OKIEM EKSPERTA

Claude Code i programowanie agentowe

Przewodnik dewelopera po systemach agentowych

Eden Marco

Helion

packt

Tytuł oryginału: Agentic Coding with Claude Code: The everyday developer's guide to agentic coding with Claude Code

Tłumaczenie: Piotr Pilch

ISBN: 978-83-289-4089-5

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

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

Printed in Poland.

Spis treści

O autorze **13**

O recenzentach **14**

Przedmowa **15**

CZĘŚĆ 1
Podstawy inżynierii kontekstu i narzędzia Claude Code

ROZDZIAŁ 1
Inżynieria kontekstu **23**

Wprowadzenie do inżynierii kontekstu 23
Od inżynierii promptów do inżynierii kontekstu 24
Narzędzie Claude Code i jego strategie inżynierii kontekstu 26
Strategia 1.: zapisywanie kontekstu i pamięć trwała 27
Strategia 2.: inteligentne uzyskiwanie kontekstu 30
Strategia 3.: kompresowanie kontekstu 30
Strategia 4.: izolowanie kontekstu za pomocą subagentów 31
Prompty systemowe i powody ich znaczenia 32
Najlepsze praktyki tworzenia promptów systemowych 32
Podsumowanie 35

ROZDZIAŁ 2
Narzędzie Claude Code w pigułce **37**

Konfigurowanie projektu w środowisku Next.js 38
Uruchamianie i weryfikacja aplikacji 40
Inicjalizacja narzędzia Claude Code 40

Połączenie serwera Playwright MCP z narzędziem Claude Code 43
Weryfikacja połączenia z serwerem MCP 44
Zastosowanie reguł środowiska Cursor jako dodatkowego kontekstu 45
Zastosowanie projektu opartego na specyfikacji 48
Przegląd wygenerowanej specyfikacji 50
Implementacja strony głównej za pomocą specyfikacji 51
Uruchomienie implementacji 52
Podsumowanie 54

ROZDZIAŁ 3
Pierwsze kroki z narzędziem Claude Code — przegląd podstawowych poleceń 55

Cennik i plany dostępne dla narzędzia Claude Code 56
Opcja 1.: użycie klucza interfejsu API Anthropic 56
Opcja 2.: korzystanie z subskrypcji modelu Claude 59
Sterowanie narzędziem Claude Code za pomocą poleceń ze znakiem „/" 61
Integracja narzędzia Claude Code ze środowiskiem IDE 63
Używanie hooków w narzędziu Claude Code 64
Tworzenie hooka powiadomienia 65
Pamięć w narzędziu Claude Code 69
Typy pamięci 69
Jak narzędzie Claude Code wczytuje pamięć? 70
Praktyczny przykład: wykorzystanie pamięci w istniejącym projekcie 72
Opcja cofania w narzędziu Claude Code 77
Zalety i znaczenie opcji cofania 77
Korzystanie z opcji cofania w praktyce 78
Niestandardowe polecenia narzędzia Claude Code ze znakiem „/" (oparte na systemie Skills) 81
Tworzenie umiejętności 81
Użycie umiejętności 82
Doskonalenie umiejętności za pomocą elementu zastępczego $ARGUMENTS 82
Ulepszanie za pomocą inżynierii kontekstu automatycznie generowanych komunikatów zatwierdzania 83
Podsumowanie 85

CZĘŚĆ 2. Rozszerzanie narzędzia Claude Code: protokoły, automatyzacja i procesy strukturalne

ROZDZIAŁ 4
Rozszerzanie narzędzia Claude Code za pomocą serwerów MCP i dodatków 89

Dlaczego protokół MCP? 90
Problem integracji 90
Problem z przenośnością 91
Protokół MCP jako warstwa abstrakcji 93
Podstawowa architektura protokołu MCP 94
Zalety protokołu MCP 95
Główne komponenty protokołu MCP 95
Zastosowanie protokołu MCP 98
Działanie bez serwera MCP 98
Łączenie się z serwerem MCP 99
Ponowne użycie tego samego serwera MCP między klientami 100
Połączenie narzędzia Claude Code z serwerem MCP 102
Dodawanie serwera MCP Context7 z poziomu wiersza poleceń 103
Zasięg i konfiguracja serwera MCP 104
Ponowne uruchamianie narzędzia Claude Code i włączanie serwera MCP 105
Wyświetlanie i sprawdzanie zainstalowanych serwerów MCP 105
Kierowanie zapytań do środowiska LangGraph za pomocą serwera MCP Context7 106
Utrwalanie konfiguracji używanego serwera MCP w pamięci projektu 107
Weryfikacja trwałości działania serwera MCP 107
Zatwierdzanie artefaktów MCP w systemie kontroli wersji 107
Podsumowanie zasięgów protokołu MCP 108
Kiedy stosować poszczególne zasięgi protokołu MCP? 109
Inżynieria kontekstu w ekosystemie protokołu MCP 110
Problem — zbyt ogólna konfiguracja protokołu MCP 111
Przykładowe repozytorium i serwer MCP 112
Konfiguracja protokołu MCP na poziomie projektu 114
Inspekcja użycia kontekstu 115
Ograniczanie kontekstu przez określanie zasięgu konfiguracji protokołu MCP 116
Wymuszanie ścisłej konfiguracji protokołu MCP 117
Dynamiczne zarządzanie protokołem MCP w ramach sesji 119

Obecne ograniczenia protokołu MCP ... 120
„Zanieczyszczenie" kontekstu ... 120
Nieefektywne działanie agenta ... 122
Problem języka natywnego — format JSON a kod ... 123
Ograniczenia definicji narzędzi i schematów ... 124
Model CodeMode firmy Cloudflare ... 125
Kluczowe obserwacje ... 126
Wywoływanie narzędzi i związane z tym ograniczenia ... 126
Rola protokołu MCP ... 127
Integracja z modelem CodeMode ... 127
Przekształcanie serwera MCP do postaci kodu TypeScript ... 128
Wykonywanie kodu w „piaskownicy" ... 128
Zarządzanie konfiguracją za pomocą dodatków narzędzia Claude Code ... 130
Zalety systemu dodatków ... 130
Przeglądanie oficjalnego rynku dodatków ... 131
Dodawanie rynku dodatków do narzędzia Claude Code ... 131
Sprawdzanie źródła dodatków i kwestie bezpieczeństwa ... 132
Instalowanie dodatków oraz zarządzanie nimi ... 133
Użycie dodatku feature-dev ... 134
Rynki dodatków i zastosowania korporacyjne ... 138
Podsumowanie ... 139

ROZDZIAŁ 5
Automatyzacja procesu projektowego przy użyciu narzędzia Claude Code i serwisu GitHub ... 140

