

RAG Optimization Techniques

Now that you've built a basic RAG system and seen how it improves your LLM's ability to answer questions using your own data, it's time to take it further.

In this chapter, we'll explore advanced RAG techniques that help you:

- Retrieve more accurate and relevant documents
- Reduce hallucinations
- Improve answer quality
- Handle complex or ambiguous queries

The story so far...

Scenario: Users are loving the new feature where they can directly discuss their day with **TaskFriend**. However, they've noticed that **TaskFriend** sometimes fails to retrieve the information they're looking for.

Goals

- Improve **TaskFriend's** ability to retrieve accurate and relevant tasks
- Reduce incorrect answers caused by missing or irrelevant context
- Help **TaskFriend** understand complex or ambiguous queries like "I'm stuck – what should I do next?"
- Teach **TaskFriend** to provide more helpful, grounded answers by optimizing the RAG pipeline

Intitializing 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 client

We'll use the openai-compatible interface provided by DashScope to interact with Qwen models (or other models we use throughout the course).

```

from openai import OpenAI

client = OpenAI(
    # If environment variables are not configured, replace the following
    line with: api_key="sk-xxx",
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    base_url="https://dashscope-intl.aliyuncs.com/compatible-mode/v1",
)

```

```

# Set global settings
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

# 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!")

```

Setting up global parameters

We'll use the same global parameters we used in our previous chapter

```

def highlight_words(text, words_to_highlight, emoji="★"):
    for word in words_to_highlight:
        if word in text:
            text = text.replace(word, f"{emoji}{word}{emoji}")
    return text

def get_rag_response_with_info(query, query_engine, highlight=None):
    print("🚀 TaskFriend Conversation (single-round)")

```

```

print("-" * 50)
print(f'👤 You: {query}')

try:
    # 🔍 Query the RAG engine
    response = query_engine.query(query)

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

    # 📖 Show source references AFTER the answer
    print("🤖 TaskFriend:", answer)
    print('\n\n' + '=' * 50)
    print('📖 References\n')

    highlight = highlight or []
    for i, source_node in enumerate(response.source_nodes, start=1):
        print(f'Chunk {i}:')

        # Highlight words in chunk
        highlighted_text = highlight_words(source_node.text,
highlight)
        print(highlighted_text)
        print()
    print('=' * 50)

    return answer

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

```

```

from llama_index.core import StorageContext, load_index_from_storage

persist_path="./knowledge_base/taskfriend"

# Import index ("knowledgebase") we built last chapter,
storage_context = StorageContext.from_defaults(
    persist_dir=persist_path,
)

index = load_index_from_storage(
    storage_context,
    embed_model=Settings.embed_model
)

print(f"✅ Index loaded from `{persist_path}`!")

# Build the query engine (used to implement RAG)

```

```

query_engine = index.as_query_engine(
    streaming=False,
    llm=Settings.llm,
    embeddings=Settings.embed_model,
)
print("✅ Query engine built!")

```

Basic Strategies for Improving RAG Performance

Now that you've learned how to evaluate your RAG system and diagnose its weak spots, it's time to roll up your sleeves and fix them. You've seen the symptoms—low recall, poor precision, hallucinations—and now it's time for the cure.

Improving a RAG pipeline isn't about magic; it's about making smart, targeted adjustments to its core components. The following diagram provides a simplified workflow of the basic optimization strategies you'll learn about in this chapter. These are also the most impactful levers you can pull right now to tune the performance of your RAG system.

```

graph TD
    classDef frameworkStyle fill:#ffffff,stroke:#1f77b4,stroke-width:2px;
    classDef retrieval fill:#e3f2fd,stroke:#1976d2,stroke-width:2px;
    classDef embedding fill:#e8f5e8,stroke:#388e3c,stroke-width:2px;
    classDef chunking fill:#fff3e0,stroke:#f57c00,stroke-width:2px;

    subgraph flowchart [RAG Optimization Pipeline]
        A[User Query] --> B[Retrieval]
        subgraph "Optimization: similarity_top_k"
            direction TB
            B --> C[Vector Store Search]
            C --> D[Retrieve Top-K Chunks]
            style D fill:#e3f2fd,stroke:#1976d2
        end
        end

        subgraph "Optimization: Embeddings"
            direction TB
            E[Embedding Model] --> C
            style E fill:#e8f5e8,stroke:#388e3c
        end
        end

        D --> F[Assemble Context]
        subgraph "Optimization: Chunking"
            direction TB
            G[Chunking Strategy] --> H[Source Documents]
            style G fill:#fff3e0,stroke:#f57c00
        end
        end

        H --> I[Create Vector Index]
        I --> E
        I --> C
    
```

```

    F --> J[LLM Generation]
    J --> K[Final Answer]
end

class flowchat frameworkStyle

class D retrieval
class E embedding
class G chunking

style A fill:#f3e5f5,stroke:#7b1fa2
style K fill:#f3e5f5,stroke:#7b1fa2

```

1. Fine-tuning your retrieval with the `similarity_top_k` parameter,
2. Choosing the right embeddings to power your semantic search,
3. Mastering your chunking strategy to ensure your knowledge is well-structured.

These aren't just technical tweaks—they're foundational decisions that shape how your AI "thinks" and finds information. We'll explore each one with practical examples and show you exactly how they can turn a struggling RAG into a high-performing one.

Let's dive in and start making TaskFriend smarter.

`similarity_top_k`: Controlling what your RAG sees

Let's go back to a familiar example - we've ran through this in the previous chapter. Let's run it again.

```

# User query
query = "How many tasks do I need to complete today?"

# Choose words to highlight
highlight = ["Today"]

response = get_rag_response_with_info(query, query_engine=query_engine,
highlight=highlight)

```

Now we can see that **TaskFriend**:

- Only gets two chunks of context from the index
- Within those two chunks, only one occurrence of the word `today` is present
- Believes that only one task is due today from the retrieved context

So why is **TaskFriend** only seeing two chunks? What happened to the rest of the context?

This can be explained by how the `query_engine` function works.

```

query_engine = index.as_query_engine(
    ...

```

```
    similarity_top_k=2          # ← The default value is 2
)
```

The `query_engine` takes the `similarity_top_k` argument, whose default value is `2`. This means that it will return the top 2 most semantically similar text chunks from the index based on the input query.

As a result, even if there are more relevant chunks in the index—such as additional tasks due today—only the two that are most closely aligned with the query in terms of semantic similarity are retrieved and passed to the LLM for reasoning. Any information outside these top-k chunks, including other instances of tasks due today, is effectively invisible to TaskFriend during its response generation.

This behavior highlights a key limitation of retrieval-augmented systems: the model can only reason over the context that is retrieved, not the entire knowledge base. If relevant information isn't among the top-k results, it won't be considered—leading to incomplete or potentially incorrect answers.

In short, this is what we're effectively getting:

```
[Vector Index of Task Chunks]
|
|— Chunk 0: "Submit Q3 report – due today"           ← ✓ Retrieved (High
relevance)
|— Chunk 1: "Team meeting – today at 3 PM"           ← ✓ Retrieved
|— Chunk 2: "Renew cloud subscription – due today"    ← ✗ Not retrieved
|— Chunk 3: "Update documentation – due tomorrow"
|— Chunk 4: "Onboard new intern – due next week"
|
↓
[LLM Input Context]

"""
You need to complete one task today:
– Submit Q3 report – due today
"""

🧠 LLM Output: "You have 2 tasks due today."
✗ Misses: "Renew cloud subscription"
```

Now, let's try playing around with the `similarity_top_k` argument:

```
# Rebuild query engine with custom similarity_top_k argument
query_engine = index.as_query_engine(
    streaming=True,
    llm=Settings.llm,
    embeddings=Settings.embed_model,
    similarity_top_k=5          # ← The default value is 2
)
print("✓ Query engine built!")
```

```
# User query
query = "How many tasks do I need to complete today?"

# Choose words to highlight
highlight = ["Today"]

response = get_rag_response_with_info(query, query_engine=query_engine,
highlight=highlight)
```

Great! We've managed to retrieve all the tasks that are due today, as we've managed to retrieve all the relevant chunks. The `similarity_top_k` argument does provide a quick and easy way for us to retrieve what we need - but is this really the best way?

By this logic, we could simply set `similarity_top_k` to the size of the entire index and retrieve all chunks, guaranteeing we never miss anything. However, there are real-world limitations that make this approach impractical:

Coupling this with our evaluation metrics we learn in the last chapter could give us some insights:

```
from langchain_openai import ChatOpenAI
from langchain_dashscope import DashScopeEmbeddings # Correct import
import os
import time

dashscope.base_http_api_url = "https://dashscope-intl.aliyuncs.com/api/v1"

# LangChain LLM for Ragas
ragas_llm = ChatOpenAI(
    model="qwen-plus",
    base_url="https://dashscope-intl.aliyuncs.com/compatible-mode/v1",
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    timeout=60,
    max_retries=3
)

# LangChain Embeddings for Ragas
ragas_embeddings = DashScopeEmbeddings(
    model="text-embedding-v2",
    api_key=os.getenv("DASHSCOPE_API_KEY"),
)

print("✅ LLM and Embeddings pipeline for Ragas successfully built!")
```

```
# Build query engine with similarity_top_k argument
query_engine = index.as_query_engine(
    streaming=True,
    llm=Settings.llm,
    similarity_top_k=5 # ← The default value is 2
```

```
)
print("✅ Query engine built!")
```

```
test_cases = [
    {
        "question": "Which tasks are due today?",
        "ground_truth": (
            "You have four tasks due today:"
            "1. Finalize Q3 OKRs is due today, "
            "2. Onboard new team member is due today, "
            "3. Update project roadmap is due today, "
            "4. Write thank-you letter is due today."
        )
    }
]
```

```
from functions.html_table import create_html_table

def run_test_cases(query_engine, test_cases):
    results = {
        "question": [],
        "answer": [],
        "contexts": [],
        "ground_truth": []
    }

    print("Generating answers based on test cases...", end="", flush=True)
    start_time = time.time()

    for case in test_cases:
        # Run query
        response = query_engine.query(case["question"])

        # Extract answer and context
        answer = str(response).strip()
        # Ensure contexts is a list of strings
        contexts = [node.get_content().strip() for node in
                    response.source_nodes]

        # Format contexts for display with [] and line breaks
        contexts_display = [f"[{ctx}]" for ctx in contexts]

        results["question"].append(case["question"])
        results["answer"].append(answer)
        results["contexts"].append(contexts)
        results["ground_truth"].append(case["ground_truth"])
```

```

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

    return results

# Run the test
test_results = run_test_cases(query_engine, test_cases)

```

```

from datasets import Dataset

# Convert to Dataset
eval_dataset = Dataset.from_dict(
    {
        "question": test_results["question"],
        "answer": test_results["answer"],
        "contexts": test_results["contexts"],
        "ground_truth": test_results["ground_truth"]
    }
)

print("Results converted to dataset!")

