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.

```
from langchain.chat_models import ChatOpenAI
from langchain.prompts import ChatPromptTemplate
from langchain.schema.output_parser 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


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

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

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

invoke_chain("ice cream")
```

LCEL

```
from langchain_core.runnables import RunnablePassthrough


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

```
from typing import Iterator


def stream_chat_model(messages:
List[dict]) -> Iterator[str]:
    stream =
client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=messages,
        stream=True,
    )
    for response in stream:
        content =
response.choices[0].delta.content
        if content is not None:
            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("ice
cream"):
```

LCEL

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

```
    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 import
ThreadPoolExecutor


def batch_chain(topics: list) ->
list:
    with
ThreadPoolExecutor(max_workers=5)
as executor:
        return
list(executor.map(invoke_chain,
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.AsyncOpenAI()

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

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

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

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

LCEL

```
chain.ainvoke("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: str) -> str:
    response = client.completions.create(
        model="gpt-3.5-turbo-instruct",
        prompt=prompt_value,
    )
    return response.choices[0].text

def invoke_llm_chain(topic: str) -> str:
```

LCEL

```
from langchain.llms import OpenAI

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

```
    prompt_value =
prompt_template.format(topic=topic)
    return call_llm(prompt_value)

invoke_llm_chain("ice cream")
```

```
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}\n\nAssistant:"
anthropic_client = anthropic.Anthropic()

def call_anthropic(prompt_value: str) -> str:
    response = anthropic_client.completions.create(
        model="claude-2",
        prompt=prompt_value,
        max_tokens_to_sample=256,
    )
    return response.completion

def invoke_anthropic_chain(topic: str) -> str:
    prompt_value = anthropic_template.format(topic=topic)
    return call_anthropic(prompt_value)

invoke_anthropic_chain("ice cream")
```

LCEL

```
from langchain.chat_models import ChatAnthropic

anthropic = ChatAnthropic(model="claude-2")
anthropic_chain = (
```

```
    {"topic": RunnablePassthrough()}
    | prompt
    | anthropic
    | output_parser
)

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_chain(
    topic: str,
    *,
    model: str = "chat_openai"
) -> 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)
    else:
        raise ValueError(
            f"Received invalid
model '{model}'."
            " Expected one of
chat_openai, openai, anthropic"
        )

def stream_configurable_chain(
    topic: str,
    *,
```

With LCEL

```
from langchain_core.runnables
import ConfigurableField


configurable_model =
model.configurable_alternatives(

ConfigurableField(id="model"),
    default_key="chat_openai",
    openai=llm,
    anthropic=anthropic,
)
configurable_chain = (
    {"topic":
RunnablePassthrough()}
    | prompt
    | configurable_model
    | output_parser
)
```

```
configurable_chain.invoke(
    "ice cream",
    config={"model": "openai"}
```

```
    model: str = "chat_openai"
) -> Iterator[str]:
    if model == "chat_openai":
        return
stream_chain(topic)
    elif model == "openai":
        # Note we haven't
implemented this yet.
        return
stream_llm_chain(topic)
    elif model == "anthropic":
        # Note we haven't
implemented this yet
        return
stream_anthropic_chain(topic)
    else:
        raise ValueError(
            f"Received invalid
model '{model}'."
            " Expected one of
chat_openai, openai, anthropic"
        )

def batch_configurable_chain(
    topics: List[str],
    *,
    model: str = "chat_openai"
) -> List[str]:
    # You get the idea
    ...

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

invoke_configurable_chain("ice
cream", model="openai")
stream =
stream_configurable_chain(
    "ice cream",
    config={"model":
"openai"}
)
for chunk in stream:
    print(chunk, end="",
flush=True)

stream =
stream_configurable_chain(
    "ice cream",
    config={"model":
"anthropic"}
)
for chunk in stream:
    print(chunk, end="",
flush=True)

