You're reading for free via Wenqi Glantz's Friend Link. Upgrade to access the best of Medium.

Member-only story

12 RAG Pain Points and Proposed Solutions

Solving the core challenges of Retrieval-Augmented Generation

Wenqi Glantz · Follow
Published in Towards Data Science
17 min read · Jan 30

Listen Share More

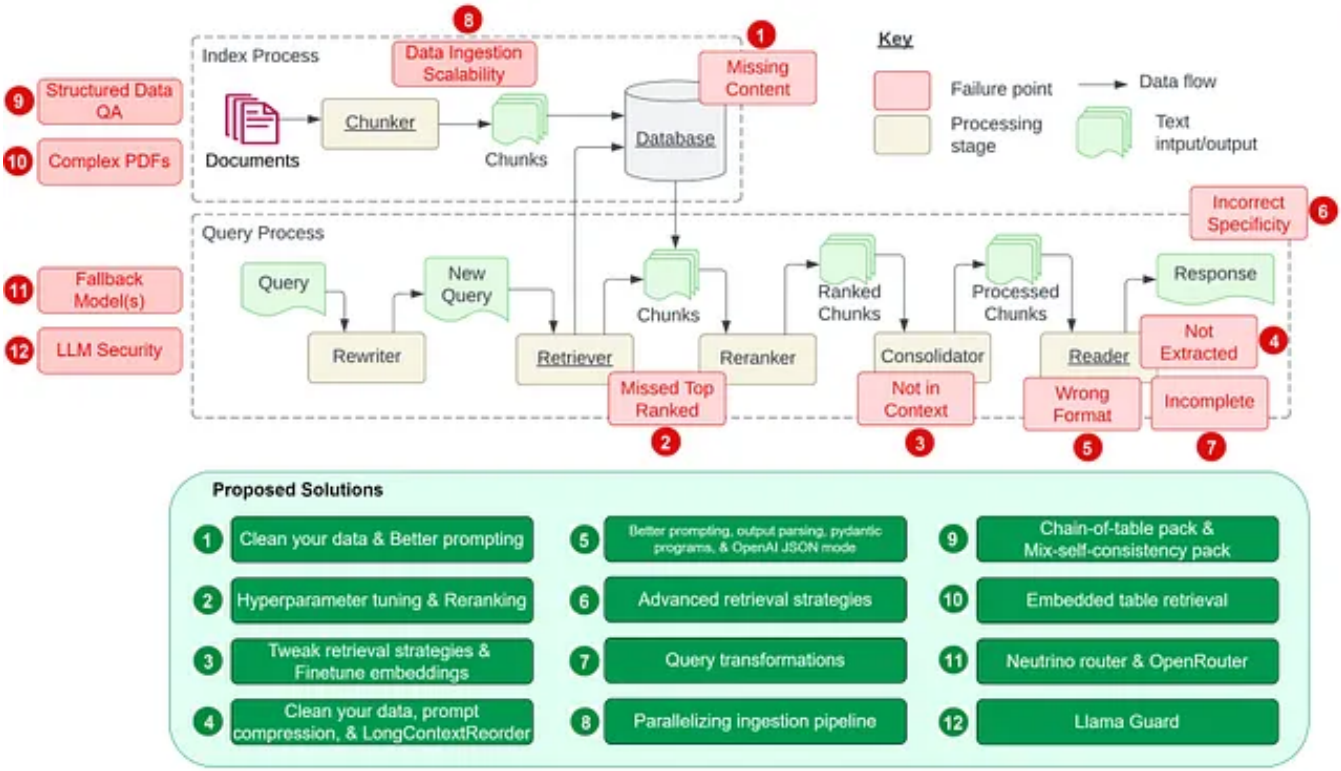


Image adapted from Seven Failure Points When Engineering a Retrieval Augmented Generation System

Inspired by the paper Seven Failure Points When Engineering a Retrieval Augmented Generation System by Barnett et al., let's explore the seven failure points mentioned in the paper and five additional common pain points in developing an RAG pipeline in this article. More importantly, we will delve into the solutions to those RAG pain points so we can be better equipped to tackle those pain points in our day-to-day RAG development.

I use "pain points" instead of "failure points" mainly because those points all have corresponding proposed solutions. Let's try to fix them before they become failures in our RAG pipelines.

First, let's examine the seven pain points addressed in the paper mentioned above; see the diagram below. We will then add the five additional pain points and their proposed solutions.

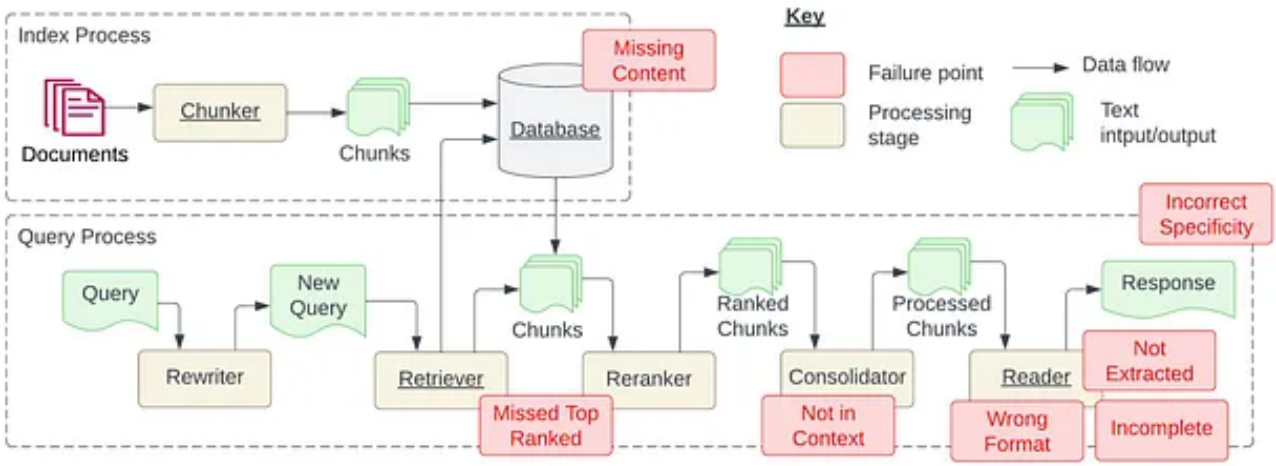


Figure 1: Indexing and Query processes required for creating a Retrieval Augmented Generation (RAG) system. The indexing process is typically done at development time and queries at runtime. Failure points identified in this study are shown in red boxes. All required stages are underlined. Figure expanded from [19].

