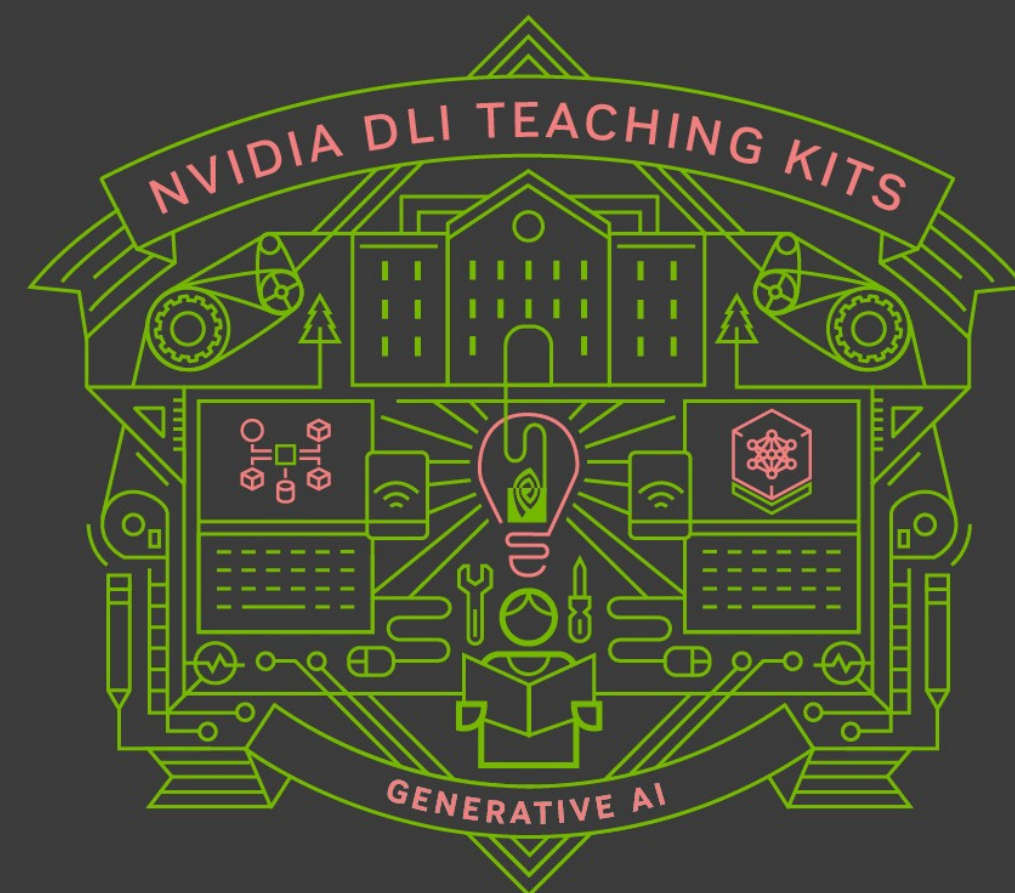




# Lecture 8.2 – LLM Compound Systems

Generative AI Teaching Kit





The NVIDIA Deep Learning Institute Generative AI Teaching Kit is licensed by NVIDIA and Dartmouth College under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

# This lecture

- Recap on Chains and LangChain
- Retrieval Augmented Generation
- Popular RAG Libraries: LlamaIndex and LangChain
- Compound systems as programs: DSPy
- Evaluation of Compound LLM Systems

# Recap on Chains and LangChain

## Chaining Logic

# Review on LLM Chains

Last time we saw that working with LLMs hinges on the prompts that we utilize

These can be from two types:

## Prompt Templates

- **Good prompts** are specific, focused, and clear, providing enough context for the LLM to generate a relevant and coherent response.
- **Bad prompts** are overly broad, vague, or based on incorrect premises, leading to potentially irrelevant, incorrect, or confusing responses.

## System Prompts

- A special instruction provided to a language model (LLM) or other AI system that guides its behavior, tone, or output style throughout the interaction.
- Unlike regular user prompts, which are dynamic and change based on user input, system prompts are generally static and set by developers or users at the start of a session. They serve as a framework within which the AI operates.

# System Prompts

These special, static, prompts are pre-pended to the user prompts and the models are trained to obey these system prompts.

- **Guidance:** It sets the rules or guidelines for how the AI should respond to user inputs. This could include instructions on maintaining a certain tone (e.g., formal, casual), following specific ethical guidelines, or limiting responses to certain types of information.
- **Context:** It provides the AI with context about the interaction. For example, the system prompt might indicate that the AI should behave as a customer service representative or as a tutor in a specific subject.
- **Consistency:** It ensures that the AI's responses are consistent throughout the interaction, adhering to the predefined rules or context.

## Example of a System Prompt

Let's say you're designing an AI assistant for a customer service chatbot. The system prompt might look something like this:

### System Prompt:

*"You are a helpful and polite customer service assistant. Always address customers respectfully, provide clear and concise answers, and offer additional help when possible. Avoid technical jargon and ensure the customer feels valued. If you cannot assist directly, offer to escalate the issue to a human representative."*

```
You are ChatGPT, a large language model trained by OpenAI, based on the GPT-4 architecture.
Knowledge cutoff: 2023-10
Current date: 2024-08-12

Image input capabilities: Enabled
Personality: v2

# Tools

## bio

The bio tool allows you to persist information across conversations. Address your message to=bio and write whatever information you want to remember. The information will appear in the model set context below in future conversations.

## dalle

// Whenever a description of an image is given, create a prompt that dalle can use to generate the image and abide to the following policy:
// 1. The prompt must be in English. Translate to English if needed.
// 2. DO NOT ask for permission to generate the image, just do it!
// 3. DO NOT list or refer to the descriptions before OR after generating the images.
// 4. Do not create more than 1 image, even if the user requests more.
// 5. Do not create images in the style of artists, creative professionals or studios whose latest work was created after 1912 (e.g. Picasso, Kahlo).
// - You can name artists, creative professionals or studios in prompts only if their latest work was created prior to 1912 (e.g. Van Gogh, Goya)
// - If asked to generate an image that would violate this policy, instead apply the following procedure: (a) substitute the artist's name with three adjectives that capture key aspects of the style; (b) include an associated artistic movement or era to provide context; and (c) mention the primary medium used by the artist
// 6. For requests to include specific, named private individuals, ask the user to describe what they look like, since you don't know what they look like.
// 7. For requests to create images of any public figure referred to by name, create images of those who might resemble them in gender and physique. But they shouldn't look like them. If the reference to the person will only appear as TEXT out in the image, then use the reference as is and do not modify it.
// 8. Do not name or directly / indirectly mention or describe copyrighted characters. Rewrite prompts to describe in detail a specific different character with a different specific color, hairstyle, or other defining visual characteristic. Do not discuss copyright policies in responses.
// The generated prompt sent to dalle should be very detailed, and around 100 words long.
// Example dalle invocation:
// {
//   "prompt": "<insert prompt here>"
// }
```

Part of ChatGPT's system prompt



# LLM Chains - Recap

LLM chains can be thought of as a way build structure and allow for flexibility.

The simplest chain consists of:

1. An input prompt template
2. An LLM
3. (optional) An output parser

Once created, a user can simply interact with the chain object, passing in the relevant information and receiving the desired output

```
# Step 1: Define the Input Template
input_template = PromptTemplate(
    template="Question: {question}\nAnswer:", input_variables=["question"])

# Step 2: Initialize the LLM (e.g., using OpenAI's GPT)
llm = OpenAI(model="text-davinci-003")

# Step 3: Define the Output Parser
# Example: Extracting a simple numerical answer from the LLM's response
# using regex
output_parser = RegexParser(
    pattern="Answer:\s*(\d+)", output_key="parsed_answer")

# Step 4: Create the LLM Chain
llm_chain = LLMChain(
    llm=llm, prompt=input_template, output_parser=output_parser)

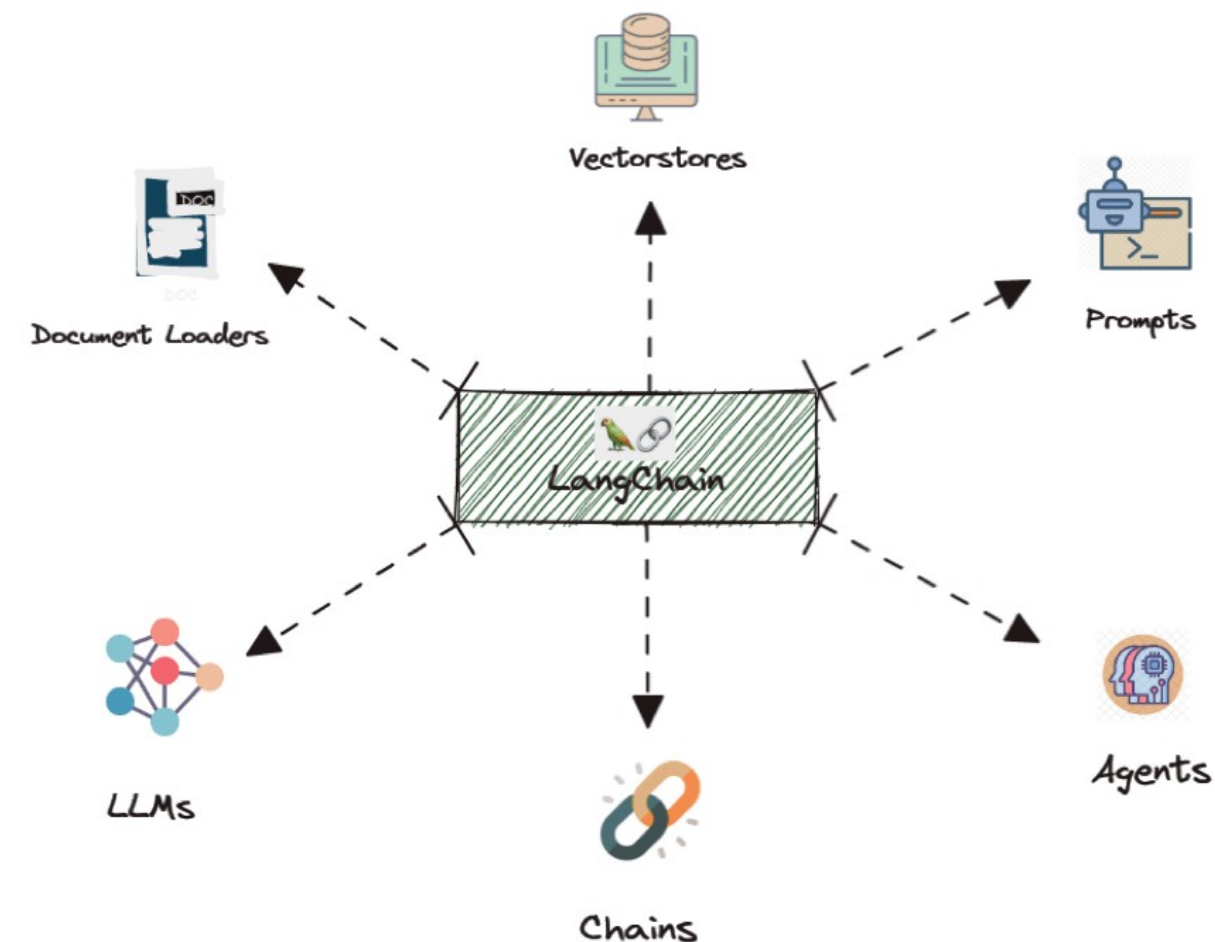
# Step 5: Define the Sequential Chain
sequential_chain = SimpleSequentialChain(chains=[llm_chain], verbose=True)
```

```
# Step 6: Run the Sequential Chain with an Example Input
input_data = {"question": "What is 2 + 2?"}
output = sequential_chain.run(input_data)
```

# Combining LLMs with other components

With chains, we can connect more than just prompts to LLMs

We can connect data sources, other LLMs, compute environments, basically anything that has a digital interface can be connected using libraires like LangChain





# Retrieval Augmented Generation

Adding the right data

# Augmenting Data for our Prompts

Last time we saw that improving prompts improves LLM performance. What if we not only improve the prompt, but add more data?

