

The background image is an aerial photograph of a natural landscape. It shows a dense forest with various shades of green and yellow autumn foliage. A light-colored, sandy path or beach runs along the edge of a dark green body of water. The overall scene is peaceful and suggests a remote, natural environment.

# Sztuczna Inteligencja

## w pracy Analityka Systemowego

– Koncepcje, Narzędzia i Metodologie



# Szkolenie

- Plan szkolenia i cele
- Aktualne obowiązki
- Pytania, dyskusje, potrzeby
- Elastyczny program szkoleniowy



# Mateusz Kulesza

- Solution Architect
- Senior Software Developer
- Consultant and Trainer



# Uczestnik

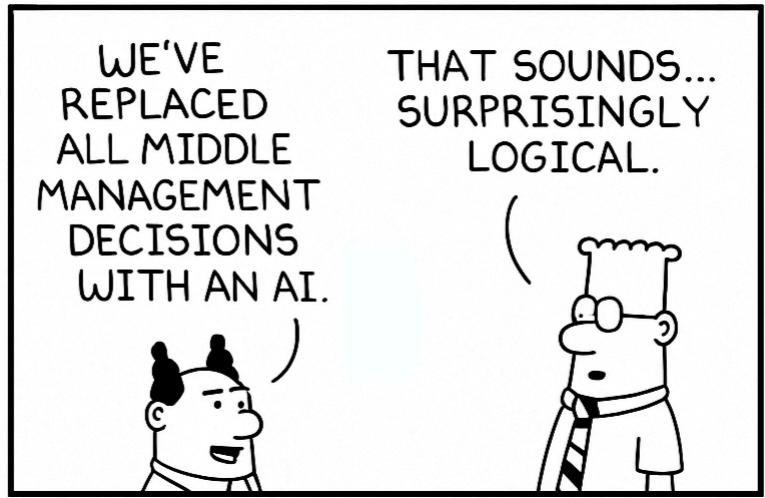
czyli "Ty"

Kilka szybkich pytań:

- Twoje codzienne obowiązki?
- Technologie i background?
- Poprzednie doświadczenie?
- Twoje cele na to szkolenie?
- Określone obszary lub tematy?

# Rzeczywistość

w czasach AI



Comic generated by ChatGPT in the style of Scott Adams.

# Jak architekt powinien odróżniać marketing AI od twardej inżynierii?



## Hype

– vs. Produktywna Rzeczywistość

🤔 Dlaczego w ogóle jest hype?

- LLM-y wyglądają jak „inteligencja” → płynny język ≠ głęboka wiedza
- Marketing producentów modeli: „agent”, „rozumowanie”, „samodzielność”
- Media i społeczność wzmacniają narrację „AI zastąpi programistów”
- Większość testów jest kontentowych, nie inżynieryjnych
  - screenshotsy promptów zamiast rzeczywistych pipelines



# Produktywna rzeczywistość

– co faktycznie działa dziś?

AI = komponent w architekturze, nie mózg systemu

- Model jest funkcją probabilistyczną, a nie deterministyczną
- Wymaga otoczenia "prawdziwego" systemu:
  - walidacja I/O
  - retry + kompensacje
  - RAG + filtracja danych
  - obserwowalność (SLI/SLO)
  - governance
  - i wiele innych ...

# Co działa 100% przewidywalnie?

– większość AI nie jest przewidywalne!

## Aby mieć gwarancje wyników:

- Kod generowany pod nadzorem → oszczędza 30–70% czasu
- AI jako parser / konwerter / etykieciarz danych
- Embeddingi + wektorowe wyszukiwanie
- Pipeline'y złożone z wielu prostych modeli, np.
  - OCR → extraction → LLM → decision engine
- Generowanie dokumentów technicznych, diagramów, testów
- Asystenci w IDE (Cursor, Copilot++)

AI jako probabilistyczna część deterministycznego systemu

# Co działa, ale z ograniczeniami?

- AI Bezpośrednio, ale pod kontrolą
- Autonomous Agents → w praktyce 1–3 zadania, nie łańcuchy 100 kroków
- Fine-tuning → sens only w przypadku dużych wolumenów i stabilnej domeny
- AI in the loop → AI jako element procesu
- Human in the loop → AI jako "asystent", człowiek weryfikuje
- Retrieval Augmented Generation, ale tylko z:
  - chunkingiem dopasowanym do schematu danych
  - filtrami domenowymi
  - guardrails
  - testami regresyjnymi promptów
  - itd...

