Technical Guide

API Architecture

Justification: We built the API to resemble OpenAI's API so that developers find it familiar and easy to use. By structuring our endpoints and payloads similarly, we leverage known patterns and client libraries. This approach accelerates adoption and ensures compatibility with minimal effort.

Endpoints and Structure: The backend exposes HTTP endpoints (e.g., RESTful API using FastAPI or Flask) for interacting with the conversational AI. The primary endpoint is a chat completion endpoint (for example, `POST /api/chat`), which accepts a JSON payload of messages and returns the model's response. The request format mirrors OpenAI's chat completions: clients send a list of messages (each with a `role` like `system`, `user`, or `assistant` and corresponding `content`), and optionally specify parameters such as `model`, `temperature`, `max_tokens`, etc. An example request body might look like:

```
json
CopyEdit
{
  "model": "gpt-3.5-turbo",
  "messages": [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Hello, how can you help me today?"}
  ],
  "temperature": 0.7,
  "functions": [ ... ],
  "function_call": "auto"
}
```

The response from this endpoint is also compatible with OpenAI's format. For instance, a successful response wraps the assistant's reply in a JSON with a `choices` list:

```
json
CopyEdit
{
  "id": "cmpl-xyz",
  "object": "chat.completion",
  "created": 1708800000,
  "model": "gpt-3.5-turbo",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "Hello! I can assist you with a variety of tasks. How may I help
you today?"
      },
      "finish_reason": "stop"
    }
  ]
}
```

By following the same schema (IDs, roles, nested message content, etc.), existing OpenAI API clients or SDKs can parse our responses easily. This design decision (part of the "OpenAI API compatibility" feature) means developers could even redirect OpenAI SDK calls to our endpoint with minimal changes.

OpenAI Completion Integration: Under the hood, when the chat endpoint is called, the backend constructs a prompt for OpenAI's ChatCompletion API. It injects the system message (if not already present), appends recent conversation history from DynamoDB, adds the latest user message, and then calls `openai.ChatCompletion.create()` with these messages. The configured parameters (model name, temperature, etc.) are passed along. The OpenAI API's response is then relayed back to the client through our endpoint. Essentially, the backend acts as a proxy that adds intelligence (like context management, function handling, etc.) around the raw OpenAI calls. For example, a Python call might be:

```
python
CopyEdit
openai.api_key = OPENAI_API_KEY
response = openai.ChatCompletion.create(
   model = chosen_model,
   messages = assembled_messages,  # system + history + new user messag
e
   functions = functions_list,     # if any functions are defined
   function_call = "auto",
   temperature = req.temperature or default_temp,
   max_tokens = req.max_tokens or default_max
)
```

This returns a JSON result, from which we extract the assistant's message (and any function call info). We then store the new assistant message into the conversation history (DynamoDB) and finally send the formatted response back to the client. By doing this, the heavy lifting of AI generation is delegated to OpenAI's managed models, while our system orchestrates the inputs and outputs.

Function Calling Mechanism: We leverage OpenAI's function calling capability to extend what the AI can do. Developers can define a set of functions with JSON schemas that describe their parameters and purpose. These definitions are included in the `functions` field of the ChatCompletion request. We typically set `function_call: "auto"` to let the model decide if a function should be called based on the conversation. For example, one might define a function to search the knowledge base:

```
python
CopyEdit
functions = [
   {
     "name": "search_documents",
     "description": "Search the document database for relevant information",
     "parameters": {
       "type": "object",
```

```
        "properties": {
          "query": {"type": "string", "description": "The search query"},
          "max_results": {"type": "number", "description": "Max documents to retr
ieve"}
        },
        "required": ["query"]
      }
    }
]
response = openai.ChatCompletion.create(
    model="gpt-4",
    messages=assembled_messages,
    functions=functions,
    function_call="auto"
)
```

In this snippet, we defined a `search_documents` function with a schema for its input file-j4dae5aggup2p2xdxtqrux. When the model's response indicates a

```
function_call
```

