Task Assignment for the Candidate - Position: AI Engineer

Title: Building a Graph Database and AI Agents for Czech Legislation Knowledge Retrieval

Objective

Design and implement a **modern AI retrieval system** for Czech legislation, combining **graph databases**, **agentic AI workflows**, and **retrieval-augmented generation (RAG)** techniques.

This assignment assesses your **software engineering maturity**, ability to work with **structured and unstructured legal data**, and implement **tool-using AI agents** for hybrid information retrieval. It emphasizes real-world, scalable architecture using new techniques.

The candidate is required to design and implement:

1. **A graph database to structure and store Czech legislative documents from eSbírka.**
2. **A tool-using React AI agent that interprets user queries/intents to solve their needs.**
3. **A preprocessing module to extract and structure legal text from PDF documents in eSbírka.**

Project Scope

You are to build a **backend legal intelligence platform** that extracts structured knowledge from Czech legislative PDFs (eSbírka) and enables AI-driven exploration of this knowledge through:

- A **Graph Knowledge Base** for laws, articles, entities, and legal references.
- An **Agentic AI layer** that interprets user queries/intents to solve their needs.
- A **Retrieval-Augmented Generation (RAG) pipeline** for deep, reference-backed answers.
- An **OpenAPI-compatible backend** with endpoints for querying legislation, graph inspection, and AI Q&A.

Task Breakdown

1. Extracting and Structuring Legal Text from PDFs

Start by turning legislative PDF files from eSbírka into clean, structured data.

What to do:

- Extract plain text from PDFs (using tools like PyMuPDF or pdfplumber).
- Clean the content:
  - Remove noise like headers, footers, and pagination.
- Identify and organize:
  - **Laws**, **Articles**, **Paragraphs**, and **Subsections**
  - **References to other laws** (e.g., "see § 3 of Act No. XYZ")
  - **Effective dates**, **Law IDs**, **Titles**
  - **Enforcing agencies** like ministries or public offices
- Save everything in a structured format like JSON or CSV or directly into the graph database.

Deliverables:

- One clean, structured **JSON file per law**, containing:
  - The law's text
  - Metadata (ID, title, dates, agency, references, structure)

2. Creating a Legal Knowledge Graph (Neo4j)

Turn the structured legal data into a queryable knowledge graph (Neo4j) enhanced with vector search (Qdrant).

What to do:

- Use **Neo4j** to build a **graph database**.
- Create a graph model where:
  - Laws are nodes (e.g., "Digital Identity Act")
  - Articles, paragraphs, and subsections are linked as nested nodes
  - References between laws are edges
  - Agencies are also nodes connected by "enforces" relationships
- Load the structured JSON data into the graph.
- Add vector embeddings.
- Make sure others can reuse your code to seed the database.

Deliverables:

- A description of your **graph schema**
- A running **Neo4j instance** with data preloaded

- Scripts to:
    - Initialize the graph schema
    - Load structured legal data into Neo4j

3. Building a Smart Legal Retrieval Agent (ReAct)

Create an AI agent that can understand user questions and figure out the best way to answer them.

What to do:

- Implement an agent that can:
    - Read a user query (e.g., “What are the requirements for electronic signatures?”)
    - Decide which tools (search strategies) to use:
        - **Graph search** using Cypher (to query Neo4j)
        - **Vector search** using Qdrant vector-based retriever for Neo4j
        - **Hybrid search**: Combine retrieval results from graph search and vector search, using rerankers or any combining techniques.
    - Generate a coherent, truth-grounded, high quality response.
    - Can validate the reply and search again if the reply does not answer the question.
    - Can detect that the user query is not relevant (e.g., “How is the weather today?”) and do not answer, or suggest other questions that are relevant to law.
- Make the agent's **reasoning process visible** (e.g., in debug mode)

Deliverables:

- You can use any existing AI Agent framework, no need to develop from scratch.
- Sample user questions and the **agent’s reasoning trace**
- (Optional) A few curated test cases and results

4. Creating an API for Your System

Wrap everything in an easy-to-use backend API that developers can query.

What to do:

- Build a RESTful API (or OpenAPI-compatible) using:
    - **FastAPI** (Python) or **Fastify** (TypeScript preferred)
- Expose endpoints for:
    - Querying the graph

    - Running vector searches
    - Interacting with the AI agent
- Add features for developers:
    - Swagger UI (for exploring and testing the API)
    - .env config support
    - Docker setup for easy deployment

Deliverables:

- A fully working **REST API** with OpenAPI schema
- A **README** that includes:
    - Local setup instructions
    - How to use each API endpoint
    - A simple **system architecture diagram**

References

Neo4j GraphRAG - Qdrant (Combine Neo4j with Qdrant)

Link to OpenAI key and raw data: Resources_for_AI_candidates