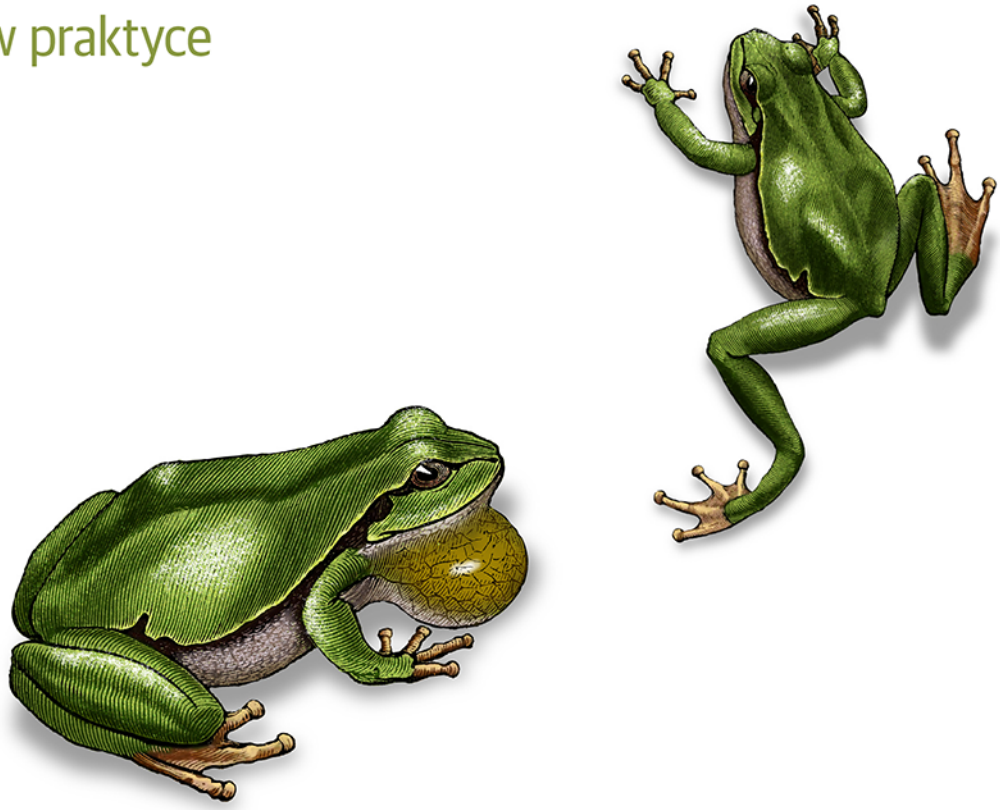
O'REILLY®

LangChain i LangGraph

Projektowanie aplikacji opartych na dużych modelach językowych w praktyce



Mayo Oshin
Nuno Campos

Helion

Tytuł oryginału: Learning LangChain: Building AI and LLM Applications
with LangChain and LangGraph

Tłumaczenie: Piotr Rajca

ISBN: 978-83-289-3072-8

Authorized Polish translation of the English edition of *Learning LangChain*
ISBN 978109816728© 2025 Olumayowa "Mayo" Olufemi Oshin

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/langch
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

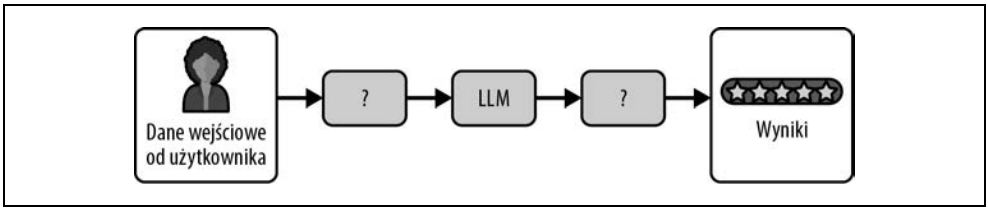
Printed in Poland.

Spis treści

ROZDZIAŁ 1.

Podstawy modeli językowych z wykorzystaniem LangChaina

Wstęp dał Ci przedsmak możliwości, jakie zapewnia odpowiednie formułowanie promptów dla modeli LLM. Czytając go, mogłeś się przekonać, jak różne techniki tworzenia promptów wpływają na wyniki generowane przez modele LLM, szczególnie gdy techniki te zostaną umiejętnie połączone. Wyzwaniem w tworzeniu dobrych aplikacji korzystających z modeli LLM jest właśnie skuteczne konstruowanie promptów przesyłanych do modelu oraz przetwarzanie jego przewidywań, aby uzyskać trafne wyniki (patrz rysunek 1.1).



Rysunek 1.1. Wyzwanie związane z wykorzystaniem modeli językowych w aplikacjach

Jeśli potrafisz rozwiązać ten problem, jesteś na dobrej drodze do tworzenia aplikacji korzystających z modeli LLM — zarówno prostych, jak i złożonych. W tym rozdziale dowiesz się więcej o tym, jak elementy składowe biblioteki LangChain odpowiadają koncepcjom modeli LLM i jak, odpowiednio połączone, umożliwiają tworzenie aplikacji korzystających z tych modeli. Najpierw jednak w ramce „Dlaczego LangChain?" znajdziesz krótkie wyjaśnienie, dlaczego uważamy, że do budowania aplikacji korzystających z modeli LLM warto używać właśnie tej biblioteki.

Dlaczego LangChain?

Oczywiście można tworzyć aplikacje korzystające z modeli LLM bez stosowania biblioteki LangChain. Najbardziej oczywistą alternatywą jest stosowanie zestawu narzędzi programistycznych (SDK) wybranego dostawcy modeli LLM (na przykład firmy OpenAI), który udostępnia metody interfejsu HTTP API jako funkcje w wybranym języku programowania.

Uważamy jednak, że nauka LangChaina z kilku powodów opłaci się zarówno w krótszej, jak i dłuższej perspektywie czasowej. A oto dlaczego tak sądzimy:

Gotowe wzorce

LangChain udostępnia referencyjne implementacje najczęściej stosowanych wzorców aplikacji korzystających z modeli LLM (niektóre z nich wymieniliśmy we wstępie: rozumowanie krok po kroku, wywoływanie narzędzi itd.). Jest to najszybszy sposób na rozpoczęcie pracy z modelami LLM i często może być wszystkim, czego będziesz potrzebować. Sugerujemy rozpoczynanie każdej nowej aplikacji od próby zastosowania tych trzech wzorców i sprawdzenia, czy wyniki są wystarczające dla Twojego przypadku użycia. Jeśli okaże się, że wyniki nie są satysfakcjonujące, będziesz musiał przejść do następnego punktu, aby poznać drugą część bibliotek LangChain.

Wymienne komponenty

Są to elementy, które można łatwo zastąpić alternatywnymi rozwiązaniami. Każdy komponent (model LLM, model konwersacyjny, parser danych wyjściowych i tak dalej — więcej na ten temat dowiesz się już niebawem) jest zgodny z pewną wspólną specyfikacją, dzięki czemu tworzona aplikacja jest odporna na przyszłe zmiany. W miarę jak dostawcy modeli udostępniają nowe funkcje i zmieniają się Twoje potrzeby, możesz rozwijać swoją aplikację bez konieczności jej ciągłego przepisywania.

