



LangChain

Introduction to LangChain

Presented by: Anum Khattak

Overview

Overview of LangChain ecosystem and its capabilities:

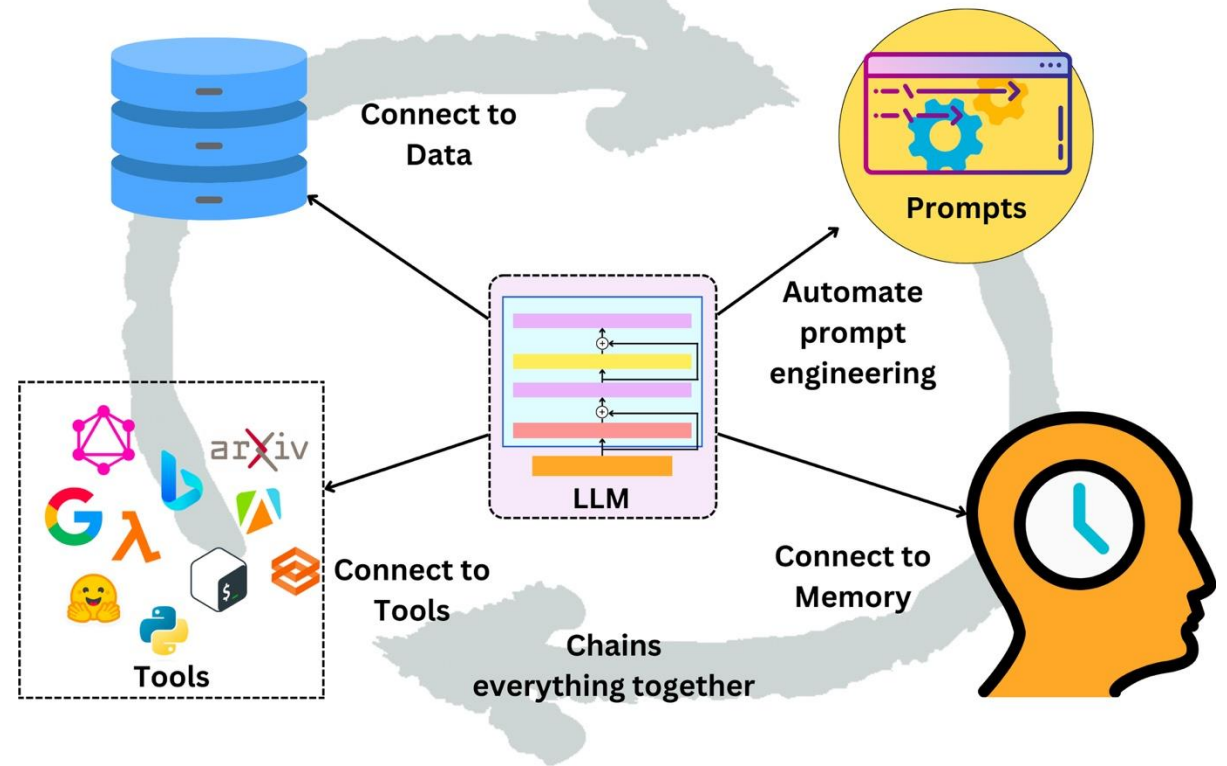
- What is LangChain?
- LangChain Developer
- High-Level Structure of LangChain
- Key Components
- Benefits of LangChain

Step-by-Step Guide on installing LangChain and LangSmith

- Setting Up Python Environment
- Installing LangChain
- Installing LangSmith
- Setting up the development environment on local machines or cloud.

What is LangChain?

Framework for building AI applications that can reason using large language models

