AI Advances · Follow publication

Member-only story

How Do I Evaluate Chunking Strategies For RAGs

Don't guess; here's how to systematically approach it.

9 min read · 6 days ago

Thuwarakesh Murallie Follow

Listen Share More

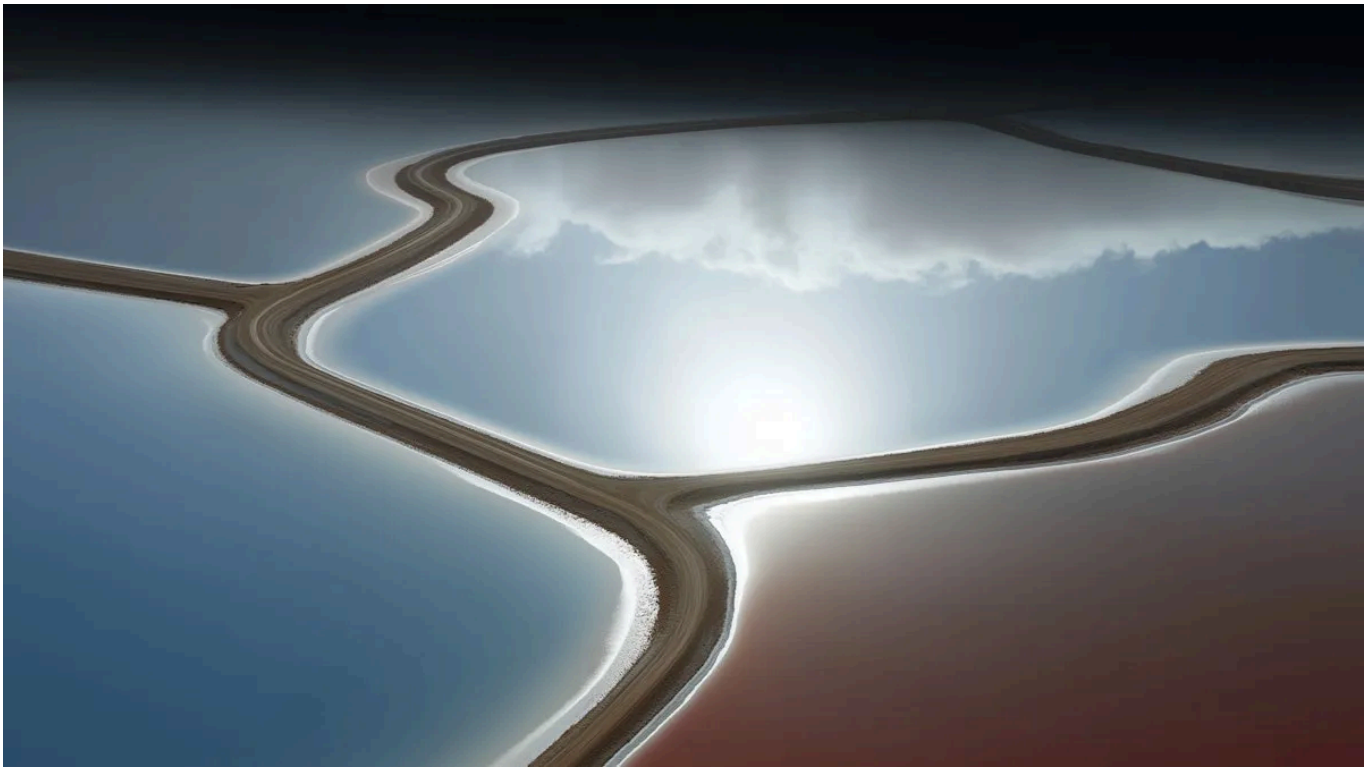


Image from Lummi.ai

I've researched RAGs extensively and know that chunking is critical to any RAG pipeline.

Many people I've talked with trust that *better models* could improve RAGs. Some put too much trust in *vector databases*. Even those who agreed that chunking is important didn't think it could significantly improve the system.

Most of them argue that *large context windows* would replace the need for chunking strategies.

But chunking techniques are here to stay. They are effective and a must for any RAG project.

