**LangChain**

LangChain is a framework for developing applications powered by language models. It enables applications that are:

* Data-aware: connect a language model to other sources of data
* Agentic: allows a language model to interact with its environment

It is a framework that allows AI developers to combine LLMs with external sources of computation and data.

## **LLM Capabilities**

The following are a few capabilities of LLMs:

1. Natural Language Understanding (NLU): To interpret human language in the form of text.
2. Natural Language Generation (NLG): The model can generate human-like text, including coherent paragraphs, articles, stories, and poems.
3. Text Completion and Prediction
4. Language Translation
5. Question-Answering
6. Chatbot Capabilities

But in order to use these in real-world applications, we want something more.

For instance, consider the following example:

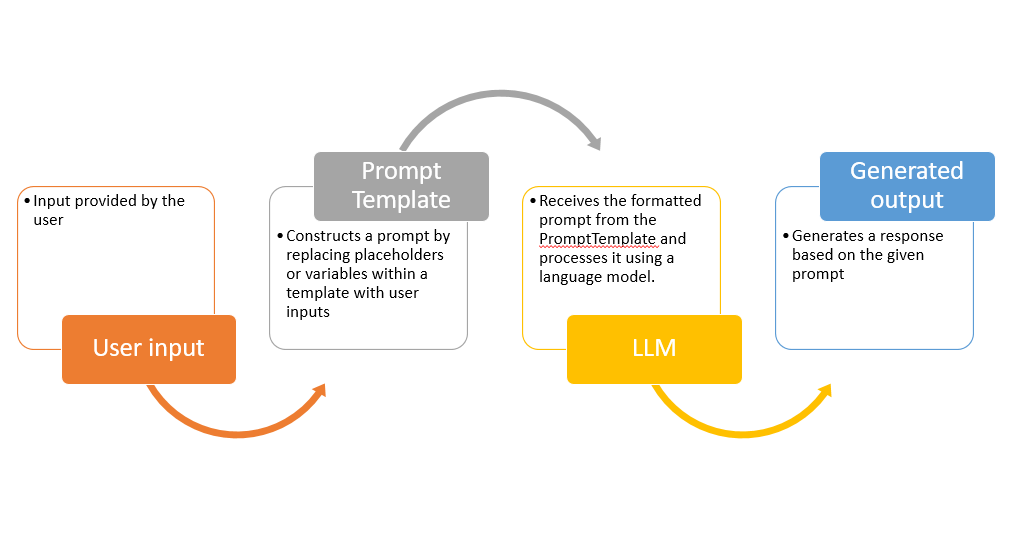
**Send an Email to ashlesha.sharma@impressico.com via Gmail to say hello and ask how is she doing.**

**OR**

**Create a chatbot for a Q&A on a few private documents.**

## **Why Lang’Chain’?**

A Chain is a sequence of steps within the LangChain framework that combines primitives and LLMs to process user input, generate prompts, and leverage the power of OpenAI large language models (LLMs) for NLP tasks.



**Other Benefits:**

1. To avoid thinking of any model's token limit.

2. Saves a lot of code by providing various wrappers like Prompt Template, ChatModels, Chains etc.

3. To do a multi-step process efficiently.

4. To get more accuracy: ChatGPT-like models struggle with generating factual statements if no context is provided. They have some general knowledge but cannot guarantee to produce a valid answer consistently.

**NOTEBOOK-> (A math problem)**

## **Langchain Modules**

LangChain provides standard, extendable interfaces and external integrations for the

following modules, listed from least to most complex:



## **Model IO**

* Wrapper over LLMs
* Can Easily switch among different models.
* This section focuses on how to create basic input prompt requests for models and how to manage their outputs.
* Langchain supports all major LLMs (OpenAI, Azure, Anthropic, Google Cloud, etc.)

### **LLMs**

There are two main types of APIs in Langchain:

* + Text Completion Model: Returns the most likely text to continue
  + Chat Models: Converses with back-and-forth messages, can also have a “system” prompt.
* LLMs supported by Langchain:

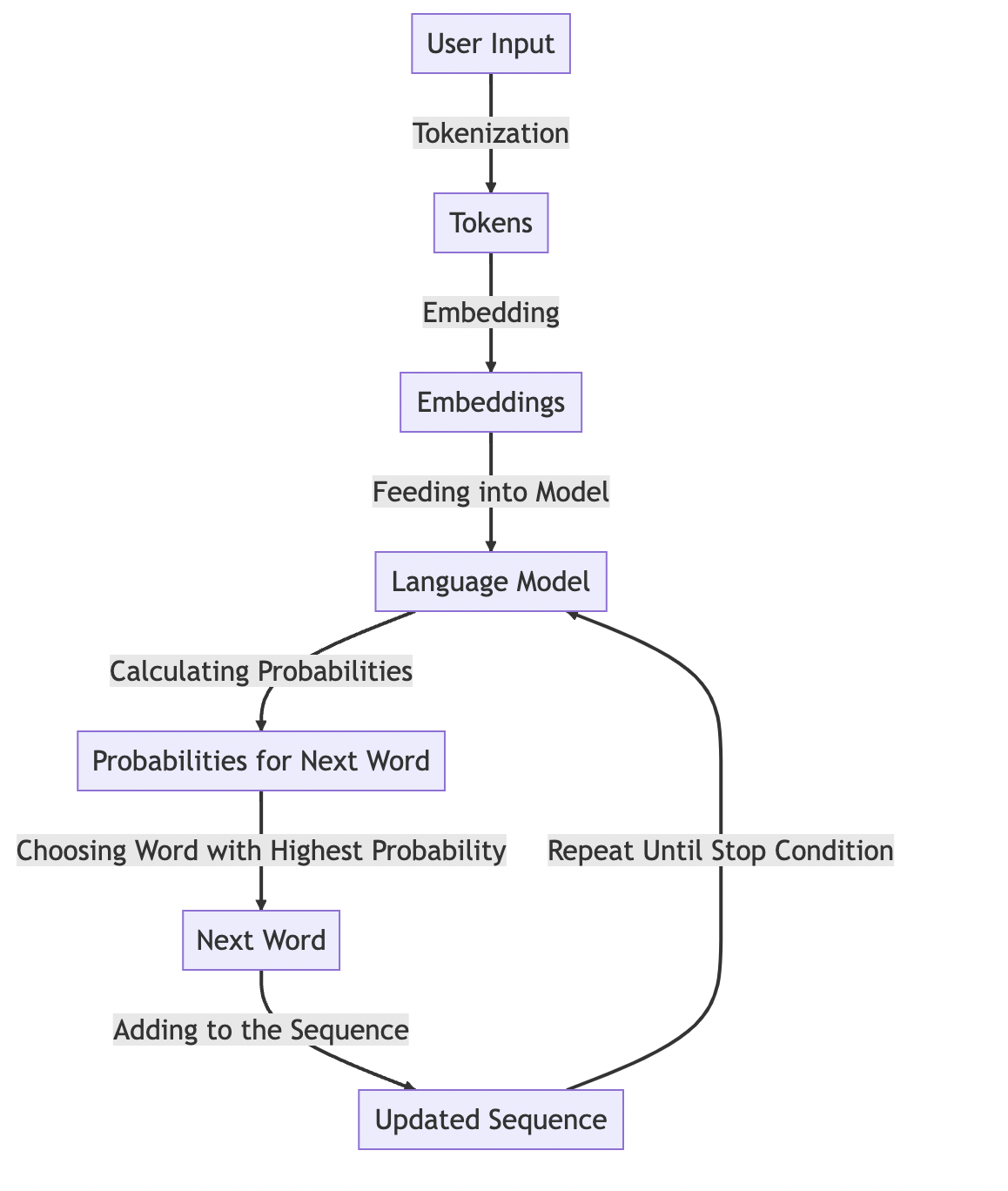
<https://python.langchain.com/docs/integrations/llms/>

**NOTEBOOK-> (Few LLM Wrapper Features)**

### **Prompts**

#### **LLM Word Prediction:**

How an LLM like GPT-4 generates text using next word prediction technique:



1. User Input: The user provides an input prompt to the model.

2. Tokenization: The input is broken down into smaller pieces called tokens. In English, a token could be as short as one character or as long as one word (e.g., 'a', 'an', 'apple').

