Medium   Q   Search   ✏ Write   ⬤

LlamaIndex Blog

A Cheat Sheet and Some Recipes For Building Advanced RAG

⬤ Andrei   (Follow)   7 min read · Jan 5, 2024

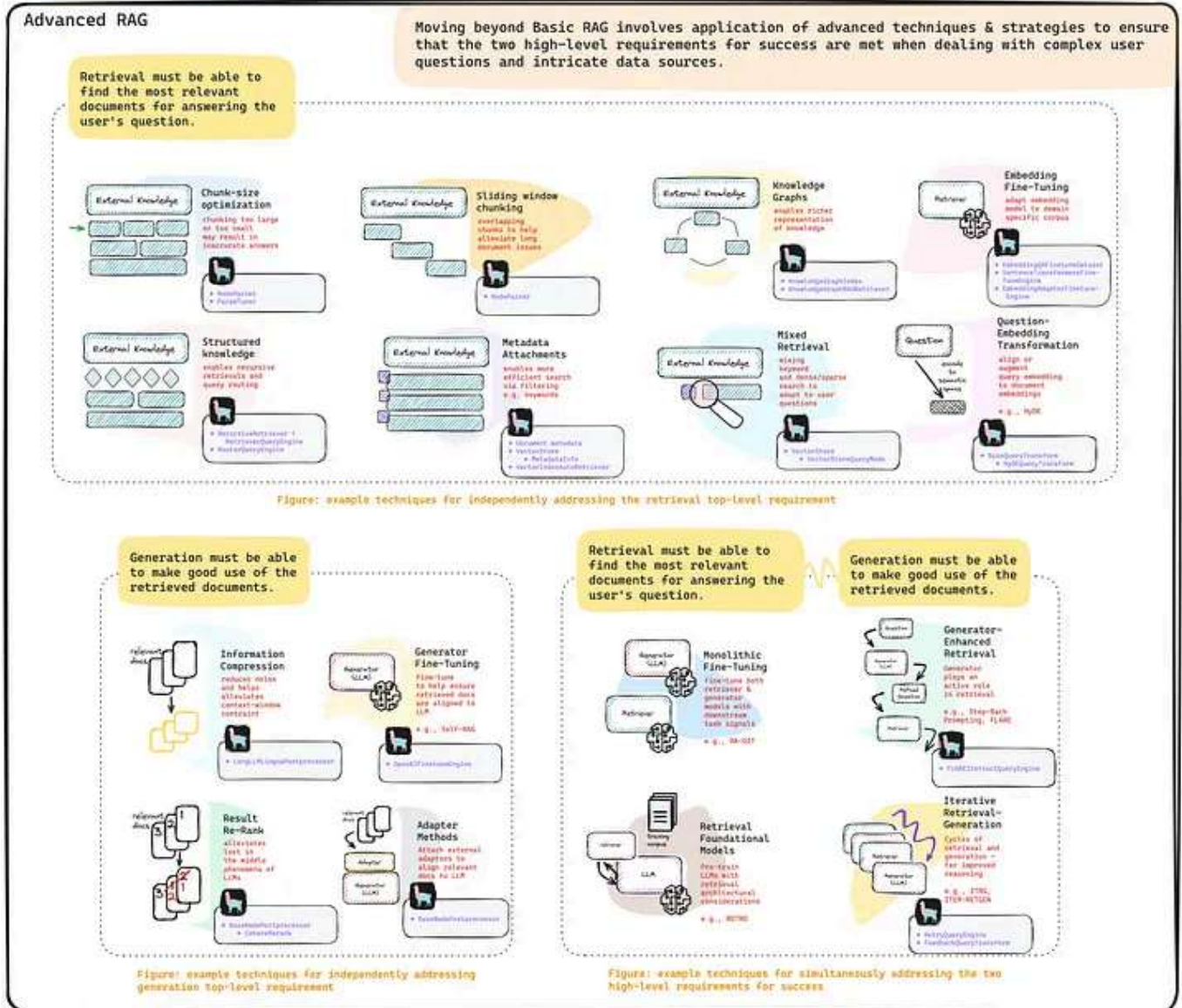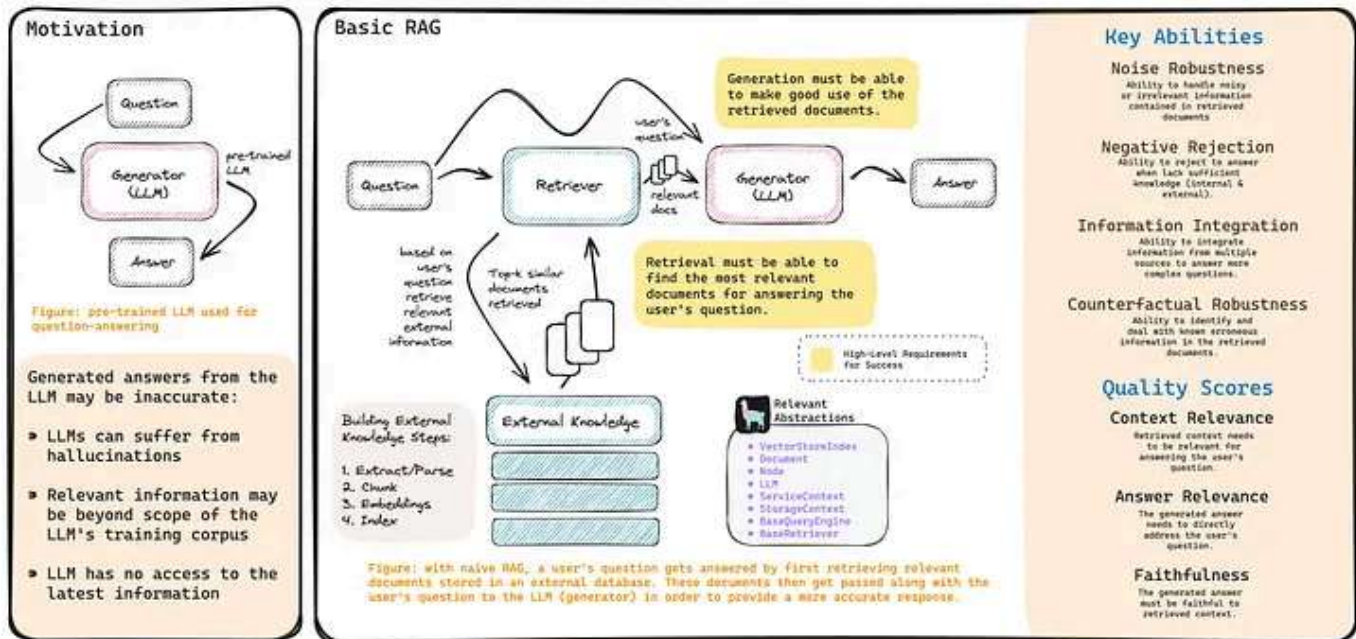👏 1.8K   💬 7   🔖 ▶ ↑

