KNOWLEDGE GRAPH CREATION USING LLM AND NLP FOR GRAPHRAG APPLICATIONS

Martin Uždil

Bachelor's thesis
Faculty of Information Technology
Czech Technical University in Prague
Department of Applied Mathematics
Study program: Informatics
Specialisation: Artificial Intelligence
Supervisor: Ing. Rastislav Tkáč
May 16, 2025

FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Knowledge Graph Creation using LLM and NLP for GraphRAG Applications
Student:	Martin Uždil
Supervisor:	Ing. Rastislav Tkáč
Study program:	Informatics
Branch / specialization:	Artificial Intelligence 2021
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2025/2026

Instructions

Retrieval Augmented Generation (RAG) applications require for the retrieval phase an information source, that provides efficient search and retrieval of relevant materials. LLM then uses the retrieved materials as a base for final output generation. The first generation of RAG applications typically use vector or full-text indexes, or a combination thereof, for retrieval. The disadvantage of these indexes is their limited ability to capture connections and relationships between records that are semantically distant. One way to overcome these limitations and improve overall relevance of the responses inferred by LLM models, is the use of graph databases, which allows knowledge representation in the form of a graph. Individual records are represented as nodes, relationships and connections as graph edges. Both nodes and edges can be enriched with properties, enabling efficient search within the graph. In effect, graph acts as a network of tags for each chunk of information, providing organising principle for the knowledge base. This approach has been termed GraphRAG.

Manual creation of knowledge graphs from hundreds of documents is usually not a viable option and therefore an automation of the creation process is necessary for such cases. NLP and LLM models can be used to process information chunks, identify properties and relations.

Objective:

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 7 November 2024 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

The aim is to design and implement a process for creating a knowledge graph usable for a GraphRAG application that meets the following requirements:
- The consumer will be an application serving as an advisor/assistant for company employees
- The input consists of dozens of electronic documents (e.g., PDF, DOCX) containing methodological guidelines, instructions, and product information
- All documents are in the same language
- The input documents contain only textual information
- The process should require minimal manual operations
- The process will only address the initial population of the graph
- The process shall use LLM and/or NLP models for input processing and graph creation
- The target platform for process implementation is Microsoft Azure cloud
- The preferred graph database is Neo4j, unless other platform can bring signifficant advantages

Guidelines for Elaboration:
1. Conduct research on LLM and NLP models, methods and tools for processing input documents and creating the knowledge graph
2. Design individual phases of the process and select specific ways to implement them. Justify each decision
3. Implement the process according to the design while adhering to the above requirements
4. Integrate the created knowledge graph into a simple RAG application implementing suitable retrieval strategies
5. Import a selected test set of documents into knowledge graph and into vector index for comparison of GraphRAG and RAG retrieval approach
6. Use suitable automated test framework and perform the same tests of the retrieval from the knowledge graph and from the vector indexes
7. Evaluate and compare the test results of the compared implementations

---

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 7 November 2024 in Prague.

I am grateful to my supervisor, Ing. Rastislav Tkáč. Whenever I got stuck, he pointed me in the right direction and provided clear and useful feedback. I also owe my friends and family my gratitude for their constant reminders about the upcoming submission deadline.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. I further declare that I have concluded an agreement with the Czech Technical University in Prague, on the basis of which the Czech Technical University in Prague has waived the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant to Section 60(1) of the Copyright Act. This fact does not affect the provisions of Section 47b of the Act No. 111/1998 Coll., on Higher Education Act, as amended.

I declare that I have used AI tools during the preparation and writing of my thesis. I have verified the generated content. I confirm that I am aware that I am fully responsible for the content of the thesis.

In Prague on May 16, 2025

Abstract

This thesis explores whether adding a knowledge graph can improve question answering over Czech-language university regulations. Thirty-seven public ČVUT documents were converted into 1 328 text chunks; entities and relations were extracted with a Czech NER model and gpt-4.1-mini, deduplicated via embeddings plus a lightweight LLM check, and stored in a Neo4j property graph enriched with vector indexes. Three retrieval pipelines were compared on 141 questions: a dense-vector RAG baseline, and two graph-augmented variants anchored on LLM or NER entities. Automated evaluation with five RAGAS metrics showed the dense baseline still delivers the highest overall accuracy, though the LLM-anchored graph slightly improves median answer relevancy. The study concludes that graph structure alone is not yet sufficient without smarter retrieval, but the delivered pipeline, entity-resolution method and evaluation harness provide a solid platform for future graph-aware QA research in Czech.

Keywords GraphRAG, knowledge graph, large language models, natural language processing, Neo4j, graph-based retrieval, document processing, vector indexes

Abstrakt

Tato práce zkoumá, zda může znalostní graf zlepšit zodpovídání dotazů nad českými univerzitními předpisy. Sedmatřicet veřejných dokumentů ČVUT bylo rozděleno do 1 328 textových úryvků; entity a vztahy byly získány pomocí českého NER modelu a gpt-4.1-mini, deduplikovány kombinací embeddingů a jednoduchého LLM volání a uloženy do property grafu Neo4j doplněného o vektorové indexy. Byly porovnány tři metody vyhledávání na 141 otázkách: vektorový RAG baseline a dvě grafem rozšířené varianty RAG založené na entitách z LLM či NER. Automatická evaluace pěti metrikami RAGAS ukázala, že baseline RAG zatím poskytuje nejvyšší celkovou přesnost, i když LLM-graf mírně zvyšuje medián relevance odpovědí. Studie uzavírá, že samotná grafová struktura nestačí bez chytřejšího vyhledávání, avšak implementovaná pipeline, metoda pro deduplikaci entit a automizovaná evaluace tvoří solidní základ pro další výzkum grafově orientovaného QA v češtině.

Klíčová slova GraphRAG, znalostní graf, velké jazykové modely, zpracování přirozeného jazyka, Neo4j, vyhledávání v grafu, zpracování dokumentů, vektorové indexy

Contents

List of Figures

List of Tables

List of Code listings

List of abbreviations

RAG	Retrieval Augmented Generation
GraphRAG	Graph-based Retrieval Augmented Generation
LLM	Large Language Model
NLP	Natural Language Processing
NER	Named Entity Recognition
KG	Knowledge Graph
QA	Question Answering
Cypher	Query language for Neo4j

Introduction

Companies, schools, archives, or any large institutions often have a vast number of documents, guidelines, and procedures that their users (employees, students) need to understand—or at least be able to search through efficiently. Traditionally, this has been handled through keyword-based search. We don't have to look far—take the academic world, for example. The National Technical Library still uses a search engine that works this way, indexing nearly *a billion* publications. Want to find all publications about cars? You search for "((car) OR (automobile))". Make a typo and write "automobele" instead of "automobile"? You get no results. Keyword search doesn't handle typos, synonyms, or semantically similar text.

In business environments, users need answers to questions like:

- How do I set up tariff X for a customer?
- Who handles complaints about product Y?
- What are the side effects of drug Z?
- How do I apply for a credit card?

These kinds of problems are solved by **embeddings**. Embeddings are vector representations of text that capture the meaning and context of words. They're generated by machine learning models that convert words, sentences, or entire paragraphs into numerical vectors that reflect their semantics and interrelationships. Similarity search can then be performed over these vectors. This allows us to build systems that can answer natural-language questions—even when the user doesn't use the exact wording found in the documentation. There's no need to know the right keywords or syntax—just ask a question in plain language.

This approach forms the basis of **retrieval-augmented generation** (RAG), where the query is first used to retrieve relevant information from a knowledge base (via embeddings and vector search), and then a generative

model (like an LLM) uses that information as context to compose the final answer.

RAG significantly improves the accuracy, freshness of answers and significantly reduces hallucinations. However, its effectiveness drops when facing more complex queries that mention multiple entities, processes, or steps scattered across documentation. For example, a query like:

> What is the return process for goods purchased via the company's online store, if the payment was made by card and the item is returned after 14 days?

requires combining information from multiple parts of the documentation—terms and conditions, internal procedures, refund policies, etc.

This is where **GraphRAG** comes in—an enhancement of classic RAG that adds an explicit knowledge graph. Instead of working solely with unstructured texts, GraphRAG extracts and stores relationships between entities (e.g., *product* → *has return period* → *14 days*) in a graph database. Queries can then be answered not only through semantic similarity, but also by navigating structured relationships. This makes it easier to understand user intent, retrieve relevant facts, and infer implicitly mentioned connections.

GraphRAG thus combines the power of language models, vector search, and formal logic of knowledge graphs—which is crucial if we want to turn documentation from a passive archive into an active assistant.

Chapter 1

Theoretical Background and Related Work

1.1 Knowledge Graph Concepts

A *knowledge graph (KG)* is a structured representation of knowledge that encodes real-world entities as nodes and explicitly models the relationships between these entities as edges.

Knowledge graphs can be represented through various graph data models. Two prominent models described by (1) are:

- **Directed Edge-labelled Graphs (DEL)**: DEL graphs, or multi-relational graphs, consist of nodes representing entities and edges explicitly labelled to represent binary relationships between these entities. RDF (Resource Description Framework) is an example of a standardized DEL graph data model, using triples (subject–predicate–object) to structurally represent knowledge.
- **Property Graphs**: Property graphs extend the basic graph model by allowing nodes and edges to have associated labels and multiple property–value pairs. This model enables flexible data annotations directly on edges and nodes, facilitating complex representations. Although property graphs are not yet standardized, they are prominently implemented in widely-used graph databases such as Neo4j.

In this thesis, we specifically focus on property graphs, consistent with their practical implementation in Neo4j.

1.1.1 Knowledge Graph Construction from Text

Building a knowledge graph from unstructured sources—known as *knowledge graph construction*—is commonly decomposed into the following stages(1):

- **Knowledge acquisition** – automatic extraction of entity mentions and relations from text;
- **Knowledge refinement** – canonicalization, duplicate resolution, and noise reduction;
- **Knowledge evolution** – incremental maintenance of the graph as data change.

This thesis focuses on the first two phases for a static corpus; evolution is left to future work.

1.1.2 Information-Extraction Tasks

Named Entity Recognition (NER) identifies and classifies entity mentions (e.g., *Person*, *Organization*, *Location*).

Relation Extraction (RE) detects semantic relations between entity pairs (e.g., *Person–worksFor–Organization*).

Early open-IE systems such as *KnowItAll* and *TextRunner* demonstrated large-scale Web extraction(2, 3). Modern approaches rely on supervised or transformer-based models, achieving state-of-the-art accuracy in well-resourced languages(4, 5).

1.2 Named Entity Recognition and Czech Language Processing

NER overview. Named Entity Recognition (NER) seeks to locate spans of text that denote real-world entities and assign them a category label (e.g., *Person*, *Organization*, *Location*). Current state-of-the-art systems fine-tune large transformer encoders such as BERT (6) or RoBERTa (7) on manually annotated corpora, achieving high recall and precision across a hierarchy of coarse and fine-grained entity types.

Czech-specific challenges. Applying NER to Czech introduces additional difficulties:

- **Morphological complexity.** Czech is highly inflected; an entity can surface in multiple case forms (*Praha* vs. *v Praze*, *Jan Novák* vs. *Janem Novákem*). A robust model must normalize these surface variants to a single canonical entity.
- **Limited resources.** Compared with English, Czech offers fewer large annotated datasets. The principal resource is the *Czech Named Entity Corpus* (CNEC 1.1) comprising ∼5.9 k sentences and 33 k annotated entities in a two-level hierarchy of "supertypes" and subtypes (8).

CNEC and the RobeCzech model. We employ the pre-trained model `stulcrad/CNEC_1_1_Supertypes_robeczech-base` (9). It builds on a `ufal/robeczech-base` fine-tuned on CNEC 1.1 and reports an F1 score of $\approx 86.7\%$ on the corpus evaluation split, covering the major categories required for this thesis (persons, organizations, locations, *etc.*) (10).

Rationale for a dedicated Czech NER. Using a language-specific NER model ensures high recall of Czech entity names and their correct type assignment, which would be difficult for a general-purpose LLM without further fine-tuning. Offloading surface-form recognition to this specialized model allows the LLM to focus on higher-level tasks such as relation extraction and reasoning, while reducing both cost and error propagation in the pipeline.

1.3 Relation Extraction and Knowledge Graph Schemas

Problem definition. Relation Extraction (RE) aims to automatically identify the relations between entities in unstructured text. Formally, given a natural-language text x, the goal is to predict a set of triplets $\{(e_1, r, e_2)\}$, where e_1 and e_2 are entity mentions in x, and r is a relation type drawn from a predefined set R (5).

Predefined relations vs. Open relation extraction.

- **Predefined relations** limits r to a fixed inventory of relation labels (plus a "no-relation" class), requiring supervised examples for each label.
- **Open relation extraction** extracts arbitrary predicate phrases r directly from text without a predefined schema (e.g. "works at", "is part of").

Techniques. Early techniques used pattern-based or dependency-parse rules with statistical classifiers. Supervised neural models require annotated corpora, while modern transformer-based and few-shot/zero-shot methods (e.g. LLM prompts) generalize to unseen relations without extensive labeled data (5).

Property graph schema. Extracted triplets populate a Neo4j property graph:

- **Nodes** are labelled by entity type (e.g. in our case `Chunk`, `NLPEntity`, `LLMEntity`[1]), each carrying properties such as `name` or `chunk_id`.
- **Edges** use the predicate r as the relationship name (e.g. `applies_to`, `works_at`).

[1] See Chapter 3 for the full node definitions and properties.

1.3.1 Large Language Models for Knowledge Extraction

LLMs as extractors and reasoners. Large language models trained on web-scale corpora enable *in-context learning*: with a prompt alone, they can emit structured outputs—JSON triples, entity lists, and so forth—without task-specific fine-tuning. Evaluations of GPT-style models show that few-shot prompting attains near–state-of-the-art F_1 on standard relation-extraction benchmarks, effectively matching dedicated supervised systems (11). When the task requires multi-hop reasoning over a knowledge graph, LLM-augmented pipelines such as GMeLLo achieve new state-of-the-art results, surpassing earlier specialized models (12). Hence, specialized neural extractors and LLMs serve complementary roles: the former provide systematic coverage where large labelled datasets exist, whereas the latter excel at filling schema gaps, inferring implicit relations, and answering complex natural-language queries.