3. Embedding: Each token is then mapped to a high-dimensional vector that represents it in the model's vocabulary.

4. Feeding into Model: These vectors are fed into the language model.

5. Calculating Probabilities: The model calculates the probability of each possible next word in the vocabulary.

6. Choosing Word with Highest Probability: The word with the highest probability is chosen as the next word in the sequence.

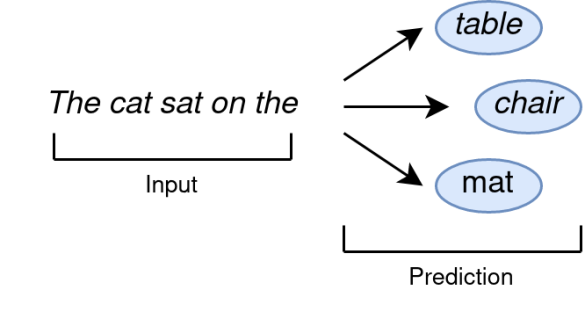
7. Adding to the Sequence: This word is added to the sequence, and the process repeats until a stop condition is met (e.g., the maximum length of the sequence, end of sentence token is generated).

This process allows the model to generate coherent and contextually relevant sentences. However, it's worth noting that while the model can generate text based on patterns it has learned during training, it does not understand the text in the same way a human would.

#### **What is a Prompt?**

In simple language, a prompt is an information that we pass to LLMs o guide the model's response

For example



The probability of predictions varies on the context and instructions we gave in the prompt:

| **Context** | **Information** | **Prediction 1** | **Prediction 2** | **Prediction 3** | **Output** |
| --- | --- | --- | --- | --- | --- |
| The room has only one table in the centre | The cat sat on the | table (95 %) | mat (20 %) | Chair (50%) | The room has only one table in the center. The cat sat on the table. |
| The baby was playing on the mat in the room | The cat sat on the | floor (80 %) | mat (80 %) | Table (50%) | The baby was playing on the mat in the room, and the cat sat on the floor nearby. |

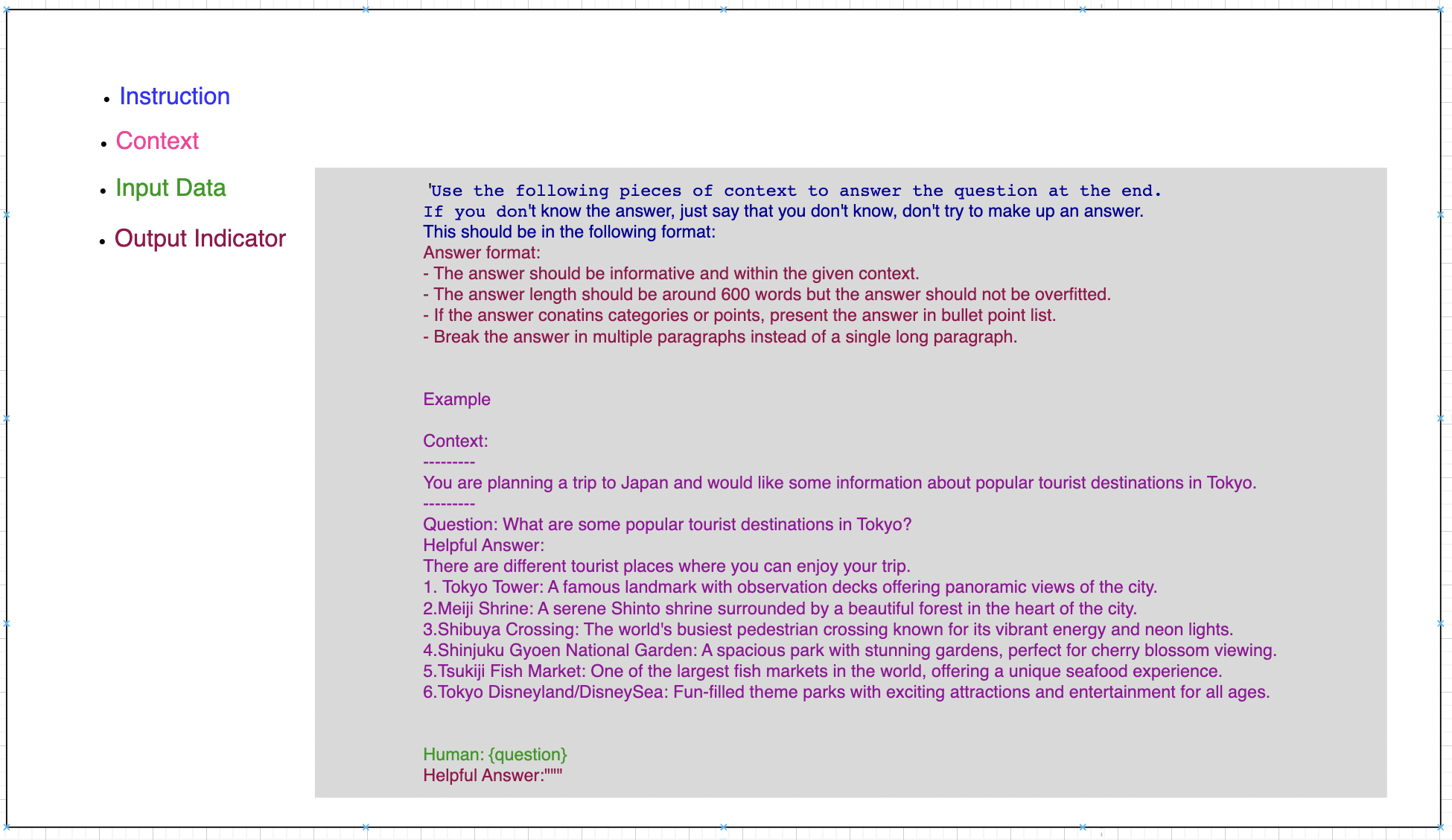
**Composition of a Formal Prompt:**

1. Instruction: This part of the prompt tells the LLM, what action needs to be performed on the input.

2. Context: The additional information to fine-tune the instruction.

3. Input Data: The data that the model will process.

4. Output Indicator: it signals the LLM model the nature of output in terms of format, sentiment etc.



### **Prompt Templates**

Prompt templates are the wrapper class around the prompts which Langchain provides.

* Templates allow us to easily configure and modify our input prompts to LLM calls.
* Templates offer a more systematic approach to passing in variables to prompts for models, instead of using f-string literals or .format() calls, the PromptTemplate converts these into function parameter names that we can pass in.
* Let’s explore how to build prompt templates for both LLM Text Completion and prompt templates for Chat Models.

