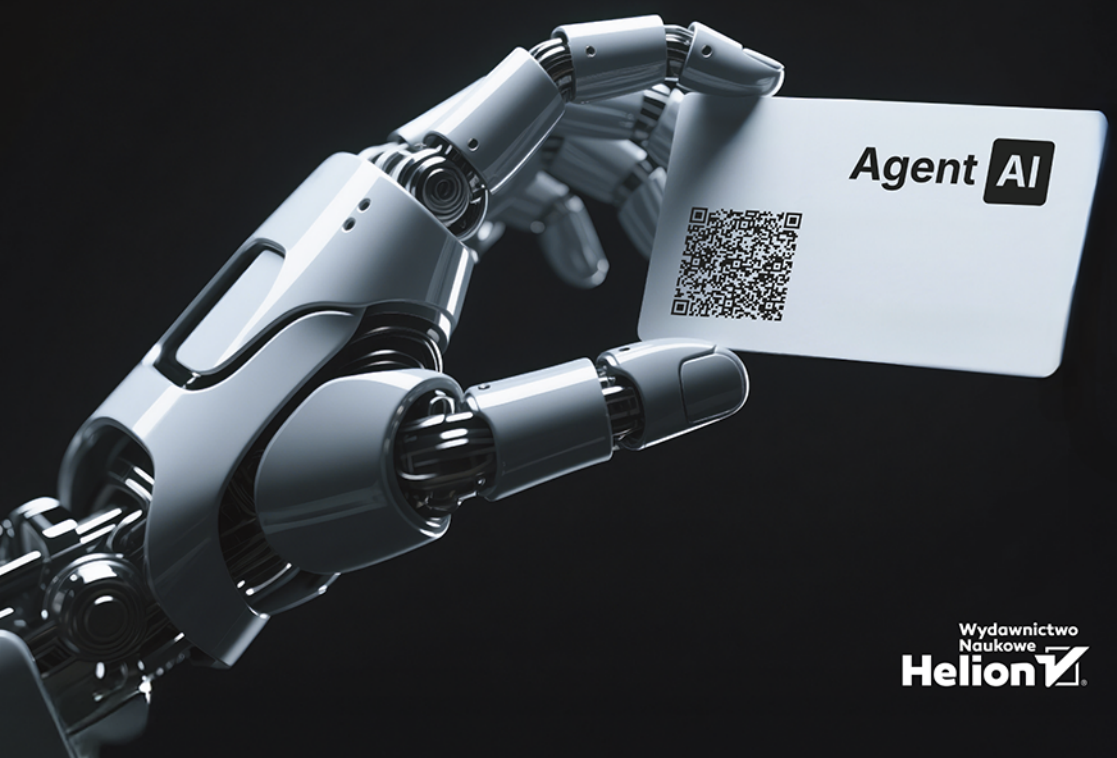
Mariusz Hofman

Agenci AI

bazujący na **modelach językowych**

istota | konfiguracje | zastosowania



Wydawnictwo Naukowe Helion

Redaktor prowadzący: Małgorzata Kulik

Recenzja naukowa: dr hab. Andrzej Sobczak, prof. Szkoły Głównej Handlowej

Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Grafika na okładce została wykorzystana za zgodą AdobeStock.com.

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)

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

ISBN: 978-83-289-2597-7

Printed in Poland.

Spis treści

ROZDZIAŁ 3.

Podstawowe implementacje agentów bazujących na modelach językowych

W tej części monografii zaprezentowane zostaną przykładowe implementacje agentów opartych na modelach językowych. Wybrane implementacje cechują się interesującą konfiguracją, zdefiniowaną specjalizacją oraz znacznym potencjałem wykorzystania poszczególnych rodzajów agentów. Omawiane rozwiązania będą dotyczyły następujących typów agentów:

1. RAG Agent — agent, który umożliwia użytkownikowi interakcję z dokumentami zawierającymi informacje w języku naturalnym, najczęściej w formacie PDF. Potrafi wyszukiwać potrzebne informacje w obszernych fragmentach tekstu lub całych zbiorach dokumentów. Uwzględnia kontekst i jest w stanie odpowiadać na pytania, dostarczać konkretne fragmenty oraz generować streszczenia i podsumowania [Singh i in., 2025].
2. Chat Agent — agent zaprojektowany do interakcji z użytkownikiem w formie naturalnego dialogu. Jego głównym zadaniem jest prowadzenie rozmowy w taki sposób, aby dostarczać użytkownikowi spersonalizowane odpowiedzi uwzględniające kontekst wynikający z dotychczasowych interakcji [Wu i in., 2023].
3. Data Agent — agent zaprojektowany do interakcji z plikami zawierającymi dane w postaci numerycznej, najczęściej zapisanymi w formatach CSV lub XLS. Pozwala użytkownikom na zadawanie pytań dotyczących danych zawartych w tych plikach, jak również na wykonywanie na tych danych (za pomocą języka naturalnego) bardziej zaawansowanych operacji, takich jak filtrowanie, sortowanie, agregowanie, analizowanie czy wizualizowanie [M. Sun i in., 2024].
4. SQL Agent — agent zaprojektowany do interakcji z bazami danych SQL za pomocą zapytań w języku naturalnym. Pozwala użytkownikom na tworzenie, modyfikowanie oraz zarządzanie danymi w bazie przy wykorzystaniu intuicyjnych interfejsów lub okien dialogowych, co eliminuje potrzebę znajomości języka SQL [Z. Wang i in., 2024].
5. State (Stateflow) Agent — agent pozwalający determinować określone ścieżki realizacji zadań na podstawie informacji obecnych w stanie (*state*) agenta. Stan agenta jest rodzajem repozytorium, w którym gromadzone są kluczowe z punktu widzenia przepływu pracy informacje. Dzięki temu może on dostosowywać swoje działania w zależności od kontekstu i aktualnej sytuacji lub brać pod uwagę dotychczasowe interakcje z użytkownikiem (albo innymi agentami).

Wymienione powyżej konfiguracje agentów zbudowane zostały przy wykorzystaniu bibliotek LangChain, AutoGen oraz LangGraph. Omawiane implementacje mają charakter demonstracyjny, a przywoływane fragmenty kodu służą jedynie zaprezentowaniu sposobu uruchamiania poszczególnych komponentów agenta (modelu, narzędzi czy modułu pamięci). Kod ilustrujący konfigurowanie agenta uwzględnia jedynie kluczowe mechanizmy charakterystyczne dla bibliotek, w ramach których jest on uruchamiany. Należy zatem pamiętać, że zaprezentowane w tej części monografii konfiguracje agentów nie mają charakteru w pełni uruchamialnego kodu.

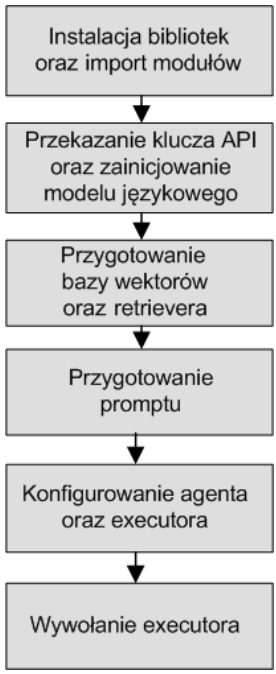
Takie podejście zostało przyjęte ze względu na gwałtowny rozwój tych bibliotek (zwłaszcza LangChain). Powoduje to, że dostępne są różne wersje dokumentacji, co może przekładać się na zmienność podejść, metod oraz sposobów konfigurowania agentów. W monografii skupiono się na logice projektowania, budowania i uruchamiania agentów, a nie na aktualnym akurat podejściu, które w momencie jej ukazania się może być już nieaktualne bądź ulec modyfikacji. Prezentowane rozwiązania mają więc na celu bardziej ukazanie zasad budowania agentów, a nie gotowych do wdrożenia implementacji.

3.1. RAG Agent (LangChain)

Jako pierwsza omówiona zostanie implementacja agenta wykorzystującego RAG i przygotowana za pomocą biblioteki LangChain. Agent ten — do generowania odpowiedzi na pytania bądź wykonywania zleconych zadań — używa specyficznego kontekstu pozyskanego z dokumentów zapisanych najczęściej w postaci plików PDF. Zawartość tych plików jest przekształcana w wektory (*embeddings*) oraz zapisywana w bazie wektorów (*vecrorestore*), w której to bazie agent wyszukuje odpowiednie fragmenty tekstu (*docs*). Tworzą one kontekst, który służy agentowi do wygenerowania odpowiedzi na pytanie użytkownika lub jest pomocny przy wykonywaniu zadania. Sekwencja uruchomienia tego typu agenta zakłada wykonanie kilku działań widocznych na rysunku 3.1.

