O'REILLY®

Helion

Vibe coding

i przyszłość kodowania

Od programisty
do dewelopera ery AI

Addy Osmani

Tytuł oryginału: Beyond Vibe Coding: From Coder to AI-Era Developer

Tłumaczenie: Piotr Pilch

ISBN: 978-83-289-3594-5

Authorized Polish translation of the English edition of *Beyond Vibe Coding*
ISBN 9798341634756 © 2025 Addy Osmani

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

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

Printed in Poland.

Spis treści

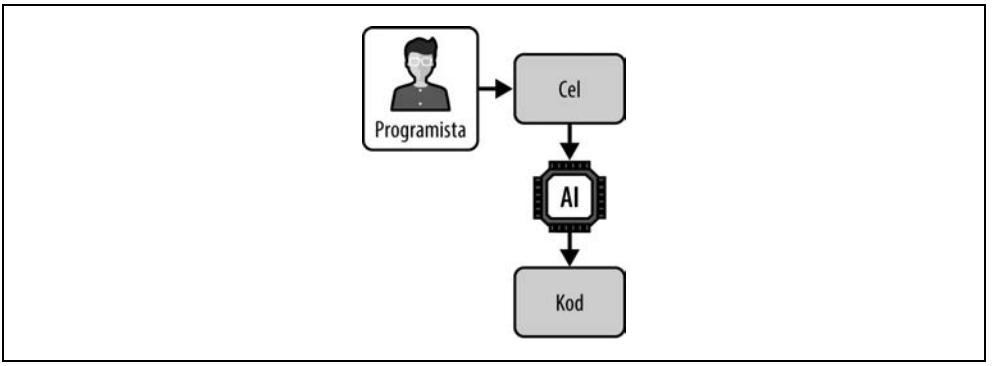
ROZDZIAŁ 1.

Wprowadzenie: czym jest vibe coding?

Sztuczna inteligencja (AI — *Artificial Intelligence*) zmienia sposób, w jaki tworzy się oprogramowanie, wprowadzając nowe modele programowania (kodowania), począwszy od swobodnego formułowania zapytań (*promptów*), a skończywszy na strukturalnym wspomaganiu. Wyobraź sobie proces tworzenia oprogramowania polegający na zwykłym *opisywaniu* tego, co ma ono zrealizować (niemal jak podczas rozmowy z kolegą z zespołu), w przypadku którego sztuczna inteligencja przekształca te pomysły do postaci kodu. Właśnie to jest istotą *vibe codingu,* czyli *programowania* lub *kodowania intuicyjnego* — eksploracyjnego podejścia opartego przede wszystkim na zapytaniach, w których za pomocą języka naturalnego opisujesz swoje zamierzenia, a duży model językowy (LLM — *Large Language Model*) wypełnia „puste miejsca". Termin ten został niedawno wymyślony przez Andreja Karpathy'ego, pioniera sztucznej inteligencji, aby opisać ten nowy sposób programowania, w przypadku którego programiści „całkowicie poddają się intuicji" wynikającej ze wspomagania przez technologię AI.

W książce zagłębię się w to, co vibe coding oznacza dla zawodowych programistów i jak wypada on w porównaniu z tym, co nazywam *inżynierią wspomaganą przez technologię AI*, czyli bardziej formalnym procesem rozszerzonego tworzenia kodu. Przeanalizuję, jak ewoluuje rola programisty w tej erze dominacji technologii AI, jakie narzędzia i przepływy pracy mogą zmaksymalizować Twoją skuteczność, a także jak radzić sobie z wyjątkowymi wyzwaniami wynikającymi z zezwolenia sztucznej inteligencji na swobodne posługiwanie się Twoją bazą kodu. Przyjrzę się również temu, gdzie vibe coding świetnie sobie radzi, gdzie napotyka trudności i jak zrównoważyć szybkość generowania kodu przez AI z mądrością towarzyszącą nadzorowi człowieka. Na koniec powinieneś mieć jasny obraz tego, jak odpowiedzialnie i skutecznie wykorzystać „intuicję" w ramach własnej praktyki programistycznej, aby w erze technologii AI stać się nie tylko szybszym programistą, ale także bardziej kreatywnym i wpływowym inżynierem oprogramowania.

W tym rozdziale przyjrzymy się temu, jak rola programisty przekształca się z tworzenia szczegółowych instrukcji dla komputerów we współpracę z technologią AI poprzez wyrażanie celu (rysunek 1.1). Dowiesz się, dlaczego ta zmiana oparta na intuicji w zakresie programowania jest tak istotna, jak sprawdza się ona na ogólnym poziomie, a także jakie niesie ze sobą możliwości i wyzwania.



Rysunek 1.1. Koncepcyjne przedstawienie programowania z wykorzystaniem celu. Programista dostarcza bardzo ogólną specyfikację (cel), a sztuczna inteligencja przekształca ją w kod. Podkreśla to przejście od tworzenia kodu wiersz po wierszu do kierowania generowaniem kodu na ogólnym poziomie

Spektrum programowania z użyciem AI: od vibe codingu do inżynierii wspomaganej sztuczną inteligencją

W ciągu ostatniego roku obserwowałem, jak dokonuje się fascynujący podział w sposobie, w jaki programiści zajmujący się tworzeniem aplikacji internetowych, a zwłaszcza ci o średnim i zaawansowanym poziomie, uwzględniają sztuczną inteligencję w swoim przepływie pracy. Na jednym końcu spektrum znajduje się vibe coding. Na drugim końcu jest to, co będę określać mianem *inżynierii wspomaganej przez technologię AI* (ang. *AI-assisted engineering*), czyli cechująca się dyscypliną metoda wplatania w ramach wyraźnych ograniczeń sztucznej inteligencji w każdą fazę procesu tworzenia oprogramowania, począwszy od projektowania, a skończywszy na testowaniu. Oba podejścia wykorzystują potężną sztuczną inteligencję, ale ich cele, odbiorcy i oczekiwania różnią się znacząco. W książce przeanalizuję te dwa skrajne podejścia i to, co oznaczają one z punktu widzenia współczesnego projektowania aplikacji internetowych.

Vibe coding: tworzenie kodu w ramach rozmowy

W vibe codingu korzystasz z wydajnych, dużych modeli językowych, które pełnią rolę partnerów podczas kodowania. Mogą one zająć się ciężką pracą polegającą na generowaniu kodu. Dzięki temu możesz skupić się na celach wyższego poziomu. Jak ujęto to w jednym z podsumowań magazynu „Business Insider" (*https://www.businessinsider.com/vibe-coding-ai-silicon-valley-andrej-karpathy-2025-2*), vibe coding „oznacza używanie narzędzi AI […] do ciężkiej pracy związanej z pisaniem kodu, aby szybko tworzyć oprogramowanie". Jak stwierdził Jensen Huang, prezes firmy NVIDIA, dzięki technologii AI „najgorętszym, nowym językiem programowania" jest angielski, a nie język Java lub Python. Zamiast ręcznie wpisywać kod każdej funkcji i poprawki błędu, prowadzisz interakcję z AI w języku naturalnym. Tworzysz zarys funkcjonalności, oceniasz sugestie i iterujesz na podstawie wyników utworzonych przez technologię AI.

Takie podejście reprezentuje dramatyczną zmianę z tradycyjnego programowania na projektowanie wspomagane przez AI. Konwencjonalne programowanie wymaga starannego planowania, dokładnej składni i często żmudnego debugowania. Vibe coding odwraca ten scenariusz: „To nie jest tak naprawdę pisanie kodu. Po prostu widzę różne rzeczy, mówię o nich, uruchamiam je oraz kopiuję i wklejam. I przeważnie to działa", tak żartował Karpathy w wypowiedzi dla magazynu „Business Insider", podkreślając, jak technologia AI może przekształcić ogólne instrukcje w działający kod przy minimalnym nakładzie ręcznych działań.

Programiści przechodzą od tworzenia szczegółowych instrukcji dla komputerów do *orkiestracji wyników* z pomocą technologii AI. Jako przykład Karpathy opisuje (*https://x.com/karpathy/status/1886192184808149383*) budowanie aplikacji internetowej w ramach procesu ciągłego akceptowania sugestii modelu AI: „Zawsze wszystko akceptuję, nie zaznajamiam się już z różnicami […] Po pojawieniu się komunikatów o błędach po prostu je kopiuję i wklejam […] Czasami modele LLM nie potrafią naprawić błędu, dlatego po prostu go omijam lub proszę o dokonanie losowych zmian do momentu, aż on zniknie". Kod przekracza wielkość, jaką normalnie miałby, gdybym napisał go sam, a jednak projekt szybko tworzy jedną całość dzięki iteracyjnemu tworzeniu zapytań i poprawianiu. Zasadniczo vibe coding traktuje tworzenie kodu jako interaktywną rozmowę prowadzoną z Twoim programistą w postaci modelu AI, a nie jako harówkę w pojedynkę polegającą na przebijaniu się przez składnię i ślady stosu. Celem są szybkość i eksploracja, co pozwala uzyskać działające rozwiązanie przy minimalnym stopniu utrudnień.

Zbiegło się kilka trendów umożliwiających takie programowanie intuicyjne. Po pierwsze, nowoczesne asystenty AI do tworzenia kodu (na przykład OpenAI Codex, ChatGPT, Anthropic Claude) stały się zdumiewająco dobre w generowaniu i poprawianiu kodu. W tym samym poście Karpathy zauważa, że jest to „możliwe, ponieważ modele LLM-y […] stają się zbyt dobre". Pozyskały ogromne obszary serwisu GitHub z kodem i mogą produkować wiarygodne rozwiązania w przypadku wielu zadań.

Po drugie, pojawiły się nowe narzędzia do projektowania, które bezproblemowo integrują te modele z przepływem pracy tworzenia kodu (więcej o tych narzędziach wkrótce). I wreszcie, nastawienie społeczności programistów ewoluuje w kierunku zaufania asystentom AI w ramach coraz większych fragmentów wykonywanej pracy. To już nie jest tylko autouzupełnianie na sterydach. Jest to przekazywanie modelowi AI całych funkcji lub plików. Z praktycznego punktu widzenia vibe coding często przypomina posiadanie nieograniczonej liczby początkujących programistów chętnych do pracy, których można zaangażować do implementacji wszystkiego, o co poprosisz, z tym że działają oni z prędkością obliczeń wykonywanych w chmurze.

Jedną z najbardziej oszałamiających obietnic vibe codingu jest wzrost produktywności. Pierwsi użytkownicy tego rozwiązania zgłaszają, że są w stanie tworzyć funkcjonalności oprogramowania lub prototypy od dziesięciu do stu razy szybciej niż wcześniej. Na przykład John Hoestje, inżynier z firmy Codeium Windsurf, zastanawia się (*https://keyholesoftware.com/codieum-windsurf-game-changing-ide-experience-part-1/*), „po co być inżynierem x10, skoro możesz być inżynierem x100". Sugeruje to, że w przypadku odpowiedniego środowiska IDE wspomaganego przez technologię AI nadzwyczajna produktywność jest w zasięgu. Takie narzędzia jak Windsurf,

czyli środowisko IDE rozszerzone o model AI, „mogą dramatycznie przyspieszyć proces projektowania, pozwalając osiągnąć produktywność x100". Choć taka wydajność może być ekstremalnym scenariuszem, nawet w przypadku bardziej konserwatywnych badań stwierdza się ogromne wzrosty wydajności.

Programiści mogą generować kod szablonowy w kilka sekund, usuwać błędy w mgnieniu oka, a nawet poinstruować model AI, by utworzył testy lub dokumentację. W ten sposób kompresuje się przepływy pracy, które kiedyś zajmowały wiele dni, do czasu liczonego zaledwie w godzinach. Nie będąc już ograniczonym szybkością pisania lub pamięcią, pojedynczy programista uzbrojony w model AI może często stworzyć prototyp aplikacji z pełnym stosem technologicznym w weekend. W przeszłości coś takiego mogło zająć małemu zespołowi wiele tygodni. To też nie jest tylko szum medialny. Jak zauważyłem w poście na blogu *Pragmatic Engineer* (*https://newsletter.pragmaticengineer.com/p/how-ai-will-change-software-engineering*) ze stycznia 2025 roku, badania pokazują, że *75% programistów* już zintegrowało jakąś formę technologii AI ze swoimi przepływami pracy, a wiele firm raportuje dwu- lub trzycyfrowe wzrosty procentowe szybkości projektowania. Krótko mówiąc, programiści wspomagający się modelem AI przekształcają mitycznego „inżyniera x10" w bardzo rzeczywiste (i osiągalne) zjawisko „inżyniera x100".

Aby zrozumieć, jak rewolucyjne to jest, rozważ konkretny przykład. Programista chce zbudować prostą aplikację internetową, która liczy słowa w skrypcie podcastu i szacuje czas czytania. Zamiast zaczynać od zera, uruchamia środowisko do tworzenia kodu wspierane przez model AI i *informuje* go o swoim pomyśle. W ciągu kilku minut model produkuje działający prototyp. Programista następnie wpisuje zapytanie: „Utwórz liczniki statystyk w jasnych kolorach i dodaj opcję eksportu do pliku PDF". Model AI odpowiednio aktualizuje kod. Rezultatem jest funkcjonalne narzędzie wdrożone jednym kliknięciem. Wszystko to osiągnięto w czasie krótszym niż 10 minut. Ten rzeczywisty scenariusz (przedstawiony przez twórcę używającego modelu AI firmy Replit; *https://www.razoroo.com/post/what-is-vibe-coding-the-ai-revolution-redefining-software-creation*) pokazuje, jak vibe coding umożliwia niezwykle szybkie, iteracyjne projektowanie oparte na bardzo ogólnych żądaniach. W podobny sposób zaczynają z tego korzystać osoby niebędące inżynierami: w tym samym artykule opisano pewnego zwolnionego marketingowca bez doświadczenia z zakresu tworzenia kodu. Użył on asystenta AI do zbudowania stu prostych narzędzi internetowych, które sumarycznie dotarły na szczyt rankingu serwisu Product Hunt. Gdy bariera dotycząca tworzenia oprogramowania jest tak niska, nie tylko zwiększa się produktywność doświadczonych programistów, ale też przede wszystkim rozszerza się grono osób, które mogą tworzyć oprogramowanie.

Programowaniu intuicyjnemu towarzyszą jednak poważne zastrzeżenia. Ponieważ tak wiele zlecasz modelowi AI, możesz skończyć z kodem, który „działa" w zgodzie z zasadą szczęśliwej ścieżki, ale skrywa w sobie „pole minowe" w postaci błędów lub kiepskich decyzji projektowych. Bez solidnego planu lub ograniczeń model LLM może wygenerować rozwiązanie, któremu brakuje odpowiedniej obsługi błędów, kontroli zabezpieczeń lub skalowalności. W rzeczywistości kod generowany przez model AI może być czasami „budową na piasku": wydaje się solidny, ale ma ukryte problemy, które ujawniają się dopiero w rzeczywistych warunkach.

Spotkałem się z przypadkami, w których programista intuicyjnie uzyskał kompletną funkcjonalność w rekordowym czasie tylko po to, by później odkryć, że kod był nieefektywny i trudny w utrzymaniu. Tego rodzaju kod przypominający „domek z kart" może zawieść w warunkach pojawienia się presji.

Wyobraź sobie na przykład, że prosisz model AI o przygotowanie na szybko systemu logowania użytkowników. Sztuczna inteligencja może szybko stworzyć działający mechanizm uwierzytelniania, ale prawdopodobnie użyje uproszczonej metody szyfrowania lub znanej i podatnej na ataki biblioteki. Jeśli wdrożysz taki system bez głębszej analizy, ślepo ufasz, że wszystko jest w porządku. Doświadczeni inżynierowie wiedzą, że to ryzykowne: kod działający w środowisku produkcyjnym musi być zrozumiały i godny zaufania. Jak to ujął jeden z ekspertów (*https://arstechnica.com/ai/2025/03/is-vibe-coding-with-ai-gnarly-or-reckless-maybe-some-of-both/*): „Vibe coding w przypadku produkcyjnej bazy kodu to oczywiste ryzyko. Większość naszej pracy jako inżynierów oprogramowania polega na rozwijaniu istniejących systemów, w których jakość i zrozumiałość podstawowego kodu mają kluczowe znaczenie". Programowanie intuicyjne w skrajnej postaci może powodować omijanie tych mechanizmów kontroli jakości.

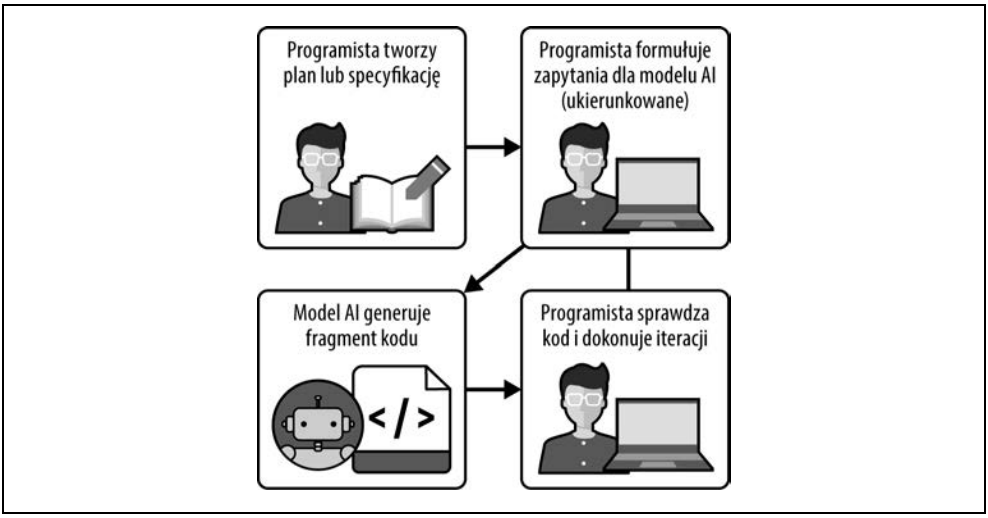
Kolejnym wyzwaniem jest to, że vibe coding bagatelizuje zwykle planowanie z góry. W tradycyjnej inżynierii oprogramowania ceni się projektowanie pod kątem przejrzystości i przemyślenie modeli danych z uwzględnieniem ograniczeń, wybór odpowiednich wzorców i przygotowanie przynajmniej minimalnej specyfikacji. Programowanie vibe coding odwraca to podejście: zaczyna się bez *żadnych ram*, po czym przechodzi się od razu do implementacji za pomocą zapytań. Może to prowadzić do chaotycznego procesu projektowania.

