Evaluating LLM-Based Cypher Query Generation from Natural Language over a CPQ Data Knowledge Graph

UNIVERSITY OF TURKU
Master of Science (Tech) Thesis
Department of Computing
Software Engineering
2025
Akseli Nuutila

Supervisors:
Erkki Kaila

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

UNIVERSITY OF TURKU
Department of Computing

Akseli Nuutila: Evaluating LLM-Based Cypher Query Generation from Natural Language over a CPQ Data Knowledge Graph

Master of Science (Tech) Thesis, 80 p., 26 app. p.
Software Engineering
December 2025

The structured configuration data used in Configure-Price-Quote (CPQ) systems is often difficult for users to access without substantial knowledge of formal query languages. This creates barriers to exploration, even for domain experts. Recent advances in large language models (LLMs) raise the question of whether natural language interfaces can support accurate querying of such structured data. This thesis evaluates the feasibility of generating Cypher queries from natural language questions for a large-scale CPQ knowledge graph.

A Neo4j knowledge graph was constructed from real CPQ data, and an evaluation pipeline was implemented to test multiple LLM configurations. Two query sets were used for the evaluation: one requiring only an understanding of the knowledge graph schema, and another requiring additional domain-specific knowledge, supplied either as a large static text file or through a retrieval-based (RAG) context construction approach.

In the controlled evaluation presented in this thesis, GPT-5-mini was able to generate correct Cypher queries for nearly all schema-based test cases. For domain-context-augmented tasks, the evaluated configurations produced widely varying results. The best-performing combinations of few-shot prompting and retrieval-based context achieved high accuracy, reduced prompt size, and enabled a more maintainable prompting strategy. These findings demonstrate that LLM-based NL-to-Cypher generation is viable for complex CPQ data when appropriate context and prompting methods are employed. However, erroneous outputs still occurred occasionally, highlighting the need for validation mechanisms before such systems can be reliably deployed.

Keywords: Large Language Models, Natural Language Querying, Knowledge Base Question Answering, Cypher Query Generation, Knowledge Graphs, Retrieval-Augmented Generation, CPQ Systems

Contents

1 Introduction **1**

1.1 Background and Motivation . . . 1

1.2 Research Problem and Questions . . . 2

1.3 Objectives and Scope . . . 3

1.4 System Overview . . . 4

1.5 Thesis Structure . . . 4

2 Background and Related Work **6**

2.1 Large Language Models . . . 6

2.2 Knowledge Graphs . . . 8

2.3 Cypher Query Language . . . 14

2.4 Knowledge Base Question Answering . . . 16

2.5 Natural Language Querying with LLMs . . . 17

2.6 Prompting Techniques for NL-to-Query Generation . . . 22

3 Methodology **24**

3.1 Knowledge Graph Construction . . . 24

3.2 Experimental System Architecture . . . 28

3.3 Query Set 1 Experiments . . . 30

3.3.1 Purpose and Design . . . 30

3.3.2 Query Set 1 Content . . . 31

3.3.3 Experimental System Implementation and Workflow 33
3.3.4 Experimental Variables 35
3.4 Query Set 2 Experiments 38
3.4.1 Nature of the Required Domain Context 40
3.4.2 Query Set 2 Contents 43
3.4.3 Static Domain Context Baseline 44
3.4.4 Dynamic Domain Context (RAG) 46
3.4.5 Experimental Variables 50
3.5 Prototype User Interface 52

4 Results and Evaluation 54
4.1 Query Set 1 Results 54
4.2 Query Set 2 Results 66

5 Conclusion 76
5.1 Summary of Key Findings 76
5.2 Answering the Research Questions 77
5.3 Implications 78
5.4 Limitations 79
5.5 Future Work 79

References 81

Appendices

A QS1 Natural Language Queries A-1

B QS2 Natural Language Queries B-1

C QS1 System Prompt C-1

D Graph Schema V2 **D-1**

E QS1 Example NLQ-Cypher Pairs **E-1**

F QS1 Per-Query Status Distributions **F-1**

G QS2 Per-Query Status Distributions **G-1**

H QS1 Results Example **H-1**

I Prototype Chat User Interface **I-1**

J Use of Generative AI **J-1**

1 Introduction

1.1 Background and Motivation

Modern configure–price–quote (CPQ) systems typically manage large amounts of structured offer and product configuration data [1]. Access to this data is usually provided through predefined reports, fixed search views, or custom-built queries against relational databases. These mechanisms are often inflexible and require technical expertise, making it difficult for non-technical users to ask ad hoc questions about offers, products, and configuration patterns. *Natural Language Querying* (NLQ) offers a more intuitive alternative, allowing users to formulate queries in their own words.

This thesis was conducted in collaboration with Wapice and is based on real offer and product configuration data from the Summium CPQ platform[1]. Wapice is interested in making Summium CPQ data more accessible for analysis and decision-making and exploring whether large language models (LLMs) could serve as a practical natural language interface for this purpose. Knowledge graphs have not yet been used to represent Summium CPQ data. Therefore, constructing a knowledge graph from this data and evaluating its suitability for LLM-based querying serves as both a technical investigation and a concrete exploration of a new way to work with Summium CPQ.

[1] `https://wapice.com/fi/tuotteet/summium-cpq/`

More broadly, this work is connected to the research areas of *Knowledge Base Question Answering* (KBQA) and *Knowledge Graph Question Answering* (KGQA). These areas study how to translate natural language questions into executable queries over structured data sources. Much of the existing work in this field focuses on SQL and SPARQL, while natural-language-to-Cypher (NL-to-Cypher) generation for property graphs has received comparatively little attention. At the same time, there is growing interest in retrieval-augmented generation (RAG) and LLM-based workflows for enterprise data access, where external knowledge sources and domain context are supplied to the model at query time. These developments provide the broader context for evaluating LLMs as a natural language interface to Summium CPQ data represented as a knowledge graph.

1.2 Research Problem and Questions

The research problem of this thesis is to assess whether modern LLMs can generate accurate and reliable Cypher queries for realistic and often complex questions over structured CPQ data represented in a knowledge graph. Prior work has mostly focused on simpler query patterns or earlier model generations, and little is known about how contemporary models such as GPT-5-mini perform in more demanding settings or when the input is provided in languages other than English (such as Finnish). This thesis evaluates this question using Summium CPQ's hierarchical offer and product configuration data as a representative example of enterprise data with rich structure.

The main research question is:

- **RQ1:** *Can large language models accurately and reliably generate Cypher queries from natural language inputs for CPQ data represented in a knowledge graph, and under what conditions?*

To address this question, the following sub-questions are investigated:

- **RQ2:** *How does performance vary across different language models, prompting styles (zero-shot vs. few-shot), and input languages (English vs. Finnish)?*
- **RQ3:** *How does the method of providing domain context (static text file vs. dynamic RAG) affect accuracy, latency, and token efficiency?*
- **RQ4:** *What are the common limitations and failure cases observed when applying LLMs to Cypher query generation?*

1.3 Objectives and Scope

The primary objective of this thesis is to evaluate the performance of LLMs in Cypher query generation under two experimental conditions:

- **Query Set 1 (QS1):** schema-based queries using only the graph structure as context.
- **Query Set 2 (QS2):** domain-context-augmented queries requiring detailed product configuration knowledge, tested with both static and dynamic (RAG-based) context delivery.

Supporting objectives are:

- to construct a Neo4j knowledge graph from Summium CPQ offer and product configuration data originating from relational and XML sources,
- to implement an experimental backend system that integrates LLMs with the knowledge graph and supports controlled automated evaluations,
- to develop a lightweight prototype user interface that demonstrates how natural language querying could be presented to end users in practice.

The scope of the thesis focuses on evaluating accuracy, reliability, and performance under clearly defined prompting and context-provision conditions. The backend system and prototype interface are secondary contributions that were developed to the extent needed to run the experiments and to illustrate a potential user-facing application. Full-scale product integration into Summium CPQ, as well as broader user experience and deployment considerations, are outside the scope of this work.

1.4 System Overview

To conduct the experiments, an experimental system was implemented that connects large language models to a Neo4j knowledge graph through a Spring Boot backend. The knowledge graph is built from real Summium CPQ offer and configuration data and models both the high-level offer structure and the detailed product configuration choices. The backend constructs prompts, communicates with the LLMs, executes the generated Cypher queries against Neo4j, and collects evaluation metrics such as accuracy, latency, and token usage.

All experimental runs were executed in an automated manner using natural language queries from the two query sets defined in this thesis. In addition to the automated evaluation setup, a simple chat-based prototype interface was implemented to illustrate a possible user-facing version of the system. The prototype UI was not used in the experiments.

1.5 Thesis Structure

The remainder of this thesis is structured as follows:

- Chapter 2 reviews background concepts and related literature, including knowledge graphs, Cypher, KBQA/KGQA, and LLM-based natural language querying.

- Chapter 3 describes the methodology, including data and knowledge graph construction, system architecture, and the experimental setups for Query Set 1 and Query Set 2.
- Chapter 4 presents the experimental results and evaluation.
- Chapter 5 discusses the key findings, implications, limitations and future work, and concludes the thesis.

2 Background and Related Work

This chapter provides the background and related literature relevant to LLM-based NL-to-query generation over knowledge graphs. The following sections summarize the fundamental concepts and techniques of the system and experiments presented later in the thesis.

2.1 Large Language Models

Natural Language Processing (NLP) is a branch of computer science focused on enabling machines to understand, interpret, and generate natural human language. NLP has advanced significantly over the past few decades. It has evolved from statistical methods to neural networks and lately to pre-trained language models (PLM) and large language models (LLM). Breakthroughs in computing power and the availability of large data sets have helped drive these advances. [2], [3]

LLMs represent the cutting edge of NLP, based on the Transformer architecture invented by the Google Brain and Google Research teams in 2017 [4]. LLMs such as OpenAI's GPT-3 include hundreds of billions of model parameters and have been pre-trained on vast amounts of diverse text data [5]. This scale has enabled LLMs to generalize to a wide range of NLP tasks, including text summarization, question answering, and language translation, with human-like accuracy. They can also perform entirely new tasks with minimal examples. Their impact is transforming industries by providing tools for content creation, code generation, and, as explored

in this thesis, natural language querying of structured databases. [2], [3]

Limitations of LLMs

LLMs have several limitations that are well-known and relevant to NL-to-query generation. One such issue is *hallucination*, where LLMs produce seemingly fluent yet factually incorrect or unsupported statements. Hallucinations arise from limitations in pre-training data, misalignment during fine-tuning, and uncertainty during inference. Recent surveys have categorized hallucinations as factual errors, context inconsistencies, and unverifiable content. [6]

LLMs can also inherit *biases* from the large-scale training data they are based on. Since their predictions reflect statistical patterns in the data, they can reproduce socially or semantically biased associations. These biases may become apparent in structured query generation when the model prioritizes certain entities or relations based on prior frequency rather than the provided schema context. [6]

Another limitation is the tendency to rely on *spurious correlations*. Rather than using genuine reasoning, models may latch onto superficial features that appear to be predictive during training but are actually only statistically correlated. This includes concept-level shortcuts, in which the model generalizes an unintended association between a concept and an outcome. Recent analyses of concept bias in language models have demonstrated this phenomenon. [7]

Another practical limitation is the unreliability of LLMs with very long contexts. Even when models support large context windows, recent evaluations show that they often fail to reliably utilize information from very long prompts, especially when key facts appear in the middle (the "lost in the middle" effect) [8], [9], [10].

Tokenization

Large language models process text as tokens rather than whole words. Subword tokenizers, such as SentencePiece [11], break text down into these units and assign them integer IDs that the model uses. Tokenization determines how text is represented internally and defines the length of model inputs, as prompt size limits are measured in tokens rather than characters.[1] [2]

2.2 Knowledge Graphs

History & Motivation

The term "knowledge graph" (KG) first appeared in academic literature as early as 1972, though it was often used in ways that were unrelated to its modern interpretation [12], [13]. However, the widespread adoption and modern understanding of knowledge graphs largely originated from Google's announcement of its Knowledge Graph in 2012 [14]. This event significantly popularized the concept, shifting search from matching keywords "strings" to understanding real-world entities and their relationships "things" [15], [16].

Knowledge graphs emerged from the need to enable machines to "understand" and leverage large, diverse, and dynamic datasets, which presents challenges in the case of traditional structured databases [12], [13]. They are motivated by the desire to organize data and relationships to reveal new insights for users or businesses [16]. Many of the foundational ideas and standards for knowledge graphs, such as RDF (Resource Description Framework), OWL (Web Ontology Language), RDFS, SPARQL, and Linked Data principles, were developed within the field of the Se-

[1] See OpenAI: Key Concepts – Tokens, `https://platform.openai.com/docs/concepts#tokens`

[2] An interactive tokenizer tool is available at `https://platform.openai.com/tokenizer`

mantic Web [13], [17].

Definition & Core Components

The modern understanding of knowledge graphs views them as organized representations of real-world entities and their relationships [16]. More formally, a knowledge graph is a type of graph designed to accumulate and present knowledge about the real world [12], [13]. In this type of graph, nodes represent entities, and edges represent the relationships between them. A knowledge graph is commonly understood as a set of triples (subject, predicate, object), each of which represents an assertion or fact about entities, relations, and semantic descriptions [12], [18]. This structured representation enables machines to process semantic information [18].

The core building blocks of a knowledge graph include:

- **Nodes (vertices)** represent real-world entities such as people, places, objects, and other concepts. Nodes have labels that identify their type. They can also have one or more optional **properties (attributes)** to describe their features. [16]

- **Relationships (edges)** link two nodes and indicate how entities are related. Relationships also have labels to identify their type and one or more optional properties. [16]

- **Organizing principles** define the structure and constraints of a knowledge graph, often via a schema or, in more formal cases, an ontology [13], [16]. A schema specifies node types (e.g., `Country`, `City`, `Language`), allowable properties (like `population`), and relationships (e.g., `CAPITAL`, `OFFICIAL_LANG`) as seen in Figure 2.1.

  When described as an *ontology*, this structure is extended with richer semantics, such as class hierarchies, domain and range restrictions, cardinality con-

straints, as well as inference rules using formal languages, such as OWL/RDFS. Developing a full ontology requires explicit definitions of concepts and their interrelations, which can be time-consuming and unnecessary with small knowledge graphs. Many practical knowledge graphs begin with a simpler schema and evolve it toward a formal ontology when more advanced reasoning or interoperability is needed.[13], [16]

Simple knowledge graphs might skip the creation of a detailed and complex ontology at first, but it's essential for establishing valid concepts and relationships, as well as enabling more in-depth insights and querying. Knowledge graphs are usually stored in graph databases, like Neo4j[3], which natively handle these interconnected data structures. [16], [18]

Figure 2.1 shows a simple knowledge graph implemented as a *property graph* [19] representing countries, cities, and languages as nodes, and their connections as named relationships. Countries are linked to their official languages with the `OFFICIAL_LANG` relationship, and to their capital cities via the `CAPITAL` relationsip. The `LOCATED_IN` relationship connects each city back to its country, illustrating how entities of different types are semantically related. Shared entities, like the language Swedish, highlight how multiple nodes can point to the same resource. Each entity also has properties such as `name` and `population`, demonstrating how descriptive data can be attached to graph nodes.

Applications

Knowledge graphs are used for many AI-driven tasks, especially those involving diverse, dynamic, and large-scale data [12]. Notable applications include improving web searches and queries (e.g., Google, Bing, and Amazon) [12], [13]. Knowledge graphs also serve as semantic databases (e.g., Wikidata) and are essential for big

[3] `https://neo4j.com/product/neo4j-graph-database/`

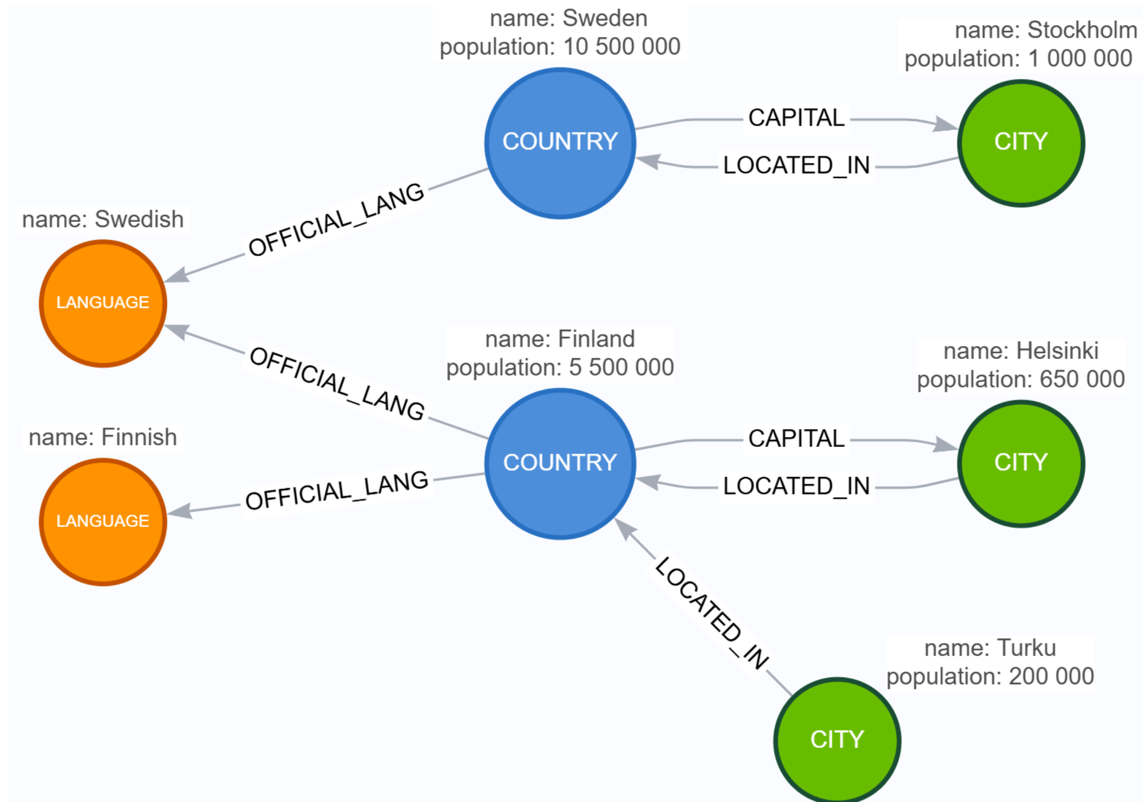

Figure 2.1: An example knowledge graph implemeted as a property graph illustrating countries, cities, and languages as entities with properties and semantic relationships.

data analytics in various industries (e.g., Walmart and financial services) [12], [13]. In the context of generative AI, knowledge graphs are used to ground LLMs across a wide range of enterprise applications that require domain-specific data, including search, question answering, and conversational agents [16]. RAG systems leverage knowledge graphs as a source of domain-specific content and also for their semantic structure, which improves response accuracy and explainability by providing contextual relationships between entities in the data. Other key use cases include connecting users in social networks, fraud detection, supply chain management, and investigative journalism. Knowledge graphs also power recommendation generation,

chatbot functionalities, decision support systems, text understanding, and more [12], [13], [16].

Construction

A systematic review [12] identified six main steps in the knowledge graph development process:

- **1. Identify Data**: The first step is to define the domain of interest and identify relevant data sources. These sources can be structured, like databases, semi-structured, like XML or JSON, or unstructured, like plain text. The type of data source influences the entire development process and how knowledge will be extracted. For example, web crawlers are often used for online content, data mining techniques for databases, and direct file access for downloadable documents. The outcome of this step is a data set that will serve as input for building the knowledge graph.

- **2. Construct the Knowledge Graph Ontology**: This step is part of the top-down approach, where a top-level structure is defined upfront, either based on an existing domain ontology or structured data. Ontologies can be developed manually by domain experts or automatically.

- **3. Extract Knowledge**: This step involves identifying the entities and relationships between them, as well as their attributes, in the acquired data. The complexity of this process depends on the type of data. Structured data allows for relatively straightforward extraction, whereas semi-structured and unstructured data require more advanced techniques. Common methods include machine learning (ML), NLP, and open information extraction (OIE). If a predefined ontology exists, it can help assign the extracted relations to known types. The extracted knowledge forms the foundation for creating triples, the

core building blocks of a knowledge graph.

- **4. Process Knowledge**: This step focuses on ensuring the quality, consistency, and completeness of the extracted knowledge prior to constructing the final knowledge graph. This involves integrating knowledge from multiple sources, cleaning noisy data, resolving duplicate or ambiguous entities, and merging semantically similar relations. Then, the extracted entities and relations are aligned with a predefined ontology, or a new one is constructed if one does not already exist. The ontology provides a structured model for organizing the knowledge and evaluating its completeness. The graph can also be enriched through reasoning and inference by deriving new relationships from existing ones, and it can be validated to ensure the included triples are meaningful and consistent. The final step is to optimize the graph by removing irrelevant or conflicting elements to improve its quality and usability.

- **5. Construct the Knowledge Graph**: This step ensures the graph is accessible and usable by storing it in a suitable database (e.g., relational, key/-value, triple stores, or graph databases like Neo4j), displaying and visualizing the graph for exploration (e.g., Google's infoboxes), and implementing tools that enable its intended applications. Visualization supports interactive exploration and understanding of the graph's structure and helps reveal hidden connections. Querying functionality is essential for retrieving and analyzing information and interacting with the graph. This functionality is typically supported through query languages such as SPARQL for RDF triple stores and Cypher for property graph databases like Neo4j. The Cypher query language is discussed in more detail in Section 2.3. The approach chosen for storing, displaying, and using the graph should align with its domain, scale, and the needs of its end users.

