One of These Will Be Your Favorite Chunking Technique For RAGs.

Document splitting and RAGs beyond the basics.

11 min read · Feb 18, 2025

Thuwarakesh Murallie Follow

Photo by Jason Abdilla on Unsplash

> *A RAG is only as good as its chunks. — Thuwarakesh (me)*

If your RAG app is not performing to your expectations, perhaps it's time to change your chunking strategy. **Better chunks mean better retrieval, which means high-quality responses.**

However, no chunking technique is better than the others. You need to choose based on several factors, including your project budget.

How does better chunking lead to high-quality responses?

If you're reading this, I can assume you know what chunking and RAG are. Nonetheless, here is what it is, in short.

LLMs are trained on massive public datasets. Yet, they aren't updated afterward. Therefore, LLMs don't know anything after the pretraining cutoff date. Also, your use of LLM can be about your organization's private data, which the LLM had no way of knowing.

Therefore, a beautiful solution called RAG has emerged. RAG asks the LLM to **answer questions based on the context provided in the prompt itself**. We even ask it not to answer even if the LLM knows the answer, but the provided context is insufficient.

How do we get the context? You can query your database and the Internet, skim several pages of a PDF report, or do anything else.

But there are two problems in RAGs.

1. LLM's **context windows sizes** are limited (Not anymore — I'll get to this soon!)
2. A large context window has a high **signal-to-noise ratio**.

First, early LLMs had limited window sizes. GPT 2, for instance, had only a 1024 token context window. GPT 3 came up with a 2048 token window. These are merely the *size of a typical blog post*.

Due to these limitations, the LLM prompt cannot include an organization's entire knowledge base. Engineers were forced to reduce the size of their input to the LLM to get a good response.

However, various models with a context window of 128k tokens showed up. This is usually *the size of an annual report* for many listed companies. It is good enough to upload a document to a chatbot and ask questions.

But, it didn't always perform as expected. That's because of the noise in the context. A large document easily contains many unrelated information and the necessary pieces. This unrelated information drives the LLM to lose its objective or hallucinate.

This is why we chunk the documents. Instead of sending a large document to the LLM, we break it into smaller pieces and only send the most relevant pieces.

However, this is easier said than done.

There are a million possible ways to break a document into chunks. For instance, you may break the document paragraph by paragraph, and I may do it sentence by sentence. Both are valid methods, but one may work better than the other in specific circumstances.

However, we won't discuss sentence and paragraph breaks, as they are trivial and have little use in chunking. Instead, we will discuss slightly more complex ones that break documents for RAGs.

In the rest of the post, I'll discuss a few chunking strategies I've learned and applied.