Why would we want to add extra data:

## Reduce Hallucination

- Focus the model to rely on in-context content

## Not enough data for fine-tuning

- May only have a few samples, but are needed for good model alignment with the task

## Privacy

- We don't want our sensitive data sent to external services

## Up-to date information

- Even if we have enough data, we might need timely/fresh information that we cannot train on everyday



# How to find Data? Similarities and Embeddings

Typically, the data we want to add will be text. How do we "search" for the right content to add?

## Converting Documents to Vectors

We can leverage the same idea behind word embedding vectors to embed text chunks. This will give us a numerical entity we can search against

### 1. Chunking Documents

We will take the raw text of our data that we want stored to search for and "chunk" it into pieces.

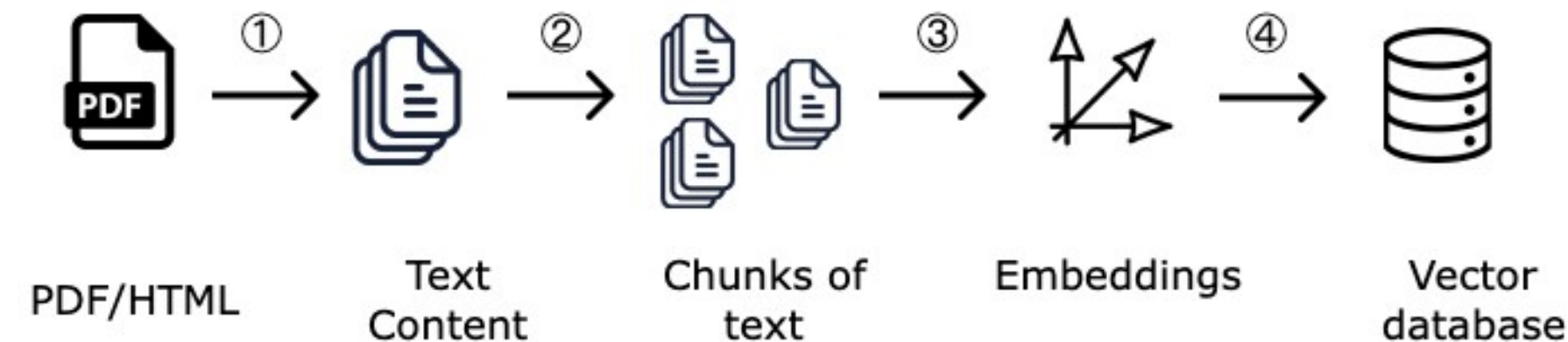
Note: We chunk so that our searching can have some finer resolution. If we only had one chunk, we'd need to average out the whole document which can reduce search performance, and means we have to add the entire document to our prompt.

### 2. Generating Embeddings from Chunks

Using an encoder-type LLM, such as BERT or variants, we can generate a high dimensional representation of the text chunk.

### 3. Vector Database

We store the text chunk embedding vectors in a new database, a vector database where we can search for these text chunks.

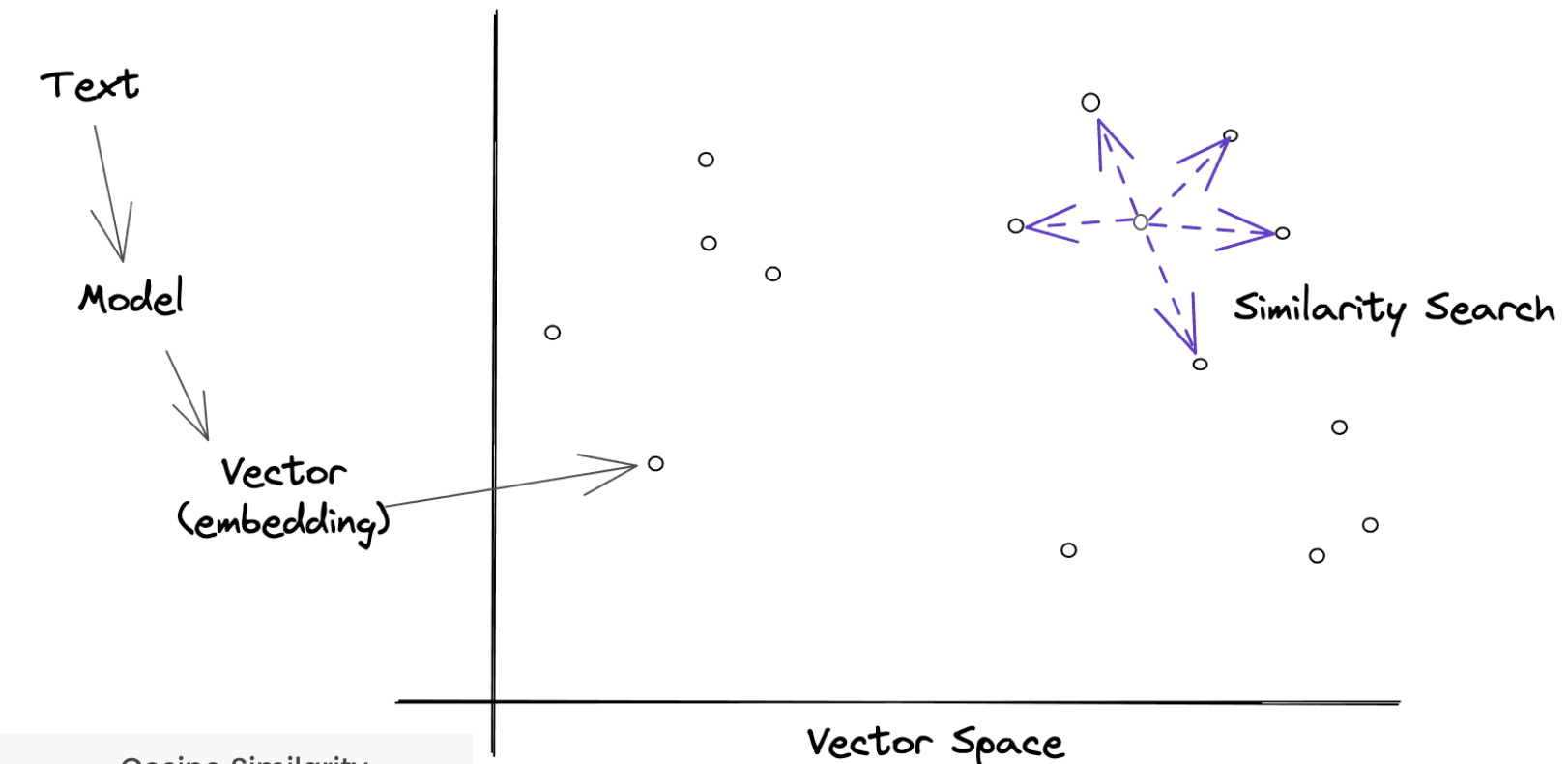


# Vector Searching

Now that we have our document chunks in vector form, how do we use them?

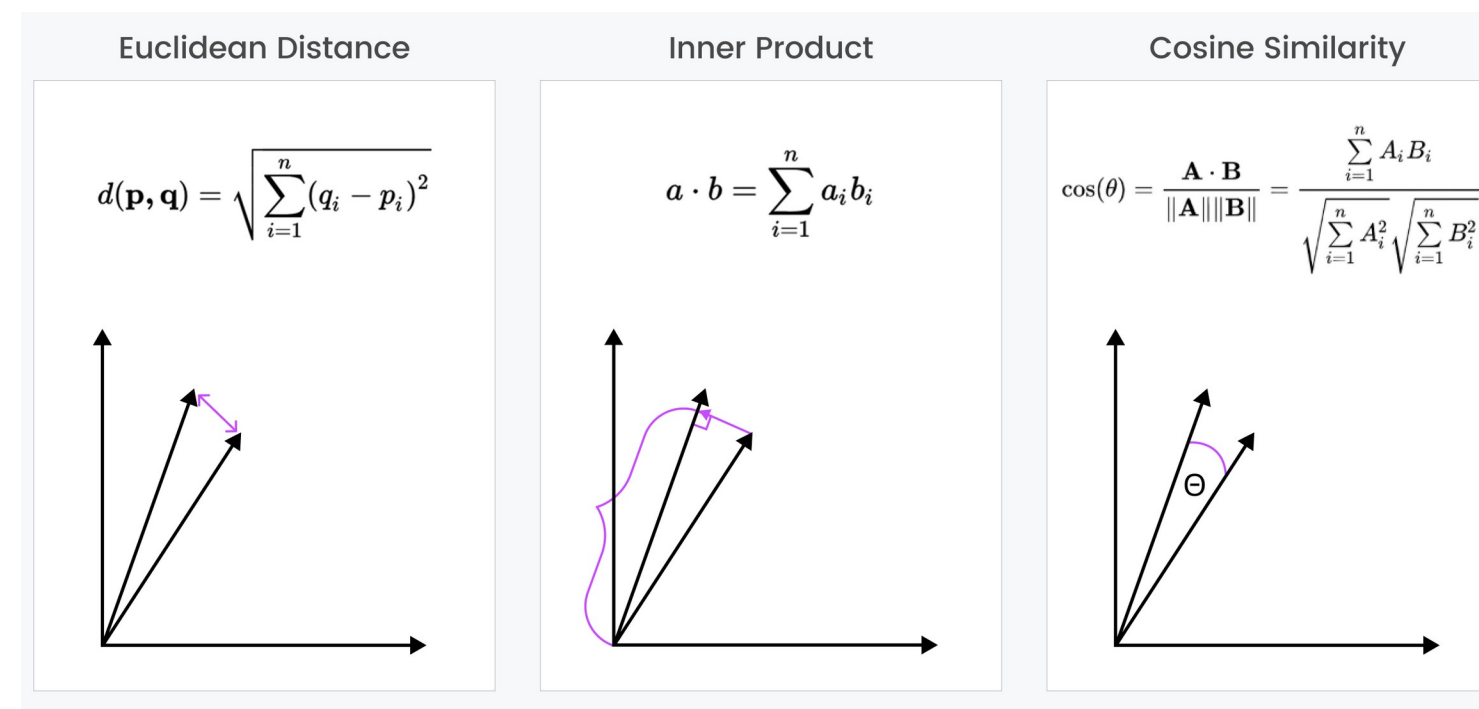
## Semantic Search: Finding similar embeddings

