Nanonets

WORKFLOW AUTOMATION

A complete Guide to LlamaIndex in 2024

by **Karan Kalra** 33 MIN READ

Published: Oct 18, 2023 • Updated: Mar 27, 2024

Table of Contents

Nanonets

Welcome to our guide of LlamaIndex!

In simple terms, LlamaIndex is a handy tool that acts as a bridge between your custom data and large language models (LLMs) like GPT-4 which are powerful models capable of understanding human-like text. Whether you have data stored in APIs, databases, or in PDFs, LlamaIndex makes it easy to bring that data into conversation with these smart machines. This bridge-

Nanonets

way for creating powerful custom LLM applications and workflows.

While discussing LlamaIndex's power to bridge data and LLMs, it's worth noting how Nanonets takes this further with Workflows Automation. Imagine seamlessly integrating such smart capabilities into your daily operations, where AI assists not only in querying data but in automating repetitive tasks across all your apps and databases. With Nanonets, you can create and enhance workflows within minutes, adding AI-driven insights and even human validation where needed, making your systems more intelligent and efficient. Learn more about revolutionizing your workflows.

Learn More

Nanonets

LlamaIndex

Initially known as GPT Index, LlamaIndex has evolved into an indispensable ally for developers. It's like a multi-tool that helps in various stages of working with data and large language models -

- Firstly, it helps in 'ingesting' data, which means getting the data from its original source into the system.
- Secondly, it helps in 'structuring' that data, which means organizing it in a way that the language models can easily understand.
- Thirdly, it aids in 'retrieval', which means finding and fetching the right pieces of data when needed.
- Lastly, it simplifies 'integration', making it easier to meld your data with various application frameworks.

When we dive a little deeper into the mechanics of LlamaIndex, we find three main heroes doing the heavy lifting.

Nanonets

data from wherever it resides, be it APIs, PDFs, databases, or external apps like Gmail, Notion, Airtable.

2. The 'data indexes' are the organized librarians, arranging your data neatly so that it's easily accessible.
3. And the 'engines' are the translators (LLMs), making it possible to interact with your data using natural language and ultimately create applications and workflows.

In the next sections, we'll explore how to set up LlamaIndex and start using it to supercharge your applications with the power of large language models.

What's what in LlamaIndex

LlamaIndex is your go-to platform for creating robust applications powered by Large Language Models (LLMs) over your customized data. Be it a sophisticated Q&A system, an interactive chatbot, or intelligent agents, LlamaIndex lays down the foundation for your ventures into the realm of **Retrieval Augmented Generation (RAG).**

Nanonets

your custom data. Components of an RAG application or workflow -

- **Knowledge Base (Input)**: The knowledge base is like a library filled with useful information such as FAQs, manuals, and other relevant documents. When a question is asked, this is where the system looks to find the answer.
- **Trigger/Query (Input)**: This is the spark that gets things going. Typically, it's a question or request from a customer that signals the system to spring into action.
- **Task/Action (Output)**: After understanding the trigger or query, the system then performs a certain task to address it. For instance, if it's a question, the system will work on providing an answer, or if it's a request for a specific action, it will carry out that action accordingly.

Based on the context of our blog, we will need to implement the following two stages using Llamaindex to provide the two inputs to our RAG mechanism -

Nanonets

2. **Querying Stage:** Harnessing the knowledge base & the LLM to respond to your queries by generating the final output / performing the final task.

Let's take a closer look at these stages under the magnifying lens of LlamaIndex.

The Indexing Stage: Crafting the Knowledge Base

LlamaIndex equips you with a suite of tools to shape your knowledge base:

- **Data Connectors:** These entities, also known as Readers, ingest data from diverse sources and formats into a unified Document representation.
- **Documents / Nodes:** A Document is your container for data, whether it springs from a PDF, an API, or a database. A Node, on the other hand, is a snippet of a Document, enriched with metadata and relationships, paving the way for precise retrieval operations.
- **Data Indexes:** Post ingestion, LlamaIndex assists in arranging

parsing, embedding, and metadata inference, and ultimately results in the creation of the knowledge base.

The Querying Stage: Engaging with Your Knowledge

In this phase, we fetch relevant context from the knowledge base as per your query, and blend it with the LLM's insights to generate a response or perform a task. This not only provides the LLM with updated relevant knowledge but also prevents hallucination. The core challenge here orbits around retrieval, orchestration, and reasoning across multiple knowledge bases.

LlamaIndex offers modular constructs to help you use it for Q&A, chatbots, or agent-driven applications.

These are the primary elements -

- **Query Engines:** These are your end-to-end conduits for querying your data, taking a natural language query and

- **Chat Engines:** They elevate the interaction to a conversational level, allowing back-and-forths with your data.
- **Agents:** Agents are your automated decision-makers, interacting with the world through a toolkit, and manoeuvring through tasks with a dynamic action plan rather than a fixed logic.

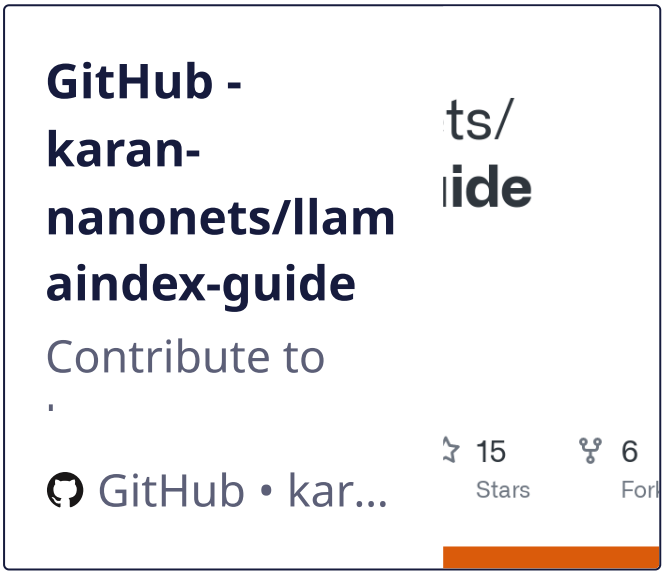
These are few common building blocks of the primary elements present in all of the elements discussed above -

- **Retrievers:** They dictate the technique of fetching relevant context from the knowledge base against a query. For example, Dense Retrieval against a vector index is a prevalent approach.
- **Node Postprocessors:** They refine the set of nodes through transformation, filtering, or re-ranking.
- **Response Synthesizers:** They channel the LLM to generate responses, blending the user query with retrieved text chunks.

Nanonets

about the above elements.

All the code examples discussed and the associated sample files used in the blog are present in this github repository.



Automate manual tasks and workflows with our AI-driven workflow builder, designed by Nanonets for you and your teams.

Installation and Setup

Before exploring the exciting features, let's first install LlamaIndex on your system. If you're familiar with Python, this will be easy. Use this command to install:

```
pip install llama-index
```

Then follow either of the two approaches below -

- By default, LlamaIndex uses OpenAI's gpt-3.5-turbo for creating text and text-embedding-ada-002 for fetching and embedding. You need an OpenAI API Key to use these. Get your API key for free by signing up on OpenAI's website. Then set your environment variable with the name OPENAI_API_KEY in your python file.

```
os.environ["OPENAI_API_KEY"]
```

- If you'd rather not use OpenAI, the system will switch to using LlamaCPP and llama2-chat-13B for creating text and BAAI/bge-small-en for fetching and embedding. These all work offline. To set up LlamaCPP, follow its setup guide here. This will need about 11.5GB of memory on both your CPU and GPU. Then, to use local embedding, install this:

```
pip install sentence-transformers
```

Creating Llamaindex Documents

Data connectors, also referred to as Readers, are essential components in LlamaIndex that facilitate the ingestion of data from various sources and formats, converting them into a simplified Document representation consisting of text and basic metadata.

Nanonets

which can be seamlessly integrated into any LlamaIndex application. All the connectors present here can be used as follows -

```
from llama_index import downl

GoogleDocsReader = download_l
loader = GoogleDocsReader()
documents = loader.load_data(
```

For example, the above loader loads data from your Google Docs into Llamaindex Documents.

See the full list of data connectors here -

The variety of data connectors here is pretty exhaustive, some of which include:

- **SimpleDirectoryReader**: Supports a broad range of file types (.pdf, .jpg, .png, .docx, etc.) from a local file directory.

- **SlackReader**: Imports data from Slack.
- **AirtableReader**: Imports data from Airtable.
- **ApifyActor**: Capable of web crawling, scraping, text extraction, and file downloading.

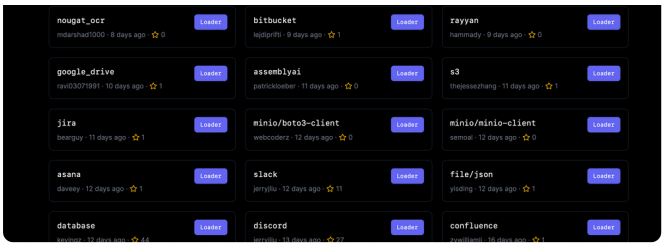
How to find the right data connector?