```

```

from ragas import evaluate
from ragas.metrics import (
    context_recall,
    context_precision,
    faithfulness,
    answer_correctness
)
from functions.rag_eval_table import create_rag_evaluation_table
from IPython.display import display

# Use the LangChain-compatible LLM for Ragas
results = evaluate(
    dataset=eval_dataset,
    metrics=[
        answer_correctness,
        context_recall,
        context_precision,
        faithfulness,
    ],
    llm=ragas_llm,
    embeddings=ragas_embeddings
)

# Convert results to pandas DataFrame
results_df = results.to_pandas()

```

```
# Extract the input fields from the dataset to add context
results_df["question"] = eval_dataset["question"]
results_df["ground_truth"] = eval_dataset["ground_truth"]
results_df["answer"] = eval_dataset["answer"]
results_df["contexts"] = eval_dataset["contexts"]

# Reorder columns for better readability
results_df = results_df[[
    "question",
    "answer",
    "ground_truth",
    "contexts",
    "answer_correctness",
    "context_recall",
    "context_precision",
    "faithfulness"
]]

print("\n📊 Evaluation Results:")
# display(results_df)
create_rag_evaluation_table(results_df)
```

Based on the results we can make several inferences (results may vary, but should be in and around these general values):

Metric	Score	Explanation	Conclusion
Answer correctness	Med (0.54)	<ul style="list-style-type: none">The model correctly identified 2 out of 4 tasks due today: Finalize Q3 OKRs and Update project roadmap.It missed two tasks: Onboard new team member and Write thank-you letter.	The answer is partially correct; it lacks completeness.
Context recall	High (0.80)	<ul style="list-style-type: none">Out of the 4 tasks that should have been retrieved (those marked as due today), 3 were found in the contexts: Finalize Q3 OKRs, Update project roadmap, and Onboard new team member.Only Write thank-you letter was missing.	The retriever is mostly effective, but still missing one relevant context.
Context precision	Zero (0.00)	<ul style="list-style-type: none">The model retrieved 5 total contexts, but only 2 (Finalize Q3 OKRs, Update project roadmap) were actually due today.The other 3 were either due later or irrelevant (Clean up inbox, Call bank, etc.).	The retriever is not precise — it's pulling in a lot of irrelevant content.

Metric	Score	Explanation	Conclusion
Faithfulness	Med (0.67)	<ul style="list-style-type: none">The model included 3 tasks in its answer, but only 2 were actually in the retrieved contexts.The task Send performance review draft to manager by 1 PM was not in the contexts and should not have been included.	The answer is mostly faithful, but includes one unsupported task — slightly hallucinated.

The RAG Pipeline: From Retrieval to Augmentation to Generation

Now that you’ve learned how adjusting **similarity_top_k** can help **TaskFriend** see more of your tasks, it's time to learn about the strategies you can use to improve RAG performance. But before we begin, we'll need to understand the RAG pipeline.

```
graph LR
    subgraph Ingest["Document Ingestion (LlamaIndex)"]
        direction TB
        KB[Knowledgebase] --> PreP["Pre-process documents"]
    end
    subgraph RAG["RAG (LlamaIndex + Langchain)"]
        direction LR
        PreP -- Docs --> Emb[Embedding Model]
        Emb -- Doc embeddings --> vDB[(VectorDB)]
        vDB -- "Prompt + query + retrieved context" --> LLM
    end
    Usr[User] --> Chatbot
    Chatbot -- User query --> Emb
    Emb -- Embedded query --> vDB
    subgraph Input
        direction LR
        Usr
        Chatbot
    end
    LLM -- Generated response --> Chatbot
```

How it works

1. Document Ingestion
- The process begins with your knowledge base (e.g., task lists, documents, FAQs).

- These documents are passed to the Pre-process documents stage, where they are cleaned, split into chunks, and prepared for indexing.
2. Embedding & Storage
- Chunks are converted into numerical vectors using an Embedding Model.
 - These document embeddings are stored in a Vector Database for fast semantic search.
3. User Query Flow
- When a User submits a query, the Chatbot sends it to the same Embedding Model.
 - The query is transformed into an embedded query — a vector in the same space as the documents.
4. Retrieval
- The system searches the VectorDB to find the most relevant document chunks (via similarity search).
 - Retrieved context is combined with the original query and a prompt template.
5. Generation
- This enriched input — "Prompt + query + retrieved context" — is sent to the LLM.
 - The LLM generates a context-aware response, which is sent back to the Chatbot and displayed to the user.

Pre-processing: Giving Your RAG the Right "Vision" Into Your Data

Pre-processing is where your RAG system gains its "vision" — the ability to see the right information at the right level of detail. It consists of two core phases: document preparation and chunking.

Matching the right knowledge to user needs

Before optimizing retrieval or generation, ensure your RAG system has the right foundation: good knowledge.

Think of it this way:

- Users ask questions based on their needs — this forms the **intent space**.
- Your knowledge base contains answers — this forms the **knowledge space**.

Aspect	Intent Space	Knowledge Space
Definition	The set of all possible user goals or underlying needs expressed through queries.	The collection of facts, documents, and structured information the RAG system can access.
Focus	"What does the user want?" — Understanding purpose, context, and query type.	"What does the system know?" — Providing factual, retrievable content.
Examples	"Compare two products", "Reset my password", "Explain how X works"	Internal docs, FAQs, product specs, research papers, databases

Aspect	Intent Space	Knowledge Space
Role in RAG	Guides retrieval by classifying the query type (e.g., factual, comparative, procedural). Enables intent-aware search.	Serves as the source for retrieval — where relevant context is pulled to augment the LLM prompt.
Key mechanism	Natural Language Understanding (NLU), intent classification models	Vector databases, semantic search, knowledge graphs
If misaligned	System misunderstands the user’s goal → retrieves wrong kind of info	System lacks supporting data → leads to hallucinations or no results
Optimization strategy	Use query rewriting, intent tagging, or multi-query retrieval to better capture user intent	Improve document coverage, clean outdated content, enhance chunking for findability

For your RAG app to work well, these two spaces must overlap.

Scenario	What it means?	How to fix
Overlap	The question is covered by your docs. → RAG can answer well.	Improve content quality and retrieval accuracy.
Missing Knowledge	The user asks something new (e.g., a new product). → RAG may "hallucinate".	Add new documents. Update your knowledge base regularly.
Irrelevant Results	Wrong or unrelated content is retrieved. → Confuses the LLM.	Improve chunking, embeddings, or retrieval logic. Remove outdated content.

Key Insight:
Intent space tells the RAG system what to look for. **Knowledge space** tells it where to look. Only when **both are aligned** can your RAG deliver accurate, relevant, and trustworthy responses.

Document preparation: Start with clean, structured input

The quality of your output depends on the quality of your input. Real-world documents come in many formats — PDFs, Word files, Keynotes — and often contain complex elements like tables, images, and mixed layouts.

Common Challenges & Solutions

Issue	Impact	Solution
Unsupported file formats (e.g., <code>.key</code> , <code>.py</code> , <code>.json</code> , etc.)	Your parser can't "read" the file, leading to gaps in knowledge space	Convert files to standard formats like <code>.pdf</code> , <code>.txt</code> , <code>.md</code> , etc.
Poorly extracted content (e.g., broken tables, missing images)	Critical data is lost or distorted	Choose parsers that preserve structure, or use LLMs to post-process and polish the output

Issue	Impact	Solution
Images with embedded text (e.g., screenshots of commands)	Text isn't captured by standard OCR	Use vision-language models (VLMs) to interpret image content and include it in your knowledge base

Converting input documents to markdown

Formats like `.pdf` are easy for humans to read, but may be challenging for machines to understand. One of the methods you can use to improve how your application understands documents is to convert it into markdown format (`.md`). The markdown format brings various benefits, such as:

- Improved structure and hierarchy
 - Markdown supports headings (`#`, `##`, etc.), lists, code blocks, and emphasis.
 - This helps preserve semantic structure, making it easier for chunking strategies to split content meaningfully (e.g., by section).
 - Better chunking → more contextually coherent passages → improved retrieval relevance.
- Enhanced Semantic Clarity
 - Headings and subheadings act as natural metadata.
 - A query like "Explain the methodology" can more easily match a chunk starting with `## Methodology` than unstructured text.
- Preservation of Key Formatting
 - Code blocks (````) and inline code (`) help distinguish technical terms or commands, which can be crucial for accurate retrieval in technical documentation.`
 - Lists and tables (in extended Markdown) preserve relationships between items.

However, it is also not without its drawbacks:

- **Poorly formatted markdown:** If headings are misused or inconsistent, the structure becomes misleading.
- **Over-chunking:** Too many small sections may break context, hurting retrieval quality.
- **Loss of rich metadata:** Markdown lacks native support for metadata (e.g., author, date), unless added via front matter (e.g., YAML).

For easier reference, here's a table on the the pros and cons of converting documents to markdown:

Aspect	Impact of markdown on RAG retrieval	Notes / Best practices
Document Structure	✔ Improves	Uses headings (<code>#</code> , <code>##</code>) to create logical hierarchy, helping retrieval systems identify relevant sections.
Semantic Clarity	✔ Improves	Headings act as natural query-matching cues (e.g., <code>## Methodology</code> matches "Tell me about the methodology").