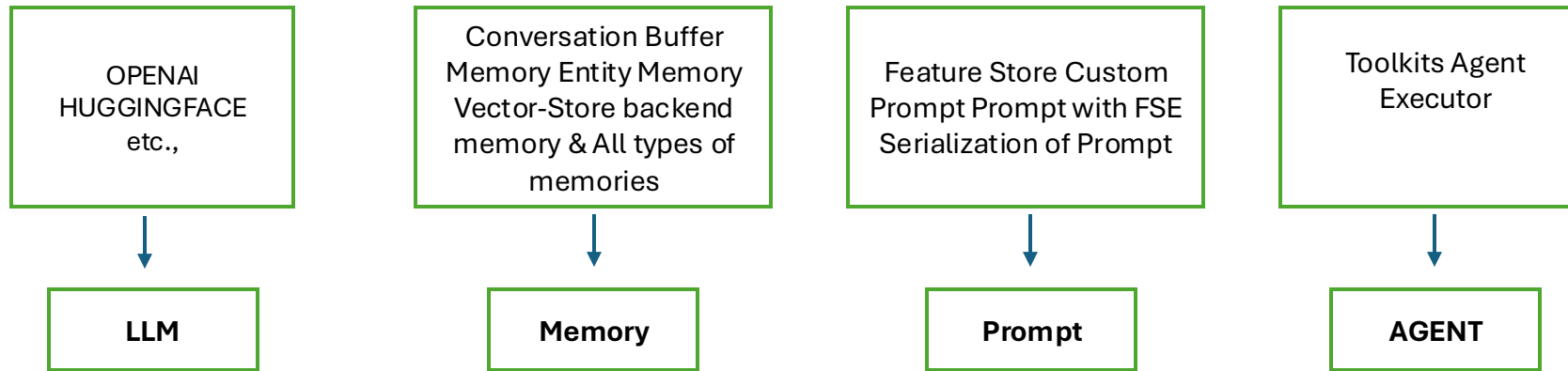


LangChain Developer

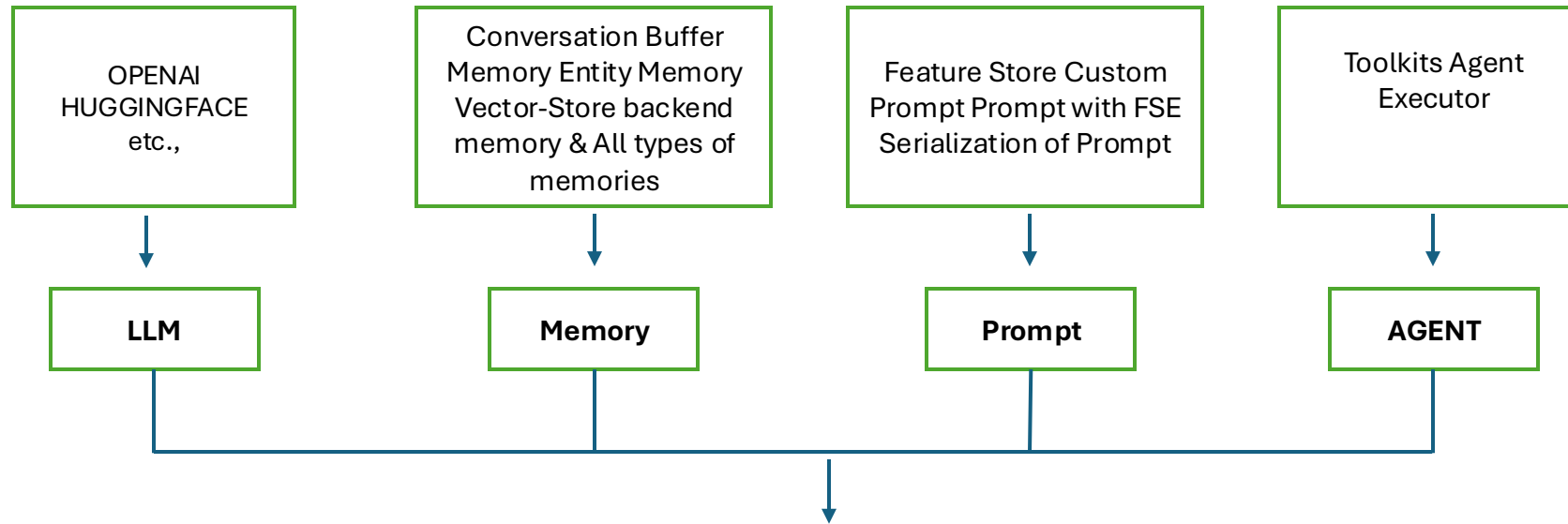
Harrison Chase



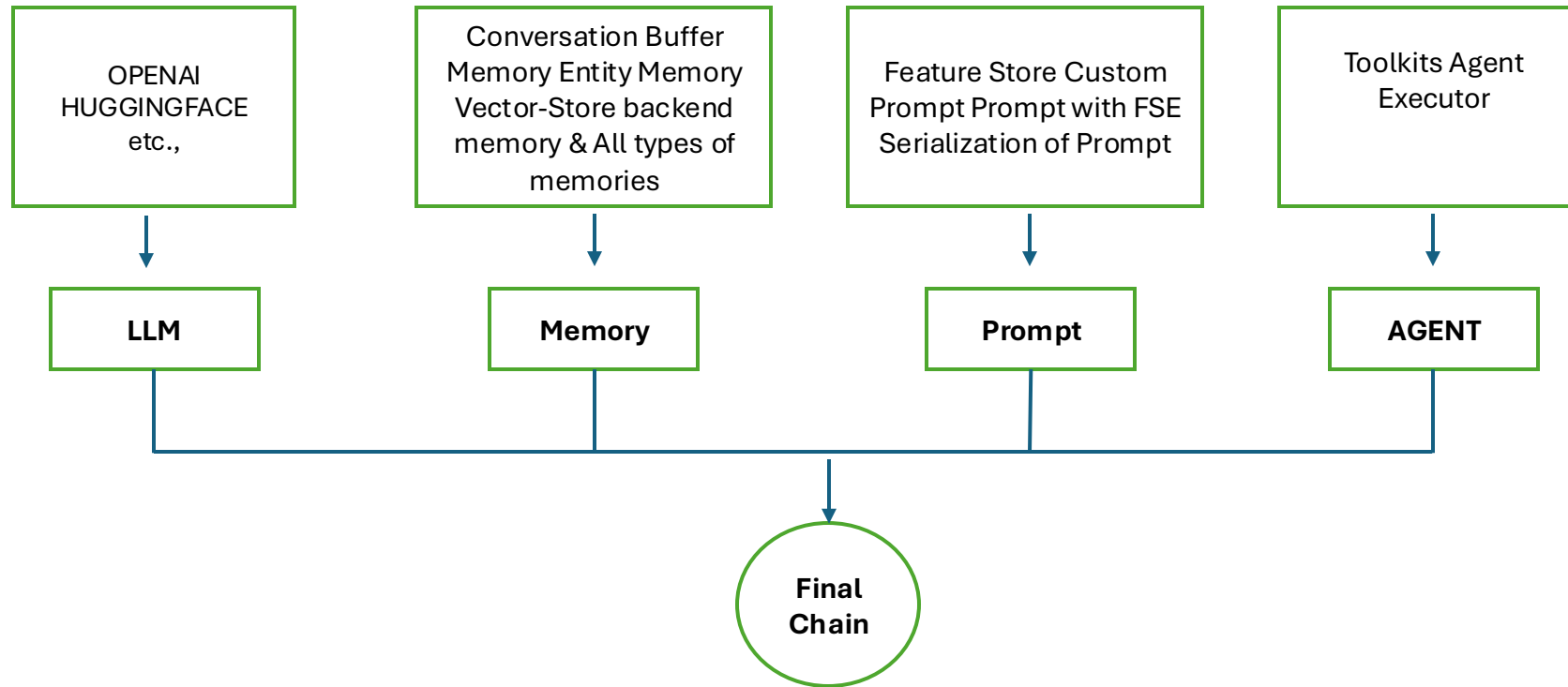
High Level Structure of LangChain



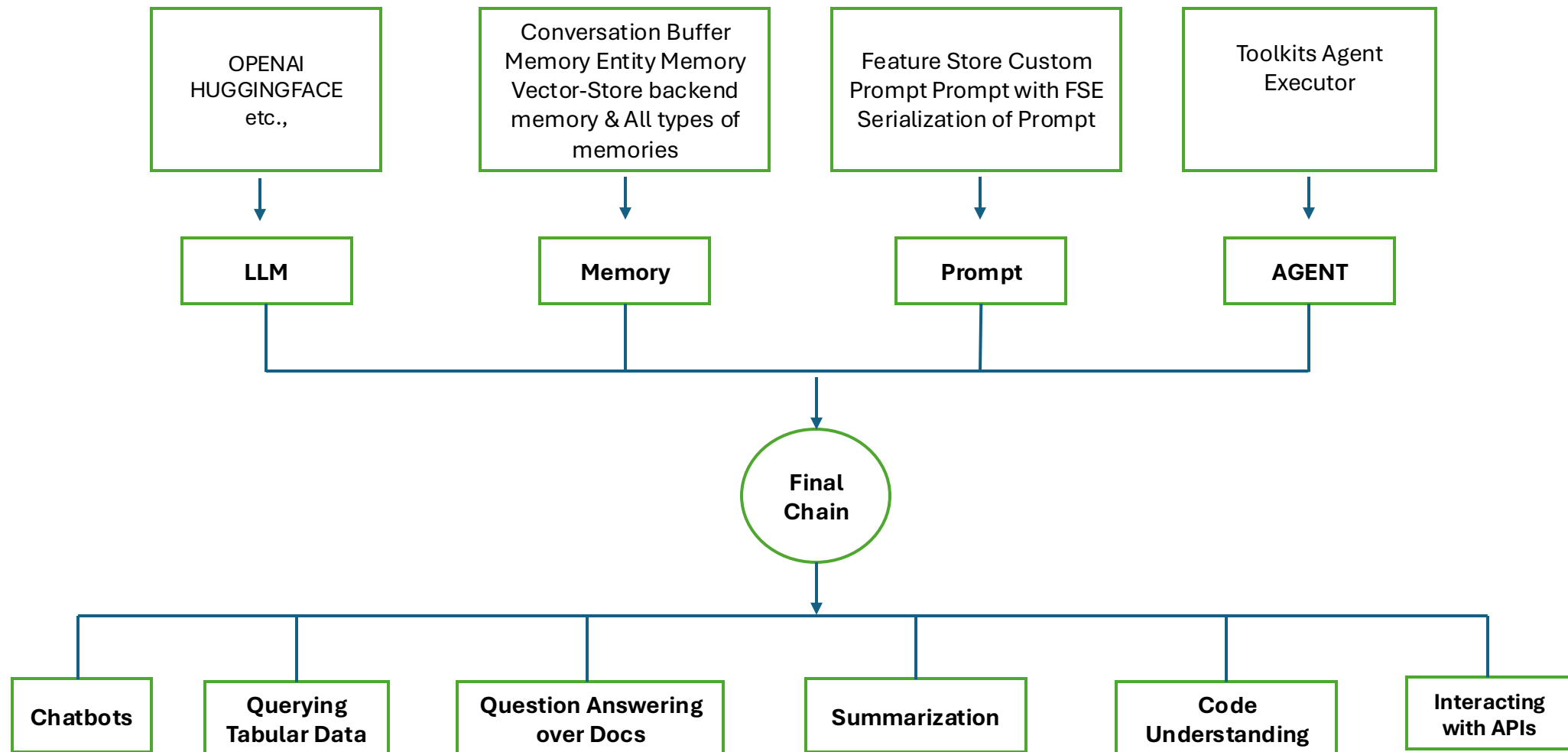
High Level Structure of LangChain



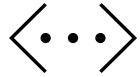
High Level Structure of LangChain



High Level Structure of LangChain



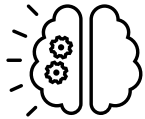
LangChain Key Components



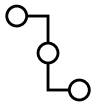
Large Language Model (LLM): Core of the LangChain framework



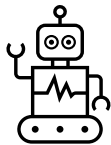
Prompts: Guide the LLM in generating the desired output



Memory: Enables LangChain to retain information over multiple interactions, allowing for more context-aware and personalized conversations.



