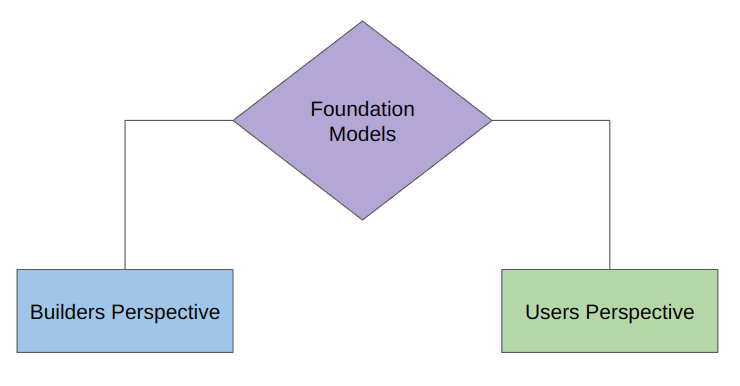
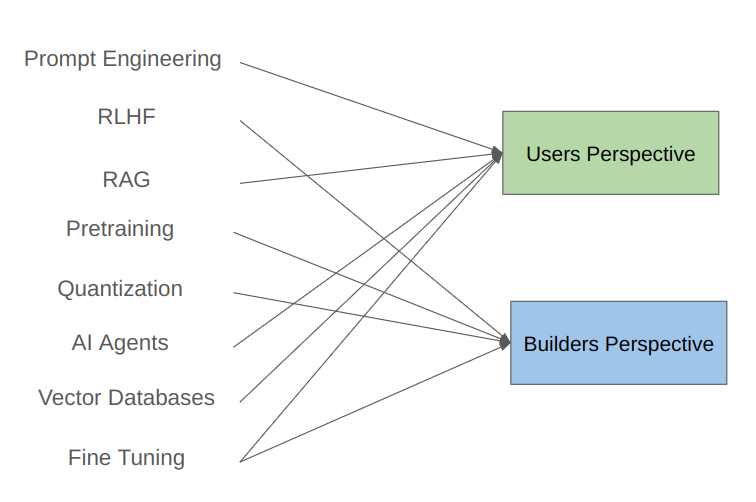
****

**Builders Perspective:**

1. Transformer Architecture
2. Types of Transformer
   1. Encoder only (BERT)
   2. Decoder only (GPT)
   3. Encoder-Decoder based (T5)
3. Pretraining
   1. Training Objectives
   2. Tokenization Strategies
   3. Training Strategies
   4. Handling Challenges
4. Optimization
   1. Training Optimization
   2. Model Compression
   3. Optimization Inference
5. Finetuning
   1. Task Specific Tuning
   2. Instruction Tuning
   3. Continual Pretraining
   4. RLHF
   5. PEFT
6. Evaluation
7. Deployment

**Users Perspective:**

1. Building Basic LLM apps
   1. Open-source vs close-source LLMs
   2. Using LLM APIs
   3. LangChain
   4. HuggingFace
   5. Ollama
2. Prompt Engineering
3. RAG
4. Fine Tuning
5. Agents
6. LLMOps
7. Miscellaneous



**LangChain**

* LangChain is an open source framework for developing applications powered by LLMs.

**Benefits:**

1. Concepts of Chains
2. Model Agnostic Development
3. Complete Ecosystem
4. Memory and State Handling

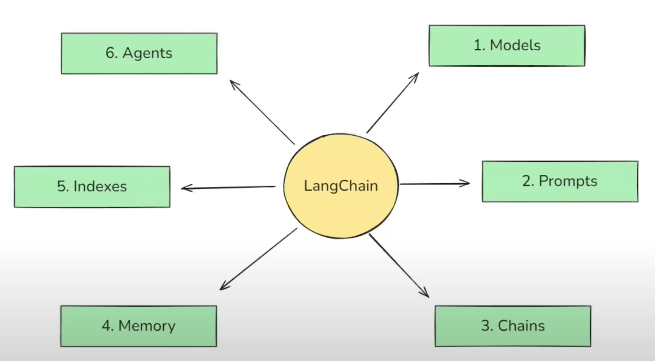
**What we can build using LangChain:**

1. Conversational Chatbots
2. AI Knowledge Assistants
3. AI Agents
4. Workflow Automation
5. Summarization/Research Helpers

**LangChain Alternatives:**

1. LlamaIndex
2. Haystack

**Components of LangChain:**

****

**Structured Output in LangChain**

* In LangChain, structured output refers to the practice of having language models return responses in a well-defined data format (for example, JSON), rather than free-form text. This makes the model output easier to parse and work with programmatically.

