ChatGPT

Dogłębna analiza: Cursor AI, ramy rozumowania LLM, GraphRAG i wykrywanie luk w wiedzy

Cursor AI – wewnętrzne mechanizmy i unikalne możliwości

Integracja z kodem i kontekst całego projektu: Cursor AI to inteligentny edytor kodu oparty na VS Code, który jest głęboko zintegrowany z całą bazą kodu programisty [1]. Oznacza to, że model językowy ma dostęp do pełnego kontekstu projektu – wszystkich plików i symboli – zamiast tylko fragmentów otwartego pliku, co odróżnia go od typowych asystentów jak ChatGPT czy Copilot [2] [3]. Dzięki temu Cursor może **świadomie proponować zmiany obejmujące wiele plików jednocześnie**. Na przykład, jeśli użytkownik zażąda refaktoryzacji funkcji, Cursor potrafi nie tylko zmienić kod w miejscu definicji, ale również **zasugerować dostosowania we wszystkich miejscach, gdzie ta funkcja jest używana** [4]. Podobnie, podczas implementacji nowych funkcjonalności, narzędzie śledzi zależności i **automatycznie tworzy nowe pliki czy klasy** zgodnie z architekturą projektu, bazując na stylu istniejącego kodu [3].

Refaktoryzacja i edycja w języku naturalnym: Jedną z wyróżniających cech Cursor AI jest możliwość **wydawania poleceń refaktoryzacji i edycji kodu w języku naturalnym**. Użytkownik może zaznaczyć fragment kodu i sformułować polecenie typu: *„Wyodrębnij ten kod do nowej klasy i przenieś do pliku* `utils/cleaner.py` *”* lub *„Podziel tę metodę na mniejsze oraz przenieś logikę logowania do dekoratora”*. Cursor zrozumie intencję i **automatycznie zastosuje odpowiednie zmiany w kodzie** [5], prezentując je w formie diff (zaznaczając usunięte i dodane linie do akceptacji) [6]. To naturalnojęzykowe sterowanie edycją znacząco przyspiesza refaktoryzację – **most między zamiarem a implementacją zostaje skrócony** dzięki temu, że Cursor *„wie, co masz na myśli i potrafi to odzwierciedlić w kodzie”* [7].

Wspomagane debugowanie: Cursor AI pełni rolę zawsze dostępnego partnera do debugowania. Gdy programista napotka błąd lub wyjątek, może wkleić komunikat błędu do okna czatu Cursor i otrzymać analizę oraz propozycję poprawek. Co więcej, **Cursor potrafi automatycznie nasłuchiwać błędów z konsoli i proponować poprawki kodu** bez wyraźnego pytania [3]. Takie *AI-assisted debugging* wykorzystuje pełny kontekst kodu – model rozumie, w jakim module wystąpił błąd i może wskazać poprawkę w odpowiadającym pliku. Przykładowo, użytkownik uruchamia aplikację i otrzymuje wyjątek typu *ModuleNotFound*. Cursor, znając strukturę projektu, może **automatycznie dodać brakujący import lub poprawić ścieżkę modułu**, często zanim programista sam zidentyfikuje przyczynę. Użytkownicy zauważają, że *“nigdy wcześniej nie widzieli debugowania na taką skalę”*, gdzie AI poprawia liczne błędy typów i zależności jednocześnie [8] [9]. Dzięki temu debugowanie staje się iteracyjne i szybkie – **Cursor pełni rolę „gumowej kaczki”** i mentora, który nie tylko wykrywa problemy, ale od razu sugeruje rozwiązania.

Generowanie testów jednostkowych: W odróżnieniu od prostych podpowiedzi testów, Cursor AI **analizuje logikę funkcji i tworzy do niej dopasowane testy jednostkowe**. Generowane testy nie są losowe – narzędzie uwzględnia edge-case’y, obsługę wyjątków oraz interakcje między klasami [10]. Na przykład, jeśli funkcja operuje na danych wejściowych różnych typów, Cursor wygeneruje testy pokrywające zarówno przypadki prawidłowe, jak i błędne. Co ważne, testy te są od razu wykonywalne – programista może je zaakceptować i uruchomić w projekcie. **Takie inteligentne generowanie testów przyspiesza TDD/BDD**, bo programista otrzymuje pełny zestaw testów jako punkt wyjścia, który można

dalej doprecyzować. To przewyższa proste szablony, gdyż Cursor dzięki kontekstowi całej bazy kodu *„rozumie”* zależności i efekty działania funkcji, pisząc testy jak doświadczony inżynier jakości [10].

Zaawansowane funkcje asystenta kodu: Poza refaktoryzacją i testami, Cursor oferuje szereg innych unikalnych możliwości: - **Inteligentna autouzupełnianie (Tab-complete):** W miarę pisania kodu, narzędzie proponuje kolejne linie, które **wydają się być dokładnie tym, co napisałby programista** – wynika to z czytania kontekstu wokół kursora i całego pliku [11]. Dzięki temu uzupełnienie nie jest tylko syntaksą, ale uwzględnia nazwy zmiennych, styl kodowania i wcześniejsze wzorce w projekcie. - **Generowanie dokumentacji i commitów:** Cursor potrafi automatycznie generować docstringi i podsumowania modułów, czerpiąc z nazewnictwa i zależności w kodzie, aby opisy były rzeczowe i użyteczne [12]. Podczas tworzenia commitów, narzędzie **analizuje diff zmian i proponuje klarowny komunikat commit** zgodny ze standardami (np. Conventional Commits), zamiast banalnego “fix bug” [13]. - **Wieloplikiwe zapytania i agenty:** Dzięki globalnemu zrozumieniu kodu, można zadać Cursorowi pytanie w stylu: *„Gdzie w projekcie obsługujemy wygasanie tokenu?”*, a on znajdzie i wskaże relevantny fragment kodu [14]. Cursor oferuje też **tryb agentów**, które pełnią rolę wirtualnych pomocników wykonujących określone zadania – np. *agent przeszukujący TODO*, *agent refaktoryzujący projekt według wytycznych* czy *agent przeglądający merge requesty w poszukiwaniu ryzykownych zmian* [14] [15]. To tak, jakby mieć zespół młodszych programistów wykonujących żmudne prace – zawsze czujnych i niezmordowanych.