- First look up and check if a relevant data connector is listed in Llamaindex documentation here -

  Module Guides - LlamaIndex 🦙 0.8.45.post1

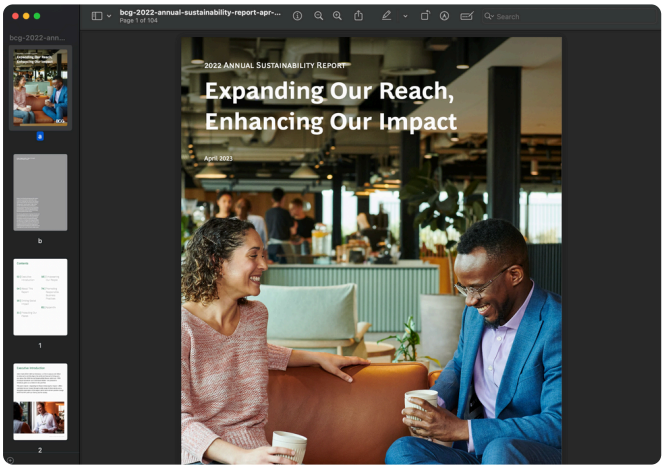
  LlamaIndex 🦙 0.8.45....

- **If not, then identify the relevant data connector on Llamahub**



For example, let us try this on a couple of data sources.

1. PDF File : We use the **SimpleDirectoryReader** data connector for this. The given example below loads a BCG Annual Sustainability Report.

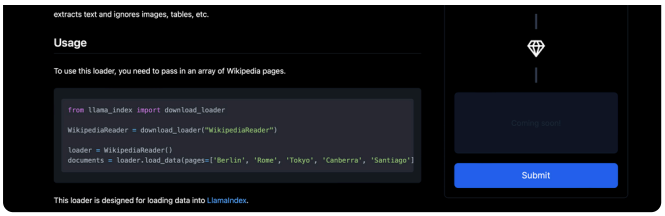



```
[6]: from llama_index import SimpleDirectoryReader

reader = SimpleDirectoryReader(
    input_files=["bcg-2022-annual-sustainability-report-apr-2023.pdf"]
)

pdf_documents = reader.load_data()
```

2. Wikipedia Page : We search Llamahub and find a relevant connector for this.



The given example below loads the wikipedia pages about a few countries from around the globe. Basically, the the top page that appears in the search results with each element of the list as a search query is ingested.

```
: from llama_index import download_loader

  WikipediaReader = download_loader("WikipediaReader")

  loader = WikipediaReader()
  wikipedia_documents = loader.load_data(pages=['Iceland Country', 'Kenya Country', 'Cambodia Country'])
```

Nanonets

Creating LlamaIndex Nodes

In LlamaIndex, once the data has been ingested and represented as Documents, there's an option to further process these Documents into Nodes. Nodes are more granular data entities that represent "chunks" of source Documents, which could be text chunks, images, or other types of data. They also carry metadata and relationship information with other nodes, which can be instrumental in building a more structured and relational index.

Basic

To parse Documents into Nodes, LlamaIndex provides NodeParser classes. These classes help in automatically transforming the content of Documents into Nodes, adhering to a specific structure that can be utilized further in index construction and querying.

Documents into Nodes.

```
from llama_index.node_parser

# Assuming documents have alr

# Initialize the parser
parser = SimpleNodeParser.fro

# Parse documents into nodes
nodes = parser.get_nodes_from
```

In this snippet, `SimpleNodeParser.from_defaults()` initializes a parser with default settings, and `get_nodes_from_documents(documents)` is used to parse the loaded Documents into Nodes.

Advanced

Various customization options include:

- `text_splitter` (default: TokenTextSplitter)
- `include_metadata` (default: True)
- `include_prev_next_rel` (default: True)

Text Splitter Customization

Customize text splitter, using either `SentenceSplitter`, `TokenTextSplitter`, or `CodeSplitter` from `llama_index.text_splitter`.

Examples:

SentenceSplitter:

```
import tiktoken
from llama_index.text_splitte

text_splitter = SentenceSplit
  separator=" ", chunk_size=1
  paragraph_separator="\n\n\n
  tokenizer=tiktoken.encoding
)

node_parser = SimpleNodeParse
```

TokenTextSplitter:

```
import tiktoken
from llama_index.text_splitte

text_splitter = TokenTextSpli
  separator=" ", chunk_size=1
  backup_separators=["\n"],
  tokenizer=tiktoken.encoding
)
```

Nanonets

CodeSplitter:

```
from llama_index.text_splitte

text_splitter = CodeSplitter(
  language="python", chunk_li
)

node_parser = SimpleNodeParse
```

SentenceWindowNodeParser

For specific scope embeddings, utilize `SentenceWindowNodeParser` to split documents into individual sentences, also capturing surrounding sentence windows.

```
import nltk
from llama_index.node_parser

node_parser = SentenceWindowN
  window_size=3, window_metad
)
```

Manual Node Creation

For more control, manually create Node objects and define attributes and relationships:

```
# Create TextNode objects
node1 = TextNode(text="<text_
node2 = TextNode(text="<text_

# Define node relationships
node1.relationships[NodeRelat
node2.relationships[NodeRelat

# Gather nodes
nodes = [node1, node2]
```

In this snippet, `TextNode` creates nodes with text content while `NodeRelationship` and `RelatedNodeInfo` define node relationships.

Let us create basic nodes for the PDF and Wikipedia page documents we have created.

```
from llama_index.node_parser import SimpleNodeParser

parser = SimpleNodeParser.from_defaults(chunk_size=1024, chunk_overlap=20)

pdf_nodes = parser.get_nodes_from_documents(pdf_documents)
wikipedia_nodes = parser.get_nodes_from_documents(wikipedia_documents)
```

Creating LlamaIndex Index

The core essence of LlamaIndex lies in its ability to build structured indices over ingested data,

Nanonets

efficient querying over the data. Let's delve into how to build indices with both Document and Node objects, and what happens under the hood during this process.

- **Building Index from Documents**

Here's how you can build an index directly from Documents using the VectorStoreIndex:

```
from llama_index import Vecto

# Assuming docs is your list
index = VectorStoreIndex.from
```

Different types of indices in LlamaIndex handle data in distinct ways:

- **Summary Index**: Stores Nodes as a sequential chain, and during query time, all Nodes are loaded into the Response Synthesis module if no other query parameters are specified.
- **Vector Store Index**: Stores each Node and a corresponding embedding in a Vector Store,

- **Tree Index**: Builds a hierarchical tree from a set of Nodes, and queries involve traversing from root nodes down to leaf nodes.
- **Keyword Table Index**: Extracts keywords from each Node to build a mapping, and queries extract relevant keywords to fetch corresponding Nodes.

To choose your index, you should carefully evaluate the module guides here and make a choice here according to your use case.

Under the Hood:

1. The Documents are parsed into Node objects, which are lightweight abstractions over text strings that additionally keep track of metadata and relationships.
2. Index-specific computations are performed to add Node into the index data structure. For example:
    - For a vector store index, an embedding model is called (either via API or locally) to compute embeddings for the Node objects.

summary.

Let us create an index for the PDF File using the above code.

```
[8]: from llama_index import VectorStoreIndex
     pdf_index = VectorStoreIndex.from_documents(pdf_documents)
```

- **Building Index from Nodes**

You can also build an index directly from Node objects, following the parsing of Documents into Nodes or manual Node creation:

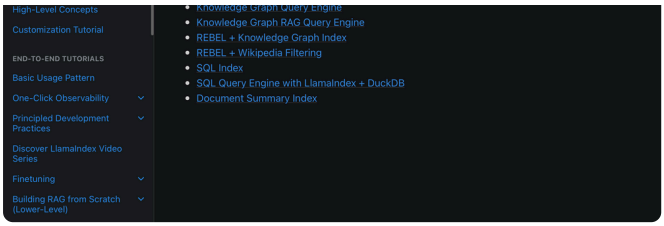
```
from llama_index import Vecto

# Assuming nodes is your list
index = VectorStoreIndex(node
```

Let us go ahead and create a VectorStoreIndex for the PDF nodes.

```
index = VectorStoreIndex(pdf_nodes)
```

Let us now create a summary index for the Wikipedia nodes. We find the relevant index from the list of supported indices, and settle on the Document Summary Index.



