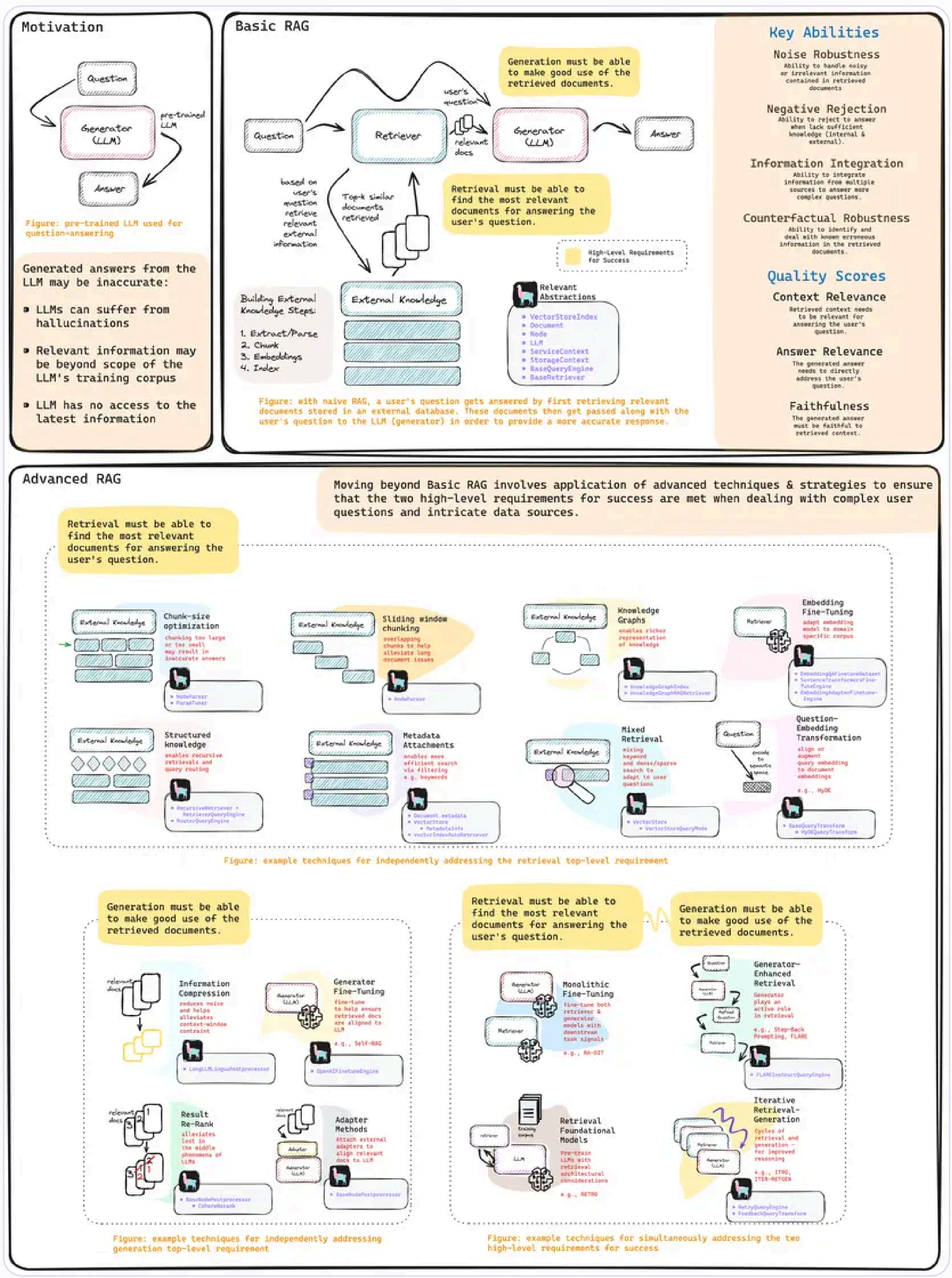

Motivation
Question
Generator (LLM)
pre-trained LLM
Answer
Figure: pre-trained LLM used for question-answering
Generated answers from the LLM may be inaccurate:
LLMs can suffer from hallucinations
Relevant information may be beyond scope of the LLM's training corpus
LLM has no access to the latest information
Basic RAG
Question
Retriever
user's question
relevant docs
Generator (LLM)
Answer
Generation must be able to make good use of the retrieved documents.
based on user's question retrieve relevant external information
Top-k similar documents retrieved
Retrieval must be able to find the most relevant documents for answering the user's question.
High-Level Requirements for Success
Building External Knowledge Steps:
1. Extract/Parse
2. Chunk
3. Embeddings
4. Index
External Knowledge
Relevant Abstractions
VectorStoreIndex
Document
Node
LLM
ServiceContext
StorageContext
BaseQueryEngine
BaseRetriever
Figure: with naive RAG, a user's question gets answered by first retrieving relevant documents stored in an external database. These documents then get passed along with the user's question to the LLM (generator) in order to provide a more accurate response.
Key Abilities
Noise Robustness
Ability to handle noisy or irrelevant information contained in retrieved documents
Negative Rejection
Ability to reject to answer when lack sufficient knowledge (internal & external).
Information Integration
Ability to integrate information from multiple sources to answer more complex questions.
Counterfactual Robustness
Ability to identify and deal with known erroneous information in the retrieved documents.
Quality Scores
Context Relevance
Retrieved context needs to be relevant for answering the user's question.
Answer Relevance
The generated answer needs to directly address the user's question.
Faithfulness
The generated answer must be faithful to retrieved context.
Advanced RAG
Moving beyond Basic RAG involves application of advanced techniques & strategies to ensure that the two high-level requirements for success are met when dealing with complex user questions and intricate data sources.
Retrieval must be able to find the most relevant documents for answering the user's question.
External Knowledge
Chunk-size optimization
chunking too large or too small may result in inaccurate answers
NodeParser
ParamTuner
Sliding window chunking
overlapping chunks to help alleviate long document issues
NodeParser
Knowledge Graphs