Aspect	Impact of markdown on RAG retrieval	Notes / Best practices
Text Chunking	✔ Improves	Enables smarter, context-aware chunking (e.g., by section), leading to more relevant retrieved passages.
Formatting Preservation	✔ Improves	Preserves code blocks, lists, and emphasis, which can be critical for technical or structured content.
Processing Efficiency	✔ Improves	Lightweight and plain-text—easier to parse, index, and clean compared to PDF/HTML.
Metadata Support	⚠ Limited	Native Markdown lacks metadata; use YAML front matter (e.g., <code>title:</code> , <code>source:</code>) to add context.
Poor Formatting Risk	✖ Can Hurt	Inconsistent or incorrect headings/lists mislead retrieval. Quality matters more than format.
Over-chunking Risk	⚠ Can Hurt	Excessively small sections may break context. Balance chunk size and coherence.
Not a Standalone Fix	⚠ Neutral	Markdown alone won't fix bad content or weak embeddings. Works best with high-quality text and models.
Best Practices	—	<ul style="list-style-type: none">• Use semantic, consistent headings• Chunk by section boundaries• Preserve code/lists/tables• Add metadata via front matter• Test retrieval performance after conversion

Now, let's do a quick experiment to convert our `./docs/taskfriend/tasks.pdf` to markdown:

```
import fitz # PyMuPDF
import pymupdf4llm # For Markdown conversion

# Convert PDF to Markdown
def pdf_to_md(pdf_path):
    doc = fitz.open(pdf_path)
    md_text = pymupdf4llm.to_markdown(doc)
    doc.close()
    return md_text

# Convert your file
md_content = pdf_to_md("./docs/taskfriend/tasks.pdf")

print("📄 Raw Markdown Output:")
print("-" * 100)
print(md_content)
```

As you can see, the raw output is similar to the chunks we were getting previously - with spelling issues and the table "breaking" across pages. These fall into the drawbacks of markdown conversion we just covered. However - this is not the end. We can actually leverage LLMs for this task: By feeding our LLM with the raw `.md` output, we can get the LLM to fix most of these issues:

```
from dashscope import Generation

def md_polisher(markdown_text, model="qwen-plus"):
    prompt = f"""
    The following Markdown text is converted from a PDF. Please fix and polish it:
    1. Fix heading hierarchy and missing info.
    2. Correct inconsistent context or tables.
    3. Ensure lists and code blocks are properly formatted.
    4. Do NOT invent new content.
    5. Output only the cleaned Markdown.

    Content:
    {markdown_text}
    """

    messages = [{'role': 'user', 'content': prompt}]

    response = Generation.call(
        model=model,
        messages=messages,
        result_format='message',
        stream=True,
        incremental_output=True # Good for long responses
    )

    result = ""
    print("✨ Polished Markdown:")
    print("-" * 50)
    for chunk in response:
        content = chunk.output.choices[0].message.content
        print(content, end='')
        result += content
    return result

# Polish the output
polished_md = md_polisher(md_content)

# Optional: Save to file
with open("./docs/tasks_polished.md", "w", encoding="utf-8") as f:
    f.write(polished_md)

print("\n\n" + "-" * 50)
print("✅ Saved polished Markdown to ./docs/tasks_polished.md")
```

Alternatively, if you prefer it to be as a `.md` list format, you can tweak the prompt to polish it up based on your requirements.

```
def md_polisher_to_list(markdown_text, model="qwen-plus"):
    prompt = f"""
The following Markdown text is converted from a PDF. Please fix and polish it:
1. Format as task list, not table
2. Each list item must contain all information that can be used independently
3. Must be human-readable
4. Must not exceed 3 sentences per list item
5. Do NOT invent new content.
6. Output only the cleaned Markdown.

## Examples
1. Task 01: <task_name>, <priority> priority, <recurring> task that is <status> due <due_date>. <description>. Stakeholders <stakeholders>
2.

Content:
{markdown_text}
"""
    messages = [{'role': 'user', 'content': prompt}]

    response = Generation.call(
        model=model,
        messages=messages,
        result_format='message',
        stream=True,
        incremental_output=True # Good for long responses
    )

    result = ""
    print("✨ Polished Markdown:")
    print("-" * 50)
    for chunk in response:
        content = chunk.output.choices[0].message.content
        print(content, end='')
        result += content
    return result

# Polish the output
polished_md = md_polisher_to_list(md_content)

# Optional: Save to file
with open("./docs/tasks_list_polished.md", "w", encoding="utf-8") as f:
    f.write(polished_md)

print("\n\n" + "-" * 50)
print("✅ Saved polished Markdown to ./docs/tasks_list_polished.md")
```

This is a simple yet effective pipeline in creating a `.md` file from a `.pdf`, as well as fixing any potential issues, so you can maximize the benefits of a more structured file format for your RAG system.

Chunking: Finding the sweet spot between context and precision

Chunking determines how your documents are split before being embedded and stored. Too small, and you lose context. Too large, and you introduce noise.

There's no *one-size-fits-all method* — but here are the most effective strategies:

Method	Best for	Key benefit
Sentence splitting	General use cases	Preserves sentence boundaries; clean and predictable
Token splitting	Models with strict context limits	Precise control over token count
Sentence window	Context-heavy queries	Adds surrounding sentences as context during retrieval
Semantic splitting	Technical or complex documents	Splits based on meaning, not just length
Markdown splitting	Documentation, <code>readme.mds</code> , structured content	Respects headings and sections — ideal for hierarchical knowledge

How to choose?

- **Just getting started?** Use sentence splitting — it works well in most cases.
- **Dealing with long-form content?** Try sentence window or semantic splitting.
- **Working with Markdown docs?** Use `MarkdownNodeParser` to respect section boundaries.
- **Hitting token limits?** Switch to token-based splitting with overlap.

No single method is **perfect**.
Test different splitters using evaluation metrics (e.g., retrieval accuracy, answer relevance), and iterate.

Before we start on the different chunking methods, let's quickly load up the dependencies and create a function to evaluate our chunking methods:

```
import pandas as pd
from llama_index.core.node_parser import (
    SentenceSplitter,
    SemanticSplitterNodeParser,
    SentenceWindowNodeParser,
    MarkdownNodeParser,
    TokenTextSplitter
)
from llama_index.core import VectorStoreIndex
from llama_index.core.prompts import PromptTemplate
```

```

from llama_index.core.postprocessor import
MetadataReplacementPostProcessor
from IPython.display import Markdown, display

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

def eval_splitter(splitter, documents, test_cases, splitter_name):
    print(f"\n{'='*70}")
    print(f"🔍 EVALUATING: {splitter_name}")
    print(f"{'='*70}")

    # 1. Split documents
    print("📄 Parsing documents into chunks...")
    nodes = splitter.get_nodes_from_documents(documents)
    print(f"    → Created {len(nodes)} chunks")

    # 2. Create index
    index = VectorStoreIndex(nodes, embed_model=Settings.embed_model)

    # 🔄 SPECIAL HANDLING: For SentenceWindowNodeParser
    if "SentenceWindow" in splitter_name or isinstance(splitter,
SentenceWindowNodeParser):
        print("🔄 Using MetadataReplacementPostProcessor to inject
sentence window...")

        # Use the official postprocessor to replace context with
sentence_window

        query_engine = index.as_query_engine(
            similarity_top_k=3,
            llm=Settings.llm,
            streaming=False,
            node_postprocessors=[
MetadataReplacementPostProcessor(target_metadata_key="window")
            ],
        )
    else:
        # Default query engine
        query_engine = index.as_query_engine(
            similarity_top_k=3,
            llm=Settings.llm,
            embed_model=Settings.embed_model,
            streaming=False
        )

    # 3. Run test cases

```

```

print("🚀 Running test cases...")
test_results = run_test_cases(query_engine, test_cases)

# Extract results
question = test_results["question"][0]
answer = test_results["answer"][0]
contexts = test_results["contexts"][0]
ground_truth = test_results["ground_truth"][0]

print(f"Answer: {answer}")

# 📌 Show retrieved contexts
print("\n📌 Retrieved Contexts:")
print("-" * 60)
for i, ctx in enumerate(contexts):
    ctx_len = len(ctx)
    print(f"Chunk {i+1} | {ctx_len} chars:\n{ctx}\n")

# 4. Prepare dataset for Ragas
eval_dataset = {
    "question": [question],
    "answer": [answer],
    "ground_truth": [ground_truth],
    "contexts": [contexts],
}
dataset = Dataset.from_dict(eval_dataset)

# 5. Evaluate with Ragas
print("📊 Evaluating with Ragas...")
try:
    results = evaluate(
        dataset=dataset,
        metrics=[
            answer_correctness,
            context_recall,
            context_precision,
            faithfulness,
        ],
        llm=ragas_llm,
        embeddings=ragas_embeddings
    )
    results_df = results.to_pandas()

    # Add metadata
    results_df["splitter"] = splitter_name
    results_df["node_count"] = len(nodes)
    avg_len = int(sum(len(c) for c in contexts) / len(contexts)) if
contexts else 0
    results_df["avg_context_length"] = avg_len
    results_df["context_count"] = len(contexts)

    # 🔍 Create a clean display version of the DataFrame
    display_df = results_df.copy()

```

```

# ✅ Replace 'contexts' with a compact summary for display
context_summary = f"{len(contexts)} chunks, {avg_len} avg chars"
display_df["context_info"] = context_summary

# ✅ Truncate long fields for table readability
display_df["answer"] = display_df["answer"].apply(
    lambda x: x[:100] + "..." if len(x) > 100 else x
)
display_df["ground_truth"] = display_df["ground_truth"].apply(
    lambda x: x[:100] + "..." if len(x) > 100 else x
)

# Reorder for clean display
display_columns = [
    "splitter",
    "node_count",
    "context_info",
    "question",
    "answer",
    "ground_truth",
    "contexts",
    "answer_correctness",
    "context_recall",
    "context_precision",
    "faithfulness"
]
display_df = display_df[display_columns]

# 📊 Show clean table
display(Markdown(display_df.to_markdown(index=False)))

# ✅ Return original results_df (with full contexts for
aggregation)
return results_df

except Exception as e:
    print(f"❌ Evaluation failed: {e}")
    return pd.DataFrame({
        "splitter": [splitter_name],
        "error": [str(e)],
        "node_count": [len(nodes)],
        "question": [question]
    })

```

```

# Set query and ground truth
query="Which tasks are due today?"
ground_truth="You have four tasks due today: \
    1. Finalize Q3 OKRs is due today, \
    2. Onboard new team member is due today, \
    3. Update project roadmap is due today, \
    4. Write thank-you letter is due today."

```

```
# Set up collection for all results from individual splitters
ALL_EVALUATION_RESULTS = []
```

Sentence splitting

How it works:

Splits text at natural sentence boundaries (using punctuation like `.`, `!`, `?`) and heuristics. Groups sentences into chunks until a character or token limit is reached.

Original Text:

Allen loves AI. He reads papers daily. LLMs are amazing!

After Sentence Splitting:

Allen loves AI.

Chunk 1

He reads papers daily.

Chunk 2

LLMs are amazing!

Chunk 3

Pros

- Preserves natural language flow
- Fast and computationally lightweight
- Easy to configure with `chunk_size` (in characters or tokens)
- Works well with most text types

Cons

- Can create overly short chunks (e.g., "Yes." or "See above.")
- May break context if a key idea spans multiple sentences
- Less precise than token-based methods for LLM context limits

Use When

- You're starting with a new RAG pipeline
- Working with well-punctuated, clean text
- Need a simple, reliable baseline chunker

```
result = eval_splitter(
    splitter=SentenceSplitter(chunk_size=128, chunk_overlap=16),
    documents=documents,
    test_cases=test_cases,
    splitter_name="SentenceSplitter"
)

ALL_EVALUATION_RESULTS.append(result)
```

Token splitting

How it works:

Splits text based on **tokens** (units understood by the LLM), not characters. Uses a tokenizer to count tokens and split when **chunk_size** is reached.