Prompting strategies for extraction.

- **Direct prompts** — "Extract all (*subject*, *relation*, *object*) triples from the paragraph below."
- **Few-shot prompts** — embed 2–3 demonstration examples so the model can imitate the required JSON format (13).
- **Chain-of-thought prompts** — ask the model to enumerate entities first, then derive relations, boosting recall on syntactically complex sentences (14).

Advantages of an LLM-based extractor.

- Offers strong multilingual support, including morphologically rich languages such as Czech.
- May capture implicit or paraphrased relations that rule-based or pattern-based systems often miss.
- Removes the need to curate large Czech-language relation datasets for supervised training.

1.3.2 Embeddings

As discussed in the Introduction, vector embeddings map words to points in a continuous space, reflecting their distributional semantics (15).

Definition. Let $\mathcal{V}$ be a vocabulary and $d \in \mathbb{N}$ a fixed dimension. A (static) word-embedding function is

$$\phi_{\text{word}} : \mathcal{V} \longrightarrow \mathbb{R}^d,$$

such that semantic similarity between tokens correlates with geometric proximity of their vectors (15).

Transformer encoders extend this idea to full sequences:

$$\phi_{\text{seq}} : \mathcal{V}^* \longrightarrow \mathbb{R}^d,$$

yielding *contextual* embeddings whose value for a token depends on its surrounding words (15). Sequence-level vectors are now standard in dense retrieval and RAG pipelines because they capture sentence- and paragraph-level semantics.

Scope in this thesis. Unless stated otherwise, we treat an embedding as a single function

$$\phi : \mathcal{V}^* \longrightarrow \mathbb{R}^d,$$

implemented by a pre-trained multilingual model. Queries, sentences, and document chunks are passed to ϕ, which returns one d-dimensional vector, even if the input may be a single word or a full paragraph.

1.3.2.1 Cosine similarity

Similarity search relies on the *cosine-similarity* function

$$S_C(A, B) = \frac{A \cdot B}{\|A\| \, \|B\|} = \frac{\sum_{i=1}^{n} A_i \, B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}, \qquad S_C : \mathbb{R}^n \times \mathbb{R}^n \longrightarrow [-1, 1],$$

where A_i and B_i denote the i-th components of vectors **A** and **B**, respectively (15).

The expression is the cosine of the angle between two vectors in n-dimensional space. Intuitively, $S_C = 1$ implies identical directions (angle $0°$), while $S_C = -1$ implies opposite directions (angle $180°$). Because the measure depends on direction rather than magnitude, it is well suited for quantifying semantic similarity in high-dimensional embedding spaces.

1.3.2.2 Structured outputs and constrained decoding

OpenAI's *Structured Output* mode allows developers to attach a JSON Schema to a prompt; the model's response *must* conform to that schema (16). Under

the hood, the schema is compiled once into a context-free grammar (CFG). During generation the decoder performs *dynamic constrained decoding*: after each token, the grammar restricts the set of valid next tokens to those that keep the partial output within the language of the CFG, effectively forcing syntactically correct JSON. Earlier IE pipelines relied on regex parsers and post-hoc validations—constrained decoding removes this engineering burden by guaranteeing syntactically valid JSON with negligible runtime overhead.

1.4 Retrieval-Augmented Generation (RAG)

Motivation. Prompting a large language model can answer many factoid questions, but three well-documented issues remain (15):

1. *Hallucination.* LLMs occasionally invent facts with high confidence, especially in specialized domains.
2. *Scope.* A frozen model cannot answer questions about proprietary or rapidly changing data.
3. *Calibration.* LLMs are not reliably aware of when they are guessing, making it hard for users to judge answer quality.

RAG addresses these problems by grounding the model's response in retrieved documents that the user can inspect.

Two-stage architecture. The canonical RAG pipeline consists of two components [15, § 14.3]:

Retriever embeds the user query, searches a document index, and returns the top-k passages whose vectors are closest by cosine similarity.

Reader / Generator is an LLM that receives the retrieved passages concatenated with the original query and autoregressively generates an answer conditioned on this augmented prompt.

This retrieve-then-generate design was popularized in open-domain QA systems such as DrQA (17) and remains the dominant pattern in industrial search assistants. The usual implementation of the RAG QA pipeline is shown in Figure 1.1, where the retriever is split into an embedding model and a vector database, while the generator is the LLM itself.

Sparse vs. dense retrieval. Early QA engines matched TF–IDF vectors; modern retrievers use *dense* BERT-style embeddings that better capture synonymy and paraphrase (15). In this thesis both the baseline system and our graph-augmented variant rely on dense vector search over chunk embeddings.

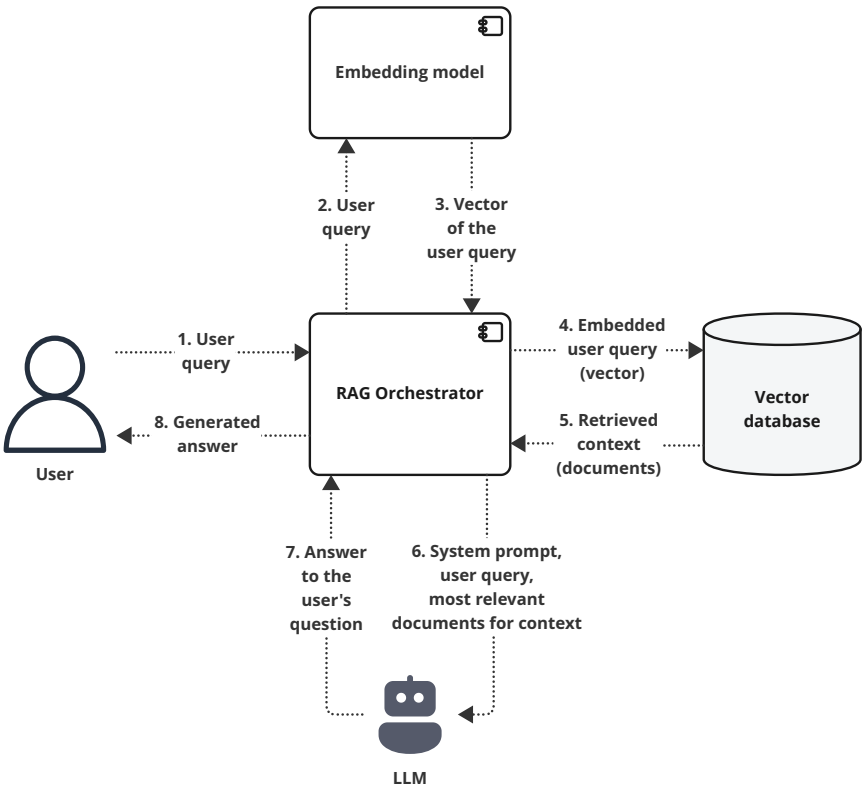

Figure 1.1 Overview of the RAG QA pipeline.

Strengths and limitations. RAG reliably answers single-hop factual questions with source-grounded evidence. Nonetheless, empirical studies highlight two persistent challenges:

- **Multi-hop reasoning**. If the answer requires combining facts from multiple passages, the LLM must infer the connection itself, often resulting in partial or incorrect answers. (15, 12)
- **Context length**. Long or holistic queries may require more passages than comfortably fit into the model's prompt window.

These limitations motivate our exploration of GraphRAG, which injects knowledge-graph structure to aid multi-hop retrieval and succinct context selection.

Evaluation tooling. Open-source frameworks such as *Haystack* and *LangChain* (18, 19) assemble off-the-shelf modules for chunking, embedding, retrieval, and prompt construction, while libraries like *RAGAS* (20) provide automated metrics for answer correctness, faithfulness, and context recall. We adopt the same metrics in Chapter 4.

1.5 Graph-enhanced Retrieval-Augmented Generation (GraphRAG)

Motivation. Vector-only RAG treats the knowledge base as an unordered collection of passages. When an answer depends on traversing explicit relations—for example, *degree → issued by → faculty → belongs to → university*—the retriever may surface the right fragments but leaves the LLM to infer the links (cf. Section 1.4). GraphRAG incorporates a knowledge graph (KG) into the retrieval loop so that graph traversal can steer which passages *and* which structured facts enter the prompt, reducing hallucination and improving multi-hop coverage (21).

High-level pipeline. Figure 1.2 sketches the similarity in implementation of RAG and GraphRAG pipelines. For a direct visual comparison, see also Figure 1.1.

Core stages.

1. **Entity extraction & linking.** The user query is passed through the same NER model as used at ingestion time to yield a set $\mathcal{E}_q$ of entity nodes in the KG.
2. **Subgraph retrieval.** A breadth-first expansion of radius $h \leq 2$ around $\mathcal{E}_q$ produces a candidate subgraph G_q. Candidate nodes are ranked by

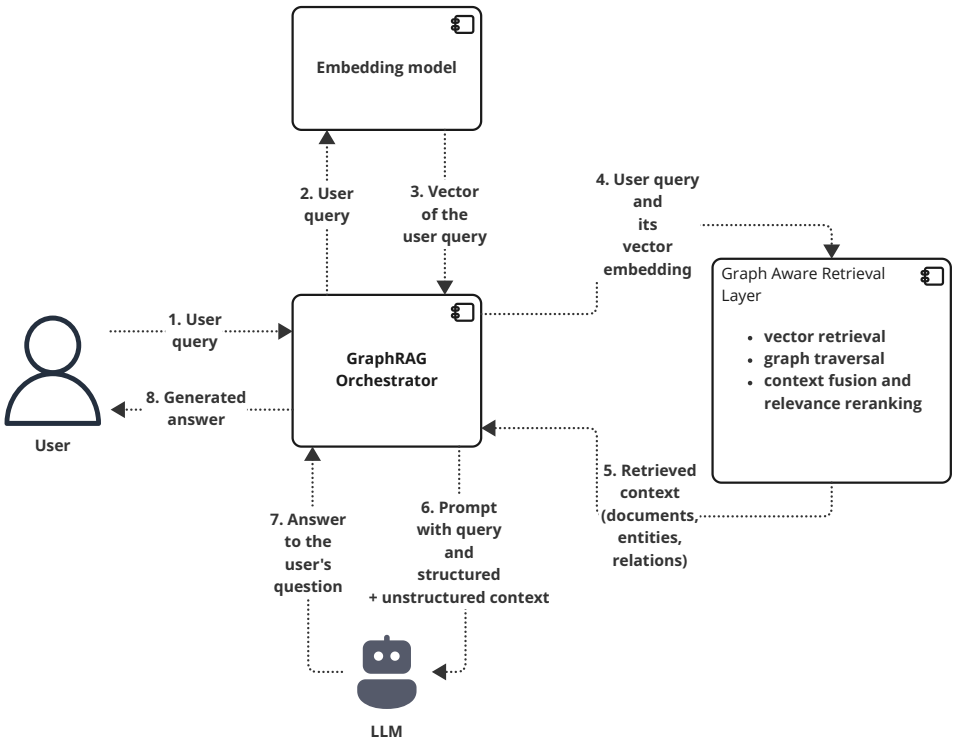

Figure 1.2 Overview of the GraphRAG pipeline.

a hybrid score that combines cosine similarity between the query embedding and node embeddings with lightweight topological weights for salient relations.

3. **Context assembly.** Text passages attached to the top-k nodes, together with a linearized list of triples $\langle e_s, r, e_t \rangle$, are appended to the LLM prompt.

4. **Answer generation.** The LLM generates the final answer conditioned on both the retrieved passages and the structured triples.

Observed benefits. Both Shavaki; Omrani; Toosi; Akhaee [21] and the Microsoft reference implementation (22) report that GraphRAG outperforms standard RAG on multi-hop reasoning tasks. Furthermore Microsoft's implementation, evaluated on proprietary support-document corpora, notes higher answer accuracy and better grounding.

Challenges. GraphRAG introduces additional considerations beyond KG construction:

- **Graph completeness.** Missing or incorrect edges can block traversal and hide relevant context.
- **Error propagation.** Extraction errors at ingestion time propagate into retrieval and ranking.

A quantitative comparison between vector-only RAG and GraphRAG on our university-document dataset is presented in Chapter 4.

1.6 Chapter Summary

This chapter reviewed the theoretical foundations and prior work that ground the thesis:

- **Knowledge graphs** formalize entities and relations. We focus on the *property-graph* model adopted by Neo4j.
- **Information extraction** enables KG construction from text. A Czech-specific NER model handles morphologically rich mentions, while LLM-based relation extraction supplements structured triples.
- **Embeddings** provide a vector space in which cosine similarity approximates semantic relatedness; dense embeddings underpin modern retrieval.
- **Retrieval-Augmented Generation (RAG)** combines a vector retriever with an LLM reader to ground answers in source passages, yet struggles with multi-hop reasoning and context limits.
- **GraphRAG** integrates a knowledge graph into the retrieval loop, guiding passage selection via graph traversal and improving multi-hop question answering.

These concepts motivate the thesis methodology: we will (i) build a property-graph KG from Czech university documents, (ii) implement baseline RAG and GraphRAG pipelines atop that graph, and (iii) evaluate their performance using automated and manual metrics. The following chapter details the end-to-end system design and implementation choices.

Chapter 2

Methodology: Knowledge-Graph Construction Pipeline

2.1 Overview of the Proposed Solution

This chapter presents the end-to-end methodology for constructing a hybrid knowledge graph from a small corpus of Czech-language university documents. The pipeline ingests PDFs, extracts structured knowledge using both NLP-based and LLM-based methods, performs entity deduplication, and writes a graph-structured representation into Neo4j. The resulting graph is augmented with dense vector embeddings to support hybrid retrieval.