**Why do we need Structured Output ?**

* Data Extraction
* API Building
* AI Agents

**Ways to get Structured Output:**

* LLM’s that can generate structured output -> **use with\_structured\_output**
* LLM’s that can’t generate structured output -> **Output Parsers**

**Ways to specify output format in with\_structured\_output:**

1. **TypedDict**

* Typed Dict is a way to define a dictionary in Python where you specify what keys and values should exist. It helps ensure that your dictionary follows a specific structure.

**Why use TypedDict?**

* It tells Python what keys are required and what types of values they should have.
* It does not validate data at runtime (it just helps with type hints for better coding).
* Simple TypedDict
* Annotated TypedDict
* Literal
* Optional

1. **Pydantic**

* Pydantic is a data validation and data parsing library for python. It ensures that the data you work with is correct, structured and type-safe.
  + Default Values
  + Optional Fields
  + Field
  + Built Validation Functions
  + Field Function (default values, constraint, description, regex)
  + Return Pydantic objects -> convert it into json/dict.

1. **Json Schema**

* Provide json schema has a type validation to get structured output from LLM.

**When to use What ?**

| **Feature** | **TypedDict** | **Pydantic** | **Json Schema** |
| --- | --- | --- | --- |
| Basic Structure | ✅ | ✅ | ✅ |
| Type Enforcement | ✅ | ✅ | ✅ |
| Data Validation | ❌ | ✅ | ✅ |
| Default Values | ❌ | ✅ | ❌ |
| Automatic Conversion | ❌ | ✅ | ❌ |
| Cross-Language Compatibility | ❌ | ❌ | ✅ |

**Output Parsers in LangChain**

* Output Parsers in LangChain help convert raw LLM responses into structured formats like JSON, CSV, Pydantic models, and more. They ensure consistency, validation, and ease of use in applications.
* Output Parsers can be used with both LLMs types (one that gives structured output and one that doesn’t give.)
* Parsers can online be used with chains.
* Different types of output parsers:
  + String Output Parsers
  + Json Output Parsers
  + Structured Output Parsers
  + Pydantic Output Parsers

1. **String Output Parsers**

* The StrOutputParser is the simplest output parser in LangChain. It is used to parse the output of LLMs and return it as a plain string.

1. **Json Output Parsers**

* It is used to parse the output of LLMs and return it as a json.
* **Note:** We cannot specify the output schema in json parser, and it is the biggest flow of it.

1. **Structured Output Parsers**

* ﻿StructuredOutputParser is an output parser in LangChain that helps extract structured JSON data from LLM responses based on predefined field schemas.
* It works by defining a list of fields (ResponseSchema) that the model should return, ensuring the output follows a structured format.
* **Note:** This parser does not provide support for data validation of the final output from this parser.

1. **Structured Output Parsers**

* PydanticOutputParser is a structured output parser in LangChain that uses Pydantic models to enforce schema validation when processing LLM responses.
* **Why Use PydanticOutputParser?**   
  + **Strict Schema Enforcement:** Ensures that LLM responses follow a well-defined structure.
  + **Type Safety:** Automatically converts LLM outputs into Python objects.
  + **Easy Validation:** Uses Pydantic's built-in validation to catch incorrect or missing data.
  + **Seamless Integration:** Works well with other LangChain components.

**Chains in LangChain**

* In LangChain, a Chain is a sequence of steps that connects components—like prompts, LLMs, tools, and other chains—so that the output of one step becomes the input to the next.
* Chains help structure multi-step reasoning, allowing you to break complex workflows into manageable building blocks.

