

Bridging Knowledge with Retrieval-Augmented Generation (RAG)

Now that our LLM can remember key decisions and maintain coherent conversations, it's time to give it **real knowledge** — about *your* tasks, *your* calendar, *your* project docs.

Right now, your **TaskFriend** app is limited to what it hears during the chat. But what if it could:

- Pull in your upcoming calendar events?
- Read your task list from a database?
- Reference your company's onboarding guide?

That's where **Retrieval-Augmented Generation (RAG)** comes in.

With RAG, your LLM doesn't just rely on pre-trained knowledge — it retrieves relevant information from external sources and uses it to generate accurate, personalized responses.

The story so far...

Scenario: Users love chatting with **TaskFriend** as they can work with it to figure out how to plan their day. The features you've built so far like streaming responses, multi-turn conversations, and a professional system prompt have been a great hit! However, there's still a gap:

TaskFriend is unable to provide satisfactory answers to questions like:

"What tasks do I have due this week?"

Or:

"Can I reschedule my presentation prep if I go to the gym tomorrow morning?"

This is not because **TaskFriend** is not smart enough, but because it has **no access to the user's actual data**. It remembers what was *said* in the conversation, but not what's *true* in the user's world.

Goals

- Understand how RAG extends LLM knowledge beyond pre-training
- Build a document retrieval system using embeddings and vector search
- Inject retrieved context into prompts to generate informed responses
- Handle private, dynamic, or frequently updated information

Initializing the environment

Setting up the API key

Before we start work on in any notebook, we'll need to load the [API key for Model Studio](#). This ensures that we can call APIs of Qwen models we'll be using throughout this course.

If you're unsure about how to find your **Model Studio** API key, refer to the [00 Setting Up the Environment](#) file.

```
# Load Model Studio API key
import os
from config.load_key import load_key
load_key(
    confirmation=False
)
```

Setting up the LLM and embedding model

We set up Alibaba Cloud's **qwen-plus** as the LLM and DashScope's **text-embedding-v3** embedding model.

For this lesson, we'll be using **OpenAILike** instead of **OpenAI**, which we were using before this.

OpenAILike is a **LlamaIndex-specific wrapper** designed for OpenAI-compatible models, including:

- Model Studio
- Dashscope
- vLLM
- Ollama
- Local LLMs with OpenAI-compatible APIs

Note: DashScope takes <https://dashscope-intl.aliyuncs.com/api/v1> as its API endpoint instead of the <https://dashscope-intl.aliyuncs.com/compatible-mode/v1> we've been using so far.

```
# Set global settings
import time
import logging
import dashscope
from llama_index.core import Settings, VectorStoreIndex, SimpleDirectoryReader
from llama_index.embeddings.dashscope import DashScopeEmbedding
from llama_index.llms.openai_like import OpenAILike
from pathlib import Path

logging.getLogger().setLevel(logging.ERROR)

# Dashscope uses https://dashscope-intl.aliyuncs.com/api/v1
# instead of https://dashscope-intl.aliyuncs.com/compatible-mode/v1
dashscope.base_http_api_url = "https://dashscope-intl.aliyuncs.com/api/v1"

Settings.llm=OpenAILike(
    model="qwen-plus",
    api_base="https://dashscope-intl.aliyuncs.com/compatible-mode/v1",
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    is_chat_model=True
```

```
)

Settings.embed_model = DashScopeEmbedding(
    model_name="text-embedding-v2",
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    encoding_format="float"
)

print("✅ Global parameters set!")
```

Limitations of Standalone LLMs

Before diving into Retrieval-Augmented Generation (RAG), it's important to understand the **inherent limitations of standalone large language models (LLMs)**. While LLMs are remarkably capable at generating fluent, coherent text, they are not omniscient or perfectly reliable. Their behavior is shaped entirely by pre-training data and prompt input—meaning they lack dynamic access to new, private, or real-time information.

Understanding these limitations helps us make informed decisions about how to enhance LLMs for real-world applications like **TaskFriend**.