However, a key question remains unanswered: How can I pick the best chunking strategy for a project?

One of These Will Be Your Favorite Chunking Technique For RAGs.
Document splitting and RAGs beyond the basics.
ai.gopubby.com

In the past, I've discussed several strategies: recursive character splitting, semantic chunking, and agentic chunking, and even argued clustering as a fast and cheap alternative to agentic chunking.

However, the bitter truth is that no strategy always works well. You can't guess which would work based on the project's nature.

The Most Valuable LLM Dev Skill is Easy to Learn, But Costly to Practice.
Here's how not to waste your budget on evaluating models and systems.
medium.com

The only way is to try them all and find the best.

So here's my approach.

Start by sampling your documents.

If you're working on a production app, you may have hundreds of gigabytes of data to process.

But you can't experiment with chunking strategies with all these data.

Cost is an apparent reason. Evaluations use LLM inferences. It may be an API call or a locally hosted LLM inference, but both come with a price tag.

Exactly How Much vRAM (and Which GPU) Can Serve Your LLM?
What if you still want to run a massive model in your small GPU?
ai.gopubby.com

The other concern is the evaluation **duration.** Your project may be on a timeline (it often is), so you don't want to waste weeks evaluating chunking techniques.

Want to move cheap & fast? Sample it.

But make sure your sample accurately represents your population.

That last one is easier said than done. I don't want to discuss sampling techniques in a post dedicated to chunking techniques, but given the nature of the project, *stratified sampling* may be a good choice.

For instance, if your organization has 100 client deliverable PPTs, 50 case study PDF documents, and 300 meeting transcripts, you can pick 10% in each as a sample. That way, you ensure every group has its representation in the sample.

Once you have the sample, you can move to the next step.

Create a test question set.

For evaluation, you need questions. The LLM will evaluate your answers to these questions.

It would be best if a human expert manually created it. However, not every organization has that luxury. Subject matter experts' time is precious and often unavailable for R&D work.

In such cases, you can get assistance from an LLM.

LLMs can create questions from the sample documents. Though I keep saying augmentation in RAG doesn't need powerful LLMs, this step does. Pick a model with good reasoning abilities to create a high-quality question set.

My suggestions? If you're on an Open AI stack, gpt-4o-mini is good enough. If not, try DeepSeek-R1-Distill-Qwen-32B.

The following code may help you get started. Make tweaks as needed for your specific requirements.

```
def create_test_questions(documents: List[Document], num_questions: int = 10) -> List[Dict[str, str]]
        """Generate test questions from documents"""
        questions = []

        # Sample text for question generation
        sample_texts = []
        for doc in documents[:3]:  # Use first 3 docs
            words = doc.page_content.split()[:500]  # First 500 words
            sample_texts.append(" ".join(words))

        combined_text = "\n\n".join(sample_texts)

        prompt = f"""Based on the following text, generate {num_questions} diverse questions that can

        Text:
        {combined_text}

        Return only the questions, one per line, without numbering or additional text."""

        response = llm.invoke(prompt)
        question_lines = [q.strip() for q in response.content.split('\n') if q.strip()]

        for i, question in enumerate(question_lines[:num_questions]):
            questions.append({
                "question": question,
                "question_id": f"q_{i+1}"
            })

        return questions
```

The code above uses Langchain. The function breaks the documents into sections of five hundred words and prompts the LLM to create questions.

Prepare for evaluation

LLM evaluation is a vast topic. We're only touching its surface in this post.

I'll be using RAGAS, a framework for evaluating RAGs. It's feature-rich, but we need only a small subset of metrics for this task.

You can install RAGAS using the command below.

```
pip install ragas
# uv add ragas
```

In case you get errors, try downgrading RAGAs to `ragas==0.1.9`.

NOTE: All the code related to this post is in this notebook. I'll discuss only snippets of this codebase in every post section.