A comprehensive RAG Cheat Sheet detailing motivations for RAG as well as techniques and strategies for progressing beyond Basic or Naive RAG builds. (high-resolution version)

It's the start of a new year and perhaps you're looking to break into the RAG scene by building your very first RAG system. Or, maybe you've built Basic RAG systems and are now looking to enhance them to something more advanced in order to better handle your user's queries and data structures.

In either case, knowing where or how to begin may be a challenge in and of itself! If that's true, then hopefully this blog post points you in the right direction for your next steps, and moreover, provides for you a mental model for you to anchor your decisions when building advanced RAG systems.

The RAG cheat sheet shared above was greatly inspired by a recent RAG survey paper ("Retrieval-Augmented Generation for Large Language Models: A Survey" Gao, Yunfan, et al. 2023).

Basic RAG

Mainstream RAG as defined today involves retrieving documents from an external knowledge database and passing these along with the user's query to an LLM for response generation. In other words, RAG involves a Retrieval component, an External Knowledge database and a Generation component.

LlamaIndex Basic RAG Recipe:

```
from llama_index import SimpleDirectoryReader, VectorStoreIndex

# load data
documents = SimpleDirectoryReader(input_dir="...").load_data()

# build VectorStoreIndex that takes care of chunking documents
# and encoding chunks to embeddings for future retrieval
index = VectorStoreIndex.from_documents(documents=documents)

# The QueryEngine class is equipped with the generator
```

```
    # and facilitates the retrieval and generation steps
    query_engine = index.as_query_engine()

    # Use your Default RAG
    response = query_engine.query("A user's query")
```

Success Requirements for RAG

In order for a RAG system to be deemed as a success (in the sense of providing useful and relevant answers to user questions), there are really only two high level requirements:

1. Retrieval must be able to find the most relevant documents to a user query.

2. Generation must be able to make good use of the retrieved documents to sufficiently answer the user query.

Advanced RAG

With the success requirements defined, we can then say that building advanced RAG is really about the application of more sophisticated techniques and strategies (to the Retrieval or Generation components) to ensure that they are ultimately met. Furthermore, we can categorize a sophisticated technique as either one that addresses one of the two high-level success requirements independent (more or less) of the other, or one that addresses both of these requirements simultaneously.

Advanced techniques for Retrieval must be able to find the most relevant documents to a user query

Below we briefly describe a couple of the more sophisticated techniques to help achieve the first success requirement.

1. **Chunk-Size Optimization:** Since LLMs are restricted by context length, it is necessary to chunk documents when building the External Knowledge database. Chunks that are too big or too small can pose problems for the Generation component leading to in accurate responses.

LlamaIndex Chunk Size Optimization Recipe (notebook guide):

```python
from llama_index import ServiceContext
from llama_index.param_tuner.base import ParamTuner, RunResult
from llama_index.evaluation import SemanticSimilarityEvaluator, BatchEvalRunner

### Recipe
### Perform hyperparameter tuning as in traditional ML via grid-search
### 1. Define an objective function that ranks different parameter combos
### 2. Build ParamTuner object
### 3. Execute hyperparameter tuning with ParamTuner.tune()

# 1. Define objective function
def objective_function(params_dict):
    chunk_size = params_dict["chunk_size"]
    docs = params_dict["docs"]
    top_k = params_dict["top_k"]
    eval_qs = params_dict["eval_qs"]
    ref_response_strs = params_dict["ref_response_strs"]

    # build RAG pipeline
    index = _build_index(chunk_size, docs)  # helper function not shown here
    query_engine = index.as_query_engine(similarity_top_k=top_k)

    # perform inference with RAG pipeline on a provided questions `eval_qs`
    pred_response_objs = get_responses(
        eval_qs, query_engine, show_progress=True
    )

    # perform evaluations of predictions by comparing them to reference
    # responses `ref_response_strs`
    evaluator = SemanticSimilarityEvaluator(...)
    eval_batch_runner = BatchEvalRunner(
        {"semantic_similarity": evaluator}, workers=2, show_progress=True
    )
    eval_results = eval_batch_runner.evaluate_responses(
        eval_qs, responses=pred_response_objs, reference=ref_response_strs
    )
```

```python
        # get semantic similarity metric
        mean_score = np.array(
            [r.score for r in eval_results["semantic_similarity"]]
        ).mean()

        return RunResult(score=mean_score, params=params_dict)

    # 2. Build ParamTuner object
    param_dict = {"chunk_size": [256, 512, 1024]} # params/values to search over
    fixed_param_dict = { # fixed hyperparams
      "top_k": 2,
        "docs": docs,
        "eval_qs": eval_qs[:10],
        "ref_response_strs": ref_response_strs[:10],
    }
    param_tuner = ParamTuner(
        param_fn=objective_function,
        param_dict=param_dict,
        fixed_param_dict=fixed_param_dict,
        show_progress=True,
    )

    # 3. Execute hyperparameter search
    results = param_tuner.tune()
    best_result = results.best_run_result
    best_chunk_size = results.best_run_result.params["chunk_size"]
```

2. Structured External Knowledge: In complex scenarios, it may be necessary to build your external knowledge with a bit more structure than the basic vector index so as to permit recursive retrievals or routed retrieval when dealing with sensibly separated external knowledge sources.

