Kritin Vongthongsri
Cofounder @ Confident AI | LLM Evals & Safety Wizard | Previously ML + CS @ Princeton Researching Self-Driving Cars

Using LLMs for Synthetic Data Generation: The Definitive Guide

Constructing a large-scale, comprehensive dataset to test LLM outputs can be a **laborious, costly, and challenging** process, especially if done from scratch. But what if I told you that it's now possible to generate the same thousands of high-quality test cases you spent weeks painstakingly crafting, in just a few minutes?

Synthetic data generation **leverages LLMs to create quality data without the need to manually collect, clean, and annotate massive datasets**. With models like GPT-4, it's now possible to synthetically produce datasets that are more comprehensive and diverse than human-labeled ones,

in far less time, which can be used to benchmark LLM (systems) with the help of some LLM evaluation metrics.

In this article, I'll teach you everything you need to know on how to use LLMs to **generate synthetic datasets** (which for example can be used to evaluate RAG pipelines). We'll explore:

- **Synthetic generation methods** (Distillation and Self-Improvement)
- What **data evolution** is, various evolution techniques, and its role in synthetic data generation
- A **step-by-step tutorial** on creating high-quality synthetic data from scratch using LLMs.
- How to use DeepEval to generate synthetic datasets in under 5 lines of code.

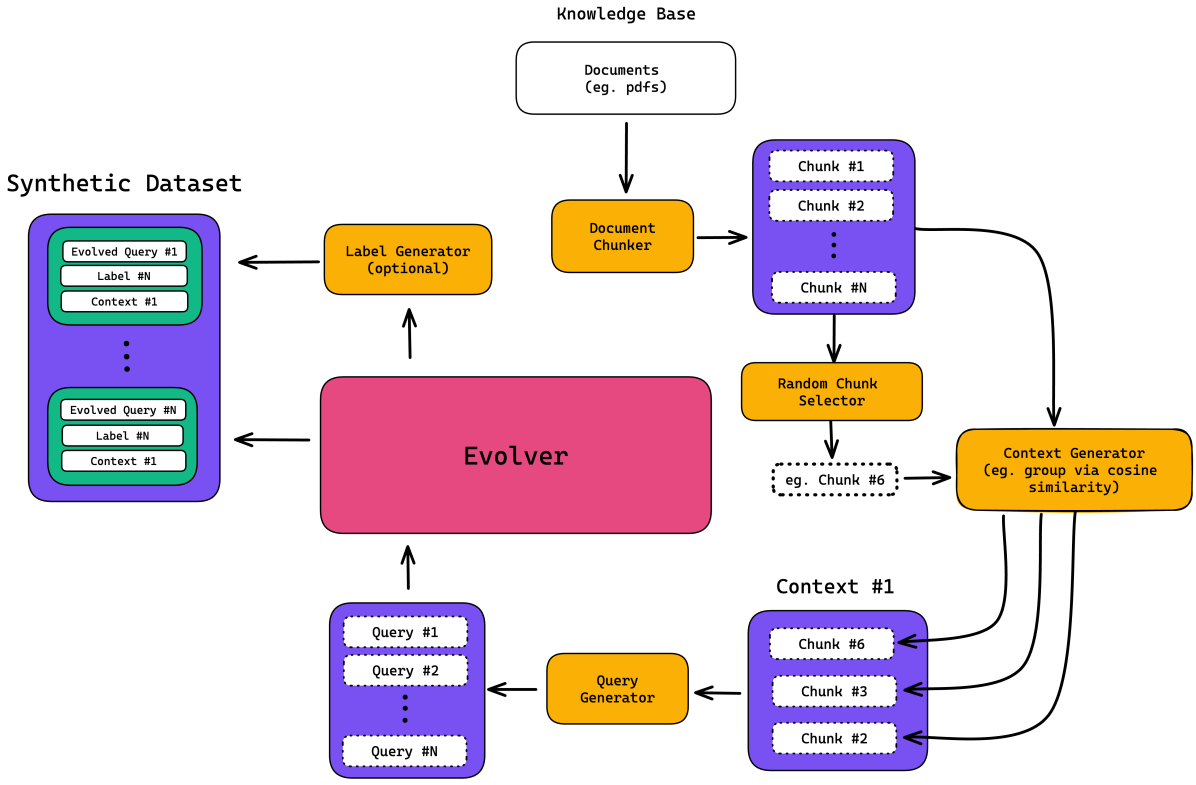
Intrigued? Let's dive in.

What is Synthetic Data Generation, Using LLMs?

Synthetic data generation using LLMs involves using an LLM to **create artificial data**, which often are datasets that can be used to train, fine-tune, and even evaluate LLMs themselves. Generating synthetic datasets is not only faster than scouring public datasets and cheaper than human annotation but also results in higher quality and data diversity, which is also imperative for LLM red teaming.

The process starts with the creation of synthetic queries, which are generated using context from your knowledge base (often in the form of documents) as the ground truth. The generated queries are then "evolved" multiple times to complicate and make realistic, and when combined with the original context it was generated from, makes up your final synthetic dataset. Although

optional, you can also choose to generate a target label for each synthetic query-context pair, which will act as the expected output of your LLM system for a given query.



A Data Synthesizer Architecture

When it comes to generating a synthetic dataset for evaluation, there are two main methods: self-improvement from using your model's output, or distillation from a more advanced model.

