This is a complete **Technical Design Document (TDD)** and **Implementation Guide** for the **MEGA-RAG** project. This document is written to be handed directly to an engineering team (or used by you and an AI agent) to build the system from scratch.

---

# Project Design Document: MEGA-RAG (Medical Evidence-Guided Augmentation)

**Project Title:** MEGA-RAG: A Tri-Brid Retrieval & Discrepancy-Aware Refinement System for Clinical LLMs

**Target Hardware:** Consumer GPU (T4 16GB / RTX 3060 12GB) or Google Colab Free Tier

**Core Model:** BioMistral-7B (Quantized)

**Primary Goal:** Mitigate hallucinations in medical QA by triangulating evidence from Vector, Keyword, and Graph sources, followed by a self-correction loop.

---

## 1. Executive Summary & Architecture

### 1.1 System Philosophy

Standard RAG systems fail in medicine because they rely solely on semantic similarity (Vector Search), which often misses specific drug names or contraindications that don't "look" similar in vector space. **MEGA-RAG** solves this by implementing a **"Tri-Brid" Retrieval Strategy** followed by an active **Refinement Loop**.

### 1.2 Architectural Diagram

The system consists of three distinct phases: **Ingestion**, **Retrieval**, and **Refinement**.

Code snippet

```
graph TD
    UserQuery[User Query] --> Router{Query Router}
```

```
  subgraph "Phase 1: Tri-Brid Retrieval"
      Router --> Vector
      Router --> Keyword
      Router --> Graph
  end

  Vector --> Aggregator[Evidence Aggregator]
  Keyword --> Aggregator
  Graph --> Aggregator

  Aggregator --> Reranker
  Reranker --> TopK

  subgraph "Phase 2: Generation & Refinement (The MEGA Loop)"
      TopK --> Generator
      Generator --> Draft

      Draft --> Critic
      TopK --> Critic

      Critic -- "Hallucination Detected" --> Refiner
      Refiner --> FinalAnswer[Final Answer]
      Critic -- "Pass" --> FinalAnswer
  end
```

---

# 2. Technical Stack & Dependencies

This stack is selected for **maximum performance on free/cheap hardware** (Google Colab T4).

| Component | Technology | Reasoning |
|---|---|---|
| **LLM (Inference)** | BioMistral-7B-DARE (GGUF/AWQ) | Best open-source medical model. DARE merge is robust. Quantized to 4-bit for T4 GPU. |
| **Orchestration** | LangGraph (LangChain) | Essential for the cyclic |

| | | |
|---|---|---|
| | | "Refinement Loop" (loops are hard in standard chains). |
| **Vector DB** | ChromaDB (Local) | Lightweight, persistent, runs in-memory or on disk. No API costs. |
| **Keyword Search** | BM25Retriever (Rank-BM25) | Critical for finding exact drug names/dosages that vectors miss. |
| **Graph DB** | Nano-GraphRAG (NetworkX) | A lightweight, in-memory implementation of GraphRAG. No Neo4j setup required. |
| **Embedding Model** | BAAI/bge-m3 | State-of-the-art dense retrieval model. |
| **Reranker** | cross-encoder/ms-marco-MiniLM-L-6-v2 | Filters out irrelevant retrieved chunks before the LLM sees them. |

---

# 3. Detailed Implementation Modules

## Module 1: Data Ingestion & Knowledge Graph Construction

**Objective:** Convert raw PDFs (WHO Guidelines) into the three indices.

**Steps:**

1. **Semantic Chunking:** Do not use simple character splitting. Use SemanticChunker (from LangChain) which breaks text based on meaning shifts.
   - *Config:* Breakpoints at percentile 95 of cosine distance.
2. **Entity Extraction (for Graph):** Pass chunks through a lightweight extractor prompt to identify triplets: (Subject, Relation, Object).
   - *Example:* (Metformin) ----> (Type 2 Diabetes)
3. **Indexing:**
   - **Vector Index:** Push chunks to ChromaDB.
   - **Sparse Index:** Push chunks to BM25Retriever.

- **Graph Index:** Use nano-graphrag to build the NetworkX graph from triplets.

**Python Guide (Pseudocode):**

Python

```
# 1. Load Documents
docs = PyPDFLoader("who_guidelines.pdf").load()

# 2. Semantic Chunking
text_splitter = SemanticChunker(embeddings=HuggingFaceEmbeddings("bge-m3"))
chunks = text_splitter.split_documents(docs)

# 3. Build Indices
vector_store = Chroma.from_documents(chunks, embedding_model)
keyword_retriever = BM25Retriever.from_documents(chunks)
graph_index = NanoGraphRAG()
graph_index.insert(chunks) # Extracts entities and builds graph
```

---

## Module 2: The Tri-Brid Retrieval Engine

**Objective:** Retrieve evidence that is semantically relevant, keyword-precise, and structurally connected.

**Logic:**

1. **Input:** User Query ("What are the contraindications of Paxlovid?")
2. **Parallel Execution:**
   - **Vector:** Finds documents discussing "Paxlovid safety".
   - **Keyword:** Finds documents with exact string "Paxlovid".
   - **Graph:** Traverses Paxlovid -> interacts_with -> Rifampin.
3. **Fusion:** Combine results into a single list (e.g., 30 snippets).
4. **Reranking:** Use the Cross-Encoder to score the 30 snippets against the question. Keep the top 5 with the highest relevance scores.

