PRAKTYCZNY PRZEWODNIK

Podręcznik inżynierii agentów AI

Od podstaw do świadomej pracy z agentami

WYDANIE 1

Adam Mateusz Brożyński

Spis treści

1. Czym właściwie jest agent AI i czym różni się od zwykłego ChatGPT? 3

2. Anatomia tokena: dlaczego za każde pytanie płacisz za całą przeszłość? 3

3. 5 koszmarnych błędów początkujących (złe vs dobre podejście) 5

4. Orkiestracja modeli: jak zarządzać subagentami i oszczędzać budżet 7

5. Święta trójca plików i żywy indeks projektu:
   jak utrzymać stan po wyczyszczeniu sesji 9

6. Dyscyplina kodu: dlaczego zasady DRY, KISS i YAGNI
   są zbawieniem przy agentach AI 11

7. Narzędzia, MCP i bezpieczeństwo środowiska wykonawczego 14

8. Dzień z życia: pełny scenariusz pracy krok po kroku 14

9. Kompletne szablony plików do skopiowania (100% pełna treść) 15

Podziękuj na suppi: https://suppi.pl/adamstrad

1. Czym właściwie jest agent AI i czym różni się od zwykłego ChatGPT?

Większość osób zna sztuczną inteligencję z okna ChatGPT lub Claude na stronie internetowej: zadajesz pytanie, model generuje odpowiedź tekstową i na tym koniec. Model jest odcięty od świata – nie wie, jakie masz pliki na komputerze, nie może uruchomić Twojego kodu ani sprawdzić, czy to, co napisał, w ogóle się kompiluje.

Agent AI to zupełnie inna klasa oprogramowania. Agent to model językowy umieszczony w tzw. uprzęży wykonawczej (Agentic Harness). Posiada on dostęp do trzech kluczowych zmysłów:

Dostęp do plików

Agent sam otwiera, czyta, tworzy i edytuje pliki w Twoim projekcie.

Dostęp do terminala

Może odpalać komendy konsoli: kompilatory, testy jednostkowe, instalację pakietów, commity w Git.

Pętla działania (Loop)

Nie tylko odpowiada, ale działa w pętli: Planuje → Wykonuje akcję → Obserwuje wynik → Poprawia błędy.

> **Kluczowy wniosek dla początkujących:** Agentowi delegujesz zadania do wykonania. Możesz traktować go jak wirtualnego stażystę lub młodszego programistę. Niejasne polecenie może prowadzić do błędów i niepotrzebnych kosztów.

2. Anatomia tokena: dlaczego za każde pytanie płacisz za całą przeszłość?

Token to jednostka tekstu przetwarzanego przez model. Może obejmować całe słowo, jego fragment, znak interpunkcyjny lub inne znaki. Podział zależy od tokenizatora używanego przez dany model.

Największy błąd myślowy początkującego:

Wielu użytkowników myśli: „Moje pytanie ma tylko 10 słów, więc zapłacę za 10 słów”. **TO CAŁKOWITA NIEPRAWDA.**

Model otrzymuje kontekst przygotowany przez narzędzie, który może zawierać wcześniejsze wiadomości, kod i wyniki poleceń. Narzędzie może też skracać historię lub korzystać z pamięci podręcznej. Sposób rozliczenia zależy od usługi i planu.

Przykładowy wzrost kontekstu w czasie rozmowy:

Wiadomość 1 (Start): Twoje pytanie (100 słów)

Przykład: 150 tokenów

Wiadomość 10: Twoje pytanie + 9 poprzednich pytań + kod wygenerowany przez AI

Przykład: 25 000 tokenów

> Wiadomość 30 (Koniec dnia): Pytanie: „Zmień kolor” + 29 pytań + 15 plików kodu + logi z konsoli
>
> Przykład: 95 000 tokenów

To ilustracyjne wielkości kontekstu, a nie prognoza rachunku. Koszt zależy m.in. od modelu, stawek i pamięci podręcznej. Długi kontekst może też utrudniać modelowi korzystanie z istotnych informacji; zjawisko to nazywa się **context rot**.

Autonomiczny strażnik kontekstu: pozwól, aby agent SAM pilnował Twojego portfela

Nie każde narzędzie pokazuje na bieżąco liczbę zużytych tokenów. Dostępność komend skracania lub czyszczenia kontekstu zależy od narzędzia. **W nowoczesnej inżynierii agentowej to sam agent staje się strażnikiem Twojego budżetu (Proactive Context Sentinel).**

Dzięki wpisaniu odpowiedniej klauzuli w pliku `AGENTS.md` agent sam analizuje stan sesji i **proaktywnie sugeruje czyszczenie kontekstu** w 3 kluczowych momentach:

1. Po kamieniu milowym (Milestone)

Gdy agent ukończy zadanie z `tasks.md` i zapisze stan w `PROJECT_INDEX.md`, może zasugerować: „Krok ukończony. Przed kolejnym zadaniem rozważ skrócenie kontekstu lub nową sesję.”

2. Nasycenie sesji (>50k tokenów)

Gdy w historii nazbiera się 15-20 tur z długimi odczytami kodu, agent wyświetla ostrzeżenie: „ Uwaga: okno kontekstu jest mocno obciążone. Koszt kolejnych zapytań gwałtownie rośnie. Zalecam /compact.”

3. Pętla debugowania (Loop Blocker)

Jeśli agent dwukrotnie nie trafił z poprawką i testy dalej są czerwone, ZATRZYMUJE SIĘ i nakazuje reset sesji zamiast brnięcia w halucynacje w zanieczyszczonym oknie kontekstu.