The code below creates and evaluates a simple RAG with a chunking strategy. It uses the Langchain framework and Chroma DB as a vector store. I like Chroma DB because it's easy to set up, but if you use a different vector store, it's best to test the code with your vector store of choice.

```
from typing import List, Dict
import time
import numpy as np
from tqdm import tqdm
from datasets import Dataset
from langchain_chroma import Chroma
from ragas import evaluate
from ragas.metrics import (
    answer_relevancy,
    faithfulness,
    context_precision,
    context_recall,
    answer_correctness
)


def evaluate_strategy(strategy_name: str,
                      strategy,
                      documents: List,
                      test_questions: List[Dict[str, str]],
                      embeddings,
                      llm,
                      chroma_client,
                      top_k: int = 5) -> 'EvaluationResult':
    """Evaluate a single chunking strategy

    Args:
        strategy_name: Name of the strategy being evaluated
        strategy: The chunking strategy object with chunk_documents method
        documents: List of documents to chunk
        test_questions: List of question dictionaries
```

```
        embeddings: Embedding function for vector store
        llm: Language model for answer generation
        chroma_client: ChromaDB client instance
        top_k: Number of documents to retrieve

    Returns:
        EvaluationResult object with metrics and performance data
    """

    print(f"\nEvaluating strategy: {strategy_name}")

    # 1. Chunk documents
    chunks = strategy.chunk_documents(documents)
    print(f"Created {len(chunks)} chunks")

    # 2. Create vector store
    collection_name = f"eval_{strategy_name}_{hash(str(time.time()))}"
    collection_name = collection_name.replace("-", "_").replace(" ", "_")

    vectorstore = Chroma(
        collection_name=collection_name,
        embedding_function=embeddings,
        client=chroma_client
    )

    # Add chunks to vector store
    vectorstore.add_documents(chunks)

    # 3. Evaluate with test questions
    evaluation_data = []

    for q_data in tqdm(test_questions, desc="Processing questions"):
        question = q_data["question"]

        # Retrieve context
        start_time = time.time()
        retrieved_docs = vectorstore.similarity_search(question, k=top_k)
        retrieval_time = time.time() - start_time

        contexts = [doc.page_content for doc in retrieved_docs]

        # Generate answer
        start_time = time.time()
        context_text = "\n\n".join(contexts)
        prompt = f"Context:\n{context_text}\n\nQuestion: {question}\n\nAnswer:"

        response = llm.invoke(prompt)
        answer = response.content
        generation_time = time.time() - start_time

        # Prepare for RAGAS
        evaluation_data.append({
            "question": question,
            "answer": answer,
            "contexts": contexts,
            "ground_truth": answer,  # Using generated answer as proxy
            "retrieval_time": retrieval_time,
            "generation_time": generation_time
        })

    # 4. Run RAGAS evaluation
    dataset = Dataset.from_list(evaluation_data)

    metrics = [
        answer_relevancy,
        faithfulness,
```

```
            context_precision,
            context_recall,
            answer_correctness
        ]

        ragas_results = evaluate(dataset, metrics=metrics)

        # 5. Calculate statistics
        avg_chunk_size = np.mean([len(chunk.page_content) for chunk in chunks])
        avg_retrieval_time = np.mean([d["retrieval_time"] for d in evaluation_data])
        avg_generation_time = np.mean([d["generation_time"] for d in evaluation_data])

        # Calculate overall score (weighted average)
        overall_score = (
            ragas_results["answer_relevancy"] * 0.25 +
            ragas_results["faithfulness"] * 0.25 +
            ragas_results["context_precision"] * 0.2 +
            ragas_results["context_recall"] * 0.2 +
            ragas_results["answer_correctness"] * 0.1
        )

        result = EvaluationResult(
            strategy_name=strategy_name,
            chunk_count=len(chunks),
            avg_chunk_size=avg_chunk_size,
            retrieval_time=avg_retrieval_time,
            generation_time=avg_generation_time,
            answer_relevancy=ragas_results["answer_relevancy"],
            faithfulness=ragas_results["faithfulness"],
            context_precision=ragas_results["context_precision"],
            context_recall=ragas_results["context_recall"],
            answer_correctness=ragas_results["answer_correctness"],
            overall_score=overall_score
        )

        # Cleanup
        try:
            chroma_client.delete_collection(collection_name)
        except:
            pass

        return result
```

We can run this code with several chunking techniques and collect the results. Each result contains five evaluation metrics. We also compute the weighted average of these metrics as the overall score. The result would also contain other vital data such as chunk size, chunk count, retrieval time, and generation time.

