Module 4 Capstone Milestone: Software Project Plan

Alexander Ricciardi

Colorado State University Global

CSC480: Capstone Computer Science

Dr. Shaher Daoud

July 6, 2025

Module 4 Capstone Milestone: Software Project Plan

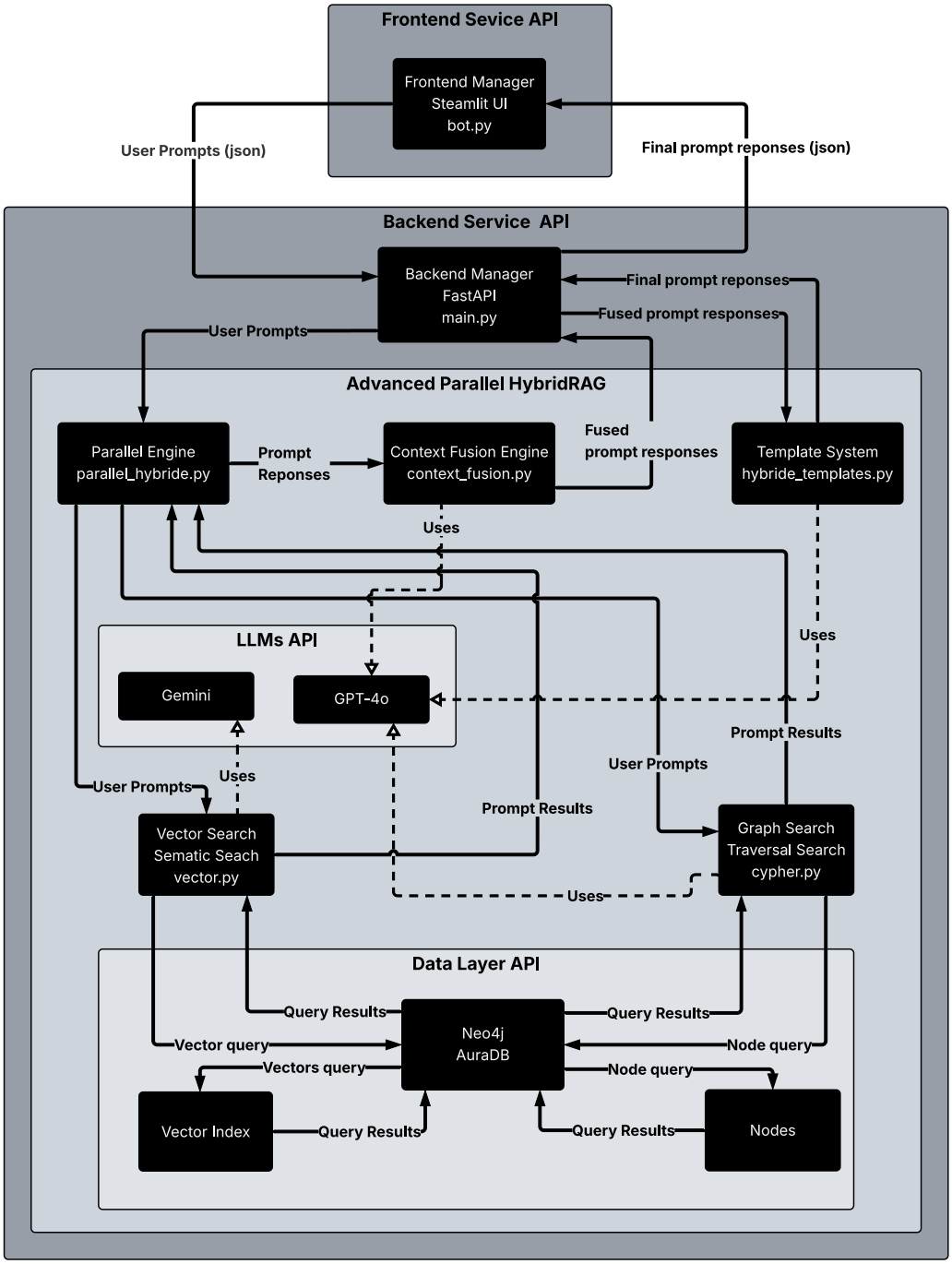
The Mining Regulatory Compliance Assistant (MRCA), as stated in the previous Module 3 - Software Design section of this project, "will provide a quick, reliable, and easy way to query Mine Safety and Health Administration (MSHA) regulations using natural language" (Ricciardi, 2025). However, after the integration of the novel Advanced Parallel HybridRAG (APH) within the application's Microservice Backends-for-Frontends (BFF) architecture, in addition to providing MSHA regulations queries, the application will also serve as a test and research tool for APH. To that end, new features were added to the MRCA design to support testing and research, including a circuit breaker for system stability, a system health monitor, various fusion algorithms, various final prompt templates, and a response metric evaluator. To illustrate the MRCA architecture core components, this document provides UML backend class diagram, a frontend components diagram, and a UML state machine diagram of the MRCA system core components.