- **Self-improvement:** involves your model generating data iteratively from its own output without external dependencies
- **Distillation:** involves using a stronger model to generate synthetic data for to evaluate a weaker model

Self-improvement methods, such as Self-Instruct or SPIN, are limited by a model's capabilities and may suffer from amplified biases and errors. In contrast, distillation techniques are only limited by the best model available, ensuring the highest quality generation.

Generating Data from your Knowledge Base

The first step in synthetic data generation involves creating *synthetic queries from your list of contexts*, which are sourced directly from your knowledge base. Essentially, contexts serve as the **ideal retrieval contexts** for your LLM application, much like how expected outputs serve as ground truth references for your LLM's actual outputs.

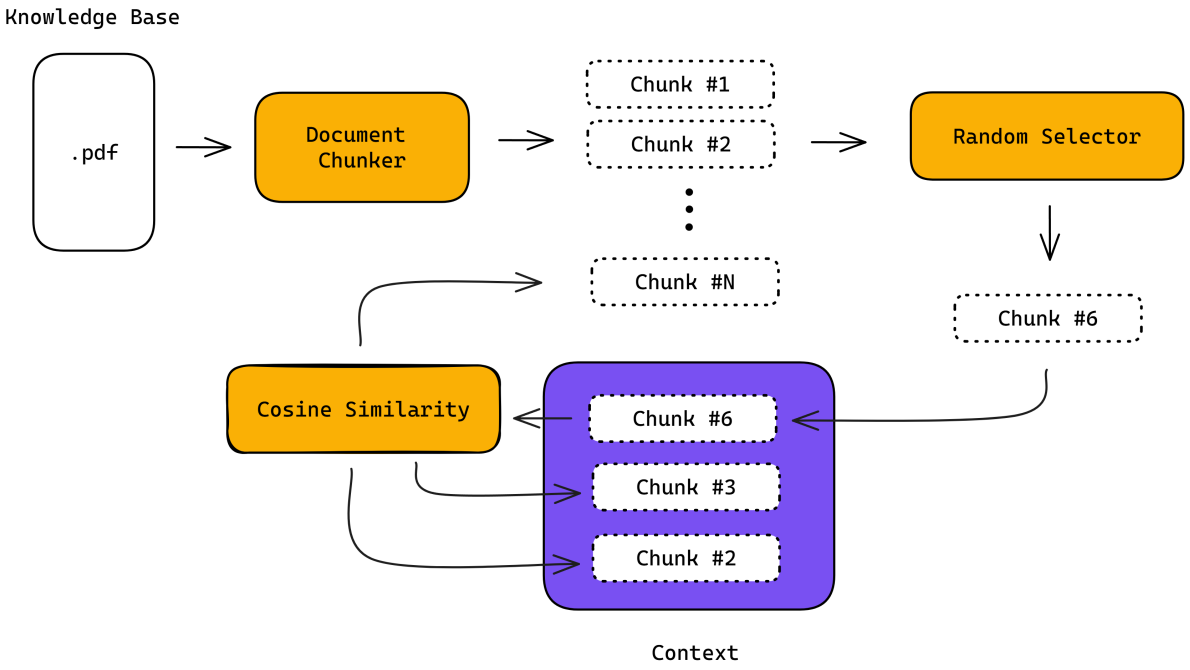
For those who wants something working immediately, here is how you can generate high quality synthetic data using **DeepEval**, the open-source LLM evaluation framework (⭐github here: [https://github.com/confident-ai/deepeval)](https://github.com/confident-ai/deepeval):

```
from deepeval.synthesizer import Synthesizer

synthesizer = Synthesizer()
synthesizer.generate_goldens_from_docs(
    document_paths=['example.txt', 'example.docx', 'example.pdf'],
)
```

Constructing Contexts

During context generation, a document — or multiple documents — from your **knowledge base** is divided into chunks using a token splitter. A random chunk is then selected, and additional chunks are retrieved and grouped with the selected one based on their **similarity**.



Context Generation using Cosine Similarity

Similarity can be calculated using several methods:

- Utilizing **similarity or distance algorithms**, such as cosine similarity
- Employing **knowledge graphs**
- Leveraging **LLMs** themselves, which, though less realistic, offer the most accuracy
- Applying **clustering** techniques to identify patterns
- Using **machine learning models** to predict groupings based on features.

In any case, the overarching goal remains the same: **to aggregate similar chunks of information effectively.**