enables richer representation of knowledge
KnowledgeGraphIndex
KnowledgeGraphRAGRetriever
Retriever
Embedding Fine-Tuning
adapt embedding model to domain specific corpus
EmbeddingQAFinetuneDataset
SentenceTransformersFineTuneEngine
EmbeddingAdapterFinetuneEngine
Structured knowledge
enables recursive retrievals and query routing
RecursiveRetriever + RetrieverQueryEngine
RouterQueryEngine
Metadata Attachments
enables more efficient search via filtering e.g. keywords
Document metadata
VectorStore
MetadataInfo
VectorIndexAutoRetriever
Mixed Retrieval
mixing keyword and dense/sparse search to adapt to user questions
VectorStore
VectorStoreQueryMode
Question
encode to semantic space
Question-Embedding Transformation
align or augment query embedding to document embeddings
e.g., HyDE
BaseQueryTransform
HyDEQueryTransform
Figure: example techniques for independently addressing the retrieval top-level requirement
Generation must be able to make good use of the retrieved documents.
relevant docs
Information Compression
reduces noise and helps alleviates context-window contraint
LongLLMLinguaPostprocessor
Generator (LLM)
Generator Fine-Tuning
fine-tune to help ensure retrieved docs are aligned to LLM
e.g., Self-RAG
OpenAIFinetuneEngine
relevant docs
Result Re-Rank
alleviates lost in the middle phenomena of LLMs
BaseNodePostprocessor
CohereRerank
relevant docs
Adapter
Generator (LLM)
Adapter Methods
Attach external adapters to align relevant docs to LLM
BaseNodePostprocessor
Figure: example techniques for independently addressing generation top-level requirement
Retrieval must be able to find the most relevant documents for answering the user's question.
Generation must be able to make good use of the retrieved documents.
Generator (LLM)
Retriever
Monolithic Fine-Tuning
fine-tune both retriever & generator models with downstream task signals
e.g., RA-DIT
Question
Generator (LLM)
Refined Question
Retriever
Generator-Enhanced Retrieval
Generator plays an active role in retrieval
e.g., Step-Back Prompting, FLARE
FLAREInstructQueryEngine
retriever
training corpus
LLM
Retrieval Foundational Models
Pre-train LLMs with retrieval architectural considerations
e.g., RETRO
Retriever
Generator (LLM)
Iterative Retrieval-Generation
Cycles of retrieval and generation - for improved reasoning
e.g., ITRG, ITER-RETGEN
RetryQueryEngine
FeedbackQueryTransform
Figure: example techniques for simultaneously addressing the two high-level requirements for success