MRCA Architecture Overview

Used as a reference from the Module 3 - Software Design section of this project, the following diagram, see Figure 1, illustrates MRCA's architecture. It illustrates the application's microservice architecture, core components' data/control flow, and related files. The system leverages an HTTP RESTful API built using FastAPI to create two micro-services. The micro-service frontend is built using Streamlit as a web User Interface (UI), and the micro-service backend leverages the APH system for retrieval and an LLM for prompt processing. Note that this diagram does not illustrate the new features adopted in the last development iteration, as well as the unloading of the MSHA data from the govinfo.gov MRCA component, and the Knowledge Graph (KG) database builder MRCA component.

Figure 1

MRCA Architecture Diagram



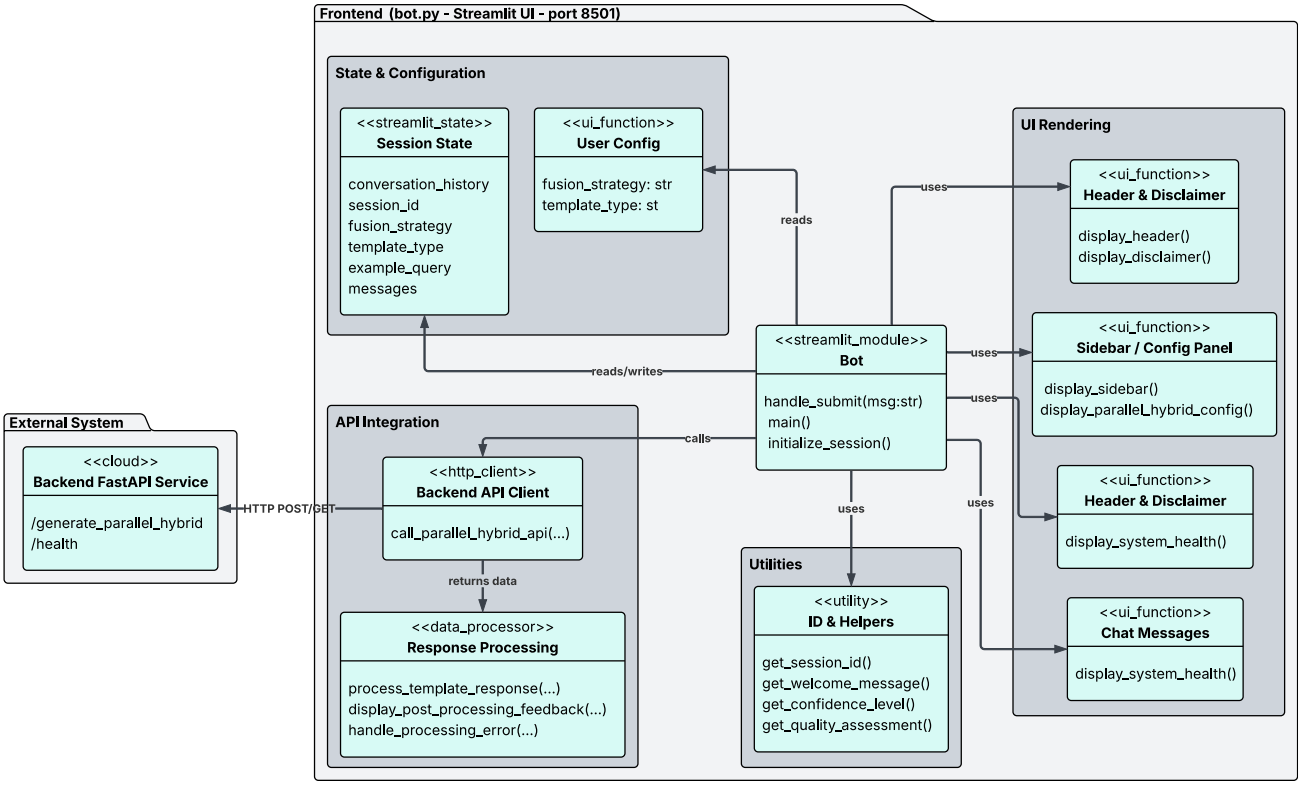
Note: The diagram illustrates MRCA's microservice architecture, core components' data/control flow, and related files. From "Module 3 Capstone Milestone: Software Design" by Ricciardi (2025).