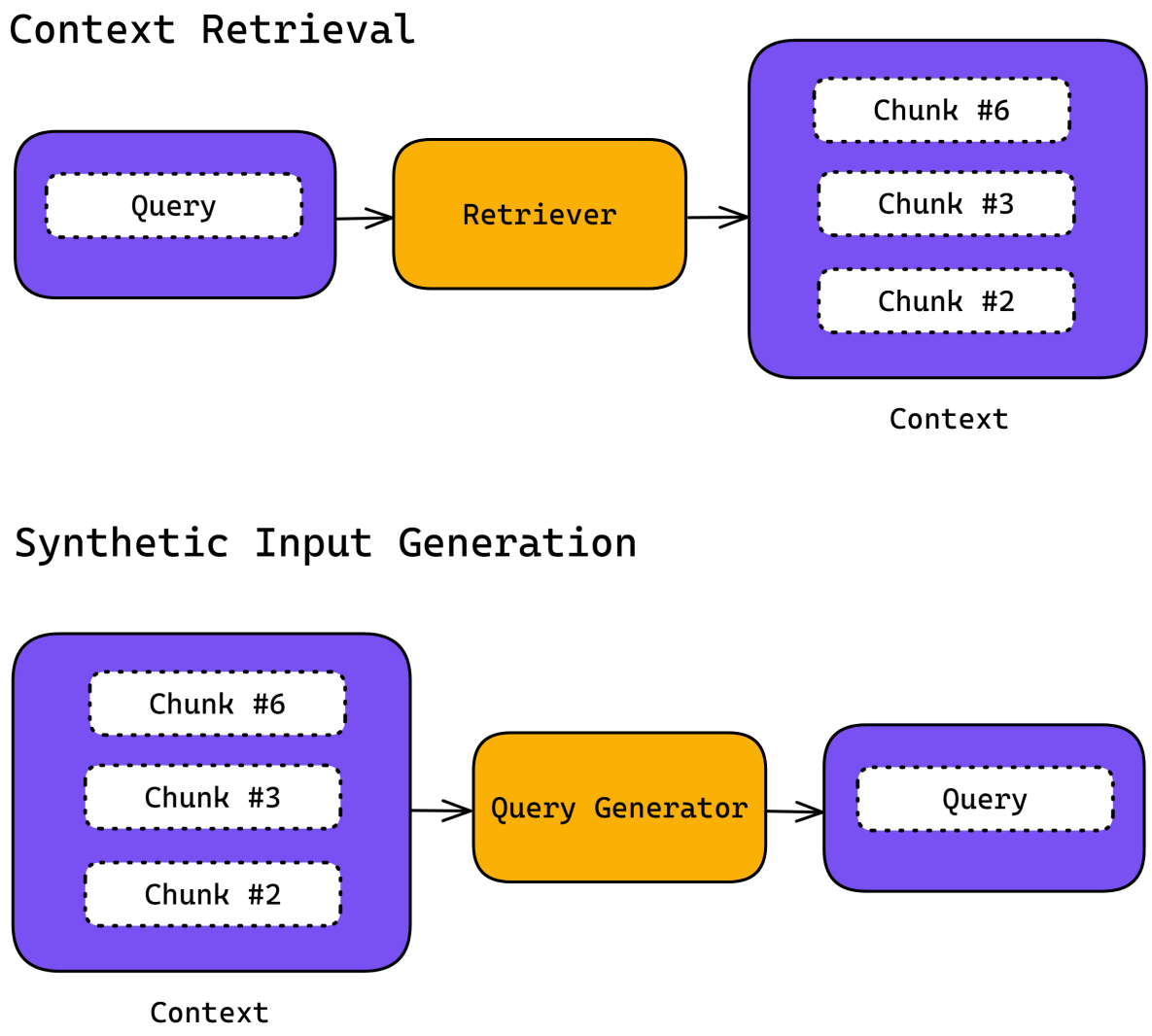
To ensure that these groupings are effective and align with the specific needs of your LLM application, *it's best practice to mirror your application's retriever logic in the context generation process*. This includes carefully considering aspects such as the token-splitting method, chunk size, and chunk overlap.

Such alignment guarantees that the synthetic data behaves consistently with your application's expectations, preventing any skew in outcomes due to differences in retriever complexity.

Generating Synthetic Inputs from Contexts

Once your contexts have been created, *synthetic inputs are subsequently generated from them*. This method **reverses the standard retrieval operation**—instead of locating contexts based on inputs, inputs are created based on predefined contexts. This ensures that every synthetic input directly corresponds to a context, enhancing relevance and accuracy.

Additionally, these contexts are used to optionally *produce the expected outputs* by aligning them with the synthetically generated inputs.



Asymmetric Approach to Query Generation

This asymmetric approach ensures that all components — inputs, outputs, and contexts — are perfectly synchronized.