Instalacja i konfiguracja integracji narzędzia Claude Code z serwisem GitHub ... 141
Instalacja wymaganych składników ... 141
Instalacja aplikacji Claude GitHub ... 142
Instalacja procesu serwisu GitHub ... 143
Obsługa zgłoszeń w repozytorium GitHub za pomocą narzędzia Claude Code ... 146
Lista zadań do wykonania modelu Claude i wykrywanie kontekstu ... 147
Przeglądanie proponowanych zmian ... 148
Dodawanie kontekstu repozytorium za pomocą pliku CLAUDE.md ... 149
Inicjalizacja kontekstu repozytorium ... 149
Podsumowanie ... 151

ROZDZIAŁ 6
Planowanie w narzędziu Claude Code i procesy z wieloma agentami 153

Tryb planowania w narzędziu Claude Code 153
Charakterystyka trybu planowania 154
Proces planowania 154
Zalety projektowania opartego na specyfikacji 154
Przykład: iteracyjne tworzenie specyfikacji rynku hooków 155
Utrwalanie specyfikacji rynku hooków 157
Równoległe użycie wielu agentów narzędzia Claude Code 157
Identyfikowanie zadań nadających się do równoległego wykonywania 158
Przykład: równoległe prace programistyczne w projekcie HookHub 159
Przydzielanie niezależnych zadań agentom 160
Współbieżne wykonywanie zadań 161
Przeglądanie i zatwierdzanie zmian 162
Podsumowanie 163

ROZDZIAŁ 7
Korzystanie z subagentów narzędzia Claude Code 164

Wprowadzenie do subagentów narzędzia Claude Code 164
Struktura plików subagenta 168
Relacja z agentami środowiska React 169
Konfiguracja pierwszego subagenta 170
Definiowanie opisu agenta 171
Wybór narzędzi dla agenta 171
Wybór modelu i ustawień wizualnych 173
Przegląd wygenerowanej konfiguracji agenta 173
Testowanie subagenta 174
Uruchamianie subagenta do sprawdzania kodu 175
Uruchamianie wielu instancji do przeglądu kodu 175
Przepływ kontekstu podczas stosowania subagentów 176
Zastosowanie okien kontekstowych 177
Przykład: tworzenie subagenta diagramów Mermaid 179
Skalowanie za pomocą współbieżnych subagentów 192
Konfiguracja środowiska 192
Instalowanie zależności 193
Polecenie inifinite używane ze znakiem „/" 194
Reorganizacja komponentu Hero 194

Prompt z poleceniem infinite 200
Faza 1.: analiza specyfikacji 201
Faza 2.: rozpoznanie katalogu danych wyjściowych 201
Faza 3.: strategia iteracji 202
Faza 4.: koordynacja równolegle działających agentów 202
Podsumowanie 204

ROZDZIAŁ 8
Tworzenie i dostosowywanie stylów danych wyjściowych 205

Tworzenie niestandardowego stylu danych wyjściowych za pomocą narzędzia Claude Code 206
Tworzenie i przeglądanie nowego stylu danych wyjściowych 207
Jak działają style danych wyjściowych? 208
Dlaczego format YAML? 209
Tworzenie danych formatu YAML jako stylu danych wyjściowych 211
Tworzenie zaawansowanego stylu danych wyjściowych na poziomie projektu 214
Definiowanie niestandardowego stylu danych wyjściowych HTML 214
Użycie stylu danych wyjściowych HTML 215
Automatyzacja generowania plików 217
Zastosowanie wielu stylów danych wyjściowych 218
Specjalistyczne procesy stylów danych wyjściowych 220
Dostosowywanie wierszy statusu w narzędziu Claude Code 221
Definiowanie polecenia niestandardowego wiersza statusu 221
Inicjowanie konfiguracji wiersza statusu 221
Aktualizacja konfiguracji 222
Implementacja logiki wiersza statusu 223
Obsługa kolorów i formatowania standardu ANSI 224
Rozszerzanie funkcjonalności wiersza statusu 224
Zaawansowany wiersz statusu: wyświetlanie ostatniego promptu użytkownika 225
Definiowanie wymagania dla wiersza statusu 226
Testowanie i finalizowanie implementacji 229
Przyszłość tworzenia kodu za pomocą sztucznej inteligencji: zespół wyspecjalizowanych agentów 230
Agentowe tworzenie kodu i wiele instancji 230
Nazewnictwo i konteksty oparte na rolach 232
Podsumowanie 232

CZĘŚĆ 3. Zaawansowane agenty i projekt głębokiego systemu

ROZDZIAŁ 9
Umiejętności agentów 237
Istota umiejętności agentów 238
Scenariusz 1.: określanie stylu bez użycia umiejętności 239
Scenariusz 2.: określanie stylu z użyciem umiejętności 239
Szczegółowa analiza umiejętności agentów 242
Eksplorowanie rynku Claude Skills 242
Sprawdzanie skryptu 245
Tworzenie niestandardowej umiejętności na poziomie projektu 245
Użycie niestandardowej umiejętności 246
Uruchamianie umiejętności w izolowanym kontekście subagenta 252
Porównanie umiejętności agenta z jego innymi podstawowymi elementami 253
Umiejętności agenta i protokół MCP 253
Umiejętności agenta i subagenty 254
Podsumowanie 256

ROZDZIAŁ 10
Użycie aplikacji Claude Code Desktop 258
Wprowadzenie do integracji z aplikacją Claude Code Desktop i agenty działające w tle 258
Instalacja narzędzia Claude Code i tryby działania: lokalny i chmurowy 259
Wybór folderu obszaru roboczego (tryb Local worktree) 260
Przełączanie na tryb chmurowy Claude Code web 261
Jak opcja Git Worktrees umożliwia proces projektowania równoległego? 262
Orkiestracja równoległych agentów lokalnych i chmurowych 263
Lokalne uruchamianie aplikacji Next.js 264
Koordynacja równoległych działań lokalnych i w chmurze 265
Równoległe tworzenie funkcjonalności 266
Struktura drzew roboczych Git 272
Przygotowanie do scalenia artefaktów 272
Znaczenie opcji Git Worktrees w narzędziu Claude Code 272
Jak narzędzie Claude Code korzysta z drzewa roboczego? 273
Równoległe użycie wielu drzew roboczych 274