Andrei • Jan 5, 2024

A Cheat Sheet and Some Recipes For Building Advanced RAG

Retrieval Augmented   LLM   Retrieval Generation   AI

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):

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

### Recipe
### Perform hyperparameter tuning as in traditional ML via grid-search
### 1. Define an objective function that ranks different parameter comb
### 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
    query_engine = index.as_query_engine(similarity_top_k=top_k)

    # perform inference with RAG pipeline on a provided questions `eval
    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=Tr
    )
```

```
        eval_results = eval_batch_runner.evaluate_responses(
            eval_qs, responses=pred_response_objs, reference=ref_response_s
        )

        # 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
    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)**:**

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

```
### 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
]
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 s
        ]
        all_nodes.extend(sub_inodes)
    # also add original node to node
    original_node = IndexNode.from_text_node(base_node, base_node.node_
    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

# 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

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 (notebook guide)**:**

```
    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 questic
    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]
)

# 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 (notebook guide)**:**

```
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 (notebook guide):

```
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 (notebook guide):

```
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
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

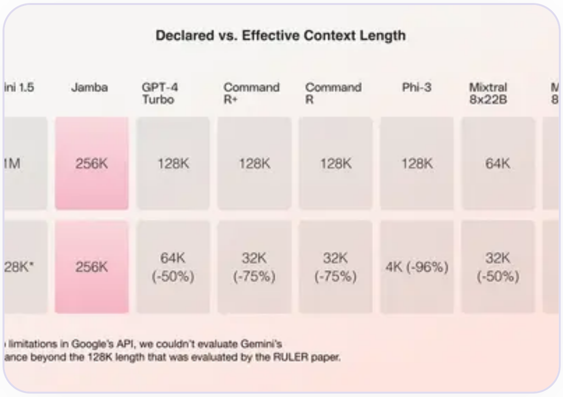
Evaluating RAG systems are, of course, of utmost importance. In their survey paper, Gao, Yunfan et al. indicate 7 measurement aspects as seen in the top-right portion of the attached RAG cheat sheet. The llama-index library consists of several evaluation abstractions as well as integrations to RAGAs 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!

RELATED ARTICLES



Declared vs. Effective Context Length

ini 1.5	Jamba	GPT-4 Turbo	Command R+	Command R	Phi-3	Mixtral 8x22B	M 8
1M	256K	128K	128K	128K	128K	64K	
28K*	256K	64K (-50%)	32K (-75%)	32K (-75%)	4K (-96%)	32K (-50%)	

limitations in Google's API, we couldn't evaluate Gemini's
ance beyond the 128K length that was evaluated by the RULER paper.

Jamba-Instruct's 256k context window on LlamaIndex

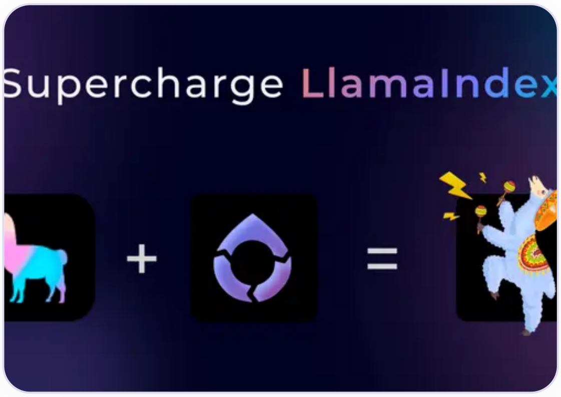
2024-07-31

LlamaIndex Newsletter 2024-04-02

2024-04-02

LlamaIndex Newsletter 2024-03-26

2024-03-26



Supercharge your LlamaIndex RAG Pipeline with UpTrain Evaluations

2024-03-19

LlamaIndex