Augment large language models with retrieval-augmented generation or fine-tuning

08/01/2025

This article explains how large language models (LLMs) can use extra data to provide better answers. By default, an LLM only knows what it learned during training. You can add real-time or private data to make it more useful.

There are two main ways to add this extra data:

- **Retrieval-augmented generation (RAG)**: Uses semantic search and contextual priming to find and add helpful information before the model answers. Learn more in Key concepts and considerations for building generative AI solutions.
- **Fine-tuning**: Retrains the LLM on a smaller, specific dataset so it gets better at certain tasks or topics.

The next sections break down both methods.

Understanding RAG

RAG enables the key "chat over my data" scenario. In this scenario, an organization has a potentially large corpus of textual content, like documents, documentation, and other proprietary data. It uses this corpus as the basis for answers to user prompts.

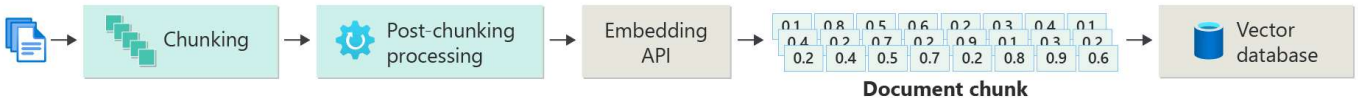
RAG lets you build chatbots that answer questions using your own documents. Here's how it works:

1. Store your documents (or parts of them, called *chunks*) in a database
2. Create an *embedding* for each chunk; a list of numbers that describe it
3. When someone asks a question, the system finds similar chunks
4. Send the relevant chunks along with the question to the LLM to create an answer

Creating an index of vectorized documents

Start by building a vector data store. This store holds the embeddings for each document or chunk. The following diagram shows the main steps to create a vectorized index of your

documents.



The diagram shows a *data pipeline*. This pipeline brings in data, processes it, and manages it for the system. It also prepares the data for storage in the vector database and makes sure it's in the right format for the LLM.

Embeddings drive the whole process. An embedding is a set of numbers that represents the meaning of words, sentences, or documents so a machine learning model can use them.

One way to create an embedding is to send your content to the Azure OpenAI Embeddings API. The API returns a vector—a list of numbers. Each number describes something about the content, like its topic, meaning, grammar, or style.

- Topic matter
- Semantic meaning
- Syntax and grammar
- Word and phrase usage
- Contextual relationships
- Style or tone

All these numbers together show where the content sits in a multi-dimensional space. Imagine a 3D graph, but with hundreds or thousands of dimensions. Computers can work with this kind of space, even if we can't draw it.

The Tutorial: Explore Azure OpenAI in Azure AI Foundry Models embeddings and document search provides a guide on how to use the Azure OpenAI Embeddings API to create embeddings for your documents.

Storing the vector and content

The next step involves storing the vector and the content (or a pointer to the content's location) and other metadata in a vector database. A vector database is like any other type of database, but with two key differences:

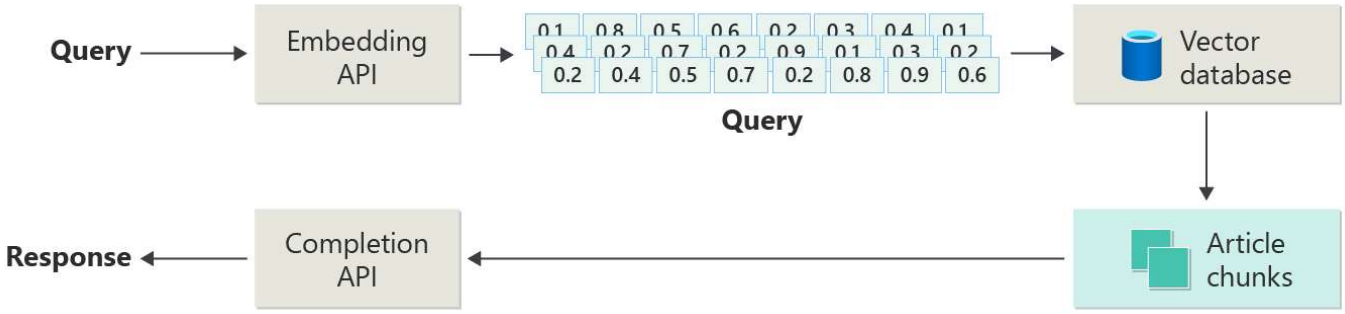
- Vector databases use a vector as an index to search for data
- Vector databases often use nearest neighbor algorithms, which can employ *cosine similarity* as a distance metric to find vectors that most closely match the search criteria

With the corpus of documents stored in a vector database, developers can build a *retriever component* to retrieve documents that match the user's query. The system uses this data to supply the LLM with what it needs to answer the user's query.