Before returning the result, we also clean the chroma db so that any remaining data won't impact the evaluation of the following strategy.

Now that we have the evaluation data, we can visualize them to make educated decisions on chunking techniques.

Visualize evaluation results

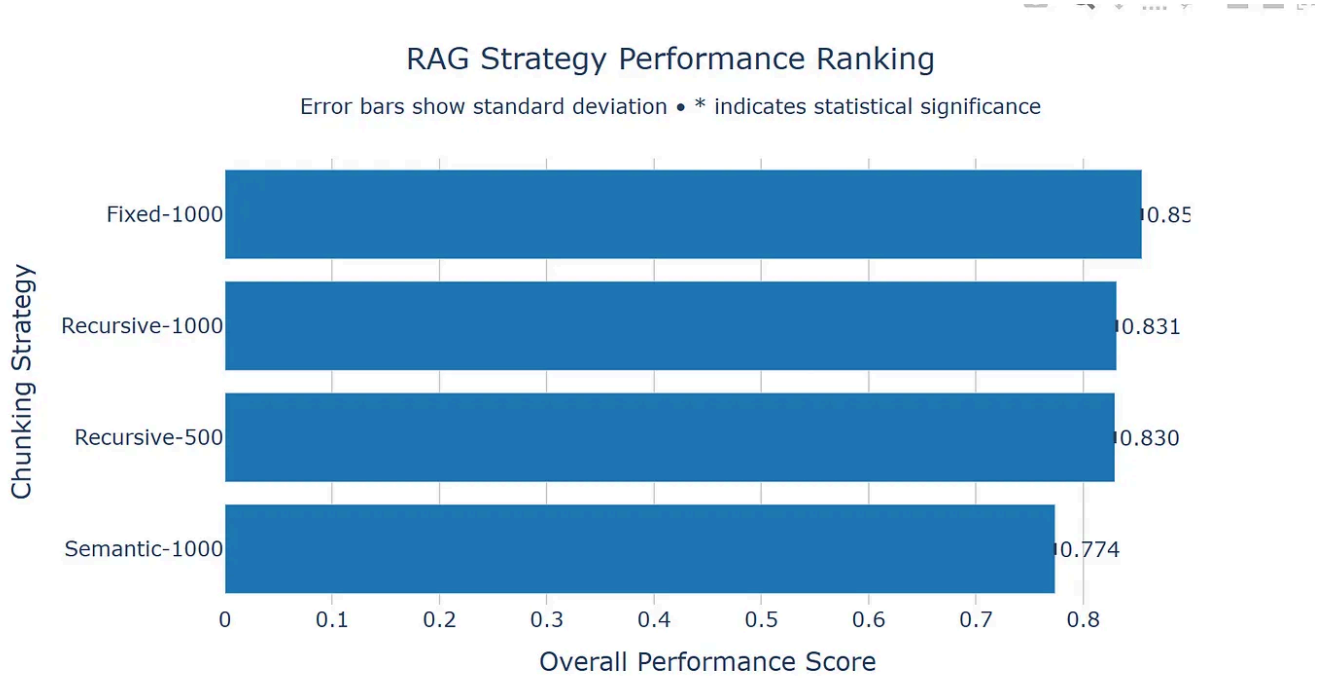
I create at least four visualizations of the results to find the best chunking technique.

You can slice and dice the data and create more helpful charts, but these would serve as a good starting point.

In the notebook I shared, I used Plotly to create the charts. I also like to create dashboards with it.

Here are the charts I create and how I use them.