Figure 2.1 provides an overview of the core components involved in building the knowledge graph:

- **Data ingestion:** Input documents are parsed and segmented into overlapping character-level chunks.
- **Entity extraction:** Each chunk is passed through two parallel extractors:
  - a Czech-specific NER model for identifying canonical NLP entities,
  - a GPT-based extractor using few-shot prompting and structured output constraints, producing both LLM entities and relations.
- **Entity deduplication:** The LLM entities undergo semantic deduplication via cosine similarity, followed by LLM-assisted disambiguation and union-find clustering.
- **Graph construction:** The KG builder module writes all nodes, edges, and relations into Neo4j, and computes embedding vectors for the chunk nodes.

- **Vector indexing:** Embeddings are stored in Neo4j's vector index to enable dense retrieval queries.

The architecture supports both classical information extraction (via the NER pipeline) and modern generative techniques (via LLM prompts), enabling robust multi-strategy retrieval downstream.

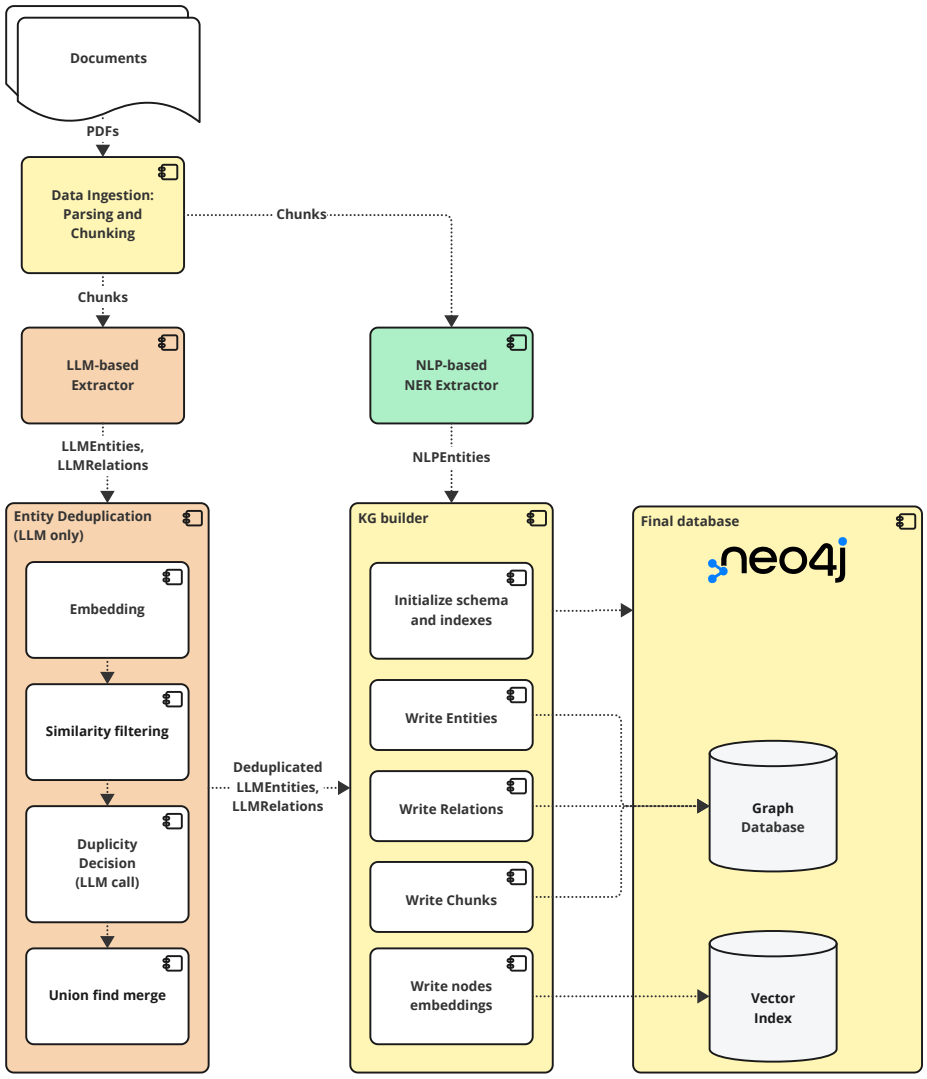

Figure 2.1 Overview of the knowledge-graph construction pipeline.

Design rationale. The pipeline reflects a pragmatic balance between engineering feasibility and information quality. It combines traditional NLP pipelines with prompt-based large language models to compensate for the

limited availability of Czech-language relation datasets. By performing early entity deduplication and using a graph-native database with hybrid search support, the system is optimized for question answering. This modular architecture also enables experimentation with components such as deduplication thresholds, embedding models, or LLM prompts.

2.2 Data Ingestion: PDF Parsing and Text Chunking

The input corpus consists of **thirty seven** official Czech-language university documents (e.g., accommodation rules, study regulations, code of ethics), each provided in PDF format. The documents contain machine-readable text and require no OCR or language-specific preprocessing.

2.2.1 Lightweight PDF extraction with `PyMuPDF`

Text content is extracted using `PyMuPDF` (23), a lightweight parser that offers reliable per-page access. It was selected over more advanced document loaders due to the homogeneous and clean structure of the input corpus.

For production use — more robust libraries such as `Markitdown` (24) would be preferred, as it handles more formats. The current design, however, did not require multiple document types.

2.2.2 Fixed-size character chunking

Each document is segmented into overlapping character-level chunks with fixed parameters:

$$\texttt{chunk\_size} = 500, \qquad \texttt{overlap} = 50.$$

This simple sliding-window approach ensures local coherence while allowing co-reference and relation spans to survive boundary splits. The chunking preserves newlines and does not normalize whitespace or punctuation.

Another possible approach would be to use semantic chunking, where the text is split into semantically coherent parts.

2.2.3 Reproducible cache format

The final output of this stage is a list of 1328 text chunks. These are cached to a `.kgcache` file for deterministic reuse for graph augmenting experimentation in later stages.

2.3 Named Entity Extraction

The pipeline identifies real-world actors and concepts with two complementary extractors: a language–specific NER model (*high precision for canonical names*) and a prompt-driven LLM extractor (*wide coverage of domain concepts and relations*). Keeping the two streams separate lets us measure their individual utility before any reconciliation.

2.3.1 Czech NER baseline

For language-specific tagging we adopt `stulcrad/CNEC_1_1_Supertypes_robeczech-base` (10). The model starts from `ufal/robeczech-base`, a Czech RoBERTa encoder pre-trained on circa 5 B tokens, and is fine-tuned on the Czech Named Entity Corpus (CNEC 1.1) (8). It reaches an F_1 of ≈86.7 % on the CNEC evaluation split, sufficient for tagging person, organisation, and location names that recur in university regulations.

- Each chunk is fed to the model; every predicted span becomes a node labelled `NLPEntity`.
- The chunk itself receives outgoing `MENTIONS` edges to all entities it contains.
- No cross-chunk merging is attempted at this stage; duplicates are handled by the global deduplication step (Section 2.5).

2.3.2 Open extraction with an LLM

Many salient concepts in the documents are *not* classical named entities—e.g. “ubytovací smlouva” (accommodation contract) or “dočasné přerušení studia” (study suspension). To capture such domain-specific terms and the relations between them we query gpt-4.1-mini in structured-output mode.

Schema-constrained prompt. Each 500-character chunk is passed to the model together with the `pydantic` schema in Listing 2.1. OpenAI’s JSON-schema decoding (16) forces the reply to instantiate:

- `Entity` (name, optional type);
- `Relationship` (source, relation, target);
- `ExtractionOutput` (lists the above).

Code listing 2.1 Simplified `pydantic` schema supplied to gpt-4.1-mini.

```
class Entity(BaseModel):
    name: str
    type: Optional[str] = None

class Relationship(BaseModel):
    source: str
    relation: str
    target: str

class ExtractionOutput(BaseModel):
    entities: List[Entity]
    relationships: List[Relationship]
```

Graph projection.

- Every returned entity becomes an `LLMEntity` node.
- Each `Relationship` is mapped to a directed edge whose label is the LLM-supplied predicate (`requires`, `governs`, *etc.*).
- The originating chunk is linked to all LLM entities via `MENTIONS` edges, mirroring the NLP pipeline.

The two extractor streams thus create parallel views of the corpus: high-confidence canonical names from Czech NER, and flexible LLM-derived concepts plus explicit *in-text* relations. Section 2.5 details how LLM entities are deduplicated and how both layers are co-indexed for downstream querying.

2.4 Relation Extraction with the LLM

The structured output returned by gpt4.1-mini already contains directed triples `(source, relation, target)`. This section explains how the triples are validated and written to the graph.

2.4.1 Minimal validation

For each chunk we iterate over the `relationships` list of the `ExtractionOutput`. A triple is accepted if and only if the *source* and *target* strings exactly match the `name` field of some `LLMEntity` extracted from that *same* chunk. Triples that reference unknown entities are logged and ignored (roughly a few dozen per full run).

2.4.2 Edge creation

Every surviving triple produces a directed edge

$$(e_s) \xrightarrow{relation} (e_t),$$

where $e_s, e_t \in$ `LLMEntity`.

2.4.3 Linking text evidence

Each chunk node is connected to every entity it mentions via a MENTIONS edge—irrespective of whether the entity originated from the Czech NER or from the LLM extractor. These links later permit users (or evaluation scripts) to trace any answer back to the exact passage in the source PDFs.

2.4.4 Graph snapshot after relation loading

- **Chunk** nodes $\longrightarrow$ MENTIONS $\longrightarrow$ **Entity** nodes
- **LLMEntity** nodes are connected by $\approx 5\,000$ relation edges drawn from roughly 500 distinct predicate labels (numbers vary slightly across runs due to the nondeterministic nature of the LLM).
- At this stage no attempt is made to merge semantically similar predicates; edge labels remain exactly as produced by gpt-4.1-mini.

The resulting graph is therefore rich but intentionally heterogeneous: relationship predicates preserve the meaning of the source text, while the MENTIONS links ensure every factual edge can be grounded in its originating chunk.

2.5 Entity Resolution and Deduplication

Processing the documents chunk-by-chunk yields many near-duplicate `LLMEntity` nodes: out of $\approx$2 100 raw mentions only $\approx$1 400 correspond to truly distinct concepts. Left unresolved, such variants fragment the graph and scatter incident relations. We therefore apply a two–stage *entity-resolution* procedure that combines fast embedding similarity with a targeted LLM check.

2.5.1 Candidate generation via cosine similarity

1. Each entity name is embedded with `text-embedding-3-large` ($d = 3\,072$), yielding vectors v_e.

2. For every unordered pair (e_i, e_j) we compute $\cos(v_{e_i}, v_{e_j})$. Pairs scoring ≥ 0.80 are flagged as *candidates* for possible merger. The threshold was tuned empirically to favour precision over recall: a lower value produced unnecessary llm calls, whereas a higher one left obvious duplicates intact.

With $\sim$2 100 entities the number of candidate pairs remains manageable; no specialized nearest-neighbor index is required for this corpus size.

2.5.2 LLM confirmation

Each candidate pair is passed to a small, cost-controlled gpt-4.1-mini instance. The model receives the two surface forms (plus optional local context) and must return a structured verdict:

- `are_equivalent` {`yes`,`no`}
- `confidence` {`high`,`medium`,`low`}
- a short justification.

Only pairs with `are_equivalent=yes` *and* `confidence=high | medium` are accepted as duplicates. Low-confidence cases are conservatively left untouched.

2.5.3 Cluster formation and canonicalization

Accepted pairs are merged into clusters with a simple parent–pointer structure (conceptually equivalent to Union–Find). For every cluster the **longest** surface form in UTF-8 characters is retained as the canonical label; shorter aliases are mapped to it.

Graph rewiring then proceeds mechanically:

1. All edges incident on an alias node are reassigned to the canonical node.
2. The redundant alias node is removed.

The procedure does *not* attempt to merge or normalize predicate labels; relation strings remain exactly as produced by the LLM.

2.5.4 Outcome

- **Nodes.** The number of `LLMEntity` nodes decreases from $\sim$2 100 to $\sim$1 400 (-33 %).
- **Edges.** All $\approx$5 000 relation edges are preserved; their endpoints are simply re-anchored to the canonical nodes where applicable.

- **Performance.** On a laptop-class CPU the entire resolution run finishes in under 15 minutes, the bulk of the time spent in embedding calls; the LLM verification is parallelized with an asynchronous semaphore limited to five concurrent queries. The time may wary as the entity set may change with each run.

Only `LLMEntity` nodes are deduplicated. `NLPEntity` nodes remain separate so that the impact of the two extraction strategies can be analyzed independently. Extending the same resolution logic to NLPEntities—or to cross-link between the two subsets—remains an avenue for future work.

2.6 Knowledge-Graph Construction in Neo4j

All data are persisted in a local Neo4j 5.27.0 (community edition) instance with the APOC and GenAI plug-ins enabled. Neo4j's labelled-property graph matches the pipeline's output and allows both pattern matching (Cypher) and vector search in one place.

2.6.1 Graph schema

Chunk 1328 nodes, one per text segment (properties: `id`, `content`, `embedding`).

NLPEntity ≈610 nodes produced by the Czech NER model (properties: `name`, `type`).

LLMEntity ≈1 700 nodes after LLM-based deduplication (properties: `name`, `embedding`).

Edges comprise

- **MENTIONS** (*Chunk*→*Entity*) – every entity occurrence in a chunk.
- **Semantic relations** ($LLMEntity \xrightarrow{r} LLMEntity$) – roughly 5 000 directed edges labelled by the verb phrase r extracted from gpt-4.1-mini.

Insertion is executed once, in batch mode, using Cypher `MERGE` so that rerunning the loader is idempotent.