**Types of Chain:**

1. **Simple Chain**

* The most basic form of a chain.
* Consists of a single input → processed by a component (e.g., LLM, prompt template) → produces a single output.
* Useful when your task doesn’t require multiple steps or intermediate reasoning.

1. **Sequential Chain**

* Executes multiple chains in a fixed sequence, where the output of one chain is passed directly as the input to the next.
* Steps are dependent on each other; earlier outputs influence later steps.
* Ideal for workflows that have a clear step-by-step dependency (e.g., generate a summary → then extract keywords from the summary).

1. **Parallel Chain**

* Runs multiple chains at the same time using the same input.
* Each chain processes independently and returns results separately.
* Good for scenarios where you want to generate different outputs from the same data simultaneously (e.g., translate text into multiple languages at once).

1. **Conditional Chain**

* Executes different sub-chains based on a condition or decision logic.
* Input is evaluated, and the flow branches to the most appropriate chain.
* Useful for workflows where the path depends on input characteristics (e.g., if a document is legal, run legal analysis chain; otherwise, run a generic summary chain).

**Runnables in LangChain**

* Runnables are the fundamental execution units in LangChain’s newer architecture.
* Anything that can be called with input and returns output (possibly asynchronously) is a Runnable—this includes LLMs, prompt templates, tools, retrievers, and even chains.
* They provide a **standardized interface** for building workflows, making it easier to connect and compose components.
* Designed to be composable, so you can chain, branch, or batch them without custom glue code.

**Core Capabilities:**

1. **Run Single Input** – Take one input and produce one output.
2. **Batch Execution** – Process multiple inputs in parallel for efficiency.
3. **Streaming** – Stream intermediate outputs as they are generated.
4. **Composition** – Easily connect multiple runnables together into pipelines.
5. **Standard API** – Unified **.invoke(), .batch(), .stream()** methods across all components.

| **Chain Name** | **Description** |
| --- | --- |
| LLMChain | Basic chain that calls an LLM with a prompt template. |
| SequentialChain | Chains multiple LLM calls in a specific sequence. |
| SimpleSequentialChain | A simplified version of SequentialChain for easier use. |
| ConversationalRetrievalChain | Handles conversational Q&A with memory and retrieval. |
| RetrievalChain | Fetches relevant documents and uses an LLM for question-answering. |
| RouterChain | Directs user queries to different chains based on intent. |
| MultiPromptChain | Uses different prompts for different user intents dynamically. |
| HydeChain (Hypothetical Document Embeddings) | Generates hypothetical answers to improve document retrieval. |
| AgentExecutorChain | Orchestrates different tools and actions dynamically using an agent |
| SQLDatabaseChain | Connects to SQL database and answers natural language queries. |

**Types / Patterns of Runnable Composition:**

1. **Runnable Sequence**

* Connects runnables in order, passing each step’s output to the next.
* Equivalent to a Sequential Chain but using the new standard API.
* Useful for fixed, dependent workflows.

1. **Runnable Map**

* Runs multiple runnables in parallel with the same input, returning a dictionary of results.
* Ideal for producing different outputs at the same time (e.g., summary, sentiment, keywords).

1. **Runnable Branch**

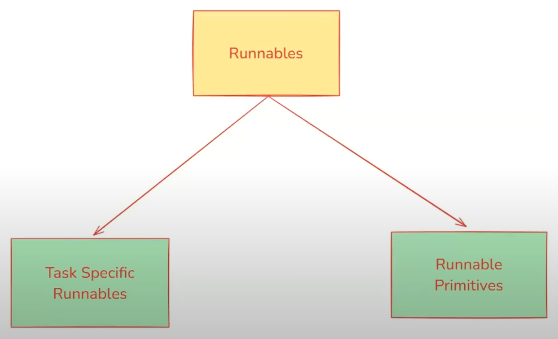
* Routes input to different runnables based on a condition.
* Works like a Conditional Chain in the old architecture.
* Enables dynamic decision-making in workflows.