Scalanie równoległych gałęzi funkcjonalności ... 274
Przesyłanie gałęzi drzew roboczych ... 276
Scalanie wszystkiego z gałęzią project/hookhub ... 278
Przeprowadzanie scalania ... 280
Testowanie przed przesłaniem do gałęzi project/hookhub ... 282
Weryfikacja poprawności wyniku integracji ... 283
Finalizowanie scalania ... 283
Uruchamianie narzędzia Claude Code w chmurze za pośrednictwem urządzenia mobilnego ... 284
Implementacja komponentu stopki ... 286
Przeglądanie żądania przesłania ... 287
Lokalna weryfikacja zmian ... 289
Kiedy współbieżne agenty są nieproduktywne? ... 289
Podsumowanie ... 290

ROZDZIAŁ 11
Głębokie agenty ... 291
Taksonomia agentów w systemach produkcyjnych ... 291
Proste agenty i ich ograniczenia ... 293
Głębokie agenty ... 294
Co sprawia, że agent jest głęboki? ... 295
Narzędzie do planowania głębokich agentów ... 296
Subagenty i hierarchiczne delegowanie ... 297
Dostęp do systemu plików w głębokich agentach ... 299
Prompt systemowy ... 302
Podsumowanie ... 304

Rozdział 1

Inżynieria kontekstu

W niniejszym rozdziale wprowadzono zagadnienie inżynierii kontekstu i wyjaśniono, dlaczego stało się ono kluczowe przy budowaniu i korzystaniu z nowoczesnych agentów AI (sztucznej inteligencji). Choć wiele systemów AI jest często opisywanych jako „zwykłe prompty opakowujące model LLM", w tym rozdziale zostanie wyjaśnione, dlaczego takie postrzeganie przestaje się sprawdzać, gdy agenty stają się bardziej złożone, działają przez długi czas, a ponadto korzystają z narzędzi. Dowiesz się, skąd bierze się kontekst, dlaczego nieustannie się rozszerza, a także, jak jego niewłaściwe zarządzanie prowadzi do spadku wydajności, wyższych kosztów, halucynacji i niespójnego sposobu działania.

Celem tego rozdziału jest zbudowanie wyraźnego modelu mentalnego na potrzeby inżynierii kontekstu i zaprezentowanie, jak stosuje się go w praktyce. Zaczniemy od wyjaśnienia, jak inżynieria kontekstu rozwinęła się z postaci inżynierii promptów, a następnie na konkretnym przykładzie w postaci narzędzia Claude Code zostanie pokazane, w jaki sposób nowoczesne agenty tworzą, wybierają, kompresują i izolują kontekst. Pod koniec rozdziału przyjrzymy się promptom systemowym. Dowiesz się, dlaczego wciąż mają znaczenie i jak je skutecznie projektować.

W niniejszym rozdziale zostaną omówione następujące zagadnienia:

- wprowadzenie do inżynierii kontekstu,
- narzędzie Claude Code i jego filozofia związana z inżynierią kontekstu,
- prompty systemowe i powody, dla których są one istotne.

Uwaga

Niniejsze omówienie opiera się na publicznie dostępnych informacjach, w tym postach na blogu inżynierów firmy Anthropic oraz analizach członków społeczności. Nie stanowi ono oficjalnego stanowiska tej firmy.

Wprowadzenie do inżynierii kontekstu

W tym podrozdziale zostanie omówione, czym jest inżynieria kontekstu. Jeśli używałeś agentów AI, a być może pracowałeś z agentami programowania, takimi jak Cursor i Claude Code, albo nawet ewentualnie sam tworzyłeś agenty AI dla swojej firmy lub na własny użytek, prawdopodobnie wiesz, że w zasadzie wszystko sprowadza się do promptu wysyłanego do modelu LLM i sporej ilości związanej z nim inżynierii.

Jest trochę prawdy w nazywaniu takich aplikacji jak Cursor czy Claude Code jedynie składnikami opakowującymi modele LLM. Jednakże budowanie naprawdę dobrych tego typu składników wymaga dogłębnej wiedzy i ogromnego nakładu pracy inżynieryjnej. *Środowisko uruchomieniowe agenta* (ang. *agent harness*) to inny termin często używany do określenia tego otaczającego systemu. Środowisko uruchomieniowe to warstwa orkiestracji wokół modelu LLM. Odpowiada ona za zarządzanie wywołaniami narzędzi, kontrolowanie pętli agenta, obsługę błędów, wymuszanie zabezpieczeń, a przede wszystkim za decydowanie, jaki kontekst jest wysyłany do modelu w ramach każdego kroku.

W praktyce większość rzeczywistej inżynierii nie jest obecna wewnątrz samego wywołania modelu LLM, lecz w jego otoczeniu. Wywołanie modelu jest często proste. O tym, czy agent działa niezawodnie, decyduje to, jak otaczający system zarządza stanem, narzędziami, pamięcią i kontekstem. Dzieje się tak dlatego, że wywołania modeli LLM zawsze wiążą się z kontekstem, który pochodzi z różnych źródeł i trwających procesów.

Kontekst może na przykład pochodzić z wielu następujących miejsc:

- od projektanta aplikacji,
- od użytkownika,
- z wcześniejszych interakcji użytkownika, wywołań narzędzi lub innych zewnętrznych danych.

Każdego dnia pojawiają się nowe źródła kontekstu, a jego ilość stale się zwiększa. Wysyłanie właściwego i istotnego kontekstu do modelu LLM nie jest tak proste, jak nam się na początku wydawało.

Od inżynierii promptów do inżynierii kontekstu

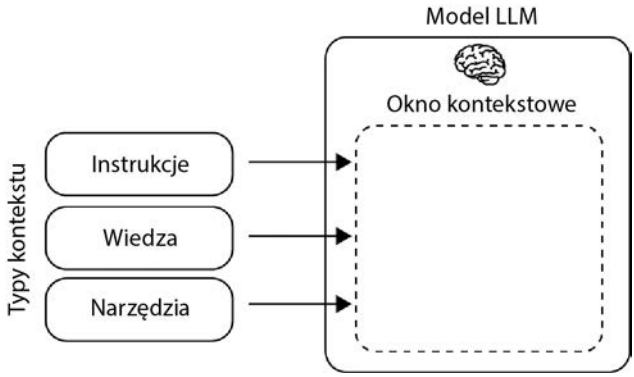
Początkowo wierzono, że inżynieria promptów wystarczy. Myśleliśmy, że wystarczy utworzyć kilka wyszukanych promptów, aby rozwiązać problem i uzyskać to, czego potrzebujemy. Problem polega jednak na tym, że prompty są statyczne, natomiast kontekst jest niezwykle dynamiczny.

Jeśli kontekst jest dynamiczny, skonstruowanie poprawnego kontekstu wymaga również dynamicznego systemu. Nie chodzi już tylko o utworzenie statycznego promptu. Z tego właśnie powodu wkraczamy w obszar **inżynierii kontekstu**. Jest to naturalna ewolucja inżynierii promptów, ale stanowi to znacznie głębszą koncepcję.