Key limitations

Knowledge cutoffs

Most LLMs are trained on static datasets with a fixed knowledge cutoff date. For example:

- **Alibaba Cloud's Qwen3**: April 2025
- **OpenAI's GPT 4.1**: June 2024
- **Google's Gemini 2.5 Pro**: Jan 2025
- **Anthropic's Claude 4 Opus**: March 2025

This means they **cannot know about events, products, or research published after that date**.

 **Example:** Ask a base LLM, "Who won the 2025 UEFA Champions League?" — it will either guess or invent an answer.


Even if the model is powerful, its knowledge is frozen in time.

No access to private or internal data

LLMs are not connected to your personal task list, company wiki, or internal CRM. Unless explicitly provided, they have **zero awareness** of:


- Your calendar
- Your project notes
- Company policies
- Customer records

This makes them **useless for personalized or enterprise tasks** without augmentation.

 **Security Note:** This isolation is actually a *feature* for privacy—but it means we must *intentionally* connect them to data when needed.

Hallucinations

When an LLM lacks sufficient information, it may **confidently generate false or fabricated content**—a phenomenon known as *hallucination*.

 **Example:**
User: "What's the deadline for the Q2 report?"
LLM: "The Q2 report is due on April 15."
Reality: No such report exists.

This is dangerous in productivity, legal, medical, or customer-facing applications.

Alternatives and their drawbacks

To overcome these limitations, several strategies exist—each with tradeoffs in cost, scalability, and maintenance.

Approach	Description	Limitations
Prompt Engineering	Crafting prompts to guide behavior (e.g., system prompts, few-shot examples)	Limited by context window; static; fragile to input changes
Fine-tuning	Retraining the model on new data to internalize knowledge or style	Expensive; hard to update; risks overfitting; not versionable
Pure Retrieval	Returning relevant documents or snippets without generation	Doesn't synthesize answers; requires user to read; no natural language output

While each approach has its place, none offers a perfect balance of **accuracy, freshness, cost, and ease of maintenance**.

The spectrum of LLM enhancement: Context vs. model optimization

There's a fundamental tradeoff in how we enhance LLMs:

Axis	Description
Model Optimization	Changing the model itself (e.g., fine-tuning, distillation, pre-training)
Context Optimization	Keeping the model fixed, but enriching the input context (e.g., RAG, prompt engineering, retrieval)

This leads to a strategic spectrum:



Source: [OpenAI - Optimizing LLM Accuracy](#)

What is Retrieval Augmented Generation (RAG)?

Retrieval-Augmented Generation (RAG) is a powerful technique that enhances large language models (LLMs) by integrating external knowledge into the generation process. In simple terms, RAG allows an AI model to look up relevant information from a knowledge base before generating a response. This makes the answers more accurate, up-to-date, and grounded in real-world data.

RAG is a hybrid approach that combines two key components:

- **Retrieval:** Finding the most relevant pieces of information from a large dataset.
- **Generation:** Using a language model to craft a natural language response based on that retrieved information.

This combination allows RAG to overcome some of the limitations of standalone LLMs, such as outdated knowledge or the tendency to hallucinate.

Why RAG matters

Traditional LLMs are trained on massive datasets, but once deployed, their knowledge is static. They cannot access real-time or private data, which limits their usefulness in many applications. RAG solves this by allowing the model to dynamically pull in the most relevant information at the time of inference.

