Why use LCEL

We recommend reading the LCEL Get started section first.

LCEL makes it easy to build complex chains from basic components. It does this by providing: 1. **A unified interface**: Every LCEL object implements the `Runnable` interface, which defines a common set of invocation methods (`invoke`, `batch`, `stream`, `ainvoke`, ...). This makes it possible for chains of LCEL objects to also automatically support these invocations. That is, every chain of LCEL objects is itself an LCEL object. 2. **Composition primitives**: LCEL provides a number of primitives that make it easy to compose chains, parallelize components, add fallbacks, dynamically configure chain internal, and more.

To better understand the value of LCEL, it's helpful to see it in action and think about how we might recreate similar functionality without it. In this walkthrough we'll do just that with our basic example from the get started section. We'll take our simple prompt + model chain, which under the hood already defines a lot of functionality, and see what it would take to recreate all of it.

%pip install –upgrade –quiet langchain-core langchain-openai

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import
ChatPromptTemplate
from langchain_core.output_parsers import
StrOutputParser


prompt =
ChatPromptTemplate.from_template("Tell me a
short joke about {topic}")
model = ChatOpenAI(model="gpt-3.5-turbo")
output_parser = StrOutputParser()

chain = prompt | model | output_parser
```

Invoke

In the simplest case, we just want to pass in a topic string and get back a joke string:

Without LCEL

```
from typing import List

import openai


```

LCEL

```
from langchain_core.runnable
import RunnablePassthrough


prompt =
```

```
prompt_template = "Tell me a
joke about {topic}"
client = openai.OpenAI()

def call_chat_model(messages
List[dict]) -> str:
    response =
client.chat.completions.crea
        model="gpt-3.5-turbo
        messages=messages,
    )
    return
response.choices[0].message

def invoke_chain(topic: str)
str:
    prompt_value =
prompt_template.format(topic
    messages = [{"role": "us
"content": prompt_value}]
    return
call_chat_model(messages)

invoke_chain("ice cream")
```

```
ChatPromptTemplate.from_temp
    "Tell me a short joke ab
{topic}"
)
output_parser = StrOutputPar
model = ChatOpenAI(model="gp
3.5-turbo")
chain = (
    {"topic":
RunnablePassthrough()}
    | prompt
    | model
    | output_parser
)

chain.invoke("ice cream")
```

Stream

If we want to stream results instead, we'll need to change our function:

Without LCEL **LCEL**

```
from typing import Iterator


def stream_chat_model(messag
List[dict]) -> Iterator[str]
    stream =
client.chat.completions.crea
        model="gpt-3.5-turbo
        messages=messages,
        stream=True,
    )
    for response in stream:
        content =
response.choices[0].delta.co
        if content is not No
            yield content

def stream_chain(topic: str)
Iterator[str]:
    prompt_value =
prompt.format(topic=topic)
    return
stream_chat_model([{"role":
"user", "content":
prompt_value}])


for chunk in stream_chain("i
cream"):
    print(chunk, end="",
flush=True)
```

```
for chunk in
chain.stream("ice
cream"):
    print(chunk, end="",
flush=True)
```

Batch

If we want to run on a batch of inputs in parallel, we'll again need a new function:

Without LCEL

```
from concurrent.futures impo
ThreadPoolExecutor


def batch_chain(topics: list
list:
    with
ThreadPoolExecutor(max_worke
as executor:
        return
list(executor.map(invoke_cha
topics))

batch_chain(["ice cream",
"spaghetti", "dumplings"])
```

LCEL

```
chain.batch(["ice cream",
"spaghetti",
"dumplings"])
```

Async

If we need an asynchronous version:

Without LCEL

```
async_client = openai.AsyncC

async def acall_chat_model(m
List[dict]) -> str:
```

LCEL

```
chain.ainvoke("ice
cream")
```

```
    response = await
async_client.chat.completion
        model="gpt-3.5-turbo
        messages=messages,
    )
    return
response.choices[0].message

async def ainvoke_chain(topi
-> str:
    prompt_value =
prompt_template.format(topic
    messages = [{"role": "us
"content": prompt_value}]
    return await
acall_chat_model(messages)
```