Filtering Synthetic Data

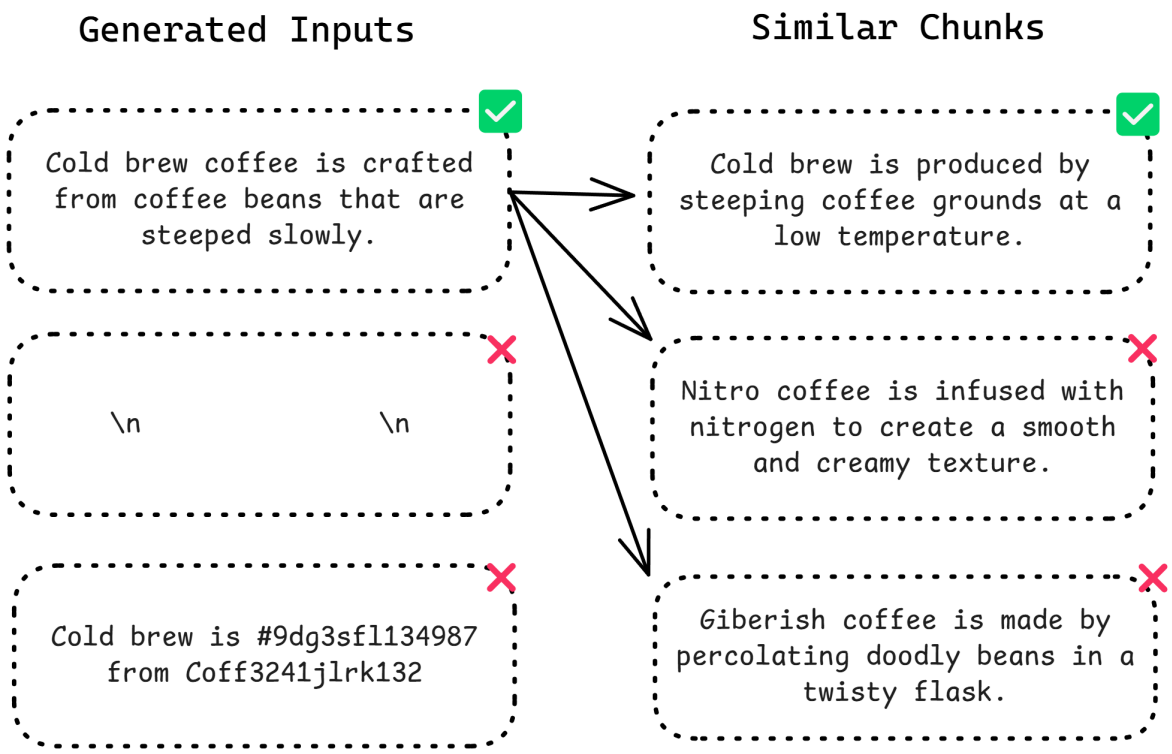
Before you begin evolving your newly generated datasets, *it's essential to conduct thorough quality checks* to avoid refining inputs that are inherently

flawed. This step is crucial to ensure that no valuable resources are wasted and that your final dataset only contains high-quality goldens.

Filtering occurs at **two critical stages of synthetic data generation**: initially during context generation, and subsequently during the generation of synthetic inputs from these contexts.

Context Filtering

During context generation, there's a chance you might randomly select a low-quality chunk. Oftentimes, your knowledge base may contain complex structures or excess whitespace that becomes unintelligible when broken down. Employing LLMs as judges is a robust method for identifying and eliminating these low-quality contexts.



Context Filtering Example

You may customize the criteria for evaluating and filtering out these contexts, but here are some foundational guidelines to consider:

- **Clarity:** Evaluate how clear and understandable the information is.

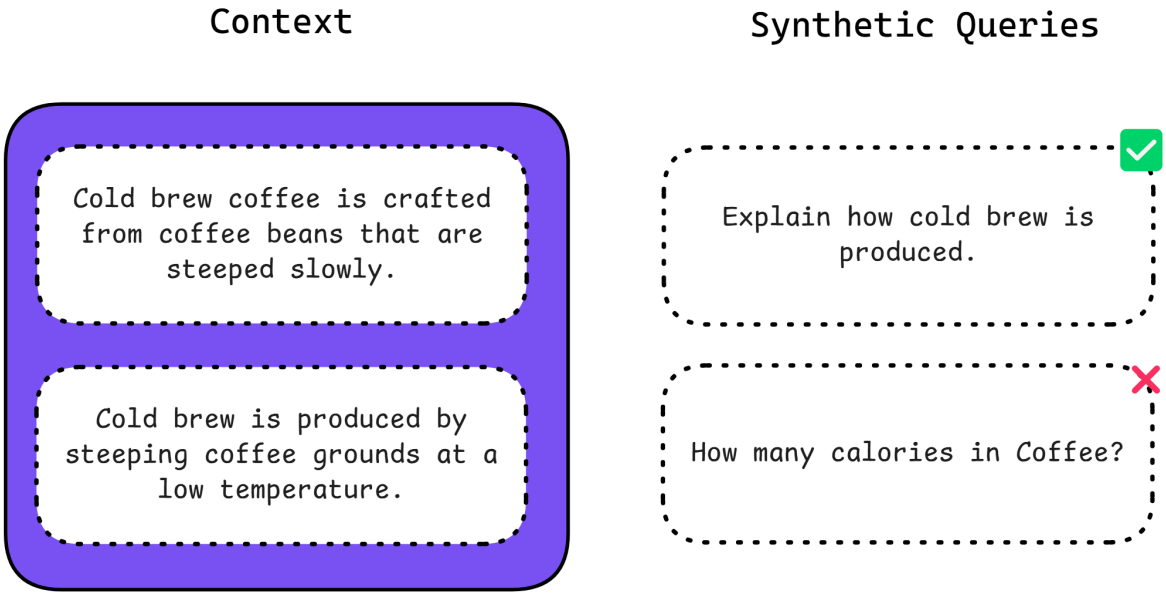
- **Depth:** Assess the level of detailed analysis and presence of original insights.
- **Structure:** Review the organization and logical progression of the content.
- **Relevance:** Determine the content's pertinence to the main topic.
- **Precision:** Gauge the accuracy and attention to detail.
- **Novelty:** Assess the uniqueness and originality of the content.
- **Conciseness:** Evaluate the brevity and efficiency of the communication.
- **Impact:** Judge the potential effect of the content on the audience.

