MASTER THESIS

An Approach to Generate Training Data for Question-to-AQL Querying Models

Marius Bernahrndt (Student ID: 7379885)

August 28, 2025

University of Cologne
Faculty of Mathematics and Natural Sciences
Department of Mathematics and Computer Science

Institute of Software Technology
Intelligent and Distributed
Systems Department

First Reviewer: Prof. Dr. Michael Felderer

Second Reviewer: Dr. Tobias Hecking

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden. Ich versichere, dass die eingereichte elektronische Fassung der eingereichten Druckfassung vollständig entspricht.

Die Strafbarkeit einer falschen eidesstattlichen Versicherung ist mir bekannt, namentlich die Strafandrohung gemäß § 156 StGB bis zu drei Jahren Freiheitsstrafe oder Geldstrafe bei vorsätzlicher Begehung der Tat bzw. gemäß § 161 Abs. 1 StGB bis zu einem Jahr Freiheitsstrafe oder Geldstrafe bei fahrlässiger Begehung.

Marius Bernahrndt

Köln, den 28.08.2025

Abstract

Graph databases are powerful tools for representing and querying complex knowledge structures. Query languages such as ArangoDB's AQL are challenging for non-expert users. Leveraging large language models (LLMs) for natural-language interfaces is an obvious step, but their preparation depends on suitable training corpora that map user questions to executable queries. For AQL, such corpora do not yet exist. This thesis introduces an approach to automatically generate such training data. The method combines schema-guided path sampling with LLM verbalization, ensuring that queries remain executable while questions are expressed in natural language. Fine-tuning an instruction-tuned model on the resulting corpus yields robust, well-formed AQL queries with high execution accuracy. Most remaining discrepancies concern semantic aspects such as collection choice, traversal direction, or operator selection, whereas syntax remains largely stable. Overall, the results demonstrate that schema-guided generation with LLM support can provide a faithful and sufficiently broad dataset, enabling the training of functional question-to-AQL models and offering a reproducible foundation for future NL2AQL research and system development.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

Abbrev.	**Full term**
API	Application Programming Interface
AQL	ArangoDB Query Language
BFS	Breadth-First Search
BPE	Byte-Pair Encoding
CoT	Chain-of-Thought
DFS	Depth-First Search
DLR	German Aerospace Center
KGQA	Knowledge Graph Question Answering
LLM	Large Language Model
LoRA	Low-Rank Adaptation
NL2AQL	Natural Language to AQL
NL2Query	Natural Language to Query
OOD	Out-of-distribution
PEFT	Parameter-efficient fine-tuning
RAG	Retrieval-Augmented Generation
RDBMS	Relational database management systems
RLHF	Reinforcement Learning from Human Feedback
RNN	Recurrent Neural Network
Seq2SQL	Sequence to SQL
SFT	Supervised Fine-tuning
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language

1 Introduction

Graph databases have gained increasing importance in recent years, as they allow the representation of complex knowledge structures in terms of nodes and edges and enable efficient traversals across multiple hops (Rodriguez & Neubauer, 2010). They form a central foundation for applications in which diverse entities and their relations must be modeled, such as knowledge management, recommendation systems, or research infrastructures. In aerospace contexts, for instance, graph databases are increasingly explored for managing mission knowledge, linking heterogeneous sensor data, and supporting maintenance workflows (C. Li et al., 2022; Yue et al., 2022). At the German Aerospace Center (DLR)[1], for instance, the multi-model database ArangoDB[2] is deployed for the management and analysis of large, heterogeneous knowledge graphs.

Despite these advantages, effective interaction with graph databases requires substantial expertise. The ArangoDB Query Language (AQL) is deliberately powerful and expressive, combining document-style queries with graph traversals and aggregations in a unified paradigm. Yet this flexibility comes at the cost of complexity. Mastering AQL requires detailed knowledge of the underlying schema as well as its traversal semantics. Occasional users must not only learn a new formal language but also internalize structural details of the data. Even small inaccuracies in operator choice, edge direction, or attribute naming can render a query invalid. While experts may cope with these challenges, the learning curve represents a barrier for non-specialists.

An obvious solution is to allow users to formulate queries directly in natural language and translate them automatically into AQL (NL2AQL). Current generative models, however, only partially bridge the gap between natural language input and correct, executable AQL queries. Even state-of-the-art large language models (LLMs) such as ChatGPT[3] frequently produce incorrect or non-executable queries. While, to the best of current knowledge, no systematic evaluation exists for AQL, empirical studies on related languages illustrate the scale of the problem: for text-to-SQL generation, for example, LLMs have been shown to generate erroneous queries in more than

[1]https://www.dlr.de
[2]https://www.arangodb.com/
[3]https://chat.openai.com

37.3% of cases (Shen et al., 2025). Crucially, however, languages such as Structured Query Language (SQL), SPARQL, and Cypher benefit from established benchmarks and datasets, such as Spider, QALD, and Text2Cypher, which provide a foundation for training, systematic evaluation, and iterative improvement. For AQL, by contrast, no such public corpus currently exists. This lack of data constitutes a central obstacle for NL2AQL research.

This thesis proposes a structure-oriented approach for the automatic generation of training corpora for natural language to AQL models, with the aim of establishing a foundation for systematic training, evaluation, and future benchmarking. At its core lies the concept of *structural retrieval*. Instead of deriving query candidates primarily from textual similarity, generation is guided by the explicit graph schema, including node and edge types, directions, and operator value compatibilities. This ensures that resulting question-query pairs are executable, even in the presence of multi-hop patterns, filters, and aggregations, while reducing hallucinations and schema drift. Complementary to this, controlled data augmentation is employed. Questions are first generated in precise, verifiable forms and subsequently transformed into more natural paraphrases. This increases linguistic variety without altering operator semantics or literal values.

Within this context, the thesis investigates the following research questions:

Research Questions

- **RQ1:** Is schema-guided path sampling with LLM verbalization a sufficient and faithful procedure to build a high-quality question-to-AQL corpus?
- **RQ2:** Is the resulting corpus adequate to train a functional question-to-AQL model that achieves high execution accuracy and remains stable under mild distribution shift?

By addressing these questions, the thesis contributes both a methodology for corpus construction and a practical resource, thereby advancing NL2AQL in the broader field of natural language interfaces to property graph databases.

2 Theoretical Background

This chapter provides the theoretical and technical background for the thesis. It introduces fundamental concepts of graph data modeling and querying, reviews relevant deep learning methods for text-to-query tasks, and discusses prompt engineering strategies that shape model behavior during inference. Together, these foundations support the design and evaluation of the proposed approach.

2.1 Graph Data Model

Graph-based data models offer a natural way to represent complex, interconnected data. They are especially relevant in application areas where relationships between entities are as important as the entities themselves. This section introduces the core principles of graph-based modeling, explains how graphs represent both data and relationships as first-class elements, and contrasts these models with traditional relational approaches.

2.1.1 Graph Databases vs. Relational Databases

Relational database management systems (RDBMS) organize data in tabular structures, relying on foreign keys and query operations to facilitate the reconstruction of relationships between entities. Although effective for many application scenarios, this approach exhibits significant performance limitations, particularly when reconstructing complex or unknown-length relationship paths (Rodrigues et al., 2023; Vicknair et al., 2010).

An illustrative example of this limitation can be seen in social network analysis, specifically when identifying connection paths between two individuals. In relational systems, queries involving paths of unknown length typically necessitate multiple query operations or recursive SQL queries. Such operations become computationally expensive and inefficient as the number of required operations increases, resulting in substantial performance degradation (Gai et al., 2014).

In contrast, graph databases organize data as interconnected networks of nodes (entities) and edges (relationships), and treat relationships as integral and equivalent elements of their data model. The explicit storage of entities and relationships allows graph databases to traverse connections directly, eliminating the need for computationally intensive query operations at runtime. Therefore, queries involving multi-hop relationships or path searches can typically be resolved via straightforward graph traversal operations without prior knowledge of the exact path length. Graph databases further enhance query efficiency through a concept known as index-free adjacency, wherein each node directly references adjacent nodes. This significantly reduces or even eliminates the need for complex query operations during data traversal (Fernandes & Bernardino, 2018).

These characteristics make graph databases particularly suited to scenarios involving highly interconnected datasets, and queries predominantly focus on relationship traversal. In particular, graph databases maintain more stable query performance as path length increases for traversal-centric workloads, offering substantial advantages in handling relationally intensive queries (Fernandes & Bernardino, 2018; Vicknair et al., 2010).

This structural difference has been examined in several comparative studies. Relational data models are designed to handle discrete data entities and reconstruct relationships at query time, typically through query operations. Graph data models, in contrast, store relationships explicitly within their structure, enabling efficient traversal of complex or multi-hop relationships. As demonstrated in multiple studies, this structural difference can yield significant performance benefits for relationship-centric queries (Fernandes & Bernardino, 2018; Gai et al., 2014; Vicknair et al., 2010).

2.1.2 Graph Model Overview

Modern graph databases commonly implement the *property graph model*, a flexible and expressive data model designed to represent complex, many-to-many relationships. In this model, nodes represent entities and edges denote directed relationships between them. A defining feature of property graphs is that both nodes and edges can hold arbitrary pairs of key-value pairs, allowing rich and descriptive representations of entities and their interconnections (Angles & Gutierrez, 2008).

Unlike relational models, which typically require decomposing entities and their relationships across multiple tables using foreign keys, property graphs offer a schema-less or schema-optional approach. This structural flexibility allows the modeling of heterogeneous and evolving data without predefined schema constraints (Angles &

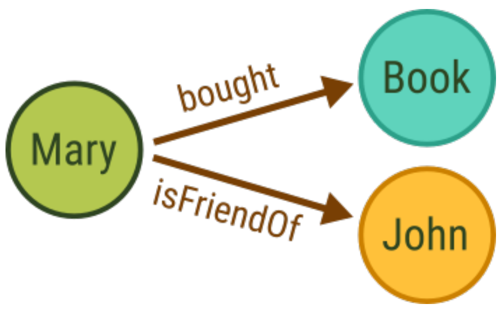

Figure 2.1: Simple property graph: Entities (nodes) and typed, directed relationships (edges). Source: ArangoDB documentation (ArangoDB GmbH, 2024a).

Gutierrez, 2008).

In addition, nodes are often annotated with one or more labels indicating their roles (e.g. `Person`, `Product`) and edges carry typed relationships (e.g. `bought`, `isFriendOf`) to encode semantic meaning. This makes the model intuitively aligned with real-world networks, where both entities and relationships are equally central. The index-free adjacency characteristic of property graphs improves performance even further. Each node maintains direct references to its neighboring nodes, thus avoiding costly query evaluations that are commonplace in relational systems during traversal operations (Fernandes & Bernardino, 2018; Robinson et al., 2015). When used in conjunction with graph-specific query languages such as Cypher or AQL, which formulate declarative graph queries, Cypher via pattern matching and AQL via explicit traversal directives, property graphs facilitate efficient querying of both topology and attributes (Paul et al., 2019).

In the graph illustrated in Figure 2.1, the entity `Mary` is connected to two other entities: `Book` via a `bought` relationship, and `John` via an `isFriendOf` relationship. Each of these relationships is modeled as a directed, semantically typed edge. This example illustrates how property graphs natively represent both entities and their relationships as first-class data elements. Rather than relying on multiple joins across relational tables, a graph database can efficiently traverse these edges directly. For example, retrieving all items Mary has purchased, or identifying her friends, can be achieved through simple graph traversals—without prior knowledge of path structure or depth. This structural and semantic alignment with real-world data patterns makes property graphs particularly suitable for domains such as social networks, recommendation engines, or contextual knowledge graphs (Fernandes & Bernardino, 2018; Paul et al., 2019; Robinson et al., 2015).

2.1.3 Graph Traversal

The term *graph traversal* refers to systematic strategies for navigating through the nodes and edges of a graph. Traversal methods are fundamental for tasks such as connectivity analysis, path search, data extraction, and synthetic data generation (Angles et al., 2017; Bojchevski et al., 2018; Gjoka et al., 2011; Robinson et al., 2015).

- **Depth-First Search (DFS)** explores a graph by following a path as deep as possible before backtracking. This makes it particularly well suited for searching for deeply nested structures, detecting cycles, and more generally topology-sensitive analyses, for example in directed acyclic graphs (Cormen et al., 2009; Tarjan, 1972).
- **Breadth-First Search (BFS)** explores a graph layer by layer, visiting all neighbors of the current frontier before expanding outward. It is the standard choice for computing the shortest paths in unweighted graphs and for efficiently analyzing local neighborhoods and distance distributions (Cormen et al., 2009; Lee, 1961).
- **Random Walks** are stochastic graph-traversal processes that view a graph as a Markov chain over its nodes. Starting from a seed node, the next node is sampled (typically uniformly) from its neighbors, and the procedure is repeated, yielding a sequence of nodes (the walk). The distribution of visited nodes mirrors structural properties such as node degrees, communities, and bottlenecks, making random walks a practical tool to collect diverse, representative path samples from large graphs (Gjoka et al., 2011; Grover & Leskovec, 2016; Lee, 1961).

2.1.4 ArangoDB as a Multimodal Graph Database

ArangoDB is a native multimodal database designed from its inception in 2011 to support graph, document, and key-value data models within a single, cohesive engine. This unified core enables all models, including graph structures and JSON documents, to be queried using the same language, the ArangoDB Query Language (AQL). Multi-model systems such as ArangoDB facilitate unified data handling and cross-model queries, eliminating the need for separate engines or context switching (ArangoDB GmbH, 2024b; Lu & Holubová, 2020).

AQL enables the seamless combination of graph traversals with document filtering or aggregation, allowing complex queries to be performed in one step. For example, a query could traverse a graph of research papers to identify the collaborators of a given scientist, while also filtering by publication year or venue, all in a single execution. This feature is particularly useful in question-to-AQL scenarios, where natural language questions often cover both relational and attribute-based data (Q. Guo et al., 2024).

ArangoDB is a multi-model database that incorporates full-text search and relevance scoring, enabling declarative information retrieval via the AQL query language. AQL supports structural and content-based conditions, as well as the combination of heterogeneous queries such as graph traversals with document-based filters. This is made possible by the close integration of graph and document data within a unified query engine (ArangoDB GmbH, 2024b).

Additionally, it has built-in clustering features for horizontal scaling. These include sharding and SmartGraphs, which partition large graphs efficiently while maintaining consistent traversal performance across distributed nodes. This architecture is therefore ideal for use with large, dynamic datasets (Fernandes & Bernardino, 2018).

Comparative studies emphasize the flexibility and scalability of ArangoDB over single-model graph systems, particularly in complex data management scenarios (Fernandes & Bernardino, 2018; Lu & Holubová, 2020).

2.2 Graph Query Languages

The process of querying graph-structured data requires the use of specialized query languages that are designed to operate effectively with the distinctive topology of graph models. In contrast to conventional SQL, these languages have been optimized for the direct expression of multi-hop traversals and structural patterns. This section introduces two widely used graph query languages, Cypher and AQL, and compares their approaches to querying property graphs and handling heterogeneous data.

2.2.1 Cypher

Cypher is Neo4j's declarative graph query language, developed specifically for property graph databases (Angles et al., 2017; Robinson et al., 2015). It was later transformed into an open standard (*openCypher*) with implementations across multiple systems. Cypher's design was influenced by SQL, with the objective of occupying a similar environmental space, but for graph data (Francis et al., 2018). Unlike SQL, where relationships are reconstructed via joins over normalized tables, Cypher expresses query logic through graph patterns. The user sketches a pattern of nodes and relationships to match, and the engine finds all subgraphs that fit that template. These patterns are written in a visual ASCII art style that is similar to graph drawings (Robinson et al., 2015).

For example, one can query for people and movies with a pattern like `(p:Person)-[:ACTED_IN]->(m:Movie)` to retrieve all `Movie` nodes connected by an `ACTED_IN` relationship from a `Person` node.

The `MATCH` clause binds graph elements on the specified pattern, and the `RETURN` clause defines the output (e.g. `m.title`). This pattern-oriented approach results in concise and human-readable queries, often shorter and more intuitive than equivalent SQL statements involving multiple `JOIN`s (Sasaki, 2016). The declarative nature of the query means that the information described *what* should be retrieved (i.e. the shape of the subgraph) rather than *how* to perform the traversal. The underlying database optimizer determines an efficient execution strategy. Cypher is a programming language that facilitates the representation of complex graph traversals, including variable-length paths, in a manner similar to drawing connections on a whiteboard (Francis et al., 2018; Sasaki, 2016). In addition to fundamental pattern matching, Cypher supports filtering with `WHERE` clauses, aggregation, and graph updates (e.g. `CREATE` and `DELETE` clauses) as part of its query language. The use of Cypher has expanded beyond the scope of Neo4j, with the advent of *openCypher* in 2015. This initiative facilitated the adoption of Cypher by multiple vendors, establishing it as a prevalent graph query language comparable to SQL within the industry. The efforts to standardize have enabled the utilization of Cypher and related query languages in systems such as SAP HANA, Graph, RediGraph, PostgreSQL, and others (Francis et al., 2018). Such developments leverage Cypher's declarative core while providing practical features for graph analysis. In general, Cypher has become a powerful tool for querying graph databases. This can be attributed to the fact that it utilizes a readable, pattern-oriented approach and a growing ecosystem. Consequently, even complex multi-hop queries can be expressed in a relatively intuitive form (Francis et al., 2018; Sasaki, 2016).

2.2.2 ArangoDB Query Language

AQL is a declarative query language for ArangoDB's native multi-model engine. Unlike pattern-matching languages such as Cypher, AQL specifies traversals explicitly: direction via `OUTBOUND/INBOUND/ANY`, variable depths (e.g., `2..4`), and bindings `v`, `e`, `p` for nodes, edges, and paths. Its JSON-centric `FOR–FILTER–RETURN` paradigm integrates traversals with document queries, true `JOIN`s across collections, and `COLLECT`-based aggregation in a single statement, enabling mixed structural and attribute conditions without context switches. This affords concise formulations of neighborhood exploration, subgraph extraction, and shortest-path tasks within one language (ArangoDB GmbH, 2024b; Q. Guo et al., 2024).

For example, the following query iterates nodes `v` reachable in exactly 2 hops *OUTBOUND* from `"Cities/Berlin"` in the graph `"Flights"`.

```
FOR v, e, p IN 2..2 OUTBOUND "Cities/Berlin"
   GRAPH "Flights"
RETURN v
```

The `OUTBOUND` keyword and numeric range `2..2` specify traversal direction and depth, while variables `v`, `e`, and `p` bind the current node, edge, and full path, enabling concise neighborhood exploration, BFS/DFS-style traversals, and shortest paths within AQL's JSON-oriented syntax (Q. Guo et al., 2024).

From a language-design perspective, AQL offers SQL-like expressiveness for projection, selection, joins, aggregation, and subqueries, while adding native traversal directives for graphs and array operations for JSON (Fernandes & Bernardino, 2018).