Zapytania mogą sprowadzić Cię w ślepą uliczkę. Powiedzmy, że model AI wybierze metodę zarządzania stanami lub bibliotekę, której nie zamierzałeś użyć, i w takiej sytuacji musisz albo naprowadzić go z powrotem na właściwą ścieżkę, albo pogodzić się z tym wyborem. Bez wstępnego planu końcowa architektura może być przypadkowa. Nie stanowi to problemu w przypadku szybkiej weryfikacji koncepcji, ale jest problematyczne w większej bazie kodu, w obrębie której spójność jest istotna.

Z natury programowanie intuicyjne nie jest „złe". W rzeczywistości jego pojawienie się jest częścią trwającej demokratyzacji programowania. Obniża ono barierę tworzenia oprogramowania podobnie jak wcześniejsze platformy niskokodowe (ang. *low-code platform*) lub języki skryptowe. Zmotywowana osoba niebędąca programistą i mająca konkretny pomysł może potencjalnie zbudować prostą aplikację wyłącznie intuicyjnie. A dla doświadczonych programistów programowanie intuicyjne może być potężnym narzędziem do prowadzenia „burzy mózgów". Przypomina to tworzenie pseudokodu, ale z natychmiastowymi rezultatami możliwymi do zastosowania. Kluczem jest rozpoznanie jego ograniczeń. Szybkość bez dyscypliny może prowadzić do „kruchego" oprogramowania, dlatego programowanie intuicyjne wymaga w pętli czujnego człowieka. Często przypominam programistom (i sobie), że „vibe coding to nie wymówka dla pracy o niskiej jakości". Powinno ono być *początkiem* rozwiązania, a nie jego końcem.

Inżynieria wspomagana sztuczną inteligencją: tworzenie struktury za pomocą asystenta AI

Na przeciwległym końcu naszego spektrum znajduje się *inżynieria wspomagana sztuczną inteligencją*, czyli bardziej uporządkowany i metodyczny sposób tworzenia oprogramowania z użyciem na każdym etapie technologii AI jako asystenta. W tym przypadku programista pozostaje główną osobą kierującą procesem. Inżynieria wspomagana przez technologię AI obejmuje wykorzystanie jej w całym cyklu tradycyjnego procesu projektowania oprogramowania. Mogą to być na przykład: automatyczne uzupełnianie oparte na modelu AI, czat, migracje kodu, wykrywanie błędów, tworzenie testów, a także zarówno szczegółowe (funkcja, moduł i komponent), jak i pełne generowanie kodu (rysunek 1.2).



Rysunek 1.2. Proces inżynierii wspomaganej sztuczną inteligencją z naciskiem na planowanie: programiści tworzą specyfikacje, formułują ukierunkowane zapytania dla systemów AI, sprawdzają wygenerowane fragmenty kodu i integrują zatwierdzone rozwiązania ze swoimi projektami

Zaczynasz od planu (nawet wtedy, gdy jest on uproszczony), określając, co musisz zbudować, i definiując z góry ograniczenia oraz kryteria akceptacji. Uwzględniasz następnie narzędzia AI w ukierunkowany sposób, aby przyspieszyć lub ulepszyć części tego planu. W przeciwieństwie do intuicyjnego tworzenia kodu opartego przede wszystkim na zapytaniach można to nazwać projektowaniem wspomaganym sztuczną inteligencją „z planem na początku". Może to być tak formalne jak minidokument wymagań produktowych (krótki dokument dotyczący opcji) lub tak proste jak lista zadań do wykonania. Kluczowa różnica polega na tym, że zanim pozwolisz działać modelowi AI, ustalasz zakres pracy w postaci *jasnych intencji i ograniczeń*.

Rozważmy przykład programisty używającego biblioteki React, którego zadaniem jest utworzenie nowego komponentu interaktywnego panelu kontrolnego. W podejściu inżynierii wspomaganej przez technologię AI może on zacząć od określenia odpowiedzialności komponentu i interfejsu API:

> Komponent panelu kontrolnego wyświetla listę kart analitycznych, obsługuje filtrowanie według zakresu dat i zawiera przyciski odświeżania oraz eksportu. Panel powinien pobierać dane z naszego interfejsu API (z odpowiednią obsługą błędów), a ponadto musi przestrzegać wytycznych naszego systemu projektowego dotyczących używania stylów.

Taki zarys to w zasadzie specyfikacja. Programista może nawet naszkicować uproszczony model danych lub zidentyfikować istniejące funkcje narzędziowe w celu ponownego użycia. Dopiero wtedy aktywuje się model AI: używając na przykład środowiska IDE z obsługą technologii AI lub asystenta tworzenia kodu do wygenerowania szkieletu komponentu na podstawie tego opisu. Model AI może dostarczyć początkową implementację komponentu biblioteki React z elementami zastępczymi umożliwiającymi pobieranie danych i zasymulowanymi procedurami obsługi zdarzeń. Ponieważ programista zapewnił jasne wytyczne, dane wynikowe modelu AI z większym prawdopodobieństwem będą zgodne z wymogami projektu (na przykład użycie odpowiednich klas systemu projektowego lub wywołanie właściwych punktów końcowych interfejsu API). Kod nie jest zaskoczeniem, gdyż stanowi efekt dobrze sformułowanego żądania.

Inżynieria wspomagana sztuczną inteligencją nie kończy się na generowaniu kodu pojedynczego komponentu. Przenika ona przez cały cykl procesu projektowania w kontrolowany sposób. W ramach rutynowych zadań z zakresu tworzenia kodu oparte na technologii AI narzędzie do automatycznego uzupełniania, takie jak GitHub Copilot, może sugerować kilka kolejnych wierszy podczas pisania kodu. Pozwala to ograniczyć liczbę naciśnięć klawiszy w trakcie implementacji znanych wzorców. Gdy na przykład tworzysz test jednostkowy, Twój asystent AI może automatycznie sugerować asercje na podstawie nazwy funkcji. Skoro mowa o testach, możesz użyć modelu AI do wygenerowania przypadków testowych po wdrożeniu funkcjonalności. W tym celu należy w zapytaniu umieścić specyfikację lub kod komponentu, aby otrzymać sugestie dotyczące przypadków brzegowych, które powinno się sprawdzić. Ideą jest tutaj *wsparcie* pracy wykonywanej przez inżyniera, a nie jej zastąpienie. Nadal przemyślasz logikę i weryfikujesz poprawność, a model AI po prostu odciąża Cię od części żmudnej pracy.

Jeśli chodzi o migrację kodu lub refaktoryzację, technologia AI może okazać się zbawieniem. Wyobraź sobie potrzebę przekształcenia komponentu biblioteki React opartego na klasach w nowoczesny komponent funkcyjny z hakami. Zamiast robić to wszystko ręcznie, możesz poprosić asystenta AI o przekształcenie kodu lub przynajmniej nakreślenie kroków. Dzięki odpowiedniemu zrozumieniu starych i nowych wzorców model LLM może przygotować wersję roboczą kodu poddanego refaktoryzacji, który następnie przejrzysz i dopracujesz. Takie użycie technologii AI w strukturalny sposób pozwala wykonać dobrze zdefiniowane zadania (takie jak: „dokonaj migracji tego kodu z biblioteki Redux do interfejsu API Context biblioteki React") jedno po drugim, zamiast zapewniać modelowi AI elastyczne upoważnienie w stylu „zbuduj cokolwiek".

Być może najbardziej dramatyczną formą inżynierii wspomaganej sztuczną inteligencją jest użycie technologii AI do generowania pełnej miniaplikacji lub funkcjonalności na podstawie szczegółowej specyfikacji. Kilka narzędzi pozwala obecnie na wprowadzenie opisu aplikacji, czyli czegoś podobnego do minidokumentu wymagań produktowych, i otrzymanie

z powrotem działającej bazy kodu lub prototypu. Programista może na przykład dostarczyć specyfikację dotyczącą:

> aplikacji listy zadań do wykonania z interfejsem użytkownika biblioteki React i warstwą serwerową środowiska Node.js, która obsługuje uwierzytelnianie użytkowników i aktualizacje w czasie rzeczywistym.

Narzędzie oparte na technologii AI stworzyłoby szkielet projektu, zbudowało kluczowe komponenty i skonfigurowało schemat bazy danych.

To nie jest magia, lecz przyspieszona wersja tego, co mógłby wykonać sumienny inżynier rozpoczynający nowy projekt (tworzenie katalogów, wybór bibliotek i pisanie rutynowego kodu). Najważniejsze jest to, że kreatywność modelu AI jest *ograniczona przez ramy określone w specyfikacji*. Rezultatem jest produkt o minimalnej funkcjonalności (MVP — *Minimum Viable Product*), który spełnia podane przez Ciebie wymagania. Podchodząc do tego wyniku właściwie, doświadczony programista nie będzie zakładał, że jest gotowy do użycia w środowisku produkcyjnym po pierwszym wygenerowaniu. Zamiast tego potraktuje go jako pierwszą wersję roboczą. Programista uruchomi aplikację, utworzy lub wygeneruje ponownie testy do sprawdzenia każdej funkcjonalności, przejrzy kod pod kątem niespójności lub niebezpiecznych konfiguracji, a ponadto udoskonali go według potrzeb. Krótko mówiąc, skorzysta on z całej swojej zwykłej, inżynierskiej skrupulatności, co jednak będzie przyspieszone dzięki zdolności modelu AI do produkowania dużych ilości kodu na podstawie planu.

Cele inżynierii wspomaganej sztuczną inteligencją różnią się od tych w przypadku kodowania intuicyjnego. Chodzi tu nie tylko o szybkie uzyskanie *działającego* kodu, ale o wydajniejsze tworzenie kodu *wysokiej jakości*. Jest to kwestia zwiększania produktywności przy zachowaniu (a nawet zwiększeniu) stopnia niezawodności rezultatu. Zespół zajmujący się inżynierią wspomaganą sztuczną inteligencją mógłby stwierdzić: „Chcemy dostarczyć tę funkcjonalność dwa razy szybciej, ale bez żadnych kompromisów w zakresie naszych standardów".

Adresatami takiego podejścia są zwykle zawodowi programiści i zespoły, które mają ustalone procesy (przegląd kodu, testowanie, potoki wdrażania) i nie zamierzają z nich rezygnować. Są to inżynierowie średniego i wyższego szczebla, którzy widzą w technologii AI nowe, potężne narzędzie w swoim arsenale, a nie jego zamiennik. Prawdopodobnie przekonali się już oni, co się dzieje, gdy idzie się na skróty, dlatego cenią praktyki, które pozwalają utrzymywać oprogramowanie (dla porównania, grupa docelowa w przypadku vibe codingu to działający w pojedynkę programiści przygotowujący wersje demonstracyjne, osoby myślące produktowo z pewną wiedzą o tworzeniu kodu, a nawet względnie początkujący programiści, którzy wykorzystują technologię AI do uzupełnienia luk w swojej wiedzy).

W inżynierii wspomaganej sztuczną inteligencją oczekuje się, że ludzie zachowują kontrolę nad decyzjami, a technologia AI zapewnia sugestie lub narzędzia przyspieszające. Jakość kodu, wydajność i bezpieczeństwo pozostają najważniejsze, dlatego każdy fragment wygenerowany przez model AI podlega takiej samej kontroli, jak w sytuacji, gdyby utworzył go początkujący programista. Traktuj model AI jak swojego stażystę, a nie jak zastępcę. Możesz mu delegować zadania, ale musisz sprawdzać efekty jego działań. Tak jak nigdy nie wdrożyłbyś kodu

napisanego przez stażystę bez uprzedniego sprawdzenia go, tak nie powinieneś stosować kodu napisanego przez model AI bez wcześniejszego zrozumienia, jak funkcjonuje. Takie nastawienie pozwala utrzymać dyscyplinę inżynierską na pierwszym planie.

Różne nastawienia i oczekiwania

Vibe coding i inżynieria wspomagana sztuczną inteligencją to dwa różne nastawienia. Kodowanie intuicyjne jest odgórne i eksploracyjne: zaczynasz od ogólnego pomysłu i pozwalasz, aby implementacja wyłoniła się w wyniku interakcji z modelem AI. To trochę jak improwizacyjny jazz, czyli minimalna struktura, dużo miejsca na kreatywne riffy, a kształt utworu odkrywa się w trakcie grania. Inżynieria wspomagana sztuczną inteligencją jest systematyczna i iteracyjna: bardziej jak kompozycja klasyczna, w ramach której zaczynasz od tematu lub motywu (Twoje wymagania) i metodycznie go rozwijasz, być może pozwalając sobie na trochę improwizacji (sugestie modelu AI) w obrębie zapisanej partytury. Oba podejścia mogą umożliwiać tworzenie „muzyki", ale proces i rodzaj rezultatu będą się różnić.

Dla średnio lub bardzo zaawansowanego projektanta aplikacji internetowych kluczowe są Twoje oczekiwania wobec każdego z podejść. Jeśli programujesz intuicyjnie, spodziewasz się zaskoczeń. Model AI może zaproponować wariant, którego sam byś nie zastosował. Być może użyje on innej biblioteki lub idiomu programistycznego, z którym jesteś mniej zaznajomiony. Część magii polega na wyciąganiu wniosków z tych niespodzianek lub szybkim pomijaniu rzeczy, które uważasz za żmudne. Musisz też jednak spodziewać się problemów. Entuzjaści vibe codingu powinni podchodzić do niego bardzo ostrożnie, wiedząc, że będą odpowiedzialni za ten trudny, ostatni etap. Magia jest prawdziwa, ale nie jest całkowita.

Jeśli praktykujesz inżynierię wspomaganą sztuczną inteligencją, w przypadku długoterminowych projektów Twoje oczekiwania są bardziej wyważone i raczej bardziej realistyczne. Spodziewasz się, że technologia AI pozwoli Ci zaoszczędzić czas i być może zainspiruje do opracowania jednego lub dwóch rozwiązań, ale nie wykona całej Twojej pracy. W rzeczywistości dobry inżynier wspomagany przez model AI może używać zapytań w stylu intuicyjnym w *małych dawkach* w ramach większej struktury. Na przykład implementując dobrze określony moduł, może on na chwilę przełączyć się w „tryb intuicyjny", aby wysłać następujące zapytanie: „Witaj modelu AI, wygeneruj szybką funkcję narzędziową do formatowania tych dat", a następnie natychmiast powrócić do trybu inżynierii w celu zintegrowania i sprawdzenia tej funkcji. Nastawienie polega tu na tym, że model AI jest współpracownikiem działającym pod Twoim kierunkiem. Przydzielasz mu zadania, w których się sprawdza (na przykład kod szablonowy i powtarzalny oraz implementacje w ogólnym zarysie), a resztę realizujesz sam (krytyczną logikę, integrację i końcowy przegląd).

Oczekiwania obejmują tutaj zwiększoną produktywność, mniej mechanicznych błędów (na przykład model AI rzadziej popełni literówkę w nazwie zmiennej) i prawdopodobnie szerszą przestrzeń poszukiwania rozwiązań (model AI może zasugerować algorytm, o którym nie pomyślałeś). Spodziewasz się też jednak, że zainwestujesz czas w weryfikację poprawności. Debugowanie kodu tworzonego przy wspomaganiu przez model AI to nadal debugowanie: uruchamiasz testy i w razie potrzeby analizujesz kod w debuggerze w trybie krokowym.

Różnica polega na tym, że możesz znaleźć się w sytuacji debugowania kodu, który został wygenerowany przez model AI, co jest nowym doświadczeniem związanym z krzywą uczenia. W rozdziale 5. szczegółowo omówiono to doświadczenie.

Cele tych dwóch podejść podkreślają fundamentalną różnicę między nimi: vibe coding optymalizuje *szybkość w krótkim okresie*, natomiast inżynieria wspomagana sztuczną inteligencją optymalizuje *trwałą szybkość i niezawodność*. Programista intuicyjny może powiedzieć: „Muszę uruchomić tę aplikację do wieczora, żeby sprawdzić, czy pomysł działa". Inżynier wspomagany przez technologię AI stwierdziłby: „Muszę szybko zbudować tę funkcjonalność, ale powinna ona być na tyle solidna, aby mogła przez lata znajdować się w naszej bazie kodu". Programista jest zadowolony, jeśli kod w zasadzie działa, a inżynier dba o to, żeby kod był wystarczająco czysty, aby inni mogli na nim bazować w trakcie swojej pracy.