You'll also need to ensure that the remaining chunks are similar enough once you finish filtering out low-quality chunks. This involves a secondary filtering process where you weed out chunks that don't pass the similarity threshold.

This structured approach to filtering ensures that only high-quality, relevant, and useful contexts proceed to the next stage of synthetic data generation.

Input Filtering

The second filtering stage focuses on the **synthetic inputs generated from these contexts**. This step is crucial because even well-curated contexts can sometimes lead to the generation of inputs that *might not meet the required standards*.



Input Filtering Example

Here are a few criteria you may want to judge your synthetic input on:

- **Self-containment:** Ensures the input is complete and can function independently without external references.
- **Clarity:** Checks that the input clearly communicates its intended message or question to avoid misinterpretation.
- **Consistency:** Assures the input aligns with the provided context or background information thematically and factually.
- **Relevance:** Verifies that the input directly pertains to the intended tasks or queries, ensuring it's purposeful and on-topic.
- **Completeness:** Confirms the input includes all necessary details for effective interaction or query resolution.

Using these criteria helps ensure that synthetic inputs are not only high in quality but also tailored perfectly to their intended applications.

Styling Synthetic Data

Finally, you may want to **tailor your queries to specific topics and customize their input and output formats** to fit your unique use cases.

For example, if your application involves converting text to SQL, the outputs should accurately reflect SQL statements. In scenarios involving an evaluative LLM, using a JSON format with keys like 'score' and 'reason' might be more appropriate.

You should plan to apply specific styling *during the initial generation, through any evolutionary changes, and after the final outputs are generated*. Revisiting the style after the initial generation is crucial, as the evolution of synthetic queries can alter the initial styling intentions.

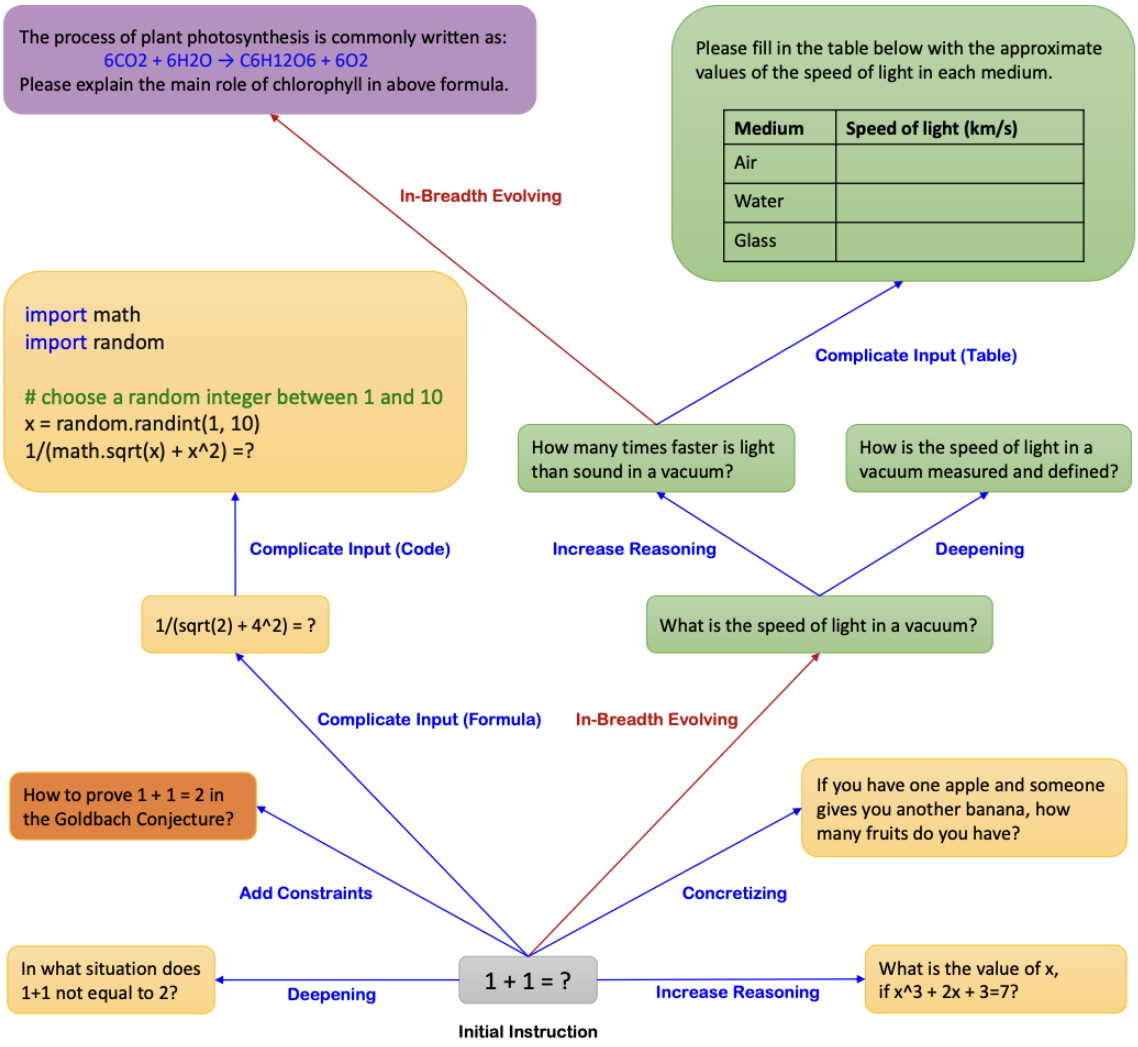
The extent of style adjustments after the first round will depend on your desired level of control over the final product and the associated cost implications.