2.3 Deep-Learning Fundamentals

Modern natural language query generation relies heavily on deep learning models, especially LLMs. It is essential to understand the core components of these models. This section provides an overview of the Transformer architecture (and its self-attention mechanism), subword tokenization methods, training paradigms (pre-training, fine-tuning, and parameter-efficient tuning such as LoRA), and recent advances in instruction-tuning and in-context learning. Each of these fundamentals has driven improvements in translating natural language questions to structured queries.

2.3.1 Transformer Architecture and Self-Attention

The Transformer architecture, introduced by Vaswani et al., 2017, represents a paradigm shift in sequence modeling. By replacing recurrent and convolutional operations with an attention mechanism, this innovative architecture has transformed the field.

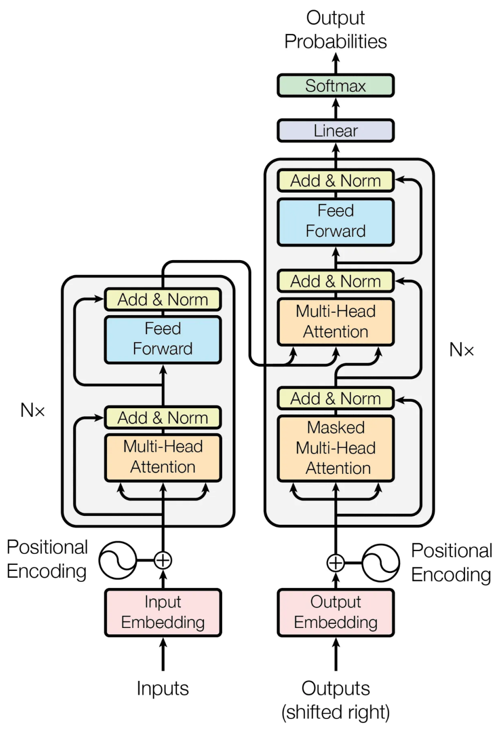

Figure 2.2: The figure shows the classic encoder-decoder structure with multi-head self-attention, masked attention, feed-forward layers, and position encodings (Vaswani et al., 2017).

The Transformer architecture illustrated in Figure 2.2 is typically composed of an encoder (a stack of self-attention and feed-forward layers that process the input sequence) and a decoder (which uses self-attention on the output sequence so far, plus encoder–decoder attention to attend to encoder outputs). The design is characterized by its high degree of parallelizability and its effectiveness in capturing long-range dependencies.

In contrast to recurrent neural networks (RNNs), which process tokens sequentially, transformers process the entire input sequence in parallel. This is achieved by the implementation of self-attention layers, which facilitate the processing of information within a sequence, enabling each position to attend to all other positions in the encoder. In a self-attention layer, the model calculates a relevance value for each token pair, determining how strongly one token contributes to the representation of another. In the context of the given set of query vectors Q, key vectors K and value vectors V (which correspond to the projections of the input or lower-layer representations), the scaled dot-product attention is defined as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{1}$$

The dimension of the keys, denoted by d_k, is used to scale the dot product (Vaswani et al., 2017). This operation produces a weighted sum of the value vectors, with weights determining how much each context token should influence the token at hand. The softmax function in the formula ensures that the attention weights on a token's keys sum to one. This mechanism enables the model to focus on the most relevant parts of the input when constructing each token's representation. Transformers use multiple such attention mechanisms in parallel, a concept known as *multi-head attention*. Each head learns to focus on different aspects or dependencies in the input sequence, and their outputs are concatenated and linearly transformed. This allows the model to jointly attend to diverse representation subspaces at different positions (Vaswani et al., 2017). The number of attention heads determines the model's ability to represent different linguistic patterns in parallel. While a single head may be limited in scope, multiple heads allow for specialization in different types of syntactic or semantic relationships.

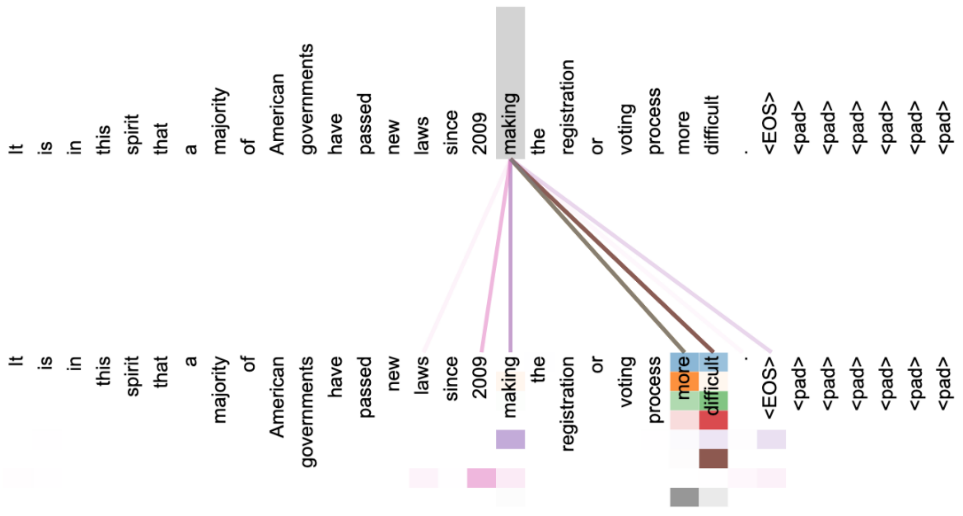

Figure 2.3: Visualization of the self-attention mechanism focusing on long-distance dependencies in the encoder. The attention heads shown attend to the word “making” and complete the phrase “making ... more difficult”. Colors represent distinct heads. Originally from Vaswani et al. (2017), reproduced via Thapa (2022).

As illustrated in Figure 2.3, this mechanism is demonstrated in action by multiple attention heads. These heads attend to semantically or syntactically relevant tokens from the perspective of the word "making". This illustrates the model’s ability to capture long-distance dependencies. In summary, a Transformer can model relationships between words regardless of their distance in the sentence (Vaswani et al., 2017).

2.3.2 Tokenization and Embedding Layers

Tokenization is the process of dividing textual input into smaller units, called *tokens*, that can be processed by neural networks. Purely word-level tokenization yields very large vocabularies and suffers from out-of-vocabulary issues. Contemporary systems therefore employ *subword* segmentation methods such as Byte-Pair Encoding (BPE), WordPiece, or the Unigram language model from SentencePiece, which keep frequent words intact while decomposing rare ones into reusable subunits. This strategy reduces the size of the vocabulary and allows the composition of unseen words from known parts (Kudo & Richardson, 2018; Sennrich et al., 2016).

Embedding layers are responsible for mapping each token ID to a dense vector in a continuous space using a learned embedding matrix. In this space, semantic similarity correlates with geometric proximity, commonly measured with cosine similarity. Earlier approaches like Word2Vec and GloVe generate *static* embeddings, where each word has a fixed vector regardless of *context* (Mikolov et al., 2013; Pennington et al., 2014). In contrast, transformer-based language models yield *contextual* embeddings, where a token's representation depends on its surrounding sequence. This enables disambiguation (e.g., distinguishing *bank* as a riverside vs. financial institution) and supports more nuanced semantic understanding (Devlin et al., 2019; Peters et al., 2018; Vaswani et al., 2017).

2.3.3 Pre-Training, Fine-Tuning, and Parameter-Efficient Tuning

Large language models are initially trained on massive, unlabeled text corpora with general self-supervised objectives such as next-token prediction or masked language modeling. During this phase, the model acquires grammar, world knowledge, and general statistics of language (Brown et al., 2020; Devlin et al., 2019; Raffel et al., 2020).

After pre-training, the model is adapted to a specific target task through supervised fine-tuning, e.g., the generation of structured queries from natural language input. This typically optimizes all parameters end-to-end on a comparatively small, labeled dataset that encodes domain- and task-specific information, thereby transferring the general representations learned during pre-training to the concrete application (Brown et al., 2020; Devlin et al., 2019; Raffel et al., 2020).

Full fine-tuning of LLMs is computationally and memory intensive. Parameter-efficient fine-tuning (PEFT) methods such as Low-Rank Adaptation (LoRA) address this by freezing the pre-trained weights and learning only a small number of additional parameters (Hu et al., 2022). Concretely, let $W_0 \in \mathbb{R}^{d\times k}$ denote a frozen pre-trained weight matrix. LoRA introduces two trainable low-rank matrices $A \in \mathbb{R}^{r\times k}$ and $B \in \mathbb{R}^{d\times r}$ with $r \ll \min(d, k)$ and parameterizes the update as

$$\Delta W = BA. \tag{2}$$

The forward pass changes from

$$h = W_0 x \tag{3}$$

to

$$h = (W_0 + BA)\, x, \tag{4}$$

while only A and B are trained. This drastically reduces the number of trainable parameters and the optimizer state, with little to no loss in downstream quality, typically by inserting low-rank adapters into the attention projection matrices W_q, W_k, W_v, W_o (Hu et al., 2022).

QLoRA extends this idea by quantizing the frozen base model to 4-bit precision and training the LoRA adapters in higher precision on top. This enables efficient fine-tuning of models with 7–13B parameters on a single GPU, while matching or exceeding the performance of full 16-bit fine-tuning on many benchmarks (Dettmers et al., 2023). In practice, PEFT/QLoRA delivers substantial memory savings and faster training by updating only a few million parameters, while producing modular adapters that can be stored, shared, composed across tasks, or merged back into the base model.

2.3.4 Prompt Engineering

Prompt engineering involves carefully designing input texts (prompts) to guide LLMs toward the desired outputs. Effective prompts specify the task, optionally provide context or demonstrations, and impose formatting or semantic constraints that reduce ambiguity. Depending on the prompt design and the provided examples, several approaches have emerged (P. Liu et al., 2023):

- **Zero-shot prompting** provides only a task description without examples. It establishes a minimal baseline and is often sufficient for simple mappings or when the model has strong prior exposure. However, it can be unreliable on complex or ambiguous inputs involving multiple constraints (Brown et al., 2020; P. Liu et al., 2023).

- **Role-play prompting** is a recent extension of zero-shot methods, where the model is instructed to take on a specific expert persona (e.g., “You are a database tutor”). This assignment of a role introduces inductive biases that improve reasoning, structure, and consistency in complex tasks, even without explicit demonstrations (Kong et al., 2024).

- **Few-shot prompting** offers a small set of input-output examples to illustrate the desired mapping. This can improve stylistic and structural consistency through in-context learning effects, though performance strongly depends on the selection, diversity, and ordering of examples (Brown et al., 2020).

- **Chain-of-Thought (CoT) prompting** is designed to encourage intermediate steps in reasoning by explicitly prompting decomposition of complex tasks. This is especially useful for compositional queries or multi-hop conditions where a single-step response may be insufficient (Wei et al., 2022).
- **Tool- and retrieval-augmented prompting** leverages external components. LLMs can be guided to call APIs, execute tools, or condition responses on retrieved context from external sources RAG, reducing hallucinations and improving factual grounding (Lewis et al., 2020; Schick et al., 2023).

In addition to prompt structure, the model's output is shaped by decoding parameters such as *temperature*, which controls randomness in token selection, and sampling strategies like *top-k* and *top-p*, which restrict candidate tokens by probability. Lower temperatures yield more deterministic outputs, while higher values increase lexical diversity (Holtzman et al., 2020).

Carefully constrained prompts using controlled natural language, such as fixed operator vocabularies, quoted literals, and strict surface templates, help ensure semantic precision and reduce drift between input structures and generated output (P. Liu et al., 2023; Zhang et al., 2023). Although prompt engineering defines structure and supervision, decoding strategies such as self-consistency (X. Wang et al., 2023) and model fine-tuning (Chung et al., 2024; Wei et al., 2022) enhance robustness, consistency, and generalization.

3 Related Work

This chapter reviews relevant literature on natural language interfaces for executable queries and on techniques for constructing synthetic datasets. The aim is to situate the present work within this broader research context. It first summarizes the state of the art in *Natural Language to Query (NL2Query)*, knowledge graph question answering (KGQA), and property graph querying, followed by approaches for constructing suitable datasets. Finally, the present work is positioned within this context.

3.1 Natural Language to Query

Research into the translation of natural language into executable database queries has made great progress, particularly in the area of text-to-SQL. A key driver of this development was the *Spider* benchmark (Yu et al., 2018), a comprehensive cross-domain dataset explicitly designed to test generalization to unseen database schemas. Containing over 10 000 natural language questions and SQL queries from 200 databases across 138 domains, without overlap between training and test data, *Spider* has established itself as the de facto standard for schema generalization evaluation. This has led to advances in areas such as schema linking, structure-aware encoding and constrained decoding.

IRNet (J. Guo et al., 2019) introduced a sketch-based neural parser with precise schema linking that improves the alignment between linguistic expressions and structural elements such as tables, columns and relationships. *RAT-SQL* (B. Wang et al., 2019) built on this by employing a transformer model that jointly encodes question tokens and schema elements to enhance transferability across databases. Earlier, *Seq2SQL* (Zhong et al., 2018) applied reinforcement learning to gradually improve query generation through reward-based optimization.

More recent systems combine LLMs with grammar-constrained decoding and execution guidance to produce syntactically valid and semantically executable SQL queries (Yang et al., 2024). Structured inputs, such as explicit schema representations and accurate tokenization, have proven essential, especially for complex compositional queries. Nevertheless, the black-box nature of LLMs poses challenges. Ambiguous or insufficiently specified inputs often lead to hallucinations or the generation of invalid or incorrect queries (Y. Liu et al., 2023).

In addition to classic text-to-SQL systems, hybrid methods that incorporate actual database content are gaining traction. For example *DART-SQL* (Mao et al., 2024) adopts a data-driven approach in which the questions are rephrased and grounded in real table values and iteratively refined through execution-based feedback.

In related domains such as natural language interfaces to knowledge graphs (KGQA), the focus is on the precise mapping of query terms to entities and representation of multi-hop constraints over relations, which are crucial for capturing complex conditions or filters. Despite structural differences, such systems face many of the same challenges as SQL-based approaches, namely generalization across domains, semantic grounding in the underlying schema and the generation of executable queries (Agarwal et al., 2024; Trivedi et al., 2017).

3.2 Dataset Generation Techniques

Scraped and mined corpora A significant portion of NL2Query resources is derived from scraped or mined data, like public knowledge graphs (e.g., *Wikidata/DBpedia/Freebase*), Web tables, SQL dumps of open databases, and sometimes query logs or Q&A forums. These sources offer scale and schema diversity at low marginal cost, but they introduce well-known challenges, like noise and drift (incomplete or outdated schemas and values), weak alignment between text and executable queries, licensing constraints and potential benchmark leakage, and executability gaps that require normalization and execution filtering (Fürst et al., 2025).

Human-annotated corpora High-quality datasets such as Spider or Knowledge Graph Question Answering (KGQA) resources are based on expert annotation, usually by domain experts or via crowdsourcing. This results in precise and richly structured pairs of natural language and formal queries. However, such datasets are expensive, difficult to transfer, and prone to obsolescence when schemas change, which increases the need for synthetic or semi-synthetic generation (C. Wang et al., 2018).

Template- and schema-guided synthesis A common route is to generate structurally valid queries against a known schema and verbalize them via templates or controlled paraphrasing. This yields strong coverage of operators, filter logic, and joins/edges without exhaustive language curation. In property-graph settings, prior work has generated Cypher from templates and then derived questions from patterns, followed by neural refinement for fluency (Mao et al., 2024).

LLM-driven synthetic data and self-refinement Instruction-tuned LLMs now support scalable creation, diversification, and verification of question-query pairs. Techniques such as self-instruction or reinforcement learning from human feedback (RLHF), for example in *InstructGPT*, enable language models to be used for automatic paraphrasing, targeted structural modification (e.g., filters, joins) and generating adversarial examples. In the field of NL2Query, LLMs are increasingly supporting tasks such as consistency checking and linguistic refinement of synthetic data (Y. Liu et al., 2023; Yang et al., 2024).

3.3 Positioning of this Thesis

Unlike prior work that depends on externally scraped corpora, this thesis follows a *structure-first* paradigm tailored to property graphs and AQL. In contrast to prior work that relies primarily on textual templates or end-to-end generation, this design keeps *graph structure* and *executability* in the driver's seat: the queries originate from valid paths in the real database, and the language component adapts to that structure. This reduces schema drift, supports heterogeneous constraints typical for AQL's multi-model setting, and isolates linguistic variability from structural correctness (Q. Guo et al., 2024; Opitz & Hochgeschwender, 2022).

- **Path synthesis on the actual graph.** Paths are constructed directly on the target database using DFS/random-walk variants. This keeps topology (depth, directions) and edge collections grounded in reality rather than in templates.

- **Schema-informed query construction.** Paths are converted to an intermediate JSON representation and compiled into AQL. Attribute filters are injected from observed schema/value catalogs.

- **Tight control over executability.** Because queries originate from valid paths in the database and adhere to collection/edge definitions, executability is prioritized over linguistic variety.

- **Role of LLMs.** Large models are used to verbalize questions from the structural patterns, paired with an automatic consistency check, after which an LLM is fine-tuned. This limits the risk of hallucinations and decouples linguistic variation from structural correctness.

In contrast to previous approaches, the contribution of this work is to use the graph structure itself as the primary source of truth and to generate the natural language on this basis. This distinguishes the approach from both purely template-based pipelines and fully open LLM decodings. Crucial properties such as the traversal semantics of AQL, for example directions such as `OUTBOUND`, `INBOUND` or `ANY` as well as the traversal depth, are explicitly retained. AQL-specific properties such as the combination of graph and document conditions are also taken into account. The aim is to avoid schema drift, reliably map heterogeneous constraints and enable evaluation along two clearly separated dimensions: the execution accuracy of the generated AQL queries and the semantic fidelity of the corresponding natural language questions.

4 Method

This chapter outlines the methodology in two parts. First, the core processing pipeline (Section 4.2) is described, which transforms raw structural information from the underlying graph database into a clean and linguistically refined dataset of question-query pairs. Each stage produces the input for the next stage, ensuring that the resulting training data is (i) clean, (ii) linguistic natural, and (iii) structurally consistent and efficient.

Second, additional analyses and experiments (Section 4.3) are presented to characterize the dataset and demonstrate its practical utility. These include a descriptive statistical analysis of the cleaned corpus and a downstream fine-tuning experiment that uses the dataset to adapt a pre-trained LLM for AQL generation.

4.1 Overview

The pipeline begins with **schema mining and path sampling** (Section 4.2.1), where the database schema is analyzed to extract structural patterns. Path-based sampling is used to capture representative node and edge configurations. Various statistics, such as path length distributions, operator frequencies, and attribute coverage, are computed to characterize the dataset and guide the subsequent augmentation stage.