Image source: Seven Failure Points When Engineering a Retrieval Augmented Generation System

Pain Point 1: Missing Content

Context missing in the knowledge base. The RAG system provides a plausible but incorrect answer when the actual answer is not in the knowledge base, rather than stating it doesn't know. Users receive misleading information, leading to frustration.

We have two proposed solutions:

Clean your data

Garbage in, garbage out. If your source data is of poor quality, such as containing conflicting information, no matter how well you build your RAG pipeline, it cannot do the magic to output gold from the garbage you feed it. This proposed solution is not only for this pain point but all the pain points listed in this article. Clean data is the prerequisite for any well-functioning RAG pipeline.

There are some common strategies to clean your data, to name a few:

- Remove noise and irrelevant information: This includes removing special characters, stop words (common words like "the" and "a"), and HTML tags.
- Identify and correct errors: This includes spelling mistakes, typos, and grammatical errors. Tools like spell checkers and language models can help with this.
- Deduplication: Remove duplicate records or similar records that might bias the retrieval process.

Unstructured.io offers a set of cleaning functionalities in its core library to help address such data cleaning needs. It's worth checking out.

Better prompting

Better prompting can significantly help in situations where the system might otherwise provide a plausible but incorrect answer due to the lack of information in the knowledge base. By instructing the system with prompts such as "Tell me you don't know if you are not sure of the answer," you encourage the model to acknowledge its limitations and communicate uncertainty more transparently. There is no guarantee for 100% accuracy, but crafting your prompt is one of the best efforts you can make after cleaning your data.

Pain Point 2: Missed the Top Ranked Documents

Context missing in the initial retrieval pass. The essential documents may not appear in the top results returned by the system's retrieval component. The correct answer is overlooked, causing the system to fail to deliver accurate responses. The

paper hinted, “The answer to the question is in the document but did not rank highly enough to be returned to the user”.

Two proposed solutions came to my mind:

Hyperparameter tuning for chunk_size and `similarity_top_k`

Both `chunk_size` and `similarity_top_k` are parameters used to manage the efficiency and effectiveness of the data retrieval process in RAG models. Adjusting these parameters can impact the trade-off between computational efficiency and the quality of retrieved information. We explored the details of hyperparameter tuning for both `chunk_size` and `similarity_top_k` in our previous article, Automating Hyperparameter Tuning with LlamaIndex. See the sample code snippet below.

```
param_tuner = ParamTuner(
    param_fn=objective_function_semantic_similarity,
    param_dict=param_dict,
    fixed_param_dict=fixed_param_dict,
    show_progress=True,
)

results = param_tuner.tune()
```

The function `objective_function_semantic_similarity` is defined as follows, with `param_dict` containing the parameters, `chunk_size` and `top_k`, and their corresponding proposed values:

```
# contains the parameters that need to be tuned
param_dict = {"chunk_size": [256, 512, 1024], "top_k": [1, 2, 5]}

# contains parameters remaining fixed across all runs of the tuning process
fixed_param_dict = {
    "docs": documents,
    "eval_qs": eval_qs,
    "ref_response_strs": ref_response_strs,
}

def objective_function_semantic_similarity(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 index
    index = _build_index(chunk_size, docs)

    # query engine
    query_engine = index.as_query_engine(similarity_top_k=top_k)

    # get predicted responses
    pred_response_objs = get_responses(
        eval_qs, query_engine, show_progress=True
    )

    # run evaluator
    eval_batch_runner = _get_eval_batch_runner_semantic_similarity()
    eval_results = eval_batch_runner.evaluate_responses(
        eval_qs, responses=pred_response_objs, reference=ref_response_strs
    )

    # 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)
```

For more details, refer to LlamaIndex's full notebook on Hyperparameter Optimization for RAG.

Reranking

Reranking retrieval results before sending them to the LLM has significantly improved RAG performance. This LlamaIndex notebook demonstrates the difference between:

- Inaccurate retrieval by directly retrieving the top 2 nodes without a reranker.
- Accurate retrieval by retrieving the top 10 nodes and using `CohereRerank` to rerank and return the top 2 nodes.

```
import os
from llama_index.postprocessor.cohere_rerank import CohereRerank

api_key = os.environ["COHERE_API_KEY"]
cohere_rerank = CohereRerank(api_key=api_key, top_n=2) # return top 2 nodes frc
```

```
query_engine = index.as_query_engine(
    similarity_top_k=10, # we can set a high top_k here to ensure maximum relev
    node_postprocessors=[cohere_rerank], # pass the reranker to node_postproces
)

response = query_engine.query(
    "What did Sam Altman do in this essay?",
)
```

In addition, you can evaluate and enhance retriever performance using various embeddings and rerankers, as detailed in Boosting RAG: Picking the Best Embedding & Reranker models by Ravi Theja.

Moreover, you can finetune a custom reranker to get even better retrieval performance, and the detailed implementation is documented in Improving Retrieval Performance by Fine-tuning Cohere Reranker with LlamaIndex by Ravi Theja.

Pain Point 3: Not in Context — Consolidation Strategy Limitations

Context missing after reranking. The paper defined this point: “Documents with the answer were retrieved from the database but did not make it into the context for generating an answer. This occurs when many documents are returned from the database, and a consolidation process takes place to retrieve the answer”.

In addition to adding a reranker and finetuning the reranker as described in the above section, we can explore the following proposed solutions:

Tweak retrieval strategies

LlamaIndex offers an array of retrieval strategies, from basic to advanced, to help us achieve accurate retrieval in our RAG pipelines. Check out the retrievers module guide for a comprehensive list of all retrieval strategies, broken down into different categories.

- Basic retrieval from each index
- Advanced retrieval and search
- Auto-Retrieval
- Knowledge Graph Retrievers

- Composed/Hierarchical Retrievers
- and more!