Różnice te przemawiają naturalnie do różnych grup odbiorców. Mniej doświadczeni programiści lub ci niezwiązani z dziedziną inżynierii mogą skłaniać się ku programowaniu intuicyjnemu, ponieważ obniża ono barierę wejścia i zapewnia natychmiastową satysfakcję. Spotkałem menedżerów produktu i projektantów, którzy próbują swoich sił w tworzeniu kodu za pomocą zapytań intuicyjnych, traktując model AI niemal jak superwydajny serwis Stack Overflow, który oferuje im kompletne rozwiązania. Z kolei doświadczeni programiści i zespoły inżynierów będą zwykle preferować inżynierię wspomaganą przez technologię AI. Wcześniej doświadczyli już problemów z powodu niestabilnego kodu, dlatego wychodzą z założenia „zróbmy to właściwie, jeśli nawet skorzystamy z nowych narzędzi w celu przyspieszenia prac". W zamian za długoterminowe korzyści wkładają na początku nieco więcej wysiłku (tworząc minidokument wymagań produktowych i przygotowując strukturę projektu).

Ustalanie swojego miejsca

Kusi, żeby zapytać: które podejście jest lepsze? Prawda jest taka, że vibe coding i inżynieria wspomagana sztuczną inteligencją to nie wzajemnie wykluczające się kategorie. Reprezentują one dwa końce spektrum, a rzeczywiste przepływy pracy często łączą elementy obu podejść. Programista może rozpocząć projekt od spontanicznego tworzenia kodu w intuicyjny sposób, żeby szybko przygotować szkielet czegoś nowego, a potem może przejść w tryb inżynieryjny w celu dopracowania tego. Generalnie może postawić na dyscyplinę wspomaganą sztuczną inteligencją, ale czasami w przypadku jakiegoś prostego i jednorazowego skryptu lub tymczasowego prototypu może stwierdzić: „Wiesz co, po prostu utworzę kod intuicyjnie i zobaczę, co z tego wyjdzie". Kluczem jest zrozumienie kompromisów i stosowanie właściwego podejścia w odpowiednim kontekście.

Pomyśl o vibe codingu jak o bardzo szybkim pojeździe eksploracyjnym: może on szybko zabrać Cię z utartych dróg i świetnie nadaje się do odkrywania. Inżynieria wspomagana technologią AI przypomina bardziej niezawodny pociąg na torach: musisz najpierw położyć szyny (opracowanie planu), ale jest to bezpieczniejsza opcja i większa szansa na dotarcie do określonego celu bez wykolejenia. Średnio i bardzo zaawansowani programiści powinni umieć prowadzić oba pojazdy, ale będą wybierać w zależności od danego zadania. Jeśli celem jest szybkie wprowadzenie innowacji lub sformułowanie koncepcji (na przykład w trakcie maratonu

programistycznego lub przy sprawdzaniu możliwości realizacji pomysłu), programowanie intuicyjne pozwala nabrać rozpędu. Pamiętaj tylko, żeby wszystko dopracować, jeśli planujesz ponownie użyć tego kodu. Jeżeli celem jest utworzenie opcji produktu, którą można utrzymywać w profesjonalnym środowisku, skłanianie się ku inżynierii wspomaganej sztuczną inteligencją gwarantuje, że nie skończysz z przypominającą „czarną skrzynkę" porcją kodu w swojej bazie, którego nikt tak naprawdę nie rozumie.

Fascynującą rzeczą, jaką zaobserwowałem, jest to, że gdy programiści nabierają doświadczenia z narzędziami opartymi na technologii AI, sposób korzystania z nich często przesuwa się naturalnie od wariantu intuicyjnego w stronę wariantu inżynierii.

Początkowo nowość polegająca na tym, że model AI generuje całe bloki kodu po wprowadzeniu jednego zapytania, jest kusząca. Kto by nie chciał spróbować w zasadzie stworzyć aplikację w ramach „pogadanki"?

Po miesiącu miodowym pojawia się jednak pragmatyzm. Programiści zaczynają dostrzegać, gdzie technologia AI błyszczy, a gdzie się potyka. Uczą się dzielić problemy na części i przekazywać je modelowi AI w kawałkach, zamiast prosić o całe rozwiązanie za jednym zamachem. W efekcie przestają być „artystami zapytań" i stają się „dyrygentami orkiestry" wspieranej przez technologię AI. Programiści nadal wykorzystują jej twórczą moc, ale kierują nią wprawną ręką i podążają za wyraźną partyturą. W mojej codziennej pracy używam zapytań w bardziej przemyślany sposób. Zamiast po prostu zadawać otwarte pytania, często tworzę niewielkie porcje pseudokodu lub komentarze, po czym proszę model AI o uzupełnienie ich. Dzięki temu zyskuję korzyści w postaci płynności podobnej jak w przypadku kodowania intuicyjnego, ale w ramach struktury, którą kontroluję.

Warto też zauważyć, że narzędzia ewoluują, żeby wspierać całe spektrum. Z jednej strony mamy interfejsy oparte na czacie i środowiska do tworzenia kodu w języku naturalnym zaprojektowane wyraźnie z myślą o programowaniu intuicyjnym, w ramach którego możesz nawet nie ujrzeć kodu, dopóki o niego nie poprosisz. Z drugiej strony środowiska IDE zapewniają opcje technologii AI, które płynnie wpisują się w tradycyjne tworzenie kodu. Są to na przykład oparte na sztucznej inteligencji narzędzia do analizy kodu sugerujące ulepszenia, generatory dokumentacji wyjaśniające kod oraz boty do kontroli wersji, które mogą automatycznie utworzyć żądanie scalenia kodu i zasugerować zmiany do sprawdzenia. Narzędzia te zachęcają do myślenia inżynieryjnego, wpisując się w zwykły przepływ pracy związanej z projektowaniem (edytowanie, przeglądanie, testowanie itd.), a jednocześnie wykorzystując technologię AI.

Rozróżnienie między programowaniem intuicyjnym a inżynierią wspomaganą sztuczną inteligencją może nawet zaniknąć z czasem, gdy pojawią się najlepsze praktyki. Można odkryć, że to, co dziś wydaje się intuicyjnym tworzeniem kodu, zyska więcej zabezpieczeń, a to, co wydaje się strukturalną inżynierią, stanie się bardziej płynne. Właściwie stwierdziłbym, że idealna przyszłość to taka, w której można bez wysiłku poruszać się w górę i w dół tego spektrum. Oznacza to eksplorowanie kreatywnych rozwiązań z użyciem technologii AI, jeśli tego chcemy, ale zawsze należy trzymać wszystko w ryzach w postaci solidnych praktyk z zakresu inżynierii, gdy przychodzi pora na zabezpieczenie i dostarczenie oprogramowania.

Takie spektrum podejść reprezentuje znaczącą ewolucję dotyczącą tego, jak dziś korzystamy z narzędzi opartych na technologii AI. Jednakże nawet wtedy, gdy doskonalimy nasze techniki współpracy z modelem AI, czy to w ramach szybkiego kodowania intuicyjnego, czy przepływów pracy strukturalnej inżynierii, kształtuje się bardziej fundamentalna transformacja. Sama natura programowania się zmienia. Odchodzimy od tradycyjnego modelu, w którym programiści musieli przekształcać swoje pomysły w wyraźne instrukcje, w stronę przyszłości, w jakiej bezpośrednio możemy wyrażać nasze intencje i zezwolić modelowi AI zająć się procesem translacji do postaci kodu.

Zmiana ta podważa nasze najbardziej podstawowe założenia dotyczące tego, czym jest bycie programistą. Przez pokolenia nasza wartość była związana z umiejętnością myślenia jak maszyny polegającą na rozkładaniu problemów na konkretne, logiczne kroki, które komputery mogą wykonać. Co się jednak dzieje, gdy maszyny stają się zdolne do rozumienia tego, czego *chcemy*, a nie tylko tego, co im każemy zrealizować? Tu właśnie pojawia się *świadome programowanie* reprezentujące nie tylko nowe narzędzie czy technikę, ale fundamentalne przewartościowanie roli programisty.

Więcej niż kod: świadome programowanie

Przez dziesięciolecia programowanie oznaczało pisanie instrukcji, czyli wiersz po wierszu kodu informującego komputer, *jak* coś zrobić. Każda funkcja, pętla i instrukcja warunkowa musiała być starannie przygotowana przez człowieka. Świadome programowanie odwraca ten schemat. Zamiast skupiać się na niskopoziomowej implementacji, programista koncentruje się na wyniku lub celu, czyli na tym, co program ma osiągnąć. Wyrażasz swój zamiar w ogólny sposób (często za pomocą języka naturalnego), a system z technologią AI sam generuje kod, który pozwoli go zrealizować.

Pomyśl o tym w następujący sposób: tradycyjne tworzenie kodu to jak dawanie komuś wskazówek krok po kroku, natomiast programowanie oparte na zamiarze to jak powiedzenie komuś, dokąd chcesz dotrzeć, i pozwolenie mu samemu ustalić najlepszą trasę. Skupiając się na tym *co*, zamiast na tym *jak*, programiści mogą pracować na wyższym poziomie abstrakcji. Takie podejście nie jest całkowicie nowe. Narzędzia, takie jak programowanie wizualne, platformy niskokodowe i generatory kodu, od dawna obiecywały zwiększenie poziomu abstrakcji. Jednakże obecne osiągnięcia w zakresie technologii AI sprawiają w końcu, że praktyczne staje się opisywanie złożonych zachowań w zwykłym języku i otrzymywanie w zamian działającego kodu.

Wzrost znaczenia zapytań: od poleceń do opisów

Sednem tej zmiany jest niepozorne *zapytanie*. Ma ono postać danych wejściowych lub pytania, które przekazujesz systemowi sztucznej inteligencji służącemu do tworzenia kodu. W istocie jest to opis tego, co program ma zrobić, a nie instrukcja określająca, w jaki sposób ma to zrealizować. Może to wydawać się bardzo odmienne od pisania kodu. Na przykład zamiast tworzyć pętlę do analizy pliku, możesz wprowadzić następujące zapytanie:

Wczytaj ten plik CSV i wyodrębnij adresy e-mail wszystkich użytkowników mających więcej niż 18 lat.

Sztuczna inteligencja spróbuje wygenerować kod, który realizuje cel zawarty w tym opisie.

Dlaczego dzieje się to teraz? Szybki postęp modeli LLM w zakresie rozumienia i generowania tekstu, w tym kodu języków programowania, okazał się przełomowy. Te modele sztucznej inteligencji zostały wytrenowane na ogromnych ilościach kodu i tekstu zapisanego w języku naturalnym. Potrafią zinterpretować zapytanie, które wygląda jak opis zachowania oprogramowania, i przekształcić je w rzeczywisty kod implementujący to zachowanie. Innymi słowy, modele nauczyły się wzorców tego, jak ludzie opisują zadania i jak przekładają się one na kod.

Rozwój projektowania opartego na zapytaniach oznacza, że jako programista coraz częściej tworzysz opisy opcji i logiki w języku naturalnym lub za pomocą pseudokodu, a sztuczna inteligencja zajmuje się ciężką pracą związaną z pisaniem kodu poprawnego składniowo. Zapytanie staje się nową jednostką myślenia. Jest to zwięzłe wyrażenie zamiaru. Przeszliśmy od instruowania komputera w stylu „zrób X, potem Y, a następnie Z" do używania sformułowań w rodzaju „potrzebuję, aby X, Y i Z zostało zrealizowane" i ufania technologii AI, że uzupełni brakujące elementy.

Godne uwagi jest to, że napisanie dobrego zapytania samo w sobie jest umiejętnością (zostanie to szczegółowo omówione w rozdziale 3.). Niejasne zapytanie może prowadzić do niepoprawnego lub nieefektywnego kodu, tak jak niedokładne wymagania mogą zmylić programistę. Im lepiej potrafisz wyrazić w zapytaniu swój zamiar, tym lepiej wynik modelu AI będzie odpowiadał Twoim potrzebom. Z tego powodu wiele osób nazywa tworzenie zapytań nową umiejętnością programistyczną.

Jak to działa: cykl iteracyjny i rola sztucznej inteligencji w generowaniu kodu

Jak zatem sztuczna inteligencja przechodzi od swobodnego opisu do faktycznego i działającego kodu? Magia tkwi w zdolności dużych modeli językowych do interpretowania kontekstu i generowania tekstu. Określenie *duży* w terminie „duży model językowy" odnosi się do liczby parametrów (wewnętrznej konfiguracji), jaką on posiada (często są to miliardy lub więcej). Umożliwia to modelowi uchwycenie złożoności języków naturalnych i programowania. Modele te zostały wytrenowane przy użyciu publicznych repozytoriów kodu, zasobów forów, dokumentacji oraz witryn z pytaniami i odpowiedziami, ucząc się zarówno składni języków programowania, jak i semantyki tego, jak kod jest stosowany do rozwiązywania problemów. Gdy wchodzisz w interakcję z programistą w postaci modelu AI, korzystasz z tej rozległej i wyuczonej wiedzy. Rozłóżmy to na następujące proste zagadnienia:

Rozumienie zapytania

Po przekazaniu zapytania (na przykład „Wygeneruj funkcję, która sprawdza, czy liczba jest pierwsza"), model AI analizuje jego treść. Nowoczesne modele firm Google, OpenAI i Anthropic zostały wytrenowane na niezliczonych przykładach języka i kodu, dlatego używają wzorców statystycznych do wywnioskowania, o co pytasz. W zasadzie model AI próbuje *przewidzieć* najbardziej prawdopodobną odpowiedź na zapytanie zawierającą kod, który ma sens.

Wykorzystywanie kontekstu

Takie systemy z modelem AI często biorą pod uwagę dodatkowy kontekst wykraczający poza zapytanie wpisane w jednym wierszu. Jeśli na przykład korzystasz ze środowiska IDE z asystentem AI, model może również uwzględnić zawartość bieżącego pliku, Twój styl kodowania, komentarze, a nawet powiązane pliki. Cały ten kontekst pomaga modelowi AI generować kod, który pasuje do Twojego projektu. Jest to podobne do tego, jak programista czyta powiązany kod i dokumentację, aby zrozumieć, co ma zrobić dalej.

Generowanie kodu

Gdy model zrozumiał Twój zamiar (lub przynajmniej odgadł go najlepiej jak potrafił), przystępuje do wygenerowania kodu. W tle model używa w tym celu jednego tokena na raz (jest to fragment słowa lub symbol kodu), korzystając z prawdopodobieństw wyuczonych podczas treningu. Model nie „myśli" w konwencjonalnym tego słowa znaczeniu. Nie zawiera on kompilatora ani środowiska uruchomieniowego, które sprawdza kod. Po prostu model bardzo dobrze kontynuuje generowanie tekstu w sposób, który ma duże szanse na bycie poprawnym kodem, ponieważ napotkał wcześniej tak wiele przykładów. Jeśli zapytanie i kontekst są przejrzyste, kod tworzony przez model może być niezwykle dokładny, a nawet może przestrzegać najlepszych praktyk, które zidentyfikował w swoich danych treningowych.

Weryfikacja poprawności w ramach nadzoru człowieka

Co ważne, technologia AI nie „ucieka" i nie wdraża Twojej aplikacji za Ciebie. Pozostajesz w pętli. Przeglądasz wygenerowany kod, testujesz go i możesz go zaakceptować lub zmodyfikować. W wielu przypadkach model AI może również oferować wyjaśnienie kodu, jeśli zostanie o to poproszony, pomagając Ci zrozumieć rezultat. Rola modelu AI przypomina pracę asystenta, który opracowuje dla Ciebie wstępną wersję kodu, ale to Ty jako programista nadal jesteś osobą podejmującą decyzje i zapewniającą, że kod jest poprawny i zgodny z wymogami projektu.

Tym, co naprawdę imponuje, jest to, że ten proces dzieje się w ciągu kilku sekund lub szybciej. Ogólnie sytuacja wygląda tak, że Twój opis (zapytanie) trafia do silnika predykcji (model LLM), który tworzy prawdopodobny kod jako wynik. Choć wewnętrzne mechanizmy działania modeli obejmują złożoną matematykę i warstwy sieci neuronowych, na poziomie użytkownika wydaje się to prawie jak współpraca z ekspertem, który może natychmiast przypomnieć sobie, jak zaimplementować niemal wszystko.

Jedną z kluczowych rzeczy do zrozumienia w przypadku kodowania intuicyjnego (opartego na zamiarze) jest to, że jest to proces iteracyjny polegający na współpracy człowieka i modelu AI. Nie wprowadzasz jedynie pojedynczego idealnego zapytania, a potem siedzisz z założonymi rękami w czasie, gdy model AI bezbłędnie generuje cały program. W praktyce angażujesz się w ciągłą wymianę danych, czyli pętlę informacji zwrotnych, która stopniowo pozwala przekształcić mało precyzyjny pomysł w dopracowany kod.

Typowy cykl może wyglądać tak:

Krok 1. Opisujesz, czego potrzebujesz.

Jest to Twoje początkowe zapytanie lub prośba. Oto przykład:

Utwórz funkcję do obliczania miesięcznych rat kredytu na podstawie jego kapitału, stopy oprocentowania i okresu spłaty.

Krok 2. Model AI zapewnia początkowe rozwiązanie.

Model AI generuje kod dla tej funkcji wraz z parametrami i wzorem obliczania rat kredytu. Może on nawet dołączyć komentarze objaśniające wzór.

Krok 3. Przeglądasz i testujesz.

Przyglądasz się kodowi. Czy ma on sens? Czy obsługuje przypadki brzegowe? Przeprowadzasz szybki test: co będzie, jeśli stopa oprocentowania wynosi 0? Czy kod działa poprawnie? Zauważasz, że może on nie radzić sobie dobrze z tym scenariuszem.