LLMOps w Cursor AI (wersjonowanie promptów, testy A/B, monitoring): Twórcy Cursor zdają sobie sprawę, że efektywne użycie AI wymaga ciągłego doskonalenia promptów i monitorowania działania modelu. W środowisku profesjonalnym **integruje się więc Cursor z narzędziami LLMOps** – przykładem jest platforma Opik od Comet, która umożliwia **logowanie wszystkich interakcji AI, monitorowanie jakości odpowiedzi oraz eksperymentowanie z różnymi podejściami** [16]. Podczas pracy z Cursorem można zachowywać historię promptów (np. kolejnych poleceń refaktoryzacji) i ich efektów, co stanowi formę wersjonowania – programista widzi, jakie zmiany w sformułowaniu polecenia dały lepszy efekt. Dzięki logom i tzw. *trace'om* z Opik, **możliwe jest porównywanie różnych wariantów promptu (A/B)**: np. czy lepszy rezultat daje polecenie *„Popraw wydajność tej funkcji”* czy bardziej szczegółowe *„Zoptymalizuj pętlę w tej funkcji, używając list comprehension”*. Wszystkie takie próby są zapisywane, co pozwala na ich późniejszą analizę i uczy najlepszych praktyk. Ponadto, integracja z narzędziami monitorującymi daje **wgląd w działanie modelu w czasie rzeczywistym** – deweloper widzi, **jakie dokładnie prompty są wysyłane do modelu „pod maską” oraz jakie odpowiedzi generuje** [17]. To zwiększa przejrzystość i zaufanie: można zauważyć, gdzie AI potencjalnie błądzi lub generuje niepożądany kod, a następnie skorygować prompt lub dostosować parametry (np. zmniejszyć temperaturę modelu dla bardziej deterministycznych poprawek). Tego typu monitoring wydajności (np. czas generacji, trafność podpowiedzi, liczba iteracji do rozwiązania problemu) pozwala firmom **oceniać realny wpływ narzędzia na produktywność** oraz szybko identyfikować regresje po aktualizacjach modelu.

Przykłady zaawansowanych komend i promptów: Cursor posiada swoje specyficzne polecenia i formaty. Poza wspomnianymi komendami w stylu *„Refaktoryzuj ten kod pod kątem czytelności”* czy *„Wyjaśnij działanie zaznaczonej funkcji”*, warto wspomnieć o funkcjonalności `.cursorrules`. Jest to plik konfiguracyjny (w formacie JSON), pozwalający zdefiniować reguły i persony AI w edytorze [18]. Za jego pomocą użytkownik może tworzyć **własne profile asystenta** – np. persona “Luna” z forum społeczności Cursor to AI, która specjalizuje się w debugowaniu z humorem (poprzez `.cursorrules` ustalono styl jej wypowiedzi i skrypty działania) [19] [20]. Tego rodzaju mechanizm umożliwia zaawansowane **dostosowanie promptów systemowych**: użytkownik określa reguły, według których AI reaguje na pewne sytuacje (np. zawsze prosi o potwierdzenie usunięcia dużego bloku kodu, lub stosuje konkretny styl formatowania). Dzięki temu Cursor staje się bardziej przewidywalny i dopasowany do potrzeb zespołu.

Podsumowanie (case study): Wyobraźmy sobie praktyczny scenariusz: programista ma istniejący projekt i chce wprowadzić nową funkcję. Zamiast ręcznie dodawać pliki i pisać kod od zera, korzysta z Cursor AI. **Krok 1:** Opisuje w czacie: *"Dodaj moduł logowania błędów: każda funkcja powinna logować wyjątki do pliku* `errors.log`*."* – Cursor generuje nowy plik modułu logowania i inserty logujące w istniejących funkcjach. **Krok 2:** Programista widzi, że testy nie przechodzą – przekazuje log błędów do Cursor, który **automatycznie poprawia importy i typy** (błyskawiczna naprawa kilku plików). **Krok 3:** Programista prosi: *"Wygeneruj testy jednostkowe do modułu logowania"* – Cursor tworzy testy, które od razu wykrywają brakujący przypadek (np. brak praw dostępu do pliku). **Krok 4:** Cursor sugeruje poprawkę – dodanie obsługi wyjątku przy braku dostępu. W efekcie w ciągu kilkunastu minut narzędzie pomogło zaimplementować funkcję, napisać do niej testy i zdebugować problemy. Taki case study pokazuje przewagę Cursor AI nad klasycznymi metodami: **skupienie się dewelopera przesuwa z "pisania kodu" na "nadzorowanie i korygowanie AI", co dramatycznie zwiększa wydajność**, gdyż AI wykonuje większość żmudnej pracy zgodnie z intencjami programisty [21] [1].

Praktyczne zastosowania ram rozumowania LLM: Tree of Thoughts, Graph of Thoughts, Adaptive GoT

Wyzwanie złożonego rozumowania: Standardowe podejścia do generowania odpowiedzi (w tym klasyczne *Chain-of-Thought*, czyli liniowe "krok po kroku") mogą okazać się niewystarczające przy zadaniach wymagających planowania, eksploracji wielu opcji czy łączenia odległych faktów. Dlatego powstały ramy rozumowania jak **Tree of Thoughts (ToT)**, **Graph of Thoughts (GoT)** oraz ich dynamiczna odmiana **Adaptive Graph of Thoughts (AGoT)**, które mają rozszerzyć zdolności LLM do *świadomego* rozwiązywania problemów. W praktyce inżynierii promptów oznacza to, że możemy **strukturą i treścią zapytań "wymusić" na modelu określony styl rozumowania** i wygenerować bardziej wyczerpujące odpowiedzi.

Tree of Thoughts (Drzewo Myśli) – eksploracja wielu ścieżek

Idea: Tree of Thoughts to koncepcja zaproponowana m.in. przez Yao i współautorów (2023), uogólniająca chain-of-thought na strukturę drzewa poszukiwań [22]. Zamiast jednej sekwencji rozumowania, model generuje **drzewo możliwych myśli (stanów pośrednich)** zmierzających do rozwiązania problemu [23]. Na każdym etapie może powstawać kilka kandydatów kolejnego kroku, z których najlepsze są wybierane na podstawie samo-ewaluacji modelu (np. oceny "czy ten kierunek rokuje osiągnięcie celu?") [24]. Pozwala to na **lookahead i backtracking** – jeśli obrana gałąź okaże się ślepa, można wrócić do wcześniejszego rozgałęzienia i podążyć alternatywną ścieżką [25] [26]. To podejście przypomina rozwiązywanie łamigłówek przez człowieka: próbujemy kilku dróg, odrzucamy te prowadzące donikąd, aż znajdziemy właściwe rozwiązanie.