2.6.2 Vector indexes

The GenAI plug-in is used to create three approximate-nearest-neighbor (HNSW) indexes:

Exact look-ups on `name` or `id` rely on the default string indexes Neo4j creates automatically when these properties are used in equality predicates.

Label	Property	Dimensions	Metric
Chunk	`embedding`	3 072	cosine
LLMEntity	`embedding`	3 072	cosine
NLPEntity	`embedding`	3 072	cosine

Table 2.1 Embedding properties used in the Neo4j graph

2.7 Embedding Nodes and Vector Indexing

Offline embedding. Text vectors are generated once with `text-embedding-3-large` (3 072 d):

- the 1328 chunk texts;
- the canonical names of all `LLMEntity` nodes.

Each vector is written to Neo4j via `db.create.setNodeVectorProperty`, after which the above indexes are built. `NLPEntity` nodes are kept without embeddings in the current prototype.

Query-time usage. User questions are embedded on-the-fly and fed into Cypher queries that combine graph patterns with vector similarity, e.g.:

```
MATCH (c:Chunk)
WHERE vector.similarity.cosine(c.embedding, $qvec) > 0.75
RETURN c.content
ORDER BY similarity DESC
LIMIT 5
```

This hybrid capability is central to the GraphRAG strategies described in Section 2.8. No graph evolution is performed in the present study; the database is loaded once for all subsequent experiments.

2.8 Query–Answering Strategies

The completed knowledge graph supports three retrieval pipelines that share an identical answer-generation back-end. All variants execute the same outer loop:

1. Embed the user question with `text-embedding-3-large` (3 072-d, cosine space).
2. Retrieve up to $k = 5$ candidate chunks.
3. Concatenate $\leq$ 10 chunks into a fixed prompt template and ask `gpt-4.1-mini` to answer in Czech, *citing only the supplied context.*

   Step 2 differs for each strategy.

2.8.1 RAG-*vector* (baseline)

(a) Perform a k-nearest-neighbour search on all `Chunk.embedding` vectors using the question embedding.

(b) Keep the five chunks whose cosine similarity is at least 0.70.

No graph traversal is involved; this mirrors a "vector-only" RAG pipeline.

2.8.2 GraphRAG-*NLP* (NER anchors)

(a) Run the Czech NER extractor on the question and collect the entites it recognises from question.

(b) Embed every recognised entity and issue a k-NN search on `NLPEntity.embedding`; keep the five closest `NLPEntity` nodes (if any).

(c) From those entities gather all directly mentioned chunks `(c)<-[:MENTIONS]-(e)`.

(d) Re-rank the distinct chunks by cosine similarity to the *question* embedding and keep the top five.

(e) If no chunks are retrieved (due to the low relevance or failure to detect any entities in the question), fall back to the baseline vector search.

`NLPEntity` nodes have no inter-entity edges, so the expansion is limited to entity → chunk links.

2.8.3 GraphRAG-*LLM* (LLM anchors)

(a) Run LLM entity extraction on the question and collect the entities it recognises.

(b) Embed every recognised entity and issue a k-NN search on `LLMEntity.embedding`; keep the five closest `LLMEntity` nodes.

(c) One-hop graph expansion gathers

1. chunks mentioned directly `(c)<-[:MENTIONS]-(e)`;
2. chunks attached to neighbouring entities reached via a single typed semantic edge `(e)-[:REL*1]-(e2)`.

(d) Deduplicate the resulting chunks, re-rank them by cosine similarity to the question embedding and keep the top five.

(e) If no chunks are retrieved (due to the low relevance or failure to detect any entities in the question), revert to the baseline vector search.

Why evaluate *both* GraphRAG variants? NER-anchored retrieval is inexpensive and highly precise, but it can only pivot on the entities that the Czech NER model recognises in the corpus it was trained on. Proper deduplication without LLM is very complex issue, since it can't be done with embeddings only or fulltext similarity. For example, dates or references to sections of text (such occur very often in law texts) are very similar by cosine similarity, but they are not the same entities. And you also need to merge entities such as "ČVUT" and "České vysoké učení technické v Praze" into one entity, so you can't choose any simple string metrics such as levenshtein distance either. LLM-anchored retrieval, on the other hand, can pivot on every concept the generative model decides is salient. This richer graph broadens recall, yet it requires additional LLM calls during extraction and thus raises cost.

Running the two approaches side-by-side therefore answers a practical question: *Does the extra recall gained from LLM-derived entities justify their higher computational (and monetary) cost compared with a lean, NER-only pipeline?*

Prompt and generation. The same zero-temperature `gpt-4.1-mini` prompt is reused in all settings. If no chunk provides the answer, the model is instructed to return the fixed sentence: *"Nemám dostatek informací v kontextu."* ("I do not have enough information in the context." in Czech).

Reproducibility. All retrieval steps are expressed in Cypher and executed through a single Python class (`Neo4jQuestionAnswerer`); parameters (k, $\theta = 0.7$, context length 10) are fixed for every experiment reported in Chapter 4.

2.9 Additional Design Choices

Graph store: why Neo4j over Cosmos DB (Gremlin API). Cosmos DB was evaluated early on, but its Gremlin interface does not ship with a built-in vector index—semantic search would have required provisioning a separate Azure AI Search instance. Neo4j 5.27 provides both labelled-property graphs and an HNSW-based vector index behind a single Cypher endpoint, so one database is enough for hybrid retrieval.

Entity-merge threshold & model choices. The 0.80 cosine cut-off emerged from hands-on probing rather than an exhaustive grid-search. Canonical synonym pairs such as "ČVUT" versus its full Czech name land well above that value and should be merged, whereas genuinely different terms (e.g. "kreditní limit" vs. "stravovací stipendium") sit far lower. Near-identical surface patterns with different referents—legal clauses like "§ 48 odst. 2" versus "§ 32 odstavec 3", or unrelated calendar dates—also register high similarity and would be wrongly unified *unless* passed through the second-stage LLM discriminator that asks whether two mentions denote the same real-world entity.

Manual spot-checks confirm that this hybrid filter removes the vast majority of false merges, though a quantitative error rate is still future work.

All LLM calls use the `gpt-4.1` family. Among its three variants, `gpt-4.1-mini` proved the best compromise of token-level accuracy, response time and API cost while fully supporting strict JSON-schema outputs required by the extractor; the larger `gpt-4.1` is more accurate yet prohibitively expensive for batch extraction, and `gpt-4.1-nano` showed less extraction precision in early tests. Embeddings come from the co-hosted `text-embedding-3-large` endpoint for deployment simplicity, even though MASSIVE-MTEB leaderboards (25, 26) list multilingual models that score higher on retrieval benchmarks; integrating those stronger but externally-hosted encoders is logged as future work once the pipeline graduates beyond Azure OpenAI endpoints.

2.10 Chapter Summary

This chapter detailed an end-to-end methodology for transforming thirty seven Czech-language university regulations into a hybrid knowledge graph that supports retrieval-augmented question answering.

- **Ingestion.** PDFs are parsed with `PyMuPDF` and segmented into 1328 character-level chunks (`size`=500, `overlap`=50), then cached for deterministic reuse.
- **Extraction.** Each chunk is processed by *two independent extractors*: a Czech RoBERTa NER model that yields high-precision `NLPEntity` nodes, and a schema-constrained `gpt-4.1-mini` prompt that emits `LLMEntity` nodes plus explicit semantic triples.
- **Deduplication.** Roughly 2 100 raw LLM entities are clustered into $\approx$1 400 canonical nodes by combining 3 072-d cosine similarity ($\theta = 0.80$) with a lightweight GPT-based equivalence check.
- **Graph persistence.** Nodes, MENTIONS links and $\sim$5 000 LLM-derived relation edges are written once to a local Neo4j 5.27 instance. Two HNSW indexes (`Chunk` / `LLMEntity`) enable vector search inside Cypher.
- **Retrieval pipelines.** Three strategies share a common gpt-4.1-mini answer generator:

  (i) a vector-only baseline,

  (ii) GraphRAG anchored on NER entities, and

  (iii) GraphRAG anchored on LLM entities with a one-hop expansion that already captures simple multi-hop semantics.

- **Design choices.** Key justifications include selecting `gpt-4.1-mini` for cost-balanced structured output, Neo4j over Cosmos DB for native vector search, and postponing deeper multi-hop traversal to future work.

The methodology yields a reproducible, modular pipeline whose components (thresholds, embedding models, hop depth) can be swapped without altering the overall architecture. The next chapter translates this blueprint into concrete code, highlighting implementation nuances, performance observations and operational tooling used during evaluation.

Chapter 3

Implementation

This chapter translates the architectural blueprint outlined in Chapter 2 into working code. It concentrates on the non-trivial *engineering tasks— entity resolution, graph construction, hybrid retrieval, and the command–line interface that orchestrates them—while routine matters (e.g. setting up a Python environment) are only touched upon briefly.*

3.1 Implementation Overview

Purpose and scope. Chapter 2 formalized *what* the pipeline should do. The present chapter documents *how* those requirements were met in practice: the concrete data structures, Cypher queries, prompt templates, and control flow that together realize a knowledge-graph-centred question–answering system. Only implementation choices that required design trade-offs or non-obvious solutions are discussed in depth; boiler-plate set-up code is omitted for brevity and can be inspected in the accompanying repository.

Technology stack. The prototype is written in **Python 3.12** and relies on:

- **Neo4j 5.27.0** (community edition) with the APOC and GenAI plug-ins for labelled-property storage and built-in HNSW vector search; (27)
- **OpenAI gpt-4.1-mini** for both relation extraction (schema-constrained) and answer generation;
- **OpenAI `text-embedding-3-large`** for 3 072-dimensional embeddings used in similarity search and deduplication;
- **LangChain** for prompt management and structured output validation; (19)

- standard scientific-Python libraries (`numpy`, `pandas`) plus `PyMuPDF` for PDF and excel parsing, and numerical computations such as cosine similarity.

The graph database runs locally in Docker; all LLM and embedding calls are served by remote Azure OpenAI endpoints.

Methodology versus implementation. Whereas the previous chapter described the pipeline conceptually, the sections that follow highlight:

(i) adaptations made when translating the design into code (e.g. batching strategies to respect API rate limits),

(ii) pragmatic decisions that influence performance or cost (choice of gpt-4.1-mini over the larger model),

(iii) auxiliary tooling—CLI commands, caching layers, logging—that make the system reproducible and testable.

Repetitions of already-stated algorithms are avoided; the focus is on implementation nuance rather than theoretical recap.

Chapter roadmap.

Section 3.2 details the construction of the Neo4j knowledge graph from PDF chunks, including schema-compliant Cypher batches and offline vector embedding.

Section 3.3 explains the full question–answering pipeline: graph and vector retrieval, deduplication, prompt assembly, and answer synthesis.

Section 3.4 presents the command-line interface that exposes five operational modes (`build`, `demo`, `interactive`, `eval`), `visualize` for reproducible experimentation.

Section 3.5 summarizes the key engineering insights and paves the way for the quantitative evaluation that follows in Chapter 4.

3.2 Knowledge-Graph Construction from Documents

The `build` sub-command of the CLI converts the thirty seven source PDFs into a Neo4j property graph enriched with vector embeddings. This section documents the *practical mechanics*—file–system cache, Cypher upserts, and embedding upload—rather than repeating the high-level pipeline that was already covered in Sections 2.2–2.6.

3.2.1 Disk Cache for Deterministic Re-runs

All expensive remote calls (LLM extraction, embeddings, equivalence checks) are memoised under `.kgcache/`:

***stem*`.chunks.json`** the list of pre-computed 500-char chunks for a single PDF;

***stem*`.nlp.json`** Czech-NER output: per-chunk entities;

***stem*`.llm.json`** gpt-4.1-mini output: per-chunk entities *and* relations;

`llm_dedup_result.json` corpus-level artefacts after entity deduplication; (canonicalized chunks and relations).

A cache entry is used only if its modification time is newer than the source PDF, keeping the logic stateless and safe to re-run. The helper module shown in Listing 3.1 handles (de)serialization of `pydantic` objects while hiding the underlying JSON.

```python
# abridged from src/knowledge_graph_creation/pipelines/cache_utils.py
def read_entities_and_rels(pdf: Path, tag: str):
    """tag  {'nlp','llm'}"""
    p = entity_path(pdf, tag)
    if p.exists() and p.stat().st_mtime > pdf.stat().st_mtime:
        data = json.loads(p.read_text('utf-8'))
        return deserialise_chunks(data["chunk_entities"]), [
            Relationship(**r) for r in data["relations"]
        ]
    return None
```

Code listing 3.1 Cache loader used by `build_graph()`.

With the cache populated, a full `build` run drops from ≈45 min (cold) to under five minutes, the remainder being spent in Neo4j writes and embedding generation.

3.2.2 Upserting Nodes and Edges

The loader walks the cached structures and executes three Cypher loops inside a single write transaction:

(a) insert `(:Chunk)` nodes;

(b) insert `(:NLPEntity)` or `(:LLMEntity)` nodes plus MENTIONS edges;

(c) insert LLM-derived relation edges.

Unlike a bulk `UNWIND`, the current implementation issues *one Cypher query per row*. Although sub-optimal for very large graphs, this decision simplifies error handling and kept the code base compact. Future work could replace the

```python
def _upsert_branch(tx, chunk_entities: list[dict], label: str) -> None:
    q_chunk = (
        "MERGE (c:Chunk {id:$cid}) "
        "ON CREATE SET c.content=$text "
        "ON MATCH  SET c.content=coalesce(c.content,$text)"
    )
    q_ent   = f"MERGE (e:`{label}` {{name:$name}}) SET e.type=$etype"
    q_link  = (
        f"MATCH (e:`{label}` {{name:$name}}), (c:Chunk {{id:$cid}}) "
        "MERGE (e)-[:MENTIONS]->(c)"
    )

    for idx, ce in enumerate(chunk_entities):
        cid = f"chunk_{idx}"
        tx.run(q_chunk, cid=cid, text=ce["chunk"])
        for ent in ce["entities"]:
            tx.run(q_ent,  name=ent.name, etype=ent.type)
            tx.run(q_link, name=ent.name, cid=cid)
```

Code listing 3.2 Actual helper used by the loader (non-batch, yet idempotent).

inner `for`-loops with batched `UNWIND` statements if ingestion speed becomes a bottleneck.