# Co NIE DZIAŁA tak, jak mówi hype

Marketingowa "ścema" i science-fiction:

- „AI zrozumie kontekst firmy i będzie jak pracownik”
- „Agent sam ogarnie wieloetapowe zadanie bez kontroli”
- „LLM zastąpi architekta”
- „Model wygeneruje od razu poprawny kod produkcyjny”
- „Dzięki Vibe-Coding nie trzeba rozumieć kodu”
- „RAG rozwiąże wszystkie problemy z wiedzą firmy”

AI nie "myśli", tylko przewiduje co będzie dalej:

- Model jest trenowany na opiniach co jest "dobrze" a co "źle"
- Im lepsze dostaje dane wejściowe tym lepsze predykcje
- Im lepiej kontrolowany proces tym mniej niespodzianek



# Rzeczywiste wyzwania,

o których hype nie mówi

- Brak transakcyjności → model może „magicznie” zmienić odpowiedź
- Halucynacje w 2–10% przypadków nawet w top modelach
- Drift danych biznesowych → model staje się nieaktualny po 3–9 miesiącach
- Zarządzanie danymi osobowymi (RODO, AI Act)
- Compliance, governance, audytowalność
- Black box problem → dlaczego model podjął taką decyzję?
- Idempotencja? → tylko pozorna
- Brak gwarancji deterministycznej interpretacji polecenia

Jak to ogarnać? 

Podstawy sztucznej inteligencji i uczenia maszynowego z perspektywy  
**anityka systemów**

# Odpowiedzialność Analityka

💡 Co analityk musi wiedzieć, a czego nie musi?

- Analityk NIE trenuje modele  
→ to rola data scientistów / ML engineerów
- AI NIE zastępuje Analityka  
→ to narzędzie a nie decydent
- AI to nie magiczna kula  
→ wymaga procesu i kontroli

- Analityk definiuje problemy:  
→ Gdzie AI może dodać wartość?
- Analityk wymienia wymagania dla AI:  
→ Jakie dane są potrzebne?  
→ Jakie są ograniczenia i ryzyka?

## Analityk WŁĄCZA AI w procesy decyzyjne

→ definiuje jak AI wpływa na zbieranie wymagań,  
walidację, dokumentację, itd ...



## Od przetwarzania informacji do wzmacniania decyzji

- AI ≠ zamiennik człowieka
- AI = narzędzie poszerzające opcje i przyspieszające pracę
- Analityk to ten, który podejmuje ostateczne decyzje:
  - weryfikuje wyniki
  - kwestionuje założenia
  - wprowadza ograniczenia biznesowe





