Contents

[GEN AI 2](#_Toc201919673)

[**Langchain:** 3](#_Toc201919674)

[**Why do we need Langchain?** 3](#_Toc201919675)

[**Benefits of using LANGCHAINS:** 5](#_Toc201919676)

[**Langchain Component or Langchain RoadMap.** 5](#_Toc201919677)

[**1. Models:** 6](#_Toc201919678)

[**2. Prompts:** 8](#_Toc201919679)

[**PromptTemplate:** 9](#_Toc201919680)

[**Structured output from LLMs** 11](#_Toc201919681)

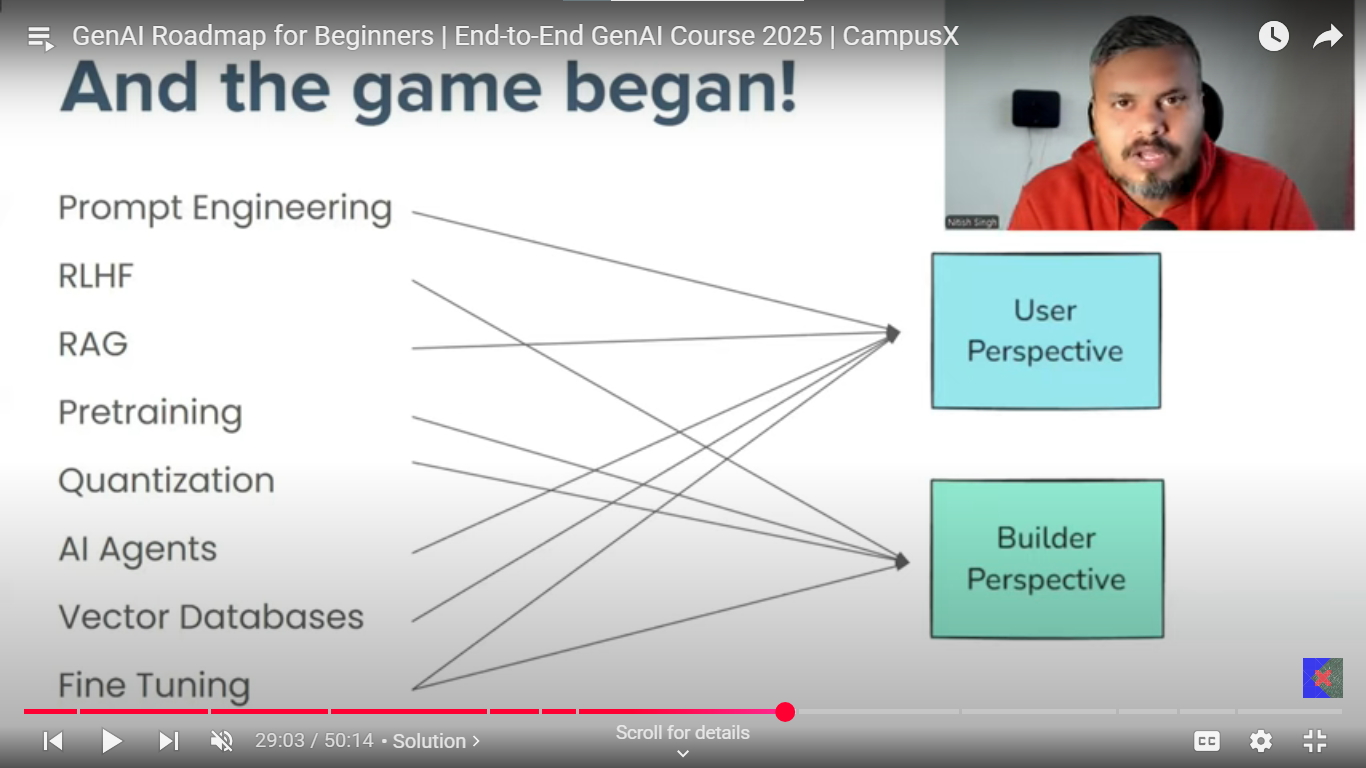
[**3 Chains:** 16](#_Toc201919682)

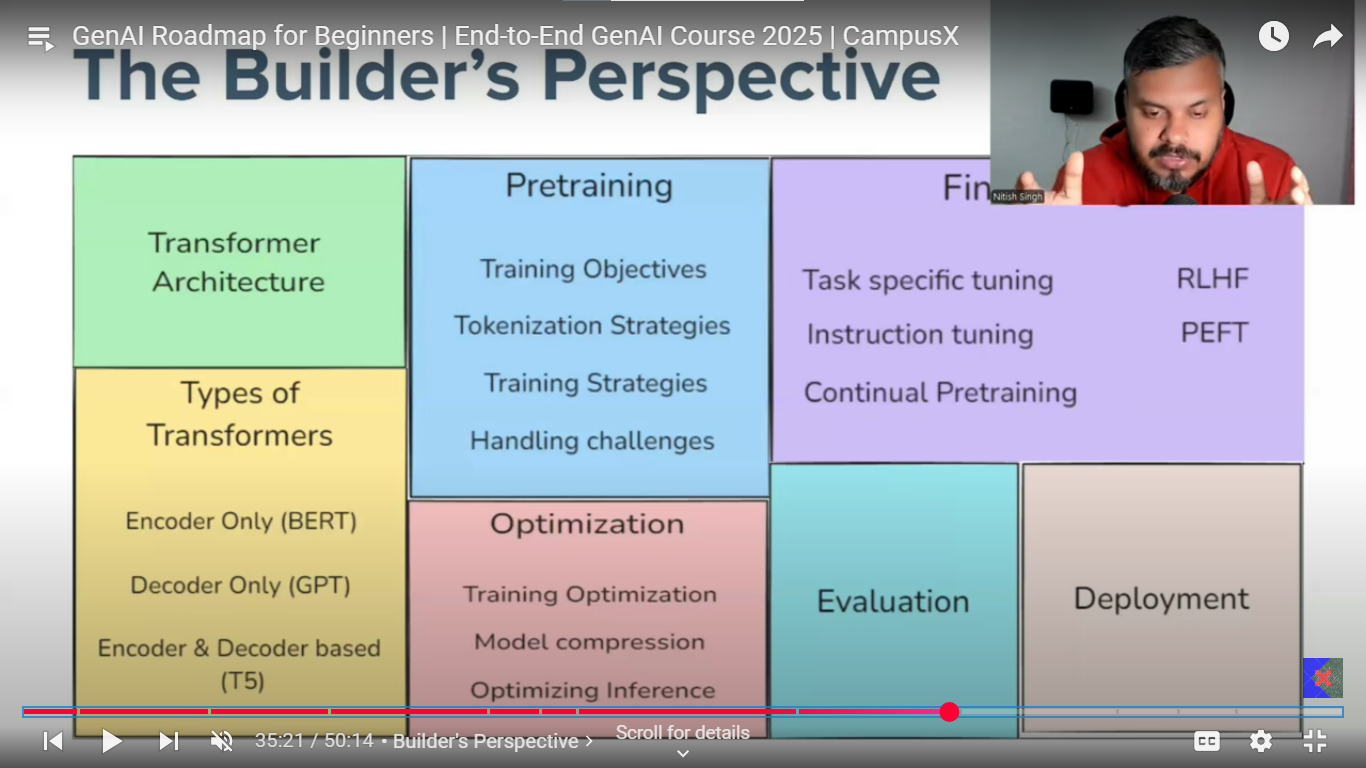
[**4. Memory:** 20](#_Toc201919683)