Chains: Sequences of actions or tasks that need to be performed to generate the desired output



Agent: Dynamic decision-makers that can interact with tools or APIs to take actions based on the model's outputs.

LangChain Key Components – Cont.



Toolkits: Collections of external tools or APIs that agents can use



Retrieval (Vector Stores & Databases): Advance retrieval mechanisms to fetch relevant data from large datasets or documents.



Final Chain: Ties together all the components—LLM, memory, prompt, agent, and any external tools—into a complete workflow

LLM – Large Language Model

- **Implements Runnable Interface:** Core building block for LangChain Expression Language (LCEL).
 - **Functionality:** Supports various methods including `invoke`, `ainvoke`, `stream`, `astream`, `batch`, `abatch`, and `astream_log`.
- **Input Compatibility:** Accepts input either as strings or objects that can be converted into string prompts.
 - Examples of coercible objects: `List[BaseMessage]`, `PromptValue`.
- **Built-in Features:** All LLMs are equipped with LangSmith tracing for enhanced debugging and operational insight.

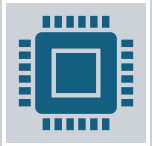
LLM – Custom LLM

- Use of a custom LLM or a different wrapper than the standard one in LangChain

Benefits:

- Minimal code modifications needed for integration.
 - Automatic inclusion as a LangChain Runnable with optimizations and async support.
-
- **Mandatory Methods:** `_call` , `_llm_type`
 - **Additional Functionalities:** `_identifying_params`, `_acall`, `_stream`, `_astream`
 - **Async Support & APIs:** Benefits from `astream_events` API and other asynchronous features out-of-the-box.

LLM - Caching



Cost Efficiency:

Reduces the number of API calls to the LLM provider, saving money when the same requests are made repeatedly.



Performance Enhancement:

Decreases the number of API calls, speeding up application response times



More Information:

https://python.langchain.com/v0.2/docs/integrations/llm_caching/



Prompt

Prompt Templates: Predefined recipes for generating prompts for language models

- Includes: instructions, examples, specific context, and task-appropriate questions

Features:

- **Tooling:** LangChain facilitates the creation and management of prompt templates.
- **Model Agnosticism:** Designed to be reusable across different language models.

Prompt – Template Details

- **Structure:** Consists of a string template accepting user-defined parameters.
- **Formatting Options:**
 - Default: f-string
 - Alternative: jinja2 (use with caution due to security risks)
- **Security:**
 - Jinja2 templates are rendered in a SandboxedEnvironment to mitigate risks.
 - Strong recommendation against using templates from untrusted sources.

Prompt – Template Parameters

input_types, input_variables, metadata, optional_variables, output_parser, partial_variables

- **Implementation in LangChain**

ChatPromptTemplate: Manages chat messages with assignable roles.

MessagesPlaceholder: Provides flexibility in message formatting during prompt creation.

Execution: Implements the Runnable interface supporting various LCEL operations (invoke, ainvoke, etc.).



Memory

Essential for Conversations

LangChain Utilities

Status: Most features are in beta due to:

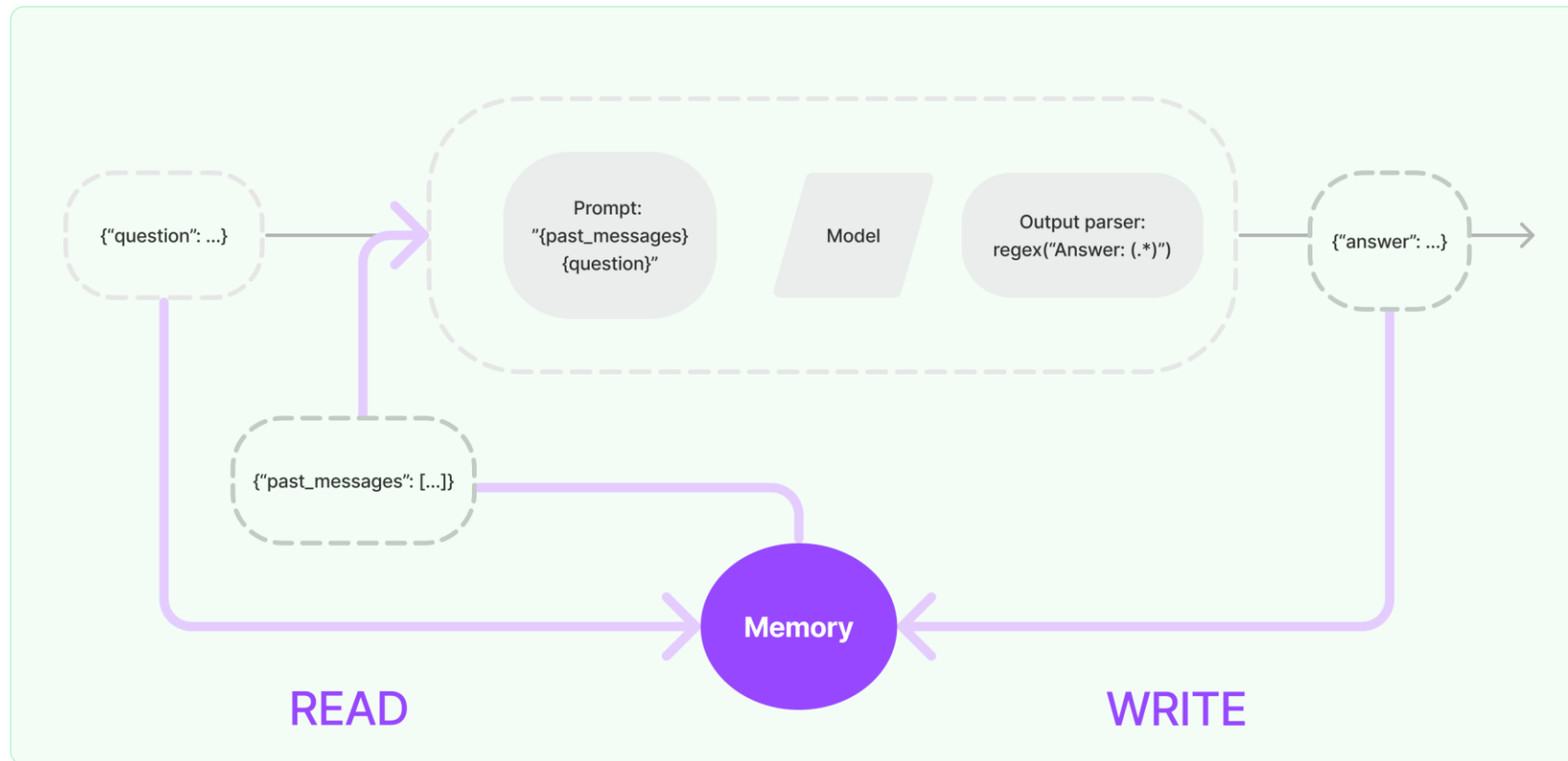
- Limited production readiness.
- Compatibility primarily with Legacy chains, not LCEL.

Exception: ChatMessageHistory is production-ready and integrates with LCEL.

Memory - Operation

Before Execution: Reads from memory to augment initial user inputs.

After Execution: Writes inputs and outputs to memory for future reference.



Memory – Building Memory System

Building a Memory System

- **Storage Decisions:**
 - **Simple Storage:** List of chat messages.
 - **Complex Systems:** Use of databases for persistence.
- **Querying Memory:**
 - **Simple Queries:** Return recent messages.
 - **Complex Queries:** Summarize past interactions or extract and return entity-related information.



Callbacks

Enhance Functionality: Useful for logging, monitoring, streaming, and more

Integration Points: Hook into various stages of LLM application execution.

Subscribing to Events

- **Use of callbacks Argument:** Pass a list of handler objects through the API to subscribe to events.
- **Handler Objects:** Implement the CallbackHandler interface.

Callbacks - Types

Constructor Callbacks

- Defined within the constructor (e.g., `LLMChain(callbacks=[handler], tags=['a-tag'])`).
- **Scope:** Object-specific; affects all calls made on that object.
- **Use Case:** Ideal for persistent needs like logging or monitoring across all operations of an object.

Request Callbacks

- Defined in method calls (e.g., `invoke(method, config={'callbacks': [handler]})`).
- **Scope:** Request-specific; affects only the particular request and its sub-requests.
- **Use Case:** Suitable for one-time needs like streaming output to a websocket for a specific request.



Callbacks – Verbose Argument

Functionality: Acts similar to passing a `ConsoleCallbackHandler` for logging all events to the console.

Usage: Available as a constructor argument across most objects (e.g., `LLMChain(verbose=True)`).

Chains

Sequences of calls, which could involve an LLM, a tool, or a data preprocessing step.

Primary Construction: Utilized through LCEL

LCEL Chains:

1. Constructed using LCEL with higher-level constructor methods.
2. Offers direct modification of chain internals via LCEL editing.
3. Supports streaming, async operations, and batch processing natively.
4. Includes built-in observability at each chain step.

Legacy Chains:

1. Constructed by subclassing from a legacy Chain class.
2. Standalone classes that do not use LCEL under the hood.


More information:

<https://python.langchain.com/v0.1/docs/modules/chains/>



Agents - Concept

Agents vs. Chains: Agents dynamically choose action sequences through language models, unlike hardcoded action sequences in chains.



Key Components:

Schema and
Abstractions

Execution and
Tools

Design
Considerations

Built-in and
Custom Agents

Agents – Schema and Abstraction



AgentAction: Data class representing an action with tool and tool_input.