```
await ainvoke_chain("ice
cream")
```

LLM instead of chat model

If we want to use a completion endpoint instead of a chat endpoint:

Without LCEL

```
def call_llm(prompt_value: s
str:
    response =
client.completions.create(
```

LCEL

```
from langchain_openai
import OpenAI

llm = OpenAI(model="gpt-
```

```
        model="gpt-3.5-turbo
instruct",
        prompt=prompt_value,
    )
    return
response.choices[0].text

def invoke_llm_chain(topic:
> str:
    prompt_value =
prompt_template.format(topic
    return call_llm(prompt_v

invoke_llm_chain("ice cream"
```

```
3.5-turbo-instruct")
llm_chain = (
    {"topic":
RunnablePassthrough()}
    | prompt
    | llm
    | output_parser
)

llm_chain.invoke("ice
cream")
```

Different model provider

If we want to use Anthropic instead of OpenAI:

Without LCEL

```
import anthropic

anthropic_template =
f"Human:\n\n{prompt_template
anthropic_client = anthropic

def call_anthropic(prompt_va
str:
    response =
anthropic_client.completions
        model="claude-2",
        prompt=prompt_value,
```

LCEL

```
from
langchain_community.chat_mod
import ChatAnthropic

anthropic =
ChatAnthropic(model="claude-
anthropic_chain = (
    {"topic":
RunnablePassthrough()}
    | prompt
    | anthropic
    | output_parser
```

```
        max_tokens_to_sample
    )
    return response.completi

def invoke_anthropic_chain(t
str:
    prompt_value =
anthropic_template.format(to
    return call_anthropic(pr

invoke_anthropic_chain("ice
```

```
)

anthropic_chain.invoke("ice
cream")
```

Runtime configurability

If we wanted to make the choice of chat model or LLM configurable at runtime:

Without LCEL

```
def invoke_configurable_chai
    topic: str,
    *,
    model: str = "chat_opena
) -> str:
    if model == "chat_openai
        return
invoke_chain(topic)
    elif model == "openai":
        return
invoke_llm_chain(topic)
    elif model == "anthropic
        return
invoke_anthropic_chain(topic
```

With LCEL

```
from langchain_core.runnable
import ConfigurableField


configurable_model =
model.configurable_alternati

ConfigurableField(id="model"
    default_key="chat_openai
    openai=llm,
    anthropic=anthropic,
)
configurable_chain = (
    {"topic":
```

```
    else:
        raise ValueError(
            f"Received inva
model '{model}'."
            " Expected one 
chat_openai, openai, anthro
        )

def stream_configurable_cha
    topic: str,
    *,
    model: str = "chat_opena
) -> Iterator[str]:
    if model == "chat_opena
        return
stream_chain(topic)
    elif model == "openai":
        # Note we haven't
implemented this yet.
        return
stream_llm_chain(topic)
    elif model == "anthropi
        # Note we haven't
implemented this yet
        return
stream_anthropic_chain(topi
    else:
        raise ValueError(
            f"Received inva
model '{model}'."
            " Expected one 
chat_openai, openai, anthro
        )

def batch_configurable_chai
    topics: List[str],
    *,
    model: str = "chat_opena
) -> List[str]:
```

```
RunnablePassthrough()}
    | prompt
    | configurable_model
    | output_parser
)
```

```
configurable_chain.invoke(
    "ice cream",
    config={"model": "opena
)
stream =
configurable_chain.stream(
    "ice cream",
    config={"model":
"anthropic"}
)
for chunk in stream:
    print(chunk, end="",
flush=True)

configurable_chain.batch(["
cream", "spaghetti",
"dumplings"])

# await
configurable_chain.ainvoke('
cream")
```

```
    # You get the idea
    ...

async def
abatch_configurable_chain(
    topics: List[str],
    *,
    model: str = "chat_opena
) -> List[str]:
    ...