- **6. Maintain the Knowledge Graph**: As data sources evolve, maintaining the knowledge graph requires continuous monitoring, evaluation, and updating. This involves tracking usage and collecting user feedback to identify gaps, improve functionality, and adapt to changing needs. Updates may involve integrating newly available data from existing sources or incorporating entirely new data sources. To ensure the knowledge graph remains relevant and accurate over time, the initial steps of the development process may need to be repeated as part of ongoing maintenance.

Knowledge graph development can follow either a top-down or a bottom-up approach. In the top-down approach, the ontology is defined first. In the bottom-up approach, knowledge is extracted from data, and then the ontology is created. The process described here focuses on creating a knowledge graph from scratch rather than maintaining or updating an existing one. While most research emphasizes the initial development phase, ongoing updates and user feedback are often overlooked. However, for real-world use, feedback and continuous improvement are important for maintaining a useful, up-to-date knowledge graph. [12]

2.3 Cypher Query Language

Cypher is a declarative query language designed specifically for property graph databases [20], [21]. Cypher was originally designed and implemented as part of the Neo4j graph database, and its development began around 2011 [20], [21], [22]. Cypher is currently the de facto standard for property graph query languages [22].

Cypher's primary purpose is to query and modify data that adheres to the property graph model, the most popular graph data model in the industry. This model consists of nodes (entities) and relationships (connections), both of which can store properties (key-value pairs). The knowledge graph shown in Figure 2.1 is an example

of this model. [20]

The language was designed as an SQL-equivalent language for graph databases. It shares many keywords and a clause syntax structure with SQL (like `WHERE` and `ORDER BY`), which helps the transition for users of relational databases. [21], [22]

In 2015, Neo4j launched the openCypher project to standardize Cypher. Cypher then became a significant contribution to the international standardization effort for ISO GQL (Graph Query Language). Today, Cypher is closely aligned with GQL. [20], [22]

Cypher is fundamentally based on pattern matching. Queries use an intuitive visual "ASCII art" syntax to describe graph patterns. Nodes are represented by rounded brackets `()`. Relationships are represented by dash-arrow notation (specifying direction and type): `-[r]->`.Both nodes and relationships can hold properties in the form of key-value pairs. [20], [21]

The queries in Cypher are structured linearly, meaning execution progresses sequentially from the beginning through the clauses. Unlike SQL, the `RETURN` projection clause comes at the end of the query. [20], [21], [22]

Cypher supports aggregation functions, such as `count()`, using the `WITH` or `RETURN` clauses. Non-aggregating expressions used with an aggregate function act as an implicit grouping key. Results can be sequenced using the `ORDER BY` clause. [20]

Cypher also includes a rich update language that utilizes visual patterns for modification. Key modification clauses include `CREATE` (for creating new entities), `DELETE` (for removing entities), `SET` (for updating properties), and `MERGE` (for matching or creating patterns). [20]

Here is a small Cypher example that queries the knowledge graph in Figure 2.1:

```
// Amount of Finnish speaking cities with a population of over 500 000
MATCH (city:CITY)-[:LOCATED_IN]->(:COUNTRY)
    -[:OFFICIAL_LANG]->(lang:LANGUAGE {name: "Finnish"})
WHERE city.population > 500000
RETURN COUNT(city) AS count_of_cities

// Returned result: [count_of_cities: 1]
```

2.4 Knowledge Base Question Answering

KBQA is an NLP task that addresses the challenge of enabling users to query structured data repositories using natural language questions [23], [24]. Rather than relying solely on knowledge internalized within models, KBQA systems extract answers by leveraging structured external knowledge sources [23], [24].

These systems can answer questions from various structured data sources, including relational databases, knowledge graphs, and other types of knowledge bases [23], [24]. A specialized area of KBQA is KGQA, which focuses specifically on finding answers to natural language questions from a knowledge graph [23], [25].

The RDF is a common framework for publishing knowledge graphs, and SPARQL has become the standard query language for accessing and retrieving information from them [23]. Systems targeting RDF knowledge bases typically generate SPARQL queries, while systems targeting relational databases typically generate SQL queries [23], [24].

Historically, KBQA and KGQA systems have relied on two main approaches: information extraction and *semantic parsing* [23]. Semantic parsing focuses on translating a natural language question into a logical form or executable query that is run against a structured data source [23], [24].

Traditional QA systems for knowledge graphs often divided the semantic parsing process into sequential phases [26]. A common pipeline approach required several complex steps to translate a question into an executable query:

1. **Entity Extraction**: Extract key entities and relations from the natural language question [25], [26].

2. **Entity Linking**: Mapping the extracted entities and relations to the corresponding vertices and predicates in the target knowledge graph [26].

3. **Logical Form Generation & Execution**: Organizing the retrieved elements to create a formal query, such as SPARQL, and executing it against the knowledge graph [26].

Before LLMs became popular, traditional QA over knowledge graphs primarily relied on ML-based methods like knowledge graph embedding (KGE), neural network modeling, and reinforcement learning (RL) [25], [27], [28].

The majority of semantic parsing research in the structured data domain focuses on two primary formal languages: SPARQL [26], [27] and SQL [24]. However, generating Cypher queries for property graph databases involves a similar core semantic parsing challenge: translating natural language into an executable formal language that aligns with the target graph schema [23], [24]. Therefore, the principles applied in SPARQL and SQL QA systems are largely transferable to Cypher QA systems. Recent studies that have investigated NL-to-Cypher generation include [29], [30], [31], [32], [33], [34].

2.5 Natural Language Querying with LLMs

Recent research [23], [24], [27], [29], [30], [31], [32], [34], [35] uses LLMs to perform semantic parsing by translating a natural language question directly into an executable

structured query. This approach treats the translation as a code generation task. This methodology often differs fundamentally from traditional multi-stage pipelines (discussed in Section 2.4) because it can combine complex, discrete tasks, such as entity linking and relation detection, into a single, implicit, end-to-end translation process.

Although this LLM-based paradigm offers substantial benefits, such as increased flexibility and reduced manual engineering, it also introduces significant limitations. LLMs often produce hallucinated queries that are syntactically correct but factually inaccurate [28], [33], [36]. Errors in graph query generation can be categorized as either structural inconsistencies, such as missing or redundant triples, or semantic inaccuracies, such as using incorrect entities or properties [27]. These semantic issues often stem from the model's limited understanding of the domain-specific data and schema [27].

LLMs also require high computational costs for training and inference. Their performance is sensitive to prompt phrasing, and they may struggle with schema drift. Overall, solutions generated by LLMs still lack complete robustness and explainability. [28], [33]

LLM-Based NL-to-Query Generation

The process of generating Cypher queries for property graphs is analogous to text-to-SQL or NL-to-SPARQL generation [34]. For example, academic advising chatbots [34] use an LLM Cypher generator (powered by GPT-4, for instance) to translate natural language into Cypher queries that can be executed on a Neo4j knowledge graph.

Several key design patterns are necessary for effective LLM-based query generation. First, schema inclusion is essential. Providing the LLM with the database or graph schema, which may include relational schemas or knowledge graph ontology

descriptions and relevant triples, enables the model to align with the schema and avoid structural errors [23], [24]. Another important pattern involves few-shot examples (in-context learning), where input-output NL-to-query pairs are supplied as guiding examples in the prompt [23], [24]. Performance is maximized by selecting these examples to balance similarity and diversity [24]. Finally, format constraints are applied by providing explicit instructions in the prompt to enforce strict control over the output. This ensures that the model only returns the executable query (e.g., "Give me only the SPARQL query, no other text") [27].

Retrieval-Augmented Generation in NL-to-Query Tasks

Retrieval-augmented generation (RAG) [37] is a key technique that tackles the shortcomings of LLMs, including hallucinations, knowledge update issues, and a lack of domain-specific expertise. RAG uses an external knowledge database to provide LLMs with relevant factual information. [36]

Retrieval-augmented generation (RAG) [37] is a key technique that addresses several limitations of LLMs, including hallucinations, knowledge-update challenges, and lack of domain-specific expertise. In RAG, the model is supplied with relevant external knowledge retrieved from a database or knowledge store, rather than relying entirely on its internal parametric knowledge. [36]

Recent needle-in-a-haystack evaluations indicate that simply increasing the size of the prompt does not guarantee reliable access to relevant information. Model accuracy often decreases as context length grows, especially when key facts are surrounded by large amounts of distracting content. [8], [9], [10]. These findings motivate supplying compact, highly relevant retrieved context instead of relying on very long static prompts, especially in settings where vast amounts of domain-specific information must be provided to the model.

In NL-to-query tasks, the relevant context retrieved by the RAG-system typically

contains schema fragments, candidate entities, properties, or relationship structures that help the LLM align natural language questions with the underlying data model [27], [32]. Prior research in Text-to-SQL indicated that augmenting prompts with such schema-level information improves the model's grounding and reduces structurally invalid or semantically incorrect queries [24]. Similarly, graph-based RAG methods demonstrate that retrieving structured subgraphs or triples enables LLMs to compensate for missing domain knowledge and significantly reduces semantic errors in multi-hop question answering [32]. However, it's important to note that misaligned or noisy low-quality RAG context can ultimately result in reduced task performance [27].

In the context of querying structured data, RAG can improve the accuracy of query generation for knowledge-intensive tasks. LLM-based query approaches often struggle due to limited exposure to domain-specific content and underlying ontological schema. Successfully RAG implementation compensates for this limitation by first retrieving and then augmenting the LLM prompts with relevant external, domain-specific knowledge, thereby enhancing the model's contextual understanding. [27]

By incorporating KG-grounded information, RAG directly tackles semantic inaccuracies, which occur when LLMs fail to link to the correct entities or properties due to their limited parametric knowledge of the underlying knowledge graph content. [27]

RAG functions as a non-parametric memory, which allows for the updating of external knowledge without the need for retraining or fine-tuning the model. This updatability is particularly useful in domains where schemas or domain knowledge evolve over time. [37]

Evaluation Practices in NL-to-Query Research

The evaluation of LLM-based NL-to-query systems generally combines execution performance metrics with string similarity metrics to comprehensively assess system quality [26], [27], [35]. In the context of generating structured queries over knowledge graphs, execution accuracy is the most meaningful measure because it directly reflects whether the system retrieves the correct result from the underlying data source. A commonly used metric is the success rate, which is defined as the proportion of generated queries that are syntactically valid, executed successfully, and return an accurate, meaningful result [34]. This metric closely aligns with real-world expectations because the practical value of a system depends on its ability to produce executable queries that lead to correct answers. Precision, recall and F1 score [33], [34], [35] are also common.

Research also reports on query string accuracy metrics, such as the exact match score [36], [38]. This metric compares the generated query to a gold-standard logical form. While these metrics offer insight into how closely the model imitates a reference query, they are less useful for evaluating the performance of an entire system in realistic settings, where different, yet logically equivalent, queries may produce the same correct result.

To better understand the limitations of LLM-based query generation, studies typically classify errors into categories:

1. **Syntax errors**, where the generated query violates the formal structure of the query language or fails to parse. [23], [27]
2. **Semantic errors**, where the structure is correct but the wrong entities, relationships, or properties are selected. [23], [27]
3. **Logical errors**, where the query is syntactically valid and uses appropriate terms but encodes incorrect reasoning relative to the knowledge graph. [23]

In addition to accuracy, some studies examine broader considerations, such as the model's robustness to variations in input phrasing, consistency of outputs across repeated runs, and execution time of the overall system. These aspects characterize the reliability and practical feasibility of approaches combining LLMs with structured data sources. [26], [34]

2.6 Prompting Techniques for NL-to-Query Generation

LLM-based query generation relies on practical prompting strategies that bridge the gap between natural language and formal query syntax. These strategies align the model's general knowledge with the specific structure and content of a target database.

Few-Shot Prompting

In few-shot prompting, demonstration examples (pairs of natural language questions and their corresponding structured queries) are incorporated directly into the prompt. These examples serve as in-context learning (ICL), guiding the model to use the correct structure and vocabulary of the target query language. An effective few-shot design balances similarity and diversity in the examples. Even a small number of demonstrations has been shown to enhance query generation and improve performance [29]. [23], [24]

Dynamic Context Construction and Selection

Dynamic context construction uses a RAG approach to retrieve only the necessary information for a specific question. Selective retrieval gathers relevant schema

fragments, entities, properties, and subgraphs, transforming them into a textual representation in the prompt [28]. This helps the model align the terms in the question with the correct schema elements. It also avoids prompt bloat by excluding irrelevant information. Long prompts can slow down the inference process of an LLM [38]. Additionally, some systems supply representative property values or extracted value candidates from the database to ensure that filtering conditions match the actual data format since schemas alone do not convey how property values appear in practice [28]. These techniques reduce noise, improve accuracy, and limit the negative effects of excessive prompt length by keeping the prompt compact and focused on the most relevant context.

Guardrails and Iterative Refinement

Systems incorporate explicit output constraints (guardrails), to ensure the generated query is executable [27], [29]. For example, the model may be required to use only specified schema elements. Some approaches use iterative refinement, in which the LLM receives feedback, such as a validation or execution error, and rewrites the query until it succeeds [29], [39], [40]. Lightweight correction layers may also correct minor structural or syntactic errors [27].

3 Methodology

This chapter describes the constructed knowledge graph, system architecture, and experimental setups used to evaluate LLMs in Cypher query generation tasks. The experiments were carried out using two structured collections of natural language queries: *Query Set 1* and *Query Set 2*. Together these query collections test LLM performance in multiple different conditions. Query Set 1 focuses on schema-based query generation using only the graph structure as context, while Query Set 2 extends the setup with additional domain knowledge supplied either statically or with a RAG-based approach.

To ensure that the query sets included realistic use cases from the CPQ domain, many of the queries were derived or refined in collaboration with a domain expert familiar with the underlying data. The remaining queries were constructed to cover additional patterns required by the experimental design.

3.1 Knowledge Graph Construction

The construction of the knowledge graph in this thesis follows the general principles described in Section 2.2, where a systematic review identified six main steps in the development process: identify data, construct an ontology, extract knowledge, process knowledge, construct the graph, and maintain it [12]. In practice, this process was considerably simplified in this case because the data was already highly structured, but the framework still provides a useful reference point for describing

the work.

For the experiments, permission was obtained to use real customer data from the Summium CPQ system. The dataset consisted of all actualized offers from a Finnish industrial company for the year 2024, covering configurable industrial luminaires. Offers can exist in several states in the database, but for this thesis only those that had been finalized and resulted in sales were included. This ensured a sufficiently large and realistic dataset while focusing on the most relevant cases.

The data resides in a PostgreSQL relational database that consists of more than 100 tables in total. Seven tables were identified as most relevant for offer and product data. Six of these tables corresponded directly to graph entities: `Offer`, `Participant`, `User`, `Content`, `Subcontent`, and `Product`. The seventh table contained XML files with detailed information about the configured products in each offer. The XML data in this table resulted in the creation five additional node types: `ProductFamily`, `Tab`, `Parameter`, `ParameterValue`, and `Attribute`. Together, these eleven node types and the relationships between them formed the graph schema used in the experiments (see Figure 3.1 and Appendix B).

The XML configuration data followed a standardized hierarchical structure, which made the mapping to graph form relatively straightforward. At the top level, each `ProductFamily` contains `Tab` elements. Tabs in turn contain `Parameter` elements, which all contain one `ParameterValue` element. All four of these element types can also have `Attribute`s that store additional metadata. Furthermore, `ProductFamily` and `Tab` elements may themselves contain nested child families or tabs, forming a recursive hierarchy. The nested structures were flattened during preprocessing to make importing easier, while preserving enough information to rebuild the hierarchical relationships in the graph.

A key technical challenge was that the XML files were very large and stored as binary data in the database, which made direct querying impossible. To address

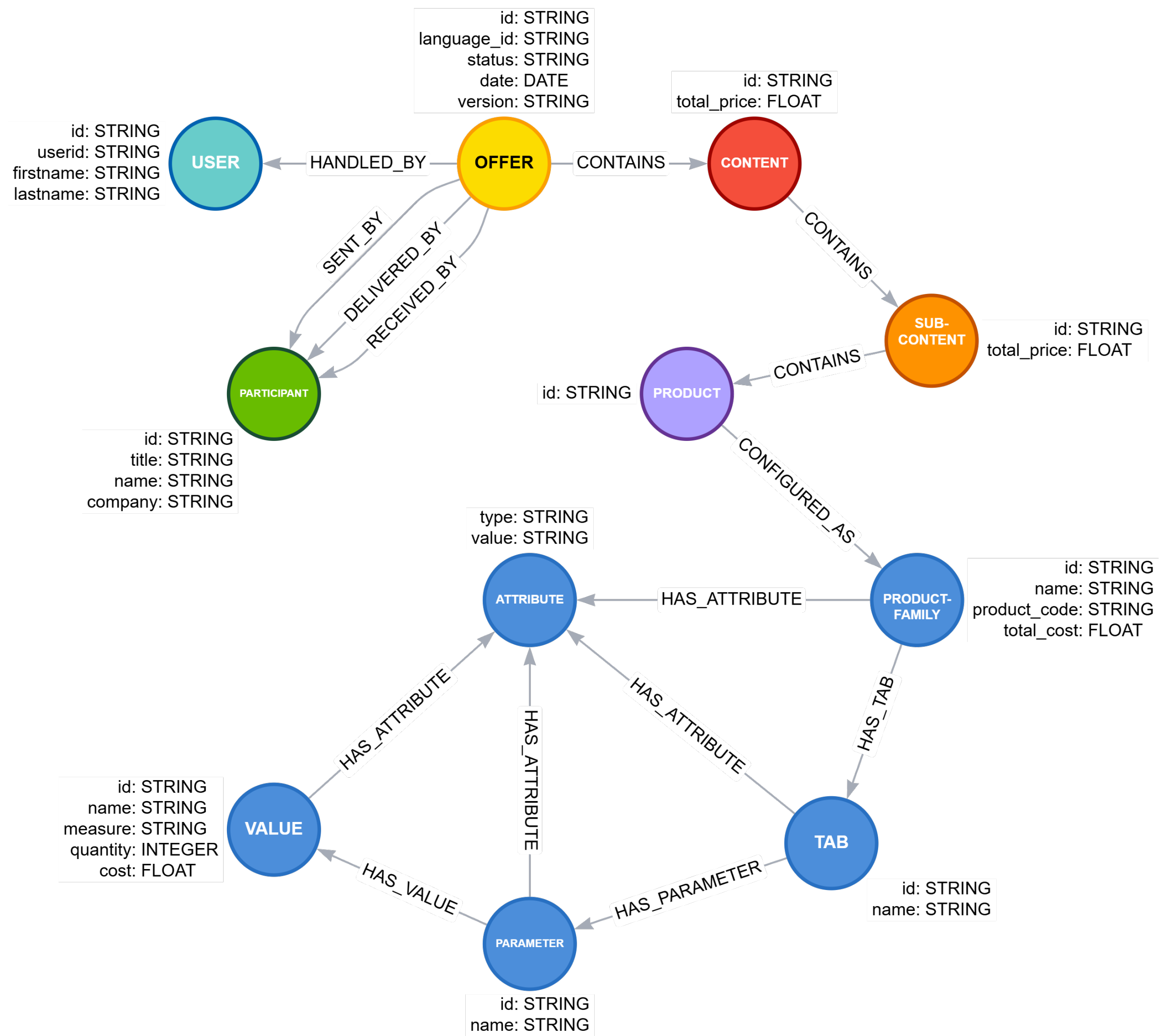

Figure 3.1: Graph database schema

this, Python preprocessing scripts were created to export the XML, convert it to UTF-8, strip irrelevant data, and write the extracted elements into separate CSV files corresponding to the target node types. This approach was considerably more efficient than using Neo4j's built-in `apoc.load.xml()` function, which proved too slow for the scale of the data.

Importing the CSV files into Neo4j required custom Cypher scripts. These scripts first created the nodes of each type from the CSV data, including a selected subset of the most relevant properties from the original relational tables, and then established the relationships between them based on foreign keys in the relational tables and references in the XML data. The complete import scripts are provided in the accompanying GitHub repository[1]. The result was a unified graph that seamlessly combined the relational offer data with the XML-based product configuration data, preserving both the high-level offer structure and the detailed configuration choices for individual products in those offers.

The final outcome was a knowledge graph, implemented as a property graph in Neo4j, consisting of eleven node types and their corresponding relationships. The resulting graph contained roughly 1.4 million nodes and 7 million relationships, providing an accurate representation of the original data sources and a realistically large setting for NLQ tasks. Although the mapping process was relatively straightforward due to the structured nature of the data, it still required substantial effort in preprocessing and scripting. In a production environment, additional automation and maintenance mechanisms would be necessary to keep the graph continuously updated as the underlying CPQ data evolves.

[1] `https://github.com/anuutila/Evaluating-LLM-Based-Cypher-Query-Generation.git`

3.2 Experimental System Architecture

The controlled experiments of this thesis were executed with an *experimental system* that integrates large language models with a Neo4j graph database through a Spring Boot backend. This system was not intended as a primary contribution of the thesis, but it was necessary to enable the evaluation of LLM-generated Cypher queries in a realistic environment. It also provides a foundation for potential future integration into the existing Summium CPQ product.

The same system framework was used in both experimental setups described in this thesis. Later sections (3.3 and 3.4) describe how this base system was configured for Query Set 1 and extended for Query Set 2 experiments. An optional chat-based prototype user interface (UI) was also implemented for interactive use, though all experimental runs were conducted in an automated way without any manual human interaction through a user interface.

Technology Stack

The backend was implemented as a Spring Boot[2] (version 3.5.5) application, using Spring AI[3] (version 1.0.3) to interact with LLMs. These technologies were chosen because they are already part of the technology stack of the Summium CPQ product. The use of Spring AI also simplifies the orchestration of prompts and responses between the backend application and the LLMs.