Data Survival-Of-The-Fittest

Let's clarify what data evolution is and why it's so important to synthetic data generation using LLMs. Data evolution, first introduced in Microsoft's Evol-Instruct, involves **iteratively enhancing an existing set of queries to generate more complex and diverse ones** through prompt engineering. This step is crucial for ensuring the quality, comprehensiveness, complexity, and diversity of the dataset. It's what makes synthetic data superior to public or human-annotated datasets.

In fact, the original authors managed to produce 250,000 instructions from just 175 human-created queries. There are 3 types of data evolution:

- **In-Depth Evolving:** Expands a simple instruction into a more detailed and intricate version.
- **In-Breadth Evolving:** Produces new, diverse instructions to enrich the dataset.
- **Elimination Evolving:** Removes less effective or failed instructions.



In-depth (blue) and in-breadth (red) evolutions of '1+1' query

There are several ways to perform in-depth evolution, such as complicating inputs, increasing the need for reasoning, or adding multiple steps to complete a task. Each approach contributes to a higher level of sophistication in the generated data.

In-depth evolution ensures the creation of **nuanced, high-quality** queries, while in-breadth evolution enhances **diversity and comprehensiveness.** By evolving each query or instruction multiple times, we increase its complexity, resulting in a rich and multifaceted dataset. But enough of me talking, let's show you how to put everything in action.

Take this query for example:

> *What is 1+1?*

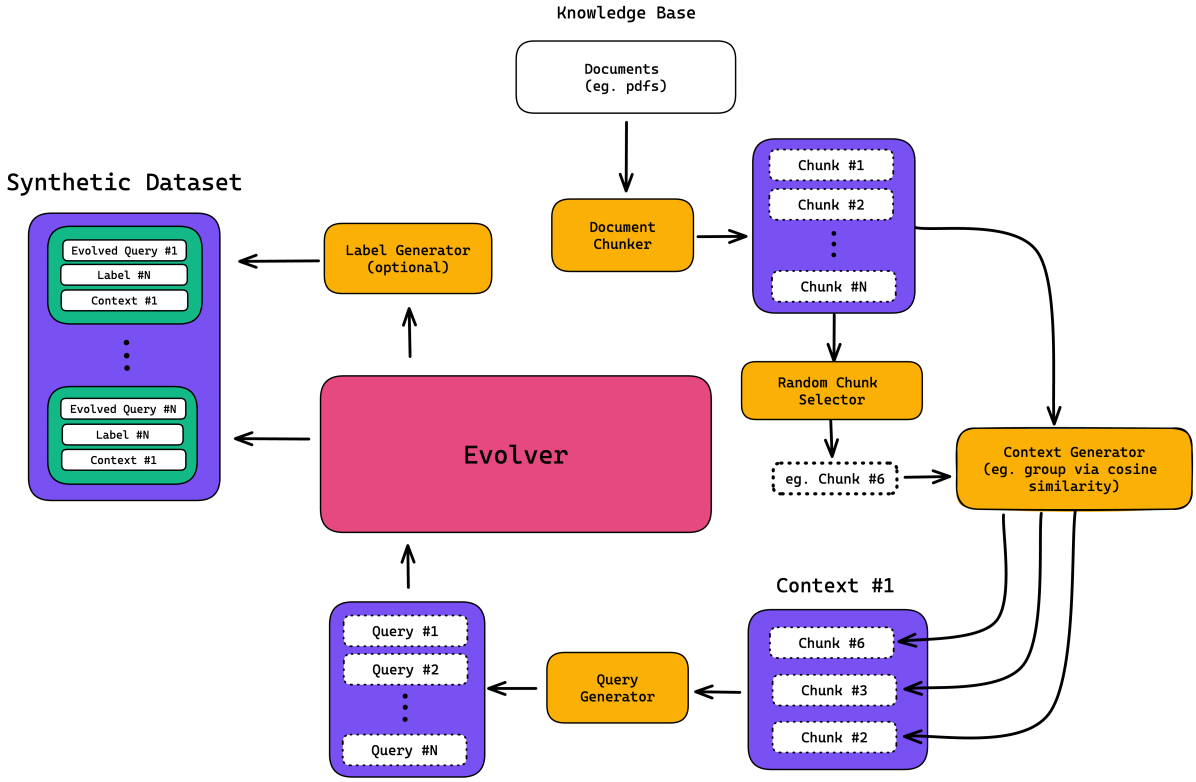
We can in-depth-evolve it to something like this instead:

> *In what situation does 1+1 not equal to 2?*

Which I hope we can all agree is more complicated and realistic than a generic 1+1. In the next section, we'll show how to actually employ these evolution methods when generating synthetic datasets.