Pierwszym krokiem jest instalacja niezbędnych pakietów oraz import potrzebnych modułów. Najczęściej instalowane są pakiety *langchain*, *langchain_core*, *langchain_community* oraz *langchain_openai*, jeżeli agent bazował będzie na modelach językowych firmy OpenAI. Jeżeli chcemy wykorzystać inne modele, np. Claude firmy Anthropic — konieczne będzie zainstalowanie pakietu *langchain_anthropic*. Jeżeli agent korzystał będzie z modeli *open source* (np. LLama 3.1) — potrzebne jest zainstalowanie pakietu *langchain_ollama*. Konieczne będzie również zainstalowanie modułu pozwalającego na budowanie baz wektorów, na przykład *Chroma*, *Faiss*, *Pinecone* lub innego preferowanego przez użytkownika.

Następnie konieczne jest przekazanie kluczy API do modeli językowych. Klucze te są zwykle definiowane jako zmienne środowiskowe lub są zapisane w plikach z rozszerzeniem .env. Aby agent mógł działać, konieczne jest zainicjowanie modelu językowego jako instancji konkretnej klasy (ChatOpenAI, ChatAnthropic, czy ChatOllama). Obrazuje to poniższy fragment kodu, w którym model językowy zainicjowany został w dwóch wariantach jako model GPT-4o lub Claude 3.5 Sonnet:



RYSUNEK 3.1. Schemat implementacji Agenta RAG w bibliotece LangChain (źródło: opracowanie własne)

```
from langchain_openai import ChatOpenAI
model = ChatOpenAI(model_name='gpt-4o', temperature=0)
```

lub:

```
from langchain_anthropic import ChatAnthropic
anthropic_model=ChatAnthropic(model="claude-3-5-sonnet-20240620", temperature=0)
```

Dalej niezbędne jest przygotowanie bazy wektorów oraz narzędzia, za pomocą którego agent będzie mógł wyszukiwać potrzebne informacje w bazie. Działanie to realizowane jest w ramach kilku kroków:

1. Wybór loadera — w zależności od formatu pliku zapisanych danych, z których ma korzystać agent. W przypadku plików PDF loader będzie instancją klasy *PyPDFLoader* — jeżeli będzie to jeden plik; lub *PyPDFDirectoryLoader* — w sytuacji, gdy wczytywanych jest wiele plików zapisanych w jednym folderze. Dla każdej klasy kluczowym parametrem jest ścieżka do pliku PDF lub folderu zawierającego pliki PDF. Wczytywanie pliku lub plików PDF wymaga użycia metody *load* w instancji jednej lub drugiej klasy. Mechanizm ten ilustruje poniższy przykład:

   ```
   from langchain.document_loaders import PyPDFLoader
   loader=PyPDFLoader("/content/drive/MyDrive/Mój_plik.pdf", extraction_mode="layout",
       extract_images=True)
   data = loader.load()
   ```

   W tym przypadku *loader* jest instancją klasy *PyPDFLoader*, zaś do konstruktora klasy przekazywane są trzy argumenty. Pierwszym z nich jest ścieżka do wczytywanego pliku. Drugim jest *extraction_mode*, który określa sposób porządkowania wczytywanego tekstu. W załączonym przykładzie przyjmuje on argument

layout, który podczas wczytywania tekstu z pliku pozwala zachować oryginalny układ dokumentu wraz z zawartymi w nim elementami graficznymi. Trzeci parametr to *extract_images*, który wskazuje, czy *loader* ma brać pod uwagę zawarte w tekście rysunki oraz elementy graficzne. W przykładzie przyjmuje on argument *True*, co oznacza że *loader*, wczytując tekst z pliku, opisuje również zawarte we wczytywanym tekście elementy graficzne. Domyślnie ten parametr przyjmuje wartość *False*.