- Let's assume the user has asked some question
- We can vectorize this question and use that to find similar content
- That content should provide information to answer the question



## Similarity Searching Methods

- Euclidean Distance
- Inner Product
- Cosine Similarity



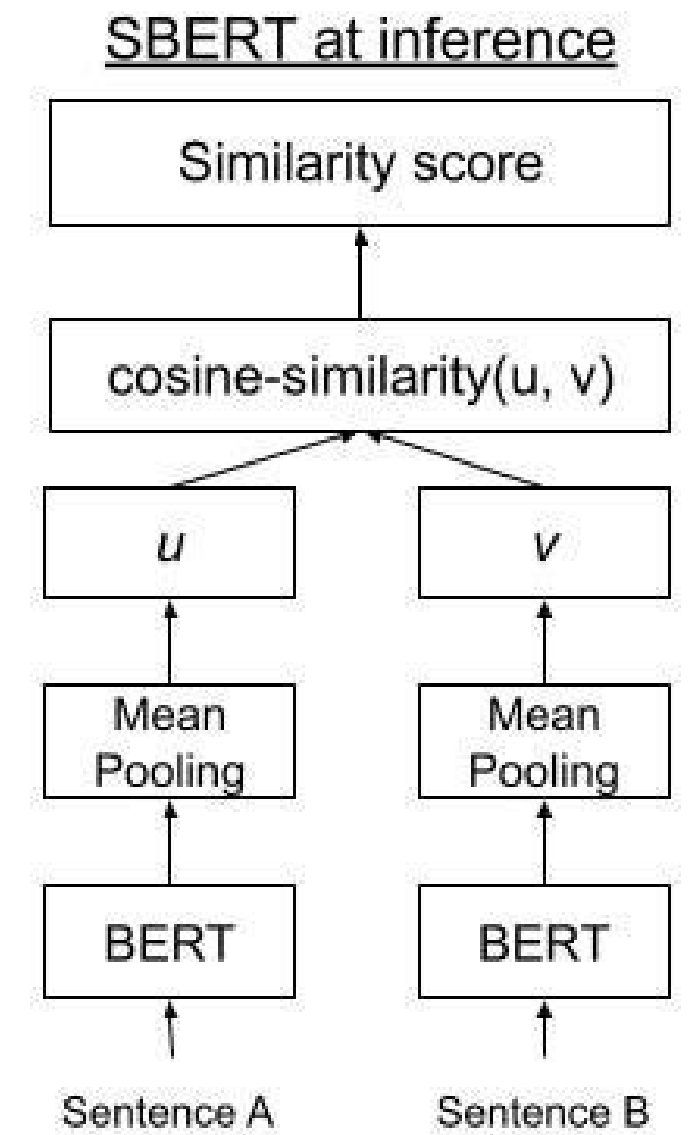
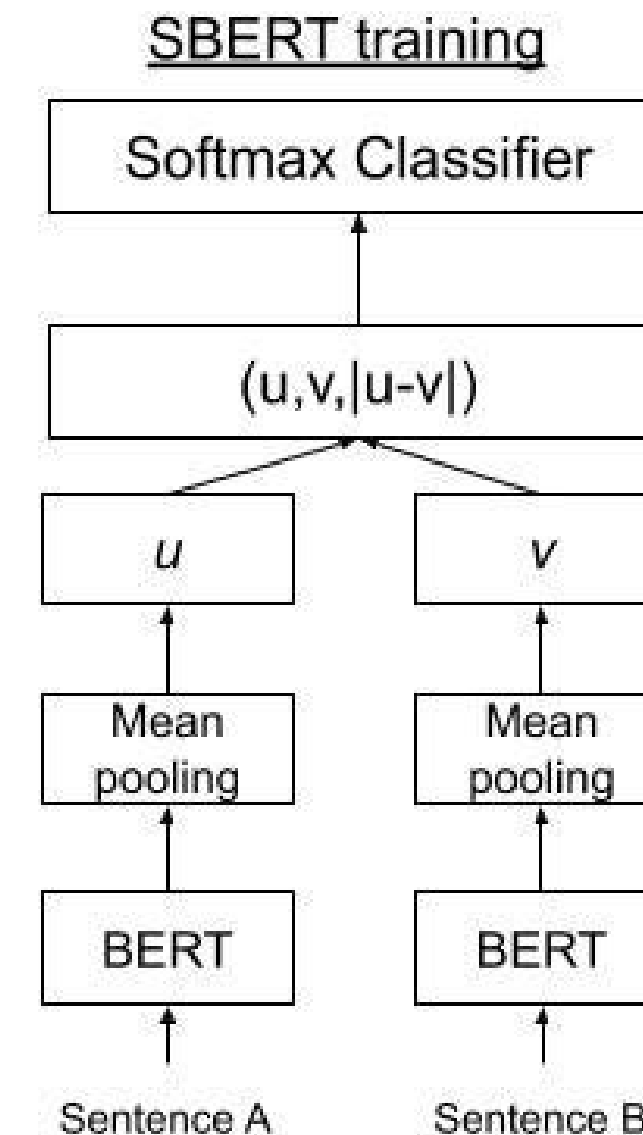
# Using Vector Search and LLMs

For RAG, we need to be able to find the relevant information quickly.

Once we have our users' query, we need to quickly find the relevant information in our vector database.

Sentence Transformers like SBERT allow us to efficiently find which vectors in our database have high similarity scores.

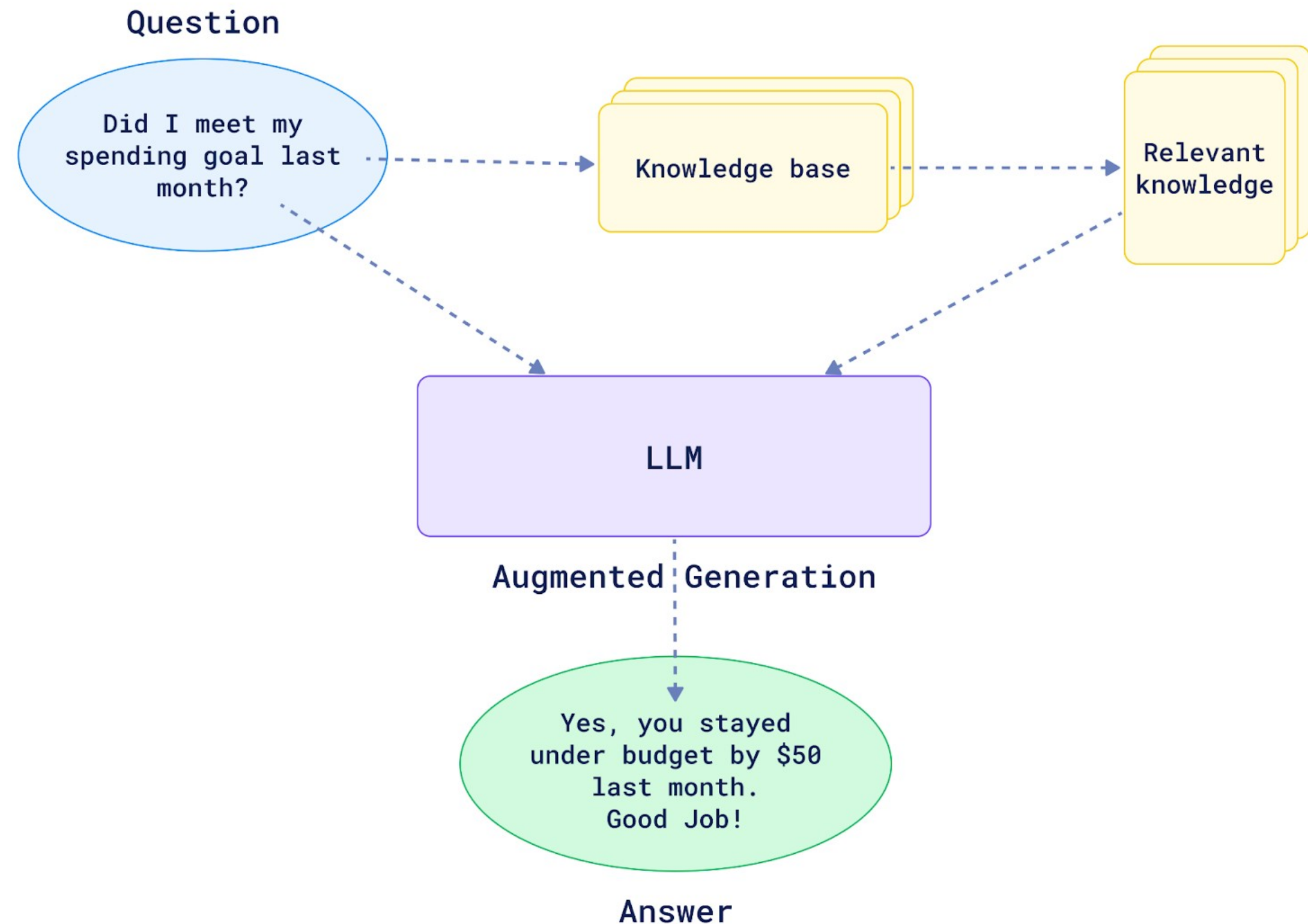
During training two sentences are passed in and the model is trained to compare the embedded vectors so that similar semantic pairs are given a higher score and dissimilar pairs a lower score.



# Retrieval Augmented Generation

The full RAG pipeline:

1. Collect user's query
2. Search knowledge base for relevant context based on user query
3. Combine user query with retrieved data from knowledge base into a prompt template
4. Pass enriched prompt to LLM with both user query and the relevant context
5. Generate output from LLM





# Factors that can Affect RAG

## Quality of Embeddings:

The effectiveness of the sentence transformers used to convert queries and documents into vector embeddings, which impacts the accuracy of information retrieval.

## Relevance of Document Corpus:

The quality, relevance, and diversity of the documents in the database, which determine the usefulness of the retrieved information.