We create the index with some customization as follows -

```
[37]: from llama_index.llms import OpenAI
      from llama_index import ServiceContext, OpenAIEmbedding
      from llama_index.indices.document_summary import DocumentSummaryIndex
      from llama_index.response_synthesizers import ResponseMode, get_response_synthesizer

      chatgpt = OpenAI(temperature=0, model="gpt-3.5-turbo")
      service_context = ServiceContext.from_defaults(llm=chatgpt, chunk_size=1024)

      response_synthesizer = get_response_synthesizer(
          response_mode="tree_summarize", use_async=True
      )

      doc_summary_index = DocumentSummaryIndex(wikipedia_nodes)

      current doc id: 407fc8b0-7649-4468-99b1-55ad64b5eae2
      current doc id: d7406f9d-5f87-4356-b3b6-bf56526dd6fd
      current doc id: 0d4b7934-e7d4-4526-8234-cf64bb08fd93
```

Storing an Index

LlamaIndex's storage capability is built for adaptability, especially when dealing with evolving data sources. This section outlines the functionalities provided for managing data storage, including customization and persistence features.

Persistence (Basic)

There might be instances where you might want to save the index for future use, and LlamaIndex makes this straightforward. With the `persist()` method, you can store

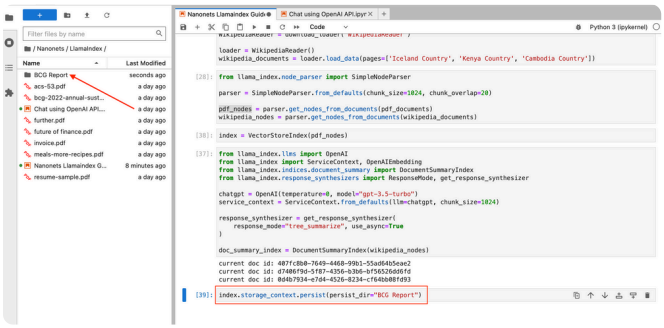
Nanonets

method, you can retrieve data effortlessly.

```
# Persisting to disk
index.storage_context.persist

# Loading from disk
from llama_index import Stora
storage_context = StorageCont
index = load_index_from_stora
```

For example, we can save the PDF index as follows -



Storage Components (Advanced)

At its core, LlamaIndex provides more customizable storage components enabling users to specify where various data elements are stored. These components include:

- **Document Stores:** The repositories for storing ingested

- **Index Stores:** The places where index metadata are kept.
- **Vector Stores:** The storages for holding embedding vectors.

LlamaIndex is versatile in its storage backend support, with confirmed support for:

- Local filesystem (as seen in the basic *persistence* example)
- AWS S3
- Cloudflare R2

These backends are facilitated through the use of the `fsspec` library, which allows for a variety of storage backends.

For many vector stores, both data and index embeddings are stored together, eliminating the need for separate document or index stores. This arrangement also auto-handles data persistence, simplifying the process of building new indexes or reloading existing ones.

LlamaIndex has integrations with various vector stores that handle the entire index (vectors + text). Check the guide here.

Nanonets

```
# build a new index
from llama_index import Vecto
from llama_index.vector_store

vector_store = DeepLakeVector
storage_context = StorageCont
index = VectorStoreIndex.from

# reload an existing index
index = VectorStoreIndex.from
```

To leverage storage abstractions, a `StorageContext` object needs to be defined, as shown below:

```
from llama_index.storage.docs
from llama_index.storage.inde
from llama_index.vector_store
from llama_index.storage impo

storage_context = StorageCont
    docstore=SimpleDocumentSt
    vector_store=SimpleVector
    index_store=SimpleIndexSt
)
```

Learn more about storage and using custom storage elements here.

Nanonets

Query Data

After having established a well-structured index using LlamaIndex, the next pivotal step is querying this index to extract meaningful insights or answers to specific inquiries. This segment elucidates the process and methods available for querying the data indexed in LlamaIndex.

Before diving into querying, ensure that you have a well-constructed index as discussed in the previous section. Your index could be built on documents or nodes, and could be a single index or composed of multiple indices.

High-Level Query API

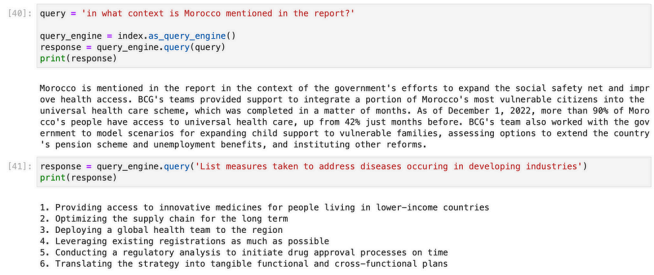
LlamaIndex provides a high-level API that facilitates straightforward querying, ideal for common use cases.

```
# Assuming 'index' is your co
query_engine = index.as_query
response = query_engine.query
print(response)
```

Nanonets

utilized to create a query engine from your index, and the `query()` method to execute a query.

We can try this out on our PDF index -



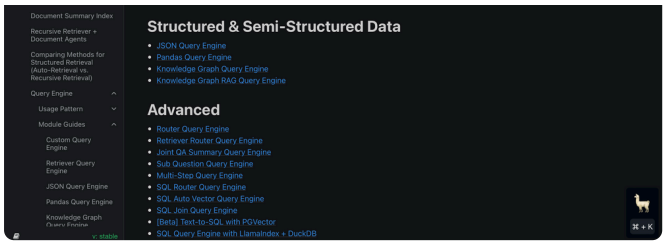
By default, index.as_query_engine() creates a query engine with the specified default settings in LlamaIndex.

You can choose your own query engine based on your use case from the list here and use that to query your index -

Module Guides - LlamaIndex 🦙 0.8.46

LlamaIndex 🦙 0.8.46

Nanonets



For example, let us use a **sub question query engine** to tackle the problem of answering a complex query using multiple data sources. It first breaks down the complex query into sub questions for each relevant data source, then gather all the intermediate reponses and synthesizes a final response. We will use it on our Wikipedia index.

We will follow the sub question query engine documentation.

Let us import required libraries and set context variable to ensure we can print the subtasks undertaken by the query engine instead of just printing the final response.

```
import nest_asyncio
from llama_index import
VectorStoreIndex,
SimpleDirectoryReader
from llama_index.tools import
QueryEngineTool, ToolMetadata
from llama_index.query_engine
```

```
LlamaDebugHandler
from llama_index import
ServiceContext

nest_asyncio.apply()

# We are using the
LlamaDebugHandler to print
the trace of the sub
questions captured by the
SUB_QUESTION callback event
type
llama_debug =
LlamaDebugHandler(print_trace
_on_end=True)
callback_manager =
CallbackManager([llama_debug]
)

service_context =
ServiceContext.from_defaults(

callback_manager=callback_man
ager
)
```

We first create a basic vector index as instructed by the documentation.

```
: vector_query_engine = VectorStoreIndex.from_documents(
    wikipedia_documents, use_async=True, service_context=service_context
).as_query_engine()
```

```
**********
Trace: index_construction
    |_node_parsing ->  0.985535 seconds
      |_chunking ->  0.499369 seconds
      |_chunking ->  0.000938 seconds
      |_chunking ->  0.481499 seconds
    |_embedding ->  2.677047 seconds
    |_embedding ->  2.659649 seconds
**********
```

```
vector_query_engine = VectorStoreIndex.from_documents(
    wikipedia_documents, use_async=True, service_context=service_context
).as_query_engine()
```

```
**********
Trace: index_construction
    |_node_parsing ->  0.985535 seconds
      |_chunking ->  0.499369 seconds
      |_chunking ->  0.000938 seconds
      |_chunking ->  0.481499 seconds
    |_embedding ->  2.677047 seconds
    |_embedding ->  2.659649 seconds
**********
```