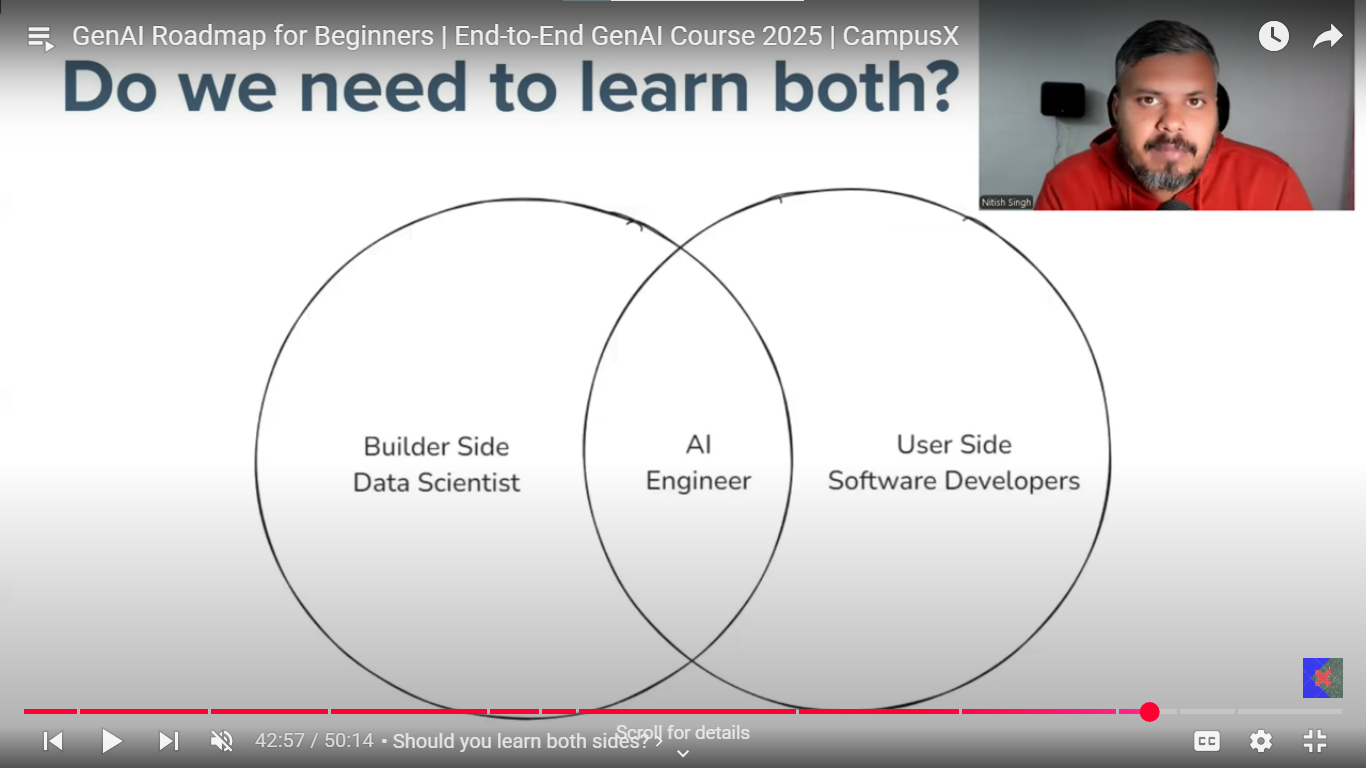
[**RAG:** 21](#_Toc201919684)

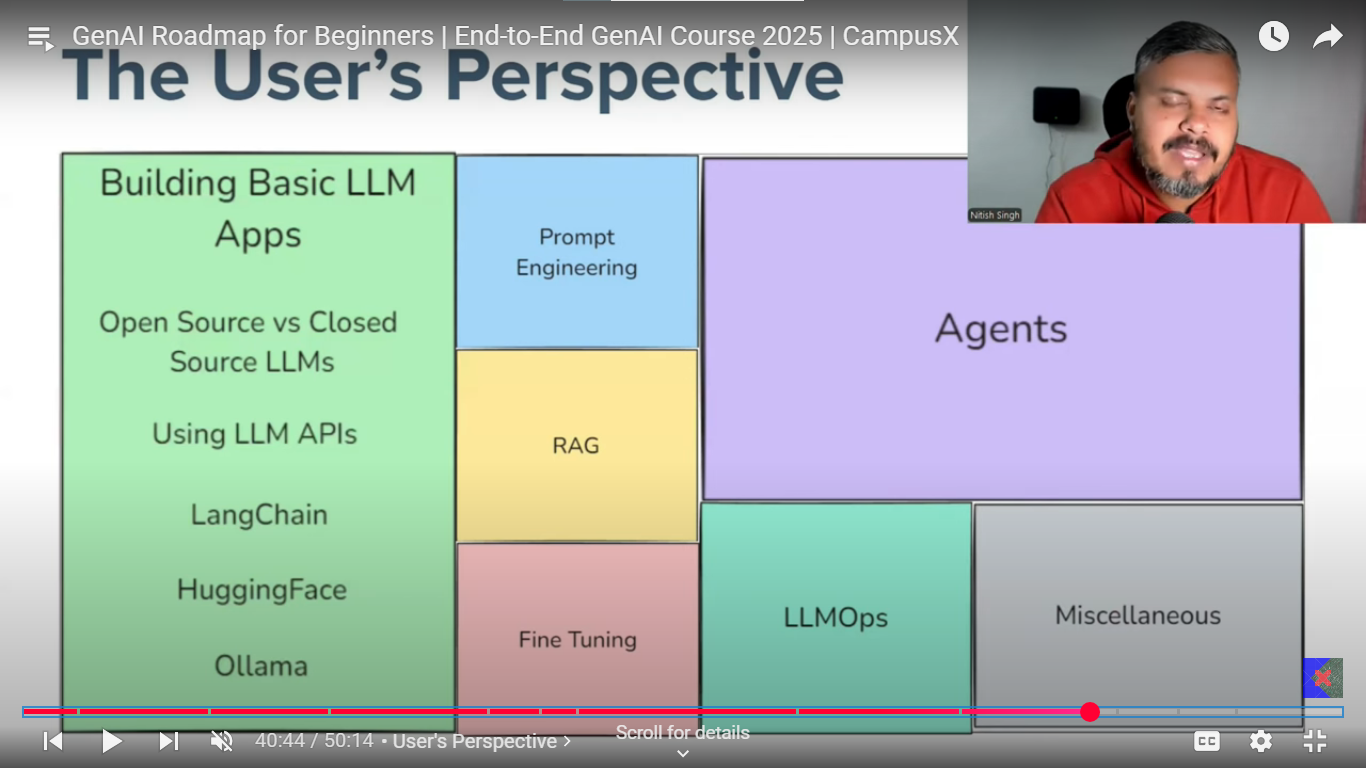
[**Agentic making in LangChain** 28](#_Toc201919685)

# GEN AI







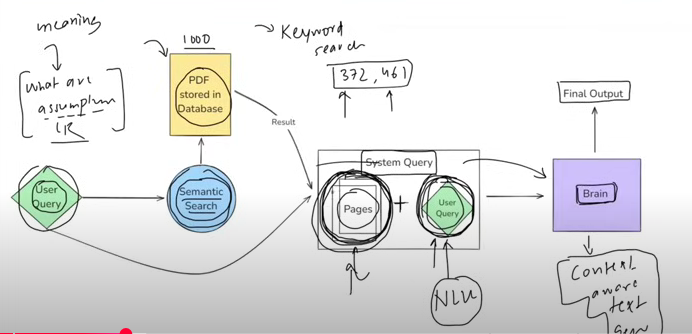


GEN AI app:

**Langchain:** open-source framework for developing applications powered by large language models.

## **Why do we need Langchain?**

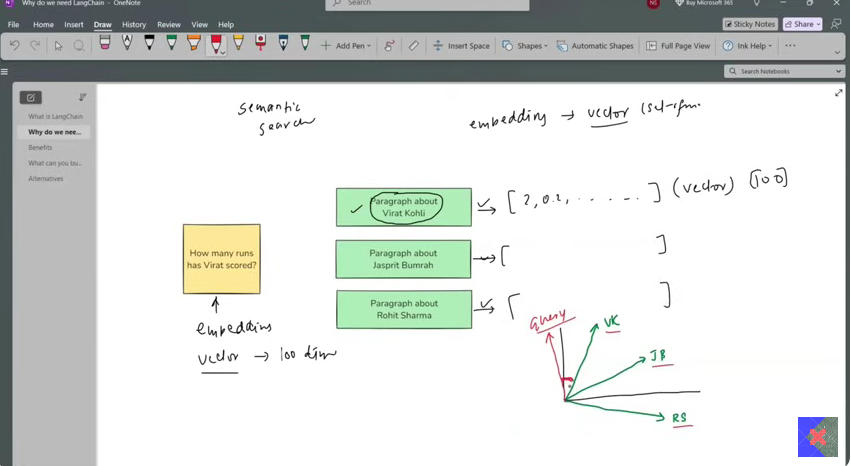
Examples of EBOOK where user can upload a book and interact with it by asking questions. Application answers the question from the book.