Dlaczego kontekst ma znaczenie dla agentów?

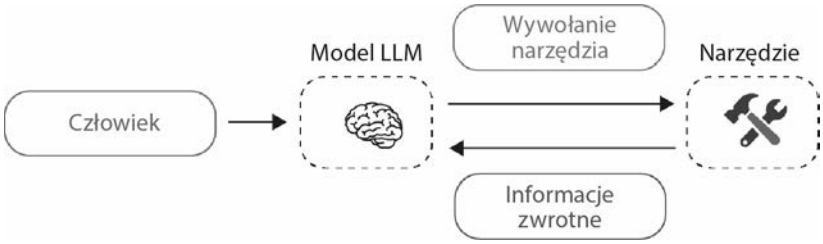
Wszyscy znamy powiedzenie *śmieci na wejściu, śmieci na wyjściu*. Jest to jeden z najczęstszych powodów, dla których systemy agentowe nie działają tak, jak powinny. Po prostu nie uzyskują one odpowiedniego kontekstu.

Duże modele językowe (LLM — *Large Language Model*) nie potrafią czytać w naszych myślach. Trzeba dostarczyć im odpowiednie informacje. Nawiasem mówiąc, nie zawsze chodzi tylko o informacje czy dane. Czasem trzeba zapewnić im właściwe narzędzia, dzięki którym mogą pozyskiwać dodatkowe informacje, podejmować działania i wykonywać zadania w naszym imieniu (rysunek 1.1).



Rysunek 1.1. Okno kontekstowe modelu LLM (opracowano na podstawie koncepcji opisywanych przez firmę LangChain, *www.langchain.com*)

Modele LLM są coraz lepsze pod względem wnioskowania. Możliwe jest wywoływanie narzędzi, a ponadto można budować agenty AI, które uruchamiają narzędzia, wywołują je, uzyskują dane wynikowe i działają w pętli do momentu zakończenia zadań (rysunek 1.2).



Rysunek 1.2. Pętla wywoływania narzędzi przez model LLM (opracowano na podstawie koncepcji opisywanych przez firmę LangChain, *www.langchain.com*)

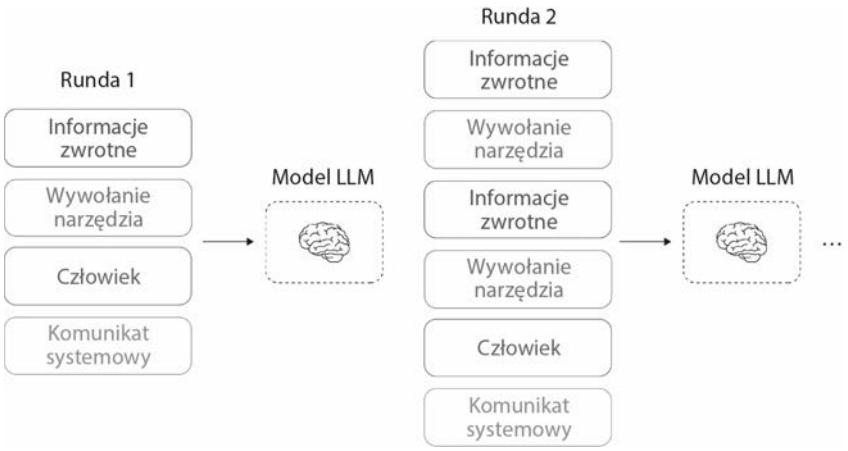
Jednakże w przypadku długotrwałych i złożonych zadań często gromadzimy informacje zwrotne w wyniku wywołań narzędzi.

Powoduje to, że okno kontekstowe stale się powiększa, wypełnione wynikami wywołań narzędzi i pośrednimi danymi wyjściowymi (rysunek 1.3).

Prowadzi to do kilku następujących problemów:

- okno kontekstowe może przekroczyć swój limit rozmiaru,
- zwiększają się koszty i opóźnienie,
- wydajność agenta zaczyna się pogarszać.

Jeśli nic z tym nie zostanie zrobione, ta degradacja stanie się nieunikniona. W ostatnich dyskusjach analizowano, dlaczego długie konteksty zawodzą w praktyce, a ponadto, jak degradacja pojawia się stopniowo w miarę gromadzenia się nieistotnych lub sprzecznych informacji. Więcej na ten temat możesz znaleźć w znakomitym artykule blogowym *How Long Contexts Fail* autorstwa Dana Breuniga (*https://www.dbreunig.com/2025/06/22/how-contexts-fail-and-how-to-fix-them.html*). Jeśli kontekst może się rozszerzać



Rysunek 1.3. Rozszerzanie się kontekstu w ramach wielu rund (opracowano na podstawie koncepcji opisywanych przez firmę LangChain, *www.langchain.com*)

bez struktury, selekcji lub kontroli, w systemach agentowych zaczynają pojawiać się różne problemy, takie jak:

- **„Zatrucie" kontekstu**. Ma miejsce, gdy halucynacja wynikająca z wywołania narzędzia lub modelu LLM trafia do kontekstu i zaczyna wpływać na przyszłe dane wyjściowe.
- **Dezorientacja kontekstowa**. Występuje, gdy zbędny kontekst wpływa na odpowiedź nawet pomimo tego, że nie jest istotny dla danego zadania.
- **Konflikt kontekstu**. Pojawia się, gdy różne części kontekstu wzajemnie sobie przeczą.

W następnym podrozdziale omówimy techniki lepszej inżynierii kontekstu. Niektóre z nich są implementowane przez twórców aplikacji (na przykład w takich narzędziach jak Claude Code). Inne techniki są obecne po stronie użytkownika.

Jako użytkownicy narzędzia Claude Code mamy duży wpływ na kontekst, który ostatecznie trafia do modelu LLM, a tym samym na otrzymywane odpowiedzi. Oznacza to, że nawet osoby niebędące programistami muszą zrozumieć inżynierię kontekstu, jeśli chcą uzyskiwać lepsze odpowiedzi od systemów i agentów AI.