In **filter augmentation** (Section 4.2.2), the sampled query skeletons are enriched with constraints by inserting filters on node and edge attributes. To increase query diversity, both filter density and the types of comparison operators are systematically varied. This includes operators such as equality, range of comparisons, and substring matches. The variation follows a probabilistic configuration to ensure that the resulting queries remain syntactically valid and structurally consistent with the underlying schema.

In the next stage, **AQL linearization and compilation** (Section 4.2.3), each structured query graph is mapped to two forms: an executable AQL statement and a Cypher-inspired linearization that compactly preserves its structure and filters for subsequent verbalization.

LLM-based verbalization (Section 4.2.4) converts the linearized structures into concise natural-language questions while preserving all constraints. Prompting and post-processing guard against hallucinated schema elements.

The generated question-query pairs are then subjected to **heuristic verification** (Section 4.2.5) to check schema conformance. Queries that do not meet these checks are flagged for review or excluded from the dataset.

In **post-hoc relabeling and dataset cleaning** (Section 4.2.6), systematic errors are corrected. One major issue addressed in this work concerns temporal filters: a significant proportion of false negatives in date-based constraints could be resolved by adjusting the handling of timestamp granularity. This targeted correction stage led to a measurable improvement in the dataset's heuristic validity.

The **question simplification** stage (Section 4.2.7) aims to improve linguistic naturalness and reduce unnecessary complexity, of the generated questions. Two approaches are employed:

- An LLM-based rewrite process that merges technical fragments into cohesive, human-readable questions.
- A rule-based "naturalization" process that replaces technical identifiers with domain-relevant descriptions and applies grammar normalization.

Finally, **data preparation** (Section 4.2.8) compacts the schema representation and constructs the training, test, and verification splits for efficient training.

Beyond the pipeline, **dataset statistics and analysis** (Section 4.3.1) provide a quantitative characterization of the cleaned corpus. Distributional statistics and correlation studies identify structural and linguistic factors affecting query validity, offering insights into both coverage and limitations of the dataset. In addition, **fine-tuning** (Section 4.3.2) adapts a pre-trained LLM for AQL generation. Using parameter-efficient adapters, the model is trained on the question–query pairs and evaluated with text-based similarity metrics and execution accuracy. This experiment demonstrates the dataset's practical utility for training specialized query generation models.

Overall, the methodology integrates structural analysis, data augmentation, linguistic refinement, and model adaptation into a coherent workflow. This ensures that the final model is trained on high-quality, representative, and efficient input, maximizing both execution correctness and linguistic alignment in generated queries. Figure 4.1 illustrates this process in a high-level schematic, highlighting the fundamental steps from graph-grounded query construction to the final dataset of question–AQL pairs.

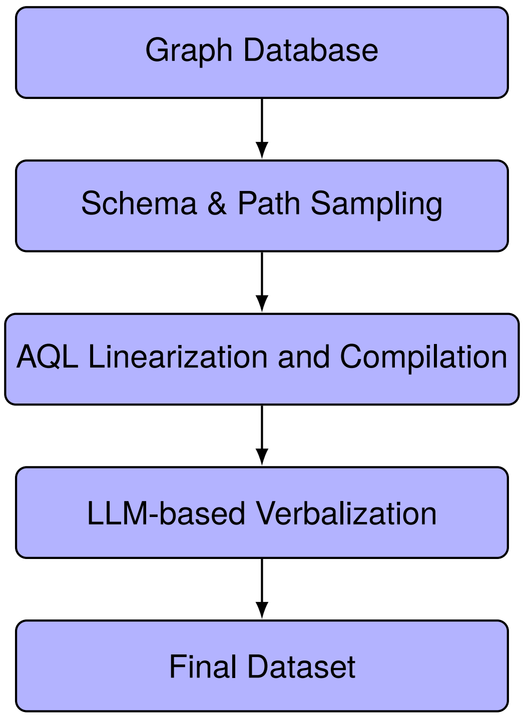

Figure 4.1: High-level overview of the question-to-AQL pipeline. The figure abstracts from implementation details and presents the core transformation steps in a fundamental visual form. The process begins with schema and path sampling from the graph database, continuing through AQL linearization and compilation, and concluding with LLM-based verbalization, which together yield the final dataset of question–AQL pairs.

4.2 Pipeline Stages

The following stages constitute the core data generation pipeline. Starting from schema extraction, each stage incrementally augments, verifies, or refines the data until a cleaned and compact dataset of question–query pairs is obtained.

4.2.1 Schema Mining & Path Sampling

The process begins with the extraction of the structural schema from the underlying property graph. A dedicated `schema_analysis` module, samples each node and edge collection to identify attribute names, their associated data types, and the relationships between entity types. The resulting structured schema includes:

- `node_props` and `edge_props` - mappings from attribute names to type labels,
- a list of typed relationships in `(from) - [: TYPE {}] -> (to)` format, and
- a catalog of representative attribute values for later use in filter generation.

This representation provides both a compact description of the database structure and a data based view of available attributes.

An excerpt of a sampled schema is shown in Listing 4.1.

```
{
  "node_props": {
    "AbbrvNode": {
      "_key": "TEXT",
      "_id": "TEXT",
      "_rev": "TEXT",
      "long-form": "TEXT",
      "abbrv": "TEXT",
      "timestamp": "DATE"
    },
    "EntityNode": {
      "_key": "TEXT",
      "_id": "TEXT",
      "_rev": "TEXT"
    }, ...
  },
  "edge_props": {
    "HasAbbreviation": {
      "_from": "TEXT",
      "_to": "TEXT",
      "_rev": "TEXT",
      "count": "NUMBER",
      "timestamp": "DATE"
    }, ...
  },
  "relationships": [
    {
      "from": "TextNode",
      "type": "HasAbbreviation",
      "to": "AbbrvNode",
      "cypher": "(TextNode) -[:HasAbbreviation {}]-> (AbbrvNode)"
    }, ...
  ]
}
```

Listing 4.1: Excerpt of a structured schema (sampled from the `InsightsNet` database).

To collect representative structural patterns, DFS is used in a random-walk mode. At each expansion step, neighbor lists are shuffled to promote structural diversity. DFS is preferred over BFS, as it naturally yields longer and more coherent paths and is better suited to randomization. The starting nodes are randomly selected across all node collections, with preference for previously unvisited candidates. This ensures balanced coverage of the graph's entity types from the outset. In line with prior analyses indicating that real-world graph queries rarely exceed three hops (Sun et al., 2021; Tang et al., 2023), this work samples each path length with a fixed per-depth budget (Table 5.2). This enforces balanced coverage across path lengths and prevents deep, infrequent structures from dominating the dataset. The neighbor expansion is also limited to a fixed number of candidates per step. Exact cycles are excluded, and a path is only accepted if its schema signature—defined by collection, edge-type and direction—is unique at that depth. A subset of sampled paths is extended into "OR-variants" by branching at a random position and adding one or two alternative continuations, provided that these remain unique. Traversal continues until the depth-specific budget is met or plateau detection indicates that further sampling is unlikely to yield new structures. To detect such plateaus, the sampling process monitors the moving average of newly discovered unique paths over the last W iterations:

$$\frac{1}{W} \sum_{k=t-W+1}^{t} \Delta_k \ , \tag{5}$$

where Δ_k denotes the number of new unique paths identified in iterations k. Sampling halts when this average falls at or below a predefined stopping threshold τ. If no improvement occurs within the stagnation window, the sampler is restarted to escape local plateaus. This criterion is integrated into the core mechanics of this sampling logic as follows:

Algorithm 1 Randomized DFS with per-depth budgets and plateau stopping

```
1:  function SAMPLE_PATHS($G, depths, budget\_per\_depth$):
2:      for $depth \in depths$:
3:          $P \leftarrow \emptyset$
4:          while $|P| < budget\_per\_depth$ and not plateau():
5:              $starts \leftarrow$ pick_unvisited_start_nodes()
6:              for $v_0 \in starts$:
7:                  DFS($v_0, [v_0], 0, depth$)
8:
9:  function DFS($v, path, depth, max\_depth$):
10:     if $depth = max\_depth$:
11:         store_signature($path$)
12:         return
13:     shuffle(neighbors($v$))
14:     for $u \in$ neighbors($v$):
15:         if $u \notin path$:
16:             DFS($u, path + [u], depth + 1, max\_depth$)
```

This combination of schema analysis, randomized DFS, and plateau-controlled stopping produces a structurally diverse depth-balanced set of query skeletons, forming a robust basis for subsequent augmentation and natural language generation. All fixed values for depths, budgets, sampling limits, and stopping thresholds are summarized in Table 5.2 for reproducibility.

4.2.2 Filter Augmentation

After structural paths have been sampled, attribute-level predicates are attached to transform them into executable query skeletons. For each node or edge on a path, candidate attributes are drawn from the schema's value catalog, and operator–type compatibility is enforced. Operators are sampled from type-compatible sets with a per-path diversity constraint (no repeated semantic group). The implemented groups cover *equality*, *text matching*, *numeric comparison*, and *date comparison*. Representative values are taken from the catalog, and attributes without cataloged values are skipped. A bounded per-structure budget governs the total number of predicates, and multiple predicates may attach to a single element. Edge predicates are inserted only with low probability, keeping most constraints on nodes. Validity is maintained by relying on schema-observed attributes, enforcing operator–type compatibility, and avoiding duplicate attribute–value selections.

An illustrative, domain-neutral JSON instance of an augmented path from a `Person` node to an `Organization` node via `WorksAt` in the internal `json_query` format is shown in Listing 4.2. Filters appear under `userData`.

```
{
  "nodes": [
    { "id": "0", "collection": "Person",
      "userData": [
        { "attribute": "birthYear", "attributeType": "number",
          "operator": { "value": "numeric_larger_than" }, "value": 1980
          ↪  },
        { "attribute": "country", "attributeType": "text",
          "operator": { "value": "equals" }, "value": "Germany" }
      ]},
    { "id": "1", "collection": "Organization",
      "userData": [
        { "attribute": "name", "attributeType": "text",
          "operator": { "value": "contains" }, "value": "University" }
      ]}
  ],
  "edges": [
    { "id": "0", "source": "0", "target": "1", "name": "WorksAt",
      "userData": [
        { "attribute": "since", "attributeType": "date",
          "operator": { "value": "date_larger_or_equal" }, "value":
          ↪  "2020-01-01" }
      ]}
  ]
}
```

Listing 4.2: Filter augmentation on a sampled path.

4.2.3 AQL Linearization & Compilation

For each filtered path, a *paired* representation is produced consisting of (i) an executable AQL statement for database execution, and (ii) a Cypher-like linearization that serves as a canonical, human-readable input for the LLM-based verbalization stage (Section 4.2.4).Both forms are generated from the same structured JSON format (`json_query`) that specifies collections, relationships, and optional attribute filters.

The AQL compilation builds upon components originally developed in Graph Detective by Opitz et al. (2024). From each `json_query`, the compiler (i) materializes collection-specific start-node subqueries whose `FILTER` clauses are induced by node predicates, (ii) emits a graph-traversal clause in which edge directions (INBOUND-/OUTBOUND/ANY) are derived from the graph definition, and (iii) injects a path-level filter that applies node and edge predicates conjunctively per index while combining alternative path variants disjunctively. Explicit lists of admissible vertex/edge collections and `LIMIT`s are added, yielding an executable statement. Unlike the original implementation, which relied on parameter placeholders, this work embeds the actual filter values directly into the AQL statements.

Cypher-like linearization Each structured query (`json_query`) is deterministically serialized into a compact, Cypher-inspired string representation. This linearization lists nodes, edges, and all attached filters in a fixed order, preserving the full structural and semantic content of the query in a concise textual format. It is both stable and human-readable, and serves as a canonical form for deduplication as well as the primary input to the subsequent verbalization stage. A representative example, corresponding to the augmented path in Listing 4.2, is shown in Listing 4.3.

```
(Person { birthYear numeric_larger_than 1980, country equals 'Germany' })
  -[WorksAt { since date_larger_or_equal '2020-01-01' }]->
  (Organization { name contains 'University' })
```

Listing 4.3: Cypher-like linearization corresponding to Listing 4.2.

AQL compilation The same `json_query` is compiled into an executable AQL statement by materializing collection-specific start-node subqueries, emitting a direction-aware traversal, and injecting a path-level filter; the resulting plain AQL is shown in Listing 4.4:

```
LET start_nodes = FLATTEN(
  FOR doc IN Person
    FILTER doc.birthYear > 1980
      AND LOWER(doc.country) == LOWER("Germany")
    LIMIT 10
    RETURN doc
)

FOR start_node IN start_nodes
  FOR v, e, p IN 1..1 OUTBOUND start_node GRAPH "InsightsNetGraph"
    FILTER IS_SAME_COLLECTION(e, "WorksAt")
       AND IS_SAME_COLLECTION(v, "Organization")
       AND CONTAINS(LOWER(v.name), LOWER("University"))
       AND DATE_TIMESTAMP(e.since) >= DATE_TIMESTAMP("2020-01-01")
    LIMIT 10
    RETURN p
```

Listing 4.4: Schematic plain AQL corresponding to Listing 4.2.

This pairing ensures that a single, structured specification (`json_query`) yields both a stable, human-readable linearization and an executable database query. Sharing the same structural origin ensures semantic consistency across all pipeline stages.

4.2.4 LLM-based Question Generation

Following the Cypher-like linearization described in Section 4.2.3, each structural pattern is rewritten as exactly one fluent English question that preserves every node, relationship, and filter condition. This is treated as a one-to-one rewriting task in which all constraints, including literal values, appear in the output with canonical operator wording and explicit enumeration for multi-value conditions.

Configuration search To balance linguistic quality and constraint fidelity, a grid search was conducted over (i) prompt styles, including a strict instruction format and several few-shot variants with role priming, (ii) several instruction-tuned LLMs, and (iii) a range of decoding temperatures. The exact search space is listed in Table 5.3.

For each `(model, prompt, temperature)` combination, a fixed set of patterns was rendered and scored by two independent evaluators: a deterministic heuristic (Section 4.2.5) and an LLM-based "judge", a separately instructed model returning a binary `valid/reason` verdict. An agreement score between the two was computed, and the final selection metric was the mean of heuristic, LLM, and agreement percentages.

Final configuration The most promising combination of model, prompt, and temperature achieved a combined score of 88.9%, with 91.7% in both heuristic and LLM evaluations and 83.3% agreement. The selected model with its configuration (see Table 5.4) was adopted for all production runs.

Execution The complete system prompt, containing explicit rewriting rules and multiple input–output examples, is given in Appendix A.1. During generation, this prompt was injected as the `system` message. The script processes inputs in small batches, streams completions via an OpenAI-compatible API, and persists outputs together with metadata `{cypher, question, model, temperature, prompt_variant}`. Checkpointing and index tracking allow the process to resume seamlessly in case of interruptions. For the linearized query in Listing 4.3, the LLM-based verbalization yields the following natural-language question, which preserves all structural and filter constraints (Listing 4.5):

```
Which persons whose birthYear is after 1980 and whose country equals
 ↪  'Germany' are linked by WorksAt since on or after January 1 2020 to
 ↪  organizations whose name contains 'University'?
```

Listing 4.5: LLM-based verbalization of the Cypher-like linearization in Listing 4.3.

4.2.5 Verification

To ensure that generated questions faithfully reflect the constraints of their originating query patterns, two automatic verification mechanisms were employed. Their goals were (i) to maintain consistency at scale between natural language questions and Cypher-like patterns, and (ii) to provide an objective, reproducible quality measure without manually reviewing all 60 000 cases. The pipeline detects mismatches in constraints, operators, and structural relations, serving both as a high-precision filter and as an evaluation tool.

Deterministic heuristic The production verifier is a deterministic, rule-based heuristic. It parses the original Cypher-like pattern into `(field, operator, value)` triples, applies type-specific normalization (text, numeric, date, multi-value), and scans the question for corresponding mentions. Operators are canonicalized (e.g., *equals* $\rightarrow$ "equals", "is equal to"), values normalized by type, and multi-value constraints checked via explicit enumeration.

To handle paraphrasing and lexical variation, a staged matching process is used:

- **Tokenization and lemmatization:** Split text into tokens and reduce to base forms, enabling matches across inflected forms.
- **Synonym lookup:** Retrieve lemma-level synonyms so that semantically equivalent expressions are matched.
- **Sentence-level embeddings:** Encode the candidate value phrase and its corresponding question span using a sentence-transformer (see Table 5.5 for model details). After L2-normalization, cosine similarity is computed:

$$\mathrm{sim}(a, b) = \frac{a \cdot b}{\|a\|\,\|b\|}. \tag{6}$$

  Matches are accepted if $\mathrm{sim} \geq 0.65$, a threshold chosen to minimize false positives while covering common paraphrases.

Semantic checks run only if simpler lexical matches fail, and all matches are anchored to operator keywords in the same sentence span to preserve constraint semantics.

LLM judge During configuration search (Section 4.2.4), a strictly instructed LLM was used as a secondary verifier. It receives both the Cypher-like pattern and the generated question, and returns a JSON verdict {`valid`, `reason`}. The LLM judge was stricter on operator semantics but tolerant to minor grammar variation, providing an independent, learned perspective complementary to the heuristic.

Statistical confidence via human annotation Although the heuristic was run on all 60 000 questions, its precision was estimated via manual annotation of a statistically representative sample. The required sample size was computed using the finite population correction:

$$n = \frac{N \cdot z^2 \cdot p(1-p)}{E^2 \cdot (N-1) + z^2 \cdot p(1-p)}, \tag{7}$$

with $N = 60\,000$, $z = 1.96$ (95% confidence), $p = 0.5$ (worst case), $E = 0.10$ (±10 percentage points, pp), yielding $n \approx 96$. A random sample of this size was manually validated, and Wilson confidence intervals used to estimate the heuristic's true accuracy.

4.2.6 Post-hoc Relabeling & Dataset Cleaning

To further improve the accuracy of the automatically assigned validity labels, a post-hoc relabeling stage was applied. During verification (Section 4.2.5), the deterministic heuristic occasionally misclassified queries involving timestamps of the form `YYYY-MM-DDhh:mm:ss`. Although the question and pattern often referred to the same calendar day, and in many cases even the same time, they were sometimes marked as invalid due to strict granularity checks during time comparisons. This behavior is consistent with the heuristic's design, which prioritizes precision over recall, but it led to a measurable number of false negatives in otherwise correct date-based constraints.

From the `verification_report`, all *invalid* cases involving date operators were extracted and grouped into three categories:

(i) `writer_missed_date`: the query contains a `date_*` operator, but the question contains no date expression,

(ii) `likely_ts_false_negative`: the question contains a date, all mismatches concern date operators, and at least one compared value is a timestamp,

(iii) `other_date_mismatch`: all remaining date-related mismatches.

In category (ii), the date in question and the date in the query usually matched at the calendar-day level, often including an identical time component. Such mismatches are therefore likely due to differences in timestamp granularity rather than genuine semantic errors.

