**Runnable in chains**

**Understanding LangChain Runnables**

2024-07-10 · 22 min read · By William Bakst

A LangChain runnable is a protocol that allows you to create and invoke custom chains. It’s designed to sequence tasks, taking the output of one call and feeding it as input to the next, making it suitable for straightforward, linear tasks where each step directly builds upon the previous one.

Runnables simplify the process of building, managing, and modifying complex workflows by providing a standardized way for different components to interact. With a single function call, you can execute a chain of operations - which is useful for scenarios where the same series of steps need to be applied multiple times.

A runnable consists of several parts. At a minimum, these include:

* **Methods:** these are the functions that a runnable can perform. The standard interface of a runnable includes methods likeinvoke , batch , stream , and their corresponding async methods (ainvoke , abatch , astream). These methods allow you to define custom chains and invoke them in a standard way.
* **Input and output schemas:** all runnables expose input and output schemas, allowing you to inspect and understand the input type a runnable expects and the output type it produces.
* **Components:** various components in LangChain implement the runnable interface, including (but not limited to) chat models, large language models (LLMs), output parsers, retrievers, [prompt templates](https://mirascope.com/blog/langchain-prompt-template), and more.

With runnables, you can easily combine components using the pipe (|) operator to build workflows for complex chains, which lets you link together prompts and function calls to architect complex flows - though at the cost of potential challenges in error handling, performance, and transparency.

One of the key characteristics of runnables is their ability to handle concurrency. The runnable interface includes async methods that can be used with [asyncio’s](https://docs.python.org/3/library/asyncio.html) await syntax for concurrency. This allows you to run multiple tasks concurrently, improving the efficiency and performance of your applications.

Moreover, if a component in a chain fails, the error will be propagated up the chain, making it easier to handle errors and exceptions, and implement fallbacks in a more consistent and predictable way.

While LangChain's **'runnable’** offers a great solution for creating and invoking custom chains - a functionality we appreciate - **runnables are another abstraction that LLM app developers have to learn**. And while the LangChain Expression Language (LCEL) works well for simple chains, they become increasingly complicated to work with as chains grow more complex.

That’s why we created [Mirascope](https://github.com/mirascope/mirascope), our Python toolkit for building with LLMs. It offers building blocks rather than a monolithic framework, and allows you to code as you normally would in Python without having to learn new abstractions.

In this article, we provide an overview of how a LangChain Runnable works, along with some of its pros and cons. Then, we contrast this with Mirascope’s approach to chaining.

How Runnables Work in LangChain

In LangChain, a [runnable can be any Python callable](https://python.langchain.com/v0.1/docs/expression_language/interface/), such as a function, a lambda expression, or an instance method of a class. However, instead of directly passing these callables around, you wrap them in a runnable object to provide additional functionality and metadata, like the function name, execution time, or custom annotations.

Here's an example of how you can create a runnable from a function:

from langchain\_core.runnables import RunnableLambda

# Define a simple function

def greet(name):

return f"Hello, {name}!"

# Wrap the function in a RunnableWrapper

greet\_runnable = RunnableLambda(lambda x: greet(x))

# Use the runnable to call the function

result = greet\_runnable.invoke("Alice")

print(result) # Output: Hello, Alice!

In the above code, we defined a simple greet function that takes a name as an argument and returns a greeting string. This function is then wrapped in a RunnableWrapper.

greet\_runnable provides additional functionality and metadata, making it easier to integrate with other parts of your code. This allows you to manage and pass around multiple callables with additional context or behavior.

One advantage of wrapping callables as runnables is you can now connect them using LangChain's chaining mechanisms, such as the pipe operator (|), the RunnableSequence class, or the .pipe( ) method.

For example, you can use [RunnableSequence](https://api.python.langchain.com/en/latest/runnables/langchain_core.runnables.base.RunnableSequence.html) to create a chain applying multiple transformations to some input data: ‍

from datetime import datetime

from langchain\_core.runnables import RunnableLambda, RunnableSequence

# Define the transformations as simple functions

def greet(name):

return f"Hello, {name}!"

def append\_datetime(text):

current\_datetime = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

return f"{text} The current date and time is {current\_datetime}."

def to\_uppercase(text):

return text.upper()

def add\_exclamation(text):

return f"{text}!"