W przykładach programistycznych prezentowanych w tej książce będziemy korzystać z następujących głównych komponentów:

- model LLM/model konwersacyjny: OpenAI,
- osadzenia: OpenAI,
- wektorowa baza danych: PGVector.

Każdy z tych elementów można zastąpić dowolnym z alternatywnych rozwiązań wymienionych na kolejnych stronach:

Modele konwersacyjne

Zobacz dokumentację biblioteki LangChain (*https://python.langchain.com/docs/integrations/chat/*). Jeśli nie chcesz korzystać z modelu firmy OpenAI (komercyjne API), sugerujemy użycie modeli firmy Anthropic (*https://python.langchain.com/docs/integrations/chat/anthropic/*) będących alternatywą komercyjną lub z Ollama (*https://python.langchain.com/docs/integrations/chat/ollama/*) jako rozwiązania otwartoźródłowego.

Osadzenia

Zobacz dokumentację LangChaina (*https://python.langchain.com/docs/integrations/text_embedding/*). Jeśli nie chcesz korzystać z rozwiązań firmy OpenAI (komercyjnego API), sugerujemy skorzystanie z Cohere (*https://python.langchain.com/docs/integrations/text_embedding/cohere/*) będącego alternatywą komercyjną lub Ollama (*https://python.langchain.com/docs/integrations/chat/ollama/*) jako rozwiązania otwartoźródłowego.

Wektorowe bazy danych

Zobacz dokumentację LangChaina (*https://python.langchain.com/docs/integrations/vectorstores/*). Jeśli nie chcesz korzystać z PGVector (otwartoźródłowego rozszerzenia popularnej bazy danych SQL Postgres), sugerujemy użycie Weaviate (*https://python.langchain.com/docs/integrations/vectorstores/weaviate/*) (wyspecjalizowanej bazy wektorowej) lub OpenSearch (*https://python.langchain.com/docs/integrations/vectorstores/opensearch/*) (są to funkcje wyszukiwania wektorowego będące częścią popularnej bazy danych wyszukiwania).

Jednak ułatwienia, jakie zapewnia biblioteka LangChain, znacznie wykraczają poza to, że wszystkie modele LLM mają te same metody, z podobnymi argumentami i zwracają podobne wyniki. Przyjrzyjmy się przykładowi modeli konwersacyjnych i dwóm popularnym dostawcom LLM, firmom: OpenAI i Anthropic. Obie udostępniają konwersacyjny interfejs API, który otrzymuje *wiadomości* (zdefiniowane jako obiekty zawierające łańcuchy znaków określające typ oraz treści) i zwraca nową wiadomość wygenerowaną przez model. Jeśli jednak spróbujesz użyć obu modeli w tej samej konwersacji, natychmiast pojawią się problemy, gdyż formaty wiadomości zwracanych przez oba modele będą nieznacznie odmienne i niezgodne ze sobą. LangChain ukrywa te różnice, umożliwiając budowanie aplikacji, które są naprawdę niezależne od konkretnego dostawcy modeli LLM. Na przykład dzięki użyciu biblioteki LangChain chatbot korzystający zarówno z modelu firmy OpenAI, jak i firmy Anthropic będzie działał bez żadnych problemów.

I w końcu, kiedy zaczniesz tworzyć aplikacje LLM korzystające z kilku takich komponentów, na pewno przydadzą Ci się możliwości orkiestracji, jakie zapewnia LangChain:

- Wszystkie główne komponenty umożliwiają stosowanie systemu wywołań zwrotnych, pozwalającego na obserwowanie ich działania (więcej informacji na ten temat znajdziesz w rozdziale 8.).
- Wszystkie główne komponenty implementują ten sam interfejs (to zagadnienie zostało szerzej opisane pod koniec tego rozdziału).
- Jeśli działanie aplikacji korzystającej z modeli LLM musi trwać bardzo długo, to można je przerywać oraz wznawiać (więcej informacji na ten temat można znaleźć w rozdziale 6.).

Konfiguracja projektu LangChain

Aby móc analizować dalszą część tego rozdziału oraz kolejne, powinieneś najpierw skonfigurować LangChain na swoim komputerze.

Jeśli jeszcze tego nie zrobiłeś, zapoznaj się z zamieszczonymi we wstępie instrukcjami dotyczącymi założenia konta OpenAI. Jeśli wolisz korzystać z innego dostawcy modeli LLM, sprawdź alternatywy wymienione w ramce „Dlaczego LangChain?” na samym początku tego rozdziału.

Następnie, w serwisie OpenAI, przejdź na stronę z kluczami API (*https://platform.openai.com/api-keys*) na stronie OpenAI (po zalogowaniu się na swoje konto), utwórz klucz API i zapisz go — wkrótce będziesz go potrzebować.

W tej książce przedstawiamy przykłady pisane zarówno w języku Python, jak i JavaScript. LangChain udostępnia te same możliwości w obu językach, więc możesz wybrać ten z nich, który jest dla Ciebie wygodniejszy, i w dalszej części książki analizować odpowiednie fragmenty kodu (w całej książce prezentujemy przykłady w obu tych językach).

Oto instrukcje konfiguracji dla Czytelników korzystających z języka Python:

1. Upewnij się, że masz zainstalowanego Pythona. Instrukcje jego pobierania i instalacji dla używanego systemu operacyjnego znajdziesz na stronie *https://wiki.python.org/moin/BeginnersGuide/Download*.
2. Jeśli chcesz uruchamiać przykłady w notatnikach, to zainstaluj Jupyter Notebook. Możesz to zrobić, wykonując w oknie terminala polecenie `pip install notebook`.
3. Zainstaluj bibliotekę LangChain. W tym celu w oknie terminala wykonaj następujące polecenie:

```
pip install langchain langchain-openai langchain-community
pip install langchain-text-splitters langchain-postgres
```

4. Wygenerowany wcześniej klucz API OpenAI udostępnij w środowisku terminala; w tym celu wykonaj polecenie:

```
export OPENAI_API_KEY=Twój_klucz_API
```

5. Nie zapomnij zastąpić fragmentu *`Twój_klucz_API`* wygenerowanym wcześniej kluczem API.
6. Uruchom serwer Jupyter przy użyciu następującego polecenia:

```
jupyter notebook
```

Po wykonaniu tych czynności będziesz gotów, by wykonywać i analizować prezentowane przykłady pisane w Pythonie.

A oto instrukcje dla Czytelników korzystających z języka JavaScript:

1. Klucz API OpenAI wygenerowany na początku tego rozdziału udostępnij w środowisku terminala. W tym celu wykonaj następujące polecenie:

```
export OPENAI_API_KEY=Twój_klucz_API
```

2. Nie zapomnij zastąpić fragmentu *`Twój_klucz_API`* faktycznym, wygenerowanym wcześniej kluczem API.
3. Jeśli chcesz uruchomić przykłady jako skrypty Node.js, zainstaluj środowisko Node zgodnie z instrukcjami dostępnymi na stronie *https://nodejs.org/en/download*.
4. Zainstaluj bibliotekę LangChain poprzez wykonanie w oknie terminala następującego polecenia:

```
npm install langchain @langchain/openai @langchain/community
npm install @langchain/core pg
```

5. Zapisz każdy p rzykład jako plik z rozszerzeniem *.js* i uruchom go przy użyciu polecenia `node ./nazwa_pliku.js`.

Stosowanie modeli LLM w bibliotece LangChain

Podsumowując, modele LLM są głównym mechanizmem napędzającym większość aplikacji generatywnej sztucznej inteligencji. Biblioteka LangChain udostępnia dwa proste interfejsy do interakcji z dowolnym dostawcą API modeli LLM:

- modele konwersacyjne,
- modele LLM.