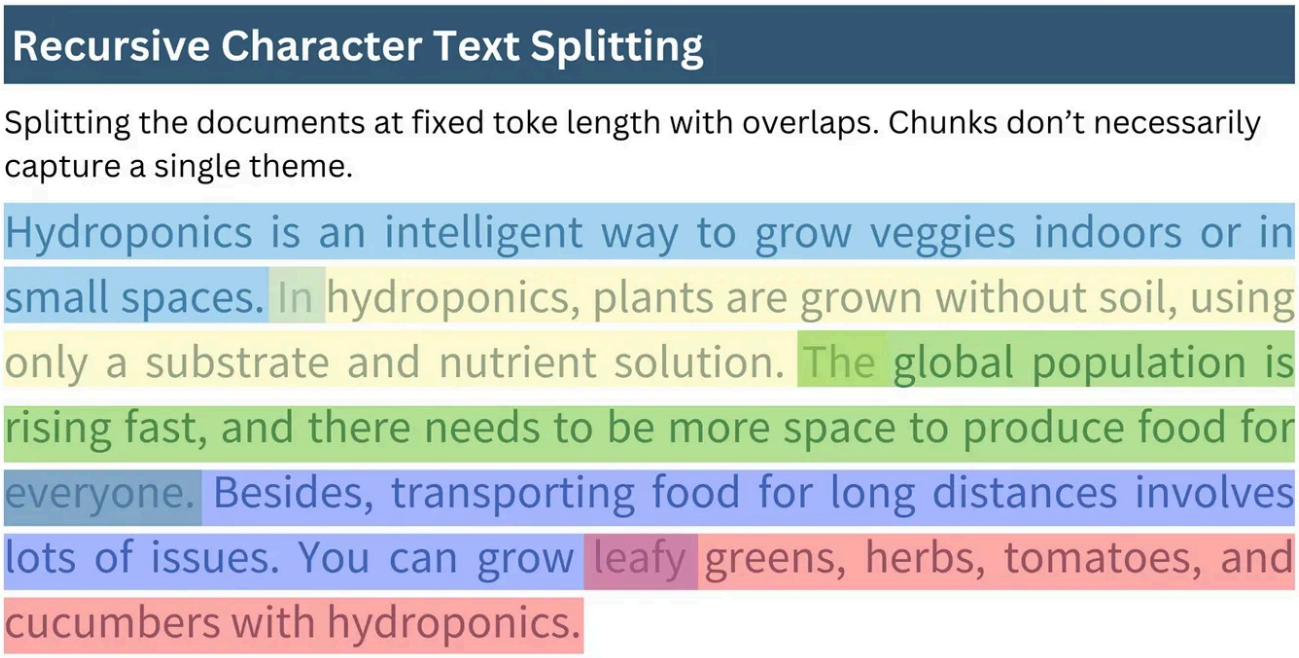
Recursive character splitting

This is the basics, you'd say.

Of course, it is. But it's also the most used in my view. Not only because it's easy to understand and implement, it's also the fastest and cheapest technique to chunk documents.

The recursive character-splitting strategy uses a **moving window of fixed chunk size** and an overlap parameter. It starts from the beginning and moves the window, keeping the overlap as specified.

Here's a visual illustration of the same.



How recursive character splitting works — Illustration by the author.

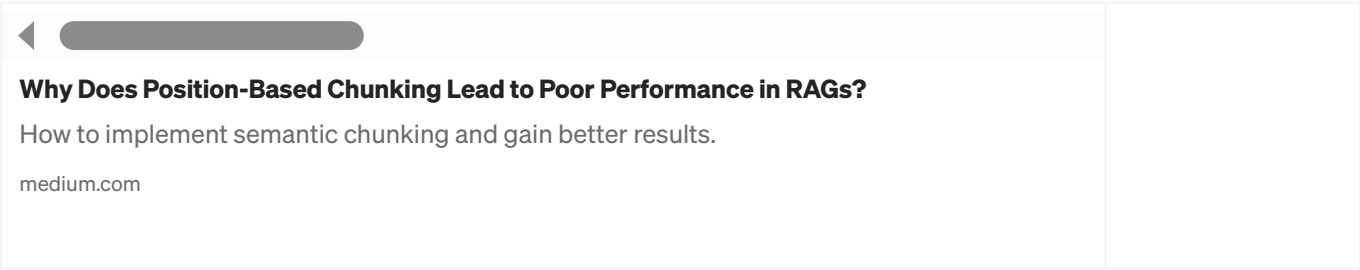
In the above illustration, the chunk size is 20 characters, and the overlap is 2. This shows how the chunks are created.

This method is very straightforward and fast. It can chunk a whole annual report in minutes. Here's how we implement it using Langchain.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

text = """
Hydroponics is an intelligent way to grow veggies indoors or in small spaces. In hydroponics, plants
"""
```