3. 5 koszmarnych błędów początkujących (złe vs dobre podejście)

Oto 5 sytuacji, w których łatwo stracić czas i pieniądze. Zobacz, jak zastąpić złe praktyki bardziej uporządkowaną pracą:

Pojedynek 1: Zadawanie zadań

Wielkie życzenie vs Kontrola intencji

ZŁE PODEJŚCIE (Katastrofa)

„Napisz mi całą aplikację do zamawiania jedzenia z restauracji. Ma mieć koszyk, płatność kartą i panel dla kuriera.”

Konsekwencje: Agent próbuje zrobić wszystko naraz. Zgaduje bazę danych, wymyśla losowe nazwy pól, tworzy 20 niespójnych plików. Kod sypie się przy kompilacji, z konta ubywa 15 zł, a Ty zostajesz z chaosem.

DOBRE PODEJŚCIE (Precyzja)

„Nie pisz teraz żadnego kodu. Chcę stworzyć moduł koszyka zakupowego. Zaproponuj mi plik `spec.md` z wymaganiami i `tasks.md` z listą 4 kroków. Czekaj na moją weryfikację.”

Zysk: Widzisz plan przed zmianą kodu, możesz skorygować architekturę i kontrolować kolejne etapy pracy.

Pojedynek 2: Długość sesji czatu

Tasiemiec vs Reset z plikiem stanu

ZŁE PODEJŚCIE (Dziura w kieszeni)

Praca w jednym i tym samym oknie czatu przez 8 godzin. 50 zapytań, 40 wygenerowanych plików w historii.

Konsekwencje: Po 20. wiadomości płacisz za wysyłanie całego kodu projektu w kółko. Model zaczyna halucynować, „zapomina” o funkcjach napisanych godzinę temu i gubi się we własnych poprawkach.

DOBRE PODEJŚCIE (Czysta sesja)

Gdy agent ukończy zadanie, aktualizuje checklistę w `tasks.md` oraz `PROJECT_INDEX.md`. Wpisujesz komendę /compact lub otwierasz nowy wątek i piszesz: „Przeczytaj `PROJECT_INDEX.md` i realizuj krok 2”.

Zysk: Skracasz kontekst lub rozpoczynasz nową sesję, zachowując ważne informacje w indeksie. Dostępne komendy i koszty zależą od narzędzia.

Pojedynek 3: Wklejanie logów błędów

Ściana tekstu vs Wycięcie błędu

ZŁE PODEJŚCIE (Spalarnia tokenów)

Kopiujesz 500 linijek z konsoli terminala (z ostrzeżeniami webpacka, pobieraniem paczek npm) i wklejasz do czatu: „Coś nie działa, napraw to”.

Konsekwencje: Właśnie wysłałeś 40 000 tokenów śmieci. Agent gubi kluczowy błąd w gąszczu nieistotnych linii info i sugeruje losowe przeinstalowanie bibliotek.

DOBRE PODEJŚCIE (Chirurgiczne cięcie)

Wklejasz wyłącznie 4 linijki: nazwę błędu oraz plik i numer linii: „TypeError: Cannot read properties of undefined (reading 'price') w `src/cart.ts` linia 42”.

Zysk: Krótszy fragment ułatwia skupienie się na błędzie. Proponowaną poprawkę nadal trzeba zweryfikować.

Pojedynek 4: Edycja kodu

Przepisywanie pliku vs Edycja punktowa (Diff)

ZŁE PODEJŚCIE (Destrukcja kodu)

Agent przepisuje cały 500-liniowy plik od nowa tylko po to, żeby zmienić jeden warunek if w środku.

Konsekwencje: Płacisz za wygenerowanie 500 linii kodu na wyjściu (output tokens są 4x droższe niż input!). Po drodze model przypadkowo ucina importy na górze pliku i kasuje inne działające funkcje.

DOBRE PODEJŚCIE (Patch / Diff)

Narzędzia CLI (np. agy, Claude Code) używają narzędzia edit_file podmieniającego konkretny blok tekstu. Agent podmienia tylko 3 linie.

Zysk: Punktowa edycja ułatwia przegląd zmian. Sprawdź diff, aby upewnić się, co zostało zmienione.

Pojedynek 5: Wiara na słowo

Ufność vs Testy deterministyczne

ZŁE PODEJŚCIE (Iluzja sukcesu)

Agent pisze: „Gotowe! Funkcja działa idealnie, dodałem obsługę kuponów”. Ty wierzysz mu na słowo, a w przeglądarce biały ekran i błąd bazy.

Konsekwencje: Modele językowe to generatory tekstu – ich celem jest brzmieć przekonująco, nawet gdy kod zawiera błędy składniowe.

DOBRE PODEJŚCIE (Bramka testowa)

W pliku `AGENT.md` wymuszasz regułę: „Zakaz oznaczania zadania jako ukończone bez odpalenia testu npm test lub pytest. Raportuj tylko wynik testu”.

Zysk: Agent sam odpala test w konsoli, widzi błąd, sam go poprawia i oddaje Ci kod, który w 100% działa w rzeczywistości.

4. Orkiestracja modeli: jak zarządzać subagentami i oszczędzać budżet

Użytkownicy pytają: „Skąd agent wie, kiedy ma użyć Haiku, a kiedy Sonnet?”. Odpowiedź brzmi: **Domyślnie NIE WIE**! Jeżeli sam mu tego nie nakażesz w pliku konfiguracyjnym, będzie wszystko robił na domyślnym, drogim modelu.

Model planuje, model wykonuje: podział zadań

Jednym z możliwych sposobów pracy jest **rozdzielenie planowania architektury od implementacji**:

Krok 1: Planowanie

Wybierasz model odpowiedni do planowania i prosisz: „Przeanalizuj problem, stwórz `spec.md` oraz rozbij go na kroki w `tasks.md`. Nie pisz jeszcze kodu.” Następnie sprawdzasz plan i uzupełniasz brakujące wymagania.