Interfejs LLM pobiera jako dane wejściowe łańcuch znaków reprezentujący prompt, przesyła go do dostawcy modelu, a następnie zwraca wynik stanowiący predykcję modelu.

W poniższym przykładzie zaimportujemy moduł OpenAI LLM biblioteki LangChain i użyjemy metody `invoke`, by pobrać odpowiedź modelu na bardzo prosty prompt:

Python

```
from langchain_openai.llms import OpenAI
model = OpenAI(model="gpt-3.5-turbo")
model.invoke("The sky is")
```

JavaScript

```
import { OpenAI } from "@langchain/openai";
const model = new OpenAI({ model: "gpt-3.5-turbo" });
await model.invoke("The sky is");
```

Wyjście:

```
Blue!
```

Zwróć uwagę na parametr `model` przekazywany do konstruktora `OpenAI`. Jest to najczęściej konfigurowany parametr stosowany podczas korzystania z modeli LLM lub modeli konwersacyjnych, określający podstawowy model LLM, który ma być używany. Większość dostawców oferuje kilka modeli cechujących się różnymi poziomami kompromisu pomiędzy możliwościami a kosztem (zazwyczaj większe modele są bardziej zaawansowane, lecz jednocześnie droższe i wolniejsze). Warto zapoznać się z informacjami na temat modeli firmy OpenAI, dostępnymi na stronie *https://platform.openai.com/docs/models*.

Inne przydatne parametry konfiguracyjne określające działanie modeli i udostępniane przez większość dostawców to:

`temperature`
: Temperatura kontroluje algorytm próbkowania używany do generowania wyników. Niższe wartości (np. `0,1`) dają bardziej przewidywalne wyniki, podczas gdy wyższe (np. `0,9`) generują bardziej kreatywne lub nieoczekiwane rezultaty. Różne zadania wymagają stosowania różnych wartości tego parametru. Na przykład generowanie ustrukturyzowanych danych wyjściowych zazwyczaj wymaga użycia niższej wartości temperatury, natomiast zadania kreatywnego pisania będą dawały lepsze wyniki w razie zastosowania wyższej wartości.

`max_tokens`
: Ogranicza rozmiar (i koszt) wygenerowanego tekstu. Niższa wartość może spowodować, że model LLM przerwie generowanie przed osiągnięciem naturalnego zakończenia, co może wyglądać jak test, w którym pominięto końcowy fragment.

Poza tymi dwoma każdy dostawca udostępnia różne zestawy parametrów. Zalecamy zapoznanie się z dokumentacją wybranego dostawcy. Na przykład parametry modeli firmy OpenAI są dostępne na stronie *https://platform.openai.com/docs/api-reference/chat/create*.

Alternatywnie, interfejs modelu konwersacyjnego umożliwia prowadzenie dialogu między użytkownikiem a modelem. Jest to oddzielny interfejs, ponieważ popularni dostawcy modeli LLM, tacy jak firma OpenAI, rozróżniają wiadomości wysyłane do modelu i przez niego zwracane na podstawie roli; gdzie rola oznacza typ zawartości wiadomości i może przyjmować wartości *użytkownik*, *asystent* i *system*:

*Rola **system***
: Używana do przekazywania instrukcji, które model powinien wykorzystać, by odpowiedzieć na pytanie użytkownika.

*Rola **użytkownik***
: Używana do przekazywania zapytania i innych treści tworzonych przez użytkownika.

*Rola **asystent***
: Używana dla treści generowanych przez model.

Interfejs modelu konwersacyjnego ułatwia konfigurację rozmów prowadzonych w aplikacjach w stylu chatbotów AI oraz zarządzanie nimi. Oto przykład wykorzystujący model `ChatOpenAI` z biblioteki LangChain:

Python

```
from langchain_openai.chat_models import ChatOpenAI
from langchain_core.messages import HumanMessage

model = ChatOpenAI()
prompt = [HumanMessage("Które miasto jest stolicą Francji?")]

model.invoke(prompt)
```

JavaScript

```
import { ChatOpenAI } from '@langchain/openai'
import { HumanMessage } from '@langchain/core/messages'

const model = new ChatOpenAI()
const prompt = [new HumanMessage('Które miasto jest stolicą Francji?')]

await model.invoke(prompt)
```

Wynik:

```
AIMessage(content='Stolicą Francji jest Paryż.')
```

W odróżnieniu od pojedynczego promptu, modele konwersacyjne wykorzystują różne typy interfejsów komunikatów, powiązane z wymienionymi wcześniej rolami. Należą do nich:

`HumanMessage`
: Wiadomość wysłana z perspektywy człowieka, z rolą *użytkownik*.

`AIMessage`
: Wiadomość wysłana z perspektywy sztucznej inteligencji, z którą człowiek wchodzi w interakcję, z rolą *asystent*.

`SystemMessage`
: Wiadomość określająca instrukcje, których model LLM powinien przestrzegać, z rolą *system*.

`ChatMessage`
: Wiadomość pozwalająca na dowolne ustawienie roli.

Przyjrzyjmy się, jak można wykorzystać instrukcję `SystemMessage` w naszym przykładzie:

Python

```
from langchain_core.messages import HumanMessage, SystemMessage
from langchain_openai.chat_models import ChatOpenAI

model = ChatOpenAI()
system_msg = SystemMessage(
    '''Jesteś pomocnym asystentem, który odpowiada na pytania, kończąc odpowiedź trzema
       znakami wykrzyknika.'''
)
human_msg = HumanMessage('Które miasto jest stolicą Francji?')

model.invoke([system_msg, human_msg])
```

JavaScript

```
import { ChatOpenAI } from "@langchain/openai";
import { HumanMessage, SystemMessage } from "@langchain/core/messages";

const model = new ChatOpenAI();
const prompt = [
  new SystemMessage(
    `Jesteś pomocnym asystentem, który odpowiada na pytania, kończąc odpowiedź trzema
     znakami wykrzyknika.`,
  ),
  new HumanMessage("Które miasto jest stolicą Francji?"),
];

await model.invoke(prompt);
```

Wynik:

```
AIMessage('Paryż!!!')
```

Jak widać, model zastosował się do instrukcji zawartej w komunikacie systemowym, mimo że nie była ona obecna w pytaniu użytkownika. Umożliwia to wstępne skonfigurowanie aplikacji AI tak, aby reagowała na dane wejściowe od użytkownika w przewidywalny sposób.

Tworzenie promptów nadających się do wielokrotnego użycia

W poprzednim podrozdziale pokazaliśmy, jak instrukcje zawarte w promptach znacząco wpływają na wyniki generowane przez model LLM. Prompty pomagają modelowi zrozumieć kontekst i tworzyć trafne odpowiedzi na zapytania.