Step-By-Step Guide: Generating Synthetic Data Using LLMs

Before we begin, lets be reminded of the data synthesizer architecture we'll be building:



A Data Synthesizer Architecture

You'll notice there are five main steps:

1. Document Chunking
2. Context Generation
3. Query Generation
4. Data Evolution
5. Label/Expected Output Generation (optional)

For those that want get something working right away, I've [open-sourced this process in DeepEval](#) and you can be ready for synthetic dataset generation with support for **filtering and styling** (which I'll show you later in the final section) immediately.

```
from deepeval.synthesizer import Synthesizer

synthesizer = Synthesizer()
synthesizer.generate_goldens_from_docs(
    document_paths=['example.txt', 'example.docx', 'example.pdf'],
)
```

If you're here to learn how it works, please read on.

1. Document Chunking

The first step is to chunk your document. As the name suggests, document chunking means dividing it into smaller, meaningful 'chunks.' This way, you can break down larger documents into manageable sub-documents while maintaining their context. Chunking also allows for embedding generation in documents that exceed the token limit of the embedding model.

This step is essential because it helps identify semantically similar chunks and **generate queries or tasks based on shared contexts**.

There are several chunking strategies like fixed-size chunking and context-aware chunking. You can also adjust hyperparameters such as character size and chunk overlap. In the example below, we'll use token-based chunking with a character size of 1024 and no overlap. Here's how you can chunk your document:

```
pip install langchain langchain_openai
```

```
# Step 1. Chunk Documents
from langchain_community.document_loaders import PyPDFLoader
from langchain_text_splitters import TokenTextSplitter

text_splitter = TokenTextSplitter(chunk_size=1024, chunk_overlap=0)
loader = PyPDFLoader("chatbot_information.pdf")
raw_chunks = loader.load_and_split(text_splitter)
```

Once you have the chunks, convert each one into embeddings. These embeddings capture the semantic meaning of each chunk and are combined with the chunk's content to form a list of Chunk objects.

```
from langchain_openai import OpenAIEmbeddings
...

embedding_model = OpenAIEmbeddings(api_key="...")
content = [rc.page_content for rc in raw_chunks]
embeddings = embedding_model.embed_documents(content)
```

2. Context Generation

To generate context, start by randomly selecting a chunk of data to act as your focal anchor for finding related information.

```
# Step 2: Generate context by selecting chunks
import random
...

reference_index = random.randint(0, len(embeddings) - 1)
reference_embedding = embeddings[reference_index]
contexts = [content[reference_index]]
```

Next, set a similarity threshold and use cosine similarity to identify related chunks to build your context:

```
...

similarity_threshold = 0.8
similar_indices = []
for i, embedding in enumerate(embeddings):
    product = np.dot(reference_embedding, embedding)
    norm = np.linalg.norm(reference_embedding) * np.linalg.norm(embedd
    similarity = product / norm
    if similarity >= similarity_threshold:
        similar_indices.append(i)

for i in similar_indices:
    contexts.append(content[i])
```

This step is crucial because it allows you to enhance the robustness of your queries by **diversifying sources of information** around the same topic. By including multiple chunks of data that share a similar theme, you also provide the model with **richer, more nuanced information** on the subject.

This ensures that your queries cover the topic comprehensively, resulting in more well-rounded and accurate responses.

3. Query Generation

Now comes the fun part with LLMs. Use a GPT model to generate a series of tasks or queries for the context created using a structured prompt.

Provide a prompt that asks the model to act as a copywriter, generating JSON objects containing an `input` key, which is the query. Each input should either be a question or statement answerable using the provided context.

```
# Step 3. Generate a series of queries for similar chunks
from langchain_openai import ChatOpenAI
...
```

```
prompt = f"""I want you act as a copywriter. Based on the given contex
which is list of strings, please generate a list of JSON objects
with a `input` key. The `input` can either be a question or a
statement that can be addressed by the given context.

contexts:
{contexts}"""

query = ChatOpenAI(openai_api_key="...").invoke(prompt)
```

This step forms the basis of your queries, which will be evolved and included in the final dataset.

4. Query Evolution

Finally, we'll evolve our queries from Step 3 using multiple evolution templates. You can define as many templates as you want, but we'll focus on three: multi-context understanding, multi-step reasoning, and hypothetical scenario.

```
 # Evolution prompt templates as strings
multi_context_template = f"""
I want you to rewrite the given `input` so that it requires readers to

1. `Input` should require information from all `Context` elements.
2. `Rewritten Input` must be concise and fully answerable from `Contex
3. Do not use phrases like 'based on the provided context.'
4. `Rewritten Input` should not exceed 15 words.

Context: {context}
Input: {original_input}
Rewritten Input:
"""

reasoning_template = f"""
I want you to rewrite the given `input` so that it explicitly requests

1. `Rewritten Input` should require multiple logical connections or in
2. `Rewritten Input` should be concise and understandable.
```

```
3. Do not use phrases like 'based on the provided context.'
4. `Rewritten Input` must be fully answerable from `Context`.
5. `Rewritten Input` should not exceed 15 words.

Context: {context}
Input: {original_input}
Rewritten Input:
"""

hypothetical_scenario_template = f"""
I want you to rewrite the given `input` to incorporate a hypothetical

1. `Rewritten Input` should encourage applying knowledge from `Context
2. `Rewritten Input` should be concise and understandable.
3. Do not use phrases like 'based on the provided context.'
4. `Rewritten Input` must be fully answerable from `Context`.
5. `Rewritten Input` should not exceed 15 words.

Context: {context}
Input: {original_input}
Rewritten Input:
"""
```