Krok 4. Udoskonalasz swoje zapytanie lub kod.

Jeśli kod nie jest idealny (a często nie będzie za pierwszym razem), udoskonalasz go. Możesz ponownie skierować zapytanie do modelu AI („Zmodyfikuj funkcję tak, aby w kontrolowany sposób obsługiwała zerową stopę oprocentowania") lub samodzielnie edytować kod i przesłać modelowi zapytanie: „Wyjaśnij tę część", jeśli coś jest niejasne. Takie wytyczne pomagają wyeliminować wszelkie nieporozumienia.

Krok 5. Model AI udoskonala rozwiązanie.

Model AI bierze pod uwagę Twoją opinię lub nowe zapytanie i dostosowuje kod. W efekcie funkcja dokonuje sprawdzenia pod kątem zerowego oprocentowania i obsługuje je w odpowiedni sposób.

Krok 6. Powtarzaj w razie potrzeby.

Kontynuujesz tę pętlę działań do momentu, aż będziesz zadowolony. Być może poprosisz następnie model AI o wygenerowanie też testów jednostkowych dla tej funkcji, aby upewnić się, że działa poprawnie. Model zajmuje się tym, a Ty uruchamiasz testy w celu sprawdzenia, czy wszystko jest w porządku.

Taka współpraca bardzo przypomina scenariusz programowania w parach, w ramach którego partnerami są człowiek i asystent AI. Człowiek wyznacza kierunek i zna ogólne wymagania, natomiast model AI oferuje sugestie, pisze kod szablonowy i przyspiesza części wiążące się ze żmudną pracą. Żaden z partnerów nie jest skuteczny samodzielnie w przypadku złożonych zadań: model AI polega na człowieku w kwestii kierunku i weryfikowania poprawności, a człowiek przekazuje mu część pracy, aby działać szybciej.

Co kluczowe, iteracja to nie tylko naprawianie błędów. Jest to także rozwijanie rozwiązania. Możesz zacząć od bardzo ogólnego zapytania, a następnie stopniowo doprecyzowywać swoje zamiary, widząc, co generuje model AI.

Zachęca to do myślenia eksperymentalnego. Jeśli pierwsza próba nie jest właściwa, nie zmarnowałeś wiele czasu. Po prostu doprecyzuj zapytanie lub popraw kod i spróbuj ponownie. W tradycyjnym programowaniu tworzenie modułu tylko po to, aby z niego zrezygnować, może być frustrujące. Jednakże w przypadku kodu generowanego przez model AI koszt nieudanego startu jest niewielki, co zachęca do eksplorowania różnych podejść.

Wydajność, dostępność i zmieniający się charakter programowania

Dlaczego świadome programowanie to tak wielka sprawa? Zmiana ta niesie ze sobą kilka znaczących konsekwencji. Oto one:

Zwiększanie wydajności programistów

Prawdopodobnie najbardziej bezpośrednią korzyścią jest szybkość. Programiści mogą wykonywać zadania szybciej, gdy model AI zajmuje się rutynowymi zadaniami. Standardowy kod, którego napisanie ręcznie mogłoby zająć wiele godzin (na przykład tworzenie modeli baz danych, punktów końcowych interfejsu API lub skryptów do czyszczenia danych), często można wygenerować w ciągu kilku minut. Wczesne badania nad asystentami AI do tworzenia kodu potwierdzają to. Programiści korzystający z takich narzędzi jak GitHub Copilot wykonują zadania znacznie szybciej (jedno z badań dostępnych pod adresem *https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/* wykazało skrócenie o 55% czasu realizacji danego zadania z użyciem asystenta Copilot). Gdy pomnożysz te oszczędności czasu dla całego projektu, rysuje się przyszłość, w której cykle tworzenia oprogramowania dramatycznie się skracają, a zespoły mogą iterować szybciej.

Utrzymywanie programistów „w rytmie"

Poza samą szybkością pojawia się korzyść natury psychologicznej. Tworzenie kodu szablonowego lub sprawdzanie składni może przerwać tok myślenia programisty i jego koncentrację. Gdy model AI zajmuje się wieloma takimi rzeczami zakłócającymi pracę, programiści mogą skupić się na problemie, który rozwiązują. Wielu użytkowników zgłasza (*https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness*), że dzięki pomocy modelu AI czują się mniej sfrustrowani żmudnymi zadaniami i mogą koncentrować się na kreatywnych i projektowych aspektach tworzenia kodu. Innymi słowy, może to uczynić tworzenie kodu bardziej przyjemnym w wyniku przejęcia przez model nudnych części pracy, co z kolei może poprawić jej jakość (szczęśliwszy programista często tworzy lepszy kod).

Obniżanie bariery wejścia

Tradycyjnie programowanie wymagało nauki dokładnej składni kodu oraz osobliwości różnych bibliotek i środowisk. W przypadku programowania opartego na zamiarze część tego obciążenia przenosi się na technologię AI. Początkujący programista może nie pamiętać dokładnej składni służącej do otwarcia pliku lub parametrów funkcji do tworzenia wykresów. Jeśli jednak potrafi opisać to, czego chce, model AI może uzupełnić te szczegóły. Nie oznacza to, że każdy może tworzyć kod złożonych systemów bez żadnej wiedzy (nadal trzeba rozumieć, co program powinien realizować), lecz to, że czas potrzebny do uzyskania użytecznych rezultatów jest krótszy. Można sobie wyobrazić, że eksperci (na przykład biolog lub ekonomista) mogliby tworzyć prototypy w swojej dziedzinie poprzez opisywanie swoich wymagań nawet wtedy, gdy nie są zawodowymi programistami. W tym sensie programowanie staje się bardziej dostępne dla osób, które mają pomysły i zamiar, ale nie posiadają dużych umiejętności z zakresu pisania kodu.

Zmieniające się role i umiejętności programistów

Gdy technologia AI w większym stopniu przejmuje generowanie kodu, rola programisty ewoluuje. Umiejętności, takie jak projektowanie architektury, dekompozycja problemów i weryfikacja poprawności, stają się jeszcze ważniejsze. Możesz stwierdzić, że spędzasz więcej czasu na decydowaniu o tym, *co* budować, i sprawdzaniu, *dlaczego* kod działa (albo nie), niż na wpisywaniu składni. Natura „umiejętności kodowania" może przesunąć się w kierunku „zdolności kierowania modelem AI w zakresie tworzenia kodu". Mogłoby to zdemokratyzować pewne aspekty projektowania oprogramowania, podnosząc jednocześnie poziom, na którym działają profesjonaliści. Prawdopodobnie ujrzymy nowe, najlepsze praktyki ukierunkowane na to, jak skutecznie kierować działaniem modelu AI (jest to temat, który zaprezentuję w rozdziale 3., a później będę do niego wracał w reszcie książki).

Wydajność kontra kreatywność

Co ciekawe, gdy model AI zajmuje się większą ilością rutynowego tworzenia kodu, programiści mogą skupić się na zadaniach kreatywnych wyższego poziomu, takich jak ulepszanie komfortu pracy użytkownika, organizowanie „burzy mózgów" dotyczącej nowych opcji lub rozwiązywanie zawiłych problemów algorytmicznych, z którymi model AI może sobie dobrze nie poradzić samodzielnie. W tym idealnym scenariuszu technologia AI zwiększa wydajność powtarzalnych zadań stanowiących 80% programowania, uwalniając energię umysłu na inwencyjną część wynoszącą 20%. Jest to zmiana w sposobie alokacji podejmowanych przez nas działań.

Nie wszystko jest jednak takie kolorowe. Ten nowy styl projektowania powoduje też następujące wyzwania:

Zaufanie i poprawność

Czy można zaufać kodowi wygenerowanemu przez model AI? Jeśli nie widzisz każdego wiersza kodu, istnieje ryzyko, że błędy przejdą niezauważone. Programiści muszą dokładnie testować i przeglądać kod generowany przez model AI. Odpowiedzialność za zapewnienie, że dane wynikowe są poprawne, bezpieczne i wydajne, spoczywa na człowieku. Ślepe zaufanie wynikom modelu AI jest ryzykowne, o czym będzie dalej mowa.

Utrata niektórych umiejętności niskiego poziomu

Jeśli polegasz na sztucznej inteligencji przy rutynowym tworzeniu kodu, czy stopniowo utracisz zdolność pisania takiego kodu od podstaw lub debugowania problemów wymagających zajmowania się szczegółami? Jest to obawa podobna do tej, że nadmierne poleganie na kalkulatorach osłabia umiejętności arytmetyczne. Programiści będą musieli świadomie równoważyć wygodę z utrzymaniem umiejętności solidnego zrozumienia podstaw.

Zmieniające się środowisko zawodowe

Gdy świadome programowanie stanie się powszechne, branża może zacząć cenić inne umiejętności. Może zmniejszyć się zapotrzebowanie na osoby potrafiące jedynie mechanicznie tworzyć szablonową logikę, a wzrośnie popyt na te, które są w stanie projektować systemy, integrować komponenty i weryfikować poprawność. Charakter pracy w branży oprogramowania może się zmienić. Sztuczna inteligencja będzie w większym stopniu obsługiwać implementację, a ludzie skupią się na projektowaniu i nadzorze.

Dodatkowo jednym z najważniejszych czynników związanych z programowaniem intuicyjnym jest rozmiar okna kontekstowego. Model Gemini oferuje najdłuższe okno kontekstowe spośród wszystkich modeli AI, co może być przełomowe w przypadku realizowania dużych projektów. Niektóre modele obsługują już okna kontekstowe przekraczające milion tokenów, co pozwala im „utrzymać świadomość" całych aplikacji. Programiści mogą przekazywać całe bazy kodu do modelu sztucznej inteligencji w celu pełnego ich zrozumienia.

Na końcu rozdziału bardziej szczegółowo przyjrzymy się tym kompromisom. Najpierw jednak warto zaznajomić się z nowymi narzędziami, które umożliwiają ten nowy sposób tworzenia kodu.

Przegląd narzędzi: rozwijający się ekosystem

Vibe coding może być filozofią, ale umożliwia go nowa generacja narzędzi opartych na sztucznej inteligencji. Doświadczeni programiści, którzy chcą zaadaptować taki przepływ pracy, będą musieli zaznajomić się z kluczowymi platformami i modelami sprawiającymi, że tworzenie kodu wspomagane sztuczną inteligencją staje się skuteczne.

W niniejszym podrozdziale dokonano krótkiego przeglądu podstawowych narzędzi wchodzących w skład zestawu narzędziowego osoby korzystającej z kodowania intuicyjnego. Obejmują one środowisko Visual Studio Code (VSCode) z powiększającym się ekosystemem opcji i rozszerzeń technologii AI, zintegrowane z nią środowiska IDE następnej generacji (na przykład Cursor i Windsurf), duże modele językowe, takie jak Claude (w różnych wersjach) i ChatGPT. W podrozdziale nie są omawiane działające w tle agenty do tworzenia kodu, ale szczegółowo zostały one opisane w rozdziale 10.

W trakcie lektury tego podrozdziału nie martw się koniecznością zapamiętywania konkretnych nazw narzędzi lub opcji. Środowisko technologiczne szybko się zmienia. Celem jest zrozumienie typów dostępnych rozwiązań.

VSCode i Copilot: zintegrowana platforma programistyczna firmy Microsoft z technologią AI

Dzięki głębokiej integracji z narzędziem GitHub Copilot środowisko VSCode (*https://code.visualstudio.com*) przeszło transformację z najpopularniejszego na świecie edytora kodu w kompleksową platformę programistyczną wspomaganą sztuczną inteligencją. Ewolucja ta odzwierciedla wizję firmy Microsoft, aby utrzymać możliwości technologii AI w znanym środowisku VSCode, z którego każdego dnia korzystają miliony programistów.

GitHub Copilot to asystent tworzenia kodu napędzany sztuczną inteligencją, który jest zintegrowany ze środowiskiem VSCode. Zapewnia on propozycje kodu, wyjaśnienia i automatyczne implementacje na podstawie zapytań formułowanych w języku naturalnym oraz kontekstu istniejącego kodu. Tym, co wyróżnia tę integrację, jest jej płynny charakter. Asystent Copilot nie jest tylko dodatkiem, ale sprawia wrażenie naturalnego rozszerzenia samego edytora.

Rdzeń możliwości technologii AI środowiska VSCode koncentruje się na trzech głównych trybach interakcji. Po pierwsze, jest to *wbudowane automatyczne uzupełnianie kodu*, w ramach którego Copilot dostarcza sugestie kodu bezpośrednio podczas jego pisania, począwszy od uzupełnień pojedynczych wierszy, a skończywszy na implementacjach całych funkcji. W trakcie tworzenia kodu pojawia się tekst w tle z sugestiami, które można zaakceptować klawiszem *Tab* lub częściowo akceptować słowo po słowie.

Po drugie, istnieje *interfejs czatu* dostępny z poziomu panelu bocznego, w którym można prowadzić rozmowy na temat kodu, zadawać pytania lub prosić o konkretne implementacje. Po trzecie, co prawdopodobnie zapewnia największe możliwości, dostępny jest *tryb agenta*, który korzysta z wywoływania narzędzi w celu uzyskania dostępu do powiększającego się zestawu możliwości wewnątrz środowiska Visual Studio. Po uzyskaniu celu tryb ten wybiera i wykonuje odpowiednie narzędzia krok po kroku. Tryb agenta może analizować Twoją bazę kodu, proponować zmiany w wielu plikach, uruchamiać polecenia terminala, reagować na błędy kompilacji i korygować się samodzielnie w ramach pętli do momentu ukończenia zadania.

Tym, co czyni implementację asystenta Copilot środowiska VSCode szczególnie atrakcyjną, jest obsługa protokołu MCP (*Model Context Protocol*). Zapewnia on modelom AI standaryzowany sposób wykrywania zewnętrznych narzędzi, aplikacji i źródeł danych oraz prowadzenia interakcji z nimi. Oznacza to, że asystent Copilot środowiska VSCode może łączyć się z bazami danych, wywoływać interfejsy API, uzyskiwać dostęp do dokumentacji, a także integrować się z całym ekosystemem służącym do projektowania. Na przykład w przypadku włączenia serwera GitHub z protokołem MCP można poprosić asystenta Copilot o „utworzenie zgłoszenia dla każdego omawianego błędu”, a on będzie bezpośrednio współpracował z interfejsem API serwisu GitHub w celu utworzenia tych zgłoszeń. Rozszerzalność za pośrednictwem protokołu MCP przekształca asystenta Copilot z generatora kodu w kompleksowego asystenta procesu programowania, który rozumie nie tylko kod, ale także cały stosowany przepływ pracy.

Aby w profesjonalnym procesie projektowania skutecznie wykorzystać środowisko VSCode z asystentem Copilot, zacznij od eksploracji różnych trybów interakcji w zależności od złożoności realizowanego zadania. W przypadku prostych uzupełnień kodu i refaktoryzacji polegaj na wbudowanych sugestiach i ikonie gwiazdki, która pojawia się przy błędach. Kliknij ją, aby uzyskać poprawki wspierane technologią AI.

Z myślą o bardziej złożonych zadaniach przełącz się na tryb agenta, otwierając panel czatu i wybierając z menu rozwijanego pozycję *Agent*. Tryb agenta zoptymalizowano pod kątem wykonywania autonomicznych zmian w wielu plikach projektu. Jest on szczególnie przydatny w przypadku złożonych zadań, które wymagają nie tylko edycji kodu, ale także wywoływania narzędzi i poleceń terminala. Połączenie znanego interfejsu środowiska VSCode z rozwijającymi się możliwościami technologii AI asystenta Copilot oferuje atrakcyjną opcję zespołom, które wymagają wsparcia przez model AI klasy korporacyjnej bez opuszczania ugruntowanego środowiska programistycznego.

VSCode i Cline: autonomiczny agent tworzenia kodu jako narzędzie open source

Zanim zostaną zaprezentowane środowiska IDE ze sztuczną inteligencją stworzone do konkretnego celu, warto przyjrzeć się temu, jak agent Cline (*https://cline.bot*; wcześniej znany jako Claude Dev) przekształca narzędzie VSCode w potężne środowisko programistyczne wspomagane sztuczną inteligencją. Agent Cline reprezentuje inną filozofię niż asystent Copilot firmy Microsoft. Zamiast być ściśle zintegrowanym asystentem, działa jako autonomiczny agent tworzenia kodu, który od początku do końca może podejmować się złożonych i wieloetapowych zadań programistycznych. To rozszerzenie *open source* wprowadza do środowiska VSCode możliwości, które często przewyższają te dostępne w komercyjnych edytorach z modelem AI, zachowując jednocześnie elastyczność i rozszerzalność, jakiej oczekują użytkownicy środowiska VSCode.

Tym, co wyróżnia narzędzie Cline, jest jego prawdziwie agentowe podejście do procesu projektowania oprogramowania. Po przekazaniu agentowi Cline ogólnej prośby typu „utwórz interfejs API REST do zarządzania użytkownikami z uwierzytelnianiem" nie generuje on po prostu szablonowego kodu. Zamiast tego analizuje strukturę projektu, planuje implementację w wielu plikach, tworzy odpowiednie hierarchie folderów, instaluje niezbędne zależności, a ponadto może nawet uruchamiać testy w celu weryfikacji implementacji. Podczas całego tego procesu agent Cline zachowuje przejrzystość, prezentując każde planowane działanie (tworzenie plików, modyfikacje i wykonywanie poleceń terminala) i oferując możliwość zatwierdzenia lub modyfikacji każdego kroku. Taki projekt *z udziałem człowieka w pętli* zapewnia idealną równowagę między automatyzacją a kontrolą, pozwalając programistom korzystać z możliwości technologii AI przy jednoczesnym zachowaniu nadzoru nad używaną bazą kodu.

