Praktyczny przewodnik wdrożenia Inżynierii Agentowej (agentic engineering) w repozytorium GitHub

1. Wprowadzenie: Od Vibe Coding do Inżynierii Agentowej

Wraz z gwałtowną ewolucją autonomicznych agentów kodujących następuje fundamentalna zmiana paradygmatu w wytwarzaniu oprogramowania: przejście od amatorskiego „vibe codingu" do profesjonalnej „inżynierii agentowej" (*Agentic Engineering*, określanej w fazie przejściowej jako *vibe engineering*). Vibe coding, zdefiniowany przez Andreja Karpathy'ego w lutym 2025r., oznacza beztroskie, szybkie i powierzchowne pisanie kodu wyłącznie na podstawie prostej interakcji z promptami. W paradygmacie tym programista całkowicie oddaje inicjatywę modelowi AI, rezygnując z analizy architektury oraz weryfikacji wygenerowanych mechanizmów i akceptując wynik, o ile kod sprawia wrażenie, że „jakoś działa" (*it mostly works*).

Jak słusznie zauważył Simon Willison (który w październiku 2025 r. zaproponował termin *vibe engineering*, zastąpiony do lutego 2026 r. przez *agentic engineering*), stosowanie takiego podejścia w odniesieniu do oprogramowania produkcyjnego przeznaczonego dla innych ludzi jest po prostu skrajnie nieodpowiedzialne (*"grossly irresponsible"*). Dzieje się tak, ponieważ słowo „vibe" – jak podkreślił Addy Osmani 4 lutego 2026 r. – niesie ze sobą zbyt duży bagaż beztroski (*"the word 'vibe' carries too much baggage. It signals casualness"*).

Inżynieria agentowa stanowi drugą stronę tego spektrum: to rygorystyczna, dojrzała dyscyplina przeznaczona dla doświadczonych inżynierów i architektów, którzy przyspieszają pracę przy użyciu modeli LLM oraz orkiestrowanych agentów (Claude Code, Codex CLI, Cursor, Windsurf, GitHub Copilot). Inżynier agentowy zachowuje pełną, jawną i osobistą odpowiedzialność za jakość, bezpieczeństwo oraz długoterminową łatwośćutrzymania systemu (*maintainability*). Kluczem do sukcesu w tym podejściu jest bezwzględna eliminacja „amnezji sesyjnej" (*session amnesia*) poprzez wdrożenie trwałych struktur pamięci oraz narzucenie sztywnych, automatycznych barier ochronnych (*guardrails*) bezpośrednio w repozytorium GitHub.

Zestawienie porównawcze: Vibe Coding a Inżynieria Agentowa (agentic engineering)

Kryterium	Vibe Coding	Inżynieria Agentowa (Agentic Engineering)
Definicja / Czym jest	Luźna praktyka: wysłanie promptu, pobieżny rzut oka na wynik i bezrefleksyjna akceptacja.	Rygorystyczna dyscyplina inżynieryjna dla doświadczonych profesjonalistów odpowiadających za jakość wdrożenia.
Autorstwo kodu	Generator / model AI pracujący bez głębszej kontroli ze strony człowieka.	Modele LLM i autonomiczni agenci kodujący pracujący iteracyjnie w ściśle wyznaczonych pętlach.
Metoda weryfikacji	Brak analizy diffa; człowiek jedynie obserwuje powierzchowne zachowanie aplikacji.	Odpowiedzialny inżynier wsparty automatycznymi testami, planowaniem architektonicznym, rygorystycznym *code review* oraz ręcznym QA.
Kryterium ukończenia	Poziom „Jakoś działa" (*It mostly works*).	Pełna jakość produkcyjna, gwarancja stabilności numerycznej i bezpieczeństwa architektonicznego.
Pochodzenie nazwy	Andrej Karpathy (wpis na platformie X, 2 lutego 2025 r.).	Simon Willison (7 paź 2025 r. jako *vibe engineering*); rozwinięte przez Sergeta Tselovalnikova (31 mar 2025 r.) i Brendana O'Leary'ego (29 kwi 2025 r.). Wpis Karpathy'ego i Osmaniego z 4 lutego 2026 r. ugruntował nazwę *Agentic Engineering* (Karpathy: *"personally my current favorite"*), co Willison oficjalnie potwierdził 23 lutego 2026 r.
Główne ryzyka / Ograniczenia	Skrajna nieodpowiedzialność w projektach produkcyjnych (*grossly irresponsible*); gwałtownie narastający dług technologiczny.	Wyczerpanie poznawcze inżyniera wynikające z orkiestracji równoległych agentów, wąskie gardła na etapie *code review* oraz próby „oszukiwania" bramek weryfikacyjnych przez agenty.

Główne tryby awarii agentów AI i ich mitygowanie przez inżynierię ustrukturyzowaną

- **Twierdzenie o sukcesie bez faktycznej weryfikacji (tzw. „oszukiwanie" / *hallucinated success*)**: Agent potrafi zignorować ukryte błędy i zakomunikować wykonanie zadania bez uruchomienia suity weryfikacyjnej. Ustrukturyzowana inżynieria eliminuje ten problem poprzez automatyczne barierki ochronne (*Verification-First*) i skrypty weryfikacyjne w potokach CI/CD, bez pomyślnego wykonania których zgłoszenie nie może zostać przyjęte.
- **Amnezja sesyjna i gubienie kontekstu architektonicznego (*Session Amnesia*)**: Wraz z upływem sesji i wyczerpywaniem się okna kontekstowego agent zapomina o ustaleniach projektowych. Inżynieria agentowa rozwiązuje to poprzez wdrożenie trwałego systemu pamięci (*Memory Bank*), odczytywanego na początku każdej sesji.