```
rc_splits = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=20, chunk_overlap=2
).split_text(text)
```

This method is a position-based splitting technique. Unfortunately, the recursive character assumes **two consecutive sentences to discuss similar things. But it doesn't have to be.**



In the same chapter of a book, we may talk about a few different things and combine them at last to show the connection. But the relationship isn't clear until the connection is revealed at the end.

Now, if you have sufficiently large chunks, you can capture the entire segment about individual ideas and their connections. However, this method doesn't work if you're working on a large project.

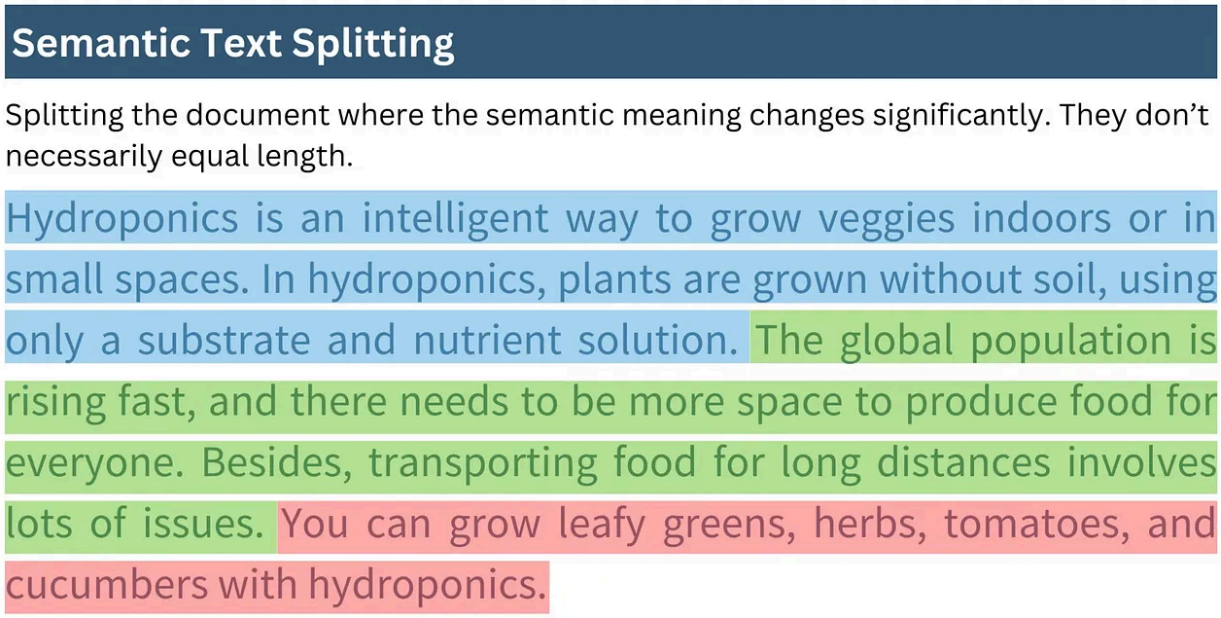
We need a technique that splits the text based on its semantic meaning — not its position.

Semantic chunking

Semantic chunking is a different approach to the problem. Instead of relying on the positions, we rely on the semantic meaning.

We scan through the document and break whenever the semantic meaning significantly changes from its current segments.

See the visualization of a semantically chunked text below.



How semantic chunking works — Illustration by the author.

The first two sentences discuss hydroponic farming, while the following two discuss global issues. Then, it returns to hydroponics.

A later paragraph reveals the link between hydroponics and global population issues. However, as of this paragraph, they are significantly different segments.

If you take another look at the illustration, the chunks created with the semantic chunking technique are of **varying lengths**. That's because we wait until the semantic meaning changes — not when a fixed length of words has been achieved.

But this method is more straightforward said than done.

The challenge is to determine the meaning of a sentence programmatically. This is done through an embedding model. Embedding models like the OpenAI's text-embedding-3-large will convert a sentence into vectors in a way that captures the semantic meaning behind them.

The semantic chunking process takes five steps to complete.

First, we break the text into sentences/paragraphs and create initial chunks by emerging adjacent ones. We **then** create vector embeddings for these chunks. The **third** step involves computing the distances between every two consecutive chunks. We can now use a threshold value to break chunks when the distance computed in the last step goes beyond a certain point — that's the **fourth** step.

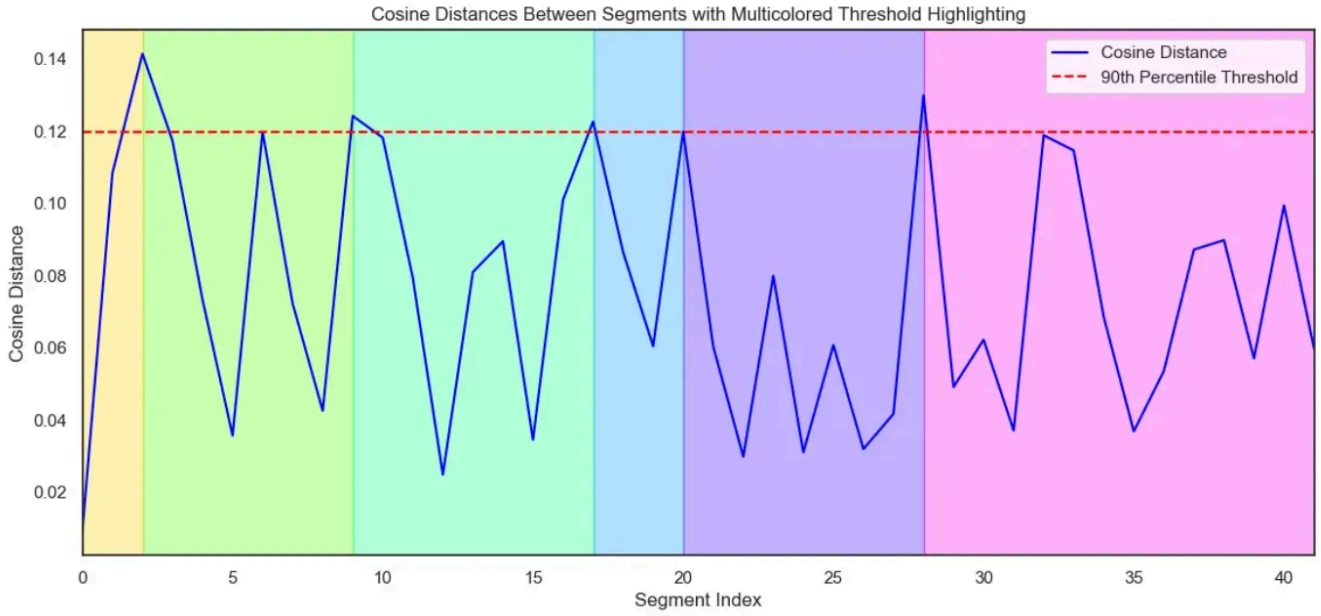
The fifth step is an optional visualization. But I'd recommend you create this as it gives you more confidence in the chunks you created. Sure, you can't do it for large projects. But you can at least do it for a sample.

Here's a complete code for doing this.

