



Interakcja z dużym modelem językowym

Zdolność dużych modeli językowych do generowania ustrukturyzowanych treści umożliwia ich integrację z logiką aplikacji, co pozwala programistycznie sterować ich zachowaniem. Na obecnym etapie rozwoju pełnią rolę narzędzia, które umożliwia przetwarzanie i generowanie danych w sposób dotąd niemożliwy do osiągnięcia programistycznie (np. z pomocą wyrażeń regularnych).

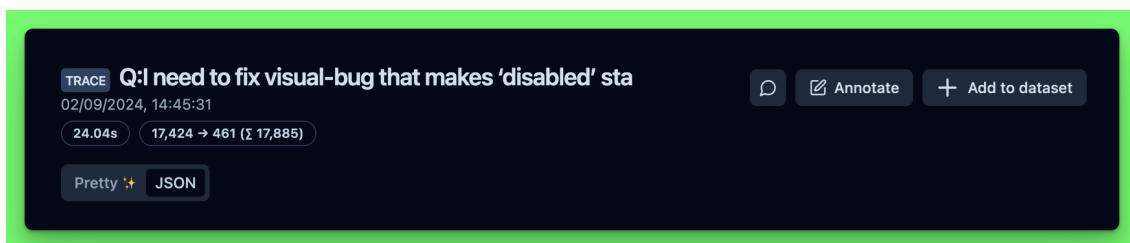
W AI_devs 3 skupimy się na programistycznej interakcji z dużymi modelami językowymi poprzez API, budując częściowo-autonomiczne narzędzia zwane "Agentami AI". To złożone rozwiązania wymagające praktycznego doświadczenia w programowaniu i dobrego zrozumienia natury dużych modeli językowych.

Narzędzia te mogą realizować najróżniejsze zadania i procesy, ale nie są uniwersalne. Dlatego **skupimy się na tworzeniu ich indywidualnych komponentów oraz modułów**. W ten sposób możliwe będzie ich połączenie w różnych konfiguracjach i dopasowanie do naszych potrzeb.

Jeszcze kilkanaście miesięcy temu wybór modelu ogólnego zastosowania praktycznie zaczynał się i kończył na OpenAI. Natomiast dziś pod uwagę możemy brać:

- OpenAI: Modele z rodziny o1, GPT, w tym także TTS, Whisper i Embedding
- Anthropic: Modele z rodziny Claude (tylko tekst + obraz)
- Vertex AI (Google): Modele Gemini oraz wybranych dostawców (np. Anthropic) i inne
- xAI: Modele Grok, które dość szybko przebiły się na szczyty rankingów (top10).
- Amazon Bedrock (Amazon): Modele Anthropic, Mistral czy Meta i inne
- Azure (Microsoft): Modele OpenAI, Meta i inne
- Groq: Modele Open Source, np. Llama
- a także kilka innych, np.: OpenRouter, Perplexity, Cerebras, Databricks, Mistral AI czy Together AI

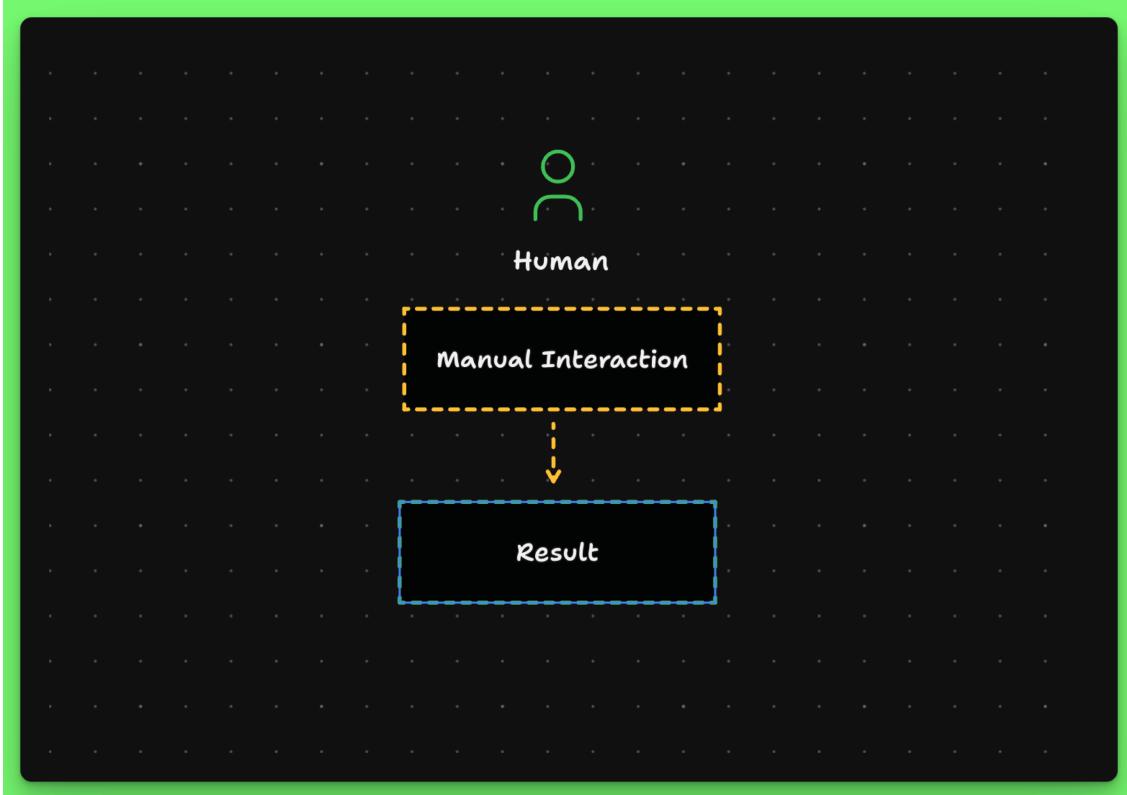
Możemy zatem wybierać między różnymi ofertami cenowymi, limitami dostępu do API, polityką prywatności i przetwarzania danych, a także samymi modelami. Jest to istotne, ponieważ agenci AI będą autonomicznie korzystać z naszych baz wiedzy lub uzyskają dostęp do narzędzi. Przełoży się to na działanie na dość dużej skali, uwzględniającej przetwarzanie nawet dziesiątek milionów tokenów, co generuje zauważalne koszty. Obrazuje to poniższy przykład zapytania z prośbą do Agenta AI o zapisanie zadań w [Linear](#), co przełożyło się na 17,400 tokenów zapytania (input) i 461 tokenów odpowiedzi (output). Warto też zwrócić uwagę na czas wykonania zapytania, czyli "aż" 24 sekundy.



Przykład zapytania do Agenta AI z prośbą o zarządzanie zadaniami obrazuje skalę przetwarzanych tokenów oraz czasu reakcji

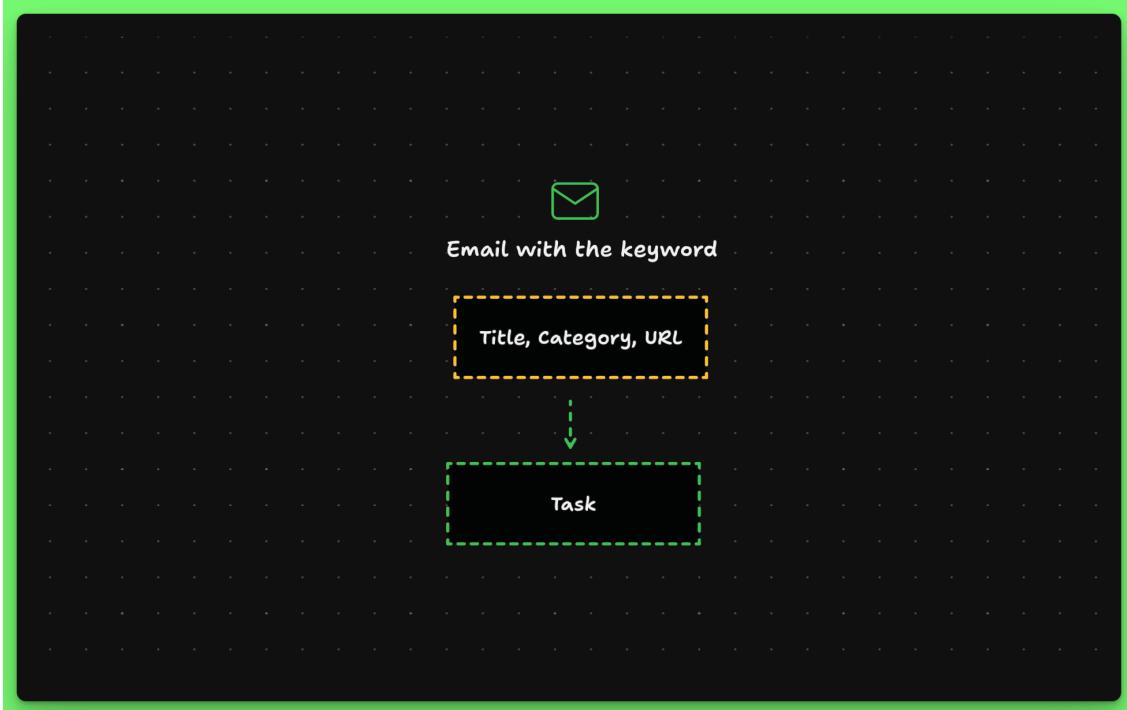
Jedna wiadomość, kilka podjętych działań, niemal 18 tysięcy tokenów i pół minuty na reakcję — z boku brzmi to, jak rozwiązanie, które nie ma sensu. Spójrzmy jednak na nie z nieco innej perspektywy.