AgentFinish: Marks the end of agent activity, containing a key-value output, usually a response string.



Intermediate Steps: List[Tuple[AgentAction, Any]] capturing past actions and outputs to inform subsequent actions.



Agent: Core decision-making chain utilizing a language model for next steps.

Agents – Execution and Tools

Agent Inputs: Key-value pairs, primarily `intermediate_steps`.

Agent Outputs: Decisions as `AgentAction`, `List[AgentAction]`, or `AgentFinish`.

Agent Executor: Manages execution cycles, handling errors, and logs activity.

Tools: Functions with specific input schemas and Python functions for execution.

Toolkits: Predefined groups of tools for common tasks, like the GitHub toolkit.

Agents - Design Consideration



Ensuring agents have access to the right tools.



Describing tools effectively for agent utilization.

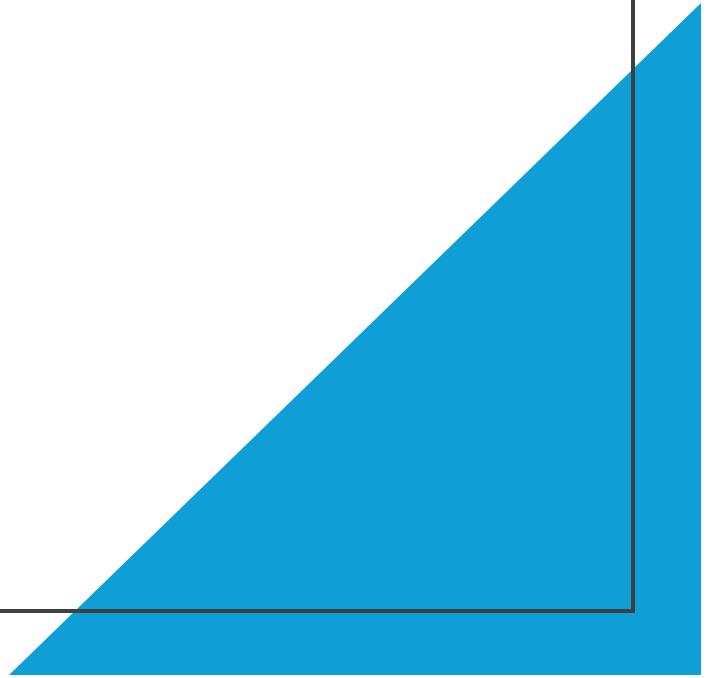


Tools Available (50+):

<https://python.langchain.com/v0.1/docs/integrations/tools/>

Agent – Built-in and Custom Agents

Extensive list of built-in agents and tools for customization



Toolkits – Custom Tools



Tool Components

Name (str)

Description (str)

Args Schema (Pydantic BaseModel)



Defining Tools

@tool Decorator

Subclass BaseTool: Offers maximum control over tool definition but requires more effort.

StructuredTool Dataclass: A hybrid approach that offers more functionality than the decorator and is more convenient than subclassing.

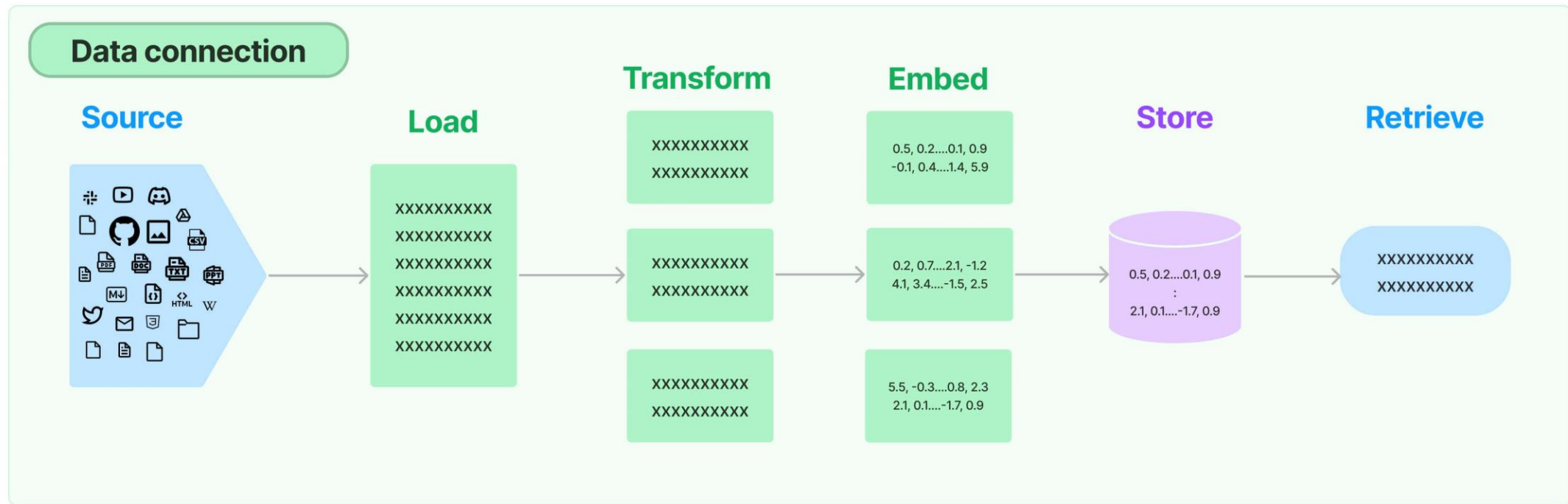
Toolkits - Error Handling

Tools may stop agent execution upon encountering an error.