- **Halucynowanie nieistniejących bibliotek i wycofanych API**: Modele wywołują nieistniejące pakiety, parametry lub przestarzałe metody. Mitygacja polega na udostępnianiu agentom plików kontekstu technicznego (`techContext.md`) z dokładnym spisem bibliotek oraz wymuszeniu analizy statycznej linterami (`ruff`, `mypy`).
- **Brak planowania i narastanie długu technologicznego**: Natychmiastowe przeskakiwanie do edycji plików generuje chaotyczny, nieprzemyślany kod. Wdrażając obowiązkowy, dwufazowy protokół *Plan-and-Act*, wymusza się najpierw analizę w trybie do odczytu i zatwierdzenie specyfikacji przez człowieka.
- **Ignorowanie przypadków brzegowych (*Edge Cases*) i rygoru numerycznego**: Agenci skupiają się na ścieżce optymistycznej (*happy path*), ignorując stany skrajne, wartości NaN/Inf oraz niestabilność numeryczną. Mitygacja polega na wprowadzeniu dedykowanych testów brzegowych w `pytest` z aktywnymi asercjami wykrywającymi anomalie numeryczne.
- **Generowanie nieoptymalnego kodu i niereprodukowalność**: Pisanie spowalniających pętli w Pythonie zamiast operacji zwektoryzowanych oraz pomijanie ziaren losowości. Inżynieria ustrukturyzowana wprowadza wzorce architektoniczne *Vectorization-First* weryfikowane drzewem AST oraz wymóg ustawiania `numpy.random.seed` dla 100% reprodukowalności symulacji.

Przełożenie powyższych zasad strategicznych na konkretną strukturę plików w repozytorium stanowi fundament nadzoru i bezpiecznego zarządzania pracą agentów AI.

2. Architektura repozytorium i uniwersalne punkty wejściowe (pointer-pattern)

Kluczowym wyzwaniem architektonicznym w pracy z agentami AI jest precyzyjne zarządzanie budżetem kontekstowym. Wczytywanie całościowych specyfikacji i rozbudowanych plików instrukcji przy każdym wywołaniu prowadzi do zjawiska *lost in the middle* (w którym agent ignoruje wytyczne umieszczone w środku tekstu) oraz szybkiego wyczerpania tokenów. Aby temu zapobiec przy jednoczesnym zapewnieniu pełnej przenośności pomiędzy różnymi edytorami i narzędziami (Cursor, Claude Code, Windsurf, GitHub Copilot, Codex CLI), stosuje się wzorzec wskaźnikowy (*Pointer-Pattern*). Polega on na utrzymywaniu zwięzłych, lekkich plików startowych dedykowanych dla poszczególnych edytorów, które wskazują precyzyjne ścieżki do wspólnego źródła prawdy znajdującego się w katalogach `.agents/` oraz `memory-bank/`.

Struktura katalogów w repozytorium

```
.
├── .agents/                    # Uniwersalne reguły i standardy wykonawcze
│   ├── general_rules.md        # Główny zestaw uniwersalnych dyrektyw
│   └── code_style.md           # Konwencje kodowania i typowania
├── memory-bank/                # Trwała pamięć projektu (Memory Bank)
│   ├── projectbrief.md         # Wymagania i cel projektu
│   ├── techContext.md          # Stos technologiczny i komendy CLI
│   ├── systemPatterns.md       # Wzorce architektoniczne i struktury
│   ├── activeContext.md        # Bieżący obszar prac i ostatnie zmiany
│   └── progress.md             # Postęp, kamienie milowe i backlog
├── .cursor/
│   └── rules/                  # Reguły wejściowe dla Cursor IDE
│       └── main.mdc
├── CLAUDE.md                   # Punkt wejścia dla Claude Code
└── .github/
    ├── copilot-instructions.md # Instrukcja dla GitHub Copilot
    ├── ISSUE_TEMPLATE/
    │   └── agent_task.md       # Szablon zgłoszenia dla agenta
    └── workflows/
        └── agent-verification.yml # Potok weryfikacyjny CI/CD
```

Limity budżetu kontekstowego dla plików dokumentacji

Plik / Typ instrukcji	Zalecana długość (znaki)	Szacowana liczba tokenów	Optymalna liczba linii
AGENTS.md / .agents/general_rules.md	1 500 – 3 500 znaków	~300 – 800 tokenów	50 – 120 linii
README.md	3 000 – 8 000 znaków	~600 – 1 800 tokenów	100 – 250 linii
ARCHITECTURE.md / systemPatterns.md	4 000 – 10 000 znaków	~800 – 2 500 tokenów	150 – 300 linii

Dynamiczne serwowanie reguł i integracja z serwerem MCP (Model Context Protocol)

W nowoczesnych środowiskach Inżynierii Agentowej wzorzec wskaźnikowy powiązany z plikami statycznymi jest uaktualniany o dynamiczną integrację przez serwery **MCP (Model Context Protocol)**. Serwer MCP działa jako uniwersalny interfejs middleware, łączący lokalne narzędzia deweloperskie (Cursor, Claude Code, Copilot) z centralnym repozytorium wiedzy organizacji.

