Our System: An End-to-End Microservices Architecture for Real-Time Query Intelligence

Introduction

We've always believed that learning is best when it's hands-on and exciting. What started as an idea to bring intelligent, responsive query handling to life turned into this fully containerized, modular architecture that blends microservices, real-time communication, retrieval-augmented generation (RAG), and large language models (LLMs).

In this documentation, we walk through our system's architecture — not just as a technical description, but as a journey of discovery and collaboration. We explain what each part does, how the pieces connect, and how it all came together to build something we're truly proud of.

---

1. The Foundation: Deployment and Automation

We kicked off with a clear goal: build a system that's **modular**, **scalable**, and **easy to deploy**.

- We began by organizing our microservices code in a GitHub repository.
- For consistency across machines and environments, we containerized each service using Docker.
- Using Docker Compose, we were able to integrate and run everything locally to test service interoperability.
- Once satisfied, we pushed these images to Docker Hub, our container registry.
- From there, Kubernetes (K8s) pulled the images and used YAML manifest files to define deployments.
- Kubernetes then created Pods for each microservice. With Calico, we enabled Pod-to-Pod networking and policy control.

This setup allowed us to automate deployments and ensure that every environment, from development to production, remained consistent.

---

2. Kubernetes Cluster and Services

Inside the Kubernetes cluster, we distributed the workload across multiple Pods, each focused on a specific responsibility:

- **UI Pod**: Hosted our frontend application, which users interacted with.
- **FastAPI Pod**: Handled HTTP requests, authentication, and WebSocket communication.
- **Kafka Pod**: Managed inter-service messaging and queues.
- **LLM Pod**: Processed prompts and generated responses using large language models.
- **RAG Pod**: Handled semantic search and context-aware query processing.
- **Storage Pod**: Included Redis and our Vector Database for storing sessions and embeddings.
- **Evaluation Pod**: Ran performance assessments and quality checks on LLM responses.

Each of these Pods played a unique and important role, and together, they allowed our architecture to stay clean, decoupled, and easy to scale.

---

3. Starting the Conversation: User Interaction and Sessions

Users are at the center of everything. They begin by accessing our web interface hosted in the UI Pod.

When a user logs in:

- Their credentials are sent to the FastAPI service.
- The FastAPI service authenticates the SRN and password.
- A session is created and stored in Redis, and additional metadata is stored in a local JSON file (`chat_history.json`).

We designed this to be fast, secure, and session-aware — so we could track and personalize the interaction over time.

---

4. Kafka as the Messaging in Motion

To handle asynchronous communication between services, we integrated Kafka.

Whenever a user submits a query:

- A Kafka Producer sends the message into relevant topics such as `Prompt`, `Query`, or `Answer`.
- Kafka Consumers pick up the messages and initiate the pre-processing pipeline.
- This includes cleaning the input (like filtering profanity), attaching metadata (like user ID and session ID), and logging the event.

Using Kafka was one of our favorite parts of the system — it brought so much flexibility and scalability, and it made inter-service communication feel effortless.

---

5. Persistent Memory: Our Storage Strategy

We needed our system to remember sessions, store conversation history, and enable efficient search.

So, we structured our storage layer into:

- **Redis**, for fast and temporary session storage.
- **chat_history.json**, for archiving past conversations.
- **Qdrant**, a high-performance vector database, which stored document embeddings used for semantic search.

These systems made it possible for us to retrieve the most relevant information at the right time, which is essential for context-aware responses.

6. Intelligent Context: The RAG Workflow

One of the most exciting parts of this project was building the **Retrieval-Augmented Generation (RAG)** pipeline.

Here's how it works:

- The user's cleaned query is embedded into a high-dimensional vector using a model like Sentence-BERT.
- That vector is used to query the Qdrant database for top-k relevant documents.
- The retrieved content is reranked and compiled into a structured prompt.
- We enrich this prompt with session history to keep the conversation consistent and contextual.

We loved seeing how RAG made responses smarter by bringing in actual documents and context instead of relying solely on LLM predictions.

---

7. Generating Answers: The LLM Integration

Once the prompt is ready, it's sent through Kafka to our LLM Pod.

Here:

- The LLM Service formats the prompt.
- The request is sent to OpenAI's GPT-3.5 (or any model we've configured).
- The generated response is returned to the WebSocket Manager, which forwards it back to the frontend in real time.

Integrating LLMs was challenging at first — especially managing rate limits and handling edge cases — but the results were deeply rewarding. We saw responses that felt natural, contextual, and relevant.

---

8. Measuring What Matters: Evaluation and Metrics

To ensure we weren't just building a "cool" system but also an **effective** one, we added a dedicated evaluation layer.

We prepared a dataset of 400+ queries along with their expected responses.

Our evaluation included:

- **Non-LLM metrics**, like keyword matching.
- **LLM-based metrics**, such as:
    - Context Precision
    - Context Recall
    - Context Relevance

- Answer Relevance
- Answer Correctness

All of this was collected into a performance report that showed how well our RAG and LLM pipelines were working in various scenarios.

---

9. Feedback and Logging: Learning from Every Query

Every query, every session, and every generated response was logged.

Our logging system stored:

- The original user query
- The contextual documents used
- The generated response
- The ground truth (if available)

This gave us a strong feedback loop that fed into both our evaluation module and our **RAGA (Retrieval-Aware Generation Analysis)** system — allowing us to learn from real usage and improve continuously.

---

10. System Endpoints and Housekeeping

Our FastAPI backend included several important endpoints:

- **/health**: Checks system status and Redis connection.
- **/session/{id}/history**: Retrieves session history.
- **/session/{id}/DELETE**: Deletes a session cleanly.

We also used FastAPI's startup and shutdown events to initialize Redis and clean up expired sessions.

These pieces ensured the system stayed stable and maintained itself properly in the background.

---

11. Looking Back: What We Learned and Loved

Building this system was a journey — full of technical challenges, architectural decisions, and countless debugging sessions. But more than anything, it was fun.

We got to explore how microservices talk to each other, how real-time systems scale, how retrieval and generation can be combined to improve responses, and how metrics and evaluation make AI systems accountable.

What started as a simple chat idea became a production-ready intelligent system — and we're incredibly proud of how every piece fits together.

Conclusion

This architecture is more than a system—it's a reflection of our curiosity, collaboration, and deep desire to learn how modern intelligent systems are designed and deployed. Each layer, from containerization with Docker to orchestration with Kubernetes, from message brokering with Kafka to backend APIs with FastAPI, and from caching with Redis to working with LLMs—was an opportunity to get our hands dirty and understand what powers the AI systems of today.

What made this journey truly meaningful wasn't just the tech stack—but the way we approached it: as a team eager to experiment, debug, and grow together. Every challenge we faced became a teaching moment. Every integration was a puzzle we were excited to solve. The modular nature of our architecture allows for flexibility and scalability, which means we're not done yet—just laying the groundwork.

We're especially thrilled about the possibilities ahead—such as exploring more advanced RAG strategies, experimenting with self-hosted LLMs, or fine-tuning models using real evaluation feedback. This system is a solid foundation for all that and more.

We would like to sincerely thank the team behind **SW1: GenAI with RAG, LLMs and a bit of Agents** for designing such an immersive and hands-on summer internship experience. Through this 8-week journey, we not only learned about cutting-edge tools like LangChain, LlamaIndex, FAISS, and Qdrant—but also how to think about building intelligent, dynamic systems that bridge unstructured data with LLM capabilities.