invoke_configurable_chain("i
cream", model="openai")
stream =
stream_configurable_chain(
    "ice_cream",
    model="anthropic"
)
for chunk in stream:
    print(chunk, end="",
flush=True)

#
batch_configurable_chain(["i
cream", "spaghetti",
"dumplings"])
# await
ainvoke_configurable_chain('
cream")
```

Logging

If we want to log our intermediate results:

Without LCEL

We'll `print` intermediate steps for illustrative purposes

```
def
invoke_anthropic_chain_with_
str) -> str:
    print(f"Input: {topic}")
    prompt_value =
anthropic_template.format(to
    print(f"Formatted prompt
{prompt_value}")
    output = call_anthropic(
    print(f"Output: {output}
    return output

invoke_anthropic_chain_with_
cream")
```

LCEL

Every component has built-in integrations with LangSmith. If we set the following two environment variables, all chain traces are logged to LangSmith.

```
import os

os.environ["LANGCHAIN_API_KE
"..."
os.environ["LANGCHAIN_TRACIN
= "true"

anthropic_chain.invoke("ice
cream")
```

Here's what our LangSmith trace looks like: https://smith.langchain.com/public/e4de52f8-bcd9-4732-b950-deee4b04e313/r

Fallbacks

If we wanted to add fallback logic, in case one model API is down:

Without LCEL

LCEL

```
def
invoke_chain_with_fallback(t
str) -> str:
    try:
        return invoke_chain(
    except Exception:
        return
invoke_anthropic_chain(topic

async def
ainvoke_chain_with_fallback(
str) -> str:
    try:
        return await
ainvoke_chain(topic)
    except Exception:
        # Note: we haven't
actually implemented this.
        return
ainvoke_anthropic_chain(topi

async def
batch_chain_with_fallback(to
List[str]) -> str:
    try:
        return batch_chain(t
    except Exception:
        # Note: we haven't
actually implemented this.
        return
batch_anthropic_chain(topics

invoke_chain_with_fallback("
cream")
# await
ainvoke_chain_with_fallback(
cream")
batch_chain_with_fallback(["
```

```
fallback_chain =
chain.with_fallbacks([anthro

fallback_chain.invoke("ice c
# await fallback_chain.ainvo
cream")
fallback_chain.batch(["ice c
"spaghetti", "dumplings"])
```

```
cream", "spaghetti",
"dumplings"]))
```

Full code comparison

Even in this simple case, our LCEL chain succinctly packs in a lot of functionality. As chains become more complex, this becomes especially valuable.

Without LCEL

```
from concurrent.futures impo
ThreadPoolExecutor
from typing import Iterator,

import anthropic
import openai


prompt_template = "Tell me a
about {topic}"
anthropic_template =
f"Human:\n\n{prompt_templat
client = openai.OpenAI()
async_client = openai.AsyncO
anthropic_client = anthropic

def call_chat_model(messages
> str:
    response =
client.chat.completions.crea
        model="gpt-3.5-turbo
```

LCEL

```
import os

from
langchain_community.chat_mod
import ChatAnthropic
from langchain_openai import
ChatOpenAI
from langchain_openai import
OpenAI
from langchain_core.output_p
import StrOutputParser
from langchain_core.prompts
ChatPromptTemplate
from langchain_core.runnable
import RunnablePassthrough,
ConfigurableField

os.environ["LANGCHAIN_API_KE
"..."
os.environ["LANGCHAIN_TRACIN
= "true"
```

```
        messages=messages,
    )
    return
response.choices[0].message

def invoke_chain(topic: str)
    print(f"Input: {topic}")
    prompt_value =
prompt_template.format(topic
    print(f"Formatted prompt
{prompt_value}")
    messages = [{"role": "us
prompt_value}]
    output = call_chat_model
    print(f"Output: {output}
    return output

def stream_chat_model(messag
-> Iterator[str]:
    stream =
client.chat.completions.crea
        model="gpt-3.5-turbo
        messages=messages,
        stream=True,
    )
    for response in stream:
        content =
response.choices[0].delta.co
        if content is not No
            yield content

def stream_chain(topic: str)
Iterator[str]:
    print(f"Input: {topic}")
    prompt_value =
prompt.format(topic=topic)
    print(f"Formatted prompt
{prompt_value}")
    stream = stream_chat_mod
```