(e.g., it returns a message with

```
role":"assistant"
```

and a

```
function_call
```

field instead of content), our backend intercepts that. The API architecture includes logic to handle such cases: it will parse the

```
function_call
```

from the OpenAI response, execute the corresponding server-side function (for instance, performing the Pinecone query for the given

```
query
```

parameter), and then feed the function's result back into the model. The function's output is appended as an assistant message (with role

```
function
```

or an injected system note containing the result), and we call the OpenAI API again so the model can continue and produce a final answer that incorporates the function's output. This loop continues until the model provides a final answer rather than another function call. All of this is abstracted behind the single chat endpoint – to the client it appears as one coherent response, but internally the architecture can perform multiple steps (AI deciding to call function → backend executes it → AI continues) in a single round. The function calling system is extensible; new functions can be added to the

```
functions
```

list based on application needs (e.g., "lookup_order_status", "book_appointment"), enabling the AI to interact with external systems safely and in a controlled manner.

Storage and Context Management

Justification: Maintaining conversational context is crucial for a chatbot to appear intelligent and coherent. Since the model does not remember past interactions by itself, we must store and retrieve conversation history externally

community.openai.com

. We chose AWS DynamoDB for this task because it offers a fast, scalable, and managed key-value store that fits our needs (low-latency access by conversation

ID, flexible schema). Using familiar AWS services also leverages our existing expertise and infrastructure for reliability

file-j4dae5aggup2p2xdxtqrux

. Additionally, to augment the context with outside knowledge, we integrate a Pinecone vector database for semantic search – this combination ensures the assistant has both short-term memory (recent messages from DynamoDB) and long-term or domain knowledge (documents via Pinecone).

DynamoDB Conversation Store: Conversation histories are stored in a DynamoDB table, using a simple schema. Each message in a conversation is an item in the table. We use the conversation ID (or a session ID/user ID combination) as the partition key, and a timestamp or incremental message index as the sort key. This way, all messages for a given conversation are grouped, and we can easily retrieve them in chronological order. A possible table structure is:

- Partition Key: `conversation_id` (string, e.g., a UUID or user ID + session marker)
- Sort Key: `message_ts` (number or ISO date string, representing message order/time)
- Attributes: `role` (string: “user” or “assistant” or “system”), `content` (string: the actual message text), plus any other metadata (function call data, etc.)

For example, after a few turns, the table might have items like:

```
makefile
CopyEdit
conversation_id = "abc123", message_ts = 1, role = "system", content = "You a
re a helpful assistant."
conversation_id = "abc123", message_ts = 2, role = "user", content = "Hello"
conversation_id = "abc123", message_ts = 3, role = "assistant", content = "Hi!
How can I help you today?"
conversation_id = "abc123", message_ts = 4, role = "user", content = "Can yo
u tell me about RAG?"
... etc.
```

When a new user request comes in, the system queries DynamoDB for the recent history of that conversation (e.g., the last N messages that fit in context window). DynamoDB's query operation on the partition key efficiently returns all messages; we can apply a limit and sort by the sort key descending to get the latest messages. These are then prepended (in correct order) to the prompt that we send to the model. Because the model can only "remember" what is in the prompt, this retrieval step is what gives the illusion of memory. By storing conversations persistently, users can come back later and the system can load their previous context so they don't start from scratch unless desired

community.openai.com

. Unique conversation IDs also allow multiple parallel chats or multi-user support without collision

community.openai.com

.

We also store each new message (user query and assistant answer) back into DynamoDB as they happen. This append-only log forms the conversation transcript. Over time, a conversation might grow too long to send entirely to the model (due to token limits). In such cases, the context manager can employ strategies like summarizing older messages or dropping the oldest context. One advanced approach we support is using the vector database for old messages: older parts of the conversation can be semantically indexed so that if they become relevant again, they can be fetched via similarity search rather than always being in the prompt. This is an optional strategy for long-term conversations.