## Efficiency of Similarity Search:

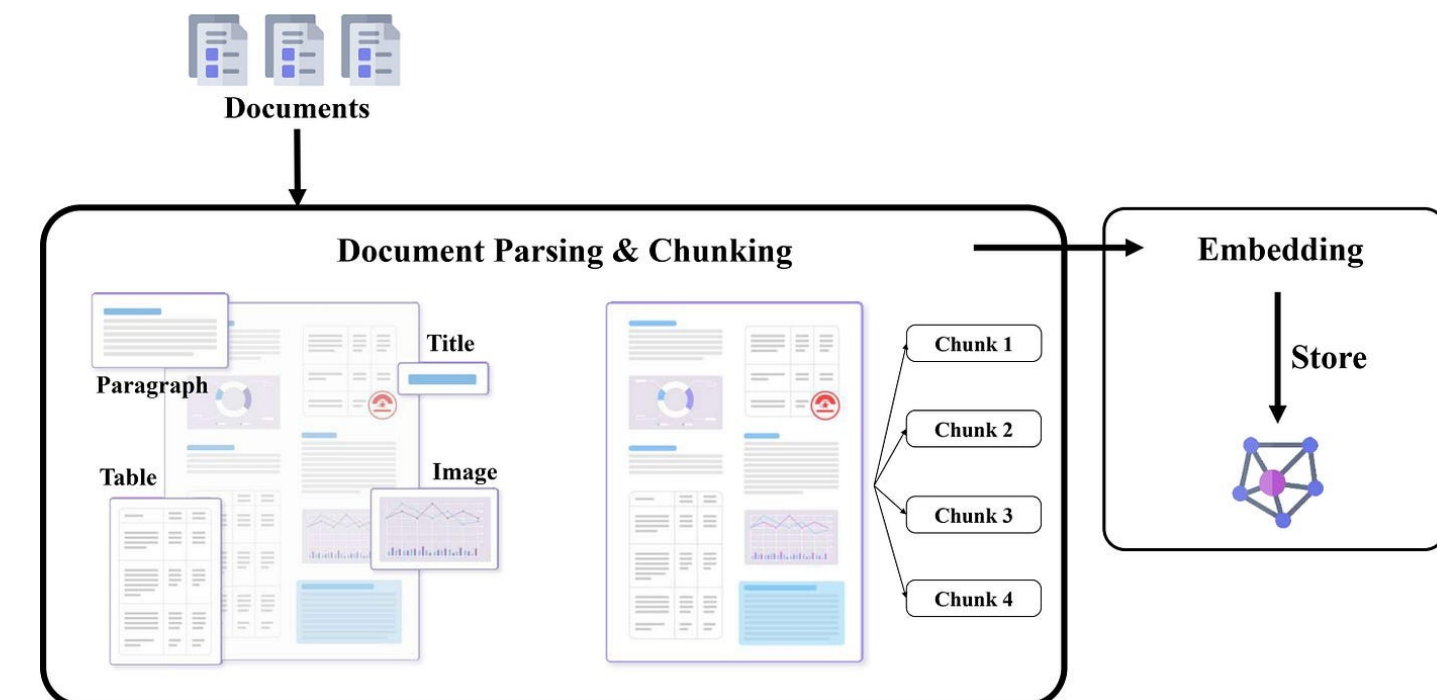
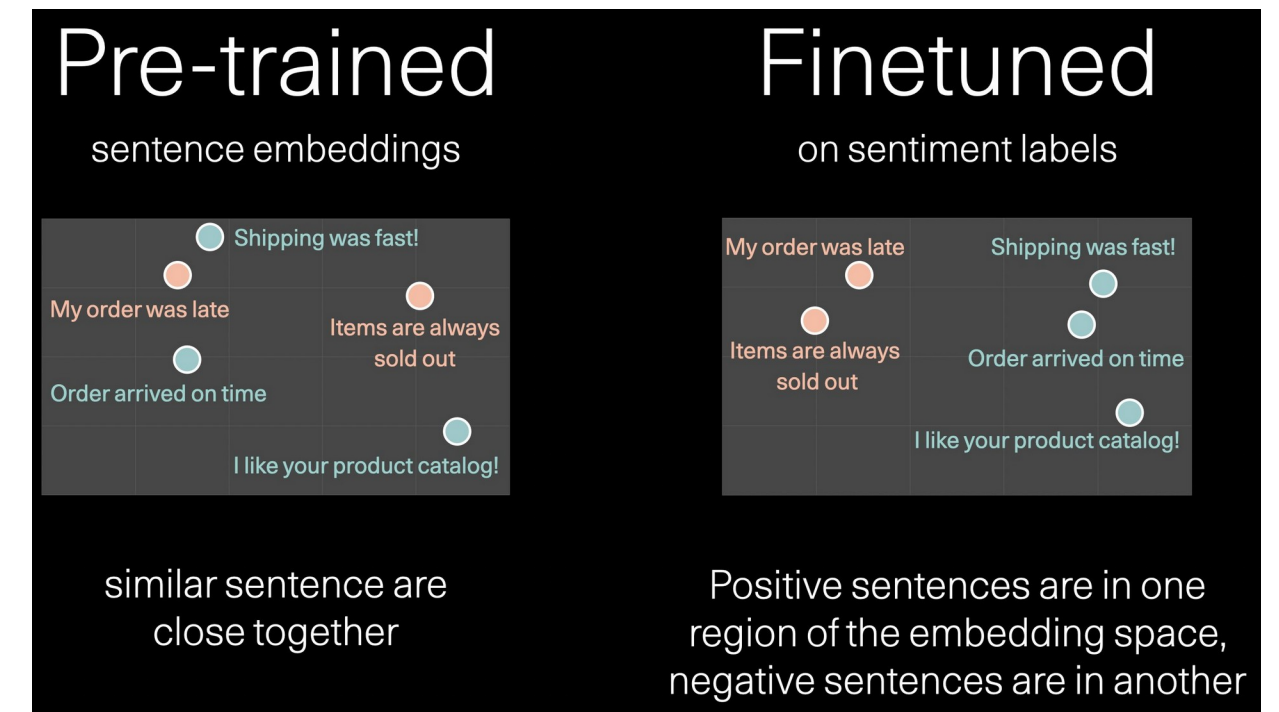
The algorithm used to search for similar vectors in the database, influencing both the speed and accuracy of the retrieval process.

## Model Fine-Tuning:

The extent to which the retrieval and generative models are fine-tuned on domain-specific data, which enhances the relevance and accuracy of responses.

## System Latency and Speed:

The overall speed at which the system processes queries, retrieves relevant documents, and generates responses, affecting the user experience.



# LlamaIndex and LangChain

The tools we build with

# Pure Python vs. Custom Libraries

Development of LLM applications is typically done using the Python language

Vanilla Python can be used, but many pieces need to be built each time

Libraries like LangChain and LlamaIndex have been used to quickly modularize and speed up development of LLM applications

```
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain

def langchain_approach(context: str, question: str) -> str:
    llm = OpenAI(temperature=0)
    template = """
    Context: {context}

    Question: {question}

    Answer: """
    prompt = PromptTemplate(template=template, input_variables=["context", "question"])
    chain = LLMChain(llm=llm, prompt=prompt)
    return chain.run(context=context, question=question)
```

## RAG with LangChain

```
from llama_index import GPTSimpleVectorIndex, Document
from llama_index.readers.string import StringReader

def llamaindex_approach(context: str, question: str) -> str:
    documents = StringReader().load_data(text=context)
    index = GPTSimpleVectorIndex.from_documents(documents)
    return str(index.query(question))
```

## RAG with LlamaIndex

```
class OpenAIHandler:
    def __init__(self, model: str = "text-davinci-002", max_tokens: int = 100):
        self.model = model
        self.max_tokens = max_tokens
        self.encoding = tiktoken.encoding_for_model(model)

    def count_tokens(self, text: str) -> int:
        return len(self.encoding.encode(text))

    def construct_prompt(self, context: str, question: str) -> str:
        return f"Context: {context}\n\nQuestion: {question}\n\nAnswer:"

    def call_api(self, prompt: str, max_retries: int = 3, backoff_factor: float = 2) -> str:
        for attempt in range(max_retries):
            try:
                response = openai.Completion.create(
                    engine=self.model,
                    prompt=prompt,
                    temperature=0,
                    max_tokens=self.max_tokens
                )
                return response.choices[0].text.strip()
            except openai.error.RateLimitError:
                if attempt < max_retries - 1:
                    sleep_time = backoff_factor ** attempt
                    logger.warning(f"Rate limit reached. Retrying in {sleep_time} seconds...")
                    time.sleep(sleep_time)
            else:
                raise

        except openai.error.OpenAIError as e:
            logger.error(f"OpenAI API error: {str(e)}")
            raise

    def get_answer(self, context: str, question: str) -> Dict[str, Any]:
        prompt = self.construct_prompt(context, question)
        token_count = self.count_tokens(prompt)
        logger.info(f"Token count: {token_count}")

        if token_count + self.max_tokens > 4000:  # Assuming 4000 is the model's max token limit
            raise ValueError("Prompt exceeds maximum token limit")

        try:
            response = self.call_api(prompt)
            return {
                "question": question,
                "answer": response,
                "token_count": token_count
            }
        except Exception as e:
            logger.error(f"Error occurred: {str(e)}")
            return {
                "question": question,
                "error": str(e),
                "token_count": token_count
            }

def vanilla_python_approach(context: str, question: str) -> str:
    handler = OpenAIHandler()
    result = handler.get_answer(context, question)
    if "answer" in result:
        return result["answer"]
    else:
        return f"Error: {result['error']}"
```

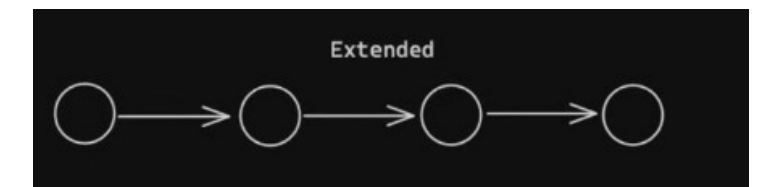
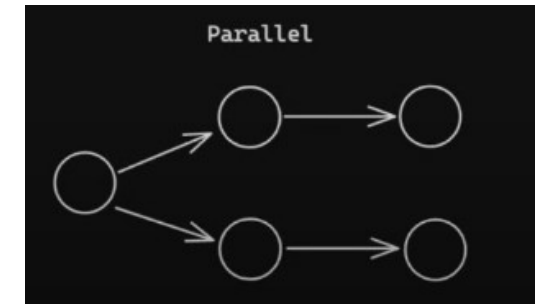
## RAG with Vanilla Python

# LangChain - Review

Previously we saw how LangChain provides a framework for working with LLM chains. We can leverage these chains to build complex LLM Compound Applications