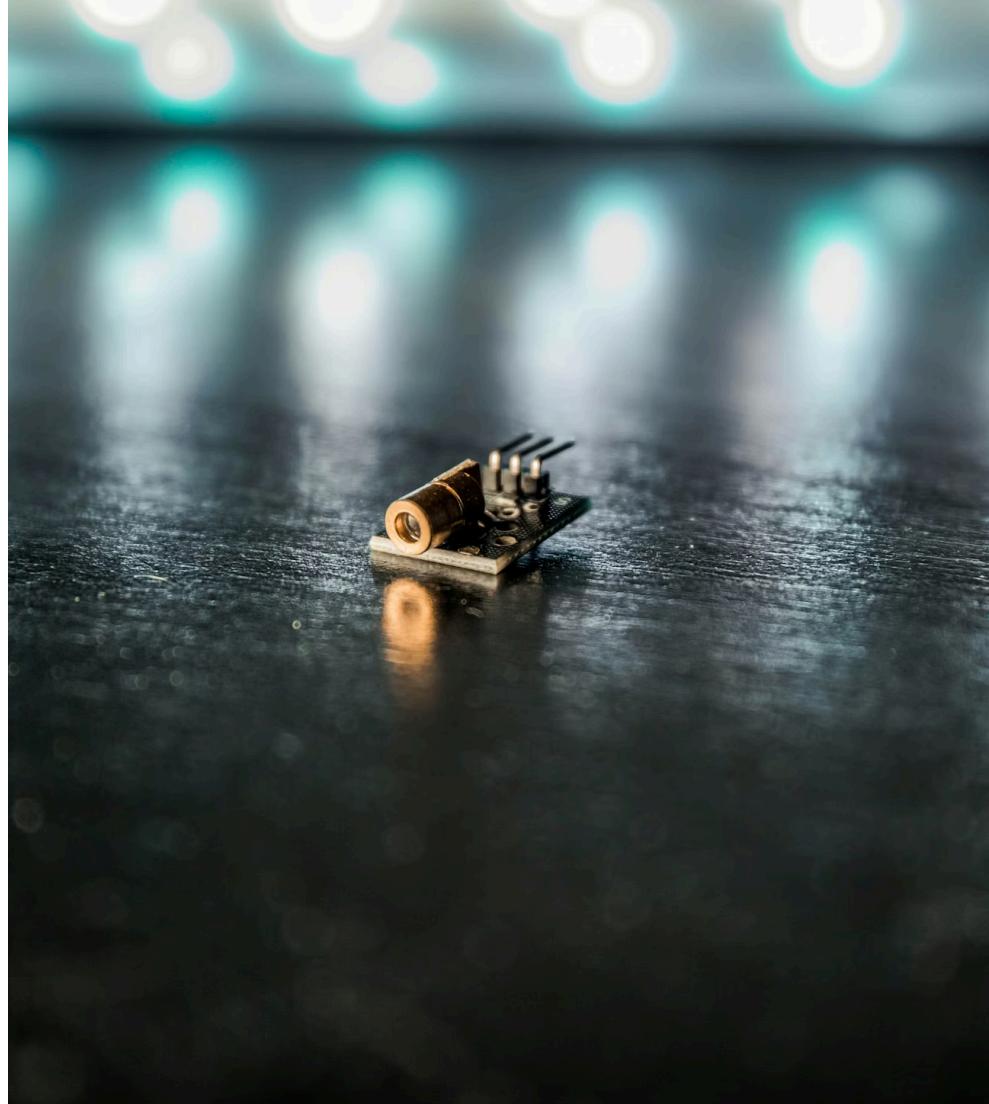
## AI jako junior analityk

- Mentalny model analityka:
  - „AI jest junior analitykiem z określonymi kompetencjami”
  
- Junior analityk powinien:
  - znać zakres swoich możliwości (domain of competence)
  - mieć jasne warunki błędów
  - być spójny (LLM) albo wiarygodny (klasyczny ML)
  - być na bieżąco (aktualne dane)

# AI jako narzędzie analytyka

– Analityk patrzy na AI jak na każde inne narzędzie:

- jaki ma **Cel** (co rozwiązuje?)
- jakie ma **ograniczenia** (co nie potrafi?)
- jakie ma **koszty**:
  - czasowe i finansowe,
  - jakościowe,
  - prawne,
  - utrzymaniowe
- jakie ma **ryzyka** (co może się nie udać?)
- jak wpływa na procesy i decyzje



Gdzie AI wspomaga analityka

Co AI potrafi?  
Czego NIE potrafi?

Gdzie jest najwartościowsze?

Gdzie analityk musi być czujny?

# Analityk:

przed i po erze AI

- Przesunięcie akcentu: z pisania na rozumienie i wnioskowanie.
- Mniej produkcji tekstu, więcej osądu i priorytetyzacji.
- Rola kuratora informacji: wybieraj, co ważne, a co szum.
- AI przyspiesza pracę, ale nie podejmuje decyzji za ludzi.
- Odpowiedzialność i rozliczalność pozostają po stronie człowieka.

