UAB
Universitat Autònoma de Barcelona

Dipòsit digital de documents de la UAB

This is the **published version** of the bachelor thesis:

Bustos Coll, Alex. *Interfície en Llenguatge Natural per a Bases de Dades de Grafs : Generació de Consultes i Respostes des de Grafs de Coneixement.* Treball de Final de Grau (Universitat Autònoma de Barcelona), 2026 (Grau en Intel·ligència Artificial)

This version is available at https://ddd.uab.cat/record/336305

under the terms of the  license.

Natural Language Interface for Graph Databases

Generating Queries and Answers from Knowledge Graphs

Alex Bustos

NIU: 1670204

Final Degree Project (TFG)

Faculty of Computer Science
Universitat Autònoma de Barcelona

June 2026

Abstract

Graph databases such as Neo4j provide powerful mechanisms for representing complex relationships between entities. However, interacting with these systems requires knowledge of Cypher, a specialized query language with limited accessibility for non-technical users. This project develops and evaluates a system that allows natural language interaction with graph databases through automatic generation of Cypher queries (Text2Cypher). The proposed architecture combines a Graph Autoencoder (GAE) with the Gemma-3 language model: the GAE encodes the structural information of the graph into a dense embedding, which is injected directly into Gemma's input sequence together with the user's question and a normalized schema description. Gemma is then fine-tuned with LoRA to learn to generate Cypher from this combined representation. The system is trained and evaluated on the public `neo4j/text2cypher-2025v1` dataset, using 8 Neo4j demo databases for training and a held-out database (Game of Thrones) for evaluation. Results and identified limitations are discussed in detail, together with a comparison against the Neo4j Text2Cypher baseline model.

Contents

1 Introduction and Motivation **2**
1.1 Objectives . . . 2

2 Related Work **2**

3 System Overview **2**
3.1 Approach 1: Graph Embedding + Language Model . . . 3
3.2 Approach 2: Iterative Subgraph Retrieval (Future Work) . . . 3

4 System Architecture **4**
4.1 Graph Autoencoder . . . 4
4.2 Graph-Conditioned Gemma with Cached Schema . . . 5

5 Training Setup **6**
5.1 Dataset . . . 6
5.2 Training Details . . . 6

6 Design Decisions and Challenges **6**
6.1 Node Feature Representation: From Semantic to Structural . . . 6
6.2 Schema Consistency . . . 7
6.3 Training Loss Alignment . . . 7
6.4 Memory Management . . . 7

7 Evaluation **7**
7.1 Metrics . . . 7
7.2 Baseline: Neo4j Text2Cypher 4B . . . 8
7.3 Results . . . 8
7.4 Qualitative Analysis . . . 8
7.5 Execution Accuracy . . . 10

8 Conclusions and Future Work **10**

1 Introduction and Motivation

During curricular internships at Accenture, I participated in the development of a dashboard connected to a graph-based database using Neo4j, allowing supply chain managers to detect potential shortages. The possibility of incorporating an AI agent that would let users interact with the database using natural language was proposed but never implemented, motivating this Final Degree Project.

Graph databases have gained significant popularity for representing complex relational data, but interacting with them still requires expertise in query languages such as Cypher. The goal of this project is to bridge this gap by developing a system that translates natural language questions into executable Cypher queries, conditioned on both the textual schema of the database and its structural graph representation.

1.1 Objectives

The main objective is to develop and evaluate a system capable of transforming natural language questions into executable Cypher queries over a Neo4j graph database. Specifically, the project aims to:

- Design a system allowing users to interact with a Neo4j database using natural language, without requiring knowledge of Cypher.
- Investigate different approaches for automatic generation of Cypher queries from natural language, focusing on the role of structural graph information.
- Implement a functional prototype capable of performing basic to intermediate queries over a graph database.
- Evaluate the advantages and limitations of the proposed approaches against a published baseline model.
- Analyse the potential integration within the Neo4j ecosystem.

Two complementary approaches are explored in this project:

1. **Approach 1 (main):** A Graph Autoencoder (GAE) combined with Gemma-3 generates Cypher directly from a graph embedding and the user question. This is the approach developed and evaluated in detail in this report.
2. **Approach 2 (future work):** An iterative pipeline that progressively narrows the relevant subgraph over multiple steps, generating partial Cypher clauses at each iteration until a convergence threshold is reached. This approach was designed but not implemented within the scope of this project.