LangChain offers the ability to use Serial and Parallel chaining structures

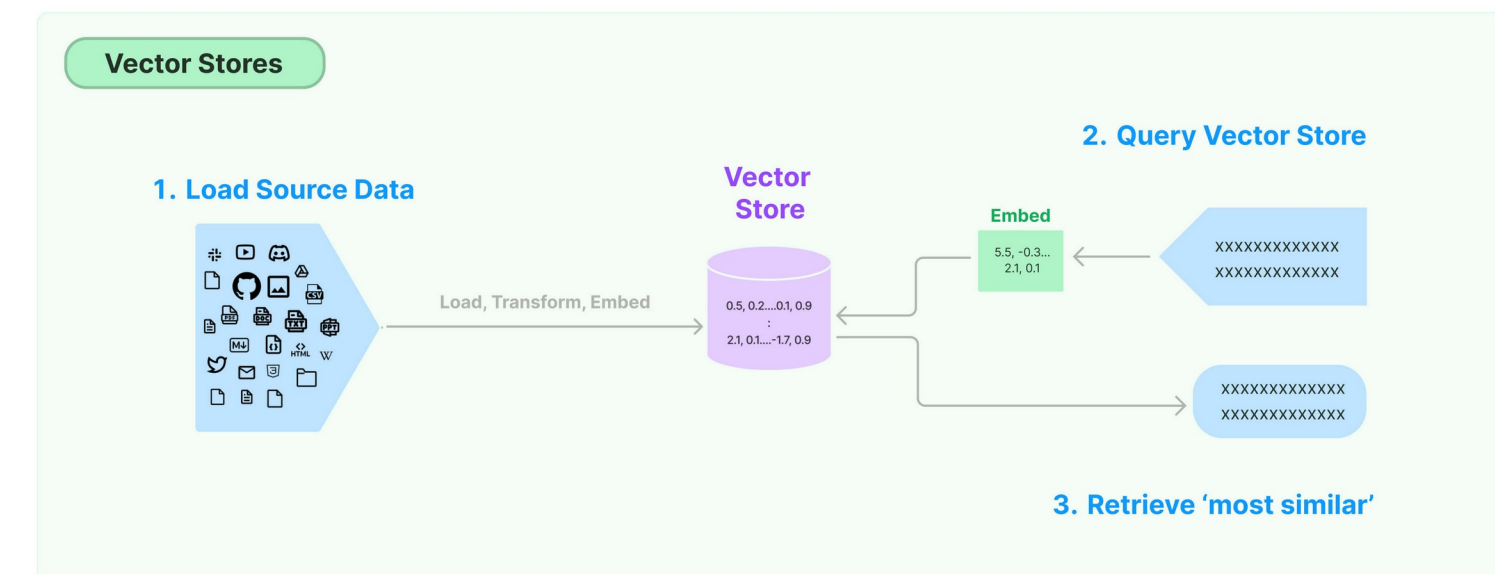
```
rag_chain = ( RunnableParallel(context = retriever | format_docs,
                               question = RunnablePassthrough()) |
             qa_prompt | llm )
```



Provides out of the box solutions to parse unstructured data with various text splitting options:

*"[The] text splitter is the recommended one for generic text. It is parameterized by a list of characters. It tries to split on them in order until the chunks are small enough. The default list is ['\n\n', '\n', ' ', '']. This has the effect of trying to keep all paragraphs (and then sentences, and then words) together as long as possible, as those would generically seem to be the strongest semantically related pieces of text."* – **LangChain Docs**

Interfaces seamlessly with many vector database providers:



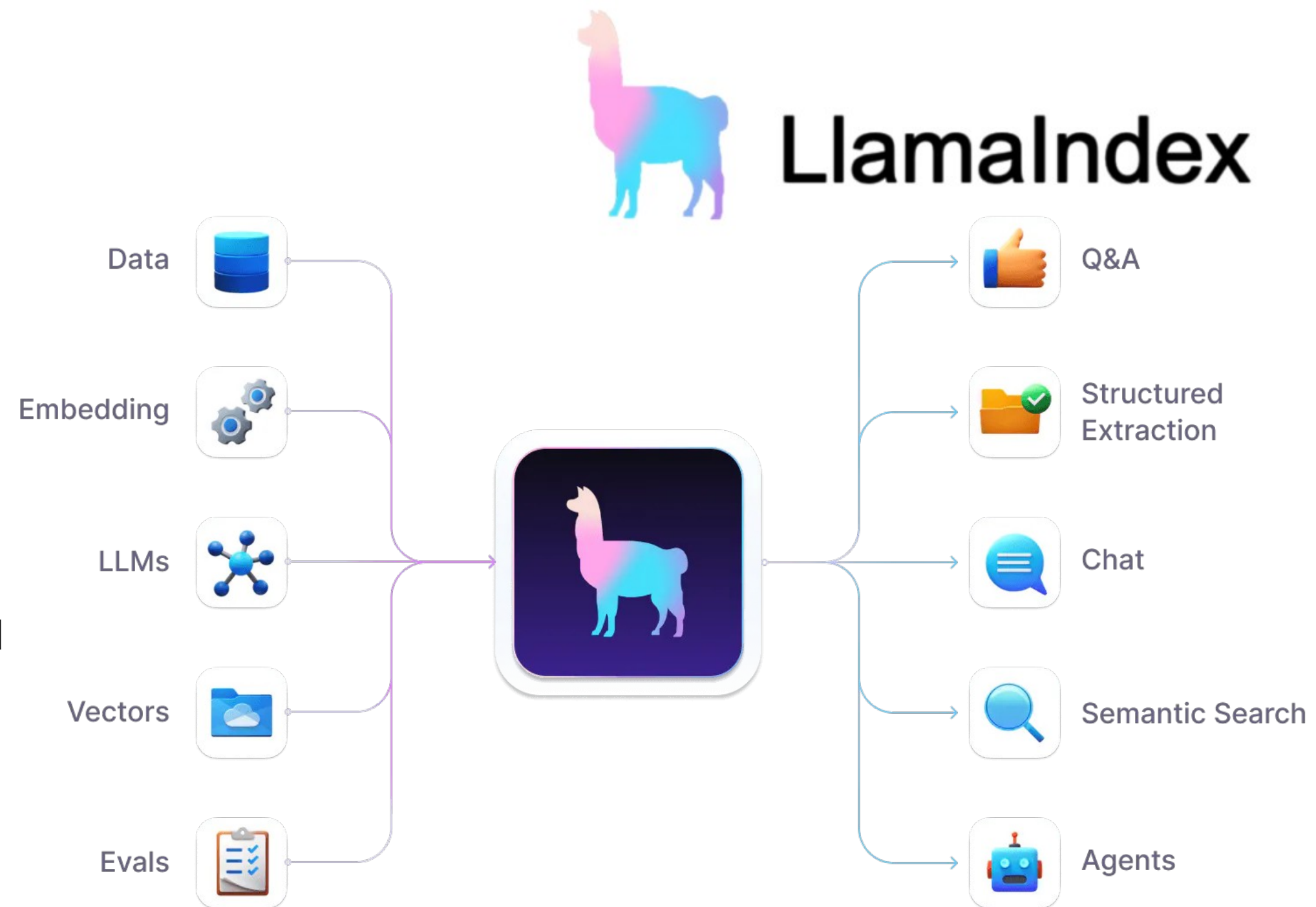


# LlamaIndex – Managing Data for RAG

## LlamaIndex vs. LangChain

Whereas LangChain's focus is on LLM components, LlamaIndex focuses more specifically on RAG and data parsing.

- LlamaIndex includes hundreds of data loaders to connect custom data sources to LLMs, including integrations with Airtable, Jira, Salesforce, and others, allowing seamless data loading from files, JSON, CSV, and unstructured data.
- LlamaIndex offers various indexing models tailored for optimizing data exploration and categorization, enabling significant performance gains when using LLMs by choosing the appropriate index type for your application.



# Data Formats and LLM Performance

There is no single best way to parse an arbitrary document:

## Document types

- PDFs
- MS Word
- Raw text
- HTML/XML pages
- JSON
- MS PowerPoint
- Markdown

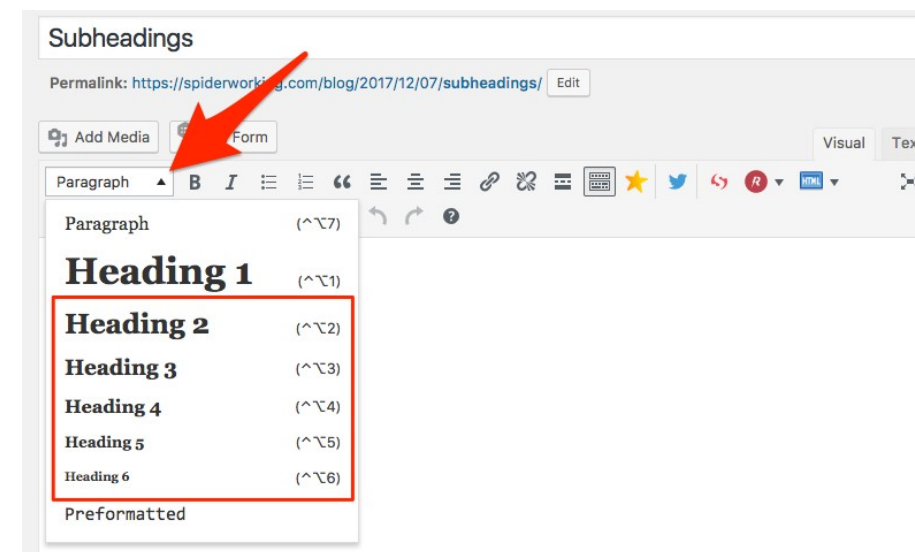


Understanding how to parse even just these requires a strong understanding of their encodings and formats, many of which are proprietary

## Formatting Differences

Even within a single document type, the semantic meaning of the document can be affected by how it is chunked:

- Newlines
- Headings/Subheadings
- Page borders





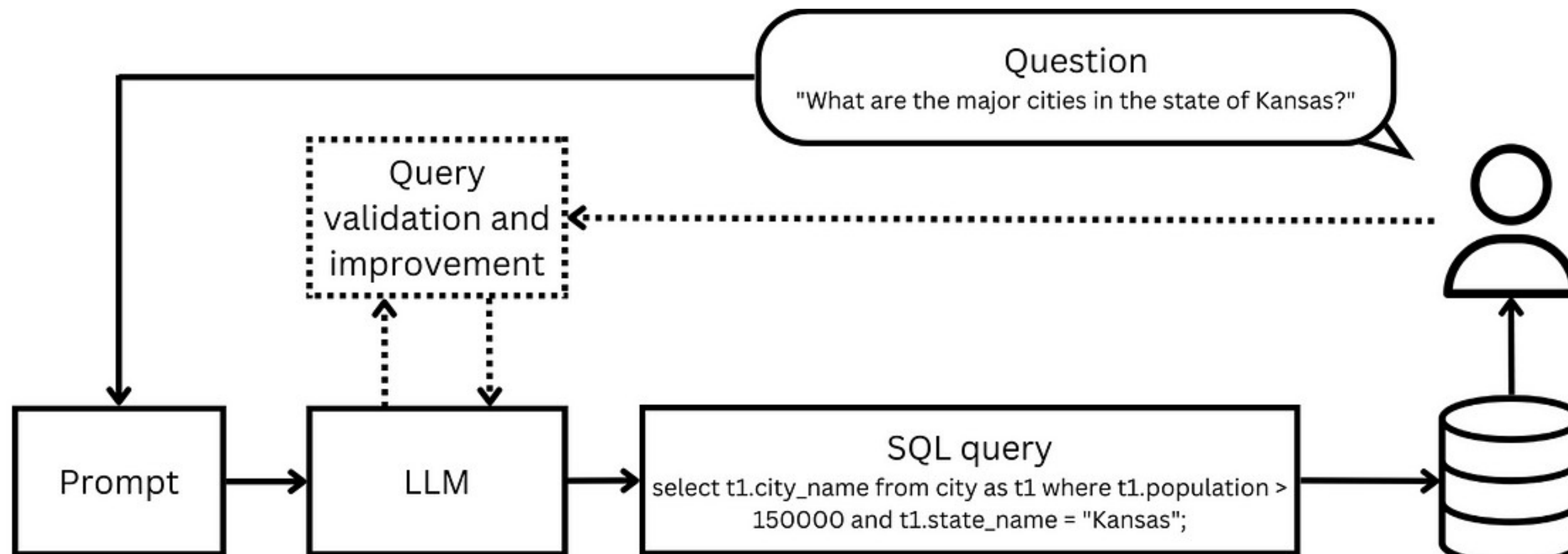
# RAG with Structured Data: SQL RAG

Unstructured data, typically text, is common in RAG. But what if we need structured data?