Performance Ranking

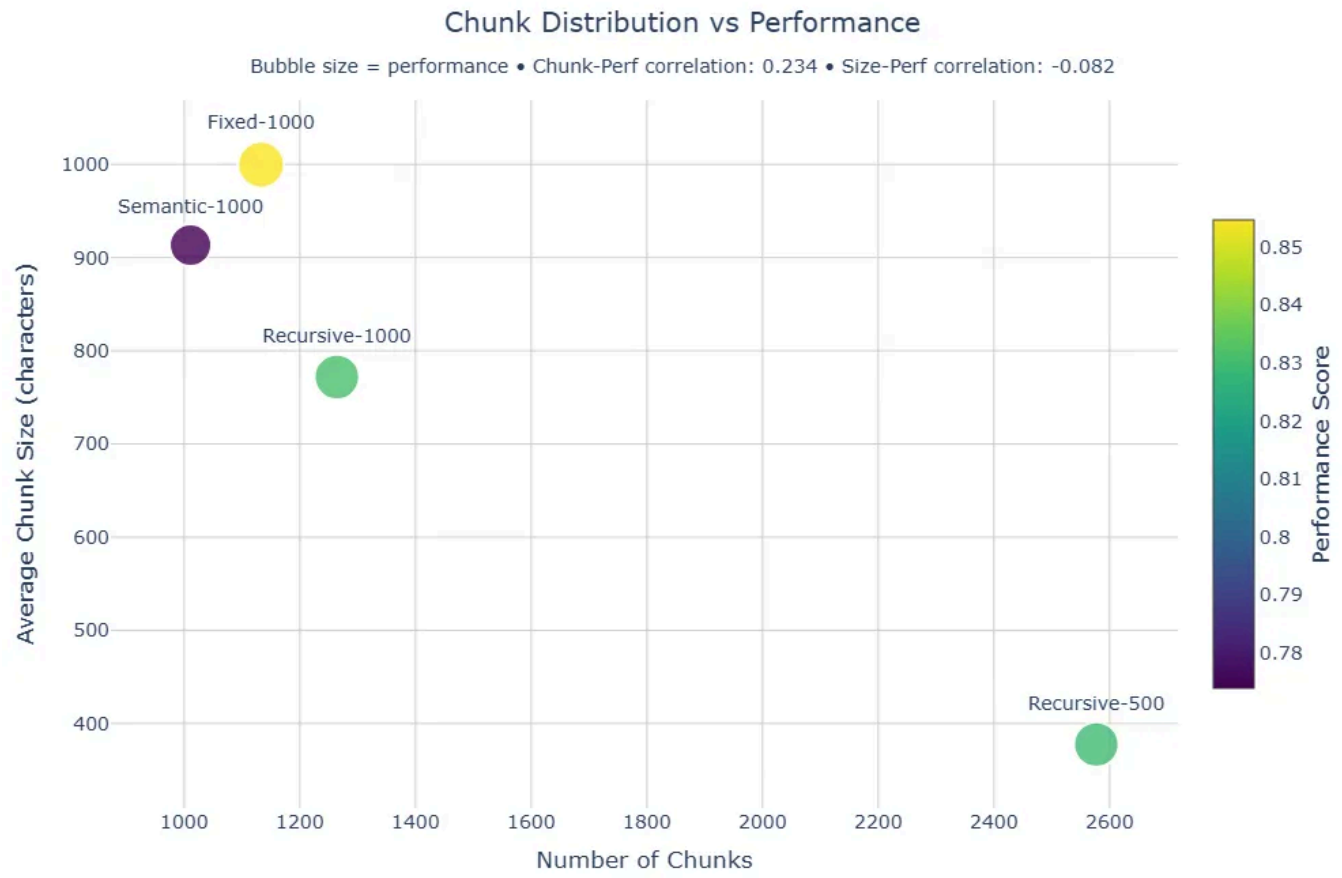


Since we calculated the overall score and the weighted average of the five individual evaluation metrics, the most sensible first step is to plot them in a bar chart.

This chart would quickly show which strategy is best and how it compares to the others.

In the example above, we can see that the fixed-1000 strategy has outperformed the other three. However, the gap seems small except for Semantic-1000, which is underperforming.

Chunk distribution vs performance



This one is a helpful chart in many ways.

In one chart, we can see the overall performance, the chunk size, and the number of chunks.