A strict relabeling rule flipped a case from *invalid* to *valid* only if: the query contained a date operator; the question contained a date string; all mismatches concerned date operators; the date in question and in the query were identical at the calendar-day level; only timestamp-family fields were involved; and the question wording matched the operator semantics.

4.2.7 Question Simplification

The question generation stage (Section 4.2.4) produced grammatically correct outputs with complete coverage of query constraints. However, their style reflected the few-shot prompts and schema-based phrasing used during generation. To adapt these questions for broader usability and to increase variation in question openings, a simplification stage was introduced. Its objectives were to (i) remove technical phrasing and (ii) improve linguistic naturalness and diversity in question openings, while strictly preserving all query constraints.

LLM-based rewrite All questions marked as *valid* were rewritten using a large instruction-tuned language model. A fixed system prompt instructed the model to use everyday language, avoid schema names and IDs, merge subclauses into a single coherent question, and preserve all literal values and operator semantics. Two prompt variants were used: a neutral version allowing any context-appropriate opening word, and an alternative explicitly discouraging *Which* to increase syntactic diversity (full prompt text in Appendix A.5). Multiple subclauses were consistently interpreted as conjunctive (`AND`). Processing included checkpoint and retry logic for robustness.

Rule-based naturalization In parallel, a strict rule-based transformation pipeline was applied to the original questions. It replaced technical node and edge type names with human-readable terms, removed schema-specific field markers, compacted IDs and corrected their grammar, and applied general grammar clean-ups. Each transformation step was logged for auditability.

Semantic integrity Both routes strictly preserved all literal values, operators, and constraint logic; no schema identifiers were introduced; timestamps and numeric ranges remained unchanged; and multiple conditions were fused under `AND` semantics. For subsequent experiments, only the LLM-based simplifications were retained, as they showed clear qualitative and quantitative advantages over the rule-based output.

An illustrative example of a question rewritten via the LLM-based simplification is shown in Listing 4.6. It demonstrates the transformation from a schema-based formulation to a more natural, readable version, while preserving all constraints.

```
Who are the people born after 1980, from Germany, who have been working
 ↪  since January 1, 2020, at places with "University" in their name?
```

Listing 4.6: Example of simplified natural-language question corresponding to the verbalization in Listing 4.5.

4.2.8 Data Preparation

To prepare the data for SFT and ensure efficient use of the model's context window, two steps were performed: (i) compaction of the database schema to reduce token usage without losing structural information, and (ii) construction of the final training, test, and verification splits.

Schema compaction The original schema text, shared across all samples, contained extensive type annotations and repeated property definitions. To address this redundancy, a compaction procedure was applied that preserved all structural and naming information relevant for query generation. This was motivated by efficiency considerations and supported by empirical findings: longer contexts can impair retrieval of mid-position information (N. F. Liu et al., 2024), whereas targeted context compression can preserve model performance while substantially reducing token usage (Y. Li et al., 2023).

The transformation applied the following rules:

- **Default type declaration:** Declare the most frequent field type for nodes and edges once globally, omitting explicit `TYPE` annotations when matching this default.
- **Common property hoisting:** List properties present in all nodes or all edges once in a *common props* block and remove them from per-collection listings.
- **Star marker:** Use a * suffix to mark node types containing the full set {_key, _id, _rev, timestamp}, omitting these fields inline.
- **Edge grouping:** Group edges by source, with the default relationship label declared globally and omitted inline when applicable.

This reduced schema length, freeing capacity for question and query content within the fixed model context.

SFT dataset construction From the post-hoc corrected dataset (Section 4.2.6), only *valid* entries were retained. For each remaining sample, the compact schema text was combined with its simplified question (Section 4.2.7) and corresponding AQL query in the following format:

```
<SCHEMA> ...</SCHEMA>
<QUESTION> ...</QUESTION>
<AQL> ...</AQL>
```

The resulting corpus was randomly split into 80% training, 10% test, and 10% verification to maximize the size of the training corpus while retaining a dedicated hold-out set for evaluation and potential refinement. Maintaining such a distinct verification set aligns with established best practices in supervised learning, where dedicated development data supports reliable model selection and mitigates overfitting (Xu & Goodacre, 2018).

Dataset By reducing redundant schema tokens, more context was available for the question and AQL content, supporting both efficiency and fidelity during model fine-tuning. The preparation stage ensured that all training samples contained complete schema information in compact form, natural language questions, and precise AQL targets.

4.3 Additional Analyses & Experiments

Beyond the pipeline itself, two complementary components were added to characterize and assess the resulting dataset. First, a statistical analysis (Section 4.3.1) quantifies structural and linguistic properties of the corpus. Second, a downstream fine-tuning experiment (Section 4.3.2) demonstrates the dataset's practical utility for adapting a pre-trained LLM to AQL generation.

4.3.1 Dataset Statistics & Analysis

To characterize the cleaned corpus and to establish descriptive baselines for subsequent modeling, structural and textual descriptors were extracted from the post-hoc corrected dataset in combination with the schema catalog. For each query, the JSON representation and its annotation block were parsed to obtain metadata describing structural features, operator usage, textual characteristics, and schema coverage.

Feature extraction The computed feature categories comprise:

- **Structure:** path length, total number of filters, filters per path element, branching indicator, and presence of `OR`-clauses.
- **Operators:** counts per operator and mapping to semantic groups (*text containment*, *text equality*, *prefix/suffix matches*, *numeric comparisons*, *date comparisons*, *ID comparisons*).
- **Textual metrics:** for all string-based filters, the maximum literal length in both words and tokens, grouped into buckets for subsequent correlation analyses, along with the total number of text-based comparisons per query.
- **Schema coverage:** usage counts of node collections, edge types, and their attributes to measure coverage at multiple levels,
- **Linguistic openings:** inspection of opening-word usage (e.g., *Which*, *What*, *Who*) in each question.

Aggregation and visualization Structural features were aggregated by path length to reveal patterns in filter density, branching, and logical composition. Operator statistics were examined individually and by semantic group, including frequency variation across depths. Schema coverage was summarized in tables and a bipartite *coverage graph* highlighting used node/edge attributes. Textual metrics were similarly aggregated to inspect length distributions and their relation to validity.

Exploratory analysis Targeted analyses tested hypotheses about structural and textual influences on heuristic validity, e.g., whether deeper paths reduce validity, higher filter density increases mismatch rates, or long literals affect parsing reliability. These analyses served to evaluate the dataset's validity, consistency, and coverage, thereby confirming its suitability as a foundation for subsequent modeling.

4.3.2 Fine-tuning

To demonstrate the practical utility of the dataset, a parameter-efficient fine-tuning strategy was applied to a pre-trained LLM for translating natural-language questions into syntactically valid ArangoDB AQL queries. Training minimized the standard next-token loss using a structured input-output format. QLoRA-style 4-bit quantization was applied for memory efficiency, with LoRA adapters inserted into the attention layers.

This setup offers three main advantages:

- **Retention of general language capabilities:** Keeping most parameters frozen preserves broad linguistic knowledge, while adapters specialize it for the question-to-AQL mapping.
- **Memory and compute efficiency:** LoRA freezes the base weights and trains only small low-rank adapters, reducing trainable parameters by orders of magnitude. QLoRA stores the frozen weights in 4-bit precision, enabling multi-billion-parameter fine-tuning on a single GPU with minimal quality loss.
- **Modularity:** LoRA adapters can be stored, re-loaded, or merged with different base models, facilitating reuse and rapid experimentation.

Input formatting Training examples were structured as short chat dialogues:

- `System prompt:` instructed the model to output exactly one valid AQL query, wrapped in `[AQL]...[/AQL]` tags, without explanations or extra text.
- `User prompt:` contained the compact database schema in a `[SCHEMA]...[/SCHEMA]` block and the natural-language question in a `[QUESTION]...[/QUESTION]` block.

During fine-tuning, the *gold* AQL query was appended as the assistant's reply in the `[AQL]` block. Here, *gold* denotes a single canonical reference query deterministically compiled from the structured representation. It serves as the training target, while acknowledging that semantically equivalent AQL formulations (e.g., reversing an INBOUND/OUTBOUND traversal or reordering filter placement) may also exist. This strict tagging scheme ensures unambiguous parsing during evaluation and enforces consistent output formatting.

Model selection Several instruction-tuned base models between 1.5B and 7B parameters were screened in a short debug phase on a reduced dataset with varied hyperparameters. Selection on a separate verification split prioritized the best compute–performance trade-off based on the metrics described in *Performance evaluation.* The selected model was then fine-tuned on the full data with LoRA adapters under 4-bit (QLoRA) quantization.

Performance evaluation Evaluation follows a three-part protocol.

(i) Text-based similarity and well-formedness. Predictions and references are minimally normalized (lowercasing, whitespace compaction, quote normalization, comment stripping) before metric computation. *BLEU* measures how many short token sequences (*n*-grams) in the generated query match the reference, indicating local phrasing accuracy. *ROUGE-L* (F1) measures the longest common subsequence, reflecting global structure and ordering. *Levenshtein distance* counts the minimal number of single-token edits needed to match the reference, quantifying syntactic deviation. Together, these metrics capture local accuracy, global structure, and overall syntactic closeness. Three well-formedness flags are recorded: *AQL-like* (regex anchored at

leading AQL keywords), *EndedOnTag* (presence of `[/AQL]`), and *TruncatedByLen* (generation hit the token cap).

(ii) Execution-based evaluation. On the verification split, gold and predicted queries are executed against the production database. Reported are *GoldOK coverage* (share of items whose gold query executes successfully) and *Execution accuracy* computed only on GoldOK items by comparing canonicalized JSON results with a small floating tolerance. Runtime failures are mapped to coarse categories (e.g., SYNTAXERROR, COLLECTIONMISSING, RUNTIMEERROR, TIMEOUT) and augmented by structural labels (WC, WJ, WF, EC, SE, IQ; definitions and frequencies in Appendix A.10).

(iii) Out-of-distribution (OOD) and robustness probes. To test generalization beyond the verification split, six hand-written questions adapted from Opitz et al. (2024) are evaluated with the same text-based metrics and executed against the database; deviations are analyzed side-by-side (start-collection shifts, edge name/direction, filter semantics, traversal depth). In addition, a syntactically well-formed but semantically nonsensical prompt probes behavior under low-signal input. These OOD and nonsense cases are reported separately for qualitative error analysis and are not included in aggregate verification metrics (examples in Appendix A.11, A.6). This separation ensures that aggregate scores remain strictly comparable, while qualitative probes provide additional insight into the model's failure modes and edge-case behavior.

5 General Setup

This chapter summarizes the experimental environment, software stack, data artifacts, and configuration parameters to ensure full reproducibility. All stages described in Section 4 were executed on a live *InsightsNet* property graph instance, with intermediate and final outputs versioned throughout the pipeline.

5.1 System Architecture

The NL2AQL pipeline is deployed in a modular, multi-stage environment spanning live database access, graph traversal, natural-language generation, verification, and fine-tuning. All stages share a unified infrastructure for compute, data access, and artifact storage, ensuring consistent execution and reproducibility across experiments.

Data layer The *InsightsNet*[4] property graph serves as the central data source for all experiments in this work. It is deployed as a production ArangoDB instance in multi-model mode and queried exclusively via AQL. Schema mining and path sampling operate directly on this live database to ensure up-to-date structural coverage (see Section 4.2.1 for details).

Compute & Hardware A dedicated Linux workstation (Table 5.1) executes all pipeline stages that do not involve LLM inference. These include schema mining and compaction; path sampling and filter augmentation; AQL compilation and linearization; dataset analysis; post-hoc relabeling; and preparation of fine-tuning data.

Table 5.1: Hardware and software specifications of the local workstation.

OS/Kernel	Ubuntu 22.04.5 LTS / 6.8
CPU	Intel Xeon W-2295 (18C/36T)
RAM	251 GiB
GPUs	2 × NVIDIA Quadro GV100 (32 GiB each)
NVIDIA driver/CUDA	550.144.03 / 12.4
Python	3.10.18
ArangoDB (server)	3.11

LLM execution Two venues are used for LLM execution workloads: (i) large-scale question generation runs remotely on the DLR Einstein server (a high-performance

[4]https://insightsnet.org/

endpoint at DLR) with an Ollama backend (Section 5.5); (ii) all other LLM-based components—including the LLM judge for verification, local question simplification, and fine-tuning with pre- and post-evaluation—run on the workstation in Table 5.1. This split maximizes generation throughput while avoiding local GPU memory contention and keeping development and evaluation self-contained.

Artifact storage All intermediate and final artifacts are stored in the project's `results/` directory, providing a single versioned source of truth for analysis and fine-tuning.

5.2 Datasets

Target graph *InsightsNet* connects multimodal digital objects such as web texts, publications, images, and videos with a focus on domains that include climate discourse and artificial intelligence. The data are stored in ArangoDB as a property graph with multiple nodes and edge collections and with tens of millions of edges in total. All queries in this thesis operate directly on that production instance.

5.3 Software & Versions

All experiments used Python 3.10.18 with pinned Conda+pip environments. Core components include ArangoDB 3.11 on the server side and the PyTorch/Transformers ecosystem for model inference and fine-tuning. The per-stage environment mapping and pin files are summarized in Appendix A.2. (Tables A.1 and A.2), with key package snapshots in Section A.4.

5.4 Traversal & Sampling Configuration

A fixed global random seed (42) is used, and all traversal and sampling parameters are managed through version-controlled configuration files. Key operational values such as traversal depths, per-depth budgets, neighbor expansion limits, and plateau-detection thresholds are summarized in Table 5.2.

All parameters are logged together with their corresponding artifacts so that every stage of traversal and sampling can be reproduced exactly.

Table 5.2: Selected configuration parameters for traversal and path sampling.

Traversal depths	0, 1, 2, 3
Paths per depth	15 000
Neighbor expansion limit	10
Schema sample size / value catalog	500 / 500
Maximum and average filters per path	4 and 2
Restart triggered after plateau	2 stagnant steps

5.5 LLM Generation Configuration

Natural-language question generation was executed on a DLR Einstein endpoint providing an OpenAI-compatible API backed by Ollama. An initial grid search (Table 5.3) explored variations in prompt style, model choice, and decoding temperature. Scoring combined heuristic validity, LLM judge validity, and their agreement rate.

Table 5.3: Search grid for LLM-based question generation.

Dimension	Values
Prompt styles	Strict instruction
	Few-shot variants with role priming
LLMs	`llama-pro`
	`llama-3.3`
	`mistral-small-24b-instruct`
	`llama-4-maverick-17b-128e-instruct`
Decoding temperatures	`0.0`
	`0.4`
	`0.7`
	`1.0`
Scoring metrics	Heuristic validity
	LLM judge validity
	Agreement

The configuration identified as optimal in the method section (Section 4.2.4) was used for all production runs and is detailed in Table 5.4.

5.6 Verification Configuration

The deterministic heuristic verifier (Section 4.2.5) employed the following tools and resources:

To ensure reproducibility beyond version pinning, the verifier runs fully offline (English-only) under a fixed seed and produces byte-identical outputs for identical inputs across reruns and machines. Preprocessing converts all text to a standard

Table 5.4: Production configuration for question generation.

Setting	Values
Endpoint	DLR Einstein
Model	`llama-4-maverick-17b-128e-instruct` (`q8_0`)
Client library	`openai 1.91.0` (OpenAI-compatible; Ollama backend)
Prompt family	`fewshot2` (Appendix A.1)
Decoding temperature	`0.7`
Output format	Exactly one question per Cypher-like pattern
Fault tolerance	Batched requests, checkpointing, retry logic
Persisted metadata	`{cypher, question, model, temperature, prompt_variant}`

Table 5.5: Core tools and models for automatic verification.

Component	Version / Identifier
Sentence transformer model	`all-MiniLM-L6-v2` (Hugging Face[5]); `sentence-transformer` 4.1.0
Tokenizer / lemmatizer	spaCy 3.8.6 (`en_core_web_md` 3.8.0)
Synonym lookup source	WordNet (NLTK 3.9.1)
Similarity threshold	0.65 (cosine; L2-normalized)

Unicode form and splits compound identifiers into words (e.g., `camelCase` → `camel case`). Dates are recognized in common formats such as ISO `YYYY-MM-DD`, `YYYYMMDD`, D/M/Y, and long English before matching. Embedding vectors are cached on-disk and reused between runs to eliminate variability from recomputation.

5.7 Fine-tuning Configuration

Two fine-tuning phases were conducted. The first was a debug phase using a reduced training set of 500 examples and an evaluation set of 100 examples to rapidly screen candidate base models and hyperparameter configurations, followed by a full production run using the best-performing setup.

Base models screened (debug runs) In the debug phase, five instruction-tuned base models of increasing parameter count were evaluated: `Qwen2-1.5B-Instruct`, `Phi-2`, `Mistral-7B-Instruct`, `Falcon-7B-Instruct`, and `CodeLlama-7B-Instruct`. This screening step served to identify a favorable trade-off between model

[5]https://huggingface.co

size, generation quality, and training efficiency prior to the full fine-tuning run.

Final model (full run) For the full run, `Qwen2-1.5B-Instruct` was selected based on its measured compute–performance trade-off in the debug screen, offering shorter training cycles while meeting the target quality thresholds.

Training configuration (final run)

- **Quantization:** 4-bit QLoRA (BitsAndBytes NF4, double quantization, compute in `float16`).
- **LoRA:** rank $r = 256$, $\alpha = 512$, dropout 0.05, init=`eva`[6]; target modules={`q_proj`, `k_proj`, `v_proj`, `up_proj`, `down_proj`, `gate_proj`, `dense`}.
- **Optimizer:** `paged_adamw_32bit`; weight decay 0.01.
- **Schedule:** cosine schedule with warmup ratio 0.05.
- **Batching:** per-device batch size 4; gradient accumulation 16 (effective batch size 64).
- **Epochs / length:** 3 epochs; max sequence length 2048; packing=`False`; group-by-length=`True`.
- **Learning rate:** 3×10^{-5}.
- **Logging / eval / save:** every 100 steps; keep best checkpoint only; Weights & Biases logging enabled.

5.8 Model Performance Evaluation

All execution-based performance measurements were conducted against the production *InsightsNet* ArangoDB instance. Unless stated otherwise, evaluations used $n = 1\,000$ items from `aql_verify_sample.jsonl` (seed 42) plus six additional questions adapted from Opitz et al. (2024). Inference employed deterministic decoding (`do_sample=False`) with a fixed stop on `[/AQL]`; execution accuracy is reported only on items whose gold query executes successfully (*GoldOK*).

[6]`eva` initializes LoRA adapter weights with small, scaled values (proportional to $1/r$) for stable low-rank updates, as implemented in the training stack.