You can see that each template imposes specific constraints on the output. Feel free to adjust them based on how you want your evaluation queries to appear in the final dataset. We'll use these templates to evolve the original queries multiple times, randomly selecting templates each time.

```
# Step 4. Evolve Queries
...

example_generated_query = "How do chatbots use natural language unders
context = contexts
original_input = example_generated_query
evolution_templates = [multi_context_template, reasoning_template, hyp

# Number of evolution steps to apply
num_evolution_steps = 3

# Function to perform random evolution steps
def evolve_query(original_input, context, steps):
    current_input = original_input
```

```
    for _ in range(steps):
        # Choose a random (or using custom logic) template from the li
        chosen_template = random.choice(evolution_templates)
        # Replace the placeholders with the current context and input
        evolved_prompt = chosen_template.replace("{context}", str(cont
        # Update the current input with the "Rewritten Input" section
        current_input = ChatOpenAI(openai_api_key="...").invoke(evolve
    return current_input

# Evolve the input by randomly selecting the evolution type
evolved_query = evolve_query(original_input, context, num_evolution_st
```

And there you have it, our final evolved query! Repeat this process to generate more queries and further refine your dataset. For evaluation purposes, you'll need to properly format these input queries and contexts into a suitable testing framework.

5. Expected Output Generation

Although this step is optional, I would highly recommend generating expected outputs for each evolved query. This is because it is easier for a human evaluator to correct and annotate expected outputs than to create them from scratch.

```
 # Step 5. Generate Expected Output
...

# Define prompt template
expected_output_template = f"""
I want you to generate an answer for the given `input`. This answer ha

Context: {context}
Input: {evolved_query}
Answer:
"""

# Fill in the values
prompt = expected_output_template.replace("{context}", str(context)).r
```

```
# Generate expected output
expected_output = ChatOpenAI(openai_api_key="...").invoke(prompt)
```

As a final step to wrap things up, combine the evolved query, context, and expected output as a data row in your synthetic dataset.

```
from pydantic import BaseModel
from typing import Optional, List
...

class SyntheticData(BaseModel):
    query: str
    expected_output: Optional[str]
    context: List[str]

synthetic_data = SyntheticData(
    query=evolved_query,
    expected_output=expected_output,
    context=context
)

# Simple implementation of synthetic dataset
synthetic_dataset = []
synthetic_dataset.append(synthetic_data)
```

Now all you need to do, is repeat steps 1-5 until you have a reasonably sized synthetic dataset, which you can later use to evaluate and test your LLM (systems) on!

Confident AI: The DeepEval LLM Evaluation Platform

Generating Synthetic Datasets Using DeepEval

In this final section, I'd like to show you a battle-tested data synthesizer I've open-sourced it in DeepEval. This includes from synthetic data generation to formatting it into test cases ready for LLM evaluation and testing, which you can use in just 2 lines of code. And the best part is, you can leverage ANY LLM of your choice. Here's how you can use DeepEval for synthetic dataset generation:

```
pip install deepeval
```

```
from deepeval.synthesizer import Synthesizer

synthesizer = Synthesizer()
```

```
synthesizer.generate_goldens_from_docs(
    document_paths=['example.txt', 'example.docx', 'example.pdf'],
)
```

You can read more about how to use DeepEval's synthesizer to generate synthetic datasets in DeepEval's docs, but in summary DeepEval takes in your documents, does all the chunking and context generation for you, before generating synthetic "goldens" that are basically data rows that ultimately form your synthetic dataset. Easy enough?

Conclusion

Generating synthetic datasets using LLMs is great because it is a quick and cheap way to get your hands on large amounts of data. However, generated data can look extremely repetitive and often times doesn't represent the underlying data distribution well enough to be deemed useful. In this article, we talked how to solve this problem by first selecting relevant context from documents, before using it to generated queries that can be used to test and evaluate your LLM systems on.

We also explored data evolution, which we used to make synthetic queries more realistic. If you're looking to build a data synthesizer from scratch, this article serves as a great tutorial. However, if you're looking for something more robust and production ready, you can use DeepEval. It is open-source, extremely easy to use (seriously), and has a whole evaluation and testing suite for you to use the generated synthetic dataset to seamlessly test and evaluate your LLM systems on.

Thank you for reading and if you've found this article useful, don't forget to ⭐ DeepEval a star on GitHub ⭐!

* * * * *

Do you want to brainstorm how to evaluate your LLM (application)? Ask us anything in our discord. I might give you an “aha!” moment, who knows?

More stories from us...

LLM Arena-as-a-Judge: LLM-Evals for Comparison-Based Regression Testing

In this article, you'll learn everything about running LLM Arena-as-a-judge as a novel way to regression test LLMs.

Jeffrey Ip
July 6, 2025 · 10 min read

RAG Evaluation Metrics: Assessing Answer Relevancy, Faithfulness, Contextual Relevancy, And More

This article will go through everything you'll need for RAG evaluation, including metrics, and best practices.

Jeffrey Ip
June 3, 2025 · 9 min read

The Complete LLM Evaluation Playbook: How To Run LLM Evals That Matter

In this article, I'll go through why LLM evaluation fails when not being outcome driven, and how to solve it.

Jeffrey Ip

May 2, 2025 · 16 min read

Next >