```
prompt =
ChatPromptTemplate.from_temp
    "Tell me a short joke ab
{topic}"
)
chat_openai =
ChatOpenAI(model="gpt-3.5-tu
openai = OpenAI(model="gpt-3
turbo-instruct")
anthropic =
ChatAnthropic(model="claude-
model = (
    chat_openai
    .with_fallbacks([anthrop
    .configurable_alternativ

ConfigurableField(id="model"
        default_key="chat_op
        openai=openai,
        anthropic=anthropic,
    )
)

chain = (
    {"topic":
RunnablePassthrough()}
    | prompt
    | model
    | StrOutputParser()
)
```

```
"user", "content": prompt_va
    for chunk in stream:
        print(f"Token: {chur
        yield chunk

def batch_chain(topics: list
    with ThreadPoolExecutor(
as executor:
        return
list(executor.map(invoke_cha

def call_llm(prompt_value: s
    response = client.comple
        model="gpt-3.5-turbo
        prompt=prompt_value,
    )
    return response.choices[

def invoke_llm_chain(topic:
    print(f"Input: {topic}")
    prompt_value =
promtp_template.format(topic
    print(f"Formatted prompt
{prompt_value}")
    output = call_llm(prompt
    print(f"Output: {output}
    return output

def call_anthropic(prompt_va
str:
    response =
anthropic_client.completions
        model="claude-2",
        prompt=prompt_value,
        max_tokens_to_sample
    )
    return response.complet

def invoke_anthropic_chain(t
```

```
str:
    print(f"Input: {topic}")
    prompt_value =
anthropic_template.format(to
    print(f"Formatted prompt
{prompt_value}")
    output = call_anthropic(
    print(f"Output: {output}
    return output

async def ainvoke_anthropic_
str) -> str:
    ...

def stream_anthropic_chain(t
Iterator[str]:
    ...

def batch_anthropic_chain(to
List[str]) -> List[str]:
    ...

def invoke_configurable_chai
    topic: str,
    *,
    model: str = "chat_opena
) -> str:
    if model == "chat_openai
        return invoke_chain(
    elif model == "openai":
        return invoke_llm_ch
    elif model == "anthropic
        return invoke_anthro
    else:
        raise ValueError(
            f"Received inval
'{model}'."
            " Expected one o
openai, anthropic"
```

```
        )

def stream_configurable_cha
    topic: str,
    *,
    model: str = "chat_opena
) -> Iterator[str]:
    if model == "chat_opena
        return stream_chain(
    elif model == "openai":
        # Note we haven't im
yet.
        return stream_llm_ch
    elif model == "anthropic
        # Note we haven't im
yet
        return stream_anthro
    else:
        raise ValueError(
            f"Received inval
'{model}'."
            " Expected one o
openai, anthropic"
        )

def batch_configurable_chain
    topics: List[str],
    *,
    model: str = "chat_opena
) -> List[str]:
    ...

async def abatch_configurabl
    topics: List[str],
    *,
    model: str = "chat_opena
) -> List[str]:
    ...
```

```
def invoke_chain_with_fallba
-> str:
    try:
        return invoke_chain(
    except Exception:
        return invoke_anthro

async def
ainvoke_chain_with_fallback(
str:
    try:
        return await ainvoke
    except Exception:
        return
ainvoke_anthropic_chain(topi

async def batch_chain_with_f
List[str]) -> str:
    try:
        return batch_chain(t
    except Exception:
        return batch_anthrop
```

Next steps

To continue learning about LCEL, we recommend: - Reading up on the full LCEL Interface, which we've only partially covered here. - Exploring the How-to section to learn about additional composition primitives that LCEL provides. - Looking through the Cookbook section to see LCEL in action for common use

cases. A good next use case to look at would be Retrieval-augmented generation.