5.9 Artifacts & Reproducibility

The following artifacts constitute the canonical record for analysis and training:

- `question_progress_posthoc_60k.json`:
  Final dataset with post-hoc label corrections and with 22 296 valid pairs out of 60 000
- `verification_report_60k.jsonl`:
  Per-sample verification evidence
- `dataset_schemas_60k.json`:
  Schema snapshot including a compact representation
- `aql_train.jsonl`, `aql_test.jsonl`, `aql_verify_sample.jsonl`:
  Splits for supervised fine-tuning
- `models/final_qwen2-1.5b-lora/`:
  Final fine-tuned LoRA adapters and merged model weights for Qwen2-1.5B-Instruct.

All artifacts are generated deterministically given the global seed and are stored in the `results/` directory. Fine-tuning checkpoints and training logs are written to the `models/` directory.

5.10 Execution Time Overview

The runtime of each pipeline stage depends on the execution venue and data volume. Table 5.6 summarizes observed wall-clock times for the major components.

Table 5.6: Wall-clock time overview for each pipeline stage.

Stage	Venue	Time (wall-clock)
Path generation for 60 000 items	Workstation	~4–5 h
Question generation for 60 000 items	DLR Einstein	~60 h
Heuristic verification	Workstation	~5–6 h
Post-hoc relabeling	Workstation	~20 min
Schema compaction	Workstation	~1–2 min
Analysis notebooks	Workstation	~20 min
Fine-tuning (final) with QLoRA	Workstation	~49 h
Inference & metrics 1 000 items	Workstation	~5 h

These timings are intended as a reproducibility budget rather than a benchmark. They document the effect of the remote–local split and the chosen scheduling policy: remote question generation and final fine-tuning form the long poles, while the remaining stages complete on the order of minutes to a few hours. All seed values and configuration files are logged with the artifacts so that deviations across runs can be attributed to venue or load rather than code changes. The complete implementation code for all pipeline components is provided; see Appendix A.3.

6 Evaluation

This chapter presents a comprehensive evaluation of the dataset, in the spirit of Spider Yu et al. (2018), examining coverage and model performance to determine whether schema-guided path sampling with LLM verbalization can automatically produce a high-quality question–AQL corpus suitable for training a functional question-to-AQL translator.

6.1 Dataset and Quality Analysis

Focus: RQ1 (corpus construction).

Data basis The dataset comprises 60 000 question–query pairs generated from graph traversals under the final configuration and corresponding verbalization pipeline described in Section 4.2.4. This corpus forms the foundation for the evaluations reported in this chapter.

Post-hoc relabeling A rule-based relabeling step addresses systematic timestamp-related errors in the heuristic. Three error classes were distinguished: (A) *writer missed date* (question without a date, query with a date operator), (B) *timestamp false negatives* (same calendar day, heuristic fails at second-level precision), and (C) *other date mismatches*. Robustly identifiable cases of types A/B were flipped from *invalid* to *valid* for 6 627 pairs, raising the overall validity from *26.1%* to *37.2%* (22 296/60 000). The further analyses in this section are computed on the basis of this corrected *valid* distribution.

Structural complexity and operator profile Validity declines with increasing path depth and filter load, the most informative structural marker is *filter density* (filters per path) rather than depth itself. Path length and filter density are strongly collinear (Spearman $\rho \approx 0.96$–0.99). OR-clauses appear in about *34.6%* of valid paths, while branching is rare but non-negligible (around *2.6%*). The operator mix is dominated by textual comparisons (`equals`, `alphabetic_starts_with`, `alphabetic_ends_with`, `contains/..._not`); numeric operators occur only sparsely. Figure 6.1 summarizes these distributions on the valid subset.

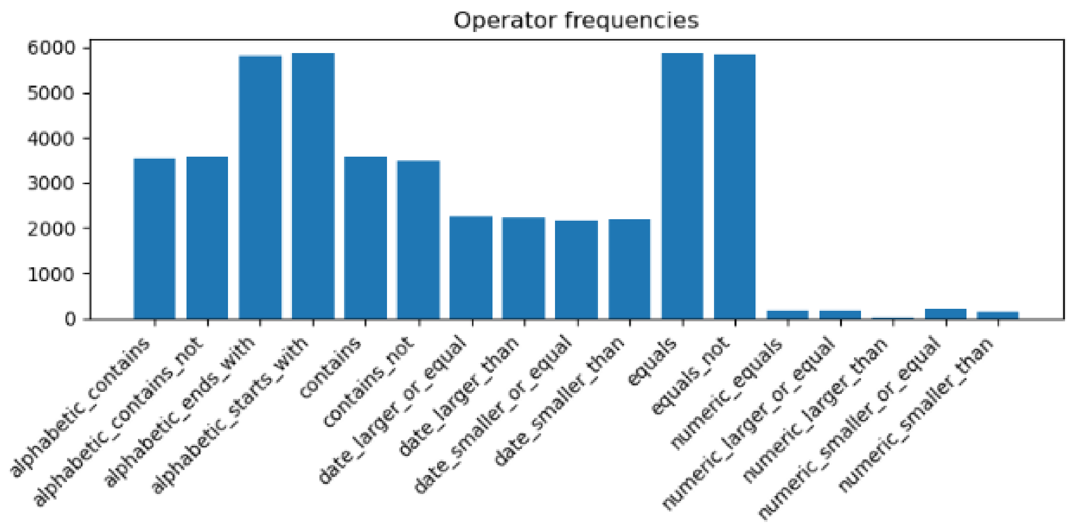

Figure 6.1: Operator frequencies across valid paths.

Text features as primary driver Among textual features, the *maximum length of the compared literals* is decisive. Across operators, the share of valid pairs declines sharply once literals exceed approximately 20 words ($\approx$ 64 tokens), and it approaches zero beyond approximately 80 words ($\approx$ 256 tokens). The logistic ablation series (Models A through D; Appendix A.6) confirms this pattern. After controlling for *filter density*, the apparent effect of `path_length` vanishes, which indicates mediation, whereas `text_len_words_max` remains the dominant predictor with an odds ratio of $\approx$ 1.056 per word (about 1.72$\times$ per additional 10 words). In Table 6.1, which reports odds ratios for the outcome *invalid* (OR $>$ 1 increases failure odds), `filters_per_path` is a strong structural risk factor (OR $\approx$ 3.01). `branched` adds independent risk (OR $\approx$ 2.82). `has_or` is protective once structural and textual complexity are held constant (OR $\approx$ 0.63). Holding density fixed, the total number of filters becomes mildly protective (OR $\approx$ 0.78). Operator-type indicators such as `equals` and `contains` are not significant once text length is included.

Term (Model D; invalidity as outcome)	OR	95% CI	Sig.
`text_len_words_max` (per word)	1.056	[1.052, 1.061]	***
`filters_per_path`	3.01	[2.11, 4.31]	***
`branched`	2.82	[1.81, 4.38]	***
`has_or`	0.63	[0.58, 0.68]	***
`num_filters`	0.78	[0.65, 0.94]	**
`path_length`	0.84	[0.64, 1.11]	n.s.

Table 6.1: Key effects in the full model (D). OR > 1 increases the odds of *invalid* pairs. Signif.: *** $p < 0.001$, ** $p < 0.01$, n.s. not significant.

Label reliability (heuristic) As detailed in Section 4.2.5, manual validation of representative sample ($n \approx 96$) yielded an estimated precision of 92.7% (95% CI: 85.7–96.4%), with no false positives and few false negatives. The valid split is therefore conservative and suitable as a basis for downstream analysis.

Annotation inconsistencies Targeted manual review surfaced a small subset of pairs in which the natural-language question and the associated Cypher-like query are each correct in isolation but originate from different source queries. These *misassigned* question–query pairs are treated as label noise and are excluded from all qualitative illustrations and from the counts underlying the post-hoc relabeling. An illustrative case is shown in Appendix Figure A.7: the ground-truth Cypher-like query traverses `WikiArticle` → `Corpus/WebResource`, whereas the question targets `TextNodes` with `HasNERAnnotation/HasAbbreviation`.

Schema coverage Coverage is reported at three granularities: collection types (nodes), relationship types (edges), and attributes on both sides. All values are computed on the *valid* split after post-hoc relabeling (22 296 pairs). An item is counted as covered if it appears at least once in a valid query, whether in a traversal, a filter, or a projection. Repeated occurrences do not increase the counts. Overall coverage is high across levels, and Table 6.2 reports the corresponding counts and percentages. A high-resolution bipartite *coverage graph* that visualizes which node and edge types and which attributes are touched by the valid paths is provided in Appendix Figure A.2.

Opening word	**Original**	**Rule-based**	**LLM**	**LLM (%)**
Which	22 296	22 296	16 922	75.9
What	0	0	4 611	20.7
Who	0	0	648	2.9
Other	0	0	115	0.5

Table 6.3: Distribution of question openings on the valid set. Original and rule-based forms start with *Which* in all cases; the LLM rewrite diversifies openings.

Level	**In schema**	**Used (valid)**	**Coverage**
Node types	30	27	90.0%
Edge types	25	20	80.0%
Node attributes	187	177	94.7%
Edge attributes	157	121	77.1%

Table 6.2: Schema coverage on the valid split. An entity/attribute is counted as covered if it occurs in at least one valid query (traversal, filter, or projection).

Schema compaction Schema length was reduced from 3 675 to 1 247 tokens (−2428; −66%), freeing substantial context budget for both the question and the corresponding AQL. On the evaluation set, questions average *55.3* tokens and AQL strings *259.8* tokens; together with the compact schema (1 247 tokens), typical samples remain well within the available context window.

Lexical diversity Manual inspection indicated that the original questions were authored in a schema-aware style and uniformly opened with *Which*. Rule-based naturalization preserved this "Which" monoculture, whereas the LLM rewrite broke the pattern and introduced alternative openings (Table 6.3), with *What*/*Who*/other accounting for *24.1%* of all questions. Beyond openings, the LLM simplification increased lexical diversity and shortened extreme length tails, as reflected by higher entropy and reduced maxima (Table 6.4), while keeping medians essentially unchanged. In essence, the LLM variant preserves constraints yet enriches the linguistic surface form.

Summary (RQ1) Overall, the validity-filtered corpus combines high schema coverage, a broad operator mix, a compact schema, and LLM-induced linguistic diversity, fitting comfortably within the context window and providing a controlled yet varied

Variant	**Uni.** H	**Bi.** H	**Min**	**Max**	**Mean**	**Median**
Original	7.799	10.679	6	333	27.59	21
Rule-based	7.812	10.625	5	327	26.03	20
LLM-simplified	8.459	11.859	4	199	24.95	21

Table 6.4: Tokenwise Shannon entropy (H) and length statistics (words) for original, rule-based, and LLM-simplified questions (valid set).

basis for subsequent model evaluation.

6.2 Model Evaluation

Focus: RQ2 (training adequacy).

Model selection `Qwen2-1.5B-Instruct` was selected as the base model due to its best overall compute–performance trade-off in the debug phase. Selection criteria included convergence speed, output quality, and training efficiency. It converged from a loss of 16 to 3 within one hour and achieved BLEU 0.25 and ROUGE 0.51. Based on this balance of quality and efficiency, it was chosen for full fine-tuning. Detailed screening results are provided in Appendix A.9.

Aggregate results Fine-tuning yields large, consistent gains across textual similarity, formatting discipline, and functional alignment. Table 6.5 reports the comparison on a subsample of the verification split. Notably, the AQL-like rate rises from 0.106 to 1.00 (+89.4 percentage points, pp), the rate that the closing tag is reached (EndedOnTag) 0.05 to 0.982 (+93.2 pp), while the truncation rate drops from 0.094 to 0.018 (−7.6 pp). A fixed-scale radar summary is shown in Figure 6.2.

Metric	**Base**	**Fine-tuned**
BLEU (mean)	0.003	**0.892**
ROUGE-L (F1, mean)	0.103	**0.945**
Levenshtein (mean; ↓)	648	**74**
AQL-like rate	0.106	**1.00**
EndedOnTag	0.05	**0.982**
Median generation length	54	**243**
Truncated-by-length rate	0.094	**0.018**
GoldOK coverage	0.847	
Execution accuracy (GoldOK only)	**0.861**	
ErrorMatch rate (same error as gold)	0.111	

Table 6.5: Base vs. fine-tuned on the verification split (n=1000). AQL-like=share of predictions that parse as AQL-like; closing tag=share that correctly emit `[/AQL]`; truncated-by-length=share that hit the generation cap. Execution accuracy is computed on items whose gold query executes successfully (*GoldOK*).

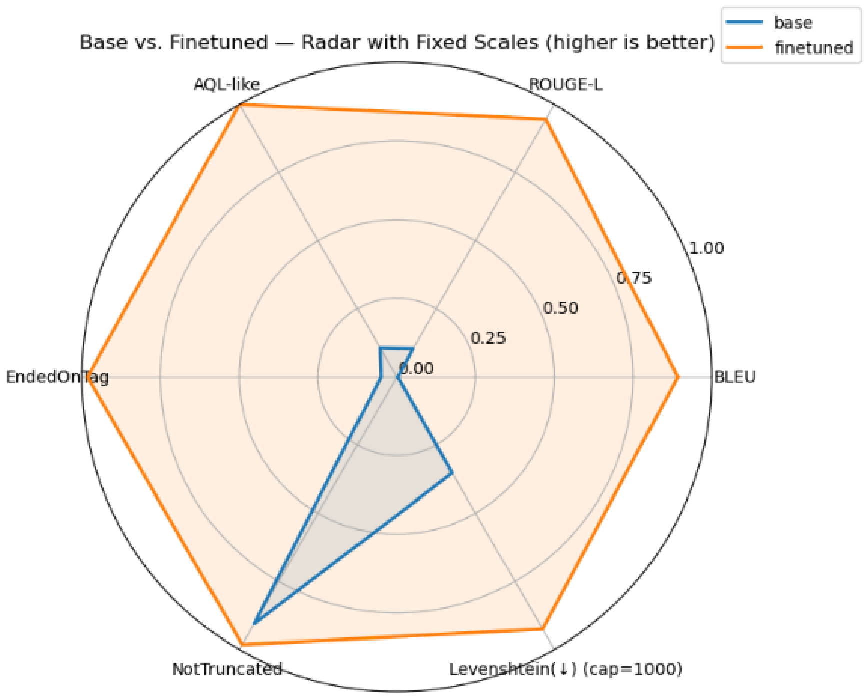

Figure 6.2: Fixed-scale radar: BLEU, ROUGE-L, AQL-like, EndedOnTag, NotTruncated (1−Truncated), and Levenshtein (mapped with fixed cap). Higher is better.

Error profile Residual failures can be assigned to five structural labels (WC, WJ, WF, EC, SE; definitions in Appendix A.10). On the verification split, *GoldOK* coverage is 0.847; conditional on *GoldOK*, execution accuracy is 0.861. Gold and prediction share the same error type in 11.1% of all cases (ErrorMatch= 0.111). Among prediction-side execution errors, the most frequent are RuntimeError (n=84), followed by CollectionMissing (*n*=52) and SyntaxError (*n*=14). Independently of runtime, the non-exclusive structural labels distribute as EC (*n*=248), SE (*n*=150), WF (*n*=132), WC (*n*=69), and WJ (*n*=3). Non-AQL outputs (IQ) are negligible after fine-tuning (cf. Table 6.5).

Generalization beyond the verification split On out-of-sample, manually written queries, predictions remain syntactically well-formed but show semantic deviations: start-collection shifts, edge-name/direction inconsistencies (e.g. `hasAlert` vs. `HasAlert`), truncated traversals, and over-strict equality where fuzzy containment is required. Representative side-by-side cases are shown in Appendix Figures A.4–A.5.

Robustness to low-signal input To assess the model's behavior in response to meaningless prompts, a syntactically well-formed yet semantically nonsensical string was provided (e.g., *"blurf tikka zondo . . . gloop . . . 'banana'"*). The fine-tuned model nevertheless generated a valid, tagged AQL (AQL-like = 1.00, EndedOnTag = 1.00) by reverting to a familiar traversal pattern (`TextNode` → `HasEmpathAnnotation`). The only semi-salient token (*gloop*) was preserved, while additional content included fabricated yet superficially plausible constants, for example `category = "fruit"` inferred from *banana*, as well as invented identifiers such as `WikiArticleRevision/2978`. Further illustrative cases are presented in Appendix Fig. A.6.

Summary (RQ2) Taken together, the fine-tuned model produces well-formed AQL and achieves high execution accuracy on the verification split. Residual errors are predominantly structural and arise chiefly on out-of-sample or low-signal inputs.

7 Discussion

This chapter interprets the evaluation with respect to the two research questions, situates the contribution within related NL2Query and property-graph research, and outlines methodological and practical implications.

Research questions

- **RQ1:** Is schema-guided path sampling with LLM verbalization a sufficient and faithful procedure to build a high-quality question-to-AQL corpus?
- **RQ2:** Is the resulting corpus adequate to train a functional question-to-AQL model that achieves high execution accuracy and remains stable under mild distribution shift?

7.1 Overview and Positioning

In contrast to template-only pipelines or unconstrained end-to-end generation, this study adopts a structure-first paradigm for property graphs and AQL: paths are sampled directly on the live database, serialized to a structured intermediate representation, compiled into executable AQL, and only then verbalized. A Cypher-like linearization stabilizes the verbalization stage. The graph structure serves as the primary source of truth, executability is prioritized, and traversal semantics (direction and depth) are kept explicit. This design reduces schema drift and decouples structural correctness from the linguistic surface.

Key empirical signals

- **Corpus quality.** 60 000 question–query pairs; after a conservative post-hoc correction for timestamp granularity, 22 296 pairs are labeled valid (37.2%). On the valid split, schema coverage is high: 90.0% of node types, 80.0% of edge types, 94.7% of node attributes, and 77.1% of edge attributes.
- **Primary drivers of validity.** Validity is chiefly shaped by maximum literal length and filter density. The share of valid pairs drops sharply beyond about 20 words ($\approx$ 64 tokens) and approaches zero beyond about 80 words ($\approx$ 256 tokens). Path depth contributes mainly through density (mediation).

- **Linguistic surface.** LLM-based simplification increases lexical diversity and breaks the prompt-induced "Which" monoculture, with non-"Which" openings rising to about one quarter.
- **Model performance.** After fine-tuning, text similarity and well-formedness improve markedly, with *AQL-like* = 1.00, *EndedOnTag* = 0.982, and execution accuracy = 0.861 at *GoldOK* coverage = 0.847.
- **OOD and low-signal behavior.** Syntactic robustness persists, while semantic deviations emerge as template-biased fallbacks on out-of-distribution ($n = 6$) and semantically nonsensical prompts.

Bridge to RQ1/RQ2 These signals frame the forthcoming answers by delimiting validity envelopes for corpus construction (RQ1), i.e., upper bounds on literal length and filter density beyond which heuristic validity drops sharply, and by establishing that the cleaned corpus can support a functional question-to-AQL translator with high execution accuracy, while delineating limits under distribution shift (RQ2).