Zarządzanie zadaniami **wymaga aktywnego działania ze strony osoby** obsługującej urządzenie z aplikacją taką jak Linear, Todoist czy ClickUp. Zadanie musi zostać nazwane, opisane, przypisane do kategorii, daty, priorytetu czy projektu — **w ten sposób pracuje większość z nas**.



Obsługa aplikacji niemal zawsze wymaga bezpośredniego zaangażowania człowieka, który kontroluje cały proces

Proces ten można częściowo zautomatyzować. Przykładowo, możemy za pomocą API monitorować skrzynkę e-mail oraz pojawiające się słowa kluczowe. Na ich podstawie możliwe jest ustawienie reguł przekierowujących wiadomość do wskazanej osoby lub nawet utworzenie nowego wpisu w aplikacji do zadań. **Tutaj zaangażowanie człowieka nie jest wymagane, lecz potrzebny jest zestaw programistycznie zdefiniowanych zasad (co nie zawsze jest możliwe)** — mówimy więc tutaj o automatyzacji procesu według ściśle określonych reguł.

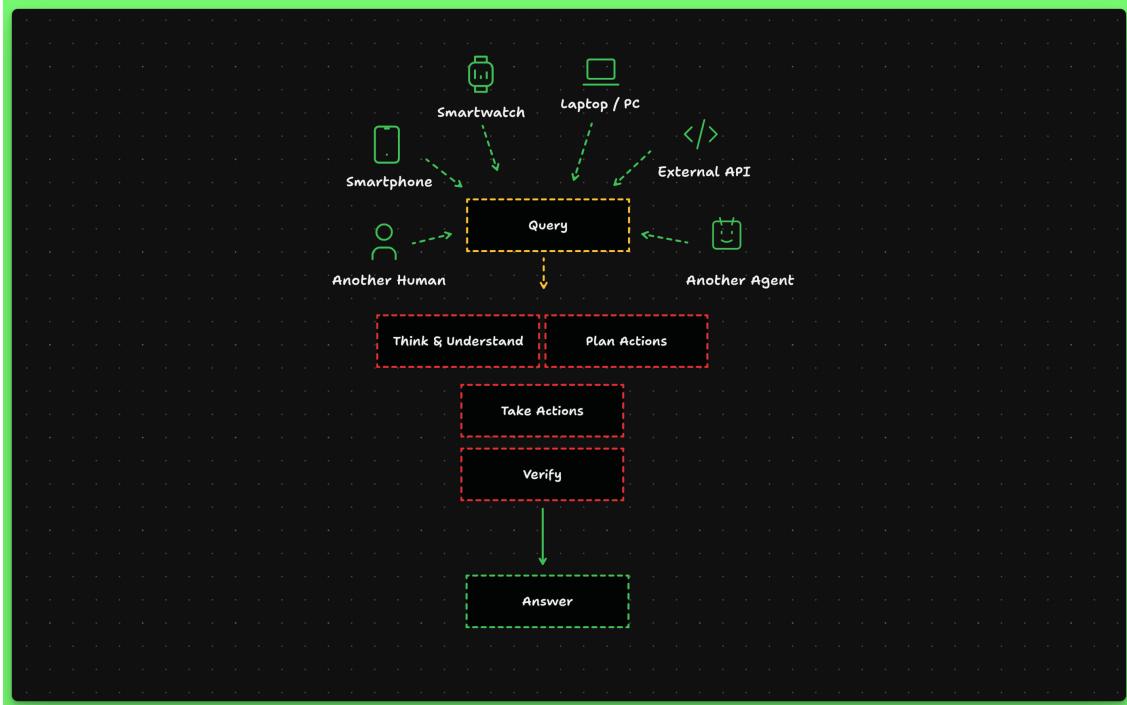


Procesy prywatne i biznesowe mogą być automatyzowane według sztywnych reguł,
np. dopasowania słów kluczowych

Teraz do budowy takiego systemu możemy wykorzystać duże modele językowe.
Dzięki nim możemy przyjmować różne formy treści pochodzące z różnych źródeł,
ponieważ za ich interpretację oraz podejmowane działania odpowiada model
połączony z logiką aplikacji.

Taki system może otrzymywać dane w formie zwykłych wiadomości pochodzących
od innej osoby, a także poprzez zdjęcie z telefonu, nagrania głosowe z zegarka czy
wiadomości przesłane na laptopie lub z zewnętrznego API. Co więcej, źródłem
danych może być nawet **inny agent AI!**

W poniższym schemacie widzimy, że wszystkie źródła danych generują **zapytanie**,
które jest interpretowane przez duży model językowy, na podstawie którego
generowany jest plan działań i podejmowane są akcje.



Automatyzacja w połączeniu z dużym modelem językowym pozwala na dość swobodne transformowanie różnych formatów treści, a także dynamiczne dostosowanie się do sytuacji

Szczególnie interesujący jest tutaj fakt, że podczas realizowania powyższej logiki, możliwe jest dynamiczne uzyskiwanie dostępu do informacji. Np. na podstawie wspomnianej nazwy projektu agent może wczytać dodatkowe informacje na jego temat, lub pobrać dane z Internetu, aby wzbogacić opis. Z kolei w sytuacji, gdy nie będzie w stanie sobie poradzić z zadaniem ... może poprosić człowieka o pomoc.

>> FILM VIDEO DOSTĘPNY JEST NA PLATFORMIE BRAVE <<

No i właśnie budowaniem takich rozwiązań, będziemy zajmować się przez najbliższe tygodnie, zatem — witaj w AI_devs 3!

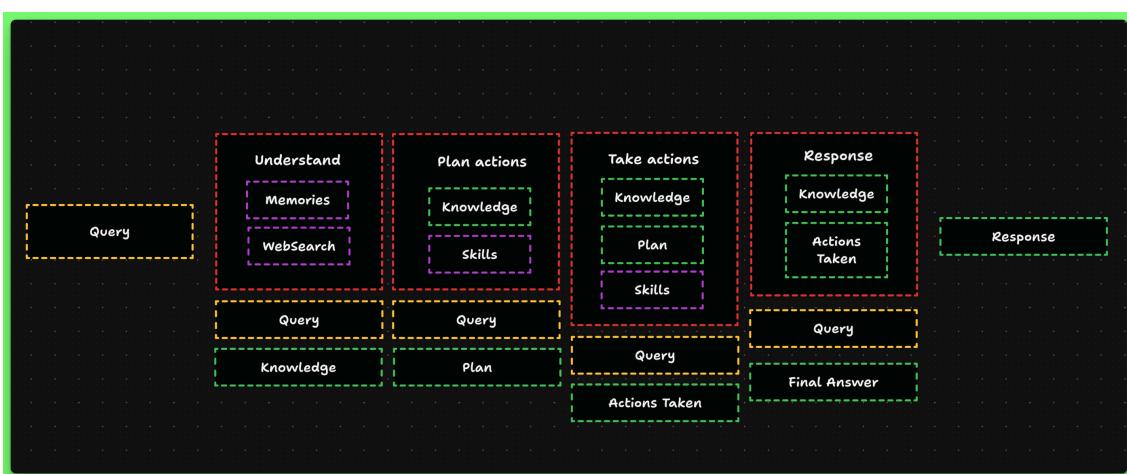
Połączenie z modelem, od praktycznej strony

Na tym etapie zakładam, że materiały wdrożeniowe do AI_devs 3 masz już za sobą lub wracasz do nas z poprzednich edycji. W obu przypadkach posiadasz przynajmniej bazową wiedzę na temat modeli językowych. Możemy więc przejść do praktycznych przykładów interakcji z modelami.