## Example: generate SQL queries

We can utilize an LLM to generate SQL queries from natural language to collect structured data as well.

These 'Text2SQL' models are becoming popular as developers want to include structured data with semantic data for RAG.



# Compound systems as programs: DSPy

A new look at LLM systems

# Prompt Engineering, Fine-Tuning, ... or Programming?

LLM Compound Systems benefit from the best alignment of the models being used

If we **don't have enough data**, we're stuck at **few-shot learning**

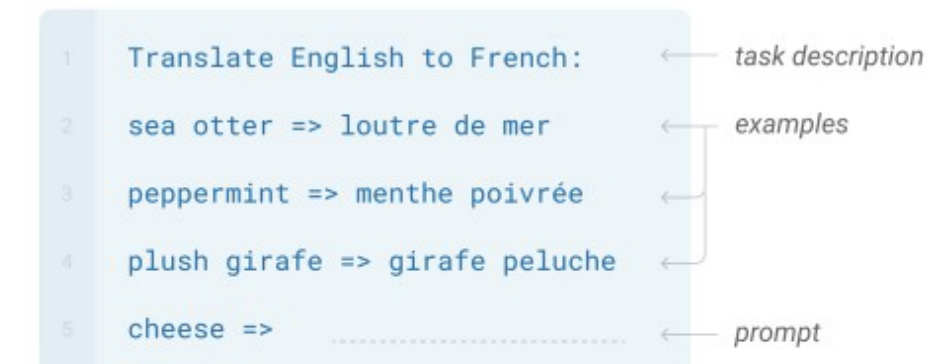
If we have **lots of good data**, we can **fine-tune** the LLM for better alignment

What if there is another way? Can we make use of the full Compound System and the data available to tune our system?

(Ans: Yes! With DSPy)

## Few-shot

In addition to the task description, the model sees a few examples of the task. No gradient updates are performed.



## Fine-tuning

The model is trained via repeated gradient updates using a large corpus of example tasks.



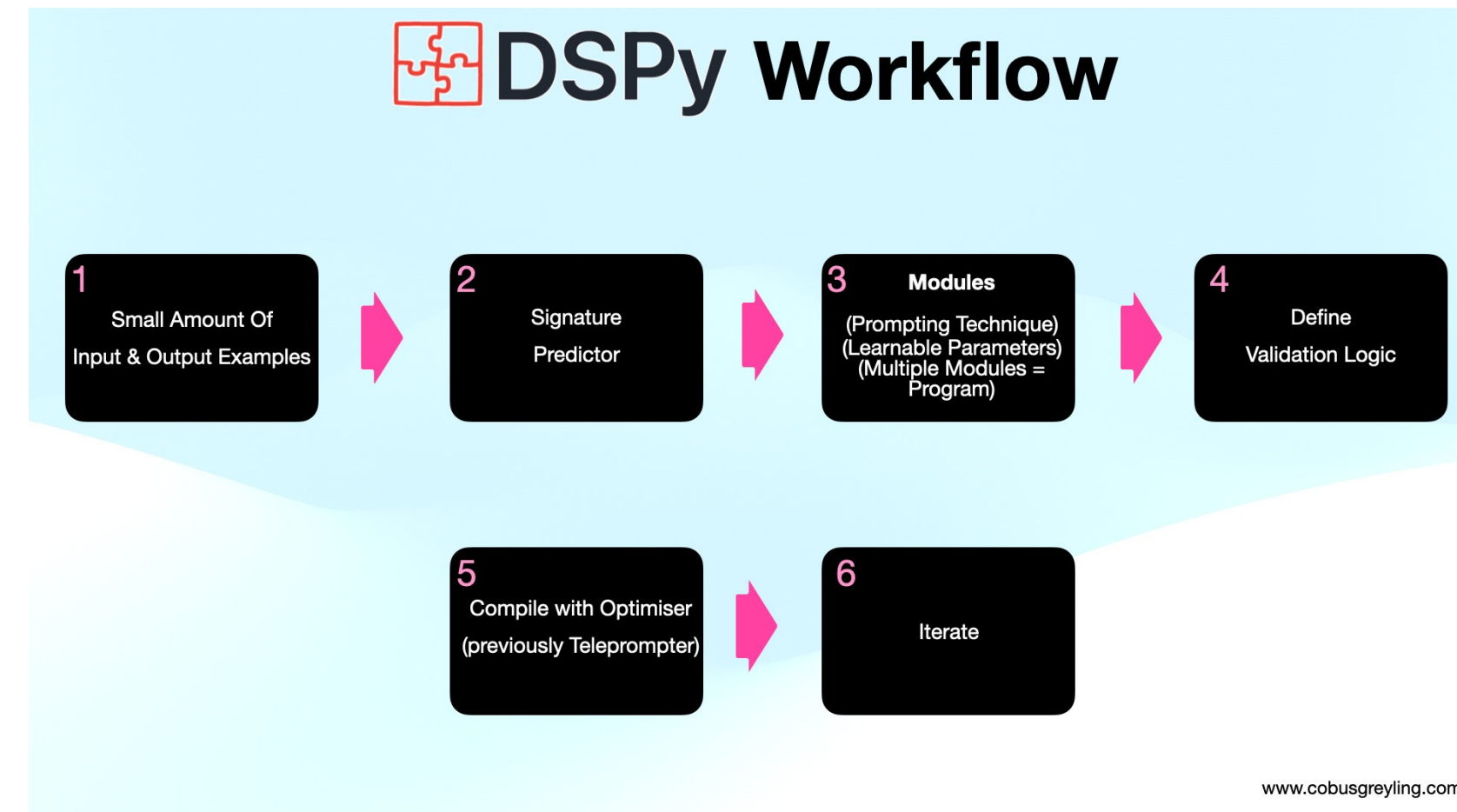
# Declarative Self-improving Language Programs, pyth<sub>o</sub>nically: DSPy

**DSPy is a framework for algorithmically optimizing LLM prompts and weights**, especially when LLMs are used one or more times within a pipeline. To use LLMs to build a complex system *without* DSPy, you generally have to:

- (1) Break the problem down into steps,
- (2) Prompt your LLM well until each step works well in isolation,
- (3) Tweak the steps to work well together,
- (4) Generate synthetic examples to tune each step, and
- (5) Use these examples to finetune smaller LLMs to cut costs.

Currently, this is hard and messy: every time you change your pipeline, your LLM, or your data, all prompts (or finetuning steps) may need to change.

**DSPy** separates the flow of the program (modules) from the parameters (LLM prompts and weights) of each step.



[www.cobusgreyling.com](http://www.cobusgreyling.com)

# DSPy Components

## Modules

A DSPy module is a building block for programs that use LMs. Each built-in module abstracts a prompting technique (like chain of thought or ReAct). Crucially, they are generalized to handle any DSPy Signature.

## Signatures

A signature is a declarative specification of input/output behavior of a DSPy module. Signatures allow you to tell the LM *what* it needs to do, rather than specify *how* we should ask the LM to do it.

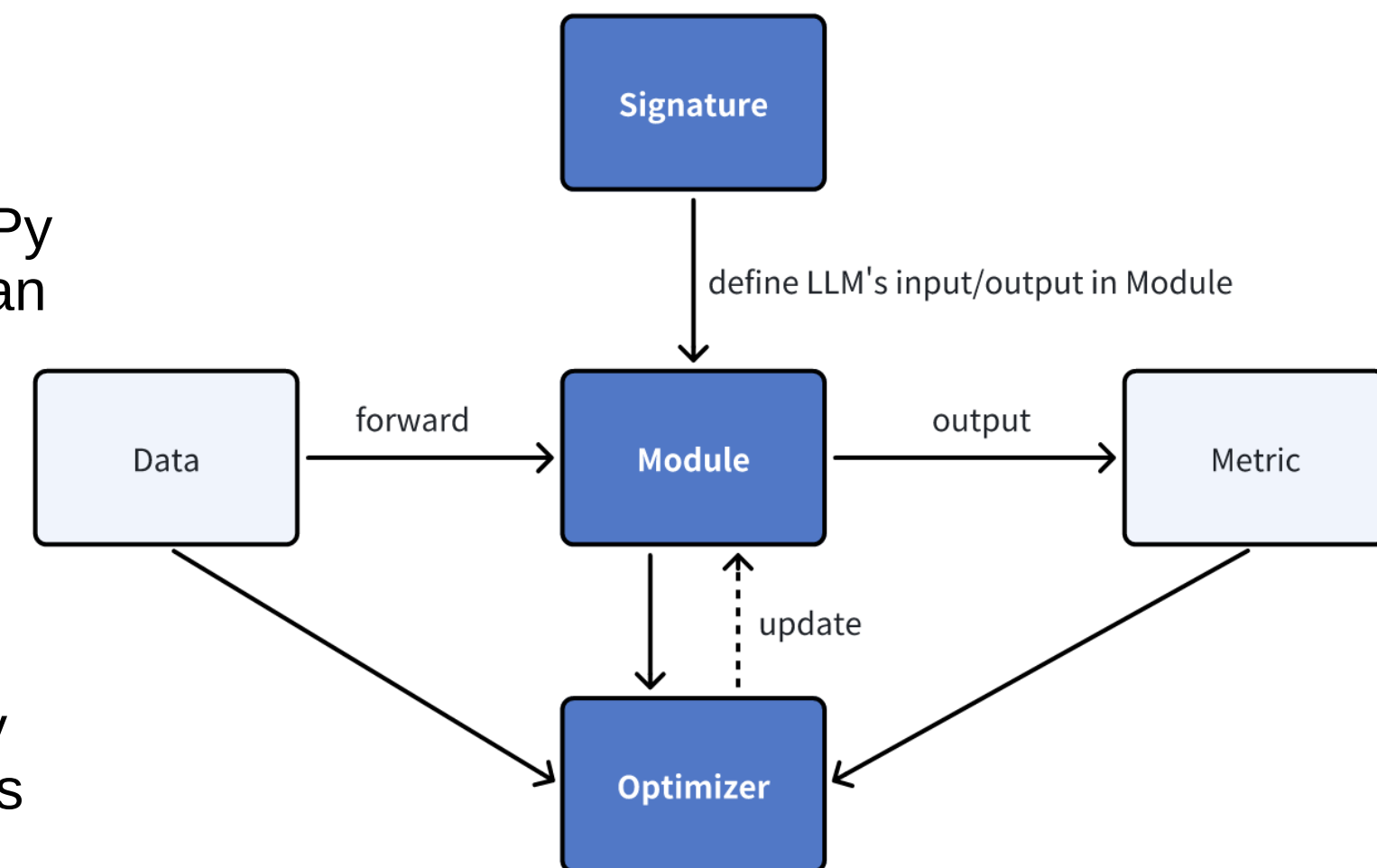
## LMs

A DSPy LM are the LLMs that we are familiar with already

## Optimizers

A DSPy optimizer is an algorithm that can tune the parameters of a DSPy program (i.e., the prompts and/or the LM weights) to maximize the metrics you specify, like accuracy.