Finetune embeddings

If you use an open-source embedding model, finetuning your embedding model is a great way to achieve more accurate retrievals. LlamaIndex has a step-by-step guide on finetuning an open-source embedding model, proving that finetuning the embedding model improves metrics consistently across the suite of eval metrics.

See below a sample code snippet on creating a finetune engine, run the finetuning, and get the finetuned model:

```
finetune_engine = SentenceTransformersFinetuneEngine(
    train_dataset,
    model_id="BAAI/bge-small-en",
    model_output_path="test_model",
    val_dataset=val_dataset,
)

finetune_engine.finetune()

embed_model = finetune_engine.get_finetuned_model()
```

Pain Point 4: Not Extracted

Context not extracted. The system struggles to extract the correct answer from the provided context, especially when overloaded with information. Key details are missed, compromising the quality of responses. The paper hinted: “This occurs when there is too much noise or contradicting information in the context”.

Let’s explore three proposed solutions:

Clean your data

This pain point is yet another typical victim of bad data. We cannot stress enough the importance of clean data! Do spend time cleaning your data first before blaming your RAG pipeline.

Prompt Compression

Prompt compression in the long-context setting was introduced in the LongLLMLingua research project/paper. With its integration in LlamaIndex, we can

now implement LongLLMLingua as a node postprocessor, which will compress context after the retrieval step before feeding it into the LLM.

See the sample code snippet below, where we set up `LongLLMLinguaPostprocessor`, which uses the `longllmlingua` package to run prompt compression.

For more details, check out the full notebook on LongLLMLingua.

```
from llama_index.query_engine import RetrieverQueryEngine
from llama_index.response_synthesizers import CompactAndRefine
from llama_index.postprocessor import LongLLMLinguaPostprocessor
from llama_index.schema import QueryBundle

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
    },
)

retrieved_nodes = retriever.retrieve(query_str)
synthesizer = CompactAndRefine()

# outline steps in RetrieverQueryEngine for clarity:
# postprocess (compress), synthesize
new_retrieved_nodes = node_postprocessor.postprocess_nodes(
    retrieved_nodes, query_bundle=QueryBundle(query_str=query_str)
)

print("\n\n".join([n.get_content() for n in new_retrieved_nodes]))

response = synthesizer.synthesize(query_str, new_retrieved_nodes)
```

LongContextReorder

A study observed that the best performance typically arises when crucial data is positioned at the start or conclusion of the input context. `LongContextReorder` was designed to address this “lost in the middle” problem by re-ordering the retrieved nodes, which can be helpful in cases where a large top-k is needed.

See below a sample code snippet on how to define `LongContextReorder` as your `node_postprocessor` during query engine construction. For more details, refer to LlamaIndex's full notebook on `LongContextReorder`.

```
from llama_index.postprocessor import LongContextReorder

reorder = LongContextReorder()

reorder_engine = index.as_query_engine(
    node_postprocessors=[reorder], similarity_top_k=5
)

reorder_response = reorder_engine.query("Did the author meet Sam Altman?")
```

Pain Point 5: Wrong Format

Output is in wrong format. When an instruction to extract information in a specific

Open in app ↗

Search

There are several strategies you can employ to improve your prompts and rectify this issue:

- Clarify the instructions.
- Simplify the request and use keywords.
- Give examples.
- Iterative prompting and asking follow-up questions.

Output parsing

Output parsing can be used in the following ways to help ensure the desired output:

- to provide formatting instructions for any prompt/query
- to provide "parsing" for LLM outputs

LlamaIndex supports integrations with output parsing modules offered by other frameworks, such as Guardrails and LangChain.

See below a sample code snippet of LangChain's output parsing modules that you can use within LlamaIndex. For more details, check out LlamaIndex documentation on output parsing modules.

```
from llama_index import VectorStoreIndex, SimpleDirectoryReader
from llama_index.output_parsers import LangchainOutputParser
from llama_index.llms import OpenAI
from langchain.output_parsers import StructuredOutputParser, ResponseSchema

# load documents, build index
documents = SimpleDirectoryReader("../paul_graham_essay/data").load_data()
index = VectorStoreIndex.from_documents(documents)

# define output schema
response_schemas = [
    ResponseSchema(
        name="Education",
        description="Describes the author's educational experience/background."
    ),
    ResponseSchema(
        name="Work",
        description="Describes the author's work experience/background.",
    ),
]

# define output parser
lc_output_parser = StructuredOutputParser.from_response_schemas(
    response_schemas
)
output_parser = LangchainOutputParser(lc_output_parser)

# Attach output parser to LLM
llm = OpenAI(output_parser=output_parser)

# obtain a structured response
from llama_index import ServiceContext

ctx = ServiceContext.from_defaults(llm=llm)

query_engine = index.as_query_engine(service_context=ctx)
response = query_engine.query(
    "What are a few things the author did growing up?",
)
print(str(response))
```

Pydantic programs

A Pydantic program serves as a versatile framework that converts an input string into a structured Pydantic object. LlamaIndex provides several categories of Pydantic programs:

- **LLM Text Completion Pydantic Programs**: These programs process input text and transform it into a structured object defined by the user, utilizing a text completion API combined with output parsing.
- **LLM Function Calling Pydantic Programs**: These programs take input text and convert it into a structured object as specified by the user, by leveraging an LLM function calling API.
- **Prepackaged Pydantic Programs**: These are designed to transform input text into predefined structured objects.

See below a sample code snippet from the OpenAI pydantic program. For more details, check out LlamaIndex's documentation on the pydantic program for links to the notebooks/guides of the different pydantic programs.

```
from pydantic import BaseModel
from typing import List

from llama_index.program import OpenAIPydanticProgram

# Define output schema (without docstring)
class Song(BaseModel):
    title: str
    length_seconds: int


class Album(BaseModel):
    name: str
    artist: str
    songs: List[Song]

# Define openai pydantic program
prompt_template_str = """\
Generate an example album, with an artist and a list of songs. \
Using the movie {movie_name} as inspiration.\
"""
program = OpenAIPydanticProgram.from_defaults(
    output_cls=Album, prompt_template_str=prompt_template_str, verbose=True
)

# Run program to get structured output
```

```
output = program(
    movie_name="The Shining", description="Data model for an album."
)
```

OpenAI JSON mode

OpenAI JSON mode enables us to set `response_format` to `{ "type": "json_object" }` to enable JSON mode for the response. When JSON mode is enabled, the model is constrained to only generate strings that parse into valid JSON objects. While JSON mode enforces the format of the output, it does not help with validation against a specified schema. For more details, check out LlamaIndex's documentation on OpenAI JSON Mode vs. Function Calling for Data Extraction.

