Module 3 Capstone Milestone: Software Design

Alexander Ricciardi

Colorado State University Global

CSC480: Capstone Computer Science

Dr. Shaher Daoud

June 29, 2025

Module 3 Capstone Milestone: Software Design

The MSHA Regulatory Conversational Agent, now renamed Mining Regulatory Compliance Assistant (MRCA), is an AI-powered web application that will provide a quick, reliable, and easy way to query Mine Safety and Health Administration (MSHA) regulations using natural language. MRCA combines Large Language Models (LLMs) with an Advanced Parallel HybridRAG (HPA) technique to minimize LLM hallucinations. This paper describes the MRCA's architectural components, such as microservices through HTTP RESTful API using the FastAPI framework, and the Advanced Parallel HybridRAG system, which uses a Neo4j AuraDB knowledge graph (KG) with vector embedding attributes (vector index) database for semantic search, traversal search, and data retrieval.

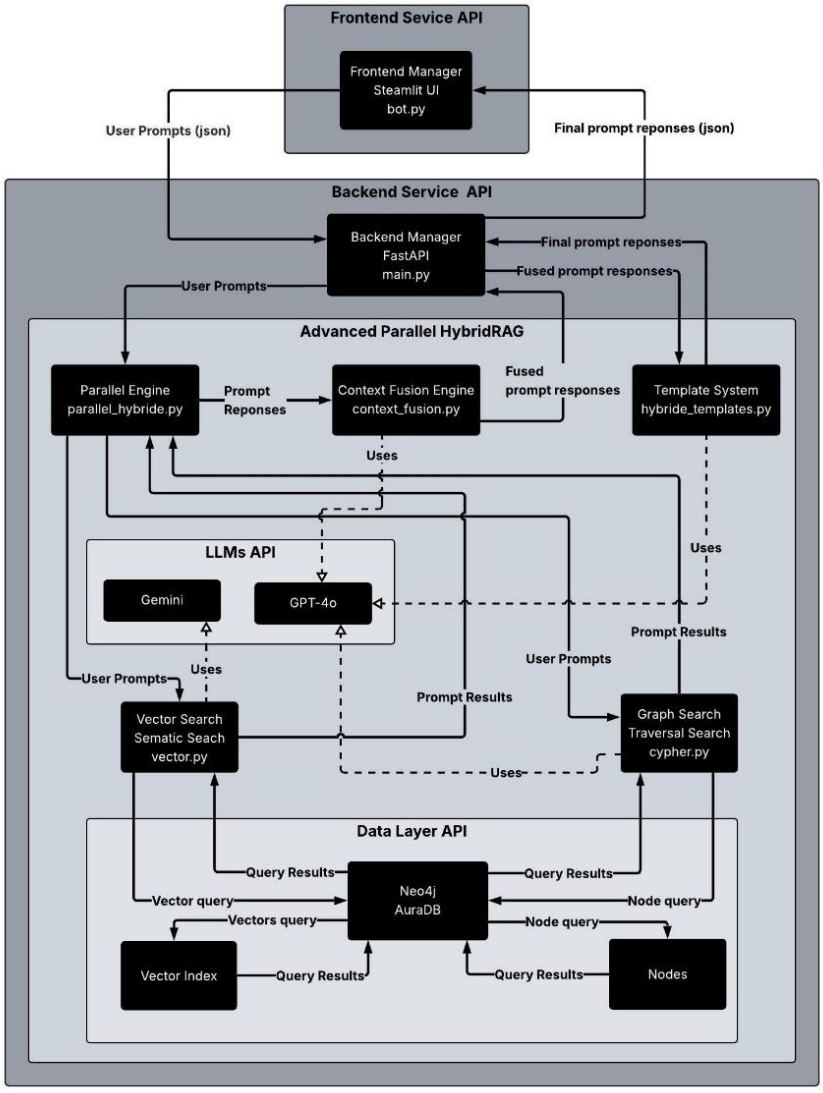
MRCA Architecture

After iterating through different architecture designs, the MRCA architecture evolved from a Modular Monolith Architecture using a sequential HybriRAG to a more advanced Microservice Backends-for-Frontends (BFF) architecture using an Advanced Parallel HybridRAG. Microservice BFF is an architecture that adds an abstraction layer (interface) between the frontend User Interface (UI) and the backend, retrieval data layer, that serves as micro-service for the UI (Abdelfattah & Cerny, 2023). The Advanced Parallel HybridRAG is a novel system that is similar to the HybridRAG concept. The HybridRAG is a RAG system that combines KG-based RAG techniques (GraphRAG) and VectorRAG techniques (Sarmah et al., 2024). The Advanced Parallel HybridRAG differs from the HybridRAG in that it does not choose between a vector-based search (VectorRAG) or a graph-based search (GraphRAG); instead, it executes both retrieval methods in parallel. Additionally, within the Neo4j AuraDB, KGs and vector embeddings are stored in the same database. MRCA stores vector embeddings as

attributes on nodes within the KG. This approach enables the same Chunk nodes to be used as both VectorRAG - semantic node search (via the textEmbedding property) and GraphRAG - traversal search (via entity relationships). Please see Figure 1 for an overview of the MRCA architecture.