How DSPy Components Collaborate with Each Other



# DSPy LMs and Signatures

Signatures can be defined as a short string, with argument names that define semantic roles for inputs/outputs.

1. Question Answering: "question → answer"
2. Sentiment Classification: "sentence → sentiment"
3. Summarization: "document → summary"

## Example A: Sentiment Classification

```
sentence = "it's a charming and often affecting journey." # example from the SST-2 dataset.  
  
classify = dsp.Predict('sentence -> sentiment')  
classify(sentence=sentence).sentiment
```

Output:

```
'Positive'
```

Your signatures can also have multiple input/output fields.

4. Retrieval-Augmented Question Answering: "context, question → answer"
5. Multiple-Choice Question Answering with Reasoning: "question, choices → reasoning, selection"



# DSPy Modules

A **DSPy module** is a building block for programs that use LMs.

Each built-in module abstracts a **prompting technique** (like chain of thought or ReAct). Crucially, they are generalized to handle any DSPy Signature.

A DSPy module has **learnable parameters** (i.e., the little pieces comprising the prompt and the LM weights) and can be invoked (called) to process inputs and return outputs.

Multiple modules can be composed into bigger modules (programs). DSPy modules are inspired directly by NN modules in PyTorch, but applied to LM programs.

1. **dspy.Predict**: Basic predictor. Does not modify the signature. Handles the key forms of learning (i.e., storing the instructions and demonstrations and updates to the LM).
2. **dspy.ChainOfThought**: Teaches the LM to think step-by-step before committing to the signature's response.
3. **dspy.ProgramOfThought**: Teaches the LM to output code, whose execution results will dictate the response.
4. **dspy.ReAct**: An agent that can use tools to implement the given signature.

# DSPy Optimizers

A **DSPy optimizer** is an algorithm that can tune the parameters of a DSPy program (i.e., the prompts and/or the LM weights) to maximize the metrics you specify, like accuracy.

There are many built-in optimizers in DSPy, which apply vastly different strategies. A typical DSPy optimizer takes three things:

Your **DSPy program**. This may be a single module (e.g., `dspy.Predict`) or a complex multi-module program.

Your **metric**. This is a function that evaluates the output of your program, and assigns it a score (higher is better).

A few **training inputs**. This may be very small (i.e., only 5 or 10 examples) and incomplete (only inputs to your program, without any labels).

If you happen to have a lot of data, DSPy can leverage that. But you can start small and get strong results.

# DSPy vs. LlamaIndex vs. LangChain

LangChain and LlamaIndex target high-level application development; they offer *batteries-included*, pre-built application modules that plug in with your data or configuration.

If you'd be happy to use a generic, off-the-shelf prompt for question answering over PDFs or standard text-to-SQL, you will find a rich ecosystem in these libraries.

**DSPy** doesn't internally contain hand-crafted prompts that target specific applications.

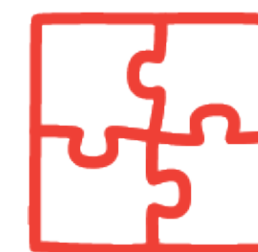
Instead, **DSPy** introduces a small set of much more powerful and general-purpose modules *that can learn to prompt (or finetune) your LM within your pipeline on your data*. when you change your data, make tweaks to your program's control flow, or change your target LM, the **DSPy compiler** can map your program into a new set of prompts (or finetunes) that are optimized specifically for this pipeline



## LangChain



## LlamaIndex



## DSPy

# Evaluation of LLMs and Compound Systems

Checking our work

# Evaluating Models: Supervised Learning

For traditional machine learning models, and for deep learning models, evaluation is straightforward as there is a specific set of inputs and outputs that the model uses to train on.

We can test how well the model performs by starting with known input and output pairs and seeing how well the model predicts the output from a given input.

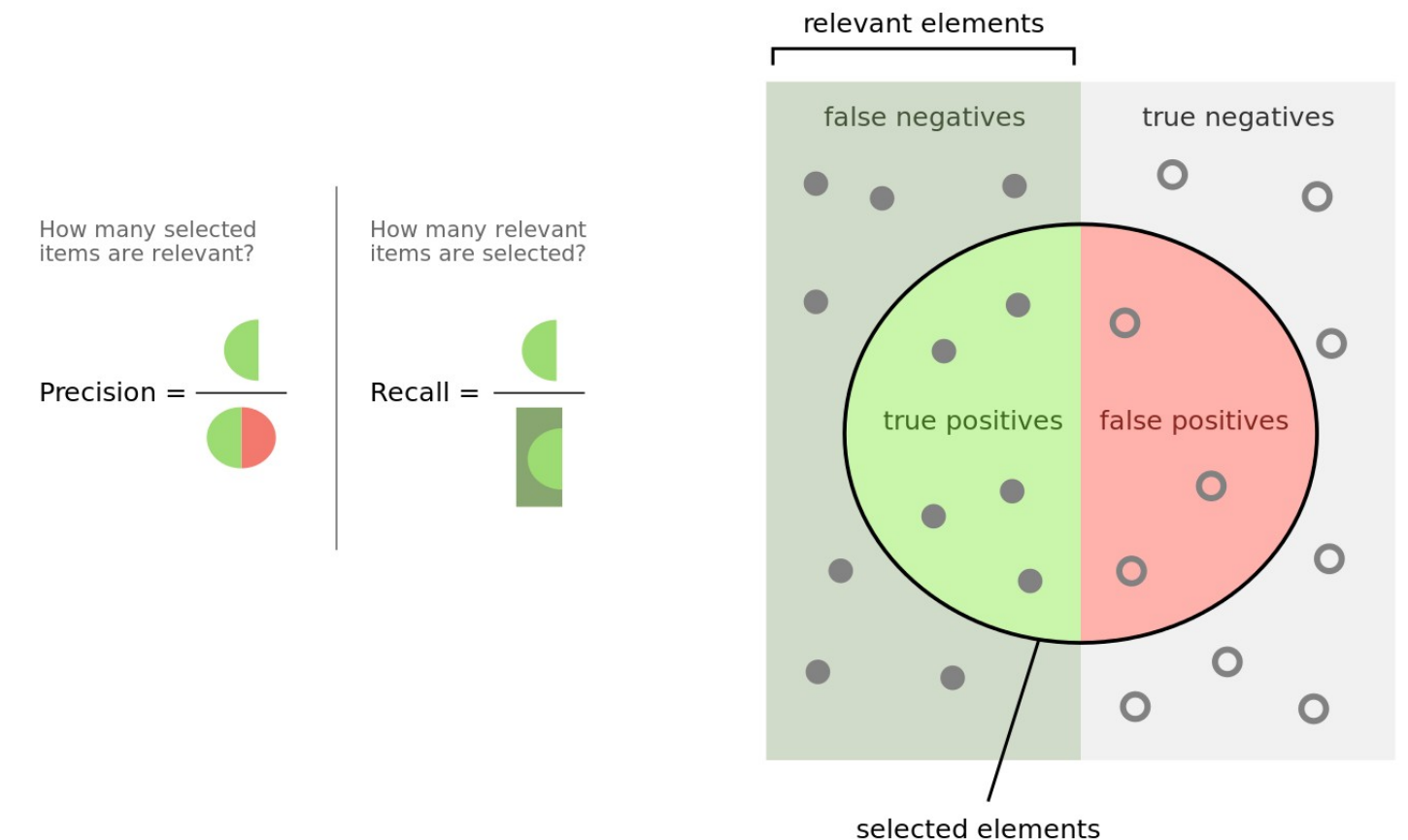
## Classification Models

- Precision
- Recall
- ROC Curve
- F1 score

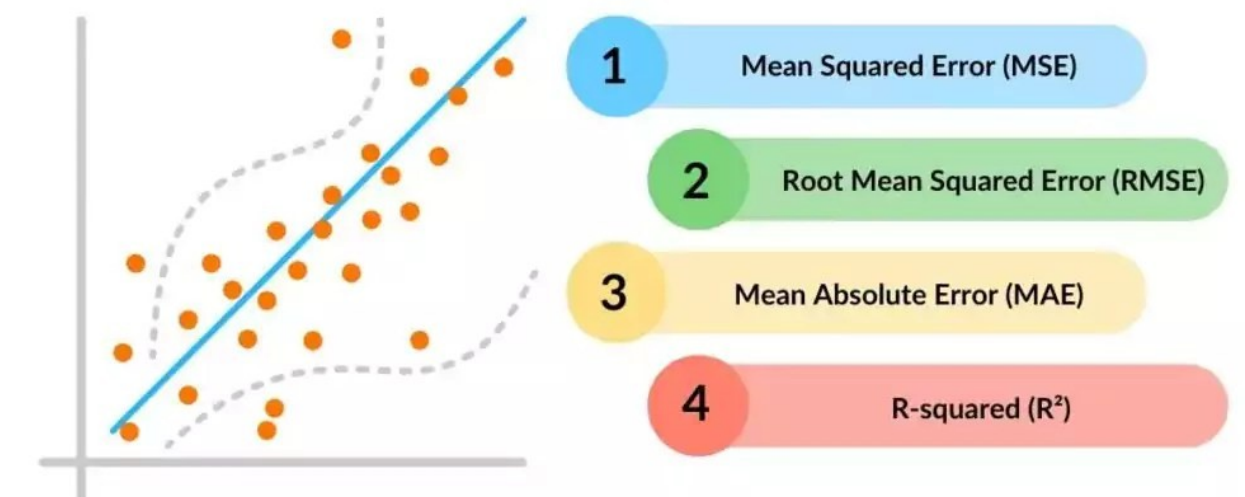
## Regression Models

- Mean Squared Error
- R-squared

What about LLMs?