Technika promptingu: Klasyczna implementacja ToT wymaga kontrolera zewnętrznego (algorytm BFS/DFS, który zadaje modelowi pytania iteracyjnie). Istnieje jednak sposób, by zasymulować Tree of Thoughts jednym przemyślanym promptem – tzw. *Tree-of-Thought Prompting* [27]. W tym ujęciu prosimy model, by sam wygenerował i ocenił równoległe ścieżki myślenia. **Przykładowy zaawansowany prompt (według Hulberta 2023):**

```
Wyobraź sobie trzech niezależnych ekspertów rozwiązujących ten problem.
Każdy z ekspertów zapisze po 1 kroku swojego rozumowania, po czym wszyscy
dzielą się tymi krokami z grupą.
Następnie wszyscy przechodzą do kolejnego kroku, i tak dalej, budując
wspólnie rozwiązanie.
```

```
Jeśli którykolwiek z ekspertów w pewnym momencie zorientuje się, że obrał
błędny kierunek, wycofuje się i przestaje udzielać.
Zadanie/problemat brzmi: [tu wstaw treść problemu].
```

Ten prompt tworzy mentalnie **drzewo**: na każdym etapie mamy do 3 równoległych myśli (kroki ekspertów), które się ujawniają i wpływają na kolejne kroki [28]. Jeśli któryś *węzeł* (ekspert) uzna swoje rozumowanie za błędne, jego gałąź jest odcinana (ekspert milknie). Model dzięki takiemu ustawieniu sam generuje coś w rodzaju *panelu dyskusyjnego* nad problemem [29]. **Praktyka:** Dla modelu GPT-4 taki prompt skutkuje odpowiedzią, w której pojawiają się **równoległe ścieżki**: np. „Ekspert A myśli: …; Ekspert B: …; Ekspert C: …", następnie kolejne iteracje kroków. Na końcu odpowiedzi zwykle wyłania się rozwiązanie, które jedna z tych ścieżek doprowadziła do celu, podczas gdy inne mogły ulec wygaszeniu. To podejście jest potężne dla zadań takich jak zagadki matematyczne (np. *24 Game*), gdzie ToT w badaniach znacząco pobiło zwykłe metody [30]. **Porada:** W praktyce inżynier może dostosować liczbę „ekspertów" (gałęzi) i liczbę kroków, a także kryterium przerwania – np. poprosić: *„oceń po każdym kroku, czy rozwiązanie jest już pewne (tak/nie/może). Jeśli tak – zakończ"*. Taka automatyczna samoewaluacja uchroni przed niepotrzebnym rozrastaniem drzewa.

Graph of Thoughts (Graf Myśli) – sieć powiązanych rozumowań

Idea: Graph of Thoughts idzie krok dalej – pozwala modelować **dowolny graf zależności między myślami** [31]. W grafie wierzchołki to jednostki informacji (myśli), a krawędzie oznaczają zależności lub wykorzystanie jednej myśli w innej [32]. Taka swoboda struktury umożliwia m.in.: **łączenie różnych ścieżek rozumowania w jedną odpowiedź, destylację kluczowych wniosków z całej sieci myśli, stosowanie pętli sprzężenia zwrotnego (feedback)** do ulepszania wcześniejszych wniosków [32] [33]. Mówiąc prościej, model może **równolegle rozważać różne wątki problemu, a następnie je scalać, poprawiać lub ponownie rozgałęziać**. To przypomina myślenie człowieka nad złożonym problemem: tworzymy mapę myśli, wracamy do wcześniejszych założeń by je zrewidować w świetle nowych ustaleń, itp. Graf umożliwia też wystąpienie konwergencji – dwie różne serie rozumowań mogą doprowadzić do tego samego pośredniego wniosku, który zostanie połączony (zamiast powtarzać te same ustalenia dwukrotnie).

Technika promptingu: W czystej postaci Graph of Thoughts wymaga zewnętrznego kontrolera, który buduje graf (węzły i krawędzie) i zadaje modelowi konkretne pod-polecenia dla każdego węzła [34] [35]. Niemniej jednak, pewne aspekty można wykorzystać w promptach. **Kluczowe jest podzielenie problemu na podproblemy i wyraźne określenie relacji między nimi** – to tworzy mentalny graf. Przykładowe podejście w promptach: - *Rozłóż problem na komponenty:* Poproś model: *„Wypisz wszystkie aspekty tego problemu lub potencjalne kroki, jakie należy rozważyć. Nazwij je A, B, C…"*. Model wypisze elementy, tworząc **węzły grafu**. - *Nawiąż zależności:* Dopytaj: *„Jak te aspekty się ze sobą łączą? Które trzeba rozwiązać najpierw, a które zależą od innych?"*. W ten sposób model może wygenerować plan w formie grafu zależności (np. *„A i B można robić równolegle, ale C wymaga wyników A"*). - *Rozwiąż podproblemy osobno:* Następnie przejdź do **iteracyjnego rozwiązywania każdego węzła**. Można to robić sekwencyjnie, dając modelowi kontekst wcześniej rozwiązanych węzłów. Przykład: *„Rozwiąż część A (tu kontekst A). Następnie rozwiąż B, mając już wynik A (tu dołącz wynik A)".* W ten sposób budujemy graf, gdzie wyniki (węzły) przepływają do kolejnych. - *Scal wyniki:* Na końcu poproś: *„Połącz teraz rozwiązania części A, B, C w spójne rozwiązanie całego problemu"*. Model, mając w pamięci wszystkie wypracowane wątki, wygeneruje **końcową odpowiedź, która integruje poszczególne myśli**.

Przykład promptu dla Graph-of-Thoughts: Załóżmy pytanie: *„Jak zaplanować podróż dookoła świata minimalnym kosztem?"*. Można zaproponować graf: 1. Węzły: A – *Wybór kontynentów/krajów*, B – *Środki transportu*, C – *Noclegi*, D – *Wyżywienie i inne koszty*. 2. Relacje: A wpływa na B (destynacje determinują transport), A wpływa na C (miejsca determinują ceny noclegów), B i C wpływają na koszty w D, itd. 3.

Rozwiązanie węzłów: Model generuje propozycje tanich krajów (A), najtańszych opcji transportu między nimi (B), budżetowych opcji zakwaterowania (C). 4. Scalenie: Model łączy to w plan podróży (to będzie finalna „odpowiedź" zebrana z grafu myśli).

Warto zaznaczyć, że Graph of Thoughts jest koncepcją ogólną, a jej praktyczna realizacja bywa wspierana kodem (np. dedykowanymi bibliotekami [36]). Jednak już samo **świadome strukturyzowanie zapytania** przez programistę w powyższy sposób powoduje, że LLM **„aktywizuje" rozumowanie grafowe**, zamiast udzielić płaskiej, liniowej odpowiedzi. Takie podejście pomaga w problemach, gdzie trzeba **łączyć wiele informacji z różnych źródeł lub dziedzin** – model może wtedy każdy wątek opracować oddzielnie, a następnie zestawić fakty (co zmniejsza ryzyko pominięcia któregoś aspektu).