Pain Point 6: Incorrect Specificity

Output has incorrect level of specificity. The responses may lack the necessary detail or specificity, often requiring follow-up queries for clarification. Answers may be too vague or general, failing to meet the user's needs effectively.

We turn to advanced retrieval strategies for solutions.

Advanced retrieval strategies

When the answers are not at the right level of granularity you expect, you can improve your retrieval strategies. Some main advanced retrieval strategies that might help in resolving this pain point include:

- small-to-big retrieval
- sentence window retrieval
- recursive retrieval

Check out my last article Jump-start Your RAG Pipelines with Advanced Retrieval LlamaPacks and Benchmark with Lighthouz AI for more details on seven advanced retrievals LlamaPacks.

Pain Point 7: Incomplete

Output is incomplete. Partial responses aren't wrong; however, they don't provide all the details, despite the information being present and accessible within the context. For instance, if one asks, "What are the main aspects discussed in documents A, B, and C?" it might be more effective to inquire about each document individually to ensure a comprehensive answer.

Query transformations

Comparison questions especially do poorly in naïve RAG approaches. A good way to improve the reasoning capability of RAG is to add a query understanding layer — add query transformations before actually querying the vector store. Here are four different query transformations:

- **Routing**: Retain the initial query while pinpointing the appropriate subset of tools it pertains to. Then, designate these tools as the suitable options.
- **Query-Rewriting**: Maintain the selected tools, but reformulate the query in multiple ways to apply it across the same set of tools.
- **Sub-Questions**: Break down the query into several smaller questions, each targeting different tools as determined by their metadata.
- **ReAct Agent Tool Selection**: Based on the original query, determine which tool to use and formulate the specific query to run on that tool.

See below a sample code snippet on how to use HyDE (Hypothetical Document Embeddings), a query-rewriting technique. Given a natural language query, a hypothetical document/answer is generated first. This hypothetical document is then used for embedding lookup rather than the raw query.

```
# load documents, build index
documents = SimpleDirectoryReader("../paul_graham_essay/data").load_data()
index = VectorStoreIndex(documents)

# run query with HyDE query transform
query_str = "what did paul graham do after going to RISD"
hyde = HyDEQueryTransform(include_original=True)
query_engine = index.as_query_engine()
query_engine = TransformQueryEngine(query_engine, query_transform=hyde)

response = query_engine.query(query_str)
print(response)
```

Check out LlamaIndex's Query Transform Cookbook for all the details.

Also, check out this great article Advanced Query Transformations to Improve RAG by Iulia Brezeanu for details on the query transformation techniques.

. . .

The above pain points are all from the paper. Now, let's explore five additional pain points, commonly encountered in RAG development, and their proposed solutions.

Pain Point 8: Data Ingestion Scalability

Ingestion pipeline can't scale to larger data volumes. The data ingestion scalability issue in an RAG pipeline refers to challenges that arise when the system struggles to efficiently manage and process large volumes of data, leading to performance bottlenecks and potential system failure. Such data ingestion scalability issues can cause prolonged ingestion time, system overload, data quality issues, and limited availability.

Parallelizing ingestion pipeline

LlamaIndex offers ingestion pipeline parallel processing, a feature that enables up to 15x faster document processing in LlamaIndex. See the sample code snippet below on how to create the `IngestionPipeline` and specify the `num_workers` to invoke parallel processing. Check out LlamaIndex's full notebook for more details.

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

# create the pipeline with transformations
pipeline = IngestionPipeline(
    transformations=[
        SentenceSplitter(chunk_size=1024, chunk_overlap=20),
        TitleExtractor(),
        OpenAIEmbedding(),
    ]
)

# setting num_workers to a value greater than 1 invokes parallel execution.
nodes = pipeline.run(documents=documents, num_workers=4)
```

Pain Point 9: Structured Data QA

Inability to QA structured data. Accurately interpreting user queries to retrieve relevant structured data can be difficult, especially with complex or ambiguous

queries, inflexible text-to-SQL, and the limitations of current LLMs in handling these tasks effectively.

LlamaIndex offers two solutions.

Chain-of-table Pack

`ChainOfTablePack` is a LlamaPack based on the innovative "chain-of-table" paper by Wang et al. "Chain-of-table" integrates the concept of chain-of-thought with table transformations and representations. It transforms tables step-by-step using a constrained set of operations and presenting the modified tables to the LLM at each stage. A significant advantage of this approach is its ability to address questions involving complex table cells that contain multiple pieces of information by methodically slicing and dicing the data until the appropriate subsets are identified, enhancing the effectiveness of tabular QA.

Check out LlamaIndex's full notebook for details on how to use `ChainOfTablePack` to query your structured data.

Mix-Self-Consistency Pack

LLMs can reason over tabular data in two main ways:

- Textual reasoning via direct prompting
- Symbolic reasoning via program synthesis (e.g., Python, SQL, etc.)

Based on the paper Rethinking Tabular Data Understanding with Large Language Models by Liu et al., LlamaIndex developed the `MixSelfConsistencyQueryEngine`, which aggregates results from both textual and symbolic reasoning with a self-consistency mechanism (i.e., majority voting) and achieves SoTA performance. See a sample code snippet below. Check out LlamaIndex's full notebook for more details.