Context Enrichment with Pinecone (RAG): In addition to conversation history, the system can inject relevant external information into the context via Retrieval-Augmented Generation. We use Pinecone (a managed vector DB) to store embeddings of knowledge documents or even older conversation summaries. When a new user query arrives, we generate an embedding for the query (using OpenAI's embedding model like `text-embedding-ada-002`) and perform a similarity search in the Pinecone index. For example:

```
python
CopyEdit
```

```
# Create embedding of the user query
embed = openai.Embedding.create(input=user_message, model="text-embed
ding-ada-002")
query_vector = embed['data'][0]['embedding']

# Query Pinecone for similar content
pinecone.init(api_key=PINECONE_API_KEY, environment=PINECONE_ENV)
index = pinecone.Index("knowledge-base")
results = index.query(vector=query_vector, top_k=4, include_metadata=True)
```

Here, `results` might return the top 4 stored pieces of text most similar to the user's question. These could be FAQ answers, documentation passages, or other conversational snippets that the user might be referring to. The system will take these retrieved pieces (for example, each has some `metadata['text']` or similar) and incorporate them into the model's prompt. A common method is to add an extra system message like: *"Knowledge Base Context: … [insert retrieved text] …"* before the assistant formulates its answer. By doing so, the model is given the relevant facts at generation time, dramatically reducing the chance of it hallucinating or giving incorrect info about that topic

docs.pinecone.io

. Essentially, Pinecone + embeddings let us extend the AI's knowledge in real-time with our own data.

The Pinecone index is kept up-to-date with whatever knowledge we need the AI to have. We can periodically add new documents or content by embedding them and upserting to Pinecone. Because this is separate from the model's training, it allows our system to have the latest information (for instance, company policies, product details, or news articles) even if the base model is static. RAG is a proven approach to give LLMs access to private or current data they otherwise wouldn't have

pinecone.io

, and our architecture is built to support it out of the box. The retrieval step is fast (vector search is efficient even with many documents), and we have configuration for it: e.g., how many results to fetch (

```
top_k
```

), a similarity threshold to filter out irrelevant matches, etc., which can be tuned to balance relevance vs. prompt length.

In summary, **context management** in this system works on two levels:

1. **Conversation memory:** DynamoDB provides short-term memory by storing chat history and retrieving it for each request, ensuring continuity in the dialogue.
2. **Augmented knowledge:** Pinecone provides long-term memory or domain-specific knowledge retrieval, feeding the model with additional context when needed.

Together, these ensure the assistant's responses are both contextually aware of the conversation and enriched with accurate information from outside the model's original training data.

System Message Handling

Justification: System messages in OpenAI's chat format act as an initial instruction that guides the assistant's behavior. By injecting a system message, we can control the persona and rules of the AI consistently across the conversation. This is important for maintaining the desired tone, enforcing policies (like not answering certain questions), or adding domain-specific role behavior (e.g., acting as a technical support agent). Our system treats system messages as a first-class element of the conversation, ensuring they are always present and up to date, which provides structure to the model's responses from the get-go.

Injection of System Messages: When a new conversation is initiated, the backend will automatically insert a predefined system message as the first message in DynamoDB (if one isn't supplied by the client). This could be something simple like: *"You are a conversational AI assistant that answers questions helpfully and concisely."* or a more elaborate set of instructions depending on the use case. On every call to OpenAI, we make sure this system prompt is the first in the `messages` list. By always including it, we avoid situations where the model might deviate or behave undesirably over long chats – it always has the initial instructions to refer to.

Our design allows this system message to be **configurable**. In a configuration file or environment variable, one can specify the content of the default system prompt. For example, a deployment for internal use might configure the system message to include company-specific guidelines or a particular style of response. If needed, the system could also support dynamic system messages: e.g., adjusting the system prompt based on user profile (a friendly tone for casual users vs. more formal for enterprise users), or even updating it during a conversation to steer the AI if the context shifts. The architecture supports multiple system messages as well (OpenAI allows more than one system role message), but typically one is sufficient, and additional context is provided via other means (like the knowledge snippets as system messages, as mentioned).

Maintaining System Instructions: As the conversation progresses, the original system message remains at the top of the context in DynamoDB. We do not remove or overwrite it. When pruning old messages due to length, we **never** drop the system message – it's always preserved. This ensures that even if we summarize or truncate the conversation, the model will still see the guiding instruction. In effect, the system message is "sticky." If we use any intermediate system-type messages (like the RAG context message or a summary of earlier conversation), those are usually inserted *after* the main system prompt, so the first message is always the primary instruction set, followed by any additional contextual system notes.