Because every query employs `MERGE`, rerunning `--mode build` is safe: existing nodes are updated rather than duplicated.

3.2.3 Embedding Upload and Index Provisioning

Embeddings are generated *once* per unique text using the remote `text-embedding-3-large` endpoint. A small helper checks for the presence of the corresponding HNSW index and creates it if missing. Because index creation is idempotent, subsequent `build` runs skip this step entirely.

Key Take-aways

- A lightweight JSON cache eliminates repeat LLM calls, making iterative development practical.
- Although the current Cypher upsert logic is row-wise, it remains idempotent and easy to reason about; batching is a clear avenue for future optimisation.
- All heavy computation (LLM, embeddings) is performed off-line; Neo4j is used *only* for storage and similarity search, keeping database CPU usage negligible.

3.3 Question–Answering Retrieval Pipeline

Figure 3.1 places the *three* retrieval strategies inside the full request–response loop. All paths share the same post-retrieval stages—deduplication, prompt

assembly, and gpt-4.1-mini answer generation—but differ in *how candidate chunks are surfaced*. The next subsections dissect those differences.

3.3.1 Vector-only RAG (baseline)

(a) **Embed question.** 3 072-d vector via `text-embedding-3-large`.

(b) **Anchor search.** A K-NN Cypher query over the `Chunk.embedding` HNSW index (Listing 3.3) returns chunks whose cosine similarity is ≥ 0.70; at most five are kept.

(c) **Proceed to prompt assembly** (Section 3.3.4).

```
MATCH (c:Chunk)
WHERE vector.similarity.cosine(c.embedding, $qvec) >= 0.70
RETURN c.id, c.content
ORDER BY similarity DESC
LIMIT 5
```

Code listing 3.3 Cypher used by the vector-only baseline.

3.3.2 GraphRAG–NLP (NER anchors)

(a) **Entity spotting.** The Czech NER extractor tags the question; each surface form is embedded and searched against the `NLPEntity` vector index. Up to five distinct anchor nodes are returned.

(b) **Chunk collection.** For every anchor e the query pulls chunks mentioned directly by e:

$$(e) \xleftarrow{\text{MENTIONS}} (c{:}\ \mathit{Chunk})$$

No inter-entity edges exist on the NLP layer, so the traversal stops here.

(c) **Re-rank & fallback.** Duplicate `chunk_id`s are collapsed; the survivors are re-ranked by cosine similarity to the *question* vector and the top five kept. If no chunk survives, the strategy falls back to the vector baseline.

(d) **Continue with prompt assembly** (Section 3.3.4).

3.3.3 GraphRAG–LLM (LLM anchors)

(a) **Entity spotting.** gpt-4.1-mini extracts entities from the question (same schema as in Section 2.3.2).

(b) **Anchor search.** Each extracted name is embedded and looked up in `LLMEntity.embedding`; five closest anchors are kept (Cypher in Listing 3.4).

(c) **One-hop expansion.** For every anchor e the query collects *(i)* chunks mentioned directly by e and *(ii)* chunks reachable through **one** semantic edge to a neighbor e_2:

$$(e) - [: \text{REL}] - (e_2) \xleftarrow{\text{MENTIONS}} (c{:}\ \mathit{Chunk})$$

Typical pattern: *Faculty* → *is_governed_by* → *Dean*; chunks mentioning either entity are returned.

(d) **De-dup & re-rank.** Duplicate chunks are collapsed; the remainder is re-ranked against the question vector and the five best are passed to the prompt builder. An empty result triggers a fallback to the vector baseline.

```
UNWIND $anchorVecs AS v
CALL db.index.vector.queryNodes('llmentity_embedding_idx', 5, v)
YIELD node AS e                          // anchors
WITH DISTINCT e, $qvec AS qvec

OPTIONAL MATCH (e)-[:MENTIONS]->(c1:Chunk)
OPTIONAL MATCH (e)-(:LLMEntity)-[:MENTIONS]->(c2:Chunk)
WITH collect(DISTINCT c1)+collect(DISTINCT c2) AS cand, qvec

UNWIND cand AS c
WITH DISTINCT c,
     vector.similarity.cosine(c.embedding, qvec) AS score
ORDER BY score DESC
LIMIT 5
RETURN c.id AS chunk_id,
       c.content AS chunk_content,
       score
```

Code listing 3.4 Cypher fragment used by the LLM-anchored strategy.

3.3.4 From Chunks to Answer

The selected chunks are concatenated with the user question and fed into gpt-4.1-mini via the fixed template in Listing 3.5.

```
You are an assistant that answers questions from the provided context.

Context:
{{ concatenated chunks }}

Question: {{ user_question }}

Answer *only* from the context. If it is insufficient, answer:
"Nemám dostatek informací v kontextu." (Always answer in Czech)
```

Code listing 3.5 Prompt template used for all three retrieval strategies.

3.3.5 Summary of Retrieval Logic

- **Vector baseline** is fast and model-agnostic but ignores graph structure.
- **NER-anchored GraphRAG** leverages canonical entities, yet struggles with broader reasoning, since there are no relations between NLPEntities.
- **LLM-anchored GraphRAG** broadens recall and captures simple relation reasoning at the cost of extra LLM calls.

All three strategies are exposed byF `Neo4jQuestionAnswerer` and can be toggled at run-time via the `--strategy` CLI flag (Section 3.4).

3.4 Command-Line Interface and Tooling

The prototype is packaged as an installable Python project managed by **uv** (28). After cloning the repository you create an isolated environment and install every recorded dependency with two commands:

```
uv venv
source .venv/bin/activate
uv sync                      # installs according to uv.lock
```

All subsequent actions are run through the same entry module:

```
uv run -m knowledge_graph_creation.main --mode <MODE> [flags]
```

3.4.1 Supported modes

3.4.2 Dispatcher skeleton

Listing 3.6 shows the condensed `argparse` driver. Only the mode flag is inspected; helper functions in the earlier sections perform the substantive work.

3.4.3 Usage examples

```
# 1 - build the knowledge graph from all PDFs
uv run -m knowledge_graph_creation.main --mode build \
        --pdf_glob "data/*.pdf"

# 2 - quick demo with LLM-anchored retrieval
uv run -m knowledge_graph_creation.main --mode demo \
        --strategy llm

# 3 - batch evaluation with RAGAS (recommended)
uv run -m knowledge_graph_creation.main --mode eval \
        --excel data/evaluation/eval_dataset.xlsx --generate
```

The CLI thus offers a single, reproducible entry-point: every experiment reported in Chapter 4 can be recreated by invoking one of the commands above.

Mode	Purpose	Principal flags
`build`	Construct and embed the knowledge graph from PDFs	`--pdf_glob "data/.pdf"`
`demo`	Print five hardcoded QA pairs from sample questions	`--strategy={rag,llm,nlp}`
`interactive`	Interactive REPL for free-form QA	`--strategy={rag,llm,nlp}`
`eval`	Evaluate answers in a CSV (RAGAS or DeepEval)	`--csv qa_merged.csv` `--evaluator={ragas, deepeval}` `--regenerate` — overwrite cached answers
`visualize`	Generate PDF plots from evaluated CSV	`--viz_csv _answers_evaluated.csv` `--viz_out outdir` (optional)

Table 3.1 Supported command-line modes and flags

3.5 Chapter Summary

This chapter has translated the conceptual pipeline of Chapter 2 into a concrete, *fully–reproducible* code base.

- **Graph construction.** Eight machine-readable PDFs are chunked, dual-extracted (Czech NER & gpt-4.1-mini), semantically deduplicated, and written to Neo4j through idempotent Cypher upserts. All costly remote calls are cached to disk, turning a 45-minute cold run into a five-minute incremental rebuild.
- **Hybrid retrieval.** A single class — `Neo4jQuestionAnswerer` — implements three retrieval strategies: a vector baseline, an NER-anchored GraphRAG, and an LLM-anchored GraphRAG with one-hop relation expansion.
- **Prompting & generation.** Retrieved chunks are injected into a fixed, schema-agnostic prompt; gpt-4.1-mini either answers in Czech or returns the fallback sentence *"Nemám dostatek informací v kontextu."* (I don't have enough information in the context).
- **Tooling.** The entire workflow is exposed by a five-mode CLI (`build`, `demo`, `interactive`, `eval`), (visualize) and packaged with **uv** (28) for one-command environment replication (`uv venv` → `uv sync`). Evaluation mode supports either RAGAS (default) or DeepEval via `--evaluator`, ensuring metric parity with the results reported later.

```
parser = argparse.ArgumentParser("knowledge-graph")
parser.add_argument("--mode",
    choices=("build", "demo", "interactive", "eval"),
    default="demo")
parser.add_argument("--strategy",
    choices=("rag", "llm", "nlp"), default="llm")
parser.add_argument("--pdf_glob", default="data/*.pdf")
parser.add_argument("--excel",
    help="Evaluation sheet (.xlsx) for --mode eval")
parser.add_argument("--generate", action="store_true")
parser.add_argument("--evaluator",
    choices=("ragas", "deepeval"), default="ragas")

args = parser.parse_args()
if args.mode == "build":
    build_graph(args.pdf_glob)
elif args.mode == "demo":
    _demo(args.strategy)
elif args.mode == "interactive":
    _interactive(args.strategy)
elif args.mode == "eval":
    if not args.excel:
        parser.error("--excel is required for mode=eval")
    _run_evaluation(args)
```

Code listing 3.6 Core of `knowledge_graph_creation.main`.

With the implementation complete and every core component callable from the command line, the stage is set for empirical assessment. Chapter 4 benchmarks the three retrieval strategies on a held-out question set, compares evaluator outputs, and discusses the trade-offs uncovered during testing.

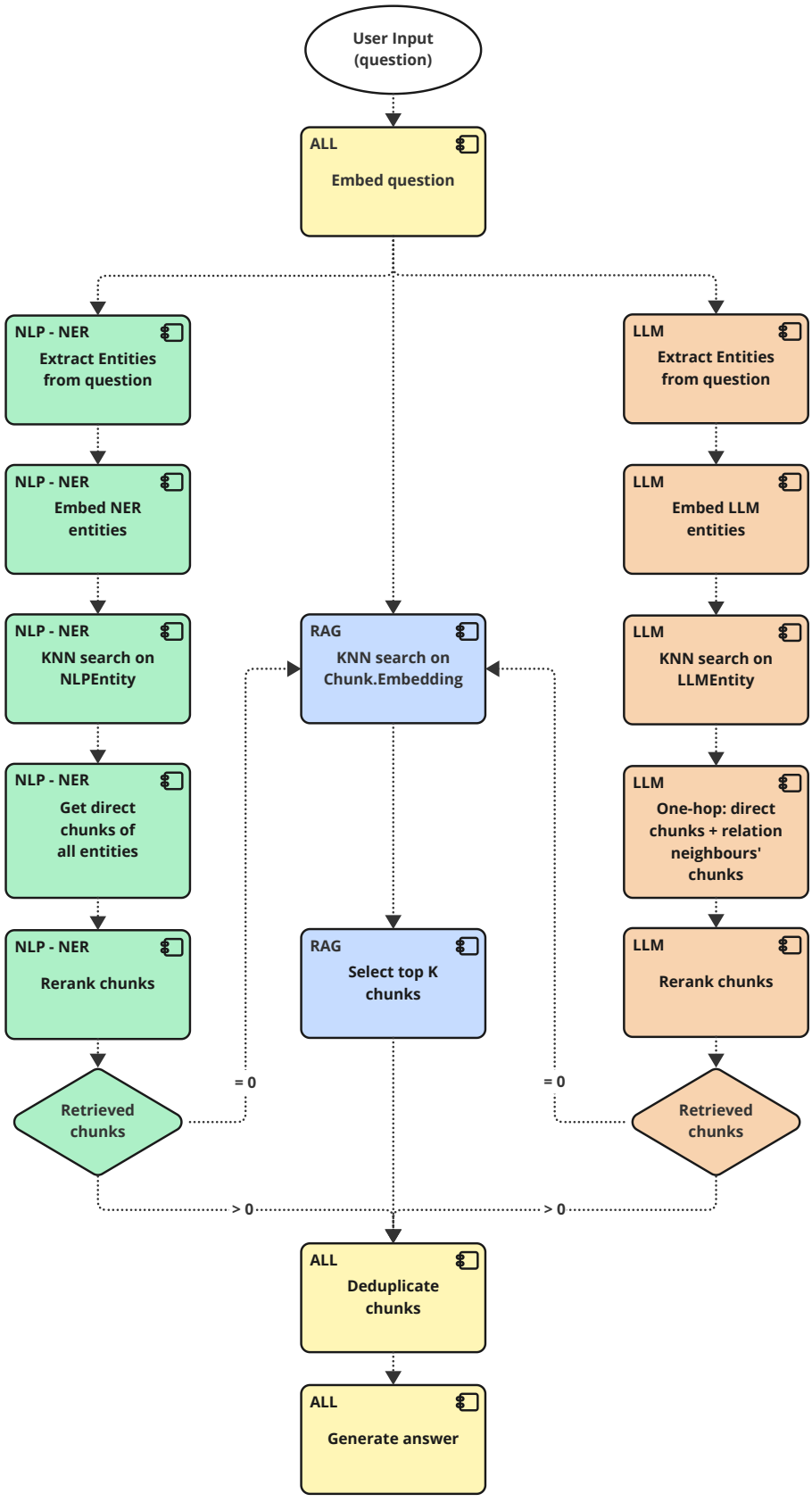

Figure 3.1 Run-time flow of the question–answering module. Yellow blocks are shared utilities; blue, green and salmon lanes are specific to Vector, NLP-Graph and LLM-Graph retrieval respectively.