Zacznijmy od tego, że domyślnie interakcja z modelem polega na budowaniu tablicy messages zawierającej treść konwersacji połączoną z instrukcją systemową, czyli format ChatML. Jednak nas interesuje kilka dodatkowych kwestii.

Mianowicie fakt, że na wygenerowanie rezultatu w przypadku Agenta AI składa się wiele zapytań i wywołań funkcji. W przykładzie poniżej widzimy 4 etapy:

- **Zrozumienie:** Wymaga wczytania pamięci i/lub dostępu do Internetu. W ten sposób wykraczamy poza bazową wiedzę modelu i zyskujemy informacje przydatne na dalszych etapach. Można to określić jako etap "zastanawiania się" lub "analizy".
- **Plan działań:** Wymaga połączenia wcześniejszych "przemyśleń" połączonych z listą dostępnych narzędzi, umiejętności lub innych agentów. Na tej podstawie tworzona jest lista akcji, która ma być zrealizowana w dalszych krokach.
- **Podejmowanie działań:** Wymaga wiedzy, planu i dostępnych umiejętności, na podstawie których model decyduje o kolejnym kroku i gromadzi informacje zwrotne.
- **Odpowiedź:** Wymaga wiedzy oraz raportu z działań w celu wygenerowania ostatecznej odpowiedzi.



Już na tym etapie trzeba mieć na uwadze to, że powyższa interakcja, **nie musi uwzględniać zaangażowania ze strony człowieka** i może być realizowana "w tle" oraz trwać od kilku sekund do nawet kilku godzin. Może też być uruchamiana automatycznie według harmonogramu lub zewnętrznego zdarzenia.

Widzimy też wyraźnie, że nie mówimy tutaj już o prostym budowaniu konwersacji przez tablice messages, lecz nowej architekturze i wzorcach projektowania aplikacji. Co ciekawe, jest to programowanie które w ~80% przypomina klasyczne aplikacje, a LLM, Prompty czy narzędzia takie jak bazy wektorowe stanowią jedynie pewną część. Natomiast poza samym kodowaniem, zdecydowanie większą rolę odgrywa praca z danymi, różnymi formatami plików, organizacją baz danych czy strategiami wyszukiwania (tzw. retrieval).

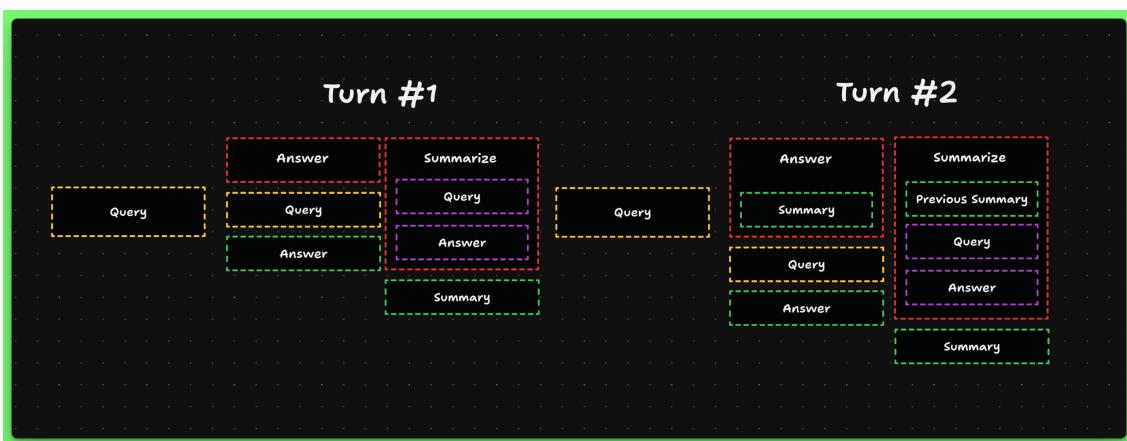
Wracając jednak do prowadzenia interakcji z modelem, to w przykładzie [thread](#) widzimy dość nietypowy, aczkolwiek bardzo przydatny sposób prowadzenia konwersacji. Zamiast każdorazowo przesyłać całą historię wiadomości do modelu, to stosujemy **podsumowanie** oraz jedynie **najnowszą wiadomość użytkownika**.

Dzięki temu, nie potrzebujemy kompletnej konwersacji, aby model zapamiętał kluczowe informacje takie jak imię użytkownika.

Aby uruchomić ten przykład, włącz serwer polecienniem bun thread i wykonaj zapytanie GET na adres localhost:3000/api/demo.

```
→ aidevs3 git:(main) ✘ bun thread
$ bun run thread/app.ts
Server running at http://localhost:3000. Listening for POST /api/chat requests
--- NEXT TURN ---
User: Hi! I'm Adam
Alice: Hi, Adam!
--- NEXT TURN ---
User: How are you?
Alice: Good, thank you. You?
--- NEXT TURN ---
User: Do you know my name?
Alice: Yes, Adam.
```

Schemat tej interakcji wygląda następująco: po udzieleniu pierwszej odpowiedzi generowane jest podsumowanie dotychczasowej rozmowy, które jest dołączane do promptu systemowego kolejnej tury. W ten sposób przekazujemy "skompresowany" wątek.



W wyniku takiej kompresji naturalnie tracimy część informacji. Nic jednak nie stoi na przeszkodzie, aby dodać mechanizm przeszukiwania wcześniejszych wątków na wypadek, gdyby podsumowanie było niewystarczające.

Przykład [thread](#) jest prosty, jednak doskonale obrazuje to, jak możemy manipulować przebiegiem konwersacji, a w rezultacie:

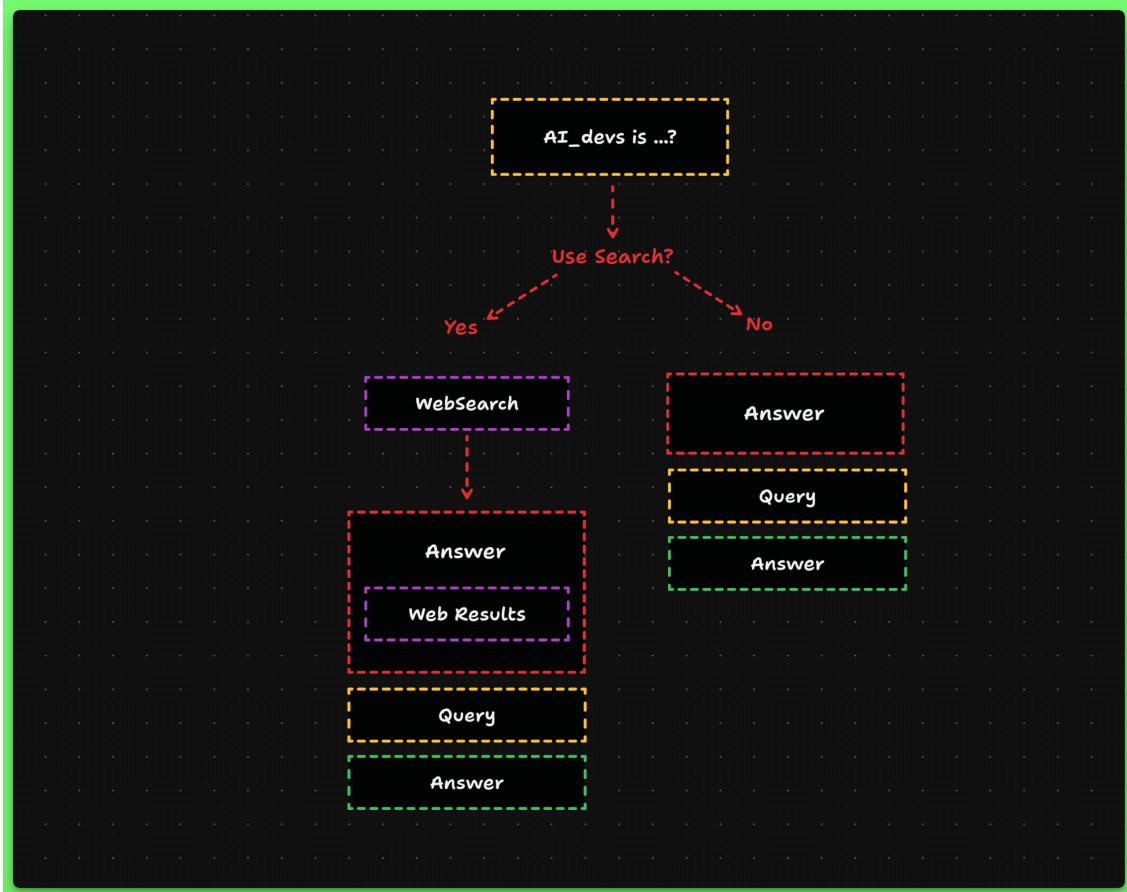
- Fakt, że do podsumowania zastosowaliśmy tańszy model, jest przykładem **optymalizacji kosztów**