Od przetwarzacza informacji

## do wzmacniacza decyzji

- AI poszerza przestrzeń opcji —> analityk zawęża do sensownych wariantów.
- Wartość często leży w tym, czego NIE budujemy (świadoma rezygnacja).
- Definiuj kryteria wyboru: koszt, ryzyko, zgodność, wartość biznesowa, czas.
- Pracuj w pętli: eksploruj → syntetyzuj → uzasadnij → zdecyduj.
- Dokumentuj uzasadnienie decyzji, nie tylko wynik (audit trail).

# Analityk

vs Architekt vs Deweloper

- Analityk: ramuje problem, definiuje potrzeby i ograniczenia biznesowe.
- Architekt: kształtuje rozwiązanie, wybiera komponenty i integracje.
- Deweloper: realizuje implementację zgodnie z wymaganiami i architekturą.
- AI wspiera wszystkich, ale niczego „nie posiada” – to narzędzie.
- Przepływ: hipotezy (Analityk) → wzorce/komponenty (Architekt) → kod (Dev).

# Odpowiedzialność

Modele generują wyniki — ludzie podejmują decyzje i biorą odpowiedzialność.

- Ustal RACI dla decyzji wspieranych przez AI (kto decyduje, kto doradza, kto zatwierdza).
- Zapewnij ślad audytowy: źródła danych, wersja modelu, parametry, prompt, wynik.
- Dbaj o zgodność: prawo, regulacje, polityki danych (data governance, DPIA).
- Minimalizuj ryzyko: walidacja, monitorowanie jakości, eskalacja do człowieka.

Gdzie AI daje

## największą dźwignię

- **Szybkość:** streszczanie dokumentów, porównywanie wariantów, generowanie list kontrolnych.
- **Spójność:** narzucona struktura artefaktów (BRD, user stories, kryteria akceptacji).
- **Pokrycie:** większy zasięg analizy (więcej źródeł, więcej scenariuszy edge-case).
- **Ideacja:** propozycje wymagań niefunkcjonalnych, ryzyk, metryk, testów UAT.
- **Mapowanie domeny:** słowniki pojęć, taksonomie, relacje (zawsze z walidacją).

Gdzie AI

# nie powinno decydować

Akceptacja ryzyka i kompromisy biznesowe — to decyzje menedżerskie.

- Interpretacja zgodności (compliance) i regulacji – wymaga kompetencji prawnych.
- Decyzje etyczne i wpływ na klientów/pracowników – potrzebna ocena człowieka.
- Sytuacje z niepełnymi danymi i wysoką niepewnością – preferuj eskalację.
- Gdy konsekwencje są krytyczne (finanse, bezpieczeństwo, reputacja) – human-in-the-loop.

AI jako współanalityk:

## Junior Analist

Traktuj AI jak juniora — świetny w produkcji draftów, wymagający przeglądu.

- Stosuj wzorce pracy: chain-of-thought, krytyk/recenzent, checklisty walidacyjne.
- Buduj guardraile: zakres, format, źródła, ograniczenia, definicje „done”.
- Weryfikuj: cytaty ze źródeł, odwołania, porównania między wariantami.
- Ucz poprzez feedback: poprawki w promptach, przykłady (few-shot), style guide.

# AI to dużo więcej niż ChatGPT

Jest wiele rodzajów sztucznej inteligencji

# Predictive Analytics

Przewidywanie na podstawie danych historycznych.

- **Forecasting** – popyt, sprzedaż, ceny
- **Risk scoring** – bankowość, ubezpieczenia
- **Churn prediction** – utrata klientów
- **Anomaly Detection** – fraudy, nieprawidłowości
- **Predictive Maintenance** – awarie maszyn
- **Credit Scoring** – scoring BIK/KNF
- **Pricing Models** – dynamiczne ceny, airline pricing

# Predictive Analytics

Przewidywanie na podstawie danych historycznych.

## Forecasting

- Prophet, NeuralProphet, ARIMA, GluonTS

## Risk / Scoring

- XGBoost, LightGBM, CatBoost

## Anomaly Detection

- Isolation Forest, LOF, PyOD, Azure Anomaly Detector

## Predictive Maintenance

- Azure ML, AWS SageMaker, TensorFlow, PyTorch

# Natural Language Processing

NLP - Rozumienie i generowanie języka naturalnego.