Now, querying and asking for the response traces the subquestions that the query engine internally computed to get to the final response.

```
response = query_engine.query(
    "Give me all similaries between Iceland, Kenya and Cambodia"
)
```

```
Generated 3 sub questions.
[countries] Q: What are the similarities between Iceland and Kenya
[countries] Q: What are the similarities between Iceland and Cambodia
[countries] Q: What are the similarities between Kenya and Cambodia
[countries] A:
Both Iceland and Cambodia have a strong sense of community and value egalitarianism. Both countries have high levels of internet access and are highly
ranked in terms of press freedom. Additionally, both countries have a rich literary tradition, with Iceland having been designated a UNESCO City of Lit
erature and Cambodia having a long history of sacred verse and rhyming epic poems.
[countries] A:
The two countries have similar demographics, with a large percentage of the population being under the age of 22. Both countries have a rich cultural h
eritage, with traditional festivals and celebrations such as the Cambodian New Year and Pchum Ben. Both countries have a variety of traditional cuisine
s, with rice being the staple grain in both countries. Additionally, both countries have a growing number of microbreweries and a variety of beverages
such as tea and coffee. Finally, both countries have high potential for developing renewable energy resources.
[countries] A:
Both Iceland and Kenya have a strong sense of community and lack of social isolation, with high levels of social cohesion attributed to the small size
and homogeneity of the population. Both countries also place a great importance on independence and self-sufficiency, and have a strong work ethic. Add
itionally, both countries have high levels of gender equality, with Iceland consistently ranked among the top three countries in the world for women to
live in, and Kenya having made significant progress in recent years in terms of women's rights. Finally, both countries have a vibrant sports culture,
with popular sports including football, track and field, handball, basketball, and horseback riding.
**********
Trace: query
    |_query ->  17.167509 seconds
      |_llm ->  3.70331 seconds
      |_sub_question ->  5.950608 seconds
        |_query ->  5.950059 seconds
          |_retrieve ->  0.518952 seconds
            |_embedding ->  0.50633 seconds
          |_synthesize ->  5.430937 seconds
            |_llm ->  5.37056 seconds
      |_sub_question ->  3.511436 seconds
        |_query ->  3.510759 seconds
          |_retrieve ->  0.591948 seconds
            |_embedding ->  0.579563 seconds
          |_synthesize ->  2.918641 seconds
            |_llm ->  2.860292 seconds
      |_sub_question ->  4.614674 seconds
        |_query ->  4.614136 seconds
          |_retrieve ->  0.413743 seconds
            |_embedding ->  0.383834 seconds
          |_synthesize ->  4.200138 seconds
            |_llm ->  4.113889 seconds
      |_synthesize ->  7.509727 seconds
        |_llm ->  7.468624 seconds
**********
```

And the final response given by the engine can now be printed.

```
print(response)
```

```
All three countries have a strong sense of community and value egalitarianism. They all have high levels of internet access and are highly ranked in te
rms of press freedom. Additionally, all three countries have a rich literary tradition, with Iceland having been designated a UNESCO City of Literature
and Cambodia having a long history of sacred verse and rhyming epic poems. They all have similar demographics, with a large percentage of the populatio
n being under the age of 22. All three countries have a rich cultural heritage, with traditional festivals and celebrations such as the Cambodian New Y
ear and Pchum Ben. They all have a variety of traditional cuisines, with rice being the staple grain in all three countries. Additionally, all three co
untries have a growing number of microbreweries and a variety of beverages such as tea and coffee. Finally, all three countries have high potential for
developing renewable energy resources.
```

Automate manual tasks and workflows with our AI-driven

Nanonets

Request a Demo

Low-Level Composition API

For more granular control or advanced querying scenarios, the low-level composition API is available. This allows for customization at various stages of the query process.

Right at the start of this blog, we mentioned that there are three supplementary blocks within the query engine / chat engine / agent that can be configured while creating them -

Retrievers

They dictate the technique of fetching relevant context from the

Nanonets

vector index is a prevalent approach.

Choose the right retriever here -

> **Module Guides - LlamaIndex 🦙 0.8.46**
>
> LlamaIndex 🦙 0.8.46

Node Postprocessors

They refine the set of nodes through transformation, filtering, or re-ranking.

Summary of LlamaIndex Postprocessors:

1. **SimilarityPostprocessor:**
   - Removes nodes below a certain similarity score.
   - Set threshold using `similarity_cutoff`.
2. **KeywordNodePostprocessor:**
   - Filters nodes based on keyword inclusion or exclusion.
   - Use `required_keywords` and `exclude_keywords`.
3. **MetadataReplacementPostProcessor:**

- Works well with `SentenceWindowNodeParser`.

4. **LongContextReorder:**
    - Addresses models' difficulty with extended contexts. It reorders nodes, which benefits situations where a large number of top results are essential.
5. **SentenceEmbeddingOptimizer:**
    - Removes irrelevant sentences based on embeddings.
    - Choose either `percentile_cutoff` or `threshold_cutoff` for relevance.
6. **CohereRerank:**
    - Uses the Cohere ReRank to reorder nodes, giving back the top N results.
7. **SentenceTransformerRerank:**
    - Uses sentence-transformer cross-encoders to reorder nodes, yielding the top N nodes.
    - Various models available with different speed/accuracy trade-offs.
8. **LLMRerank:**

9. **FixedRecencyPostprocessor:**
    - Returns nodes sorted by date. Requires a date field in node metadata.
10. **EmbeddingRecencyPostprocessor:**
    - Ranks nodes by date, but also removes older similar nodes based on embedding similarity.
11. **TimeWeightedPostprocessor:**
    - Reranks nodes with a bias towards information not recently returned.
12. **PIINodePostprocessor (Beta):**
    - Removes personally identifiable information. Can utilize either a local LLM or a NER model.
13. **PrevNextNodePostprocessor (Beta):**
    - Based on node relationships, retrieves nodes that come before, after, or both in sequence.
14. **AutoPrevNextNodePostprocessor (Beta):**
    - Similar to the above, but lets the LLM decide the relationship direction.

Nanonets

for further instructions. Read them here.

Response Synthesizers

They channel the LLM to generate responses, blending the user query with retrieved text chunks.

Response synthesizers might sound fancy, but they're actually tools that help generate a reply or answer based on your question and some given text data. Let's break it down.

Imagine you have a bunch of pieces of text (like a pile of books). Now, you ask a question and want an answer based on those texts. The response synthesizer is like a librarian who goes through the texts, finds relevant information, and crafts a reply for you.

Think of the whole process in the query engine as a factory line:

1. First, a machine pulls out relevant text pieces based on your question. We have already discussed this. (**Retriever**)
2. Then, if needed, there's a step that might fine-tune these pieces. We have already

Nanonets

3. Finally, the response synthesizer takes these pieces and gives you a neatly crafted answer. (**Response Synthesizer**)

Response synthesizers come in various styles:

- **Refine:** This method goes through each text piece, refining the answer bit by bit.
- **Compact:** A shorter version of 'Refine.' It bunches the texts together, so there are fewer steps to refine.
- **Tree Summarize:** Imagine taking many small answers, combining them, and summarizing again until you have one main answer.
- **Simple Summarize:** Just cuts the text pieces to fit and gives a quick summary.
- **No Text:** This one doesn't give you an answer but tells you which text pieces it would have used.
- **Accumulate:** Think of this as getting a bunch of mini-answers for each text piece and then sticking them together.

Nanonets

If you're tech-savvy, you can even build your custom synthesizer. The primary job of any synthesizer is to take a question and some text pieces and give back a string of text as an answer.

Below is a basic structure that every response synthesizer should have. They should be able to take in a question and parts of text and then give back an answer.

```
class BaseSynthesizer(ABC):
    """Response builder class

    def __init__(
        self,
        service_context: Opti
        streaming: bool = Fal
    ) -> None:
        """Init params."""
        self._service_context
        self._callback_manage
        self._streaming = str

    @abstractmethod
    def get_response(
        self,
        query_str: str,
        text_chunks: Sequence
        **response_kwargs: An
    ) -> RESPONSE_TEXT_TYPE:
```

Nanonets

```
    @abstractmethod
    async def aget_response(
        self,
        query_str: str,
        text_chunks: Sequence
        **response_kwargs: An
    ) -> RESPONSE_TEXT_TYPE:
        """Get response async
        ...
```

Using a response synthesizer directly (without the other steps we did in previous sections) is also possible and can be as simple as:

```
from llama_index import get_r

# Set up the synthesizer
my_synthesizer = get_response

# Ask a question
response = my_synthesizer.syn
```

Using it in your index along with your configured retrievers and node postprocessors can be done as follows -

```
from llama_index import (
    VectorStoreIndex,
    get_response_synthesizer,
```

Nanonets

```
from llama_index.indices.post

# Build index and configure r
index = VectorStoreIndex.from
retriever = VectorIndexRetrie
    index=index,
    similarity_top_k=2,
)

# Configure response synthesi
response_synthesizer = get_re

# Assemble query engine with
query_engine = RetrieverQuery
    retriever=retriever,
    response_synthesizer=resp
    node_postprocessors=[
        SimilarityPostprocess
    ]
)

# Execute the query
response = query_engine.query
print(response)
```

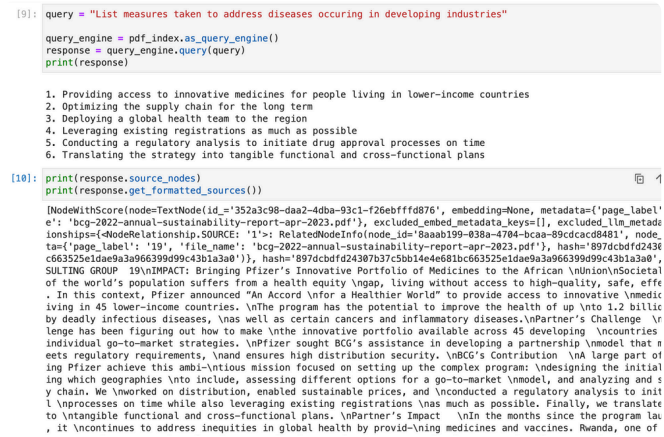
In the snippet above, the `VectorIndexRetriever`, `RetrieverQueryEngine`, and `SimilarityPostprocessor` are utilized to construct a customized query engine. This example demonstrates a more controlled query process.

returned which contains the response text and the sources of the response.

```
response = query_engine.query

# Get response
print(str(response))

# Get sources
print(response.source_nodes)
print(response.get_formatted_
```



This structure allows for a detailed examination of the query output and the sources contributing to the response.