Text:

The quick brown fox jumps over the lazy dog near the park.

Tokens:

[The][quick][brown][fox][jumps] [over][the][lazy][dog][near] [the][park][.]

Chunks:

The quick brown fox jumps

Chunk 1

over the lazy dog near

Chunk 2

the park .

Chunk 3

Pros

- Matches LLM context limits exactly (e.g., 512, 1024, 8192 tokens)
- Most accurate way to avoid **context_length_exceeded** errors
- Supports overlap to preserve context across chunks

Cons

- Can split mid-word or mid-sentence, harming readability
- Requires matching the correct tokenizer to your model
- Relatively complex vs. character-based splitting

Use When

- You need precise control over token usage
- Integrating with models that have tight context windows
- Building production-grade RAG systems

```
result = eval_splitter(  
    splitter=TokenTextSplitter(chunk_size=128, chunk_overlap=16),  
    documents=documents,  
    test_cases=test_cases,  
    splitter_name="TokenTextSplitter"  
)  
  
ALL_EVALUATION_RESULTS.append(result)
```

Sentence window

How it works:

Stores small, narrow chunks (e.g., one sentence), but during retrieval, returns the matching sentence **plus N neighboring sentences** before and after (the "window").

Original Text:

[S1] [S2] [S3] [S4] [S5] [S6] [S7]

Stored Nodes (window_size=2):

... S2 S3 [S4] S5 S6 ... ← Node for S4

Query matches S4 → returns full window

✓ Pros

- 📖 Retrieves rich context even from fine-grained chunks
- 🎯 Improves **answer faithfulness** and grounding
- 🔍 Enables high-precision retrieval with full context

✗ Cons

- 🗄️ Increases metadata storage (window context)
- ⌚ Slightly slower due to post-processing
- 🧠 Requires post-retrieval context assembly

🔧 Use When

- You want to maximize answer correctness and grounding
- Using sentence-level indexing
- Prioritizing **retrieval quality** over speed

```
result = eval_splitter(
    splitter=SentenceWindowNodeParser(
        window_size=1, # ← window_size = 1 means that it
                        # will take one sentence before and after the target sentence, i.e. 3
                        # sentences total
        window_metadata_key="window",
        original_text_metadata_key="original_text"),
    documents=documents,
    test_cases=test_cases,
    splitter_name="SentenceWindow"
)

ALL_EVALUATION_RESULTS.append(result)
```

Semantic splitting

How it works:

Uses an embedding model to detect **meaning shifts** in text. Splits only when the semantic similarity between adjacent chunks drops significantly.

Original:

Kitty plays guitar. She loves music. ▶ She works as a lawyer.

↑
Semantic shift: hobby → job

After Split:

Kitty plays guitar. She loves music.

She works as a lawyer.

✅ Pros

- 🧠 Respects meaning and topic boundaries
- 🎯 Keeps related ideas together
- 📈 Often leads to best retrieval performance

❌ Cons

- 🐢 Slow (requires computing embeddings)
- 📄 Needs sufficient text to detect shifts (fails on short docs)
- 📦 Requires a good embedding model

🔧 Use When

- Processing complex, multi-topic documents
- Standard splitters break logical units
- You care more about **quality than speed**

```
result = eval_splitter(  
    splitter=SemanticSplitterNodeParser(  
        buffer_size=1,  
        embed_model=Settings.embed_model  
    ),  
    documents=documents,  
    test_cases=test_cases,  
    splitter_name="SemanticSplitter"  
)  
  
ALL_EVALUATION_RESULTS.append(result)
```

Markdown splitting

How it works:

Parses Markdown documents by structural elements (**# headers**, **- lists**, **code blocks**). Keeps content under each header together, preserving document hierarchy. For example:

```
# Project Overview

## Tasks
- Finalize Q3 OKRs by Friday
- Review team feedback
- Schedule retro

## Notes
Meeting was productive.
Next steps: improve docs.

##
```

✅ Pros

- 📁 Preserves document structure (sections, subsections)
- 📄 Ideal for technical docs, wikis, and READMEs
- 🔗 Maintains context within logical sections

❌ Cons

- 🚫 Only works well on Markdown files
- 📄 May create very large chunks under big headers
- ❌ Useless for plain text or PDFs unless converted

🔧 Use When

- Processing documentation sites (e.g., GitHub, Notion exports)
- Your data has clear headings and sections
- You want **hierarchy-aware chunking**

```
result = eval_splitter(
    splitter=MarkdownNodeParser(),
    documents=documents,
    test_cases=test_cases,
    splitter_name="MarkdownSplitter"
)

ALL_EVALUATION_RESULTS.append(result)
```

Comparing different chunking results

```

print("\n" + "="*50)
print("🏆 FINAL COMPARISON: All Chunking Strategies")
print("="*50)

# Combine all results
if not ALL_EVALUATION_RESULTS:
    print("❌ No results collected.")
else:
    # Concatenate all results
    final_df = pd.concat(ALL_EVALUATION_RESULTS, ignore_index=True)

# Keep only relevant metrics
metric_cols = ["answer_correctness", "context_recall",
               "context_precision", "faithfulness"]

# Show average performance per splitter
summary = final_df.groupby("splitter")[metric_cols].mean().round(3)
display(summary.style.highlight_max(color="lightgreen", axis=0))

```

Embedding models: Choosing your RAG's "brain" for search

We've seen early on how adjusting `similarity_top_k` can help **TaskFriend** see more of your tasks, it's time to ask a deeper question: *How does **TaskFriend** even know which tasks are similar to your query?*

The answer lies in **embeddings** — the invisible mathematical fingerprints that power semantic search in RAG systems. Think of them as the "brain" behind retrieval: they transform words, sentences, or entire chunks into vectors in a high-dimensional space, where meaning is measured by proximity.

Just like humans understand that "urgent" is closer to "critical" than to "optional," a good embedding model learns to place similar concepts close together in vector space. But not all models do this equally well.

In this section, we'll peel back the curtain on how embeddings work, demonstrate their magic with a classic AI analogy (`king - man + woman ≈ queen`), and show you how choosing the right embedding model can dramatically improve **TaskFriend's** ability to find what you're looking for — no matter how you phrase your question.

What are embeddings?

At the heart of every RAG pipeline is a simple idea: **find the most relevant information based on meaning, not keywords.**

But how do you teach a machine to understand meaning?

Enter **embeddings** — numerical representations of text as vectors in a high-dimensional space (often 384 to 1,024 dimensions). These vectors are generated by deep learning models trained on massive amounts of text, so they capture semantic relationships.

For example:

- "happy" and "joyful" will have very similar vectors.
- "car" and "vehicle" are close.
- "apple" (fruit) is closer to "banana" than to "iPhone" — assuming the model understands context!

When you ask **TaskFriend**, *"What tasks are due today?"*, the system doesn't look for exact matches of the word "today." Instead, it converts your query into a vector and searches the knowledge base for chunks whose vectors are **closest** in this semantic space.

👉 So, the quality of your embedding model directly determines how smart your retrieval is.

```
import numpy as np
from dashscope import TextEmbedding
import os

embedding_model='text-embedding-v3'

# Create function to get embedding
def get_embedding(text, embedding_model):
    from dashscope import TextEmbedding
    response = TextEmbedding.call(
        model=embedding_model,
        input=text,
        api_key=os.getenv("DASHSCOPE_API_KEY")
    )
    if response.status_code == 200:
        return np.array(response.output['embeddings'][0]['embedding'])
    else:
        raise Exception(f"Embedding failed: {response.message}")
```

```
# Get embeddings
words = ["king", "man", "woman", "queen"]
embeddings = {word: get_embedding(word, embedding_model) for word in words}

king, man, woman, queen = embeddings["king"], embeddings["man"],
embeddings["woman"], embeddings["queen"]

# Compute the analogy: king - man + woman
result_vector = king - man + woman

from sklearn.metrics.pairwise import cosine_similarity

# Reshape for sklearn
result_vector = result_vector.reshape(1, -1)
queen_vector = queen.reshape(1, -1)
```

```

similarity = cosine_similarity(result_vector, queen_vector)[0][0]
print("Computed cosine similarity!")
print(f"Cosine similarity between '(king - man + woman)' and 'queen':
{similarity:.4f}")

```

```

import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Stack all original vectors
vectors = np.array([king, man, woman, queen])
labels = ['King', 'Man', 'Woman', 'Queen']

# Add the result vector (king - man + woman)
result_vec = king - man + woman
vectors = np.vstack([vectors, result_vec])
labels.append('Result')

# Apply PCA
pca = PCA(n_components=2)
vectors_2d = pca.fit_transform(vectors)

# Create plot
plt.figure(figsize=(10, 8))

# Plot points
colors = ['blue', 'blue', 'red', 'red', 'purple']
plt.scatter(vectors_2d[:, 0], vectors_2d[:, 1], c=colors, s=120,
            edgecolor='k')

# Label points
for i, label in enumerate(labels):
    plt.annotate(label, (vectors_2d[i, 0], vectors_2d[i, 1]),
                 textcoords="offset points", xytext=(8, 5), fontsize=12,
                 ha='left')

# Draw the parallelogram arrows
# Arrow 1: man -> king
plt.arrow(vectors_2d[1, 0], vectors_2d[1, 1],
          vectors_2d[0, 0] - vectors_2d[1, 0],
          vectors_2d[0, 1] - vectors_2d[1, 1],
          head_width=0.02, fc='gray', ec='gray',
          length_includes_head=True, linestyle='--', linewidth=1)

# Arrow 2: woman -> queen
plt.arrow(vectors_2d[2, 0], vectors_2d[2, 1],
          vectors_2d[3, 0] - vectors_2d[2, 0],
          vectors_2d[3, 1] - vectors_2d[2, 1],
          head_width=0.02, fc='gray', ec='gray',
          length_includes_head=True, linestyle='--', linewidth=1)

# Arrow 3: king -> result (should point to queen if perfect)