Here the user query “What are assumptions of the linear regression”, this user query is searched with the pdf stored in database, but the search is not just straight keyword search (which matches the exact keyword in the pdf dataset) instead semantic search is done to match the exact relevant text parts from the pdf.search which interprets the meaning and intent behind a user’s query rather than just matching the keywords.

Now this user query along with semantically related query combinly called as system query which is fed to the model (called brain) has the ability of natural language understanding to understand the system query i.e. user query and pages relations with meaning i.e. understanding the question finding it’s correlation with pages. It should also have the ability of context aware text generation.

**Semantic search: SEARCH** based on the meaning and context rather than just word matching, where embedding is done prior by various methods of the text to search and of the dataset as well, then the closest embedded vector is treated as similar one.



Final product: How is the flow?

* Start with the PDF upload on the AWS(S3), load the PDF with data loader and then splits the content of pdf into different pages with text splitter generate the embedding of all the pages, suppose there are 1000 pages then generate the 1000 pages of embedding. store all the embedding into database. Then user query is taken which is after embedding is match with the pages embeddings stored in databases with semantic search, this gives the top related pages(content). Along with this related content and user query forms a system query. This system query is fed to the brain which generates the context aware data.

Now the main issue in the above product is developing this brain. With transformers and new work in NLP, LLMs are the best brains. But there are few problems here:

* Developing the LLMs, which itself is a huge task, but you can use already developed LLMs’.
* Secondly, to run this LLMs on the local machines or maintaining your servers for it is extremely complicated, costly and computationally heavy task, API for different models can be used in this case which are made available for different LLMs runs which on their severs.
* Third issue in the above product is the managing number of component with different data processes(tasks) which takes place, this process of using different component like LLMs, text splitters, databases, AWS with different processes like embedding, semantic search makes the entire flow management very complex, that’s where the LANGCHAIN comes into picture which makes managing these processes and component very smooth. Explains the use of LANGCHAIN.

## **Benefits of using LANGCHAIN:**

* Langchain uses the concept of chains where sequence of interconnected component or steps that are executed in a defined order or process.
* Agnostic development which means model can be changed by just changing the LLM model simply.
* Complete ecosystem: Langchain provides complete ecosystem of all the different tools to select from.
* Memory and state handling: Mechanism that allows chains and agents to remember and utilize the previous interactions within a conversation of workflow.

What can you build with LangChain?

1. **Conversational chatbots:** AI chatbot
2. **AI knowledge Assistance.**
3. **AI agents used**, chatbots on steroids, can do task for you instead of just conversations like book a ticket or something.
4. **Workflow automation**
5. **Summarization / Research Helpers**

## **Langchain Component or Langchain RoadMap.**

1. Models
2. Prompt
3. Chains
4. Memory
5. Indexes
6. Agents
7. **Models:** This component of the langchain is the core interface through which you interact with AI models. So, technically the models give you the ability to connect to different APIs of the different LLMs with very simple code implementations.

Let’s understand the issue in detail, initially the building chatbots was the main issue in NLP, where to make the chatbot which has NLU (natural language understanding) ability along with ability of context aware text generation. This issue is solved by LLMs. Then the second issue arises which is to develop these models and deployed it on servers and access this huge model, so the LLMs developers deployed this model on their servers and make it available the access of this LLM with API connection, but connecting to each LLM is so different in a way we write a code to connect to the API. And now this issue is solved by the langchain which kind of made the standard procedure of hitting this APIs and connecting to any of the LLM model is now very much easy and standardize.

Put the remaining part from **the local**

**Setup:**

**Mentioned in local**

**Open-source models:** Open-source models are freely available to download, modified, fine-tuned and deployed without restriction from the central provider, unlike closed source.

**Open-source AI models** are freely available for anyone to use, modify, and deploy. They offer maximum control and customization, allowing organizations to adapt the models to their specific needs and maintain full data privacy by hosting everything on their own infrastructure. However, using open-source models typically requires more technical expertise and resources for setup, maintenance, and scaling. In contrast, **closed source AI models** are proprietary solutions provided by companies as a service, usually accessed through APIs. These models are easier to deploy and come with professional support, but they give you less control over customization and data privacy, as your data is processed on the provider’s servers. Additionally, closed source models often involve ongoing costs based on usage or licensing, and you may be dependent on the vendor for updates and continued access.

Open-source model is not uploaded on the developer servers rather they are free to download, and you can find these on hugging face, largest repository open-source models. Hugging face also provides the API interface which lets you connect to many open-source models without downloading these models.

Now we are doing 2 things, Use API inference of hugging face and download

a model on local machine.

**Temperature** in models, decides the **creativity and deterministic** nature of the model output, if the temperature values are set to 0 then in that case most deterministic and probable values are selected which lacks diversity whereas for values towards the 1 are more random and non-probable values are selected leading to creative and diverse response each time.

**Embedding model Application:**

**Document similarity search application.**

In this application you have document with statements and based on the query you to return the most similar statement from the document.

Get the embedding of the documents, then get the embeddings of the query compare which one is closest by co-sine similarity and return the closest one as the result.

Implementation in code.

### **Prompts:**

Prompts very much decide the kind of input you might be getting from the LLMs’, just the small change in the input could change the output of the LLM by large factor.

There are different kinds of the prompts:

1. Dynamic and reusable prompts (Structured/Template Prompts):

These prompts use template with variables that can be filled dynamically making then adaptable and reusable for many tasks. It has dynamic placeholders for custom inputs.

e.g. from langchain. prompts import PromptTemplate

template = "Summarize the following text: **{text}**"

prompt = PromptTemplate.from\_template(template)

1. Role based prompts: This prompt assigned specific persona or role to the LLM, guiding its response.

E.g. template = "**You are a helpful assistant**. Answer the user's question: **{question}**"

prompt = PromptTemplate.from\_template(template)

1. Few shots prompting: It provides few examples to model in prompts to perform a similar task. Model does not get trained on this example to understand this example and make prediction it, rather model understand the context within the prompts and use it for generation of the required output based of the similar context as in earlier examples.

* E.g. from langchain.prompts.few\_shot import FewShotPromptTemplate
* examples = [
* {"input": "What is the capital of France?", "output": "Paris"},
* {"input": "What is the capital of Germany?", "output": "Berlin"}
* ]
* example\_prompt = PromptTemplate (input\_variables = ["input", "output"], template="Q: {input}\nA: {output}")
* prompt=FewShotPromptTemplate(examples=examples, example\_prompt=example\_prompt, suffix="Q: {question}\nA:", input\_variables=["question"])

Prompts can be text based as well as multimodal (image, sound, video input) prompts.

Static and dynamic, dynamic and reusable prompt templates designed to be flexible and adaptable can be used for different inputs. In static prompts we don’t have high level of control over prompts given as an input to the LLMs, which can lead to inconsistent and low-quality responses by the LLMs. So, to make sure the LLMs give best and consistent responses dynamic prompts with prewritten format is used in which dynamically keywords are filled. Code in VS code.

#### **PromptTemplate:**

used in langchain for structured way of creating dynamic prompts by inserting variables into a predefined template, instead of hardcoding prompts, it allows to define the placeholders which can be filled at the runtime with different input.