Frontend Components

The MRCA frontend does not include classes, as the application is built in Python. Unlike Java, Python is not a strictly object-oriented programming language; it allows a blend of procedural, functional, and object-oriented programming, and depending on the framework used and requirements, classes are not always the best solution. Thus, the MRCA based on the Streamlit framework does not use classes. MRCA frontend, the web UI micro-service, is coded using a functional programming style. The following diagram, see Figure 2, illustrates the frontend core component logic that cross-references to the `Front Manager Streamlit UI` component of the MRCA Architecture diagram.

Figure 2

MRCA Frontend Core Components Diagram



Note: The diagram illustrates MRCA's frontend web UI logic.

The following table, see Table 1, illustrates the frontend functionality shown in the MRCA Frontend Core Components diagram and how it cross-references to the MRCA Architecture diagram.

Table 1

MRCA Architecture and Frontend Components Cross-references

MRCA Architecture Component	MRCA Frontend Module	Key Functions/Attributes
Frontend Service API	**Bot** (`streamlit_module`)	Handles the frontend message and initializes sessions. `main():handle_submit(msg:str)` `initialize_session()`
	State & Configuration (`streamlit_state`, `ui_function`)	Handles session states. `session_state: conversation_history`, `session_id` `User Config`: `fusion_strategy`
	UI Rendering (`ui_function`, `ui_module`)	Displays the UI header, disclaimer, sidebar, configuration panel, and chat messages.
	API Integration (`http_client`, `data_processor`)	Handles communication with the backend using the `Backend API Client` and processes responses using `Response Processing`.
	Utilities (`utility`)	Provides helper functions: `get_session_id()`, `get_welcome_message()`, and `get_confidence_level()`.

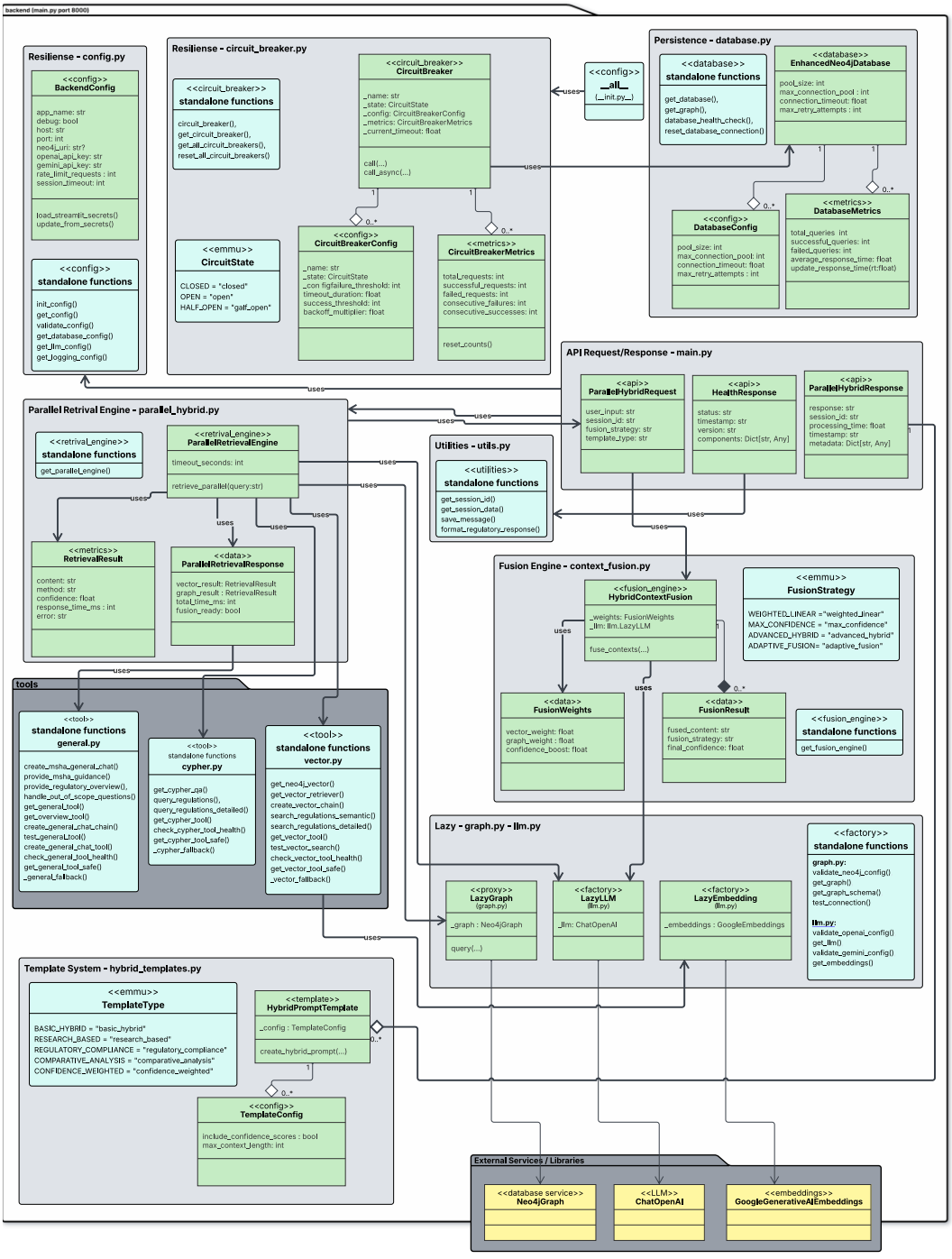
Note: The table illustrates the frontend core functionality and cross-references to the Frontend Core Components with the MRCA Architecture diagram.