To continue execution:

- Raise a `ToolException` with `handle_tool_error` set appropriately.
- If `handle_tool_error` is `True`, the agent handles the exception and continues.
- The exception can be handled as a unified string or a function taking `ToolException` as an argument.

Retrieval



Retrieval – Document Loaders

Document is a structured format comprising text and associated metadata

Primary Function: load method

Advanced Feature: Optional lazy load

```
from langchain_community.document_loaders import TextLoader

loader = TextLoader("./index.md")
loader.load()
```


Retrieval – Custom Document Loaders

Extract and format data from databases or files (like PDFs) into a usable format for LLMs

Document Objects: Encapsulate extracted text (page_content) along with metadata (e.g., author's name, publication date)

Key Abstractions

- 1.Document:** Contains text and metadata.
- 2.BaseLoader:** Converts raw data into Document objects.
- 3.Blob:** Binary data representation, located in file or memory.
- 4.BaseBlobParser:** Parses a Blob to yield Document objects.

Retrieval – Text Splitters

Transforming Documents

- Splitting long documents into manageable chunks for model processing

Goal

- Maintain semantic coherence when breaking down texts.

Methodology:

- **Step 1:** Split text into small, semantically meaningful chunks.
- **Step 2:** Combine chunks until a preset size is reached.
- **Step 3:** Start a new chunk with some overlap for contextual continuity.

Retrieval – Text Splitters

Customization Axes

- Text Splitting Method
- Chunk Size Measurement

Types of Text Splitters in LangChain

- Recursive, HTML, Markdown, Code, Token, Character, [Experimental] Semantic Chunker, AI21 Semantic Text Splitter
- **Features:**
 - Various classes implement different methods (e.g., HTML characters, Markdown characters, code-specific characters).
 - Some splitters add metadata about chunk origins.

Evaluate Text Splitters

- **Tool:** Chunkviz by Greg Kamradt.
- **Function:** Visualize and tune text splitting parameters.

Retrieval – Split by

Character (simplest method)

- splits based on characters (by default "\n\n") and measure chunk length by number of characters.

Code

- split code with multiple languages supported

Semantic Chunking

- Splits the text based on semantic similarity

Token

- Split text into manageable chunks to stay within token limits

Retrieval – Embedding Models

Standard Interface:

- Designed to interact uniformly with various text embedding models from providers like OpenAI, Cohere, and Hugging Face.

Functionality of Text Embeddings:

- **Vector Representation:**
 - Converts text into vector form, facilitating operations in vector space.
- **Applications:**
 - Enables semantic search by comparing text similarities within the vector space.

Retrieval – Methods of Embedding Class



Document Embedding

Takes multiple texts as input, suitable for texts that will be searched over.



Query Embedding

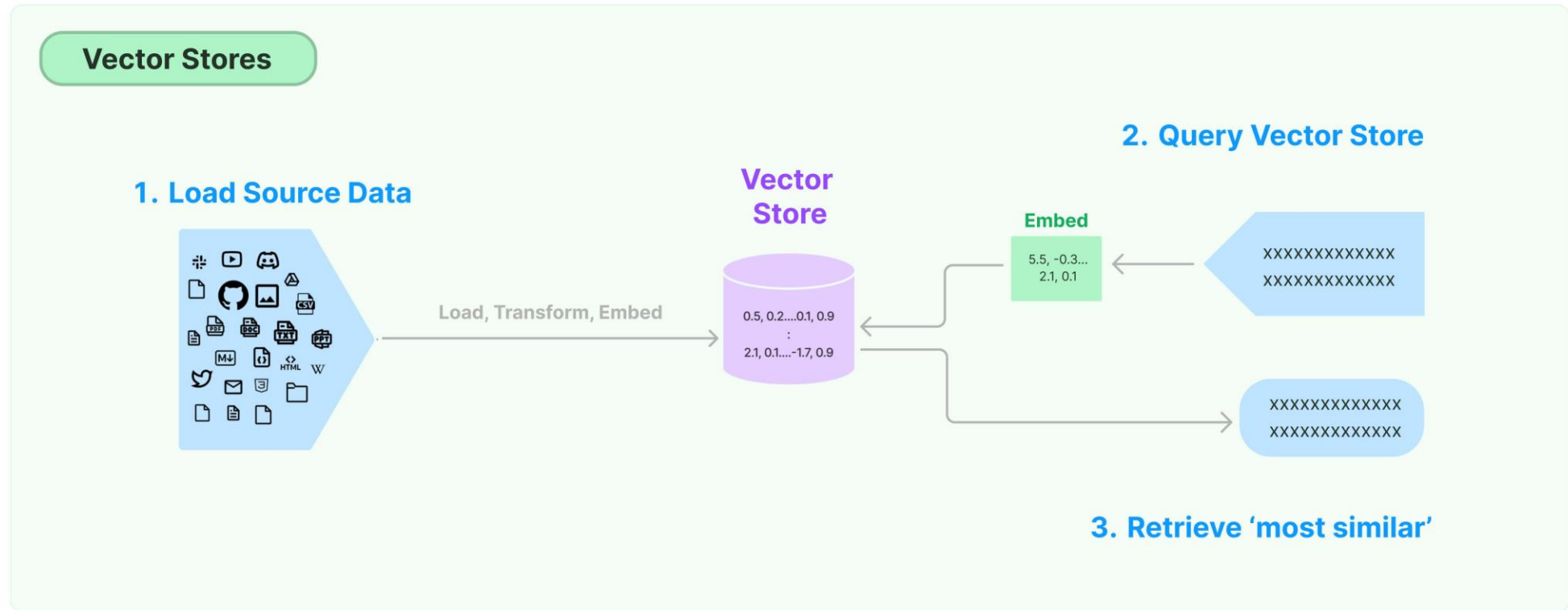
Processes a single text, optimized for use as a search query.