Adaptive Graph of Thoughts (AGoT) – dynamiczne dostosowanie rozumowania

Idea: Adaptive GoT to najnowsze rozwinięcie tych technik (przedstawione w 2025 roku), które **dynamicznie łączy zalety chain-of-thought, drzewa i grafu** w trakcie działania modelu [37]. Zamiast ustalonej z góry struktury (jak jedno drzewo), AGoT buduje **acykliczny graf myśli w locie**, **rekurencyjnie dekomponując problem na podproblemy** i decydując, które wątki wymagają dalszego rozwinięcia [37] [38]. Najważniejszą cechą jest tu **selektywna eksploracja**: model **alokuje wysiłek obliczeniowy tam, gdzie jest najbardziej potrzebny** – jeśli jakiś podproblem jest trudny lub niepewny, będzie drążony głębiej, a prostsze elementy nie będą nadmiernie rozbudowywane [39]. To adaptacyjne podejście zwiększa efektywność: osiągnięto znaczne poprawy jakości rozumowania (np. +46% w zadaniach naukowych) bez trenowania modelu od nowa, jedynie poprzez mądrzejsze wykorzystywanie go na etapie *inference* [40] [41].

Technika promptingu: Całkowite wdrożenie AGoT również wymaga zaawansowanego kontrolera i mechanizmów oceny niepewności. Jednak w inżynierii promptów możemy wdrożyć **zasadę „dziel i rządź adaptacyjnie"**: - **Rozpoznanie trudności:** Możemy poprosić model na początku: *„Zidentyfikuj, które części tego zadania mogą być najbardziej złożone lub niejasne."* To pozwoli modelowi wskazać potencjalne punkty trudności. - **Rekurencyjne pogłębianie:** Dla każdego z trudnych aspektów, nakłaniamy model do szczegółowego rozbicia go na mniejsze kroki lub do **zadania pytań pomocniczych** (co jest pewną innowacją AGoT – model *pyta sam siebie* o brakujące informacje). Przykładowo, jeśli model stwierdzi: *„Najtrudniejsze jest ustalenie, gdzie zdobyć dane do analizy"*, możemy kazać mu wygenerować sub-pytanie: *„Jakie źródła danych są dostępne?"* i na nie odpowiedzieć. - **Dynamiczna kontrola ścieżek:** Możemy zawrzeć w promptach instrukcje typu: *„Jeśli podczas rozwiązywania zauważysz, że jakiś wątek staje się zbyt skomplikowany albo brakuje Ci informacji – zatrzymaj się i najpierw rozwiąż pomniejsze kwestie"*. Dzięki temu model nie będzie brnąć w niejasności, tylko sam zainicjuje *odgałęzienie* (np. uzupełni założenia, wyjaśni definicje). - **Łączenie wyników:** Po takich adaptacyjnych krokach – podobnie jak w GoT – zlecamy zebranie wszystkiego w jedną spójną odpowiedź.

Przykład podejścia AGoT: Załóżmy pytanie złożone: *„Oblicz optymalną trajektorię lotu kosmicznego z uwzględnieniem grawitacji pośrednich planet."* Model może uznać, że największy problem to rozwiązanie równań trajektorii (A) i dostępność danych astronomicznych (B). **Krok 1:** Model identyfikuje te dwa wątki. **Krok 2:** Dla wątku A stwierdza, że potrzebuje założeń (masa statku, paliwo) – zamiast strzelać, pyta (lub my we wkładzie promptu pytamy): *„Jakie założenia przyjąć dla masy i paliwa?"*. **Krok 3:** Model dostaje lub ustala założenia, rozwiązuje wątek A. Wątek B (dane) może rozwiązać np. stwierdzając, że *„przyjmie uśrednione dane orbitalne znanych planet"*. **Krok 4:** Łączy A i B, generuje trajektorię. Dzięki adaptacyjnemu podejściu, **model nie utknął** na niejawnych założeniach – **sam je wykrył i uzupełnił**, zamiast halucynować czy pomijać. Co ważne, w AGoT model **nie wykonuje nadmiarowej pracy tam, gdzie nie trzeba** – proste aspekty załatwia od razu, a energię skupia na trudnych (to tak, jakby w drzewie myśli dynamicznie regulować głębokość poszukiwań dla różnych gałęzi).

Podsumowanie wskazówek: Dla inżyniera promptów praktyczne zastosowanie tych ram sprowadza się do **świadomego kierowania uwagą modelu**: - W ToT – wymuszamy eksplorację wielu możliwości i ich ocenę. - W GoT – rozbijamy problem na sieć elementów i pozwalamy modelowi iterować między nimi. - W AGoT – zachęcamy model do aktywnego wykrywania własnych braków wiedzy lub złożoności i adaptacyjnego dopytywania/rozbijania problemu.

Warto eksperymentować z promptami zawierającymi frazy typu *„Jeśli uważasz, że potrzebne są dodatkowe informacje, wyjaśnij czego brakuje”* albo *„Rozważ kilka podejść i wybierz najlepsze wraz z uzasadnieniem”*. Takie instrukcje, poparte odpowiednim formatowaniem (np. numerowane listy dla podproblemów, role ‘ekspertów’), **znacznie zwiększają szansę uruchomienia u LLM procesu głębszego rozumowania** zgodnie z opisanymi ramami.

Integracja LLM z grafami wiedzy w RAG – techniki GraphRAG

RAG a problem złożonych zapytań: Retrieval-Augmented Generation (RAG) polega na podawaniu modelowi zewnętrznych danych kontekstowych (poprzez wyszukiwanie) celem poprawy trafności i ograniczenia halucynacji. Typowy pipeline RAG to wektorowa baza wiedzy + LLM [42]. Jednak tradycyjny RAG, oparty na podobieństwie semantycznym fragmentów tekstu, **ma trudności z pytaniami wymagającymi wieloetapowego rozumowania lub łączenia rozproszonych informacji** [43]. Przykładowo pytanie: *„Jak miał na imię syn mężczyzny, który pokonał uzurpatora Allectusa?”* wymaga **dwóch kroków**: najpierw ustalić, kto pokonał Allectusa, a potem znaleźć imię syna tej osoby [44]. Zwykłe wektorowe wyszukiwanie może nie powiązać tych faktów, bo każdy z nich osobno może występować w innym dokumencie. Efekt – model nie znajdzie odpowiedzi albo *“zgaduje”*, co prowadzi do błędów.

GraphRAG – co to jest: Naukowcy z Microsoft Research zaproponowali podejście **GraphRAG**, które wzbogaca standardowy RAG o **wykorzystanie grafów wiedzy (Knowledge Graphs)** [45]. W GraphRAG dane nie są przechowywane wyłącznie jako luźne dokumenty czy embedingi, lecz jako **graf powiązanych encji i relacji**. Węzły grafu reprezentują jednostki wiedzy (pojęcia, obiekty, osoby, daty), a krawędzie – relacje między nimi (np. „X jest synem Y”, „X pokonał Z”). Dzięki temu mechanizm wyszukiwania może **uwzględnić związki między faktami**. W pytaniu o Allectusa system znajdzie w grafie ścieżkę: Allectus -> pokonany przez -> Asclepiodotus (np.) -> ojciec syna -> imię syna (tu: imię). **Zamiast szukać osobno w tekstach**, GraphRAG mógłby dotrzeć do odpowiedzi, bo graf jawnie łączy te kropki.