Configurability: Operators of the system can modify settings related to system messages without code changes. For instance, the text of the default system prompt can be loaded from a config file. There may also be a toggle for whether to enable or disable automatic system message injection (in case a client wants to fully control prompts themselves – though by default we inject it for safety). Additionally, the position of inserted context (as system or assistant role) could be configurable. We chose the approach of using system role for added context (like knowledge) so that the model treats it as given information rather than something a user said, which tends to make it follow that information more faithfully.

In summary, system message handling is about **establishing control and consistency**. By injecting a carefully crafted system message at the start of each conversation, the backend ensures the AI operates within the desired bounds and persona. This mechanism is simple but powerful – it's how we can make the same

base model behave like a friendly customer service rep in one deployment, or a strict and factual instructor in another, just by changing the system instructions.

Fine-Tuning OpenAI Models

Justification: While prompt engineering and retrieval can go a long way, there are scenarios where having a fine-tuned model yields better performance. Fine-tuning allows the model to internalize patterns specific to our domain or use case, which can improve response quality and reduce the need to always provide lengthy context. It can also potentially reduce latency or API costs if the fine-tuned model can achieve the same results with shorter prompts. Our system includes a fine-tuning pipeline so that we can continuously improve the AI with real conversation data or curated datasets, creating a virtuous cycle where the model gets better over time on our specific tasks.

Generating Fine-Tuning Datasets: The first step in fine-tuning is preparing a training dataset from our existing data or target Q&A pairs. The system provides a way to export conversation logs (with appropriate filters) into a fine-tuning dataset format. For OpenAI's API, the dataset must be in JSONL (JSON Lines) format, where each line is a training example. Since we're dealing with a chat model, we use the chat completion fine-tune format: each example is a JSON object with a `"messages"` array containing role-content pairs for a conversation snippet. Typically, that includes a system role (if we want to bake in the same initial instruction), a user role with a prompt, and an assistant role with the expected completion. For example, an entry might look like:

```
json
CopyEdit
{"messages": [
  {"role": "system", "content": "You are a helpful math tutor."},
  {"role": "user", "content": "What is 12 times 8?"},
  {"role": "assistant", "content": "96"}
]}
```

This format teaches the model how to respond given the user prompt (with the system role giving context). Our system can construct such examples either from

actual logged conversations (filtering out any sensitive info and ensuring quality) or from manually prepared Q&A. We might have a script or an admin tool that collects all user questions of a certain type and the approved best answers, then format them accordingly. The inclusion of the system message in each training sample is optional, but including it can help the fine-tuned model learn the intended persona or constraints directly

dev.to

. OpenAI's guidelines note that the content should differ depending on chat vs completion models

github.com

– for chat models like GPT-3.5-turbo, we use the structured message format as shown.

Before fine-tuning, these JSONL files are usually uploaded to OpenAI via their API or CLI. The system might facilitate this by providing a command or endpoint to upload the file. (For instance, using the OpenAI Python SDK: `openai.File.create(file=open("train.jsonl"), purpose="fine-tune")` to get a file ID.)

Kicking Off Fine-Tuning Jobs: We have an endpoint (available to administrators or automated pipelines) to initiate a fine-tune. This could be a REST endpoint like `POST /api/fine-tune` which takes parameters such as the training file (or references to it), target base model, and any hyperparameters. Internally, this will call OpenAI's fine-tuning API to create a job. The OpenAI fine-tuning endpoint is `POST /v1/fine_tuning/jobs`

community.openai.com

. We use the OpenAI Python library or direct HTTP call to submit the job, e.g.:

```
python
CopyEdit
response = openai.FineTuningJob.create(
    training_file = file_id,
    model = base_model,          # e.g. "gpt-3.5-turbo-0613"
    n_epochs = 2,                # (if we allow specifying hyperparams)
    suffix = "my-domain-v1"      # optional suffix for easier identification
)
```

```
job_id = response.id
```

Once submitted, OpenAI will queue and run the fine-tuning process on their servers. Our system can track the status of the fine-tune job (we might poll the status or use webhooks if available). When the job is completed, it results in a new fine-tuned model ID (which usually looks like `ft:model-name:...`). We store this model ID in our configuration or database.