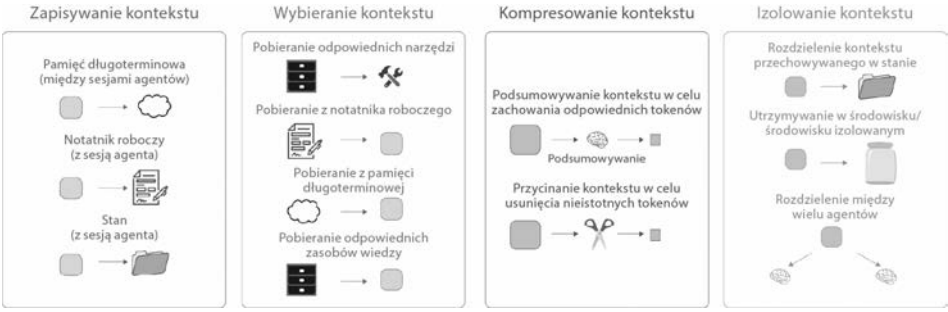
Agenty programowania, takie jak Claude Code, są doskonałym przykładem tego, jak techniki inżynierii kontekstu są stosowane zarówno po stronie programisty, jak i użytkownika. Dokładnie to zostanie przeanalizowane w następnym podrozdziale, a także to, jak lepiej formułować używany kontekst.

Narzędzie Claude Code i jego strategie inżynierii kontekstu

W tym podrozdziale omówimy narzędzie Claude Code, jego filozofię związaną z inżynierią kontekstu oraz sposoby, w jakie radzi sobie ono z przedstawionymi trudnościami.

Narzędzie to stosuje te strategie inżynierii kontekstu, implementując wszystkie cztery następujące podejścia:

- **Zapisywanie kontekstu**. Systemy pamięci trwałej.
- **Wybieranie kontekstu**. Inteligentne uzyskiwanie i wstawianie kontekstu.
- **Kompresowanie kontekstu**. Skuteczne reprezentowanie i podsumowywanie kontekstu.
- **Izolowanie kontekstu**. Zarządzanie kontekstem z użyciem wielu agentów przy ograniczonym zasięgu (rysunek 1.4).



Rysunek 1.4. Kategorie inżynierii kontekstu (opracowano na podstawie koncepcji omawianych przez firmę LangChain, www.langchain.com)

Zacznijmy od pierwszego podejścia.

Strategia 1.: zapisywanie kontekstu i pamięć trwała

Pierwsza strategia dotyczy **zapisywania kontekstu i jego architektury pamięci trwałej**. Model Claude dysponuje wielowarstwowym systemem pamięci, a narzędzie Claude Code implementuje trójwarstwową hierarchię pamięci, która zachowuje kontekst między sesjami podczas programowania z użyciem tego rozwiązania.

- Jako pierwsza pojawia się **pamięć projektu** (plik *./CLAUDE.md*). Jest to współdzielony przez zespół kontekst dotyczący architektury projektu, standardów lub wszystkiego, co jest związane z konkretnym projektem. Ten rodzaj pamięci podlega kontroli wersji i dostępny jest dla wszystkich członków zespołu.

```
# Project Context

## Architecture Overview
This is a microservices application using:
- Node.js with Express for API services
- React with TypeScript for frontend
- PostgreSQL with Prisma ORM
- Redis for caching
```

```
## Coding Standards
- Use functional components with hooks
- Implement error boundaries for all route components
- Follow RESTful API conventions
- Write unit tests for all business logic
```

- W dalszej kolejności jest **pamięć użytkownika** (plik *~/.claude/CLAUDE.md*). Jest ona przechowywana w pliku *CLAUDE.md* znajdującym się w folderze *.claude* katalogu domowego użytkownika. Plik zawiera osobiste preferencje i skróty obowiązujące we wszystkich projektach danego użytkownika. Nie jest on zatwierdzany w serwisie GitHub i powiązany jest z konkretnym użytkownikiem. Każdy użytkownik będzie używał tutaj innych wartości, a plik ten jest utrzymywany między wszystkimi sesjami narzędzia Claude Code.

```
# Personal Development Preferences

## Code Style
- Always use explicit return types in TypeScript
- Prefer const assertions over type annotations
- Use descriptive variable names, avoid abbreviations

## Workflow Shortcuts
- When writing tests, use Jest with React Testing Library
- Always run
`

npm

 run lint`

 before commits
- Prefer composition over inheritance
```

- Na końcu znajdują się **dynamiczne importy pamięci**. Pozwalają one na importowanie danych z innych plików pamięci za pomocą symbolu „@” i składni. Działa to podobnie do ładowania zwykłego kontekstu do narzędzia Claude Code, ale tym razem mogą istnieć dedykowane pliki pamięci z konkretnymi informacjami, do których można się odwoływać wewnątrz używanych plików pamięci.

```
# W dowolnym pliku CLAUDE.md
@sciezka/do/pliku/pamieci.md
@./sciezka/wzgledna/kontekst.md
@~/globalny/kontekst/uzytkownika.md
```

Możemy również dostosowywać sposób działania, pisząc skrypty, które dynamicznie aktualizują kontekst w zależności od gałęzi repozytorium Git. Taki skrypt może następnie zostać połączony z hookiem przełączania kontekstu. Dzięki temu budujemy bardziej adaptacyjne procesy.

Ważnym szczegółem, o którym warto pamiętać, jest to, że powinniśmy unikać wielokrotnego dołączania do pliku *CLAUDE.md* tych samych odwołań do kontekstu. Jeśli skrypt po prostu stosuje ciąg `>>` przy każdym uruchomieniu, plik będzie powiększał się w nieskończoność o zduplikowane pozycje. Aby temu zapobiec, dodajemy proste zabezpieczenie, które sprawdza, czy dane odwołanie do kontekstu już istnieje, zanim zostanie dołączone.

Oto ulepszona wersja skryptu, który unika duplikowania importów:

```
# Script to dynamically update context based on git branch
#!/bin/bash
# context-switcher.sh
# Dynamically load relevant context based on user query
# Safe against duplicate imports

CLAUDE_MD="CLAUDE.md"

# Create CLAUDE.md if it doesn't exist
touch "$CLAUDE_MD"

add_context() {
  local context_ref="$1"
  grep -qxF "$context_ref" "$CLAUDE_MD" || echo "$context_ref" >> "$CLAUDE_MD"
}

# --- Branch-based context ---
branch=$(git branch --show-current 2>/dev/null)

case $branch in
  "feature/auth-"*)
    add_context "@./context/auth-system.md"
    ;;
  "feature/payment-"*)
    add_context "@./context/payment-flow.md"
    ;;
  "hotfix/"*)
    add_context "@./context/production-hotfix.md"
    ;;
esac

# --- Query-based context ---
user_input="$1"

if [[ -n "$user_input" ]]; then
  if [[ $user_input == *"database"* || $user_input == *"migration"* ]]; then
    add_context "@./context/database-context.md"
  elif [[ $user_input == *"API"* || $user_input == *"endpoint"* ]]; then
    add_context "@./context/api-context.md"
  elif [[ $user_input == *"frontend"* || $user_input == *"component"* ]]; then
    add_context "@./context/frontend-context.md"
  fi
fi
```

Kluczową zmianą jest funkcja pomocnicza `add_context`. Używa ona polecenia `grep -qxF`, aby sprawdzić, czy odwołanie do kontekstu już istnieje w pliku *CLAUDE.md*. Jeśli tak, nic nie jest dołączane. Dzięki temu skrypt jest idempotentny i można go bezpiecznie uruchamiać przy każdym wysłaniu promptu.