Figure 1

MRCA Architecture Diagram



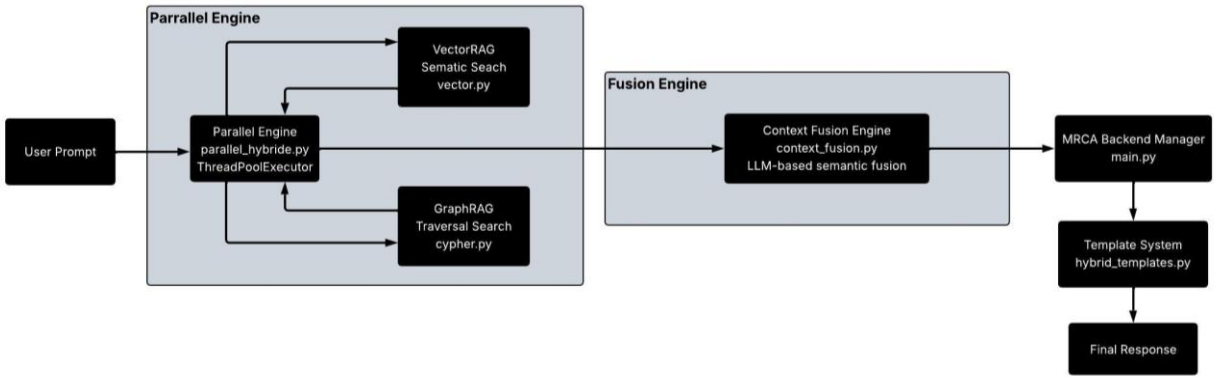
Note: The diagram illustrates the MRCA architecture with its core components and their data/control flow.

MRCA Advanced Parallel HybridRAG

The Advanced Parallel HybridRAG (APH) technique processes user prompts by using the VectorRAG and GraphRAG retrieval methods in parallel. In other words, within the APH parallel engine, the entirety of a user prompt is processed in parallel by both the VectorRAG and GraphRAG methods. Then, the results of the RAG queries are fused using an LLM power context fusion engine, and finally, the fused prompt is processed by an LLM using a response template to produce the final response. See Figure 2 for an illustration of the APH parallel engine and the context fusion engine.

Figure 2

APH Parallel and Context Fusion Engines



Note: The diagram illustrates the structure of APH parallel and context fusion engines.

MRCA Architecture Component

MRCA Microservice BFF's frontend and backend APIs are implemented using FastAPI. The frontend service is a web interface built with Streamlit. The interface files are located in the `frontend` directory, which is defined as a Python package (using `__init__.py`). The core file of the interface is `bot.py`, which hosts the section manager functionality, handles persistent conversation history, displays chat conversation, and captures the user's prompts. The backend

service files are located in the `backend` directory, which is also defined as a Python package. The `main.py` file is the core component of the backend interface. It orchestrates the entire APH process, see Figure 1 and Figure 2. Additionally, to construct the database, the system uses a set of standalone Python executable scripts, `cfr_downloader.py` to download the Title 30 CFR PDF files from govinfo.gov, and `build_hybrid_store.py` to build the Neo4j AuraDB database. Google Gemini 2.5 Pro is used to construct the knowledge graph, and Google embedding-001 is used to build the vector index. Finally, see Table A.1 in the Appendix section of this paper for a detailed list of the project codebase structure, including directories, files, classes, and functions.

Conclusion

The MRCA Microservice BFF architecture provides modular cohesion and low coupling through frontend and backend service separation, allowing MRCA to be a scalable, maintainable, resilient system. Additionally, MRCA Advanced Parallel HybridRAG (HPA) is a novel approach to RAG that differs from traditional sequential RAG systems by using both VectorRAG and GraphRAG in parallel and fusing their query results. This provides the LLM with more accurate and reliable data based on the fused results of a semantic search and a transversal search of a KG database, which significantly reduces the LLM's hallucinations. Finally, the overall MRCA architecture meets all the quality attributes that should drive any architectural decision-making mentioned by Keeling (2017), such as modifiability, testability, availability, performance, and security.