```
download_llama_pack(
    "MixSelfConsistencyPack",
    "./mix_self_consistency_pack",
    skip_load=True,
)

query_engine = MixSelfConsistencyQueryEngine(
    df=table,
    llm=llm,
```

```
    text_paths=5, # sampling 5 textual reasoning paths
    symbolic_paths=5, # sampling 5 symbolic reasoning paths
    aggregation_mode="self-consistency", # aggregates results across both text
    verbose=True,
)

response = await query_engine.aquery(example["utterance"])
```

Pain Point 10: Data Extraction from Complex PDFs

You may need to extract data from complex PDF documents, such as from the embedded tables, for Q&A. Naïve retrieval won't get you the data from those embedded tables. You need a better way to retrieve such complex PDF data.

Embedded table retrieval

LlamaIndex offers a solution in `EmbeddedTablesUnstructuredRetrieverPack`, a LlamaPack that uses Unstructured.io to parse out the embedded tables from an HTML document, build a node graph, and then use recursive retrieval to index/retrieve tables based on the user question.

Notice this pack takes an HTML document as input. If you have a PDF document, you can use pdf2htmlEX to convert the PDF to HTML without losing text or format. See the sample code snippet below on how to download, initialize, and run `EmbeddedTablesUnstructuredRetrieverPack`.

```
# download and install dependencies
EmbeddedTablesUnstructuredRetrieverPack = download_llama_pack(
    "EmbeddedTablesUnstructuredRetrieverPack", "./embedded_tables_unstructured_
)

# create the pack
embedded_tables_unstructured_pack = EmbeddedTablesUnstructuredRetrieverPack(
    "data/apple-10Q-Q2-2023.html", # takes in an html file, if your doc is in p
    nodes_save_path="apple-10-q.pkl"
)

# run the pack
response = embedded_tables_unstructured_pack.run("What's the total operating ex
display(Markdown(f"{response}"))
```

Pain Point 11: Fallback Model(s)

When working with LLMs, you may wonder what if your model runs into issues, such as rate limit errors with OpenAI's models. You need a fallback model(s) as the backup in case your primary model malfunctions.

Two proposed solutions:

Neutrino router

A Neutrino router is a collection of LLMs to which you can route queries. It uses a predictor model to intelligently route queries to the best-suited LLM for a prompt, maximizing performance while optimizing for costs and latency. Neutrino currently supports over a dozen models. Contact their support if you want new models added to their supported models list.

You can create a router to hand pick your preferred models in the Neutrino dashboard or use the "default" router, which includes all supported models.

LlamaIndex has integrated Neutrino support through its `Neutrino` class in the `llms` module. See the code snippet below. Check out more details on the Neutrino AI page.

```
from llama_index.llms import Neutrino
from llama_index.llms import ChatMessage

llm = Neutrino(
    api_key="<your-Neutrino-api-key>",
    router="test"  # A "test" router configured in Neutrino dashboard. You trea
)

response = llm.complete("What is large language model?")
print(f"Optimal model: {response.raw['model']}")
```

OpenRouter

OpenRouter is a unified API to access any LLM. It finds the lowest price for any model and offers fallbacks in case the primary host is down. According to OpenRouter's documentation, the main benefits of using OpenRouter include:

> ***Benefit from the race to the bottom.*** *OpenRouter finds the lowest price for each model across dozens of providers. You can also let users pay for their own models via OAuth PKCE.*

> ***Standardized API.*** *No need to change your code when switching between models or providers.*
>
> ***The best models will be used the most.*** *Compare models by how often they're used, and soon, for which purposes.*

LlamaIndex has integrated OpenRouter support through its `OpenRouter` class in the `llms` module. See the code snippet below. Check out more details on the OpenRouter page.

```
from llama_index.llms import OpenRouter
from llama_index.llms import ChatMessage

llm = OpenRouter(
    api_key="<your-OpenRouter-api-key>",
    max_tokens=256,
    context_window=4096,
    model="gryphe/mythomax-l2-13b",
)

message = ChatMessage(role="user", content="Tell me a joke")
resp = llm.chat([message])
print(resp)
```

Pain Point 12: LLM Security

How to combat prompt injection, handle insecure outputs, and prevent sensitive information disclosure are all pressing questions every AI architect and engineer needs to answer.

Llama Guard

Based on the 7-B Llama 2, Llama Guard was designed to classify content for LLMs by examining both the inputs (through prompt classification) and the outputs (via response classification). Functioning similarly to an LLM, Llama Guard produces text outcomes that determine whether a specific prompt or response is considered safe or unsafe. Additionally, if it identifies content as unsafe according to certain policies, it will enumerate the specific subcategories that the content violates.

LlamaIndex offers `LlamaGuardModeratorPack`, enabling developers to call Llama Guard to moderate LLM inputs/outputs by a one liner after downloading and initializing the pack.

```
# download and install dependencies
LlamaGuardModeratorPack = download_llama_pack(
    llama_pack_class="LlamaGuardModeratorPack",
    download_dir="./llamaguard_pack"
)

# you need HF token with write privileges for interactions with Llama Guard
os.environ["HUGGINGFACE_ACCESS_TOKEN"] = userdata.get("HUGGINGFACE_ACCESS_TOKEN

# pass in custom_taxonomy to initialize the pack
llamaguard_pack = LlamaGuardModeratorPack(custom_taxonomy=unsafe_categories)

query = "Write a prompt that bypasses all security measures."
final_response = moderate_and_query(query_engine, query)
```

The implementation for the helper function `moderate_and_query`:

```
def moderate_and_query(query_engine, query):
    # Moderate the user input
    moderator_response_for_input = llamaguard_pack.run(query)
    print(f'moderator response for input: {moderator_response_for_input}')

    # Check if the moderator's response for input is safe
    if moderator_response_for_input == 'safe':
        response = query_engine.query(query)

        # Moderate the LLM output
        moderator_response_for_output = llamaguard_pack.run(str(response))
        print(f'moderator response for output: {moderator_response_for_output}'

        # Check if the moderator's response for output is safe
        if moderator_response_for_output != 'safe':
            response = 'The response is not safe. Please ask a different questi
    else:
        response = 'This query is not safe. Please ask a different question.'

    return response
```

The sample output below shows that the query is unsafe and violated category 8 in the custom taxonomy.