The LLMs were accessed through the Azure OpenAI[4] service, using the `gpt-4o` (version 2024-08-06) and `gpt-5-mini` (version 2025-08-07) model variants. Azure was chosen because it is already used in the company's cloud infrastructure. This ensured compatibility with existing practices and made it easy to deploy the models.

[2] `https://spring.io/projects/spring-boot`

[3] `https://spring.io/projects/spring-ai`

[4] `https://azure.microsoft.com/en-us/products/ai-services/openai-service`

The graph database was implemented using Neo4j[5] (version 2025.05.0), the most widely used graph database management system [41]. Its native property graph model and specialized query language (Cypher) made it a natural choice for the experiments in this thesis.

High-Level Architecture

Figure 3.2 illustrates the high-level components and connections of the experimental system. The chat UI forwards user queries to the backend and the backend constructs prompts and communicates with the LLM and Neo4j. Detailed processing flows and step-by-step sequences are described later for both Query Set setups.

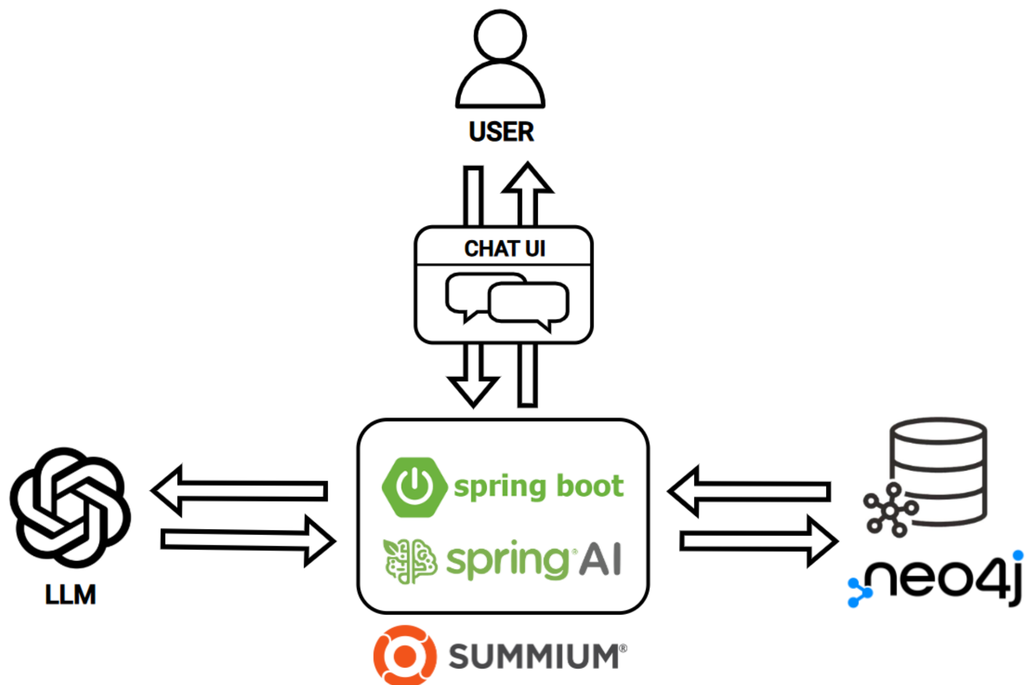

Figure 3.2: High-level architecture of the experimental system.

[5] https://neo4j.com/

3.3 Query Set 1 Experiments

3.3.1 Purpose and Design

The first set of experiments (Query Set 1) aimed to measure the capability of LLMs to generate valid Cypher queries when only the graph schema and the natural language query, in addition to minimal instructions, were given in the Cypher generation prompt. No domain-specific context was provided beyond the graph schema itself, so all entity and relationship names appearing in the natural language queries of Query Set 1 directly correspond to schema elements. This setup creates a controlled baseline for evaluating models' reasoning and syntax generation skills in an idealized, fully schema-aligned environment.

The natural language queries were manually designed to represent coherent but deliberately constrained information retrieval tasks. These queries are not representative of how end users would phrase questions in a production system since real users are unlikely to know or use the exact schema terminology. However, this restriction was necessary in order to isolate the models' ability to correctly generate Cypher statements when all the necessary context is explicitly included in the natural language query itself. The queries were first created in English, strictly following the schema terminology. Then, they were translated into Finnish to examine how language differences affect LLM performance and how well the models can preserve semantic accuracy when schema terms are no longer verbatim matches.

To cover a representative range of query variations, all queries in Query Set 1 were categorized by *Type* and *Complexity Level*:

- **Types:**
  - *Lookup*: retrieves a single property of a known node.
  - *Single-hop*: follows one relationship to a directly related node.

  - *Multi-hop*: traverses two or more relationships in sequence.
  - *Aggregation*: performs summary operations such as `COUNT` or `SUM`.
  - *Analytics*: conducts analytics operations such as averaging or ranking.

- **Complexity Levels:**
  - *L1* (Very Simple): a single-property or simple one-hop query without filtering.
  - *L2* (Intermediate): includes filtering, short multi-hop traversals, or basic aggregation.
  - *L3* (Hard): combines multi-hop traversals with filters, aggregation, or advanced analytical operations.

Not all type–level combinations are meaningful.

- *Lookup* only applies to *L1*, as it lacks structural complexity.
- *Single-hop* queries can be either *L1* or *L2* (e.g., when filters are added) but not *L3*.
- *Multi-hop*, *Aggregation*, and *Analytics* are applicable at *L2* and *L3*, as they inherently involve traversal, summarization, or more demanding logical reasoning.

This structure aims to ensure that each query's complexity corresponds to its logical challenge and prevents nonsensical combinations. Table 3.1 lists the valid type–level combinations used to build the query set.

3.3.2 Query Set 1 Content

Query Set 1 consists of 18 distinct information-retrieval tasks, each written in both English and Finnish (36 queries in total). The queries were constructed to ensure

Table 3.1: Valid type-level combinations. Complexity Levels: *L1* (Very Simple), *L2* (Intermediate) and *L3* (Hard).

Type	L1	L2	L3
Lookup	✔	-	-
Single-hop	✔	✔	-
Multi-hop	-	✔	✔
Aggregation	-	✔	✔
Analytics	-	✔	✔

balanced coverage of the nine valid Type–Level combinations defined in the previous section, with exactly two tasks representing each combination. All Query Set 1 queries use schema-aligned terminology so that every entity or relationship mentioned in the natural-language description corresponds directly to an element in the graph schema.

Table 3.2 shows a couple representative examples from Query Set 1. The complete set of all 36 queries is provided in Appendix A.

Table 3.2: Queries Q01 and Q18 from Query Set 1

ID	Lvl/Type	Lang	Natural Language Query
Q01E	L1–Lookup	EN	What is the version of offer {offer_id}?
Q01F	L1–Lookup	FI	Mikä on tarjouksen {offer_id} versio?
Q18E	L3–Analytics	EN	List the top 3 participants grouped by name and ranked by number of offers dated in Q3 2024 that they sent, including the offer counts and the combined total price of those offers.

ID	Lvl/Type	Lang	Natural Language Query
Q18F	L3–Analytics	FI	Listaa kolme osallistujaa, ryhmiteltynä nimen mukaan, jotka lähettivät eniten vuoden 2024 kolmannelle vuosineljännekselle päivättyjä tarjouksia, ja näytä tarjousten määrät sekä yhteenlasketut kokonaishinnat.

3.3.3 Experimental System Implementation and Workflow

Figure 3.3 shows the functional flow of the experimental system used to execute the Query Set 1 experiments. This setup builds upon the baseline system framework introduced in Section 3.2.

The process proceeds as follows:

1. A natural language query is entered in the chat-based user interface (or automatically read from a file containing the whole query set) and passed to the Spring Boot backend.

2. The backend constructs a prompt using Spring AI, combining the natural language query with the system instructions and additional context such as the graph schema and few-shot examples, when applicable.

3. The prompt is sent to the Azure-hosted LLM, which generates a Cypher query based on the provided context. The resulting Cypher query is returned to the Spring Boot backend.

4. The Cypher query is executed on the Neo4j graph database.

5. Neo4j returns a raw JSON response containing the query results.

6. The backend combines this raw response with the original natural language query into a second prompt.

7. The second prompt is sent to the LLM, which reformulates the output into a final natural language answer that is returned to the backend.

8. The final answer and collected metrics are displayed in the chat UI (or written to a results file in automated runs).

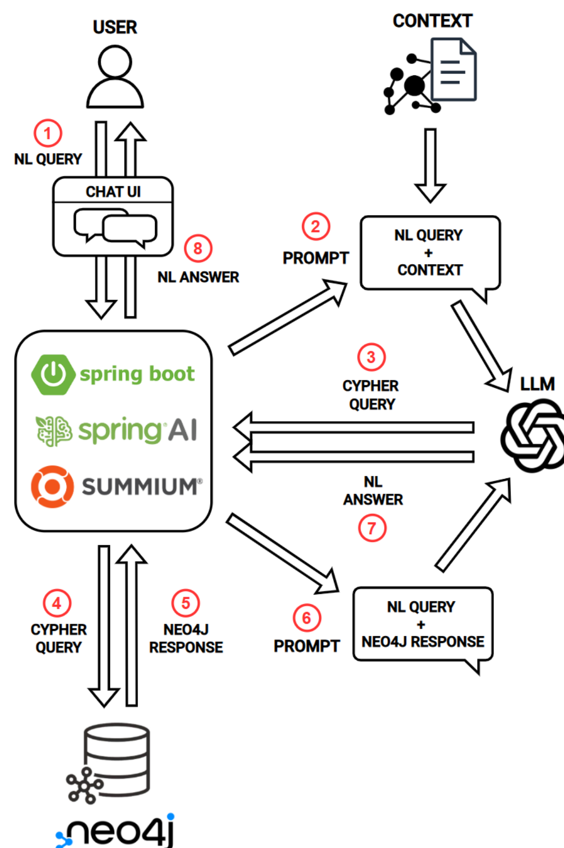

Figure 3.3: High-level architecture and data flow of the experimental system implementation for Query Set 1. The numbers (1–8) indicate the sequence of steps from the initial natural language query to the system's response.

In the Query Set 1 experiments, the process was executed entirely in an automated way. The evaluation setup was designed to process all 36 queries (18 English and 18 Finnish), each repeated three times to account for the nondeterministic behavior of LLMs. For each query, the generated Cypher, summary answers, and associated metrics, including end status, latency, and token usage, were automati-

cally collected and written to a results file. The controlled experiments involved no human interaction or graphical user interface.

3.3.4 Experimental Variables

To systematically examine the factors impacting LLM performance in Cypher query generation, a set of independent and dependent variables was defined. The independent variables represent the configurable conditions of the experimental setup, and the dependent variables measure performance in terms of accuracy, efficiency, and cost.

Independent Variables

The following independent variables were defined and tested in the Query Set 1 experiments:

- **LLM**: Three OpenAI model configurations were compared: `GPT-4o` and `GPT-5-mini` with reasoning effort set to low (*rl*), and `GPT-5-mini` with reasoning effort set to high (*rh*).
- **Prompt style**: either *zero-shot* or *few-shot* (see Section 3.3.4), depending on whether example NL–Cypher pairs were included in the prompt or not. This variable tests whether including few-shot examples improves the model's ability to generalize relevant Cypher query patterns in this particular context.
- **Graph schema representation**: Two alternative schema formats were provided to the model:
  - *V1 – Verbose JSON schema*: the raw export from Neo4j using the `apoc.meta.schema` procedure, including detailed and verbose metadata about all the nodes and relationships in the graph database (available

in the GitHub repository[6]). This representation mirrors a realistic but information-heavy format produced by automated schema extraction tools.

- *V2 – Minimal text schema*: a manually condensed version containing only relevant node labels, relationship types, and property names, structured as a human-readable summary (see Appendix D). This format tests whether a compact, low-token schema improves comprehension and reduces cost.

The comparison between these two formats focuses on the impact of schema verbosity and structure on the accuracy of Cypher generation and token usage.

- **Query language**: Each query was executed in both English and Finnish to assess whether linguistic variation affects model accuracy.

Prompt Style Conditions

In the *zero-shot* prompting style, the system prompt contained only minimal instructions (Appendix C) and the graph schema (Appendix D). In the *few-shot* prompting style, three example pairs of natural-language queries and their corresponding correct Cypher queries were added to the Cypher generation prompt (Appendix E).

Few-shot prompting typically uses a small number of examples to balance instructional coverage with prompt length. The goal here was not to optimize the number of examples, but rather to test whether including any illustrative examples measurably improves LLM performance compared to zero-shot prompting. To avoid contaminating the evaluation, none of the examples were drawn directly from the evaluation query set. Instead, separate English examples were created to demonstrate some of the core patterns found in Query Set 1: (1) direct lookups along

[6] `https://github.com/anuutila/Evaluating-LLM-Based-Cypher-Query-Generation.git`

single relationships, (2) multi-hop traversals through nested structures, and (3) aggregations with date filters and ordering.

Dependent Variables

The following dependent variables were measured to evaluate model performance:

- **Status**: The qualitative outcome assigned to each query attempt, determined by whether the LLM produced a valid and correct Cypher query and whether it executed successfully in Neo4j. The status values were defined as follows: `OK`: the query executed successfully and returned correct results, `WRONG_RESULT`: the query executed successfully but returned incorrect or incomplete results, `SYNTAX_ERROR`: the generated Cypher query contained syntactic errors and could not be executed, `NO_ATTEMPT`: the model explicitly declined to generate a query.

- **Accuracy**: The configuration-level accuracy, i.e., the proportion of successful (`OK`) queries out of all attempts, reported both as a percentage and as the raw count (e.g., 74/108).

- **Total time**: The macro-average end-to-end latency (Cypher query generation + Neo4j execution + summary generation) in milliseconds. For each query ID, the mean latency of each stage is first computed across its successful runs. The reported total value is then obtained by summing together the averages of these per-query-ID means for each stage. This ensures that each query ID contributes equally, regardless of how many successful runs it has.

- **Prompt tokens**: The macro-average number of input tokens sent to the model across both LLM calls (Cypher generation + summary generation).

- **Completion tokens**: The macro-average number of tokens generated by the

model in both LLM calls, including reasoning and output tokens[7]. The amount of prompt and completion tokens is the main factor in cost efficiency.

The `NO_ATTEMPT` status originated from a safeguard instruction in the system prompt that directed the model to decline when it determined that a Cypher query could not be generated based on the available context. Although all queries in Query Set 1 could in principle be solved with the provided information, this rule was included to simulate realistic guardrail behaviour: in practical NL-to-query systems, safety mechanisms prevent the model from producing hallucinated or irrelevant answers to unexpected or unclear user questions. The four status categories align with the error classifications discussed in Section 2.5: `SYNTAX_ERROR` corresponds to syntax violations, while `WRONG_RESULT` captures both semantic and logical errors. The `OK` status aligns with the concept of successful execution commonly used in NL-to-query research, where a query must be syntactically valid, executable, and return a correct result.

Using macro averages for total time and token counts ensures comparability across different runs even when some queries fail or only partially complete. Together, these three metrics (accuracy, total time, and token counts) allow for a direct comparison of effectiveness, efficiency, and cost across the different experimental settings.

3.4 Query Set 2 Experiments

The Query Set 1 experiments established a controlled baseline for evaluating schema-based Cypher query generation. They showed that large language models can interpret the structure of the graph schema and produce correct Cypher queries when all necessary information is directly represented in the provided schema and the

[7] `https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them`

schema-aligned natural language queries. However, such queries are not representative of real user behavior. In real-world usage, users are unlikely to know the exact schema terminology or the node and property names used internally. Instead, they ask questions that depend on knowledge about the application domain and the data itself, which is not represented in the graph schema.

Query Set 2 was therefore designed to extend the experiments to a more realistic setting where generating a correct Cypher query requires additional domain knowledge not contained in the schema. Typical examples of such queries concern the configured features of industrial luminaire products, such as their *dimensions*, *color*, or *power*, and how the values of these features vary across offers, product families and time periods for example. Answering such queries requires the model to understand how this configuration data is represented in the graph.

The Query Set 2 had two goals. First, it aimed to test how well LLMs can generate accurate Cypher queries when the prompts include more contextual information about the domain. Compared to the compact schema used in Query Set 1, the prompts in these experiments were much larger, particularly with the static domain context approach. This allowed us to examine how a significant increase in prompt size affects the model's ability to focus on relevant information and maintain accuracy in tasks requiring high syntactic and logical precision.

Second, the experiments compared two different ways of supplying this domain context and analyzed how each method affects the system's accuracy, latency, and cost. The first approach provided a precompiled text file containing a large set of feature identifiers, parameter names, and value examples. The other approach retrieved only the relevant subset of this information dynamically during runtime. These two approaches represent a trade-off between simplicity and scalability: the static file is straightforward to build for a limited dataset but becomes inefficient and difficult to maintain as the data grows, whereas the retrieval-based method requires

more implementation effort but scales better and keeps the prompts smaller and more focused.

3.4.1 Nature of the Required Domain Context

The Neo4j knowledge graph models offers that contain industrial lighting products and their configurations at a detailed level. Each `Offer` node contains one or more `Product` nodes, and each product is configured as a `ProductFamily`. The product family defines the available configuration options through a hierarchical structure of nodes: `ProductFamily`–`Tab`–`Parameter`–`ParameterValue`. The parameters represent configurable features such as *length*, *color*, or *power*, and the `ParameterValue` nodes store the actual selections for those features. Figure 3.4 illustrates a simplified example subgraph of this data structure.

Suppose a user asks:

> *"What was the most popular configured length for Snep Mode P products in 2024?"*

The correct Cypher query for this task would be:

```
MATCH (offer:Offer)-[:CONTAINS]->(:Content)-[:CONTAINS]->(:Subcontent)
     -[:CONTAINS]->(prod:Product)-[:CONFIGURED_AS]->
     (pf:ProductFamily {name: "SNEP MODE P"})
MATCH (pf)-[:HAS_TAB]->(:Tab)-[:HAS_PARAMETER]->(p:Parameter)
     -[:HAS_ATTRIBUTE]->(:Attribute {type:'ext_id', value:'pituus'})
MATCH (p)-[:HAS_VALUE]->(pv:ParameterValue)
WHERE offer.date.year = 2024
RETURN pv.name AS length, COUNT(*) AS count
ORDER BY count DESC
LIMIT 1
```

This query can be generated by an LLM only if it knows (1) the retrieval pattern by which feature information is stored in the graph, (2) the canonical internal name of the product family `"SNEP® MODE P"`, (3) the external feature ID representing product length `'pituus'`, and (4) how the actual length value is stored within the `ParameterValue` node's properties. None of these facts appear in the graph schema, which defines the database structure but not the semantic retrieval patterns or property-level values and conventions. Therefore, such queries cannot be generated correctly without additional domain-specific context that links user-facing terms, canonical identifiers, and the semantic organization of the data.

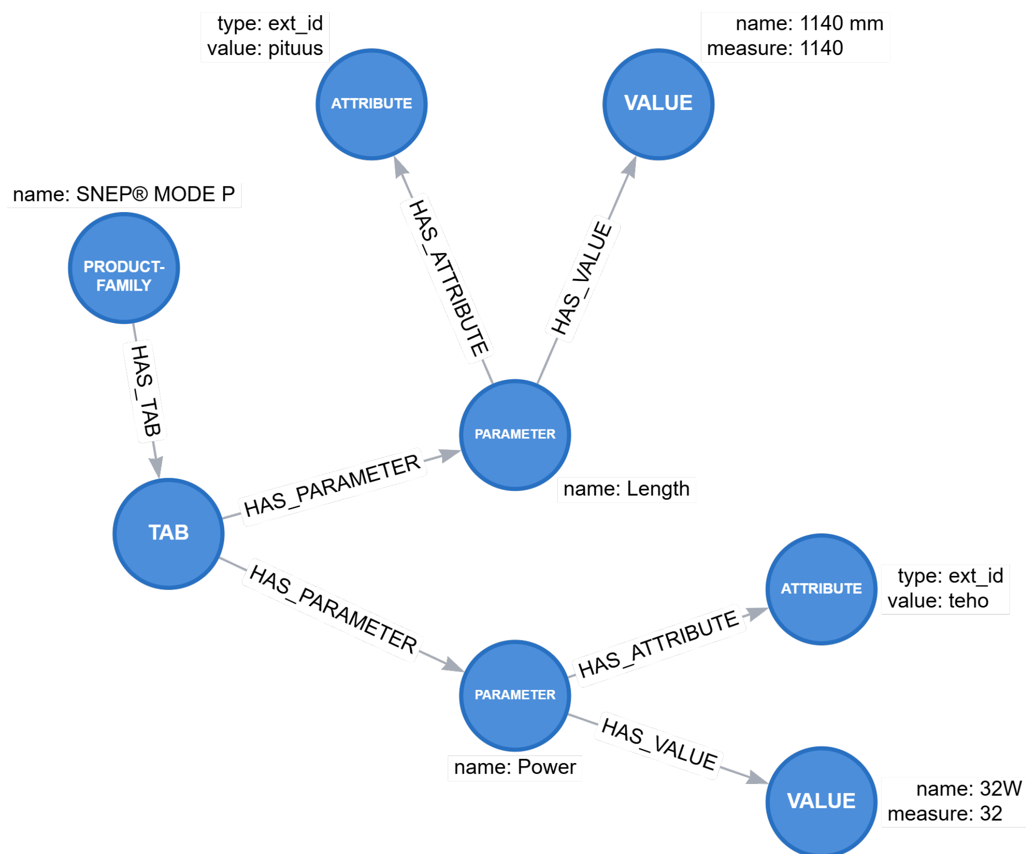

Figure 3.4: A simple subgraph of the product configuration data in Neo4j. Each `Parameter` node defines a configurable feature (e.g., *length* or *power*) and connects to an `Attribute` node storing a unique external feature ID (`type: 'ext_id'`). The linked `ParameterValue` node stores the actual selected value for that feature.