Structured Outputs

Nanonets

streamlined workflow. LlamaIndex understands this and taps into the capabilities of Large Language Models (LLMs) to deliver structured results. Let's explore how.

Structured results aren't just a fancy way of presenting data; they are crucial for applications that depend on the precision and fixed structure of parsed values.

LLMs can give structured outputs in two ways:

Method 1 : Pydantic Programs

With **function calling APIs**, you get a naturally structured result, which then gets molded into the desired format, using **Pydantic Programs.** These nifty modules convert a prompt into a well-structured output using a Pydantic object. They can either call functions or use text completions along with output parsers. Plus, they gel well with search tools. LlamaIndex also offers ready-to-use Pydantic programs that change certain inputs into specific output types, like data tables.

Nanonets

data about these countries from the unstructured Wikipedia articles.

We create our pydantic output object -

```
from typing import List
from pydantic import BaseModel


class CountryInfo(BaseModel):
    """Data model for getting structured data about countries"""

    name: str
    official_languages: List[str]
    neighbouring_counties: List[str]
    form_of_government: str
    size_in_square_kilometer: int
```

We then create our index using the wikipedia document objects.

```
from llama_index import VectorStoreIndex, ServiceContext
from llama_index.llms import OpenAI

llm = OpenAI(model="gpt-3.5-turbo", temperature=0.1)
service_context = ServiceContext.from_defaults(llm=llm)

index = VectorStoreIndex.from_documents(
    wikipedia_documents, service_context=service_context
)
```

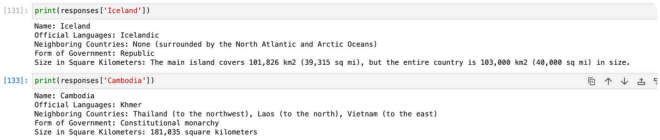
We initiate our query engine and specify the Pydantic output class.

```
query_engine = index.as_query_engine(
    output_cls=CountryInfo, response_mode="compact"
)
```

We can now expect structured response from the query engine. Let us retrieve this information for the three countries.

Nanonets

Let us inspect the responses object now.



Remember, while any LLM can technically produce these structured outputs, integration outside of OpenAI LLMs is still a work in progress.

Method 2 : Output Parsers

We can also use **generic completion APIs**, where text prompts dictate inputs and outputs. Here, the output parser ensures the final product aligns with the desired structure, guiding the LLM before and after its task. This is done with the help of **Output Parsers,** which acts as gatekeepers just before the final response is generated. They sit before and after an LLM text response and ensure everything's in order.

However, if you're using LLM functions discussed above that already give

Nanonets

We will follow the Output Parsers documentation here.

Let us import the LangChain output parser now.

```
from llama_index.output_parsers import LangchainOutputParser
from llama_index.llm_predictor import StructuredLLMPredictor
from langchain.output_parsers import StructuredOutputParser, ResponseSchema
```

We now define the structured LLM and the response format as shown in the documentation.

```
llm_predictor = StructuredLLMPredictor()

from llama_index.prompts import Prompt
from llama_index.prompts.default_prompts import (
    DEFAULT_TEXT_QA_PROMPT_TMPL,
    DEFAULT_REFINE_PROMPT_TMPL,
)

response_schemas = [
    ResponseSchema(
        name="Time",
        description=(
            "Time of occurence of event"
        ),
    ),
    ResponseSchema(
        name="Place",
        description="Place of occurence of event",
    ),
    ResponseSchema(
        name="Description",
        description="Event Description",
    )
]
```

We define the output parser and it's query template using the response_schemas defined above.

```
lc_output_parser = StructuredOutputParser.from_response_schemas(
    response_schemas
)
output_parser = LangchainOutputParser(lc_output_parser)

fmt_qa_tmpl = output_parser.format(DEFAULT_TEXT_QA_PROMPT_TMPL)
fmt_refine_tmpl = output_parser.format(DEFAULT_REFINE_PROMPT_TMPL)

qa_prompt = Prompt(fmt_qa_tmpl, output_parser=output_parser)
refine_prompt = Prompt(fmt_refine_tmpl, output_parser=output_parser)

print(fmt_qa_tmpl)
```

```
Context information is below.
---------------------
{context_str}
---------------------
Given the context information and not prior knowledge, answer the question: {query_str}


The output should be a markdown code snippet formatted in the following schema, including the leading and trailing "```json" and "```":

```json
{{
        "Time": string  // Time of occurence of event
        "Place": string  // Place of occurence of event
        "Description": string  // Event Description
}}
```
```

We define the query engine and pass the structured output parser template to it while creating it.

Nanonets

```
)
```

Running any query now fetches a structured json output!

```
response = query_engine.query(
    "Who was the first president of Iceland?",
)

print(response)
```

```
```json
{
        "Time": "17 June 1944",
        "Place": "Iceland",
        "Description": "Sveinn Björnsson became the first president of Iceland."
}
```
```

```
response = query_engine.query(
    "Describe the most important event in the history of Cambodia in the 21st century."
)

print(response)
```

```
```json
{
        "Time": "2004",
        "Place": "Cambodia",
        "Description": "Norodom Sihamoni was crowned Cambodia's king in 2004 after his father Sihanouk's abdication."
}
```
```

Structured outputs with LlamaIndex make parsing and downstream processes a breeze, ensuring you get the most out of your LLMs.

Automate manual tasks and workflows with our AI-driven workflow builder, designed by Nanonets for you and your teams.

Using Index to Chat with Data

Engaging in a conversation with your data takes querying a step further. LlamaIndex introduces the concept of a Chat Engine to facilitate a more interactive and contextual dialogue with your data. This section elaborates on setting up and utilizing the Chat Engine for a richer interaction with your indexed data.

Understanding the Chat Engine

A Chat Engine provides a high-level interface to have a back-and-forth conversation with your data, as opposed to a single question-answer interaction facilitated by the Query Engine. By maintaining a history of the conversation, the Chat Engine can provide answers that are contextually aware of previous interactions.

Nanonets

Getting Started with Chat Engine

Initiating a conversation is straightforward. Here's how you can get started:

```
# Build a chat engine from th
chat_engine = index.as_chat_e

# Start a conversation
response = chat_engine.chat("

# For streaming responses
streaming_response = chat_eng
for token in streaming_respon
    print(token, end="")
```

For example, we can start a chat with our PDF document on the BCG Annual Sustainability Report as follows.

You can choose the chat engine based on your use case. LlamaIndex

Nanonets

different needs and levels of sophistication. These engines are designed to facilitate conversations and interactions with users, each offering a unique set of features.

1. **SimpleChatEngine**

The SimpleChatEngine is a basic chat mode that does not rely on a knowledge base. It provides a starting point for chatbot development. However, it might not handle complex queries well due to its lack of a knowledge base.

2. **ReAct Agent Mode**

ReAct is an agent-based chat mode built on top of a query engine over your data. It follows a flexible approach where the agent decides whether to use the query engine tool to generate responses or not. This mode is versatile but highly dependent on the quality of the language model (LLM) and may require more control to ensure accurate responses. You can customize the LLM used in ReAct mode.

Implementation Example:

Nanonets

```
from llama_index.llms import

service_context = ServiceCont
data = SimpleDirectoryReader(
index = VectorStoreIndex.from

# Configure chat engine
chat_engine = index.as_chat_e

# Chat with your data
response = chat_engine.chat("
```

3. **OpenAI Agent Mode**

OpenAI Agent Mode leverages OpenAI's powerful language models like GPT-3.5 Turbo. It's designed for general interactions and can perform specific functions like querying a knowledge base. It's particularly suitable for a wide range of chat applications.

Implementation Example:

```
# Load data and build index
from llama_index import Vecto
from llama_index.llms import

service_context = ServiceCont
data = SimpleDirectoryReader(
index = VectorStoreIndex.from
```


```
# Chat with your data
response = chat_engine.chat("
```

4. **Context Mode**

The ContextChatEngine is a simple chat mode built on top of a retriever over your data. It retrieves relevant text from the index based on the user's message, sets this retrieved text as context in the system prompt, and returns an answer. This mode is ideal for questions related to the knowledge base and general interactions.