Using Fine-Tuned Models: After fine-tuning, clients can specify the fine-tuned model in the conversation requests (since our API allows selecting the model). Alternatively, we might decide to deploy the fine-tuned model as the default for certain tasks. For example, if we fine-tuned a model for customer support dialogues, the system might automatically use that model for conversations in the support domain. The swappable backend abstraction makes this easy – it's just another model in the OpenAI provider. We simply call `openai.ChatCompletion.create` with the fine-tuned model's name instead of a base model. All other features (context injection, function calling) continue to work as before.

We also provide an easy way to generate *validation* data or to evaluate the fine-tuned model. For instance, the system could hold out some conversation examples and after fine-tuning, run them through both the base and fine-tuned model to compare results, ensuring the fine-tune is beneficial.

It's important to note that fine-tuning is complementary to RAG: fine-tuning imparts static knowledge or style into the model, whereas RAG provides dynamic, up-to-date information at query time. Our system uses both: fine-tune for improving general behavior in our domain, and RAG for on-demand retrieval of facts. Both are configured via API – e.g., an admin can trigger a fine-tune job when enough new data has been collected, without shutting down the system.

Process Summary:

1. **Collect Data:** Gather conversation data or Q&A pairs that represent the desired knowledge/behavior.
2. **Format Data:** Convert to JSONL with chat format messages. Ensure each example has clear user prompt and ideal assistant response.

   dev.to

3. **Upload & Launch Job:** Call OpenAI fine-tuning API (via our endpoint). The job runs on OpenAI's side – our service just initiates and monitors it.
4. **Update Model Reference:** Once fine-tuned, update configuration to use the new model ID for relevant requests.
5. **Iterate:** Continue gathering data and fine-tuning as needed. The system could even schedule periodic fine-tunes if continuous learning is desired (with human-approved data to avoid drift).

By supporting fine-tuning, the conversational backend can become more specialized over time, offering users more accurate and context-aware assistance beyond what the base models provide out-of-the-box.

Deployment Strategy

Justification: We opted for a manual deployment on EC2 to keep things simple and accessible. Early-stage or small-scale applications often don't require the complexity of container orchestration or serverless architectures, which can introduce unnecessary overhead. A single EC2 instance can comfortably run this backend, and using it gives us direct control over the environment and easier debugging. In comparison, container-based solutions (like AWS ECS or Kubernetes) offer scalability but at the cost of more setup and maintenance – since our initial load is moderate, we chose the path of least resistance. In short, **an EC2 VM provides a straightforward, low-friction deployment** that aligns with our need for quick iteration and full-stack ownership

file-j4dae5aggup2p2xdxtqrux

.

Moreover, the team is already familiar with AWS and managing EC2, which means we can leverage that knowledge (security groups, IAM roles, monitoring) without learning a new platform

file-j4dae5aggup2p2xdxtqrux

. This familiarity and directness can accelerate development and troubleshooting. Should the need arise to scale out or use containers later, the transition will be considered, but the MVP deploys on a single instance.

Manual EC2 Deployment Steps: We outline a simple CLI-based deployment process to set up the service on an Amazon Linux or Ubuntu EC2 instance:

1. **Provision an EC2 Instance:** Launch an EC2 (e.g., t3.medium) in your preferred region. Attach an IAM role with permissions for DynamoDB, S3, and any other AWS services the system uses (or alternatively, store AWS credentials on the instance, though an IAM role is safer). Ensure the security group allows inbound traffic on the port we'll run the API (for example, port 80 or 8000) and SSH for access.
2. **Install System Dependencies:** SSH into the instance. Install necessary system packages – primarily Python (or Anaconda) and any build tools needed for Python packages. For example, update apt and then `sudo apt-get install build-essential python3-dev git`. If using Anaconda, download the Anaconda or Miniconda installer for Linux.
3. **Set Up Python Environment:** Rather than using a Docker container, we use an Anaconda environment to isolate dependencies. This approach gives many benefits of containerization with less complexity. Create a new conda environment (e.g., `conda create -n chatbot-env python=3.10`) and activate it. Anaconda will handle the Python version and package isolation.