# Wrap the functions in RunnableWrapper

greet\_runnable = RunnableLambda(lambda x: greet(x))

datetime\_runnable = RunnableLambda(lambda x: append\_datetime(x))

uppercase\_runnable = RunnableLambda(lambda x: to\_uppercase(x))

exclamation\_runnable = RunnableLambda(lambda x: add\_exclamation(x))

# Create a RunnableSequence with the wrapped runnables

chain = RunnableSequence(

first=greet\_runnable,

middle=[datetime\_runnable, uppercase\_runnable],

last=exclamation\_runnable,

)

# Apply the chain to some input data

input\_data = "Alice"

result = chain.invoke(input\_data)

print(

result

) # Output example: "HELLO, ALICE! THE CURRENT DATE AND TIME IS 2024-06-19 14:30:00!"

Here we have four simple functions: greet , append\_datetime , to\_uppercase , and add\_exclamation , each of which takes input and performs a specific transformation on it. RunnableLambda takes a function as its argument, and creates a runnable object.

We can then create a RunnableSequence by passing these runnables to its constructor:

chain = RunnableSequence(

first=greet\_runnable,

middle=[datetime\_runnable, uppercase\_runnable],

last=exclamation\_runnable,

)

RunnableSequence executes these runnables in sequential order, using the output of one runnable as input to the next.

The result of a chain is a RunnableSequence which is still a runnable that can still be piped, invoked, streamed, etc.

Creating A Runnable with the Chain Decorator

The @chain decorator allows you to turn any function into a chain. Below, the decorator creates a custom chain that combines multiple components, such as [prompts](https://mirascope.com/blog/prompt-engineering-tools), models, and output parsers, and defines a function (custom\_chain) that encapsulates the sequence of operations:

from langchain\_core.output\_parsers import StrOutputParser

from langchain\_core.prompts import ChatPromptTemplate

from langchain\_core.runnables import chain

from langchain\_openai import ChatOpenAI

prompt1 = ChatPromptTemplate.from\_template("Tell me a joke about {topic}")

prompt2 = ChatPromptTemplate.from\_template("What is the subject of this joke: {joke}")

@chain

def custom\_chain(text):

prompt\_val1 = prompt1.invoke({"topic": text})

output1 = ChatOpenAI().invoke(prompt\_val1)

parsed\_output1 = StrOutputParser().invoke(output1)

chain2 = prompt2 | ChatOpenAI() | StrOutputParser()

return chain2.invoke({"joke": parsed\_output1})

custom\_chain.invoke("bears")

# Output: 'The subject of this joke is bears.'

invoke,batch, andstreamMethods

As previously mentioned, LangChain runnables provide three key methods to execute and interact with your chains:

* invoke: executes a runnable with a single input, and is typically used when you have a single piece of data to process.
* batch: allows you to process multiple inputs in parallel. This method is useful when you have a list of inputs and want to run them through the chain simultaneously. ‍
* stream: processes input data as a stream, handling one piece of data at a time and providing results as they are available. This method is ideal for handling streamed output for real-time data processing or for large datasets that you want to process incrementally. At the time of this writing, streaming support for retries is being added for higher reliability without any latency cost (as explained in their docs).

Key Runnable Types in LangChain

Within LangChain, you have access to various runnable types that allow you to execute and manage tasks:

* RunnableParallel for parallelizing operations.
* RunnablePassthrough for passing data unchanged from previous steps for use as input in later steps.
* RunnableLambda for converting a Python callable into a runnable.

RunnableParallel

This runs a mapping of runnables in parallel and returns a mapping of their outputs. It’s essentially a dictionary whose values are runnables, and it invokes them concurrently, providing the same input to each.

A [RunnableParallel](https://python.langchain.com/v0.1/docs/expression_language/primitives/parallel/) can be instantiated directly or by using a dictionary literal within a sequence. This is particularly useful when you want to parallelize operations or manipulate the output of one runnable to match the input format of the next runnable in a sequence.

Below is an example that uses functions to illustrate how RunnableParallel works.

import asyncio

from langchain\_core.runnables import RunnableLambda

def add\_one(x: int) -> int:

return x + 1

def mul\_two(x: int) -> int:

return x \* 2

def mul\_three(x: int) -> int:

return x \* 3

runnable\_1 = RunnableLambda(add\_one)

runnable\_2 = RunnableLambda(mul\_two)

runnable\_3 = RunnableLambda(mul\_three)

sequence = runnable\_1 | { # this dict is coerced to a RunnableParallel

"mul\_two": runnable\_2,

"mul\_three": runnable\_3,

}

# Or equivalently:

# sequence = runnable\_1 | RunnableParallel(

# {"mul\_two": runnable\_2, "mul\_three": runnable\_3}

# )

# Also equivalently:

# sequence = runnable\_1 | RunnableParallel(

# mul\_two=runnable\_2,

# mul\_three=runnable\_3,

# )

print(sequence.invoke(1))