```

```
plt.arrow(vectors_2d[0, 0], vectors_2d[0, 1],
          vectors_2d[4, 0] - vectors_2d[0, 0],
          vectors_2d[4, 1] - vectors_2d[0, 1],
          head_width=0.02, fc='orange', ec='orange',
          length_includes_head=True, linestyle='--', linewidth=1)

# Optional: Add legend
from matplotlib.patches import Patch
legend_elements = [
    Patch(facecolor='blue', edgecolor='k', label='Male'),
    Patch(facecolor='red', edgecolor='k', label='Female'),
    Patch(facecolor='purple', edgecolor='k', label='Prediction'),
    plt.Line2D([0], [0], color='gray', linewidth=2, label='Observed
vector'),
    plt.Line2D([0], [0], color='orange', linestyle='--', linewidth=2,
label='Computed step')
]
plt.legend(handles=legend_elements, loc='best')

plt.title(f"Vector Analogy: king - man + woman ≈ queen\nEvaluated with
text-embedding-v3", fontsize=14)
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

Understanding the Results

Let’s analyze the outcome of our vector analogy experiment:

- **The obvious:**
 - Cosine similarity is 0.7855 - this is considered a **strong similarity**
 - "King" and "Man" are far apart, indicating the model distinguishes between male royalty and commoners.
 - "Woman" and "Queen" are similarly separated, suggesting the model captures gender roles.
 - The computed result (king - man + woman) lands close to "Queen", demonstrating some understanding of the analogy.

In short, cosine similarity is a measure of directional alignment between two vectors in high-dimensional space — it quantifies how closely two pieces of text are related in meaning, regardless of their magnitude. A high score means the vectors point in nearly the same direction, which in RAG translates to retrieving context that is semantically relevant to the query, even if the words don’t match exactly.

For reference, here's a table of general guidelines for cosine similarity:

Cosine Similarity	Interpretation	What It Means in RAG
-------------------	----------------	----------------------

Cosine Similarity	Interpretation	What It Means in RAG
> 0.85	Very High / Excellent	The texts are nearly synonymous or express the same concept in different words. Ideal for high-stakes retrieval (e.g., medical QA, legal doc search). Models achieving this consistently are considered top-tier.
0.75 – 0.85	High / Strong	Clear semantic relationship. Suitable for most RAG applications. This is a common threshold for relevance in vector search (e.g., filter chunks above 0.8). Captures paraphrasing and topic-level matching well.
0.60 – 0.75	Moderate / Acceptable	Some meaningful overlap in meaning, but not precise. May retrieve tangentially related content. Can work in broad-domain retrieval, but increases noise. Often requires re-ranking or hybrid search to improve precision.
0.50 – 0.60	Low / Weak	Marginal similarity. The texts share some keywords or general topic, but likely differ in intent or detail. Risk of false positives. Not recommended as a standalone retrieval threshold.
< 0.50	Negligible / Unrelated	Little to no semantic connection. The vectors represent distinct concepts. In RAG, matches below this level should typically be discarded.

This result of `0.7855` is promising — it shows that the `text-embedding-v3` model has a solid grasp of semantic structure. However, it’s not perfect. In real-world RAG systems, even small gaps in embedding quality can compound during retrieval, leading to missed context or hallucinated answers. That’s why choosing the right embedding model isn’t just a technical detail — it’s choosing the "brain" of your retrieval system. A smarter brain means more accurate, reliable, and faithful responses.

Before we move on to the next section - we'll do a quick comparison between the `text-embedding-v3` model and the `text-embedding-v2` model we've been using up till now:

```
# Define sentences to compare (2 sentences)
sentences = [
    "Which tasks should be completed today?",
    "Which tasks are due today and need to be finished?"
]

# Models to compare
model = ["text-embedding-v2", "text-embedding-v3"]

# Store results: model -> similarity score
results = {}

from sklearn.metrics.pairwise import cosine_similarity

print("Comparing sentences:")
print(f"{sentences}")
print("=" * 50 + "\n")
```

```
# Evaluate each model
for model in model:
    try:
        print(f"🧠 Testing {model} on sentence similarity...")

        # Get embeddings for both sentences
        emb1 = get_embedding(sentences[0], embedding_model=model)
        emb2 = get_embedding(sentences[1], embedding_model=model)

        # Reshape for sklearn
        emb1 = emb1.reshape(1, -1)
        emb2 = emb2.reshape(1, -1)

        # Compute cosine similarity
        sim = cosine_similarity(emb1, emb2)[0][0]
        print(f"✅ Similarity between sentences: {sim:.4f}\n")
        results[model] = sim

    except Exception as e:
        print(f"❌ Error with {model}: {e}\n")
        results[model] = None

# 📊 Final Comparison
print("\n📊 SENTENCE SIMILARITY COMPARISON")
print("-" * 50)
for model, sim in results.items():
    status = f"{sim:.4f}" if sim is not None else "Failed"
    print(f"{model:>20} : {status}")
```

Both embeddings show different levels of cosine similarity, but the results tell a clear story: **text-embedding-v3 captures semantic equivalence far more effectively.**

With a similarity score of **0.9623**, **text-embedding-v3** treats the two sentences — *"Which tasks should be completed today?"* and *"Which tasks are due today and need to be finished?"* — as nearly identical in meaning. That's ideal for RAG: it means the system will retrieve the right context no matter how the user phrases their question.

Even **text-embedding-v2** performs well at **0.9114**, which falls in the "High / Strong" range — but that small gap (0.05) can make a big difference when scaled across hundreds of queries.

In practical terms:

- A higher similarity score means **better context recall** — fewer missed tasks.
- It also means **cleaner retrieval**, reducing noise and improving faithfulness.
- Ultimately, it leads to **more accurate, confident answers** — with less hallucination.

This isn't just a marginal improvement — it's the difference between a helpful assistant and a frustrating one.

How embeddings affect RAG outcomes

Think of embeddings as the **semantic GPS** of your retrieval system. They don't just map words — they map *meaning*. When you ask TaskFriend, *"What's due today?"*, it doesn't search for exact matches of *"today."* Instead, it converts your query into a vector and finds chunks whose vectors are closest in meaning — whether they say *"due today,"* *"must be finished by EOD,"* or *"urgent: complete now."*

A better embedding model means:

- 🎯 **Higher context recall:** More relevant chunks are retrieved, even if phrased differently.
- 🔍 **Better context precision:** Irrelevant or outdated tasks are filtered out more effectively.
- 🧠 **Improved faithfulness:** The LLM receives cleaner, more focused context — reducing the temptation to hallucinate.
- 💬 **Robustness to paraphrasing:** Users can say things like *"What should I finish today?"* or *"Any urgent tasks?"* and still get the same correct answer.

In short, upgrading your embedding model is like upgrading the brain of your RAG system. It doesn't just make retrieval faster — it makes it *smarter*.

So how does it work in the real world? Let's test it out:

```
def build_query_engine_with_embedding(embed_model_name):
    """Build a query engine using a specific DashScope embedding model."""
    from llama_index.embeddings.dashscope import DashScopeEmbedding

    # Create a temporary embed model
    temp_embed_model = DashScopeEmbedding(
        model_name=embed_model_name,
        api_key=os.getenv("DASHSCOPE_API_KEY"),
        encoding_format="float"
    )

    # Rebuild index with new embedding model
    storage_context =
StorageContext.from_defaults(persist_dir=persist_path)
    index = load_index_from_storage(
        storage_context,
        embed_model=temp_embed_model
    )
```

```
# Define test cases
test_cases = [
    {
        "question": "What tasks are due today?",
        "ground_truth": (
            "The tasks due today are:"
            "1. Finalize Q3 OKRs is due today, "
            "2. Onboard new team member is due today, "
            "3. Update project roadmap is due today, "
```

```

        "4. Write thank-you letter is due today."
    )
}

]

```

```

# Load the nodes from storage (once) – this is your raw document chunks
storage_context = StorageContext.from_defaults(persist_dir=persist_path)
# You can reload all the nodes
node_ids = list(storage_context.docstore.docs.keys())
nodes = storage_context.docstore.get_nodes(node_ids)

# List of models to compare
embedding_models = ["text-embedding-v2", "text-embedding-v3"]

# Store evaluation results
evaluation_results = []

# Run evaluation for each model
for model_name in embedding_models:
    print(f"🔧 Building index and evaluating with: {model_name}")

    try:
        # ✅ Create embedding model
        embed_model = DashScopeEmbedding(
            model_name=model_name.strip(), # Handle any whitespace
            api_key=os.getenv("DASHSCOPE_API_KEY"),
            encoding_format="float"
        )

        # ✅ Rebuild index using this embed_model
        index = VectorStoreIndex(
            nodes,
            embed_model=embed_model
        )

        # ✅ Build query engine
        query_engine = index.as_query_engine(
            streaming=True,
            llm=Settings.llm,
            similarity_top_k=3
        )

        # ✅ Run test cases
        test_results = run_test_cases(query_engine, test_cases)

        print(f"Beginning evaluation of test cases...")
        start_time = time.time()

        # ✅ Convert to Dataset
        eval_dataset = Dataset.from_dict({

```

```

        "question": test_results["question"],
        "answer": test_results["answer"],
        "contexts": test_results["contexts"],
        "ground_truth": test_results["ground_truth"]
    })

# 🛠️ Print answers after running test cases
print("\n🛠️ TEST CASE STATS")
print("=" * 50)
for i in range(len(test_results["answer"])):
    print(f"\nQuestion: {test_results['question'][i]}")
    print(f"Ground Truth: {test_results['ground_truth'][i]}")
    print(f"Answer: {test_results['answer'][i]}")
    print("-" * 50)

# ✅ Evaluate using Ragas
results = evaluate(
    dataset=eval_dataset,
    metrics=[answer_correctness, context_recall,
context_precision, faithfulness],
    llm=ragas_llm,
    embeddings=DashScopeEmbeddings(model=model_name.strip(),
api_key=os.getenv("DASHSCOPE_API_KEY"))
)

# ✅ Convert to dictionary and add model name
result_dict = results.to_pandas().to_dict('records')[0]
result_dict['embedding_model'] = model_name.strip()
evaluation_results.append(result_dict)

load_time = time.time() - start_time
print(f"✅ Evaluation complete for {model_name} ({load_time:.1f}
seconds)\n")

except Exception as e:
    print(f"❌ Error with {model_name}: {e}\n")
    evaluation_results.append({
        'embedding_model': model_name.strip(),
        'answer_correctness': None,
        'context_recall': None,
        'context_precision': None,
        'faithfulness': None
    })

# Convert to DataFrame
import pandas as pd
results_df = pd.DataFrame(evaluation_results)

# Reorder columns
results_df = results_df[[
    'embedding_model',
    'question',
    'answer',
    'ground_truth',

```

