Runnables in LangChain

1. The Core Problem LangChain Solved: LLM Interaction Standardization

LangChain's inception around November 2022 coincided with the public release of ChatGPT and OpenAI's APIs. The LangChain team quickly identified a growing demand for LLM-based applications. A primary challenge was the **disparate APIs offered by various LLM providers** (OpenAI, Anthropic, Google, Mistral, etc.).

- **Problem:** "हर कंपनी का एपीआई अलग तरीके से बिहेव करता था" (Every company's API behaved differently).
- **LangChain's Initial Solution:** Create a framework that allowed developers to interact with any LLM API with minimal code changes. This standardization of LLM interaction was LangChain's first major contribution and led to its initial popularity.

2. Expanding Beyond LLM Interaction: Comprehensive AI Application Development

The LangChain team soon realized that interacting with LLMs was only a small part of building a complete LLM-based application. Many other crucial tasks were involved, such as:

- **Document Loading:** Fetching documents from various sources.
- **Text Splitting:** Breaking down large documents into smaller, manageable chunks.
- **Embedding Generation:** Converting text chunks into numerical representations (embeddings).
- **Vector Database Storage:** Storing embeddings for efficient retrieval.
- **Retrieval (Semantic Search):** Finding relevant text chunks based on user queries.
- **Output Parsing:** Structuring and displaying the LLM's response to the user.
- **Problem:** "एलएलएम से इंटरेक्ट करना इस पूरे एप्लीकेशन को बिल्ड करने का एक बहुत छोटा पार्ट है" (Interacting with LLMs is a very small part of building the entire application).
- **LangChain's Solution:** Develop a wide range of **components** for each of these tasks. Examples include Document Loaders, Text Splitters, Embedding Models, Vector Databases, Retrievers, and Output Parsers. This transformed LangChain into a powerful, comprehensive library for building LLM applications.

3. The Rise and Fall of "Chains": An Attempt at Abstraction (and its Pitfalls)

With a plethora of components available, developers began manually connecting them to build complex applications. The LangChain team observed common workflows, like generating a prompt and sending it to an LLM.

- **Observation:** Certain sequences of component interactions were frequently reused across different LLM applications. For instance, prompt creation and LLM interaction were universal.
- **LangChain's Idea (Chains):** Automate these common multi-component tasks by creating "built-in functions" called **Chains**.
- **LLMChain:** The simplest chain, automating prompt generation and sending it to an LLM. "एलएलएम चेन जिसका काम है कि अगर आप उसको एक एलएलएम प्रोवाइड करते हो और एक प्रम्प्ट प्रोवाइड करते हो तो वो प्रम्प्ट जनरेट करके एलएलएम के पास उस प्रम्प को भेज सकता है" (LLMChain's job is that if you provide it an LLM and a prompt, it can generate the prompt and send it to the LLM).
- **RetrievalQAChain:** For RAG (Retrieval Augmented Generation) applications, it automates retrieving relevant documents, forming a prompt, and sending it to the LLM.

Chains significantly simplified development, reducing lines of code and abstracting manual steps. LangChain then created **"too many chains"** for various use cases (e.g., SimpleSequentialChain, SequentialChain, SQL chains, API chains, Math chains).

- **Problem (Anticipated):** This proliferation of Chains led to two major disadvantages:

1. **Large Codebase & Maintenance Burden:** "लैंग चेन का जो कोड बेस है वो बहुत बड़ा हो गया" (LangChain's codebase became very large).

2. **Steep Learning Curve:** "जो नए एआई इंजीनियर्स लैंग चेन सीख रहे थे उनको यही समझ में नहीं आ रहा था कि कौन से यूज़ केस के लिए कौन सा चेन होता है" (New AI engineers learning LangChain couldn't understand which chain to use for which use case). The learning curve became "very steep."