Krok 2: Wykonawca (Sonnet / Haiku / Flash)

Przełączasz się na model roboczy. Agent realizuje plan, uruchamia testy i aktualizuje listę zadań. Koszt i jakość zależą od modeli, zadania i sposobu weryfikacji.

Hierarchia zespołu (metafora z życia wzięta):

1. Stażysta / Zwiadowca

Model do zwiadu

Niemal darmowy pracownik. Robi brudną robotę: przeszukuje pliki, czyta logi terminala, filtruje JSON. Posiada uprawnienia **TYLKO DO ODCZYTU**.

2. Główny programista

Model do implementacji

Główny model, który pisze funkcje, tworzy testy jednostkowe i refaktoryzuje kod na podstawie gotowego planu. Ma prawa do zapisu w plikach.

3. Główny architekt

Model do złożonego rozumowania

Najdroższy mózg. Konsultujesz go TYLKO na etapie tworzenia specyfikacji (`spec.md`), projektowania bazy danych lub gdy Sonnet utknie w pętli błędu.

Kompletny plik reguł dla orkiestratora (wklej to do swojego projektu):

Poniższy tekst to przykładowa polityka delegowania. Dostosuj ją do możliwości narzędzia oraz jego aktualnej dokumentacji; sama instrukcja nie gwarantuje dostępności subagentów ani wyboru modelu:

Szablon: dyrektywa orkiestracji wielomodelowej

```
# MULTI-AGENT ORCHESTRATION & MODEL ROUTING POLICY

Jesteś głównym Inżynierem-Orkiestratorem. Twoim zadaniem jest
zarządzanie projektem przy zachowaniu maksymalnej ekonomii
tokenów i czystości okna kontekstu.

## 1. ZASADY DELEGACJI SUBAGENTÓW (MODEL ROUTING)
Bezwzględnie przestrzegaj poniższego podziału zadań na modele:

1. ZADANIA EKSPLORACYJNE (Tani model: Haiku / Flash-Lite):
   - Kiedy musisz przeszukać pliki (grep, find, listowanie
katalogów) lub przeczytać pliki powyżej 100 linii:
   - ZAKAZ czytania ich bezpośrednio w głównym wątku.
   - Oddeleguj to zadanie do subagenta zwiadowczego (Haiku /
Flash-Lite) z uprawnieniami read-only.
   - Wymagaj od zwiadowcy zwrócenia wyłącznie ZWIĘZŁEGO
PODSUMOWANIA (maksymalnie 10 linii z nazwami plików i numerami
linii błędu).

2. ZADANIA PROGRAMISTYCZNE (Standardowy model: Sonnet / Pro):
   - Wszelką implementację kodu, pisanie testów i refaktoring
wykonujesz osobiście na modelu Sonnet.
   - Pracujesz wyłącznie na podstawie planu w tasks.md,
realizując jedno zadanie naraz.

3. KONSULTACJE ARCHITEKTONICZNE (model do złożonego rozumowania):
   - Rozważ konsultację z modelem do złożonego rozumowania, gdy:
     a) Projektujesz od zera architekturę modułu lub tworzysz
plik spec.md,
```

```
     b) Model Sonnet utknął w pętli naprawiania tego samego błędu
przez więcej niż 2 iteracje.
   - Do modelu architekta wysyłaj tylko wyizolowany, mały problem
logiczny bez całego kodu projektu.

## 2. HIGIENA OKNA KONTEKSTU I PAMIĘĆ
- Nigdy nie wklejaj do głównego czatu surowych logów terminala
powyżej 15 linii.
- Zwracaj wyłącznie punktowe diffy zmian zamiast przepisywania
całych plików.
- Po ukończeniu zadania zaktualizuj plik tasks.md oraz
PROJECT_INDEX.md i poproś o zatwierdzenie.
```

Przykładowy opis roli zwiadowcy

To schemat zakresu zadania, nie gotowy plik konfiguracyjny. Format i uprawnienia sprawdź w dokumentacji używanego narzędzia:

Opis roli

```
# Zwiadowca
Zadanie: przeszukuj kod i analizuj logi.
Zakres: tylko odczyt; zwracaj zwięzłe wyniki z nazwami plików i
numerami linii.
Uprawnienia tylko do odczytu skonfiguruj w narzędziu.
```

Przykładowe polecenie dla zwiadowcy

Przekaż je subagentowi, jeśli narzędzie obsługuje delegowanie:

Treść zadania

```
Przeszukaj build.log i znajdź linie z błędem Exception.
Zwróć maksymalnie trzy istotne linie. Nie edytuj plików.
```

5. Święta trójca plików i żywy indeks projektu: jak utrzymać stan po wyczyszczeniu sesji

Dlaczego inżynierowie IBM twierdzą, że „kod stał się tani, a plan jest prawdziwym artefaktem”?

Bo wygenerowanie 500 linii kodu zajmuje modelowi 8 sekund. Ale jeśli kod realizuje zły pomysł, to te 500 linii jest bezużytecznym śmieciem. Aby agent pracował jak w zegarku, w Twoim projekcie muszą istnieć **pliki sterujące markdown**:

1. `AGENT.md`

Konstytucja agenta

Zasady ogólne projektu: zakaz lania wody, zakaz komend rm -rf, wymóg testów przed raportem.

2. `spec.md`

Co i dlaczego budujemy

Specyfikacja konkretnej funkcji: pola, zachowanie systemu i przypadki brzegowe. Plan wymaga przeglądu człowieka.

3. `tasks.md`

Atomowa checklista

Lista małych zadań z polami [ ] i [x]. Realizujesz jedno po drugim bez gubienia wątku.