2 Related Work

Several research efforts have focused on generating database queries from natural language, most notably for SQL. RAT-SQL [1] encodes relational schema structures using relation-aware attention mechanisms, highlighting the importance of structural context for query generation. In the knowledge graph domain, hybrid methods combining language models with graph reasoning have been proposed: GNN-RAG [4] extracts relevant graph structures with a GNN which are then used by an LLM; QA-GNN [2] constructs a joint graph representation including both question context and graph entities; PullNet [3] iteratively constructs a question-specific subgraph to reduce the reasoning space. These works share a common finding: providing explicit structural information from the underlying graph consistently improves query or answer generation compared to relying on language models alone, which directly motivates our approach.

For the Cypher-specific domain, the Neo4j `text2cypher-2025v1` dataset [5] provides 35,946 question–Cypher pairs with database schemas across 16 named Neo4j demo databases, enabling systematic evaluation.

3 System Overview

The implemented system follows the same underlying idea as RAT-SQL and GNN-RAG: structural information from the database is encoded explicitly

and provided to the language model, rather than relying solely on a textual description of the schema. In our case, this structural information takes the form of a graph embedding rather than relation-aware attention or a retrieved subgraph.

Two approaches were considered for this project. Approach 1 is fully implemented and evaluated. Approach 2 is proposed as a design and left for future work due to the constraints described below.

3.1 Approach 1: Graph Embedding + Language Model

The architecture has three main components, trained in separate phases and then combined into a single inference pipeline:

1. **Graph Autoencoder (GAE):** trained first and independently on a diverse set of heterogeneous graphs, so that it learns general-purpose graph representations rather than features tied to a single domain. Once trained, its encoder half is frozen and reused.

2. **Graph Embedding:** for any given question, the frozen encoder processes the corresponding Neo4j database graph and produces a single dense vector summarising its structural properties. This is precomputed once per database and cached for all subsequent queries.

3. **Gemma-3 with LoRA and cached schema:** the graph embedding and a normalised textual schema (also precomputed once per database) are injected into Gemma as additional context, prepended to the tokenised question. Gemma is then fine-tuned with LoRA to generate the corresponding Cypher query.

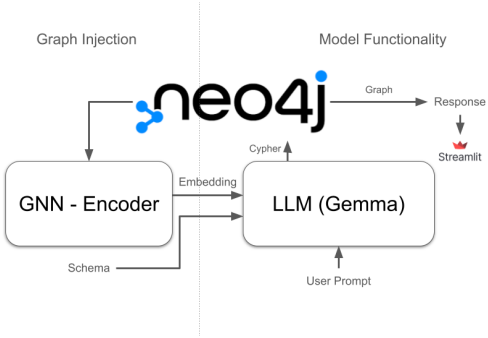


Figure 1: Approach 1: the graph embedding and schema embeddings are prepended to the user question before Gemma generates the Cypher query. Both the graph embedding and schema are precomputed once per database and cached.

The main contribution with respect to the Neo4j baseline is that the model receives structural information of the graph as an embedding, not only the schema as text. From Gemma's perspective, the graph embedding is just another vector in the 2560-dimensional embedding space, which allows the model to attend to graph structure information through its existing self-attention layers without any architectural changes to the transformer itself.

3.2 Approach 2: Iterative Subgraph Retrieval (Future Work)

The second approach proposes an iterative pipeline inspired by PullNet and GNN-RAG, where the query is not generated in a single pass but through a sequence of retrieval steps that progressively narrow the relevant subgraph until a convergence threshold is reached.

The core idea is as follows. Starting from the full database graph, the model generates a partial Cypher clause at each iteration that retrieves a subgraph relevant to the question. The retrieved subgraph is then fed back into the model as updated context for the next iteration, which generates the next clause and retrieves a smaller, more focused subgraph. This continues until either the model determines the answer is complete, or a maximum number of iterations is reached.