Dzięki integracji MCP, zamiast polegać wyłącznie na lokalnych plikach reguł, lokalny agent za pośrednictwem Pointer-Patternu wysyła zapytanie do serwera MCP (np. `10xRules` lub wewnętrzny serwer firmowy), pobierając w czasie rzeczywistym:

1. Aktualne, centralnie zarządzane schematy narzędzi i bibliotek promptów.
2. Zaktualizowane polityki bezpieczeństwa i standardy kodowania obowiązujące w organizacji.
3. Dynamiczny kontekst z zewnętrznych systemów (np. aktualny schemat bazy danych czy statusy zgłoszeń z systemu JIRA/GitHub Issues).

Integracja MCP eliminuje ryzyko stosowania nieaktualnych reguł lokalnych w rozproszonych zespołach inżynierskich.

Gotowe szablony plików wejściowych

1. Plik uniwersalny: .agents/general_rules.md

```
# UNIVERSAL AGENT CORE DIRECTIVES

You are an autonomous AI Coding Agent. You MUST strictly adhere to the following core rules:

## 1. EXECUTION PROTOCOL (PLAN-AND-ACT)
- You MUST operate in a two-phase cycle: PLAN and ACT.
- In Stage 1 (PLAN), you are FORBIDDEN from editing, creating, or deleting code files. You MUST inspect the codebase and write a plan.
- ALWAYS wait for explicit human approval ("APPROVED") before moving to Stage 3 (ACT).

## 2. VERIFICATION-FIRST & QUALITY GATES
- Before presenting completed work, you MUST run all verification commands defined in `memory-bank/techContext.md`.
- You MUST ensure 100% passing tests in `pytest`, zero errors in `ruff check`, zero `mypy` type errors, and clean LaTeX compilation (`latexmk -pdf`).
- NEVER claim a task is successful without executing verification scripts.

## 3. MEMORY BANK SYNCHRONIZATION
- BEFORE planning, you MUST read all files in `memory-bank/`.
- AFTER execution, you MUST update `memory-bank/activeContext.md` and `memory-bank/progress.md`.
```

2. Wskaźnik dla Claude Code: CLAUDE.md

```
# Claude Code Configuration Entry Point

Before taking any action or generating code, you MUST read the universal project rules and memory context:

1. ALWAYS read `.agents/general_rules.md` for core directives and execution bounds.
2. ALWAYS read all Markdown files in `memory-bank/` (`projectbrief.md`, `techContext.md`, `systemPatterns.md`, `activeContext.md`, `progress.md`) to establish full repository context.
3. Strictly follow the Plan-and-Act workflow and run `bash scripts/verify_agent_work.sh` upon completion.
```

3. Wskaźnik dla Cursor IDE: .cursor/rules/main.mdc

```
---
description: Main Pointer Rule for Universal Agent Governance
globs: "**/*"
---
# CURSOR AGENT DIRECTIVE

- ALWAYS read and enforce rules from `.agents/general_rules.md`.
- Read persistent context from `memory-bank/` prior to planning.
- FORBIDDEN from modifying code files during the PLAN phase.
- Execute local verification scripts (`pytest`, `ruff`, `mypy`) before finalizing output.
```

Lekkie punkty wejściowe sprawnie przekazują uprawnienia wykonawcze do trwałego systemu pamięci projektu, co eliminuje utratę kontekstu między niezależnymi sesjami.

3. Wdrożenie Trwałej Pamięci Projektu: System Memory Bank

Wdrożenie wzorca *Memory Bank* pozwala całkowicie wyeliminować zjawisko amnezji sesyjnej u agentów AI. Tworzy ono trwałe repozytorium wiedzy o kodzie podzielone na wyspecjalizowane moduły, co pozwala chronić budżet kontekstowy – agent wczytuje dynamiczne pliki pamięci tylko wtedy, gdy wymaga tego dane zadanie.

Ewaluacja plików struktury Memory Bank

1. **`memory-bank/projectbrief.md`**: Definiuje biznesowy i techniczny cel projektu, zakreśla granice systemu oraz określa główne kryteria jakościowe (*Quality Gates*). Charakter: **statyczny** (zmienia się rzadko, przy redefinicji założeń). Updates: na początku projektu lub przy zmianie wymagań.
2. **`memory-bank/techContext.md`**: Zawiera opis stosu technologicznego, wersji języka, bibliotek, struktury katalogów oraz dokładnych komend CLI weryfikacyjnych. Charakter: **statyczny / częściowo dynamiczny**. Updates: przy zmianie zależności lub dodaniu narzędzi.
3. **`memory-bank/systemPatterns.md`**: Określa wzorce architektoniczne, konwencje kodowania, zasady rygoru numerycznego oraz standardy dokumentacji. Charakter: **statyczny**. Updates: przy wdrożeniu nowych wzorców projektowych.
4. **`memory-bank/activeContext.md`**: Rejestruje bieżący obszar skupienia (*current focus*), ostatnio introduced zmiany, aktywne decyzje architektoniczne oraz najbliższe kroki. Charakter: **wysoce dynamiczny**. Updates: po zakończeniu każdego zadania w fazie ACT.
5. **`memory-bank/progress.md`**: Śledzi stan realizacji projektu, odznaczone kamienie milowe, działające moduły, znane problemy (*known issues*) oraz backlog. Charakter: **wysoce dynamiczny**. Updates: po zweryfikowaniu prac i zamknięciu zadania.