Answering queries by using your documents

A RAG system first uses semantic search to find articles that might be helpful to the LLM when it composes an answer. The next step involves sending the matching articles with the user's original prompt to the LLM to compose an answer.

The following diagram depicts a simple RAG implementation (sometimes called *naive RAG*):



In the diagram, a user submits a query. First, the system turns the user's prompt into an embedding. Then, it searches the vector database to find the documents or chunks that are most similar to the prompt.

Cosine similarity measures how close two vectors are by looking at the angle between them. A value near 1 means the vectors are very similar; a value near -1 means they're very different. This approach helps the system find documents with similar content.

Nearest neighbor algorithms find the vectors that are closest to a given point. The *k-nearest neighbors (KNN) algorithm* looks for the top *k* closest matches. Systems like recommendation engines often use KNN and cosine similarity together to find the best matches for a user's needs.

After the search, send the best matching content and the user's prompt to the LLM so it can generate a more relevant response.

Challenges and considerations

A RAG system comes with its own challenges:

- **Data privacy**: Handle user data responsibly, especially when retrieving or processing information from outside sources.
- **Computational requirements**: Expect both retrieval and generation steps to use significant computing resources.
- **Accuracy and relevance**: Focus on delivering accurate, relevant responses and watch for bias in your data or models.

Developers need to address these challenges to build RAG systems that are efficient, ethical, and valuable.

To learn more about building production-ready RAG systems, see Build advanced retrieval-augmented generation systems.

Want to try building a generative AI solution? Start with Get started with the chat using your own data sample for Python. Tutorials are also available for .NET, Java, and JavaScript.

Fine-tuning a model

Fine-tuning retrains an LLM on a smaller, domain-specific dataset after its initial training on a large, general dataset.

During pretraining, LLMs learn language structure, context, and general patterns from broad data. Fine-tuning teaches the model with new, focused data so it can perform better on specific tasks or topics. As it learns, the model updates its weights to handle the details of the new data.

Key benefits of fine-tuning

- **Specialization**: Fine-tuning helps the model do better on specific tasks, like analyzing legal or medical documents or handling customer service.
- **Efficiency**: Fine-tuning uses less data and fewer resources than training a model from scratch.
- **Adaptability**: Fine-tuning lets the model learn new tasks or domains not covered in the original training.
- **Improved performance**: Fine-tuning helps the model understand the language, style, or terminology of a new domain.
- **Personalization**: Fine-tuning can make the model's responses fit the needs or preferences of a user or organization.

Limitations and challenges

Fine-tuning also has some challenges:

- **Data requirement**: You need a large, high-quality dataset for your specific task or domain.
- **Risk of overfitting**: With a small dataset, the model might do well on training data but poorly on new data.
- **Cost and resources**: Fine-tuning still needs computing power, especially for large models or datasets.
- **Maintenance and updating**: You need to update fine-tuned models as your domain changes.
- **Model drift**: Fine-tuning for a specific task can make the model less effective at general language tasks.

Customize a model through fine-tuning explains how to fine-tune a model.

Fine-tuning vs. RAG

Fine-tuning and RAG both help LLMs work better, but each fits different needs. Pick the right approach based on your goals, the data and compute you have, and whether you want the model to specialize or stay general.

When to choose fine-tuning

- **Task-specific performance**: Pick fine-tuning when you need top results for a specific task and have enough domain data to avoid overfitting.
- **Control over data**: Use fine-tuning if you have unique or proprietary data that's very different from what the base model was pretrained on.
- **Stable content**: Choose fine-tuning if your task doesn't need constant updates with the latest information.

When to choose RAG

- **Dynamic or changing content**: Use RAG when you need the most current information, like for news or recent events.
- **Wide topic coverage**: Pick RAG if you want strong results across many topics, not just one area.

- **Limited resources**: Go with RAG if you don't have lots of data or compute for training, and the base model already does a good job.

Final thoughts for application design

Decide between fine-tuning and RAG based on what your app needs. Fine-tuning is best for specialized tasks, while RAG gives you flexibility and up-to-date content for dynamic scenarios.