- Dzięki podsumowaniu model przetwarza mniejszą ilość treści, **przez co jego uwaga jest bardziej skupiona na aktualnym zadaniu — WAŻNE!** w modelach klasy GPT-4o jest to bardzo istotna kwestia, wpływająca na skuteczność działania modelu
- Podsumowanie pozwala także uniknąć limitu okna kontekstu, co ma znaczenie w przypadku modeli Open Source, które mogą odpowiadać za wybrany element interakcji (np. anonimizację)
- Podsumowanie może być również wykorzystane w innych częściach logiki agenta, a także jako element interfejsu użytkownika lub raportu pracy agent
- W tym przypadku zastosowaliśmy podsumowanie, jednak ten sam schemat będziemy stosować np. przy rozpoznawaniu obrazu, dźwięku czy wideo. Tam również dodatkowe zapytania do modelu będą wykorzystywane jako kontekst promptu systemowego ## Rodzaje interakcji

Złożona logika agentów składa się z modułów oraz pojedynczych akcji. Trudno mówić o dobrze działającym systemie, jeśli nie zadbamy o detale zarówno po stronie promptów, jak i po stronie kodu. Dlatego na tym etapie przejdziemy przez kilka przykładów elementarnych akcji, takich jak podejmowanie decyzji, klasyfikacja, parsowanie, transformacja i ocena. Wykorzystamy także narzędzie PromptFoo, którego uruchomienie omówiłem w lekcji S00E02 — Prompt Engineering i o którym będziemy jeszcze mówić w dalszych lekcjach, a aktualnie wystarczy nam jego uruchomienie.

Przykładem pojedynczej akcji, może być **podejmowanie decyzji** przez model na podstawie dostępnych danych. Można to porównać do instrukcji warunkowej if lub switch. Różnica polega na elastyczności, kosztem deterministycznego rezultatu.

Poniższy scenariusz prezentuje logikę sprawdzającą czy do danego zapytania potrzebujemy skorzystać z wyszukiwarki internetowej. W pierwszej chwili taki scenariusz sugeruje zastosowanie Function Calling / Tool Use. Nie zawsze jednak będzie to oczywiste.



Mianowicie zakładamy tutaj, że natychmiast mamy komplet niezbędnych informacji potrzebnych do uruchomienia wyszukiwania, co zwykle nie jest prawdą. Połączenie np. z FireCrawl może wymagać pobrania listy dopuszczalnych domen, czy wygenerowania słów kluczowych na podstawie dodatkowego kontekstu wczytywanego z bazy danych.

W przykładzie [use_search](#) (wymaga zainstalowanego PromptFoo), mamy prompt odpowiadający za podejmowanie decyzji o zastosowaniu wyszukiwarki. Jego zadaniem jest wygenerowanie 0 lub 1 w celu klasyfikacji zapytania. Z tego powodu uwzględniałem w nim przykłady Few-Shot oraz zdefiniowałem zestaw zasad dopasowany do początkowych założeń. Następnie działanie takiego promptu jest automatycznie weryfikowane na kilkudziesięciu przykładach.

query	[openai:gpt-4o-2024-08-06] use_search/use_search.js
Who is John Wick	[PASS] 1
How are you???	[PASS] 0
What writing system is used for the text below? For which languages and in what part of the world is it used? What are the names of the peoples who speak these languages?	[PASS] 0
Okay, I need to better understand how the Svelte state management works	[PASS] 0
Look, I have a couple of questions for you: 1. What is the latest OpenAI multimodal models 2. What is the latest OpenAI multimodal models	[PASS] 1
Explain to me how the latest OpenAI multimodal models work	[PASS] 1
Where's my Tesla	[PASS] 0
I have a meeting with Michael at 2pm	[PASS] 0
The name of this actor	[PASS] 0
Play Queen on Spotify	[PASS] 0
Switch the music to the latest single of Nora En Pure	[PASS] 1
Can you explain how you work	[PASS] 0
Tell me about the history of quantum computing	[PASS] 0
Define epistemology	[PASS] 0

Podejmowanie decyzji przez LLM może uwzględniać potrzebę wybrania wielu opcji, a nie tylko jednej. Wówczas mówimy o klasyfikacji zapytania. Skoro jesteśmy już przy przeszukiwaniu Internetu, to pomocny będzie także prompt wybierający domeny, do których zawężymy wyszukiwanie. Jest to przydatne, ponieważ autonomiczne przeglądanie stron www szybko prowadzi do niskiej jakości źródeł czy serwisów, które wymagają logowania lub blokują dostęp do treści.

Warto więc wypisać sobie listę adresów i opisać je tak, aby LLM mógł zdecydować kiedy je uwzględnić, a kiedy nie. Prompt realizujący to zadanie znajduje się w przykładzie [pick_domains](#).

+ aidevss git:(main) ✘ bun pick_domains	[openai:gpt-4o-2024-08-06] pick_domains/pick_domains.js
\$ bun promptoo eval -c pick_domains/pick_domains.yaml	
(node:93501) [DEP0040] DeprecationWarning: The 'punycode' module is deprecated. Please use a userland alternative instead.	
(Use `node --trace-deprecation ...` to show where the warning was created)	
] 100% ETA: 0s 3/3 openai:gpt-4o-2024-08-06 "async func" queryWhat	
] 100% ETA: 0s 2/2 openai:gpt-4o-2024-08-06 "async func" queryConf	
] 100% ETA: 0s 2/2 openai:gpt-4o-2024-08-06 "async func" querySolv	
] 100% ETA: 0s 2/2 openai:gpt-4o-2024-08-06 "async func" queryFind	
query	
Tell me about John Wick and his movies	[PASS] { "_thoughts": "1. Movie details and character background available on Wikipedia. 2. User reviews and ratings on IMDb. 3. Video analyses and trailers on YouTube.", "queries": [{"q": "John Wick character background", "url": "wikipedia.org"...
Find an overment videos about functional programming on YouTube	[PASS] { "_thoughts": "1. User is interested in videos about functional programming. 2. YouTube is a suitable platform for video content. 3. No specific domain for 'overment' videos, so focus on YouTube.", "queries": [{"q": "Functional programmi...
How to configure Nginx on DigitalOcean and set up a PostgreSQL database for high performance	[PASS] { "_thoughts": "1. Involves configuring Nginx and PostgreSQL on DigitalOcean. 2. DigitalOcean for setup guides. 3. Nginx for configuration specifics. 4. PostgreSQL for performance tuning. 5. YouTube for visual tutorials.", "queries": [{"q...
Setup Kubernetes cluster on AWS	[PASS] { "_thoughts": "1. Task involves Kubernetes and AWS. 2. Use Kubernetes documentation for setup instructions. 3. AWS documentation for cloud-specific configurations. 4. YouTube for step-by-step video tutorials.", "queries": [{"q": "Setting...

W powyższym prompcie, w celu zwiększenia skuteczności, zastosowaliśmy także wariant Thought Generation, a konkretnie "zero-shot chain of thought". Mówimy tutaj o podniesieniu skuteczności, ponieważ w ten sposób dajemy LLM "czas na myślenie", które domyślnie widoczne jest w modelach o1. Dodatkowo fakt, że właściwość "_thoughts" jest generowana na początku, jest powiązany z faktem, że modele językowe są obecnie autoregresywne i treść tej pierwszej właściwości wpływa na treść kolejnych — zwiększając w ten sposób prawdopodobieństwo uzyskania oczekiwanych rezultatów.

Pozostając w temacie przeszukiwania Internetu, jesteśmy gotowi na wykonanie zapytania do wyszukiwarki. Jednak na tym etapie możemy uzyskać jedynie wyniki w formacie znanym z Google czy DuckDuckGo. Oznacza to, że nie będą to wystarczające informacje do udzielenia finalnej odpowiedzi, ale możemy na ich podstawie wskazać strony, które będziemy chcieli wczytać.

W przykładzie rate znajduje się prompt oceniający to, czy zwrócony wynik może zawierać interesujące nas informacje. Na podstawie zwróconych ocen wybierzemy te strony, których zawartość będziemy chcieli wczytać z pomocą np. FireCrawl.

