**GENEATIVE AI Pre-requisitions..**

1. Python programming language

2. Statistics

3. NLP and Computer visions..

4. NLP Deep learning..RNN and its variants.Lstm, Gru, encorder decorder

5. Transformer berts and its variats.., attention models , attention is all you need .

**LEARNING GEN AI Pre-requistions..**

-Frameworks to develop Gen Ai applications. Such as Launching, lama index, chainlet, hugging fales.

-LLM Multi models,Open source models.

-Fine tunning techniques like Quora , rora ,mistol, lama 2 ,Gemma models.

**TO BE A DIFFERENTIATOR.**

-MLops -inferencing techniques..Groqs.. Generally a life cycle of a Gen AI project will be handled through this.. Eg Vortex ai from Google are providing this kind of platforms to do this.

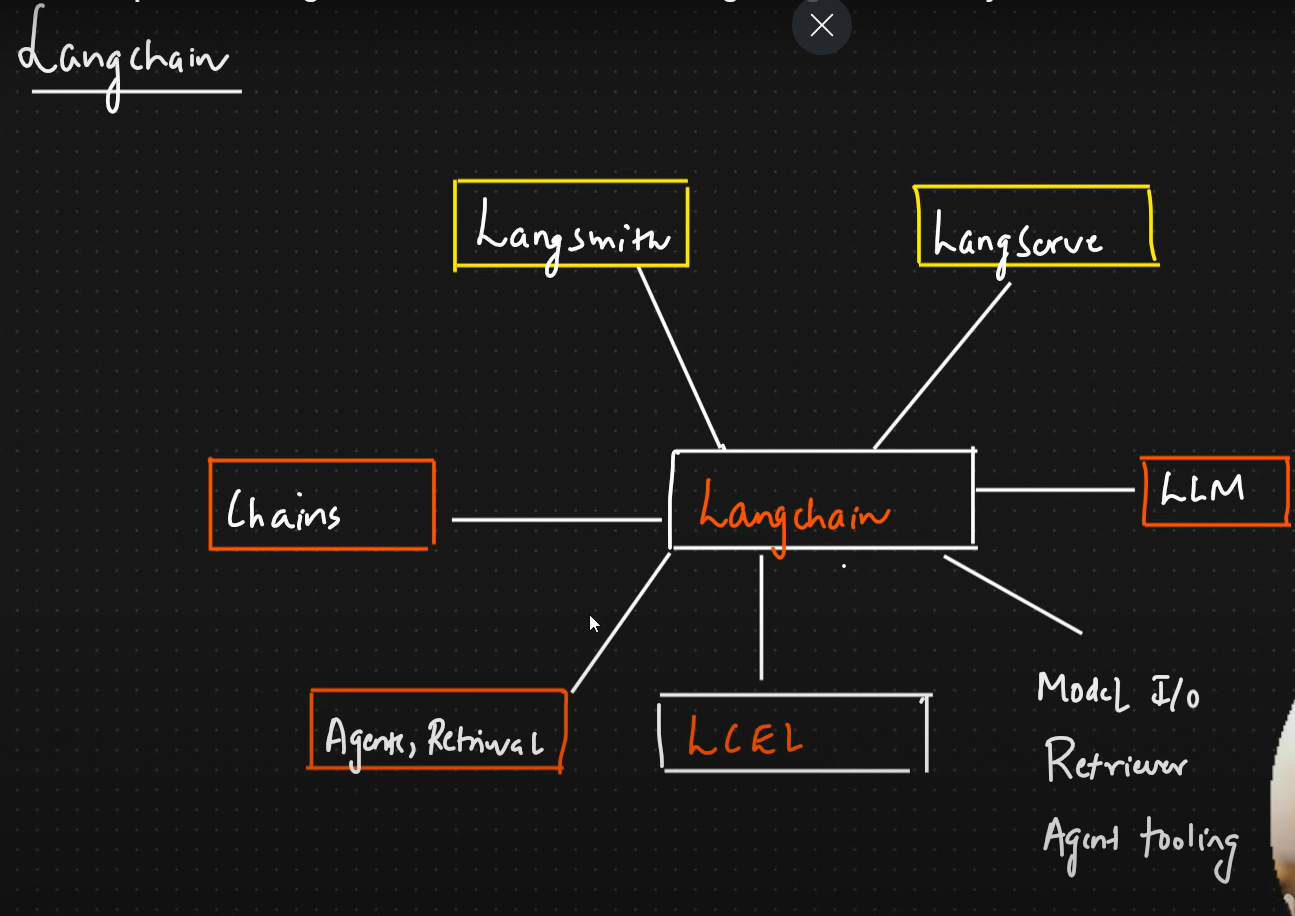
**LangChain** is a framework for developing applications powered by language models. This framework consists of several parts.

**LangChain Libraries:** The Python and JavaScript libraries. Contains interfaces and integrations for a myriad of components, a basic run time for combining these components into chains and agents, and off-the-shelf implementations of chains and agents.

**LangChain Templates:** A collection of easily deployable reference architectures for a wide variety of tasks.

**LangServe**: A library for deploying LangChain chains as a REST API.

**LangSmith**: A developer platform that lets you debug, test, evaluate, and monitor chains built on any LLM framework and seamlessly integrates with LangChain.