2. Określenie sposobu podziału wczytywanego tekstu na mniejsze elementy jest kluczowe dla skutecznego działania retrievera, z którego korzysta agent. Jest to istotna część konfiguracji agenta, decyduje bowiem o jakości pozyskiwanego kontekstu oraz przekłada się na precyzję i rzetelność generowanej przez agenta odpowiedzi lub rezultat zadania, które agent ma wykonać. Sposób podziału wczytywanego tekstu na mniejsze elementy (*docs*) ilustruje poniższy fragment kodu:

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
text_splitter=RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=40)
docs = text_splitter.split_documents(data)
```

Splitter jest w powyższym przykładzie inicjowany jako instancja klasy *RecursiveCharacterTextSplitter*. Wymaga to podania parametrów dla konstruktora klasy, którymi są *chunk_size* — wskazujące, jaka będzie długość każdego wyodrębnionego fragmentu tekstu; oraz *chunk_overlap* — czyli liczba znaków, o którą kolejne fragmenty będą na siebie nachodzić. Oczywiście można wykorzystywać inne rodzaje splitterów na przykład *SemanticChunker*, który dzieli ładowany tekst na fragmenty, bazując na ich podobieństwie semantycznym. Można też stosować jeszcze bardziej zaawansowane splittery semantyczne (np. *AI21SemanticTextSplitter*) lub wyspecjalizowane w dzieleniu tekstu zapisanego w określonym formacie (np. *HTMLHeaderTextSplitter*, *HTMLSection Splitter* lub *RecursiveJsonSplitter*). Należy jednak pamiętać, że odpowiedni podział ładowanego tekstu na fragmenty (chunkowanie) ma istotny wpływ na skuteczność działania retrievera używanego przez agenta jako narzędzie. Dlatego należy zwrócić szczególną uwagę na sposób, w jaki tekst jest dzielony — uwzględniając zarówno jego rodzaj, jak i format, w jakim jest zapisany.

3. Konwersja załadowanego i podzielonego na fragmenty tekstu na wektory oraz zapisanie ich w bazie wektorów (*embeddings*). Tę sekwencję działań ilustruje poniższy fragment kodu:

```
from langchain_openai.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Chroma
embedding = OpenAIEmbeddings()
vectorstore = Chroma.from_documents(documents=docs, embedding=embedding)
```

Najpierw wybierany jest model, za pomocą którego tekst zostanie przekształcony na wektory (numeryczną reprezentację tekstu). Model ten jest w zaprezentowanym powyżej przykładzie instancją klasy *OpenAIEmbeddings*. Niemniej można w tym celu wykorzystywać inne dostępne oraz darmowe modele, które są wtedy instancją klasy *OllamaEmbeddings*. Może to być następujące wywołanie modelu: *embedding = OllamaEmbeddings(model="llama3.1:8b")*. W tym wariancie konieczne jest podanie nazwy modelu jako argumentu przekazywanego do konstruktora klasy. Baza wektorów jest inicjowana jako instancja klasy *Chroma* i wykorzystuje metodę *from_documents*, aby przekształcić poszczególne

fragmenty tekstu (*docs*) w wektory oraz zapisać je w jednym repozytorium, zachowując jednocześnie metadane pozwalające określić, z którego fragmentu tekstu oraz dokumentu one pochodzą. Wymaga to przekazania jako argumentu zmiennej, do której przypisany jest podzielony tekst (*documents=docs*). Proces budowania bazy wektorów wymaga również wskazania wcześniej zainicjowanego modelu, którego konfiguracja przekazywana jest jako argument do konstruktora klasy (*embedding=embeddings*). Jak już wcześniej wspomniano, zamiast korzystać z klasy z pakietu *Chroma*, do budowy bazy wektorów możemy wykorzystać także inne moduły i klasy, na przykład *FAISS* lub *Pinecone*.

4. Definiowanie retrievera. W tym miejscu inicjujemy obiekt, który wykorzystuje wcześniej przygotowaną bazę wektorów oraz metodę *as_retriever*, aby wyszukiwać w niej fragmenty tekstu, bazując na podobieństwie ich reprezentacji numerycznych (wektorów). Mówiąc bardziej obrazowo *retriever* pozwala wyszukiwać najlepiej pasujące do zapytania fragmenty tekstu. Zainicjowanie retrievera za pomocą metody *as_retriever* bez podania konkretnych argumentów spowoduje, że domyślnie metoda ta wykorzysta mechanizm *similarity_search*, czyli wyszukiwania opartego na podobieństwie semantycznym. Niemniej możliwe jest bardziej precyzyjne zdefiniowanie parametrów retrievera, w tym wykorzystanie innych, nieco bardziej zaawansowanych mechanizmów wyszukiwania. Na przykład MMR (Maximum Marginal Relevance), który pozwala sterować w procesie wyszukiwania trafnością oraz różnorodnością wyszukiwanych wyników. Ilustruje to poniższy fragment kodu:

```
pdf_retriever = vectorstore.as_retriever(search_type="mmr",
                                         search_kwargs={'k': 3, 'lambda_mult': 0,
                                                        'fetch_k': 10}
)
```

   W tym przypadku do metody *as_retriever* zostały przekazane dwa parametry. Pierwszym z nich był *search_type*, który przyjął argument *"mmr"*. Tym samym *retriever* będzie korzystał z innej niż domyślna metody wyszukiwania wyników, w tym przypadku *maximum marginal relevance*. Drugi, *search_kwargs*, którego argumentem jest słownik zawierający następujące klucze:

   4.1. *k: 3*, który wskazuje, że *retriever* ma ostatecznie zwrócić 3 najbardziej pasujące do zapytania fragmenty (*docs*).

   4.2. *lambda_mult: 0*, który steruje równoważeniem trafności i różnorodności wyników. W tym przypadku przekazana wartość (którą jest 0) wskazuje, że mechanizm wyszukiwania retrievera nie wprowadza dodatkowego aspektu, którym jest różnicowanie wyszukanych wyników. Priorytetem pozostaje trafność wyszukanych fragmentów tekstu.

   4.3. *fetch_k: 10*, który wskazuje, że na etapie wstępnym *retriever* wyszukuje 10 fragmentów tekstu (wyników), spośród których algorytm MMR wybiera 3 najlepsze.

   Należy również pamiętać, że parametry i ich argumenty są kluczowe dla skutecznego działania retrievera oraz powinny być dostosowane do specyfiki zadań, które będzie realizował agent wykorzystujący go jako narzędzie.

5. Ostatnim krokiem jest zdefiniowanie retrievera jako narzędzia, z którego będzie korzystał agent. Ilustruje to poniższy fragment kodu:

```
from langchain.tools.retriever import create_retriever_tool
retriever_tool = create_retriever_tool(
    pdf_retriever,
    "PDF_Document_Retriever",
    "Użyj tego narzędzia zawsze, gdy ktoś poprosi cię o przeszukanie pliku PDF
    i wyszukanie tam informacji."
)
tools = [retriever_tool]
```

Definiowanie narzędzia polega na utworzeniu nowego obiektu (*retriever_tool*) za pomocą funkcji *create_retriever_tool*. Funkcja ta generuje uruchamialny (*runnable*) obiekt, który agent może wywoływać — przekazując dane wejściowe (zapytanie) — oraz otrzymywać dane wyjściowe w postaci listy wyszukanych fragmentów tekstu (*docs*). Wywołanie funkcji tworzącej narzędzie wymaga podania trzech argumentów:

5.1. Retrievera — obiektu odpowiedzialnego za wyszukiwanie (zdefiniowanego w poprzednim kroku — *pdf_retriever*).

5.2. Nazwy — unikalnej etykiety, która umożliwia agentowi poprawnie identyfikować narzędzie. W omawianym przypadku nazwa to *PDF_Document_Retriever*, która na ogólnym poziomie wskazuje agentowi, do czego służy to narzędzie.

5.3. Opisu — który pozwala agentowi dokładnie zrozumieć, czym jest narzędzie, do czego służy oraz w jakich sytuacjach agent powinien z tego narzędzia korzystać. W prezentowanym przykładzie opis to: *"Użyj tego narzędzia zawsze, gdy ktoś poprosi cię o przeszukanie pliku PDF i wyszukanie tam informacji"*.

Ostatnim krokiem jest wpisanie narzędzia *retriever_tool* na listę narzędzi dostępnych dla agenta — *tools*. W załączonym przykładzie lista narzędzi zawiera jeden element: *tools = [retriever_tool]*. Jednakże na liście może znajdować się więcej narzędzi, a każde z nich musi być zdefiniowane jako narzędzie oraz posiadać unikalną nazwę i opis, który pozwala agentowi wybrać odpowiednie z nich do realizacji konkretnego zadania.

Dalej konieczne jest zdefiniowanie promptu agenta. Możemy skorzystać z gotowych szablonów promptów, które oferuje biblioteka LangChain. Można wykorzystać w tym celu klasę *hub* oraz pobrać (za pomocą metody *pull*) gotowe szablony, które są przygotowane i skonfigurowane pod kątem określonych zastosowań agenta. Argumentem przekazywanym do metody *pull* jest w takim przypadku nazwa pobieranego z repozytorium promptu zapisana jako zmienna tekstowa (*string*). Sposób przypisywania agentowi gotowego szablonu promptu może w takim przypadku wyglądać następująco:

```
from langchain import hub
prompt = hub.pull("hwchase17/openai-tools-agent").
```

Jest to dobre rozwiązanie w sytuacjach, gdy nie ma potrzeby indywidualnego profilowania agenta lub wskazywania mu oczekiwanych sposobów rozumowania i działania. W przypadku, gdy agent będzie korzystał z customizowanego promptu, konieczne jest zdefiniowanie szablonu promptu zawierającego wszystkie przekazywane w nim informacje (zob. sekcja 2.2.3). Najczęściej modyfikowany jest

komunikat *SystemMessage*, który zawiera profil agenta oraz opis sposobu rozumowania. Ilustruje to poniższy przykład:

```
from langchain_core.prompts import ChatPromptTemplate
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", " Jesteś agentem RAG. Twoim zadaniem jest wyszukiwanie
         i analizowanie informacji, aby dostarczać precyzyjne odpowiedzi.
         Używaj dostępnych narzędzi, aby pobrać potrzebne informacje."),
        MessagesPlaceholder("chat_history", optional=True),
        ("human", "{input}"),
        MessagesPlaceholder("agent_scratchpad"),
    ]
)
```

Należy pamiętać, że w tym szablonie promptu placeholder *chat_history* jest opcjonalny. Oznacza to, że w momencie wywoływania agenta nie jest konieczne przekazywanie argumentu tego parametru (szablon zwróci dla tego placeholdera pustą listę). Natomiast ustawienie tego parametru na *False* pozwoli przekazać do promptu listę komunikatów stanowiących dotychczasową historię konwersacji użytkownika z agentem. Jest to użyteczne i jednocześnie elastyczne rozwiązanie pozwalające w razie potrzeby uwzględnić historię rozmowy oraz lepiej kontekstualizować odpowiedzi agenta.

Ostatnim elementem implementacji jest przygotowanie instancji agenta oraz *executora*. Ilustruje to poniższy fragment kodu:

```
agent = create_openai_tools_agent(model, tools, prompt)
executor = AgentExecutor(agent=agent, tools=tools)
```

Agent jest inicjowany poprzez wywołanie funkcji *create_openai_tools_agent* oraz przekazanie jako argumentów: instancji modelu językowego, listy narzędzi oraz szablonu promptu. Są to wymagane przez funkcje parametry wejściowe. Można oczywiście inicjować inne typy agentów, wykorzystując inne funkcje (np. *create_openai_functions_agent* czy *create_openai_tools_agent*). Konieczne jest wtedy jednak załadowanie (przy pomocy klasy *hub* oraz metody *pull*) gotowego szablonu promptu dla danego typu agenta lub przygotowanie odpowiedniego szablonu za pomocą klasy *ChatPromptTemplate*. Należy również pamiętać, że poszczególne typy agentów mogą wymagać przekazania różnych informacji do szablonów promptów w formie placeholderów.

Executor jest inicjowany jako instancja klasy *AgentExecutor*, zaś jako argumenty konstruktora klasy przekazywana jest konfiguracja agenta (obiekt przygotowany wcześniej) oraz ponownie lista narzędzi, z których agent może korzystać. Możemy przekazać również inne argumenty, np. *verbose* (z argumentem *True*), jeżeli działania agenta mają być widoczne. Lub *handle_parsing_errors* (również z argumentem *True*), jeżeli *executor* ma sam radzić sobie z ewentualnymi problemami, których źródłem są różne formaty przetwarzanych przez niego danych wejściowych i wyjściowych. Możliwe jest również zintegrowanie tej konfiguracji agenta z modułem pamięci za pomocą parametru *memory*, w którym jako argument należy przekazać skonfigurowany bufor pamięci, na przykład jako obiekt klasy *ConversationBufferMemory*. Taka integracja działa poprawnie, choć należy zaznaczyć, że tego typu rozwiązanie traktowane jest jako przestarzałe (*legacy*) i niezalecane w nowych

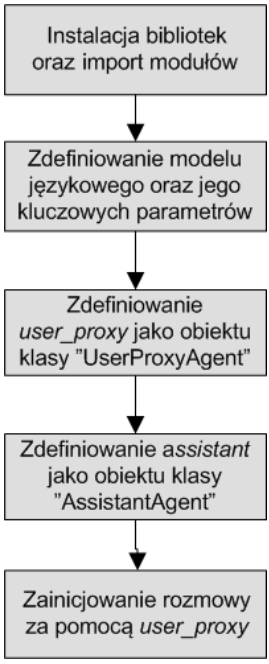
konfiguracjach agentów (stosowany jest wtedy mechanizm opisany w rozdziale 2.4). *Executor* działa zgodnie z logiką omówioną we wcześniejszych częściach monografii i jest wywoływany za pomocą metody *invoke*, co prezentuje poniższy fragment kodu:

```
result = executor.invoke(
    {"input": "Na podstawie pozyskanych informacji scharakteryzuj rodzaje RAG"}
)
```

W tym przypadku wykorzystywana jest metoda *invoke* dla obiektu, którym jest *agent_executor*. Argumentem przekazywanym do tej metody jest słownik (*dict*), w którym wymagany przez szablon promptu klucz *input* zawiera przypisane mu pytanie użytkownika. Wynik przypisywany jest do zmiennej *result* i może być wyświetlony przy pomocy funkcji *print*.

3.2. Chat Agent (AutoGen)

Implementacja agenta konwersacyjnego przeprowadzona zostanie za pomocą biblioteki AutoGen. Wymaga ona zdefiniowania dwóch współpracujących ze sobą agentów. Pierwszym z nich jest User Proxy, czyli agent pośredniczący pomiędzy użytkownikiem a drugim agentem, którym jest Assistant Agent. Logikę tej implementacji przedstawia rysunek 3.2.



RYSUNEK 3.2. Schemat implementacji agenta konwersacyjnego w bibliotece AutoGen (źródło: opracowanie własne)

Skorowidz

J

K

L

M

N

O

P

R

Agenci, którzy stoją po stronie biznesu

Agenci AI to algorytmy wykorzystujące modele językowe jako *reasoning engine*. Są one zdolne do postrzegania otoczenia, rozumowania i podejmowania decyzji, co czyni je przydatnymi w wielu dziedzinach biznesu, między innymi:

- **w spersonalizowanej obsłudze klienta**
- **w automatyzacji procesów biznesowych**
- **w zaawansowanej analityce biznesowej**
- **we wspieraniu ludzi pracujących w takich działach jak HR czy R&D**

Użycie agentów AI może przynieść firmom wymierne oszczędności, usprawnić proces podejmowania decyzji i w efekcie zagwarantować trwałą przewagę konkurencyjną.

Autor tej książki stawia sobie za cel wyjaśnienie istoty agentów opartych na modelach językowych, a także omówienie ich kluczowych architektur — od prostych, wyspecjalizowanych rozwiązań po złożone systemy współdziałających ze sobą agentów. Dodatkowo prezentuje przykłady zastosowań wybranych konfiguracji w realiach quasi-biznesowych.

Mariusz Hofman — doktor habilitowany nauk ekonomicznych, profesor Uniwersytetu Marii Curie-Skłodowskiej w Lublinie. Doświadczony wykładowca i konsultant w zakresie zarządzania projektami i procesami. Entuzjasta agentów AI, propagujący ich zastosowanie w organizacjach do automatyzacji rutynowych zadań i poszerzania ludzkiego potencjału.

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 978-83-289-2597-7

9 788328 925977

Cena: 59,90 zł