```
# Step 1 : Create initial chunks by combining concecutive sentences.
# ---------------------------------------------------------------------

#Split the text into individual sentences.
sentences = re.split(r"(?<=[.?!])\s+", text)
initial_chunks = [
    {"chunk": str(sentence), "index": i} for i, sentence in enumerate(sentences)
]

# Function to combine chunks with overlapping sentences
def combine_chunks(chunks):
    for i in range(len(chunks)):
        combined_chunk = ""

        if i > 0:
            combined_chunk += chunks[i - 1]["chunk"]

        combined_chunk += chunks[i]["chunk"]

        if i < len(chunks) - 1:
            combined_chunk += chunks[i + 1]["chunk"]

        chunks[i]["combined_chunk"] = combined_chunk

    return chunks

# Combine chunks
combined_chunks = combine_chunks(initial_chunks)


# Step 2 : Create embeddings for the initial chunks.
# ---------------------------------------------------------------------

# Embed the combined chunks
chunk_embeddings = embeddings.embed_documents(
    [chunk["combined_chunk"] for chunk in combined_chunks]

    # If you haven't created combined_chunk, use the following.
    # [chunk["chunk"] for chunk in combined_chunks]
)

# Add embeddings to chunks
for i, chunk in enumerate(combined_chunks):
    chunk["embedding"] = chunk_embeddings[i]


# Step 3 : Calculate distance between the chunks
# ---------------------------------------------------------------------

def calculate_cosine_distances(chunks):
    distances = []
    for i in range(len(chunks) - 1):
        current_embedding = chunks[i]["embedding"]
        next_embedding = chunks[i + 1]["embedding"]

        similarity = cosine_similarity([current_embedding], [next_embedding])[0][0]
        distance = 1 - similarity

        distances.append(distance)
        chunks[i]["distance_to_next"] = distance

    return distances

# Calculate cosine distances
distances = calculate_cosine_distances(combined_chunks)
```

```
# Step 4 : Find chunks with significant different to it's previous ones.
# ------------------------------------------------------------------------

import numpy as np

threshold_percentile = 90

threshold_value = np.percentile(cosine_distances, threshold_percentile)

crossing_points = [
    i for i, distance in enumerate(distances) if distance > threshold_value
]

len(crossing_points)