# > {'mul\_two': 4, 'mul\_three': 6}

print(sequence.batch([1, 2, 3]))

# > [{'mul\_two': 4, 'mul\_three': 6}, {'mul\_two': 6, 'mul\_three': 9}, {'mul\_two': 8, 'mul\_three': 12}]

async def async\_invoke(sequence, x):

return await sequence.ainvoke(x)

async def async\_batch(sequence, x):

return await sequence.abatch(x)

print(asyncio.run(async\_invoke(sequence, 1)))

# > {'mul\_two': 4, 'mul\_three': 6}

print(asyncio.run(async\_batch(sequence, [1, 2, 3])))

# > [{'mul\_two': 4, 'mul\_three': 6}, {'mul\_two': 6, 'mul\_three': 9}, {'mul\_two': 8, 'mul\_three': 12}]

RunnablePassthrough

This is a [runnable](https://python.langchain.com/v0.1/docs/expression_language/primitives/passthrough/) that passes inputs through unchanged or with additional keys. It behaves almost like the identity function, except that it can be configured to add additional keys to the output, if the input is a dictionary.

It’s often used in conjunction with RunnableParallel to pass data through to a new key in the map, which allows you to keep the original input intact while adding some extra information.

# !pip install -qU langchain langchain-openai

import os

from langchain\_core.runnables import RunnableParallel, RunnablePassthrough

os.environ["OPENAI\_API\_KEY"] = "YOUR\_API\_KEY"

runnable = RunnableParallel(

passed=RunnablePassthrough(),

modified=lambda x: x["num"] + 1,

)

runnable.invoke({"num": 1})

# Output: {'passed': {'num': 1}, 'modified': 2}

Here, the passed key was called with RunnablePassthrough, passing on the input data {'num': 1} without changes. And, the modified key was set using a lambda that added 1 to num, resulting in modified having the value 2.

RunnableLambda

[RunnableLambda](https://api.python.langchain.com/en/latest/runnables/langchain_core.runnables.base.RunnableLambda.html) is a LangChain abstraction that allows Python-callable functions to be transformed into functions compatible with LangChain's pipeline operations.

Wrapping a callable in a RunnableLambda makes the callable usable within either a sync or async context and can be composed as any other runnable. ‍

# This is a RunnableLambda

from langchain\_core.runnables import RunnableLambda

def add\_one(x: int) -> int:

return x + 1

runnable = RunnableLambda(add\_one)

runnable.invoke(1) # returns 2

runnable.batch([1, 2, 3]) # returns [2, 3, 4]

# Async is supported by default by delegating to the sync implementation

await runnable.ainvoke(1) # returns 2

await runnable.abatch([1, 2, 3]) # returns [2, 3, 4]

# Alternatively, can provide both synd and sync implementations

async def add\_one\_async(x: int) -> int:

return x + 1

runnable = RunnableLambda(add\_one, afunc=add\_one\_async)

runnable.invoke(1) # Uses add\_one

await runnable.ainvoke(1) # Uses add\_one\_async

As shown above, the code handles individual values and batches of data, using the provided sync and async implementations.

Chaining with Mirascope

[Mirascope](https://mirascope.com/docs/mirascope) dispenses with abstractions like runnables and offers two ways of chaining: computed fields and function arguments.

Chaining with Computed Fields

An example of chaining using Python's native functionality is shown below, where explain\_book first calls recommend\_book once and injects the result into the prompt template as a computed field:

from mirascope.core import openai, prompt\_template

@openai.call("gpt-4o-mini")

@prompt\_template(

"""

Recommend a popular book in the {genre} genre.

Give me just the title.

"""

)

def recommend\_book(genre: str): ...

@openai.call("gpt-4o-mini")

@prompt\_template(

"""

SYSTEM:

You are the world's greatest librarian.

Your task is to explain why the book "{book\_title}" is popular in the {genre} genre.

USER:

Explain why "{book\_title}" in the {genre} genre is popular.

