Contents

[GEN AI 2](#_Toc200687761)

[**Langchain:** 3](#_Toc200687762)

[**Why do we need Langchain?** 3](#_Toc200687763)

[**Benefits of using LANGCHAINS:** 5](#_Toc200687764)

[**Langchain Component or Langchain RoadMap.** 5](#_Toc200687765)

[**1. Models:** 6](#_Toc200687766)

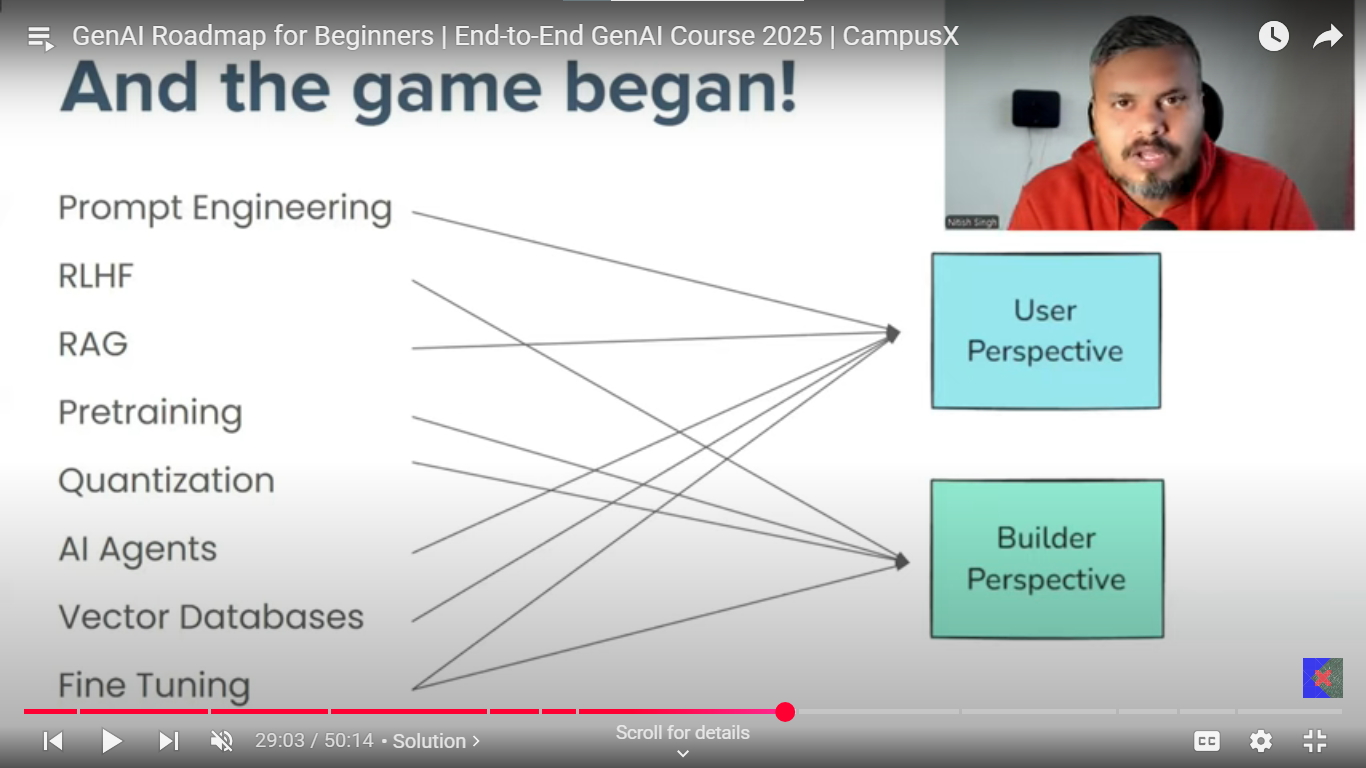
[**2. Prompts:** 8](#_Toc200687767)