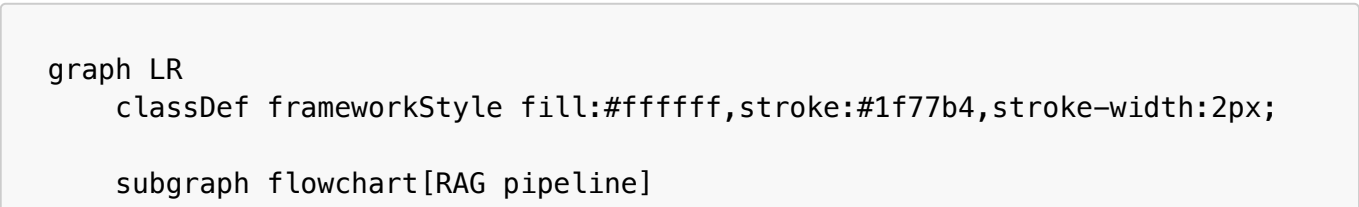
Problem	Without RAG	With RAG
"What tasks are due this week?"	Can't answer (no access to data)	Retrieves actual task list
"I need to break down my project"	General advice only	Uses real project notes
"Are there any company policies on remote work?"	Might hallucinate	Pulls from HR docs

This makes RAG especially valuable in:

- Enterprise environments (e.g., internal knowledge bases).
- Research and academic settings (e.g., answering questions from scientific papers).
- Customer support (e.g., answering queries using product documentation).

With RAG, our LLM transforms from a *reactive chatbot* into a **proactive knowledge assistant!**

How RAG works: The core pipeline



```

    subgraph "Your data"
      A[(Database)]
      B[Document]
      C[API]
    end

    U((User))
    I[Index]
    L[LLM]

    A -- structured --> I
    B -- unstructured --> I
    C -- programmatic --> I

    U -- query --> I
    I -- "prompt +
query +
relevant data" --> L
    L -- response --> U
  end

  class flowchat frameworkStyle;

```

The RAG system operates in **three distinct stages**:

Stage 1: Retrieval

When a user asks a question, the system first needs to find the most relevant pieces of information. This is done using a **retrieval model** that converts the query into a numerical representation (embedding) and searches a database of pre-embedded documents for the most similar matches.

For example, if the user asks “What is the capital of France?”, the system might retrieve a document that says “Paris is the capital of France.”

Stage 2: Augmentation

Once the system retrieves relevant documents, it injects them directly into the prompt as **context**. This transforms a generic query into a data-rich instruction the LLM can act on.

Example:

```

graph LR
  A["User Query:  
'What is on my shopping list?'" ] --> D["Prompt")
  B["Retrieved Context:  
'- Bread  
- Milk  
- Jam'"] --> D
  D --> C["LLM"]
  C --> Response["Response:  
'Your shopping list is:  
bread, milk, jam'"]

```

```
subgraph "Augmented Prompt"
  D --> E["'Based on the following:
- Bread
- Milk
- Jam
Answer: What tasks are due today?'"']
end

E --> C
```

Stage 3: Generation

The LLM generates a natural language response **grounded in the retrieved data**, reducing hallucinations and increasing accuracy.

Understanding Embeddings and Vector Search

At the heart of RAG is vector search, powered by embeddings — dense numerical representations of text.

What is an embedding?

An embedding is a fixed-length vector (e.g., length **1024**) that represents the semantic meaning of a piece of text. Similar texts have similar embeddings.

💡 Analogy: Think of documents as books in a library. Embeddings are like GPS coordinates — they help us find the closest matches.

How does vector search work?

- The user query is converted into an embedding.
- The system searches a vector index for the most similar embeddings.
- The top matching documents are returned and used to augment the prompt.

This process is powered by **cosine similarity** — a way to measure how alike two pieces of text are in meaning, even if their words differ.

Why "angle" matters more than "distance"

Imagine each document (and the query) lives as a point in a high-dimensional space — say, **1024** dimensions. You can't visualize that, but here's the key idea:

Cosine similarity looks at the angle between two vectors, not how far apart they are.

- If two vectors point in nearly the same direction → high similarity
- If they point in opposite directions → low similarity

Pro tip:

Think of embeddings like arrows shot from the origin.

Even if one arrow is longer (e.g., a longer document), what matters is where it's aiming.