Langchain EcoSystem:

V0.3

A screenshot of a computer

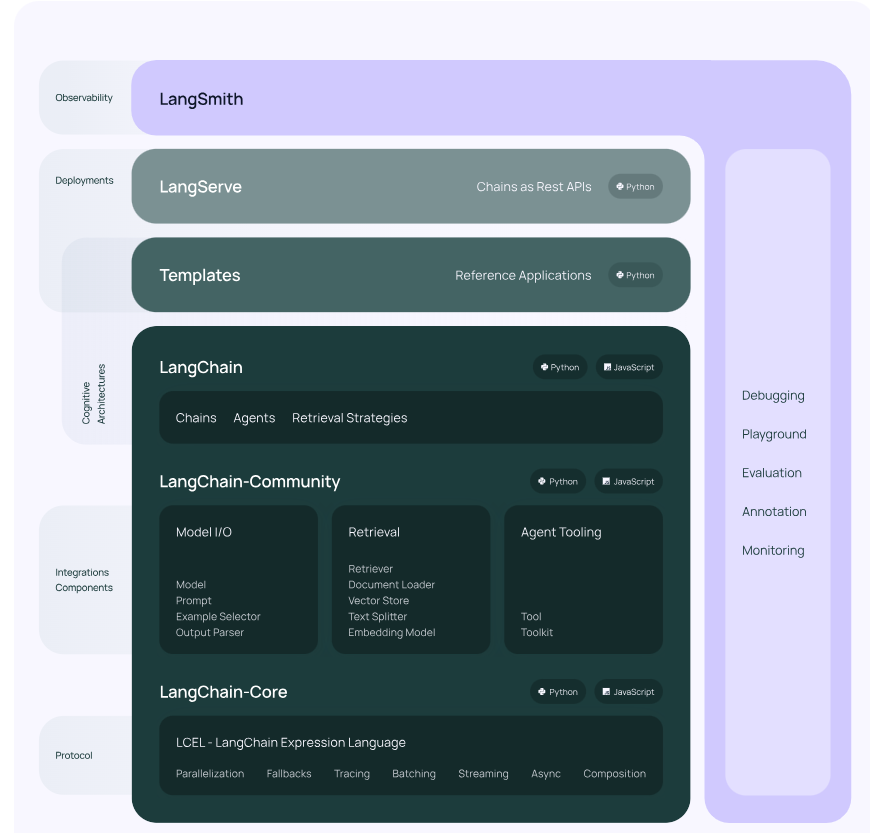
AI-generated content may be incorrect.

V0.2

A screenshot of a computer screen

AI-generated content may be incorrect.

V0.1



[LangChain](https://www.langchain.com/)

Login using Google Id.

Will See LangSmith dashboards

C:\AIandGenAI\_Krish\_May25

**Commands to create Virtual Environment:**

1. Go to Anaconda prompt
2. Route to the folder where you want to create project
3. Write code .
4. VScode will open -> then open terminal
5. Write following command to create environment -> conda create -p venv python==3.12 -y
6. conda activate venv/

git init

echo "# GenAIAndAgenticAI" >> README.md

git add README.md

git branch -M main

git add .

git commit -m "first commit"

git remote add origin https://github.com/itforankit/GenAIAndAgenticAI.git

git push -u origin main

To remove **all files** from the staging area, you can use the following command:

Git reset

git reset <file>

**How to Remove Added Files in Git with**

git rm <file>

git rm --cached <file>

**Installing Packages**

pip install -r requirement.txt

conda install -p c:\AIandGenAI\_Krish\_May25\venv ipykernel --update-deps --force-reinstall

pip install ipykernel

Project Folder: C:\AIandGenAI\_Krish\_May25

First Program -> 1.1-gettingstarted.ipynb

[Output Parsers | 🦜️🔗 LangChain](https://python.langchain.com/v0.1/docs/modules/model_io/output_parsers/)

**Runnable ([langchain\_core.runnables.base.Runnable — 🦜🔗 LangChain 0.2.17](https://api.python.langchain.com/en/latest/runnables/langchain_core.runnables.base.Runnable.html" \l "langchain_core.runnables.base.Runnable))**

**langchain\_core.runnables.base.Runnable**

A unit of work that can be invoked, batched, streamed, transformed and composed.

**Key Methods:**

* invoke/ainvoke: Transforms a single input into an output.
* batch/abatch: Efficiently transforms multiple inputs into outputs.
* stream/astream: Streams output from a single input as it’s produced.
* astream\_log: Streams output and selected intermediate results from an input.

