My Weekend Fun Project: Building an Intelligent Agentic RAG System! 🚀

Welcome to the thrilling sprint of building an intelligent Agentic RAG (Retrieval Augmented Generation) system from scratch, **all accomplished in just two days: Saturday and Sunday!** This project was an intense, yet incredibly rewarding challenge, combining cutting-edge AI with robust backend development in a lightning-fast timeframe.

The Big Idea: Smarter Q&A for My Knowledge Base, FAST! 💡

I wanted a system that could answer questions not just from its general knowledge, but also from my specific internal documents (like HR policies, employee handbooks, etc.). But I didn't just want a simple chatbot; I wanted an *intelligent agent* that could decide the best way to answer, learn from its interactions, and be super fast. And I wanted it done this weekend! This led me to the world of Agentic RAG.

Day 1 - Saturday: Laying the AI Foundations & Adding Intelligence (Python) 🐍🧠💨

Morning Goal: Get a basic RAG system up and running, capable of answering questions from a document.

I kicked off Saturday morning with Python, the go-to language for AI/ML.
What I Did:

1. **Environment Setup:** Got my Python virtual environment ready.
2. **LLM Integration:** Hooked up with Google's Gemini models (gemini-2.0-flash for generation, text-embedding-004 for embeddings). This was my brain!
3. **Document Processing:** Figured out how to load my internal PDFs, split them into manageable chunks, and convert those chunks into numerical "embeddings" (vector representations) using text-embedding-004.
4. **Basic Retrieval:** Implemented a simple vector search using FAISS (Facebook AI Similarity Search) to find document chunks semantically similar to a user's query.
5. **Initial Generation:** Prompted the LLM to generate answers using the retrieved chunks as context.

Aha! Moment: Seeing the LLM actually *answer from my own documents* was pure magic! ✨

Outcome (Morning): A foundational RAG system that could take a query, find

relevant info, and generate an answer. Ready for intelligence!

Afternoon/Evening Goal: Make my RAG system smarter with decision-making capabilities and introduce a caching layer for speed.

The afternoon was dedicated to adding the "Agentic" part and tackling performance.
What I Did:

1. **Query Classification (Agent 1):** Introduced a first-line agent that uses an LLM to classify incoming queries. Is it about "Company HR policies" or "General knowledge"? This helped me route queries efficiently.
2. **Hybrid Retrieval:** Realized that pure semantic search isn't always enough. I integrated BM25 (a keyword-based search algorithm) alongside FAISS to get the best of both worlds – semantic understanding *and* keyword precision. This gave me "hybrid retrieval."
3. **Confidence Scoring:** I built a module to calculate how "confident" my RAG system was in its answer. This was crucial for deciding if an answer was good enough to return or if I needed to try another approach.
4. **Redis Caching - The Challenge & The Fix!**
    - **The Idea:** Store query-answer pairs in Redis (a super-fast in-memory data store) so I don't have to hit the LLM every time for similar questions.
    - **The Problem:** I initially struggled with Invalid numeric value errors when trying to store and retrieve vector embeddings in Redis using a HASH index type. It seemed like redis-py (my Python client) wasn't playing nice with RediSearch (Redis's search module) in that specific configuration.
    - **The Fix! 🥳:** After much head-scratching and debugging, I discovered that using **JSON indexing** in Redis (storing embeddings as JSON arrays within documents) with decode_responses=True on the Redis client was the secret sauce! This allowed me to store and retrieve vectors flawlessly for the cache.
5. **Flask API Wrapper:** Encapsulated my entire RAG logic within a Flask REST API endpoint (/api/agentic_qna). This made my RAG system a standalone microservice, ready to be consumed by other applications.

Aha! Moment: The Redis JSON indexing fix was a huge breakthrough! It felt great to solve that tricky compatibility issue. Also, seeing the "Cache hit!" in the logs was incredibly satisfying. ⚡

Outcome (End of Day 1): A powerful, intelligent RAG engine with decision-making, hybrid retrieval, confidence scoring, and a blazing-fast caching layer, exposed as a Flask API. I finished Saturday knowing the core AI was solid!

Day 2 - Sunday: Building the Brain's Gateway & API

(C#) 🌐🔐

Morning Goal: Create a robust ASP.NET Core backend to manage users, conversations, and act as the API gateway for my frontend.