# Step 5 (Optional) : Create a plot of chunk distances to get a better view
# ------------------------------------------------------------------------

import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np


def visualize_cosine_distances_with_thresholds_multicolored(
    cosine_distances, threshold_percentile=90
):
    # Calculate the threshold value based on the percentile
    threshold_value = np.percentile(cosine_distances, threshold_percentile)

    # Identify the points where the cosine distance crosses the threshold
    crossing_points = [0]  # Start with the first segment beginning at index 0

    crossing_points += [
        i
        for i, distance in enumerate(cosine_distances)
        if distance > threshold_value
    ]

    crossing_points.append(
        len(cosine_distances)
    )  # Ensure the last segment goes to the end

    # Set up the plot
    plt.figure(figsize=(14, 6))
    sns.set(style="white")  # Change to white to turn off gridlines

    # Plot the cosine distances
    sns.lineplot(
        x=range(len(cosine_distances)),
        y=cosine_distances,
        color="blue",
        label="Cosine Distance",
    )

    # Plot the threshold line
    plt.axhline(
        y=threshold_value,
        color="red",
        linestyle="--",
        label=f"{threshold_percentile}th Percentile Threshold",
    )

    # Highlight segments between threshold crossings with different colors
```

```
        colors = sns.color_palette(
            "hsv", len(crossing_points) - 1
        )  # Use a color palette for segments
        for i in range(len(crossing_points) - 1):
            plt.axvspan(
                crossing_points[i], crossing_points[i + 1], color=colors[i], alpha=0.3
            )

        # Add labels and title
        plt.title(
            "Cosine Distances Between Segments with Multicolored Threshold Highlighting"
        )
        plt.xlabel("Segment Index")
        plt.ylabel("Cosine Distance")
        plt.legend()

        # Adjust the x-axis limits to remove extra space
        plt.xlim(0, len(cosine_distances) - 1)

        # Display the plot
        plt.show()

        return crossing_points

    # Example usage with cosine_distances and threshold_percentile
    crossing_poings = visualize_cosine_distances_with_thresholds_multicolored(
        distances, threshold_percentile=bp_threashold
    )
```



Seaborn chart to illustrate semantic chunking — Image by the author.

As you can see, whenever the distance between the chunks spikes beyond 0.12, we create a bigger chunk by combining every smaller chunk up to that point. This process has created six chunks of varying lengths.

Agentic chunking

You may be tired of hearing the word agent these days. I get it.

Agentic chunking is an intelligent process to create chunking. I'd say it's a step further than semantic chunking.

Here's the prime difference. People don't think in a strict order. We tend to jump here and there with our thoughts — And so do our writings.

On a side note, this is one easy way to determine if AI writes something.

Back to the discussion,

> The drawback of recursive character splitting and semantic chunking is that they assume similar ideas are always close to each other in a document.

This is rarely the case in many documents we see.

Agentic chunking splits the document like an actual human would do. We **use an LLM** to skim through the text to find new ideas; if the text falls to an existing idea, that part is put into the existing bucket (chunk).

How to Achieve Near Human-Level Performance in Chunking for RAGs

The costly yet powerful splitting technique for superior RAG retrieval

medium.com

But if the agent sees the idea for the first time, it creates a new bucket (chunk) to put this text.

Here's the code implementation.

```
from langchain import hub
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.pydantic_v1 import BaseModel, Field

# Step 1: Convert paragraphs to propositions.
# ----------------------------------------------

# Load the propositioning prompt from langchain hub
obj = hub.pull("wfh/proposal-indexing")

# Pick the LLM
llm = ChatOpenAI(model="gpt-4o")

# A Pydantic model to extract sentences from the passage
class Sentences(BaseModel):
    sentences: List[str]

extraction_llm = llm.with_structured_output(Sentences)

# Create the sentence extraction chain
extraction_chain = obj | extraction_llm

# NOTE: text is your actual document
paragraphs = text.split("\n\n")

propositions = []
```

```
for i, p in enumerate(paragraphs):
    propositions = extraction_chain.invoke(p

    propositions.extend(propositions)

# Step 2: Create a placeholder to store chunks
chunks = {}

# Step 3: Deine helper classes and functions for agentic chunking.
class ChunkMeta(BaseModel):
    title: str = Field(description="The title of the chunk.")
    summary: str = Field(description="The summary of the chunk.")