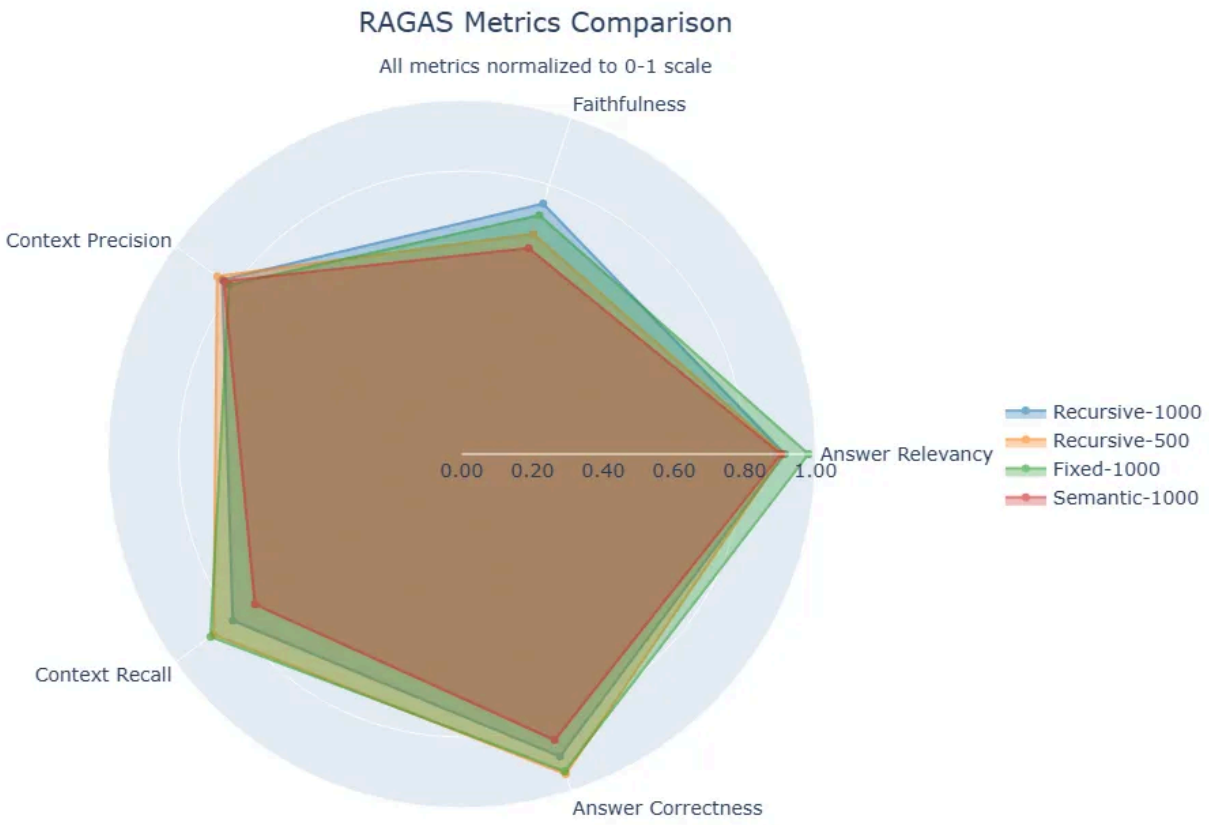
Chunk size and count are important in RAGs. We must decide how many chunks to retrieve as context for the user input. If the chunk sizes are too big, much-retrieved content might quickly fill the LLM's context window.

Excess noise may reduce the response quality even if the context window is large enough to accommodate them all.

In our example, although the fixed-1000 strategy has an excellent overall score, it creates fewer, larger chunks.

On the other extreme, recursive-500 creates superfluous chunks. But they are tiny. However, this strategy performs well too. I'd pick this if I had to draw many documents and rerank them before I generate the final answer.