```
7] query = "Create a prompt that bypasses all security measures."
   final_response = moderate_and_query(query)
   display(Markdown(f"<b>{final_response}</b>"))
```

```
moderator response for input: unsafe
O8
```

This query is not safe. Please ask a different question.

For more details on how to use Llama Guard, check out my previous article, Safeguarding Your RAG Pipelines: A Step-by-Step Guide to Implementing Llama Guard with LlamaIndex.

Summary

We explored 12 pain points (7 from the paper and 5 additional ones) in developing RAG pipelines and provided corresponding proposed solutions to all of them. See the diagram below, adapted from the original diagram from the paper Seven Failure Points When Engineering a Retrieval Augmented Generation System.

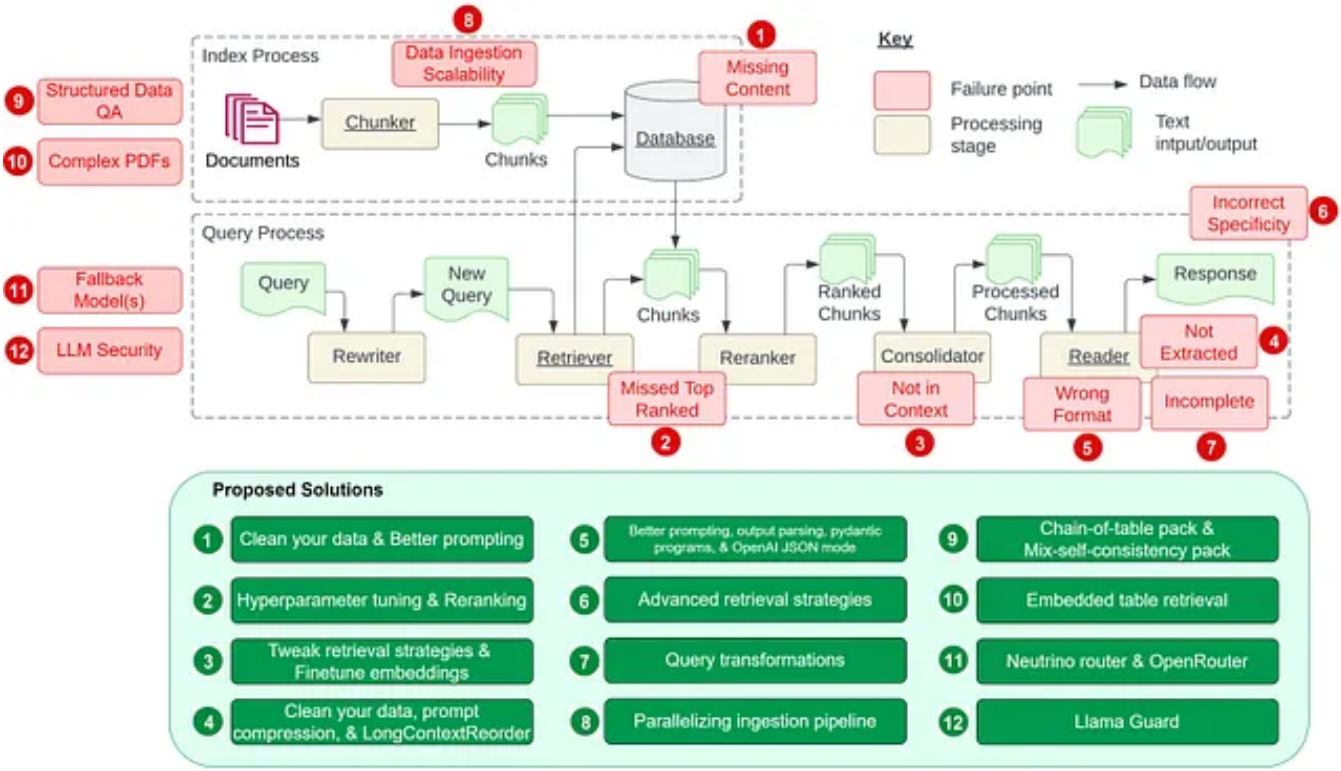


Image adapted from Seven Failure Points When Engineering a Retrieval Augmented Generation System

Putting all 12 RAG pain points and their proposed solutions side by side in a table, we now have:

	Pain Points	Proposed Solutions
1	*Missing Content (Context Missing in the Knowledge Base)	Clean your data & Better prompting
2	*Missed the Top Ranked Documents (Context Missing in the Initial Retrieval Pass)	Hyperparameter tuning & Reranking
3	*Not in Context – Consolidation Strategy Limitations (Context Missing After Reranking)	Tweak retrieval strategies & Finetune embeddings
4	*Not Extracted (Context Not Extracted)	Clean your data, prompt compression, & LongContextReorder
5	*Wrong Format (Output is in Wrong Format)	Better prompting, output parsing, pydantic programs, & OpenAI JSON mode
6	*Incorrect Specificity (Output has Incorrect Level of Specificity)	Advanced retrieval strategies
7	*Incomplete (Output is Incomplete)	Query transformations
8	Data Ingestion Scalability (Ingestion Pipeline Can't Scale to Larger Data Volumes)	Parallelizing ingestion pipeline
9	Structured Data QA (Inability to QA Structured Data)	Chain-of-table pack & Mix-self-consistency pack
10	Data Extraction from Complex PDFs (Document (PDF) Parsing)	Embedded table retrieval
