# Knowledge Assistant Prototype — Architectural & Design Decisions

This document outlines the key architectural, tooling, and design decisions for the **Knowledge Assistant prototype**, built for the AI coding challenge.

The system is designed to analyze customer support tickets and generate **MCP-compliant responses** using a minimal **RAG pipeline**.

The focus of these decisions is to balance **simplicity, correctness, and production-aware thinking**. While the prototype remains lightweight, each choice considers **future extensibility, maintainability, and reproducibility**.

## 1. Language & API Framework

**Decision**: Use **Python** with **FastAPI** for the API endpoint /resolve-ticket.

**Reasoning**: - Python provides excellent support for LLM integration, FAISS vector stores, and embedding libraries. - FastAPI simplifies asynchronous requests and guarantees JSON outputs. - This combination allows quick prototyping while remaining production-ready.

**Trade-offs**: - Choosing Python over Go for the API may diverge from company stack preferences. - However, it accelerates LLM experimentation and integration.

## 2. LLM Choice

**Decision**: Use **cpp-llama** as a local model.

**Reasoning**: - Running the LLM locally ensures **privacy** and **reproducibility**. - CPU/GPU support allows flexibility for small-scale deployment without reliance on external APIs. - The LLM receives a clear **role, task, and context**, ensuring **MCP-compliant output**.

**Why not Ollama, Torch, or OpenAI**: - **Ollama**: Easier to use but less control over deployment and model versioning; external dependencies may limit reproducibility. - **Torch-based models**: Require more setup, dependencies, and GPU resources; adds complexity for a small prototype. - **OpenAI API**: Introduces external network calls, costs, and potential privacy concerns; reduces local reproducibility and control over prompts. Additionally, OpenAI **limits the number of LLM variants you can try** in parallel, restricting experimentation and iterative evaluation to improve accuracy without adding extra implementation.

## 3. Vector Database

**Decision**: Use **FAISS** for the vector index.

**Reasoning**: - Lightweight, open-source, and fast for local prototyping. - Good ecosystem support with Python. - Easily replaceable with a cloud-hosted alternative (e.g., Pinecone, Weaviate) in production.

**Trade-offs**: - Lacks advanced features like multi-tenancy, durability, and automatic scaling. - For prototype purposes, the simplicity outweighs these limitations.

## 4. Embeddings

**Decision**: Use **sentence-transformers (all-MiniLM-L6-v2)** for embeddings.

**Reasoning**: - Strong performance on semantic similarity tasks with small resource usage. - Widely used in RAG pipelines and well-supported in FAISS workflows. - Maintains a balance between quality and speed for ticket-scale datasets.

**Trade-offs**: - Larger embedding models (e.g., OpenAI text-embedding-ada-002) may provide better recall but add API latency and costs. - The chosen model is optimal for local, reproducible testing.

## 5. Retrieval Strategy

**Decision**: Use **top-k retrieval** with k=3.

**Reasoning**: - Simple and interpretable baseline. - Avoids excessive context size that could overwhelm the LLM. - Small k ensures faster lookups while still providing enough grounding.

**Future Extensions**: - Add reranking models for improved precision. - Explore hybrid retrieval (keyword + vector search).

## 6. Prompting & MCP Compliance

**Decision**: Use **structured system prompts** to enforce MCP response format.

**Reasoning**: - The LLM is instructed with: - **Role**: Support assistant. - **Task**: Answer ticket based only on retrieved docs. - **Output**: JSON response matching MCP spec. - This reduces hallucination and ensures output consistency.

**Trade-offs**: - Prompt engineering adds complexity and may need iteration per model. - However, it is essential for correctness and downstream API reliability.

## 7. Prompt & Embedding Fallback

**Decision** - Store prompts as JSON files (e.g., latest.json) and pre-embed support documents before deployment. - At runtime, the application first looks for **updated prompts and embeddings** in the mounted volume or artifacts. - If no updated files are found, the system **falls back to the default built-in prompts and pre-embedded documents**, ensuring there is always usable data for the RAG pipeline.

**Reasoning** - Guarantees **robust behavior in production** even if the latest updates are missing. - Supports iterative improvements post-deployment without risking downtime or empty responses. - Enables evaluators to see default behavior and updated behavior separately.

**Trade-offs** - Hot-reloading of prompts or embeddings is **not implemented**; updates require replacing files in the volume or rebuilding the container. - Full version tracking or advanced MLflow-style logging is left for future work.

## 8. Error Handling & Output Validation

**Decision**: Implement a **post-generation validation step**.

**Reasoning**: - Ensures the LLM output matches the expected MCP JSON schema. - Falls back to a safe error response if schema validation fails. - Protects downstream services from malformed responses.

## 9. Testing Strategy

**Decision**: Use **unit tests + integration tests** for the pipeline.

**Reasoning**: - Unit tests validate each step: embeddings, FAISS retrieval, JSON validation. - Integration tests simulate an end-to-end ticket resolution flow. - Lightweight pytest framework ensures quick feedback.

**Future Extensions**: - Add **golden test cases** for MCP outputs. - Benchmark retrieval recall and response latency.

## 10. Deployment & Containerization

**Decision**: Containerize with **Docker** and orchestrate with **Docker Compose** for deployment only.

**Reasoning**: - Guarantees reproducibility across environments. - Lightweight deployment; developers do **not** need to run the model in Docker. - Docker Compose simplifies running multiple services (API, pre-embedded RAG data, volumes) in a single deployment. - Ensures a path to production-readiness without over-complicating the development workflow.

**Future Extensions**: - Add observability (logging, metrics). - Swap FAISS with a cloud vector database. - Scale API horizontally. - Integrate CI/CD pipelines for automated builds and deployments. - Explore deployment on **Kubernetes** for large-scale environments.

## Disclaimer on AI Assistance

Throughout this project, an AI assistant was used to support planning, decision-making, and research. Specifically, the AI contributed to:

* Evaluating and comparing different technologies, frameworks, and LLM options.
* Suggesting trade-offs, best practices, and alternative approaches for RAG pipelines.
* Providing references, example code snippets, and architectural inspiration.
* Structuring documentation, diagrams, and decision records for clarity.

All final choices, implementations, and code were reviewed and validated by the project author. The AI was used as a planning and research aid, not as the sole source of technical decisions or code production.

# Summary

The Knowledge Assistant prototype is designed with a **lean but extensible architecture**: - **Python + FastAPI** for rapid API development. - **cpp-llama** for local LLM inference. - **FAISS + sentence-transformers** for simple, effective RAG. - **Structured prompts + validation** for MCP compliance. - **Dockerized deployment** for portability.

This setup balances **prototype speed** with **production-aware design**, allowing future extension into a fully reliable knowledge assistant system.