In the production environment, the names of the `Parameter` nodes correspond to the user-facing feature names that appear in the product configurator UI. For instance, customers selecting a product's "Length" or "Frame color" are interacting with these parameters through the UI. In the database, however, each parameter node is also linked to an `Attribute` node containing a unique external identifier (`type:'ext_id'`), such as `'pituus'` for length or `'vari'` for frame color. These external IDs serve as canonical references that remain consistent across all product families, even if the displayed parameter names differ slightly or change over time. Therefore, when forming Cypher queries, it is essential to reference the external IDs rather than the potentially inconsistent parameter names.

A similar issue exists for product family names. The database stores each family under a canonical label such as `"SNEP® MODE P"`, which includes exact casing and special characters. Users, however, typically refer to product families in simplified form (for example, "Mode P" or "Snep Mode P"). To ensure correct filtering in Cypher queries, the model must know the canonical product family names so it can correctly interpret such variations in natural language queries.

Some user queries also require filtering or aggregation based on specific feature values, such as finding all luminaires with power below 100W or length above 2000mm. To handle such requests, the model must understand what the actual stored data looks like for each feature. Providing representative examples of parameter values, such as `name: "57W"` and `measure: "57"` for the power feature, enables the model to create correct numerical and textual filtering logic in Cypher.

In summary, generating valid and precise Cypher queries in this environment requires additional contextual knowledge that bridges the gap between user terminology and the database's canonical identifiers. The required domain context therefore includes:

- A concise description of how feature data is represented and retrieved in the

graph, including the relevant node and relationship patterns

- Canonical product family names used in the database
- Mappings between feature IDs (external IDs) and their corresponding parameter names
- Representative samples of parameter values showing how data for each feature is stored.

The upcoming subsections describe two alternative ways this domain context was supplied to the model: first with a static text file containing all domain information, and then with a dynamic RAG-based process that retrieved only the relevant context during query generation.

3.4.2 Query Set 2 Contents

Query Set 2 consists of 14 natural-language questions designed to reflect how users typically phrase information needs in a CPQ environment. Unlike Query Set 1, which was engineered to cover a balanced set of Type–Level combinations, Query Set 2 focuses on realistic user phrasing and naturally occurring query structures. Imposing a synthetic type–level taxonomy would have biased the set away from genuine usage patterns, so no such categorization was applied.

Each Query Set 2 question includes a small set of descriptive tags (e.g., *TopK*, *Feature*, *SingleFamily*, *Temporal*), which were used during dataset construction to ensure conceptual variety. These tags are not used directly in the evaluation but are retained in the full listing for reference.

Table 3.3 shows a few representative examples. The complete set of all 14 queries, together with their associated tags, is provided in Appendix B.

Table 3.3: Queries Q01, Q09 and Q14 from Query Set 2

ID	Natural Language Query	Tags
Q01	What was the most popular configured length for SNEP Mode P lights in 2024?	TopK, Feature, SingleFamily
Q09	How many Mode C products with white color were ordered by {company_name} in 2024?	FeatureFilter, EntityFilter, SingleFamily
Q14	What were the 3 most popular selected combinations of CRI and optics among all luminaire products in each quarter in 2024?	TopK, MultiFeature, Temporal, AllFamilies

3.4.3 Static Domain Context Baseline

The first experimental configuration for Query Set 2 used a single pre-compiled text file to provide the domain-specific context required for Cypher generation. In this setup, the LLM was prompted using a slightly modified version of the system instructions from Query Set 1, combined with the same graph schema information. In addition to these base elements, the prompt also included an appended text block containing the domain information needed to interpret and generate the new queries.

The purpose of this configuration was to establish a simple baseline for evaluating how the system performs when all the necessary contextual information for the entire query set is supplied at once, without any retrieval or filtering logic. This setup required no changes to the experimental system used in Query Set 1 experiments beyond appending the domain context file to the Cypher generation prompt before sending it to the model.

Contents of the Static Context File

The static domain context file contained canonical product family names, feature identifiers, and representative value samples for a selected subset of luminaire product features. In total, the file included roughly 23 000 tokens of text content. This amount was chosen as a practical compromise: it is large enough to cover all the domain information needed for every query in the Query Set 2 (and some additional examples), but still feasible to process within token and cost constraints. The complete file is available in the accompanying GitHub repository[8].

Advantages and Limitations

The static domain context approach is straightforward to implement for small or moderately sized datasets. Once compiled, the file can be reused to answer a wide range of questions without additional retrieval steps or database access. However, this simplicity also introduces several important limitations.

First, every prompt sent to the model contains the entire context, even when only a fraction of it is relevant to the current query. In other words, the static file cannot adapt its contents to the current query, since it always provides the same information regardless of what is being asked. This increases token usage and may reduce accuracy if the model's attention becomes distracted by unrelated details.

The method also scales poorly as the database grows. Adding new product families, features, or value samples requires regenerating the entire text file, and the prompt size quickly becomes impractical for large or frequently updated datasets.

Despite these drawbacks, the static domain context configuration serves as a useful reference point. It provides a controlled baseline for evaluating how the retrieval-augmented approach affects accuracy, latency, and token efficiency, and it demonstrates the practical upper limit of prompt size and processing cost within

[8] `https://github.com/anuutila/Evaluating-LLM-Based-Cypher-Query-Generation.git`

this experimental setup.

3.4.4 Dynamic Domain Context (RAG)

To address the limitations of the static domain context configuration, a second experimental setup was developed that retrieves only the relevant contextual information dynamically for each individual query. The goal was to reduce redundancy in the prompt and make the system more scalable and maintainable when applied to larger or evolving datasets like Summium CPQ's database. This configuration introduces a simple RAG-based workflow that constructs the Cypher-generation prompt at runtime based on the luminaire product models and features mentioned in the input query.

The goal of this setup was not to develop a full-fledged KBQA system, but to examine how the use of dynamically retrieved, query-specific prompts affects performance. In particular, the experiments aimed to measure whether the significantly smaller and more focused prompts produced by the retrieval mechanism lead to differences in accuracy, latency, and cost compared to the static approach, in which the model is provided with all domain context at once.

Experimental System Extension and Workflow

To implement this retrieval capability, the backend was extended with additional Neo4j and LLM calls that take place before Cypher generation. Figure 3.5 illustrates the ten-step interaction sequence between the user, the backend (Spring Boot and Spring AI), the Neo4j graph database, and the Azure-hosted LLM. The chat UI shown in the figure is optional and was not used in the automated experimental runs.

The workflow proceeds as follows:

1. The user submits a natural-language query through the chat interface (or, in

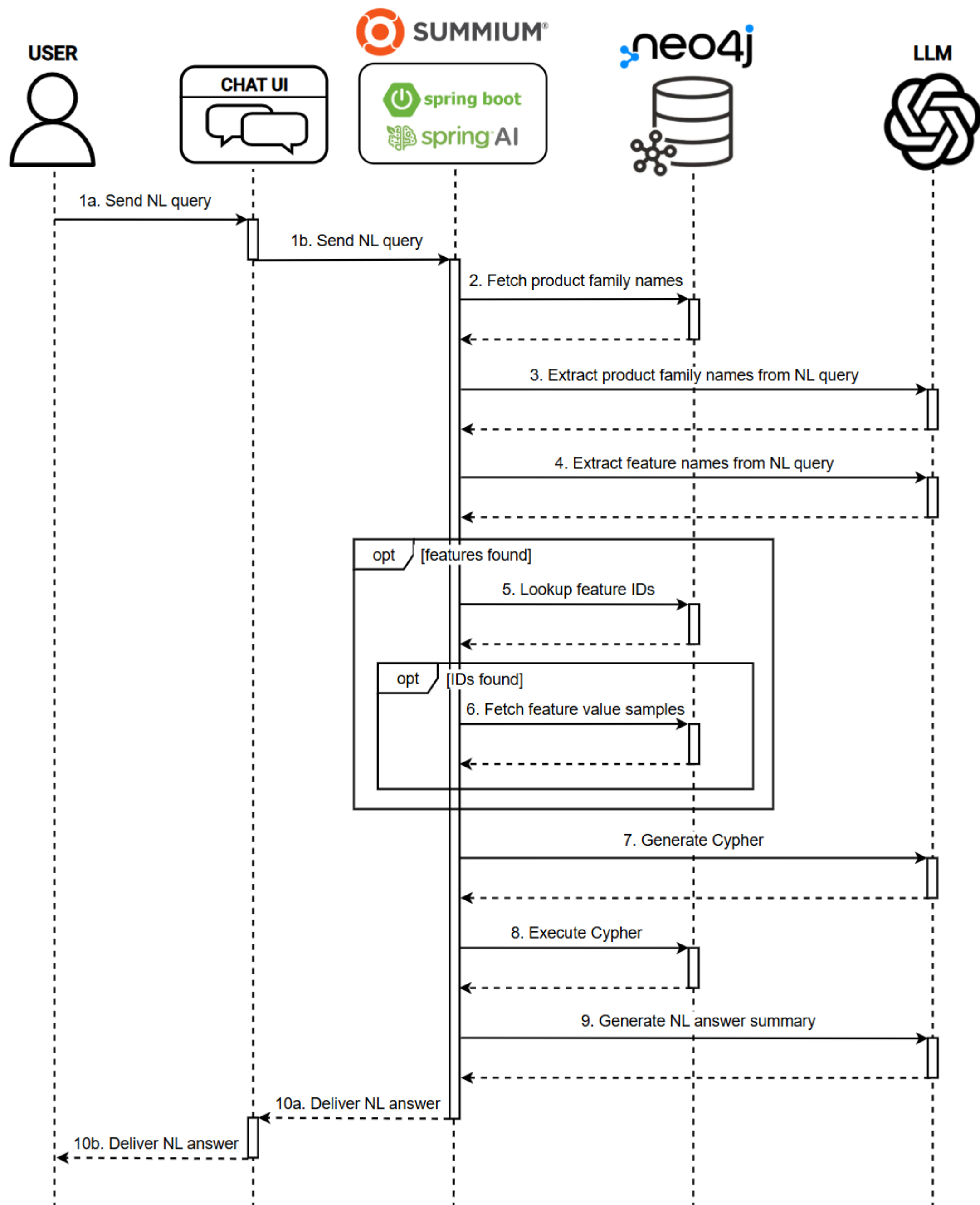

Figure 3.5: Sequence diagram of the retrieval-augmented workflow used in the Query Set 2 experiments. The workflow includes three additional Neo4j retrieval calls and two LLM calls before the Cypher generation step. Optional branches are executed only when relevant entities have been identified in the previous step.

automated runs, read from a file containing the query set).

2. The backend fetches a list of all the valid product family names from Neo4j to serve as a reference context.

3. The query text is sent to the LLM, which extracts any product family names mentioned in the natural-language query and maps them to the canonical names retrieved in Step 2.

4. A second LLM call extracts possible luminaire product feature entities (e.g., *length*, *color*, *power*) from the same query.

5. If features are identified, the backend utilizes them to perform a lookup in Neo4j using a full-text index[9] that links parameter names (the user-facing luminaire feature names) to their corresponding external feature IDs. The full-text search returns the feature ID with the best match to the search term.

6. For each retrieved feature ID, the backend fetches representative parameter-value samples from Neo4j to illustrate how that feature's data appears in the database. If a product family was identified earlier, only samples related to that product family are retrieved.

7. The retrieved contextual information (canonical product families, feature IDs, and value samples) is compiled into a concise text block that replaces the large static domain context file. This block is appended to the Cypher-generation prompt along with the system instructions and schema. The complete prompt is sent to the LLM, which generates the Cypher query.

8. The generated Cypher query is executed on the Neo4j database, and the resulting records are returned to the backend.

[9] `https://neo4j.com/docs/cypher-manual/4.3/indexes-for-full-text-search/`

9. Finally, the query result and original question are combined into a summary prompt that the LLM reformulates into a natural-language answer

10. The natural language answer is sent to be displayed in the chat UI (or, in automated runs, written to an output file).

Implementation Rationale

The retrieval logic was designed following the principles of *LLM workflows* as defined by Anthropic [39]. In this distinction, *workflows* are orchestrations of models and tools that follow predefined code paths, while *agents* are systems where the model autonomously decides which tools to use and how to perform each step. As Anthropic recommends, beginning with well-defined workflows provides greater reliability and control, and additional complexity should only be introduced when necessary.

The workflow implemented in this thesis follows that philosophy. Retrieval was carried out through a small sequence of predefined LLM and Neo4j calls. No vector databases or embedding-based similarity searches were used.

The implementation was informed by best practices for retrieval-augmented querying and workflow orchestration, including Spring AI's documentation on effective agents [42], LangChain's SQL QA tutorial [43], and LangGraph workflow examples [44]. The design was intentionally kept lightweight to isolate the effects of dynamic context retrieval itself. Therefore, the experiments focus on how dynamically generated, smaller, and more relevant prompts influence model performance in Cypher query generation rather than on optimizing the retrieval pipeline as a standalone system.

Advantages and Limitations

The dynamic context workflow offers several advantages compared to the static baseline. It produces smaller and more focused prompts, which reduces token usage and cost. It also scales better to larger or evolving databases, since only the relevant context is retrieved at runtime. Furthermore, it supports a more maintainable architecture: new features or product families can be added to the graph without requiring any manual updates to a static text file.

On the other hand, this configuration also introduces more moving parts and requires greater implementation effort due to the additional retrieval and orchestration logic. The overall performance depends on how reliably the intermediate LLM steps identify the correct features and product families. If these extraction steps fail, the retrieved context may be incomplete or irrelevant, which can affect the quality of the generated Cypher queries. Despite these challenges, the workflow provides a more flexible and scalable foundation for retrieval-augmented Cypher generation and enables direct comparison against the static baseline in terms of accuracy, efficiency, and token usage.

3.4.5 Experimental Variables

The experimental variables for Query Set 2 largely follow the same structure as in Query Set 1, with the addition of a new key independent variable that distinguishes the two domain context configurations introduced in this section. Only the aspects that differ from the earlier setup are described here.

Independent Variables

The following independent variables were defined for the Query Set 2 experiments:

- **Domain context method:** The main independent variable in Query Set 2

was the method used to supply the required domain context to the model. Two configurations were tested:

- *Static:* The model was prompted with the full precompiled context file described in Section 3.4.3.
- *Dynamic (RAG):* The model received only the relevant, query-specific context retrieved dynamically at runtime through the retrieval-augmented generation workflow described in Section 3.4.4.

This variable isolates the effect of prompt size and relevance on performance.

- **Prompt style:** Both *zero-shot* and *few-shot* prompting conditions were tested. The few-shot examples were redesigned for Query Set 2 to represent the new feature-based query types and to collectively cover all ten unique query tags introduced in Appendix B. The examples are available in the accompanying GitHub repository[10].
- **Query language:** All queries in Query Set 2 were executed in English only. The experiments with Query Set 1 already confirmed that language choice had virtually no effect on accuracy.

Dependent Variables

The same evaluation metrics were used as in Query Set 1, with one modification to the status categorization:

- **Status:** Each query was assigned one of three outcome categories — `OK`, `WRONG_RESULT`, or `SYNTAX_ERROR`. The `NO_ATTEMPT` status used in Query Set 1 was removed. Based on the findings from Query Set 1 experiments, the instruction that allowed the model to decline Cypher generation was removed

[10] `https://github.com/anuutila/Evaluating-LLM-Based-Cypher-Query-Generation.git`

for Query Set 2 to prevent the models from becoming overly cautious. Since all Query Set 2 queries were solvable with the provided context, the guardrail was considered unnecessary for the controlled evaluation.

- **Accuracy:** The proportion of successful (`OK`) queries out of all attempts, reported as both a percentage and a raw count (e.g., 30 / 42).
- **Total time:** The macro-average end-to-end latency, including all Neo4j and LLM calls in the retrieval and generation phases.
- **Prompt tokens:** The macro-average number of input tokens sent across all four LLM calls in the workflow.
- **Completion tokens:** The macro-average number of tokens generated by the model across all four LLM calls.

All experiments were executed automatically using the same batch evaluation framework described earlier, adapted to the extended retrieval-augmented workflow. The results directly compare the two context-provision methods in terms of accuracy, efficiency, and token usage.

3.5 Prototype User Interface

In addition to the automated evaluation setup used in the experiments, a *prototype user interface* was developed for demonstration purposes. This interface was not used in the experiments, but it allowed the system to be explored in a more intuitive way and illustrated how natural language querying could be integrated into future products. The interface was implemented using Thymeleaf[11], a server-side Java template engine, but since it was created only for demonstration purposes, it is not described in further technical detail.

[11]`https://www.thymeleaf.org/`

Screenshots and additional details about the prototype UI are provided in Appendix I.

4 Results and Evaluation

4.1 Query Set 1 Results

The first set of experiments (Query Set 1) tested the Cypher query generation capabilities of gpt-4o and gpt-5-mini. The LLMs were only given natural language queries from Query Set 1 (Table A.1), minimal system instructions (Appendix C) with and without generated example queries (Appendix E) and the graph database schema (Figure 3.1) V1[1] or V2 (Appendix D) as context.

Each query in the query set was executed three times under every combination of the independent variables to account for the nondeterministic behavior of the LLMs. This repetition results in 108 total query runs per experimental configuration (36 queries $\times$ 3 runs). The results are reported as aggregated performance measures for each experimental configuration.

Overall Results

Table 4.1 summarizes the aggregated results across all twelve experimental configurations of Query Set 1. Each row corresponds to one combination of LLM, prompting strategy, and schema version (see Section 3.3.4 for definitions of the independent and dependent variables). The overall trends are also visualized in Figure 4.1, which plots accuracy and total time across all configurations.

[1] `https://github.com/anuutila/Evaluating-LLM-Based-Cypher-Query-Generation.git`

Table 4.1: Query Set 1 Overall Results. Colors use a gradient: worst → yellow, best → green.

Conf.	Model	Prompting	Schema	Accuracy	Total time	Prompt tokens	Completion tokens
1	GPT-4o	zero-shot	V1	36% (39/108)	1667 ms	4241	89
2	GPT-4o	zero-shot	V2	72% (78/108)	1683 ms	929	111
3	GPT-4o	few-shot	V1	91% (98/108)	1422 ms	4689	112
4	GPT-4o	few-shot	V2	93% (100/108)	1375 ms	1246	115
5	GPT-5-mini (rl)	zero-shot	V1	94% (101/108)	6505 ms	4304	604
6	GPT-5-mini (rl)	zero-shot	V2	95% (104/108)	5450 ms	934	478
7	GPT-5-mini (rl)	few-shot	V1	98% (106/108)	5674 ms	4634	456
8	GPT-5-mini (rl)	few-shot	V2	98% (106/108)	5457 ms	1249	396
9	GPT-5-mini (rh)	zero-shot	V1	99% (107/108)	30562 ms	4311	3323
10	GPT-5-mini (rh)	zero-shot	V2	97% (105/108)	21289 ms	936	2432
11	GPT-5-mini (rh)	few-shot	V1	100% (108/108)	29681 ms	4638	2747
12	GPT-5-mini (rh)	few-shot	V2	100% (108/108)	19354 ms	1253	1915

Accuracy increased systematically with stronger model configurations, schema version 2 and few-shot prompting, though the effects varied considerably depending on the model. The weakest setup (Config 1, GPT-4o, zero-shot, schema V1) produced an accuracy of only 36%, which was expected to be poor, but was still strikingly low. In contrast, the strongest setups (Configs 11–12, GPT-5 mini with high reasoning, few-shot, both schema versions) achieved perfect accuracy.

The impact of schema representation and prompting style depended on the model. For GPT-4o, using schema V2 and few-shot prompting produced dramatic accuracy gains (from 36–72% to 91–93%), while total response times remained roughly the same. For GPT-5-mini with low reasoning effort, however, the same changes produced only marginal accuracy gains (94–95% to 98%) and minor efficiency improvements. For GPT-5-mini with reasoning effor set to high, the accuracy was already almost perfect, and the main effect of schema V2 was a substantial reduction in response times (from 30 seconds to 20 seconds). The schema representation was also the main factor influencing prompt token usage: prompts using the

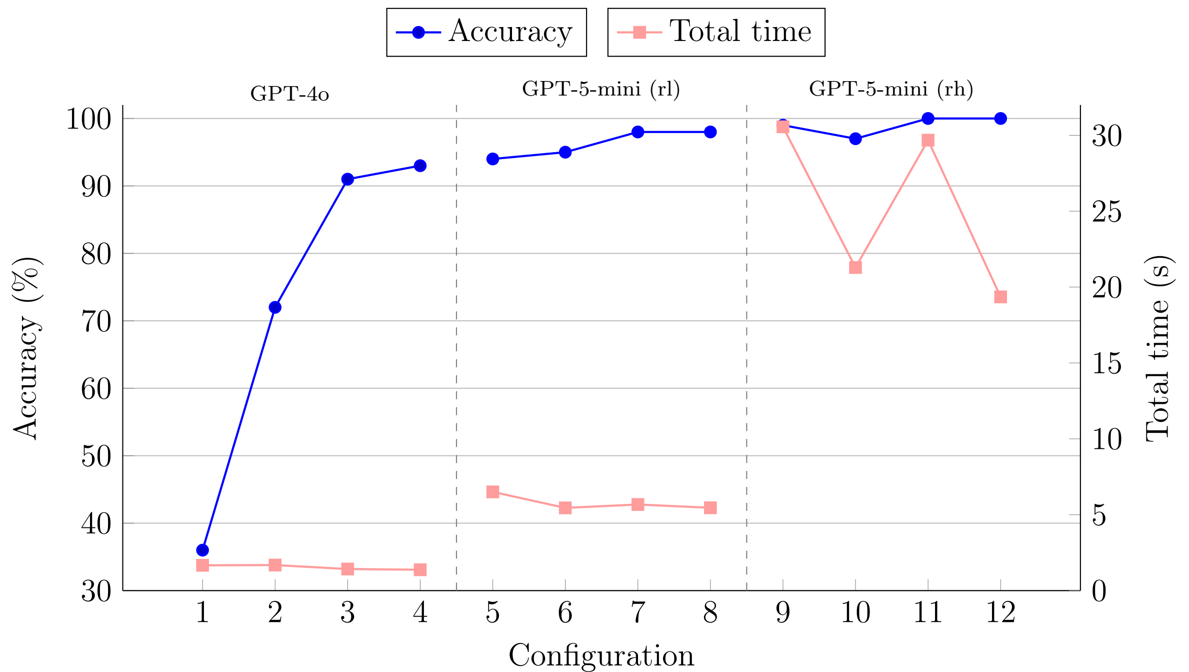

Figure 4.1: Accuracy and total time across all experimental configurations (Query Set 1).

verbose JSON schema (V1) required around 4 000 tokens, whereas the condensed text schema (V2) reduced this to under 1 000.