**NOTEBOOK-> (Few Prompt template examples)**

### **Serialization (Saving Prompts)**

This section shows how can we save and load the saved prompt.

**NOTEBOOK-> (Prompt serialisation)**

### **Parsing Outputs**

* Often when connecting LLM output you need it in a particular format, for example, you want a Python datetime object or a JSON object.
* LangChain comes with Parse utilities allowing you to easily convert outputs into precise data types or even your own custom class instances with Pydantic.
* **Parsers**

Consists of two key elements:

* + - format\_instructions   
      An extra string that Langchain adds to the end of a prompt to assist with formatting.
    - parse() method:  
      A method for using eval() internally to parse the string reply to the exact Python object you need.
* **Parsers Example**

If You need a datetime response from an LLM. There are two Main Issues:

* + LLM always replies back with a string: ex - “2020-01-01”
  + Could be formatted in many ways: “Jan 1st, 2020”
* Parsers use format\_instructions to take care of the first issue and eval() to take care of the second issue.

**NOTEBOOK-> (Few Parsing examples)**

* **Pydantic library:**
  + Using the Pydantic library for type validation, you use Langchain’s PydanticOutputParser to directly attempt to convert LLM replies to your own custom Python objects (as long as you built them with Pydantic).
  + Note, this requires you to have some Pydantic knowledge and pip install the Pydantic library.
  + Let’s explore a simple example!
* **OpenAI Function Calling**

**NOTEBOOK->(Pydantic library example)**

## **Data Connections**

In this section we’ll explore how to load documents, transform them to vector embeddings, and then store and query those vector embeddings.

* Section Overview:
  + Document Loading and Integrations
  + Document Transformers
  + Text Embedding
  + Vector Stores
  + Queries and Retrievers

### **Document Loader**

* Langchain comes with built-in loader tools to quickly load files to its own Document object.
* Note that many of these loaders require other libraries, for example, PDF loading requires the pypdf library and HTML loading requires the Beautiful Soup library. Make sure to pip install the required libraries before using the loader (the loaders will inform you if they can’t find the installed libraries)
* <https://python.langchain.com/docs/integrations/document_loaders/email>

**NOTEBOOK->**

### **Chunks and its Significance**

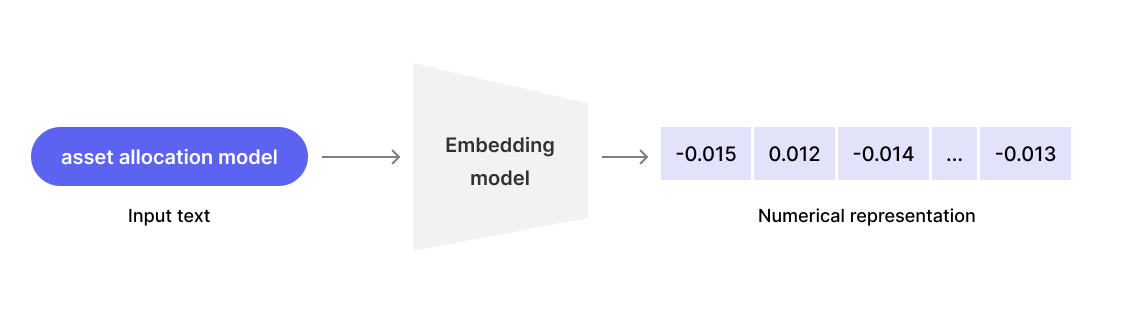
In machine learning, data is sometimes broken into smaller pieces called chunks or batches for several reasons:

* Memory Efficiency: Large datasets may not fit entirely into the memory of a machine learning system. By breaking the data into smaller chunks, the system can process one batch at a time, minimizing memory usage.
* Computational Efficiency: Processing data in smaller batches can improve computational efficiency. Many machine learning algorithms can take advantage of parallel processing, where multiple batches can be processed simultaneously on different computational resources, leading to faster training or inference.
* Gradient Descent Optimization: In optimization algorithms like gradient descent, breaking the data into batches allows for updating model parameters incrementally based on each batch. This approach, called stochastic gradient descent (SGD), enables faster convergence and more frequent parameter updates compared to using the entire dataset.
* Regularization and Generalization: Training on smaller batches of data can introduce regularization effects. By exposing the model to different subsets of data in each batch, it helps prevent overfitting and improves generalization performance by encouraging the model to learn from a more diverse set of examples.
* Streaming or Online Learning: In scenarios where new data arrives continuously, breaking the data into chunks allows for incremental learning or online learning. The model can be updated iteratively as new batches of data become available, enabling real-time learning and adaptation to evolving patterns.

By breaking data into smaller chunks, machine learning systems can handle large datasets more efficiently, achieve faster computation, facilitate optimization algorithms, enhance generalization, and support streaming or online learning paradigms.

### 

### **Vectors**

* In machine learning, a vector refers to an ordered collection of numerical values, often represented as an array or a list. Vectors are fundamental data structures used to represent various types of data, such as features, inputs, outputs, or parameters in machine learning models.  
  
* Vectors can be used for various types of searches, depending on the specific context and application. Here are some different types of searches that can be performed on vectors
  + Similarity Search: Finding vectors that are similar or close to a given query vector, often used in recommendation systems, content similarity analysis, and nearest neighbor searches.
  + Semantic Search: Searching for vectors that capture similar semantic meaning or context, allowing for more nuanced search queries in natural language processing tasks.
  + Clustering: Grouping vectors together based on their similarity or distance, enabling tasks such as document clustering, image segmentation, or customer segmentation.
  + Anomaly Detection: Identifying vectors that significantly deviate from the normal patterns or distributions, helping to detect anomalies in various domains like fraud detection or network intrusion detection.
  + Dimensionality Reduction: Reducing the dimensionality of high-dimensional vectors while preserving their important characteristics, commonly used in visualization, compression, or feature extraction tasks.
  + Contextual Search: Incorporating contextual information along with vectors to perform context-aware searches, such as context-aware recommendation systems or personalized search.

These are just a few examples of the diverse range of searches that can be conducted on vectors, demonstrating the flexibility and utility of vector-based approaches in various domains.