7.2 RQ1: Is schema-guided path sampling with LLM verbalization sufficient and faithful for corpus construction?

Scope and criteria Assessment follows two criteria. *Sufficiency* is understood as broad coverage of the underlying schema. *Fidelity* is understood as preservation of the operator and value semantics encoded in the queries within the corresponding questions.

Corpus scale and coverage The pipeline yields 60 000 question-query pairs. After a narrowly scoped post-hoc correction for timestamp granularity, 22 296 pairs remain labeled as valid, which corresponds to 37.2%. All coverage values refer to this *valid* split. At least once, 90.0% of node types, 80.0% of edge types, 94.7% of node attributes, and 77.1% of edge attributes are observed. Repeated occurrences do not increase the counts. Coverage is broad, structure driven, and aligned with executable paths in the live schema.

Determinants of validity Validity is shaped by a small set of controllable factors. The maximum length of textual literals has the strongest effect. The share of valid pairs drops sharply beyond about twenty words and approaches zero beyond roughly eighty words. Logistic ablations confirm a significant rise in the odds of invalidity with each additional word. Among structural features, filter density is the primary risk factor. Path depth contributes mainly through this density. Branching increases risk, whereas carefully introduced `OR` clauses are mildly protective. The operator profile also reflects the data layer. In the schema-less target environment, fields that look numeric (e.g., `cited_by_count`) are often stored as strings at import. Without fixed typing or reliable type checks, only text-based operators are effectively available, so numeric comparisons are rarely generated or verified. This shifts the learning signal toward text comparisons and explains the low prevalence of numeric operators.

Label reliability and timestamp handling Validity labels are sufficiently reliable. An audit sample of $n \approx 96$ cases estimates the heuristic's accuracy at 92.7% with a 95% interval of 85.7% to 96.4% and no false positives. The main challenges involve dates and times. The heuristic checks at the calendar-day level, that is, the presence and agreement of the date in both question and pattern. In practice, heterogeneous time-of-day displays and differing formats caused conservative rejections despite semantic agreement. Post-hoc adjustments were applied only when the calendar day matched and the operator wording was consistent. Differences in time-of-day display were ignored. A small number of *misassigned pairs* also occurred, meaning that question and structural pattern are coherent on their own but do not originate from the same query source. These cases were excluded from illustrations and quantitative analyses so that coverage and validity rates are not distorted.

Two-stage generation for verifiability The generation process is oriented toward verifiability and control. The first stage produces narrowly scoped, technically precise questions with canonical operator wording and explicit mention of all literal values. This enables the deterministic verifier to identify field-operator-value triples reliably and reduces paraphrase-induced misclassifications. A Cypher-like linearization anchors structure and language. Only pairs that pass this check proceed to the second stage, where naturalization or simplification diversifies the linguistic surface, varies openings, and removes technical markers without altering operator semantics or literal values. Empirically, this second stage yields measurable variety. The prompt-induced *Which* monoculture is broken and alternative openings rise to 24.1% (*What* 20.7%, *Who* 2.9%, *Other* 0.5%; see Table 6.3). Lexical diversity increases and extreme lengths shorten, while the median length remains essentially stable (see Table 6.4).

Conditional answer to RQ1 Under these conditions, the answer is affirmative. The procedure yields a high-coverage and label-reliable corpus. Quality is bounded by a validity envelope defined chiefly by literal length and filter density. *Misassigned* cases and granularity details must be controlled explicitly.

7.3 RQ2: Does the corpus support a functional question-to-AQL model?

Strong gains and output discipline On the verification split, the fine-tuned model shows large improvements over the base. BLEU rises from 0.003 to 0.892, ROUGE-L from 0.103 to 0.945, and mean Levenshtein distance drops from 648 to 74. Output form is near perfect: *AQL-like* = 1.00, *EndedOnTag* = 0.982, and the truncation rate falls to 0.018. Note that BLEU, ROUGE-L, and Levenshtein weight surface form and order, and therefore penalize paraphrases or differently ordered yet semantically equivalent structures.

Execution level and practical meaning *GoldOK* quantifies the portion of items whose gold query executes on the target database, here 0.847. Conditional on this set, *Execution Accuracy* reaches 0.861, meaning the model reproduces the gold result in 86.1% of evaluable cases. The two figures are similar in magnitude but measure different aspects: *GoldOK* reflects dataset and environment executability, while *Execution Accuracy* isolates translation quality on the executable subset. This separation prevents database or gold-side failures from being attributed to the model. Unconditional end-to-end success on the verify split—treating non-GoldOK items as failures—thus equals $0.847 \times 0.861 = 0.729$ ($\approx 72.9\%$; $n \approx 729/1000$), which is a conservative lower bound.

Transparent error profile Non-AQL outputs are negligible after fine-tuning. Residual errors concentrate in structural-semantic categories: EC (expression or condition count mismatch, $n = 248$), SE (structural or execution failure, $n = 150$), WF (wrong or missing fields, $n = 132$), less often WC (wrong collections, $n = 69$) and WJ (wrong join or direction, $n = 3$). Runtime classes are dominated by RuntimeError ($n = 84$), CollectionMissing ($n = 52$), and SyntaxError ($n = 14$). In 11.1% of all examples, gold and prediction share the same error category (*ErrorMatch*), which points to issues beyond the model, such as schema drift between data generation and evaluation, missing views or permissions, or varying resource limits.

Syntax versus semantics High form fidelity is a direct consequence of the training setup. Strictly tagged examples, a compact schema context, and fixed stop tokens stabilize bracketing, keywords, and tag boundaries. By contrast, semantic deviations occur when structural details drift from the target. This includes schema linking, start collection, edge naming, directionality, or operator choice. Semantics tends to fail when the signal favors text-based filters over numeric ones, when edge attributes are underrepresented, when no execution check is available at inference time, or when low-signal prompts trigger template-biased fallbacks.

A further limitation is that the *gold* query denotes one deterministic canonicalization of the structured representation rather than the full set of semantically valid AQL formulations. Because AQL admits multiple equivalent encodings (e.g., reversing an `INBOUND/OUTBOUND` traversal, reordering filters, or distributing joins), text-based similarity metrics systematically underestimate functional accuracy by penalizing harmless surface variants. Execution-based evaluation mitigates this bias by judging semantic equivalence at the level of result sets rather than surface form.

Stability under mild distribution shift On six hand-written out-of-distribution questions, outputs remain well formed. Deviations cluster around start-collection selection, edge naming and direction, and a tendency to use strict equality where fuzzy containment is intended. Under deliberately low-signal prompts, syntax stays stable while predictions revert to familiar traversal templates. This confirms that output discipline generalizes, whereas semantic precision remains bounded by the validity envelopes established in RQ1.

Conditional answer to RQ2 Within the RQ1 envelopes, the corpus is sufficient to train a functional question-to-AQL model. Execution quality is high, output discipline is effectively complete, and stability holds under mild shift. Limits appear in traversal and filtering semantics, in particular around schema linking and operator choice. These limits can be reduced by strengthening numeric and edge-attribute coverage, adding explicit schema-coherence checks, and, where feasible, incorporating execution guidance at inference time. While increasing base-model size alone need not resolve semantic drift, multi-database SFT with rotating schemas is expected to reduce template fallback and improve robustness to schema/phrasing shifts.

7.4 Unexpected Findings

This section reports three unexpected but instructive phenomena that informed subsequent design choices and mitigations.

Spurious mismatches with long textual literals At first sight, longer textual literals in filters seemed to degrade verbalization fidelity: the incidence of heuristic *invalid* cases rose with the maximum literal length. Subsequent analysis traced this primarily to the batch pipeline rather than to the verbalizer. Generation ran in parallel; completions were consumed in order of arrival via `as_completed(...)` but paired with index batches in order of submission. Under concurrency this implicit FIFO assumption fails. Longer or more complex inputs incur higher latency, return later, and were disproportionately matched to the wrong index batch. The correlation with literal length is therefore explained by out-of-order returns that induce pairing errors, not by a systematic failure to verbalize long literals. The result is label noise in the form of mispaired question–AQL items: the heuristic evaluates overlap against the *assigned* structure, so a semantically correct sentence can be marked *invalid* if the pairing is wrong. This bias primarily affects RQ1 (corpus construction quality); RQ2 (training adequacy) remains robust because fine-tuning relied on the filtered, cleaned valid split, rendering the reported execution accuracies conservative. Practical safeguards include enforcing submission–result order with `executor.map(...)`, echoing stable sample IDs in prompt and answer and verifying them before writing.

Prompt-induced *Which* monoculture The initial verbalization exhibited a strong preference for *Which* as the opening word. The dominant cause is in-context priming by the production prompt (Appendix A.1): both demonstrations begin with *Which*, and the instruction asks for "clear English questions" without actively encouraging alternatives. In-context models imitate demonstrated forms, so *Which* becomes a stylistic anchor independent of content. Rule-based naturalization tends to preserve the opening and thus conserves this style. While not a semantic error—operators and constraints are preserved—the uniformity reduces surface variance and may lower robustness to naturally varying user phrasing. The downstream LLM simplification breaks this pattern: non-*Which* openings rise to 24.1% (*What* 20.7%, *Who* 2.9%, *Other* 0.5%; Table 6.3), and lexical diversity increases while extreme lengths shorten with a stable median (Table 6.4). Mitigations include a balanced few-shot set with mixed openings (*What/Which/Who/When/How many* . . .), an explicit style-jitter instruction that encourages varied openings, and a light stochastic choice for the first token with otherwise deterministic decoding. These adjustments preserve first-stage verifiability while increasing linguistic variety earlier in the pipeline.

Syntactically valid AQL from nonsense inputs For semantically meaningless prompts (e.g., "blurf tikka zondo . . . gloop . . . 'banana'"), the fine-tuned model still emits well-formed, tagged AQL (*AQL-like* = 1.00; *EndedOnTag* = 1.00). It "rescues" a single semi-salient token ("gloop"), falls back to a frequent traversal template (`TextNode` → `HasEmpathAnnotation`), and inserts plausible but unwarranted constants such as `category = "fruit"` in the vicinity of "banana". This is not a syntax issue but an abstention/control issue: the instruction and strict output tagging condition the model to always produce a valid query, and there is no explicit "no mapping" option. With weak signal, the model selects a mode answer from its training support and fills gaps with associative constants. Formal well-formedness therefore does not imply semantic grounding. Practical safeguards include an explicit abstain pathway (e.g., emitting `[NOAQL]` when signal is insufficient or schema alignment fails), schema-bound decoding masks restricting collections, fields, and operator–value types to schema-consistent choices, grammar-/parser-guided decoding to bound the search space, simple post-generation semantic checks that trigger abstention under low overlap or unlikely combinations, and training diversification across multiple graph databases to reduce template bias. Scaling alone does not resolve the issue: larger base models may detect inconsistencies better, yet without an abstention mechanism they still tend to produce valid but semantically irrelevant queries.

7.5 Practical Implications

The findings yield concrete guidance for corpus construction, verification, training, and inference. The following principles balance verifiability, coverage, and linguistic naturalness while preserving execution fidelity.

- **Constrain textual literals.** Keep compared text spans short. Aim for a maximum of about twenty words and avoid literals beyond eighty words. Where long strings are unavoidable, prefer anchored substring conditions (prefix, suffix, contains) over full-string equality and select distinctive substrings.

- **Manage filter density across the path.** Budget the number of constraints per sample and distribute them over hops rather than concentrating them on a single node or edge. Allow `OR` variants in a controlled manner to add diversity without inflating structural complexity.

- **Handle time-of-day consistently in verification.** Canonicalize timestamps at ingestion and evaluation, compare at an explicitly chosen granularity, and treat time-of-day uniformly. Expose the comparison granularity as a parameter of the heuristic to avoid brittle outcomes.

- **Maintain compaction.** Keep a compact schema representation that preserves structural information while reducing token footprint, thereby safeguarding context for the question and AQL segments.

- **Diversify prompts to reduce style priming.** Mix opening words (*What*, *Which*, *Who*, *When*, *How many*) and vary operator phrasing in demonstrations. If verification requires determinism, sample only the first token and decode the remainder greedily to retain reproducibility.

- **Adopt a two-stage generation protocol.** First generate narrowly scoped, canonical questions that support reliable heuristic checking, then apply naturalization or simplification only after passing verification. Preserve stable identifiers and logs to keep provenance and decisions auditable.

- **Teach abstention.** Include explicit `[NOAQL]` examples for low-signal inputs and schema mismatches. Negative and borderline cases help the model learn to abstain rather than produce a plausible but irrelevant query.
- **Constrain decoding with schema-bound semantic masks.** At inference time, restrict collections, fields, and operator–value types to those that exist in the schema and are type compatible. Combine grammar-guided decoding for syntax with parser masks for semantics to bound the search space while preserving validity.
- **Prioritize gold cleanup and continuous integration.** Operate a *GoldOK*-first pipeline. Execute gold queries regularly, version schema snapshots, and monitor drift. Fix or remove items that do not execute to prevent attributing data or environment faults to model behavior.
- **Train across multiple graphs.** Use multiple property graphs to decouple values from structure, broaden operator and attribute exposure, reduce template bias, and improve robustness under mild distribution shift.

7.6 Limitations and Threats to Validity

Internal validity For RQ1, *misassigned pairs* reduce interpretability only to a limited extent, since analyses consider the *valid* split exclusively and identified misassignments were excluded in advance. The deterministic heuristic's label accuracy in the audit sample is high but not perfect ($\approx 92.7\%$). The narrowly scoped post-hoc timestamp corrections can introduce bias in edge cases.

Construct validity (measurement adequacy) BLEU, ROUGE-L and Levenshtein primarily reward token order and surface similarity and therefore capture semantic equivalence only imperfectly. *AQL-like* and *EndedOnTag* measure formal well-formedness rather than semantic coverage. Execution accuracy is computed only for *GoldOK* items in which the gold query executes; cases where the gold query fails are not scored. In addition, the *gold* query denotes a single deterministic canonicalization rather than the full set of semantically valid AQL formulations. Because AQL admits multiple equivalent encodings (e.g., reversed traversals, reordered filters, distributed joins), text-based similarity metrics systematically underestimate functional accuracy. Execution-based evaluation mitigates this bias by tolerating structural variants and rewarding semantic equivalence at the level of result sets.

External validity (generalizability) Findings are based on training and evaluation within a single live graph and a closely related schema family, so the degree of transfer to unrelated domains remains open. The out-of-distribution probe is small (six hand-written questions) and domain coverage is limited. A larger multi-domain corpus across several databases would strengthen external validity, yet suitable property-graph datasets are scarce and often unbalanced, for example IMDb with a dominant node type and GDELT with few types and edges.

Conclusion validity (statistical reliability) The verify subsample size of $n \approx 1000$ is dictated by the available compute budget, which implies greater sampling variability in the reported point estimates.

Compute cost and reproducibility The pipeline is deterministically reproducible through fixed seeds and versioned artifacts, yet it is resource intensive. Question generation requires long runtimes, and fine-tuning the selected 1.5B model took about 50 hours in the present setup. Hyperparameters were not fixed ad hoc; they were selected via several short screenings of 30–90 minutes on smaller datasets and models, followed by a full run on the entire corpus. Budget constraints limit full ablations and extensive sensitivity analyses.

Source-to-sink restriction Query skeletonization follows a source-to-sink paradigm: start and end points are structurally prescribed, while cycles, internal start points and paths that do not terminate in a sink are ignored. Traversal depth is bounded by the length of the source-to-sink chains. Together with directions aggregated per edge type and fixed sampling limits for start nodes and neighbors, this reduces path diversity and can miss valid routes in the database graph. The findings therefore primarily reflect sourde-to-sink traversals.

7.7 Synthesis

The guiding hypothesis holds in a conditional form. Schema-guided path sampling combined with LLM verbalization can produce a viable question-AQL corpus when broad schema coverage and constraint fidelity are explicitly enforced.

Answer to RQ1 The pipeline yields a broadly covering and conservatively validated corpus. Quality is bounded by a clear validity envelope that is controlled by limits on literal length and by the management of filter density, together with robust pairing integrity and consistent handling of dates and times. Under these conditions, corpus construction is sufficient and faithful.

Answer to RQ2 The corpus supports a functional question-to-AQL model. Textual similarity and well-formedness improve markedly after fine-tuning, and execution accuracy is high on *GoldOK* items. Residual errors concentrate on start collection, edge direction, and operator choice, indicating that remaining gaps are primarily semantic rather than syntactic.

Overall conclusion The approach constitutes a robust baseline rather than a finished system. With modest dataset curation-numeric typing where appropriate, pairing integrity checks, controlled complexity, and targeted training adjustments such as an abstention option, schema-aware decoding, and grammar guidance, it becomes a solid, benchmark-ready pipeline for NL2AQL on property graphs.

7.8 Significance

Text–to–query quality hinges less on model size than on the composition of a structure-first design, formal verification, and compact schema context. Methodologically, the two-stage pipeline—first a verifier-friendly rendering, then naturalization—turns validity into an explicit design constraint and cleanly separates structural correctness from style. Empirically, schema-guided sampling yields a corpus that supports high execution accuracy when schema coverage and constraint fidelity are enforced; it also explains operator skew via numeric-as-string typing and motivates *GoldOK* as a standard for execution-based reporting. Practically, the blueprint provides low-risk control measures: bounds on literal length, management of filter density, pairing-integrity checks, an abstention (`NOAQL`) option for low-signal inputs, and schema- or grammar-guided decoding. Combined with a compact schema representation, these measures enabled a 1.5B model with QLoRA to achieve strong gains at modest cost. Field-wide, the work positions property-graph NL2Query as a first-class target with explicit traversal semantics and encourages execution-centered, reproducible evaluation with versioned artifacts.

7.9 Open Questions and Future Work

Pairing integrity Future runs should enforce strict result ordering under parallelism (e.g., `executor.map(...)` rather than `as_completed(...)`) and adopt an ID echo: each input carries a sample ID that the model must reproduce in its output; only matching IDs are committed. This guards against misassignments even under load.

Abstention / NOAQL An explicit no-mapping option is required for low-signal or schema-mismatch inputs. A standardized token (e.g., `[NOAQL]`) or a brief abstention rationale prevents plausible yet irrelevant queries. Triggers include weak schema alignment, missing operator anchors, or unusual value patterns.

Schema- and grammar-guided decoding Constrain decoding to collections, fields, and operator types present in the schema and enforce type compatibility through semantic masks. Combine this with grammar- or parser-guided decoding to bound the search space syntactically. The hybrid control reduces both form errors and semantic drift.

OOD breadth and multi-database training To strengthen transfer, assemble a multi-domain corpus spanning several graph databases. This decorrelates values from structure, supports schema-invariant mappings, and enables more informative out-of-distribution tests.