Możliwości techniczne agenta Cline wykraczają daleko poza generowanie kodu. Może on wykorzystywać *funkcję automatyzacji przeglądarki* do badania dokumentacji interfejsu API, debugować złożone problemy drogą analizy śladów błędów w wielu plikach, a nawet współdziałać z zewnętrznymi usługami dzięki obsłudze protokołu MCP. Z myślą o debugowaniu możesz wkleić komunikat o błędzie, a agent Cline przeanalizuje stosowaną bazę kodu, aby zidentyfikować główną przyczynę, zaproponować poprawkę, wdrożyć ją i dodać odpowiednią obsługę błędów w celu zapobiegnięcia podobnym problemom. Integracja protokołu MCP oznacza, że agent Cline może połączyć się z Twoją bazą danych, aby zrozumieć schematy przed wygenerowaniem zapytań, uzyskać dostęp do narzędzi zarządzania projektami w celu dostosowania implementacji do wymagań lub współdziałać z dowolną inną usługą zgodną z tym protokołem. Taka rozszerzalność przekształca agenta Cline z generatora kodu w kompleksowego partnera w procesie projektowania, który rozumie cały ekosystem techniczny.

W przypadku zespołów agent Cline oferuje kilka istotnych zalet. Dzięki temu, że jest to rozwiązanie *open source*, zespoły mogą sprawdzać kod agenta, wprowadzać ulepszenia lub tworzyć własne wersje dostosowane do konkretnych potrzeb, co jest kluczowe dla organizacji o szczególnych wymaganiach z zakresu bezpieczeństwa lub zgodności. Agent obsługuje wielu dostawców technologii AI, w tym model Claude firmy Anthropic, modele firmy OpenAI, model Gemini firmy Google, a nawet modele lokalne za pośrednictwem narzędzia Ollama.

Zapewnia to zespołom elastyczność dotyczącą wyboru modelu na podstawie wydajności, kosztów lub wymagań odnoszących się do lokalizacji danych.

Aby skutecznie korzystać z agenta Cline, twórz szczegółowe zapytania uwzględniające kontekst i ograniczenia projektu. Korzystaj z jego zdolności analizowania całej bazy kodu przed wprowadzaniem zmian oraz jego możliwości iteracyjnego projektowania. Po zaimplementowaniu funkcjonalności przez agenta Cline możesz natychmiast ją przetestować i poprosić o udoskonalenia w tym samym kontekście rozmowy. Połączenie dojrzałego ekosystemu środowiska VSCode z autonomicznymi możliwościami agenta Cline oferuje zespołom potężną, elastyczną i opłacalną ścieżkę prowadzącą do projektowania wspomaganego sztuczną inteligencją bez konieczności rezygnowania z istniejących narzędzi i przepływów pracy.

Cursor: edytor kodu wspomagany sztuczną inteligencją

Cursor to jedno z flagowych narzędzi ruchu związanego z programowaniem intuicyjnym. Jest to środowisko IDE wzbogacone o sztuczną inteligencję, które szybko zyskało popularność wśród programistów poszukujących możliwości płynniejszego pisania kodu. Cursor to w zasadzie edytor kodu stawiający technologię AI na pierwszym planie (tak naprawdę wywodzi się on ze środowiska VSCode), który bezpośrednio ze środowiskiem programistycznym integruje najnowocześniejsze możliwości generowania i rozumienia kodu.

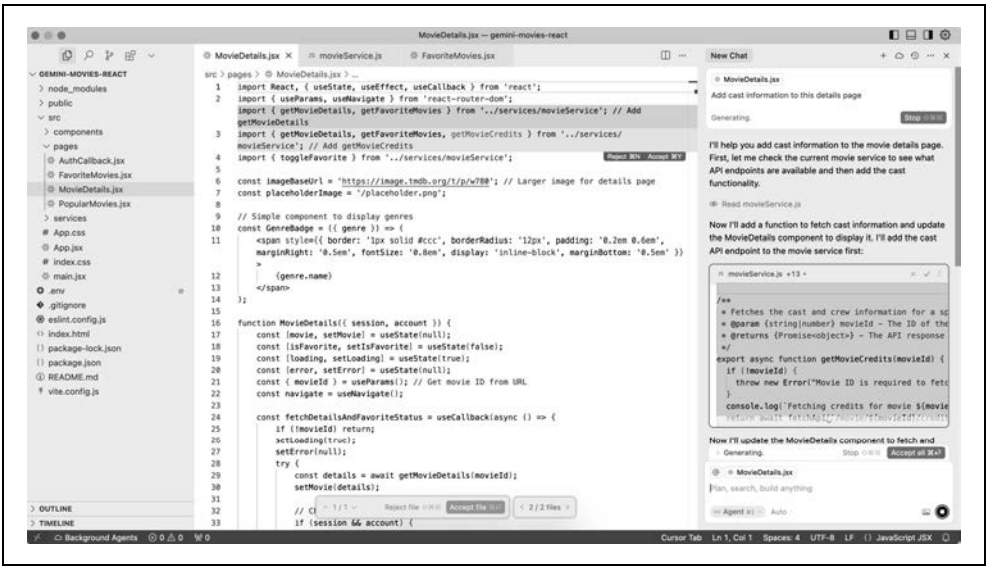
Opis tego narzędzia brzmi: „edytor kodu z modelem AI". Zostało ono tak zaprojektowane, aby umożliwić tworzenie i modyfikowanie kodu za pomocą instrukcji w naturalnym języku. Możesz na przykład zaznaczyć funkcję i poprosić agenta Cursor o „jej optymalizację" lub „dodanie w tym miejscu obsługi błędów", a on natychmiast zasugeruje zmiany w kodzie. Model AI agenta Cursor rozpoznaje kontekst projektu. Indeksuje użytą bazę kodu i rozumie kontekst plików, dlatego może przedstawiać bardziej trafne sugestie (znacznie wykraczające poza zwykłe automatyczne uzupełnianie). Środowisko IDE agenta Cursor integruje w swoim głównym interfejsie możliwości dużych modeli językowych. Jest to jak narzędzie ChatGPT, które zna Twoją bazę kodu.

Do zasilania swoich funkcjonalności w tle agent Cursor stosuje zaawansowane modele językowe (w zależności od konfiguracji jest to często model Claude firmy Anthropic lub są to modele firmy OpenAI). Agent oferuje boczny panel czatu, w którym możesz prowadzić rozmowy na temat swojego kodu, a nawet tryb Composer służący do wieloetapowego generowania kodu. W swoich eksperymentach związanych z programowaniem intuicyjnym sam Andrej Karpathy używał tego trybu agenta Cursor w przypadku modelu o nazwie Sonnet. Taka konfiguracja pozwoliła mu dosłownie *rozmawiać* z edytorem (z użyciem opcji SuperWhisper przekształcającej mowę w tekst) i obserwować, jak pojawia się kod, który następnie akceptował lub ulepszał.

Agent Cursor potrafi nie tylko generować kod, ale także *edytować istniejący kod* na żądanie. Możesz na przykład utworzyć następujące pytanie:

> Czy mógłbyś ułatwić przełączanie certyfikatów w module nasłuchiwania warstwy transportu?

Agent Cursor zrozumie, że odnosisz się do swojego kodu, i zaproponuje bezpośrednie zmiany w odpowiednim pliku lub dokona odczytu właściwych plików, takich jak plik specyfikacji w formacie Markdown (rysunek 1.3). W wersji darmowej agent często przedstawia różnice w czacie do zatwierdzenia. W wersji profesjonalnej może automatycznie zastosować zmiany w Twoim obszarze roboczym.



Rysunek 1.3. Interfejs agenta Cursor to przykład nowszej generacji środowisk IDE zintegrowanych ze sztuczną inteligencją. Dzięki indeksowaniu projektu i iteracyjnemu doskonaleniu zapytań narzędzia takie jak Cursor umożliwiają „pozostawienie uruchomionego edytora, pójście na kawę i powrót do w pełni działających funkcjonalności", co zapewnia wzrost produktywności w tempie wykładniczym

Aby skutecznie korzystać z agenta Cursor w ramach profesjonalnego przepływu pracy, powinieneś systematycznie używać jego możliwości. Zacznij od otwarcia czatu agenta i opisz funkcjonalność lub poprawkę, której wymagasz. Oto przykład: dodaj formularz logowania użytkownika z polami adresu e-mail i hasła wraz z weryfikacją i komunikatami o błędach. Agent Cursor wygeneruje niezbędny kod (tworząc nowe pliki lub modyfikując istniejące) w wersji roboczej. Możesz przejrzeć te zmiany (agent pokazuje różnice lub podgląd), a następnie kliknąć przycisk *Apply*, aby scalić je ze swoją bazą kodu. Wielu programistów stosuje następującą pętlę: zapytanie → przegląd → akceptacja. Jeśli sugestia nie jest idealna, możesz doprecyzować swoje zapytanie (na przykład „Użyj środowiska Tailwind CSS do określenia stylów formularza") lub po prostu poprosić agenta Cursor o usunięcie dostrzeżonych problemów („Zajmij się teraz przypadkiem, gdy adres e-mail jest już zarejestrowany"). Zasadniczo prowadzisz dialog ze swoim kodem, dopóki nie będzie wyglądał dobrze.

Agent Cursor doskonale radzi sobie również ze zrozumieniem błędów i dzienników. Jeśli uruchomisz kod i otrzymasz komunikat o błędzie lub ślad stosu, możesz wkleić go do czatu agenta Cursor. Często model AI będzie analizować go i zasugeruje rozwiązanie. Przekształca to debugowanie w formę współpracy: zamiast ręcznego przeszukiwania za pomocą wyszukiwarki

Google lub witryny Stack Overflow, model AI agenta Cursor nierzadko potrafi wskazać problem, a nawet utworzyć poprawkę. Mimo to warto weryfikować poprawki, ponieważ model AI nie zawsze dobrze zadziała za pierwszym razem.

Oto kolejna rada profesjonalisty: korzystaj z możliwości agenta Cursor pozwalających uwzględniać wiele plików. Aby podczas generowania kodu agent brał pod uwagę całą bazę kodu, możesz wybrać zestaw plików (lub poinformować go o kontekście projektu w zapytaniu). Oto przykład: dodaj po stronie serwerowej nowy punkt końcowy interfejsu API do obsługi formularza logowania i połącz go z formularzem interfejsu, który właśnie utworzyliśmy. Agent Cursor przypomni sobie kod interfejsu, który właśnie wygenerował, i pomoże w stworzeniu odpowiedniej logiki serwerowej. Taki kontekst obejmujący cały projekt to przełom w porównaniu z wcześniejszymi asystentami kodowania, które działały tylko na poziomie poszczególnych plików.

Podsumowując: dysponować agentem Cursor to tak, jakby mieć wsparcie programisty w postaci modelu AI *wewnątrz* środowiska IDE dostępnego przez całą dobę. Jest on intuicyjny (komunikujesz się z nim za pomocą zwykłego języka), a ponadto może bezpośrednio modyfikować Twój kod. Im więcej ćwiczysz dzielenie zadań na części i przekazywanie agentowi Cursor jasnych instrukcji, tym więcej zdołasz osiągnąć w krótkim czasie. Szczególnie znakomicie agent sprawdza się w ramach projektowania iteracyjnego: tworzysz fragment kodu, uruchamiasz go i sprawdzasz wyniki, a następnie od razu prosisz agenta Cursor o dostosowanie lub rozszerzenie kodu, po czym powtarzasz całość.

Windsurf: środowisko IDE ze sztuczną inteligencją i pełnym indeksowaniem bazy kodu

Kolejną wschodzącą gwiazdą w zestawie narzędzi do vibe codingu jest Windsurf, czyli środowisko programistyczne oparte na sztucznej inteligencji, które wynosi rozumienie kodu na nowy poziom. Środowisko, które zostało stworzone przez zespół stojący za narzędziem Codeium (*https://windsurf.com*), wyróżnia się tym, że indeksuje całą bazę kodu i wykorzystuje techniki wyszukiwania, aby w trakcie pracy dostarczać odpowiednie fragmenty modelowi AI. W praktyce oznacza to, że Windsurf doskonale radzi sobie z dużymi projektami, w przypadku których odpowiedź na Twoje pytanie może być rozproszona w wielu plikach. Główny komponent tego środowiska używa czegoś określanego mianem *generowania wspomaganego wyszukiwaniem* (RAG — *Retrieval-Augmented Generation*), co w prostszych słowach oznacza, że wyszukiwane są porcje kodu istotne dla Twojego zapytania i taki kontekst jest dostarczany modelowi AI. Dzięki temu jego sugestie są spójne z istniejącym kodem.

Jak to wygląda z perspektywy projektanta? Załóżmy, że zaznajamiasz się dopiero z dużą bazą kodu i musisz dodać nową funkcjonalność. Używając środowiska Windsurf, możesz sformułować zapytanie w języku naturalnym:

> Gdzie w bazie kodu obsługiwana jest logika uwierzytelniania użytkowników?

Narzędzie to przeszuka indeks i wskaże Ci odpowiedni plik, a nawet funkcję. Możesz następnie otworzyć czat (w środowisku Windsurf nosi on nazwę widoku *Cascade* uruchamianego za pomocą skrótu klawiaturowego *Cmd+L*) i wpisać:

> Dodaj do procesu logowania dwuskładnikowe uwierzytelnianie oparte na telefonie.

Ponieważ środowisko Windsurf dysponuje kontekstem Twojej logiki uwierzytelniania, to aby ją zaimplementować, może wygenerować zmiany obejmujące wiele plików (bazę danych, interfejs API, interfejs użytkownika), podejmując przemyślane decyzje zgodne ze strukturą Twojego systemu.

Tryb zapisu Write środowiska Windsurf może śmiało wprowadzać zmiany za Ciebie: zamiast tylko sugerować różnice na panelu bocznym, utworzy ono nowe pliki lub automatycznie zmodyfikuje istniejące. Może to oznaczać ogromną oszczędność czasu: zamiast kopiować i wklejać z treści sugestii, widzisz w jednym miejscu, jak Twój projekt ewoluuje. W zasadzie środowisko Windsurf próbuje podejmować działania w Twoim imieniu, gdy jest pewne siebie, zachowując się jak niezależny, początkujący programista implementujący funkcjonalności w całej bazie kodu (filozofia działania środowiska Cursor jest nieco bardziej konserwatywna; prosi ono o potwierdzenie, choć jego wersja Pro zawiera również opcję automatycznego zatwierdzania).

Aby skutecznie korzystać ze środowiska Windsurf, warto zrozumieć jego mocne strony:

Pytania i odpowiedzi związane z bazą kodu

Zapytania dotyczące swojej bazy kodu możesz tworzyć w języku naturalnym niemal tak samo jak własne pytania w serwisie Stack Overflow w przypadku realizowanego projektu. Jest to znakomite rozwiązanie w odniesieniu do dużych i starszych projektów, w których znalezienie miejsca, w jakim coś zostało zdefiniowane, może zająć wiele godzin. Korzystając z zaindeksowanego kodu, środowisko Windsurf udzieli odpowiedzi w kilka sekund.

Sugestie z globalnym kontekstem

Ponieważ środowisko Windsurf dostarcza modelowi odpowiednie pliki, może ono obsługiwać takie zadania jak „Dokonaj refaktoryzacji modułu płatności, aby korzystał z nowego narzędzia do rejestrowania, które utworzyliśmy", ponieważ zna zarówno moduł płatności, jak i narzędzie do rejestrowania.

Tryby działania

Środowisko Windsurf oferuje wiele trybów (na przykład Autocomplete, Chat, Command i tryb Cascade, o którym już wspomniano). Tryb Cascade to coś w rodzaju superczatu, który może uwzględnić szerszy kontekst. Tryb Write (w obrębie czatu) faktycznie dokonuje zmian. Jako inżynier możesz zdecydować, ile autonomii przyznasz temu trybowi.

W przypadku zespołu środowisko Windsurf można zintegrować z codziennym procesem projektowania (podobnie jak środowisko Cursor). Dokonując wyboru środowiska, niektórzy programiści preferują narzędzie Windsurf ze względu na jego szybkość i otwartość w podejmowaniu działań (zauważają, że wydaje się ono szybsze pod względem generowania i wprowadzania zmian), a także możliwość indeksowania przy realizowaniu bardzo dużych projektów. Z kolei interfejs środowiska Cursor może wydawać się bardziej znajomy użytkownikom środowiska VSCode. Niekoniecznie jest to wybór typu „albo-albo". Część programistów trzyma pod ręką oba narzędzia, a zespoły mogą dokonać standaryzacji z użyciem jednego z nich.

Podsumowując: środowisko Windsurf to doskonałe narzędzie, jeśli szukasz asystenta AI do tworzenia kodu, który naprawdę „czyta dokumentację i kod" przed rozpoczęciem generowania.

Minimalizuje ono ryzyko wymyślonych funkcji lub błędnie nazwanych zmiennych, ponieważ może je sprawdzić. Aby w pełni wykorzystać to środowisko, przekazuj do niego jasne instrukcje i pozwól mu działać w trybie Write przy dużych zadaniach, ale swobodnie używaj go też w bardziej kontrolowany sposób w przypadku nieznacznych zmian. Zawsze przeglądaj zmiany, które środowisko wprowadza (wskaże Ci je), zwłaszcza w kluczowym kodzie. Środowisko Windsurf jest inteligentne, lecz nie nieomylne. Gdy jest mądrze używane, przypomina hiperinteligentne środowisko IDE, które zna cały Twój projekt i może implementować pomysły w jego obrębie, znacząco zwiększając Twoją produktywność.