### **Embeddings and Vector Store (Pinecone)**

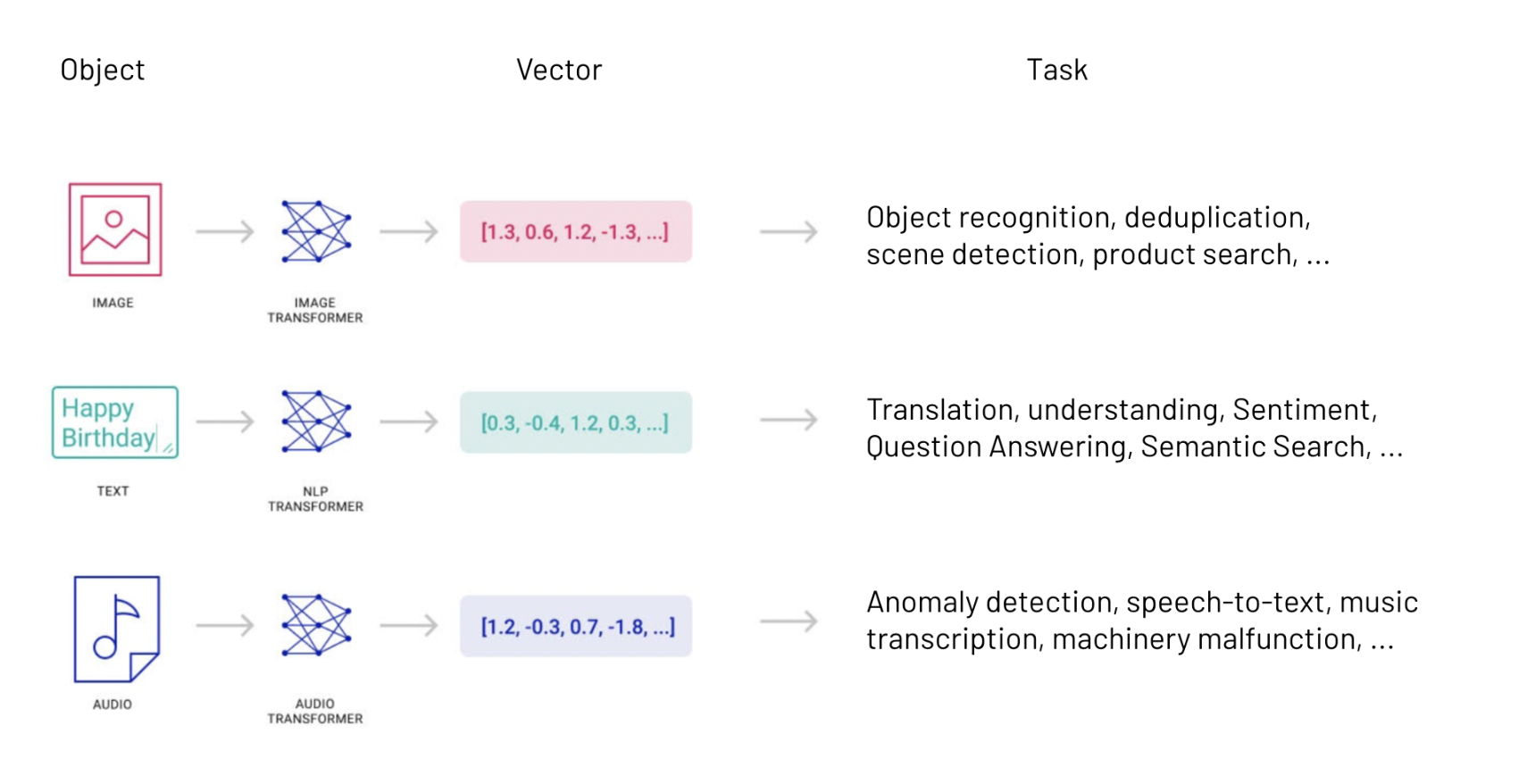
[Pinecone](https://www.pinecone.io/) is a managed vector database that provides vector search (or “similarity search”) for developers.

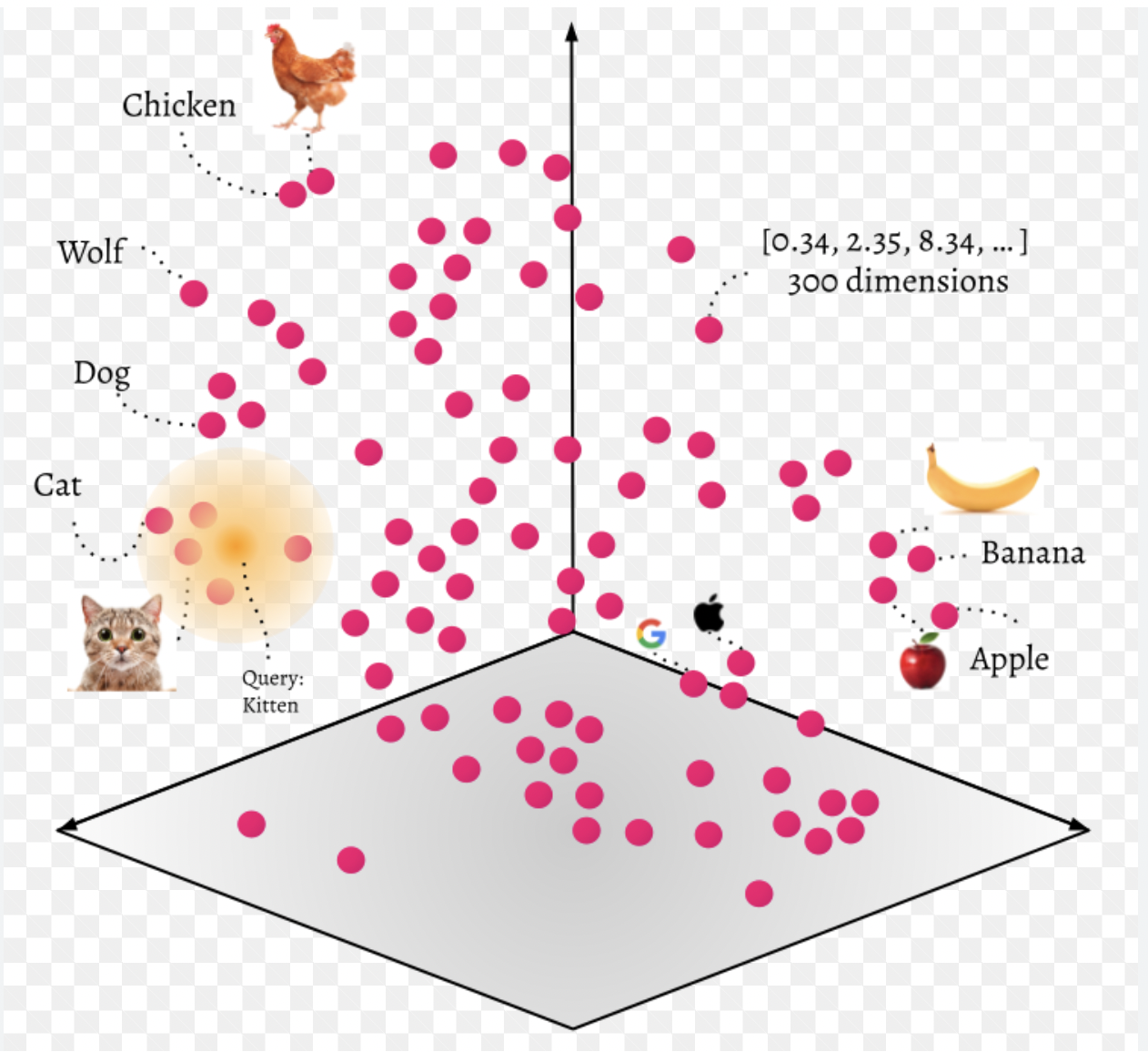
Regardless of the types of objects Pinecone is searching through, Pinecone uses “vectors” or “vector embeddings” to convert the data for analysis into simpler representations. Then, they check for similarity on those representations while still maintaining the deeper meaning of the objects themselves.

[Vector embeddings](https://www.pinecone.io/learn/vector-embeddings/) are really just a simplified numerical representation of complex data, used to make it easier to run generic machine-learning algorithms on sets of that data. By taking real-world objects and translating them to vector embeddings — numerical representations — those numbers can be fed into machine learning algorithms to determine [semantic similarity](https://www.pinecone.io/learn/vector-search-basics/learn/semantic-search/).