Built-in optimizations:

* Batch: By default, batch runs invoke() in parallel using a thread pool executor. Override to optimize batching.
* Async: Methods with “a” suffix are asynchronous. By default, they execute the sync counterpart using asyncio’s thread pool. Override for native async.
* LCEL and Composition
* The LangChain Expression Language (LCEL) is a declarative way to compose Runnables into chains. Any chain constructed this way will automatically have sync, async, batch, and streaming support.
* The main composition primitives are RunnableSequence and RunnableParallel.
* **RunnableSequence** invokes a series of runnables sequentially, with one Runnable’s output serving as the next’s input. Construct using the *|* operator or by passing a list of runnables to RunnableSequence.
* **RunnableParallel** invokes runnables concurrently, providing the same input to each. Construct it using a dict literal within a sequence or by passing a dict to RunnableParallel.
* Standard Methods
* All Runnables expose additional methods that can be used to modify their behavior (e.g., add a retry policy, add lifecycle listeners, make them configurable, etc.).
* These methods will work on any Runnable, including Runnable chains constructed by composing other Runnables. See the individual methods for details.

**langchain\_core.prompts.chat.ChatPromptTemplate**

**ChatPromptTemplate** ([langchain\_core.prompts.chat.ChatPromptTemplate — 🦜🔗 LangChain 0.2.17](https://api.python.langchain.com/en/latest/prompts/langchain_core.prompts.chat.ChatPromptTemplate.html))

ChatPromptTemplate implements the standard [Runnable Interface](https://api.python.langchain.com/en/latest/runnables/langchain_core.runnables.base.Runnable.html#langchain_core.runnables.base.Runnable). 🏃

The [Runnable Interface](https://api.python.langchain.com/en/latest/runnables/langchain_core.runnables.base.Runnable.html#langchain_core.runnables.base.Runnable) has additional methods that are available on runnables, such as [with\_types](https://api.python.langchain.com/en/latest/runnables/langchain_core.runnables.base.Runnable.html" \l "langchain_core.runnables.base.Runnable.with_types" \o "langchain_core.runnables.base.Runnable.with_types), [with\_retry](https://api.python.langchain.com/en/latest/runnables/langchain_core.runnables.base.Runnable.html" \l "langchain_core.runnables.base.Runnable.with_retry" \o "langchain_core.runnables.base.Runnable.with_retry), [assign](https://api.python.langchain.com/en/latest/runnables/langchain_core.runnables.base.Runnable.html#langchain_core.runnables.base.Runnable.assign), [bind](https://api.python.langchain.com/en/latest/runnables/langchain_core.runnables.base.Runnable.html#langchain_core.runnables.base.Runnable.bind), [get\_graph](https://api.python.langchain.com/en/latest/runnables/langchain_core.runnables.base.Runnable.html" \l "langchain_core.runnables.base.Runnable.get_graph" \o "langchain_core.runnables.base.Runnable.get_graph), and more.

Bases: [BaseChatPromptTemplate](https://api.python.langchain.com/en/latest/prompts/langchain_core.prompts.chat.BaseChatPromptTemplate.html" \l "langchain_core.prompts.chat.BaseChatPromptTemplate" \o "langchain_core.prompts.chat.BaseChatPromptTemplate)

Prompt template for chat models.

Use to create flexible templated prompts for chat models.

[Document loaders | 🦜️🔗 LangChain](https://python.langchain.com/v0.1/docs/modules/data_connection/document_loaders/) -> try with different types of documents (Need to work on this closing today here)

**#Tool calling (**[**Tool/function calling | 🦜️🔗 LangChain**](https://python.langchain.com/v0.1/docs/modules/model_io/chat/function_calling/)**)**

We use the term "tool calling" interchangeably with "function calling". Although function calling is sometimes meant to refer to invocations of a single function, we treat all models as though they can return multiple tool or function calls in each message.

Tool calling allows a model to respond to a given prompt by generating output that matches a user-defined schema. While the name implies that the model is performing some action, this is actually not the case! The model is coming up with the arguments to a tool, and actually running the tool (or not) is up to the user - for example, if you want to [extract output matching some schema](https://python.langchain.com/v0.1/docs/use_cases/extraction/) from unstructured text, you could give the model an "extraction" tool that takes parameters matching the desired schema, then treat the generated output as your final result.

A tool call includes a name, arguments dict, and an optional identifier. The arguments dict is structured {argument\_name: argument\_value}.

Many LLM providers, including [Anthropic](https://www.anthropic.com/), [Cohere](https://cohere.com/), [Google](https://cloud.google.com/vertex-ai), [Mistral](https://mistral.ai/), [OpenAI](https://openai.com/), and others, support variants of a tool calling feature. These features typically allow requests to the LLM to include available tools and their schemas, and for responses to include calls to these tools. For instance, given a search engine tool, an LLM might handle a query by first issuing a call to the search engine. The system calling the LLM can receive the tool call, execute it, and return the output to the LLM to inform its response. LangChain includes a suite of [built-in tools](https://python.langchain.com/v0.1/docs/integrations/tools/) and supports several methods for defining your own [custom tools](https://python.langchain.com/v0.1/docs/modules/tools/custom_tools/). Tool-calling is extremely useful for building [tool-using chains and agents](https://python.langchain.com/v0.1/docs/use_cases/tool_use/), and for getting structured outputs from models more generally.

**RAG**

**Basic RAG**

C:\AIandGenAI\_Krish\_May25\[2. Langchain.pdf](file:///C:\AIandGenAI_Krish_May25\2.%20Langchain.pdf)