Two arrows aiming in the same direction represent similar meanings — and cosine similarity captures that.

```
# Step 1: Use configured embedding model
embed_model = Settings.embed_model

# Step 2: Sample documents
docs = [
    "Paris is the capital of France.",
    "The Eiffel Tower is in Paris.",
    "Berlin is the capital of Germany.",
    "Tokyo is the capital of Japan.",
    "Machine learning is a subset of artificial intelligence."
]

# Step 3: Import and plot
from functions.vector_visualization import plot_vector_search

query = "What is the capital city of France?"

plot_vector_search(embed_model, docs, query)
```

Building the RAG System Step-by-Step

Let's walk through how to build a working RAG system using `llama_index`, `DashScope`, and our local documents.

Step 1: Load the Documents

LlamaIndex provides the `SimpleDirectoryReader`, which we will use to load files from the `./docs/taskfriend` directory.

Note: files may be separated into multiple pieces by `SimpleDirectoryReader`.
For our example, the embedder takes a maximum of 10 pieces.

```
# Load the documents
documents = SimpleDirectoryReader(
    input_dir="./docs/taskfriend",
    required_exts=[".pdf"],
    recursive=False
).load_data()

print(f"\n📄 Raw documents loaded: {len(documents)}")
for doc in documents:
    print(f" - {Path(doc.metadata['file_path']).name} (Text len: {len(doc.text)})")
```

Step 2: Build and Save the Index

Next, we use LlamaIndex's `VectorStoreIndex.from_documents()` function to build a vector index from the documents we loaded, and persist it to disk.

Pro tip: Persisting to disk helps improve the speed of our RAG since we don't need to rebuild the index every time.

```
# Build index from documents
print("Creating index...", end="", flush=True)
start_time = time.time()

index = VectorStoreIndex.from_documents(
    documents,
    embed_model=Settings.embed_model
)

load_time = time.time() - start_time
print(f" Done ✓ ({load_time:.1f} seconds)")

# Save index
index.storage_context.persist("knowledge_base/taskfriend")
print("✅ Index built and saved")
```

```
from llama_index.core import SimpleDirectoryReader
import logging

logging.getLogger().setLevel(logging.ERROR)

documents = SimpleDirectoryReader(
    input_dir="./docs/taskfriend",
    required_exts=[".pdf"],
    recursive=False
).load_data()

print("Raw chunks:")
for i, doc in enumerate(documents):
    print(f"\n--- Chunk {i} ---\n")
    print(doc.text)
```

Step 3: Query the RAG System

Now, use the `index.as_query_engine()` function to create the `query_engine`. Then, we'll build a wrapper for multi-turn conversations call it from our **TaskFriend** app.

```
from taskfriend.chat import chat_interface, wrap_rag_for_chat
```

```

# Build the query engine (used to implement RAG)
query_engine = index.as_query_engine(
    streaming=True,
    llm=Settings.llm,
)

# 📝 Define & initialize full_conversation
full_conversation = []

def get_rag_response(question, query_engine):
    try:
        # 🔍 Query the RAG engine
        response = query_engine.query(question)

        # 🧠 Extract the answer
        if hasattr(response, 'response'):
            answer = response.response
        else:
            answer = str(response)

        return answer

    except Exception as e:
        print(f"[RAG Error] {e}")
        return "[Error retrieving response]"

# Wrap function for compatibility
wrapped_rag = wrap_rag_for_chat(
    get_rag_response,
    query_engine=query_engine,
)

# Start chat with RAG
chat_interface(
    full_conversation=full_conversation,
    # client=client,
    call_llm_fn=wrapped_rag,
)

```

Now, try asking your model the following questions:

```

"What tasks are due today?"
"What tasks are due this week?"

```

Congratulations! You've successfully created your first RAG!