Modele AI: rozwiązania do generowania kodu

Rozwiązania do tworzenia kodu oparte na sztucznej inteligencji przeszły dramatyczną transformację. Obecnie o uwagę programistów rywalizuje wiele zaawansowanych modeli, w tym modele z rodzin Claude, Gemini i OpenAI. Tam, gdzie kiedyś mógł dominować jeden model, dzisiejszy ekosystem oferuje bogaty wybór opcji, z których każda charakteryzuje się odrębnymi zaletami czyniącymi je odpowiednimi dla różnych scenariuszy programistycznych.

Kategorie modeli

Dzisiejsze modele do tworzenia kodu można ogólnie podzielić na kilka następujących kategorii w zależności od ich podejścia i mocnych stron:

Zoptymalizowane pod kątem szybkości

Takie modele priorytetowo traktują natychmiastowe odpowiedzi i są idealne do uzupełniania kodu w czasie rzeczywistym oraz do szybkich iteracji. Zazwyczaj oferują mniejsze opóźnienia kosztem nieco ograniczonej dokładności przy złożonych zadaniach.

Dogłębne wnioskowanie

Modele te potrzebują więcej czasu na „przemyślenie" problemów, ale wyróżniają się w przypadku złożonego debugowania, decyzji architektonicznych i wieloetapowego rozwiązywania problemów. Modele z zaawansowanymi możliwościami wnioskowania potrafią dokonywać analizy złożonych błędów krok po kroku.

Multimodalne potęgi

Niektóre modele mogą przetwarzać nie tylko kod i tekst, ale także obrazy i diagramy, a nawet treści wideo. Dzięki temu są one szczególnie cenne w analizie dokumentacji wizualnej lub podczas pracy z elementami UI/UX.

Alternatywy w postaci narzędzi open source

Model DeepSeek wyróżnia się tym, że oferuje poziom mocy sztucznej inteligencji porównywalny z modelami bez dostępnego kodu źródłowego bez konieczności płacenia lub rejestracji. W modelu tym może jednak brakować niektórych funkcjonalności, takich jak generowanie obrazów lub możliwości przeglądania witryn internetowych.

Wybór odpowiedniego modelu pod kątem swojego zadania

Zamiast szukać jednego „najlepszego” modelu, doświadczeni programiści dopasowują obecnie modele do konkretnych zadań:

- Do szybkiego prototypowania i tworzenia ogólnego kodu nadają się dobrze modele zoptymalizowane pod kątem szybkości i szerokiej obsługi języków.
- W przypadku złożonego debugowania i projektowania systemów dobrym wyborem są modele z dogłębnym wnioskowaniem, które potrafią metodycznie prześledzić logikę.
- Z myślą o pracy z dużymi bazami kodu warto wybrać modele z rozszerzonymi oknami kontekstu, które potrafią utrzymać świadomość całego projektu.
- Zespołom dbającym o budżet modele *open source* zapewniają doskonałą wartość bez kosztów subskrypcji.

Wiele narzędzi obsługuje obecnie różne modele sztucznej inteligencji, w tym warianty modeli OpenAI, Claude i Gemini, a także modele zastrzeżone, umożliwiając programistom przełączanie się między nimi w zależności od wykonywanego zadania.

Praktyczne wskazówki dotyczące każdego modelu

Niezależnie od tego, który model sztucznej inteligencji wybierzesz, określone praktyki zawsze poprawiają wyniki. Przede wszystkim dostarczaj rozbudowanych informacji kontekstowych. Nie pytaj jedynie o „funkcję obsługi płatności”. Zamiast tego podziel się swoimi modelami danych, istniejącymi wzorcami kodu, metodami obsługi błędów oraz wszelkimi szczególnymi wymaganiami. Im więcej kontekstu zapewnisz, tym lepiej dane wynikowe będą dopasowane do Twojej bazy kodu.

Większość nowoczesnych modeli do generowania kodu doskonale radzi sobie z przeglądaniem własnych rezultatów. Po otrzymaniu wygenerowanego kodu poproś model o sprawdzenie potencjalnych problemów, zasugerowanie ulepszeń lub wyjaśnienie swojego sposobu wnioskowania. Taka samoocena często pozwala wychwycić subtelne błędy lub zasugerować optymalizacje.

Wykorzystaj zdolność modelu do utrzymywania kontekstu rozmowy. Zacznij od podstawowej implementacji, a następnie stopniowo ją udoskonalaj w ramach kolejnych zapytań. Takie iteracyjne podejście często daje lepsze rezultaty niż próba określenia wszystkiego z góry.

Każdy model ma subtelne różnice dotyczące sposobu podchodzenia do problemów. Niektóre z nich są bardziej szczegółowe w swoich wyjaśnieniach, natomiast inne są bardziej zwięzłe. Część modeli domyślnie używa nowszej składni, a część posługuje się nią z zachowaniem bezpieczeństwa. Poznanie tych tendencji pomaga tworzyć lepsze zapytania.

Główne modele

Rozwiązania do tworzenia kodu z użyciem sztucznej inteligencji zmieniają się co miesiąc. Nowe modele regularnie rzucają wyzwanie dotychczasowym liderom. Konkurencja stała się tak intensywna, że programiści zyskują bezprecedensowe możliwości wyboru i usprawnienia funkcjonalności. Najważniejsze nie jest wybranie „idealnego" modelu, lecz zrozumienie, jak wykorzystać mocne strony dostępnych narzędzi.

Wiele zespołów projektowych stosuje obecnie podejście portfelowe. Wykorzystują one szybkie modele do rutynowych zadań, wydajne modele do złożonych wyzwań oraz wyspecjalizowane modele w konkretnych dziedzinach, takich jak optymalizacja baz danych lub tworzenie interfejsów użytkownika. Niektóre środowiska IDE umożliwiają nawet płynne przełączanie między modelami w trakcie wykonywania zadania.

Sukces wynika ze zrozumienia tych możliwości i strategicznego stosowania ich w celu przyspieszenia procesu projektowania.

Google Gemini: multimodalna potęga programistyczna

Rodzina modeli Gemini firmy Google (*https://gemini.google.com*) reprezentuje fundamentalną zmianę w projektowaniu wspomaganym sztuczną inteligencją, co jest możliwe dzięki ich wbudowanym możliwościom multimodalnym. W przeciwieństwie do modeli trenowanych głównie na tekście i kodzie model Gemini został zaprojektowany od podstaw tak, aby płynnie rozumieć i obsługiwać tekst, kod, obrazy, wideo oraz inne formaty danych. Sprawia to, że jest on wyjątkowo skuteczny w przypadku nowoczesnych procesów projektowania, w których kontekst wizualny ma równie duże znaczenie co informacje tekstowe.

Multimodalna natura modelu Gemini okazuje się szczególnie cenna w scenariuszach tworzenia aplikacji internetowych. Programiści mogą udostępniać zrzuty ekranu z szablonami interfejsów, a model ten potrafi wygenerować implementacje idealnie odwzorowujące styl wizualny. Model doskonale radzi sobie z analizą wykresów, diagramów i elementów interfejsu użytkownika, co czyni go idealnym partnerem przy przekształcaniu projektów wizualnych w funkcjonalny kod. Możliwość ta wykracza daleko poza zwykłe rozpoznawanie obrazów: model Gemini potrafi wnioskować w związku z elementami wizualnymi, rozumieć wzorce projektowe i utrzymywać spójność estetyczną w całym projekcie.

Integracja modelu Gemini z procesami projektowania za pomocą popularnych edytorów (VSCode, Cursor, Windsurf) oraz wtyczek, takich jak Cline i Code Assist, oferuje programistom zaawansowane opcje dostosowywania, które skalują się od indywidualnych preferencji do standardów obowiązujących w całym zespole. Programiści mogą tworzyć niestandardowe polecenia do realizowania powtarzalnych zadań, ustanawiać reguły stosowane w każdym generowaniu kodu oraz utrzymywać spójne wzorce tworzenia kodu w dużych bazach kodu. Pokaźny plan darmowy sprawia, że narzędzie jest dostępne dla studentów, hobbystów i start-upów, natomiast opcje korporacyjne wychodzą naprzeciw złożonym wymaganiom organizacyjnym.

Model Gemini wśród wielu narzędzi programistycznych wyróżnia zdolność do głębokiego analizowania problemów przy zachowaniu praktycznej szybkości działania. Model potrafi przełączać się między szybkimi odpowiedziami w przypadku prostych zadań a rozszerzonym wnioskowaniem dla złożonych wyzwań, dostosowując swoje podejście do danego problemu. W połączeniu ze zdolnościami rozumienia obrazów ta elastyczność czyni ten model szczególnie skutecznym w tworzeniu aplikacji full-stack, w których logika serwerowa i estetyka interfejsu użytkownika mają równie duże znaczenie.

Claude: wirtuoz wnioskowania

Podejście modelu Claude firmy Anthropic (*https://anthropic.com/claude*) do wspomagania programowania koncentruje się na przejrzystości i możliwościach dogłębnego wnioskowania. Rodzina modeli Claude, a zwłaszcza modele Sonnet, charakteryzuje się wyjątkowymi możliwościami dotyczącymi wykonywania złożonych zadań z zakresu inżynierii oprogramowania wymagających uważnej analizy i rozwiązywania problemów krok po kroku. Model Claude wyróżnia się możliwością prezentowania procesu myślenia. Dzięki temu programiści mogą śledzić jego tok rozumowania i weryfikować logikę przed wdrożeniem rozwiązań.

Opcja Artifacts stanowi istotną zmianę w sposobie, w jaki programiści współpracują z asystentami AI służącymi do tworzenia kodu. Zamiast po prostu dostarczać kod w interfejsie czatu, model Claude tworzy wydzielony obszar roboczy, w którym kod można wyświetlać, edytować i podglądać w czasie rzeczywistym. To interaktywne środowisko sprawdza się szczególnie dobrze w tworzeniu interfejsów użytkownika, wizualizacji danych i w każdej sytuacji, w której natychmiastowa informacja zwrotna przyspiesza proces projektowania. Programiści mogą iterować projekty, testować funkcjonalność i dopracowywać implementacje w ramach tej samej rozmowy.

Model Claude demonstruje wyjątkową wydajność w rzeczywistych testach porównawczych wykonywanych w ramach inżynierii oprogramowania, konsekwentnie plasując się w czołówce modeli służących do realizowania takich zadań jak usuwanie błędów, implementacja opcji i refaktoryzacja kodu. Jego mocną stroną jest nie tylko generowanie kodu, ale również rozumienie szerszego kontekstu projektów oprogramowania. Model Claude potrafi analizować istniejące bazy kodu, identyfikować wzorce i antywzorce, sugerować ulepszenia architektury oraz zachowywać spójność z ustalonymi stylami kodowania. Dzięki temu jest on nieoceniony zarówno w nowych projektach, jak i w przypadku utrzymywania starszych systemów.

Sposób, w jaki model zarządza pamięcią i kontekstem, pozwala mu na stopniowe wnioskowanie podczas długich sesji programistycznych. Przy realizowaniu dużych projektów model Claude może wyodrębniać i zachowywać kluczowe informacje o strukturze bazy kodu, decyzjach projektowych i wzorcach specyficznych dla danego projektu. Ta nagromadzona wiedza pozwala mu dostarczać coraz bardziej trafne i kontekstowe sugestie w miarę postępu prac projektowych. Sprawia to wrażenie, że model jest bardziej członkiem zespołu, który z upływem czasu coraz bardziej zaznajamia się z projektem, a nie pełni jedynie roli bezstanowego asystenta.

ChatGPT: wszechstronny towarzysz programisty

Model ChatGPT (*https://openai.com/pl-PL/index/chatgpt/*) ugruntował swój status „szwajcarskiego scyzoryka oficerskiego" wśród asystentów AI tworzących kod. Jest on ceniony nie za wyspecjalizowane opcje, ale za niezwykłą wszechstronność i obszerną bazę wiedzy. Jego miejsce w zestawie narzędziowym programisty jest wyjątkowe. Inne modele mogą integrować się bezpośrednio ze środowiskami IDE lub oferować wyspecjalizowane środowiska programistyczne, natomiast model ChatGPT pełni rolę zawsze dostępnego konsultanta programistycznego, którego programiści mają aktywnego w swojej przeglądarce w trakcie całego dnia pracy.

Interfejs konwersacyjny narzędzia ChatGPT czyni je wyjątkowo skutecznym w eksploracyjnym rozwiązywaniu problemów i nauce. Programiści regularnie używają go do debugowania z „gumową kaczką", wklejając problematyczny kod i analizując kwestie w ramach zwykłej rozmowy. Rozszerzone trenowanie tego modelu pozwala mu rozpoznawać wzorce w praktycznie każdym języku programowania, środowisku i narzędziu będącym w powszechnym użyciu. Czy debuguje się wyrażenie regularne, analizuje niejasny komunikat o błędzie, czy eksploruje dokumentację nieznanej biblioteki, model ChatGPT może dostarczyć trafnych spostrzeżeń uzyskiwanych z jego rozbudowanej bazy wiedzy.

Siła narzędzia ChatGPT tkwi w jego zdolności do wypełniania luki między ludzkim zamiarem a implementacją kodu. Wyróżnia się ono w *tłumaczeniu dwukierunkowym*, czyli przekształcaniu opisów w języku naturalnym do postaci działającego kodu i wyjaśnianiu złożonego kodu za pomocą języka potocznego.

Czyni to ten model nieocenionym w przypadku dokumentacji, sprawdzania kodu i przekazywania wiedzy w zespołach. Programiści mogą wkleić nieznany kod i otrzymać jasne objaśnienia jego funkcjonalności lub mogą opisać żądane zachowanie i uzyskać odpowiednie implementacje oparte na wielu modelach programowania.

Wszechstronność modelu ChatGPT wykracza poza tradycyjne języki programowania, obejmując pliki konfiguracyjne, skrypty, formaty danych i języki specjalistyczne. Wyspecjalizowane narzędzia programistyczne wyróżniają się w swoich wąskich dziedzinach, natomiast model ChatGPT zapewnia wartościową pomoc w całym spektrum zadań związanych z projektowaniem oprogramowania. Ta obszerność czyni go szczególnie przydatnym podczas działań na styku różnych technologii lub w trakcie zajmowania się problemami obejmującymi wiele dziedzin. Zdolność tego modelu do utrzymywania kontekstu w długich rozmowach pozwala programistom iteracyjnie eksplorować złożone problemy i udoskonalać rozwiązania w ramach wspólnego dialogu.

Wybór modelu odpowiedniego do swoich potrzeb

Dostępność tych zaawansowanych asystentów AI do tworzenia kodu oznacza fundamentalną zmianę w zakresie praktyk projektowania oprogramowania. Zamiast traktować je jako konkurencyjne opcje, skuteczni programiści dostrzegają, że każda rodzina modeli zapewnia unikalne, mocne strony w różnych aspektach procesu projektowania. Model Gemini firmy Google

sprawdza się doskonale, gdy kluczowe są kontekst wizualny i wnioskowanie multimodalne, a szczególnie w tworzeniu interfejsów użytkownika (UI/UX) i pracy ze specyfikacjami projektowymi. Model Claude firmy Anthropic błyszczy w sytuacjach wymagających głębokiego wnioskowania, złożonej refaktoryzacji i przejrzystych metod rozwiązywania problemów. Rodzina modeli firmy OpenAI zapewnia niezrównaną wszechstronność i szeroką wiedzę, co czyni ją idealną do nauki, debugowania i wyzwań obejmujących wiele dziedzin.

Wiele zespołów projektowych stosuje obecnie podejście portfelowe, wykorzystując w ramach tego samego projektu różne modele do odmiennych zadań. Typowy przepływ pracy może obejmować użycie modelu Gemini do przekształcania szablonów projektowych w początkowe implementacje, modelu Claude do złożonych decyzji architektonicznych i przeglądów kodu, a modelu ChatGPT do ogólnego rozwiązywania problemów i tworzenia dokumentacji. To wielomodelowe podejście maksymalizuje produktywność dzięki dopasowaniu mocnych stron każdego narzędzia do konkretnych wyzwań programistycznych.

W miarę jak te modele nadal się rozwijają, klucz do skutecznego projektowania wspomaganego przez sztuczną inteligencję tkwi nie w wyborze jednej „najlepszej" opcji, lecz w zrozumieniu tego, jak zapewnić orkiestrację wielu asystentów AI w celu przyspieszenia i ulepszenia każdego aspektu cyklu projektowania oprogramowania.

Ten ekosystem jest młody i szybko się zmienia. Nowi gracze i możliwości pojawiają się co kilka miesięcy. Kluczowy wniosek jest taki, że nie musisz od podstaw budować własnego rozwiązania opartego na sztucznej inteligencji, aby móc skorzystać ze świadomego programowania. Istnieje mnóstwo narzędzi, które sprawiają, że takie możliwości są na wyciągnięcie ręki. W książce omówię różne platformy i sposób, w jaki wpisują się one w przepływ pracy związany z programowaniem intuicyjnym.

Zalety i ograniczenia vibe codingu: zróżnicowany punkt widzenia

Ważne jest, aby rozpoznać sytuacje, w których projektowanie wspomagane sztuczną inteligencją naprawdę się sprawdza, a także te, gdzie wciąż może ono zawodzić. Przyjrzyjmy się kilku idealnym przypadkom użycia, w których kodowanie intuicyjne doskonale się sprawdza. Przeanalizujmy też sytuacje, w których dostępna obecnie sztuczna inteligencja nadal ma trudności lub wymaga znacznej interwencji człowieka.