4. `PROJECT_INDEX.md`

Żywy indeks & pamięć

Aktualizowana na bieżąco mapa plików, baza wiedzy i wnioski z błędów. Magiczny klucz do nowej sesji!

Żywy indeks projektu (`PROJECT_INDEX.md`): jak pokonać amnezję agenta

Gdy kończysz dzień pracy lub czyścisz czat komendą `/compact`, model traci 100% pamięci operacyjnej. W nowej sesji początkujący zaczynają od nowa tłumaczyć projekt i wklejać pliki, przepalając kolejne dolary.

Złota praktyka: Nakazujesz orkiestratorowi, aby po KAŻDYM zadaniu dopisywał do pliku `PROJECT_INDEX.md` trzy rzeczy:

• **Mapę plików:** Co gdzie jest (np. `src/services/cart.ts` → silnik koszyka, `tests/cart.test.ts` → testy jednostkowe). Agent nie musi marnować tokenów na przeszukiwanie całego dysku.

• **Rejestr wniosków i post-mortems (Co NIE działało):** Np. „Próba użycia paczki X spowodowała błąd wersji w Node 20. Zastąpiono biblioteką Y. Nie próbuj ponownie instalować X!”. To ratuje agenta przed powtórzeniem tego samego błędu!

• **Aktualny stan i otwarte zadanie:** Dokładna informacja, co jest gotowe, a co leży na warsztacie.

Magiczny jednolinijkowy prompt po resecie sesji:

„Nowa sesja. Przeczytaj `PROJECT_INDEX.md` oraz `tasks.md` i podejmij realizację kolejnego otwartego zadania.”

W 300 tokenach agent natychmiast odzyskuje pełną pamięć, wie czego unikać i od razu przystępuje do pisania kodu!

Porównanie z prawdziwego życia: zły vs dobry plik specyfikacji

ZŁY PLIK SPEC (Ogólniki):

```
# Specyfikacja Koszyka
- Chcemy ładny koszyk zakupowy.
- Żeby klient mógł kliknąć kup teraz.
- Żeby dało się wpisać kod rabatowy.
```

Rezultat: Agent nie wie, co to znaczy „ładny”. Nie wie, jak działa rabat, czy rabaty się sumują, ani co zrobić przy ujemnej cenie. Tworzy kod z lukami bezpieczeństwa.

DOBRY PLIK SPEC (Ścisłe reguły):

```
# Specyfikacja: Koszyk Zakupowy
1. Pola: productId (string), ilosc (int, min 1).
2. Kupon rabatowy: pole kod (string).
   - Kod "RABAT10": odejmij 10% od sumy.
   - Błędny kod: zwróć błąd HTTP 400 z komunikatem.
3. Edge case: cena końcowa nie może spaść poniżej 0 zł.
4. Kryterium DoD: testy w cart.test.ts przechodzą w 100%.
```

Rezultat: Zero domysłów. Agent pisze kod dokładnie pod te punkty.

6. Dyscyplina kodu: dlaczego zasady DRY, KISS i YAGNI są zbawieniem przy agentach AI

Jeżeli nie narzucisz agentowi twardych zasad inżynierskich, zachowa się on jak nadgorliwy uczeń: na prośbę o zrobienie prostej herbaty zaprojektuje automatyczną fabrykę porcelany, założy plantację na Sri Lance i wdroży system subskrypcji imbryków w chmurze AWS.

Modele LLM były trenowane na gigantycznych repozytoriach korporacyjnych (Enterprise Java, rozbudowane frameworki C# i TypeScript). Mają one wrodzoną skłonność do tzw. **overengineeringu (przekombinowania)**. Generują setki linii kodu tam, gdzie wystarczyłoby pięć.

DRY

Don't Repeat Yourself

Nie powtarzaj się (Reuse First)

Grzech agenta: Zamiast sprawdzić, czy w projekcie istnieje funkcja walidacji e-maila lub kalkulator podatkowy, agent w nowym pliku pisze własną, nieco inną wersję. W projekcie powstają 4 różne funkcje robiące to samo.

> **Lekarstwo:** Nakaz przeszukania projektu (przez zwiadowcę Haiku) przed napisaniem jakiegokolwiek nowego helpera.

KISS

Keep It Simple, Stupid

Pisz prosto, bez przekombinowania

Grzech agenta: Prosisz o prosty filtr produktów po cenie. Agent generuje interfejs generyczny, klasę abstrakcyjną, wzorzec Strategy, Factory oraz 3 pliki konfiguracyjne (łącznie 200 linii nieczytelnego kodu).

> **Lekarstwo:** Reguła najprostszego mechanizmu: jeśli zadanie da się zrealizować 1 czystą funkcją, zakaz tworzenia klas i wzorców projektowych.

YAGNI

You Aren't Gonna Need It

Nie buduj na zapas

Grzech agenta: Prosisz o dodanie rabatu kwotowego. Agent dopisuje „na przyszłość” obsługę kryptowalut, punkty lojalnościowe, integrację ze Stripe i multi-tenancy.

> **Lekarstwo:** Żelazna dyscyplina: implementujesz TYLKO to, co znajduje się w pliku `spec.md`. Każda nieproszona linijka kodu to błąd architektoniczny.

Porównanie w praktyce: co daje narzucenie tych reguł

Pojedynek inżynierski: Obliczenie zniżki w koszyku

Overengineering vs Czysta prostota

BEZ ZASAD (Syndrom korporacyjny)

```
// 120 linii kodu w 4 plikach:
interface IDiscountStrategy { calculate(...): number; }
class PercentageDiscountStrategy implements IDiscountStrategy {
... }
class FixedDiscountStrategy implements IDiscountStrategy { ... }
class DiscountStrategyFactory { ... }
class DiscountCalculationPipelineManager { ... }
// Zduplikowany formatCurrency() w każdym pliku
// Obsługa 8 walut, których nikt nie potrzebuje
```

Koszt tokenów: ~4 500 tokenów wyjściowych (drogo i wolno).