Backend Class Diagram

The MRCA backend micro-service implements the APH system and FastAPI. The files are located in the `backend` directory, with the `main.py` components orchestrating the entire functionality of the backend. The backend system combines object-oriented and functional programming. It also uses external services through API calls. Services such as OpenAI and Google APIs for LLMs use and Neo4j AuraDB for a graph database. The following diagram, see Figure 3, illustrates the MRCA backend class diagram, standalone functions, and related files.

Figure 3

MRCA Backend Class Diagram



Note: The diagram illustrates the MRCA backend class diagram, standalone functions, and related files.

The following table, see Table 2, cross-references the MRCA Architecture diagram with the Backend Class diagram.

Table 2

MRCA Architecture Components and Backen Classe Cross-references

MRCA Architecture Component	MRCA Backend Classes Cross-Reference
Frontend Service API	Not shown in the backend classes diagram, it interacts with `main.py`.
Backend Service API	The `main.py` (`API Request/Response` and `main.py`). The `ParallelHybridResponse` class is also part of the API's data.
Advanced Parallel HybridRAG	Cross-references to several backend components. It represents the interaction between the `Parallel Retrieval Engine`, `Context Fusion Engine`, and `Template System`.
Parallel Engine	The `parallel_hybride.py`, the `ParallelRetrievalEngine` class.
Context Fusion Engine	The `context_fusion.py`, the `FusionEngine` class, and the `FusionStrategy` enum.
Template System	The `hybride_templates.py`, the `HybridPromptTemplate` class, and the `TemplateType` enum.
LLMs API	The `graph.py` and `llm.py`, the `LazyGraph`, `LazyLLM`, and `LazyEmbeddings`, which are wrappers for the LLM APIs.
Vector Search	`vector.py`, within the `standalone functions` definition.
Graph Search	`cypher.py` within the standalone functions definition.
Data Layer API (Neo4j AuraDB)	`database.py`, the `EnhancedNeo4jDatabase` class, including the `DatabaseConfig` class.

Note: The table cross-references the MRCA Architecture diagram components with the Backend Class diagram classes.