```
'answer_correctness',  
'context_recall',  
'context_precision',  
'faithfulness'  
]]  
  
print("\n📊 FINAL EVALUATION RESULTS (Comparison)")  
print("=" * 50)  
display(results_df)
```

Depending on the results - there might be variation in the `answer_correctness` metric, but more often than not `text-embedding-v3` outperforms the older `text-embedding-v2`. In short, better embeddings help your model *better understand* the relationships between words, phrases, and concepts and produce useful output.

The Recall Game: Finding the Right Answer in a Sea of Chunks

At first glance, retrieval might seem like the quiet middle child of the RAG pipeline — just a simple “search” step between ingestion and generation. But in reality, retrieval is the make-or-break moment. No matter how clean your data or powerful your LLM, if the system fails to pull the right context, the answer will be wrong.

In an ideal scenario, users will always know what they're looking for - but as AI becomes more and more ingrained into our lives, it becomes more of our co-pilot or second brain. Soon, queries shift from

```
"What urgent tasks are due today?"  
to  
"I'm stuck — what should I do next?"
```

These are not just search queries, but actual people looking for actual advice.

In real-world scenarios, the knowledge needed to answer such questions is rarely contained in a single, perfectly matched chunk. It's scattered across guidance documents, best practices, and process frameworks. The model sees only what you give it — and if that context is incomplete or noisy, hallucination follows.

So how do we close the gap?

It starts with reframing retrieval not as a lookup, but as a strategic reasoning process — one that can be optimized before and after the initial search. In this section, we'll explore three high-impact techniques that turn retrieval into a robust, intelligent operation:

1. **Query rewriting:** Help the system understand what the user *really* means.
2. **Metadata tagging:** Add structure to chaos with metadata-driven filtering.
3. **Reranking chunks:** Promote the best context to the top — because relevance isn't always obvious at first glance.

Let's explore how we can improve recall performance for our RAG-enhanced **TaskFriend** application.

Query rewriting: Speak the knowledge base's language

Rewriting queries reframe the queries written by users into more precise queries that the RAG system can use. There are multiple ways we can use to refine our queries, and in this section we'll explore 3 of the most widely used rewriting strategies:

- **LLM rewrite:** Uses an LLM to rephrase a user's query into a more precise, retrieval-friendly format.
- **Multi-step query:** Breaks a complex question into smaller sub-questions, retrieves answers step by step, then synthesizes a final response.
- **HyDE rewrite:** Generates a hypothetical answer first, then uses that as the search query. Works well for emotional or vague queries where keywords are missing.

Before we start, let's first create a function to compare our query strategies:

```
def eval_query_strategy(
    strategy_fn,
    strategy_name,
    index,
    test_cases,
    ragas_llm,
    ragas_embeddings,
    node_count
):

    print(f"\n{'='*70}")
    print(f"🔍 EVALUATING: {strategy_name}")
    print(f"{'='*70}")

    # Detect if strategy returns a query engine (e.g., Multi-Step, HyDE)
    test_query = test_cases[0]["question"]
    result = strategy_fn(test_query)

    # ✅ Log original query for ALL strategies
    print(f"Original query → '{test_query}'")

    if hasattr(result, "query"): # It's a query engine
        print(f" → Using transformed query engine: {strategy_name}")
        query_engine = result
        final_query = test_query
    else: # It's a rewritten query string
        final_query = result
        print(f"Rewritten query → '{final_query}'")
        query_engine = index.as_query_engine(
            similarity_top_k=3,
            llm=Settings.llm,
            streaming=False
        )

    # Run test case
```

```

try:
    response = query_engine.query(final_query)
    answer = str(response).strip()
    contexts = [node.get_content().strip() for node in
response.source_nodes]
    source_nodes = response.source_nodes
    print(f"\n🗨️ ANSWER:\n{answer}")
except Exception as e:
    print(f"❌ Error during query: {e}")
    answer = f"Error: {e}"
    contexts = []
    source_nodes = []

# ✅ NEW: Print retrieved contexts
print(f"\n📄 RETRIEVED CONTEXTS ({len(contexts)} chunks):")
print("-" * 60)
if contexts:
    for i, ctx in enumerate(contexts):
        print(f"📄 Chunk {i+1} (Length: {len(ctx)} chars)")
        print(f"    {ctx.strip()}")
        print(f"{'-'*60}")
else:
    print("    No contexts retrieved.")
    print("-" * 60)

# Prepare for Ragas
dataset = Dataset.from_dict({
    "question": [test_query],
    "answer": [answer],
    "contexts": [contexts],
    "ground_truth": [test_cases[0]["ground_truth"]]
})

# Evaluate with Ragas
print(f"\n📊 EVALUATING WITH RAGAS...")
try:
    results = evaluate(
        dataset=dataset,
        metrics=[
            answer_correctness,
            context_recall,
            context_precision,
            faithfulness,
        ],
        llm=ragas_llm,
        embeddings=ragas_embeddings
    )
    results_df = results.to_pandas()
    results_df["strategy"] = strategy_name
    results_df["node_count"] = node_count
    avg_len = int(sum(len(c) for c in contexts) / len(contexts)) if
contexts else 0
    results_df["avg_context_length"] = avg_len
    results_df["context_info"] = f"{len(contexts)} chunks, {avg_len}

```

```

avg_chars"

    # Clean display
    display_df = results_df[[
        "strategy", "node_count", "context_info", "question",
        "answer",
        "ground_truth", "answer_correctness", "context_recall",
        "context_precision", "faithfulness"
    ]].copy()
    display_df["answer"] = display_df["answer"].apply(lambda x:
x[:100] + "..." if len(x) > 100 else x)
    display_df["ground_truth"] =
display_df["ground_truth"].apply(lambda x: x[:100] + "..." if len(x) > 100
else x)

    print(f"\n✅ RESULTS: {strategy_name}")
    display(Markdown(display_df.to_markdown(index=False)))

    return {
        "results_df": results_df,
        "contexts": contexts,
        "source_nodes": source_nodes,
        "answer": answer,
        "final_query": final_query
    }

except Exception as e:
    print(f"❌ Evaluation failed: {e}")
    return {
        "results_df": pd.DataFrame({"strategy": [strategy_name],
"error": [str(e)]}),
        "contexts": [],
        "source_nodes": [],
        "answer": "",
        "final_query": final_query
    }

```

Rephrase your question: LLM rewrite

This method is one of the easiest to implement - plug your original query into the LLM, and use that to rewrite your query:

```

# Strategy 0: No rewriting
def no_rewrite(query):
    return query

# Strategy 1: LLM-based rewriting
rewrite_prompt = PromptTemplate("""
You are a query optimization assistant for a structured task management
system.

```

Your goal is to **rewrite ambiguous user questions** into **precise, retrieval-friendly queries** that match the database schema.

Database Schema

- Fields: ID, Description, Type, Priority, Due, Status, Stakeholders, Notes
- Time mappings: "today" → "Due == 'Today'", "urgent" → "Priority == 'High'"
- Status: "not started", "in progress", "completed"

Rules

1. Do NOT generate answers – only rewrite the query.
2. Map natural language to field-based filters.
3. Use logical operators: AND, OR, NOT.
4. Output only the rewritten query – no explanations.

Examples

Original: "What tasks are due <when>?"

Rewritten: "Find all tasks where Due == <when>"

Original: "I'm stuck on <task> – what should I do?"

Rewritten: "Find guidance on what to do when stuck on a <task> (note priority, etc.)>, or how to start <task> when stuck"

Original: "How do I prioritize my day?"

Rewritten: "Find the prioritization framework for daily planning"

Original question: {query_str}

Rewritten query:

""")

```
def llm_rewrite(query):
    response =
Settings.llm.complete(rewrite_prompt.format(query_str=query))
    return str(response.text).strip()
```

As we've covered early on, the knowledge needed to answer more complex questions may come from other documents in the knowledgebase. For **TaskFriend**, you want to create a guideline on how to advise users when they're stuck that ties in with your brand - helping them make work more manageable and promoting efficiency, while ensuring users get to celebrate their little wins.

For this, we'll be using the entire `./docs/taskfriend` folder, which also includes the documents you've prepared for the **TaskFriend Coach**, a helpful assistant that provides users with answers that help steer them in the right direction.

```
from llama_index.core import SimpleDirectoryReader

documents = SimpleDirectoryReader(
    "./docs/taskfriend",
```

```

        recursive=True
    ).load_data()

    print(f"✅ Loaded {len(documents)} documents")

    splitter = SentenceSplitter(chunk_size=128, chunk_overlap=16)
    nodes = splitter.get_nodes_from_documents(documents)
    index = VectorStoreIndex(nodes, embed_model=Settings.embed_model)

    test_cases = [
        {
            "question": "I'm overwhelmed and stuck on finalizing the Q3 OKRs – what should I do next?",
            "ground_truth": "Identify most important steps. Draft objectives, Align with leads on objectives, Get approval. Break it down into smaller steps."
        }
    ]

```

```

# Initialize list to collect results
QUERY_REWRITING_RESULTS = []

# Evaluate each strategy
strategies = [
    ("No Rewrite", no_rewrite),
    ("LLM Rewrite", llm_rewrite),
]

for name, fn in strategies:
    result = eval_query_strategy(
        strategy_fn=fn,
        strategy_name=name,
        index=index,
        test_cases=test_cases,
        ragas_llm=ragas_llm,
        ragas_embeddings=ragas_embeddings,
        node_count=len(nodes)
    )
    QUERY_REWRITING_RESULTS.append(result)

```

Let's dissect our results. The **LLM Rewrite** strategy we used here transformed the basic query to highlight the user's needs, focusing on the users' (*hidden*) key question within the query: "**I need guidance and support**", and the model responded by providing clear, actionable answers.

However, take note that the **answer_correctness metric** is highly dependent on your ground truth – meaning that any tuning we're doing is conditioning the model towards the behavior you want to instill via the **ground_truth**. In actual practice, you'll need to be able to work with your customers and target audience in defining what your app needs to do.

Break the problem down: Multi-step query decomposition

When users ask simple questions like **"What tasks are due today?"**, a single vector search might suffice. But real-world queries are rarely so clean.

Consider this:

"I'm overwhelmed and stuck on finalizing the Q3 OKRs — what should I do next?"

This isn't a single question — it's a complex cognitive state wrapped in a sentence. To answer it well, the system must:

- Disentangle the emotional signal ("overwhelmed") from the task at hand
- Identify the core blockers
- Recall relevant frameworks (e.g., prioritization, focus techniques)
- Synthesize a response that's both practical and empathetic

This is where **Multi-Step Query Decomposition** shines.

Rather than treating retrieval as a one-shot search, this technique reframes it as a structured reasoning process. It takes an ambiguous, high-level question and breaks it down into a sequence of smaller, targeted sub-queries — each designed to retrieve a specific piece of knowledge.

For example, given the question above, the system might generate:

"What might cause the user to be overwhelmed?"

"Are there subtasks that the user can prioritize when writing their OKRs?"

"How can the user focus their attention and keep on-track?"

Each sub-query is executed independently, retrieving focused context from the knowledge base. The final answer is then synthesized from this aggregated, multi-source context — not from a single **top-k** match.

This is a great example of decomposition, as it takes real questions and answers them from the perspective of the user. This works for the LLM because:

- **Handles complexity:** Real problems are rarely atomic. Decomposition mirrors how humans solve them — step by step.
- **Improves recall:** By exploring multiple angles, it reduces the risk of missing critical information.

Multi-step query decomposition excels when:

- The user's intent is unclear or layered
- The answer requires synthesizing multiple pieces of knowledge
- Simple keyword or vector search fails to retrieve the right guidance

It's not always the fastest approach — but it's one of the most robust for building AI assistants that don't just retrieve, but understand.