Few-shot prompting improved accuracy consistently across all models, though the impact varied. For GPT-4o, for example, it was essential, raising accuracy by even more than 50 percentage points. For GPT-5-mini, however, it only provided a modest improvement in accuracy (1–3 percentage points) and minor changes in response times. A small exception was observed (Config 6 vs. Config 8), where the few-shot configuration was slightly slower, though this difference was negligible.

A clear trade-off became obvious between reasoning effort and efficiency. With reasoning effort set to *low*, GPT-5-mini already achieved 94–98% accuracy, with total times of around 5–6 seconds per query. Increasing the reasoning effort to *high* raised the accuracy to 97–100%, but this came at the cost of response times of 20–30

seconds, i.e., 4–6 times slower. Within the high reasoning runs, schema version had a significant impact on total time, with schema V2 reducing response times by up to a third. Configuration 8 (GPT-5-mini with low reasoning, few-shot and schema V2) appears to offer the best balance, providing 98% accuracy alongside much lower total time and token usage compared to the high reasoning runs.

Figure 4.2 breaks down the total response time into *query generation* (t_1) and *summary generation* (t_3), omitting *execution time* (t_2), which was consistently negligible at 50–100 ms. The results show that nearly all of the latency originates from the two language model steps. With GPT-4o, both steps took less than one second. However, GPT-5-mini required several seconds when reasoning effort was set to *low* and over ten seconds when set to *high*. The final natural language summary could potentially always be produced by a lighter faster model, such as GPT-4o, since this step may not require the same depth of reasoning as Cypher query generation. This has not been systematically tested, but it provides a potential method for reducing total response times.

Overall, these results suggest that schema simplification and few-shot prompting are beneficial for all models, although the extent of this benefit varies depending on the strength of the model. For weaker models, such as GPT-4o, these techniques are essential for achieving reasonable accuracy. However, for stronger models such as GPT-5-mini, which have high reasoning capabilities, the benefits of schema and prompting are primarily seen in efficiency, as accuracy is already near-perfect.

Accuracy by Query Type

Table 4.2 illustrates the accuracy of Query Set 1 for different query types and complexity levels. As expected, the simplest lookup queries (L1) achieved a 100% success rate. Accuracy remained high for single-hop and multi-hop queries, mostly above 90%, but decreased gradually with increasing complexity. Analytics queries

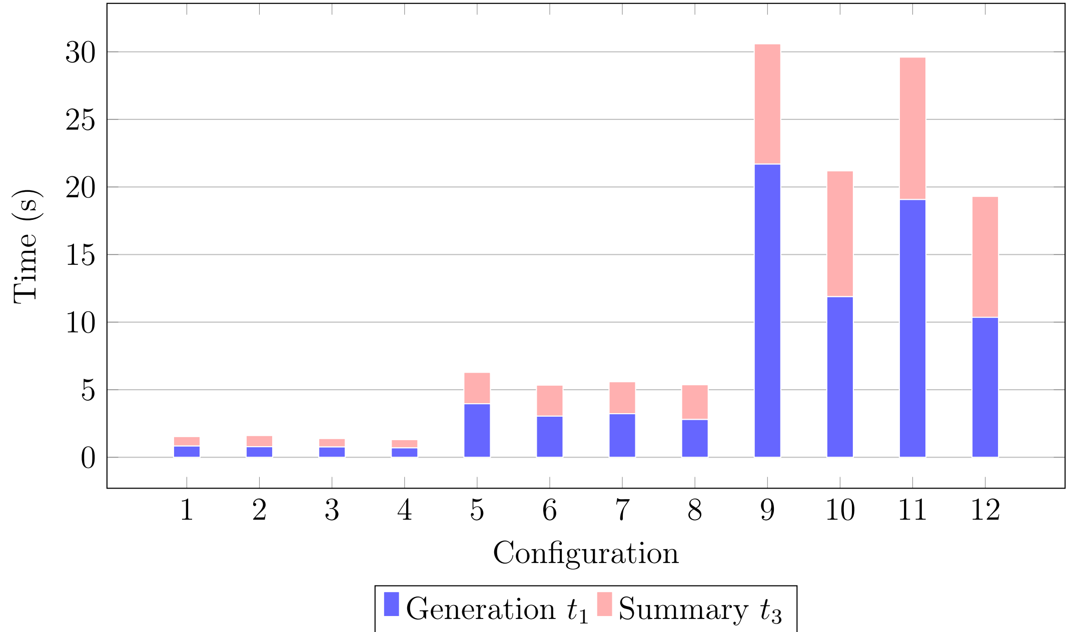

Figure 4.2: Breakdown of response times into query generation (t_1) and summary generation (t_3) across the twelve configurations. Execution time (t_2) is omitted as it remained consistently negligible (50–100 ms).

Table 4.2: Accuracy of Query Set 1 by query type and complexity. Cells show percentage and (OK/total). Colors use a gradient: worst → yellow, best → green.

Type	**L1**	**L2**	**L3**	**Overall**
Lookup	100% (144/144)	–	–	100% (144/144)
Single-hop	96% (138/144)	92% (133/144)	–	94% (271/288)
Multi-hop	–	95% (137/144)	90% (129/144)	92% (266/288)
Aggregation	–	95% (137/144)	68% (98/144)	82% (235/288)
Analytics	–	85% (123/144)	84% (121/144)	85% (244/288)
Overall	98% (282/288)	92% (530/576)	81% (348/432)	90% (1160/1296)

performed slightly worse, with accuracies of around 84–85%. The level 3 aggregation queries were clearly the most difficult, with an accuracy rate of only 68%,

significantly below the other categories. These results suggest that, although large language models are highly effective at generating simple Cypher queries, they still struggle with more demanding tasks involving the combination of multiple hops and aggregation operations.

Accuracy by Language

As shown in Table 4.3 and Figure 4.3, no substantial performance difference was observed between English and Finnish queries. Across all three model configurations, the accuracy for Finnish queries was within 1–3 percentage points of English, with no consistent trend favoring one language. It is reasonable to assume that the minor variations detected between the two languages can be considered as normal variation. The differences would likely decrease if the sample size was larger. When aggregated across the entire query set, the results were strikingly even: English achieved 89.4% accuracy and Finnish 89.7%. This suggests that the models generate Cypher queries with almost identical accuracy when the user queries are in English or Finnish.

Table 4.3: Overall accuracy of Query Set 1 by language

Language	**Accuracy**
English	89.4% (579/648)
Finnish	89.7% (581/648)

It is also worth noting that Query Set 1 was designed so that all natural language queries strictly used terminology drawn directly from the graph schema. This condition applies precisely for the English queries, since the schema labels themselves are in English. The Finnish queries, however, were created as direct translations of the English queries, and therefore do not use the English schema terminology directly. Despite this mismatch, the models were able to interpret the translated terms correctly and still generate valid Cypher queries at virtually the same accuracy as for

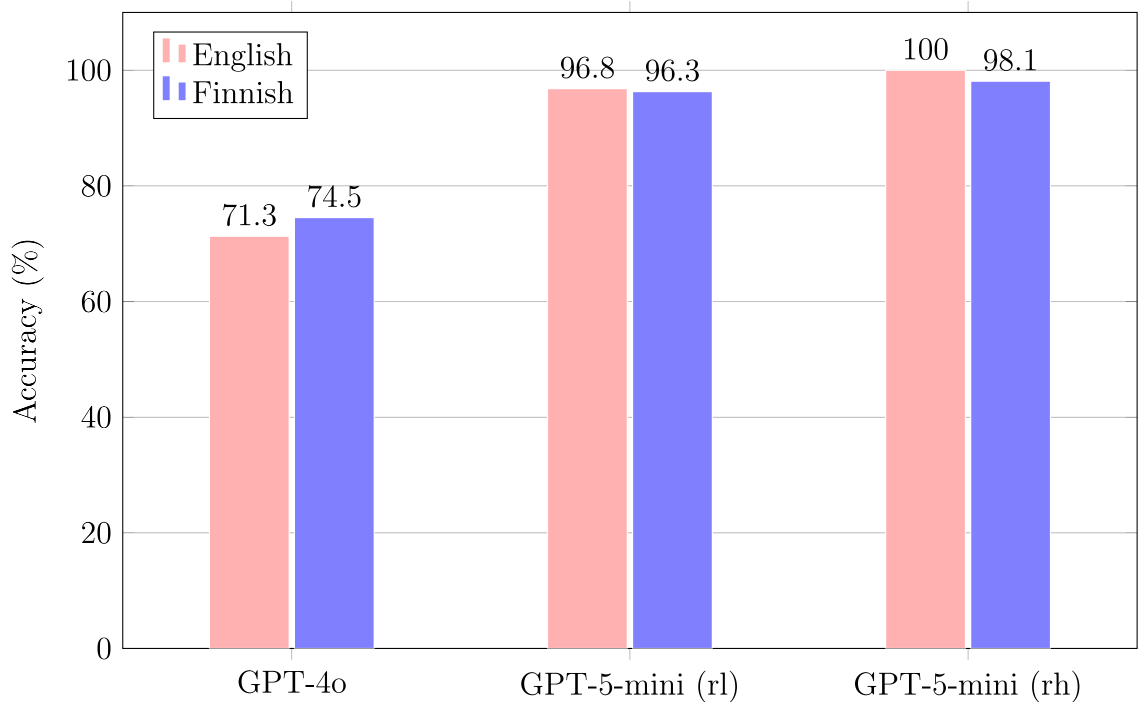

Figure 4.3: Overall accuracy by model and language for Query Set 1.

English. This demonstrates the models' ability to map translated terms correctly to the underlying English-based schema context.

When examining the results from individual configurations, most showed less than a two-percentage-point difference between English and Finnish. Only three configurations (Config 1, 2, and 10 from Table 4.1) had a larger gap of more than three percentage points. There was no evidence that any of the failures were caused by misinterpretation of translated terminology.

These findings may generalize to other widely spoken languages as well. Finnish is a relatively small language that is often considered challenging for NLP systems. Its performance on par with English in this setup suggests that other major languages are also likely to achieve similar accuracy for Cypher query generation with current LLMs.

An example of the full per-query results is provided in Appendix H. The com-

plete set of detailed results for all experimental configurations is available in the accompanying GitHub repository[2].

Failure Analysis

In addition to overall accuracy, it is important to examine how and why the models failed. This analysis provides insights into the limitations of LLM-based query generation and helps to explain the observed differences in performance between models. The failures were categorized into three types: `NO_ATTEMPT`, `SYNTAX_ERROR`, and `WRONG_RESULT`. Appendix F provides the full distribution of failure types for each individual query. As well as the quantitative results, qualitative evidence from the generated answers was also reviewed to identify common patterns behind these errors.

Table 4.4 shows the overall distribution of failure types in Query Set 1. Out of a total of 136 failures, nearly half (48.5%) were cases where the model made no attempt to generate a query. Another 47.8% of failures were syntactically valid queries that returned an incorrect answer. Syntax errors were rare, accounting for only 3.7% of failures. While `NO_ATTEMPT` and `SYNTAX_ERROR` failures are straightforward to detect and can simply prompt the user to retry in a chat interface, `WRONG_RESULT` failures are more problematic: the system produces a syntactically valid query and returns an answer, but there is no way for the system to flag that the answer is incorrect. This makes `WRONG_RESULT` failures the most critical source of error, since users cannot easily distinguish between correct and incorrect answers without manually verifying them.

Figure 4.4 shows the breakdown of failures across models. The results highlight the clear difference between GPT-4o and GPT-5-mini. GPT-4o was responsible for the overwhelming majority of failures, with 66 `NO_ATTEMPT` and 49 `WRONG_RESULT`

[2] `https://github.com/anuutila/Evaluating-LLM-Based-Cypher-Query-Generation.git`

Table 4.4: Failure distribution in Query Set 1 experiments.

Failure type	**Failure count**	**Share of failures**	**Share of all attempts**
NO_ATTEMPT	66	48.5%	5.1% (66/1296)
SYNTAX_ERROR	5	3.7%	0.4% (5/1296)
WRONG_RESULT	65	47.8%	5.0% (65/1296)
Total	136	100%	10.5% (65/1296)

cases. In contrast, GPT-5-mini with the reasoning effort set to *low* eliminated NO_ATTEMPT failures entirely. It also reduced WRONG_RESULT cases to 14. The reasoning effor *high* setting further lowered WRONG_RESULT cases to just 2. Both model configurations produced only 1-2 SYNTAX_ERRORs. GPT-4o often failed to utilize the schema or examples, resulting in NO_ATTEMPTs. In contrast, GPT-5 interpreted the provided context more effectively, producing more consistent query attempts and fewer failures.

The choice of prompting strategy also influenced failures. For GPT-4o, zero-shot runs (Configs 1 and 2) produced many NO_ATTEMPTs (48 and 16), while few-shot runs (Configs 3 and 4) produced few (2 and 0). A similar trend was visible in the incorrect results: zero-shot prompting yielded 20 and 13 WRONG_RESULT cases, while few-shot prompting reduced these to 8 and 8. These results demonstrate that the prompting strategy was the primary factor contributing to the variation. Few-shot prompting was effective in reducing both NO_ATTEMPTs and WRONG_RESULTs.

A qualitative analysis of failure cases revealed the most common causes of errors in each of our three failure categories:

- NO_ATTEMPT: the most frequent reason was the model's refusal or inability to generate a query based on the provided context. A typical response looked as follows:

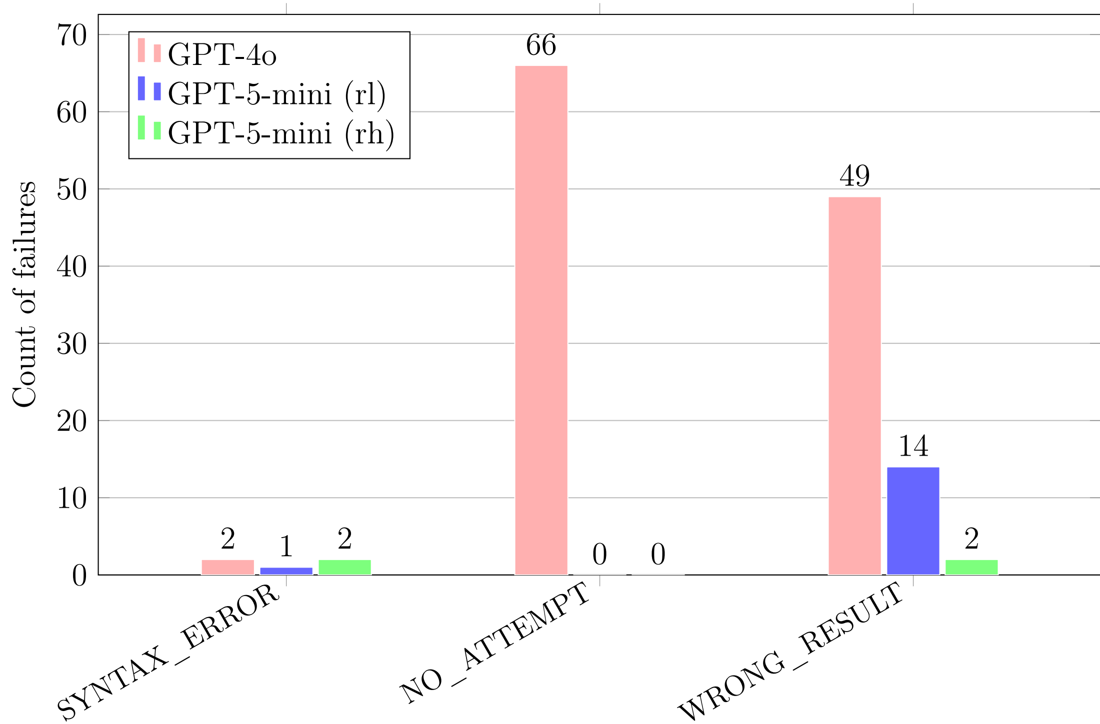

Figure 4.4: Distribution of failure types by model in Query Set 1 experiments.

> "Sorry, I couldn't generate a Cypher query for that. The schema does not provide a direct relationship or property to count distinct products within an offer."

This reflects the model's difficulty in handling multi-hop relationships. The connection in question was not explicitly represented in the schema as a single relationship, but rather, it could be inferred by combining multiple existing relationships. GPT-4o often failed to make such deductions, even though the information was implicitly present. However, GPT-5-mini, with its reasoning abilities, was able to reliably perform these inferences.

It is worth noting that these outcomes were influenced by a guardrail instruction in the system prompt (see Appendix C). This instruction allowed the model to decline query generation if it determined that the available context was insufficient. While this rule was redundant in the controlled experiments,

since all queries were solvable, it demonstrated how conservative behavior can very easily result from the phrasing of instructions alone. The guardrail affected GPT-4o in particular, causing it to decline more uncertain cases. GPT-5-mini, however, was unaffected. Although we cannot know with certainty how many of these failures would have succeeded without the guardrail instruction in place, many cases showed that the same model configuration produced both successful and `NO_ATTEMPT` outcomes for the same query. This indicates that the model often could generate the correct Cypher but opted not to do so because the guardrail encouraged conservative behavior under uncertainty.

- `SYNTAX_ERROR`: Only five syntax errors were recorded in total, indicating that syntax rarely caused problems. Two of these errors occurred when the model attempted to use a variable before it was defined (e.g., summing a variable introduced later in the query). Two other errors occurred when the model attempted to call non-existent date functions, such as year() or month(), which are not part of Cypher 5. The final error occurred when a `"-"` character was missing, which broke an otherwise valid query.

- `WRONG_RESULT`: These errors were more varied, but they still showed recurring patterns. A frequent cause was misunderstandings about the schema, particularly confusion about the direction of relationships between nodes. For example, GPT-4o frequently reversed the direction of the `RECEIVED_BY` relationship between Offer and Participant nodes, resulting in queries that executed but returned no participants.

  Additionally, there was a common error involving the misuse of node variables in aggregations. In the correct query, aggregation is performed over a property, such as `p.company`, to ensure that the results are grouped by company. However, in erroneous queries, aggregation was applied to the node variable `p`

itself, producing wrong results. An example pair of generated queries for *Q17*:

```
// correct
MATCH (o:Offer)-[:RECEIVED_BY]->(p:Participant)
WHERE o.date >= date("2024-07-01") AND o.date <= date("2024-12-31")
MATCH (o)-[:CONTAINS]->(c:Content)
WITH p.company AS company, o, SUM(c.total_price) AS offerTotal
WITH company, AVG(offerTotal) AS avgContentPrice
RETURN company, avgContentPrice
ORDER BY avgContentPrice DESC
LIMIT 10

// wrong
MATCH (p:Participant)<-[:RECEIVED_BY]-(o:Offer)
WHERE o.date >= date("2024-07-01") AND o.date <= date("2024-12-31")
MATCH (o)-[:CONTAINS]->(c:Content)
WITH p, o, SUM(c.total_price) AS offerTotal
WITH p, AVG(offerTotal) AS avgContentTotalPrice
RETURN p.company AS company, avgContentTotalPrice
ORDER BY avgContentTotalPrice DESC
LIMIT 10
```

Another type of mistake occurred in nested content structures. In multiple occasions, the model calculated the total price of a Content node and its Subcontent, then added them together. However, since the content already encapsulates its subcontent, this resulted in double-counting and inflated totals. The correct query summed only the `Content.total_price`.

These examples illustrate that the incorrect results were not random. Rather, they reflected repeated misunderstandings about how the schema entities should be combined. Such recurring issues could potentially be mitigated by provid-

ing clarifications or usage notes in the system prompt to guide the model away from these mistakes.

4.2 Query Set 2 Results

The second set of experiments (Query Set 2) evaluated how large language models generate Cypher queries when the domain context is provided either as a static, precompiled text block or dynamically through retrieval from the underlying Neo4j database. Unlike in Query Set 1, where the models received only the graph schema and schema-aligned natural language queries, Query Set 2 aimed to assess how well the models can interpret and use domain-specific information when it is supplied in these two alternative forms. The same two models were tested: GPT-4o and GPT-5-mini with low and high reasoning configurations. Each model was evaluated under zero-shot and few-shot prompting conditions, resulting in twelve experimental configurations in total (see Section 3.4.5 for details on the independent variables). Based on the results of Query Set 1, which showed Schema V2 to be consistently superior to Schema V1, all Query Set 2 experiments were conducted using only Schema V2.