def create_new_chunk(chunk_id, proposition):
    summary_llm = llm.with_structured_output(ChunkMeta)

    summary_prompt_template = ChatPromptTemplate.from_messages(
        [
            (
                "system",
                "Generate a new summary and a title based on the propositions.",
            ),
            (
                "user",
                "propositions:{propositions}",
            ),
        ]
    )

    summary_chain = summary_prompt_template | summary_llm

    chunk_meta = summary_chain.invoke(
        {
            "propositions": [proposition],
        }
    )

    chunks[chunk_id] = {
        "summary": chunk_meta.summary,
        "title": chunk_meta.title,
        "propositions": [proposition],
    }


def add_proposition(chunk_id, proposition):
    summary_llm = llm.with_structured_output(ChunkMeta)

    summary_prompt_template = ChatPromptTemplate.from_messages(
        [
            (
                "system",
                "If the current_summary and title is still valid for the propositions return them."
                "If not generate a new summary and a title based on the propositions.",
            ),
            (
                "user",
                "current_summary:{current_summary}\n\ncurrent_title:{current_title}\n\npropositions:{
            ),
        ]
    )

    summary_chain = summary_prompt_template | summary_llm

    chunk = chunks[chunk_id]
```

```
    current_summary = chunk["summary"]
    current_title = chunk["title"]
    current_propositions = chunk["propositions"]

    all_propositions = current_propositions + [proposition]

    chunk_meta = summary_chain.invoke(
        {
            "current_summary": current_summary,
            "current_title": current_title,
            "propositions": all_propositions,
        }
    )

    chunk["summary"] = chunk_meta.summary
    chunk["title"] = chunk_meta.title
    chunk["propositions"] = all_propositions


# Step 5: The main functino that creates chunks from propositions.
def find_chunk_and_push_proposition(proposition):

    class ChunkID(BaseModel):
        chunk_id: int = Field(description="The chunk id.")

    allocation_llm = llm.with_structured_output(ChunkID)

    allocation_prompt = ChatPromptTemplate.from_messages(
        [
            (
                "system",
                "You have the chunk ids and the summaries"
                "Find the chunk that best matches the proposition."
                "If no chunk matches, return a new chunk id."
                "Return only the chunk id.",
            ),
            (
                "user",
                "proposition:{proposition}" "chunks_summaries:{chunks_summaries}",
            ),
        ]
    )

    allocation_chain = allocation_prompt | allocation_llm

    chunks_summaries = {
        chunk_id: chunk["summary"] for chunk_id, chunk in chunks.items()
    }

    best_chunk_id = allocation_chain.invoke(
        {"proposition": proposition, "chunks_summaries": chunks_summaries}
    ).chunk_id

    if best_chunk_id not in chunks:
        best_chunk_id = create_new_chunk(best_chunk_id, proposition)
        return

    add_proposition(best_chunk_id, proposition)
```

In the above example, we did something called propositioning before working on the chunking process. It's a method to convert every sentence to be self-explanatory. Consider the following example:

```
Original text
====================
A crow sits near the pond. It's a white one.

Propositioned text
==================
A crow sits near the pond. Thise crow is a white one.
```

This helps the LLM understand the text more clearly. It doesn't have to wonder what “It” and “that” means in every sentence.

Chunks created with an agentic method like this don’t have to have only the neighboring texts. In a large document, agentic chunking can bring similar ideas wherever they are from.

In summary,

In this post, I’ve discussed three chunking strategies that are popularly used. Each has its own merits. But none of them is a silver bullet that works every time.

Recursive character splitting is the most straightforward and cheapest. This is your budget-friendly option to try out. It’s very good for prototyping, and if you’re lucky, you’d get decent accuracy in production, too.

Semantic chunking is slower but not too costly. It can create chunks that discuss one specific idea. This isn’t possible with recursive splitting. Thus, it can produce better results.

However, both these methods assume that everything related is spoken together. But in reality, we don’t do this. A better approach involves LLM—agentic chunking. Unfortunately, this can be very slow and expensive.

But depending on your use case, you will find one of these your favorite.

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

Llm | OpenAI | Data Science | Retrieval Augmented | Agentic Ai

Follow

Published in AI Advances

36K followers · Last published 15 hours ago

Democratizing access to artificial intelligence