Oto przykład szczegółowego promptu:

```
Odpowiedz na pytanie na podstawie podanego poniżej kontekstu. Jeśli na pytanie nie można
odpowiedzieć na podstawie dostarczonych danych, to odpowiedz: "Nie wiem".
Kontekst: Najnowsze osiągnięcia w dziedzinie przetwarzania języka naturalnego (NLP) są
napędzane przez duże modele językowe (LLM). Modele te przewyższają swoje mniejsze
odpowiedniki i stały się nieocenionym narzędziem dla programistów tworzących aplikacje
z funkcjami NLP. Programiści mogą korzystać z tych modeli za pośrednictwem
biblioteki `transformers` udostępnianej przez Hugging Face lub przy użyciu rozwiązań
oferowanych przez OpenAI i Cohere za pośrednictwem bibliotek: `openai` i `cohere`.
Question: Którzy dostawcy modeli udostępniają modele LLM?
Answer:
```

Choć powyższy prompt wygląda jak prosty łańcuch znaków, to określenie, co powinien zawierać i jak powinien się zmieniać w zależności od danych wejściowych podanych przez użytkownika, jest sporym wyzwaniem. W tym przykładzie wartości kontekstu (fragment `Context:`) i pytania (fragment `Question:`) są zapisane na stałe. A co w sytuacji, gdybyśmy chcieli przekazywać je dynamicznie?

Na szczęście biblioteka LangChain udostępnia interfejsy szablonów promptów, które ułatwiają tworzenie promptów konstruowanych z wykorzystaniem dynamicznych danych wejściowych.

Python

```
from langchain_core.prompts import PromptTemplate

template = PromptTemplate.from_template("""Odpowiedz na pytanie na podstawie podanego
    poniżej kontekstu. Jeśli na pytanie nie można odpowiedzieć na podstawie
    dostarczonych danych, to odpowiedz: "Nie wiem".

Context: {context}

Question: {question}

Answer: """)

template.invoke({
    "context": """Najnowsze osiągnięcia w dziedzinie przetwarzania języka naturalnego
        (NLP) są napędzane przez duże modele językowe (LLM). Modele te przewyższają
        swoje mniejsze odpowiedniki i stały się nieocenionym narzędziem dla programistów
        tworzących aplikacje z funkcjami NLP. Programiści mogą korzystać z tych modeli
        za pośrednictwem biblioteki `transformers` udostępnianej przez Hugging Face lub
        przy użyciu rozwiązań oferowanych przez OpenAI i Cohere za pośrednictwem
        bibliotek: `openai` i `cohere`.""",
    "question": "Którzy dostawcy modeli udostępniają modele LLM?"
})
```

JavaScript

```
import { PromptTemplate } from '@langchain/core/prompts'

const template = PromptTemplate.fromTemplate(`Odpowiedz na pytanie na podstawie
  podanego poniżej kontekstu. Jeśli na pytanie nie można odpowiedzieć na podstawie
  dostarczonych danych, to odpowiedz: "Nie wiem".

  Context: {context}

  Question: {question}

  Answer: `)

await template.invoke({
  context: `Najnowsze osiągnięcia w dziedzinie przetwarzania języka naturalnego (NLP)
    są napędzane przez duże modele językowe (LLM). Modele te przewyższają swoje mniejsze
    odpowiedniki i stały się nieocenionym narzędziem dla programistów tworzących
    aplikacje z funkcjami NLP. Programiści mogą korzystać z tych modeli za pośrednictwem
    biblioteki \`transformers\` udostępnianej przez Hugging Face lub przy użyciu rozwiązań
oferowanych przez OpenAI i Cohere za pośrednictwem bibliotek:
    \`openai\` i \`cohere\`.`,
  question: "Którzy dostawcy modeli udostępniają modele LLM?"
})
```

Wyniki:

```
StringPromptValue(text=' Odpowiedz na pytanie na podstawie podanego poniżej
    kontekstu. Jeśli na pytanie nie można odpowiedzieć na podstawie dostarczonych
    danych, to odpowiedz: "Nie wiem".\n\nKontekst: Najnowsze osiągnięcia w dziedzinie
    przetwarzania języka naturalnego (NLP) są napędzane przez duże modele językowe
    (LLM). Modele te przewyższają swoje mniejsze odpowiedniki i stały się nieocenionym
    narzędziem dla programistów tworzących aplikacje z funkcjami NLP. Programiści mogą
    korzystać z tych modeli za pośrednictwem biblioteki `transformers` udostępnianej
    przez Hugging Face lub przy użyciu rozwiązań oferowanych przez OpenAI i Cohere za
    pośrednictwem bibliotek: `openai` i `cohere`.\n\nPytanie: Którzy dostawcy modeli
    udostępniają modele LLM?\n\nOdpowiedź: ')
```

Ten przykład wykorzystuje dynamiczny prompt zamiast statycznego. Zmienna `template` zawiera strukturę końcowego promptu wraz z definicją miejsc, w których zostaną wstawione dynamiczne dane wejściowe.

Dzięki temu szablon może służyć jako przepis do tworzenia wielu statycznych, konkretnych promptów. Po uzupełnieniu szablonu konkretnymi wartościami — w tym przypadku kontekstem (`context`) i pytaniem (`question`) — otrzymujemy statyczny prompt gotowy do przekazania do modelu LLM.

Jak widać, argument `question` jest przekazywany dynamicznie w wywołaniu funkcji `invoke`. Domyślnie w bibliotece LangChain do definiowania dynamicznych parametrów prompty używają składni f-łańcuchów języka Python — każde słowo zapisane w nawiasach klamrowych, np. `{question}`, jest zastępowane wartościami przekazanymi podczas wykonania. W poprzednim przykładzie `{question}` zostało zastąpione łańcuchem znaków `"Którzy dostawcy modeli udostępniają modele LLM?"`.

Zobaczmy teraz, w jaki sposób można przekazać tak przygotowany prompt do modelu LLM firmy OpenAI, używając do tego celu możliwości biblioteki LangChain.

Python

```
from langchain_openai.llms import OpenAI
from langchain_core.prompts import PromptTemplate

# Zmiennych `template` i `model` można używać wiele razy
template = PromptTemplate.from_template("""Odpowiedz na pytanie na podstawie podanego
    poniżej kontekstu. Jeśli na pytanie nie można odpowiedzieć na podstawie
    dostarczonych danych, to odpowiedz: "Nie wiem".

Kontekst: {context}

Pytanie: {question}

Odpowiedź: """)

model = OpenAI()

# Zmienne `prompt` i `completion` są wynikami użycia zmiennych `template` i `model`
prompt = template.invoke({
    "context": """Najnowsze osiągnięcia w dziedzinie przetwarzania języka naturalnego
        (NLP) są napędzane przez duże modele językowe (LLM). Modele te przewyższają
        swoje mniejsze odpowiedniki i stały się nieocenionym narzędziem dla programistów
        tworzących aplikacje z funkcjami NLP. Programiści mogą korzystać z tych modeli
        za pośrednictwem biblioteki `transformers` udostępnianej przez Hugging Face lub
        przy użyciu rozwiązań oferowanych przez OpenAI i Cohere za pośrednictwem
        bibliotek: `openai` i `cohere`.""",
    "question": "Którzy dostawcy modeli udostępniają modele LLM?"
})

completion = model.invoke(prompt)
```