Utrzymanie: Koszmar. Debugowanie 4 plików dla prostej zniżki 10%.

Ryzyko: Wysokie prawdopodobieństwo błędu w fabryce lub pipeline.

ZASADY DRY + KISS + YAGNI (Inżynieria)

```
// 12 linii czytelnego, deterministycznego kodu:
import { formatMoney } from '@/utils/currency'; // DRY: ponowne
użycie

export function applyDiscount(total: number, coupon?: Coupon):
number {
  if (!coupon) return total; // KISS: proste warunki brzegowe
  if (coupon.type === 'percent') return Math.max(0, total * (1 -
coupon.value / 100));
  if (coupon.type === 'fixed') return Math.max(0, total -
coupon.value);
  return total;
}
// YAGNI: tylko to co w specyfikacji, zero zbędnych klas
```

Koszt tokenów: Krótszy kod może wymagać mniej tokenów.

Utrzymanie: Prosta implementacja ułatwia przegląd.

Testowanie: Testy powinny obejmować wymagania i przypadki brzegowe.

> **Co zyskujesz w skali całego projektu?** Zasady DRY, KISS i YAGNI w pliku `AGENTS.md` mogą pomóc ograniczyć zbędny kod i ułatwić utrzymanie projektu. Wynik nadal wymaga przeglądu i testów.

7. Narzędzia, MCP i bezpieczeństwo środowiska wykonawczego

Czym jest MCP (Model Context Protocol) po ludzku?

Wyobraź sobie, że model językowy to mózg zamknięty w szklanej kuli. **Narzędzia i protokół MCP to ręce i oczy tego mózgu.**

Zamiast kopiować ręcznie dane z bazy danych lub dokumentacji i wklejać do czatu (marnując czas i tokeny), podłączasz do agenta tzw. serwer MCP. Agent sam odpytuje bazę przez uporządkowany protokół, pobiera dokładnie 1 rekord i natychmiast wraca do pracy.

Zasady bezpieczeństwa w terminalu (dlaczego AI nie wolno bezgranicznie ufać):

1. Zakaz komend niszczących

W pliku `AGENT.md` wpisz twardy zakaz używania poleceń: `rm -rf`, `git reset --hard`, `drop database` bez wyraźnej, ręcznej zgody w danej turze.

2. Ochrona plików sekretów (.env)

Nigdy nie pozwalaj agentowi odczytywać ani wypisywać w czacie plików `.env`, kluczy API ani tokenów chmurowych. Agent powinien operować wyłącznie na plikach `.env.example`.

3. Reguła Git Checkpoint (Twój pas bezpieczeństwa)

Przed pracą zapisz potrzebne zmiany w commicie i sprawdź stan repozytorium. Polecenie `git restore .` odrzuca niezapisane w indeksie zmiany w śledzonych plikach; nie cofa commitów ani nie usuwa nowych, nieśledzonych plików. Przed jego użyciem sprawdź zmiany, które zostaną utracone.

8. Dzień z życia: pełny scenariusz pracy krok po kroku

Zobacz, jak wygląda idealna, profesjonalna godzina pracy z agentem (np. w Claude Code, agy lub Cursorze) przy dodawaniu nowej funkcji:

• **Krok 1: Pas bezpieczeństwa** – Robisz `git checkout -b feature/nowy-koszyk` i commitujesz obecny stan.

• **Krok 2: Planowanie** – Dajesz polecenie: „Zapoznaj się z projektem. Stwórz `spec.md` oraz `tasks.md` dla modułu koszyka. Nie pisz kodu, czekaj na akceptację.”

• **Krok 3: Weryfikacja planu** – Otwierasz wygenerowany `spec.md`. Widzisz, że zapomniał o podatku VAT. Dopisujesz jedną linijkę w pliku.

• **Krok 4: Egzekucja krok po kroku na modelu Sonnet** – Piszesz: „Plan zatwierdzony. Zrealizuj krok 1 z `tasks.md`. Uruchom testy i pokaż diff.”

• **Krok 5: Konsolidacja pamięci i aktualizacja indeksu** – Po ukończeniu zadania agent dopisuje wnioski do `PROJECT_INDEX.md`. W razie potrzeby skracasz kontekst dostępną w narzędziu komendą i przechodzisz do kolejnego zadania.

9. Kompletne szablony plików do skopiowania (100% pełna treść)

Żadnych wielokropków i skrótów. Gotowe do wklejenia do Twojego projektu:

Zapisz jako `AGENTS.md` / `.agent/AGENT.md` / `.claude/CLAUDE.md` / .cursorrules w katalogu głównym projektu

```
# UNIVERSAL AGENT OPERATING CONSTITUTION (AGENTS.md)
# Uniwersalny standard inżynierski dla autonomicznych agentów
programistycznych
# Kompatybilny z: Claude Code, agy (Antigravity CLI), Cursor
(.cursorrules), OpenAI Codex
## 1. ROLA I DETERMINIZM INŻYNIERSKI
- Działasz jako starszy inżynier oprogramowania w trybie ściśle
deterministycznym.
- Kategoryczny zakaz „vibe codingu” (zgadywania intencji, pisania
kodu na oko, samowolnego dodawania niepotrzebnych bibliotek i
modyfikowania niepowiązanych plików).
- Każda modyfikacja kodu musi wynikać wprost z pliku `spec.md`
oraz aktywnego zadania z `tasks.md`.
- Zasada atomowości: Realizujesz dokładnie JEDNO zadanie z listy
naraz. Po jego wykonaniu i przetestowaniu zatrzymujesz się i
prosisz o akceptację. Kategoryczny zakaz przechodzenia do zadania
N+1 bez zgody użytkownika.

## 2. PAMIĘĆ PROJEKTU I ŻYWY INDEKS (PROJECT_INDEX.md)
- Początek każdej sesji: Zanim podejmiesz jakąkolwiek akcję,
przeczytaj `PROJECT_INDEX.md` oraz `tasks.md`. W ten sposób
natychmiast odzyskujesz pełny kontekst projektu bez marnowania
tokenów na przeszukiwanie dysku.
- Koniec zadania: Po pomyślnym zrealizowaniu każdego kroku masz
OBOWIĄZEK zaktualizować `PROJECT_INDEX.md`:
  1. Mapę plików (jeśli utworzyłeś lub zmieniłeś pliki – dopisz
ich ścieżkę i rolę w architekturze).
  2. Rejestr decyzji architektonicznych (ADR) – jednozdaniowe
uzasadnienie wybranego rozwiązania.
  3. Rejestr post-mortems (co NIE zadziałało i dlaczego – zanotuj
```

```
biblioteki, konflikty wersji lub błędy, których kolejny agent w
nowej sesji ma unikać).
  4. Aktualny stan projektu i precyzyjne wskazanie najbliższego
otwartego zadania.
- Zasada Progressive Disclosure: Główny plik konfiguracyjny ma
być zwięzły (<150 linii). Szczegółowe zasady dla podmodułów
sprawdzaj w ich dedykowanych folderach.

## 3. ORKIESTRACJA WIELOMODELOWA (MODEL ROUTING & SUBAGENCI)
Bezwzględnie przestrzegaj hierarchii modeli i uprawnień:
1. ZADANIA EKSPLORACYJNE (Tani model: Haiku / Flash-Lite /
GPT-4o-mini) — TRYB READ-ONLY:
   - Wszelkie przeszukiwanie bazy kodu (`grep`, `glob`, czytanie
katalogów) oraz analizę logów powyżej 50 linii oddeleguj do
zwiadowcy w trybie tylko do odczytu.
   - Zakaz wczytywania wielkich plików bezpośrednio do głównego
okna kontekstu.
   - Wymagaj od zwiadowcy wyłącznie syntetycznego ekstraktu (do
10 linii z nazwami plików i numerami linii błędu).
2. ZADANIA PROGRAMISTYCZNE (Standardowy model: Sonnet / Pro /
GPT-4o):
   - Wszelką implementację logiki biznesowej, pisanie testów
jednostkowych i refaktoryzację wykonujesz na modelu standardowym.
   - Pracujesz wyłącznie w granicach aktywnego kroku z `tasks.md`.
3. KONSULTACJE ARCHITEKTONICZNE (model do złożonego rozumowania):
   - Uruchamiany WYŁĄCZNIE na etapie tworzenia specyfikacji
(`spec.md`), dekompozycji planu (`tasks.md`), projektowania
schematów baz danych lub gdy model roboczy dwukrotnie utknie w
pętli naprawy tego samego błędu.
   - Do architekta wysyłaj tylko wyizolowany, minimalny problem
logiczny bez zbędnego kodu całego repozytorium.

## 4. BEZPIECZEŃSTWO, SEKRETY I GRANICE SANDBOXU
- BEZWZGLĘDNY ZAKAZ dotykania plików zawierających hasła, klucze
i sekrety: `.env`, `.env.*`, `*.pem`, `*.key`, `id_rsa`,
`credentials.json`, `token.json`.
- Nigdy nie wypisuj wartości zmiennych środowiskowych w
odpowiedziach. Odwołuj się wyłącznie do szablonu `.env.example`.
- BEZWZGLĘDNY ZAKAZ wykonywania komend niszczących:
  - `rm -rf`, `rmdir /s`, `git reset --hard`, `git clean -fdx`,
`drop database`, `drop table`, `mkfs`, `format`.
  - Wykonanie jakiejkolwiek komendy usuwającej pliki wymaga
jawnej, pisemnej zgody użytkownika w bieżącej turze.
- Reguła Git Checkpoint: Przed modyfikacją kodu upewnij się, że
pracujesz na dedykowanej gałęzi lub stan repozytorium jest czysty
(`git status`).

## 5. HIGIENA OKNA KONTEKSTU I EKONOMIA TOKENÓW
- Zwięzłość techniczna: Odpowiadaj maksymalnie krótko. Podawaj
wyłącznie fakty, diffy kodu i wyniki komend.
- Zakaz lania wody: Zero wstępów typu „Oczywiście, chętnie
pomogę!”, zero grzecznościowych podsumowań i parafrazowania pytań
użytkownika.
- Edycja punktowa (Diff): Kategoryczny zakaz przepisywania całych
plików, jeśli zmieniasz kilka linii. Używaj narzędzi edycji
blokowej (patch/diff).
- Logi terminala: Zakaz wklejania do głównego kontekstu logów
```

```
powyżej 15 linii. W razie błędu kompilacji wyodrębnij tylko 3-5
kluczowych linii stack trace (nazwa wyjątku, plik, linia).

## 6. DETERMINISTYCZNA BRAMKA WERYFIKACYJNA (DEFINITION OF DONE)
- Agent NIGDY nie ocenia poprawności kodu „na oko” i nie
raportuje sukcesu bez weryfikacji maszynowej.
- Zadanie z `tasks.md` uznaje się za ukończone (`[x]`) WYŁĄCZNIE
wtedy, gdy:
  1. Został uruchomiony odpowiedni test jednostkowy lub
integracyjny (`npm test`, `pytest`, `cargo test`) i zakończył się
kodem wyjścia 0 (wszystkie testy zielone).
  2. Linter i kompilator typów (`npm run lint`, `tsc --noEmit`,
`flake8`) nie zgłaszają żadnych nowych błędów ani ostrzeżeń.
- W raporcie końcowym przedstaw wyłącznie:
  - Status testów (np. `Tests: 14 passed, 0 failed, exit code 0`),
  - Krótki diff zmienionych plików,
  - Zaktualizowany wpis w `tasks.md` oraz `PROJECT_INDEX.md`.

## 7. PROTOKÓŁ ZAMKNIĘCIA SESJI (PRZED /compact LUB WYJŚCIEM)
- Przed zresetowaniem czatu lub zakończeniem dnia pracy:
  1. Oznacz ukończone kroki w `tasks.md`.
  2. Uzupełnij `PROJECT_INDEX.md` o stan bieżący, napotkane
wnioski oraz wskaż dokładnie, od czego ma zacząć kolejny agent w
nowej sesji.
  3. Zrób commit w Git podsumowujący wykonany krok: `git commit
-m "feat/fix: opis kroku"`.

## 8. DYSCYPLINA ARCHITEKTURY KODU: DRY, KISS ORAZ YAGNI
- **KISS (Keep It Simple, Stupid):** Wybieraj zawsze najprostsze
działające rozwiązanie. Kategoryczny zakaz wprowadzania
niepotrzebnych wielowarstwowych abstrakcji (wzorców Factory,
Strategy, Adapter) dla trywialnych operacji. Pisz kod prosty,
czytelny i liniowy.
- **YAGNI (You Aren't Gonna Need It):** Implementuj WYŁĄCZNIE to,
co zostało wprost zdefiniowane w `spec.md`. Kategoryczny zakaz
dopisywania funkcji „na zapas” (obsługa nieużywanych walut,
nieproszone parametry konfiguracyjne, przedwczesne optymalizacje
wydajnościowe).
- **DRY (Don't Repeat Yourself & Reuse First):** Zanim utworzysz
nową funkcję pomocniczą (np. walidator danych, formatowanie kwot,
parser), masz OBOWIĄZEK sprawdzić istniejący kod projektu. Jeśli
rozwiązanie istnieje – zaimportuj je i użyj ponownie, zamiast
tworzyć kolejną wersję w nowym pliku.

## 9. AUTONOMICZNY MONITORING KONTEKSTU (PROACTIVE CONTEXT
SENTINEL)
Agent ma obowiązek aktywnie dbać o budżet tokenów użytkownika i
zapobiegać zjawisku Context Rot.
Agent MA OBOWIĄZEK proaktywnie sugerować wykonanie komendy
`/compact` lub `/clear` w następujących sytuacjach:
1. **Po ukończeniu zadania (Milestone):** Gdy krok z `tasks.md`
zostanie zrealizowany, przetestowany i odnotowany w
`PROJECT_INDEX.md`, agent w podsumowaniu wypisuje:
   `[WSKAZÓWKA] Wskazówka: Krok zakończony i utrwalony w
indeksie. Przed kolejnym zadaniem rozważ skrócenie kontekstu lub
otwarcie nowej sesji.`
2. **Nasycenie kontekstu (>15-20 tur lub >60k tokenów):** Gdy
```

```
historia czatu zawiera dużo długich odczytów plików i logów,
agent ostrzega o rosnącym koszcie kolejnych wiadomości i zaleca
oczyszczenie kontekstu.
3. **Pętla błędu (2 nieudane próby naprawy):** Jeśli testy
jednostkowe nie przechodzą po 2 kolejnych poprawkach, agent
ZATRZYMUJE SIĘ i nakazuje:
   `[STOP] Pętla naprawcza: Zatrzymuję pętlę. Zalecam zapisanie
problemu w spec.md i ponowną analizę architektury.`
```