Gotowe szablony Memory Bank (Dla projektu: Python, LaTeX, symulacje numeryczne)

1. Szablon: memory-bank/projectbrief.md

```
# Project Brief

## Cel i Zakres Projektu
Opracowanie i prowadzenie symulacji komputerowych w języku Python oraz równoległe tworzenie artykułu naukowego w LaTeX
w ustrukturyzowanym paradygmacie Inżynierii Agentowej.

## Kluczowe Wymagania i Wyniki (Deliverables)
- **Moduły Obliczeniowe**: Wykorzystanie zwektoryzowanych bibliotek (`numpy`, `scipy`) do wykonania obliczeń numerycznych.
- **Zestaw Testów**: Automatyczne testy jednostkowe, brzegowe i zbieżności w `pytest`.
- **Artykuł Naukowy**: Dokument LaTeX (`paper/main.tex`) z bibliografią BibTeX (`references.bib`) i wektorowymi wykresami PDF.

## Kryteria Jakościowe (Quality Gates)
1. **Stabilność Numeryczna**: Brak wartości NaN/Inf na każdym etapie symulacji oraz wymóg stałego ziarna losowości (`seed=42`).
2. **Bezbłędna Kompilacja**: Dokument LaTeX musi kompilować się komendą `latexmk -pdf` bez błędów (0 errors) i bez ostrzeżeń o braku odwołań (*undefined references*).
3. **Plan-and-Act**: Zakaz edycji plików bez zatwierdzonego planu w fazie PLAN.
```

2. Szablon: memory-bank/techContext.md

```
# Technical Context

## Stos Technologiczny

### 1. Środowisko Python
- **Wersja**: Python 3.10+
- **Biblioteki Obliczeniowe**: `numpy`, `scipy`, `matplotlib`, `seaborn`
- **Jakość Kodu i Testy**: `pytest`, `ruff` (linter/formatter), `mypy` (kontrola typów)

### 2. Środowisko LaTeX
- **Silnik Kompilacji**: `latexmk -pdf -halt-on-error`
- **Zarządzanie Bibliografią**: BibTeX (`references.bib`)
- **Główne Pakiety**: `amsmath`, `amssymb`, `graphicx`, `booktabs`, `hyperref`, `cleveref`

### 3. Układ Katalogów Repozytorium
```text
.
├── .agents/              # Uniwersalne wytyczne i reguły dla agentów
├── memory-bank/          # Pamięć projektu (projectbrief, techContext, systemPatterns, activeContext, progress)
├── src/                  # Kod źródłowy w Pythonie (models, solvers, utils)
```
```

```
├── tests/              # Testy pytest (numeryczne, brzegowe, guardraile)
├── paper/              # Dokumentacja LaTeX (main.tex, sections/, figures/, references.bib)
├── scripts/            # Skrypty weryfikacyjne (verify_agent_work.sh)
└── requirements.txt    # Zależności bibliotek w Pythonie
```

Polecenia CLI Weryfikacyjne

- **Testy jednostkowe**: `pytest tests/ -v --maxfail=1`
- **Linter kodu**: `ruff check src/ tests/`
- **Sprawdzanie typów**: `mypy src/ --ignore-missing-imports`
- **Kompilacja LaTeX**: `cd paper && latexmk -pdf -halt-on-error main.tex`
- **Pełna weryfikacja**: `bash scripts/verify_agent_work.sh`

3. Szablon: `memory-bank/systemPatterns.md`

```markdown
# System Patterns & Architecture

## 1. Wzorce Kodowania i Symulacji (Python)

### A. Wzorzec Trzystopniowy Symulacji (Simulation Pipeline Pattern)
Każdy moduł symulacji numerycznej musi realizować sekwencję:
1. **Setup & Initialization**: Definicja siatki, warunków brzegowych/początkowych i ustawienie ziarna losowości (`numpy.random.seed(42)`).
2. **Solver Execution Loop**: Wykonanie zwektoryzowanych obliczeń w NumPy/SciPy z aktywną kontrolą wartości NaN/Inf.
3. **Data Export & Sanity Plotting**: Zapis wyników oraz generowanie wektorowych wykresów weryfikacyjnych do PDF/PNG dla dokumentu LaTeX.