JavaScript:

```
import { PromptTemplate } from '@langchain/core/prompts'
import { OpenAI } from '@langchain/openai'

const model = new OpenAI()
const template = PromptTemplate.fromTemplate(`Odpowiedz na pytanie na podstawie
  podanego poniżej kontekstu. Jeśli na pytanie nie można odpowiedzieć na podstawie
  dostarczonych danych, to odpowiedz: "Nie wiem".

Kontekst: {context}

Pytanie: {question}

Odpowiedź: `)

const prompt = await template.invoke({
  context: `Najnowsze osiągnięcia w dziedzinie przetwarzania języka naturalnego (NLP) są
    napędzane przez duże modele językowe (LLM). Modele te przewyższają swoje mniejsze
    odpowiedniki i stały się nieocenionym narzędziem dla programistów tworzących
    aplikacje z funkcjami NLP. Programiści mogą korzystać z tych modeli za pośrednictwem
    biblioteki \`transformers\` udostępnianej przez Hugging Face lub przy użyciu
    rozwiązań oferowanych przez OpenAI i Cohere za pośrednictwem bibliotek: \`openai\`
```

```
    i \`cohere\`.`,
  question: "Którzy dostawcy modeli udostępniają modele LLM?"
})

await model.invoke(prompt)
```

Wyniki:

```
OpenAI i Cohere udostępniają modele LLM.
```

Jeśli chcesz stworzyć aplikację konwersacyjną opartą na sztucznej inteligencji, możesz użyć szablonu `ChatPromptTemplate`. Pozwala on na dynamiczne przekazywanie danych w zależności od roli wiadomości w konwersacji.

Python

```
from langchain_core.prompts import ChatPromptTemplate

template = ChatPromptTemplate.from_messages([
    ('system', '''Odpowiedz na pytanie na podstawie podanego poniżej kontekstu. Jeśli
      na pytanie nie można odpowiedzieć na podstawie dostarczonych danych, to
      odpowiedz: "Nie wiem".'''),
    ('human', 'Kontekst: {context}'),
    ('human', 'Pytanie: {question}'),
])

template.invoke({
    "context": """Najnowsze osiągnięcia w dziedzinie przetwarzania języka naturalnego
        (NLP) są napędzane przez duże modele językowe (LLM). Modele te przewyższają
        swoje mniejsze odpowiedniki i stały się nieocenionym narzędziem dla programistów
        tworzących aplikacje z funkcjami NLP. Programiści mogą korzystać z tych modeli
        za pośrednictwem biblioteki `transformers` udostępnianej przez Hugging Face lub
        przy użyciu rozwiązań oferowanych przez OpenAI i Cohere za pośrednictwem
        bibliotek: `openai` i `cohere`.""",
    "question": "Którzy dostawcy modeli udostępniają modele LLM?"
})
```

JavaScript

```
import { ChatPromptTemplate } from '@langchain/core/prompts'

const template = ChatPromptTemplate.fromMessages([
  ['system', `Odpowiedz na pytanie na podstawie podanego poniżej kontekstu. Jeśli
    na pytanie nie można odpowiedzieć na podstawie dostarczonych danych, to
    odpowiedz: "Nie wiem".`],
  ['human', 'Kontekst: {context}'],
  ['human', 'Pytanie: {question}'],
])

await template.invoke({
  context: `Najnowsze osiągnięcia w dziedzinie przetwarzania języka naturalnego (NLP) są
    napędzane przez duże modele językowe (LLM). Modele te przewyższają swoje mniejsze
    odpowiedniki i stały się nieocenionym narzędziem dla programistów tworzących
    aplikacje z funkcjami NLP. Programiści mogą korzystać z tych modeli za pośrednictwem
    biblioteki \`transformers\` udostępnianej przez Hugging Face lub przy użyciu
    rozwiązań oferowanych przez OpenAI i Cohere za pośrednictwem bibliotek: \`openai\`
    i \`cohere\`.`,
  question: "Którzy dostawcy modeli udostępniają modele LLM?"
})
```

Wyniki:

```
ChatPromptValue(messages=[SystemMessage(content=' Odpowiedz na pytanie na podstawie
    podanego poniżej kontekstu. Jeśli na pytanie nie można odpowiedzieć na podstawie
    dostarczonych danych, to odpowiedz: "Nie wiem".'), HumanMessage(content="Kontekst:
    Najnowsze osiągnięcia w dziedzinie przetwarzania języka naturalnego (NLP) są
    napędzane przez duże modele językowe (LLM). Modele te przewyższają swoje mniejsze
    odpowiedniki i stały się nieocenionym narzędziem dla programistów tworzących
    aplikacje z funkcjami NLP. Programiści mogą korzystać z tych modeli za pośrednictwem
    biblioteki `transformers` udostępnianej przez Hugging Face lub przy użyciu
    rozwiązań oferowanych przez OpenAI i Cohere za pośrednictwem bibliotek: `openai`
    i `cohere`."), HumanMessage(content='Pytanie: Którzy dostawcy modeli udostępniają
    modele LLM?')])
```

Zwróć uwagę, że prompt zawiera instrukcje zapisane jako `SystemMessage` oraz dwie instancje `HumanMessage`, które zawierają dynamiczne zmienne `context` oraz `question`. Nic nie stoi na przeszkodzie, by sformatować szablon w ten sam sposób i uzyskać statyczny prompt, który następnie przekażemy do modelu LLM w celu wygenerowania przewidywanego wyniku:

Python

```
from langchain_openai.chat_models import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate

# Zmiennych `template` i `model` można używać wiele razy
template = ChatPromptTemplate.from_messages([
    ('system', '''Odpowiedz na pytanie na podstawie podanego poniżej kontekstu. Jeśli
      na pytanie nie można odpowiedzieć na podstawie dostarczonych danych, to
      odpowiedz: "Nie wiem".'''),
    ('human', 'Kontekst: {context}'),
    ('human', 'Pytanie: {question}'),
])

model = ChatOpenAI()

# Zmienne `prompt` i `completion` są wynikami użycia zmiennych `template` i `model`

prompt = template.invoke({
    "context": """Najnowsze osiągnięcia w dziedzinie przetwarzania języka naturalnego
        (NLP) są napędzane przez duże modele językowe (LLM). Modele te przewyższają
        swoje mniejsze odpowiedniki i stały się nieocenionym narzędziem dla programistów
        tworzących aplikacje z funkcjami NLP. Programiści mogą korzystać z tych modeli
        za pośrednictwem biblioteki `transformers` udostępnianej przez Hugging Face lub
        przy użyciu rozwiązań oferowanych przez OpenAI i Cohere za pośrednictwem
        bibliotek: `openai` i `cohere`.""",
    "question": "Którzy dostawcy modeli udostępniają modele LLM?"
})

model.invoke(prompt)
```