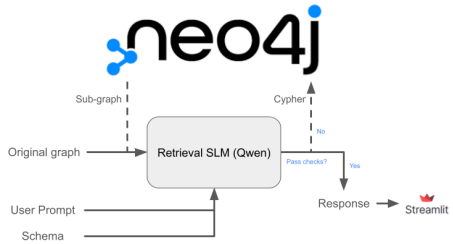


Figure 2: Approach 2: an iterative pipeline that progressively reduces the graph at each step, generating partial Cypher clauses until a convergence threshold is reached or a maximum number of iterations is exceeded.

Formally, at iteration t the model receives the original question q, the partial query generated so far $c_{1..t-1}$, and the subgraph $\mathcal{G}_t$ retrieved in the previous step. It produces the next clause c_t and a retrieval query that reduces $\mathcal{G}_t$ to a smaller subgraph $\mathcal{G}_{t+1}$. The process stops when $|\mathcal{G}_t| \leq \tau$ (a node count threshold τ) or $t = T_{\max}$, at which point the concatenated clauses $c_1 \cdot c_2 \cdots c_t$ form the final Cypher query.

The key advantage over Approach 1 is that the model never needs to compress the entire database into a single fixed vector: at each step it only attends to the current subgraph, which shrinks progressively. This makes the representation query-specific and avoids the information loss that occurs when pooling a large heterogeneous graph into 2560 dimensions.

This approach was not implemented within the scope of this project for two reasons. First, time constraints: the development and debugging of Approach 1 consumed the majority of the available project time. Second, the `text2cypher-2025v1` dataset provides only the final Cypher query, not the intermediate subgraphs or partial clauses that would serve as ground truth for each iteration step. Constructing this supervision signal automatically would require executing partial queries against all 16 databases at each iteration and extracting the traversed subgraphs, an infrastructure investment that was not feasible within the project timeline.

4 System Architecture

4.1 Graph Autoencoder

The autoencoder consists of an Encoder and a Decoder based on SAGEConv (GraphSAGE) [8]. SAGEConv is a graph convolution operator that aggregates neighbourhood information for each node by sampling and averaging features from its neighbours, rather than using the full adjacency matrix. This makes it scalable to graphs of arbitrary size, which is essential given the diversity of databases used (from 36 nodes in grandstack to 83,847 in network). The encoder stacks two SAGEConv layers to capture two-hop neighbourhood information, followed by a global max pooling operation that collapses the entire node feature matrix into a single fixed-size vector — the graph embedding.

The key intuition behind SAGEConv is that a node's representation is built by combining its own features with a summary of its immediate neighbours' features. After two layers, each node's representation captures not only its direct connections but also the connections of its neighbours — a two-hop receptive field. This is particularly useful for heterogeneous graph databases, where a node's role is largely determined by what types of entities it connects to: a `Movie` node connected to many `Person` nodes via `ACTED_IN` has a very different structural signature than one connected via `DIRECTED`. SAGEConv was chosen over alternatives such as GCN (which requires the full adjacency matrix and does not scale to large graphs) and GAT (which adds attention over neighbours but at higher computational cost) precisely because scalability and simplicity were priorities given the hardware constraints of this project.

Comp.	Architecture	Params
Encoder	Linear→SAGEConv×2 →global_max_pool	in=20, h=512, z=2560
Decoder	SAGEConv×2	z=2560, h=512, out=20

Table 1: Graph Autoencoder architecture.

The Encoder uses `global_max_pool` to collapse any graph into a single 2560-dimensional vector (matching Gemma's hidden dimension). Node features are fixed-dimension 20-vectors constructed as follows for each node v:

- **Dimension 0:** node type index — an integer identifying the label of node v (e.g. `Movie`=0, `Person`=1), mapped from the set of all node labels in the database.
- **Dimensions 1–k:** outgoing edge type counts — for each distinct (relation_type, destination_label) pair observed in the database, the number of edges of that type originating from v. These capture the local connectivity pattern of v with respect to each relation type.
- **Dimensions k+1–19:** destination node type counts — for each distinct destination node label, the total number of outgoing edges from v towards nodes of that label, regardless of relation type. These capture the neighbourhood type distribution of v.

The exact split between dimensions 1–k and k+1–19 depends on the vocabulary of the database; the total is always padded or truncated to 20 dimensions for consistency across databases.