Metrics Radar



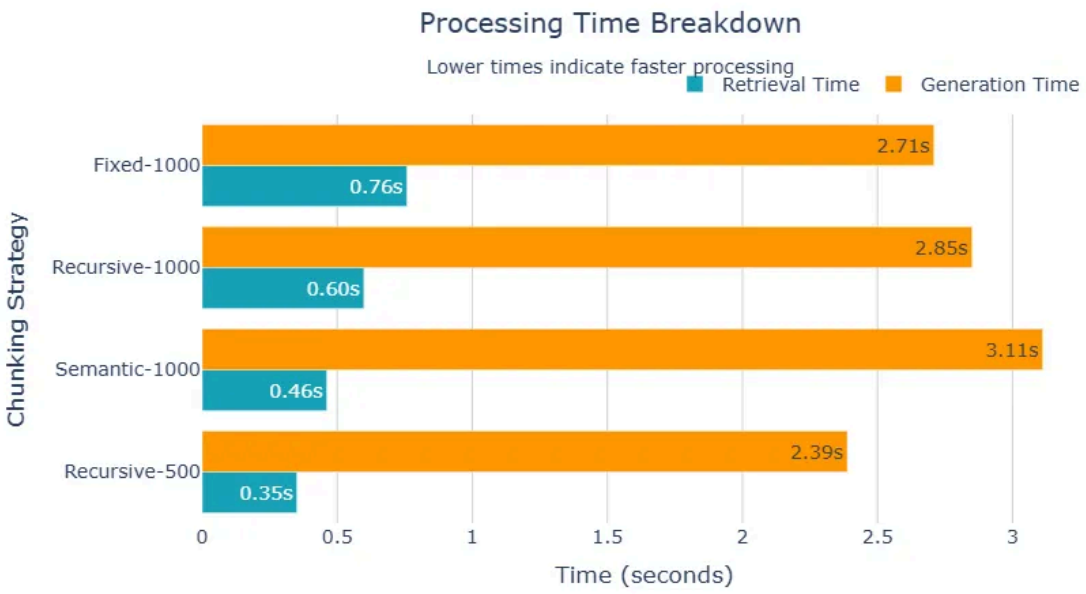
So far, we've only discussed overall scores. We haven't paid any attention to the details.

Here in this chart, we do.

A radar plot would tell a lot about the results. For specific applications, faithfulness can be paramount. However, for more casual applications, a high relevancy score is sufficient.

You must decide what is best. But you need to see them in one place, and this is the best way to do so.

Processing time breakdown



The next chart I'd like to see shows how long each strategy takes to complete. This is an operational decision we need to make.

For apps that need lower latency, this view is a must.

In our example, semantic chunking takes significantly longer than the other two. Recursive-500 is the fastest.

We have both generation time and retrieval time. Generation time is the time it took to create the chunks. In real apps, this can be a large number. Therefore, differences save the company lots of money.

The retrieval time is what contributes to latency.

Again, which one to prioritize depends on your specific application.

How do I decide (my thought process)?

So far, we've done a lot.

We've picked a sample to evaluate chunking techniques, created questions to evaluate the system, created a RAG pipeline to change chunking techniques and evaluate them with RAG, and finally, created charts with our collected results.

Now what?

Let's revisit our example.

Semantic chunking is out. Its performance is meager. It takes a lot longer to generate chunks.

Fixed-1000 has the highest overall score, but its relevancy score mainly boosts. If that's what we want above everything else, it's still a good choice. However, given the longer retrieval time and large chunk sizes, I'd still avoid it.

Both recursive methods score well in evaluation scores, but their chunk size, generation time, and retrieval time are significantly different.

I've already said I'd go for recursive-500 if I use a reranking model. However, with the additional information on processing time, I now think this is the best option.

Final thought

Chunking makes or breaks a RAG. It's the heart of RAGs.

The more time we spend on chunking techniques, the more our RAG apps will be more helpful.

However, different applications need to prioritize different aspects. Therefore, we need to evaluate chunking techniques systematically.

In this post, I've shared how I'd approach this problem.

I still have to make some guesses, but this time, they are supported by data and explainable via charts.

*Thanks for reading, friend! Besides **Medium**, I'm on **LinkedIn** and **X**, too!*

Retrieval Augmented Gen | Large Language Models | Langchain | Python | Programming

Follow

Published in AI Advances

36K followers · Last published 15 hours ago

Democratizing access to artificial intelligence

Follow

Written by Thuwarakesh Murallie

5.3K followers · 145 following

Data Science Journalist & Independent Consultant