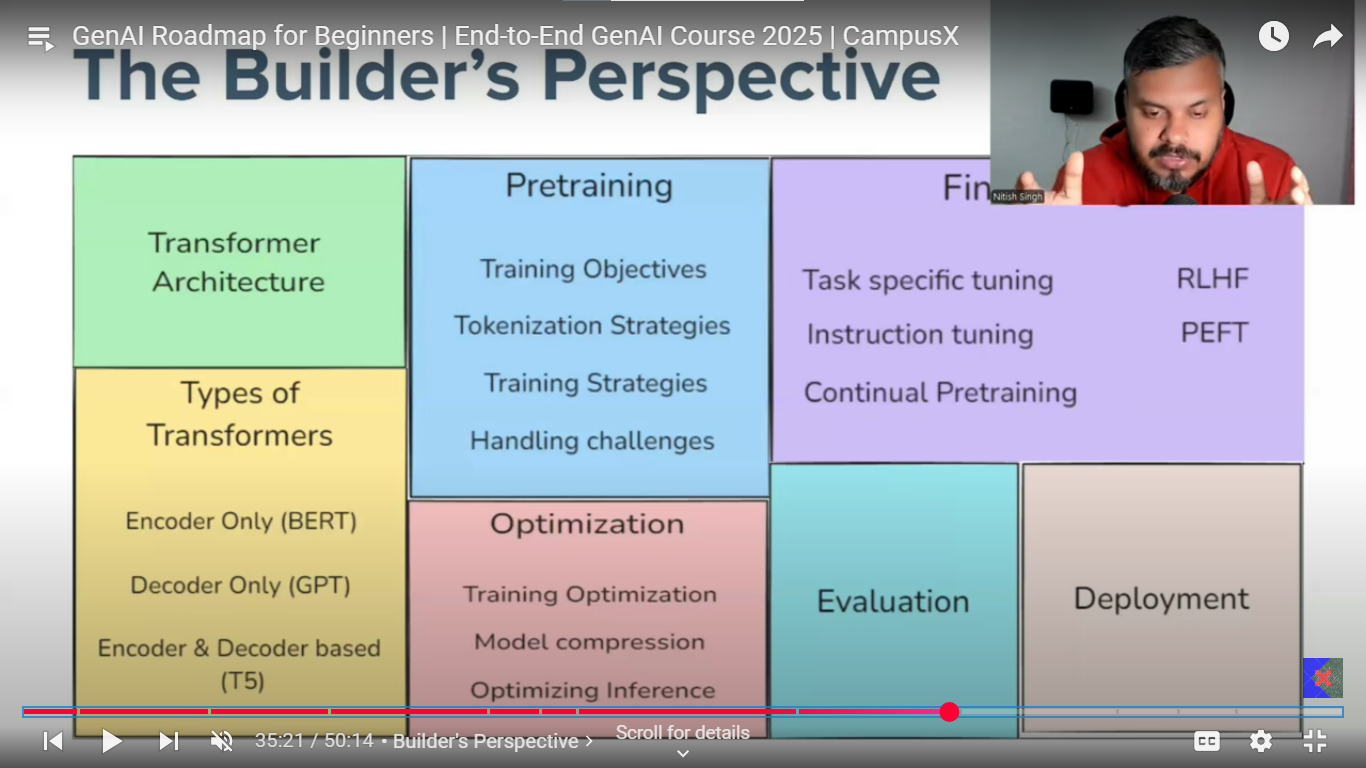
[**PromptTemplate:** 9](#_Toc200687768)