"""

)

def explain\_book(genre: str) -> openai.OpenAIDynamicConfig:

return {"computed\_fields": {"book\_title": recommend\_book(genre)}}

explanation = explain\_book("science fiction")

print(explanation)

# > "Dune," written by Frank Herbert, has garnered immense popularity in the science fiction genre...

Additionally, computed\_fields includes the output at every step of the chain in the final dump:

**Runnables** in LangChain provide a structured and modular way to define, execute, and manage tasks within a pipeline. They encapsulate various components such as models, tools, chains, and custom logic, making them reusable and composable. This feature is particularly useful for building complex workflows, as it allows developers to break down tasks into smaller, manageable units.

**Key Concepts of Runnables**

**Modularity**:

* A **Runnable** is a building block that represents a single task or operation.
* Tasks can include running a language model, processing data, or chaining multiple operations together.

**Composability**:

* Multiple Runnables can be linked together to form a pipeline.
* This facilitates building complex workflows from smaller, reusable components.

**Reusability**:

* Once defined, a Runnable can be used in different workflows without modification.
* This is particularly helpful for tasks like data preprocessing or standard operations that recur across projects.

**Asynchronous Execution**:

* Runnables support asynchronous execution, enabling faster and more efficient task management, especially in workflows involving I/O-bound operations.

**Key Features of Runnables**

**Ease of Integration**:

* Runnables integrate seamlessly with LangChain’s ecosystem of tools, models, chains, and utilities.

**Parallel Execution**:

* They can be configured to execute in parallel where appropriate, optimizing performance for batch tasks.

**Error Handling**:

* Runnables often include mechanisms for capturing and handling errors, ensuring robustness in workflows.

**Logging and Debugging**:

* They support logging and metadata propagation, making it easier to monitor and debug workflows.

**Types of Runnables**

**Basic Runnables**:

* Perform simple operations such as string manipulation or data transformation.

**Model-Based Runnables**:

* Interact with language models like OpenAI’s GPT, Hugging Face models, or other custom LLMs.
* Example: Prompting an LLM for summarization.

**Tool Runnables**:

* Utilize external tools such as search engines, APIs, or custom scripts.

**Chain Runnables**:

* Combine multiple components into a sequential or parallel workflow.

**Custom Runnables**:

* Allow developers to define their logic using Python functions or classes.

**Core API Components**

**1. Runnable: Base Class**

The Runnable class serves as the foundational building block. All other specialized Runnables inherit from this class.

from langchain.schema.runnable import Runnable  
  
class MyRunnable(Runnable):  
 def invoke(self, input):  
 return input.upper()  
  
# Create an instance of MyRunnable  
runnable = MyRunnable()  
  
# Test with a sample input  
result = runnable.invoke("hello world")  
print(result) # Output: HELLO WORLD  
  
# Try another example  
result = runnable.invoke("LangChain is awesome")  
print(result) # Output: LANGCHAIN IS AWESOME

**2. RunnableMap**

Executes multiple Runnables in parallel and aggregates their results

from langchain.schema.runnable import RunnableMap  
  
runnable\_map = RunnableMap({  
 "uppercase": lambda x: x.upper(),  
 "reverse": lambda x: x[::-1],  
})  
  
result = runnable\_map.invoke("langchain")  
# Output: {'uppercase': 'LANGCHAIN', 'reverse': 'niahcnagL'}

**3. RunnableSequence**

Chains Runnables sequentially, passing the output of one as input to the next.

from langchain.schema.runnable import RunnableSequence  
  
runnable\_sequence = RunnableSequence([  
 lambda x: x.lower(),  
 lambda x: x[::-1],  
])  
  
result = runnable\_sequence.invoke("LangChain")  
# Output: 'niahcnag'

**4. RunnableLambda**

Wraps a simple Python function in a Runnable.

from langchain.schema.runnable import RunnableLambda  
  
uppercase\_runnable = RunnableLambda(lambda x: x.upper())  
result = uppercase\_runnable.invoke("langchain")  
# Output: 'LANGCHAIN'

**Example: End-to-End Workflow**

**Problem Statement:**

Process customer feedback, classify its sentiment, and summarize it.

**Solution with Runnables:**

from langchain.prompts import PromptTemplate  
from langchain.llms import OpenAI  
from langchain.schema.runnable import Runnable, RunnableSequence, RunnableMap,RunnableLambda  
  
# Define individual Runnables  
sentiment\_analysis\_runnable = RunnableLambda(lambda text: "Positive" if "good" in text.lower() else "Negative")  
  
summarization\_runnable = RunnableSequence([  
 PromptTemplate(input\_variables=["text"], template="Summarize this: {text}"),  
 OpenAI()  
])  
  
# Combine Runnables into a pipeline  
pipeline = RunnableMap({  
 "sentiment": sentiment\_analysis\_runnable,  
 "summary": summarization\_runnable  
})  
  
# Invoke the pipeline  
feedback = "The product quality is really good and exceeded expectations."  
result = pipeline.invoke(feedback)  
  
print(result)  
# Output:  
# {  
# "sentiment": "Positive",  
# "summary": "The product quality is excellent."  
# }

**Benefits of Runnables**

**Simplified Workflow Creation**:

* Runnables abstract away boilerplate code, focusing on core logic.