[] 100% ETA: 0s 2/2 openai:gpt-4o-2024-08-06 "async func" query=Find [] 100% ETA: 0s 1/1 openai:gpt-4o-2024-08-06 "async func" query=List [] 100% ETA: 0s 1/1 openai:gpt-4o-2024-08-06 "async func" query=Find [] 100% ETA: 0s 1/1 openai:gpt-4o-2024-08-06 "async func" query=List		
context	query	[openai:gpt-4o-2024-08-06] rate/rate.js
Resource: https://brain.overment.com/ Title: brain.overment.com brain.overment.com Snippet: The most important thing to me is performance of the hardware, minimalistic design and my workflow. This is possible thanks to many Apps which are avail...	List Hardware mentioned on this page https://brain.overment.com/	[PASS] { "reason": "Snippet mentions 'performa... of the hardware' from query but does not specify any hardware. Page may contain hardware details.", "score": 0.5 }
Resource: https://brain.overment.com/ Title: Apps brain.overment.com Snippet: This is the latest (Q2 2024) list of apps I use: Arc – The best web browser I know. Alice – OpenAI / Anthropic / Ollama desktop experience. ia Writer – My favorite tex...	List Hardware mentioned on this page https://brain.overment.com/	[PASS] { "reason": "Snippet lists 'apps' and software, not 'hardware'. No indication of hardware information on the page.", "score": 0.1 }
Resource: https://youtube.com Title: How to Think Computationally About AI, the Universe and Everything Stephen Wolfram TED Link: https://www.youtube.com/watch?v=fLMZAHy... Snippet: Drawing on his decades-long mission to formulate the world ...	Find me Stephen Wolfram latest TED talk on youtube	[PASS] { "reason": "Snippet mentions 'Stephen Wolfram' and 'TED', matching query keywords 'Stephen Wolfram' and 'TED talk'. Likely contains relevant video", "score": 0.9 }

Automatyczny test promptu oceniającego to, jak istotny jest context z punktu widzenia zapytania

Zbierając to w całość, mamy już:

- Decyzję o tym czy wyszukiwarka internetu jest potrzebna

- Decyzyję o tym, jakie zapytania chcemy do niej skierować
- Możliwość filtrowania zwróconych wyników

Pozostaje nam więc już tylko wygenerowanie odpowiedzi na oryginalne pytanie na podstawie pobranych danych. Zobaczmy, jak możemy to wszystko połączyć w całość w przykładzie [websearch](#). Jego logika umożliwia zwykłą rozmowę z LLM, ale gdy wykryta zostanie konieczność skorzystania z wyszukiwarki, oryginalne zapytanie użytkownika zostaje wykorzystane do przeszukiwania sieci, co zresztą można zobaczyć na poniższym filmie.

>> FILM WIDEO DOSTĘPNY JEST NA PLATFORMIE BRAVE <<

```

26
27 app.post('/api/chat', async (req, res) => {
28   await fs.writeFile('prompt.md', '');
29
30   const { messages }: { messages: Message[] } = req.body;
31
32   try {
33     const latestUserMessage = messages.filter(m => m.role === 'user').pop();
34     if (!latestUserMessage) {
35       throw new Error('No user message found');
36     }
37
38     const shouldSearch = await webSearchService.isWebSearchNeeded(latestUserMessage.content as string);
39     let mergedResults: SearchResult[] = [];
40
41     if (shouldSearch) {
42       const { queries } = await webSearchService.generateQueries(latestUserMessage.content as string);
43       if (queries.length > 0) {
44         const searchResults = await webSearchService.searchWeb(queries);
45         const filteredResults = await webSearchService.scoreResults(searchResults, latestUserMessage.content as string);
46         const urlsToLoad = await webSearchService.selectResourcesToLoad(latestUserMessage.content as string, filteredResults);
47         const scrapedContent = await webSearchService.scrapeUrls(urlsToLoad);
48         mergedResults = filteredResults.map(result => {
49           const scrapedItem = scrapedContent.find(item => item.url === result.url);
50           return scrapedItem
51             ? { ...result, content: scrapedItem.content }
52             : result;
53         });
54       }
55     }
56
57     const promptWithResults = answerPrompt(mergedResults);
58     const allMessages: ChatCompletionMessageParam[] = [
59       { role: 'system', content: promptWithResults, name: 'Alice' },
60       ...messages as ChatCompletionMessageParam[]
61     ];
62
63     const completion = await openaiService.completion(allMessages, "gpt-4o", false);
64
65     return res.json(completion);
66   } catch (error) {
67     console.error('Error in chat processing:', error);
68     res.status(500).json({ error: 'An error occurred while processing your request' });
69   }
70 }
71 );

```

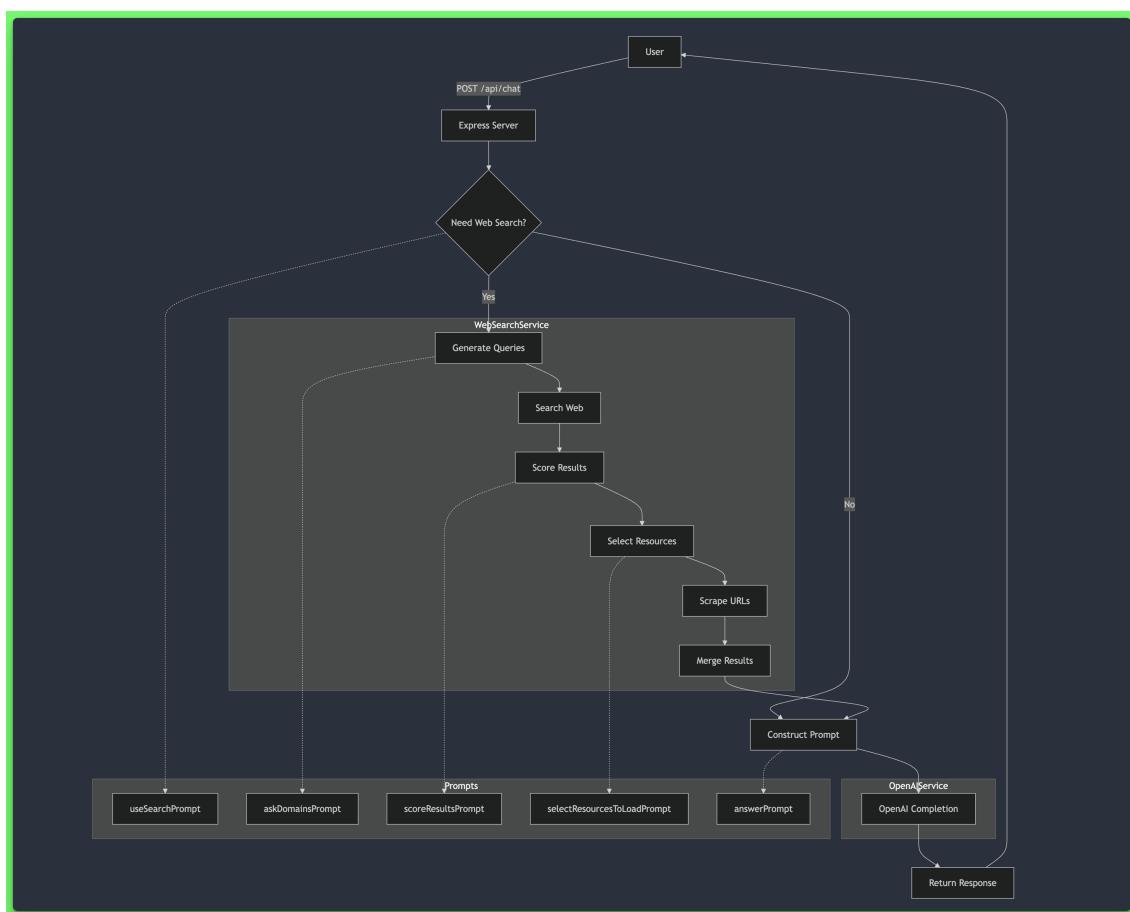
Przykład kodu łączącego duży model z wyszukiwarką internetową