Chapter 4

Results and Evaluation

This chapter benchmarks three retrieval–augmented QA pipelines on Czech-language university regulations: a dense-vector RAG baseline, a GraphRAG variant anchored on LLM-extracted *entities, and a GraphRAG variant anchored on* Czech-specific NER *entities. Performance is quantified with the reference-free RAGAS metric suite, allowing a multi-faceted comparison of retrieval quality, answer faithfulness, and factual correctness.*

4.1 Evaluation Objectives and Scope

This chapter provides a systematic comparison of three retrieval strategies: (i) a baseline dense-vector RAG approach, (ii) GraphRAG$_{\text{LLM}}$, and (iii) GraphRAG$_{\text{NER}}$. The goal is to determine how each retrieval mechanism influences answer quality and reliability when answering natural-language questions over a corpus of Czech university documents. Evaluation relies on the RAGAS framework (29), which scores every answer on context relevance, faithfulness to the retrieved evidence, and factual correctness. Comparing these metrics across strategies reveals which approach best balances precise retrieval with accurate, well-grounded generation.

4.2 Evaluation Process and Methodology

All three retrieval pipelines were run on an identical question set to ensure a fair, head-to-head comparison. For each query the experiment followed three deterministic steps:

1. **Retrieval**: passages were retrieved with the selected strategy.
2. **Generation**: a single, fixed gpt-4.1-mini prompt generated the answer; no prompt engineering or fine-tuning varied across strategies.

3. **Evaluation**: the question, retrieved context, and system answer were passed to RAGAS, which returned scores for *context precision*, *context recall*, *faithfulness*, *answer relevancy*, and *answer correctness*.

Because each question is accompanied by a reference answer derived from the source documents, the `answer_correctness` metric variant that explicitly compares the generated answer to ground truth was activated. All scores stem from this single, uniform evaluation prompt, ensuring metric parity across strategies. The resulting per-question scores are aggregated in the following sections to compare the dense-vector baseline with the two GraphRAG variants.

4.3 Evaluation Metrics

To capture both retrieval quality *and* answer quality we adopt the five metrics implemented in the RAGAS library(30). Each metric returns a score in the closed interval $[0, 1]$, where larger values uniformly denote better performance.

4.3.1 Context Precision

Let $C = \langle c_1, \ldots, c_K \rangle$ be the (ranked) list of passages returned by the retriever and let $R \subseteq C$ be the (latent) subset that is actually relevant to the query. Denote by $\mathbf{1}_{\{k \in R\}} \in \{0, 1\}$ an indicator that passage c_k is relevant. RAGAS defines[1]

$$\begin{aligned} \text{Precision}_k &= \frac{\text{TP}_k}{\text{TP}_k + \text{FP}_k}, \\ \text{ContextPrecision}_K &= \frac{1}{K} \sum_{k=1}^{K} \text{Precision}_k \, \mathbf{1}_{\{k \in R\}}. \end{aligned} \tag{4.1}$$

where TP_k (FP_k) counts the relevant (respectively irrelevant) passages among the first k items of C(31). A high score indicates that little extraneous text was retrieved; any *additional* still-relevant passages are rewarded by recall rather than precision.

4.3.2 Context Recall

Context recall measures how completely the retrieval step recovered the evidence required for a correct answer. It is approximated by

$$\text{ContextRecall} \;=\; \frac{|C \cap R|}{|R|} \quad \in [0, 1],$$

[1] The implementation averages over K because most RAG systems use a fixed top–K cut-off during evaluation.

where missing relevant passages ($|R \setminus C|$) are detected by decomposing the ground-truth answer into atomic claims and asking an LLM whether each claim is supported by *some* passage in C(32). A low recall thus flags omissions that may cause downstream errors even when precision is high.

4.3.3 Faithfulness

Given the set of factual statements $A = \{a_1, \ldots, a_m\}$ extracted from the generated answer, and the retrieved context C, the faithfulness score is

$$\text{Faithfulness} \;=\; \frac{|\{\, a_i \mid a_i \text{ is entailed by } C \,\}|}{m} \quad \in [0,1],$$

where "entailed by C" is decided by an LLM verifier(33). The metric penalizes hallucinated claims; an answer can be perfectly faithful yet incomplete if recall is low.

4.3.4 Answer Relevancy

Answer relevancy quantifies how directly the generated answer a addresses the user question q, independent of factual accuracy. RAGAS constructs N synthetic questions $g_1, \ldots, g_N$ by *back-prompting* an LLM on the answer and then averages their semantic similarity to the original question:

$$\text{AnswerRelevancy} \;=\; \frac{1}{N}\sum_{i=1}^{N} \cos\!\big(\mathbf{E}_{g_i}, \mathbf{E}_q\big),$$

where $\mathbf{E}_q$ and $\mathbf{E}_{g_i}$ are sentence embeddings of q and g_i, and $N = 3$ by default(34). Higher scores signal that the answer is on-topic and complete.

4.3.5 Answer Correctness

When a canonical reference answer g is available, correctness is computed as an F_1-style overlap of factual statements:

$$\text{AnswerCorrectness} \;=\; \frac{|\text{TP}|}{|\text{TP}| + 0.5\big(|\text{FP}| + |\text{FN}|\big)},$$

where TP = facts present in both a and g, FP = facts only in a, FN = facts only in g(35). This metric rewards factual agreement even when the verbal surface form differs, complementing faithfulness (grounding) and relevancy (focus).

Interpretation. Together, these five metrics disentangle:

- **Retrieval performance**: context precision & recall,

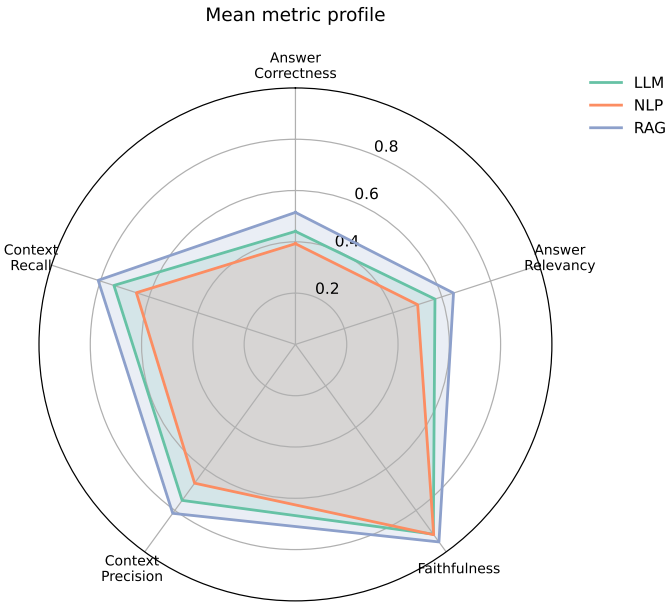

Figure 4.1 Radar plot with mean value of every metric.

- **Grounding quality**: faithfulness,
- **Topical focus**: answer relevancy,
- **Factual accuracy**: answer correctness.

The following sections analyse their empirical distributions to expose the strengths and weaknesses of the dense-vector baseline versus the two GraphRAG variants.

4.4 Aggregate Results per Metric

The evaluation runs 141 questions on a corpus created from 37 public ČVUT regulations (1 328 chunks, 1 790 LLM–entities, 611 NER–entities, 7 590 edges), resulting in 2115 evaluations (questions * strategies * metrics). Figures 4.1–4.4 visualize the five RAGAS metrics for the dense baseline (**rag**), the LLM-anchored graph (**llm**) and the NER-anchored graph (**nlp**).

Key observations.

(a) **Baseline dominates on averages.** Figure 4.1 shows the dense RAG has the highest *mean* score on *all five* metrics.

(b) **LLM-graph edges ahead on median relevancy.** In Figure 4.2 the median of *answer relevancy* is slightly higher for **llm** than for the baseline; on every other metric its median trails.

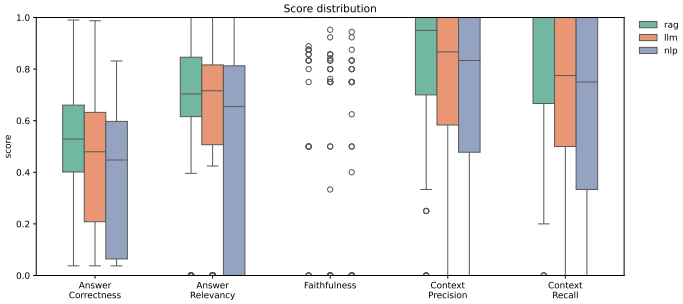

Figure 4.2 Full score distributions (median, IQR, 1.5 × IQR).

(c) **NER-graph is consistently weakest.** Both mean and median values are lowest for **nlp**—expected, because the current KG stores only MENTIONS edges, no typed relations.

(d) **Variance tells a cautionary story.** **llm** exhibits the widest IQR on precision and correctness; noisy or generic entity anchors occasionally lead the retriever off–topic. The baseline shows the tightest spreads overall.

(e) **Error-coupling patterns.** Heat-map (Figure 4.3) confirms that both graph variants mostly lose ground to the baseline; the biggest drop is correctness for **nlp**. Correlation plots (Figure 4.4–4.6) highlight that correctness tracks recall more strongly than faithfulness: missing evidence hurts more than hallucination.

In short, the dense vector retriever still provides the best end-to-end factual accuracy. LLM-anchored GraphRAG can improve topicality for some questions but at the cost of higher variance; the current NER-graph offers no net gain and often degrades performance because its structure is too shallow.

4.5 Discussion

The quantitative evaluation reported in section 4.4 crowns the **dense RAG baseline** as the current winner. Yet the goal of this thesis was broader: to prove that *knowledge-graph construction from Czech PDFs is feasible* and to pave the way for richer retrieval policies. From that vantage-point the project is still a success. Key contributions are summarized below.

- **Retrieval—not representation—is the bottleneck.**
  The graph contains all passages present in the vector store *and* extra structure. Lower scores therefore stem from the first-round, embedding-only graph search. Future work—agentic traversal, hybrid re-ranking, learned path scoring—could unlock the additional signal.

- **Turn-key ingestion and QA interface.**
  The pipeline reads raw PDFs, splits text, extracts entities (LLM or Czech NER), deduplicates them, predicts relations and persists everything to Neo4j. A command-line chatbot then lets users compare three retrieval modes with a single flag.

- **Extensible evaluation harness.**
  The RAGAS wrapper accepts any spreadsheet with `question`, `ground_truth` and result columns. Adding a new retriever or metric is a *simple change* (one function call), making the setup reusable for future experiments.

- **Practical entity-resolution.**
  Duplicate surface forms are collapsed with a union-find–style algorithm that combines *embedding cosine similarity* with *selective LLM comparisons*. A manual audit shows the graph now contains far fewer false duplicates, while precision remains high—illustrating that purely statistical clustering can be strengthened with light-weight LLM checks.

- **Empirical insight into Czech IE.**
  Entity recognition works with both LLMs and classical NLP, but robust *relation* extraction for Czech is still missing. A hybrid route—NER for entities, LLM function-calling for relations—looks promising, though cost savings are uncertain.

Take-away. Dense retrieval is tough to beat on a coherent Czech corpus, but the software artefacts delivered here—a KG builder, a tri-modal chatbot, an evaluation kit and a deduplication module—set the stage for smarter graph-aware retrieval that could convert structural knowledge into higher factual accuracy in future iterations.

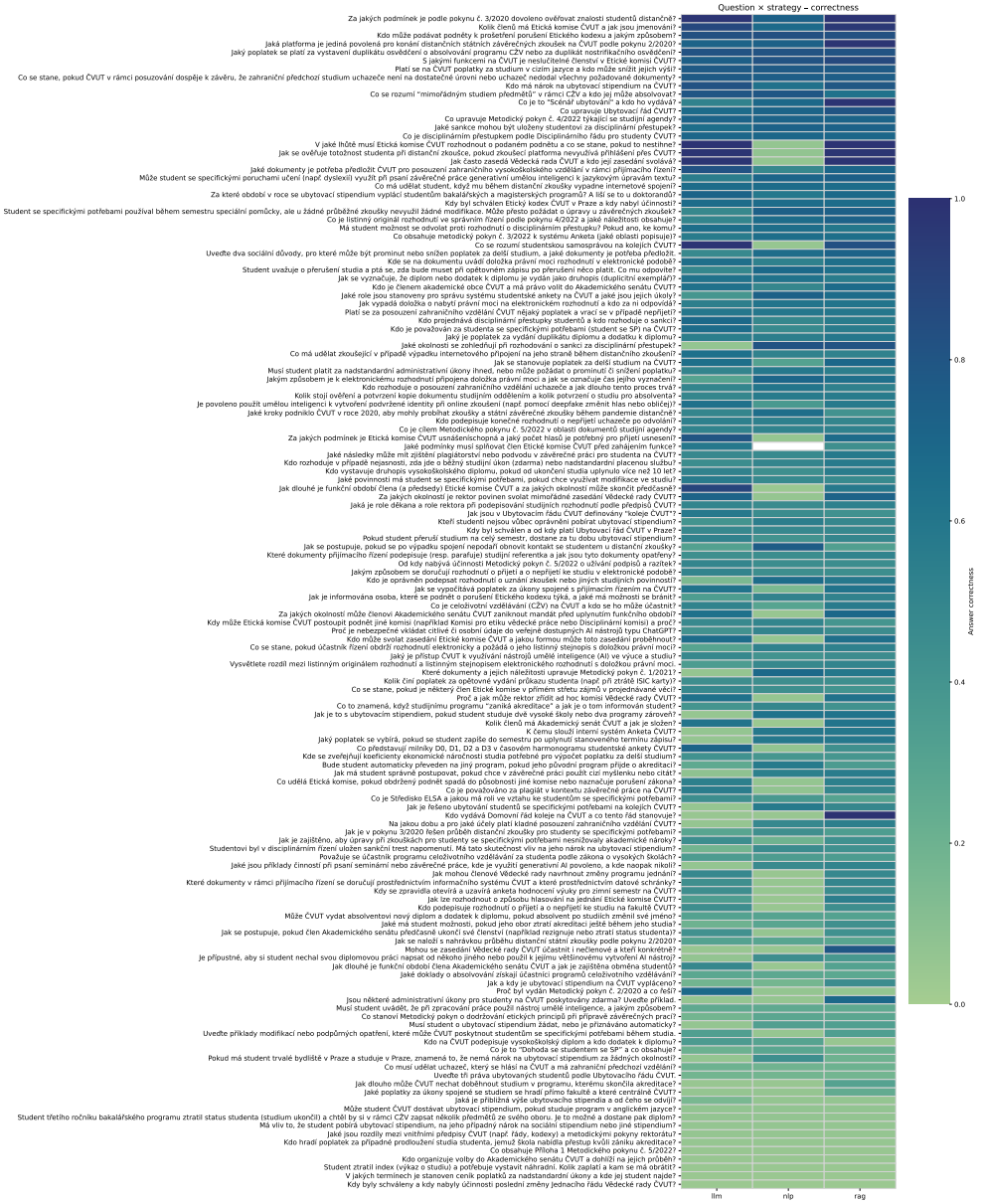

Figure 4.3 Relative % difference of each graph variant to the baseline (blue = better, green = worse).