1. **Runnable Paralle**l

* Executes multiple runnables simultaneously, merging results when done.
* Different from Runnable Map in that it’s not necessarily keyed by runnable name—it’s about parallel execution for speed.

**Why Runnables Matter ?**

* **Modern Replacement for Chains:** More flexible, composable, and performant.
* **Unified Interface:** You can mix LLMs, tools, retrievers, and custom logic without worrying about different method names.
* **Optimized for Async + Streaming:** Handles real-time output better than old chains.
* **Encourages Modular Design** – Easy to replace or reorder components.



1. **Task Specific Runnables**

* **Definition:** These are core LangChain components that have been converted into Runnables so they can be used in pipelines.
* **Purpose:** Perform task-specific operations like LLM calls, prompting, retrieval, etc.
* Examples:
  + **ChatOpenAI**→ Runs an LLM model.
  + **PromptTemplate** → Formats prompts dynamically.
  + **Retriever** →Retrieves relevant documents.

1. **Runnable Primitives**

* **Definition:** These are fundamental building blocks for structuring execution logic in Al workflows.
* **Purpose:** They help orchestrate execution by defining how different Runnables interact (sequentially, in parallel, conditionally, etc.).
* **Examples:**
  + **RunnableSequence** : Runs steps in order (| operator).
  + **RunnableParallel** : Runs multiple steps simultaneously. RunnableHap - Maps the same input across multiple functions.
  + **RunnableBranch** : Implements conditional execution (if-else logic). **RunnableLambda** : Wraps custom Python functions into Runnables.
  + **RunnablePassthrough** : Just forwards input as output (acts as a placeholder).

**RunnableSequence:**

* RunnableSequence is a sequential chain of runnables in LangChain that **executes each step one after another**, passing the output of one step as the input to the next.
* It is useful when you need to compose multiple runnables together in a structured workflow.

**RunnableParallel:**

* RunnableParallel is a runnable primitive that **allows multiple runnables to execute in parallel.**
* Each runnable receives the same input and processes it independently, producing a dictionary of outputs.

**RunnablePassThrough:**

* RunnablePassThrough is a special runnable primitive that **simply returns the input as output** without modifying it.

**RunnableLambda:**

* RunnableLambda is a runnable primitive that **allows you to apply custom Python functions** within an Al pipeline.
* It acts as a middleware between different Al components, enabling preprocessing, transformation, API calls, filtering, and post-processing in a LangChain workflow.

**RunnableBranch:**

* RunnableBranch is a control flow component in LangChain that **allows you to conditionally route input data to different chains or runnables** based on custom logic.
* It functions like an if/elif/else block for chains - where you define a set of condition functions, each associated with a runnable (e.g., LLM call, prompt chain, or tool). The first matching condition is executed. If no condition matches, a default runnable is used (if provided).

**LangChain Expression Language (LCEL)**

* LCEL is a declarative way to build LangChain pipelines using the Runnable interface.
* Instead of writing verbose Python glue code, you use operator overloading (like | for chaining) to connect components in a fluent, readable, and composable style.
* Think of it as a "pipeline syntax" for LangChain—making workflows look clean and intuitive, while still benefiting from the performance and features of Runnables.

**Benefits:**

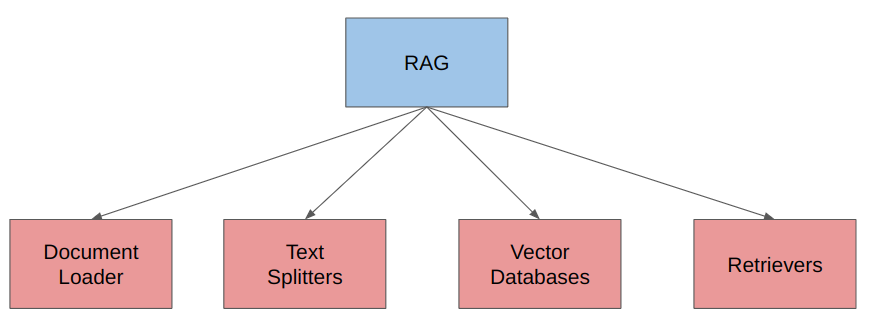
* **Readability:** Pipelines look like logical expressions rather than verbose wiring code.
* **Performance:** Inherits async, batching, and streaming optimizations from Runnables.
* **Composability:** Easily mix LLMs, prompt templates, retrievers, and custom functions.
* **Maintainability:** Easier to modify, reorder, or extend pipelines without refactoring large classes.
* **Uniform Behavior:** Works the same for single runs, batch runs, or streaming.