Implementation Example:

```
# Load data and build index
from llama_index import Vecto
data = SimpleDirectoryReader(
index = VectorStoreIndex.from

# Configure chat engine
from llama_index.memory impor
memory = ChatMemoryBuffer.fro
chat_engine = index.as_chat_e
    chat_mode="context",
    memory=memory,
    system_prompt=(
        "You are a chatbot, a
        " about an essay disc
    ),
```

Nanonets

```
response = chat_engine.chat("
```

5. **Condense Question Mode**

The Condense Question mode generates a standalone question from the conversation context and the last message. It then queries the query engine with this condensed question to provide a response. This mode is suitable for questions directly related to the knowledge base.

Implementation Example:

```
# Load data and build index
from llama_index import Vecto
data = SimpleDirectoryReader(
index = VectorStoreIndex.from

# Configure chat engine
chat_engine = index.as_chat_e

# Chat with your data
response = chat_engine.chat("
```

Configuring the Chat Engine

Configuration of a Chat Engine is akin to that of a Query Engine.

conversation according to your needs.

- **High-Level API**

```
chat_engine = index.as_chat_e
    chat_mode='condense_quest
    verbose=True
)
```

Different chat modes are available:

`best` : Optimizes the chat engine for use with a ReAct data agent or an OpenAI data agent, depending on the LLM support.

`context` : Retrieves nodes from the index using every user message, inserting the retrieved text into the system prompt for more contextual responses.

`condense_question` : Rewrites the user message to be a query for the index based on the chat history.

`simple` : Direct chat with the LLM, without involving the query engine.

`react` : Forces a ReAct data agent.

- **Low-Level Composition API**

For granular control, the low-level composition API allows explicit construction of the ChatEngine object:

```
from llama_index.prompts impo
from llama_index.llms import

# Custom prompt template
custom_prompt = PromptTemplat
... (template content) ...
""")

# Custom chat history
custom_chat_history = [ ... ]

# Query engine from index
query_engine = index.as_query

# Configuring the Chat Engine
chat_engine = CondenseQuestio
    query_engine=query_engine
    condense_question_prompt=
    chat_history=custom_chat_
    verbose=True
)
```

In this example, a custom prompt template and chat history are used to configure the

Nanonets

Streaming Responses

To enable streaming of responses, simply use the `stream_chat` endpoint:

```
streaming_response = chat_eng
for token in streaming_respon
    print(token, end="")
```

This feature provides a way to receive and process the response tokens as they are generated, which can be beneficial in certain interactive or real-time scenarios.

Resetting the Chat History

To start a new conversation or to discard the current conversation history, use the `reset` method:

```
chat_engine.reset()
```

Interactive Chat REPL

For an interactive chat session, use the `chat_repl` method which provides a Read-Eval-Print Loop (REPL) for chatting:

Nanonets

The Chat Engine in LlamaIndex extends the querying capability to a conversational paradigm, allowing for a more interactive and context-aware interaction with your data. Through various configurations and modes, you can tailor the conversation to suit your specific needs, whether it's a simple chat or a more complex, context-driven dialogue.

Llamaindex Tools and Data Agents

LlamaIndex Data Agents take natural language as input, and **perform actions instead of generating responses.**

The essence of constructing proficient data agents lies in the art of tool abstractions.

But what exactly is a tool in this context? Think of Tools as API interfaces, tailored for agent interactions rather than human touchpoints.

comes with a generic interface and some fundamental metadata like name, description, and function schema.

- **Tool Spec**: This dives deeper into the API details. It outlines a comprehensive service API specification, ready to be translated into an assortment of Tools.

There are different flavors of Tools available:

1. **FunctionTool**: Transform any user-defined function into a Tool. Plus, it can smartly infer the function's schema.
2. **QueryEngineTool**: Wraps around an existing query engine, and given our agent abstractions are derived from BaseQueryEngine, this tool can also embrace agents (which we will discuss later).

You can either custom design LlamaHub Tool Specs and Tools or effortlessly import them from the llama-hub package. Integrating them into agents is straightforward.

Nanonets

LlamaHub.

Data Agents in LlamaIndex are powered by Language Learning Models (LLMs) and act as **intelligent knowledge workers** over your data, executing both "read" and "write" operations. They automate search and retrieval across diverse data types—unstructured, semi-structured, and structured. Unlike our query engines which only "read" from a static data source, Data Agents can dynamically ingest, modify, and interact with data across various tools. They can call external service APIs, process the returned data, and store it for future reference.

The two building blocks of Data Agents are:

1. **A Reasoning Loop:** Dictates the agent's decision-making process on which tools to employ, their sequence, and the

Nanonets

2. **Tool Abstractions:** A set of APIs or Tools that the agent interacts with to fetch information or alter states.

The type of reasoning loop depends on the agent; supported types include OpenAI Function agent and a ReAct agent (which operates across any chat/text completion endpoint).

Here's how to use an OpenAI Function API-based Data Agent:

```
from llama_index.agent import
from llama_index.llms import
... # import and define tools
llm = OpenAI(model="gpt-3.5-t
agent = OpenAIAgent.from_tool
```

Now, let us go ahead and use the Code Interpreter tool available in LlamaHub to write and execute code directly by giving natural language instructions. We will use this Spotify dataset (which is a .csv file) and perform data analysis by making our agent execute python code to read and manipulate the data in pandas.

We first import the tool.

Nanonets

Let's start chatting.

```
print(
    agent.chat(
        "Can you help me write some python code to pass to the code_interpreter tool"
    )
)
```

```
Of course! I'd be happy to help you write some Python code. What specific task or problem would you like the code to solve?
```

We first ask it to fetch the list of columns. Our agent executes python code and uses pandas to read the column names.

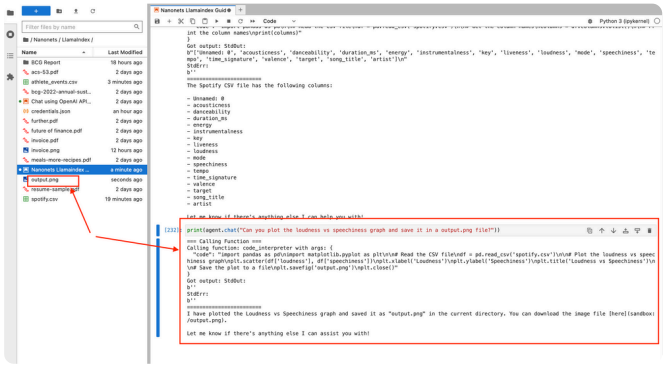
```
print(
    agent.chat(
        """There is a spotify.csv file in the current directory (relative path).
                Can you write and execute code to tell me columns does it have?"""
    )
)
```

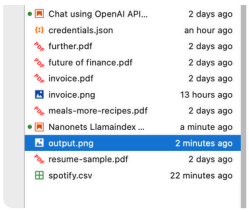
```
=== Calling Function ===
Calling function: code_interpreter with args: {
  "code": "import pandas as pd\n\n# Read the CSV file\ndf = pd.read_csv('spotify.csv')\n\n# Get the column names\ncolumns = df.columns.tolist()\n\n# Pr
int the column names\nprint(columns)"
}
Got output: StdOut:
b"['Unnamed: 0', 'acousticness', 'danceability', 'duration_ms', 'energy', 'instrumentalness', 'key', 'liveness', 'loudness', 'mode', 'speechiness', 'te
mpo', 'time_signature', 'valence', 'target', 'song_title', 'artist']\n"
StdErr:
b''
========================
The Spotify CSV file has the following columns:

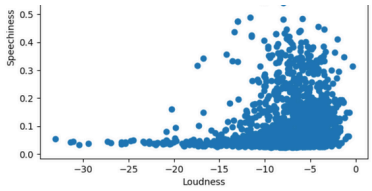
- Unnamed: 0
- acousticness
- danceability
- duration_ms
- energy
- instrumentalness
- key
- liveness
- loudness
- mode
- speechiness
- tempo
- time_signature
- valence
- target
- song_title
- artist

Let me know if there's anything else I can help you with!
```

We now ask it to plot a graph of loudness vs 'speechiness' and save it in a output.png file on our system, all by just chatting with our agent.







We can perform EDA in detail as well.

```
print(agent.chat("Can you give top 5 artists with at least 10 songs with highest average energy?"))
```

```
=== Calling Function ===
Calling function: code_interpreter with args: {
    "code": "import pandas as pd\n\n# Read the CSV file\ndf = pd.read_csv('spotify.csv')\n\n# Group by artist and count the number of songs\nartist_count
s = df.groupby('artist').size()\n\n# Filter artists with at least 10 songs\nartists_with_10_songs = artist_counts[artist_counts >= 10]\n\n# Filter the
dataframe for artists with at least 10 songs\ndf_filtered = df[df['artist'].isin(artists_with_10_songs.index)]\n\n# Calculate the average energy for ea
ch artist\naverage_energy = df_filtered.groupby('artist')['energy'].mean()\n\n# Sort the artists by average energy in descending order\ntop_5_artists =
average_energy.nlargest(5)\n\n# Print the top 5 artists\nprint(top_5_artists)"
}
Got output: StdOut:
b'artist\nWALK THE MOON      0.819200\nDisclosure         0.777750\nRick Ross          0.754769\nBackstreet Boys    0.736000\nDrake              0.5647
50\nName: energy, dtype: float64\n'
StdErr:
b''
========================
The top 5 artists with at least 10 songs and the highest average energy are:

1. WALK THE MOON - Average Energy: 0.819200
2. Disclosure - Average Energy: 0.777750
3. Rick Ross - Average Energy: 0.754769
4. Backstreet Boys - Average Energy: 0.736000
5. Drake - Average Energy: 0.564750

Let me know if there's anything else I can assist you with!
```

Now, let us go ahead and use another tool to send emails from our Gmail account with an agent using natural language input.