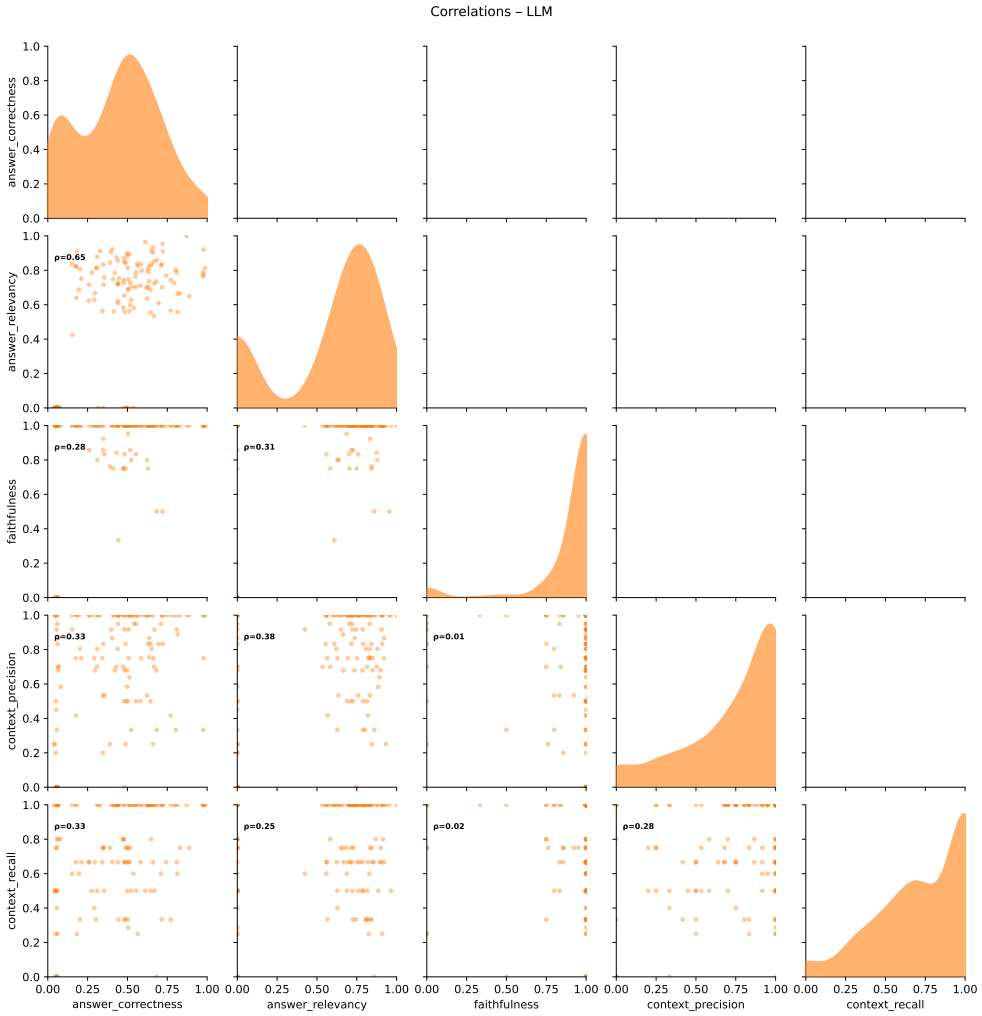

Figure 4.4 LLM anchored PairGrid correlation plot.

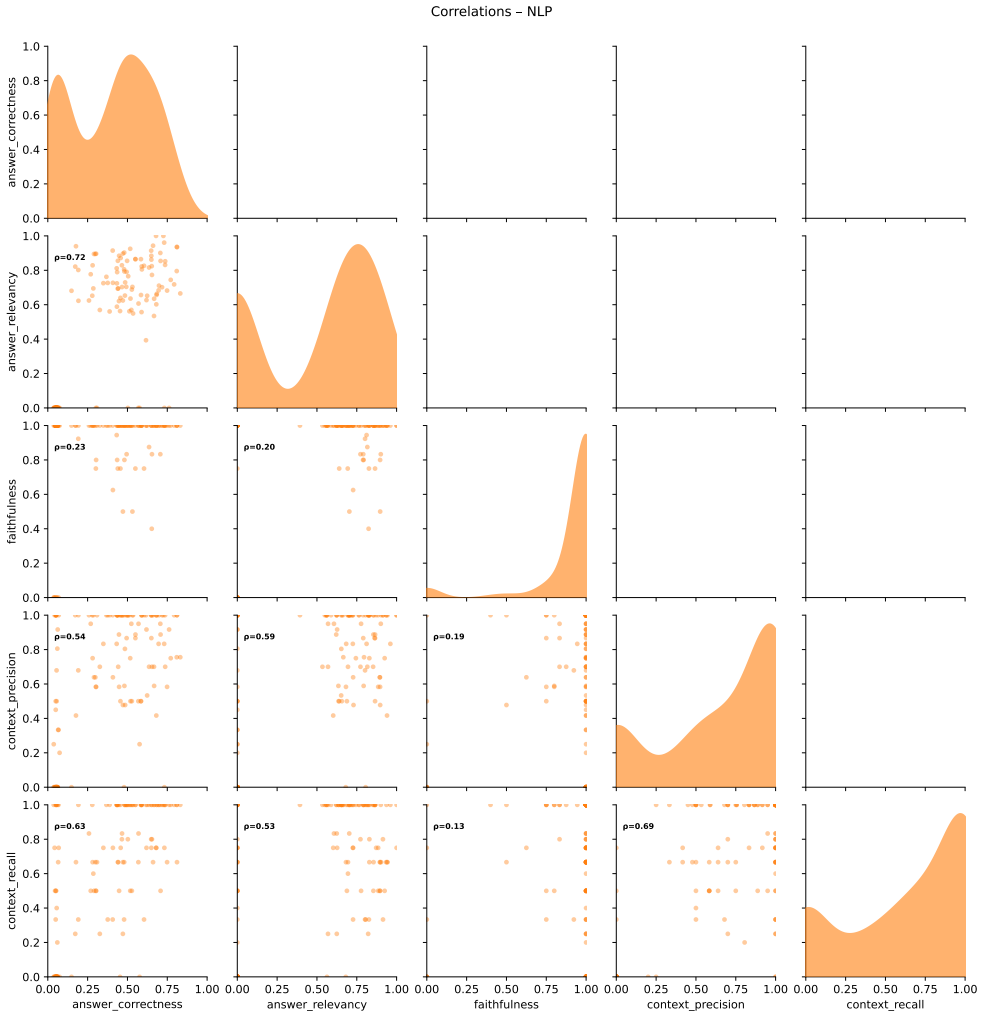

Figure 4.5 NER anchored PairGrid correlation plot.

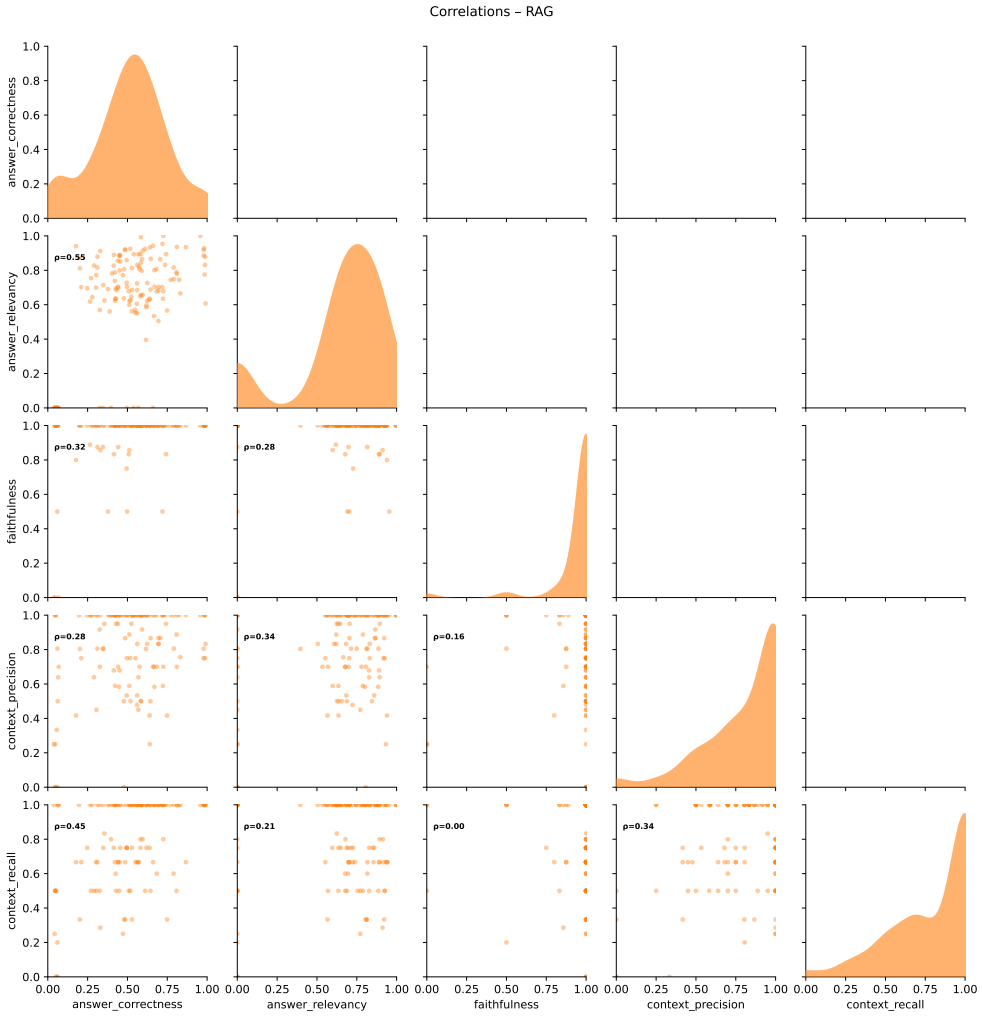

Figure 4.6 RAG baseline PairGrid correlation plot.

Chapter 5

Conclusion

This thesis examined whether introducing an explicit knowledge graph can improve question answering over Czech university regulations. The work combined traditional natural–language processing with large–language–model techniques and evaluated their impact on a retrieval-augmented QA pipeline. The main achievements and findings are summarized below.

1. **End-to-end pipeline.** A fully automated workflow was implemented that ingests 37 ČVUT documents, splits them into 1328 fixed-length chunks, extracts entities and relations by a Czech RoBERTa NER model and by gpt-4.1-mini in structured-output mode, merges duplicate entities via cosine similarity and an LLM equivalence check, and stores the resulting entities and relations in a Neo4j property graph equipped with vector indexes.

2. **Entity-resolution method.** Duplicate surface forms produced by the LLM extractor were clustered with a union–find algorithm that first proposes candidates by embedding similarity and then confirms identity through a lightweight LLM call.

3. **Three retrieval strategies.** The graph supports a dense-vector baseline (RAG), GraphRAG anchored on LLM entities, and GraphRAG anchored on NER entities. All three share an identical gpt-4.1-mini answer generator and a uniform prompt.

4. **Quantitative evaluation.** For 141 held-out questions each (strategy, answer) pair was scored by the RAGAS metric suite—context precision, context recall, faithfulness, answer relevancy, and correctness—yielding more than two thousand individual measurements.

5. **Results.** The dense-vector baseline achieved the highest *mean* score on every metric. The LLM-anchored graph attained a slightly higher

median answer-relevancy, suggesting potential once retrieval is improved. The analysis indicates that the present bottleneck is the retrieval policy rather than the knowledge-graph representation itself.

6. **Reproducible tooling.** All components—graph building, interactive chat, and metric evaluation—are exposed through a single command-line interface. Adding a new retriever or metric requires only a minor code change, facilitating future experiments.

Overall, the thesis demonstrates that constructing a knowledge graph from Czech legal and administrative texts is both feasible and operationally useful. Although the initial graph-augmented retrievers did not yet surpass a well-tuned dense baseline in overall accuracy, the graph provides structured evidence that can be exploited by more advanced retrieval policies. The delivered pipeline, evaluation framework, and empirical insights lay a solid foundation for future work on graph-aware search, multi-hop reasoning, and improved information extraction for morphologically rich languages such as Czech.