Poprawa precyzji i kontekstu dzięki grafom: Włączenie bazy typu **Neo4j** (popularna baza grafowa) do ekosystemu GenAI umożliwia **bogatszy retriever**: nie tylko zwraca fragmenty tekstu, ale całe *podgrafy wiedzy* związane z pytaniem. To daje LLM kontekst, który zawiera nie tylko izolowane fakty, lecz też ich powiązania. Przekłada się to na: - **Większą trafność odpowiedzi:** Graf potrafi wydobyć *ukryte połączenia*, które nie są oczywiste w samym tekście [46]. Model opierający się na takiej sieci danych jest mniej skłonny do pominięcia ważnego elementu – np. nie “zapomni”, że dwie informacje są powiązane, bo dostaje je razem. - **Mniej halucynacji:** LLM działający w RAG ma za zadanie bazować wyłącznie na dostarczonym kontekście. Jeśli kontekst jest dokładny i powiązany (dzięki grafowi), zmniejsza się pokusa modelu do *“wymyślania”* brakujących ogniw. Innymi słowy, graf **zwiększa wiarygodność** – odpowiedzi są oparte na faktach i relacjach z wiedzy dziedzinowej, a nie na luźnych skojarzeniach modelu [47] [48]. W zastosowaniach biznesowych to kluczowe, bo klasyczne LLM generują “pomysłowe” ale fałszywe uzasadnienia, co GraphRAG stara się wyeliminować [47]. - **Lepsze odpowiedzi na pytania złożone (multi-hop):** Jak wspomniano, proste RAG słabo radzi sobie z pytaniami wymagającymi przejścia przez kilka faktów. GraphRAG jest wręcz zaprojektowany do *wieloskokowych* zapytań – graf dostarcza naturalną strukturę do takich operacji. W badaniach wykazano np. **wzrost skuteczności sortowania o 62%** w

GraphRAG względem Tree-of-Thoughts, przy jednoczesnym obniżeniu kosztów obliczeń o 31% [32] (co sugeruje, że grafowe podejście pozwala efektywniej wykorzystywać zapytania do LLM).

Architektura GraphRAG – jak to działa: Cały proces można podzielić na **indeksowanie** (budowa grafu wiedzy) oraz **zapytania** (wykorzystanie grafu do generacji odpowiedzi) [49].

- **Indeksowanie (budowa grafu):** Najczęściej zaczynamy od korpusu tekstów (dokumenty, artykuły, np. dokumentacja firmy czy zbiory artykułów). GraphRAG wykorzystuje LLM lub pipeline NLP, by **wydobyć z tekstów kluczowe encje, relacje i twierdzenia** [50]. Przykładowo, ze zdania "Jan Kowalski założył firmę X w 2020 roku" wyciągniemy encje: Jan Kowalski (osoba), Firma X (organizacja), 2020 (rok) oraz relację: (Jan Kowalski) –[założył w roku]–> (Firma X, 2020) [50]. Takie trójki (s, r, o) są wprowadzane do grafu Neo4j. GraphRAG może także **klasteryzować encje w społeczności tematyczne** (np. wszystkie encje dot. jednej dziedziny) za pomocą algorytmów grafowych jak Leiden [51], a następnie **tworzyć streszczenia każdej społeczności** przy użyciu LLM [52]. Rezultat: mamy graf wiedzy z warstwą ogólnych podsumowań (komunitów) oraz szczegółowymi połączeniami między encjami. Już na etapie budowy model językowy pomaga – np. generuje krótkie opisy węzłów i relacji (co może zostać użyte potem jako kontekst tekstowy).

- **Zapytania (retrieval + generacja odpowiedzi):** W GraphRAG przewiduje się dwa główne tryby zapytań: **globalne** i **lokalne** [53].

    - *Globalne* nadają się do pytań ogólnych o cały zbiór danych. Wtedy używa się wspomnianych **podsumowań społeczności**: dla pytania wybiera się te grupy encji, które mogą być istotne, i pobiera ich opisy jako kontekst [54]. LLM generuje z nich tzw. *Rated Intermediate Responses* – cząstkowe odpowiedzi z wypunktowanymi faktami i ich ważnością [55]. Następnie system rankuje te fakty i wybiera najważniejsze, by z nich złożyć **ostateczną odpowiedź** [56]. Dzięki temu z wielu dokumentów/grafów wyłuskiwane jest sedno potrzebne do odpowiedzi.
    - *Lokalne* zapytania dotyczą konkretnych encji. Wtedy GraphRAG **rozszerza kontekst o bezpośrednie połączenia danej encji w grafie** – np. jeśli pytanie brzmi *"Co osiągnęła firma X założona przez Jana Kowalskiego?"*, system znajdzie w grafie węzeł "Firma X", pobierze jej właściwości, powiązania (np. "założyciel: Jan Kowalski", "przychód: …", "produkty: …") i przekaże te informacje modelowi [57] [58]. Model nie musi wtedy szukać wolno w dokumentach – dostaje uporządkowaną paczkę faktów. W razie potrzeby, system może też generować dynamicznie zapytania Cypher (język zapytań grafowych): np. *"znajdź wszystkie projekty powiązane z technologią Y realizowane po 2020"* – to zrealizuje baza grafowa, a LLM tylko sformułuje odpowiedź na bazie wyniku.

Wzorce promptów dla GraphRAG: Integracja grafu wpływa też na sposób formułowania promptów do LLM: - **Przy budowie grafu:** używamy promptów ekstrakcyjnych. Np. *"Przeczytaj poniższy tekst i wypisz wszystkie pary (podmiot – relacja – przedmiot) oraz ważne fakty."* albo *"Stwórz notatkę opisującą relacje między wymienionymi encjami."* Taki prompt kieruje LLM by zwrócił dane, które bezpośrednio zasilą graf (trójki, atrybuty). **Przykład:** *"Tekst: [paragraf]. Wypisz znalezione fakty w formacie: [Encja1] – [Rodzaj relacji] – [Encja2]."* – LLM generuje np. "Asclepiodotus – pokonał – Allectus; Asclepiodotus – ojciec – [imię syna]". - **Przy zapytaniu globalnym:** prompt dla LLM będzie zawierał **skondensowaną zawartość grafu**. Można to sformułować: *"Na podstawie poniższych informacji z bazy wiedzy odpowiedz na pytanie... Informacje: [tu lista punktów z grafu]. Pytanie: …"*. Ważne jest tu polecenie dla modelu, by korzystał tylko z dostarczonych faktów. Np.: *"Użyj WYŁĄCZNIE podanych faktów, aby odpowiedzieć na pytanie, unikając własnych domysłów."* – to bezpośrednio zapobiega halucynacjom. Model będzie czuł się "związany" kontekstem. - **Przy zapytaniu lokalnym:** być może zastosujemy **prompt dwufazowy**: najpierw LLM jako *planner* generuje zapytanie Cypher na podstawie pytania użytkownika (np. *"Znajdź w grafie: syna mężczyzny, który pokonał*