Why do we need **PromptTemplate** in langchain, simple text format or f string would have done the same thing. There are 3 main reasons for it:

**Default validation:** You get instant validation in with prompt before sending the prompt to the LLM itself, with the parameter called as the validate\_template = True, which basically checks for all the placeholders and see if all those values are filled or not?

**Reusable:** you can define the prompt once in Json and can be used multiple times by just calling the prompt.

**LangChain Ecosystem** works very well with langchain ecosystem, like chain and all, like you can integrate invoke of the template with model invoke with chains.

**Messages**

Building a chatbot**,** simple code is in VScode, the simpler code is you run while loop and send each input as a prompt to the LLM and print the output till you get exit from the user.

But out model cannot keep track of the previous prompts and responses of LLM. So, one solution for the same is you try to store every input and each response in a list and pass a document i.e. list of all the inputs and output with new query as well is passed as a document to the LLM for each new query, so for each new response to generate it does not just have query but the entire history as well.

But as the conversation goes bigger and bigger it is difficult to recognize which ones are queries and which ones are responses by the LLM. So, you maintain the dictionary of prompt and response for each instant and likewise send the history in this format to the user.

Types of the Messages?

* Human Messages: Prompt
* AI Messages: responses
* System Message: Initial input/prompt given to the LLM to set the behavior, tone or persona of the chat model.

Messages can be specified for single prompt as well for multiple prompts i.e. conversations by ChatPromptTemplate. (method do is bit different).

Message Placeholder: To load the history of previous chats, message placeholder is used. Through which previous history is made available to model.

This is it for this course of the LANGCHAIN but there is far more to it than just this.

#### **Structured output from LLMs**

Refers to the practice of constraining the output of the LLM models to certain format, so that it can be understood by other systems. The output generated is consistent, machine readable, easier to integrate with downstream applications.

Where is structured output used?

* Data extraction: data details like YOE, CGPA, Skills could be extracted from the resume and stored to get the specific, structured information could be use down the line for different application.
* API Interactions: Ensuring LLM outputs match the expected format for downstream API calls
* Agents: data to the agents could be given in structured form to understand what to do on what info.

There are two types of models in LANGCHAIN, few can provide structured output, and few cannot. Open AI models usually can provide the structured output.

CAN: with\_structured\_output

CAN’T: output parser

##### **With\_structured\_output:**

Ways to specify the format:

* Typed dictionary
* Pydantic
* Json\_schema

1. **Typed dictionary**: dictionary where you specify the key with is constrained data type. This is not the hardcore constrain it is only for representation purpose, other formats may be specified without throwing the errors.

From typing import TypedDict

Class Person (TypedDict):

Name: str

Age: Int

New\_person: Person = (‘name’: ’Nitish’, ‘age’:35)

Print(new\_person)

Implementation in the code.

There is one more thing which is annotation which could be specified along with the datatypes for so, that model understands the description in many details.

How does it work? model takes the class as an input in the with\_structured\_output which basically makes the model understands the in what format the output is expected, where the key is the expected thing from the model with the value as the datatype of the key’s output. Where value could be explained in bit more details along with just datatype with another module called annotation. Likewise, even the pair if optional could also be specified and so does the list of datatypes if the outputs expected are not single outputs.

**Intuition**, TypedDict approach, the keys and their annotated descriptions together form a detailed, field-wise prompt for the LLM, specifying both what the output should look like and what each field should contain.

**Why do we need with\_structured\_Output, specifically?**

* Simply we can tell in prompts which format we want the output into, but why do we still need that to prior to prompt invoke?
* **No validation of the output with automatic parsing:** LLM might return the output in the required format there won’t be any validation check done on that output and there won’t be any automatic parsing.
* **No consistency in output:** output won’t be consistent

**How does it work in the backend.**

When you use a TypedDict with with\_structured\_output, the TypedDict defines the expected output structure by specifying keys, their types, and optional descriptions. The backend extracts this schema and combines the keys and descriptions to form a clear, detailed prompt that guides the LLM on what information to generate and how to format it. The model then produces a structured JSON response matching this schema. Finally, the backend parses this JSON output into a Python dictionary that adheres to the TypedDict, providing you with clean, validated, and easy-to-use structured data.

When you use with\_structured\_output(schema) in LangChain, it configures the language model to return responses in a specific format defined by your schema. This is done by setting up the right API parameters (like JSON mode or function calling) for models that support structured output. After the model generates a response, LangChain automatically parses and validates the output according to your schema, so you get a ready-to-use structured object instead of raw text.

**One of the above two is valid.**

**The flow** is as such:

Sure! Here are five simple points:

1. You define the output format using a schema (like a Pydantic model).
2. LangChain uses your schema to set up how to parse the model’s output.
3. The model generates its output (usually as text or JSON).
4. LangChain converts and checks the output using your schema.
5. You get a ready-to-use structured object instead of raw text.

Options in schema definition is, optional, annotation etc, all of these are used only for output parsing and do not have any role in generation of the output by the LLM.

**Annotations** (like field descriptions) are mainly for developers to document what each part of your schema means, and sometimes for generating prompts or error messages—they are not used by the LLM or output parser to “understand” what you want. If you want the LLM to generate output in a specific way, you should put clear instructions or keywords in your prompt; the parser only uses the schema’s structure (like field names and types), not the literal meaning of the annotation text.

Share

Export

Rewrite

1. **Pydantic:**

**What?** It is data validation and data parsing library for python which ensures that the data you work with is correct, structured and type safe.The main difference from the typedict it imposes hardcore constrained and throws error when the datatypes and constrains are not matched.

**TYPE COERCING:** Type coercing, also called type coercion, is when a programming language automatically converts a value from one data type to another during an operation or comparison, without the programmer explicitly asking for the conversion.

**Why?**

It is same as typedict just the bit harsher on the validation. LLMs naturally produce unstructured or free-form text, but applications often need predictable, structured data. Pydantic ensures that outputs from LLMs are validated, formatted, and easy to process—making integration with other systems reliable and error-resistant. This is especially important for APIs, data pipelines, and automated workflows.

**How?**

**Let’s first understand how Pydantic works,**

* **Install Pydantic:**  
  First, you install the Pydantic library using a package manager like pip.
* **Create a Model Class:**  
  You create a class that inherits from BaseModel. In this class, you define your fields with type hints and add constraints like default values, field options, value constraints, and descriptions.
* **Prepare Your Data:**  
  You collect your information as a dictionary (key-value pairs).
* **Validate and Create a Pydantic Object:**  
  You instantiate your model class by passing the dictionary (typically by unpacking it with \*\*), which creates a Pydantic object.  
  Pydantic automatically checks that all your data matches the defined types and constraints. If anything is invalid, it raises an error.
* **Use the Validated Object:**  
  If validation passes, you get a Pydantic object with all your fields and methods.  
  You can use this object directly or convert it to a dictionary or JSON for further use.
* Main difference is in syntax of how you assign the datatype and particularly in annotation/description rest is similar almost.
* **Ref Pydantic demo for code understanding.**

**How it is used in LangChain:**

In LangChain flow, you create the Model Class and then just pass the Model Class to the with\_structured\_output. That’s it.

1. **JSON**

* What: like above two
* Why: when working on multi language project.
* How:

A dictionary is specified with 5 kinds of information, Title, Description, Type, Properties, Optional. Where each is a dictionary.

**Output Parsers:** used in case where the LLM models are not able to generate the structured output.

**There are four types of Output parsers:**

* + StrOutputParser
  + JsonOutputParser
  + StructuredOuputParser
  + PydanticOutputParser
    1. **StrOutputParser:** returns the string format return, then the obvious question is why we even use it, makes sure the return is always string and using the string output parser we can **build chains**, which is not possible else, where we must manually extract the outputs**.**

**langchain\_core is that mini library where the most used langchain classes are kept.**

* + 1. **JSON output Parser:**

**Why:** A JSON Output Parser is used to convert the raw text output from a language model into structured Python dictionaries, ensuring that the data is predictable and easy to work with. This structured approach improves reliability, makes it easier to automate workflows, and simplifies integration with other tools, databases, or systems that require clean, machine-readable data

**What:**

**How:** JSON output parser object is created which is passed after the model to get the parsed output. In template a **partial\_variable** could be defined called as **format\_instruction** whose value is preset and not on runtime value is passed.