```
# Strategy 2: Multi-Step Query Decomposition
def multistep_rewrite(query):
    from llama_index.core.indices.query.query_transform.base import
    StepDecomposeQueryTransform
    from llama_index.core.query_engine import MultiStepQueryEngine
```

```

# Build base engine
base_engine = index.as_query_engine(
    similarity_top_k=3,
    llm=Settings.llm,
    streaming=False
)

# Teach your engine how to decompose the query
decompose_prompt = PromptTemplate(
    "You are a coaching assistant to help users untangle their queries. "
    "Break the query into 2-3 specific sub-questions that will help find the right guidance. "
    "Each sub-question should target a specific part of the query."
    "Do NOT include any explanations, formatting, or introductions. "
    "Output only the questions, one per line.\n"
    "Now, decompose this query:\n"
    "User: {query_str}\n"
    "Output:"
)

# Create transform with EXPLICIT, helpful index_summary
step_transform = StepDecomposeQueryTransform(
    llm=Settings.llm,
    verbose=True,
    step_decompose_query_prompt=decompose_prompt
)

return MultiStepQueryEngine(
    query_engine=base_engine,
    query_transform=step_transform,
    index_summary=(
        "A collection of user guidance documents for TaskFriend, an AI task management app. "
        "Includes how to handle being stuck, prioritize tasks, manage overwhelm, and plan daily work. "
        "Content is structured as advice, frameworks, and best practices for productivity."
    ),
)

```

```

# Initialize list to collect results
QUERY_REWRITING_RESULTS = []

# Evaluate each strategy
strategies = [
    ("Multi-Step Decomposition", multistep_rewrite),
]

```

```

for name, fn in strategies:
    result = eval_query_strategy(
        strategy_fn=fn,
        strategy_name=name,
        index=index,
        test_cases=test_cases,
        ragas_llm=ragas_llm,
        ragas_embeddings=ragas_embeddings,
        node_count=len(nodes)
    )
    QUERY_REWRITING_RESULTS.append(result)

```

Let's dissect our results from **Multi-step Query Decomposition**. We notice that the LLM now breaks down our query based on the instructions we provided in **decompose_prompt**, and generated sub-questions that further break down the requirements. This gives the LLM a better understanding of your needs, and helps it to recall chunks that are more relevant based on the multiple questions.

Then, it takes these chunks and generates a more complete, precise answer based on the aggregate of these chunks.

Imagine to retrieve: Hypothetical Document Embeddings (HyDE)

What if we could retrieve answers we haven't written yet?

That's the idea behind **HyDE (Hypothetical Document Embeddings)** — a technique that turns the RAG pipeline on its head. Instead of starting with a query, HyDE starts with a hypothetical answer.

Here's how it works:

1. The user asks: **"I'm overwhelmed and stuck on finalizing the Q3 OKRs — what should I do next?"**
2. The LLM generates a detailed, coaching-style response — not as the final answer, but as a search query.
3. This hypothetical answer is embedded and used to search the vector index.
4. The real documents that best match this synthetic response are retrieved.
5. The final answer is generated from those real documents.

Why is this powerful?

Because good answers are better search queries than good questions. A well-structured, empathetic answer contains rich semantic cues — like "5-minute rule", "break it down", "align with stakeholders" — that are more likely to match coaching content than the vague original query.

HyDE is especially effective once your data grows to a certain volume, especially when more and more documents are added in various formats and structures.

```

from llama_index.core.indices.query.query_transform.base import
HyDEQueryTransform
from llama_index.core.query_engine import TransformQueryEngine

# Strategy 3: HyDE (Hypothetical Document Embeddings)

```

```
def hyde_rewrite(query):
    # Build base engine
    base_engine = index.as_query_engine(
        similarity_top_k=3,
        llm=Settings.llm,
        streaming=False
    )

    hyde_prompt = PromptTemplate(
        "You are a personal productivity coach. "
        "Generate a detailed, helpful answer to the user's question.\n\n"
        "Question: {context_str}\n\n"
        "Hypothetical Answer:"
    )

    hyde = HyDEQueryTransform(
        llm=Settings.llm,
        include_original=True,
        hyde_prompt=hyde_prompt
    )

    return TransformQueryEngine(
        query_engine=base_engine,
        query_transform=hyde
    )
```

```
# Test HyDE directly
print("🔪 TESTING HYDE: Generating hypothetical document...")
hyde = HyDEQueryTransform(
    llm=Settings.llm,
    hyde_prompt=PromptTemplate(
        "You are a personal productivity coach. "
        "Generate a detailed, helpful answer to the user's question.\n\n"
        "Question: {context_str}\n\n"
        "Hypothetical Answer:"
    ),
    include_original=True
)

query = test_cases[0]["question"]
bundle = hyde(query)

print("\n🔪 HYPOTHETICAL DOCUMENT GENERATED:")
print("-" * 60)
print(bundle.embedding_strs[0])
print("-" * 60)
```

```
# Initialize list to collect results
QUERY_REWRITING_RESULTS = []
```

```
# Define all strategies
strategies = [
    ("HyDE", hyde_rewrite),
]

# Evaluate each
for name, fn in strategies:
    result = eval_query_strategy(
        strategy_fn=fn,
        strategy_name=name,
        index=index,
        test_cases=test_cases,
        ragas_llm=ragas_llm,
        ragas_embeddings=ragas_embeddings,
        node_count=len(nodes)
    )
    QUERY_REWRITING_RESULTS.append(result)
```

Let's dissect our results, and compare them to the previous strategies. [[COMPLETE THIS FOR ME]]

Conclusion

We began with a user's simple question: *"I'm stuck – what should I do next?"*

From that, we've built a sophisticated RAG system that doesn't just retrieve tasks — it delivers **coaching**, **clarity**, and **confidence**.

Our evaluation of three query rewriting strategies reveals a powerful insight:

- **LLM Rewrite** improves retrieval by translating intent into structured queries.
- **Multi-Step Decomposition** enables deep reasoning but can over-fetch.
- **HyDE** imagines the ideal answer, then finds it — a uniquely powerful approach for guidance-based systems.

No single strategy wins in all dimensions. The best choice depends on what you need your LLM to do.

Metadata tagging: Add structure to chaos with metadata filtering

We've seen how query rewriting helps the system understand user intent. But what if we could go further — not just improve the query, but improve the index itself?

In real-world applications, knowledge bases grow messy. You have:

- Task descriptions
- Coaching guides
- Meeting notes
- User feedback
- Process frameworks

All mixed together.

When a user asks, “I’m stuck – what should I do next?”, the system should ideally:

- Recognize this as a coaching request
- Filter to guidance content
- Retrieve only the most relevant frameworks

But how?

Enter **metadata tagging** — a powerful technique that adds structure to chaos by enriching your documents with semantic labels.

Why tagging matters

Imagine searching a library without categories. You ask for a book on mindfulness, and the librarian returns:

- A novel with a calm character
- A neuroscience textbook
- A yoga manual

All are technically “related” — but you're expecting something more aligned with your requirements.

Without metadata, your RAG system is that librarian.

By tagging documents with labels like:

- "topic": "being-stuck"
- "framework": "5-minute-rule"
- "type": "coaching-guide"

...you're giving your RAG system more information to match your requirements, and narrow the search space before retrieval.

This is especially valuable when:

- Your knowledge base is large and diverse
- Users ask ambiguous questions
- You want to route queries to specialized content

Extracting tags automatically with LLMs

Manually tagging hundreds of documents isn’t scalable. But LLMs can do it for you.

Using a technique called LLM-based metadata extraction, you can automatically analyze each document and assign structured tags.

```
# Create LLM engine to extract tags
system_message = """
You are a metadata tagging expert for TaskFriend, an AI productivity
coach.
Analyze the text and extract structured tags that help route user queries
to the right guidance.

### Supported Tag Types
```

```

- topic
- content_type
- audience
- task_stage

### Output Rules
1. Output only a JSON object: {"tags": [{"key": "tag_type", "value": "tag_value"}]}
2. Each tag can have multiple values, store as JSON array in the object
3. Do not include tags and values that are not found
4. Keep values concise and normalized
5. Output only JSON – no explanations

---
Text to analyze:
"""

def extract_metadata_tags(text):
    """Extract structured metadata tags from a document."""
    try:
        completion = client.chat.completions.create(
            model="qwen-plus", # Fast and cost-effective
            messages=[
                {"role": "system", "content": system_message},
                {"role": "user", "content": text}
            ],
            response_format={"type": "json_object"}, # Ensure clean JSON
            temperature=0.2 # Low temperature for consistency
        )
        return completion.choices[0].message.content
    except Exception as e:
        return f'{{"error": "{str(e)}"}}'

```

And let's extract some tags based on the files we have in `./doc/taskfriend/`. For clarity, we're just going to copy the text from two of those files here.

```

# Example 1: "What to Do When You're Stuck"
when_stuck_text = """
# What to Do When You're Stuck

It's normal to feel stuck. The key is not to stay stuck.