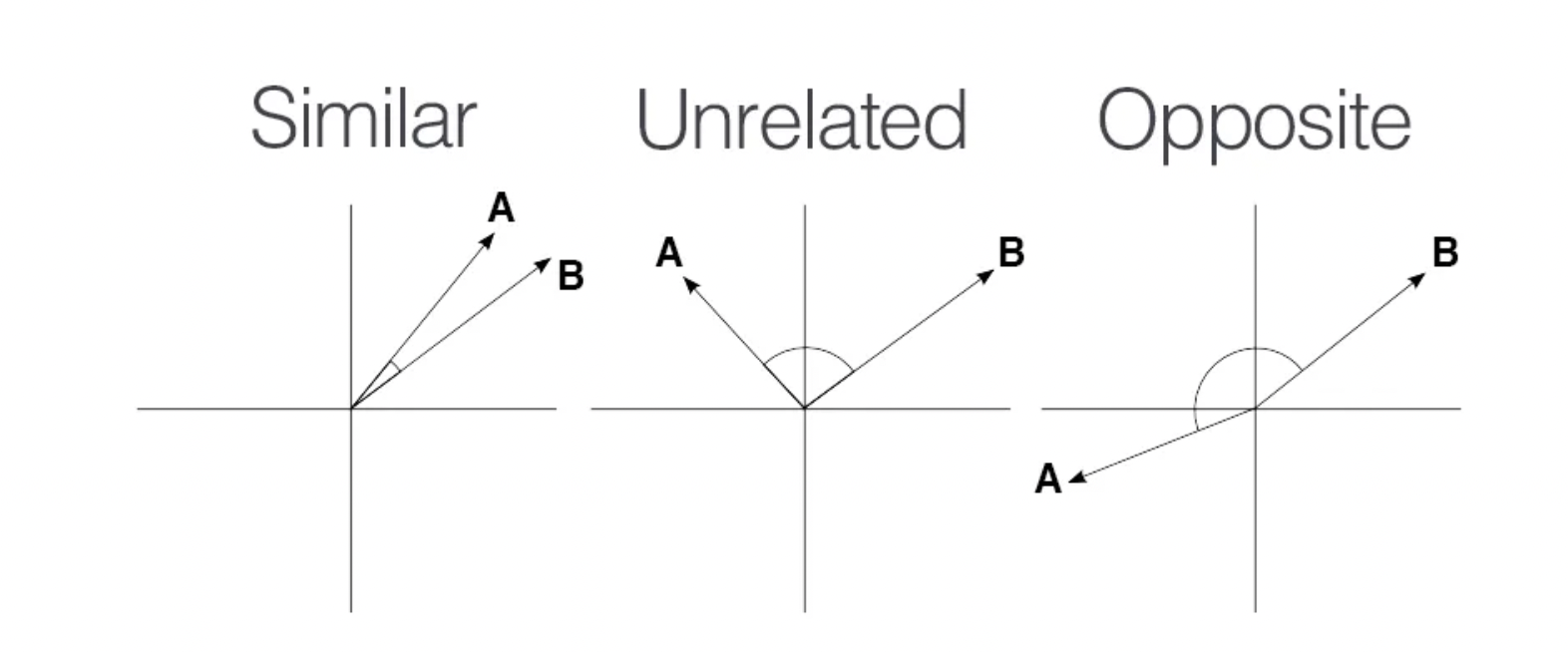
Regardless of the model you’re using, though, Pinecone can help you search through the generated vector embeddings to find similar items.

Once you have a vector embedding, you need to be able to run queries against it. This is where Pinecone comes in. Rather than requiring you to learn all kinds of techniques for searching through your data, Pinecone provides managed vector search. You store vector embeddings with IDs that tie your data back to the objects they represent, allowing you to search through that data with a straightforward API and client. A store of vector embeddings and their IDs is called a “vector index.”