**Structured Output Parser:**

Used to extract the structured JSON data from LLM responses based on the **predefined field Schemas**.

Schema is defined with Response\_Schemas object with the use of the name and description.

Flow in the LangChain pipeline is almost same except the definition of the parser object.

* + 1. **PydanticOutputParser:** Structured Output Parser does not have the ability to validate the schema, it can only define the schema. That’s where this parser is used and it can validate the output, with parser formed with Pydantic object. Pydantic object is created with the field like normally done and it is then passed while forming the parser object. Once the object is formed the rest process is similar.

### **Chains:**

Important elements of the langchain, so important that langchain is named cause of chains. A concept refers to sequence of automated steps or components called as links that work together to process of users’ query and generate context aware response from the LLM. Each link represents specific action or transformation whose output is linked/connected to the next link.

Various kinds of applications can be made by use of langchain. A simple pipeline can be made with the help of chains where the input of the one component is fed to the second element.

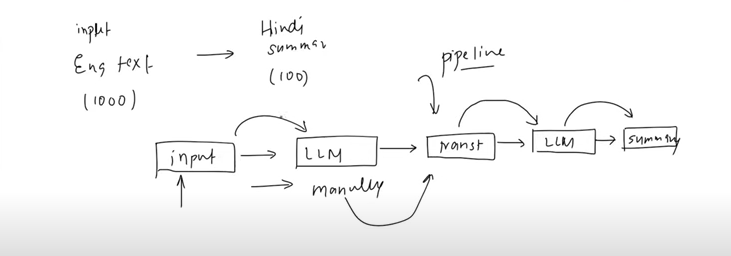
There are three kinds of the Chain:

Sequential

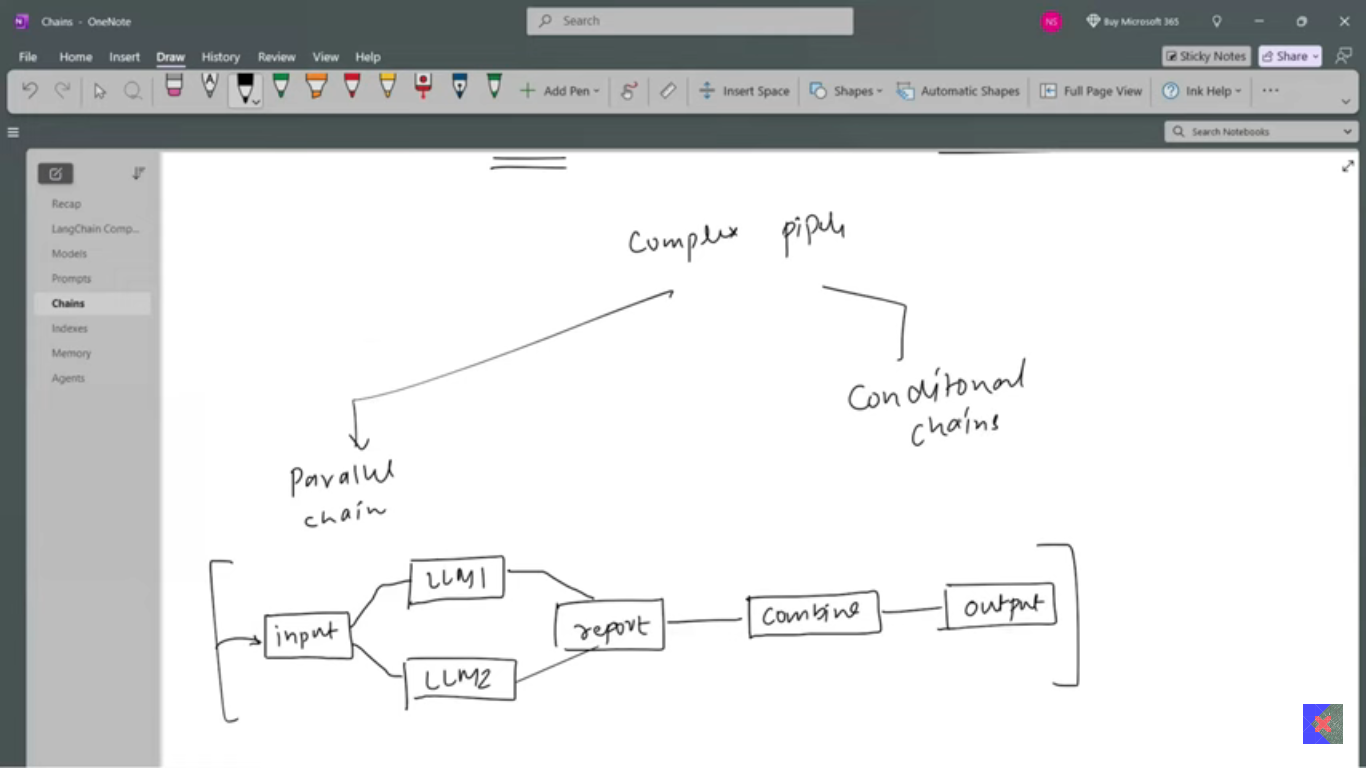
Parallel

Conditional

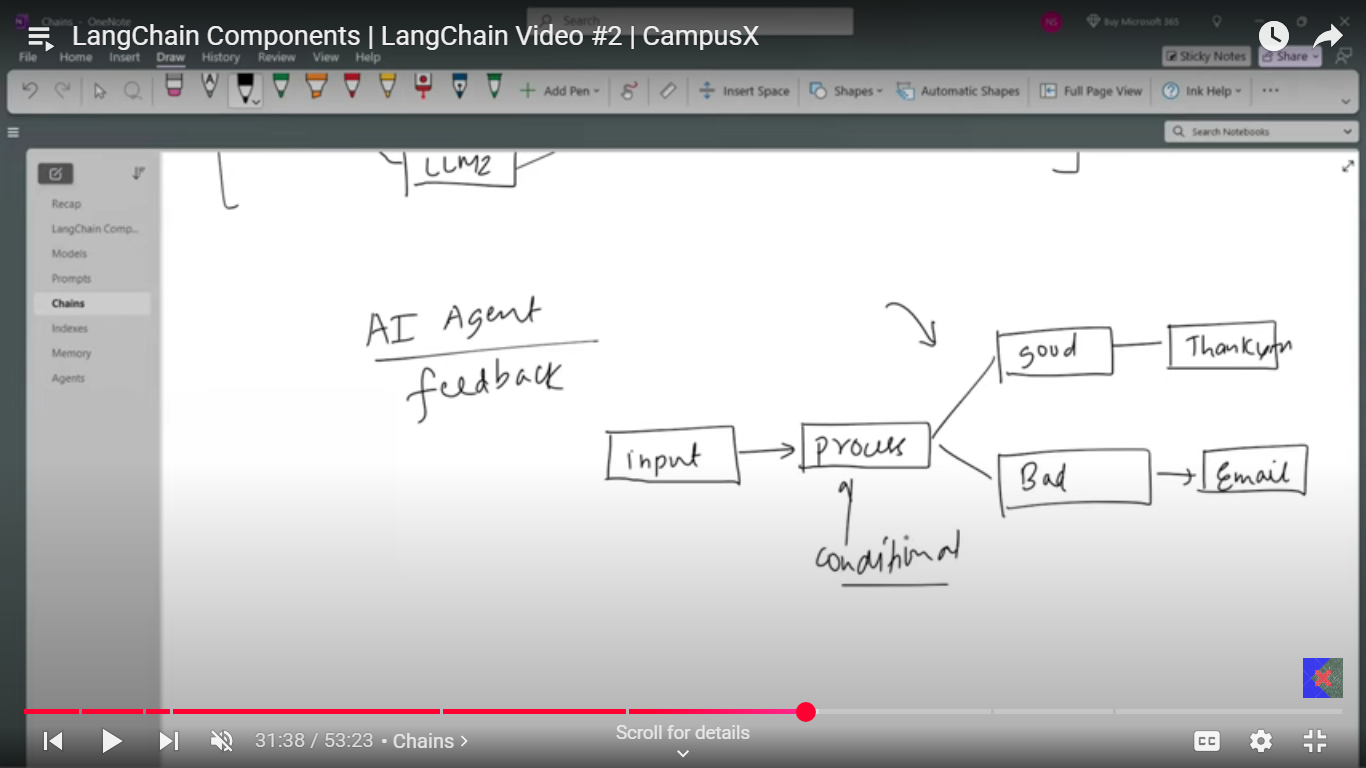
E.g. Building bot to take 1000 words English para and return a 100 words hindi summary.