Each natural language query from Query Set 2 was executed three times under every combination of the independent variables, yielding 42 individual executions per configuration (14 queries $\times$ 3 runs). The results are reported as aggregated performance measures for each experimental configuration.

Overall Results

Table 4.5 and Figure 4.5 summarize the results of all twelve Query Set 2 configurations. Each configuration combines a model, a prompting style, and a domain-context provisioning method. The overall accuracy ranged from 26% to 95%, and

Table 4.5: Query Set 2 Overall Results. Colors use a gradient: worst → yellow, best → green.

Conf.	Domain context	Model	Prompting	Accuracy	Total time (ms)	Prompt tokens	Completion tokens
1	static	GPT-4o	zero-shot	43% (18/42)	3684	24298	232
2	static	GPT-4o	few-shot	81% (34/42)	2603	25260	286
3	static	GPT-5-mini (rl)	zero-shot	76% (32/42)	26918	24336	1453
4	static	GPT-5-mini (rl)	few-shot	83% (35/42)	16006	25280	790
5	static	GPT-5-mini (rh)	zero-shot	26% (11/42)	139777	24270	8988
6	static	GPT-5-mini (rh)	few-shot	91% (38/42)	61679	25281	5187
7	dynamic	GPT-4o	zero-shot	74% (31/42)	2923	2576	247
8	dynamic	GPT-4o	few-shot	79% (33/42)	3788	3775	283
9	dynamic	GPT-5-mini (rl)	zero-shot	67% (28/42)	28557	2641	1394
10	dynamic	GPT-5-mini (rl)	few-shot	88% (37/42)	18270	3673	976
11	dynamic	GPT-5-mini (rh)	zero-shot	38% (16/42)	152504	2522	8479
12	dynamic	GPT-5-mini (rh)	few-shot	95% (40/42)	81928	3638	5740

total latencies spanned from a few seconds to well over two minutes, depending primarily on the model and reasoning effort. The highest accuracy was achieved by GPT-5-mini (rh) with the dynamic context and few-shot prompting (Config 12), which produced correct Cypher queries for 95% of the inputs. The weakest configuration was the same model with the static context and zero-shot prompting (Config 5), which achieved only 26%. The lighter GPT-4o model produced far lower absolute latencies but also lower accuracy: between 43% and 81% with static context and 74% to 79% with dynamic domain context. For the GPT-5-mini (rl) variant, accuracy stayed between 67% and 88% across the two context provision methods.

Few-shot prompting improved performance across all models, though the extent of improvement varied considerably. This effect was most noticeable for the high-reasoning variant of GPT-5-mini, which demonstrated the greatest absolute improvement of all configurations. With the static context, accuracy increased from 26% to 91% (Configs 5 to 6), and with the dynamic context, it increased from 38% to

95% (Configs 11 to 12). In both cases, few-shot prompting transformed the weakest configuration into the strongest. The low-reasoning variant improved more moderately, rising from 76% to 83% with the static context and from 67% to 88% with the dynamic context (Configs 3 to 4 and 9 to 10). GPT-4o only showed a large effect in the static case (43% to 81%), while the improvement with the dynamic context remained small (74% to 79%). Few-shot prompting with GPT-5-mini also resulted in a 40-55% reduction in the system's total end-to-end latency.

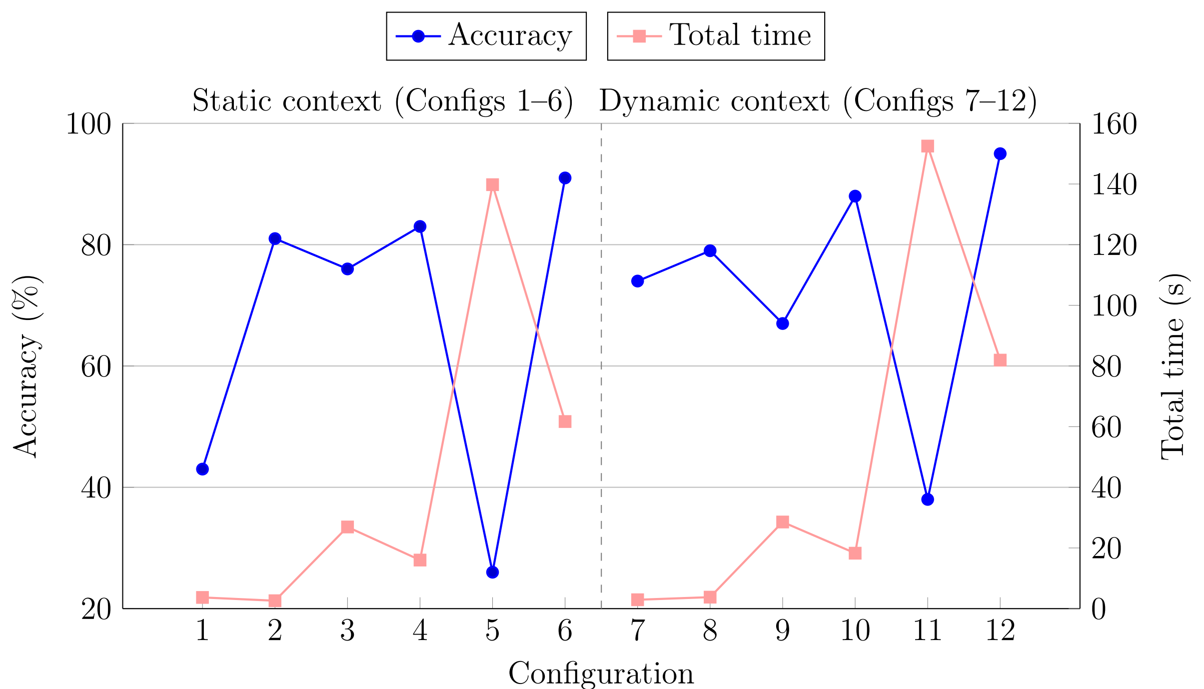

Figure 4.5: Accuracy and total time across all experimental configurations (Query Set 2).

This pattern suggests that GPT-5-mini's high reasoning mode is very sensitive to the prompt design. When left unguided in zero-shot configurations, the model's reasoning process appears to diverge from the intended query-generation objective, amplifying irrelevant or speculative reasoning chains. However, when anchored by concrete examples, that same reasoning capability becomes advantageous, producing

the best overall accuracy among all tested configurations. The difference between configurations 5 and 6 (or 11 and 12) illustrates how explicit guidance stabilizes the model's reasoning, whereas its absence severely reduces accuracy in tasks that require strict precision and adherence to rules.

The comparison between static and dynamic domain contexts revealed that the two approaches produced relatively similar accuracy, but the efficiency difference was substantial. Static prompts contained the large pre-compiled subset of domain data and averaged around 25 000 tokens per input. In contrast, dynamic prompts constructed through the retrieval of only the relevant domain information averaged just around 3 000 tokens. This corresponds to a near 90% reduction in prompt size. The static prompt used in these experiments was already a truncated version of the full dataset. A complete prompt including all product family and feature information would grow proportionally with the database. Although the static approach occasionally produced slightly higher accuracy in this controlled setup (Configs 2 and 3), it is not an easy-to-maintain, scalable alternative for real-world deployments where the knowledge base is large and continues to grow. The dynamic approach achieves nearly the same or better accuracy with smaller, more maintainable prompts. These results align with recent long-context evaluations, which demonstrate that simply increasing prompt length does not guarantee reliable access to relevant information and that model performance often degrades as context grows [8], [9], [10].

Latency patterns followed the same overall hierarchy as accuracy, with large differences between the models and smaller differences between the context provision methods. GPT-4o completed all runs in about 2–4 seconds. GPT-5-mini (rl) required about 16–28 seconds per query, and GPT-5-mini (rh) took between 60 and over 150 seconds, depending on the configuration. Figure 4.6 illustrates how this time was distributed across the main processing stages. For all models, the Cypher generation phase was by far the largest contributor to latency, typically accounting

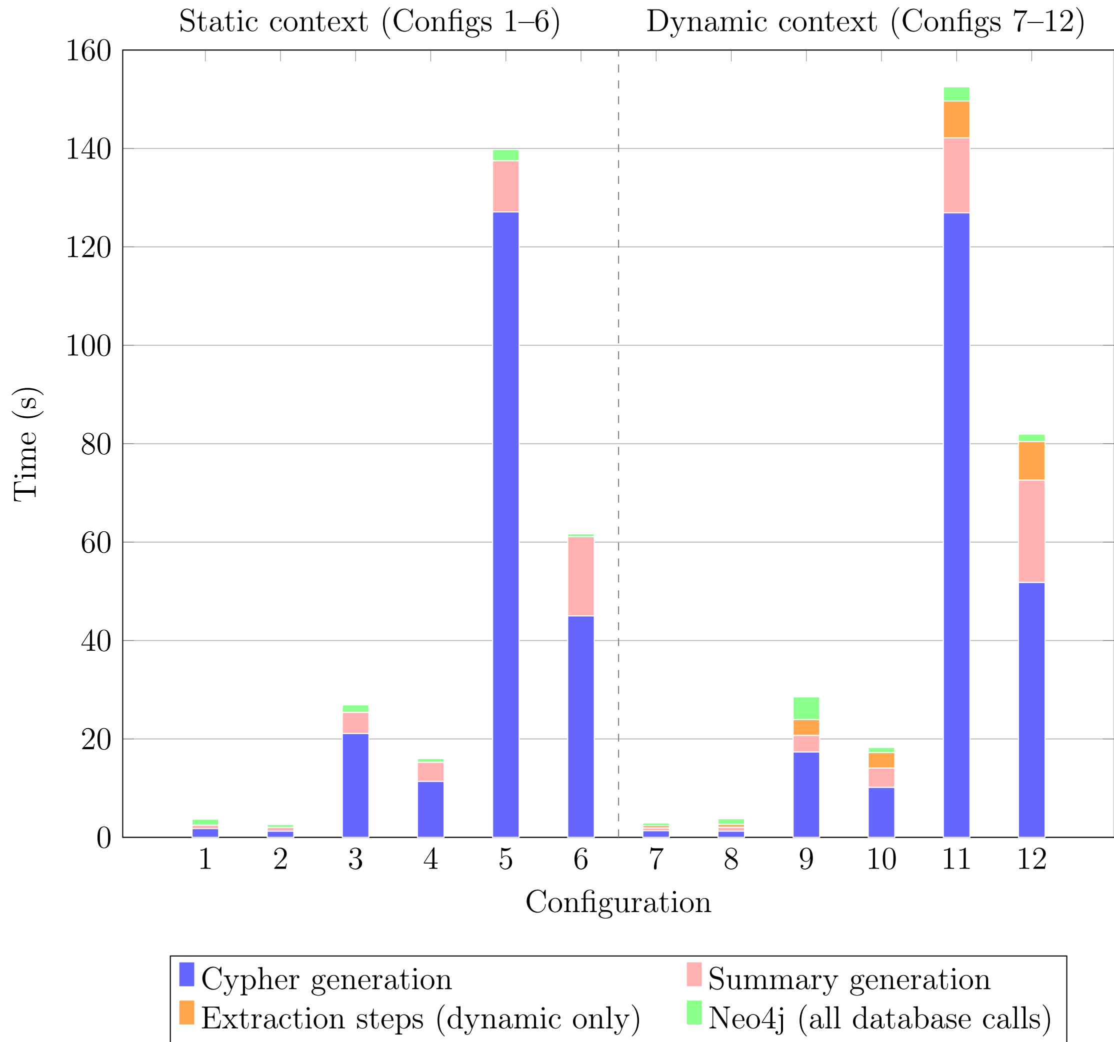

Figure 4.6: Latency breakdown for all twelve experimental configurations of Query Set 2. Each bar shows the average end-to-end latency divided into key processing stages. In the static context configurations (1–6), the Neo4j segment represents only the execution of the generated Cypher query. In the dynamic context configurations (7–12), an additional extraction segment appears that corresponds to the two LLM-based steps that identify product family and feature names before Cypher generation. The Neo4j segment in these dynamic runs aggregates all four separate database interactions. Across all configurations, Cypher generation dominates total latency, while database operations remain comparatively minor in duration.

for 70–90% of the total latency of each configuration. Summary generation was the second largest component, adding a few seconds in smaller models and over 15 seconds in the high-reasoning setups. The Neo4j segment was consistently minor: less than 2 s in nearly all runs. With the dynamic context, the additional extraction step introduced by the two preliminary LLM calls added between 0.5 seconds (GPT-4o) and approximately 8 seconds (GPT-5-mini-rh). Despite these additional steps, the dynamic context provision method remained competitive in terms of overall speed. This shows that the added retrieval and preprocessing overhead was almost negligible compared to the time consumed by the Cypher generation step.

Token consumption showed clear differences across configurations. As mentioned earlier, dynamic prompting required far fewer prompt tokens. Completion tokens primarily varied based on the model and reasoning setting. GPT-4o produced short completions of around 250–300 tokens. In contrast, GPT-5-mini generated much longer completions, especially in high-reasoning mode. For example, Config 5 produced almost 9 000 tokens, while Config 12 produced approximately 5,700. In these cases, the majority of the completion consists of reasoning tokens rather than the generated Cypher query output.

Overall, the results demonstrate that few-shot prompting improves accuracy across all models, dynamic domain context reduces prompt size and in most cases also increases accuracy, and total latency grows primarily with model complexity. GPT-5-mini's high-reasoning configuration with few-shot prompting remains the most accurate, but it is also the slowest and most costly. GPT-4o offers very fast responses at moderate accuracy. The dynamic few-shot setup with GPT-5-mini (rl) (Config 10) provides the best balance between accuracy (88%), latency (18 s), and token consumption. These findings indicate that the RAG-based dynamic context provisioning generally achieves equal or higher accuracy than static full-context prompting while requiring far fewer tokens.

Failure Analysis

Failures in Query Set 2 were classified into two types: `WRONG_RESULT` and `SYNTAX_ERROR`. The `NO_ATTEMPT` outcome was removed from Query Set 2 experiments (see Section 3.4.5). Table 4.6 shows the overall distribution of failures across all 504 query attempts, while Figure 4.7 compares how these failures were distributed across the different model families. Appendix G shows the full distribution of failure types for each individual query.

Table 4.6: The overall failure distribution in Query Set 2 experiments.

Failure type	Failure count	Share of failures	Share of all attempts
`SYNTAX_ERROR`	41	27.7%	8.1% (41/504)
`WRONG_RESULT`	107	72.3%	21.2% (107/504)
Total	148	100%	29.4% (148/504)

Across all configurations, the majority of errors (107 instances) were `WRONG_RESULT` cases where the model produced a syntactically valid Cypher query that executed successfully but returned an incorrect answer. In addition, 41 cases were classified as `SYNTAX_ERROR`. Both failure types occurred far more frequently than in the Query Set 1 experiments. In Query Set 1, syntax errors accounted for only 5 out of 1,296 attempts (0.4%), whereas Query Set 2 produced 41 syntax errors out of 504 attempts (8.1%). `WRONG_RESULT` cases also increased substantially from 65 out of 1,296 attempts (5.0%) in Query Set 1 to 107 out of 504 attempts (21.2%) in Query Set 2.These differences likely result from the increased structural and analytical complexity of the Query Set 2 tasks combined with the broader, more detailed domain context employed in these experiments. These factors appear to provide more opportunities for the model to make mistakes when constructing multi-step queries.

A key observation is that none of the incorrect outputs were caused by errors in the retrieval process of the dynamic context pipeline. In every dynamic run, the extracted entities and retrieved data samples were accurate. All failures originated from the Cypher generation step. This confirms that the retrieval stage did not introduce errors into the evaluation. The observed errors are a result of limitations in the generation process rather than missing or incorrect domain context in the prompts.

The `SYNTAX_ERROR`s observed in Query Set 2 took several forms that did not follow a clear recurring pattern. These included undefined or redeclared variables, mismatched data types, and calls to functions not supported in Cypher 5 (e.g., `year()`, `month()`, or `APOC` procedures). Some errors appeared in queries that were so long that maintaining consistent variable scope and grouping order became difficult.

The distribution of these issues across the query set was not uniform. Query 04 presented a significant challenge, resulting in 17 syntax errors, the highest number of errors among all queries. This query required multi-stage aggregation, ranking, and percentage calculations. These operations are syntactically fragile in Cypher and prone to break when a single ordering or grouping step is misplaced. The high number of syntax errors in Q04 reflects the inherent structural complexity involved in formulating this query in Cypher.

Some failures, particularly in the GPT-5-mini zero-shot configurations, involved completely hallucinated or unnecessary match patterns. These failures fell into two categories:

- **Relationships that do not exist in the schema at all**, such as `(:Product)-[:HAS_VALUE]->(:ParameterValue)` or `(:Attribute)-[:HAS_ATTRIBUTE]->(:Attribute)`.
- **Valid, but unnecessary relationships**, such as `(:ParameterValue)-[:HAS_ATTRIBUTE]->(:Attribute)`

or `(:ProductFamily)-[:HAS_ATTRIBUTE]->()<-[:HAS_ATTRIBUTE]-(:ParameterValue)`

These relationship are possible and exist in the schema, but none of the evaluation queries required this kind of traversing, and using them produced incorrect results.

These hallucinations were largely eliminated by few-shot prompting, which reinforces the earlier observation that the high-reasoning configuration behaves unpredictably under zero-shot prompting, but becomes far more stable when anchored by examples.

The `WRONG_RESULT` errors were also irregular and varied, but typically arose from issues such as aggregations applied at the wrong level, misordered or misplaced filters, incorrect relationship directions in multi-hop patterns, or the use of an incorrect denominator when computing percentages. Among all queries, Q07 generated the largest number of logical failures, with 18 `WRONG_RESULT` cases. This query required calculating a percentage over a filtered subset of products, which in Cypher must be constructed through a multi-stage grouping pipeline. Such queries are sensitive to the precise ordering of `WITH` clauses and to how intermediate variables are scoped. Small deviations from the intended structure often resulted in queries that executed but returned incorrect results.

In contrast with Query Set 1, where the differences between models were substantial, the failure distributions in Query Set 2 were far more uniform across the three models (Figure 4.7). The only clear divergence appeared in the hallucinated graph structures, which occurred primarily in the GPT-5-mini zero-shot configurations. This pattern, however, is not directly visible in Figure 4.7, since the figure aggregates all the different experimental configuration outputs for each model.

Across all configurations, failures in Query Set 2 were more irregular than those in Query Set 1. The strongest configurations best illustrate the practical implica-

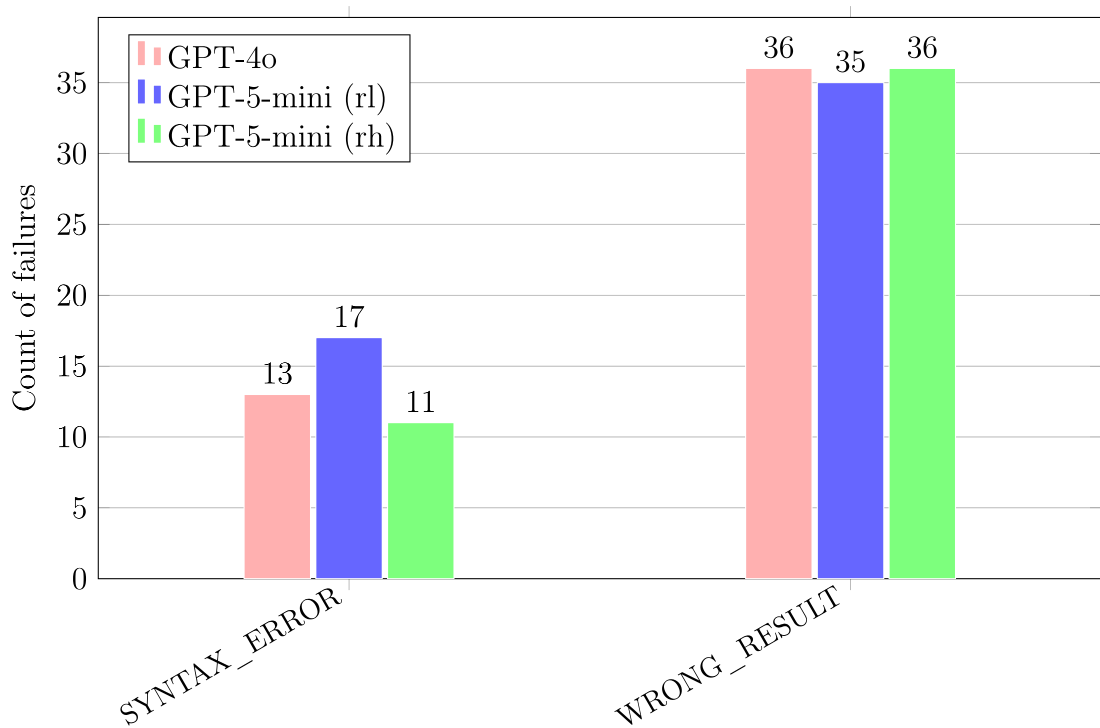

Figure 4.7: Distribution of failure types by model in Query Set 2 experiments.

tions of this irregularity. Configuration 10 (GPT-5-mini, low reasoning, dynamic context, few-shot prompting) produced only five failures in total, yet these errors did not share a common underlying cause. Instead, they consisted of isolated syntax errors and minor logical inaccuracies . Since these errors do not stem from a clear recurring pattern that could be corrected by refining the prompt, they likely reflect the difficulty of expressing complex analytical logic in Cypher rather than a deficiency in the prompting strategy.