LlamaIndex Recursive Retrieval Recipe (notebook guide):

```python
    from llama_index import SimpleDirectoryReader, VectorStoreIndex
    from llama_index.node_parser import SentenceSplitter
    from llama_index.schema import IndexNode
```

```python
### Recipe
### Build a recursive retriever that retrieves using small chunks
### but passes associated larger chunks to the generation stage

# load data
documents = SimpleDirectoryReader(
    input_file="some_data_path/llama2.pdf"
).load_data()

# build parent chunks via NodeParser
node_parser = SentenceSplitter(chunk_size=1024)
base_nodes = node_parser.get_nodes_from_documents(documents)

# define smaller child chunks
sub_chunk_sizes = [256, 512]
sub_node_parsers = [
    SentenceSplitter(chunk_size=c, chunk_overlap=20) for c in sub_chunk_sizes
]
all_nodes = []
for base_node in base_nodes:
    for n in sub_node_parsers:
        sub_nodes = n.get_nodes_from_documents([base_node])
        sub_inodes = [
            IndexNode.from_text_node(sn, base_node.node_id) for sn in sub_nodes
        ]
        all_nodes.extend(sub_inodes)
    # also add original node to node
    original_node = IndexNode.from_text_node(base_node, base_node.node_id)
    all_nodes.append(original_node)

# define a VectorStoreIndex with all of the nodes
vector_index_chunk = VectorStoreIndex(
    all_nodes, service_context=service_context
)
vector_retriever_chunk = vector_index_chunk.as_retriever(similarity_top_k=2)

# build RecursiveRetriever
all_nodes_dict = {n.node_id: n for n in all_nodes}
retriever_chunk = RecursiveRetriever(
    "vector",
    retriever_dict={"vector": vector_retriever_chunk},
    node_dict=all_nodes_dict,
    verbose=True,
)

# build RetrieverQueryEngine using recursive_retriever
query_engine_chunk = RetrieverQueryEngine.from_args(
    retriever_chunk, service_context=service_context
)
```

```
    # perform inference with advanced RAG (i.e. query engine)
    response = query_engine_chunk.query(
        "Can you tell me about the key concepts for safety finetuning"
    )
```

Other useful links

Get Andrei's stories in your inbox

Join Medium for free to get updates from this writer.

Enter your email   Subscribe

We have several of guides demonstrating the application of other advanced techniques to help ensure accurate retrieval in complex cases. Here are links to a select few of them:

1. Building External Knowledge using Knowledge Graphs

2. Performing Mixed Retrieval with Auto Retrievers

3. Building Fusion Retrievers

4. Fine-tuning Embedding Models used in Retrieval

5. Transforming Query Embeddings (HyDE)

Advanced techniques for Generation must be able to make good use of the retrieved documents

Similar to previous section, we provide a couple of examples of the sophisticated techniques under this category, which can be described as

ensuring that the retrieved documents are well aligned to the LLM of the Generator.

1. **Information Compression:** Not only are LLMs are restricted by context length, but there can be response degradation if the retrieved documents carry too much noise (i.e. irrelevant information).

LlamaIndex Information Compression Recipe (<u>notebook guide</u>)**:**

```python
from llama_index import SimpleDirectoryReader, VectorStoreIndex
from llama_index.query_engine import RetrieverQueryEngine
from llama_index.postprocessor import LongLLMLinguaPostprocessor

### Recipe
### Define a Postprocessor object, here LongLLMLinguaPostprocessor
### Build QueryEngine that uses this Postprocessor on retrieved docs

# Define Postprocessor
node_postprocessor = LongLLMLinguaPostprocessor(
    instruction_str="Given the context, please answer the final question",
    target_token=300,
    rank_method="longllmlingua",
    additional_compress_kwargs={
        "condition_compare": True,
        "condition_in_question": "after",
        "context_budget": "+100",
        "reorder_context": "sort",  # enable document reorder
    },
)

# Define VectorStoreIndex
documents = SimpleDirectoryReader(input_dir="...").load_data()
index = VectorStoreIndex.from_documents(documents)

# Define QueryEngine
retriever = index.as_retriever(similarity_top_k=2)
retriever_query_engine = RetrieverQueryEngine.from_args(
    retriever, node_postprocessors=[node_postprocessor]
)
```

```python
# Used your advanced RAG
response = retriever_query_engine.query("A user query")
```