**Klasyfikacja tekstu** – spam, ryzyka, tonacja

**Named Entity Recognition** – firmy, osoby, kwoty

**Information Extraction** – dane z umów, polis, pism sądowych

**Intent Detection** – routing spraw, call centers

**Machine Translation** – tłumaczenia techniczne

**Sentiment & Emotion Analysis** – social listening

**Semantic Search** – RAG, wektory, chat z dokumentacją

**Summarization** – skróty dokumentów, protokoły

# Natural Language Processing

NLP - Rozumienie i generowanie języka naturalnego.

## Klasyfikacja, NER

Modele BERT, spaCy, HuggingFace Transformers

Flair NLP, fastText

## Semantic Search / RAG

OpenAI Embeddings, Voyage, Cohere Embed,  
HuggingFace sentence-transformers

PgVector, Pinecone, Weaviate, ChromaDB, Milvus

## Summarization & QA

LLM: GPT-4/5, Claude, Gemini, Llama, Mistral

Pipeline: LangChain, LlamaIndex, OpenAI Assistants API

## Machine Translation

MarianMT, M2M-100, NLLB, DeepL API

# Computer Vision

AI operująca na obrazach, wideo i sygnałach wizualnych.

- **OCR / ICR** – faktury, KYC, dowody, paszporty
- **Doc Understanding** – klasyfikacja dokumentów, tabelaryzacja
- **Object Detection** – YOLO, DETR, autonomiczne pojazdy
- **Image Segmentation** – medycyna, analizy satelitarne
- **Face Recognition** – dostęp fizyczny
- **Quality Control** – linie produkcyjne
- **Medical Imaging** – RTG, MRI, CT
- **Satellite/Drone Vision** – geodezja, rolnictwo

# Computer Vision

AI operująca na obrazach, wideo i sygnałach wizualnych.

- **OCR / ICR**

Tesseract OCR, Google Vision OCR, Azure Form Recognizer, AWS Textract

PaddleOCR, EasyOCR, Kofax

- **Object Detection**

YOLOv5/v7/v8, Detectron2, DETR, OpenCV DNN

Frameworki: PyTorch, TensorFlow, Keras

- **Image Segmentation**

U-Net, Mask R-CNN, Segment Anything (SAM)

- **Face Recognition**

dlib, FaceNet, DeepFace, OpenCV

# Speech AI

Audio, głos, mowa, sygnały akustyczne.

- ASR – Automatic Speech Recognition
- Speech-to-Text – transkrypcje, call center
- Text-to-Speech – wirtualni asystenci
- Speaker Recognition – biometria głosowa
- Emotion Detection – analiza stresu w call center

# Speech AI

Audio, głos, mowa, sygnały akustyczne.

## Speech-to-Text (ASR)

Whisper, DeepSpeech, Google Speech API, Amazon Transcribe

## Text-to-Speech (TTS)

ElevenLabs, Microsoft Neural TTS, Coqui TTS

## Speaker Recognition

DeepSpeaker, speechbrain

# Generative AI

Tworzenie nowych treści.

- LLM – generowanie tekstu, kodu, dokumentacji
- Code Generation – Copilot, PR review
- Image Generation – obrazy, UI mockup
- Video Generation – animacje, generowanie scen
- Audio Generation – muzyka, narracja
- 3D/Modeling – CAD, VR, gry
- Diagram Generation – sekwencje, architektury
- Agent-based Automation – agenci wieloetapowi

# Generative AI

Tworzenie nowych treści.

## Text (LLM)

- OpenAI GPT-4.1/5, Claude 3.x, Gemini 2.x
- Llama 3.x, Mistral 8x7B, Qwen 2

## Image Generation

- Stable Diffusion (SDXL, SD 3)
- Midjourney, DALL-E 3/4, Adobe Firefly

## Video Generation

- Sora, Pika Labs, RunwayML Gen-2

## Audio

- ElevenLabs, Sunō, Tortoise-TTS, Whisper

## 3D / CAD

- Gaussian Splatting, NerfStudio

## Code Generation

- GitHub Copilot, Cursor, Codeium, Aider

# Knowledge AI & RAG

Praca na wiedzy, danych, dokumentach, bazach.