e.g. Suppose you want to generate a response based of the multiple LLM models. Parallel conditional pipeline can be made.



e.g. AI Agents feedback and response, based on the AI agent’s processes, if the feedback is good then return the greetings whereas the feedback is bad you send email to our customer executives’ team.



1. Chains Sequence Components

A chain is a sequence of modular components—such as prompt templates, language models, and output parsers—connected. Each component performs a specific function in the workflow.

2. Input Starts at the First Component

When you run a chain, you provide your input (like a user question or data) only to the first component. This is the entry point for your data into the chain.

3. Output Passes Step-by-Step

The output of the first component becomes the input for the second component. This pattern continues, with each component receiving the previous one’s output, until the end of the chain.

4. Unified Method: .invoke()

LangChain standardizes how each component is called by using the. invoke() method. Regardless of whether the component is a prompt template, a language model, or a parser, the chain always calls. Invoke() on it.

5. Internal Logic per Component

Inside each component, the .invoke() method handles the specific logic needed:

For prompt templates: .invoke() formats the input into a prompt.

For LLMs: .invoke() sends the prompt to the model and gets a response.

For parsers: .invoke() parses the model’s output (and internally calls .parse()).

6. No Manual Handling Between Steps

You don’t need to manually extract, transform, or parse data between steps. The chain automatically manages data flow and method calls for you.

7. Final Output

After the last component processes the data, the final output is returned to you. This output is ready for use in your application, already formatted or structured as needed.

Let’s understand how LangChains happened, LangChain provided the ability to connect to different LLMs and access those LLMs though APIs or local servers. As the library got developed, langchain now has classes and components for each kind of task and step needed in building LLM application. So, now using those classes and components for each sub-task separately in the flow LLM application can be built in langchain. Developer of langchain observed that there are lot of tasks are which are very repetitive across various kinds of applications, so they found a way to simply the implementation of these task with the help of chains, where a unique chain for each task could be connected and have an execution of these tasks. Where a class would take the components and have execution of this components and returned the output.

But few issues happened,

Codebase got too big and difficult to maintain

Too many chains for each specific task integration.

And now we have big codebase with lot of chains which made difficult to maintain the codebase and learning curved became steep. Where the intent of letting AI engineers connect different components seamlessly got completely wrong and now it’s headache.

What was the reason, the individual components developed by langchain were not standardized, they are developed individually and behave differently. Like format for prompt, invoke for model, parse for parser etc. and hence for each custom component we have custom code or function cause the method/function of the components are different for the use of that component. Like to connect LLM and Prompt they created LLMChain and for retrieval RetrievalQaChain.

The solution was to build standardized components for seamless connection. And this is done by Runnables.

What are Runnables:

They are the unit of work.

Four ways Runnables can be defines:

Unit of work: where each component can do specific work by taking the input, process and returns the output.

Common interface: (Invoke, Batch, Stream)

Connect: One Runnable connects to the other Runnable.

Returns a Runnable: Each runnable connection returns a workflow which is a runnable itself.

LEGO blocks resemble the Runnables.

Where each LEGO block is a unit of work with unique purpose, where each piece has same connected characteristic, connecting one piece to another and finally the new connection formed is also has similar characteristics.

The most important idea behind runnable in LangChain is that the standardized.invoke() method is implemented for all modules via the Runnable abstraction class, creating a single, common way to execute any component—whether it’s a function, language model, retriever, or parser. This unified method means you can interact with all parts of your workflow in the same way, making it easy to connect, chain, and compose different tasks. By enforcing this standard interface, LangChain lets developers build complex AI pipelines where every module can be managed, combined, and executed predictably and consistently, greatly simplifying the development and maintenance of advanced language model applications.

Runnable Second part

1. **Memory:** You’ll learn in LangGraph.

### **RAG:**

Technique which combines information retrieval with language generation, where a model retrieves relevant documents from knowledge base and then uses then uses them as context to generate accurate and grounded responses**.** Where basically a knowledge base of your is used to retrieve the information which then used to generate the text from the LLM.

RAG works by first retrieving relevant information from your own dataset that relates to the user's query. This retrieved information is then combined with the original query and sent to the language model, which uses both to generate a more accurate, informed, and context-aware response. This approach allows the model to leverage up-to-date and specific knowledge beyond its original training data.

There are four Components of RAG based Applications:

* **Documents Loaders**
* **Text Splitters**
* **Vector Databases**
* **Retrievers**

**Documents Loaders:**

Components in langchain used to load data from various sources into a standardized format (usually a document object) which is then be used for chunking, embedding, retrieval and generation.

Different format documents are loaded into a common document object format which looks like,

Document (

Page content = “……...”,

Metadata = {}

)

**Text Loader**: Simple and commonly used document loader in LangChain that reads plain text and converts them into a Documents object.

Use cases: Ideal for loading log chats logs, scraped text, transcripts, code snippets or any plain text data into a langchain pipeline. But works only with text files.

**PyDocument Loader:**

Till retrievers is in the local files. **Update these.**

**Now what is RAG and Why do we need RAG?**

LLMs are trained on the huge data where they store information in their parameter’s values. So, any query is solved through these parameter values, but it has three main drawbacks:

* **Private Data:** RAG allows organizations to use their internal or proprietary data securely without having to retrain the underlying language model. This means sensitive or business-specific information can be incorporated into responses while minimizing privacy risks.
* **Recent Data:** RAG enables access to up-to-date information by retrieving from external or frequently updated sources. This overcomes the limitation of language models being trained on static datasets, which can become outdated.
* **Hallucination:** RAG reduces the likelihood of hallucinations (plausible but incorrect or fabricated answers) by grounding responses in retrieved facts and information, rather than relying solely on the model’s internal knowledge.

Ways to solve this problem and how do you make sure your current/private data in hand is worked by the LLMs, there are two solutions to it,

* + - 1. **Fine tuning:** Training an already trained model on your private data, so basically on technical level you add few layers to your model, where model learns the patterns from your private data with weight updates of your model. It uses the principles of transfer learning where the already trained model on general data (understanding the general data patterns) with combinations of fine tuning on the new private, specific data. Types of fine-tuning techniques:
* **Supervised Fine tuning**
* **Continued Pretraining (unsupervised training)**
* **RLHF (Reinforcement learning with human feedback)**

Let’s try an understand the supervised fine tuning:

* Collect supervise data, (Prompts with desired output)
* Choose method: Full parameter, LoRa, QLoRA.
* Train for few epochs: keeping the base wights freeze, train all the weights, train only the subset.
* Evaluate with metrics, or exact matches etc.

Fine tuning does solve the problems mentioned above, but comes with certain drawbacks as well:

* High-cost training.
* Complexity in training this huge model.
* Overfitting risk: As it is trained on the very small dataset, so the fine-tuned model could be overfitted.
* Catastrophic Forgetting: Model could forget the earlier trained context as part of the fine tuning on the specific dataset.
* Loss of flexibility.