2. Result Re-Rank: LLMs suffer from the so-called "Lost in the Middle" phenomena which stipulates that LLMs focus on the extreme ends of the prompts. In light of this, it is beneficial to re-rank retrieved documents before passing them off to the Generation component.

LlamaIndex Re-Ranking For Better Generation Recipe (<u>notebook guide</u>):

```python
import os
from llama_index import SimpleDirectoryReader, VectorStoreIndex
from llama_index.postprocessor.cohere_rerank import CohereRerank
from llama_index.postprocessor import LongLLMLinguaPostprocessor

### Recipe
### Define a Postprocessor object, here CohereRerank
### Build QueryEngine that uses this Postprocessor on retrieved docs

# Build CohereRerank post retrieval processor
api_key = os.environ["COHERE_API_KEY"]
cohere_rerank = CohereRerank(api_key=api_key, top_n=2)

# Build QueryEngine (RAG) using the post processor
documents = SimpleDirectoryReader("./data/paul_graham/").load_data()
index = VectorStoreIndex.from_documents(documents=documents)
query_engine = index.as_query_engine(
    similarity_top_k=10,
    node_postprocessors=[cohere_rerank],
)

# Use your advanced RAG
response = query_engine.query(
    "What did Sam Altman do in this essay?"
)
```

Advanced techniques for simultaneously addressing Retrieval and Generation success requirements

In this sub section, we consider sophisticated methods that use the synergy of retrieval and generation in order to achieve both better retrieval as well as more accurate generated responses to user queries).

1. **Generator-Enhanced Retrieval:** These techniques make use of the LLM's inherent reasoning abilities to refine the user query before retrieval is performed so as to better indicate what exactly it requires to provide a useful response.

LlamaIndex Generator-Enhanced Retrieval Recipe (<u>notebook guide</u>):

```python
from llama_index.llms import OpenAI
from llama_index.query_engine import FLAREInstructQueryEngine
from llama_index import (
    VectorStoreIndex,
    SimpleDirectoryReader,
    ServiceContext,
)
### Recipe
### Build a FLAREInstructQueryEngine which has the generator LLM play
### a more active role in retrieval by prompting it to elicit retrieval
### instructions on what it needs to answer the user query.

# Build FLAREInstructQueryEngine
documents = SimpleDirectoryReader("./data/paul_graham").load_data()
index = VectorStoreIndex.from_documents(documents)
index_query_engine = index.as_query_engine(similarity_top_k=2)
service_context = ServiceContext.from_defaults(llm=OpenAI(model="gpt-4"))
flare_query_engine = FLAREInstructQueryEngine(
    query_engine=index_query_engine,
    service_context=service_context,
    max_iterations=7,
    verbose=True,
)

# Use your advanced RAG
```

```
response = flare_query_engine.query(
    "Can you tell me about the author's trajectory in the startup world?"
)
```

2. Iterative Retrieval-Generator RAG: For some complex cases, multi-step reasoning may be required to provide a useful and relevant answer to the user query.