Idealne przypadki użycia vibe codingu

Tak jak określone architektury są odpowiednie w konkretnych zastosowaniach, tak vibe coding ma swoje optymalne obszary zastosowań w zakresie projektowania oprogramowania.

Tworzenie produktu od zera do pierwszej wersji

Vibe coding to przełom w inicjowaniu zupełnie nowych projektów. Spopularyzowane przez Petera Thiela pojęcie *od zera do pierwszej wersji* (ang. *zero-to-one*) odnosi się do tworzenia czegoś całkowicie nowego od podstaw. Dzięki sztucznej inteligencji możesz pokonać etap od pustej kartki do funkcjonalnego prototypu w błyskawicznym tempie. Wymagasz uruchomienia aplikacji internetowej, która nigdy wcześniej nie istniała? W ramach jednej intensywnej sesji z wysyłanymi zapytaniami możesz wygenerować kod szablonowy interfejsu użytkownika, składników serwerowych, schematu bazy danych, a nawet skryptów wdrożeniowych. Jest to idealne rozwiązanie w przypadku start-upów lub projektów maratonów programistycznych mających na celu szybką weryfikację pomysłu. Zamiast spędzać tygodnie na budowaniu „rusztowania" projektu (całości powtarzalnego kodu konfiguracyjnego), możesz zlecić to zadanie modelowi AI, aby zrealizował je w kilka minut.

Wielu programistów opowiada, jak przez weekend z pomocą sztucznej inteligencji jako partnera programistycznego utworzyło produkt o minimalnej funkcjonalności. Jest to coś, co wcześniej oznaczało miesiąc pracy w pojedynkę. Dzięki szybkiemu przekształceniu pomysłu w działający produkt możesz znacznie wcześniej rozpocząć testowanie go z udziałem użytkowników lub interesariuszy. Model AI świetnie radzi sobie z ogólnymi zadaniami (konfiguracją routingu, podstawowymi komponentami interfejsu oraz typowymi operacjami takimi jak tworzenie, odczytywanie, aktualizowanie i usuwanie), co pozwala Ci skupić się na nowatorskich aspektach produktu.

Gdy jednak Twój produkt o minimalnej funkcjonalności zyska na popularności i zmierza do fazy produkcji, musisz zmienić podejście. Tu właśnie niezbędna staje się inżynieria wspomagana sztuczną inteligencją. Choć kodowanie intuicyjne pomogło Ci szybko dokonać eksploracji i weryfikacji, skalowanie wymaga obecnie bardziej przemyślanych praktyk. Konieczna będzie refaktoryzacja takiego szybko wygenerowanego kodu z użyciem odpowiedniej obsługi błędów, a także uwzględnienie pełnego pokrycia testami i ustanowienie jasnych granic architektonicznych. Przejście z prototypu do produktu oznacza naturalną ewolucję od eksploracyjnej swobody towarzyszącej kodowaniu intuicyjnemu do uporządkowanej dyscypliny związanej z inżynierią. Mądre zespoły dostrzegają ten punkt zwrotny i odpowiednio dostosowują sposób wykorzystania przez siebie sztucznej inteligencji. Utrzymują one tempo działań, a jednocześnie wprowadzają zabezpieczenia niezbędne do zapewnienia zrównoważonego wzrostu.

Prototypowanie funkcji i aplikacje z operacjami CRUD

Znaczna część inżynierii oprogramowania, zwłaszcza w aplikacjach biznesowych, uwzględnia operacje CRUD (*Create, Read, Update, Delete*), czyli tworzenie, odczyt, aktualizację i usuwanie danych. Jest to schematyczna praca, w ramach której sztuczna inteligencja radzi sobie wyjątkowo dobrze, ponieważ zaznajomiła się z niezliczonymi przykładami takich rozwiązań. Jeśli, przykładowo, musisz dodać nowy moduł magazynu do swojego systemu z oknami operacji CRUD i interfejsami API, vibe coding pozwala poradzić sobie z tym znakomicie. Może umożliwić wygenerowanie kodu migracji bazy danych, modele odwzorowania ORM (*Object-Relational Mapping*), punkty końcowe interfejsu API oraz formularze interfejsu użytkownika

z weryfikowaniem poprawności. Jest to w zasadzie cały stos technologiczny w dużej mierze pozbawiony błędów, ponieważ te wzorce są tak powszechne w danych treningowych modelu. Jeśli nawet Twoja aplikacja stosuje niestandardowe reguły, możesz je określić w zapytaniu i otrzymać przyzwoitą pierwszą wersję. Rezultat: to, co kiedyś było tygodniowym zadaniem polegającym na nudnym łączeniu elementów, staje się popołudniową sesją z zapytaniami i testowaniem. W przypadku generowania narzędzi wewnętrznych lub paneli administracyjnych (które w gruncie rzeczy są dużymi aplikacjami z operacjami CRUD) możesz niemal całkowicie polegać na modelu AI, biorąc pod uwagę to, że zwykle te narzędzia lub panele są proste, choć czasochłonne.

Podejście inżynieryjne staje się kluczowe, gdy te operacje CRUD obejmują złożoną logikę biznesową, reguły weryfikacji danych lub integrację z istniejącymi systemami. Vibe coding pozwala szybko wygenerować podstawową strukturę, natomiast inżynieria wspomagana sztuczną inteligencją zapewnia, że Twój moduł magazynu prawidłowo obsługuje przypadki brzegowe, takie jak równoczesne aktualizacje, a ponadto zachowuje integralność referencyjną i przestrzega ustalonych wzorców Twojej firmy. Vibe coding możesz na przykład zastosować do wygenerowania początkowego szkieletu operacji CRUD, a następnie przełączyć się na tryb inżynieryjny, aby zaimplementować reguły specyficzne dla domeny, takie jak alerty dotyczące progów magazynowych, logika alokacji w wielu magazynach lub integracja z istniejącymi systemami uwierzytelniania i autoryzacji. Kluczem jest rozpoznanie tego, kiedy należy dokonać przejścia z szybkiego generowania na staranne dopracowywanie.

Kod łączący i integracja

Czy musisz zintegrować ze sobą dwie usługi lub interfejsy API? Wiąże się to zwykle z czytaniem dokumentacji i pisaniem kodu, który przekształca dane z jednego formatu w drugi. Modele AI były często trenowane za pomocą dokumentacji interfejsów API oraz przykładów kodu, co oznacza, że mogą one znacznie przyspieszyć działania w ramach integracji. Poproś model ChatGPT, aby pokazał, jak wywołać interfejs API usługi A z poziomu kodu napisanego w języku B. Jest spora szansa, że wygeneruje on przykładowy kod z odpowiednimi punktami końcowymi, a może nawet z przykładem uwierzytelniania. Łączenie wielu systemów (na przykład podpięcie bramy płatności do systemu zamówień czy połączenie z zewnętrznym pakietem SDK służącym do analityki) staje się łatwiejsze, gdy model AI może zaproponować szablonowy kod i obsługę przypadków brzegowych. Szczególnie dobrze model radzi sobie z takimi standardowymi wzorcami integracji.

Wykorzystanie nowoczesnych środowisk

Asystenci AI do tworzenia kodu skutecznie przyswoili sobie dokumentację wszystkich popularnych środowisk, takich jak React, Angular, Django, Rails, Node/Express i Flutter. Oznacza to, że jeśli korzystasz z dobrze znanych środowisk, sztuczna inteligencja może generować dla nich kod idiomatyczny. Na przykład model AI może utworzyć nowy komponent środowiska React z punktami zaczepienia oraz zarządzaniem stanem lub nowy model środowiska Django z odpowiednią klasą administrowania i serializatorem.

Korzyść polega na tym, że nie musisz pamiętać każdego szczegółu. Sztuczna inteligencja wypełnia luki. Vibe coding sprawdza się szczególnie dobrze przy zadaniach związanych z tworzeniem nowoczesnych aplikacji internetowych, takich jak generowanie kodu HTML/JSX z odpowiednimi klasami lub podłączanie punktów końcowych kontrolera, ponieważ są to zadania, z którymi modele AI miały do czynienia wielokrotnie. To jak posiadanie zawsze u boku eksperta od środowisk, który pisze kod szablonowy, a Ty decydujesz o szczegółach, jakie funkcjonalność powinna zapewnić.

Powtarzalne generowanie kodu

Czasami trzeba stworzyć wiele podobnych fragmentów kodu (na przykład wiele podobnych punktów końcowych lub klas dla każdego typu w jakimś schemacie). W przypadku człowieka może to być żmudne i podatne na błędy. Z kolei model AI uwielbia powtarzające się struktury. Wystarczy zaprezentować mu jeden lub dwa przykłady, a on potrafi konsekwentnie wygenerować resztę. Takie masowe generowanie kodu może zaoszczędzić mnóstwo czasu. Jeśli na przykład tworzysz klasy modeli danych dla 50 typów rekordów, możesz podać w zapytaniu jeden przykład i poprosić model AI o wygenerowanie klas dla wszystkich tych typów zgodnie z tym wzorcem. Prawdopodobnie model wykona to bezbłędnie i w ciągu kilku sekund. W efekcie nie zmarnujesz całego dnia na monotonne pisanie kodu.

Kiedy inżynieria wspomagana sztuczną inteligencją powinna mieć pierwszeństwo?

Choć vibe coding sprawdza się w pewnych sytuacjach, inżynieria wspomagana sztuczną inteligencją staje się niezbędna w innych przypadkach. Zrozumienie tych okoliczności pomaga programistom wybrać właściwe podejście od samego początku, unikając kosztownego przebudowywania kodu lub długu technicznego. Złożone implementacje algorytmiczne wymagają podejścia inżynieryjnego. Gdy tworzysz zaawansowane struktury danych, implementujesz algorytmy krytyczne z punktu widzenia wydajności lub rozwiązujesz nowatorskie problemy obliczeniowe, wymagasz dokładnej kontroli nad każdym aspektem implementacji.

W tym przypadku sztuczna inteligencja pełni rolę kompetentnego asystenta, a nie generatora kodu. Możesz poprosić model AI o wyjaśnienie metod algorytmicznych lub dokonanie przeglądu implementacji pod kątem poprawności, ale zachowujesz bezpośrednią kontrolę nad decyzjami dotyczącymi architektury i optymalizacji. Model pomaga Ci przemyśleć problemy, zamiast rozwiązywać je hurtowo.

Systemy o kluczowym znaczeniu od samego początku wymagają rygoru inżynieryjnego. W przypadku transakcji finansowych, aplikacji medycznych, infrastruktury bezpieczeństwa i innych dziedzin wysokiego ryzyka nie można sobie pozwolić na eksploracyjny charakter kodowania intuicyjnego. W takich kontekstach każdy wiersz kodu wymaga starannego przemyślenia, kompleksowych testów i często zgodności z przepisami. Model AI wspiera działania, sugerując najlepsze praktyki, identyfikując potencjalne luki w zabezpieczeniach i pomagając w zapewnieniu zgodności ze standardami, ale to programista zachowuje ścisłą kontrolę nad implementacją.

Koszt awarii w tych systemach znacznie przewyższa wszelkie korzyści wynikające z szybkości generowania kodu. Integracja ze starszymi systemami stwarza unikalne wyzwania, w ramach których dyscyplina inżynieryjna ma zasadnicze znaczenie. W trakcie pracy z bazami kodu liczącymi dziesięciolecia, zastrzeżonymi protokołami lub systemami z obszernym długiem technicznym często zawodzi dopasowywanie wzorców charakterystyczne dla programowania intuicyjnego. Takie scenariusze wymagają dogłębnego zrozumienia istniejących ograniczeń, starannego planowania punktów integracji i metodycznego refaktoryzowania. Model AI może pomóc dzięki wyjaśnianiu wzorców starszego kodu lub sugerowaniu strategii modernizacji, ale faktyczna implementacja wymaga precyzji, którą zapewnia jedynie uporządkowane podejście inżynieryjne.

Optymalizacja wydajności to kolejna dziedzina, w której inżynieria przeważa nad programowaniem intuicyjnym. Choć sztuczna inteligencja potrafi szybko generować funkcjonalny kod, rzadko tworzy optymalne rozwiązania na potrzeby ścieżek krytycznych z punktu widzenia wydajności. Takie zadania jak zarządzanie pamięcią, optymalizacja pamięci podręcznej, przetwarzanie równoległe lub redukcja opóźnień wymagają dokładnego zrozumienia sprzętu, systemów operacyjnych i złożoności algorytmicznej. Tutaj model AI sprawdza się najlepiej jako asystent badawczy pomagający eksplorować techniki optymalizacji lub porównywać różne podejścia, natomiast to Ty podejmujesz świadome decyzje dotyczące implementacji.

W takich sytuacjach rozpoznawanie wzorców przez model AI i jego szybkość działania idealnie pasują do realizowania zadania. Zasadniczo vibe coding sprawdza się najlepiej w przypadku zadań, które w świecie programowania są dobrze rozpoznanym terytorium (na przykład operacje CRUD lub typowe struktury aplikacji internetowych), a także tych korzystających z szybkich metod prób i błędów (prototypów, nowych pomysłów). Można to porównać do początkującego programisty, który zaznajomił się z każdym repozytorium w serwisie GitHub i może natychmiast przypomnieć sobie, jak zwykle się to realizuje, a następnie napisać kod, który przejrzysz. Jest to niezwykle potężne narzędzie do szybkiego wprawiania różnych rzeczy w ruch.

Rozpoznawanie momentów przejścia

Sztuka nowoczesnego projektowania wspomaganego przez model AI polega nie na wyborze jednego podejścia kosztem drugiego, lecz na rozpoznawaniu momentu, *kiedy należy przejść z jednego na drugie*. Skuteczni programiści wykształcają w sobie intuicję pozwalającą wyczuwać te punkty zwrotne. Czy zaczynasz tworzyć nową funkcjonalność? Rozpocznij od kodowania intuicyjnego, aby szybko zbadać możliwości. Czy zauważasz, że kod staje się złożony lub ma styczność z kluczowymi systemami? Przejdź do trybu inżynieryjnego. Czy opracowujesz weryfikację koncepcji w celu zademonstrowania klientowi? Vibe coding szybko Cię do tego doprowadzi. Czy przekształcasz tę weryfikację w system produkcyjny? Pora na dyscyplinę inżynieryjną.

Taka płynność, czyli umiejętność bezproblemowego przechodzenia od szybkiej eksploracji do starannego procesu budowania, wyróżnia naprawdę skutecznych programistów korzystających ze sztucznej inteligencji. Rozumieją oni, że programowanie intuicyjne i inżynieria wspomagana przez model AI to narzędzia wzajemnie się uzupełniające w ich arsenale, z których

każde jest odpowiednie na różnych etapach cyklu projektowego. Celem nie jest opowiedzenie się po którejś stronie, lecz strategiczne wykorzystanie obu podejść, z maksymalizacją zarówno szybkości, jak i jakości w całym procesie tworzenia oprogramowania.

Gdzie sztuczna inteligencja wciąż sobie nie radzi?

Choć obecne narzędzia do tworzenia kodu oparte na sztucznej inteligencji są imponujące, nie są żadną magią. Z niezawodnym rozwiązywaniem problemów z niektórych kategorii model AI nadal ma trudności. Często wymagają one spostrzeżeń człowieka lub tradycyjnych technik tworzenia kodu. Znajomość tych ograniczeń pomaga ustalić właściwe oczekiwania i zaplanować, kiedy warto polegać na sztucznej inteligencji, a kiedy lepiej przejąć kontrolę.

Ograniczenia obejmują następujące obszary:

Bardzo złożone systemy

Jeśli masz do czynienia z bardzo złożonymi algorytmami lub nowatorskimi problemami, z którymi model AI prawdopodobnie wcześniej nie miał do czynienia, może się on pogubić. Na przykład tworzenie zupełnie nowego algorytmu na podstawie artykułu naukowego albo kompilatora lub systemu o wysokim stopniu współbieżności to zadania obejmujące zawiłą logikę wymagającą prawdziwego zrozumienia i często kreatywnych przeskoków myślowych. Model AI może próbować, ale może popełnić subtelne błędy.

W tak złożonych dziedzinach skłonność sztucznej inteligencji do tworzenia kodu w przybliżeniu poprawnego, lecz nie do końca dokładnego, może prowadzić do wielu poprawek. Jak dowiesz się z rozdziałów 3. i 4., ostatnie około 30% poprawności jest bardzo trudne do osiągnięcia przez model AI. Wiąże się to z tym, co nazywam *problemem 70%*. Sztuczna inteligencja szybko prowadzi Cię przez większość drogi, ale ostatnia część jest trudna. Doświadczony programista może w przypadku takich złożonych zadań użyć modelu AI do generowania szkieletów lub funkcji pomocniczych, ale podstawową logikę tworzy sam.

Optymalizacje niskopoziomowe i programowanie systemowe

Obecne modele AI są głównie trenowane z użyciem języków wysokiego poziomu i abstrakcji. Jeśli musisz wykonywać niskopoziomowe operacje bitowe, tworzyć mocno zoptymalizowany kod C dla konkretnego mikrokontrolera lub generować zwektoryzowane instrukcje SIMD, model AI może okazać się zawodny. Może on utworzyć kod, który wygląda wiarygodnie, ale tak naprawdę na poziomie sprzętowym nie jest optymalny, a nawet poprawny.