Architektura aplikacji

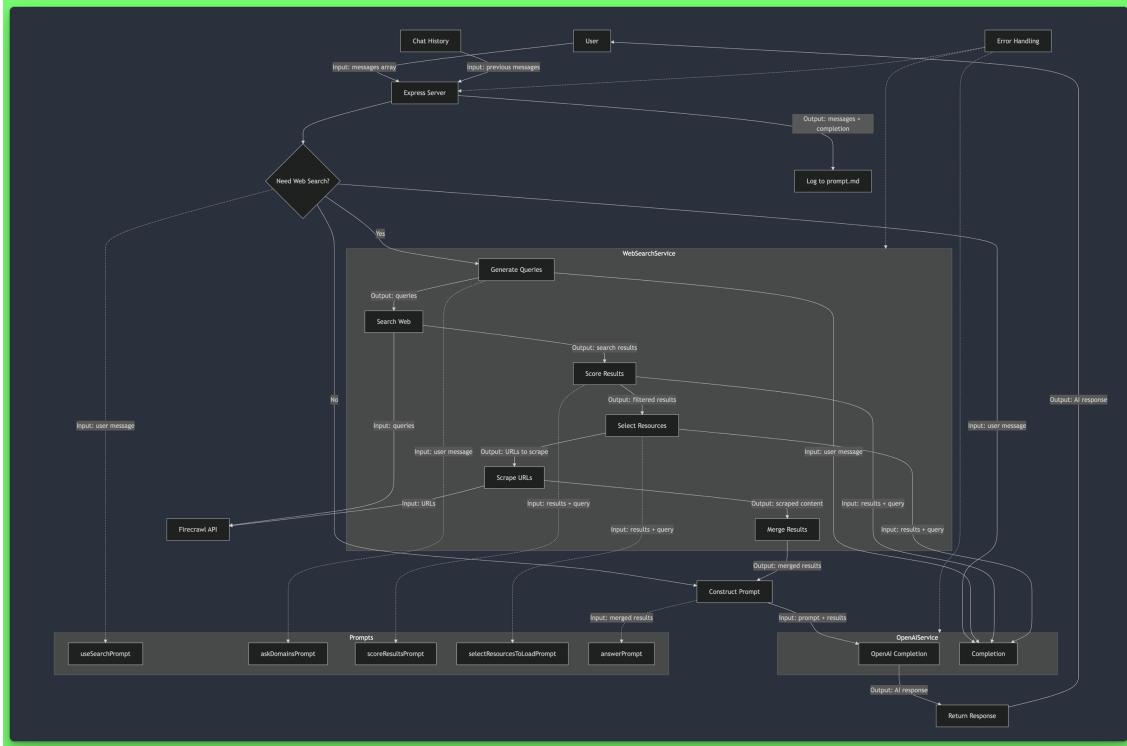
Patrząc nawet na przykład [websearch](#), można zauważyc, że faktycznie ~80% kodu przypomina klasyczną aplikację. Jednak zgodnie z tym, co omawialiśmy w lekcji S00E04 — Programowanie, w kodzie zaczyna pojawiać się język naturalny oraz elementy, które do tej pory mogły odgrywać nieco mniejszą rolę w zależności od projektu.

Poza strukturą katalogów, podziałem odpowiedzialności, architekturą bazy danych czy samym stackiem technologicznym, pod uwagę musimy wziąć także rolę dużych modeli językowych oraz promptów. Nie chodzi tutaj wyłącznie o wybór modelu, hostingu czy napisaniu instrukcji, ale przede wszystkim o sposób przepływu danych.

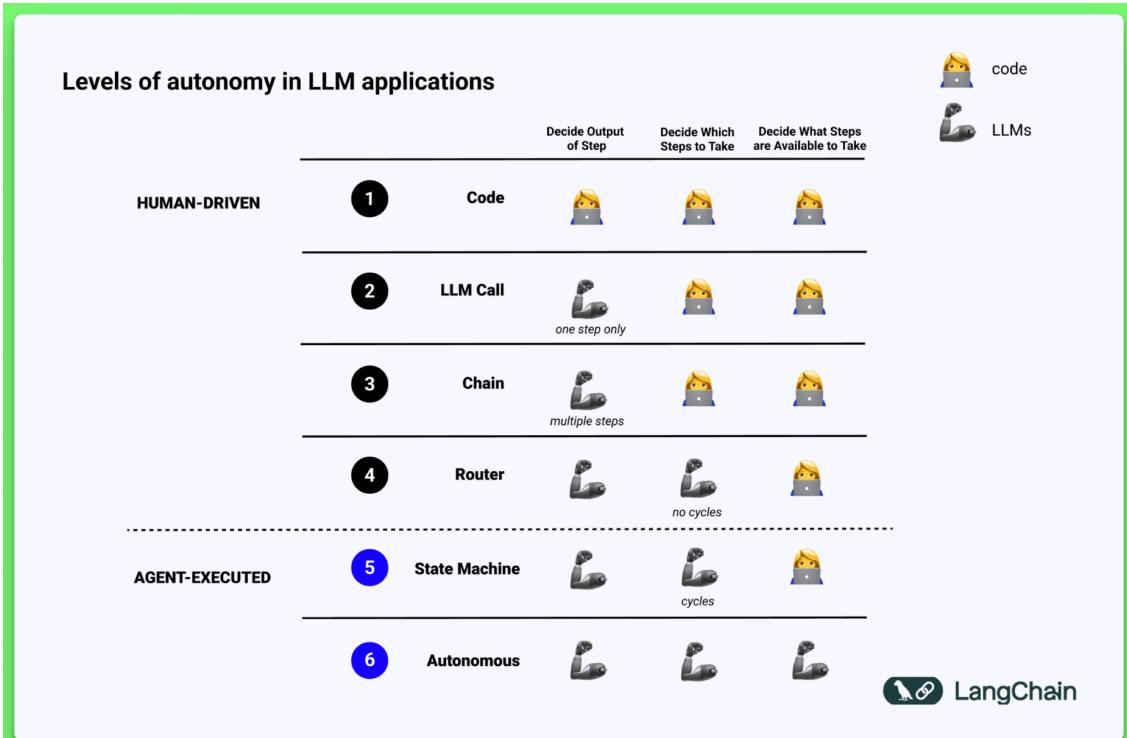
Jeśli spojrzymy teraz na wizualizację przykładu [websearch](#), to jasno widzimy, że działanie kolejnych promptów jest uzależnione od rezultatów poprzednich. Choć każdy z nich budujemy indywidualnie, to robiąc to, musimy brać pod uwagę dane na których będzie pracować, oraz to w jaki sposób generowane przez niego dane będą wykorzystane później.



Nieco bardziej rozbudowana wizualizacja pokazuje te zależności nieco wyraźniej. I tak nie jest to wszystko, bo mamy tutaj do czynienia z łańcuchem promptów i akcjami następującymi po sobie, a nie zawsze tak będzie.



Na blogu LangChain można przeczytać [o podstawach architektury kognitywnej](#), gdzie uwzględniony jest podział na Code, LLM Call, Chain, Router, a także State Machine i w pełni autonomiczne systemy o których będziemy jeszcze mówić.



Tymczasem spróbujmy spojrzeć na to z szerokiej perspektywy, uwzględniając elementy, które w tej chwili pominęliśmy w celu uniknięcia dużej złożoności.

- **Baza danych (np. PostgreSQL):** teraz nie tylko historia konwersacji rozpoczyna się za każdym razem od nowa. Treść wyników wyszukiwania oraz zawartość wczytanych stron również znikają po zakończeniu żądania. Zatem jeśli w kolejnej wiadomości użytkownik zada pytanie pogłębiające, będziemy musieli ponownie wczytywać te same dane. Widzimy więc, że **będziemy chcieli zapisać zarówno historię rozmowy, jak i kontekst wykorzystany do ich generowania**. Ogólny mechanizm widoczny jest we wcześniejszym przykładzie [thread](#), ale uwzględniał on wyłącznie treść rozmowy, bez dodatkowego kontekstu
- **Silnik wyszukiwania (np. Qdrant):** zapisując wspomniane dane, szybko dojdziemy do momentu gdy wczytanie ich wszystkich do kontekstu stanie się nieopłacalne lub wprost niemożliwe. Wówczas kluczowe będzie ich skuteczne odszukiwanie. Musimy zatem pomyśleć o tym, jak je skutecznie zorganizować i opisać, a potem przeszukiwać oraz przekazywać do modelu.
- **Zarządzanie stanem:** podobnie jak w przypadku klasycznych aplikacji, tutaj także do gry wchodzi zarządzanie stanem. Jednak tutaj przechowywane dane będą obejmować historię promptów czy historię uruchomionych narzędzi wraz z informacją zwrotną.
- **API:** w przypadku [websearch](#) mamy do czynienia tylko z dwoma narzędziami (web search i web scrapping), jednak zwykle będzie ich znacznie więcej. Każde z nich musi zostać zbudowane tak, aby LLM mógł się nim posługiwać, rozumieć odpowiedzi oraz obsługiwać błędy
- **Ewaluacja promptów:** Patrząc na powyższy schemat, staje się jasne, dlaczego wcześniej przechodziliśmy przez PromptFoo oraz dlaczego będziemy modyfikować prompty, testując je automatycznie na wybranych zestawach testowych.
- **Wersjonowanie i kopie zapasowe:** wersjonowanie odnosi się już nie tylko do historii kodu oraz promptów, lecz także do zmian wprowadzanych przez model. Przykładowo, agent zarządzający listą zadań może przypadkowo zmodyfikować wpisy, których nie chcemy edytować, i musimy mieć łatwy sposób ich przywrócenia.
- **Kontrola uprawnień:** w przykładzie [websearch](#) programistycznie ograniczyłem listę domen z którymi LLM może się skontaktować. W podobny sposób będziemy określać uprawnienia modelu w celu zwiększenia stabilności aplikacji.
- **Monitorowanie aplikacji:** w przykładzie [websearch](#) historię wykonanych zapytań zapisałem w pliku markdown. Naturalnie, nie będzie to wystarczające w produkcyjnych aplikacjach, gdzie będziemy korzystać z LangFuse czy podobnych rozwiązań do zaawansowanego monitorowania.
- **Asynchroniczność:** narzędzie [websearch](#) pokazuje, że LLM może działać w tle, co wydłuża czas reakcji. W takim przypadku sensowne jest uruchamianie skryptu "w tle" lub utworzenie kolejki, po której wykonaniu użytkownik otrzyma powiadomienie lub e-mail z informacją o zakończeniu zadania.
- **Interfejs:** narzędzie [websearch](#) może być wykorzystane w interfejsie czatu, co pokazałem na filmie. Jednak równie dobrze mógłby to być formularz umożliwiający dodanie listy