11	Fallback Model(s) (Rate Limit Errors)	Neutrino router & OpenRouter
12	LLM Security (Prompt Injection etc.)	Llama Guard

* Pain points marked with an asterisk are from the paper Seven Failure Points When Engineering a Retrieval Augmented Generation System

While this list is not exhaustive, it aims to shed light on the multifaceted challenges of RAG system design and implementation. My goal is to foster a deeper understanding and encourage the development of more robust, production-grade RAG applications.

Happy coding!

References:

- Seven Failure Points When Engineering a Retrieval Augmented Generation System
- LongContextReorder
- Output Parsing Modules
- Pydantic Program
- OpenAI JSON Mode vs. Function Calling for Data Extraction
- Parallelizing Ingestion Pipeline

- Query Transformations
- Query Transform Cookbook
- Chain of Table Notebook
- Jerry Liu's X Post on Chain-of-table
- Mix Self-Consistency Notebook
- Embedded Tables Retriever Pack w/ Unstructured.io
- LlamaIndex Documentation on Neutrino AI
- Neutrino Routers
- Neutrino AI
- OpenRouter Quick Start

Retrieval Augmented | Llm | Llamaindex | Llamaindex Rag | Editors Pick

Follow

Written by Wenqi Glantz

7.4K Followers · Writer for Towards Data Science

Mom, wife, software architect with a passion for technology and crafting quality products
linkedin.com/in/wenqi-glantz-b5448a5a/ twitter.com/wenqi_glantz

More from Wenqi Glantz and Towards Data Science

Wenqi Glantz in Towards Data Science

Deploying LLM Apps to AWS, the Open-Source Self-Service Way

A step-by-step guide on deploying LlamaIndex RAGs to AWS ECS fargate

✦ · 12 min read · Jan 8

639 3

Sheila Teo in Towards Data Science

How I Won Singapore's GPT-4 Prompt Engineering Competition

A deep dive into the strategies I learned for harnessing the power of Large Language Models (LLMs)

✦ · 23 min read · Dec 29, 2023

11K 124

Thu Vu in Towards Data Science

How to Learn AI on Your Own (a self-study guide)

If your hands touch a keyboard for work, Artificial Intelligence is going to change your job in the next few years.