Feature preprocessing. Raw degree counts vary by orders of magnitude across datasets (e.g. AMiner reaches values up to 31,045 versus 52 for IMDB). The following pipeline normalises this variability: (1) `log1p` to compress extreme degree counts; (2) Z-score per column to standardise to zero mean and unit variance; (3) Clamp to $[-3, 3]$ to remove residual outliers.

The autoencoder is trained on 7 heterogeneous graphs loaded from PyTorch Geometric [9] (AMiner, DBLP, OGB-MAG, IMDB, LastFM, ACM, Freebase) — a standard benchmark suite covering academic citation networks, movie databases, social networks, and knowledge graphs from different domains. These were chosen because they are publicly available, cover diverse graph structures (varying density, node types, and relation types), and are large enough to train a general-purpose encoder without overfitting to a single domain. Training uses MSE reconstruction loss, achieving convergence at ≈0.39 from epoch 5. Once trained, the encoder is frozen and reused for all subsequent training and inference.

4.2 Graph-Conditioned Gemma with Cached Schema

The combined model integrates the graph embedding module with Gemma-3-4B-it by replacing the original vision tower with the graph encoder. The key architectural decision is to separate the *fixed context* (graph embedding and schema) from the *variable input* (user question), and cache the fixed context once per database rather than recomputing it for every example.

Input construction. For each forward pass, the input sequence is constructed by concatenation in the embedding space:

$$\mathbf{E}_{\text{input}} = [\ \underbrace{\mathbf{z}_{\text{graph}}}_{\text{graph token}}\ ;\ \underbrace{\mathbf{E}_{\text{schema}}}_{\text{s_len tokens}}\ ;\ \underbrace{\mathbf{E}_{\text{question}}}_{\text{q_len tokens}}\ ]$$

where $\mathbf{z}_{\text{graph}} \in \mathbb{R}^{2560}$ is the precomputed graph embedding (projected from the GNN's output), $\mathbf{E}_{\text{schema}}$ are the token embeddings of the normalized schema text (precomputed once per database), and $\mathbf{E}_{\text{question}}$ are the token embeddings of the user's question (computed per example). The total prefix length is prefix_len $= 1 + s_{\text{len}}$.

Normalized schema generation. Rather than using the inconsistent schema text from the dataset (which had 2–3 format variants per database), the schema is generated directly from the real graph structure extracted from Neo4j, by querying the server for `MIN`/`MAX` statistics per property. This produces a single canonical schema per database in a uniform verbose format with property type, Min/-Max for numeric values, and an example for string values.

Training. Gemma's parameters are frozen except for LoRA adapters ($r = 4$) on `q_proj` and `v_proj` in all 34 decoder layers (136 trainable tensors). The loss is standard next-token CrossEntropy, shifted to compare logits at position prefix_len$+i$ against the label at position $i + 1$, masking all prompt tokens with -100.

Component	Status	Params
Graph Embedding (SAGEConv)	Frozen	∼2M
Projector (Linear+GELU+LN)	Frozen	∼5M
Gemma-3-4B base	Frozen	∼4B
LoRA (`q_proj`, `v_proj`, $r = 4$)	Trainable	∼few M

Table 2: Training configuration per component.

5 Training Setup

5.1 Dataset

The `neo4j/text2cypher-2025v1` dataset contains 35,946 question–Cypher pairs. Of these, 18,976 reference one of 16 named Neo4j demo databases. Only examples whose referenced database has a local graph available are used.

Database	Examples
recommendations	2,146
movies	1,942
northwind	1,668
grandstack	1,492
fincen	1,346
gameofthrones	1,296
buzzoverflow	1,106
network	1,093
Total	**12,089**

Table 3: Training databases. All 8 databases were used for training. Gameofthrones is used both for training and evaluation, as no fully independent test set was available within the project scope.

Databases with more than 100k nodes (neoflix, offshoreleaks) were excluded due to GPU memory constraints. The full training set covers 12,089 examples across 8 databases of diverse domains (movies, finance, social networks, infrastructure, knowledge graphs).