```text
[ Config / Init ] ──> [ Solver Loop (NumPy/SciPy) ] ──> [ Data Logger ] ──> [ Export & PDF Plots ]
```

B. Wzorzec Vectorization-First

- **Zakaz pętli Python**: Obliczenia numeryczne MUSZĄ wykorzystywać operacje zwektoryzowane NumPy. Surowe pętle `for` oraz list comprehensions w funkcjach obliczeniowych są zabronione i weryfikowane przez AST.
- **Jawne typowanie**: Wszystkie funkcje muszą posiadać adnotacje typów (`typing`, `numpy.ndarray`).

2. Wzorce Dokumentacji (LaTeX)

- **Modularność**: Główny plik `paper/main.tex` łączy rozdziały z podkatalogu `paper/sections/` przy użyciu `\include` lub `\input`.
- **Etykiety i Odwołania**: Wzory `\label{eq:...}` odwoływane przez `\eqref{eq:...}`, rysunki `\label{fig:...}`, tabele `\label{tab:...}` odwoływane przez `\cref{}`. Unikalne klucze BibTeX w `paper/references.bib`.

3. Standardy Weryfikacji (verification-first)

- Porównywanie wyników numerycznych z rozwiązaniami analitycznymi z użyciem tolerancji (`numpy.testing.assert_allclose`).
- Kompilacja `latexmk -pdf` z wynikiem 0 błędów.
- Statyczne pliki pamięci wyznaczają twarde granice architektoniczne niezbędne do bezpiecznego prowadzenia dynamicznych cykli wykonawczych.

---

4. Automatyzacja Protokołu Plan-and-Act i Synchronizacja Pamięci

Operacyjny rygor inżynierii agentowej wymusza stosowanie dwufazowego cyklu wykonawczego *Plan-and-Act*. Zapobiega to sytuacji, w której agent natychmiastowo przechodzi do edycji kodu na podstawie błędnych założeń, zużywając tokeny na niepoprawne iteracje.
```

```text
┌─────────────────────────────────────────────────────────────┐
│ STAGE 1: PLAN PHASE (Read-Only Analysis & Markdown Specification)│
└─────────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────────┐
│ STAGE 2: HUMAN APPROVAL CHECKPOINT (Wait for "APPROVED")    │
└─────────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────────┐
│ STAGE 3: ACT PHASE (Verification-First Execution & Testing) │
└─────────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────────┐
│ STAGE 4: POST-ACT MEMORY BANK SYNC (Update activeContext/progress)│
└─────────────────────────────────────────────────────────────┘
```

Szczegółowy przebieg cyklu zadania agenta

- **Stage 1 (PLAN – Faza do odczytu)**: Agent odczytuje katalog `memory-bank/`, analizuje kod, po czym formułuje plan wykonawczy w formacie Markdown zawierający: cel zadania, wykaz plików do modyfikacji, podejście algorytmiczne oraz strategię weryfikacji.
- **Stage 2 (Human Approval Checkpoint)**: Agent zatrzymuje pracę i jawnie oczekuje na weryfikację i odpowiedź człowieka („APPROVED"). Praca w kodzie na tym etapie jest zabroniona.
- **Stage 3 (ACT – Faza wykonania)**: Po uzyskaniu akceptacji agent tworzy lub aktualizuje testy w `pytest`, wprowadza zmiany w kodzie Python lub dokumentacji LaTeX i uruchamia skrypty weryfikacyjne.
- **Stage 4 (Post-ACT Memory Bank Sync)**: Po pomyślnym przejściu weryfikacji agent aktualizuje status w plikach `memory-bank/activeContext.md` oraz `memory-bank/progress.md`.

Gotowy szablon reguły dla agenta (.github/copilot-instructions.md / .clinerules / .cursor/rules)

```
# AGENT INSTRUCTION: STRICT PLAN-AND-ACT WORKFLOW WITH MEMORY BANK SYNC

You are an autonomous AI Coding Agent. You MUST operate strictly following the **Plan-and-Act** pattern.

---

## STAGE 1: PLAN PHASE (Read-Only & Analysis)
1. Read all files in `memory-bank/` (`projectbrief.md`, `techContext.md`, `systemPatterns.md`, `activeContext.md`, `progress.md`).
2. Inspect existing code, `.tex` files, and `pytest` suites.
3. Formulate a structured Markdown Execution Plan:
  - **Objective**: Clear scope of changes.
  - **Target Files**: List of files to edit/create.
  - **Algorithmic/Mathematical Strategy**: Solvers, NumPy vectorization, or LaTeX sections.
  - **Verification Strategy**: Exact `pytest` cases and `latexmk` commands.
4. STOP and request approval with:
  > *"Plan generation complete. I am waiting for Human Architect review. Reply 'APPROVED' to execute or provide feedback."*
5. **RESTRICTION**: You are FORBIDDEN from editing or creating files during Stage 1.

---

## STAGE 2: HUMAN APPROVAL
- Wait for explicit user confirmation ("APPROVED").

---

## STAGE 3: ACT PHASE (Execution & Verification)
1. Write or update tests in `pytest` alongside implementation.
2. Modify Python scripts or LaTeX source files in logical steps.
3. Run verification: execute `bash scripts/verify_agent_work.sh`. Ensure 0 errors.

---

## STAGE 4: POST-ACT MEMORY BANK SYNCHRONIZATION
Immediately after verification, update `memory-bank/`:
- `activeContext.md`: Log current focus, recent changes, active decisions.
- `progress.md`: Mark completed tasks as `[x]` and log new milestones.
```

Conclude with:
> *"Task executed, verified, and Memory Bank files (`activeContext.md`, `progress.md`) successfully synchronized."*