- Vector Search
- Retrieval-Augmented Generation
- Enterprise Knowledge Graphs
- Document QA – chatboty domenowe
- Semantic ETL – automatyczne przetwarzanie danych

Historia AI

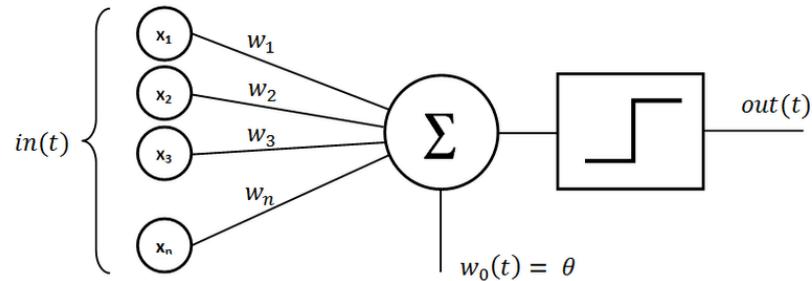
w pigułce



# Wczesne sieci neuronowe

(lata 60–80)

Perceptron - uczenie przez stopniowe dobieranie wag i biasów (progów) tak, by wyniki podawane na wyjściu sieci były coraz bardziej zgodne z wartościami oczekiwanyimi

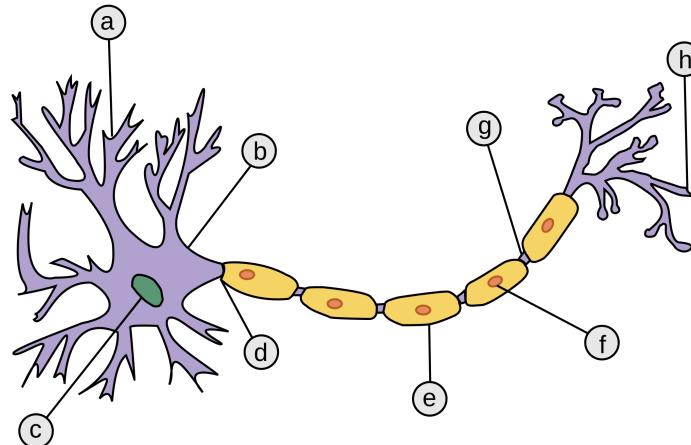


Kamienie milowe:

- perceptron (Rosenblatt, 1957)
- sieci jednokierunkowe → ograniczona moc
- brak efektywnego uczenia wielowarstwowego

Problem:

Minsky, Papert: perceptron nie potrafi XOR - Zima AI



## Perceptron – podstawy

Perceptron to najprostszy model sztucznego neuronu używany do liniowej klasyfikacji.

### Model matematyczny

Wejścia:  $x_1, x_2, \dots, x_n$

Wagi:  $w_1, w_2, \dots, w_n$

Bias:  $b$

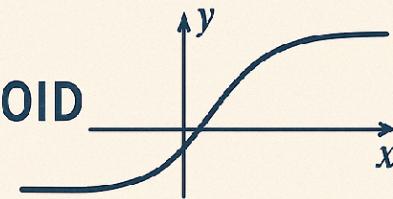
Obliczenie aktywacji neuronu:

$$z = \sum_{i=1}^n w_i x_i + b$$

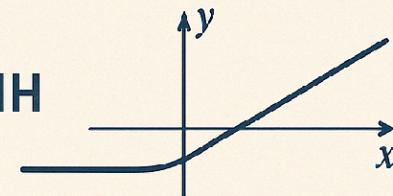
Wyjście po funkcji aktywacji (np. sign):

$$\hat{y} = \text{sign}(z)$$

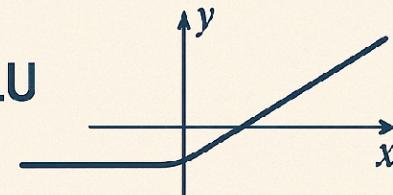
SIGMOID



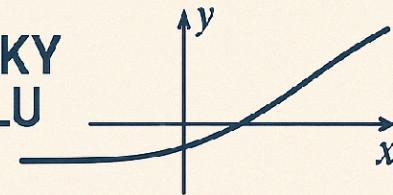
TANH



RELU



LEAKY  
RELU



GELU



# Backpropagation

(1986)