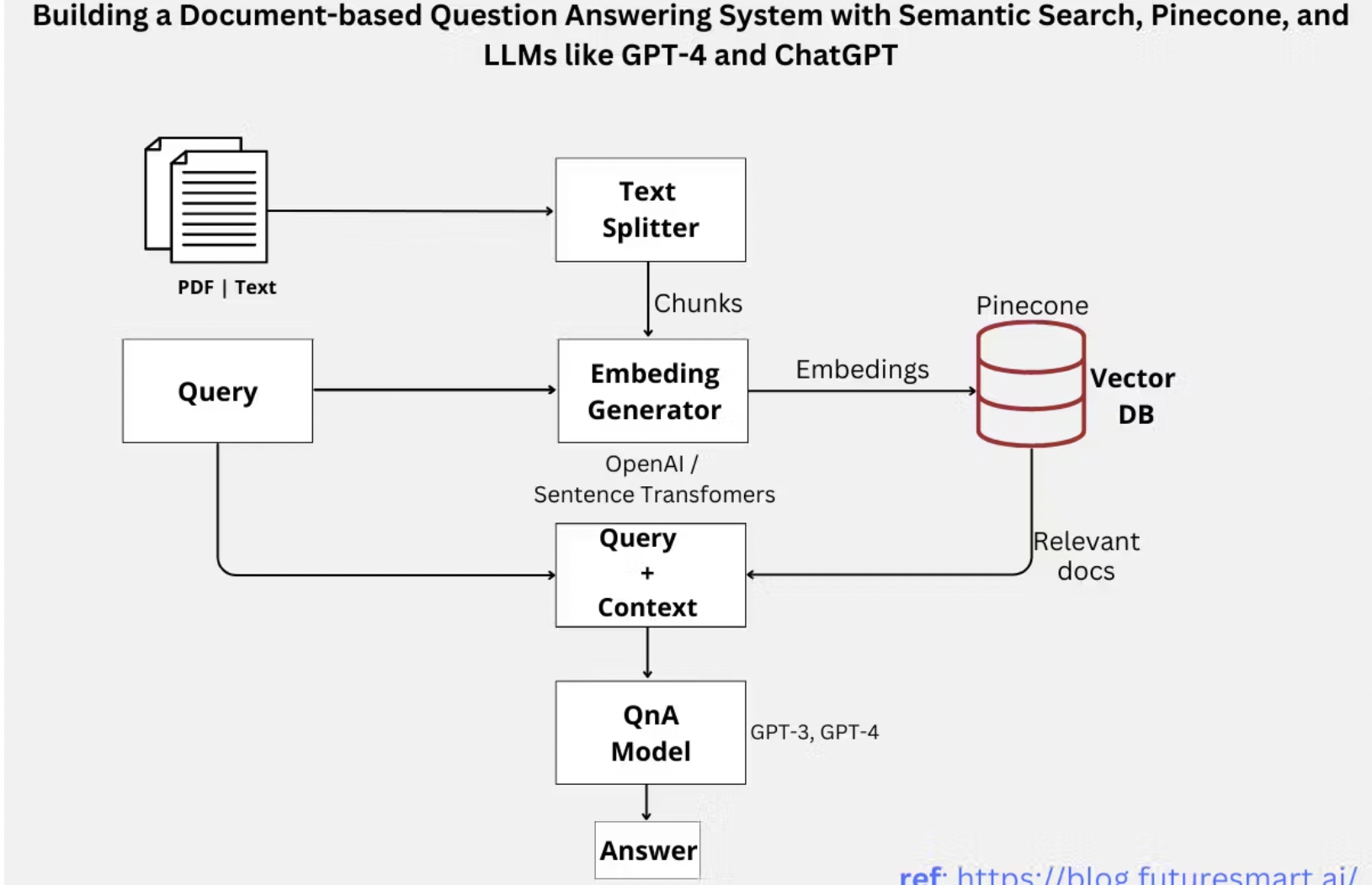


### **Cosine Similarity:**



### **A Conversational Bot**

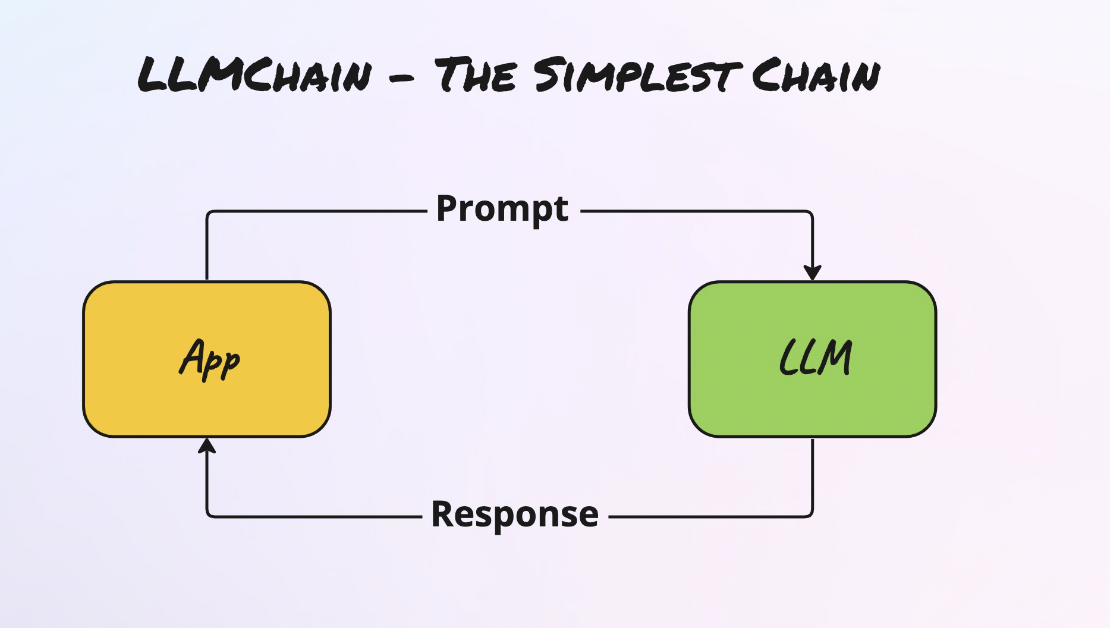
### 



**NOTEBOOK->**

## **Chains**

* Chains allow you to link the output of one model to be the input of another model call.
* LangChain Advantage:
  + Easily chain together different LLM calls to separate out work between models, allowing you to swap LLMs in the middle of a chain easily.



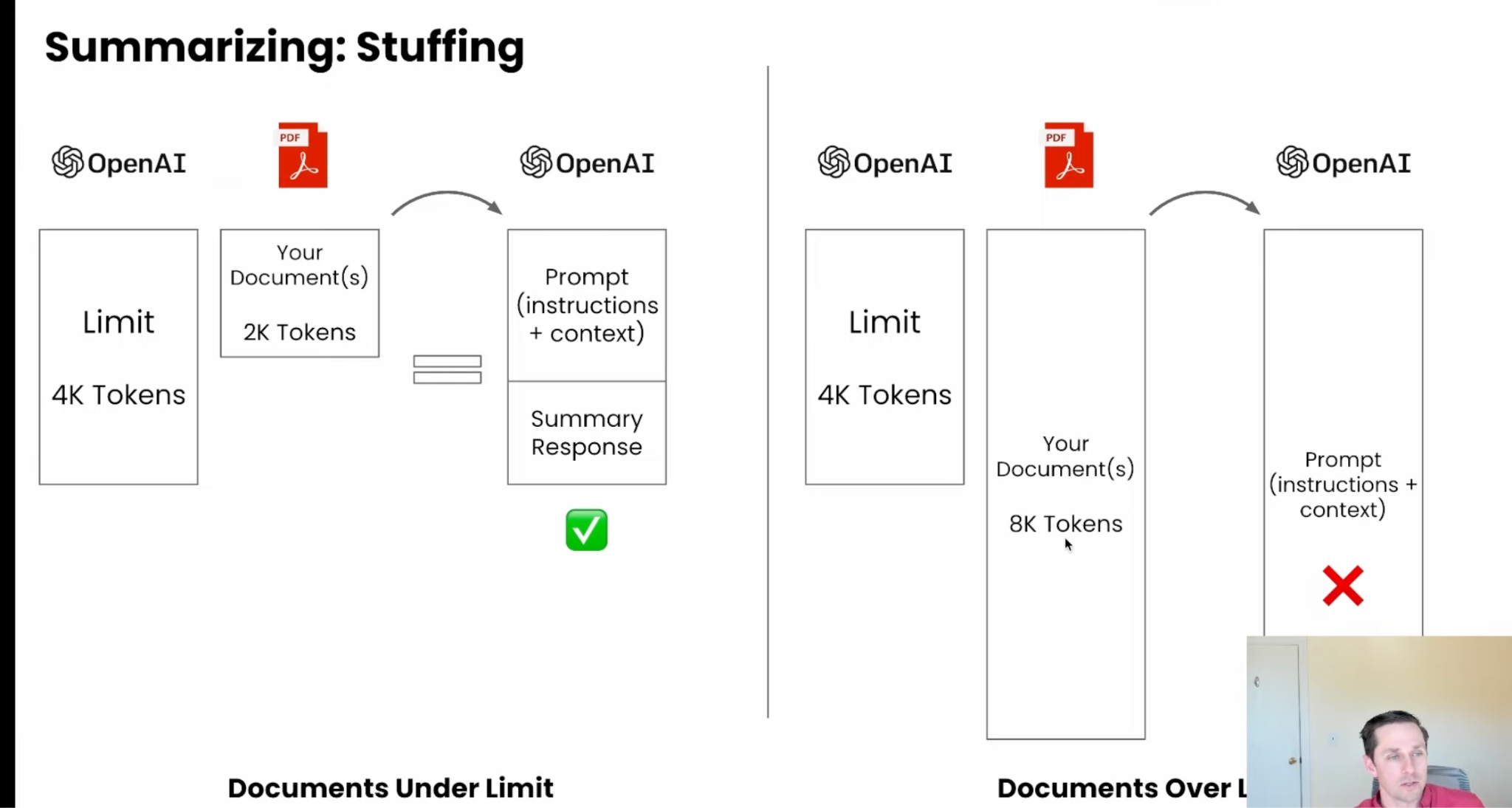
* Just like our most basic LLM and ChatModel calls earlier in Model IO, Chains have a basic building block known as an LLMChain object.
* You can think of the LLMChain as just a simple LLM call that will have an input and an output.
* Later on, we can use these objects in sequence to create more complex functionality!