JavaScript

```
import { ChatPromptTemplate } from '@langchain/core/prompts'
import { ChatOpenAI } from '@langchain/openai'

const model = new ChatOpenAI()
const template = ChatPromptTemplate.fromMessages([
  ['system', ` Odpowiedz na pytanie na podstawie podanego poniżej kontekstu. Jeśli
```

```
    na pytanie nie można odpowiedzieć na podstawie dostarczonych danych, to
    odpowiedz: "Nie wiem".`],
  ['human', 'Kontekst: {context}'],
  ['human', 'Pytanie: {question}'],
])

const prompt = await template.invoke({
  context: `Najnowsze osiągnięcia w dziedzinie przetwarzania języka naturalnego (NLP)
    są napędzane przez duże modele językowe (LLM). Modele te przewyższają swoje mniejsze
    odpowiedniki i stały się nieocenionym narzędziem dla programistów tworzących
    aplikacje z funkcjami NLP. Programiści mogą korzystać z tych modeli za pośrednictwem
    biblioteki \`transformers\` udostępnianej przez Hugging Face lub przy użyciu
rozwiązań oferowanych przez OpenAI i Cohere za pośrednictwem bibliotek: \`openai\`
    i \`cohere\`.`,
  question: "Którzy dostawcy modeli udostępniają modele LLM?"
})

await model.invoke(prompt)
```

Wyniki:

```
AIMessage(content="OpenAI i Cohere udostępniają modele LLM.")
```

Uzyskiwanie od modeli LLM wyników w określonych formatach

Proste wyniki tekstowe są przydatne, ale czasami będziemy chcieli, aby model LLM generował *dane ustrukturyzowane* — czyli wynik w formacie czytelnym dla komputera, takim jak JSON, XML, CSV, a nawet w określonym języku programowania, takim jak Python lub JavaScript. Jest to szczególnie użyteczne, gdy chcemy przekazać ten wynik do innej części kodu, umożliwiając modelowi LLM odegranie pewnej roli w działaniu naszej większej aplikacji.

Dane wyjściowe w formacie JSON

Najczęściej używanym formatem danych generowanych przez modele LLM jest JSON. Dane w formacie JSON można na przykład przesyłać do kodu frontendowego lub zapisywać w bazie danych.

Przy generowaniu danych w formacie JSON pierwszym krokiem jest zdefiniowanie schematu, którego model LLM ma przestrzegać podczas tworzenia wyniku. Następnie należy umieścić ten schemat w prompcie wraz z tekstem źródłowym. Przeanalizujmy następujący przykład:

Python

```
from langchain_openai import ChatOpenAI
from langchain_core.pydantic_v1 import BaseModel

class AnswerWithJustification(BaseModel):
    '''Odpowiedź na pytanie użytkownika wraz z jej uzasadnieniem'''
    answer: str
    '''Odpowiedź na pytanie użytkownika'''
    justification: str
```

```
    '''Uzasadnienie odpowiedzi'''

llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0)
structured_llm = llm.with_structured_output(AnswerWithJustification)
structured_llm.invoke("""Co waży więcej: kilo cegieł czy kilo pierza?""")
```

JavaScript

```
import { ChatOpenAI } from '@langchain/openai'
import { z } from "zod";

const answerSchema = z
  .object({
    answer: z.string().describe("Odpowiedź na pytanie użytkownika"),
    justification: z.string().describe(`Uzasadnienie odpowiedzi`),
  })
  .describe(`Odpowiedź na pytanie użytkownika wraz z jej uzasadnieniem.`);

const model = new ChatOpenAI({
  model: "gpt-3.5-turbo",
  temperature: 0,
}).withStructuredOutput(answerSchema)
await model.invoke("Co waży więcej: kilo cegieł czy kilo pierza?")
```

Wyniki:

```
{
  answer: "Ważą tyle samo",
  justification: " Obie rzeczy ważą tyle samo, czyli jeden kilogram. Kilo cegieł i kilo
    pierza mają tę samą masę, ponieważ"... 180 more characters
}
```

Zacznijmy od zdefiniowania schematu. W Pythonie najłatwiej to zrobić przy użyciu biblioteki Pydantic (służącej do walidacji danych względem schematów). W JavaScripcie analogiczną rolę pełni biblioteka Zod. Metoda `with_structured_output` wykorzystuje ten schemat na dwa sposoby:

- Schemat zostaje przekonwertowany na obiekt `JSONSchema` (format JSON opisujący strukturę danych JSON, ich typy, nazwy i opisy), który jest następnie przekazywany do modelu językowego. LangChain dobiera najlepszą metodę realizacji tego zadania dla każdego modelu, zazwyczaj wykorzystując wywołania funkcji lub odpowiednie podpowiedzi.
- Schemat służy również do walidacji wyniku zwróconego przez model językowy przed jego ostatecznym przekazaniem; gwarantuje to, że wygenerowane dane są zgodne z przekazanym schematem.

Inne formaty czytelne dla oprogramowania z wykorzystaniem parserów wyjściowych

Można również użyć modeli LLM lub modeli konwersacyjnych do generowania wyników w innych formatach, takich jak CSV lub XML. W takich przypadkach przydatne będą *parsery danych wyjściowych*. Są to klasy, które pomagają ustrukturyzować odpowiedzi zwracane przez modele LLM. Pełnią one dwie funkcje:

Dostarczają instrukcji dotyczących formatowania

Parsery danych wyjściowych można wykorzystać do dodania do promptu dodatkowych wskazówek, które pomogą modelowi LLM wygenerować tekst w formacie, który parser potrafi przetworzyć.

Określają sposoby walidacji i przetwarzania wyników

Główną funkcją parserów jest przetworzenie tekstowego wyniku zwróconego przez model LLM lub model konwersacyjny na bardziej ustrukturyzowany format, taki jak lista, XML itp. Może to obejmować usuwanie zbędnych informacji, poprawianie niekompletnych danych oraz walidację przetworzonych wartości.

Poniżej przedstawiliśmy przykład działania parsera danych wyjściowych:

Python

```
from langchain_core.output_parsers import CommaSeparatedListOutputParser
parser = CommaSeparatedListOutputParser()
items = parser.invoke("jabłka, banany, wiśnie")
```

JavaScript

```
import { CommaSeparatedListOutputParser } from '@langchain/core/output_parsers'

const parser = new CommaSeparatedListOutputParser()

await parser.invoke("jabłka, banany, wiśnie")
```

Wyniki:

```
['jabłka', 'banany', 'wiśnie']
```

Biblioteka LangChain udostępnia różnorodne parsery danych wyjściowych, przeznaczone do różnych zastosowań i obsługujące różne formaty, w tym CSV, XML i inne. W następnym podrozdziale dowiesz się, jak łączyć parsery danych wyjściowych z modelami i promptami.

Łączenie wielu elementów aplikacji korzystającej z modelu LLM

Kluczowe elementy, które poznałeś do tej pory, stanowią podstawowe bloki konstrukcyjne biblioteki LangChain. To prowadzi nas do istotnego pytania: jak efektywnie połączyć te elementy, aby zbudować aplikację korzystającą z modeli LLM?

Wykorzystanie interfejsu Runnable

Jak zapewne zauważyłeś, wszystkie przedstawione do tej pory przykłady do generowania wyników zwracanych przez model LLM (lub szablonu zapytania czy parsera danych wyjściowych) korzystają z podobnego interfejsu i metody `invoke()`. Wszystkie te komponenty mają następujące cechy:

- Implementują ten sam interfejs udostępniający następujące metody:
  - `invoke`: przekształca pojedyncze dane wejściowe w wyjściowe;
  - `batch`: efektywnie przekształca wiele danych wejściowych w wiele wyników;
  - `stream`: strumieniuje wyniki z pojedynczego wejścia w miarę ich tworzenia.
- Dostępne są wbudowane narzędzia do obsługi ponownego wykonania, alternatyw, schematów i zapewniania możliwości konfigurowania w trakcie działania.
- W Pythonie każda z tych trzech metod ma swój odpowiednik asynchroniczny.