5.2 Training Details

The model is trained with gradient accumulation over 8 steps (effective batch size of 8 with `batch_size=1`), AdamW optimizer (lr $= 10^{-4}$), gradient clipping at 1.0, and 4-bit quantization (NF4) for the Gemma base. Schema embeddings are precomputed once per database (truncated to 400 tokens) and graph embeddings are precomputed once per database. The training dataset is cycled indefinitely with random shuffling.

The model was trained for over 5,500 steps, achieving a training loss of **0.087** at checkpoint step 5500, consistently decreasing from an initial loss of ∼21.6.

6 Design Decisions and Challenges

6.1 Node Feature Representation: From Semantic to Structural

An initial direction explored for node feature representation was a **semantic autoencoder**: rather than using fixed structural features, each node's properties would be concatenated into a natural language sentence (e.g. "Movie with title The Matrix, votes 5259, released 1999") and encoded with a frozen sentence transformer (`all-MiniLM-L6-v2`, 384 dimensions). A GRU-based generative decoder was added to reconstruct the original sentence from the reconstructed node embedding, providing a richer training signal than pure MSE reconstruction.

This approach was explored extensively but ultimately set aside in favour of the structural 20-dimensional feature representation. The main reasons were:

- **Dataset imbalance:** databases with many nodes (Recommendations: 28,866; Twitter: 43,325) dominated the vocabulary and caused the GRU decoder to collapse to generating templates from the largest database, regardless of the input node.
- **Heterogeneous property noise:** many nodes contained numerical embeddings stored

as graph properties (e.g. `fastrf_embedding`, `book1PageRank`) that produce very long, noisy sentences with little semantic value for Cypher generation.

- **Computational cost:** encoding all nodes of large databases with a sentence transformer before each training step was prohibitively slow on the available hardware.

The final structural features — a 20-dimensional vector encoding node type index, edge type counts, and destination node type counts per outgoing relation — proved simpler, faster, and sufficient for the graph encoder to learn discriminative database-level representations. The vector is normalised with `log1p`, Z-score, and clamping to $[-3, 3]$ to handle the wide range of degree counts across databases.

6.2 Schema Consistency

The `schema` field in the dataset had 2–3 format variants per database (verbose with bullets vs. compact inline), making caching by database alias incorrect. This was resolved by generating normalised schemas directly from the real Neo4j graph structure by querying the server for `MIN/MAX` statistics per property, producing exactly one canonical schema per database regardless of which dataset rows are sampled.

6.3 Training Loss Alignment

A subtle but critical issue was discovered in the loss computation: with a prefix of prefix_len positions (graph token + schema tokens) prepended to the input, the CrossEntropy shift was starting at position prefix_len-1 instead of prefix_len. This off-by-one caused the model to systematically learn to predict each answer token one position late, which meant the signal for the very first answer token (`MATCH`, present in 99.9% of examples) was never correctly learned. This was diagnosed by comparing teacher-forcing predictions against greedy generation and confirmed numerically before applying the fix.

6.4 Memory Management

Training a 4B model on a GPU with 8.6 GB VRAM (RTX 3070 Ti Laptop) required combining several techniques: 4-bit quantization (NF4), gradient checkpointing, `batch_size=1` with 8 accumulation steps, explicit deletion of batch tensors after each backward pass, and the environment variable `PYTORCH_CUDA_ALLOC_CONF` set to `expandable_segments:True` to reduce memory fragmentation that caused out-of-memory errors after extended training runs.

7 Evaluation

7.1 Metrics

Three metrics are used, consistent with the Neo4j baseline:

- **Exact Match (EM):** the generated query is identical to the expected one (case-insensitive). Strict: a single wrong character counts as a full miss.
- **Token Accuracy:** fraction of tokens correct at each position (aligned by prefix up to the shorter length). Captures partial correctness in cases where the query structure is right but details differ.
- **Structure Match:** the set of Cypher keywords (`MATCH`, `WHERE`, `RETURN`, `WITH`, `LIMIT`, etc.) in the generated query matches the expected set exactly. Measures whether the model understood the logical structure of the query.

Evaluation note. Both our model and the baseline were trained on the same `text2cypher-2025v1` split, as no separate public test set exists. Results may therefore be optimistically biased for both systems, particularly for the baseline which was fine-tuned on the full dataset. All 8 databases including gameofthrones were used for training, so the evaluation is performed on in-distribution data. The evaluation covers 100 examples sampled across all available databases with greedy decoding.