Retrieval - Vector Stores

Storing & Searching Unstructured Data

Process:

1. Embed unstructured data and store the resulting embedding vectors.
2. At query time, embed the query.
3. Retrieve the vectors most similar to the query's embedding.



Retrieval – Benefits of Vector Stores



Efficiency



Accuracy

Retrieval - Retrievers

An interface that returns documents in response to an unstructured query
Focuses on retrieving documents without the need to store them.

How does it differ from a Vector Store?

Vector Store: Can act as the backbone for a retriever by storing and indexing documents.

Retriever: More general; does not necessarily store documents but retrieves them using various mechanisms.

- **Input:** Accepts a string query.
- **Output:** Returns a list of documents (List[Document]).

Retrieval - Indexing

Streamlines the process of loading and syncing documents into a vector store

Key Benefits:

- Prevents duplication of content.
- Avoids rewriting unchanged content.
- Reduces the need for re-computing embeddings, saving time and costs.

Advanced Handling:

- Effectively processes documents even after multiple transformations (e.g., text chunking).



Retrieval – How Indexing Works?

Record Manager (RecordManager): Manages document writes into the vector store.

Data Tracked by Record Manager:

- Document hash (including content and metadata).
 - Write time and source ID to trace document origins.
-

Retrieval – Indexing



Deletion Mode

Options: None, Incremental, Full.

None: No automatic cleanup; manual management required.

Incremental: Cleans up mutations and derived documents continuously.

Full: Comprehensive cleanup at the end of indexing; ideal for batch operations.



Requirements and Compatibility

Vectorstores Supported: Includes AnalyticDB, AstraDB, AzureCosmosDBVectorSearch, etc.

Usage Note: Not suitable for pre-populated stores without API integration.

Retrieval – Indexing [Cautions]



Time Sensitivity



Potential Risk

Benefits of using LangChain



ACCELERATED
DEVELOPMENT



ENHANCED USER
EXPERIENCE



FUTURE PROOF
INVESTMENT



SCALABILITY AND
FLEXIBILITY



IMPROVED
PRODUCTIVITY

Step-by-Step Guide: Installing LangChain and LangSmith

Step 1: Installing IDE - VSCode



Visit the VSCode website: <https://code.visualstudio.com/>.



Download the appropriate installer for your operating system.



Follow the installation instructions.



After installation, open VSCode and install the Python extension from the Extensions Marketplace.

Step 2: Setting Up Python Environment

1. Ensure Python (v3.7+) is installed. You can download Python from <https://python.org>.
2. Create a new directory for your project.
3. Inside VSCode, open a terminal and create a virtual environment:
 - ``python -m venv venv``
4. Activate the virtual environment:
 - Windows: ``venv\Scripts\activate``
 - MacOS/Linux: ``source venv/bin/activate``
5. Install the necessary Python packages using pip.

Step 3: Installing LangChain

1. Open the terminal in VSCode within your project directory.
2. Run the following command to install LangChain:
 - ``pip install langchain``
3. Verify the installation by running:
 - ``python -m langchain --version``

You're ready to start developing with LangChain!

Step 4: Installing LangSmith

LangSmith is a **debugging and monitoring tool** for **LangChain** applications.

1. In the same VSCode terminal, run the following command to install LangSmith:
 - ``pip install langsmith``
2. Verify the installation by running:
 - ``python -m langsmith --version``

LangSmith is now installed and ready to use.

Step 5: Setting Up Local Development Environment

- Ensure your virtual environment is activated in VSCode.
- Install any additional dependencies by running ``pip install -r requirements.txt`` (if applicable).
- You can now begin developing and running your LangChain and LangSmith projects locally in VSCode.

Step 6: Setting Up Cloud Development Environment



Choose a cloud provider (AWS, GCP, Azure).



Set up a virtual machine with Python installed.



Install VSCode and connect remotely via SSH or use a cloud development platform like GitHub Codespaces.



Follow the same steps as the local setup:

Install Python, LangChain, LangSmith, and any other dependencies.



You are now ready to work in the cloud.