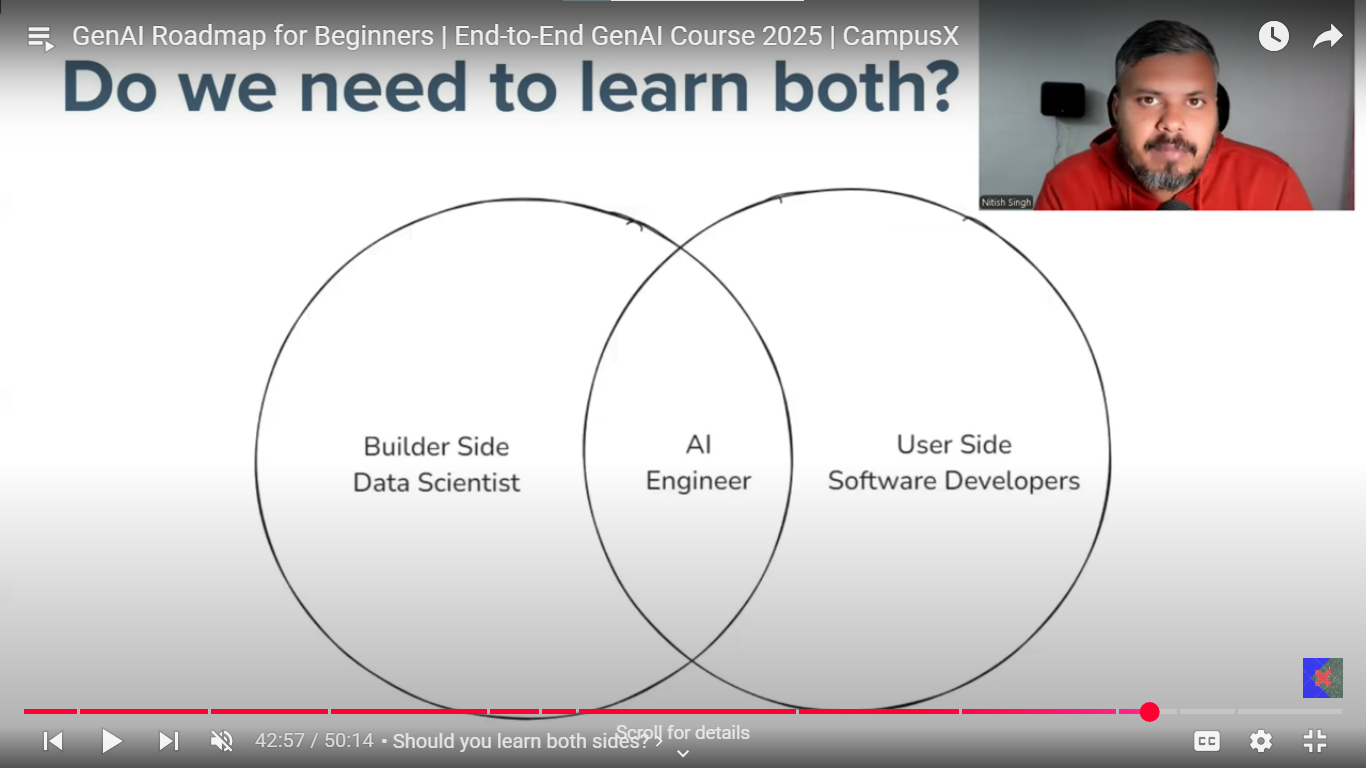
[**Structured output from LLMs** 11](#_Toc200687769)