Zapisz jako `spec.md`

```
# FEATURE SPEC: Moduł Koszyka Zakupowego z Kuponami Rabatowymi
## 1. CEL BIZNESOWY
Umożliwienie użytkownikowi dodawania produktów do koszyka,
naliczania podatku VAT (23%) oraz wprowadzania kodów promocyjnych
obniżających wartość zamówienia.

## 2. WYMAGANIA FUNKCJONALNE
- [ ] Funkcja 1: Dodawanie produktu do koszyka:
  - Wejście: `productId` (string, UUIDv4), `quantity` (int, min:
1, max: 99).
  - Walidacja: Jeśli `quantity <= 0`, wyrzuć błąd
`InvalidQuantityError` (HTTP 400).
- [ ] Funkcja 2: Aplikowanie kodu rabatowego:
  - Kod "PROMO10": obniża sumę produktów o 10%.
  - Kod "MINUS50": obniża sumę zamówienia o 50 zł (tylko dla
zamówień od 200 zł).
  - Wprowadzenie nieistniejącego kodu: zwróć kod błędu
`CouponNotFoundError` (HTTP 404).
- [ ] Funkcja 3: Kalkulacja sumy końcowej:
  - Wzór: `Suma_Końcowa = (Suma_Produktów - Rabat) * 1.23 (VAT)`.

## 3. PRZYPADKI BRZEGOWE (EDGE CASES)
- Co jeśli rabat przekracza wartość produktów? Cena po rabacie
nie może spaść poniżej 0 zł (wartość minimalna: 0.00 zł).
- Co jeśli produkt został usunięty z bazy w trakcie trwania sesji
koszyka? Zwróć kod błędu `ProductUnavailableError` i usuń element
z koszyka.
- Zaokrąglenia walutowe: Wszystkie kwoty muszą być zaokrąglane do
dokładnie 2 miejsc po przecinku w górę (metoda HALF_UP).

## 4. KRYTERIA UKOŃCZENIA (DEFINITION OF DONE)
- Napisane testy jednostkowe w pliku `cart.test.ts` pokrywające
100% przypadków brzegowych.
- Brak jakichkolwiek ostrzeżeń lintera (`npm run lint`).
- Kod nie zawiera hardkodowanych wartości stawek podatku (stawka
pobierana z konfiguracji).
```