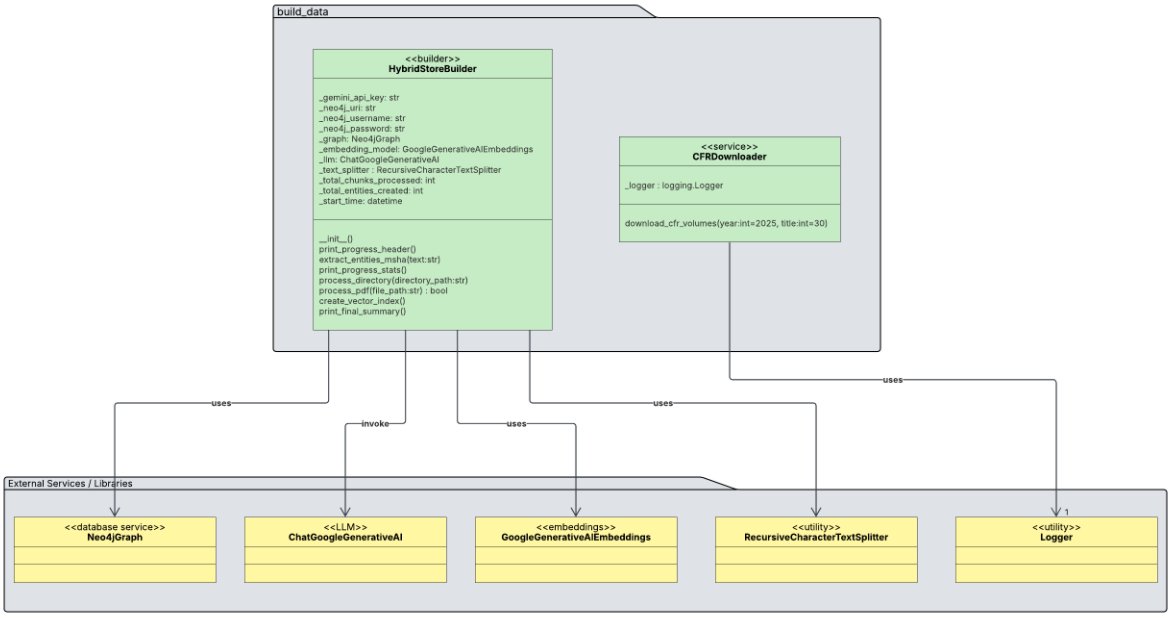
The Build Data Class Diagram

The built data components are not illustrated in the original MCRA Architecture diagram. However, it is crucial to describe their logic as they play a crucial role. The built data includes two standalone scripts stored in the `build_data` directory. The standalone `cfr_downloader.py` script is responsible for downloading the MSHA regulations Title 30 CFR PDFs using the class

`CFRDownloader`. The standalone `build_hybrid_store.py` script is responsible for building the Neo4j KB database with property-indexed vectors using the class `HybridStoreBuilder`. Note that the `HybridStoreBuilder` class uses a Google LMM to generate the KG and an embedding model to create the vector indexes. The following diagram, see Figure 5, illustrates both classes found in the data build standalone scripts.