It is not the most go to technique to solve this problem especially for the application which are constantly updating and required often fine tuning.

There is one more technique which does this is,

**In-context Learning:**

It is a core capability of the models (huge LLMs like gpt3 and onwards) where model learn to solve the problem by just seeing the examples in the prompts only. So, you basically give the examples in the prompts called as one-shot prompts while asking the question in query as well.

In context learning is the **emergent property** of the LLMs, it is the property which appears in the model/system when they reach certain level of complexity - even though it was not explicitly programmed into the model/system.

**Language models are few shot learner:** When we say language models are "few-shot learners," it means they can perform new tasks or generate accurate responses after being shown only a small number of examples—sometimes as few as three to ten—rather than needing thousands of training samples as in traditional machine learning

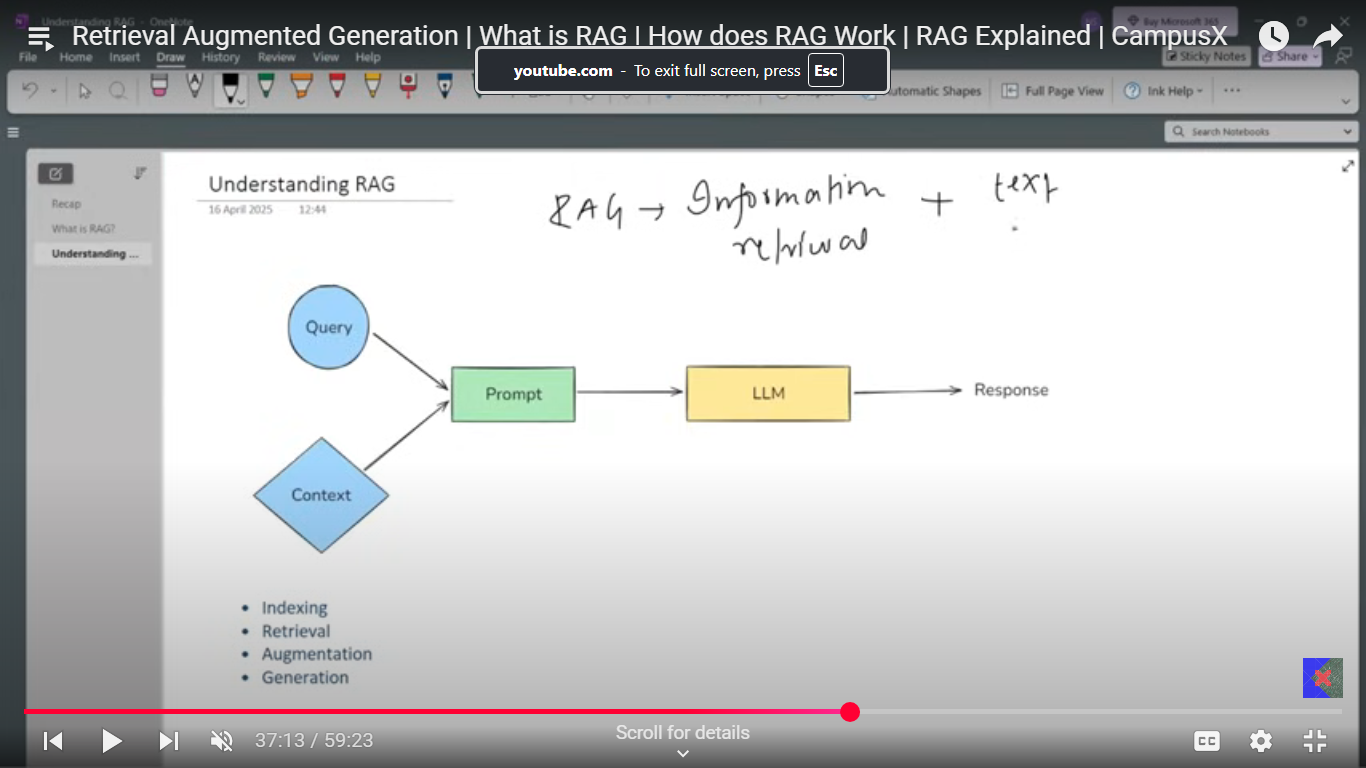
In a **RAG** system, the LLM works like a helpful assistant that receives your question along with relevant information pulled from external sources, such as documents or databases. The model reads both the question and the provided information together, then uses its language understanding to find connections and generate an answer based on what’s in the retrieved content. It doesn’t memorize or learn from this information; instead, it uses the data only for the current response, much like someone looking up an answer in a book and replying to you on the spot.

The main difference between in-context learning and Retrieval-Augmented Generation (RAG) is how the model gets the information it needs to answer your query:

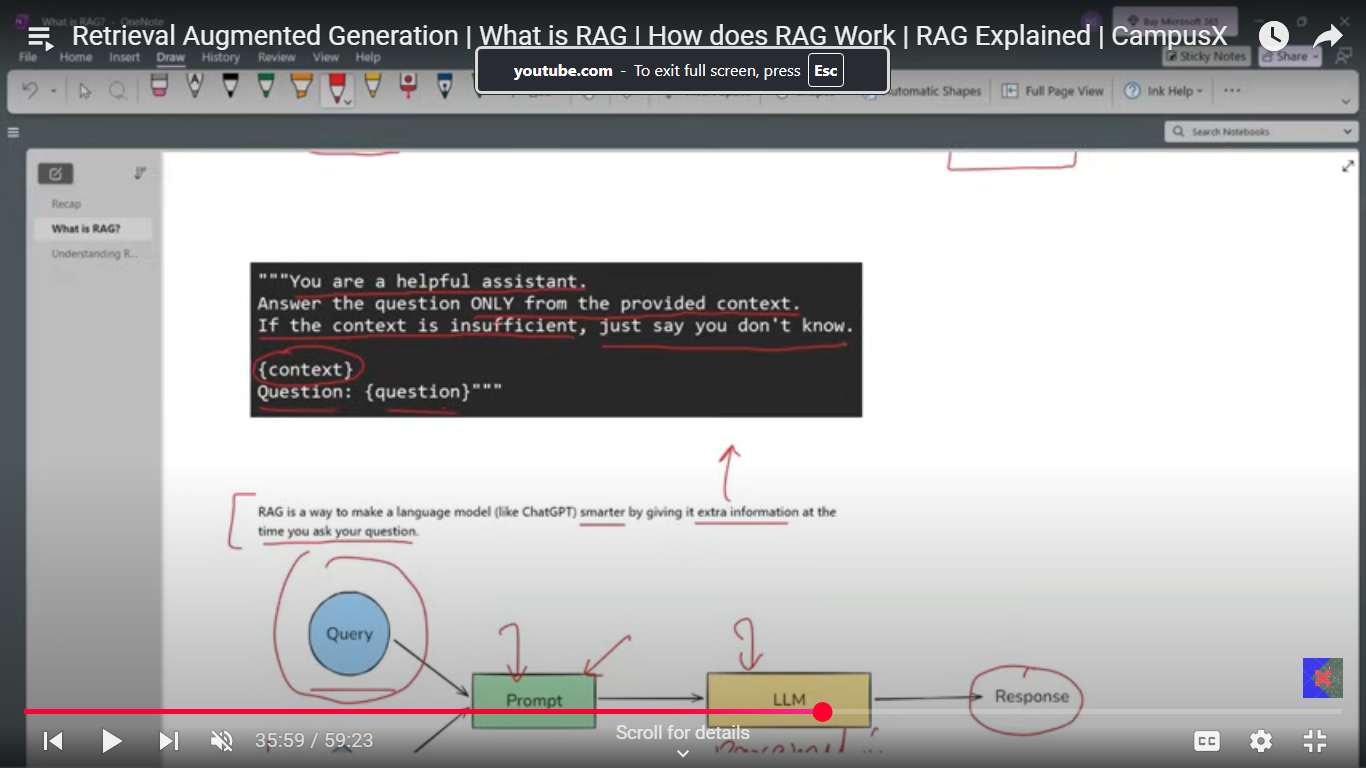
* **In-context learning**: You provide the model with examples or relevant information directly in the prompt. The model uses these examples to understand and perform the task. This approach relies on what you can fit into the prompt and the model's existing knowledge.
* **RAG**: The model automatically retrieves relevant information from an external database or document collection and uses that as additional context to generate its response. This allows the model to access up-to-date or specialized information without needing you to supply it manually in the prompt.