**Mental Model:**

* **Without LCEL:** You write procedural Python code connecting components manually.
* **With LCEL:** You “draw” the pipeline in a single expression, where the data flows through operators and containers.

**Retrieval Augmented Generation (RAG)**

* RAG is a technique that combines information retrieval with language generation, where a model retrieves relevant documents from a knowledge base and then uses them as context to generate accurate and grounded responses.
* Benefits of using RAG:
  + Use of up-to-date information
  + Better privacy
  + No limit of document size

****

**Document Loader**

* Document loaders are components in LangChain used to load data from various sources into a standardized format (usually as Document objects), which can then be used for chunking, embedding, retrieval, and generation.

Document(

page\_content="The actual text content",

metadata={"source": "filename.pdf", ...}

)

1. **Text Loader**

* TextLoader is a simple and commonly used document loader in LangChain that reads plain text (.txt) files and converts them into LangChain Document objects.
* Use Case
  + Ideal for loading chat logs, scraped text, transcripts, code snippets, or any plain text data into a LangChain pipeline.
* Limitation
  + Works only with .txt files

1. **PDF Loader**

* PyPDFLoader is a document loader in LangChain used to load content from PDF files and convert each page into a Document object.

[  
 Document(page\_content="Text from page 1", metadata={"page": 0, "source": "file.pdf"}},

Document(page\_content="Text from page 2", metadata={"page": 1, "source": "file.pdf"}),

]

* Limitation
  + It uses the PyPDF library under the hood - not great with scanned PDFs or complex layouts.

| **UseCase** | **Recommended Loader** |
| --- | --- |
| Simple, Clean PDFs | PyPDFLoader |
| PDFs with tables/columns | PDFPlumberLoader |
| Scanned/Image PDFs | UnstructuredPDFLoader or AmazonTextractPDFLoader |
| Need layout and Image data | PyMuPDFLoader |
| Want best structure extraction | UnstructuredPDFLoader |

1. **Directory Loader**

* It is a document loader that lets you load multiple documents from a directory of files.

| **Glob Pattern** | **What it loads** |
| --- | --- |
| “\*\*/\*.txt” | All .txt files in all subfolders |
| “\*.pdf” | All .pdf files in the root directory |
| “data/\*.csv” | All .csv files in the data/ folder |
| “\*\*/\*” | All files (any type, all folders) |

\*\* = recursive search through subfolders

* load()
  + Eager Loading (loads everything at once)
  + Returns: A list of Document objects
  + Loads all documents immediately into memory.
  + Best When:
    - The number of documents is small.
    - You want everything loaded upfront.
* lazy\_load()
  + Lazy loading (loads on demand).
  + Returns: A generator of Document objects.
  + Documents are not all loaded at once: they’re fetched one at a time as needed.
  + Best when:
    - Dealing with large documents or lots of files.
    - Want to stream processing (e.g Chunking, embedding) without using lots of memory.

1. **WebBaseLoader**

* WebBaseLoader is a document loader in LangChain used to load and extract text content from web pages (URLs).
* It uses BeautifulSoup under the hood to parse HTML and extract visible text.
* When to Use:
  + For blogs, news articles, or public websites where the content is primarily text-based and static.
* Limitations:
  + Doesn't handle JavaScript-heavy pages well (use Selenium URLLoader for that).
  + Loads only static content (what's in the HTML, not what loads after the page renders).

1. **CSVLoader**

* CSVLoader is a document loader used to load CSV files into langchain Document objects, one per row, by default.

