Building information extraction via natural language query using multi-agent LLM system

Егор Сурков, Николай Михайлов, Егор Кошелев, Анна Никифорова

Introduction

Building Information Modeling (BIM) data in the IFC format often reaches massive scale and complex structure, making direct interrogation both slow and error-prone. Architects, engineers, and contractors must navigate deeply nested entities—walls, slabs, openings, and their interconnections—to extract even simple facts, which demands specialized expertise and significant processing time.

To bridge this gap, our goal is to enable users to pose plain-English questions against IFC repositories and obtain fast, accurate responses without needing to master the IFC specification or write low-level queries. By abstracting away the underlying complexity, we empower stakeholders to interact with BIM data as intuitively as they would with any modern information system.

We achieve this by transforming IFC models into lighter, query-friendly representations—relational (SQL), document-oriented (JSON), or graph (Memgraph)—and leveraging a large language model to translate natural-language questions into optimized Cypher or SQL queries. The resulting workflow delivers concise, actionable answers directly to end users, dramatically reducing both response time and the technical barrier to BIM data exploration.

Memgraph Optimization for IFC Graph Queries

The task: enable fast, accurate retrieval of building-model data stored as an IFC-derived property graph in Memgraph.

Must:

- Express property filters (e.g., *IsExternal* walls) with simple Cypher.
- Raise the success rate of LLM-generated queries.
- Eliminate run-time syntax errors.

Nice: keep data migration effort low; maintain backward compatibility where possible.

The idea: keep the graph but redesign how properties are reached—either by flattening them or by splitting query logic across two stages.

The solution:

- **Data-flattening ETL** – expand nested *Property Sets* into top-level node attributes (e.g., `Pset_WallCommon_IsExternal`).
- **Two-stage retrieval** (quick interim fix) –
    1. Cypher returns candidate nodes + raw JSON properties.

2. The LLM post-filters that list for the user's criterion.

- **Few-shot library** – curate a comprehensive NL↔Cypher dataset so the LLM learns the new, simpler schema.

Results (current baseline): graph imported; LLM integration works but struggles on three pain points (nested JSON access, precision, rare syntax glitches). Implementation of the flattening pipeline is expected to cut query latency and boost LLM accuracy significantly; quantitative re-evaluation is scheduled.

Limitations:

- Memgraph has no native JSON-path syntax.
- One-off ETL is needed to flatten millions of properties.
- LLM still needs guarded execution to catch residual syntax slips.

What didn't work:

- Direct Cypher like `n.Pset_WallCommon.IsExternal` (unsupported).
- Prompt-only tweaking without schema changes—accuracy plateaued.

SQL ToolManager

The task: given a query to an IFC file, return the correct answer to the query.
Must: correct answer, scalable architecture
Nice: explanation of how the answer was found
The idea:

1. Convert IFC file to a relational database (sqlite)
2. Use an LLM Agent to interact with the database via SQL and answer the user's question

The solution:

- Open-source SOTA LLM that can run locally — Qwen8B
- SQLToolManager — custom Python toolset that allows the LLM to query the database (sql_db_list_tables, sql_db_schema, sql_db_query_checker, sql_db_query)
- Specialized system prompt
- Try-except-rewrite logic

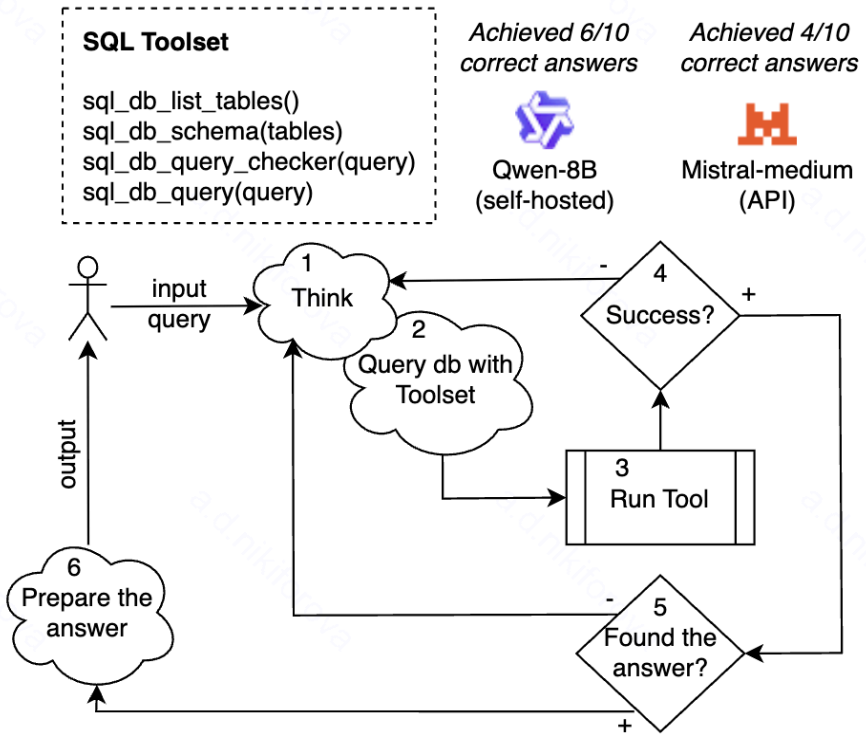
Results: 6/10 queries from the paper are answered correctly
Limitations:

- Memory — cannot pass the whole database description to the context
- Speed — 2-7 minutes per query (2 V100 GPUs)

What didn't work:

- LangChain & LangGraph — unnecessary abstractions led to numerous mistakes
- Summarization to reduce context length — the Summarizer would omit important details like queries and error tracebacks, not allowing the Agent to learn from its mistakes

- Validator agent — after questions from the Validator, the Agent would start doubting and endlessly repeat the same query
- Used Mistral in the same setting — the model performed poorly (4/10)



Gemini 2.5 Flash Text-to-SQL Pipeline

The task: convert a natural-language question into correct SQL and deliver a human-readable answer.

Must: accurate SQL; self-healing execution flow.
Nice: transparent explanation of how the answer was produced.

The idea: break the job into small, verifiable steps—prune schema, show real data, generate SQL, auto-fix, then validate semantics.

The solution:

- **Model:** Google Gemini-2.5 Flash.
- **Stage 1 – Table Pruning**: `identify_relevant_tables` prompt returns only the tables that matter.
- **Stage 2 – Contextual Previews**: fetch `SELECT * LIMIT k` rows to show formats and implicit relations.
- **Stage 3 – SQL Generation** with those previews.
- **Stage 4 – Recursive Self-Correction** (`execute_and_evaluate` with try/except → `fix_sql` prompt).
- **Stage 5 – Semantic Validation**: a secondary LLM call flags illogical results (e.g., zero area for a floor).

- **Stage 6 – Answer Formalisation**: `formalizing_response` crafts a clear Russian reply.
- **Infrastructure:** all LLM outputs are JSON-validated via Pydantic models, keeping the code path deterministic.

Results: the pipeline consistently auto-repairs both syntax and semantic errors; internal tests show high answer correctness (exact figures pending broader evaluation).

Limitations:

- No dependency graph yet—queries run independently even when later steps could reuse earlier IDs.
- Single-turn only; a dialogue mode for follow-up clarifications is on the roadmap.

What didn't work:

- Passing the full DB schema blew out the context window.
- A standalone *Validator agent* caused infinite loops.
- Summarising error traces removed vital details, preventing effective self-correction.
- Swapping Gemini for Mistral 7B cut accuracy from 6/10 to 4/10 on the reference benchmark.

Conclusion

Two complementary pipelines were developed to bridge the gap between natural-language questions and IFC data queries. The first pipeline embeds a textual description of the database schema directly into the LLM's context, which allowed the model to generate correct SQL or Cypher queries for nine out of ten test questions without any preliminary database inspection. However, as BIM models grow in size and complexity, maintaining and updating this in-prompt schema becomes increasingly cumbersome and difficult to scale.

The second pipeline relies on tool calls to explore the database at runtime—invoking commands like SHOW TABLES or DESCRIBE—to build an internal understanding of the schema. This automation reduces the need for manual schema maintenance, but query quality currently lags behind expectations, succeeding on roughly six out of ten questions. Ongoing work on agent orchestration, error handling, and retrieval strategies is aimed at closing this gap.

Overall, our open-source solution surpasses the baseline in transparency and customizability, and it supports three interchangeable data back-ends—relational (SQL), document (JSON), and graph (Memgraph)—to suit different query patterns. Next steps include optimizing the schema-in-prompt approach for very large models and strengthening the tool-based agent framework to improve consistency and accuracy.