In-short:  
**In-context learning** depends on user-supplied examples in the prompt, while **RAG** dynamically fetches the most relevant external information to help the model answer accurately.

**RAG** just makes model smarter by giving the extra information to the model at the time a question is asked to the model.



This is how a prompt looks like,



RAG is combination of two topics, information retrieval and Text Generation (LLMs).

There are four steps in RAG:

* + - * 1. **Indexing**

Indexing is the foundational step where all relevant documents or data sources are gathered, broken down into smaller, manageable chunks, and converted into vector embeddings using an embedding model. These embeddings capture the semantic meaning of each chunk and are stored in a vector database, enabling fast and efficient similarity search when a query is made

* + - 1. **Retrieval**

When a user submits a query, the system converts the query into an embedding and searches the vector database for the most similar document chunks. This similarity search ensures that only the most relevant pieces of information are selected to address the user's question, optimizing both accuracy and relevance

* + - 1. **Augmentation**

The retrieved document chunks are then combined with the original user query to create an augmented prompt. This step ensures that the language model receives both the user's question and the most pertinent external context, providing a richer foundation for generating a grounded and informed response

* + - 1. **Generation**

Finally, the language model processes the augmented prompt, using both its own internal knowledge and the retrieved external context to generate a coherent, accurate, and contextually relevant answer for the user

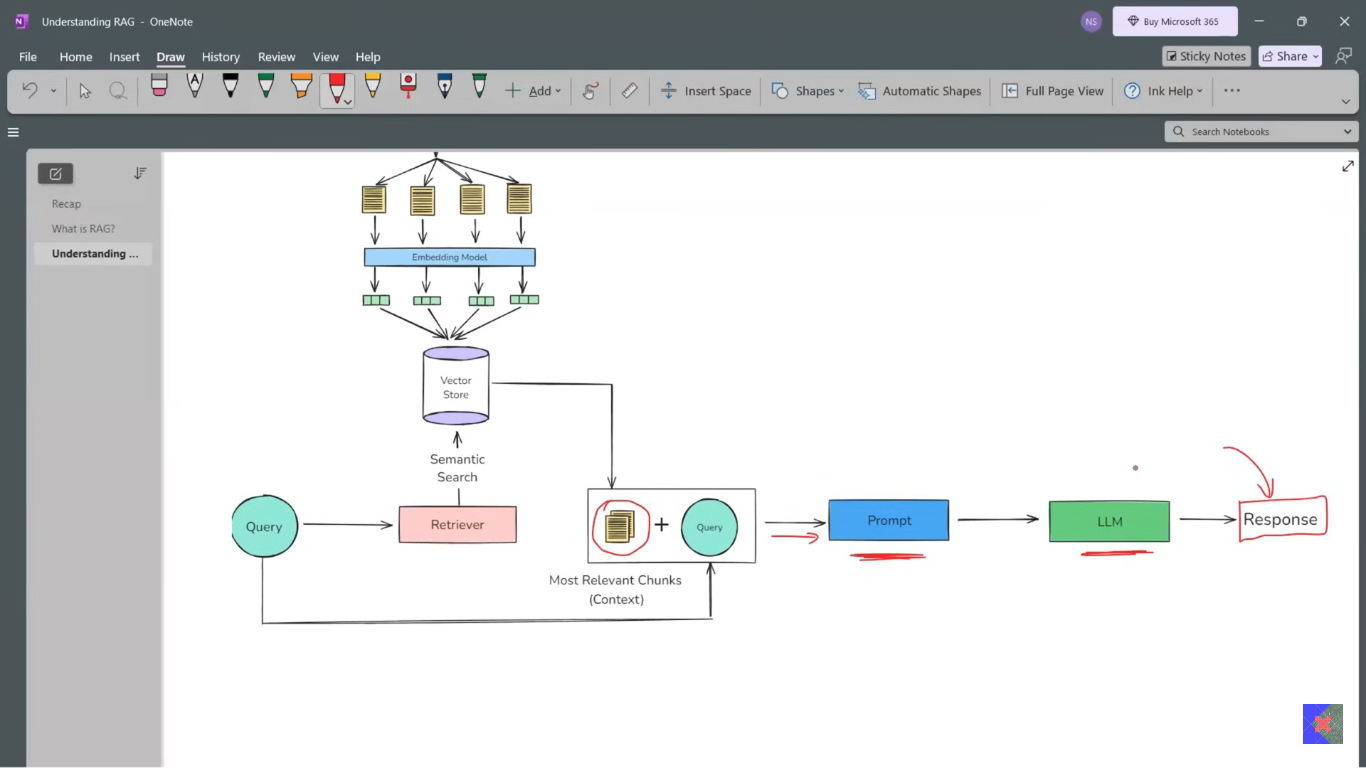
* + - * 1. **Indexing:** It is a process of preparing your knowledge base so that it can be efficiently searched at query time. This step consists of 4 Sub-steps.
  1. Document ingestion: You load your source knowledge into Memory with the help of different document loaders. Files like Word, pdf, SQL records, scrapped pages etc.
  2. Text Chunking: Breaks small documents into small, semantically meaning chunks. (Recursive Character text splitters)
  3. Embedded Generation: Generating the text into a dense vector embedding.
  4. Storage in vector store: Store the vector along with the original chunk text + metadata in a vector database.

1. **Retrieval:** A real time process of finding the most relevant pieces of information from pre-built index from user’s question.

Multiple steps in it:

* 1. Embed the query
  2. Search the query vectors related vectors.
  3. Rank these vectors.

1. **Augmentation:** Combines the query vector with search context.
2. **Generation:** Model generates the output.



**RAG Project:**

### **Agentic making in LangChain**

**Tools:** A tool in an LLM is a Python function or API that is a packaged in a way the LLM can understand and all call when needed.

**LLMs are great** at reasoning and language generation but they cannot do task, it’s like they can understand and have brain to process but no hands and legs to do some dynamic tasks. LLMs can’t access data, call APIs, run code, interact with systems (like database etc.)

There are two kinds of the Tools:

* **Built-in Tools:** Built in are the tools which are readily available in the Langchain.

E.g. Web Search via duck\_duck go, Wikipedia summary, run raw python, run shell commands, Make HTTP get requests, send email via Gmail etc.

* **Custom tools: A custom tool which you define yourself. Used when:**
* You want to call your own APIs.
* You want to encapsulate business logic.
* You want the LLM to interact with your database, product or gap.

How do you create your own Custom tool: There are three ways of creating custom tools

**Using @tool decorator** (most used)

* First add the doc string in your function. Highly advisable.
* Define your input and output data types.
* Use tool decorator on the custom function and now it behaves just like the in-built tool, which is a runnable with invoke, description, name (name of the function usually), description (doc string content), args methods.
  + - * 1. **Using structuredTool and Pydantic:** Instead of simply defining the input and output datatypes while building tools, Pydantic Schema is used to define the input, output structure.
        2. **Using BaseTool class:**

Every other tool in langchain even the @tool, structured tool or even built in tools have inherited a class called as BaseTool class.

**Tool Kit:**

A **toolkit** in LangChain is a collection of tools that are designed to be used together for specific tasks or objectives. While a *tool* is a single utility function or integration that can be called by a language model (such as searching the web, sending emails, or running code), a *toolkit* groups multiple related tools into one package to streamline workflows and enable more complex or multi-step operations.

* + - * 1. **Tool Calling:** Way to call tool in LANGCHAIN.
        2. **Agents:**

1. Indexes:
2. **Agents**