We use the data connector with Hubspot from LlamaHub to fetch a structured list of leads which were created yesterday.

```
from llama_index import download_loader
import openai
import os
openai.api_key = "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"

HubspotReader = download_loader('HubspotReader')

reader = HubspotReader("your_api_key_here")
documents = reader.load_data()
```

```
from llama_index.llms import OpenAI
from llama_index import ServiceContext, OpenAIEmbedding
from llama_index.indices.document_summary import DocumentSummaryIndex
from llama_index.response_synthesizers import ResponseMode, get_response_synthesizer

chatgpt = OpenAI(temperature=0, model="gpt-3.5-turbo")
service_context = ServiceContext.from_defaults(llm=chatgpt, chunk_size=1024)

response_synthesizer = get_response_synthesizer(
    response_mode="tree_summarize", use_async=True
)

doc_summary_index = DocumentSummaryIndex.from_documents(documents=documents)
```

```
from typing import List
from pydantic import BaseModel


class HubspotLeadsOneDayEmail(BaseModel):
    """Data model for leads eligible for one day follow up emails."""

    lead_emails: List[str]
```

```
query_engine = index.as_query_engine(
    output_cls=HubspotLeadsOneDayEmail, response_mode="compact"
)

response = query_engine.query("Give me list of email addresses of leads created yesterday")
```

Nanonets

create a Gmail Agent to write and send one day follow-up emails to these folks.

We start by creating a credentials.json file by following the documentation and add it to our working directory.

We then proceed to provide the email writing prompt.

```
import openai

openai.api_key = "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"
from llama_index.agent import OpenAIAgent

from llama_hub.tools.gmail.base import GmailToolSpec
tool_spec = GmailToolSpec()

agent = OpenAIAgent.from_tools(tool_spec.to_tool_list(), verbose=True)

print(agent.chat(
    """I want to write follow up emails to leads who registered on Nanonets yesterday.
    For each of these leads, write an email draft elaborating how their specific company can benefit from Nanonets OCR - """ + response
))
```

Upon running the code, we can see the execution chain writing the emails and saving them as drafts in your Gmail account.

```
=== Calling Function ===
Calling function: create_draft with args: {
  "to": ["kevin@accounts_payable.com", "rachel@accounts_payable.com"],
  "subject": "Benefits of OCR in Accounts Payable Automation",
  "message": "Dear Kevin and Rachel,\n\nI hope this email finds you well. I wanted to bring to your attention the benefits of OCR (Optical Character Recognition) in accounts payable automation.\n\nOCR technology allows for the automatic extraction of data from invoices, receipts, and other financial documents. This eliminates the need for manual data entry, reducing errors and saving time.\n\nBy implementing OCR in accounts payable processes, organizations can streamline their invoice processing, improve accuracy, and enhance efficiency. OCR can also help with data validation, fraud detection, and compliance.\n\nI believe th
karan@Karan LlamaIndex % python3 gmail.py
=== Calling Function ===
Calling function: create_draft with args: {
  "to": ["kevin@accountspayable.com", "rachel@accountspayable.com"],
  "subject": "Benefits of OCR in Accounts Payable Automation",
  "message": "Dear Kevin and Rachel,\n\nI hope this email finds you well. I wanted to reach out to discuss the benefits of OCR (Optical Character Recognition) in accounts payable automation.\n\nOCR technology has revolutionized the way businesses handle invoice processing and accounts payable tasks. By automatically extracting data from invoices, OCR eliminates the need for manual data entry, reducing errors and saving valuable time.\n\nHere are some key benefits of OCR in accounts payable automation:\n\n1. Increased Efficiency: OCR technology can process invoices at a much faster rate compared to manual data entry, allowing businesses to streamline their accounts payable processes.\n\n2. Cost Savings: By automating invoice processing with OCR, businesses can reduce labor costs associated with manual data entry and improve overall operational efficiency.\n\n3. Improved Accuracy: OCR technology ensures accurate data extraction from invoices, minimizing errors and reducing the risk of payment discrepancies.\n\n4. Enhanced Data Visibility: With OCR, businesses can gain better visibility into their accounts payable data, enabling them to make more informed financial decisions.\n\nI would be happy to provide more information or schedule a meeting to discuss how OCR can benefit your accounts payable department. Please let me know your availability.\n\nBest regards,\n[Your Name]"
}
Got output: {'id': 'r4817409398252475927', 'message': {'id': '18b4647baab24717', 'threadId': '18b4647baab24717', 'labelIds': ['DRAFT']}}
========================
I have created a new email draft to Kevin and Rachel at kevin@accountspayable.com and rachel@accountspayable.com. The subject of the email is "Benefits of OCR in Accounts Payable Automation". Here is the draft message:

```
Dear Kevin and Rachel,

I hope this email finds you well. I wanted to reach out to discuss the benefits of OCR (Optical Character Recognition) in accounts payable automation.

OCR technology has revolutionized the way businesses handle invoice processing and accounts payable tasks. By automatically extracting data from invoices, OCR eliminates the need for manual data entry, reducing errors and saving valuable time.

Here are some key benefits of OCR in accounts payable automation:

1. Increased Efficiency: OCR technology can process invoices at a much faster rate compared to manual data entry, allowing businesses to streamline their accounts payable processes.

2. Cost Savings: By automating invoice processing with OCR, businesses can reduce labor costs associated with manual data entry and improve overall operational efficiency.

3. Improved Accuracy: OCR technology ensures accurate data extraction from invoices, minimizing errors and reducing the risk of payment discrepancies.

4. Enhanced Data Visibility: With OCR, businesses can gain better visibility into their accounts payable data, enabling them to make more informed financial decisions.

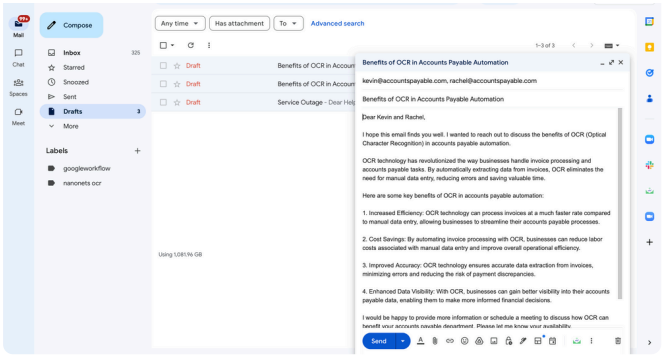
I would be happy to provide more information or schedule a meeting to discuss how OCR can benefit your accounts payable department. Please let me know your availability.

Best regards,
[Your Name]
```

Please review the draft and make any necessary changes before sending it.
```

When we visit our Gmail Draft box, we can see the draft that the agent

Nanonets



In fact, we could have also scheduled this script to run daily and instructed the Gmail agent to send these emails directly instead of saving them as drafts, resulting in a end-to-end fully automated workflow that -

- connects with hubspot daily to fetch yesterday's leads
- uses the domain from the email addresses to create personalized follow up emails
- sends the emails from your Gmail account directly.

Thus, in addition to querying and chatting with data, LlamaIndex can be also be used to fully execute tasks by interacting with applications and data sources.

And that's a wrap!

Level up with Nanonets

LlamaIndex is a great tool to connect your data, irrespective of its format, with LLMs and harness their prowess to interact with it. We have seen how to use natural language with your data and apps in order to generate responses / perform tasks.

However, for businesses, using LlamaIndex for enterprise applications might pose a few problems -

Nanonets

of data connectors, this list is still not exhaustive and misses out on providing connections with some of the major workspace apps.

2. This trend is even more increasingly prevalent in the context of tools, which take natural language as input to perform tasks (write and send emails, create CRM entries, execute SQL queries and fetch results, and so on). This limits the number of supported apps and ways in which the agents can interact with them.
3. Finetuning the configuration of each element of the LlamaIndex pipeline (retrievers, synthesizers, indices, and so on) is a cumbersome process. On top of that, identifying the best pipeline for a given dataset and task is time consuming and not always intuitive.
4. Each task needs a unique implementation. There is no one-stop solution for connecting your data with LLMs.
5. LlamaIndex fetches static data with data connectors, which is not updated with new data

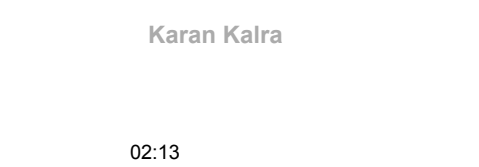
Nanonets

Enter Nanonets Workflows!