Relaxing the source-to-sink paradigm Extend beyond source-to-sink chains to include cycles, internal start points, and paths without sink termination. Preserve edge-level directionality rather than global aggregation and enrich edge attributes. This increases path diversity and brings the query class closer to real-world use.

8 Conclusion

This chapter summarizes the core contributions of the thesis, highlights their methodological and practical relevance, and outlines directions for future research.

This work has examined whether schema-guided path sampling in combination with LLM-based verbalization is suitable for constructing a high-quality question–to–AQL corpus, and whether this corpus can serve as the foundation for a functional NL2AQL model.

The results show:

- With a *structure-first* approach and a verifiable two-stage generation process, it is possible to create a high-quality and broadly covering corpus. Key dimensions of schema coverage were achieved, although the complexity of literals and the density of filters impose clear boundaries.
- On this basis, parameter-efficient fine-tuning yields a model that generates robust and executable AQL queries. It achieves stable *form fidelity* and high *execution accuracy*, although residual errors in operator choice and direction semantics highlight the need for further safeguards.

The scientific contribution of this work lies in providing a methodological blueprint for NL2Query in the context of property graphs: structure and executability are consistently prioritized over linguistic surface, verification is an integral part of the design, and a compact schema ensures robustness under limited context. In practical terms, the work delivers clear guidelines for corpus and system design, such as limiting literal lengths, distributing filters, introducing mechanisms for controlled non-answers (via an explicit `[NOAQL]` abstention path), and schema-guided decoding.

Overall, this establishes a solid starting point: a reproducible, benchmark-ready approach for NL2AQL that can be further developed through multi-graph training, extended decoding controls, and well-defined abstention strategies. In this way, the work lays a foundation on which both research and applications can build.

A Appendix

A.1 Final Prompt for Question Generation

The complete system prompt used in production for the LLM-based question generation step (Section 4.2.4) is given below. The placeholder `{{CY}}` is replaced at runtime with the Cypher-like pattern(s) to be verbalized.

```
You are an expert technical writer. Transform Cypher-style patterns into
clear English questions.
Rules:
• Preserve every node, relationship, and filter.
• Wording: equals, is not, contains, does not contain, starts with, ends
with, smaller than,
  at least, before, on or after.
• Wrap literals in single quotes.
• Multi-value lists → parentheses with “or”.
• Output exactly one question per pattern, do not enumerate.
• Separate multiple questions by a blank line.

Example:
Input:
  (events { _rev alphabetic_ends_with X;...;Y, sourceurls
  alphabetic_contains_not URL1,
            tone alphabetic_ends_with A;B })
  -[event_locations { relation equals located_in }]-> (locations)
  (events { numarts alphabetic_starts_with 1, _key equals_not key42 })
  -[event_persons]-> (persons)

Output:
  Which events whose revision ends with 'X' or ends with 'Y', whose
  sourceurls do not contain
  'URL1', and whose tone ends with 'A' or ends with 'B' are linked by
  event_locations
  (relation = located_in) to locations?

Input:
  (events { _key alphabetic_starts_with event_20250330_16,
            date date_larger_or_equal 2025-03-30,
            numarts alphabetic_ends_with 1 })

Output:
  Which events have a key that starts with 'event_20250330_16', occurred
  on or after
  March 30 2025, and whose 'numarts' value ends with the digit '1'?

Now convert the following patterns:
{{CY}}
```

A.2 Environments and Versioning

Table A.1: Mapping of pipeline stages to environments and entry points.

Stage	Environment	Entry point
Sampling & filter augmentation	`mth-path`	`sampling_main.py`
LLM question generation	`mth-qgen`	`llm/question_writer.py`
Heuristic verification	`mth-qgen`	`heuristic_verifier_main.py`
Question simplification	`mth-qgen`	`simplifier_main.py`
Analysis notebooks	`mth-ana`	`notebooks/01-08`
Fine-tuning & inference	`mth-tune`	`finetuning_main.py`

Table A.2: Conda environments and pin files (roles: see Table A.1)

Env	Conda YAML	Pip pins
`mth-path`	`env/mth-path.yml`	`env/mth-path-pip.txt`
`mth-qgen`	`env/mth-qgen.yml`	`env/mth-qgen-pip.txt`
`mth-ana`	`env/mth-ana.yml`	`env/mth-ana-pip.txt`
`mth-tune`	`env/mth-tune.yml`	`env/mth-tune-pip.txt`

A.3 Code Availability

The complete implementation of the question-to-AQL pipeline, including pre-processing, generation, verification, and fine-tuning scripts, is archived on the accompanying storage medium provided with this thesis. All code is versioned and executable under the pinned environments described in Appendix A.2.

A.4 Key Versions (Thesis Snapshot)

Table A.3: Key library versions per environment (see Table A.2 for YAML/pin files).

Environment	Key versions
`mth-path`	Python 3.10.18
	`python-arango 8.2.0`
	`requests 2.32.4`
	`python-dotenv 1.1.1`
	`tabulate 0.9.0`
`mth-qgen`	Python 3.10.18
	`openai 1.91.0`
	`ollama 0.5.1`
	`spacy 3.8.6` (+ `en-core-web-md 3.8.0`)
	`nltk 3.9.1`
	`pandas 2.3.0`
	`scikit-learn 1.7.0`
	`tiktoken 0.9.0`
	`regex 2024.11.6`
`mth-ana`	Python 3.10.18
	`pandas 2.3.0`
	`numpy 2.2.6`
	`matplotlib 3.10.3`
	`seaborn 0.13.2`
	`scikit-learn 1.7.0`
	`statsmodels 0.14.4`
	`jupyterlab 4.4.3`
	`networkx 3.4.2`
	`tiktoken 0.9.0`
`mth-tune`	Python 3.10.18
	PyTorch `2.5.1` (CUDA `pytorch-cuda 12.1`)
	`transformers 4.48.0`
	`peft 0.14.0`
	`trl 0.13.0`
	`bitsandbytes 0.46.1`
	`accelerate 1.2.1`
	`datasets 3.2.0`
	`evaluate 0.4.3`
	`sentencepiece 0.2.0`
	`wandb 0.19.3`
	`scikit-learn 1.7.1`
	`numpy 2.2.6`
	`pandas 2.3.1`
	`tokenizers 0.21.1`

Exact lockfiles live in `env/*-explicit.txt` (Conda) and `env/*-pip.txt` (pip).

A.5 Prompt for Question Simplification

The complete prompt set used in the LLM-based question simplification step (Section 4.2.7) is given below. The placeholder **{}** in the system prompt is replaced at runtime with either `PROMPT_NEUTRAL` or `PROMPT_AVOID_WHICH` and **{question}** in the user prompt, replaced at runtime with the actual question to be simplified.

System Prompt

```
You will receive a single natural language query at a time. This query
may contain multiple related subquestions that currently include
database-specific terminology, property names, and technical details.

Your task is to rewrite the entire input as a single, logically
equivalent inquiry that:

- Uses only natural, everyday language understandable by someone without
any database or technical knowledge.
- Avoids all references to database-specific names, collections,
properties, or internal IDs.
- Elegantly integrates all subquestions into one coherent, fluid
question, as a typical person might naturally ask it, using appropriate
question words like what, who, how, {}
- Preserves all specific values, filters, and conditions (such as dates,
keywords, numerical values, etc.) exactly as given.
- Important: Treat multiple subquestions as connected by AND, meaning
**all** conditions must be true in the combined inquiry.
- When multiple subquestions reference the same entities or concepts,
integrate their conditions so that the final question clearly reflects
that **all relevant conditions apply collectively** to those entities.

- Carefully consider the types of attributes referenced in the original
query, ensuring that the rewritten question accurately reflects the
intended meaning of attribute-based conditions (for example,
distinguishing when a condition specifies an attribute's value versus
when it references the presence of a keyword or text).
- Find appropriate, colloquial synonyms for database entities and
properties (except the values), which a "normal person" would use.

Do not include explanations or references to the original technical
query. Only return the newly created, natural-sounding question.
```

Two variants were used for the **{}** placeholder:

```
PROMPT_NEUTRAL: or which, depending on the context, when seeking the
same information.
PROMPT_AVOID_WHICH: but avoiding which to encourage more conversational
phrasing.
```

User Prompt

~~~
This is the query (enclosed in triple backticks):

```
{question}
```
~~~

A.6 Logistic Ablations and Interactions

Model	Pseudo-R^2	AIC	BIC
A	0.066	73986.7	74031.7
B	0.066	73927.9	73990.9
C	0.068	73789.0	73924.0
D	**0.117**	**69950.5**	**70094.5**

Table A.4: Model quality across the A→D ladder.

Note. Higher pseudo-R^2 and lower AIC/BIC indicate better fit; Model D provides the strongest overall fit

Term (OR)	A	B	C	D
`path_length`	2.14	0.75	0.75	0.84
`filters_per_path`	—	3.83	3.90	3.01
`text_len_words_max`	—	—	—	1.056

Table A.5: Selected odds ratios across models (OR > 1 increases invalidity).

Note. Odds ratios >1 increase the odds of an *invalid* pair. The apparent path-length effect in A vanishes once filter density and text length enter (D), while `text_len_words_max` is per *word*.

Interaction term	OR	p
`path_length` × `num_filters`	0.85	$< 10^{-15}$
`num_filters` × (`text_len_words_max`/10)	0.97	0.040
`text_len_words_max`/10	1.79	$< 10^{-25}$

Table A.6: Interaction diagnostics (compact). Effects as odds ratios.

Note. OT < 1 denotes attenuation: the impact of additional filters shrinks at longer paths. The text-length effect is reported per +10 words.

A.7 Misassigned Question–Query Pair (Illustrative Example)

Cypher (ground truth, excerpt)

```
( WikiArticle {
_key alphabetic_ends_with 227015837
}) - [ BelongsTo ] → ( Corpus
{ _id alphabetic_starts_with
Corpus/ClimateAnalyticsCorpusNode,
    timestamp date_larger_than
2023-06-02T22:08:21.425268 })
( WikiArticleRevision {
_rev contains _hfzPh7e-_a })
- [ BelongsTo ] → ( WikiArticle {
_key alphabetic_ends_with 227015837
})
( WikiArticleRevision {
_rev contains _hfzPh7e-_a })
- [ RefersTo ] → ( WebResource )
```

Corresponding Question

Which TextNodes that contain the text *‘Source : http : //www.dryiceinfo.com/science.htm .’* are connected by HasAbbreviation relationships to AbbrvNodes whose abbreviation starts with *‘SBT’*?

Which TextNodes that contain the same text are connected by HasNERAnnotation relationships to EntityNodes labeled ‘ORG’ ?

Which TextNodes whose key is not *‘41989’* are connected by HasNERAnnotation relationships to EntityNodes labeled ‘ORG’ ?

Figure A.1: Misassigned question–query pair: the Cypher ground truth (left) traverses WikiArticle → Corpus / WebResource via BelongsTo / RefersTo , while the question (right) actually belongs to a different query and targets TextNodes with HasNERAnnotation / HasAbbreviation . Green marks ground-truth entities/edges; Red marks elements of the question that do not correspond to that ground truth.

A.8 Coverage Graph

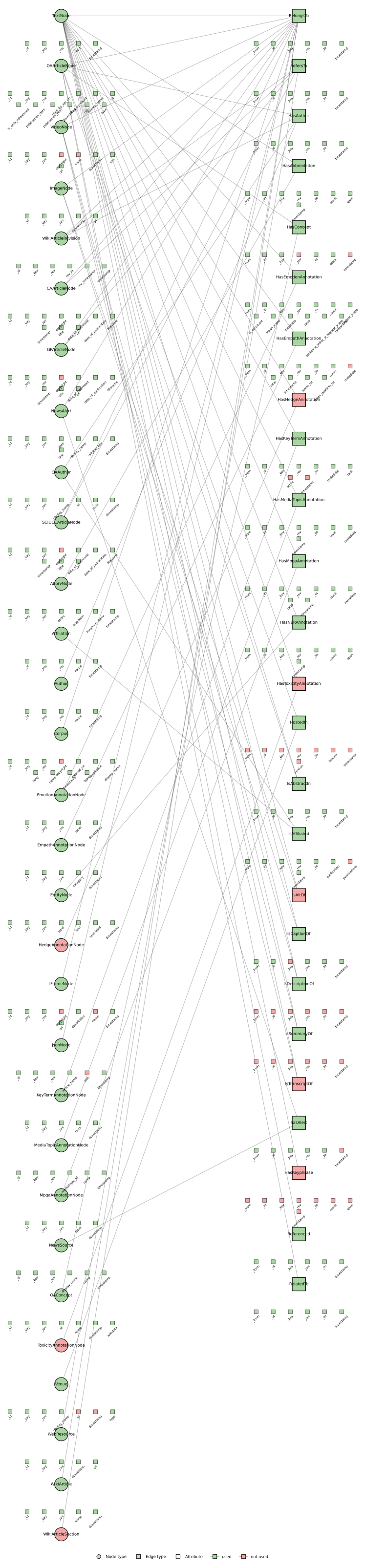

Figure A.2: Coverage graph showing which node/edge types and attributes are touched. Due to its size, the figure is scaled down for space, but serves to illustrate the overall breadth and distribution of schema coverage.

A.9 Model Screening Results (Debug Phase)

Five instruction-tuned LLMs were fine-tuned for 3 epochs on a reduced corpus (500 training, 100 evaluation examples) to evaluate convergence behavior, runtime efficiency, and output quality. Default settings were used per model family. Table A.7 reports BLEU, ROUGE, Levenshtein distance, loss progression, and approximate runtime. `Qwen2-1.5B-Instruct` was selected based on its favorable quality–efficiency trade-off.

Model	**BLEU**	**ROUGE**	**Levenshtein**	**Loss (start–end)**	**Time**
`Qwen2-1.5B-Instruct`	0.25	0.51	584.33	$16 \rightarrow 3$	~1h
`Phi-2`	0.0018	0.0415	1040.67	$20 \rightarrow 15$	~1h
`Falcon-7B-Instruct`	0.10	0.24	722.00	$1.27 \rightarrow 0.57$	~3h
`Mistral-7B-Instruct`	0.79	0.82	309.33	$7 \rightarrow 1$	~4h
`CodeLlama-7B-Instruct`	0.67	0.82	253.67	$12 \rightarrow 3$	~4.5h

Table A.7: Metrics from debug-phase fine-tuning. BLEU/ROUGE measure lexical similarity to gold questions; Levenshtein reports raw edit distance; loss reflects training convergence. All runtimes are approximate wall-clock durations.

A.10 Error Taxonomy and Distribution

Label	Name	Definition (prediction vs. gold)
IQ	Invalid Query	Output does not parse as AQL-like.
WC	Wrong Collections	Different set of collections used.
WJ	Wrong Join/Direction	Mismatch in traversal direction or join pattern.
WF	Wrong/Missing Fields	Missing or extraneous attribute references.
EC	Expr. Count Mismatch	Different number of filter expressions/conditions.
SE	Structural/Execution Failure	Execution fails (syntax/runtime/collection missing) not covered above.

Table A.8: Structural error labels used in the analysis. Labels are non-exclusive. Total samples in verify split: 1000.

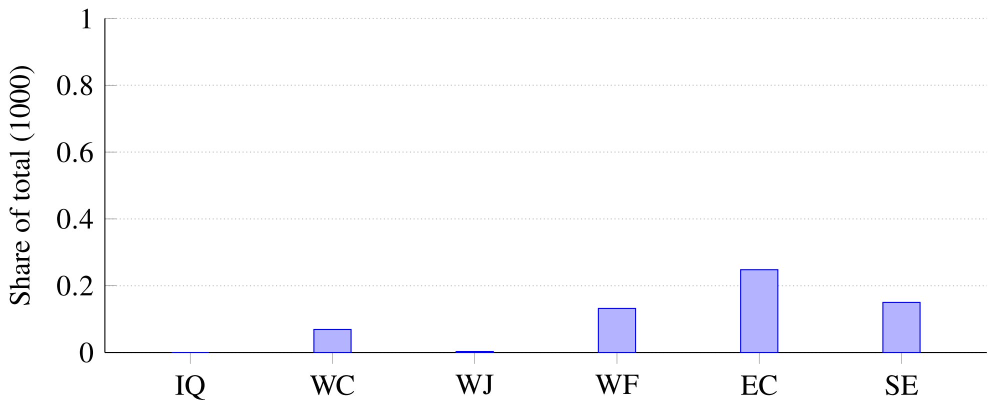

Figure A.3: Relative frequency of structural error labels on the verification split (1000 samples).

Label	Count	Share of total
IQ	0	0.000
WC	69	0.069
WJ	3	0.003
WF	132	0.132
EC	248	0.248
SE	150	0.150

Table A.9: Absolute counts and relative frequencies of structural error labels on the verification split (1000 samples).

A.11 Beyond Verify - Gold vs. Prediction

```
Question
Find all news alerts published by the news source ‘elib‘ of the DLR.
```

```
Gold AQL (excerpt)
LET doc_0 = FLATTEN(
  FOR doc IN NewsAlert
    LIMIT 10
  RETURN doc)
LET start_nodes = INTERLEAVE(doc_0, [])
FOR start_node IN start_nodes
  FOR v, e, p IN 0..1 ANY start_node INBOUND hasAlert
    OPTIONS { vertexCollections: [ "NewsSource" , "NewsAlert" ], edgeCollections:
[ "hasAlert" ] }
    FILTER (
      (IS_SAME_COLLECTION(p.vertices[0], "NewsAlert" ))
      AND (IS_SAME_COLLECTION(p.vertices[1], "NewsSource" ))
      AND (CONTAINS(LOWER(p.vertices[1].name), LOWER( "elib" )))
      AND (CONTAINS(LOWER(p.vertices[1].name), LOWER( "dlr" ))))
LIMIT 10
RETURN p
```

```
Prediction (model)
LET doc_0 = FLATTEN(
  FOR doc IN NewsSource
    FILTER LOWER(doc. display_name ) == LOWER( "elib" )
    LIMIT 10
  RETURN doc)
LET start_nodes = INTERLEAVE(doc_0, [])
FOR start_node IN start_nodes
  FOR v, e, p IN 0..1 ANY start_node OUTBOUND HasAlert
    OPTIONS { vertexCollections: [ "NewsSource" , "NewsAlert" ], edgeCollections:
[ "HasAlert" ] }
    FILTER (
      (IS_SAME_COLLECTION(p.vertices[0], "NewsSource" ))
      AND (LOWER(p.vertices[0]. display_name ) == LOWER( "elib" ))
      AND (IS_SAME_COLLECTION(p.vertices[1], "NewsAlert" ))
      % missing filter for "dlr" )
LIMIT 10
RETURN p
```

Notes: Start collection and direction swapped; field selection `display_name` vs. `name`; missing *dlr* filter. Labels: **WC**, **WJ**, **WF**, **EC**. Metrics (this example): BLEU=0.665, ROUGE-L=0.798, Levenshtein=158.