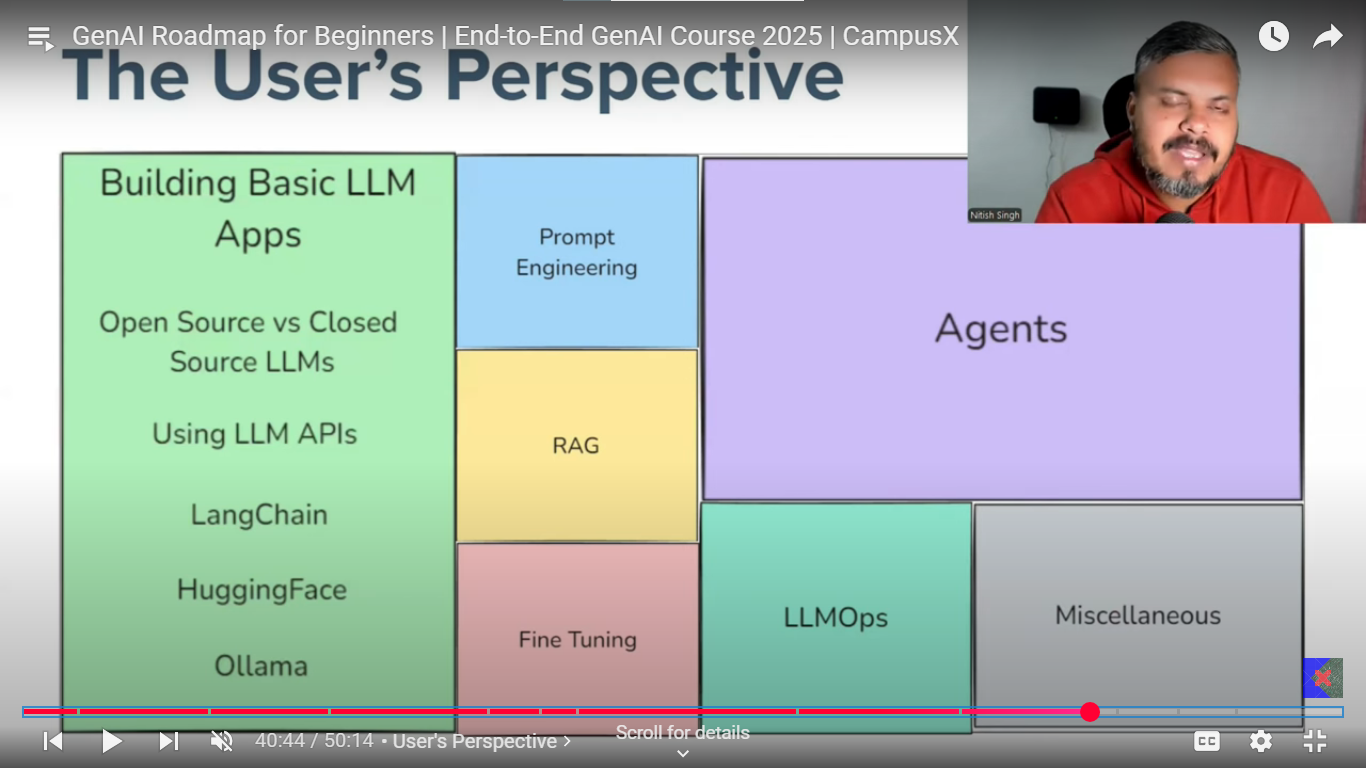
[**3. Chains** 13](#_Toc200687770)

# GEN AI







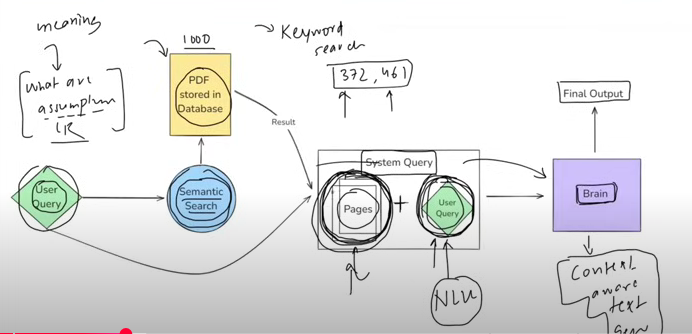


GEN AI app:

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

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

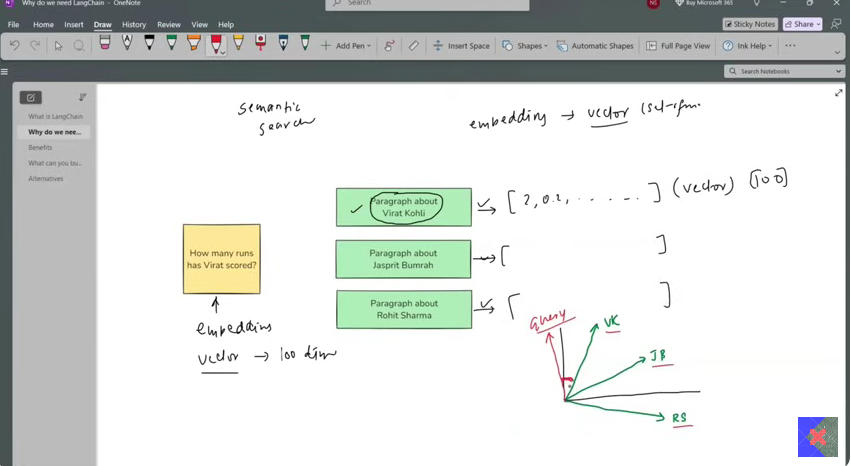
Examples of EBOOK where use can upload a book and interact with it by asking questions. Application answers the question form the book.