Harnessing the Power of Workflow Automation: A Game-Changer for Modern Businesses

In today's fast-paced business environment, workflow automation stands out as a crucial innovation, offering a competitive edge to companies of all sizes. The integration of automated workflows into daily business operations is not just a trend; it's a strategic necessity. In addition to this, the advent of LLMs has opened even more opportunities for automation of manual tasks and processes.

Welcome to Nanonets Workflow Automation, where AI-driven technology empowers you and your team to automate manual tasks and construct efficient workflows in minutes. Utilize natural language to effortlessly create and manage workflows that seamlessly integrate



Our platform offers not only seamless app integrations for unified workflows but also the ability to build and utilize custom Large Language Models Apps for sophisticated text writing and response posting within your apps. All the while ensuring data security remains our top priority, with strict adherence to GDPR, SOC 2, and HIPAA compliance standards.

To better understand the practical applications of Nanonets workflow automation, let's delve into some real-world examples.

- **Automated Customer Support and Engagement Process**

Nanonets

Karan Kalra

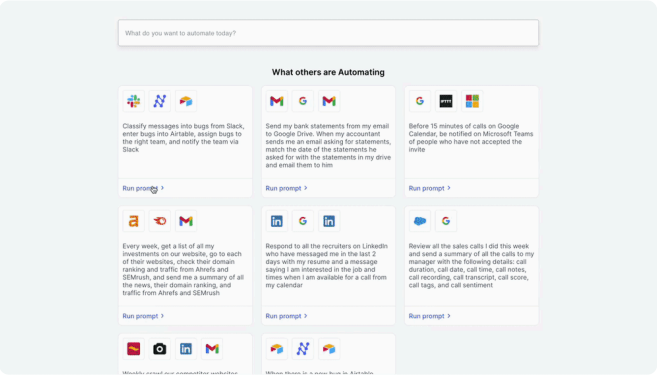
02:04

- **Ticket Creation – Zendesk**: The workflow is triggered when a customer submits a new support ticket in Zendesk, indicating they need assistance with a product or service.
- **Ticket Update – Zendesk**: After the ticket is created, an automated update is immediately logged in Zendesk to indicate that the ticket has been received and is being processed, providing the customer with a ticket number for reference.
- **Information Retrieval – Nanonets Browsing**: Concurrently, the Nanonets Browsing feature searches through all the knowledge base pages to find relevant information and possible solutions related to the customer's issue.
- **Customer History Access – HubSpot**: Simultaneously, HubSpot is queried to retrieve the customer's previous interaction records, purchase history, and any past tickets to

Nanonets

- **Ticket Processing – Nanonets AI**: With the relevant information and customer history at hand, Nanonets AI processes the ticket, categorizing the issue and suggesting potential solutions based on similar past cases.
- **Notification – Slack**: Finally, the responsible support team or individual is notified through Slack with a message containing the ticket details, customer history, and suggested solutions, prompting a swift and informed response.

- **Automated Issue Resolution Process**



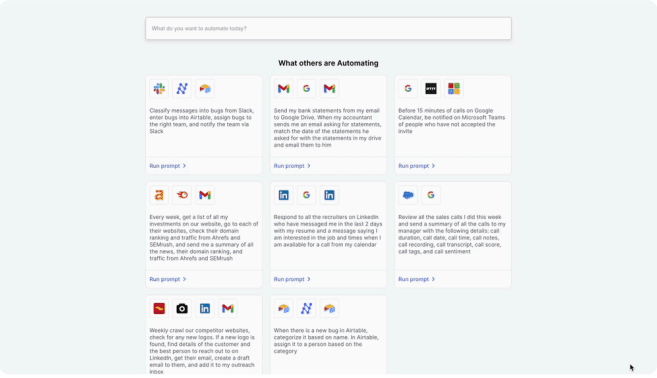
1. **Initial Trigger – Slack Message**: The workflow begins when a customer service representative receives a new message in a dedicated channel on Slack, signaling a customer

2. **Classification – Nanonets AI**: Once the message is detected, Nanonets AI steps in to classify the message based on its content and past classification data (from Airtable records). Using LLMs, it classifies it as a bug along with determining urgency.
3. **Record Creation – Airtable**: After classification, the workflow automatically creates a new record in Airtable, a cloud collaboration service. This record includes all relevant details from the customer's message, such as customer ID, issue category, and urgency level.
4. **Team Assignment – Airtable**: With the record created, the Airtable system then assigns a team to handle the issue. Based on the classification done by Nanonets AI, the system selects the most appropriate team – tech support, billing, customer success, etc. – to take over the issue.
5. **Notification – Slack**: Finally, the assigned team is notified through Slack. An automated message is sent to the team's

Nanonets

link to the Airtable record, and prompting a timely response.

- **Automated Meeting Scheduling Process**



1. **Initial Contact – LinkedIn**: The workflow is initiated when a professional connection sends a new message on LinkedIn expressing interest in scheduling a meeting. An LLM parses incoming messages and triggers the workflow if it deems the message as a request for a meeting from a potential job candidate.
2. **Document Retrieval – Google Drive**: Following the initial contact, the workflow automation system retrieves a pre-prepared document from Google Drive that contains information about the meeting agenda, company overview, or any relevant briefing materials.

Nanonets

Google calendar to get available times for the meeting. It checks the calendar for open slots that align with business hours (based on the location parsed from LinkedIn profile) and previously set preferences for meetings.

4. **Confirmation Message as Reply – LinkedIn**: Once a suitable time slot is found, the workflow automation system sends a message back through LinkedIn. This message includes the proposed time for the meeting, access to the document retrieved from Google Drive, and a request for confirmation or alternative suggestions.

- **Invoice Processing in Accounts Payable**

- **Data Extraction - Nanonets OCR**: The system automatically extracts relevant data (like vendor details, amounts, due dates).
- **Data Verification - Quickbooks:** The Nanonets workflow verifies the extracted invoice data against purchase orders and receipts.
- **Approval Routing - Slack**: The invoice is routed to the appropriate manager for approval based on predefined thresholds and rules.
- **Payment Processing - Brex**: Once approved, the system schedules the payment according to the vendor's terms and updates the finance records.
- **Archiving - Quickbooks**: The completed transaction is archived for future reference and audit trails.

- **Internal Knowledge Base Assistance**

Nanonets

- **Initial Inquiry – Slack**: A team member, Smith, inquires in the #chat-with-data Slack channel about customers experiencing issues with QuickBooks integration.
- **Automated Data Aggregation - Nanonets Knowledge Base**:
    - **Ticket Lookup - Zendesk**: The Zendesk app in Slack automatically provides a summary of today's tickets, indicating that there are issues with exporting invoice data to QuickBooks for some customers.
    - **Slack Search** - Slack: Simultaneously, the Slack app notifies the channel that team members Patrick and Rachel are actively discussing the resolution of the QuickBooks export bug in another channel, with a fix scheduled to go live at 4 PM.
    - **Ticket Tracking – JIRA**: The JIRA app updates the channel about a ticket created by Emily titled "QuickBooks export failing for QB Desktop

Nanonets

issue.

- **Reference Documentation – Google Drive**: The Drive app mentions the existence of a runbook for fixing bugs related to QuickBooks integrations, which can be referenced to understand the steps for troubleshooting and resolution.
- **Ongoing Communication and Resolution Confirmation – Slack**: As the conversation progresses, the Slack channel serves as a real-time forum for discussing updates, sharing findings from the runbook, and confirming the deployment of the bug fix. Team members use the channel to collaborate, share insights, and ask follow-up questions to ensure a comprehensive understanding of the issue and its resolution.
- **Resolution Documentation and Knowledge Sharing**: After the fix is implemented, team members update the internal documentation in Google Drive with new findings and any additional steps taken to resolve the issue. A summary of the incident, resolution, and any lessons learned are already shared in the Slack channel. Thus, the team's internal knowledge base is

Nanonets

to focus on what truly matters.

Related content

WORKFLOW AUTOMATION

Nanonets

WORKFLOW AUTOMATION

Order entry automation simplified

WORKFLOW AUTOMATION

What is the Role of AI in Lending and Loan Management?

WORKFLOW AUTOMATION

Nanonets

SOLUTIONS

AP Automation

Touchless Invoice Processing

Email Parsing

ERP Integrations

RESOURCES

Customer Success Stories

Blog

Help Center

API Documentation

COMPANY

About

Investors

Careers

Privacy Policy

Terms of Servi

CONTACT

+1-650-381-0077

info@nanonets.com

2261 Market Street #4010,
San Francisco, CA 94114, USA