LlamaIndex Iterative Retrieval-Generator Recipe (<u>notebook guide</u>**):**

```python
from llama_index.query_engine import RetryQueryEngine
from llama_index.evaluation import RelevancyEvaluator

### Recipe
### Build a RetryQueryEngine which performs retrieval-generation cycles
### until it either achieves a passing evaluation or a max number of
### cycles has been reached

# Build RetryQueryEngine
documents = SimpleDirectoryReader("./data/paul_graham").load_data()
index = VectorStoreIndex.from_documents(documents)
base_query_engine = index.as_query_engine()
query_response_evaluator = RelevancyEvaluator() # evaluator to critique
                                                # retrieval-generation cycles

retry_query_engine = RetryQueryEngine(
    base_query_engine, query_response_evaluator
)

# Use your advanced rag
retry_response = retry_query_engine.query("A user query")
```

Measurement Aspects of RAG

Evaluating RAG systems are, of course, of utmost importance. In their survey paper, Gao, Yunfan et al. indicate 7 measurement aspects as seen in the top-

right portion of the attached RAG cheat sheet. The llama-index library consists of several evaluation abstractions as well as integrations to RAGAs in order to help builders gain an understanding of the level to which their RAG system achieves the success requirements through the lens of these measurement aspects. Below, we list a select few of the evaluation notebook guides.

1. Answer Relevancy and Context Relevancy

2. Faithfulness

3. Retrieval Evaluation

4. Batch Evaluations with BatchEvalRunner

You're Now Equipped To Do Advanced RAG

After reading this blog post, we hope that you feel more equipped and confident to apply some of these sophisticated techniques for building Advanced RAG systems!

Retrieval Augmented   Llm   Retrieval Generation   AI

Published in LlamaIndex Blog   Follow

2K followers · Last published Mar 4, 2024

Official Blog for LlamaIndex

Written by Andrei   Follow

670 followers · 26 following

Founding software/machine-learning engineer at LlamaIndex.
(https://ca.linkedin.com/in/nerdai)

Responses (7)

Write a response

What are your thoughts?

Doug Creates
Jan 8, 2024

Just awesome. 10.8k billion papers per day summarized into one page.

👏 11   Reply

Artsiom Baranouski
Jan 7, 2024

I don't understand 90% of that, but some points are good. I will save cheat sheet and will study more to understand 70% next year

👏 8   Reply

李涛
Jan 9, 2024

Hello, I saw the picture you drew and it's really great. I would like to know what software you used to draw it. Looking forward to your reply.

👏 5   💬 1 reply   Reply

See all responses

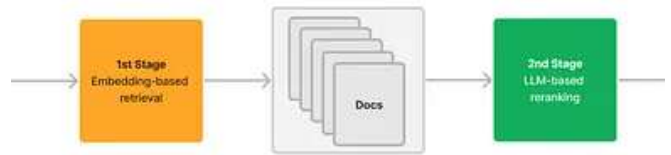# More from Andrei and LlamaIndex Blog
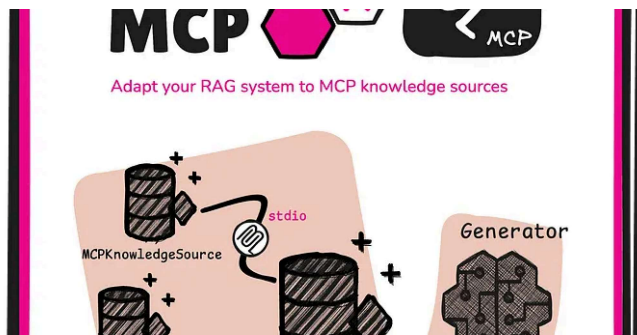
DSC In Data Science Collective by Andrei

NoEncode RAG with MCP—A new FedRAG core abstraction
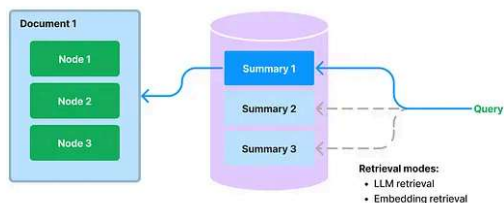
Hello, again!

Jun 2   👋 3

In LlamaIndex Blog by Jerry Liu

Using LLM's for Retrieval and Reranking

Summary

May 17, 2023   👋 341   💬 6

In LlamaIndex Blog by Jerry Liu

A New Document Summary Index for LLM-powered QA Systems

DSC In Data Science Collective by Andrei

Goodnews MCP—good news at your fingertips

In this blog post, we introduce a brand new   Building my first MCP server
LlamaIndex data structure: a Document...