W dalszej kolejności skrypt ten jest łączony z hookiem:

```
{
  "hooks": {
    "UserPromptSubmit": {
```

```
        "command": "./scripts/context-switcher.sh \"$PROMPT\""
        "description": "Dynamically load relevant context based on
          branch and user query"
      }
    }
  }
```

W przypadku takiej konfiguracji przełączanie kontekstu odbywa się dynamicznie, ale pozostaje stabilne z upływem czasu.

Strategia 2.: inteligentne uzyskiwanie kontekstu

Druga strategia to **inteligentne uzyskiwanie kontekstu**, które zwykle jest realizowane w ramach dynamicznego wykrywania kontekstu.

Model Claude automatycznie przeszukuje foldery w poszukiwaniu przydatnych plików kontekstowych. Jeśli wybrano podfolder, model pobiera również kontekst z folderów nadrzędnych. Gdy jednak dostępne są bardziej szczegółowe informacje, to one są używane. Dodatkowo model priorytetowo traktuje informacje ostatnio stosowane i często żądane.

Jest to inżynieria kontekstu na poziomie aplikacji. Logikę tę implementują deweloperzy korzystający z narzędzia Claude Code. Jednakże użytkownicy również mogą samodzielnie dodawać trwały kontekst za pomocą polecenia `/memory`. Oto przykład:

```
/memory add Always use descriptive variable names
```

Model Claude zapyta, czy konkretna pamięć powinna być przechowywana na poziomie projektu czy użytkownika, a następnie automatycznie zaktualizuje odpowiedni plik *CLAUDE.md*. Dzięki temu użytkownicy mogą wpływać na przyszłe działanie modelu bez ręcznego edytowania plików kontekstowych.

W zależności od używanego narzędzia do modelu LLM przekazywane są różne konteksty. Gdy na przykład narzędzie Claude Code ma zamiar edytować plik, automatycznie przypomina sobie o takich rzeczach jak sprawdzenie najpierw istniejącego stylu kodu oraz poszukiwanie istniejących funkcji przed utworzeniem nowych. Jest to kontekst istotny dla narzędzia edycji.

Inny przykład to sytuacja, w której narzędzie Claude Code ma wykonać polecenie w terminalu. W tym przypadku pamięta ono o innych rzeczach, takich jak sprawdzenie przed uruchomieniem poleceń, czy istnieje już skrypt programu `npm`, albo upewnienie się przed wykonaniem pliku, czy jego ścieżka jest poprawna.

Strategia 3.: kompresowanie kontekstu

Trzecią strategią jest **kompresowanie kontekstu**, czyli efektywna reprezentacja informacji kontekstowych. Narzędzie Claude Code ma wbudowane polecenia do kompresowania.

- Polecenie `/clear` resetuje historię konwersacji w bieżącym oknie kontekstowym, zachowując jednocześnie podstawową pamięć projektu i pamięć użytkownika.

Przydaje się ono, gdy bieżąca konwersacja zmierza w niepożądanym kierunku lub wtedy, gdy postanowisz rozpocząć nową interakcję, nie tracąc tego, czego system już się nauczył o projekcie.

- Polecenie `/compact` podsumowuje dotychczasową konwersację w postaci krótszej formy. Zamiast pozbywać się wszystkiego, zachowuje ono kluczowe decyzje i istotne informacje, pomijając mniej ważne szczegóły. Skompresowana historia zajmuje mniej miejsca w oknie kontekstowym, co zmniejsza koszty i opóźnienia, a jednocześnie zapewnia modelowi więcej przestrzeni na nowe dane wejściowe.

Strategia 4.: izolowanie kontekstu za pomocą subagentów

Czwartą strategią jest izolowanie kontekstu. W tym celu narzędzie Claude Code korzysta z subagentów. Każdy subagent działa we własnym, odizolowanym oknie kontekstowym i nie dziedziczy pełnej historii konwersacji głównego agenta. Strategia ta polega na tworzeniu różnych wyspecjalizowanych wersji narzędzia Claude Code do wykonywania różnorodnych zadań, z których każda dysponuje własną, ukierunkowaną wiedzą.

Zamiast próbować robić za pomocą narzędzia Claude Code wszystko naraz z całością dostępnych informacji, tworzymy wyspecjalizowanych subagentów, którzy pełnią funkcję ekspertów w różnych obszarach. Możemy dysponować głównym agentem Claude, który działa jako menedżer i deleguje zadania. Można zastosować agenta do sprawdzania kodu, który skupia się na jakości kodu i bezpieczeństwie agenta testowego koncentrującego się na tworzeniu i uruchamianiu testów, a także agenta badawczego zajmującego się wyszukiwaniem informacji i najlepszych praktyk. Oto przykład:

```
// Code review agent – focused context
Task(
  description: "Code review",
  prompt: "Review this PR focusing only on security and performance",
  subagent_type: "code-reviewer"
)

// Research agent – broad context
Task(
  description: "Research implementation",
  prompt: "Find best practices for OAuth2 implementation",
  subagent_type: "general-purpose"
)
```

Jest to pomocne, ponieważ w przypadku braku izolacji narzędzie Claude Code jest zdezorientowane, próbując robić wszystko naraz, mając dostęp do wszystkich możliwych informacji. Dzięki izolacji każdy specjalista dysponuje konkretnym zakresem wiedzy i wykorzystuje ją do skutecznego wykonania swojego zadania.

Uwaga

Powyższa składnia ma wyłącznie cel ilustracyjny i została uproszczona na potrzeby demonstracji. Narzędzie Claude Code nie stosuje obecnie dokładnie takiego formatu. W praktyce definiowanie subagentów wymaga konfiguracji formatu YAML z sekcją frontmatter, co zostanie omówione w dalszej części książki.

Prompty systemowe i powody ich znaczenia

W tym podrozdziale omówimy prompty systemowe i ich znaczenie w kontekście inżynierii kontekstu. Prawdopodobnie tysiąc razy czytałeś w serwisach X i LinkedIn, że prompty systemowe są ważne, a ponadto, że trzeba nad nimi pracować, iterować i dopracowywać je, aby były naprawdę dobre. Stwierdzenie „miej dobry prompt systemowy" to prawdopodobnie najbardziej ogólna rada w obszarze inżynierii sztucznej inteligencji.