## Step 1: Name the Blocker
Ask yourself:
- Am I waiting for input from someone?
- Is the task unclear?
- Am I avoiding it because it's hard?

```

Write it down. Just naming it helps.

Step 2: Start and Apply 5-minute Rule

Commit to working on the task for just 5 minutes. Often, starting is the hardest part.

Step 3: Break It Down

Can you split the task into smaller steps?

Example:

"Write report" → "1. Outline sections, 2. Draft intro, 3. Add data"
 ""

```
print("\n📌 'When You're Stuck' – Extracted Tags:")
print(extract_metadata_tags(when_stuck_text))
```

```
# Example 2: "Managing Overwhelm"
```

```
overwhelm_text = ""
```

```
# Feeling Overwhelmed? Here's How to Manage It
```

When your task list feels too long:

1. Pause and Breathe

Take 2 minutes. Close your eyes. Breathe.

2. Filter by “Due Today + High Priority”

Only look at what *actually* matters today.

3. Hide the Rest

Use TaskFriend’s “Focus Mode” to hide non-urgent tasks.

4. Pick One Thing

Ask: “If I only finish one thing today, what should it be?”
 ""

```
print("\n📌 'Managing Overwhelm' – Extracted Tags:")
print(extract_metadata_tags(overwhelm_text))
```

Though simple, this process helps create **helpful tags** that your app can use. For example, if the user is having a rough time breaking down tasks, we can rewrite queries to look for chunks tagged with **"topic": "task-breakdown"** to help the app retrieve more relevant chunks and provide better answers.

This technique is a helpful addition to any RAG system, especially systems that use highly structured documents or databases.

Reranking chunks: Putting what's most relevant at the top

[[NOTE THIS ENTIRE PART, THERE ARE MULTIPLE ISSUES WE NEED TO FIX. 1. EXAMPLES, 2. THE CODE AND ITS OUTPUT ISNT RERANKING AT ALL]]

You’ve optimized your chunks. You’ve chosen the right embeddings. You’ve rewritten queries, added metadata, and even broken problems down into sub-questions.

But here's a hard truth: *not all retrieved chunks are created equal*.

Vector similarity search does a good job of finding relevant content — but it doesn't always rank the most relevant content at the top.

Why reranking matters

Reranking is the process of reordering retrieved chunks using a more sophisticated relevance model — often a cross-encoder — that deeply analyzes the relationship between the query and each chunk.

Unlike vector search (which is fast but approximate), reranking:

- Reads the full query and chunk together
- Understands nuanced relevance
- Promotes the best-matching context to the top

It's the difference between:

- "These documents are vaguely related" → vector search
- "This document directly answers your question" → reranking

For **TaskFriend**, this is critical. When a user says:

"I'm stuck — what should I do next?"

They don't just need any advice. They need the right advice — like the 5-minute rule — at the top of the context window.

Reranking ensures that.

```
test_cases = [
    {
        "question": "Which tasks are due today?",
        "ground_truth": ("""
The tasks due today are:
1. Finalize Q3 OKRs is due today,
2. Onboard new team member is due today,
3. Update project roadmap is due today,
4. Write thank-you letter is due today
""")
    }
]
```

```
from llama_index.postprocessor.dashscope_rerank import DashScopeRerank
from llama_index.core.postprocessor import SimilarityPostprocessor,
LLMRerank, SentenceTransformerRerank

# Reset index to read all documents in the `./docs/taskfriend` directory
documents = SimpleDirectoryReader(
    "./docs/taskfriend",
```

```

        recursive=True
    ).load_data()
    print(f"✅ Loaded {len(documents)} documents")

    splitter = TokenTextSplitter(chunk_size=128, chunk_overlap=16)

    print("🔪 Parsing documents into chunks...")
    nodes = splitter.get_nodes_from_documents(documents)
    print(f"    → Created {len(nodes)} chunks")

    index = VectorStoreIndex(nodes, embed_model=Settings.embed_model)

    rerank_query_engine = index.as_query_engine(
        similarity_top_k=45,
        streaming=True,
        llm=Settings.llm,
        node_postprocessors=[
            LLMRerank(
                choice_batch_size=15,
                top_n=5,
            )
        ]
    )

    # Custom code to show our results and chunks being referenced.
    def print_test_results(results, strategy_name=""):
        if strategy_name:
            print(f"\n{'='*50}")
            print(f"🔍 RESULTS: {strategy_name}")
            print(f"{'='*50}")

        for i, question in enumerate(results["question"]):
            print(f"\n? Question: {question}")
            print(f"💬 Answer: {results['answer'][i]}")

            print(f"\n📌 RETRIEVED CONTEXTS ({len(results['contexts'][i])}
chunks):")
            print("-" * 50)
            for j, ctx in enumerate(results['contexts'][i]):
                print(f"📄 Chunk {j+1} | {len(ctx)} chars:")
                print(f"    {ctx}")
                print(f"{'-'*50}")

    def debug_response(response):
        print("\n🔍 DEBUG: Response object:")
        print(f"Type: {type(response)}")
        print(f"Has response: {'response' in dir(response)}")
        print(f"Has source_nodes: {'source_nodes' in dir(response)}")
        if hasattr(response, "source_nodes"):
            print(f"Number of source nodes: {len(response.source_nodes)}")

    # Run baseline (no reranking)
    print("\n🚀 BASELINE: Vector Search Only")

```

```

baseline_engine = index.as_query_engine(similarity_top_k=5,
llm=Settings.llm)
baseline_results = run_test_cases(baseline_engine, test_cases)

# Run with reranking
print("\n🚀 ENHANCED: Vector Search + Reranking")
rerank_results = run_test_cases(rerank_query_engine, test_cases)

# Print results
print_test_results(baseline_results, "Baseline")
print_test_results(rerank_results, "Reranked")

```

The results are pretty stunning - we can see that the default ranker placed many irrelevant chunks higher up, resulting in an answer that was pretty far away from what we were expecting. But, with the **LLMRerank** postprocessor, we got 3 key pieces of information placed on top, resulting in an answer that is more accurate and in-line with the question.

The Optimized RAG Pipeline

In this chapter, you transformed **TaskFriend** from a basic RAG system into a smart, context-aware assistant by optimizing every stage of the pipeline:

```

graph TD
    A[User Query] --> B{Query Rewriting}
    B -->|Rewritten Query| C[Metadata Filtering]
    C -->|Filtered Chunks| D[Vector Similarity Search]
    D -->|Top-K Chunks| E[Reranking]
    E -->|Reordered Context| F[LLM Generation]
    F --> G[Final Answer]

    subgraph "TaskFriend Knowledge Base"
        direction TB
        H[Raw Documents] --> I[Chunking]
        I --> J[Metadata Tagging]
        J --> K[Embedding Model]
        K --> L[(Vector DB)]
    end

    J -->|topic: being-stuck  
framework: 5-minute-rule| C
    L --> D
    E -. ->|Uses cross-encoder  
for relevance scoring| L

    style A fill:#4CAF50,stroke:#388E3C,color:white
    style G fill:#4CAF50,stroke:#388E3C,color:white
    style B fill:#2196F3,stroke:#1976D2,color:white
    style C fill:#FF9800,stroke:#F57C00,color:white
    style E fill:#9C27B0,stroke:#7B1FA2,color:white

```

```
style L fill:#607D8B,stroke:#455A64,color:white

classDef process fill:#2196F3,stroke:#1976D2,stroke-
width:2px,color:white;
classDef enhancement fill:#FF9800,stroke:#F57C00,stroke-
width:2px,color:white;
classDef final fill:#4CAF50,stroke:#388E3C,stroke-
width:2px,color:white;
classDef db fill:#607D8B,stroke:#455A64,stroke-width:2px,color:white;

class B,C,E process
class J enhancement
class A,G final
class L db
```

What's next?

Quiz yourself!

► 1. Which of the following best describes the purpose of the `similarity_top_k` parameter in a RAG system?

- A) It limits how many times the LLM can generate an answer
- B) It controls how many chunks are retrieved from the vector store
- C) It determines how many times the query is rewritten
- D) It sets the maximum number of characters in a chunk

View answer →

✅ **Correct answer:** B) It controls how many chunks are retrieved from the vector store

📝 **Explanation :**

- The `similarity_top_k` parameter determines how many top-matching document chunks are retrieved from the vector store based on the query's similarity to the stored embeddings.
- This is a key tuning parameter that affects both:
 - Recall (did we retrieve all relevant info?), and
 - Precision (are the retrieved chunks actually relevant?).

Takeaways

- **Why optimize RAG?**
 - **Basic RAG often fails on complex queries** — especially those with emotional, ambiguous, or multi-part intent.
 - **Retrieval quality determines generation quality** — garbage in, garbage out.

- **Optimization is not optional** — it's how you move from "kind of works" to "production-ready."
- **The goal is not just relevance, but precision, grounding, and empathy.**
- **Document preprocessing**
 - **How you chunk and format documents affects retrieval quality.**
 - **Markdown pros:**
 - Preserves structure (headings, lists)
 - Human-readable
 - **Markdown cons:**
 - Over-chunking breaks context
 - Poor metadata support
 - **Best practices:**
 - Use semantic chunking (e.g., by section)
 - Add metadata (e.g., `source: coaching_guide`)
 - Avoid excessive splitting
- **Synthesis**
 - **No single technique solves everything** — combine:
 - Query rewriting → for intent clarity
 - Multi-step → for reasoning
 - HyDE → for empathy
 - Reranking → for precision
 - **The future of RAG is not retrieval — it's understanding.**
 - **Your goal:** Build systems that don't just answer, but *help*.
- **Query rewriting**
 - **Natural language is messy** — users say "I'm stuck," not "Find guidance on task blockers."
 - **Query rewriting translates intent into search-friendly queries** using an LLM.
 - **It improves retrieval accuracy** by mapping phrases to structured filters (e.g., "due today" → `Due == 'Today'`).
 - **Best for:**
 - Ambiguous queries
 - Emotional language
 - Implicit needs
 - **Always preserve context** — don't lose urgency or tone in translation.
- **Multi-step decomposition**
 - **Complex questions require complex reasoning** — decompose them into sub-questions.
 - **Example:**
 - "I'm stuck on OKRs" →
 - 1. What are the steps to finalize OKRs?

2. What help is available?
3. How do I start when overwhelmed?

- **Use tools like QueryEngineTool** to route sub-queries to different knowledge bases.
- **Trade-off:** More steps = more API calls, but deeper reasoning.

- **HyDE**

- **HyDE (Hypothetical Document Embeddings)** flips the script: generate a *hypothetical answer*, then use it as the search query.
- **It works because** the hypothetical answer matches the style and content of existing guidance documents.
- **Ideal for:**
 - Coaching or advice-based systems
 - Queries with no direct keyword match
 - Emotional or open-ended questions
- **Risk:** May retrieve irrelevant content if the hypothetical answer is off-track.

- **Reranking**

- **Vector search finds relevant content, but doesn't rank by usefulness.**
- **Reranking reorders results** using a cross-encoder model (e.g., CohereRerank) that deeply understands query-context alignment.
- **It ensures critical content (e.g., 5-minute rule) appears at the top** of the context window.
- **Best practice:** Retrieve more (e.g., top-10), then rerank to top-3.
- **Impact:** Higher faithfulness, fewer hallucinations, better answers.