Dzięki temu wszystkie komponenty działają w ten sam sposób, a poznanie interfejsu jednego z nich umożliwi stosowanie wszystkich pozostałych.

Python[1]

```
from langchain_openai.llms import ChatOpenAI

model = ChatOpenAI()

completion = model.invoke('Hi there!')
# Hi!

completions = model.batch(['Hi there!', 'Bye!'])
# ['Hi!', 'See you!']

for token in model.stream('Bye!'):
    print(token)
    # Good
    # bye
    # !
```

JavaScript

```
import { ChatOpenAI } from '@langchain/openai'

const model = new ChatOpenAI()

const completion = await model.invoke('Hi there!')
// Hi!

const completions = await model.batch(['Hi there!', 'Bye!'])
// ['Hi!', 'See you!']

for await (const token of await model.stream('Bye!')) {
  console.log(token)
  // Good
  // bye
  // !
}
```

[1] Przykład celowo pozostawiono w języku angielskim, aby wiernie zademonstrować działanie procesu generowania odpowiedzi przez model językowy token po tokenie, który jest zoptymalizowany pod kątem przetwarzania tekstów właśnie w tym języku — *przyp. tłum.*

W tym przykładzie przedstawiliśmy działanie trzech podstawowych metod:

- `invoke()`, która pobiera pojedynczą daną wejściową i zwraca pojedynczy wynik;
- `batch()`, dla której daną wejściową jest lista i która zwraca listę wyników;
- `stream()`, która pobiera pojedynczą daną wejściową i zwraca iterator przekazujący kolejne części wyników, w miarę jak stają się one dostępne.

W niektórych przypadkach, gdy bazowy komponent nie obsługuje danych wyjściowych mających charakter iteracyjny, zostanie zwrócona pojedyncza część zawierająca cały wynik.

Te komponenty można łączyć ze sobą na dwa sposoby:

Imperatywny
: Wywołuje komponenty bezpośrednio, na przykład za pomocą `model.invoke(...)`.

Deklaratywny
: Do tego celu służy LangChain Expression Language (LCEL), który przedstawiliśmy pobieżnie w dalszej części rozdziału.

Różnice pomiędzy tymi dwoma sposobami zostały podsumowane w tabeli 1.1, natomiast sposoby korzystania z każdego z nich zaprezentowaliśmy w kolejnych punktach rozdziału.

Tabela 1.1. Główne różnice między imperatywnym a deklaratywnym sposobem łączenia komponentów LangChain

	Metoda imperatywna	Metoda deklaratywna
Składnia	W całości w Pythonie lub JavaScripcie	LCEL
Realizacja współbieżna	Python: z użyciem wątków i koprocedur JavaScript: z użyciem `Promise.all`	Automatycznie
Strumieniowanie	Z użyciem słowa kluczowego `yield`	Automatycznie
Realizacja asynchroniczna	Z użyciem funkcji asynchronicznych	Automatycznie

Łączenie imperatywne

Programowanie imperatywne to po prostu wymyślna nazwa dla pisania kodu w sposób, do którego jesteś przyzwyczajony, czyli polegającego na łączeniu komponentów w funkcje i klasy. Oto przykład pokazujący, jak w taki sposób można łączyć prompty, modele i parsery wyjścia:

Python

```
from langchain_openai.chat_models import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import chain

# Elementy konstrukcyjne

template = ChatPromptTemplate.from_messages([
    ('system', 'Jesteś użytecznym asystentem.'),
    ('human', '{question}'),
])
```

```
model = ChatOpenAI()

# Połączenie elementów konstrukcyjnych w funkcję
# Dekorator @chain dodaje ten sam interfejs Runnable do każdej napisanej funkcji

@chain
def chatbot(values):
    prompt = template.invoke(values)
    return model.invoke(prompt)

# Użycie funkcji

chatbot.invoke({"question": "Którzy dostawcy modeli udostępniają modele LLM?"})
```

JavaScript

```
import {ChatOpenAI} from '@langchain/openai'
import {ChatPromptTemplate} from '@langchain/core/prompts'
import {RunnableLambda} from '@langchain/core/runnables'

// Elementy konstrukcyjne

const template = ChatPromptTemplate.fromMessages([
  ['system', 'Jesteś użytecznym asystentem.'],
  ['human', '{question}'],
])

const model = new ChatOpenAI()

// Połączenie elementów konstrukcyjnych w funkcję
// RunnableLambda dodaje do dowolnej funkcji ten sam interfejs Runnable

const chatbot = RunnableLambda.from(async values => {
  const prompt = await template.invoke(values)
  return await model.invoke(prompt)
})

// Użycie funkcji

await chatbot.invoke({
  "question": "Którzy dostawcy modeli udostępniają modele LLM?"
})
```

Wynik[2]*:*

```
AIMessage(content=" Kilku dostawców udostępnia modele językowe oparte na dużo większych
danych szkoleniowych, takie jak duże językowe modele (ang. Large Language Models, LLM),
w tym:
   1.  **OpenAI**: Jest jednym z wiodących dostawców modeli językowych, a ich model
GPT-3...")
```

Powyższy przykład przedstawia kompletną implementację chatbota, korzystającą z promptu i modelu konwersacyjnego. Jak widać, używa on znajomej składni Pythona i pozwala na dodanie do zaimplementowanej funkcji dowolnej niestandardowej logiki.

[2] Zamieszczone wyniki zostały celowo skrócone, gdyż nowe modele językowe generują na zadane pytanie rozbudowane odpowiedzi, których pełna treść nie jest istotna dla przykładu — *przyp. tłum.*

Z drugiej strony, jeśli chcesz umożliwić strumieniowanie lub działanie asynchroniczne, musisz odpowiednio zmodyfikować swoją funkcję, tak by na to pozwalała. Na przykład wsparcie dla strumieniowania można dodać w następujący sposób:

Python

```
@chain
def chatbot(values):
    prompt = template.invoke(values)
    for token in model.stream(prompt):
        yield token

for part in chatbot.stream({
    "question": "Którzy dostawcy modeli udostępniają modele LLM?"
}):
    print(part)
```

JavaScript

```
const chatbot = RunnableLambda.from(async function* (values) {
  const prompt = await template.invoke(values)
  for await (const token of await model.stream(prompt)) {
    yield token
  }
})

for await (const token of await chatbot.stream({
  "question": "Którzy dostawcy modeli udostępniają modele LLM?"
})) {
  console.log(token)
}
```

Wyniki:

```
AIMessageChunk(content="Hugging")
AIMessageChunk(content=" Face's")
AIMessageChunk(content=" `transformers`")
    ...
```

A zatem zarówno w języku JavaScript, jak i w Pythonie można zapewnić działanie strumieniowe swojej funkcji poprzez zwracanie wartości, które chcesz udostępniać strumieniowo, a następnie wywołanie tej funkcji z użyciem strumienia.