May 8, 2023   👏 464   💬 10   Mar 21   👏 74
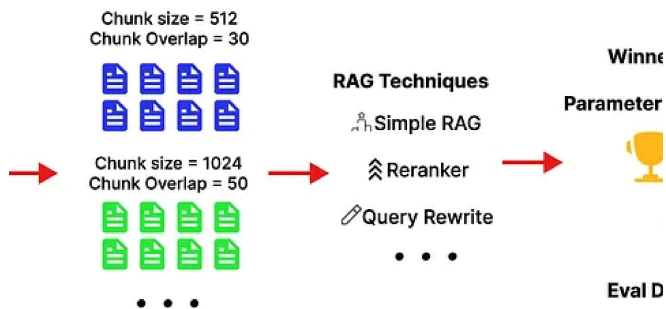
See all from Andrei   See all from LlamaIndex Blog

Recommended from Medium

🔵 In Level Up Coding by Fareed Khan   🔷 In AI Simplified in Plain English by kishore

Creating the Best RAG Finder   **Building an Uncensored Multi-**
Pipeline for Your Dataset   **Agent AI Debate System locally...**

Step by Step Implementation   What if AI agents could debate like real
                                        humans—structured, timed, and even...

✦   Apr 30   👏 434   💬 4   ✦   Jun 16   👏 52
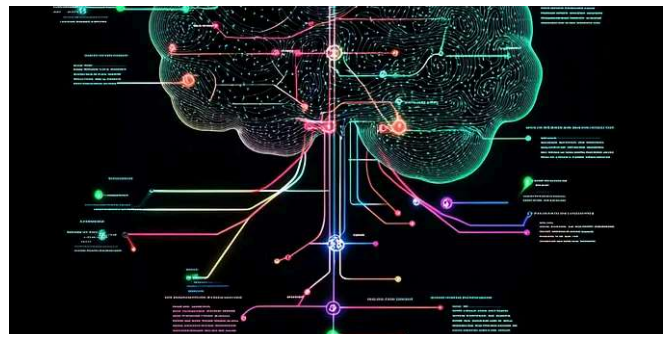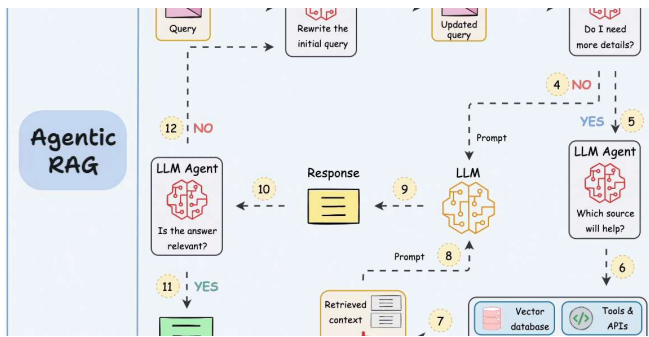
AI In Artificial Intelligence in Plain ... by Piyush Agni...

Building Agentic RAG with LangGraph: Mastering Adaptive...

Build intelligent RAG systems that know when to retrieve documents, search the web, or...

✦ Jul 20   👏 1.93K   💬 28

Tuhin Sharma

Understanding Adaptive-RAG: Smarter, Faster, and More Efficien...

Large Language Models (LLMs) like ChatGPT and GPT-4 are powerful tools, but they have ...

Mar 30   👏 7

S Shankar Angadi

AI : Agent vs LLM vs RAG

The artificial intelligence landscape is rapidly evolving, with three distinct architectural...

✦ Sep 1   👏 257   💬 4

DSC In Data Science Collective by Wei Yi

Reinforcement Learning from Human Feedback (RLHF) — Model...

This article introduces the four models used in RLHF, how they are setup, their loss and...

✦ May 26   👏 288   💬 2

[See more recommendations]