   stackoverflow.com
4. **Fetch Application Code:** Pull the latest code from the repository. This could be via `git clone <repo_url>` or copying the files through SCP. The project structure (see **Scaffolding Code** below) will include requirements or an environment file.
5. **Install Python Dependencies:** If the project has a `requirements.txt`, run `pip install -r requirements.txt` within the activated environment. If an `environment.yml` (conda environment file) is provided, use `conda env update -f environment.yml`. This will install libraries like FastAPI/Flask, boto3 (for AWS), openai, pinecone-client, etc., as required by the backend. The aim is to reproduce the development environment on the server.
6. **Configure Environment Variables:** Set the necessary environment variables on the EC2 instance. This includes:
   - `OPENAI_API_KEY` for OpenAI access,
   - `PINECONE_API_KEY` (and environment) for Pinecone access,

- AWS credentials or rely on IAM role,
- Any other config like table names or default parameters.These can be added to the shell profile or a `.env` file that the application loads. It's critical to secure these secrets; since this is a single server, simple methods like an `.env` file readable by the app (but not committed to code) are fine.

7. **Run the Service:** Launch the backend application. If it's a FastAPI app, for example, you might use Uvicorn: `uvicorn app:api --host 0.0.0.0 --port 80`. We configure it to listen on all interfaces (0.0.0.0) so that the EC2's public IP or domain will be accessible. For a production setup, you might run it behind an Nginx reverse proxy for stability and to serve it on port 80/443 (with SSL), but initially running directly on a port is fine. You can use a process manager or `nohup` to keep it running. A simple approach is to use `tmux` or `screen` to start the server in a long-lived session, or better, set up a systemd service for it.
8. **Verification:** Once running, test the endpoint. From your local machine, you might `curl` the EC2 API endpoint to ensure it returns a well-formed response. Also test a simple chat interaction to verify connectivity with OpenAI, DynamoDB, Pinecone, etc. Monitor the logs for any errors. On EC2, logs printed to stdout can be viewed via the console (or set up logging to CloudWatch).
9. **Monitoring & Maintenance:** With a manual deployment, you'll want to ensure the process stays alive. A basic health-check cron job can periodically hit a status endpoint of the API. If using systemd, make sure to enable the service to start on boot and restart on failure. Regularly update the code by pulling from git and restarting the service as needed (during a low-traffic period).

This manual deployment strategy emphasizes simplicity. We deliberately avoid Docker here – by using Conda on the host, we sidestep containerization overhead while still keeping dependencies isolated. This means we don't need to rebuild images for code changes; we can edit code or pull updates and just restart. It also avoids any file or hardware access complications that containers might introduce (everything runs natively on the VM)

stackoverflow.com

. The trade-off is that scaling beyond one machine or ensuring absolute consistency across environments might be harder than with containers, but for our

current scope, one well-configured EC2 is sufficient.

If the user base grows, we can then consider containerizing this app and using a service like ECS or EKS for horizontal scaling, or even AWS Lambda for a serverless approach. The codebase is structured in a way (stateless API server with external state in DynamoDB/Pinecone) that moving to those would not be too difficult. But until needed, we enjoy the ease of a straightforward EC2 deployment.

Using Anaconda for Execution

We specifically use Anaconda on the EC2 instance to create a consistent runtime environment. Anaconda/Miniconda allows us to specify exact library versions (via environment files) and reproduce the development environment on the server without dealing with system Python or package conflicts. It acts like a lightweight container. By activating the dedicated environment for our app, we ensure that only the required libraries (and correct versions) are present, reducing the chances of “it works on my machine” issues in production.

Anaconda also simplifies any scientific or ML dependencies. If we later add any heavy libraries (for example, for data processing or if hosting a small local model), conda can handle compiled binaries and CUDA dependencies more gracefully than pip in some cases. All these benefits come while still running directly on the host OS.

In short, **Conda gives us isolation and reproducibility**. The deployment instructions include creating the environment and installing requirements, which encapsulates our app. Should we need to update the app, we can update that environment (or create a new one) accordingly.

Using this approach, our deployment is effectively “containerized” by the environment – but without Docker. This avoids needing root privileges to manage containers or dealing with container networking. The application process runs as a normal process on EC2, which is straightforward to manage.