Figure 4

MRCA *Build Data Class Diagram*



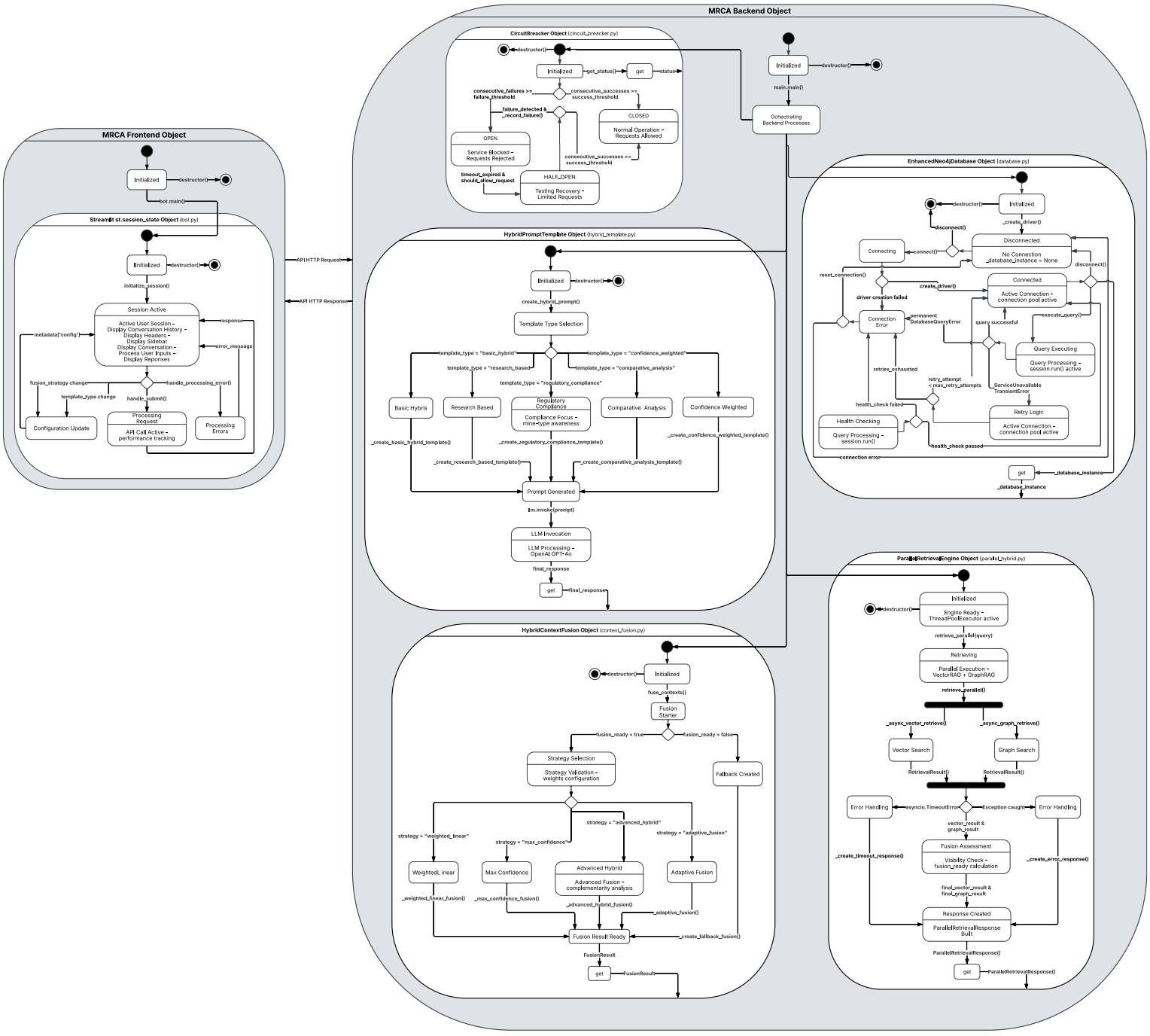
Note: The diagram illustrates the MRCA building KG with vectors embedding property class and the MSHA regulation data downloading class.

MRCA Machine State Diagram

The folowing diagram, see Figure 5, illustrates the runtime state of different objects related to the architecture components found in the MRCA Architecture diagram and the Frontend Core Components diagram. The objects are based on the class objects found in the MRCA Backend Class diagram.

Figure 5

MRCA Machine State Diagram



Note: The diagram illustrates the runtime state of various frontend and backend MRCA objects.

The following table, see Table 3, cross-references the component found in the MRCA Architecture diagram with the MRCA object states found in the MRCA Machine State diagram.

Table 3

MRCA Architecture Components and MRCA Object States Cross-references

MRCA Architecture Component	MRCA State Diagram Cross-Reference
Frontend Service API	`MRCA Frontend Object, Streamlit st.session_state Object (bot.py)`
Backend Service API	`MRCA Backend Object` (`backend` directory and `main.py`)
Advanced Parallel HybridRAG	Logical components of `ParallelRetrievalEngine`, `HybridFusionEngine`, and `HybridPromptTemplate objects`.
Parallel Engine	`ParallelRetrievalEngine Object (engine_instance.py)`
Context Fusion Engine	`HybridFusionEngine Object (fusion_instance.py)`
Template System	`HybridPromptTemplate Object (template_instance.py)`
LLMs API	`LLM Invocation` states within the `HybridPromptTemplate Object.`
Vector Search	Illustrates an object state `Vector Search` in the `ParallelRetrievalEngine - Object's Parallel Execution` state.
Graph Search	Illustrates an object state `Graph Search` in the `ParallelRetrievalEngine - Object's Parallel Execution` state
Data Layer API (Neo4j AuraDB)	`EnhancedNeo4jDatabase Object (database.py)`

Note: The table cross-references the MRCA Architecture diagram components with the State diagram object states.

Conclusion

This document provides a detailed class diagram of the MRCA backend, a class diagram of the build data classes, and state machine diagrams illustrating the state of MRCA objects at runtime. The diagrams showcase the functionality, modularity, scalability, maintainability, and resilience of the MRCA system. Specifically, by showcasing the backend classes and objects' state (behavior), the digraphs illustrate well the functionality of the novel APH system found in the backend micro-service of the MRCA architecture.

References

Ricciardi (2025, June 29). Module 3 Capstone Milestone: Software Design [Student Essay]. CSC480 Capstone Computer Science. CSU Global.