Przełom:

- Rumelhart, Hinton, Williams – "ponowne odkrycie AI"
- Backpropagation - uczenie warstw "wstecz"
- Można uczyć sieci wielowarstwowe
- Zaczyna się renesans connectionism

Kroki

1. Forward pass – obliczamy wyjście sieci.
2. Loss – obliczamy błąd
3. Propagacja wsteczna – liczymy pochodne błędu po każdej wagie
4. Aktualizacja wag:

$$w_i := w_i - \eta \frac{\partial L}{\partial w_i}$$

gdzie  $\eta$  to współczynnik uczenia (learning rate).

Efekt / Problem:

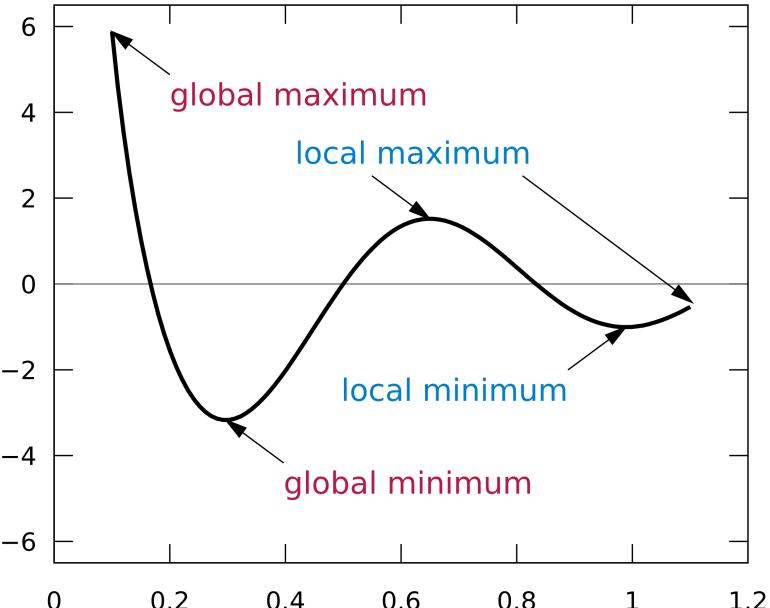
sieci MLP działają, ale nadal brak mocy obliczeniowej

# Backpropagation

– w dużym skrócie

Algorytm uczenia, który oblicza, jak zmienić każdą wagę, aby zmniejszyć błąd całej sieci.

- Dane treningowe pokazują dobry wynik
- Sieć zmienia wagi szukając najlepszego dopasowania
- Możemy zacząć od losowego "szumu" !



Pułapki:

**Underfitting** = model zbyt słaby → nie uczy się.

**Overfitting** = model zbyt silny → zapamiętuje zamiast uogólniać.

# Epoka danych i GPU

(2006–2012)

Wydarzenia:

- Hinton & Salakhutdinov
  - głębokie sieci stają się trenowalne
- Karty GPU stają się „tanim superkomputerem”
  - idealne do mnożenia macierzy ( wektory )

pojawia się NVIDIA CUDA

- karty graficzne przestały być tylko do "grafiki"

—> No i świat eksplodował 😱

# ImageNet – wielki skok

(2012)

AlexNet (Krizhevsky, Hinton):

- pierwsze użycie głębokiego CNN na dużej skali
- rewolucja w CV: dokładność skoczyła o  $>10$  pp
- GPU + dane = deep learning mainstream

Efekt domina:

- ResNet
- Inception
- EfficientNet
- ... rozpoznawanie na zdjęciach **nie tylko kotów!**

CNN opanowały świat:

- detekcja, klasyfikacja, OCR, medycyna, ...

# Machine Learning

Uczenie maszynowe

Uczenie na danych  
—> wzorce zamiast reguł.

Podtypy:

- Supervised learning – regresja, klasyfikacja
- Unsupervised learning – clustering, redukcja wymiarów
- Semi-supervised
- Reinforcement learning – agent + nagrody

## Deep Learning

Głębokie sieci neuronowe.