Istnieje kilka publicznych repozytoriów, które zbierają przykłady promptów systemowych używanych przez dobrze znane agenty AI. Repozytoria te zazwyczaj agregują prompty, które zostały publicznie udostępnione lub omówione w publicznych źródłach. Wiele przykładów koncentruje się na agentach programowania, takich jak Claude Code, Cursor i Devin, ale mogą też zawierać prompty z innych systemów agentowych.

Celem odwoływania się do tych repozytoriów nie jest weryfikacja autentyczności każdego promptu, lecz zilustrowanie, jak duże, ustrukturyzowane i starannie zaprojektowane prompty systemowe bywają w praktyce.

Celem tego podrozdziału nie jest szczegółowa analiza każdego promptu ani wyjaśnianie każdej zastosowanej techniki. Samym promptom systemowym można by poświęcić całą książkę lub kurs.

Moim celem jest tutaj pokazanie, że prompty systemowe są istotne. Prompty te nieustannie ewoluują. Wraz z ewolucją modeli LLM rozwijają się prompty systemowe. Ogromny wysiłek inżynieryjny wkładany jest w ich opracowywanie i ciągłe ulepszanie. Jest to proces iteracyjny.

Najlepsze praktyki tworzenia promptów systemowych

Omówmy najlepsze praktyki dotyczące opracowywania promptów systemowych i zajmowania się nimi. Przypomina to trochę przekazywanie komuś wskazówek dotyczących dojazdu.

Jeśli powiesz komuś tylko: „Jedź tam", nie będzie to precyzyjne. Osoba ta nie będzie wiedzieć, dokąd ma jechać. Jeśli jednak przekażesz jej 50-stronicową instrukcję opisującą każdy możliwy zakręt i każdą ulicę, przytłoczysz ją nadmiarem informacji, przez co nadal nie dotrze ona tam, gdzie chce.

Wskazane jest zatem zadbanie o przejrzystość, konkretność i podanie dokładnie tylu informacji, ile potrzeba, żeby ktoś dotarł do celu. Trudność sprawia znalezienie tej optymalnej ilości informacji.

Ekosfera dla promptów systemowych

Tworząc prompty systemowe, szukamy czegoś, co firma Anthropic określa mianem **ekosfery** (ang. *Goldilocks zone*). Niezbyt ogólnie, nie za szczegółowo, lecz w sam raz.

Możesz to sobie wyobrazić jako skalę. Na jej lewym końcu znajdują się prompty zbyt dokładne. Na prawym końcu skali są prompty zbyt ogólne. To, czego szukamy, jest dokładnie w środku (rysunek 1.5).

Rysunek 1.5. Ekosfera dla promptów systemowych (źródło: firma Anthropic, Effective Context Engineering for AI Agents, *https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents*)

Przeanalizujmy to.

Problem ze zbyt szczegółowymi promptami

Po lewej stronie skali są bardzo dokładne prompty. Główny problem polega na tym, że model LLM traktujemy jak deterministyczną maszynę stanów, a nie jak inteligentnego agenta. Logika jest ustalana na sztywno.

Można na przykład stwierdzić: jeśli intencją użytkownika jest rozstrzygnięcie incydentu, zadaj trzy pytania uzupełniające. *Dlaczego akurat trzy? Co będzie, jeśli wystarczą dwa? A co, jeśli potrzebujemy pięciu pytań?*

Spotkać się można też z drobiazgowym wyliczeniem, w przypadku którego próbuje się wymienić każdy możliwy scenariusz eskalacji. Takiej listy nie da się skompletować, a ponadto wymusza ona na modelu podążanie za z góry ustalonymi ścieżkami, które mogą nie pasować do faktycznych danych wejściowych. Staje się to też koszmarem z punktu widzenia utrzymywania, gdyż każdy nowy skrajny przypadek wymaga zmian w prompcie.

Jeśli w tym momencie wszystko jest z góry ustalone, być może w ogóle zbędny jest autonomiczny agent. Może wystarczyłby deterministyczny proces.

Problem ze zbyt ogólnymi promptami

Na przeciwnym końcu spektrum znajdują się bardzo ogólne prompty. Zasadniczy problem polega na tym, że dostarczają one zbyt mało informacji, by zapewnić spójne działanie.

W takich promptach często brakuje praktycznych wskazówek. Rozważmy na przykład następujący prompt:

You are a bakery assistant. You should attempt to solve customers' issues in a manner consistent with the principles and essence of the company brand. Escalate to a human if needed.

Jakie są to zasady?

Problemem jest błędne założenie istnienia wspólnego kontekstu. Prompt zakłada, że model zna firmę, markę i normy obsługi klienta, a tak nie jest.

Widoczne są także niezdefiniowane granice, takie jak: „W razie potrzeby przekaż sprawę człowiekowi". Kiedy jest to konieczne? Model nie ma możliwości stwierdzenia tego.

Brakuje schematu lub struktury, które pozwalałyby na podejście do problemów w sposób systematyczny. Prowadzi to do niespójnego działania, w ramach którego różne uruchomienia zapewniają bardzo odmienne podejścia do tego samego problemu. W zasadzie prompt instruuje: „Zrób to w odpowiedni sposób", nie definiując, co „odpowiedni" właściwie oznacza.

Jak wygląda dobry prompt stanowiący kompromis?

Pora przyjrzeć się kompromisowi. Rozważmy przykład:

You are a customer support agent for Claude's Bakery.

You specialize in assisting customers with their orders and basic questions about the bakery. Use the tools available to you to resolve issues efficiently and professionally.

You have access to order management systems, product catalogs, and store policies. Your goal is to resolve issues quickly when possible. Start by understanding the complete situation before proposing solutions, and ask follow-up questions if you do not understand.