**Text Splitters**

* Text Splitters are utilities in LangChain that break large text into smaller chunks so they can be efficiently processed by LLMs or stored in vector databases.
* Splitting ensures that:
  + The text fits within model token limits.
  + Retrieval and search work with manageable, context-rich segments.
* Splitting strategy impacts how much context and meaning is preserved in each chunk.

1. **Text Based Splitting**

* **Logic:** Splits text purely based on character count or token count, often with some overlap to preserve context.
* **Pros:**
  + Simple and fast.
  + Works for all text formats.
* **Cons:**
  + Ignores sentence or paragraph boundaries, which can break meaning mid-sentence.
* **Use Case:** Large unstructured text where structure doesn’t matter (e.g., raw logs, scraped HTML).

1. **Text Structure Based Splitting**

* **Logic:** Splits text according to natural language boundaries—such as sentences, paragraphs, or bullet points—while respecting a target chunk size.
* **Pros:**
  + Preserves readability and meaning better than pure length-based splitting.
  + More natural for LLM processing.
* **Cons:**
  + Slightly slower because it needs to parse boundaries.
* **Use Case:** Articles, essays, and other prose where sentence structure matters.

1. **Document-Structure Based Splitting**

* **Logic:** Splits based on explicit document elements like headings, sections, tables, or metadata tags.
* **Pros:**
  + Maintains logical sections and context.
  + Works well for structured documents like Markdown, PDFs, HTML.
* **Cons:**
  + Requires documents with clear structure.
* **Use Case:** Reports, academic papers, technical documentation.

1. **Semantic Meaning Based Splitting**

* **Logic:** Uses semantic similarity or embeddings to determine where to split, grouping text so each chunk has a coherent meaning.
* **Pros:**
  + Keeps semantically related content together, even if length varies.
  + Best for retrieval tasks needing high contextual accuracy.
* **Cons:**
  + More computationally expensive.
  + Requires embedding models or advanced NLP processing.
* **Use Case:** Knowledge bases, Q&A systems, semantic search pipelines.

**Vector Store**

* A vector store is a system designed to store and retrieve data represented as numerical vectors.
* **Key Features:**
  + Storage: Ensures that vectors and their associated metadata are retained, whether in- memory for quick lookups or on-disk for durability and large-scale use.
  + Similarity Search - Helps retrieve the vectors most similar to a query vector.
  + Indexing: Provide a data structure or method that enables fast similarity searches on high-dimensional vectors (e.g., approximate nearest neighbor lookups).
  + CRUD Operations: Manage the lifecycle of data-adding new vectors, reading them, updating existing entries, removing outdated vectors.
* **Use-cases**
  + Semantic Search
  + RAG
  + Recommender Systems 4. Image/Multimedia search

**Vector Store vs Vector Database**

**Vector Store**

* Typically refers to a lightweight library or service that focuses on storing vectors (embeddings) and performing similarity search.
* May not include many traditional database features like transactions, rich query languages, or role-based access control.
* Ideal for prototyping, smaller-scale applications
* Examples FAISS where you store vectors and can query them by similarity, but you handle persistence and scaling separately).

**Vector Database**

* A full-fledged database system designed to store and query vectors.
* Offers additional "database-like" features:
  + Distributed architecture for horizontal scaling
  + Durability and persistence (replication, backup/restore)
  + Metadata handling (schemas, filters)
  + Potential for ACID or near-ACID guarantees
  + Authentication/authorization and more advanced security
* Geared for production environments with significant scaling, large datasets
* Examples: Milvus, Qdrant, Weaviate, Pine Cone

**Note:** A vector database is effectively a vector store with extra database features (e.g., clustering, scaling, security, metadata filtering, and durability)

**VectorStore in LangChain**

* **Supported Stores:** LangChain integrates with multiple vector stores (FAISS, Pinecone, Chroma, Qdrant, Weaviate, etc.), giving you flexibility in scale, features, and deployment.
* **Common Interface:** A uniform Vector Store API lets you swap out one backend (e.g., FAISS) for another (e.g., Pinecone) with minimal code changes.
* **Metadata Handling:** Most vector stores in LangChain allow you to attach metadata (e.g., timestamps, authors) to each document, enabling filter-based retrieval.