adresów oraz związane z nimi zadania (np. "pobierz najnowszy artykuł") wraz z harmonogramem uruchomienia.

Wszystkimi z wyżej wymienionych punktów będziemy zajmować się w dalszych lekcjach, ale nie wszystkie będą wymagane za każdym razem. Będziemy tworzyć zarówno proste narzędzia odpowiedzialne za nieskomplikowane akcje, jak i rozbudowane rozwiązania wspierające złożone procesy.

Optymalizacja skuteczności

Instrukcje z przykładów [pick_domains](#), [use_search](#) czy [rate](#) zawierają od kilku do kilkunastu przykładów Few-Shot. W niektórych przypadkach może być ich nawet kilkadziesiąt czy kilkaset i wówczas mówimy o "in-context learningu w oparciu o przykłady 'many-shot'" o których możemy przeczytać w [Many-Shot In Context Learning](#).

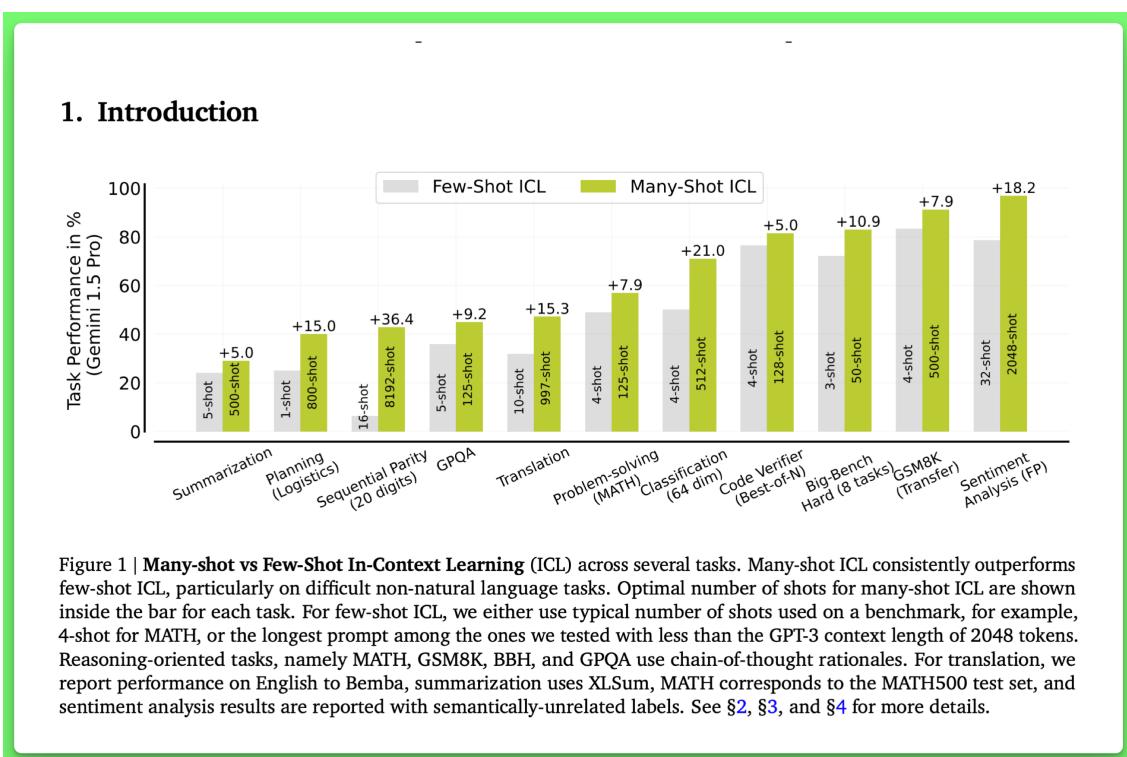


Figure 1 | **Many-shot vs Few-Shot In-Context Learning (ICL)** across several tasks. Many-shot ICL consistently outperforms few-shot ICL, particularly on difficult non-natural language tasks. Optimal number of shots for many-shot ICL are shown inside the bar for each task. For few-shot ICL, we either use typical number of shots used on a benchmark, for example, 4-shot for MATH, or the longest prompt among the ones we tested with less than the GPT-3 context length of 2048 tokens. Reasoning-oriented tasks, namely MATH, GSM8K, BBH, and GPQA use chain-of-thought rationales. For translation, we report performance on English to Bemba, summarization uses XLSum, MATH corresponds to the MATH500 test set, and sentiment analysis results are reported with semantically-unrelated labels. See §2, §3, and §4 for more details.

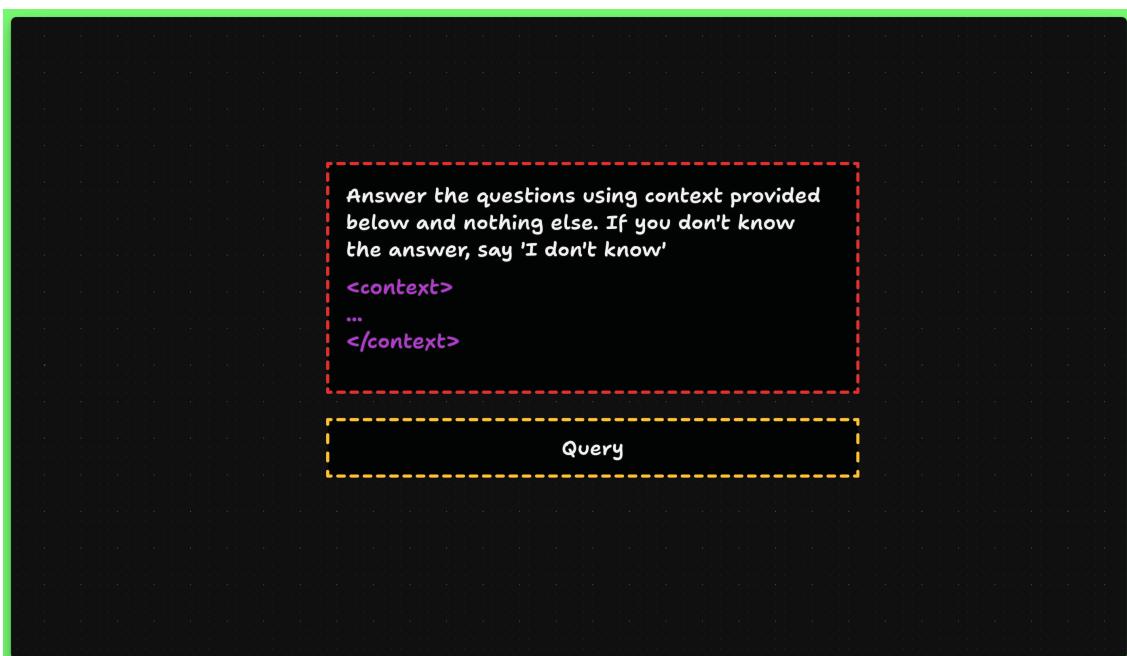
Uwzględnienie przykładów jest pierwszą techniką, którą powinniśmy brać pod uwagę przy optymalizacji skuteczności promptu. Poprzez prezentowanie oczekiwanej zachowania w ten sposób, możemy wzmacnić treść głównej instrukcji.