Przykład stanów plików pamięci po synchronizacji w Etapie 4

Stan pliku memory-bank/activeContext.md:

```
# Active Context

## Aktualny Obszar Prac (Current Focus)
- Implementacja zwektoryzowanego solvera numerycznego oraz kompilacja sekcji wyników w LaTeX.

## Ostatnio Wprowadzone Zmiany (Recent Changes)
- **Python**: Zaimplementowano funkcję całkowania w `src/solvers/numerical_solver.py`.
- **pytest**: Dodano testy stabilności i braku wartości NaN/Inf w `tests/test_solver.py`.
- **LaTeX**: Wygenerowano wykres `paper/figures/convergence.pdf` i dołączono do `paper/sections/03_results.tex`.

## Aktywne Decyzje i Kompromisy (Active Decisions & Trade-offs)
- Zastosowano operacje macierzowe NumPy zamiast pętli dla zapewnienia wydajności przy gęstej siatce.
- Ustawiono stałe ziarno losowości (`numpy.random.seed(42)`).

## Najbliższe Kroki (Immediate Next Steps)
1. Przeprowadzić przegląd diffa z Architektem.
2. Przygotować Pull Request do gałęzi głównej.
```

Stan pliku memory-bank/progress.md:

```
# Project Progress

## Status Projektu i Kamienie Milowe
- [x] Utworzenie i skonfigurowanie struktury **Memory Bank** (`projectbrief`, `techContext`, `systemPatterns`).
- [ ] **Moduł Symulacyjny (Python)**:
  - [x] Implementacja bazy solvera numerycznego.
  - [x] Wdrożenie testów zbieżności i stabilności numerycznej w `pytest`.
- [ ] **Dokumentacja i Publikacja (LaTeX)**:
  - [x] Konfiguracja głównego pliku `paper/main.tex`.
  - [x] Generowanie wektorowych wykresów PDF z symulacji.
  - [ ] Końcowa weryfikacja bibliografii BibTeX.
```

Plany wykonawcze są jednak tylko tak niezawodne, jak automatyczne testy i guardraile, które je walidują.

5. Podejście Verification-First: Skrypty Skrajnej Walidacji i Guardraile

Podejście *Verification-First* polega na zdefiniowaniu barier ochronnych (testów, linterów, kontrolerów typowania) przed przystąpieniem do generowania kodu lub równolegle z nim. Zapobiega to fałszywym komunikatom o sukcesie, halucynacjom oraz brakowi stabilności numerycznej (wartościom NaN/Inf).

Configurations i Kod Guardrailowy

1. Konfiguracja Lintera i Formattera: pyproject.toml

```
[tool.ruff]
target-version = "py310"
line-length = 88

[tool.ruff.lint]
select = [
    "E",   # Błędy pycodestyle (PEP 8)
    "F",   # Błędy Pyflakes (nieużywane importy, braki zmiennych)
    "UP",  # Unowocześnienia składni pyupgrade
    "B",   # Wykrywanie pułapek i błędów logicznych (flake8-bugbear)
    "SIM", # Uproszczenia kodu (flake8-simplify)
    "TCH", # Sprawdzanie adnotacji typów (flake8-type-checking)
]
ignore = []
```

2. Guardraile numeryczne i statyczne (AST): `tests/test_agent_guardrails.py`

W przeciwieństwie do naiwnej analizy tekstowej (która daje fałszywe alarmy na komentarzach lub pomija `list comprehensions`), poniższy test wykorzystuje moduł `ast` (Abstract Syntax Tree) Pythona. Precyzyjnie parsuje on drzewo składniowe funkcji symulacyjnej i zgłasza błąd w przypadku wykrycia pętli `for`, `async for` lub wyrażeń listowych.

```
import ast
import inspect
import numpy as np
import pytest
import src.solvers.numerical_solver as solver


class LoopVisitor(ast.NodeVisitor):
    """Skanuje drzewo AST w poszukiwaniu niewektoryzowanych konstrukcji pętli."""

    def __init__(self):
        self.found_loops = []

    def visit_For(self, node):
        self.found_loops.append(f"Pętla 'for' w linii {node.lineno}")
        self.generic_visit(node)

    def visit_AsyncFor(self, node):
        self.found_loops.append(f"Pętla 'async for' w linii {node.lineno}")
        self.generic_visit(node)

    def visit_ListComp(self, node):
        self.found_loops.append(f"List comprehension w linii {node.lineno}")
        self.generic_visit(node)


def test_random_seed_reproducibility():
    """Weryfikuje, czy algorytm respektuje ziarno losowości (reprodukowalność)."""
    np.random.seed(42)
    res1 = solver.run_simulation(seed=42)

    np.random.seed(42)
    res2 = solver.run_simulation(seed=42)

    np.testing.assert_allclose(
        res1,
        res2,
        err_msg="Symulacja nie jest reprodukowalna przy tym samym seedzie!",
    )


def test_numerical_stability_no_nan_inf():
    """Wykrywa obecność niepożądanych wartości NaN lub Inf w wynikach."""
    result = solver.run_simulation(seed=42)

    assert not np.isnan(
        result
    ).any(), "Wykryto wartości NaN w wynikach symulacji!"
    assert not np.isinf(
        result
    ).any(), "Wykryto wartości Inf w wynikach symulacji!"


def test_vectorization_no_raw_python_loops():
    """Statyczna inspekcja AST: wymusza stosowanie operacji zwektoryzowanych NumPy."""
    source_code = inspect.getsource(solver.run_simulation)
    tree = ast.parse(source_code)

    visitor = LoopVisitor()
    visitor.visit(tree)

    assert not visitor.found_loops, (
        "Wykryto niewektoryzowane pętle w kodzie symulacji numerycznej:\n"
        + "\n".join(visitor.found_loops)
        + "\nZastosuj operacje zwektoryzowane NumPy."
    )
```