In practice, this suggests that a small margin of error is unavoidable for analytically demanding queries. Mitigating these cases may require additional post-generation validation steps or performing the numerical calculations outside of Cypher.

5 Conclusion

5.1 Summary of Key Findings

This thesis examined whether modern large language models can generate accurate and reliable Cypher queries for structured product configuration data represented in a knowledge graph. The evaluation was carried out in two controlled settings: schema-based querying (Query Set 1) and domain-context–augmented querying (Query Set 2). The key findings are:

- LLMs are highly effective in schema-based query generation when all required information is available directly in the prompt. GPT-5-mini achieved 98–100% accuracy in most configurations.
- Few-shot prompting consistently improved accuracy across all models, with especially large gains for GPT-4o and for GPT-5-mini in the domain-context experiments.
- The simplified schema representation (V2) reduced prompt size by 70–80% and improved or maintained accuracy for all models.
- In domain-context-augmented tasks, dynamic RAG-based prompting reduced prompt size by nearly 90% and achieved accuracy comparable to or higher than static full-context prompting.

- GPT-5-mini with high reasoning mode achieved the highest accuracy in both query sets but at very high latency and token cost. GPT-5-mini (low reasoning) with few-shot prompting and dynamically retrieved domain context offered the best balance between accuracy and latency.
- `WRONG_RESULT` errors were the most critical failure type in both query sets. Syntax errors were rare in Query Set 1 but more common in Query Set 2 due to the increased structural complexity of the queries.

5.2 Answering the Research Questions

RQ1. The experiments show that modern LLMs *can* generate Cypher queries accurately and reliably for structured enterprise data, but only under specific prompting and context conditions. GPT-5-mini achieved near-perfect accuracy in the schema-only setting and up to 95% accuracy in the domain-context setting when few-shot prompting was used. Zero-shot prompting, especially in complex domain-context tasks, led to significant accuracy drops. Reliability depends strongly on providing examples, supplying relevant context, and avoiding highly complex multi-stage analytical queries.

RQ2. Model choice and prompting style had the largest effect on performance. GPT-5-mini outperformed GPT-4o in all configurations. Few-shot prompting consistently improved accuracy. The language of the NLQ (English vs. Finnish) had no meaningful effect in Query Set 1.

RQ3. Static and dynamic domain context achieved similar accuracy, but dynamic context was substantially more efficient, reducing prompt size from approximately 25000 to about 3000 tokens. Latency differences between static and dynamic prompting were small relative to the latency introduced by the Cypher-generation step itself. The dynamic approach is more scalable and easier to maintain.

RQ4. The most common and problematic failures were `WRONG_RESULT` cases caused by a variety of subtle logical and semantic errors. Syntax errors appeared mainly in the more complex analytical queries of Query Set 2. `NO_ATTEMPT` failures occurred mainly in GPT-4o due to conservative guardrail behavior. No failures were caused by the retrieval step in the dynamic context pipeline. All errors originated from the Cypher generation step.

5.3 Implications

The results demonstrate that LLM-based natural language querying is feasible for CPQ data when prompts are carefully designed and relevant context is supplied explicitly.This aligns with recent research emphasizing that LLMs perform substantially better on structured query-generation tasks when prompts are enriched with relevant schema-grounded context [23], [24]. Few-shot prompting and schema simplification consistently improved accuracy, which aligns with studies showing that strategically designed examples in prompts can enhance multi-hop reasoning and query construction [30]. Other work has similarly found that structured few-shot examples can improve performance in certain settings [29].

A RAG-based approach provides a scalable, cost-efficient way to supply relevant domain knowledge to LLMs. This approach mirrors others that retrieve candidate entities, properties, or subgraphs to better align user queries with the structure and content of the underlying data model [27], [32].

These findings suggest that LLM-based querying could be integrated into systems like Summium CPQ to support interactive exploration of offer and configuration data.

However, the presence of `WRONG_RESULT` errors shows that real-world deployments still require validation layers, particularly for analytically complex queries. Prior work has reached similar conclusions, demonstrating the value of post-generation

correction mechanisms such as query-checking algorithms [29] or structural consistency validators [27] to mitigate schema- and syntax-level errors. Overall, while contemporary LLMs are capable of generating sophisticated Cypher queries, the reliability of NL-to-Cypher systems depends heavily on the choice of model, prompting strategy, context formulation, and query complexity.

5.4 Limitations

The evaluation was conducted under controlled conditions that constrain the generalizability of the results. All natural language queries were carefully phrased and did not reflect the variability, ambiguity, or noise typically found in real user input. In the Query Set 2 experiments, only a curated subset of the product configuration data was used. Therefore, the results do not reflect the full complexity of a production-scale Summium CPQ deployment. The study also evaluated only OpenAI models (GPT-4o and GPT-5-mini), so the findings may not directly apply to other LLMs.

Each query was executed three times per configuration, which limits statistical robustness for configurations that exhibited highly variable behavior. The system also did not incorporate any post-generation validation layers, although such mechanisms would likely improve robustness against `WRONG_RESULT` failures in real deployments. Finally, some of the analytically demanding queries used in the evaluation are inherently difficult to express in Cypher, and the observed errors partly reflect limitations of the query language itself rather than the prompting strategy alone.

5.5 Future Work

Several areas for improvement naturally follow from the limitations and findings of this thesis. First, robustness could be improved by adding post-generation valida-

tion layers or refinement loops to reduce `WRONG_RESULT` failures. Second, the system should be evaluated with real, unconstrained user queries to assess how well the models handle natural variation and ambiguity. Regarding the system, the dynamic retrieval pipeline could be extended and optimized to support larger or continuously evolving product datasets. Additionally, hybrid model pipelines, such as using lighter models for summary generation or dynamically selecting between lighter and heavier models based on query complexity, could reduce latency and cost.

Advanced directions include exploring AI agent systems in which an LLM autonomously selects tools or retrieval operations, and investigating whether fine-tuning LLMs on domain-specific data and Cypher patterns results in measurable improvements. Finally, integrating the approach into a real-world CPQ environment and observing how it behaves under real usage conditions would provide valuable insight into its practical viability.

References

[1] M. Jordan, G. Auth, O. Jokisch, and J.-U. Kühl, “Knowledge-based systems for the configure price quote (cpq) process – a case study in the it solution business”, *Online Journal of Applied Knowledge Management*, vol. 8, no. 2, pp. 17–30, Sep. 2020. DOI: https://doi.org/10.36965/ojakm.2020.8(2)17-30.

[2] T. Teubner, C. M. Flath, C. Weinhardt, W. van der Aalst, and O. Hinz, “Welcome to the era of chatgpt et al.”, *Business & Information Systems Engineering*, vol. 65, no. 2, pp. 95–101, Mar. 2023. DOI: https://doi.org/10.1007/s12599-023-00795-x.

[3] H. Naveed et al., *A Comprehensive Overview of Large Language Models*. Apr. 2024. [Online]. Available: https://arxiv.org/pdf/2307.06435.

[4] A. Vaswani et al., *Attention Is All You Need*. Jun. 2017. [Online]. Available: https://arxiv.org/pdf/1706.03762.

[5] T. Brown et al., *Language Models are Few-Shot Learners*. Jul. 2020. [Online]. Available: https://arxiv.org/pdf/2005.14165.

[6] L. Huang et al., “A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions”, *ACM Transactions on Information Systems*, vol. 43, no. 2, pp. 1–55, Jan. 2025. DOI: https://doi.org/10.1145/3703155.

[7] Y. Zhou, P. Xu, X. Liu, B. An, W. Ai, and F. Huang, *Explore spurious correlations at the concept level in language models for text classification*, 2023. [Online]. Available: `https://arxiv.org/abs/2311.08648`.

[8] N. F. Liu et al., *Lost in the middle: How language models use long contexts*, 2023. [Online]. Available: `https://arxiv.org/abs/2307.03172`.

[9] H. Dai, D. Pechi, X. Yang, G. Banga, and R. Mantri, *Deniahl: In-context features influence llm needle-in-a-haystack abilities*, 2024. [Online]. Available: `https://arxiv.org/abs/2411.19360`.

[10] Y. Gao, Y. Xiong, W. Wu, Z. Huang, B. Li, and H. Wang, *U-niah: Unified rag and llm evaluation for long context needle-in-a-haystack*, 2025. [Online]. Available: `https://arxiv.org/abs/2503.00353`.

[11] T. Kudo and J. Richardson, *Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing*, 2018. [Online]. Available: `https://arxiv.org/abs/1808.06226`.

[12] G. Tamašauskaitė and P. Groth, "Defining a knowledge graph development process through a systematic review", *ACM Transactions on Software Engineering and Methodology*, 2022. DOI: `https://doi.org/10.1145/3522586`.

[13] A. Hogan et al., *Knowledge Graphs*. 2021. [Online]. Available: `https://arxiv.org/pdf/2003.02320`.

[14] A. Singhal, *Introducing the knowledge graph: Things, not strings*, May 2012. [Online]. Available: `https://blog.google/products/search/introducing-knowledge-graph-things-not/`.

[15] L. Ehrlinger and W. Wöß, *Towards a Definition of Knowledge Graphs*. 2016. [Online]. Available: `https://ceur-ws.org/Vol-1695/paper4.pdf`.

[16] J. Stegeman, *What is a knowledge graph?*, Jul. 2024. [Online]. Available: `https://neo4j.com/blog/genai/what-is-knowledge-graph/`.

[17] P. Hitzler, "A review of the semantic web field", *Communications of the ACM*, vol. 64, no. 2, pp. 76–83, Jan. 2021. DOI: https://doi.org/10.1145/3397512.

[18] D. Fensel et al., "Introduction: What is a knowledge graph?", in *Knowledge Graphs: Methodology, Tools and Selected Use Cases*. Cham: Springer International Publishing, 2020, pp. 1–10, ISBN: 978-3-030-37439-6. DOI: 10.1007/978-3-030-37439-6_1.

[19] R. Howard, *Rdf vs. property graphs: Choosing the right approach for implementing a knowledge graph - graph database analytics*, Jun. 2024. [Online]. Available: https://neo4j.com/blog/knowledge-graph/rdf-vs-property-graphs-knowledge-graphs/.

[20] N. Francis et al., "Cypher", *Proceedings of the 2018 International Conference on Management of Data - SIGMOD '18*, 2018. DOI: https://doi.org/10.1145/3183713.3190657.

[21] 2025. [Online]. Available: https://neo4j.com/docs/cypher-manual/current/introduction/cypher-overview/.

[22] A. Bridgwater, *Neo4j cto: Gql is here: The evolution from cypher opencypher*, 2024. [Online]. Available: https://www.computerweekly.com/blog/CW-Developer-Network/Neo4j-CTO-GQL-is-here-the-evolution-from-Cypher-openCypher#:~:text=Cypher%20is%20a%20property%20graph,users%20write%20queries%20in%20Cypher.

[23] I.-V. Hernandez-Camero, E. Garcia-Lopez, A. Garcia-Cabot, and S. Caro-Alvaro, "Context-aware few-shot learning sparql query generation from natural language on an aviation knowledge graph", *MAKE*, vol. 7, no. 2, p. 52, Jun. 2025. DOI: https://doi.org/10.3390/make7020052.

[24] L. Nan et al., *Enhancing few-shot text-to-sql capabilities of large language models: A study on prompt design strategies*, 2023. [Online]. Available: https://arxiv.org/abs/2305.12586.

[25] X. Huang, J. Zhang, D. Li, and P. Li, "Knowledge graph embedding based question answering", pp. 105–113, Jan. 2019. DOI: https://doi.org/10.1145/3289600.3290956.

[26] R. Omar, O. Mangukiya, P. Kalnis, and E. Mansour, *Chatgpt versus traditional question answering for knowledge graphs: Current status and future directions towards knowledge graph chatbots*, 2023. [Online]. Available: https://arxiv.org/abs/2302.06466.

[27] X. Pan, d. Boer, and v. Ossenbruggen, *Firesparql: A llm-based framework for sparql query generation over scholarly knowledge graphs*, 2025. [Online]. Available: https://arxiv.org/abs/2508.10467.

[28] M. Liu and J. Xu, *Nli4db: A systematic review of natural language interfaces for databases*, 2025. [Online]. Available: https://arxiv.org/abs/2503.02435.

[29] L. Pusch and T. Conrad, *Combining LLMs and Knowledge Graphs to Reduce Hallucinations in Biomedical Question Answering*. 2024. [Online]. Available: https://arxiv.org/pdf/2409.04181.

[30] M. Shah et al., *Improving LLM-based KGQA for multi-hop Question Answering with implicit reasoning in few-shot examples*. 2024. [Online]. Available: https://aclanthology.org/2024.kallm-1.13.pdf.

[31] S. Sivasubramaniam, C. Osei-Akoto, Y. Zhang, K. Stockinger, and J. Fürst, "Sm3-text-to-query: Synthetic multi-model medical text-to-query benchmark", 2024. [Online]. Available: https://arxiv.org/pdf/2411.05521.

[32] A. Saleh, G. Tur, and Y. Saygin, "Sg-rag: Multi-hop question answering with large language models through knowledge graphs", 2024. [Online]. Available: https://aclanthology.org/2024.icnlsp-1.45.pdf.

[33] I. Tsampos and E. Marakakis, "Domain- and language-adaptable natural language interface for property graphs", *Computers*, vol. 14, no. 5, pp. 183–183, May 2025. DOI: https://doi.org/10.3390/computers14050183.

[34] G. Ayman et al., "Building a smart academic advising chatbot with llms and knowledge graphs: A case study at nile university", *2025 International Conference on Machine Intelligence and Smart Innovation (ICMISI)*, pp. 319–323, May 2025. DOI: https://doi.org/10.1109/icmisi65108.2025.11115413.

[35] Z. Li, L. Deng, H. Liu, Q. Liu, and J. Du, *Unioqa: A unified framework for knowledge graph question answering with large language models*, 2024. [Online]. Available: https://arxiv.org/abs/2406.02110.

[36] S. Wu et al., *Retrieval-augmented generation for natural language processing: A survey*, 2024. [Online]. Available: https://arxiv.org/abs/2407.13193.

[37] P. Lewis et al., *Retrieval-augmented generation for knowledge-intensive nlp tasks*, 2020. [Online]. Available: https://arxiv.org/abs/2005.11401.

[38] X. Wang et al., *Searching for best practices in retrieval-augmented generation*, 2024. [Online]. Available: https://arxiv.org/abs/2407.01219.

[39] *Building effective agents*, 2024. [Online]. Available: https://www.anthropic.com/engineering/building-effective-agents.

[40] 2025. [Online]. Available: https://docs.langchain.com/oss/python/langchain/sql-agent.

[41] *Db-engines ranking of graph dbms*, 2025. [Online]. Available: https://db-engines.com/en/ranking/graph+dbms.

[42] *Building effective agents*, 2025. [Online]. Available: https://docs.spring.io/spring-ai/reference/api/effective-agents.html.

[43] *Build a question/answering system over sql data*, 2021. [Online]. Available: https://python.langchain.com/docs/tutorials/sql_qa/?utm_source=chatgpt.com#dealing-with-high-cardinality-columns.

[44] *Workflows and agents*, 2025. [Online]. Available: https://langchain-ai.github.io/langgraph/tutorials/workflows/.

Appendix A QS1 Natural Language Queries

This appendix contains the complete set of natural-language questions defined in Query Set 1. The set consists of 18 distinct information-retrieval tasks, each written in both English and Finnish (36 queries in total). The queries cover all nine valid Type–Level combinations defined in Section 3.3, with exactly two queries representing each combination.

Table A.1 lists all queries in both languages.

Table A.1: Query Set 1

ID	Lvl/Type	Lang	Natural Language Query
Q01E	L1–Lookup	EN	What is the version of offer {offer_id}?
Q01F	L1–Lookup	FI	Mikä on tarjouksen {offer_id} versio?
Q02E	L1–Lookup	EN	What is the date of offer {offer_id}?
Q02F	L1–Lookup	FI	Mikä on tarjouksen {offer_id} päivämäärä?
Q03E	L1–Single-hop	EN	Which participant sent offer {offer_id}?
Q03F	L1–Single-hop	FI	Kuka osallistuja lähetti tarjouksen {offer_id}?
Q04E	L1–Single-hop	EN	Which participant received offer {offer_id}?
Q04F	L1–Single-hop	FI	Kuka osallistuja vastaanotti tarjouksen {offer_id}?

ID	Lvl/Type	Lang	Natural Language Query
Q05E	L2–Single-hop	EN	Which offers dated July 16th 2024 were sent by participant {participant_name}?
Q05F	L2–Single-hop	FI	Mitkä 16. heinäkuuta 2024 päivätyt tarjoukset osallistuja {participant_name} lähetti?
Q06E	L2–Single-hop	EN	Which offers dated in January 2024 were handled by user with id {user_id}?
Q06F	L2–Single-hop	FI	Mitkä tammikuulle 2024 päivätyt tarjoukset käsitteli käyttäjä, jonka id on {user_id}?
Q07E	L2–Multi-hop	EN	What are the names of the product families configured in offer {offer_id}?
Q07F	L2–Multi-hop	FI	Minkä nimiset tuoteperheet on konfiguroitu tarjoukseen {offer_id}?
Q08E	L2–Multi-hop	EN	What is the total_price of the subcontent of offer {offer_id}?
Q08F	L2–Multi-hop	FI	Mikä on tarjouksen {offer_id} alisisällön kokonaishinta?
Q09E	L3–Multi-hop	EN	For offer {offer_id}, list the measure values for all parameters that have Attribute(type = ”ext_id”, value = ”{ext_id_value}”).
Q09F	L3–Multi-hop	FI	Listaa measure-arvot kaikilta tarjouksen {offer_id} parametreilta, joilla on Attribute(type = ”ext_id”, value = ”{ext_id_value}”).
Q10E	L3–Multi-hop	EN	List the measure values of the parameter named ”Length” for all products in offer {offer_id}.
Q10F	L3–Multi-hop	FI	Listaa ”Length”-nimisen parametrin measure-arvot kaikille tuotteille tarjouksessa {offer_id}.

ID	Lvl/Type	Lang	Natural Language Query
Q11E	L2–Aggregation	EN	How many offers have been sent by participant {participant_name}?
Q11F	L2–Aggregation	FI	Kuinka monta tarjousta osallistuja {participant_name} on lähettänyt?
Q12E	L2–Aggregation	EN	How many different products are contained in offer {offer_id}?
Q12F	L2–Aggregation	FI	Kuinka monta eri tuotetta sisältyy tarjoukseen {offer_id}?
Q13E	L3–Aggregation	EN	What is the number and combined total price of offers sent by {participant_name} that only contain products configured as “SNEP® MODE P”?
Q13F	L3–Aggregation	FI	Mikä on osallistujan {participant_name} lähettämien tarjousten määrä ja yhteenlaskettu kokonaishinta, kun tarjoukset sisältävät ainoastaan tuotteita, jotka ovat konfiguroitu malliksi “SNEP® MODE P”?
Q14E	L3–Aggregation	EN	List the 5 companies that received the most offers dated in the first half of 2024, including the number of received offers and their combined total price.
Q14F	L3–Aggregation	FI	Listaa 5 yritystä, jotka vastaanottivat eniten vuoden 2024 ensimmäiselle puoliskolle päivättyjä tarjouksia, sekä tarjousten lukumäärä ja niiden yhteenlaskettu kokonaishinta.
Q15E	L2–Analytics	EN	What is the average content total price of offers dated in March 2024?
Q15F	L2–Analytics	FI	Mikä on maaliskuulle 2024 päivättyjen tarjousten sisällön keskimääräinen kokonaishinta?

ID	Lvl/Type	Lang	Natural Language Query
Q16E	L2–Analytics	EN	What is the average number of different products in offers received by company {participant_company}?
Q16F	L2–Analytics	FI	Mikä on yrityksen {participant_company} vastaanottamien tarjousten sisältämien eri tuotteiden keskimääräinen määrä?
Q17E	L3–Analytics	EN	Which 10 participant companies have the highest average content total price of their received offers that are dated in the second half of 2024?
Q17F	L3–Analytics	FI	Mitkä 10 osallistujayritystä omaavat korkeimman vastaanotettujen tarjousten sisällön keskimääräisen kokonaishinnan, kun tarkastellaan vuoden 2024 toiselle puoliskolle päivättyjä tarjouksia?
Q18E	L3–Analytics	EN	List the top 3 participants grouped by name and ranked by number of offers dated in Q3 2024 that they sent, including the offer counts and the combined total price of those offers.
Q18F	L3–Analytics	FI	Listaa kolme osallistujaa, ryhmiteltynä nimen mukaan, jotka lähettivät eniten vuoden 2024 kolmannelle vuosineljännekselle päivättyjä tarjouksia, ja näytä tarjousten määrät sekä yhteenlasketut kokonaishinnat.

Appendix B QS2 Natural Language Queries

This appendix contains the complete set of 14 natural-language questions defined in Query Set 2. These queries were designed to reflect realistic user phrasing in the CPQ domain and each query is accompanied by a set of descriptive tags (e.g., TopK, Feature, SingleFamily, Temporal). Although the tags were not used directly in the evaluation metrics, they document the conceptual variety of the set and helped ensure coverage of different query characteristics.

Table B.1 lists all queries and their associated tags.