Samym projektowaniem przykładów będziemy zajmować się w dalszych lekcjach, natomiast na teraz musisz wiedzieć, że:

- Przykłady zwykle mają formę par prezentujących dane wejściowe (wiadomość użytkownika) oraz dane wyjściowe (odpowiedź modelu)
- Liczba przykładów zwykle nie przekracza ~3 - 40 par
- Przykłady powinny prezentować oczekiwane zachowanie i być zróżnicowane oraz uwzględniać sytuacje brzegowe (np. takie w których zachowanie modelu ma być inne niż oczekiwane)
- Przykłady muszą być dobrane starannie, lecz do ich generowania możemy wykorzystać pomoc ze strony modelu
- Przykłady docelowo mogą być wykorzystane na potrzeby Fine-Tuning, a ich warianty na potrzeby automatycznych testów
- Duża liczba przykładów może być połączona z mechanizmem cache'owania w celu optymalizacji kosztów oraz wydajności

Technikami dobierania przykładów i pracy z nimi, będziemy zajmować się w dalszej części AI_devs 3. Tymczasem Few-Shot możesz zapamiętać jako nieodłączny element praktycznie każdego promptu. ## Podstawy pamięci długoterminowej

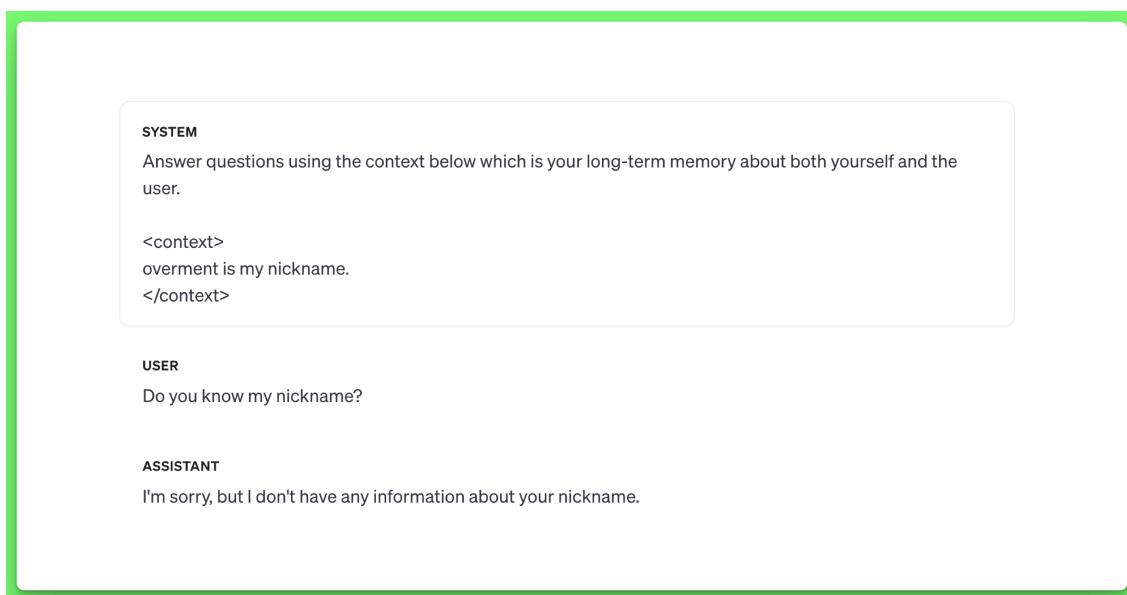
Wczytywanie treści z wyszukiwarki oraz stron www do promptu w celu odpowiadania na pytania na ich podstawie to przykład Retrieval-Augmented Generation. Tutaj z pomocą FireCrawl rozszerzyliśmy bazową wiedzę modelu dokładnie w taki sposób, jak będziemy robić to w przypadku baz wiedzy czy pamięci tymczasowej oraz długoterminowej agenta. Jest to prawdopodobnie znany Ci już schemat, widoczny poniżej, w którym wskazujemy modelowi treść, którą ma wykorzystać przy generowaniu odpowiedzi.



Analogicznie będziemy wczytywać dane z plików, baz danych czy zewnętrznych usług. Przykład [websearch](#) pokazał nam jednak, że na odszukanie informacji będzie składać się szereg dodatkowych kroków, związanych z parafrazą zapytania, generowaniem dodatkowych zapytań (tzw. Self-Querying), ocenianiem wyników (tzw. Re-rank) i ich filtrowaniem.

Należy mieć tutaj na uwadze fakt, że podłączanie zewnętrznych źródeł wiedzy do naszego systemu może mieć bardzo negatywny wpływ na jego działanie. To właśnie z tego powodu ograniczyłem listę domen dla FireCrawl, ale takich sytuacji jest więcej. Chociażby wczytywanie dokumentu PDF może wiązać się z utratą formatowania, co zaburzy zrozumienie jego zawartości.

Problemy z formatowaniem to także nie wszystko, ponieważ stale musimy mieć na uwadze ograniczoną wiedzę LLM na temat naszego kontekstu. Przykładowo jeśli powiemy "Zapamiętaj, że overment to mój nickname", to system powinien zapamiętać "Nickname Adama to overment". W przeciwnym razie w przyszłości może uznać, że 'overment' to jego nickname, czego przykład mamy poniżej.



O pamięci długoterminowej dla modelu będziemy jeszcze mówić w module trzecim AI_devs 3. Na ten moment zapamiętaj, że:

- Jakość wypowiedzi modelu zależy od promptu, ale także od dostarczonych danych
- Instrukcja powinna zawierać informacje na temat tego, jak model powinien wykorzystywać kontekst w swoich wypowiedziach
- Jeden prompt może zawierać wiele zewnętrznych kontekstów, jednak powinny być one wyraźnie od siebie oddzielone
- Model powinien posiadać instrukcję u zachowaniu w sytuacji, gdy dostarczony kontekst jest niewystarczający do udzielenia odpowiedzi

- Musimy zadbać nie tylko o jakość źródeł dostarczanych informacji, ale także o sposób ich przechowania i dostarczenia do modelu. Wspomniana wyżej parafraza wspomnienia pokazuje, że zawsze musimy zadawać sobie pytanie: **Jak model będzie wykorzystywać dostarczoną wiedzę?**

Podsumowanie

Niniejsza lekcja to przedsmak tego, czym będziemy zajmować się w nadchodzących tygodniach. Jej celem było pokazanie szerokiej perspektywy na temat aplikacji wykorzystujących LLM, a przede wszystkim tego, że w dużym stopniu są one budowane tak samo, jak oprogramowanie, które tworzymy na co dzień.

To właśnie z tego powodu, jednym z wymagań AI_devs 3 była znajomość przynajmniej jednego języka programowania. Co prawda nie wszystkie z omawianych elementów będziemy wdrażać samodzielnie i nie musimy posiadać doświadczenia w budowie baz danych czy optymalizacji silników wyszukiwania. Pomimo tego w nadchodzących tygodniach będziemy mieć kontakt z najróżniejszymi zagadnieniami, które mogą być dla Ciebie zupełnie nowe. Potraktuj to więc jako możliwość doświadczenia szerokiej perspektywy rozwoju aplikacji, a najwięcej uwagi skieruj na obszary, które Ciebie dotyczą (np. front-end, back-end czy bazy danych).

Po dzisiejszej lekcji spróbuj przynajmniej uruchomić przykład [websearch](#) i zadać mu kilka pytań w celu sprawdzenia jak się w nich odnajduje. Istnieje dość duże prawdopodobieństwo, że nie odpowie skutecznie na Twoje pytania — zastanów się wtedy dlaczego tak się dzieje. Przejdz przez prompty z pliku prompts.ts oraz sprawdź jak skutecznie FireCrawl radzi sobie z wczytywaniem treści stron, z którymi chcesz pracować.

Możesz także poświęcić chwilę na pracę z PromptFoo, którego podstawową konfigurację omawiałem w materiale wdrożeniowym i lekcji S00E02 — Prompt Engineering. Do pracy z tym narzędziem wykorzystaj Cursor IDE z wczytaną dokumentacją, co ułatwi generowanie plików konfiguracyjnych oraz szybsze zrozumienie tego rozwiązania.

Zamykając klamrą dzisiejszą lekcję, to już na tym etapie powinno być dla Ciebie zrozumiałe to, że LLM w kodzie aplikacji pozwala na sprawne przetwarzanie języka naturalnego, a także różnych formatów danych (np. audio czy obrazu). Daje to nam nowe możliwości, ale nadal fundamentem rozwoju aplikacji pozostaje Twoja wiedza oraz doświadczenie.