**Why this matters:** The Graph layer will find side effects linked via *other* drugs that Vector search would miss.

---

## Module 3: The Refinement Loop (SEAE & DISC)

This is the "MEGA" part of the project. It implements two specific algorithms often cited in advanced RAG literature.

### A. SEAE (Semantic-Evidential Alignment Evaluation)

- **Role:** The Auditor.
- **Action:** After the LLM generates a draft answer, this module calculates an **Entailment Score**. It checks: "Does the retrieved evidence *actually* support every sentence in the draft?"
- **Implementation:** A simple NLI (Natural Language Inference) check or a self-reflection prompt:
  - *Prompt:* "Premise:. Hypothesis:. Does the premise support the hypothesis? (Yes/No)"

### B. DISC (Discrepancy-Identified Self-Clarification)

- **Role:** The Fixer.
- **Action:** If SEAE finds a "No" (Hallucination), the DISC module kicks in.
  - It isolates the hallucinatory sentence.
  - It attempts to rewrite it using *only* the evidence.
  - If the evidence is missing, it replaces the claim with "Evidence not found."

**LangGraph Workflow:**

Python

```
# Define the State Graph
class AgentState(TypedDict):
    question: str
    evidence: List[str]
    draft_answer: str
    critique_score: float
    final_answer: str

# Define Nodes
def retrieve(state):...
def generate_draft(state):...
def seae_audit(state):
    # Check if answer is grounded in evidence
    if score < 0.8: return "needs_refinement"
    return "approved"
```

```
def disc_refine(state):...

# Build Graph
workflow = StateGraph(AgentState)
workflow.add_node("retrieve", retrieve)
workflow.add_node("generate", generate_draft)
workflow.add_node("audit", seae_audit)
workflow.add_node("refine", disc_refine)

# Add Edges
workflow.add_edge("retrieve", "generate")
workflow.add_conditional_edges("generate", seae_audit, {
   "needs_refinement": "refine",
   "approved": END
})
workflow.add_edge("refine", END)
```

---

# 4. Development Roadmap (12 Weeks)

| Phase | Duration | Deliverables |
|---|---|---|
| **Phase 1: Setup** | Week 1-2 | Environment (Colab/Local), Dependency Installation, Dataset Collection (StatPearls/WHO). |
| **Phase 2: Retrieval** | Week 3-5 | Implement Vector (Chroma) and Keyword (BM25) retrieval. Implement Nano-GraphRAG. Verify retrieval quality. |
| **Phase 3: Generation** | Week 6-7 | Set up BioMistral 4-bit. Connect Retrieval to Generation. Build the "Naive RAG" baseline. |
| **Phase 4: The Loop** | Week 8-9 | Implement LangGraph. Build the SEAE (Audit) and |

| | | |
|---|---|---|
| | | DISC (Refine) nodes. This is the core innovation. |
| **Phase 5: Evaluation** | Week 10-11 | Run RAGAS metrics (Faithfulness). Compare "Naive RAG" vs "MEGA-RAG". |
| **Phase 6: Report** | Week 12 | Finalize thesis document. Create flowcharts of the Refinement Loop. |

---

# 5. Evaluation Plan (The "Proof")

To get an 'A' grade, you must prove your system is better than a basic chatbot.

1. **Baseline:** Standard RAG (Vector only, no refinement).
2. **Your System:** MEGA-RAG (Tri-Brid + Refinement).
3. **Metrics (using RAGAS library):**
   - **Faithfulness:** (Does the answer lie?) Target: >0.9
   - **Answer Relevancy:** (Is it on topic?) Target: >0.85
   - **Context Precision:** (Did we find the right graph node?)
4. **Dataset:** Use **MedHallu** (a dataset of medical hallucinations) to test if your system refuses to answer trick questions.

---

# 6. How to Run This on Google Colab (Free Tier)

Since you are a student, resource constraints are real. Here is the exact strategy:

1. **Runtime:** Select T4 GPU.
2. **Quantization:** You **must** load BioMistral using bitsandbytes in 4-bit precision.

```Python
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.float16,
)
model = AutoModelForCausalLM.from_pretrained(
    "BioMistral/BioMistral-7B",
    quantization_config=bnb_config
```

```
)
```

3. **Graph Storage:** Use NetworkX (RAM-based) instead of Neo4j (Database). It is faster for small projects and requires no installation.

---

# 7. Deliverables for Submission

1. **Codebase:** A GitHub repo with requirements.txt and a clear main.py.
2. **The "Black Box":** A simple Streamlit UI where a user types a symptom and gets a verified, cited answer.
3. **The Report:** Focusing heavily on the *Refinement Loop* logic. This is your "algorithm" contribution.

Developer Note:
The hardest part will be the Graph Construction. It can be slow. For the minor project, do not try to index all medical knowledge. Pick one specific domain (e.g., "Cardiology Guidelines" or "Diabetes Management") and only index documents related to that. This keeps the graph small, fast, and highly accurate.