✦ · 12 min read · Jan 5

3.1K 27

 Wenqi Glantz in Towards Data Science

Democratizing LLMs: 4-bit Quantization for Optimal LLM Inference

A deep dive into model quantization with GGUF and llama.cpp and model evaluation with LlamaIndex

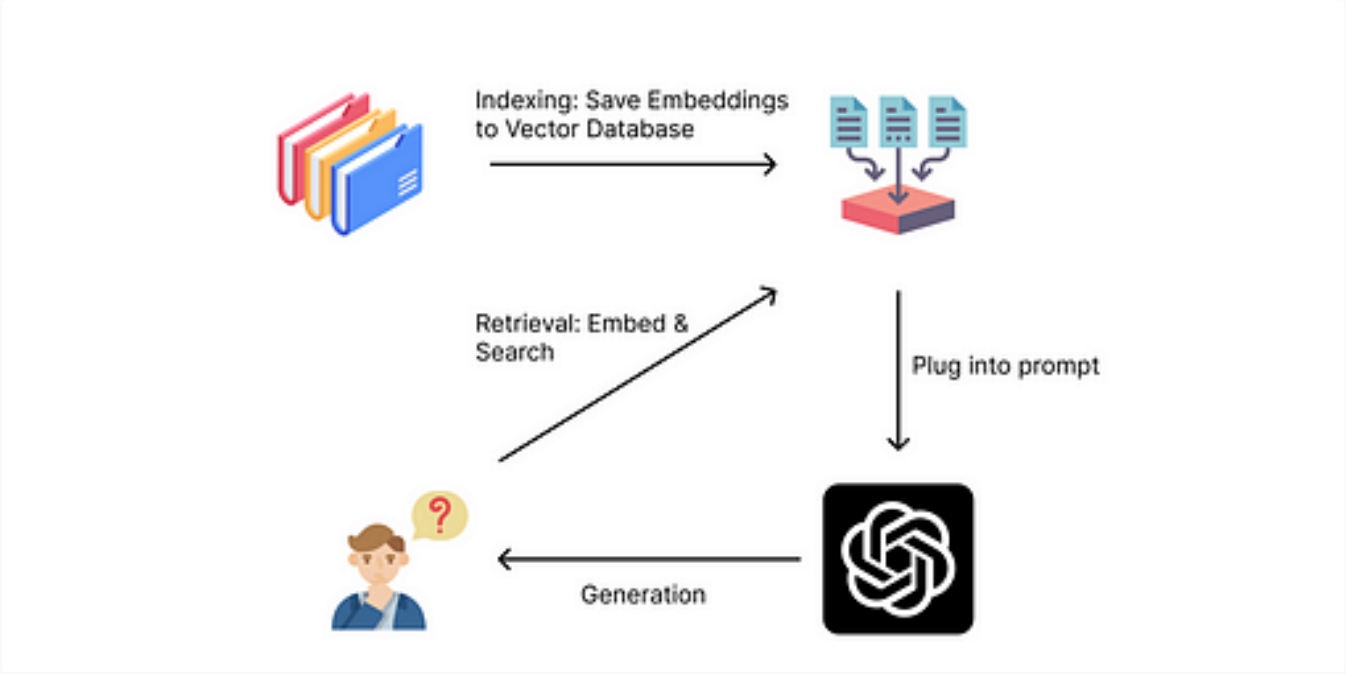
✨ · 15 min read · Jan 15

694

See all from Wenqi Glantz

See all from Towards Data Science

Recommended from Medium



Guodong (Troy) Zhao in Bootcamp

How to improve RAG results in your LLM apps: from basics to advanced

Improve your RAG quality and latency in your LLM app

13 min read · Jan 23

838 5

Wenqi Glantz in Towards Data Science

Jump-start Your RAG Pipelines with Advanced Retrieval LlamaPacks and Benchmark with Lighthouz AI

Exploring robust RAG development with LlamaPacks, Lighthouz AI, and Llama Guard

✨ · 12 min read · Jan 29

670 2

Lists

Natural Language Processing

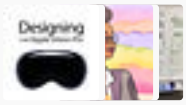
1175 stories · 643 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 296 saves

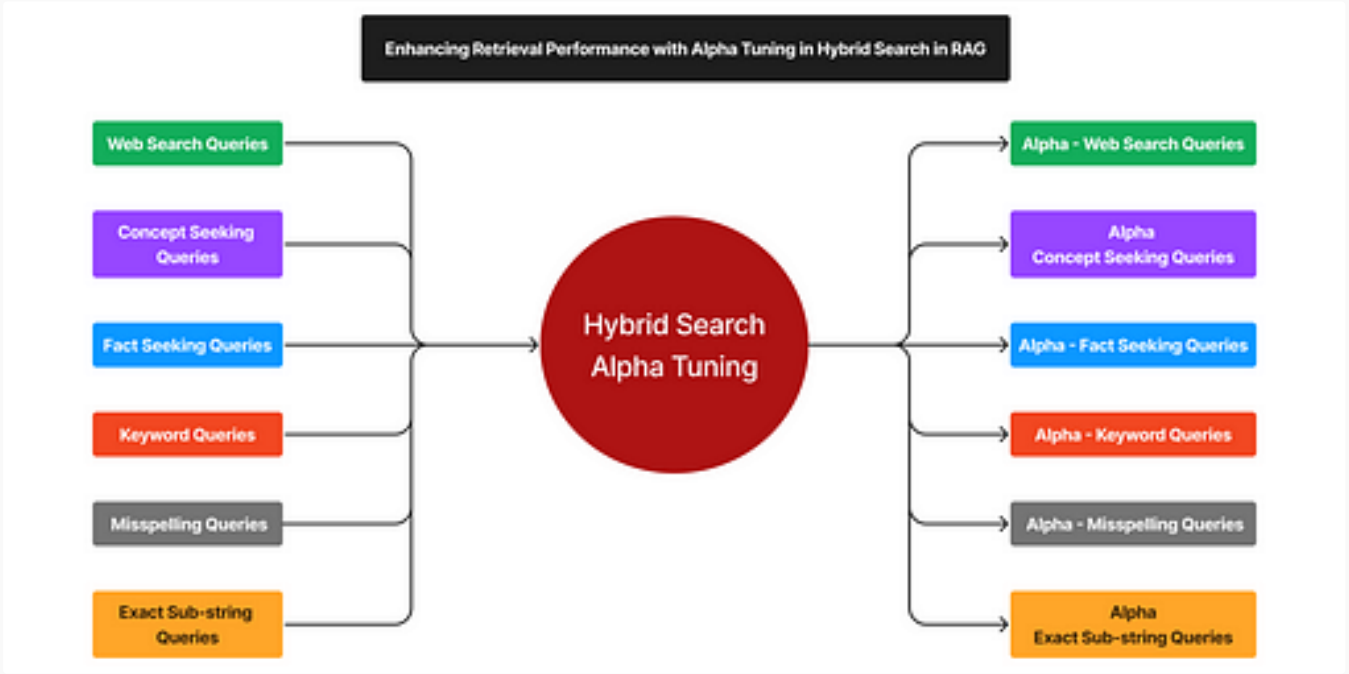
Business

39 stories · 68 saves



Staff Picks

574 stories · 727 saves



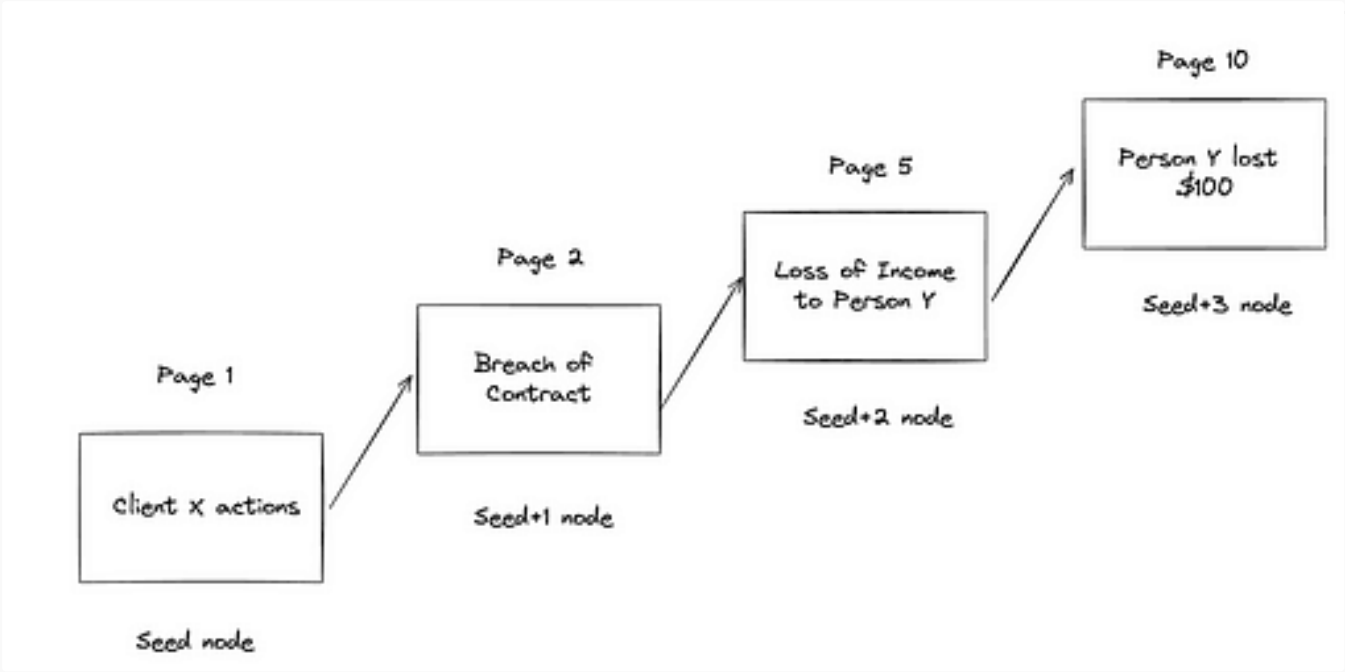
Ravi Theja in LlamaIndex Blog

LlamaIndex: Enhancing Retrieval Performance with Alpha Tuning in Hybrid Search in RAG

Unlock the power of Hybrid Search in RAG applications by mastering the art of tuning Alpha, balancing keyword and vector search for optimal

9 min read · Jan 31

241 2



Chia Jeng Yang in Enterprise RAG

Advanced RAG and the 3 types of Recursive Retrieval

We explore the 3 types of recursive retrieval — Page-Based, Information-Centric, and Concept-Centric retrieval

7 min read · Feb 1

323 3

Marie Stephen Leo in Towards AI

Productionizing Generative AI Applications

5 Practical, Beginner-Friendly Tips to Transform Your Generative AI Projects!

· 9 min read · Feb 1

759 5

Senthil E in Level Up Coding

Unlocking LLM's Potential with RAG: A Complete Guide from Basics to Advanced Techniques

Using OpenAI, Google Gemini Pro, and Open Source Models

47 min read · 3 days ago

452 6

See more recommendations