Table B.1: Query Set 2

ID	Natural Language Query	Tags
Q01	What was the most popular configured length for SNEP Mode P lights in 2024?	TopK, Feature, SingleFamily
Q02	What were the amounts of the three most popular selected lengths for SNEP Mode P luminaires in Q1 2024?	TopK, Feature, Temporal, Single-Family
Q03	What were the amounts of the three most popular selected lengths for SNEP Mode P luminaires in each quarter of 2024?	TopK, Feature, Temporal, Single-Family

ID	Natural Language Query	Tags
Q04	What were the amounts of the three most popular selected lengths for SNEP Mode P luminaires and their percentage shares in each quarter of 2024?	TopK, Feature, Temporal, Percentage, SingleFamily
Q05	How many Mode S luminaires were configured with the gray frame color in Q4 2024?	FeatureFilter, Temporal, SingleFamily
Q06	How many mode C lights did not have the black frame color in Q1 2024?	FeatureFilter, Temporal, SingleFamily
Q07	Out of all ordered SNEP MODE C luminaires in 2024, what percentage had a configured length greater than 2000 mm?	FeatureFilter, Percentage, SingleFamily
Q08	What was the configured power for each luminaire in offer {offer_id}?	Feature, EntityFilter, SingleFamily
Q09	How many Mode C products with white color were ordered by {company_name} in 2024?	FeatureFilter, EntityFilter, SingleFamily
Q10	How many SNEP Mode S luminaires sold by {participant_name} were configured with power between 60W and 100W?	FeatureFilter, EntityFilter, SingleFamily
Q11	Which 5 optics were the most popular among all products in 2024? List their counts.	TopK, Feature, AllFamilies
Q12	In 2024, how many snep mode CR and snep mode P luminaires had a configured CCT of 4000K or higher?	FeatureFilter, MultiFamily

ID	Natural Language Query	Tags
Q13	What were the 5 most popular combinations of configured control, connections, and cable for SNEP MODE S luminaires in 2024?	TopK, MultiFeature, SingleFamily
Q14	What were the 3 most popular selected combinations of CRI and optics among all luminaire products in each quarter in 2024?	TopK, MultiFeature, Temporal, AllFamilies

Appendix C QS1 System Prompt

This appendix contains the system prompt instructions that were given for the LLM in the Query Set 1 experiments of this thesis.

```
You are a professional Neo4j expert. Your task is to generate a
Cypher statement to query a graph database.

Instructions:
- Use only the provided node labels, relationship types, and
    properties in the schema.
- Do not use any other node labels, relationship types or
    properties that are not provided.
- If a question cannot be answered based on the schema, respond with:
    "Sorry, I couldn't generate a Cypher query for that."
    Also mention the reason why the query couldn’t be generated.

Output:
- Provide only the raw Cypher query in your response.
- Do not include explanations, comments, or any additional text
    before or after the generated query.
- Do not wrap it in any markdown code fences (```...```).
- Return no extra text or annotations.

Schema:
«graphSchema»
```

Appendix D Graph Schema V2

This appendix contains the custom made text version (V2) of the graph schema that includes only the strictly necessary information about the nodes and relationships in the graph database.

```
Nodes:

User {id: STRING, userid: STRING, firstname: STRING, lastname: STRING}

Participant {id: STRING, title: STRING, name: STRING, company: STRING}

Offer {id: STRING, language_id: STRING, status: STRING, date: DATE,
    version: STRING}

Content {id: STRING, total_price: FLOAT}

Subcontent {id: STRING, total_price: FLOAT}

Product {id: STRING}

ProductFamily {id: STRING, name: STRING, product_code: STRING,
    total_cost: FLOAT}

Tab {id: STRING, name: STRING}

Parameter {id: STRING, name: STRING}

ParameterValue {id: STRING, measure: STRING, name: STRING,
    quantity: INTEGER, parameterId: STRING, cost: FLOAT}

Attribute {value: STRING, type: STRING}
```

```
Relationships:
(Offer)-[:HANDLED_BY]->(User) [1:1]
(Offer)-[:SENT_BY]->(Participant) [1:1]
(Offer)-[:DELIVERED_BY]->(Participant) [1:1]
(Offer)-[:RECEIVED_BY]->(Participant) [1:1]
(Offer)-[:HANDLED_BY]->(User) [1:1]
(Offer)-[:CONTAINS]->(Content) [1:1]
(Content)-[:CONTAINS]->(Subcontent) [1:1]
(Subcontent)-[:CONTAINS]->(Product) [1:N]
(Product)-[:CONFIGURED_AS]-(ProductFamily) [1:1]
(ProductFamily)-[:HAS_TAB]->(Tab) [1:N]
(Tab)-[:HAS_PARAMETER]->(Parameter) [1:N]
(Parameter)-[:HAS_VALUE]->(ParameterValue) [1:1]
(ProductFamily)-[:HAS_ATTRIBUTE]->(Attribute) [1:N]
(Tab)-[:HAS_ATTRIBUTE]->(Attribute) [1:N]
(Parameter)-[:HAS_ATTRIBUTE]->(Attribute) [1:N]
(ParameterValue)-[:HAS_ATTRIBUTE]->(Attribute) [1:N]
```

Appendix E QS1 Example NLQ-Cypher Pairs

This appendix contains the natural language query and Cypher query pairs that were added as examples to the system prompt when few-shot prompting was used with Query Set 1.

```
# Who sent offer 12345?
MATCH (o:Offer {id: "12345"})-[:SENT_BY]->(p:Participant)
RETURN p.name AS participantName

# For offer 12345, list the names of the parameter values for
parameters that have an attribute with type "ext_id" and value "vari".
MATCH (o:Offer {id: "12345"})
  -[:CONTAINS]->(:Content)
  -[:CONTAINS]->(:Subcontent)
  -[:CONTAINS]->(:Product)
  -[:CONFIGURED_AS]->(:ProductFamily)
  -[:HAS_TAB]->(:Tab)
  -[:HAS_PARAMETER]->(param:Parameter)
MATCH (param)
  -[:HAS_ATTRIBUTE]->(attr:Attribute {type: "ext_id", value: "vari"})
MATCH (param)-[:HAS_VALUE]->(pv:ParameterValue)
RETURN pv.name AS name

# Which 3 users handled the most offers dated in Q1 2024,
including the number of offers and the total content price?
MATCH (o:Offer)-[:HANDLED_BY]->(u:User)
WHERE o.date >= date("2024-01-01") AND o.date <= date("2024-03-31")
MATCH (o)-[:CONTAINS]->(c:Content)
WITH u, COUNT(o) AS offerCount, SUM(c.total_price) AS totalPrice
RETURN u.id AS userId, u.firstname AS firstName,
u.lastname AS lastName, offerCount, totalPrice
ORDER BY offerCount DESC
LIMIT 3
```

Appendix F QS1 Per-Query Status Distributions

Table F.1 summarizes, for each query from Query Set 1, the distribution of end statuses across all experimental configurations ($12 \times 3 = 36$ attempts per query). Values are shown as percentages with the absolute counts in parentheses. The final column reports total accuracy (OK%). The queries are in a descending order based on the total accuracy..

Table F.1: Aggregated per-query status distributions (percentage with absolute count in parentheses)

ID	OK	NO_ATTEMPT	WRONG_RESULT	SYNTAX_ERROR	Accuracy
Q01E	100.0% (36)	0.0% (0)	0.0% (0)	0.0% (0)	100.0%
Q01F	100.0% (36)	0.0% (0)	0.0% (0)	0.0% (0)	100.0%
Q02E	100.0% (36)	0.0% (0)	0.0% (0)	0.0% (0)	100.0%
Q02F	100.0% (36)	0.0% (0)	0.0% (0)	0.0% (0)	100.0%
Q04E	97.2% (35)	0.0% (0)	2.8% (1)	0.0% (0)	97.2%
Q11E	97.2% (35)	0.0% (0)	2.8% (1)	0.0% (0)	97.2%
Q03F	97.2% (35)	2.8% (1)	0.0% (0)	0.0% (0)	97.2%
Q04F	97.2% (35)	2.8% (1)	0.0% (0)	0.0% (0)	97.2%

ID	OK	NO_ATTEMPT	WRONG_RESULT	SYNTAX_ERROR	Accuracy
Q07E	97.2% (35)	2.8% (1)	0.0% (0)	0.0% (0)	97.2%
Q12F	97.2% (35)	2.8% (1)	0.0% (0)	0.0% (0)	97.2%
Q08E	94.4% (34)	0.0% (0)	5.6% (2)	0.0% (0)	94.4%
Q08F	94.4% (34)	2.8% (1)	2.8% (1)	0.0% (0)	94.4%
Q10F	94.4% (34)	2.8% (1)	2.8% (1)	0.0% (0)	94.4%
Q11F	94.4% (34)	2.8% (1)	2.8% (1)	0.0% (0)	94.4%
Q05E	94.4% (34)	5.6% (2)	0.0% (0)	0.0% (0)	94.4%
Q06F	94.4% (34)	2.8% (1)	0.0% (0)	2.8% (1)	94.4%
Q07F	94.4% (34)	5.6% (2)	0.0% (0)	0.0% (0)	94.4%
Q12E	94.3% (33)	0.0% (0)	5.7% (2)	0.0% (0)	94.3%
Q03E	91.7% (33)	0.0% (0)	8.3% (3)	0.0% (0)	91.7%
Q10E	91.7% (33)	2.8% (1)	5.6% (2)	0.0% (0)	91.7%
Q17E	91.7% (33)	5.6% (2)	2.8% (1)	0.0% (0)	91.7%
Q05F	91.7% (33)	8.3% (3)	0.0% (0)	0.0% (0)	91.7%
Q09E	88.9% (32)	0.0% (0)	11.1% (4)	0.0% (0)	88.9%
Q06E	88.9% (32)	8.3% (3)	2.8% (1)	0.0% (0)	88.9%
Q16F	88.9% (32)	8.3% (3)	2.8% (1)	0.0% (0)	88.9%
Q15F	88.9% (32)	8.3% (3)	0.0% (0)	2.8% (1)	88.9%
Q17F	88.9% (32)	8.3% (3)	0.0% (0)	2.8% (1)	88.9%
Q14E	86.1% (31)	0.0% (0)	8.3% (3)	5.6% (2)	86.1%
Q09F	83.3% (30)	2.8% (1)	13.9% (5)	0.0% (0)	83.3%
Q14F	83.3% (30)	11.1% (4)	5.6% (2)	0.0% (0)	83.3%
Q15E	83.3% (30)	16.7% (6)	0.0% (0)	0.0% (0)	83.3%
Q18E	80.6% (29)	16.7% (6)	2.8% (1)	0.0% (0)	80.6%
Q16E	80.6% (29)	19.4% (7)	0.0% (0)	0.0% (0)	80.6%

ID	OK	NO_ATTEMPT	WRONG_RESULT	SYNTAX_ERROR	Accuracy
Q18F	79.4% (27)	17.6% (6)	2.9% (1)	0.0% (0)	79.4%
Q13E	59.4% (19)	9.4% (3)	31.2% (10)	0.0% (0)	59.4%
Q13F	56.2% (18)	9.4% (3)	34.4% (11)	0.0% (0)	56.2%

Appendix G QS2 Per-Query Status Distributions

Table G.1 summarizes, for each query from Query Set 2, the distribution of end statuses across all experimental configurations ($12 \times 3 = 36$ attempts per query). Values are shown as percentages with the absolute counts in parentheses. The final column reports total accuracy (`OK`%). The queries are sorted in descending order of total accuracy.

Table G.1: Aggregated per-query status distributions for Query Set 2 (percentage with absolute count in parentheses)

ID	`OK`	`WRONG_RESULT`	`SYNTAX_ERROR`	Total	Accuracy
Q08	97.2% (35)	2.8% (1)	0.0% (0)	36	97.2%
Q06	91.7% (33)	2.8% (1)	5.6% (2)	36	91.7%
Q05	86.1% (31)	13.9% (5)	0.0% (0)	36	86.1%
Q10	86.1% (31)	11.1% (4)	2.8% (1)	36	86.1%
Q02	83.3% (30)	16.7% (6)	0.0% (0)	36	83.3%
Q01	77.8% (28)	22.2% (8)	0.0% (0)	36	77.8%
Q09	72.2% (26)	27.8% (10)	0.0% (0)	36	72.2%
Q11	72.2% (26)	25.0% (9)	2.8% (1)	36	72.2%

ID	OK	WRONG_RESULT	SYNTAX_ERROR	Total	Accuracy
Q12	69.4% (25)	22.2% (8)	8.3% (3)	36	69.4%
Q13	69.4% (25)	30.6% (11)	0.0% (0)	36	69.4%
Q03	66.7% (24)	16.7% (6)	16.7% (6)	36	66.7%
Q14	55.6% (20)	30.6% (11)	13.9% (5)	36	55.6%
Q07	33.3% (12)	50.0% (18)	16.7% (6)	36	33.3%
Q04	27.8% (10)	25.0% (9)	47.2% (17)	36	27.8%

Appendix H QS1 Results Example

Table H.1 shows an example of the full per-query results for one experimental configuration (GPT-4o, zero-shot prompting, graph schema version 2) with Query Set 1. The complete results for all configurations are provided in the GitHub repository. The columns in the results table are defined as follows:

Column	**Description**
`Status`	Outcome of query generation and execution: `OK` – correct result `SYNTAX_ERROR` – query contained a syntax error `NO_ATTEMPT` – model failed to generate a query `WRONG_RESULT` – query executed but result was incorrect
t_1	Avg. Cypher generation time over 3 runs (ms)
t_2	Avg. Cypher execution time in Neo4j over 3 runs (ms)
t_3	Avg. result summary generation time over 3 runs (ms)
t_{total}	Avg. total time over 3 runs (ms)
$\texttt{tokens}_1$	Avg. number of input tokens for Cypher generation prompt
$\texttt{tokens}_2$	Avg. number of input tokens for final NL summary prompt

Table H.1: Results 1 (GPT-4o, zero-shot, schema V2)

ID	status	t_1	t_2	t_3	t_{total}	$tokens_1$	$tokens_2$
Q01E	OK 3/3	329	10	655	994	606	188
Q01F	OK 3/3	301	8	395	704	608	190
Q02E	OK 3/3	376	20	405	801	606	193
Q02F	OK 3/3	1056	9	357	1422	610	197
Q03E	OK 3/3	395	17	411	823	604	232
Q03F	OK 3/3	465	7	403	875	609	206
Q04E	OK 3/3	374	9	527	910	604	236
Q04F	OK 3/3	738	12	500	1250	609	218
Q05E	OK 3/3	629	38	1047	1714	616	417
Q05F	OK 3/3	540	4	2277	2822	622	422
Q06E	OK 2/3 NO_ATTEMPT 1/3	593	47	2346	2986	613	422
Q06F	OK 2/3 NO_ATTEMPT 1/3	628	14	1391	2032	620	430
Q07E	OK 3/3	978	18	479	1475	611	317
Q07F	OK 3/3	666	17	491	1174	617	328
Q08E	OK 3/3	627	11	329	967	611	220
Q08F	OK 3/3	484	6	341	831	614	221
Q09E	OK 2/3 WRONG_RESULT 1/3	1045	35	642	1722	625	327
Q09F	WRONG_RESULT 3/3	990	292	436	1719	631	287
Q10E	OK 3/3	1025	233	581	1840	616	337
Q10F	OK 3/3	911	174	608	1693	621	334

ID	status	t_1	t_2	t_3	t_{total}	$tokens_1$	$tokens_2$
Q11E	OK 2/3 WRONG_RESULT 1/3	690	47	379	1115	610	215
Q11F	OK 3/3	920	15	363	1299	614	218
Q12E	OK 3/3	706	94	322	1123	590	226
Q12F	OK 3/3	581	79	475	1135	595	235
Q13E	WRONG_RESULT 3/3	1354	197	569	2121	627	320
Q13F	WRONG_RESULT 3/3	1241	165	703	2109	651	328
Q14E	OK 1/3 SYNTAX_ERROR 1/3 WRONG_RESULT 1/3	1345	238	1560	3143	628	506
Q14F	OK 2/3 NO_ATTEMPT 1/3	1158	122	1534	2812	647	498
Q15E	NO_ATTEMPT 3/3	-	-	-	-	-	-
Q15F	OK 2/3 NO_ATTEMPT 1/3	886	38	515	1439	623	249
Q16E	NO_ATTEMPT 3/3	-	-	-	-	-	-
Q16F	OK 2/3 WRONG_RESULT 1/3	971	126	480	1578	608	281
Q17E	OK 3/3	941	91	2251	3283	624	584
Q17F	OK 3/3	1134	127	2673	3934	652	607
Q18E	NO_ATTEMPT 3/3	-	-	-	-	-	-
Q18F	NO_ATTEMPT 3/3	-	-	-	-	-	-

Appendix I Prototype Chat User Interface

Figures I.1 and I.2 illustrate the chat-based UI used to demonstrate the system. The interface allows users to enter natural-language queries, view the generated Cypher, examine model metadata, and inspect execution latencies and token usage.

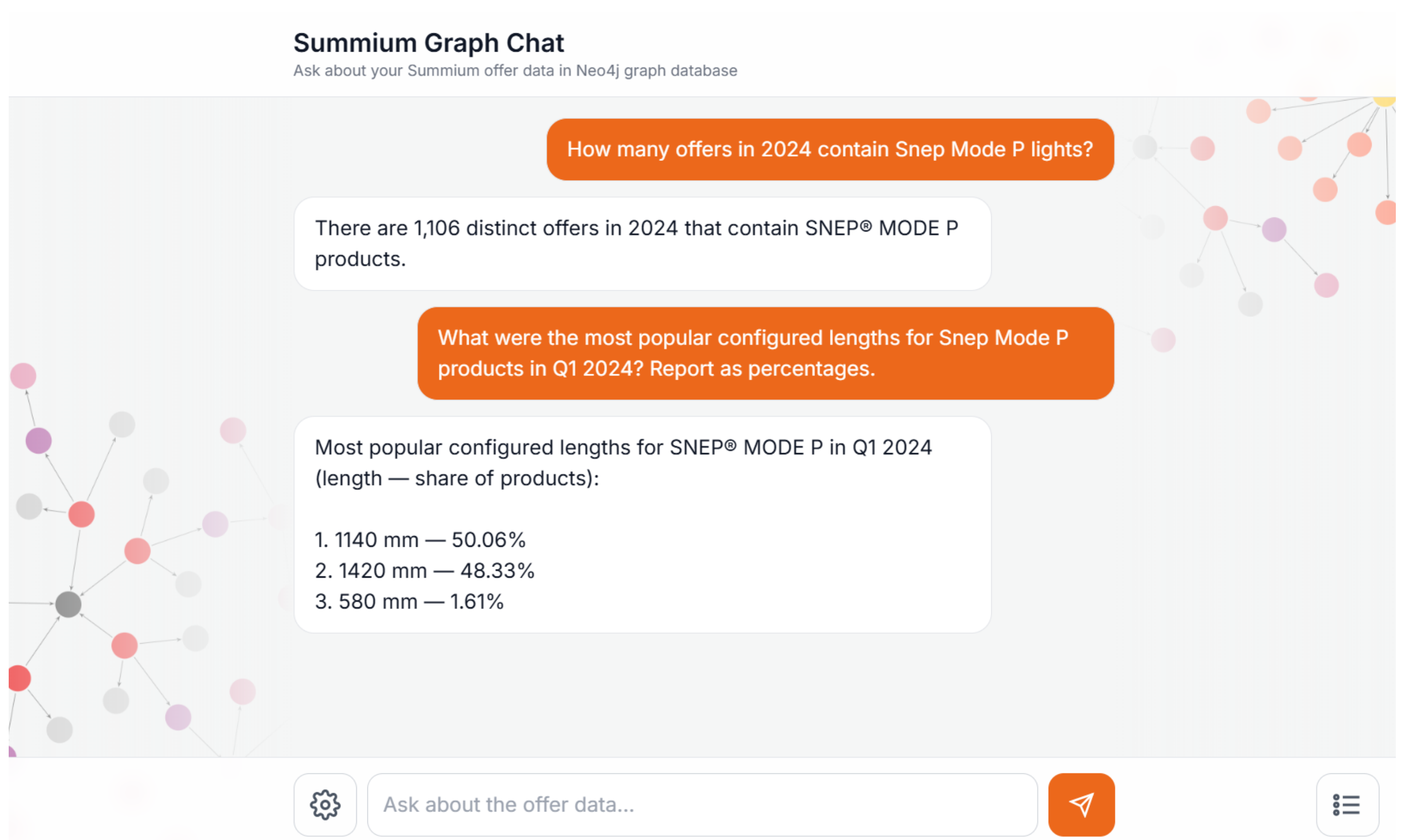

Figure I.1: The chat interface of the prototype system, showing example queries and answers.

Summium Graph Chat

Ask about your Summium offer data in Neo4j graph database

There are 1,106 distinct offers in 2024 that contain SNEP® MODE P products.

▼ Query Details

Model:

Model name	gpt-5-mini
Reasoning effort	low
Temperature	1.0

Metrics:

Query generation time	4003 ms
Neo4j execution time	121 ms
Summary generation time	1230 ms
Total time	5834 ms
Prompt 1 tokens	2166
Completion 1 tokens	296
Prompt 2 tokens	1524
Completion 2 tokens	96
Total input tokens	3690
Total output tokens	392

Generated Cypher:

```
MATCH (o:Offer)-[:CONTAINS]->(:Content)-[:CONTAINS]->(:Subcontent)
    -[:CONTAINS]->(prod:Product)-[:CONFIGURED_AS]-(pf:ProductFamily {name: "SNEP® MODE P"})
WHERE o.date >= date("2024-01-01") AND o.date <= date("2024-12-31")
RETURN COUNT(DISTINCT o) AS offerCount
LIMIT 100
```

Neo4j Response:


```
[ {
  "offerCount" : 1106
} ]
```

Ask about the offer data...

Figure I.2: The details view of a query in the prototype system, including model info, metrics, the generated Cypher query, and the raw Neo4j response.

Appendix J Use of Generative AI

Generative AI tools were used to improve the wording, grammar, and structure of the text in this thesis. These tools were not used to generate the substance of the thesis.