Allectusa" -> LLM generuje `MATCH (m)-[:pokonał]->(Allectus), (m)-[:ojciec]->(syn) RETURN syn.name`). Potem ten wynik wstawiamy do kolejnego promptu, np.: *"Pytanie użytkownika: … Odpowiedź na podstawie bazy: [wynik zapytania]. Udziel pełnej odpowiedzi."*. Ten wzorzec – **LLM jako tłumacz zapytań i potem jako generujący odpowiedź** – jest często stosowany w GraphRAG [59]. Można wspomagać model, podając kilka przykładów few-shot jak mapować pytanie na Cypher (co stanowi *dynamiczny prompt few-shot* dla generowania zapytań do grafu [60]).

Aktualizacja grafu i unikanie przestarzałych danych: Ważnym aspektem jest utrzymanie grafu wiedzy aktualnym, zwłaszcza jeśli RAG ma minimalizować halucynacje (często wynikające z braków w wiedzy). **Architektoniczne podejścia** obejmują: - Regularne przepuszczanie nowych dokumentów przez LLM ekstrakcyjny i dodawanie nowych węzłów/relacji do grafu (automatyczne rozszerzanie wiedzy). - **Wersjonowanie grafu**: w krytycznych systemach można utrzymywać historię zmian grafu i meta-informacje (np. źródło i data każdej krawędzi). W promptach da się wtedy uwzględnić świeżość wiedzy: *"(Fakt X, źródło: dokument z 2024)"*. Model widząc daty będzie ostrożniejszy z informacjami historycznymi. - **Feedback pętla:** jeżeli mimo wszystko LLM wygeneruje niepewną odpowiedź, można dodać mechanizm weryfikacji – np. jeszcze raz przepytać graf (przez kolejny prompt: *"Sprawdź w grafie poprawność powyższej odpowiedzi"*). Co prawda to bardziej aspekt systemowy niż czystego promptu, ale idea jest taka, by **monitorować wydajność**: czy odpowiedzi faktycznie bazują na grafie i czy nie pomijają istotnego węzła.

Odkrywanie informacji w danych narracyjnych: Grafy szczególnie błyszczą, gdy dane wejściowe są *narracyjne* (długie artykuły, opowiadania, dokumenty). Zamiast dawać LLM długi tekst i liczyć, że "znajdzie igłę w stogu siana", GraphRAG **wydobywa strukturę narracji** w postaci grafu zdarzeń, postaci, miejsc. To ułatwia odkrywanie informacji, bo np. można zapytać graf: *"pokaż wszystkie zdarzenia powiązane z postacią X"* albo *"jakie są związki między wątkiem A a wątkiem B w tej historii?"*. LLM mając takie usystematyzowane dane, odpowie precyzyjniej. **Przykład:** W powieści fantasy graf wiedzy może połączyć postacie z rodami, mieczami, miejscami bitew. Pytanie *"Kto ostatecznie pokonał czarnoksiężnika i przy pomocy którego artefaktu?"* – to dwa wątki historii, które w tekście mogą być oddalone o setki stron. W grafie pewnie istnieje ścieżka: Czarnoksiężnik -> pokonany przez -> Bohater, oraz Bohater -> użył -> Artefakt. GraphRAG błyskawicznie **wyszuka taką ścieżkę i przekaże ją modelowi**, który sformułuje odpowiedź bez błądzenia po tekście. Zatem, integracja grafów z RAG **wydatnie zwiększa zdolność systemu do odkrywania powiązań w dużych zbiorach danych narracyjnych**, czyniąc odpowiedzi bardziej kompletne i kontekstowe.

Wykrywanie i uzupełnianie luk w wiedzy promptów oraz person AI

Definicja problemu: *Luki w wiedzy promptu* to braki lub niejasności w informacjach dostarczonych w zapytaniu do LLM. Gdy użytkownik pomija istotny kontekst, nie precyzuje wymagań albo zadaje pytanie zbyt ogólne, model może wygenerować odpowiedź błędną lub nieprzydatną. *Luki w wiedzy persony AI* odnoszą się z kolei do sytuacji, gdy model przyjmuje określoną rolę/osobowość (personę), ale **nie posiada pełnej wiedzy lub spójności wymaganej od tej roli**. Na przykład, jeśli prosimy LLM by udawał eksperta medycznego, a on nie zna najnowszych wytycznych – pojawia się luka między oczekiwaną wiedzą persony a faktyczną wiedzą modelu.

Te luki mogą prowadzić do **nieefektywnych interakcji** – model daje odpowiedzi wymijające, halucynuje brakujące dane albo dopytuje wielokrotnie o doprecyzowanie (wydłużając rozmowę). W kontekście programistycznym analizowano np. konwersacje deweloper–ChatGPT i stwierdzono, że **nieudane rozmowy w ponad połowie przypadków zawierały luki wiedzy w promptach**, podczas gdy w skutecznych dialogach tylko ~13% promptów miało takie braki [61]. Najczęstszym problemem okazał się

Missing Context, czyli brak kluczowych informacji o problemie [62]. Oznacza to, że **uzupełnienie brakującego kontekstu często decyduje o sukcesie** – bez tego rozmowa staje się długa i jałowa [63].

Standardowe metody vs. zaawansowane strategie: Typowe podejścia do radzenia sobie z lukami to: - *Samoaudyt modelu:* LLM generuje odpowiedź, po czym weryfikuje sam siebie (np. polecenie *"Sprawdź swoją odpowiedź pod kątem poprawności faktów"*). To jednak często działa na dane, które model już posiada – nie wprowadza nowej wiedzy. - *Walidacja przez użytkownika:* Użytkownik musi zauważyć, że model pominął coś istotnego i ręcznie doprecyzować kolejnym pytaniem. Jest to mało wydajne i nie zawsze oczywiste (skąd użytkownik ma wiedzieć, czego model *nie wie*?).

Zaawansowane metodologie idą dalej: **proaktywnie wykrywają luki i starają się je zapełnić zanim nastąpi błąd lub nieporozumienie**.

