Product Requirements Document: LLM Arena - RAG Evaluation & Benchmarking Suite

- **Version:** 1.0
- **Date:** October 26, 2023
- **Author:** [Your Name/Company Name]
- **Status:** Proposed

---

1. Introduction & Vision

1.1. Product Overview

The "LLM Arena" is an integrated Python suite for building, evaluating, and benchmarking Retrieval-Augmented Generation (RAG) systems. The suite provides a complete workflow: from creating distinct RAG pipelines for multiple LLM providers, to generating evaluation datasets, and finally, enabling human-in-the-loop grading via a simple user interface.

1.2. Problem Statement

Choosing the best LLM and RAG configuration for a specific knowledge base is a significant challenge. It requires a standardized process for:

1. Building comparable RAG systems for different models.
2. Generating unbiased, high-quality test questions based on the source data.
3. Collecting and comparing answers from each system.
4. Facilitating a fair, human-led judgment of answer quality.

This suite automates this entire pipeline, enabling developers and ML engineers to make data-driven decisions about which LLM performs best for their specific domain.

1.3. User Personas

- **Machine Learning Engineer:** Responsible for building and optimizing LLM-based applications.
- **AI Application Developer:** Integrates LLM capabilities into software products.
- **Data Scientist / QA Analyst:** Responsible for testing and validating the quality and accuracy of AI models.

---

2. System Workflow Overview

The software operates in a sequential, multi-stage process:

1. **Build:** Create separate FAISS-based RAG systems for each configured LLM.

2. **Generate Questions:** Use all created RAG systems to generate a pool of questions based on the source documents. These are then de-duplicated into a final evaluation set.
3. **Generate Answers:** For each question in the final set, query every LLM (via its RAG system) to get an answer.
4. **Evaluate:** Launch a UI application that allows a human user to review the question and the competing answers, selecting the best one. Results are persisted to a database.

---

3. Functional Requirements (FR)

FR-1: Configuration

- **FR-1.1:** The system MUST be configured via a single `config.yaml` file.
- **FR-1.2:** The `config.yaml` file MUST support:
    - `source_documents_path`: Path to the folder with input documents (`.pdf`, `.txt`, `.md`).
    - `vector_store_path`: Path to the root folder where FAISS vector stores will be saved.
    - `output_data_path`: Path to a folder for saving intermediate and final data files (questions, answers).
    - `database_path`: Path to the SQLite database file for storing evaluation results.
    - `llm_providers`: A list of objects defining each LLM.
- **FR-1.3:** Each object in `llm_providers` MUST contain:
    - `name`: A unique identifier (e.g., "gemini-1.5-pro", "openai-gpt-4o", "claude-3-opus").
    - `api_key`: The API key (the system should prioritize an environment variable if present, e.g., `GOOGLE_API_KEY`).
    - `embedding_model`: The model used for embeddings (e.g., `models/embedding-001`).
    - `generation_model`: The model used for generation (e.g., `models/gemini-1.5-pro-latest`).

FR-2: Stage 1 - RAG System Builder

- **FR-2.1:** The software MUST provide a CLI command: `python main.py build`.
- **FR-2.2:** The `build` command will iterate through each provider in `config.yaml` and, for each one:
    1. Create a provider-specific subdirectory in `vector_store_path` (e.g., `./vector_stores/gemini-1.5-pro/`).

2. Load, parse, and chunk all documents from `source_documents_path`. Chunking parameters (size, overlap) should have sensible defaults but be optionally configurable.
3. Use the provider's `embedding_model` to create vector embeddings for each chunk.
4. Build a FAISS index from the embeddings and save it as `index.faiss` in the provider's subdirectory.
5. Save a mapping file (`documents.pkl` or similar) that links index entries to their source text and document metadata.

FR-3: Stage 2 - Question Generation & Curation

- **FR-3.1:** A CLI command `python main.py generate_questions` MUST perform this stage.
- **FR-3.2:** The command will:
    1. Initialize all RAG systems built in Stage 1.
    2. For each RAG system, generate a set of N questions (e.g., N=50, configurable) by sampling text chunks and prompting the LLM to create a question answerable from that chunk.
    3. Collect all generated questions from all LLMs into a single list.
    4. **De-duplicate** this list to create a final, clean set of questions. De-duplication should be case-insensitive and ignore minor punctuation/whitespace differences.
    5. Save the final, unique list of questions as `curated_questions.json` in the `output_data_path`. Each entry should contain a unique ID and the question text.

FR-4: Stage 3 - Answer Generation

- **FR-4.1:** A CLI command `python main.py generate_answers` MUST perform this stage.
- **FR-4.2:** This command requires that `curated_questions.json` exists.
- **FR-4.3:** The command will:
    1. Load the `curated_questions.json` file.
    2. Initialize all RAG systems.
    3. For **each question** in the JSON file:
       a. For **each LLM provider**:
       i. Use that provider's RAG system (retriever) to find relevant context for the question.
       ii. Use that provider's `generation_model` to generate an answer based on the question and the retrieved context.
    4. Save the results to `generated_answers.json` in the `output_data_path`.