7.2 Baseline: Neo4j Text2Cypher 4B

The reference baseline used for comparison is `neo4j/text-to-cypher-Gemma-3-4B-Instruct-2025.04.0`, a Gemma-3-4B model fine-tuned by Neo4j directly on the full `text2cypher-2025v1` dataset. It represents the current state-of-the-art publicly available model for Cypher generation on this benchmark.

Unlike our system, the baseline receives the database schema as plain text embedded directly in the user prompt, following the standard instruct-tuning format:

```
Schema:
Node properties: ...
The relationships: ...

Question: Which movies have more than
5000 votes?
```

The model then generates the Cypher query autoregressively, without any explicit graph structure encoding. This is a simpler and more direct conditioning signal than our approach — the schema text is exactly the format the dataset was constructed with — which gives the baseline a structural advantage in terms of format alignment.

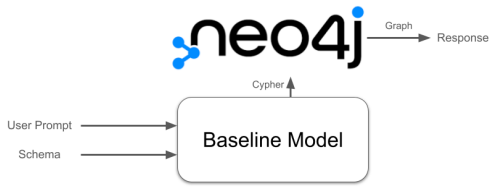


Figure 3: Architecture of the Neo4j baseline (Approach 0). The schema is provided as plain text in the prompt; no graph structure encoding is used. Greedy decoding is applied at inference time.

The key differences with respect to our system are summarised below:

- **Schema conditioning:** plain text in the prompt (baseline) vs. precomputed schema embeddings injected before the question (ours).
- **Graph structure:** not used (baseline) vs. graph embedding from a frozen SAGEConv encoder (ours).
- **Training data:** full dataset, 35,946 examples (baseline) vs. 12,089 examples from 8 databases (ours).
- **Fine-tuning:** full supervised fine-tuning (baseline) vs. LoRA $r = 4$ on `q_proj/v_proj` only (ours).

Both models use the same Gemma-3-4B base and are evaluated with greedy decoding on the same 100 examples.

7.3 Results

7.4 Qualitative Analysis

The cross-comparison between the two systems on the same 100 examples reveals consistent and interpretable failure patterns, rather than random errors.

Summary. Out of 100 examples: the baseline wins exclusively in 23 cases, our model wins exclusively in 6 cases, both are correct in 8 cases, and both fail in 63 cases.

Cases where our model wins (6/100). Our model consistently outperforms the baseline on queries where the expected answer returns the full node rather than specific properties. For example:

- *"List the top 5 orders with the highest freight cost"*
  Expected: `MATCH (o:Order) RETURN o ORDER BY o.freight DESC LIMIT 5`
  Ours: `MATCH (o:Order) RETURN o ORDER BY o.freight DESC LIMIT 5` ✓
  Baseline: `... RETURN o.orderID, o.freight ORDER BY toFloat(o.freight) DESC LIMIT 5` ×
- *"What is the total revenue from Adventure genre movies?"*
  Both queries are semantically equivalent but

System	Architecture	EM	Tok. Acc.	Struct.
Baseline (Neo4j 4B)	Fine-tuned LLM, schema in prompt	31.0%	65.0%	88.0%
Sys. B: GNN + Gemma	GAE + Gemma-3-4B + LoRA + cached schema	4.0%	47.3%	28.0%

Table 4: Evaluation results over 100 examples across 8 databases. Both systems evaluated with greedy decoding on the same examples. The baseline uses the full schema as text in the prompt; our system uses precomputed schema embeddings and a graph embedding token.