Sunday morning was dedicated to setting up the persistent backend.
What I Did:

1. **Why .NET?** I decided on .NET for its strong typing, performance, security features, and excellent ecosystem for building enterprise-grade APIs.
2. **High-Level Requirements:** Brainstormed the essential features: user registration/login, conversation creation, continuing conversations, retrieving history, and linking everything to a user.
3. **Database Design (PostgreSQL):** Designed my relational database schema: Users, Conversations, Messages.
4. **C# Models & DbContext:** Translated my database design into C# model classes (User, Conversation, Message) and set up my ApplicationDbContext using Entity Framework Core (EF Core).
5. **Fluent API Relationships:** Configured the one-to-many relationships, ensuring proper foreign keys and cascade delete behavior.
6. **EF Core Migrations:** Set up EF Core migrations to automatically create and update my PostgreSQL database schema from my C# models.

Aha! Moment: Seeing the dotnet ef database update command successfully create all my tables in PostgreSQL was a satisfying step towards persistent data. 💾

Outcome (Morning): A solid data persistence layer for my .NET backend, ready to store all user and conversation data.

Afternoon/Evening Goal: Implement the core business logic, expose my API endpoints, and secure them with JWT authentication.

The final push on Sunday afternoon brought all the pieces together.
What I Did:

1. **Data Transfer Objects (DTOs) & AutoMapper:** Defined all the DTOs for API contracts and set up AutoMapper for seamless data mapping.
2. **Repository Layer Implementation:** Wrote the concrete implementations for IUserRepository, IConversationRepository, and IMessageRepository, providing clean CRUD operations.
3. **Service Layer Implementation:** This was the heart of the business logic:

    - AuthService: Handled user registration (password hashing with BCrypt.Net-Next), login, and **JWT token generation**.
    - FlaskApiService: Encapsulated all HTTP communication with my Python Flask RAG API.
    - ConversationService: The orchestrator! It managed the entire conversation flow, from creating new conversations to retrieving history and calling the Flask API.
4. **API Controllers Implementation:** Created AuthController and ConversationController:
    - Exposed RESTful endpoints for all my user and conversation management features.
    - Implemented input validation.
    - **JWT Authentication:** Secured the ConversationController endpoints using the [Authorize] attribute.
5. **Swagger UI Configuration:** Configured Swashbuckle to understand and display JWT Bearer token authentication in the Swagger UI, making API testing much easier.
6. **Testing & Troubleshooting 🐛➡️🦋:**
    - I tested the login endpoint, got my JWT token, and tried to use it with protected endpoints.
    - **The Final Authentication Fix! 🥳:** I encountered persistent www-authenticate: Bearer error="invalid_token" and IDX14102 errors. This was the last major hurdle. Through meticulous debugging (checking curl -v output, ensuring consistent Encoding.UTF8.GetBytes for the JWT key, and verifying middleware order), I finally pinpointed and resolved the subtle issue, getting the JWT authentication working flawlessly! This was a huge victory for security and API access, making the entire system functional.

Aha! Moment: Successfully authenticating with a JWT and then getting a RAG response through the entire .NET -> Flask -> Redis -> LLM -> Redis -> Flask -> .NET pipeline was incredibly rewarding. The final "invalid_token" fix was the perfect capstone to the weekend's work! 🎉

Outcome (End of Day 2): A fully functional, secure, and robust .NET backend API, acting as the central hub for my intelligent Agentic RAG system.

Conclusion: A Powerful, Intelligent System Built in a Weekend! 🌟

This "weekend fun project" transformed from a simple idea into a sophisticated, multi-layered intelligent RAG system in just two intense days. I've built:

- A **smart Python RAG engine** that classifies queries, uses hybrid retrieval, scores confidence, and caches responses efficiently.
- A **robust .NET backend** that handles user authentication, manages persistent conversation history, and orchestrates the entire RAG workflow through a clean REST

API.

This project was a fantastic, fast-paced learning experience, showcasing the power of combining different technologies (Python, Flask, Redis, C#, ASP.NET Core, PostgreSQL) to create a truly intelligent application. The journey was filled with challenges, but each "aha!" moment made it incredibly rewarding.

Ready for the next challenge? Perhaps a beautiful frontend to bring it all to life! 🎨