Podobnie model AI nie może się pochwalić faktycznym pojęciem o zarządzaniu pamięcią lub ograniczeniach czasu rzeczywistego (nie symuluje w swojej „głowie” pamięci podręcznej procesora). Z tego powodu w przypadku kodu kluczowego pod względem wydajności warto albo dokładnie przetestować sugestie modelu AI, albo utworzyć odpowiednie fragmenty kodu ręcznie. Model może jednak nadal pomóc, dostarczając szablon początkowy lub objaśniając assembler. Nie można jednak ślepo mu ufać w takich scenariuszach.

Unikalne lub niszowe środowiska

Jeżeli używasz bardzo nowego lub mało znanego środowiska, które nie istniało podczas trenowania modelu AI, nie będzie on o nim wiedział. Wówczas model może próbować generalizować albo tworzyć kod, który wygląda na pasujący, ale w rzeczywistości wywołuje nieistniejące funkcje (halucynacje) lub używa przestarzałych wersji interfejsu API. Jeśli na przykład nowa wersja środowiska aplikacji internetowych pojawiła się w zeszłym miesiącu z przełomowymi zmianami, model AI nie będzie dysponował informacjami o tych zmianach. Może zapewnić Ci kod dla starej wersji. W takich sytuacjach musisz wspierać się dokumentacją i być może nawet konieczne będzie trenowanie modelu AI w celu dostarczenia mu w swoim zapytaniu kontekstu z dokumentacji (w zasadzie ucząc go na bieżąco).

Kreatywny projekt interfejsu (UI/UX)

Jeśli poprosisz model AI o zaprojektowanie zupełnie nowatorskiego interfejsu użytkownika lub metody pracy, nie sprawdzi się on najlepiej w takich kreatywnych przeskokach myślowych. Może wygenerować kod interfejsu użytkownika dla znanych wzorców (na przykład standardowy formularz lub panel kontrolny). Jeśli jednak oczekujesz innowacyjnego interfejsu użytkownika bez wyraźnych precedensów, model AI może nie zapewnić Ci czegoś inspirującego. Może po prostu posklejać znane komponenty. Projektanci i programiści tworzący interfejsy użytkownika są nadal bardzo potrzebni do opracowywania nowych metod pracy użytkowników. Mówiąc językiem programowania, model AI może szybko stworzyć standardowo wyglądający interfejs, ale w celu uzyskania takiego specjalnego i niestandardowego rozwiązania musisz go pokierować lub wprowadzać ręcznie ulepszenia.

Interpretowanie intencji i wymagań

Czasami model AI ma trudności, gdy wymagania są ukryte lub sprzeczne. Nie potrafi w pełni zrozumieć ostatecznego celu poza tym, co mu wprost przekażesz. Jeżeli wymagania są niejasne („zrób to skutecznie” — co to dokładnie oznacza?), model AI może nie odgadnąć, na czym Ci zależy (na przykład pamięć w zestawieniu z szybkością). Ludzie lepiej radzą sobie z wyjaśnianiem intencji, szczególnie gdy adresatami są interesariusze bez wiedzy technicznej. Model AI może też źle zinterpretować instrukcje, zwłaszcza gdy nie zna kontekstu domenowego (na przykład reguł biznesowych). Może utworzyć logicznie poprawne rozwiązanie, które w rzeczywistości nie rozwiązuje prawdziwego problemu, ponieważ niuans został zatracony w przekazie.

Oto dobry przykład scenariusza łączącego te kwestie: wyobraź sobie projektowanie nowego silnika grafiki trójwymiarowej (złożony system) w języku Rust (poziom systemowy, kluczowe pod względem wydajności). Dysponujesz nowatorskimi algorytmami renderowania (unikalne problemy). Model AI mógłby pomóc napisać trochę kodu szablonowego, ale w przypadku głównych składników w dużej mierze polegałbyś na ludzkiej pomysłowości. Model AI mógłby pomóc Ci rozpocząć od określenia okna i podstawowej pętli renderowania (typowe zadania), ale przy specjalistycznych częściach postępowałbyś zgodnie z tradycyjnym i ostrożnym programowaniem. Być może otrzymałbyś od modelu pomoc algorytmiczną w postaci pseudokodu.

Jeślibyś poprosił go o optymalizację pętli wykonującej się bardzo często w assemblerze, musiałbyś zweryfikować każdą instrukcję.

Sztucznej inteligencji brakuje także umiejętności prawdziwego wglądu w proces rozwiązywania problemów. W gruncie rzeczy jej sposób działania to dopasowywanie wzorców. Jeśli zatem Twój problem wymaga spostrzeżenia w stylu *aha!*, model AI może po prostu błądzić, prezentując rzeczy, które wyglądają jak kod, ale problemu nie rozwiązują. Tu właśnie sytuację może uratować człowiek, który cofnie się o krok, pomyśli abstrakcyjnie lub sięgnie po realne doświadczenie. Gdy już dysponujesz takim spostrzeżeniem, możesz wykorzystać model AI do szybkiego wdrożenia rozwiązania.

Zrozumienie tych mocnych i słabych stron zapewni, że będziesz stosować techniki programowania intuicyjnego w odpowiednich sytuacjach. Aby zmaksymalizować sukces, zastosuj model AI do tego, w czym jest dobry (do znanych wzorców), a swoją kreatywnością wykaż się w unikalnych częściach aplikacji. Bądź gotowy do interwencji w tych obszarach, w których model AI notorycznie ma problemy. Na przykład dokładnie przejrzyj każdy wygenerowany przez niego kod związany z bezpieczeństwem, ponieważ model może przegapić jeden lub dwa przypadki brzegowe.

Używaj modelu AI jako uzupełnienia mocnych stron ludzi: pozwól mu zająć się ogólnymi działaniami (mnóstwem kodu, szablonem), natomiast sam zajmij się szczegółami (złożoną logiką, architekturą). Używaj go jako wspomagacza tam, gdzie się świetnie sprawdza, i nie bój się przejąć kontroli na trudniejszych odcinkach drogi. Metoda ta wykorzystuje mocne strony obu podejść i zapewnia najlepszy rezultat. Wiedza o tym, kiedy użyć modelu AI, a kiedy polegać na ludzkich umiejętnościach, sprawi, że staniesz się bardzo efektywnym programistą w tej nowej erze.

Każdej nowej technologii towarzyszą atuty i zastrzeżenia. Gdy korzystamy ze wzrostu produktywności i kreatywności wynikającego z projektowania wspomaganego przez modele AI, ważne jest, aby podchodzić do tego z wnikliwym zrozumieniem ograniczeń i kompromisów. Oto kluczowe korzyści:

Krótsze cykle projektowe

Projekty mogą znacznie szybciej przejść od koncepcji, przez prototyp, do gotowego produktu. Model AI potrafi w mgnieniu oka wygenerować kod szkieletowy (na przykład konfigurację szablonu dla nowego projektu), dlatego możesz spędzać więcej czasu nad unikalnymi częściami swojej aplikacji.

Ulepszone prototypowanie i eksperymentowanie

Ponieważ koszt wypróbowania czegoś jest niższy (wystarczy opisać modelowi AI, czego chcesz, aby szybko otrzymać od niego wersję roboczą), programiści mogą czuć się swobodniej podczas eksperymentowania. Możesz tworzyć prototypy wielu podejść do problemu, kierując do modelu AI różne zapytania, a potem wybrać najlepsze rozwiązania. Taka iteracyjna wymiana pomysłów może prowadzić do bardziej kreatywnych rozwiązań.

Wiedza na wyciągnięcie ręki

Duże modele językowe są trenowane na ogromnej ilości wiedzy programistycznej. Często są im znane rozwiązania z niejasnymi interfejsami API lub komunikatami o błędach. W praktyce modele mogą ujawniać rozwiązania lub pomysły, o których mogłeś nie pomyśleć, czyniąc Cię osobą skuteczniej rozwiązującą problemy.

Spójność i standaryzacja

W przypadku zespołów dzięki generowaniu kodu o spójnym stylu asystent AI może pomóc w wymuszaniu standardów tworzenia kodu i najlepszych praktyk. Jeśli zostanie on skonfigurowany zgodnie z wytycznymi przewodnika stylów Twojego projektu, może to zapewnić, że kod od każdej osoby będzie bazował na podobnych wzorcach. Nawet bez wyraźnego treningu modele AI często generują kod idiomatyczny (ponieważ uczyły się na milionach przykładów). Może to zmniejszyć nakład pracy związany z przeglądem kodu, ponieważ funkcje modelu mogą wyglądać znajomo i domyślnie przestrzegać powszechnych konwencji.

Oto niektóre ograniczenia i kompromisy do rozważenia:

Zmienna jakość wyników

Modele nie są nieomylne. Mogą generować kod, który wygląda poprawnie, ale zawiera subtelne błędy lub nieefektywności. Mogą też wybrać przestarzałą metodę, ponieważ ich dane treningowe zawierały dużo starszego kodu. Jako programista musisz zachować czujność. Tak jak nie skopiowałbyś kodu z internetu bez jego zrozumienia, tak nie powinieneś bezmyślnie akceptować kodu generowanego przez model AI. W drugiej części książki zostaną omówione techniki dokładnego weryfikowania i testowania kodu wygenerowanego przez sztuczną inteligencję.

Niejednoznaczność w zapytaniach prowadzi do niejednoznaczności w kodzie

Jeśli Twoje zapytanie nie jest wystarczająco dokładne, model AI musi odgadywać Twoje intencje, dlatego może sobie z tym nie poradzić. Jeśli na przykład poinstruujesz go, aby „posortował listę imion", może domyślnie zastosować sortowanie alfabetyczne, ale może miałeś na myśli coś innego (na przykład sortowanie według długości imienia). Model AI nie stwierdzi różnicy, dopóki mu tego nie wyjaśnisz. Właśnie dlatego precyzja w zapytaniach (jest to tematem rozdziału 2.) jest kluczowa. Nauczysz się przewidywać, jakie szczegóły musisz doprecyzować.

Nadmierne poleganie i zanik umiejętności

Jeśli w pisaniu kodu początkujący programiści zawsze polegają na modelu AI, to czy wykształcą tak samo głęboką znajomość algorytmów i procesu debugowania? Istnieje ryzyko zaniku umiejętności podobnie jak w przypadku polegania na systemie GPS w nawigacji, co może osłabić Twoje poczucie kierunku. Aby temu zapobiec, ważne jest stosowanie modelu AI jako narzędzia do nauki (zwracaj uwagę na kod, jaki dostarcza, i pytaj dlaczego), a ponadto czasami ćwiczenie programowania bez niego w celu zachowania podstawowych umiejętności.

Obawy dotyczące prywatności i bezpieczeństwa

Korzystanie z chmurowych narzędzi AI do tworzenia kodu często oznacza wysyłanie swojego kodu (który może być zastrzeżony lub poufny) do usługi zewnętrznej w celu analizy. Firmy muszą to brać pod uwagę. Wiele narzędzi rozwiązuje ten problem, udostępniając modele lokalne lub dając gwarancje, że nie przechowują kodu, ale to wciąż wymaga rozważenia. Istnieje też ryzyko, że model AI może nieumyślnie wygenerować kod bardzo podobny do czegoś z jego danych treningowych, co może być objęte licencją *open source* (na przykład GPL). Choć jest to mało prawdopodobne (i wprowadzono środki zapobiegające pełnym wynikom w dosłownej formie), podkreśla to potrzebę przeglądu i zrozumienia tego, co model tworzy, zanim zostanie to zintegrowane. W rozdziale 8. zagłębiono się w kwestie bezpieczeństwa i niezawodności.

Stronniczość w wynikach modeli AI

Modele AI mogą odzwierciedlać uprzedzenia obecne w ich danych treningowych. W kontekście tworzenia kodu może to być tak niewinne jak preferowanie określonych nazw zmiennych lub tak znaczące jak używanie przykładów zakładających konkretne atrybuty użytkownika. Na przykład model może używać nazwy `foo` lub `bar` dla każdej przykładowej zmiennej (ponieważ w wielu przykładach tak postąpiono) lub może zakładać rzeczy dotyczące ustawień regionalnych użytkowników. Zwykle nie stanowi to dużego problemu w przypadku generowania kodu w porównaniu z innymi zastosowaniami sztucznej inteligencji, ale warto być świadomym tej możliwości. Bardziej subtelnie model AI może być stronniczy w odniesieniu do rozwiązań, które napotkał częściej, nawet wtedy, gdy nie są one najlepsze z w Twoim przypadku. W rozdziale 9. omówiono stronniczość i inne kwestie etyczne.

Czynniki ludzkie i zaufanie

Nie wszyscy programiści od razu czują się komfortowo z tym stylem pracy. Tworzeniu kodu towarzyszą pewna przyjemność i artyzm, a niektórzy mogą czuć, że jest to pomniejszane przez zaangażowanie sztucznej inteligencji. Może też wystąpić początkowy brak zaufania („Czy naprawdę model zrobił to dobrze?”), który można przezwyciężyć tylko dobrymi praktykami i czasem. Zespoły adaptujące model AI powinny przewidzieć okres przystosowania oraz zachęcać do dzielenia się doświadczeniami i wskazówkami. Z czasem, jak z każdym narzędziem, większość ustali stan równowagi, w którym wkład modeli AI jest doceniany, a wiedza ludzi skupi się na tym, z czym radzą sobie one najlepiej.

Podsumowanie i dalsze kroki

Intuicyjne przejście w kierunku świadomego programowania niesie ze sobą ogromny potencjał, który może uczynić proces tworzenia oprogramowania szybszym, bardziej dostępnym i pod wieloma względami przyjemniejszym. Uświadomienie sobie tego potencjału oznacza jednak zrozumienie nowej dynamiki, czyli tego, jak skutecznie komunikować się z modelem AI, jak weryfikować jego wyniki i jak odpowiedzialnie uwzględnić go w procesie projektowania.

Mój punkt widzenia wynikający z używania tych narzędzi i obserwowania wielu projektów jest taki, że najlepsze zastosowanie sztucznej inteligencji polega na połączeniu kreatywnej intuicji z solidną higieną inżynieryjną. Zachęcaj różne osoby do śmiałych pomysłów i szybkich wersji roboczych, które model AI może oferować. Są to nowe supermoce, którymi dysponujemy. Jednakże kieruj nimi z mądrością, którą wraz z rozwojem oprogramowania gromadzono przez dziesięciolecia, czyli mając na uwadze znaczenie planowania, testowania i rozumienia tego, co budujesz.

Po osiągnięciu tej równowagi otrzymuje się to, co najlepsze z obu światów. Uzyskuje się oprogramowanie budowane szybciej i potencjalnie bardziej pomysłowo, ale także takie, któremu się ufa, które utrzymuje się i rozwija z pewnością siebie. Ostatecznie w ten sposób podnosimy poziom naszego rzemiosła w erze sztucznej inteligencji: nie wybierając intuicji zamiast inżynierii lub odwrotnie, ale opanowując całe spektrum między nimi.

W rozdziale 2. przeanalizowano sztukę tworzenia zapytań i współpracy z modelami AI. Będąc świadomym podstawowych koncepcji z tego rozdziału, jesteś gotowy do eksploracji praktycznej strony tej nowej ery programowania. Pozwoli to przygotować grunt pod praktyczne przykłady i dokładniejsze techniki tworzenia zapytań w kolejnych rozdziałach.

Opanuj sztukę intuicyjnego programowania z AI

Sztuczna inteligencja rewolucjonizuje tworzenie oprogramowania, wprowadzając nowy paradygmat programowania opartego na intencjach zamiast szczegółowej implementacji. Kodowanie intuicyjne (ang. *vibe coding*) to eksploracyjne podejście, w którym programiści współpracują z modelami AI, opisując w języku naturalnym pożądane rezultaty. Ta transformacja przesuwa rolę programistów z pisania kodu linijka po linijce do kierowania procesem generowania przez AI, co może zwiększyć produktywność nawet stukrotnie.

Książka stanowi kompleksowy przewodnik po projektowaniu wspomaganym sztuczną inteligencją, od podstaw inżynierii promptów po zaawansowane techniki współpracy z modelami AI. Autor szczegółowo omawia narzędzia takie jak GitHub Copilot, Cursor, Windsurf czy Claude, pokazując, jak je skutecznie integrować z codziennym przepływem pracy. Szczególny nacisk kładzie na „problem 70%" — zjawisko, w którym AI doskonale radzi sobie z rutynowymi zadaniami, ale ostatnie 30% wymaga ludzkiej ekspertyzy w zakresie architektury, bezpieczeństwa i optymalizacji.

Sztuczna inteligencja zmienia reguły inżynierii oprogramowania, a ta książka to podręcznik, którego potrzebuje każdy współczesny twórca.

— Sergio Pereira, przedsiębiorca i dyrektor do spraw technologii w startupie

W książce:

- Praktyczne techniki tworzenia skutecznych promptów
- Strategie przechodzenia od prototypów do kodu produkcyjnego
- Najlepsze praktyki bezpieczeństwa i audytu kodu
- Autonomiczne agenty do tworzenia kodu
- Etyczne aspekty AI w programowaniu
- Ewolucja ról programistów
- Zarządzanie zespołami hybrydowymi

Addy Osmani to główny lider zespołu inżynieryjnego w Google, kierujący pracami nad przeglądarką Chrome. Ma 25 lat doświadczenia w branży IT, jest autorem wielu książek o najlepszych praktykach inżynierii oprogramowania. Intensywnie testował narzędzia AI takie jak Cursor, Copilot, Claude Code, a jego publikacje o programowaniu wspomaganym AI wpłynęły na tysiące programistów.

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 978-83-289-3594-5
9 788328 935945
Cena: 89,00 zł