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 this makes managing these processes and component very smooth. Explains the use of LANGCHAIN.

## **Benefits of using LANGCHAINS:**

* 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 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 can not 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 like wise 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 form 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.

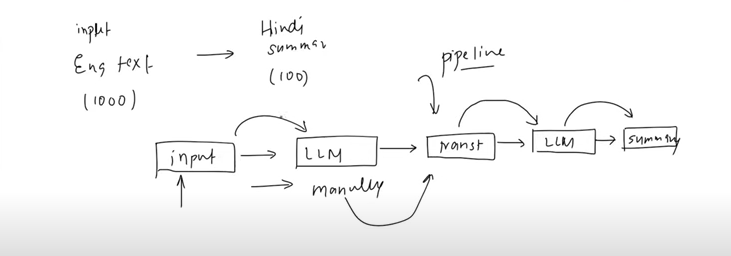
**How does it work in the backend.**

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.

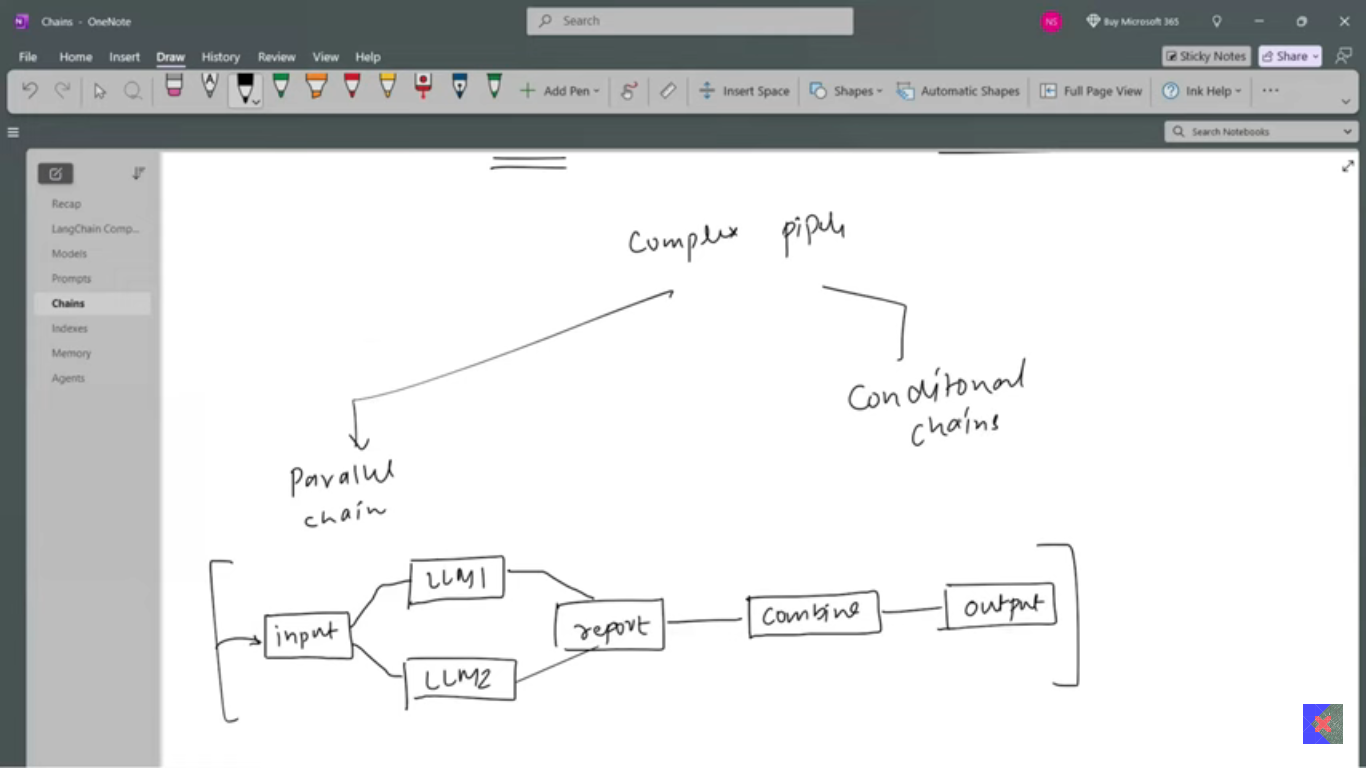
1. **Pydantic:**
2. **Chains:** Important elements of the langchain, so important that langchain is named cause of chains. A concept refers to sequence of automated steps or components called as links that work together to process of users’ query and generate context aware response from the LLM. Each 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 langchai. A simple pipeline can be made with the help of chains where the input of the one component is fed to the second element.