Bibliography

1. HOGAN, Aidan; BLOMQVIST, Eva; COCHEZ, Michael; D'AMATO, Claudia; MELO, Gerard De; GUTIERREZ, Claudio; KIRRANE, Sabrina; GAYO, José Emilio Labra; NAVIGLI, Roberto; NEUMAIER, Sebastian; NGOMO, Axel-Cyrille Ngonga; POLLERES, Axel; RASHID, Sabbir M.; RULA, Anisa; SCHMELZEISEN, Lukas; SEQUEDA, Juan; STAAB, Steffen; ZIMMERMANN, Antoine. Knowledge Graphs. *ACM Computing Surveys*. 2021, vol. 54, no. 4, pp. 1–37. ISSN 1557-7341. Available from DOI: 10.1145/3447772.
2. ETZIONI, Oren; CAFARELLA, Michael; DOWNEY, Doug; POPESCU, Ana-Maria; SHAKED, Tal; SODERLAND, Stephen; WELD, Daniel S.; YATES, Alexander. Unsupervised named-entity extraction from the Web: An experimental study. *Artificial Intelligence*. 2005, vol. 165, no. 1, pp. 91–134. ISSN 0004-3702. Available from DOI: https://doi.org/10.1016/j.artint.2005.03.001.
3. YATES, Alexander; BANKO, Michele; BROADHEAD, Matthew; CAFARELLA, Michael; ETZIONI, Oren; SODERLAND, Stephen. TextRunner: Open Information Extraction on the Web. In: CARPENTER, Bob; STENT, Amanda; WILLIAMS, Jason D. (eds.). *Proceedings of Human Language Technologies: The Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*. Rochester, New York, USA: Association for Computational Linguistics, 2007, pp. 25–26. Available also from: https://aclanthology.org/N07-4013/.
4. YADAV, Vikas; BETHARD, Steven. A Survey on Recent Advances in Named Entity Recognition from Deep Learning models. In: BENDER, Emily M.; DERCZYNSKI, Leon; ISABELLE, Pierre (eds.). *Proceedings of the 27th International Conference on Computational Linguistics*. Santa Fe, New Mexico, USA: Association for Computational Linguistics, 2018, pp. 2145–2158. Available also from: https://aclanthology.org/C18-1182/.

5. ZHAO, Xiaoyan; DENG, Yang; YANG, Min; WANG, Lingzhi; ZHANG, Rui; CHENG, Hong; LAM, Wai; SHEN, Ying; XU, Ruifeng. *A Comprehensive Survey on Relation Extraction: Recent Advances and New Frontiers*. 2024. Available from arXiv: 2306.02051 [cs.CL].

6. DEVLIN, Jacob; CHANG, Ming-Wei; LEE, Kenton; TOUTANOVA, Kristina. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: BURSTEIN, Jill; DORAN, Christy; SOLORIO, Thamar (eds.). *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, 2019, pp. 4171–4186. Available from DOI: 10.18653/v1/N19-1423.

7. LIU, Yinhan; OTT, Myle; GOYAL, Naman; DU, Jingfei; JOSHI, Mandar; CHEN, Danqi; LEVY, Omer; LEWIS, Mike; ZETTLEMOYER, Luke; STOYANOV, Veselin. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. Available from arXiv: 1907.11692 [cs.CL].

8. STRAKOVÁ, Jana; STRAKA, Milan; ŠEVČÍKOVÁ, Magda; ŽABOKRTSKÝ, Zdeněk. Czech Named Entity Corpus. In: *Handbook of Linguistic Annotation*. Springer Netherlands, 2017, pp. 855–873. Available from DOI: 10.1007/978-94-024-0881-2_31.

9. ŠTULC, Radek. *Named Entity Recognition in Czech Using Pre-trained Language Models*. 2024. Available also from: https://dspace.cvut.cz/bitstream/handle/10467/115618/F3-BP-2024-Stulc-Radek-NamedEntityRecognition.pdf.

10. ŠTULC, Radek. *CNEC_1_1_Supertypes_robeczech-base*. 2024. Available also from: https://huggingface.co/stulcrad/CNEC_1_1_Supertypes_robeczech-base. Fine-tuned ufal/robeczech-base on CNEC 1.1. F1 score: 86.7%.

11. WADHWA, Somin; AMIR, Silvio; WALLACE, Byron. Revisiting Relation Extraction in the era of Large Language Models. In: ROGERS, Anna; BOYD-GRABER, Jordan; OKAZAKI, Naoaki (eds.). *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Toronto, Canada: Association for Computational Linguistics, 2023, pp. 15566–15589. Available from DOI: 10.18653/v1/2023.acl-long.868.

12. CHEN, Ruirui; JIANG, Weifeng; QIN, Chengwei; RAWAL, Ishaan Singh; TAN, Cheston; CHOI, Dongkyu; XIONG, Bo; AI, Bo. LLM-Based Multi-Hop Question Answering with Knowledge Graph Integration in Evolving Environments. In: AL-ONAIZAN, Yaser; BANSAL, Mohit; CHEN, Yun-Nung (eds.). *Findings of the Association for Computational Linguistics: EMNLP 2024*. Miami, Florida, USA: Association for Computational Lin-

guistics, 2024, pp. 14438–14451. Available from DOI: 10.18653/v1/2024.findings-emnlp.844.

13. BROWN, Tom B.; MANN, Benjamin; RYDER, Nick; SUBBIAH, Melanie; KAPLAN, Jared; DHARIWAL, Prafulla; NEELAKANTAN, Arvind; SHYAM, Pranav; SASTRY, Girish; ASKELL, Amanda; AGARWAL, Sandhini; HERBERT-VOSS, Ariel; KRUEGER, Gretchen; HENIGHAN, Tom; CHILD, Rewon; RAMESH, Aditya; ZIEGLER, Daniel M.; WU, Jeffrey; WINTER, Clemens; HESSE, Christopher; CHEN, Mark; SIGLER, Eric; LITWIN, Mateusz; GRAY, Scott; CHESS, Benjamin; CLARK, Jack; BERNER, Christopher; MCCANDLISH, Sam; RADFORD, Alec; SUTSKEVER, Ilya; AMODEI, Dario. *Language Models are Few-Shot Learners*. 2020. Available from arXiv: 2005.14165 [cs.CL].

14. WEI, Jason; WANG, Xuezhi; SCHUURMANS, Dale; BOSMA, Maarten; ICHTER, Brian; XIA, Fei; CHI, Ed; LE, Quoc; ZHOU, Denny. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. Available from arXiv: 2201.11903 [cs.CL].

15. JURAFSKY, Daniel. *Speech and Language Processing* [https://web.stanford.edu/~jurafsky/slp3/ed3book.pdf]. 2025. [Accessed 09-05-2025].

16. OPENAI. *Structured Output with JSON Schemas*. 2024. Available also from: https://openai.com/index/introducing-structured-outputs-in-the-api/. Accessed: 2025-05-07.

17. CHEN, Danqi; FISCH, Adam; WESTON, Jason; BORDES, Antoine. Reading Wikipedia to Answer Open-Domain Questions. In: BARZILAY, Regina; KAN, Min-Yen (eds.). *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, 2017, pp. 1870–1879. Available from DOI: 10.18653/v1/P17-1171.

18. PIETSCH, Malte; MÖLLER, Timo; KOSTIC, Bogdan; RISCH, Julian; PIPPI, Massimiliano; JOBANPUTRA, Mayank; ZANZOTTERA, Sara; CERZA, Silvano; BLAGOJEVIC, Vladimir; STADELMANN, Thomas; SONI, Tanay; LEE, Sebastian. *Haystack: the end-to-end NLP framework for pragmatic builders*. 2019. Available also from: https://github.com/deepset-ai/haystack.

19. CHASE, Harrison. *LangChain*. 2022. Available also from: https://github.com/langchain-ai/langchain.

20. EXPLODINGGRADIENTS. *Ragas: Supercharge Your LLM Application Evaluations* [https://github.com/explodinggradients/ragas]. 2024.

21. SHAVAKI, Mahdi Amiri; OMRANI, Pouria; TOOSI, Ramin; AKHAEE, Mohammad Ali. Knowledge Graph Based Retrieval-Augmented Generation for Multi-Hop Question Answering Enhancement. In: *2024 15th International Conference on Information and Knowledge Technology (IKT)*. 2024, pp. 78–84. Available from DOI: 10.1109/IKT65497.2024.10892619.

22. RESEARCH, Microsoft. *GraphRAG: Unlocking LLM Discovery on Narrative Private Data*. 2024. Available also from: https://www.microsoft.com/en-us/research/blog/graphrag-unlocking-llm-discovery-on-narrative-private-data/. Accessed 2025-05-09.

23. DEVELOPERS, PyMuPDF. *PyMuPDF: A lightweight PDF and XPS parser*. 2024. Available also from: https://github.com/pymupdf/PyMuPDF. Accessed 2025-05-09.

24. MICROSOFT. *Markitdown*. 2025. Available also from: https://github.com/microsoft/markitdown.

25. ENEVOLDSEN, Kenneth; CHUNG, Isaac; KERBOUA, Imene; KARDOS, Márton; MATHUR, Ashwin; STAP, David; GALA, Jay; SIBLINI, Wissam; KRZEMIŃSKI, Dominik; WINATA, Genta Indra; STURUA, Saba; UTPALA, Saiteja; CIANCONE, Mathieu; SCHAEFFER, Marion; SEQUEIRA, Gabriel; MISRA, Diganta; DHAKAL, Shreeya; RYSTRØM, Jonathan; SOLOMATIN, Roman; ÇAĞATAN, Ömer; KUNDU, Akash; BERNSTORFF, Martin; XIAO, Shitao; SUKHLECHA, Akshita; PAHWA, Bhavish; POŚWIATA, Rafał; GV, Kranthi Kiran; ASHRAF, Shawon; AURAS, Daniel; PLÜSTER, Björn; HARRIES, Jan Philipp; MAGNE, Loïc; MOHR, Isabelle; HENDRIKSEN, Mariya; ZHU, Dawei; GISSEROT-BOUKHLEF, Hippolyte; AARSEN, Tom; KOSTKAN, Jan; WOJTASIK, Konrad; LEE, Taemin; ŠUPPA, Marek; ZHANG, Crystina; ROCCA, Roberta; HAMDY, Mohammed; MICHAIL, Andrianos; YANG, John; FAYSSE, Manuel; VATOLIN, Aleksei; THAKUR, Nandan; DEY, Manan; VASANI, Dipam; CHITALE, Pranjal; TEDESCHI, Simone; TAI, Nguyen; SNEGIREV, Artem; GÜNTHER, Michael; XIA, Mengzhou; SHI, Weijia; LÙ, Xing Han; CLIVE, Jordan; KRISHNAKUMAR, Gayatri; MAKSIMOVA, Anna; WEHRLI, Silvan; TIKHONOVA, Maria; PANCHAL, Henil; ABRAMOV, Aleksandr; OSTENDORFF, Malte; LIU, Zheng; CLEMATIDE, Simon; MIRANDA, Lester James; FENOGENOVA, Alena; SONG, Guangyu; SAFI, Ruqiya Bin; LI, Wen-Ding; BORGHINI, Alessia; CASSANO, Federico; SU, Hongjin; LIN, Jimmy; YEN, Howard; HANSEN, Lasse; HOOKER, Sara; XIAO, Chenghao; ADLAKHA, Vaibhav; WELLER, Orion; REDDY, Siva; MUENNIGHOFF, Niklas. MMTEB: Massive Multilingual Text Embedding Benchmark. *arXiv preprint arXiv:2502.13595*. 2025. Available from DOI: 10.48550/arXiv.2502.13595.

26. FACE, Hugging. *MMTEB Leaderboard*. 2025. Available also from: https://huggingface.co/spaces/mteb/leaderboard. Accessed 2025-05-10.

27. INC., Neo4j. *GenAI integrations - Cypher Manual — neo4j.com* [https://neo4j.com/docs/cypher-manual/current/genai-integrations/]. 2025. [Accessed 10-05-2025].

28. SOFTWARE, Astral. *uv: The next-generation Python package manager* [https://docs.astral.sh/uv/]. 2024. Accessed 2025-05-9.

29. ES, Shahul; JAMES, Jithin; ESPINOSA-ANKE, Luis; SCHOCKAERT, Steven. *Ragas: Automated Evaluation of Retrieval Augmented Generation*. 2025. Available from arXiv: 2309.15217 [cs.CL].

30. CONTRIBUTORS, RAGAS. *RAGAS Documentation* [https://docs.ragas.io/]. 2024. Accessed: 2025-05-14.

31. CONTRIBUTORS, RAGAS. *Context Precision — RAGAS Documentation* [https://docs.ragas.io/en/stable/concepts/metrics/available_metrics/context_precision/]. 2024. Accessed: 2025-05-14.

32. CONTRIBUTORS, RAGAS. *Context Recall — RAGAS Documentation* [https://docs.ragas.io/en/stable/concepts/metrics/available_metrics/context_recall/]. 2024. Accessed: 2025-05-14.

33. CONTRIBUTORS, RAGAS. *Faithfulness — RAGAS Documentation* [https://docs.ragas.io/en/stable/concepts/metrics/available_metrics/faithfulness/]. 2024. Accessed: 2025-05-14.

34. CONTRIBUTORS, RAGAS. *Answer Relevance — RAGAS Documentation* [https://docs.ragas.io/en/v0.1.21/concepts/metrics/answer_relevance.html]. 2024. Accessed: 2025-05-14.

35. CONTRIBUTORS, RAGAS. *Answer Correctness — RAGAS Documentation* [https://docs.ragas.io/en/v0.1.21/concepts/metrics/answer_correctness.html]. 2024. Accessed: 2025-05-14.

Content of the attachment

- `/`
  - `code` ..... source code directory
    - `data` ..... input data for the code
      - `evaluation` ... directory with evaluation datasets with questions
      - `*.pdf` ..... input documents in PDF
    - `src` ..... source codes of the implementation
    - `README.md` ..... instructions for running the code
  - `theory` ..... source code of thesis in LaTeX
    - `main.pdf` ..... thesis in PDF