The model can now read from the `tasks.pdf` file in `./docs/taskfriend`, giving you answers about the tasks you have. Here's a table of the tasks (if you can't find `tasks.pdf`):

ID	Description	Type	Priority	Due	Status	Stakeholders	Notes
01	Finalize Q3 OKRs by 3pm	One-off	High	Today	Not Started	Department Leads, Executive Team, HRBP	Collaborate with department heads to align on measurable objectives. Ensure KPIs are SMART (Specific, Measurable, Achievable, Relevant, Time-bound). Submit final version to leadership for approval before 3 PM.
02	Prepare presentation for client review	One-off	High	This Week	Not Started	Client Success, Sales Lead, Product Manager	Focus on deliverables from Q2, highlight success metrics, and outline next steps. Use company-branded template. Include visual dashboards for performance trends. Share draft with manager by 1 PM for feedback.
03	Onboard new team member	One-off	High	Today	Not Started	HR, IT Support, Team Mentor, New Hire	Complete HR onboarding checklist: assign laptop, set up email, grant access to DingTalk, Jira, and internal wikis. Schedule intro meetings with team members. Assign mentor for first 30 days. Send welcome email with onboarding schedule.

ID	Description	Type	Priority	Due	Status	Stakeholders	Notes
04	Review team feedback survey results	One-off	Med	This Week	Not Started	Team Members, People Ops, Department Leads	Analyze anonymous feedback from recent engagement survey. Identify top 3 pain points and 2 strengths. Summarize findings in a report and propose action items for team improvement. Present insights in next leadership meeting.
05	Update project roadmap in Dingtalk	Recurring	Med	Today	Not Started	Project Leads, Engineering, Design, QA	Sync with project leads to reflect latest timelines, milestones, and resource allocations. Highlight any delays or risks. Ensure all stakeholders are tagged and notified upon update. Use Gantt view for clarity.
06	Schedule 1:1s with team	One-off	Med	This Month	Not Started	All Team Members, Team Leads	Book 30-minute slots with each team member over the next 4 weeks. Focus agenda on career development, workload balance, and feedback. Send calendar invites with pre-read form to gather talking points in advance.

ID	Description	Type	Priority	Due	Status	Stakeholders	Notes
07	Research AI tools for task automation	One-off	Low	This Year	Not Started	Tech Team, Security, Finance, Admin Ops	Investigate tools like Notion AI, Zapier, Microsoft Copilot, and Tongyi Qwen for automating repetitive tasks (e.g., email sorting, report generation). Evaluate cost, integration ease, and security compliance. Prepare comparison matrix and present by Q4.
08	Clean up email inbox	Recurring	Low	This Year	Not Started	Personal task – no external stakeholders	Archive old threads, delete spam, and create filters for newsletters and notifications. Aim to keep inbox under 100 messages. Use rules to auto-sort by project or sender. Repeat quarterly.
09	Call bank regarding home loan	One-off	High	This Week	Not Started	Partner, Financial Advisor, Bank Officer	Contact customer service to inquire about refinancing options. Compare current interest rate with market rates. Ask about early repayment penalties and eligibility for better terms. Bring loan agreement and ID to the call. Goal: reduce monthly payment or term.

ID	Description	Type	Priority	Due	Status	Stakeholders	Notes
10	Weekly report: Project Phoenix	Recurring	Med	This Week	Started	Project Manager, Stakeholder Group, Execs	Compile progress on deliverables, blockers, and resource usage. Include burn-down chart and risk log. Share with stakeholders via DingTalk by Friday EOD. Tag project manager for review.
11	Organize desktop files	Recurring	Low	This Year	Not Started	Personal task – no external stakeholders	Create structured folders: Projects, Finance, Personal, Archives. Move all loose files into appropriate categories. Delete duplicates and outdated drafts. Backup critical files to cloud storage. Repeat every 6 months.
12	Develop 3-year plan	One-off	Med	This Year	Not Started	Executive Leadership, Strategy Team, PMO	Based on company strategy shifts and market trends, draft a long-term vision for the team. Include goals for talent development, technology adoption, and innovation. Align with executive leadership's roadmap. Present draft at annual planning retreat.