**NOTEBOOK->**

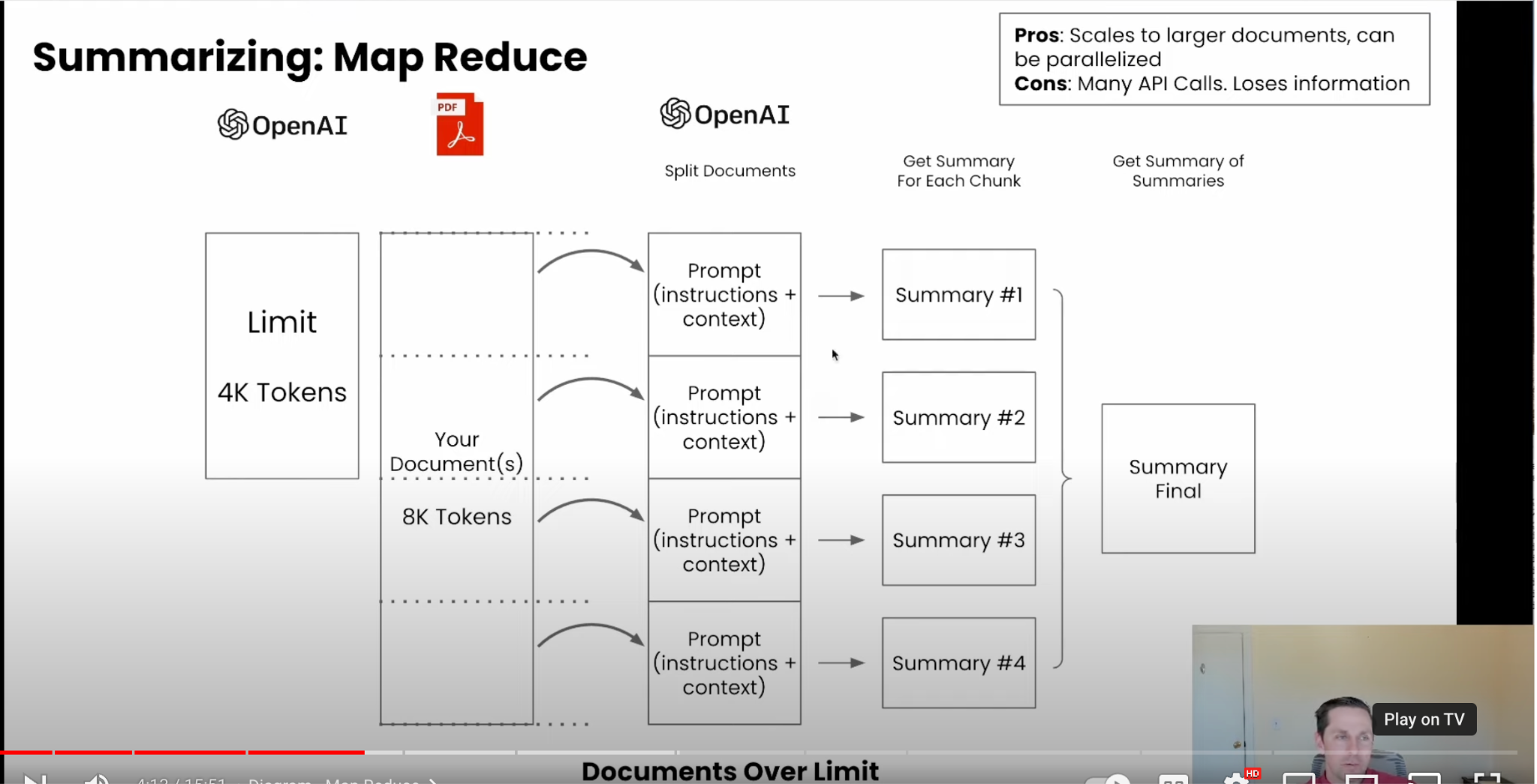
### **Foundational Chains**

* + LLMChain
  + SequentialChain
  + LLMRouterChain
  + TransformChain
  + MathChain