	Baseline (Neo4j 4B)		Sys. B (GNN+Gemma)	
Database	**EM**	**Struct.**	**EM**	**Struct.**
recommendations	5/20 (25%)	18/20 (90%)	1/20 (5%)	7/20 (35%)
movies	4/15 (27%)	15/15(100%)	0/15 (0%)	5/15 (33%)
northwind	4/13 (31%)	11/13 (85%)	1/13 (8%)	1/13 (8%)
grandstack	5/13 (38%)	13/13(100%)	1/13 (8%)	4/13 (31%)
gameofthrones	5/12 (42%)	10/12 (83%)	0/12 (0%)	3/12 (25%)
network	3/9 (33%)	8/9 (89%)	0/9 (0%)	4/9 (44%)
buzzoverflow	4/11 (36%)	8/11 (73%)	1/11 (9%)	3/11 (27%)
fincen	1/7 (14%)	5/7 (71%)	0/7 (0%)	1/7 (14%)
Total	**31 (31%)**	**88 (88%)**	**4 (4%)**	**28 (28%)**

Table 5: Per-database breakdown comparing both systems.

differ in traversal direction: our model generates `(m)-[:IN_GENRE]->(g)` matching the expected direction, while the baseline reverses it to `(g)<-[:IN_GENRE]-(m)`.

The pattern is clear: our model learned a more conservative return style (`RETURN node` rather than `RETURN node.prop1, node.prop2`), which matches the dataset's ground truth more often for simple retrieval queries.

Cases where the baseline wins (23/100). The baseline's failures of our model fall into four recurring categories:

1. **Alias naming (frequent):** Our model uses different but valid alias names. Example: expected `moviesDirected`, our model generates `numMovies`. The query is logically correct but fails EM due to naming. Similarly, expected `Character1`, our model generates `c1`.

2. **Incomplete generation (critical):** Our model truncates the query before completion. Example for "top 3 directors by budget":

   Ours: `... WITH d, count(m) AS numMovies ORDER BY numMovies DESC`
   Expected: `... LIMIT 3 RETURN d.name, moviesDirected`

   The model generates the correct structure but stops before `LIMIT` and `RETURN`, suggesting `max_new_tokens=50` is too short for complex multi-clause queries.

3. **Quote style (trivial):** Expected uses double quotes (`"France"`), our model generates single quotes (`'France'`). Both are valid Cypher but differ character by character under EM.

4. **Filter style:** Our model sometimes uses inline property filters (`MATCH (m:Movie {released: 1999})`) while the expected uses `WHERE` (`WHERE m.released = 1999`). Both are semantically equivalent.

These findings suggest that the gap is not primarily one of semantic understanding but of surface-level consistency: alias naming conventions, quote style, and filter placement account for a large fraction of failures.

7.5 Execution Accuracy

Beyond text-based metrics, the generated queries were executed against the real Neo4j demo server to measure their functional correctness. Three additional metrics were computed:

- **Executable:** the query runs without syntax or schema errors.
- **Non-empty:** the query executes and returns at least one result.
- **Result Match:** the query returns exactly the same results as the expected query, regardless of query text differences. This is fairer than EM: two syntactically different queries can be semantically equivalent (e.g. different traversal direction or alias names).

Metric	Ours	Baseline
Executable	75.0%	97.0%
Non-empty	71.0%	90.0%
Result Match	27.0%	42.0%
Exact Match (text)	4.0%	31.0%

Table 6: Execution accuracy on the real Neo4j demo server (100 examples). Result Match is fairer than EM: queries with different syntax but equivalent semantics are counted as correct.

Several observations stand out from these results. First, the gap between Result Match (27%) and EM (4%) for our model confirms that a significant fraction of failures under EM are due to surface differences rather than semantic errors: our model generates correct queries that happen to use different aliases, quote styles, or filter placement. The same gap is visible for the baseline (42% Result Match vs 31% EM), though smaller.

Second, the executability gap (75% vs 97%) reveals that our model sometimes generates syntactically invalid Cypher. Inspection of the 25 failing queries shows a consistent cause: **truncated generation**. The model generates a correct partial query but stops before completing it (e.g. ending mid-clause after `ORDER BY` without the column name, or after `LIMIT` without the number). This is directly caused by `max_new_tokens=50` being too short for multi-clause queries. Increasing this limit to 120–150 tokens is expected to recover most of the 25 execution failures.

Third, **network** achieves 100% executability for our model — every generated query is syntactically valid and runs correctly — which is consistent with it being the database where our model ties with the baseline on EM. The regular hierarchical schema (DataCenter → Rack → Machine → Type) produces consistent multi-hop traversal patterns that our model has learned reliably.

Fincen remains the hardest database: 42.9% executability and 0% Result Match for our model, reflecting the complexity of its financial schema with many interrelated entity types and less regular query patterns.