## 4 Common Regression Metrics



# Why LLM systems are hard to evaluate

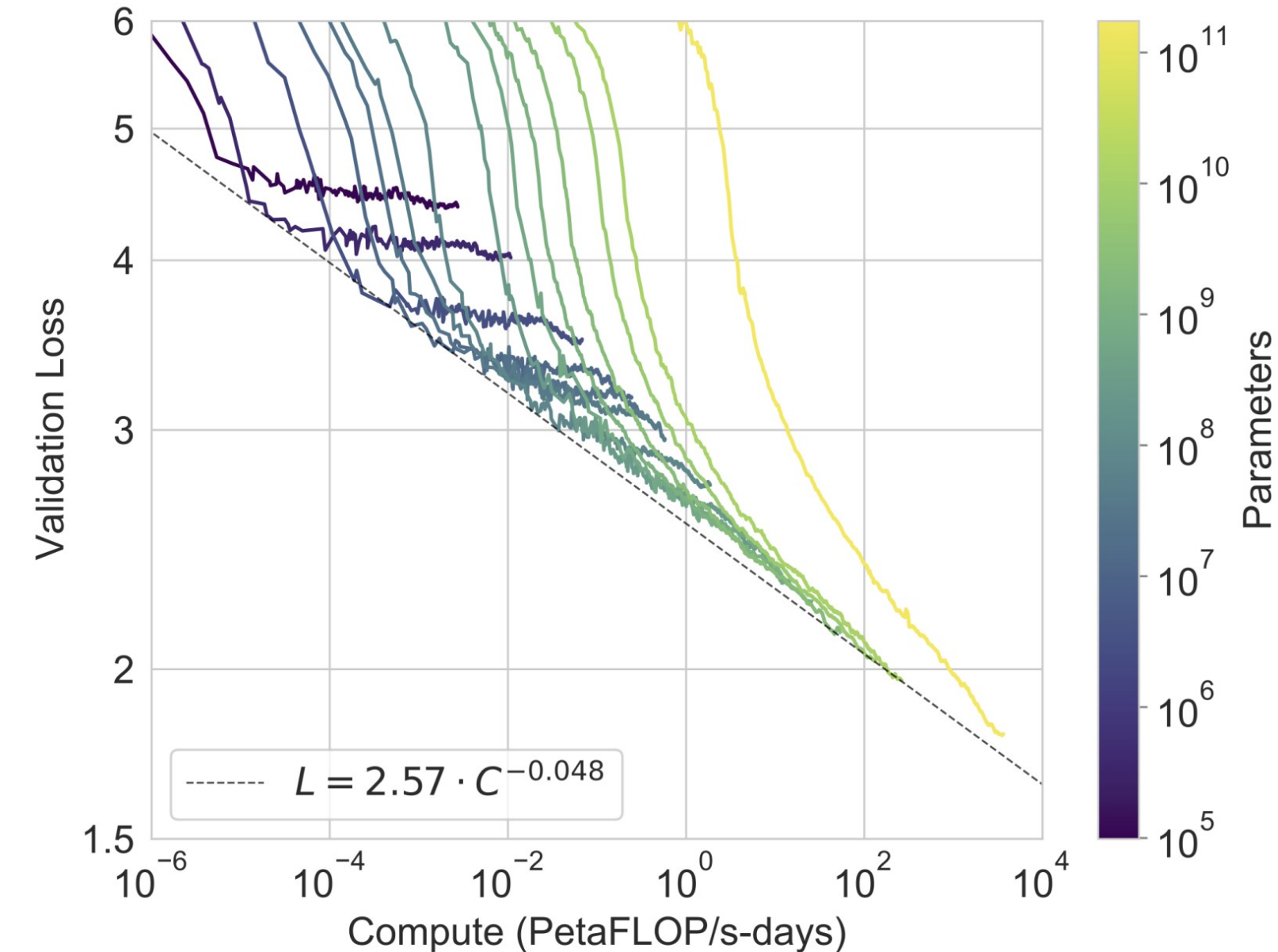
Consider how an LLM like GPT-3 is trained

- LLM predicts the next token.
- We know, in the training data, which is the “correct” token, this is how we calculate loss.
- So evaluation should be simple, follow the loss?

Nope!

Recall that the real power of GPT-3 and the models that came after it relied on in-context learning and the power of few-shot learning.

The loss curves indicate that the model can understand language, but to evaluate something like a RAG system, this doesn't give us any information at all...





# Evaluating a RAG System

Let's consider what the task is for RAG and see where we can evaluate the performance.

## RAG Pipeline:

1. Take user's input and search for similar content
2. Retrieve similar content from Vector Database
3. Combine user input and context into a single prompt
4. Generate the answer to the user's query

We can look at each of these stages and consider how we might evaluate it.

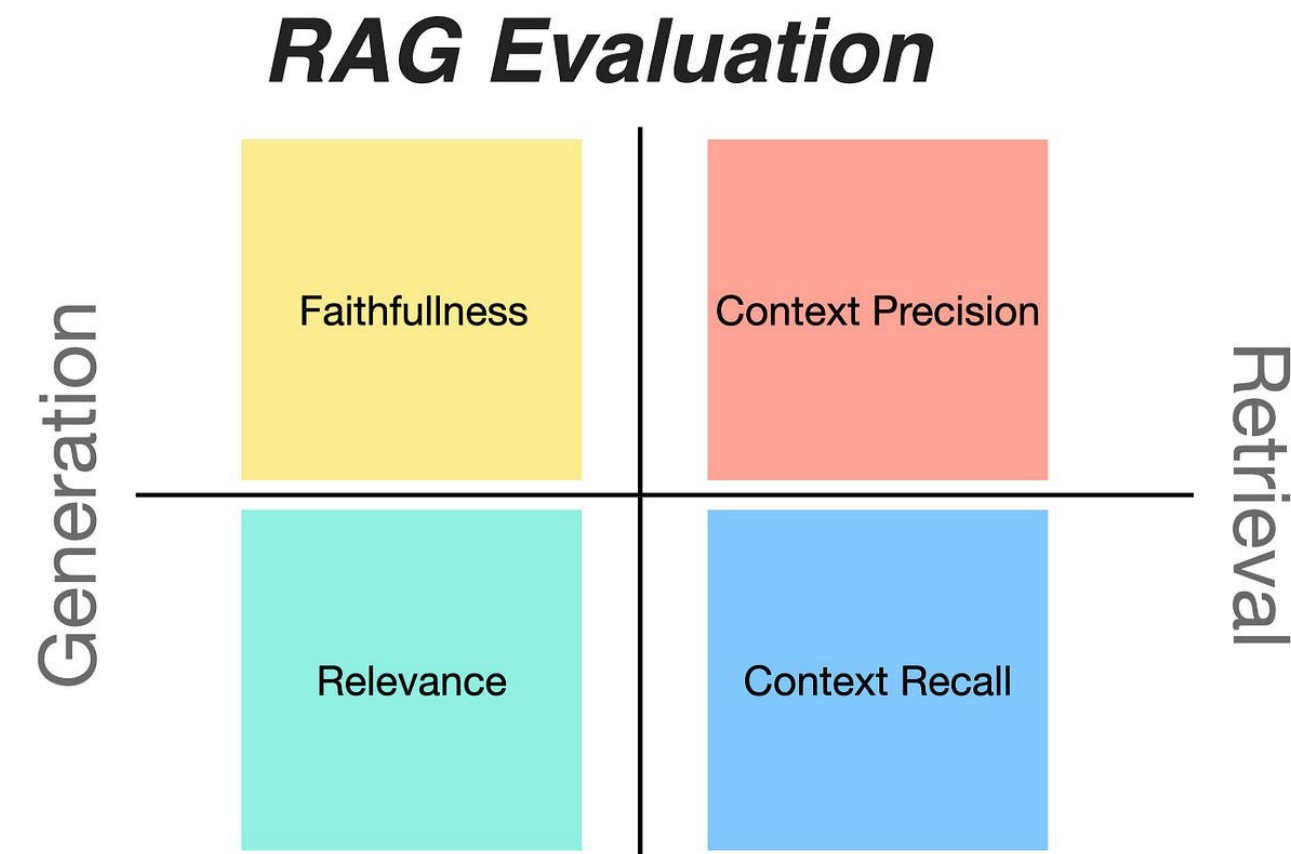
## Retriever Evaluation

- How well does the retriever find the right content?
- If multiple records are returned, is the best one at the top?

## Generation Evaluation

- How well does the model use the content to answer the user's query?
- How much of the content retrieved is used in answering the user's query?

Note that we are using very subjective terminology. "How well" something is done can be very hard to measure in natural language.



# Using an LLM to judge another LLM?

LLMs excel at interpreting their context window. More so than generate new text. This means that we can use an LLM to effectively judge the performance of an LLMs generated answer.

We can look at 3 LLM-as-a-judge variations. They can be implemented independently or in combination:

1. **Pairwise comparison.** An LLM judge is presented with a question and two answers, and tasked to determine which one is better or declare a tie.
2. **Single answer grading.** Alternatively, an LLM judge is asked to directly assign a score to a single answer.
3. **Reference-guided grading.** In certain cases, it may be beneficial to provide a reference solution if applicable with a grading scale.

Each of these methods require different amounts of data but offer more structured approaches to evaluating the performance of an LLM system.




# Evaluation Frameworks

Frameworks like LangChain and MLflow offer support to implement evaluation of LLMs

```
from langchain.evaluation import load_evaluator

evaluator = load_evaluator("labeled_pairwise_string")

evaluator.evaluate_string_pairs(
    prediction="there are three dogs",
    prediction_b="4",
    input="how many dogs are in the park?",
    reference="four",
)
```



```
with mlflow.start_run() as run:
    system_prompt = "Answer the following question in two sentences"
    # Wrap "gpt-4" as an MLflow model.
    logged_model_info = mlflow.openai.log_model(
        model="gpt-4",
        task=openai.chat.completions,
        artifact_path="model",
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": "{question}"},
        ],
    )

    # Use predefined question-answering metrics to evaluate our model.
    results = mlflow.evaluate(
        logged_model_info.model_uri,
        eval_data,
        targets="ground_truth",
        model_type="question-answering",
    )
    print(f"See aggregated evaluation results below: \n{results.metrics}")


    # Evaluation result for each data record is available in `results.tables`.
    eval_table = results.tables["eval_results_table"]
    print(f"See evaluation table below: \n{eval_table}")
```





# Evaluation Frameworks: RAGAS and ARES

We are also seeing the development of new evaluation-focused frameworks such as RAGAS and ARES



*Evaluation framework for your Retrieval Augmented Generation (RAG) pipelines*

release v0.1.14 Made with Python license Apache-2.0 Open in Colab explodinggradients 2128 members Open Source

[Documentation](#) | [Installation](#) | [Quickstart](#) | [Community](#) | [Open Analytics](#) | [Hugging Face](#)

🚀 Dedicated solutions to evaluate, monitor and improve performance of LLM & RAG application in production including custom models for production quality monitoring. [Talk to founders](#)

Ragas is a framework that helps you evaluate your Retrieval Augmented Generation (RAG) pipelines. RAG denotes a class of LLM applications that use external data to augment the LLM's context. There are existing tools and frameworks that help you build these pipelines but evaluating it and quantifying your pipeline performance can be hard. This is where Ragas (RAG Assessment) comes in.

Ragas provides you with the tools based on the latest research for evaluating LLM-generated text to give you insights about your RAG pipeline. Ragas can be integrated with your CI/CD to provide continuous checks to ensure performance.

## ARES: An Automated Evaluation Framework for Retrieval-Augmented Generation Systems

Table of Contents: [Installation](#) | [Requirements](#) | [Quick Start](#) | [Citation](#)

release v0.5.7 Read ARES Paper Read documentation Open in Colab Made with Python

ARES is a groundbreaking framework for evaluating Retrieval-Augmented Generation (RAG) models. The automated process combines synthetic data generation with fine-tuned classifiers to efficiently assess context relevance, answer faithfulness, and answer relevance, minimizing the need for extensive human annotations. ARES employs synthetic query generation and Prediction-Powered Inference (PPI), providing accurate evaluations with statistical confidence.

# Wrap Up

## LLM Compound Systems

- Today we reviewed more concepts of LLM Chains in LangChain
  - Introduced the Retrieval Augmented Generation LLM System
  - Explored the libraries that have been developed to build LLM Compound Systems like LlamaIndex and DSPy
  - Discussed the issues with evaluating LLMs and Compound systems
- 

In the next lesson we will introduce the concept of LLM agents and explore how LLMs can be used as reasoning engines inside more complex applications





Thank you!