ID	Description	Type	Priority	Due	Status	Stakeholders	Notes
13	Plan for trip to Norway	One-off	Med	This Year	Started	Partner, Children, Travel Agent (optional)	Research best time to visit (June–August). Estimate budget for flights, stays, and activities. Apply for passport/visas if needed. Look into family-friendly tours and outdoor experiences (nords, hiking, Northern Lights). Book initial flights by end of Q3.
14	Write thank-you letter to penpal in Korea	One-off	Low	Today	Started	Penpal	Thank penpal for the help they provided.

However, you'll notice that your RAG is not perfect – the answer it gave you isn't representative of all the tasks you have. And as you continue to talk to **TaskFriend**, you'll realize that there are some questions it still can't answer correctly. We'll cover this in the next chapter.

What's next?

Quiz yourself!

► 1. Which of the following is a key limitation of standalone LLMs that RAG helps solve?

- A) High API costs
- B) Inability to generate fluent text
- C) Lack of access to private or real-time data
- D) Slow inference speed

View answer →

✔ **Correct answer:** C) Lack of access to private or real-time data

📝 **Explanation :**

- RAG enables LLMs to retrieve and use up-to-date, user-specific information (e.g., tasks, calendars).

► 2. In the RAG pipeline, what happens during the "Augmentation" stage?

- A) Embeddings are retrained
- B) The model fine-tunes on new documents
- C) The user is shown raw search results
- D) Retrieved documents are added to the prompt as context

View answer →



Correct answer: D) Retrieved documents are added to the prompt as context



Explanation :

- This allows the LLM to generate responses grounded in actual data.

Takeaways

- **Limitations of standalone LLMs**
 - **Knowledge cutoffs** mean LLMs cannot know about events or data after their training date (e.g., Qwen3: April 2025).
 - **No access to private or internal data** — LLMs don't see your calendar, tasks, or company docs unless explicitly provided.
 - **Hallucinations** occur when models lack information and invent plausible-sounding but false answers.
 - **Alternatives have tradeoffs:**
 - *Prompt engineering*: Limited by context window.
 - *Fine-tuning*: Expensive, hard to update.
 - *Pure retrieval*: Returns raw text, no natural language synthesis.
 - **RAG solves these** by dynamically injecting real, up-to-date, user-specific context at inference time.
- **RAG systems**
 - **RAG bridges the gap** between general knowledge and specific, private, or real-time data.
 - It combines two powerful components:
 - **Retrieval**: Find relevant documents from a knowledge base.
 - **Generation**: Use an LLM to generate a natural language response based on retrieved content.
 - **RAG transforms LLMs** from static chatbots into dynamic knowledge assistants.
 - It enables accurate, personalized responses to questions like:
 - "What tasks are due this week?"
 - "Can I reschedule my presentation prep?"
 - **RAG is context optimization**, not model optimization — the LLM stays fixed, but the input is enriched.

- **Embeddings and vector search**

- **Embeddings** are dense numerical vectors (e.g., length 1024) that represent semantic meaning.
- **Similar texts have similar embeddings** — this enables semantic search beyond keyword matching.
- **Vector search** finds the most relevant documents by comparing query and document embeddings.
- **Cosine similarity** measures semantic alignment by the *angle* between vectors, not distance:
 - Small angle → high similarity
 - Opposite directions → low similarity
- **Embeddings allow the system to understand** that “capital of France” and “Paris” are related, even if the words don’t match exactly.

- **Building a RAG system**

- **Step 1: Load documents** using tools like **SimpleDirectoryReader** to ingest PDFs, text files, or APIs.
- **Step 2: Build a vector index** — convert documents into embeddings and store them for fast retrieval.
- **Step 3: Persist the index** to disk so it doesn’t need to be rebuilt every time.
- **Step 4: Query with augmentation** — retrieve relevant context and inject it into the prompt.
- **The RAG pipeline:**
 1. **Retrieval:** Convert query to embedding → find top-matching documents.
 2. **Augmentation:** Add retrieved content to the prompt as context.
 3. **Generation:** LLM generates a grounded, accurate response.
- **RAG is iterative** — your first version may not be perfect, but it’s a foundation for improvement.