References

Abdelfattah, A., & Cerny, T. (2023). *Filling the gaps in microservice frontend communication: Case for new frontend patterns*. In *Proceedings of the 13th International Conference on Cloud Computing and Services Science (CLOSER 2023), 1,* 184–193, SciTePress. https://doi.org/10.5220/0011812500003488

Keeling, M. (2017). Chapter 5: Dig for architecturally significant requirements. *Design it! From programmer to software architect*. Pragmatic Bookshelf. ISBN-13: 978-1-680-50209-1

Sarmah, B., Patel, S., Hall, B., Pasquali, S., Rao, R., & Mehta, D. (2024, August 9). HybridRAG: Integrating Knowledge Graphs and Vector Retrieval Augmented Generation for Efficient Information Extraction. *arXiv*. https://arxiv.org/abs/2408.04948

Appendix

Table A.1

MRCA Codebase Structure Table

File	Description	Components (Functions & Classes)
Root		
`build_hybrid_store.py`	Knowledge graph and vector index building script using Gemini 2.5 Pro	**Classes:** GraphBuilder **Functions:** main()
`cfr_downloader.py`	Downloads Title 30 CFR PDFs from govinfo.gov	**Classes:** CFRDownloader **Functions:** main()
`docker-compose.yml`	Multi-service container for frontend/backend	Configuration file
frontend/		
`__init__.py`	Frontend package	Package documentation
`bot.py`	Main Streamlit chat interface - configuration	**Functions:** get_session_id(), write_message(), display_header(), display_disclaimer(), display_sidebar(), get_welcome_message(), call_parallel_hybrid_api(), handle_submit(), initialize_session(), main()
`requirements.txt`	Frontend dependencies (Streamlit, requests, etc.)	Dependencies file
`Dockerfile.frontend`	Container configuration for Streamlit frontend service	Docker configuration
frontend/.streamlit/		
`config.toml`	Streamlit theme and UI configuration settings	
`secrets.toml`	Frontend secrets and API keys (git-ignored)	
backend/		
`__init__.py`	Backend package	Package documentation + exports
`main.py`	FastAPI server, orchestrates Advanced Parallel Hybrid processes	**Classes:** ParallelHybridRequest, ParallelHybridResponse, HealthResponse **Functions:** FastAPI endpoints
`parallel_hybrid.py`	Core parallel engine component for VectorRAG + GraphRAG	**Classes:** RetrievalResult, ParallelRetrievalResponse, ParallelRetrievalEngine **Functions:** get_parallel_engine()

`context_fusion.py`	Core fusion engine - fusion algorithm(s)	**Classes:** HybridContextFusion **Functions:** get_fusion_engine()
`hybrid_templates.py`	Prompt template(s) used to create the final response – part of the fusion engine	**Classes:** HybridPromptTemplate **Functions:** get_template_engine(), create_hybrid_prompt()
`config.py`	Pydantic-based configuration management	**Classes:** BackendConfig **Functions:** init_config(), get_config(), validate_config(), get_database_config(), get_llm_config(), get_logging_config()
`llm.py`	OpenAI GPT-4o and Gemini embeddings	**Classes:** LazyLLM, LazyEmbeddings **Functions:** validate_openai_config(), get_llm(), validate_gemini_config(), get_embeddings()
`graph.py`	Neo4j database connection and utilities	**Classes:** LazyGraph **Functions:** validate_neo4j_config(), get_graph(), get_graph_schema(), test_connection()
`utils.py`	Backend utility functions for sessions and formatting	**Functions:** get_session_id(), get_session_data(), save_message(), format_regulatory_response()
`requirements.txt`	Backend dependencies (FastAPI, LangChain, Neo4j, etc.)	Dependencies file
`Dockerfile.backend`	Container configuration for FastAPI backend service	Docker configuration
backend/tools/		
`__init__.py`	Tools package documentation for MSHA regulatory queries	Package documentation
`vector.py`	VectorRAG tool for semantic search	**Functions:** get_neo4j_vector(), get_vector_retriever(), create_vector_chain(), search_regulations_semantic(), search_regulations_detailed(), get_vector_tool()
`cypher.py`	GraphRAG tool for Cypher query generation and graph traversal	**Functions:** get_cypher_qa(), query_regulations(), query_regulations_detailed(), get_cypher_tool()

`general.py`	General MSHA tool for fallback and overview responses	**Functions:** create_msha_general_chat(), provide_msha_guidance(), provide_regulatory_overview(), handle_out_of_scope_questions(), get_general_tool(), get_overview_tool(), create_general_chat_chain(), test_general_tool(), create_general_chat_tool()

Note: The table illustrates a detailed list of the MRCA codebase structure, including a short description of each MRCA file.