Response Framework:

1. *Identify the core issue – Look beyond surface complaints to understand what the customer actually needs.*
2. *Gather necessary context – Use available tools to verify order details, check inventory, or review policies before responding.*
3. *Provide clear resolution – Offer concrete next steps with realistic timelines.*
4. *Confirm satisfaction – Ensure the customer understands the resolution and knows how to follow up if needed.*

Guidelines:

- *When multiple solutions exist, choose the simplest one that fully addresses the issue.*
- *Check the status of an order if a user mentions it before suggesting next steps.*
- *Call the human assistance tool.*
- *For legal issues, health/allergy emergencies, or situations requiring financial adjustments beyond standard policies, call the human assistance tool.*
- *Acknowledge frustration or urgency in the user's tone and respond with appropriate empathy.*

Można tu dostrzec następujące rzeczy:

- Zaczynamy od jasnego określenia tożsamości i zakresu działania. Wyznacza to natychmiast granice. Jest to obsługa klienta, a nie marketing lub sprzedaż. Obejmuje ona zamówienia i podstawowe pytania, a nie złożone operacje biznesowe.
- Zamiast ograniczać, oferujemy też swobodę działania. Zamiast precyzyjnie określać z góry, jakiego narzędzia użyć w każdej sytuacji, definiujemy cel, taki jak skuteczne i profesjonalne rozwiązanie problemu. Heurystyka polega tu na tym, że ufamy agentowi w tym, że w razie potrzeby sam wybierze odpowiednie narzędzia.
- Dostarczamy także strukturę rozumowania zamiast schematu blokowego. Stosowany jest tu schemat odpowiedzi złożony z 4 kroków: identyfikacja głównego problemu, określenie niezbędnego kontekstu, zaproponowanie jasnego rozwiązania i potwierdzenie zadowolenia klienta. Tego rodzaju schemat powinien się sprawdzać w różnych scenariuszach, a nie stanowić sztywnej logiki rozgałęzień.
- I wreszcie, ustaliliśmy również wyraźne granice i zasady. Jeśli na przykład istnieje wiele rozwiązań, wybierz najprostsze. Jest to heurystyka, a nie ścisła reguła, która przypomina podejście zachłanne stosowane w informatyce.

Zbyt szczegółowy prompt próbuje myśleć za model LLM i zawodzi, gdy sytuacja nie pasuje do skryptu. Zbyt ogólny prompt nie zapewnia modelowi wystarczająco dużo informacji, aby mógł działać.

Optymalny prompt wykorzystuje to, w czym nowoczesne modele LLM są naprawdę dobre, czyli rozpoznawanie wzorców i stosowanie ogólnych zasad w konkretnych sytuacjach. Taki prompt dobrze radzi sobie z nowymi sytuacjami, ponieważ uczy zasad, a nie reguł. Jest wydajny, gdyż nie marnuje słów. Każda wytyczna obejmuje wiele scenariuszy. Zasady są zwięzłe, a instrukcje nie nakładają się na siebie ani sobie nie przeczą.

Podsumowanie

W tym rozdziale wprowadzono pojęcie inżynierii kontekstu i wyjaśniono, dlaczego jest ona niezbędna do budowania niezawodnych i skalowalnych agentów AI. Przeanalizowano, czym kontekst różni się od promptów statycznych, skąd pochodzi i dlaczego brak zarządzania kontekstem prowadzi do takich problemów jak spadek wydajności, halucynacje czy niespójne działanie. Na praktycznym przykładzie obejmującym użycie narzędzia Claude Code omówiono 4 kluczowe strategie inżynierii kontekstu: zapisywanie kontekstu przy użyciu pamięci trwałej, dynamiczne wybieranie odpowiedniego kontekstu, kompresowanie kontekstu w celu zapewnienia możliwości zarządzania nim oraz izolowanie kontekstu za pomocą wyspecjalizowanych subagentów. Przybliżono też rolę promptów systemowych, ukazując, jak skuteczne prompty zachowują równowagę między zbyt sztywnymi a zbyt ogólnymi instrukcjami. Na tym etapie powinieneś już mieć wyobrażenie o tym, jak współczesne agenty AI zarządzają kontekstem i w jaki sposób zarówno deweloperzy, jak i użytkownicy mogą wpływać na ich zachowanie. Na bazie tych podstawowych informacji w następnym rozdziale zaczniemy używać bardziej zaawansowanych

procesów agentowych, wykorzystując projekt HookHub do sprawdzenia, jak narzędzie Claude Code koordynuje zadania, zarządza kontekstem i w swoim działaniu wykracza poza proste, jednoetapowe interakcje.

W kolejnym rozdziale przyjrzymy się bliżej narzędziu Claude Code i temu, jak wchodzi w interakcję z bazą kodu.

Opanuj programowanie agentowe z Claude Code!

Sztuczna inteligencja rewolucjonizuje sposób tworzenia oprogramowania, a narzędzia oparte na agentach AI stają się nieodzownym elementem warsztatu współczesnego programisty. Claude Code — jeden z najpopularniejszych agentów programistycznych — pozwala automatyzować złożone zadania projektowe, od generowania kodu po zarządzanie całymi procesami CI/CD. Jednak większość użytkowników korzysta z zaledwie ułamka jego możliwości. Tymczasem prawdziwa siła tego narzędzia ujawnia się dopiero wtedy, gdy traktujemy je jako rozszerzalną platformę agentową z zaawansowanymi mechanizmami kontroli kontekstu, orkiestracją wielu agentów i strukturalnymi procesami automatyzacji.

Książka przedstawia pełne spektrum możliwości Claude Code — od podstawowej konfiguracji i poleceń, przez zarządzanie pamięcią trwałą i hookami, aż po zaawansowane wzorce z wieloma agentami i głębokie procesy programistyczne. Autor szczegółowo omawia protokół Model Context Protocol (MCP) jako fundament współczesnego ekosystemu agentowego, wyjaśnia różnice między umiejętnościami, subagentami i serwerami MCP. Praktyczne przykłady oparte na projekcie HookHub w środowisku Next.js ilustruje rzeczywistymi zastosowaniami omawianych technik. Książka kończy się analizą architektury głębokich agentów — systemów zdolnych do realizacji długoterminowych, wieloetapowych zadań programistycznych z zachowaniem jakości i niezawodności.

Najważniejsze zagadnienia:

- Inżynieria kontekstu i zarządzanie pamięcią trwałą w systemach agentowych
- Protokół MCP jako warstwa integracji narzędzi i źródeł danych
- Automatyzacja procesów za pomocą hooków, umiejętności i niestandardowych poleceń
- Orkiestracja wielu agentów z użyciem subagentów i Git Worktrees
- Projektowanie oparte na specyfikacjach i tryb planowania
- Architektura głębokich agentów do zadań długoterminowych
- Integracja z GitHub Actions i aplikacją Claude Desktop

Eden Marco jest specjalistą do spraw dużych modeli językowych (LLM) w Google Cloud i ambasadorem firmy LangChain. Ma wieloletnie doświadczenie w inżynierii oprogramowania i architekturze chmurowej, był jednym z pierwszych inżynierów w firmie Orca Security. Absolwent informatyki na uczelni Technion, wykładowca na Uniwersytecie Reichmana. Tworzy praktyczne kursy gotowe do zastosowania w środowisku produkcyjnym — dzieli się nimi za pośrednictwem platformy Udemy i własnej witryny.

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 978-83-289-4089-5
9 788328 940895
Cena: 89,00 zł
packt