



LangChain Expression Language (LCEL)

❗ PREREQUISITES

- [Runnable Interface](#)

The LangChain Expression Language (LCEL) takes a **declarative** approach to building new **Runnables** from existing Runnables.

This means that you describe what *should* happen, rather than *how* it should happen, allowing LangChain to optimize the run-time execution of the chains.

We often refer to a `Runnable` created using LCEL as a "chain". It's important to remember that a "chain" is `Runnable` and it implements the full **Runnable Interface**.

❗ NOTE

- The [LCEL cheatsheet](#) shows common patterns that involve the Runnable interface and LCEL expressions.
- Please see the following list of [how-to guides](#) that cover common tasks with LCEL.
- A list of built-in `Runnables` can be found in the [LangChain Core API Reference](#). Many of these Runnables are useful when composing custom "chains" in LangChain using LCEL.

Benefits of LCEL

LangChain optimizes the run-time execution of chains built with LCEL in a number of ways:

- **Optimized parallel execution:** Run Runnable in parallel using [RunnableParallel](#) or run multiple inputs through a given chain in parallel using the [Runnable Batch API](#). Parallel execution can significantly reduce the latency as processing can be done in parallel instead of sequentially.
- **Guaranteed Async support:** Any chain built with LCEL can be run asynchronously using the [Runnable Async API](#). This can be useful when running chains in a server environment where you want to handle large number of requests concurrently.
- **Simplify streaming:** LCEL chains can be streamed, allowing for incremental output as the chain is executed. LangChain can optimize the streaming of the output to minimize the time-to-first-token (time elapsed until the first chunk of output from a [chat model](#) or [llm](#) comes out).

Other benefits include:

- **Seamless LangSmith tracing** As your chains get more and more complex, it becomes increasingly important to understand what exactly is happening at every step. With LCEL, all steps are automatically logged to [LangSmith](#) for maximum observability and debuggability.
- **Standard API:** Because all chains are built using the Runnable interface, they can be used in the same way as any other Runnable.
- **Deployable with LangServe:** Chains built with LCEL can be deployed using for production use.

Should I use LCEL?

LCEL is an [orchestration solution](#) -- it allows LangChain to handle run-time execution of chains in an optimized way.

While we have seen users run chains with hundreds of steps in production, we generally recommend using LCEL for simpler orchestration tasks. When the application requires complex state management, branching, cycles or multiple agents, we recommend that users take advantage of [LangGraph](#).

In LangGraph, users define graphs that specify the application's flow. This allows users to keep using LCEL within individual nodes when LCEL is needed, while making it easy to define complex orchestration logic that is more readable and maintainable.

Here are some guidelines:

- If you are making a single LLM call, you don't need LCEL; instead call the underlying `chat model` directly.
- If you have a simple chain (e.g., prompt + llm + parser, simple retrieval set up etc.), LCEL is a reasonable fit, if you're taking advantage of the LCEL benefits.
- If you're building a complex chain (e.g., with branching, cycles, multiple agents, etc.) use `LangGraph` instead. Remember that you can always use LCEL within individual nodes in LangGraph.

Composition Primitives

LCEL chains are built by composing existing `Runnables` together. The two main composition primitives are `RunnableSequence` and `RunnableParallel`.

Many other composition primitives (e.g., `RunnableAssign`) can be thought of as variations of these two primitives.

NOTE

You can find a list of all composition primitives in the [LangChain Core API Reference](#).

RunnableSequence

`RunnableSequence` is a composition primitive that allows you "chain" multiple runnables sequentially, with the output of one runnable serving as the input to the next.

```
from langchain_core.runnables import RunnableSequence
chain = RunnableSequence([runnable1, runnable2])
```

API Reference: RunnableSequence

Invoking the `chain` with some input:

```
final_output = chain.invoke(some_input)
```

corresponds to the following:

```
output1 = runnable1.invoke(some_input)
final_output = runnable2.invoke(output1)
```

NOTE

`runnable1` and `runnable2` are placeholders for any `Runnable` that you want to chain together.

RunnableParallel

`RunnableParallel` is a composition primitive that allows you to run multiple runnables concurrently, with the same input provided to each.

```
from langchain_core.runnables import RunnableParallel
chain = RunnableParallel({
    "key1": runnable1,
    "key2": runnable2,
})
```

API Reference: RunnableParallel

Invoking the `chain` with some input:

```
final_output = chain.invoke(some_input)
```

Will yield a `final_output` dictionary with the same keys as the input dictionary, but with the values replaced by the output of the corresponding runnable.

```
{  
    "key1": runnable1.invoke(some_input),  
    "key2": runnable2.invoke(some_input),  
}
```

Recall, that the runnables are executed in parallel, so while the result is the same as dictionary comprehension shown above, the execution time is much faster.

NOTE

`RunnableParallel` supports both synchronous and asynchronous execution (as all `Runnables` do).

- For synchronous execution, `RunnableParallel` uses a `ThreadPoolExecutor` to run the runnables concurrently.
- For asynchronous execution, `RunnableParallel` uses `asyncio.gather` to run the runnables concurrently.

Composition Syntax

The usage of `RunnableSequence` and `RunnableParallel` is so common that we created a shorthand syntax for using them. This helps to make the code more readable and concise.

The `|` operator

We have **overloaded** the `|` operator to create a `RunnableSequence` from two `Runnables`.

```
chain = runnable1 | runnable2
```

is Equivalent to:

```
chain = RunnableSequence([runnable1, runnable2])
```

The `.pipe` method

If you have moral qualms with operator overloading, you can use the `.pipe` method instead. This is equivalent to the `|` operator.

```
chain = runnable1.pipe(runnable2)
```

Coercion

LCEL applies automatic type coercion to make it easier to compose chains.

If you do not understand the type coercion, you can always use the `RunnableSequence` and `RunnableParallel` classes directly.

This will make the code more verbose, but it will also make it more explicit.

Dictionary to RunnableParallel

Inside an LCEL expression, a dictionary is automatically converted to a `RunnableParallel`.

For example, the following code:

```
mapping = {  
    "key1": runnable1,  
    "key2": runnable2,  
}  
  
chain = mapping | runnable3
```

It gets automatically converted to the following:

```
chain = RunnableSequence([RunnableParallel(mapping), runnable3])
```

⚠ CAUTION

You have to be careful because the `mapping` dictionary is not a `RunnableParallel` object, it is just a dictionary. This means that the following code will raise an `AttributeError`:

```
mapping.invoke(some_input)
```

Function to RunnableLambda

Inside an LCEL expression, a function is automatically converted to a `RunnableLambda`.

```
def some_func(x):  
    return x  
  
chain = some_func | runnable1
```

It gets automatically converted to the following:

```
chain = RunnableSequence([RunnableLambda(some_func), runnable1])
```

⚠ CAUTION

You have to be careful because the lambda function is not a `RunnableLambda` object, it is just a function. This means that the following code will raise an `AttributeError`:

```
lambda x: x + 1.invoke(some_input)
```

Legacy chains

LCEL aims to provide consistency around behavior and customization over legacy subclassed chains such as `LLMChain` and `ConversationalRetrievalChain`. Many of these legacy chains hide important details like prompts, and as a wider variety of viable models emerge, customization has become more and more important.

If you are currently using one of these legacy chains, please see [this guide for guidance on how to migrate](#).

For guides on how to do specific tasks with LCEL, check out [the relevant how-to guides](#).

[✎ Edit this page](#)

Was this page helpful?





An error occurred: API rate limit exceeded for langchain-ai/langchain. Sign in to increase the

Write

Preview

Aa

Sign in to comment

Sign in with GitHub