4. The Root Cause of Chain Proliferation: Lack of Component Standardization

The fundamental reason for the "too many chains" problem was that LangChain's initial components (LLM, Prompt, Parsers, Retrievers) were **not standardized**. They were developed independently with different interaction methods.

- **Example:** "अगर आप बात करो एलएलएम वाले कंपोनेंट की तो इससे इंटरेक्ट करने के लिए आपको प्रिडिक्ट फंक्शन को कॉल करना होता है वेयर एज आप प्रप टैंपलेट के लिए फॉर्मेट फंक्शन यूज़ करते हो आप रिट्रीवर के लिए गेट रेलेवेंट डॉक्यूमेंट्स फंक्शन को कॉल करते हो" (If you talk about the LLM component, you call the predict function to interact with it, whereas for the Prompt Template you use the format function, and for the Retriever you call the get_relevant_documents function).
- **Consequence:** Because components had incompatible interfaces, every time the LangChain team wanted to connect two or more components, they had to write **custom code** to bridge these differences, leading to the creation of a new Chain function. "उनको हर नए यूज़ केस के लिए नया कस्टम कोड एज इन नया फंक्शन बनाना पड़ा" (They had to create new custom code, i.e., a new function, for every new use case).

5. The Solution: Runnables – Standardized, Composable Units of Work

LangChain's team realized their mistake: they needed to rebuild their components to be **standardized and seamlessly connectable**. This realization led to the concept of **Runnables**.

- **Definition:** "रनेबल्स को आप एक यूनिट ऑफ वर्क बुला सकते हो" (Runnables can be called a unit of work).
- **Key Characteristics of Runnables:**

1. **Unit of Work:** Each Runnable has a specific purpose, takes an input, processes it, and provides an output.

2. **Common Interface:** All Runnables adhere to a consistent interface, meaning they have the "same set of methods" with the "same names."

- **invoke():** The most important method, taking a single input and returning a single output.
- **batch():** Processes multiple inputs simultaneously and returns multiple outputs.
- **stream():** Enables streaming output.

1. **Composability:** Because of their common interface, Runnables can be seamlessly connected, forming complex workflows. The output of one Runnable automatically becomes the input for the next. "आप रनेबल्स को किसी भी तरीके से आपस में कनेक्ट कर करके कितना भी कॉम्प्लेक्स वर्क फ्लो बना सकते हो" (You can connect Runnables in any way to create any complex workflow).

2. **Nested Runnables:** When multiple Runnables are connected to form a workflow (a Chain), the resulting workflow *itself* is also a Runnable. This allows for hierarchical composition, enabling the creation of extremely large and complex structures by connecting smaller workflows.

- **Analogy:** Runnables are like **Lego blocks**. Each block (component) has a purpose, they all have a standard connection mechanism (common interface), they can be connected in any way to build structures (composability), and the resulting structure can itself be treated as a new Lego block (nested Runnables).

6. Implementation of Runnables: Abstraction and Forcing Standardization

To enforce the common interface, LangChain uses **Abstract Base Classes (ABCs)** in Python.

- A base Runnable abstract class is defined with abstract methods like invoke().
- All component classes (e.g., LLM, PromptTemplate, OutputParser) inherit from Runnable.
- This inheritance *forces* these component classes to implement the invoke() method (and other common methods). If they don't, an error is thrown.
- Existing methods like predict() (for LLM) and format() (for PromptTemplate) are deprecated and developers are encouraged to use invoke() instead.

This standardization ensures that any Runnable, regardless of its underlying function, can be interacted with using the same set of methods. This unified interface is what enables the flexible and powerful chaining capabilities without the need for endless custom Chain functions.

The RunnableConnector class shows how a generic class can connect any list of Runnables, as the output of one is seamlessly passed as input to the next, thanks to the standardized invoke() method.

In essence, Runnables represent LangChain's evolution towards a more robust, flexible, and maintainable framework, addressing the limitations of its earlier "Chains" by enforcing a universal standard for component interaction.