1. Automatyczne wykrywanie luk w promptach: Badania sugerują, że można zbudować narzędzia analizujące tekst promptu pod kątem brakujących elementów [64] [65]. Na przykład, w domenie programistycznej zidentyfikowano heurystyki oceny jakości promptu: **Specyficzność, Bogactwo Kontekstu, Jasność** [64]. Jeśli prompt jest zbyt ogólny (niska specyficzność), nie zawiera wystarczających danych wejściowych (niska kontekstowość) lub jest niejednoznaczny (brak jasności), system może **oznaczyć go jako potencjalnie problematyczny** [64] [66]. Zaawansowana implementacja to np. **wtyczka do przeglądarki** czy IDE, która w czasie pisania promptu podkreśla: "Tutaj może brakować informacji o wersji środowiska" albo "Polecenie nieprecyzyjne – co konkretnie znaczy *'optymalny'*?". Taki prototyp został opisany przez Zhou i in. (2023) – potrafił on **dynamicznie oflagować niejasny prompt i zaproponować szablon poprawek** zanim użytkownik wyśle go do modelu [66] [67]. Z praktycznego punktu widzenia, inżynier promptów może też samemu stosować checklistę: czy uwzględniłem *kto*, *co*, *kiedy*, *w jakim kontekście*? Im więcej konkretów dostarczymy modelowi, tym mniejsza szansa na lukę.

2. Model jako inkwizytor – dopytywanie zamiast zgadywania: Bardzo obiecującą strategią jest nauczenie modelu, by **zamiast zgadywać brakującą wiedzę – zadawał pytania pomocnicze**. To naśladuje ludzką reakcję: jeśli pytanie jest niejasne, najpierw prosimy o doprecyzowanie. Framework **CPER (Conversation Preference Elicitation and Recommendation)** pokazuje, że LLM można zmusić do takiej postawy w dialogu [68] [69]. Działa to tak: model ocenia własną niepewność co do kontekstu użytkownika i jeśli wykryje *persona knowledge gap* (np. brak kluczowej informacji o preferencjach rozmówcy), wtrąca do rozmowy pytanie uzupełniające [70] [69]. Przykład: Użytkownik prosi *"Poleć mi film na wieczór"*, a model ma personę doradcy filmowego. Zamiast od razu rzucać tytułami na ślepo, **model pyta:** *"Jakie gatunki filmowe lubisz najbardziej?"*. To wypełnia lukę wiedzy o preferencjach. Dopiero po uzyskaniu odpowiedzi, model rekomenduje film, trafiając lepiej w gust (bo zebrał brakującą informację). W testach A/B takie podejście znacząco poprawiło ocenę odpowiedzi – ludzie woleli model, który umiał dopytać i spersonalizować rozmowę [71] [72].

Jak wykorzystać to w promptach? Można jawnie zachęcić model: *"Jeśli brakuje Ci informacji by udzielić dobrej odpowiedzi, zapytaj użytkownika o doprecyzowanie zamiast zgadywać."* Albo: *"Jeśli Twoja odpowiedź wymaga założeń, wypisz czego potrzebujesz zanim odpowiesz."* Tego rodzaju instrukcje w System Message lub w pierwszym promptcie mogą drastycznie zmienić zachowanie LLM – stanie się ostrożniejszy i interaktywny, zamiast konfabulować. To **wykracza poza standardowy tryb**, w którym model udziela odpowiedzi bazując tylko na bieżącym wejściu.

3. Samoanaliza i iteracja wiedzy: Inną techniką jest wymuszenie na modelu **eksploracji swojej "wiedzy" i wskazania dziur**. Można to osiągnąć promptem typu: *"Wypisz wszystko, co wiesz na ten temat oraz wskaż aspekty, których nie jesteś pewien."* Model w odpowiedzi może wygenerować listę faktów oraz np. punkt z oznaczeniem "[?]" przy tych, co do których nie ma pewności. Następnie można zareagować: *"Punkt X jest niejasny – oto dodatkowe informacje: … Teraz udziel pełnej odpowiedzi."* W ten sposób model

niejako **sam prosi o informacje** (poprzez przyznanie się do niepewności). Taka otwarta "metaprompta" jest formą zaawansowanego sterowania – wymaga, by model wewnętrznie ocenił stan swojej wiedzy. GPT-4 i nowsze modele często potrafią to zrobić: np. zapytane o aktualne wydarzenie (po 2021), przyznają *"Nie mam danych po 2021"*. Wystarczy to wykorzystać.

4. Wykorzystanie drugiego modelu lub walidatora: W ramach LLMOps można wprowadzić drugi model jako **audytora wiedzy**. Przykładowo, jedna instancja LLM generuje odpowiedź, a druga dostaje polecenie: *"Sprawdź, czy w powyższej odpowiedzi brakuje jakiejś informacji potrzebnej użytkownikowi"*. Taki walidator, szczególnie jeśli ma dostęp do szerszej bazy wiedzy, może wychwycić pominięte kwestie. Jeśli wykryje lukę, system może automatycznie zadać oryginalnemu modelowi dodatkowe pytanie lub dołączyć brakujący kontekst. To przypomina "podwójne myślenie": jeden model rozwiązuje zadanie, inny ocenia, czy rozumowanie było kompletne. Jest to **bardziej zaawansowane niż autokrytyka**, bo drugi model może być inaczej strojony (np. bardziej rygorystyczny, mniej skłonny do upraszczania).

Przykładowe procesy/promy do dynamicznych testów: - *Checklist Q&A:* Po odpowiedzi modelu można zadać jemu samemu serię pytań kontrolnych: *"Czy odpowiedziałeś na wszystkie części pytania? Czy założenia zostały wyjaśnione? Czy potrzebne były jakieś dane, których nie miałeś?"*. Taki **mini-cross-exam** przez model pozwala ujawnić przeoczenia. Jeśli model odpowie np. *"Nie wyjaśniłem, skąd wziąłem tę wartość, bo nie byłem pewien"*, to jasny sygnał luki – można promptem dopytać o tę wartość. - *Dynamiczne uzupełnianie persony:* Jeżeli tworzymy personę AI (np. lekarza), dobrym zwyczajem jest **przetestowanie jej wiedzy** zestawem pytań kontrolnych z danej dziedziny. Jeśli model-persona nie zna odpowiedzi, to wskazuje, gdzie są braki. Wtedy: 1. Albo dostarczamy mu dodatkowy kontekst (np. *"Oto streszczenie najnowszych badań na ten temat..."*), 2. Albo modyfikujemy prompt definicji persony, aby zawierał brakujące elementy (np. dopisujemy do opisu roli: "Masz dostęp do bazy danych X w razie potrzeby" lub "Jeśli nie znasz odpowiedzi, skieruj do specjalisty"). Ten drugi punkt to ciekawa metoda – **przez sprytne prompty możemy zasymulować uzupełnienie wiedzy**: model nie realnie się nauczy, ale możemy nakazać mu pewne reakcje, gdy napotka lukę. Np. *"Jako lekarz, jeśli nie znasz jakiegoś szczegółu, posługujesz się swoją encyklopedią medyczną zanim odpowiesz"* – w efekcie model może wymusić na sobie krok "wyszukania" (w praktyce to kolejny chain-of-thought, ewentualnie z integracją narzędzia).

