Stateless API

A "stateless" API means the model doesn't inherently remember past interactions. Each query is treated as a brand new one.

- **Solution: Conversation History Management** The most common solution is to have your application **store the conversation history**. For each new message from the user, you combine it with a summary of the past conversation and send the entire package to the LLM. This provides the necessary context for the model to generate a coherent, "stateful" response. For very long conversations, you can use another LLM call to periodically summarize the history to keep it from exceeding the context limit.

Not Trained on Your Data & Limited Data Size

LLMs are trained on vast public datasets, not your private documents. You also can't just paste a 500-page manual into a single prompt due to context window limits.

- **Solution: Retrieval-Augmented Generation (RAG)** This is the most powerful and popular technique to solve both problems. Instead of retraining the model, you provide it with the relevant information *at the time of the query*.
    1. **Index Your Data:** You take your private documents (e.g., company policies, product manuals, user data), break them into manageable chunks, and convert them into numerical representations called **embeddings** using an AI model.
    2. **Store Embeddings:** These embeddings are stored in a special database called a **vector database**.
    3. **Retrieve & Augment:** When a user asks a question, your system converts the question into an embedding and uses the vector database to find the most relevant chunks of your original documents (semantic search).
    4. **Generate Answer:** You then feed these relevant chunks to the LLM along with the original question and instruct it: "Answer this question using *only* the provided information."

    This approach securely uses your private data, sidesteps context window limits by only providing small, relevant pieces of data, and is much cheaper than retraining a model.

Prone to Hallucinations

Hallucinations are instances where the model confidently states incorrect or nonsensical "facts."

- **Solution 1: Grounding with RAG** The RAG technique described above is the best defense against hallucinations. By forcing the model to base its answers on specific, verified documents you provide, you are **grounding** it. You can even prompt it to provide citations from the source material, making its answers verifiable.
- **Solution 2: Adjusting Temperature** LLMs have a parameter called **temperature**, which controls randomness. A high temperature encourages more creative and diverse outputs, while a low temperature makes the output more deterministic and focused. For factual Q&A applications, setting the temperature to a low value (like `0.1` or even `0.0`) can significantly reduce the likelihood of hallucinations.

Not Aware of Your APIs & Real-Time Data

An LLM's knowledge is frozen at the time of its training, and it can't browse the web or interact with other software on its own.

- **Solution: Function Calling & Tool Use** Modern LLMs support **function calling** or **tool use**. You can describe a set of tools (like your company's API or a public weather API) to the LLM in its system prompt.
    1. The user asks a question that requires external data, like "What's the current ticket status for USER-123?" or "What's the weather like in Fairfield, Iowa right now?"
    2. The LLM recognizes that it needs to use a tool. Instead of answering, it outputs a structured JSON object, like: `{"tool": "get_ticket_status", "parameters": {"ticket_id": "USER-123"}}`.
    3. Your application code sees this, calls your actual internal API with the provided parameters, and gets the result (e.g., `{"status": "In Progress"}`).
    4. You then feed this result back to the LLM in a second call and ask it to formulate a natural language response for the user.

    This technique effectively gives the LLM access to any real-time data or external service you want, breaking it out of its static knowledge box.