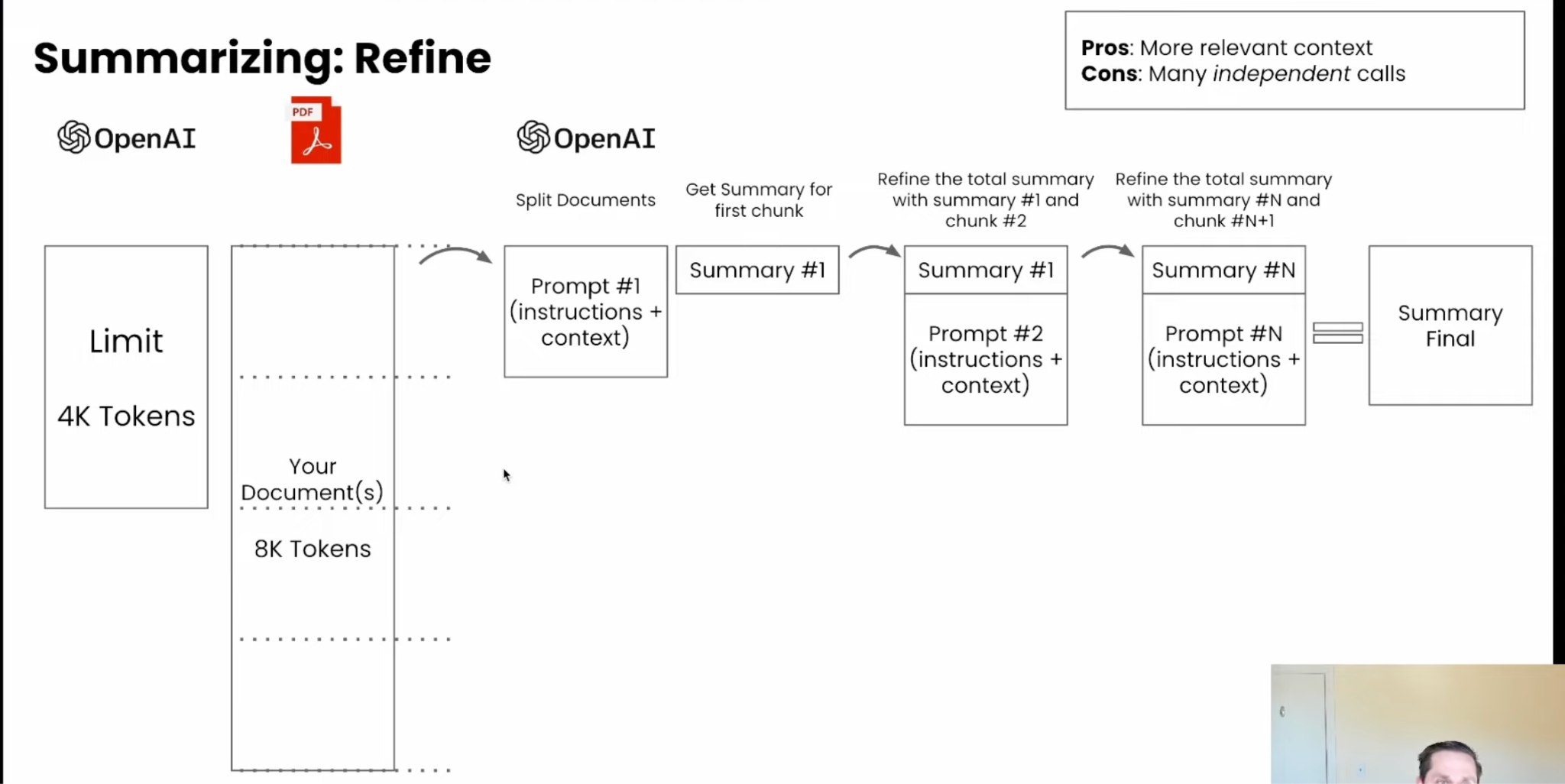
### **Document Chains:**



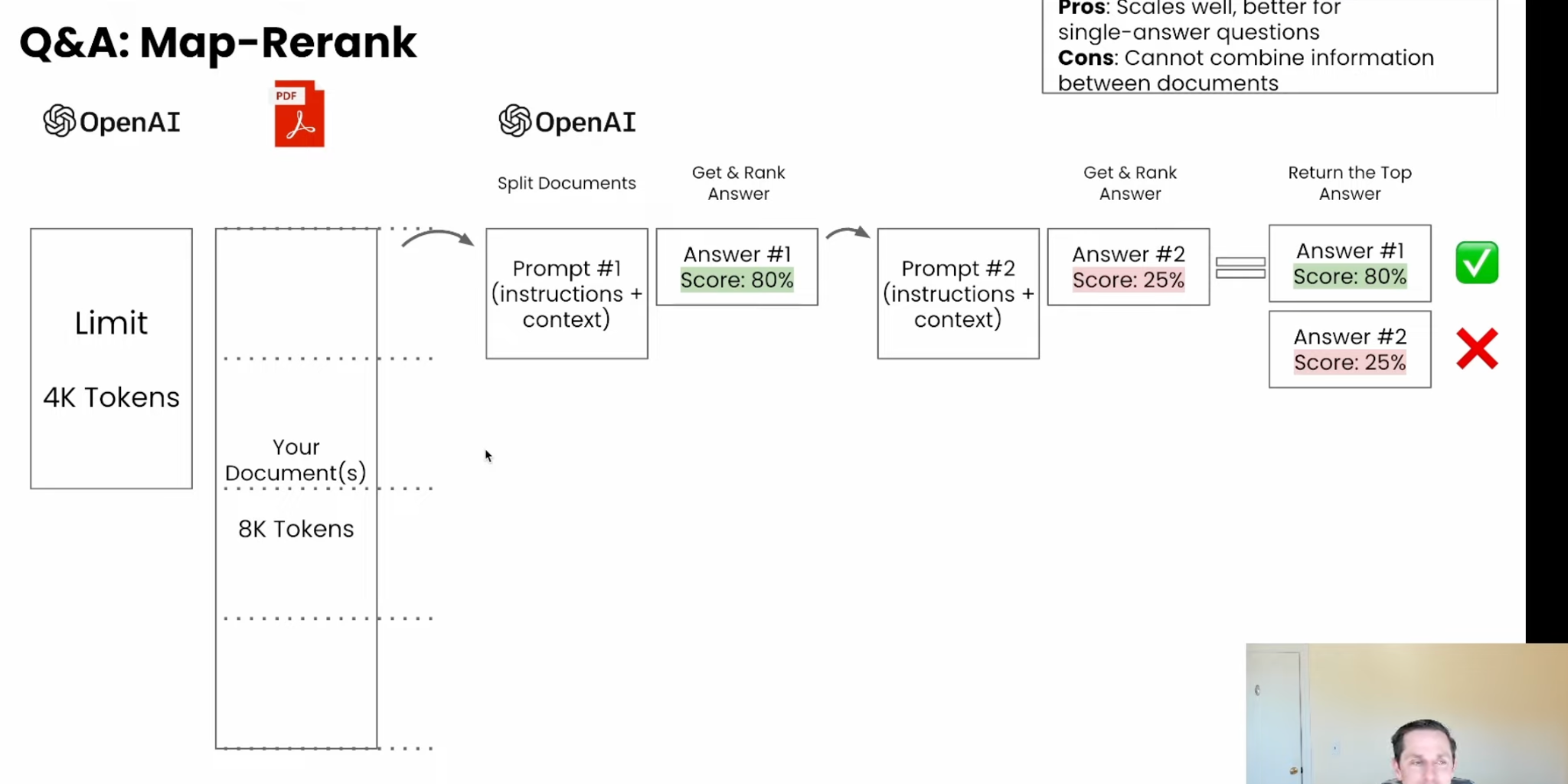
Map - reduce:



Refine:



Map-rerank:



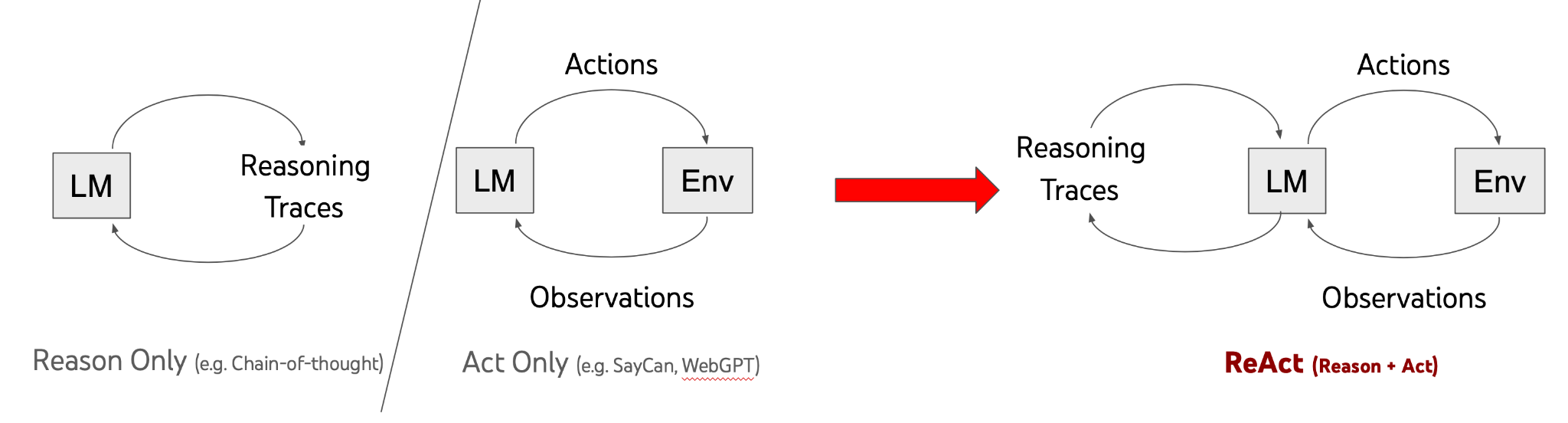
## **Memory**

* Memory allows your models to retain the historical context of previous interactions.
* LangChain Advantage:
  + Easily save historical conversations or results from LLMs or Chat Models and reload them for future use.
* When LangChain refers to the term “memory”, they are usually referring to keeping track of message interaction history.
* LangChain provides a few different options for storing a conversation memory:
  + Storing all messages
  + Storing a limited number of interactions
  + Storing a limited number of tokens
* It also comes with specialized chains that store a memory with other transformations, such as a vector store or auto-summarizing conversations.

### **Different Memories:**

* ChatMessageHistory
* ConversationBufferMemory
* ConversationBufferWindowMemory
* ConversationSummaryMemory
* Entity Memory
* Conversation Knowledge Graph Memory
* ConversationSummaryBufferMemory
* ConversationTokenBufferMemory
* Vector Store - backed memory

## **Agents**

* By combining what we already learned about Model IO, Data Connections, and Chains, we’ve already approached applications that are similar to agents that allow us to create more robust applications, Agents takes this one step further with the ReACT framework.
* At their core, agents allow LLMs to connect to tools (e.g. Wikipedia, Calculator, Google Search, etc…) and conduct a structured approach to complete a task based on ReAct: Reasoning and Acting.
* ReAct prompting can help an LLM reason about a task and perform correct actions based on observations:
* 
* Here is an example of a complex question that an Agent can answer:
* 

### **Agent Section**

* + Agent Basics
  + Agent Tools
  + Custom Tools
  + Conversation Agents