W przypadku działania asynchronicznego funkcję należałoby zaimplementować w następujący sposób:

Python

```
@chain
async def chatbot(values):
    prompt = await template.ainvoke(values)
    return await model.ainvoke(prompt)

await chatbot.ainvoke({"question": "Którzy dostawcy modeli udostępniają modele LLM?"})
# > AIMessage(content="""Kilku dostawców udostępnia modele językowe oparte na dużo większych danych
szkoleniowych, takie jak duże językowe modele (ang. Large Language Models, LLM), w tym...""")
```

To rozwiązanie dotyczy wyłącznie Pythona — w języku JavaScript działanie asynchroniczne jest jedyną dostępną możliwością.

Łączenie deklaratywne

LCEL to *deklaratywny język* służący do łączenia elementów biblioteki LangChain. LangChain kompiluje wyrażenia zapisane w języku LCEL do postaci *zoptymalizowanego planu wykonania*, zapewniającego możliwości automatycznego wykonywania równoległego, strumieniowania, śledzenia oraz działania asynchronicznego.

Przeanalizujmy przykład z poprzedniego punktu, zaimplementowany z użyciem LCEL:

Python

```
from langchain_openai.chat_models import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate

# Elementy konstrukcyjne

template = ChatPromptTemplate.from_messages([
    ('system', 'Jesteś użytecznym asystentem.'),
    ('human', '{question}'),
])

model = ChatOpenAI()

# Połączenie elementów przy użyciu operatora |

chatbot = template | model

# Wykonanie

chatbot.invoke({"question": "Którzy dostawcy modeli udostępniają modele LLM?"})
```

JavaScript

```
import { ChatOpenAI } from '@langchain/openai'
import { ChatPromptTemplate } from '@langchain/core/prompts'
import { RunnableLambda } from '@langchain/core/runnables'

// Elementy konstrukcyjne

const template = ChatPromptTemplate.fromMessages([
  ['system', 'Jesteś użytecznym asystentem.'],
  ['human', '{question}'],
])

const model = new ChatOpenAI()

// Połączenie elementów w jedną funkcję

const chatbot = template.pipe(model)

// Wykonanie

await chatbot.invoke({
  "question": "Którzy dostawcy modeli udostępniają modele LLM?"
})
```

Wyniki:

```
AIMessage(content="Kilku dostawców udostępnia modele językowe oparte na dużo większych
danych szkoleniowych, takie jak duże językowe modele (ang. Large Language Models, LLM),
w tym...")
```

Co najważniejsze, ostatni wiersz kodu w obu przykładach jest taki sam — oznacza to, że funkcji oraz sekwencji LCEL można używać w ten sam sposób, wywołując je za pomocą metod `invoke`, `stream` lub `batch`. W tej wersji przykładu, aby skorzystać z działania strumieniowego, wystarczy ograniczyć się do wywołania funkcji `stream` zamiast `invoke`:

Python

```
chatbot = template | model

for part in chatbot.stream({
    "question": "Którzy dostawcy modeli udostępniają modele LLM?"
}):
    print(part)
    # > AIMessageChunk(content="Kil")
    # > AIMessageChunk(content="ku")
    # > AIMessageChunk(content=" do")
    # ...
```

JavaScript

```
const chatbot = template.pipe(model)

for await (const token of await chatbot.stream({
  "question": "Którzy dostawcy modeli udostępniają modele LLM?"
})) {
  console.log(token)
}
```

W przypadku Pythona sytuacja wygląda podobnie, jeśli chcemy skorzystać z techniki programowania asynchronicznego:

Python

```
chatbot = template | model

await chatbot.ainvoke({
    "question": "Którzy dostawcy modeli udostępniają modele LLM?"
})
```

Podsumowanie

W tym rozdziale poznałeś podstawowe elementy i kluczowe komponenty niezbędne do tworzenia aplikacji korzystających z modeli LLM przy użyciu biblioteki LangChain. Aplikacje te składają się zasadniczo z łańcucha obejmującego model LLM służący do generowania predykcji, z promptu stanowiącego instrukcje kierujące model w stronę pożądanego wyniku oraz opcjonalnego parsera danych wynikowych, służącego do zwracania wyniku w pożądanym formacie.

Wszystkie komponenty biblioteki LangChain mają ten sam interfejs udostępniający metody `invoke`, `stream` i `batch`, które są używane do obsługi różnych danych wejściowych i wyjściowych. Komponenty te można łączyć i wykonywać w sposób imperatywny — wywołując je bezpośrednio — lub deklaratywny przy użyciu języka LCEL.

Podejście imperatywne jest przydatne w sytuacjach, gdy zamierzamy napisać dużo własnej logiki, natomiast deklaratywne sprawdza się lepiej w przypadkach prostego łączenia istniejących komponentów i stosowania nieznacznych modyfikacji.

W rozdziale 2. dowiesz się, jak przekazywać do chatbota AI dane zewnętrzne w formie *kontekstu*, co pozwala zbudować aplikację korzystającą z modelu LLM i umożliwiającą prowadzenie „rozmowy" z Twoimi danymi.

Ta książka sprawia, że nauka LangChain jest interesująca i zabawna!

— Rajat K. Goel, IBM

Jak zacząć tworzyć aplikacje AI, które potrafią wnioskować i wyszukiwać zewnętrzne dane w celu zapewniania odpowiedniego kontekstu? Sięgnij po LangChain! To popularne narzędzie programistyczne służy do tworzenia, uruchamiania aplikacji LLM, a także zarządzania nimi. LangChain jest używany przez wiele znanych firm, a jego popularność dynamicznie rośnie.

Jeśli znasz Pythona lub JavaScript i chcesz rozpocząć przygodę z aplikacjami AI — ta książka jest dla Ciebie! Krok po kroku zbudujesz agenta AI korzystającego ze wskazanego źródła danych, wyposażysz go w mechanizmy autoryzacji działań i umożliwisz mu zdobywanie dodatkowych informacji lub wyjaśnień. Dowiesz się, jak przygotować go do pracy w środowisku produkcyjnym z uwzględnieniem kwestii opóźnień, niezawodności i bezpieczeństwa. Nauczysz się również monitorować i stale ulepszać swoje aplikacje.

To lektura obowiązkowa dla programistów, którzy chcą przesuwać granice możliwości LangChain!

— Tom Taulli, konsultant IT i autor książek

Najciekawsze zagadnienia:

- zwiększanie dokładności LLM za pomocą techniki RAG
- inteligentna komunikacja aplikacji AI z użytkownikami
- architektura agentowa z wykorzystaniem LangGraph
- integracja z interfejsami API i narzędziami firm trzecich
- monitorowanie, testowanie i ewaluacja aplikacji AI
- praktyczne zastosowanie biblioteki LangChain

Mayo Oshin jest doradcą do spraw sztucznej inteligencji. Był wczesnym współtwórcą biblioteki LangChain, a także pionierem w popularnym ruchu *AI chat with data*.

Nuno Campos jest inżynierem, jednym z twórców biblioteki LangChain. Ma dziesięcioletnie doświadczenie jako inżynier oprogramowania, architekt i opiekun rozwiązań tworzonych w językach Python i JavaScript.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-3072-8
9 788328 930728
Cena: 79,00 zł