**Improved Scalability**:

* Their composable nature makes scaling up workflows straightforward.

**Enhanced Debugging**:

* Logs and modular design make troubleshooting easier

**Flexibility**:

* Custom Runnables allow complete control over behavior, ensuring adaptability to various use cases.

**When to Use Runnables?**

* **Data Processing Pipelines**: Modularize tasks like data cleaning, transformation, and extraction.
* **Model-Oriented Workflows**: Create end-to-end pipelines for tasks like text summarization, Q&A, or sentiment analysis.
* **Complex Applications**: Build AI systems involving multiple steps or interacting components.
* **Reusable Components**: Encapsulate frequently used operations for better maintainability.

**1. RunnableBinding**

A RunnableBinding allows you to bind fixed arguments to a callable and create a reusable Runnable. It's useful when you want to fix certain parameters while still using the callable with different inputs.

**Example:**

from langchain.schema.runnable import RunnableBinding  
  
# Bind fixed arguments to a lambda function  
binding = RunnableBinding(lambda x, y: x + y, y=10)  
result = binding.invoke(5)  
print(result) # Output: 15

**1. RunnableBindingBase**

This is a base class for creating Runnables that bind specific inputs or arguments to a callable, making it reusable with fixed parameters.

**Use Case:**

When you need a reusable Runnable with pre-bound arguments.

**Example:**

from langchain.schema.runnable import RunnableBindingBase  
  
class AddConstant(RunnableBindingBase):  
 def \_\_init\_\_(self, constant):  
 self.constant = constant  
  
 def invoke(self, input):  
 return input + self.constant  
  
add\_five = AddConstant(5)  
result = add\_five.invoke(10) # Adds 5 to the input  
print(result) # Output: 15

**2. RunnableEach**

Executes a Runnable on each element of a collection (e.g., a list or dictionary) independently and aggregates the results.

**Use Case:**

Parallel processing of individual items in a list or dictionary.

**Example:**

from langchain.schema.runnable import RunnableEach  
  
uppercase\_each = RunnableEach(lambda x: x.upper())  
result = uppercase\_each.invoke(["langchain", "runnables"])  
print(result) # Output: ['LANGCHAIN', 'RUNNABLES']

**3. RunnableEachBase**

The base class for Runnables that operate on collections. Similar to RunnableEach, but used for implementing custom logic.

**Use Case:**

When you want to create custom logic for iterating over collections.

**Example:**

from langchain.schema.runnable import RunnableEachBase  
  
class CustomEach(RunnableEachBase):  
 def invoke(self, inputs):  
 return [x[::-1] for x in inputs] # Reverses each string in the collection  
  
custom\_runnable = CustomEach()  
result = custom\_runnable.invoke(["langchain", "runnables"])  
print(result) # Output: ['niahcnagL', 'selbannuR']

**4. RunnableGenerator**

Generates outputs iteratively for a given input, often used for streaming use cases.

**Use Case:**

Stream outputs from a model or a long-running process.

**Example:**

from langchain.schema.runnable import RunnableGenerator  
  
class StreamWords(RunnableGenerator):  
 def invoke(self, input):  
 for word in input.split():  
 yield word  
  
stream\_runnable = StreamWords()  
for word in stream\_runnable.invoke("Streaming is powerful"):  
 print(word)  
# Output:  
# Streaming  
# is  
# powerful

**5. RunnableLambda**

Wraps a Python lambda or callable as a Runnable, allowing simple functions to integrate into pipelines.

**Use Case:**

Quickly wrap a lambda or function as a Runnable.

**Example:**

from langchain.schema.runnable import RunnableLambda  
  
reverse\_text = RunnableLambda(lambda x: x[::-1])  
result = reverse\_text.invoke("langchain")  
print(result) # Output: 'niahcnagL'

**6. RunnableMap**

Applies multiple Runnables to a single input and combines their outputs into a dictionary.

**Use Case:**

Split an input into multiple tasks and combine results.

