High-Level Overview

Purpose: The system serves as a backend for conversational AI, integrating OpenAI's models with retrieval-augmented generation (RAG), fine-tuning capabilities, and straightforward deployment. The goal is to provide a flexible chat AI service that retains conversation context, can call external functions, and incorporate private knowledge, all while being easy to deploy on AWS. This design addresses common LLM limitations (being frozen in time and lacking domain-specific data

pinecone.io

) by allowing custom data integration (via RAG and fine-tunes) and giving developers full control over the stack.

Key Features:

- **OpenAI API Compatibility:** The backend's API is designed to mimic OpenAI's Chat Completion API structure for seamless integration. Clients can send chat requests with the same format (model, messages, functions, etc.) and receive responses in a similar JSON schema as they would from OpenAI, making it easy to switch to this system with minimal code changes.
- **Extensible Function Calling:** The system supports OpenAI's function calling mechanism, allowing the AI to invoke custom functions for specialized tasks (e.g. database queries, calculations). Developers can register new functions with schemas, which the model can call when appropriate. This extends the assistant's capabilities beyond language responses – for example, the AI could call a `search_documents` function to query a vector database and then use the result in its reply.

  file-j4dae5aggup2p2xdxtqrux
- **Message Storage with Context Enrichment:** Every conversation turn is stored in DynamoDB for persistence. Because chat models are stateless and only retain memory through provided context, the system retrieves past messages from the database to include in each prompt. It can also enrich the context by

retrieving relevant data (using RAG) when needed – for instance, pulling in prior conversation snippets or knowledge base entries related to the user's query. This ensures continuity and depth even in long-running conversations.

community.openai.com

- **System Message Injection:** The backend automatically prepends a system message at the start of each conversation (or each prompt) to guide the assistant's behavior. This system message can define the AI's role, tone, or constraints (for example, "You are a helpful assistant..."). It is maintained throughout the conversation to consistently enforce these guidelines. The content is configurable, allowing adjustments to the AI's persona or policies without changing code.
- **Configurable API Parameters:** Parameters such as model choice, temperature, max tokens, and others are configurable per request or via server settings. The API exposes these options so that clients can tweak the creativity (temperature), response length, etc., as needed – similar to OpenAI's API. Sensible defaults are provided, but developers can override them to suit different use cases or to experiment with various OpenAI models.
- **Swappable Backend Provider Abstraction:** The system is built with an abstraction layer for the language model provider. While it currently uses OpenAI's API, this design allows plugging in other providers (or even local models) in the future with minimal changes. In practice, an interface defines operations like generating a completion or embedding, and OpenAI is one implementation of it. This means we can "start with OpenAI integration" but later expand to custom or local models, even orchestrating multiple AI systems behind the same API.

  file-j4dae5aggup2p2xdxtqrux
- **Fine-Tuning Support:** To improve performance on domain-specific conversations, the backend can fine-tune OpenAI models on custom datasets. It provides tools to export conversation logs or feed custom Q&A pairs into a training dataset, and an endpoint to trigger the fine-tuning job. By fine-tuning, we can obtain a model that better understands the specific context or jargon of our application, yielding more accurate and efficient responses. (For example, fine-tuning can imbue the model with knowledge of company-

specific terminology or preferred styles.) Fine-tuned models with chat format data use role-tagged JSONL training files, and once a job is completed, the custom model can be deployed through the same API.

dev.to

- **Retrieval-Augmented Generation (RAG) via Pinecone:** The system integrates with a Pinecone vector database to enable RAG. RAG is an approach that supplements the AI's responses with relevant external data fetched at runtime, which helps prevent hallucination by providing the model with knowledge it otherwise lacks. In our setup, documents or facts are indexed as vectors in Pinecone. When a user query comes in, the system generates an embedding for it and searches Pinecone for related content. Any highly relevant snippets are then injected into the conversation context (typically as an additional system message or as part of the prompt) so the model can use them when formulating its answer. This allows the assistant to give answers based on up-to-date or proprietary information not contained in its base training data.

  docs.pinecone.io

  pinecone.io

- **Simple Deployment on EC2:** Designed for low overhead in production, the entire service can be deployed as a lightweight application on an AWS EC2 instance. This avoids the complexity of container orchestration or serverless pipelines for a small-scale deployment. By using a direct EC2 VM, developers have full control and visibility into the running system, and we sidestep additional layers that might introduce latency or complexity. The deployment process is straightforward (essentially setting up Python/Anaconda and running the service), making it accessible for teams that want quick iteration without managing Kubernetes or Docker in production.