3. Automatyczny skrypt weryfikacyjny: `scripts/verify_agent_work.sh`

Operacje na podkatalogach wykonywane są w bezpiecznym podpowłoce (*subshell*), co zapobiega zakleszczeniu katalogu roboczego w przypadku błędów kompilacji.

```
#!/usr/bin/env bash
set -e

echo "=== 1. Analiza Statyczna Lintera (Ruff) ==="
ruff check src/ tests/

echo "=== 2. Statyczna Kontrola Typów (Mypy) ==="
mypy src/ --ignore-missing-imports

echo "=== 3. Wykonanie Testów Numerycznych i Guardrailów (Pytest) ==="
pytest tests/ -v --maxfail=1

echo "=== 4. Kompilacja Dokumentacji LaTeX ==="
if [ -f "paper/main.tex" ]; then
    (cd paper && latexmk -pdf -halt-on-error main.tex)
fi

echo "Weryfikacja zakończona sukcesem (Status 0). Kod gotowy do Code Review."
```

Lokalne skrypty weryfikacyjne bezpośrednio łączą się z automatycznymi potokami CI/CD chroniącymi produkcyjne gałęzi repozytorium.

6. Integracja CI/CD w GitHub Actions oraz szablony zgłoszeń i PR

Automatyczna integracja w środowisku GitHub wymusza bezkompromisowe bramki jakościowe (*Quality Gates*) przy każdym zdarzeniu Pull Request tworzonym przez agenta AI.

Pełna konfiguracja potoku CI/CD: `.github/workflows/agent-verification.yml`

Uwaga dotycząca wydajności DevEx: Pobieranie pakietów `texlive` przez `apt-get` przy każdym wywołaniu PR może dodawać 2–4 minuty do czasu wykonywania potoku. W środowiskach produkcyjnych zaleca się stosowanie gotowych akcji kontenerowych (np. `xu-cheng/latex-action@v1`) lub wykorzystanie pamięci podręcznej GitHub Actions (*caching*).

```
name: Agent Verification Pipeline

on:
  pull_request:
    branches: [ main, master ]
  workflow_dispatch:

jobs:
  verify-agent-work:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.10'
          cache: 'pip'

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
          pip install ruff mypy pytest numpy scipy matplotlib

      - name: 1. Linting and Formatting Check (Ruff)
        run: |
          ruff check src/ tests/

      - name: 2. Static Type Check (Mypy)
        run: |
          mypy src/ --ignore-missing-imports

      - name: 3. Numerical & Unit Tests (Pytest)
        run: |
          pytest tests/ -v --maxfail=1

      - name: 4. LaTeX Compilation Check
        if: hashFiles('paper/main.tex') != ''
```

```
        uses: xu-cheng/latex-action@v1
        with:
          root_file: paper/main.tex
          args: "-pdf -halt-on-error"
```

Szablony procesowe dla systemu GitHub

1. Szablon Zadania: `.github/ISSUE_TEMPLATE/agent_task.md`

```
---
name: Task for AI Agent
about: Zadanie programistyczne dla agenta AI z wymogiem Plan-and-Act
title: '[AGENT] '
labels: agent-task, plan-and-act
assignees: ''
---

## Cel Zadania
Opisz zwięźle cel (np. implementacja solvera, dodanie sekcji w LaTeX).

## Ograniczenia i Wymagania
- Stos: Python / LaTeX
- Zasady: Operacje zwektoryzowane NumPy, brak pętli `for` (weryfikacja AST), rygor typowania, stały
random seed.
- Wymóg Plan-and-Act: Praca WYŁĄCZNIE w trybie odczytu do momentu zatwierdzenia planu.

## Etap 1: Wymagany Plan (Do wypełnienia przez Agenta)
- [ ] Odczytano pliki z `memory-bank/` oraz `.agents/`.
- [ ] Przedstawiono wykaz plików do modyfikacji.
- [ ] Zdefiniowano przypadki testowe w `pytest`.
- [ ] Oczekiwanie na zatwierdzenie planu (odpowiedź "APPROVED").

## Etap 2: Kryteria Akceptacji (Faza ACT)
- [ ] Testy `pytest` przechodzą (brak wartości NaN/Inf, zachowana reprodukowalność).
- [ ] Kod przeszedł weryfikację linterem `ruff` oraz `mypy`.
- [ ] Dokument LaTeX kompiluje się bez błędów (`latexmk -pdf`).
- [ ] Zaktualizowano `memory-bank/activeContext.md` oraz `memory-bank/progress.md`.
```

2. Szablon PR: `.github/PULL_REQUEST_TEMPLATE.md`

```
## Opis Zmian
Zwięzłe podsumowanie modyfikacji wprowadzonych przez agenta AI.

## Lista Kontrolna Weryfikacji (Verification Checklist)
- [ ] Kod przeszedł automatyczne testy numeryczne (`pytest`).
- [ ] Statyczna analiza kodu nie wykazuje błędów (`ruff check`, `mypy`).
- [ ] Kompilacja LaTeX zakończyła się sukcesem (0 błędów, status 0).
- [ ] Stałe ziarno losowości (`random seed`) zostało ustalone.

## Synchronizacja Pamięci (Memory Bank Sync)
- [ ] Zaktualizowano `memory-bank/activeContext.md` (bieżący punkt skupienia).
- [ ] Zaktualizowano `memory-bank/progress.md` (odznaczone wykonane zadania).
```

Ustrukturyzowane potoki i szablony w prosty sposób umożliwiają prowadzenie efektywnych przeglądów kodu z udziałem człowieka w pętli.