**Example:**

from langchain.schema.runnable import RunnableMap  
  
pipeline = RunnableMap({  
 "uppercase": lambda x: x.upper(),  
 "reverse": lambda x: x[::-1],  
})  
  
result = pipeline.invoke("langchain")  
print(result) # Output: {'uppercase': 'LANGCHAIN', 'reverse': 'niahcnagL'}

**7. RunnableParallel**

Runs multiple Runnables in parallel and combines the results.

**Use Case:**

Optimize performance by executing tasks concurrently.

**Example:**

from langchain.schema.runnable import RunnableParallel  
  
parallel\_tasks = RunnableParallel({  
 "uppercase": lambda x: x.upper(),  
 "reverse": lambda x: x[::-1],  
})  
  
result = parallel\_tasks.invoke("langchain")  
print(result) # Output: {'uppercase': 'LANGCHAIN', 'reverse': 'niahcnagL'}

**8. RunnableSequence**

Chains multiple Runnables sequentially, where the output of one is the input for the next.

**Use Case:**

Create a linear pipeline of operations.

**Example:**

from langchain.schema.runnable import RunnableSequence  
  
sequence = RunnableSequence([  
 lambda x: x.lower(),  
 lambda x: x[::-1],  
])  
  
result = sequence.invoke("LangChain")  
print(result) # Output: 'niahcnag'

**9. RunnableSerializable**

Allows Runnables to be serialized (e.g., to save and reload them).

**Use Case:**

Serialize Runnables for sharing or storage.

**Example:**

from langchain.schema.runnable import RunnableSerializable  
  
class SerializableRunnable(RunnableSerializable):  
 def invoke(self, input):  
 return input.upper()  
  
serializable = SerializableRunnable()  
result = serializable.invoke("langchain")  
print(result) # Output: 'LANGCHAIN'

**10. RunnableBranch**

Selectively executes one of several branches based on input.

**Use Case:**

Implement conditional workflows.

**Example:**

from langchain.schema.runnable import RunnableBranch  
  
branch = RunnableBranch({  
 "is\_positive": lambda x: "Positive" if x > 0 else "Negative",  
 "is\_even": lambda x: "Even" if x % 2 == 0 else "Odd",  
})  
  
result = branch.invoke(10)  
print(result)  
# Output: {'is\_positive': 'Positive', 'is\_even': 'Even'}

**12.DynamicRunnable**

Creates a Runnable dynamically based on a provided function or callable. Useful for quick prototyping or dynamic behavior.

**Example:**

from langchain.schema.runnable import DynamicRunnable  
  
dynamic\_runnable = DynamicRunnable(lambda x: x \* 2)  
result = dynamic\_runnable.invoke(5)  
print(result) # Output: 10

**13.RunnableConfigurableAlternatives**

Allows switching between different Runnable implementations based on configuration or input.

**Example:**

from langchain.schema.runnable import RunnableConfigurableAlternatives  
  
configurable = RunnableConfigurableAlternatives({  
 "method1": lambda x: x + 1,  
 "method2": lambda x: x \* 2,  
})  
  
result = configurable.invoke(5, config={"choice": "method2"})  
print(result) # Output: 10

**14.RunnableWithMessageHistory**

Tracks input and output history for debugging or logging purposes.

**Example:**

from langchain.schema.runnable import RunnableWithMessageHistory  
  
class HistoryRunnable(RunnableWithMessageHistory):  
 def invoke(self, input):  
 return input.upper()  
  
runnable = HistoryRunnable()  
result = runnable.invoke("langchain")  
print(result) # Output: LANGCHAIN  
print(runnable.history) # Output: [{'input': 'langchain', 'output': 'LANGCHAIN'}]

**15.RunnableAssign**

Allows assigning static values or additional fields to the input or output.

**Example:**

from langchain.schema.runnable import RunnableAssign  
  
assign = RunnableAssign(assignments={"greeting": "Hello"})  
result = assign.invoke({})  
print(result) # Output: {'greeting': 'Hello'}

**16. RunnablePassthrough**

A simple Runnable that passes its input unchanged to the output. Useful for testing or as a placeholder.

**Example:**

from langchain.schema.runnable import RunnablePassthrough  
  
passthrough = RunnablePassthrough()  
result = passthrough.invoke("input\_value")  
print(result) # Output: input\_value

Runnables in LangChain provide a powerful and flexible framework to build modular, composable, and reusable pipelines, significantly enhancing productivity and maintainability in AI and data workflows.