Zapisz jako `tasks.md` w projekcie

```
# IMPLEMENTATION TASKS: Koszyk Zakupowy
```

```
Status Ogólny: W toku
Bieżące zadanie: Krok 2 (Testy jednostkowe TDD)

## Faza 1: Modele Danych i Walidacja
- [x] Krok 1: Zdefiniowanie interfejsów TypeScript: `CartItem`,
`Coupon`, `CartState` w pliku `types/cart.ts`.
- [ ] Krok 2: Napisanie zestawu testów jednostkowych (TDD) w
`tests/cart.test.ts` sprawdzających dodawanie produktów, limity
ilościowe oraz nieistniejące kupony.
- [ ] Krok 3: Implementacja klasy `CartManager` w
`services/cart.ts` obsługującej dodawanie i usuwanie pozycji.

## Faza 2: Logika Rabatowa i Podatki
- [ ] Krok 4: Implementacja silnika kuponów `CouponEngine` z
obsługą kodów "PROMO10" i "MINUS50" oraz zabezpieczeniem przed
sumą ujemną.
- [ ] Krok 5: Dodanie kalkulatora podatkowego z zaokrąglaniem
kwot do 2 miejsc po przecinku.

## Faza 3: Integracja i Weryfikacja
- [ ] Krok 6: Podłączenie endpointu kontrolera API `POST
/api/cart/add`.
- [ ] Krok 7: Uruchomienie pełnego audytu lintera i testów
regresyjnych (`npm test`).
```

Zapisz jako `PROJECT_INDEX.md` (Żywy indeks i pamięć)

```
# PROJECT INDEX & REPOSITORY MEMORY

Ostatnia aktualizacja: 2026-11-21 10:45
Bieżąca gałąź: feature/shopping-cart
Ostatni commit: 4f9a12c ("types: dodano interfejsy koszyka")

## 1. MAPA PLIKÓW I ARCHITEKTURA (CO GDZIE JEST)
- `src/types/cart.ts` — Główne definicje typów (CartItem, Coupon,
CartState).
- `src/services/cart.ts` — Logika biznesowa koszyka (CartManager).
- `src/services/coupons.ts` — Silnik rabatowy (CouponEngine).
- `src/controllers/cart.ts` — Kontroler REST API dla Express /
Fastify.
- `tests/cart.test.ts` — Zestaw testów jednostkowych (Jest /
Vitest).
- `config/tax.ts` — Stałe stawek podatkowych pobierane ze
środowiska.

## 2. REJESTR DECYZJI ARCHITEKTONICZNYCH (ADR)
- [ADR-01] Zrezygnowano z biblioteki float-math na rzecz
BigNumber.js dla operacji walutowych, aby uniknąć błędów
zaokrągleń IEEE 754.
- [ADR-02] Wszystkie kupony rabatowe są walidowane lokalnie przed
zapytaniem do bazy w celu odciążenia I/O.

## 3. POST-MORTEMS I WNIOSKI (CO NIE DZIAŁAŁO — NIE POWTARZAJ!)
- UWAGA: Próba użycia biblioteki `currency-formatter` powoduje
```

```
błąd kompilacji pod Node 20. NIE INSTALUJ JEJ. Używamy natywnego
`Intl.NumberFormat`.
- UWAGA: Endpoint `/api/cart` wymaga nagłówka `X-Session-ID`, w
przeciwnym razie middleware rzuca błąd 401. Pamiętaj o tym w
testach integracyjnych.

## 4. AKTUALNY STAN PRAC I NAJBLIŻSZY KROK
- Ukończono: Krok 1 z tasks.md (Interfejsy TypeScript).
- NASTĘPNE ZADANIE DO REALIZACJI: Krok 2 z tasks.md — napisanie
testów jednostkowych w `tests/cart.test.ts`.
```

Prompt 1: Planowanie

Działasz jako Główny Architekt Oprogramowania. Moim celem jest dodanie modułu koszyka zakupowego. Na podstawie założeń stwórz plik `spec.md` z wymaganiami i przypadkami brzegowymi oraz `tasks.md` z atomową listą kroków. Kategoryczny zakaz pisania kodu źródłowego na tym etapie. Czekaj na moją weryfikację i akceptację planu.

Prompt 2: Wykonanie kroku (Wklejasz modelowi Sonnet)

Plan został zatwierdzony. Zrealizuj wyłącznie Krok 1 z pliku `tasks.md`. Po zakończeniu implementacji uruchom testy jednostkowe. Zwróć tylko zwięzły wynik testów oraz punktowy diff zmian. Nie przechodź samowolnie do Kroku 2.

Prompt 3: Reset po wyczyszczeniu sesji (Nowy dzień / /compact)

Otwieramy nową sesję roboczą. Przeczytaj reguły w `.agent/AGENT.md` oraz pamięć w `PROJECT_INDEX.md`. Zapoznaj się z bieżącym stanem w pliku `tasks.md` i przystąp do realizacji pierwszego niezrobionego zadania o statusie [ ].

Prompt 4: Tura konsolidacji przed zamknięciem pracy

Kończymy bieżącą sesję roboczą. Zaktualizuj checkboxy w `tasks.md` oraz dopisz do `PROJECT_INDEX.md`: mapę zmodyfikowanych dziś plików, napotkane problemy i wnioski oraz wskaż dokładnie od czego ma zacząć kolejny agent w nowej sesji.