FR-5: Stage 4 - Human Evaluation UI

- **FR-5.1:** A CLI command `python main.py launch_ui` MUST start the evaluation application.
- **FR-5.2:** The UI MUST be a web application (e.g., built with Streamlit or Gradio).
- **FR-5.3:** On startup, the UI will:
    1. Load the `generated_answers.json` file.
    2. Connect to the SQLite database at `database_path` and retrieve all previously graded questions to avoid showing them again.
    3. Display the first ungraded question.
- **FR-5.4:** The UI layout for each evaluation MUST display:
    1. The current question text prominently.
    2. A list or side-by-side view of the answers, each clearly labeled with the `name` of the LLM provider that generated it.
    3. A "Select as Best" button next to each answer.
- **FR-5.5:** When a user clicks a "Select as Best" button:
    1. The application MUST write a new record to the evaluation database.
    2. A success message (e.g., "Result saved!") should be briefly displayed.
    3. The UI MUST automatically refresh to show the next ungraded question.
- **FR-5.6:** The UI should display progress, such as "Graded 35 / 150". If all questions are graded, it should display a "Completed!" message.

---

4. Data Models & Schemas

4.1. `generated_answers.json` Schema

A JSON list where each element is an object representing a question and its set of answers.

Generated json

```
[
  {
    "question_id": "q_001",
    "question_text": "What was the primary function of the Mark II
computer?",
    "answers": {
      "gemini-1.5-pro": "The Mark II computer's primary function was to
calculate complex ballistic trajectories for the US Navy.",
      "openai-gpt-4o": "Developed by Howard Aiken, the Mark II was used for
calculating ballistic trajectories.",
      "claude-3-opus": "The main purpose of the Mark II was to compute tables
for ballistics."
```

```
    }
  },
  {
    "question_id": "q_002",
    "question_text": "...",
    "answers": { ... }
  }
]
```

content_copydownload

Use code with caution.Json

4.2. Evaluation Database Schema

A single SQLite table named `evaluation_results`.

- `id`: INTEGER, PRIMARY KEY, AUTOINCREMENT
- `question_id`: TEXT, NOT NULL
- `question_text`: TEXT, NOT NULL
- `best_answer_text`: TEXT, NOT NULL
- `winning_llm_name`: TEXT, NOT NULL
- `evaluation_timestamp`: TIMESTAMP, DEFAULT CURRENT_TIMESTAMP

---

5. Non-Functional Requirements (NFR)

- **NFR-1 (Performance):** API-dependent stages should implement appropriate request handling (e.g., batching, parallel requests if possible) to minimize execution time.
- **NFR-2 (Usability):** The CLI must provide clear logging for each stage (e.g., "Building RAG for gemini-1.5-pro...", "Generating answers for question 5/150..."). The UI must be intuitive and require no training for a technical user.
- **NFR-3 (Modularity):** The architecture must cleanly separate the logic for each LLM provider. Adding a new provider should ideally only require adding an entry to `config.yaml` and a corresponding adapter/connector class if its API differs significantly.
- **NFR-4 (Reliability):** The application must have robust error handling for API failures (rate limits, timeouts, invalid keys), file I/O errors, and database connection issues.
- **NFR-5 (Maintainability):** Code must be well-documented, follow PEP 8 standards, and be organized into logical modules (e.g., `config.py`, `rag_builder.py`, `eval_ui.py`).

---

6. Technical Stack & Constraints

- **Language:** Python 3.9+
- **CLI Framework:** `Click` or `argparse`.

- **RAG & LLM Orchestration:** `LangChain` or `LlamaIndex` is highly recommended.
- **Vector Store:** `faiss-cpu`.
- **LLM SDKs:** `google-generativeai`, `openai`, `anthropic`, etc.
- **UI Framework:** `Streamlit` or `Gradio`.
- **Database:** `sqlite3` (standard library) with `SQLAlchemy` for ORM (optional but recommended).
- **Environment:** A `requirements.txt` file must be provided. The developer should assume the user can manage environment variables for API keys.

---

7. Acceptance Criteria

The project is complete when a user can successfully execute the entire workflow:

1. Configure `config.yaml` with at least three LLM providers and paths.
2. Run `python main.py build` and see the respective RAG system folders and files created correctly.
3. Run `python main.py generate_questions` and find a `curated_questions.json` file in the output path.
4. Run `python main.py generate_answers` and find a `generated_answers.json` file populated with answers from all LLMs for every question.
5. Run `python main.py launch_ui` and see the web application launch successfully.
6. In the UI, view a question and its answers, click "Select as Best" for one of them, and verify that a corresponding record is created in the SQLite database file.
7. Verify that after grading a question, the UI automatically proceeds to the next ungraded one.
8. The final submission includes a `README.md` file with setup instructions, environment variable definitions, and command execution order.