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

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

```
    "ice_cream",
    model="anthropic"
)
for chunk in stream:
    print(chunk, end="",
flush=True)

#
batch_configurable_chain(["ice
cream", "spaghetti",
"dumplings"])
# await
ainvoke_configurable_chain("ice
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_logging(topic: str) -> str:
    print(f"Input: {topic}")
    prompt_value = anthropic_template.format(topic=topic)
    print(f"Formatted prompt: {prompt_value}")
    output = call_anthropic(prompt_value)
    print(f"Output: {output}")
    return output

invoke_anthropic_chain_with_logging("ice 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_KEY"] = "..."
os.environ["LANGCHAIN_TRACING_V2"] = "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

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

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

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

invoke_chain_with_fallback("ice cream")
```

```
# await ainvoke_chain_with_fallback("ice cream")
batch_chain_with_fallback(["ice cream", "spaghetti", "dumplings"]))
```

LCEL

```
fallback_chain = chain.with_fallbacks([anthropic_chain])

fallback_chain.invoke("ice cream")
# await fallback_chain.ainvoke("ice cream")
fallback_chain.batch(["ice 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 import ThreadPoolExecutor
from typing import Iterator, List, Tuple

import anthropic
import openai


prompt_template = "Tell me a short joke about {topic}"
anthropic_template = f"Human:\n\n{prompt_template}\n\nAssistant:"
client = openai.OpenAI()
async_client = openai.AsyncOpenAI()
anthropic_client = anthropic.Anthropic()

def call_chat_model(messages: List[dict]) -> str:
    response = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=messages,
    )
```

```
    return response.choices[0].message.content

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

def stream_chat_model(messages: List[dict]) -> Iterator[str]:
    stream = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=messages,
        stream=True,
    )
    for response in stream:
        content = response.choices[0].delta.content
        if content is not None:
            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_model([{"role": "user", "content":
prompt_value}])
    for chunk in stream:
        print(f"Token: {chunk}", end="")
        yield chunk

def batch_chain(topics: list) -> list:
    with ThreadPoolExecutor(max_workers=5) as executor:
        return list(executor.map(invoke_chain, topics))

def call_llm(prompt_value: str) -> str:
    response = client.completions.create(
        model="gpt-3.5-turbo-instruct",
        prompt=prompt_value,
    )
    return response.choices[0].text
```

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

def call_anthropic(prompt_value: str) -> str:
    response = anthropic_client.completions.create(
        model="claude-2",
        prompt=prompt_value,
        max_tokens_to_sample=256,
    )
    return response.completion

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

async def ainvoke_anthropic_chain(topic: str) -> str:
    ...

def stream_anthropic_chain(topic: str) -> Iterator[str]:
    ...

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

def invoke_configurable_chain(
    topic: str,
    *,
    model: str = "chat_openai"
) -> 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)
    else:
        raise ValueError(
            f"Received invalid model '{model}'."
            " Expected one of chat_openai, openai, anthropic"
        )

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

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

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

def invoke_chain_with_fallback(topic: str) -> str:
    try:
        return invoke_chain(topic)
```

```
    except Exception:
        return invoke_anthropic_chain(topic)

async def ainvoke_chain_with_fallback(topic: str) -> str:
    try:
        return await ainvoke_chain(topic)
    except Exception:
        return ainvoke_anthropic_chain(topic)

async def batch_chain_with_fallback(topics: List[str]) -> str:
    try:
        return batch_chain(topics)
    except Exception:
        return batch_anthropic_chain(topics)
```

```
import os

from langchain.chat_models import ChatAnthropic, ChatOpenAI
from langchain.llms import OpenAI
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough

os.environ["LANGCHAIN_API_KEY"] = "..."
os.environ["LANGCHAIN_TRACING_V2"] = "true"

prompt = ChatPromptTemplate.from_template(
    "Tell me a short joke about {topic}"
)
chat_openai = ChatOpenAI(model="gpt-3.5-turbo")
openai = OpenAI(model="gpt-3.5-turbo-instruct")
anthropic = ChatAnthropic(model="claude-2")
model = (
    chat_openai
    .with_fallbacks([anthropic])
    .configurable_alternatives(
        ConfigurableField(id="model"),
        default_key="chat_openai",
        openai=openai,
        anthropic=anthropic,
    )
)
```

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

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.