8 Conclusions and Future Work

This project developed a Text2Cypher system that integrates a Graph Autoencoder with a fine-tuned Gemma-3 model, conditioned on both a structural graph embedding and a precomputed schema. Several non-trivial engineering challenges were identified and resolved during development: a loss shift off-by-one bug that suppressed learning of the first answer token, schema inconsistency requiring normalized generation from the real graph structure, and memory management for training a 4B model on 8.6 GB VRAM.

The system was trained for over 5,500 steps achieving a training loss of 0.087, and evaluated on 100 examples across 8 databases against the Neo4j `text-to-cypher-Gemma-3-4B` baseline. The baseline achieves 31% EM, 65% Token Accuracy, and 88% Structure Match. Our system achieves 4% EM, 47.3% Token Accuracy, and 28% Structure Match. The Token Accuracy gap (17.7 percentage points) and especially the Structure Match gap (60 points) indicate that the main remaining challenge is structural correctness of queries, which requires further training with more diverse examples and the corrected loss.

Future work. The qualitative analysis identified concrete and actionable improvements:

- **Increase `max_new_tokens`:** 25 out of 100 generated queries fail to execute due to truncation — the model generates a correct partial query but stops before completing it (e.g. ending after `ORDER BY` without the sort column, or after `LIMIT` without the number). Increasing from 50 to 120–150 tokens is expected to recover most of these failures and push executability from 75% closer to the baseline's 97%.
- **Richer GNN node features:** the current 20-dimensional structural features encode only node type, edge type counts, and destination node type counts. A natural extension would be to incorporate database-specific features derived directly from the schema: property names as token embeddings, property types (STRING, INTEGER, FLOAT), or node degree distributions. This would allow the GNN to produce embeddings that carry semantic information about the database schema rather than purely structural topology, potentially improving performance on complex schemas like fincen where purely structural features proved insufficient.
- **Schema-free inference mode:** the current architecture always prepends the precomputed schema embeddings to the input, relying on them as the primary source of property and label names. An alternative would be to train a variant where the schema embedding is optionally excluded, forcing the model to rely more heavily on the graph embedding alone. This would test whether the graph encoder captures enough structural information to guide generation without explicit schema text, and would reduce the input sequence length substantially — which directly alleviates the memory pressure that limits training to `batch_size=1`.
- **Surface normalisation:** quote style (`"` vs `'`), alias naming, and filter placement (`WHERE` vs inline `{key: value}`) account for a large fraction of EM failures despite semantic correctness. Post-processing normalisation or training with augmented data covering both styles would help.
- **Extend to more databases:** adding neoflix, twitch, and other databases once connections are available would increase training diversity and coverage.
- **Implement Approach 2:** the iterative subgraph retrieval pipeline (Qwen-based) would complement Approach 1 for multi-hop reasoning queries where the full schema is harder to compress into a single embedding.
- **Proper train/test split:** once a public held-out benchmark becomes available for `text2cypher-2025v1`, re-evaluate to obtain unbiased metrics.

References

[1] Wang et al., *RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers*, ACL 2020.

[2] Yasunaga et al., *QA-GNN: Reasoning with Language Models and Knowledge Graphs*, NAACL 2021.

[3] Sun et al., *PullNet: Open Domain Question Answering with Iterative Retrieval*, EMNLP 2019.

[4] Luo et al., *GNN-RAG: Graph Neural Retrieval for Large Language Model Reasoning*, 2024.

[5] Neo4j, *text2cypher-2025v1 dataset*, https://huggingface.co/datasets/neo4j/text2cypher-2025v1.

[6] Google DeepMind, *Gemma 3 Technical Report*, 2025.

[7] Hu et al., *LoRA: Low-Rank Adaptation of Large Language Models*, ICLR 2022.

[8] Hamilton et al., *Inductive Representation Learning on Large Graphs*, NeurIPS 2017.

[9] Fey and Lenssen, *Fast Graph Representation Learning with PyTorch Geometric*, ICLR Workshop 2019.

[10] Mangrulkar et al., *PEFT: State-of-the-art Parameter-Efficient Fine-Tuning*, https://github.com/huggingface/peft, 2022.