5. Beyond verification – continuous learning setup: Choć bieżąca generacja LLM nie uczy się w trakcie rozmowy (nie zmienia wag modelu), można skonfigurować system, który **zapamiętuje uzupełnioną wiedzę** na poziomie rozmowy czy sesji. Np. jeśli persona AI spyta użytkownika o brakujące preferencje (jak w przykładzie filmowym) i dostanie odpowiedź, to zapisujemy to jako część stanu konwersacji. Zaawansowane systemy tworzą w tle **profil użytkownika/konwersacji**, który jest sukcesywnie rozbudowywany. W kolejnych turach prompt ten profil jest dołączany, dzięki czemu model nie zapomni już uzupełnionych informacji (co rozwiązuje lukę wiedzy długoterminowo w obrębie sesji). To trochę analogia do pamięci długotrwałej – coś jak Knowledge Graph użytkownika, do którego model może się odwoływać.

Podsumowując, **proaktywne zarządzanie lukami wiedzy** to nowy ważny obszar w inżynierii promptów. Kluczowe elementy to: - **Wczesna detekcja**: analiza promptu i kontekstu pod kątem braków (automatyczna lub manualna z checklistą). - **Strategia „nie wiem – pytam"**: skłonienie modelu do przyznania się do niewiedzy i zadania pytań uzupełniających zamiast halucynacji. - **Adaptacja persony**: zapewnienie, że jeśli model ma odgrywać eksperta, to ma dostarczone "ściśle tajne" informacje eksperckie albo ma procedurę co robić, gdy czegoś nie wie. - **Iteracja i feedback**: pętle walidacji odpowiedzi i doprecyzowania w czasie rzeczywistym.

Dla inżynierów promptów oznacza to, że tworząc rozwiązania oparte na LLM, **nie wystarczy optymalizować samego jednego promptu**. Trzeba projektować *cały proces dialogu*: przewidywać, gdzie model może mieć lukę i zawczasu dostarczać mechanizmy jej zapełnienia. Często najlepsze wyniki daje

połączenie podejść – np. najpierw model generuje plan (co potrzebuje), potem pyta o brakujące dane, potem generuje odpowiedź, a na końcu inny model to sprawdza. Takie wieloetapowe konstrukcje promptów (i ewentualnie kodu je orkiestrującego) stanowią praktyczne ramy dla pewnych i spójnych odpowiedzi AI, nawet w obliczu początkowych braków wiedzy. Dzięki temu użytkownik dostaje mniej *"Nie wiem, co masz na myśli"*, a więcej **trafnych i rzetelnych wyników, z jasno uzupełnionym kontekstem**. [65] [70].

Literatura: W powyższym omówieniu przytoczono przykłady i wyniki z najnowszych publikacji (2023-2025) oraz doświadczeń społeczności. M.in. praca Zhou et al. (2023) o lukach w promptach deweloperskich [73], Baskar et al. (2025) o personie knowledge gap i frameworku CPER [68] [69], czy raporty Microsoft dotyczące GraphRAG (2024) [48]. Zachęcamy do zapoznania się z tymi źródłami po więcej szczegółów technicznych i ewaluacji opisanych metod. Wszystkie opisane techniki mają na celu to samo: **zwiększyć niezawodność i inteligencję systemów opartych o LLM poprzez świadome kierowanie ich rozumowaniem i wiedzą** – czy to w edytorze kodu, czy w dialogu, czy w systemie pytanie-odpowiedź wspartym grafem wiedzy. Dzięki temu AI staje się nie tylko elokwentnym generatorem tekstu, ale prawdziwym asystentem rozwiązującym problemy w sposób przemyślany i wiarygodny.

---

[1] [16] [17] [21] Beyond Vibe Coding: AI-Assisted Coding With Cursor and Opik
https://www.comet.com/site/blog/cursor-vibe-coding/

[2] [3] Anyone using Cursor AI and barely writing any code? Anything better than Cursor AI ? : r/ChatGPTCoding
https://www.reddit.com/r/ChatGPTCoding/comments/1c1o8wm/anyone_using_cursor_ai_and_barely_writing_any/

[4] [5] [7] [10] [11] [12] [13] [14] [15] Why Cursor Is the Code Assistant You Didn't Know You Needed (But Totally Deserve) | Blog | Conclusive Engineering
https://conclusive.tech/blog/cursor-code-assistant/

[6] Cursor AI: A Guide With 10 Practical Examples | DataCamp
https://www.datacamp.com/tutorial/cursor-ai-code-editor

[8] [9] [18] [19] [20] Prompting the perfect coding partner through .Cursorrules - Discussions - Cursor - Community Forum
https://forum.cursor.com/t/prompting-the-perfect-coding-partner-through-cursorrules/39907

[22] [23] [24] [25] [26] [27] [28] [29] [30] Tree of Thoughts (ToT) | Prompt Engineering Guide
https://www.promptingguide.ai/techniques/tot

[31] [32] [2308.09687] Graph of Thoughts: Solving Elaborate Problems with Large Language Models
https://arxiv.org/abs/2308.09687

[33] Graph of Thoughts: A New Paradigm for Elaborate Problem-Solving in Large Language Models - KDnuggets
https://www.kdnuggets.com/graph-of-thoughts-a-new-paradigm-for-elaborate-problem-solving-in-large-language-models

[34] [35] [36] LLMs Graph of Thoughts Framework. Case study | by Jomsborg Lab | Medium
https://medium.com/@JacekWo/llms-graph-of-thoughts-framework-c5607a46aa9a

[37] [38] [39] [40] [41] Adaptive Graph of Thoughts: Test-Time Reasoning
https://www.emergentmind.com/papers/2502.05078

[42] [43] [44] [45] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] GraphRAG Explained: Enhancing RAG with Knowledge Graphs | by Zilliz | Medium
https://medium.com/@zilliz_learn/graphrag-explained-enhancing-rag-with-knowledge-graphs-3312065f99e1

46 47 48 What Is GraphRAG?
https://neo4j.com/blog/genai/what-is-graphrag/

59 Neo4j graphRAG POC : r/Rag - Reddit
https://www.reddit.com/r/Rag/comments/1l8dvs8/neo4j_graphrag_poc/

60 Building a robust GraphRAG System for a specific use case - Medium
https://medium.com/infinitgraph/building-a-robust-graphrag-system-for-a-specific-use-case-part-three-59d5dc64e3f4

61 62 63 64 65 66 67 73 Towards Detecting Prompt Knowledge Gaps for Improved LLM-guided Issue Resolution
https://arxiv.org/html/2501.11709v1

68 69 70 71 72 (CPER) From Guessing to Asking: An Approach to Resolving the Persona Knowledge Gap in LLMs during Multi-Turn Conversations
https://arxiv.org/html/2503.12556v1