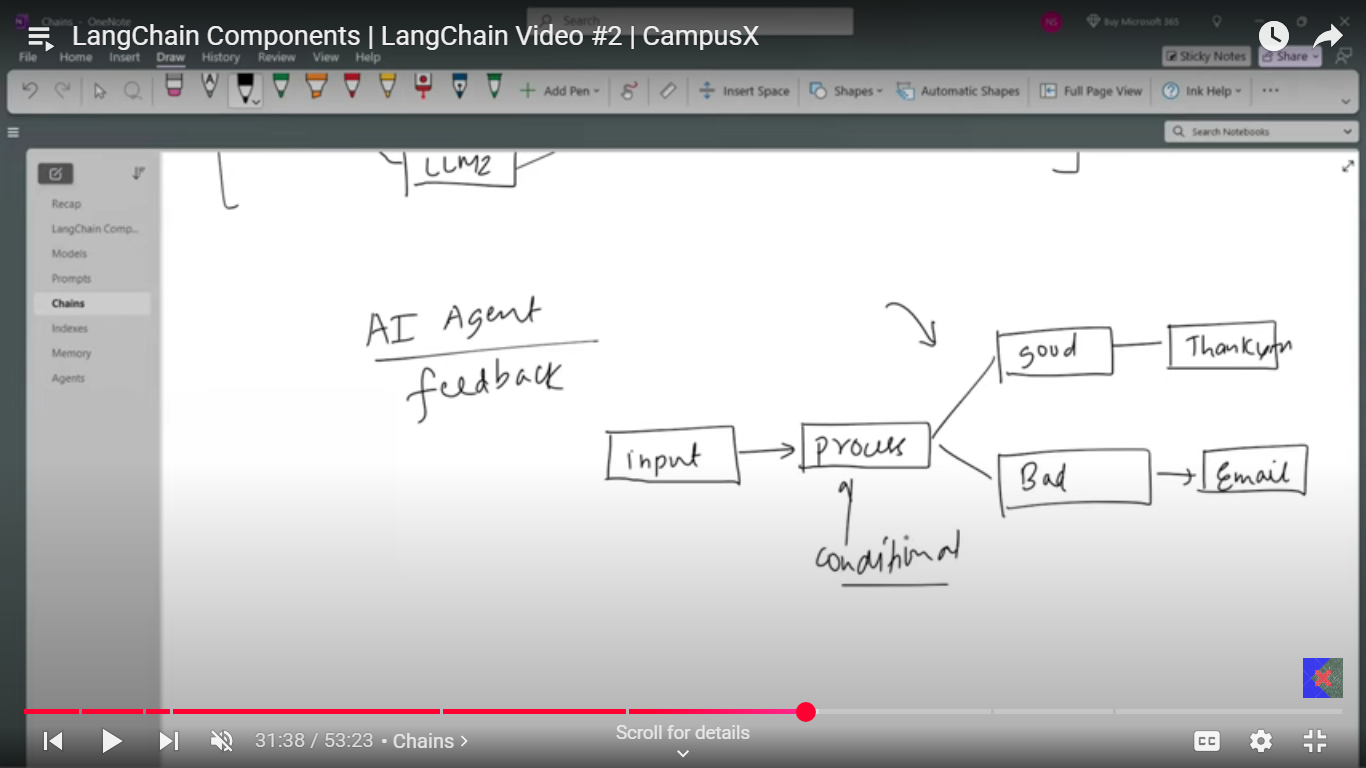
**Eg. Builiding bot to take 1000 words English para and return a 100 words hindi summary.**



**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. Memory
2. **Agents**