7. Dobre praktyki zarządzania flotą agentów, operacje na diffach i code review

W inżynierii agentowej rola programisty przechodzi zasadniczą ewolucję: z pisarza linii kodu inżynier staje się Głównym Architektem i Senior Reviewerem zarządzającym cyfrowymi stażystami. Human code review przestaje być powierzchownym sprawdzaniem literówek – staje się zaawansowanym audytem architektonicznym.

Wytyczne dla procesu Code Review w przepływach agentowych

1. **Szczegółowa analiza różnic w kodzie (*Diff-based Review*)**: W przeciwieństwie do vibe codingu, zabrania się akceptowania zmian bez czytania diffa. Architekt ma obowiązek przeanalizować każdy zmodyfikowany plik i wycinek kodu pod kątem ukrytych usterek logicznych.
2. **Kolejność weryfikacji (*Verification-First*)**: Przegląd ludzki następuje dopiero po tym, jak kod przejdzie automatyczne barierki ochronne (lintery, typowanie, testy AST i kompilację LaTeX) w potoku CI/CD.

3. **Weryfikacja spójności architektonicznej**: Należy sprawdzić, czy agent zrealizował dokładnie zakres uzgodniony w fazie PLAN oraz czy zachował wzorce określone w `memory-bank/systemPatterns.md` (np. operacje zwektoryzowane).
4. **Badanie skrajnych przypadków i ręczne QA**: Inżynier musi przeanalizować nieoczywiste scenariusze brzegowe (*edge cases*) oraz przeprowadzić ręczne testy logiki biznesowej.
5. **Środowiska podglądowe (*Preview Environments*)**: Kod wygenerowany przez agenta należy przed scaleniem z gałęzią główną uruchamiać w bezpiecznych środowiskach podglądowych.
6. **Rygor historii Git (*Git Hygiene*)**: Wymuszanie pracy na małych, czystych i atomowych commitach przypisanych do osobnych gałęzi Git (np. `feature/issue-12`), co ułatwia analizę i ewentualne cofanie błędnych iteracji.

Skalowanie floty i architektura wielozespołowa

Gdy do realizacji wielomodułowych zadań wykorzystywana jest równolegle floty agentów, standardowe, pojedyncze pliki pamięci mogą ulec fragmentacji. W skali przedsiębiorstwa wdraża się następujące wzorce:

- **Struktury Podów Agentowych (*Multi-Agent Pods*)**: Przydzielanie wyspecjalizowanych agentów pod nadzorem jednego inżyniera-architekta (np. Pod Numeryczny, Pod LaTeX/Dokumentacyjny, Pod CI/CD).
- **Architektura Pamięci Federacyjnej (*Federated Memory Architecture*)**: Każdy pod lub zespół utrzymuje lokalny katalog `memory-bank/` powiązany z centralnym repozytorium wiedzy organizacji, ze ściśle zdefiniowanymi protokołami synchronizacji.
- **Przestrzenie Nazw i Kontekst Gałęzi (*Context Namespacing & Branch Awareness*)**: Odczyt kontekstu dostosowany do gałęzi Git. Przy przełączaniu gałęzi agent ładuje wyłącznie deltę wiedzy zmodyfikowaną na danym *branchu*, dbając o spójność architektury globalnej.

Top 5 zasad operacyjnych do natychmiastowego wdrożenia w repozytorium

1. **Stwórz strukturę pointer-pattern**: Dodaj uniwersalne zasady w `.agents/general_rules.md` oraz lekkie pliki startowe (`CLAUDE.md`, `.cursor/rules/main.mdc`, `.github/copilot-instructions.md`) nakazujące ich odczyt.
2. **Inicjalizuj Memory Bank**: Utwórz katalog `memory-bank/` z 5 plikami Markdown (`projectbrief`, `techContext`, `systemPatterns`, `activeContext`, `progress`) w celu eliminacji amnezji sesyjnej.
3. **Wymuś protokół Plan-and-Act**: Kategorycznie zabroń agentowi edycji kodu w pierwszej fazie pracy do momentu uzyskania akceptacji planu przez człowieka.
4. **Wdrożyj skrypt weryfikacji lokalnej z guardrailem AST**: Utwórz `scripts/verify_agent_work.sh` wraz z testem parsera AST i zobowiąż agenta w pliku reguł do jego uruchamiania przed zgłoszeniem zakończenia zadania.
5. **Uruchom potok CI/CD w GitHub Actions**: Skonfiguruj `.github/workflows/agent-verification.yml`, aby żaden PR utworzony przez agenta nie został scalony bez zaliczenia linterów, typowania, testów numerycznych i kompilacji dokumentacji.