Figure A.4: Gold vs. Prediction (Example 1: NewsAlert/NewsSource). Green = Ground-truth elements; Red = deviations in the prediction; Gray = literals/other values.

```
Question
Identify the Wikipedia articles about ‘transportation‘ and find the authors that created
revisions for those articles during or after the year 2022.
```

```
Gold AQL (excerpt)
LET doc_0 = FLATTEN(
  FOR doc IN WikiArticle
    FILTER CONTAINS(LOWER(doc.name), LOWER( "transportation" ))
    LIMIT 10
  RETURN doc)
LET start_nodes = INTERLEAVE(doc_0, [])
FOR start_node IN start_nodes
  FOR v, e, p IN 0..2 ANY start_node INBOUND BelongsTo , OUTBOUND HasAuthor
    OPTIONS { vertexCollections: [ "WikiArticleRevision" , "Author" , "WikiArticle" ],
edgeCollections: [ "BelongsTo" , "HasAuthor" ] }
    FILTER (
      (IS_SAME_COLLECTION(p.vertices[0], "WikiArticle" ))
      AND (CONTAINS(LOWER(p.vertices[0].name), LOWER( "transportation" )))
      AND (IS_SAME_COLLECTION(p.vertices[1], "WikiArticleRevision" ))
      AND (p.vertices[1]. rev_timestamp >= DATE_ISO8601( "2022-01-01T00:00:00.000Z" ))
      AND (IS_SAME_COLLECTION(p.vertices[2], "Author" )))
LIMIT 10
RETURN p
```

```
Prediction (model)
LET doc_0 = FLATTEN(
  FOR doc IN WikiArticleRevision
    FILTER doc. timestamp >= DATE_ISO8601( "2022-01-01" )
    LIMIT 10
  RETURN doc)
LET start_nodes = INTERLEAVE(doc_0, [])
FOR start_node IN start_nodes
  FOR v, e, p IN 0..1 ANY start_node OUTBOUND HasAuthor
    OPTIONS { vertexCollections: [ "WikiArticleRevision" , "Author" ], edgeCollections:
[ "HasAuthor" ] }
    FILTER (
      (IS_SAME_COLLECTION(p.vertices[0], "WikiArticleRevision" ))
      AND (p.vertices[0]. timestamp >= DATE_ISO8601( "2022-01-01" ))
      AND (IS_SAME_COLLECTION(p.vertices[1], "Author" ))
      AND (CONTAINS(LOWER(p.vertices[1].name),
      LOWER( "Transportation2001-07-24 19:35:01+00:00" ))))
    % missing INBOUND BelongsTo hop from WikiArticle
LIMIT 10
RETURN p
```

Notes: Incorrect start in `WikiArticleRevision`; missing `BelongsTo`-hop; field `timestamp` instead of `rev_timestamp`; corrupted literal in the name filter. Labels: **WC**, **WJ**, **WF**, **EC**. Metrics (this ecample): BLEU=0.491, ROUGE-L=0.710, Levenshtein=271.

Figure A.5: Gold vs. Prediction (Example 2: WikiArticle → Revision → Author). Green = Ground-truth; Red = deviations in the prediction; Gray = literals/other values.

A.12 Robustness to Nonsense Input Example

```
Question
blurf tikka zondo from weeble snorf > 300 and gloop matches "banana"
```

```
Gold AQL
(none - no meaningful mapping exists)
```

```
Prediction (finetuned model)
LET doc_0 = FLATTEN(
  FOR doc IN TextNode
    FILTER CONTAINS(LOWER(doc.text), LOWER( "gloop" ))
    LIMIT 10
  RETURN doc)
LET start_nodes = INTERLEAVE(doc_0, [])
FOR start_node IN start_nodes
  FOR v, e, p IN 0..1 ANY start_node OUTBOUND HasEmpathAnnotation
    OPTIONS { vertexCollections:  [ "EmpathAnnotationNode" ,  "TextNode" ],
edgeCollections:  [ "HasEmpathAnnotation" ] }
    FILTER (
      (IS_SAME_COLLECTION(p.vertices[0],  "TextNode" )
         AND CONTAINS(LOWER(p.vertices[0].text), LOWER( "gloop" ))
         AND STARTS_WITH(REVERSE(p.edges[0]._from), REVERSE( "WikiArticleRevision/2978" ))
         AND IS_SAME_COLLECTION(p.vertices[1],  "EmpathAnnotationNode" )
         AND LOWER(p.vertices[1].category) == LOWER( "fruit" ))
     OR
      (IS_SAME_COLLECTION(p.vertices[0],  "TextNode" )
         AND CONTAINS(LOWER(p.vertices[0].text), LOWER( "gloop" ))
         AND IS_SAME_COLLECTION(p.vertices[1],  "EmpathAnnotationNode" )
         AND LOWER(p.vertices[1].category) == LOWER( "fruit" )))
LIMIT 10
RETURN p
```

Notes: The model preserves the token "gloop" from the input but injects irrelevant constants such as "fruit" and "WikiArticleRevision/2978" , reflecting template bias under low-semantic input. It still produces well-formed AQL and reaches the closing tag (AQL-like = 1.00; EndedOnTag = 1.00; GenLen = 313).

Figure A.6: Robustness to nonsense input: despite a semantically meaningless prompt, the finetuned model emits syntactically valid AQL and defaults to familiar traversal templates with spurious constants. No gold query is available for metric computation.

References

Agarwal, D., Das, R., Khosla, S., & Gangadharaiah, R. (2024, June). Bring your own KG: Self-supervised program synthesis for zero-shot KGQA. In K. Duh, H. Gomez, & S. Bethard (Eds.), *Findings of the association for computational linguistics: Naacl 2024* (pp. 896–919). Association for Computational Linguistics. https://doi.org/10.18653/v1/2024.findings-naacl.57

Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., & Vrgoč, D. (2017). Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, *50*(5). https://doi.org/10.1145/3104031

Angles, R., & Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys*, *40*(1), 1–39. https://doi.org/10.1145/1322432.1322433

ArangoDB GmbH. (2024a). ArangoDB Documentation – Graph Data Model [Accessed: 2025-07-05].

ArangoDB GmbH. (2024b). Arangodb documentation – version 3.11 [Accessed: 2025-07-06].

Bojchevski, A., Shchur, O., Zügner, D., & Günnemann, S. (2018). Netgan: Generating graphs via random walks [arXiv:1803.00816 [cs]]. *arXiv preprint arXiv:1803.00816*. http://arxiv.org/abs/1803.00816

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., . . . Amodei, D. (2020). Language models are few-shot learners. *Proceedings of the 34th International Conference on Neural Information Processing Systems*.

Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tai, Y., Fedus, W., Li, Y., Wang, X., Dehghani, M., Brahma, S., Webson, A., Gu, S. S., Dai, Z., Suzgun, M., Chen, X., Chowdhery, A., Castro-Ros, A., Pellat, M., Robinson, K., . . . Wei, J. (2024). Scaling instruction-finetuned language models. *J. Mach. Learn. Res.*, *25*(1).

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed). MIT Press.

Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). Qlora: Efficient finetuning of quantized llms. *Proceedings of the 37th International Conference on Neural Information Processing Systems*.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding [arXiv:1810.04805 [cs]]. *arXiv preprint arXiv:1810.04805*. https://doi.org/10.48550/arXiv.1810.04805

Fernandes, D., & Bernardino, J. (2018). Graph databases comparison: Allegrograph, arangodb, infinitegraph, neo4j, and orientdb. *Proceedings of the 7th International Conference on Data Science, Technology and Applications*, 373–380. https://doi.org/10.5220/0006910203730380

Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., & Taylor, A. (2018). Cypher: An evolving query language for property graphs. *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*, 1433–1445. https://doi.org/10.1145/3183713.3190657

Fürst, J., Kosten, C., Nooralahzadeh, F., Zhang, Y., & Stockinger, K. (2025). Evaluating the data model robustness of text-to-SQL systems based on real user queries. https://doi.org/10.48786/EDBT.2025.13

Gai, L., Chen, W., Xu, Z., Qiu, C., & Wang, T. (2014). Towards efficient path query on social network with hybrid rdf management. https://doi.org/10.1007/978-3-319-11116-2_48

Gjoka, M., Kurant, M., Butts, C. T., & Markopoulou, A. (2011). Practical recommendations on crawling online social networks. *IEEE Journal on Selected Areas in Communications*, *29*(9), 1872–1892. https://doi.org/10.1109/JSAC.2011.111011

Grover, A., & Leskovec, J. (2016). Node2vec: Scalable feature learning for networks. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 855–864. https://doi.org/10.1145/2939672.2939754

Guo, J., Zhan, Z., Gao, Y., Xiao, Y., Lou, J.-G., Liu, T., & Zhang, D. (2019). Towards complex text-to-SQL in cross-domain database with intermediate representation. *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. https://doi.org/10.18653/v1/p19-1444

Guo, Q., Zhang, C., Zhang, S., & Lu, J. (2024). Multi-model query languages: Taming the variety of big data. *Distributed and Parallel Databases*, *42*(1), 31–71. https://doi.org/10.1007/s10619-023-07433-1

Holtzman, A., Buys, J., Du, L., Forbes, M., & Choi, Y. (2020). The curious case of neural text degeneration. *International Conference on Learning Representations*. https://openreview.net/forum?id=rygGQyrFvH

Hu, E. J., yelong shen, Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2022). LoRA: Low-rank adaptation of large language models. *International Conference on Learning Representations*. https://openreview.net/forum?id=nZeVKeeFYf9

Kong, A., Zhao, S., Chen, H., Li, Q., Qin, Y., Sun, R., Zhou, X., Wang, E., & Dong, X. (2024, June). Better zero-shot reasoning with role-play prompting. In K. Duh, H. Gomez, & S. Bethard (Eds.), *Proceedings of the 2024 conference of the north american chapter of the association for computational linguistics: Human language technologies (volume 1: Long papers)* (pp. 4099–4113). Association for Computational Linguistics. https://doi.org/10.18653/v1/2024.naacl-long.228

Kudo, T., & Richardson, J. (2018, November). SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In E. Blanco & W. Lu (Eds.), *Proceedings of the 2018 conference on empirical methods in natural language processing: System demonstrations* (pp. 66–71). Association for Computational Linguistics. https://doi.org/10.18653/v1/D18-2012

Lee, C. Y. (1961). An algorithm for path connections and its applications. *IEEE Transactions on Electronic Computers*, *EC-10*(3), 346–365. https://doi.org/10.1109/TEC.1961.5219222

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. *Proceedings of the 34th International Conference on Neural Information Processing Systems*.

Li, C., Yang, X., Luo, S., Song, M., & Li, W. (2022). Towards domain-specific knowledge graph construction for flight control aided maintenance. *Applied Sciences*, *12*(24), 12736. https://doi.org/10.3390/app122412736

Li, Y., Dong, B., Guerin, F., & Lin, C. (2023, December). Compressing context to enhance inference efficiency of large language models. In H. Bouamor, J. Pino, & K. Bali (Eds.), *Proceedings of the 2023 conference on empirical methods in natural language processing* (pp. 6342–6353). Association for Computational Linguistics. https://doi.org/10.18653/v1/2023.emnlp-main.391

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2024). Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, *12*, 157–173. https://doi.org/10.1162/tacl_a_00638

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., & Neubig, G. (2023). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.*, *55*(9). https://doi.org/10.1145/3560815

Liu, Y., Han, T., Ma, S., Zhang, J., Yang, Y., Tian, J., He, H., Li, A., He, M., Liu, Z., Wu, Z., Zhao, L., Zhu, D., Li, X., Qiang, N., Shen, D., Liu, T., & Ge, B. (2023). Summary of ChatGPT-related research and perspective towards the future of large language models. *Meta-Radiology*, *1*(2), 100017. https://doi.org/10.1016/j.metrad.2023.100017

Lu, J., & Holubová, I. (2020). Multi-model databases: A new journey to handle the variety of data. *ACM Computing Surveys*, *52*(3), 1–38. https://doi.org/10.1145/3323214

Mao, W., Wang, R., Guo, J., Zeng, J., Gao, C., Han, P., & Liu, C. (2024, August). Enhancing text-to-SQL parsing through question rewriting and execution-guided refinement. In L.-W. Ku, A. Martins, & V. Srikumar (Eds.), *Findings of the association for computational linguistics: Acl 2024* (pp. 2009–2024). Association for Computational Linguistics. https://doi.org/10.18653/v1/2024.findings-acl.120

Mikolov, T., Chen, K., Corrado, G. S., & Dean, J. (2013). Efficient estimation of word representations in vector space. *International Conference on Learning Representations*. https://api.semanticscholar.org/CorpusID:5959482

Opitz, D., Hamm, A., Baff, R. E., Korte, J., & Hecking, T. (2024). Graph detective: A user interface for intuitive graph exploration through visualized queries. *Proceedings of the ACM Symposium on Document Engineering 2024*, 1–9. https://doi.org/10.1145/3685650.3685660

Opitz, D., & Hochgeschwender, N. (2022). From zero to hero: Generating training data for question-to-cypher models. *Proceedings of the 1st International Workshop on Natural Language-based Software Engineering*, 17–20. https://doi.org/10.1145/3528588.3528655

Paul, S., Mitra, A., & Koner, C. (2019). A review on graph database and its representation. *2019 International Conference on Recent Advances in Energy-efficient Computing and Communication (ICRAECC)*, 1–5. https://doi.org/10.1109/ICRAECC43874.2019.8995006

Pennington, J., Socher, R., & Manning, C. (2014). Glove: Global vectors for word representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1532–1543. https://doi.org/10.3115/v1/D14-1162

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018, June). Deep contextualized word representations. In M. Walker, H. Ji, & A. Stent (Eds.), *Proceedings of the 2018 conference of the north American chapter of the association for computational linguistics: Human language technologies, volume 1 (long papers)* (pp. 2227–2237). Association for Computational Linguistics. https://doi.org/10.18653/v1/N18-1202

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, *21*(1).

Robinson, I., Webber, J., & Eifrem, E. (2015). *Graph databases: New opportunities for connected data* (2nd). O'Reilly Media, Inc.

Rodrigues, C., Jain, M., & Khanchandani, A. (2023). Performance comparison of graph database and relational database. https://doi.org/10.13140/RG.2.2.27380.32641

Rodriguez, M. A., & Neubauer, P. (2010). The graph traversal pattern. *ArXiv*, *abs/1004.1001*. https://api.semanticscholar.org/CorpusID:7168330

Sasaki, B. M. (2016, March). Rdbms & graphs: Sql vs. cypher query languages [Accessed: 2025-07-05].

Schick, T., Dwivedi-Yu, J., Dessí, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. *Proceedings of the 37th International Conference on Neural Information Processing Systems*.

Sennrich, R., Haddow, B., & Birch, A. (2016, August). Neural machine translation of rare words with subword units. In K. Erk & N. A. Smith (Eds.), *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 1: Long papers)* (pp. 1715–1725). Association for Computational Linguistics. https://doi.org/10.18653/v1/P16-1162

Shen, J., Wan, C., Qiao, R., Zou, J., Xu, H., Shao, Y., Zhang, Y., Miao, W., & Pu, G. (2025). A study of in-context-learning-based text-to-sql errors. *ArXiv*, *abs/2501.09310*. https://api.semanticscholar.org/CorpusID:275570824

Sun, S., Chen, Y., He, B., & Hooi, B. (2021). Pathenum: Towards real-time hop-constrained s-t path enumeration. *Proceedings of the 2021 International Conference on Management of Data*, 1758–1770. https://doi.org/10.1145/3448016.3457290

Tang, X., Zhou, J., Shi, Y., Liu, X., & Lin, K. (2023). Efficient processing of k-hop reachability queries on directed graphs. *Applied Sciences*, *13*(6), 3470. https://doi.org/10.3390/app13063470

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, *1*(2), 146–160. https://doi.org/10.1137/0201010

Thapa, S. (2022, November). *Survey on self-supervised multimodal representation learning and foundation models*. https://doi.org/10.48550/arXiv.2211.15837

Trivedi, R., Dai, H., Wang, Y., & Song, L. (2017). Know-evolve: Deep temporal reasoning for dynamic knowledge graphs. *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, 3462–3471.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 6000–6010.

Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., & Wilkins, D. (2010). A comparison of a graph database and a relational database: A data provenance perspective. *Proceedings of the 48th Annual Southeast Regional Conference*, 1–6. https://doi.org/10.1145/1900008.1900067

Wang, B., Shin, R., Liu, X., Polozov, O., & Richardson, M. (2019). Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *Annual Meeting of the Association for Computational Linguistics*. https://api.semanticscholar.org/CorpusID:207863446

Wang, C., Tatwawadi, K., Brockschmidt, M., Huang, P.-S., Mao, Y., Polozov, O., & Singh, R. (2018). Robust text-to-SQL generation with execution-guided decoding [arXiv:1807.03100 [cs]]. *arXiv preprint arXiv:1807.03100*. https://doi.org/10.48550/arXiv.1807.03100

Wang, X., Wei, J., Schuurmans, D., Le, Q. V., Chi, E. H., Narang, S., Chowdhery, A., & Zhou, D. (2023). Self-consistency improves chain of thought reasoning in language models. *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=1PL1NIMMrw

Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., & Le, Q. V. (2022). Finetuned language models are zero-shot learners. *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. https://openreview.net/forum?id=gEZrGCozdqR

Xu, Y., & Goodacre, R. (2018). On splitting training and validation set: A comparative study of cross-validation, bootstrap and systematic sampling for estimating the generalization performance of supervised learning. *Journal of Analysis and Testing*, *2*(3), 249–262. https://doi.org/10.1007/s41664-018-0068-2

Yang, J., Jin, H., Tang, R., Han, X., Feng, Q., Jiang, H., Zhong, S., Yin, B., & Hu, X. (2024). Harnessing the power of llms in practice: A survey on chatgpt and beyond. *ACM Trans. Knowl. Discov. Data*, *18*(6). https://doi.org/10.1145/3649506

Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., & Radev, D. (2018, October). Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In E. Riloff, D. Chiang, J. Hockenmaier, & J. Tsujii (Eds.), *Proceedings of the 2018 conference on empirical methods in natural language processing* (pp. 3911–3921). Association for Computational Linguistics. https://doi.org/10.18653/v1/D18-1425

Yue, S., Xiao, L., Li, J., & Wang, N. (2022). Research on application of knowledge graph for aircraft maintenance. *Advances in Mechanical Engineering*, *14*(7), 16878132221107429. https://doi.org/10.1177/16878132221107429

Zhang, H., Song, H., Li, S., Zhou, M., & Song, D. (2023). A survey of controllable text generation using transformer-based pre-trained language models. *ACM Comput. Surv.*, *56*(3). https://doi.org/10.1145/3617680

Zhong, V., Xiong, C., & Socher, R. (2018). Seq2SQL: Generating structured queries from natural language using reinforcement learning. https://openreview.net/forum?id=Syx6bz-Ab