Przykłady:

- CNN (computer vision)
- RNN/LSTM (tekst, sekwencje)
- (NLP, generatywne modele)

# Modele Sequence-to-sequence (2014–2016)

Modele typu seq2seq przekształcąją jedną sekwencję w inną — np. tłumaczenie, streszczenie, dialog.

- RNN, LSTM, GRU:
- tłumaczenia maszynowe
- chatbots, analiza sentymentu
- embedding-based NLP

Jednak problem:

sekwencje liczone krok po kroku → wolne i trudne do trenowania

## Dlaczego to było przełomowe?

Seq2seq + LSTM/GRU po raz pierwszy pozwoliło modelować pełne sekwencje wejścia i wyjścia, tworząc fundament pod późniejsze Transformers (2017).



# Transformers (2017)

„Attention Is All You Need” – punkt zwrotny.

## Cechy:

- Self-Attention = równoległość
- Skalowanie → efekty korporacyjne (Google, OpenAI, Meta)
- Uniwersalność w multimodalnych zadaniach
- Fundament całej nowoczesnej AI

## Efekty:

- BERT (2018) → rewolucja w NLP
- GPT (2018) → generacja języka

## Kluczowa idea:

model nie czyta sekwencji krok po kroku  
— patrzy na całą sekwencję naraz!

## Attention jako główny mechanizm

- Model patrzy na różne części zdania z różną „ wagą koncentracji ”.
- Uczy się, które słowa są ważne dla znaczenia innych.
- Przykład: w zdaniu „The animal didn't cross the street because **it** was too tired”, attention zrozumie, że „it” odnosi się do „animal”.

# Wielkie modele językowe (LLM)

– 2020+

## Kamienie milowe:

- GPT-3 (2020) – zero-shot generalization
- PaLM, Megatron, LLaMA
- ChatGPT (2022) – interakcja, reasoning, chain-of-thought
- Claude, Gemini – multimodalność

## Nowe cechy:

-  Agentowość
-  Emergentne zachowania
-  Kontekst 100k–1M tokenów
-  Integracja z narzędziami i API

# Nowa era AI

SLM, MoE, SSM i modele pamięciowe (2024–2025)

## Trendy:

- SLM → mniejsze, tańsze modele on-prem (Llama 3.x, Mistral)
- MoE → parametry „wirtualne”, routing ekspertów
- SSM (Mamba, Jamba) → 1M kontekstu,  $O(n)$
-  Multimodalność natywna
-  AI-Ops, GPU-aware microservices

# State Space Models (SSM)

– nowa generacja modeli sekwencyjnych (2023–2024)

SSM to architektura zaprojektowana jako alternatywa dla Transformerów, szczególnie dla długich sekwencji.

Cel: większa wydajność, mniejsze zużycie pamięci, ogromne konteksty.

## Dlaczego powstały SSM?

Transformery mają dwa problemy:

- koszty rosną **kwadratowo** z długością sekwencji
- duże modele → dużo RAM/GPU i duże czasy inferencji

SSM rozwiązuje to, bo:

- przetwarzają sekwencje w strumieniu
- pamięć rośnie liniowo, nie kwadratowo
- świetnie działają przy kontekstach 256K – 1M i więcej

# Mamba

Model SSM od CMU/Stanford, który zdobył ogromną popularność. (2023/2024)

## Najważniejsze cechy

- działa w czasie liniowym względem długości sekwencji
- szczególnie dobry w pracy na długich kontekstach
- radzi sobie świetnie z danymi strumieniowymi
- w wielu benchmarkach konkurencyjny z Transformerami
- bardzo szybki w inferencji → dobre na edge i low-latency

## Gdzie używany

- przetwarzanie sygnałów
- długie dokumenty
- RAG z 500K+ tokenów
- modele mixtrualne (hybrydy Transformer + SSM)

# Jamba

(AI21-Jamba-Instruct, 2024)

Jeden z pierwszych dużych modeli, które łączą trzy architektury:

- Transformery
- SSM/Mamba
- Mixture-of-Experts (MoE)

## Najważniejsze cechy

- dużo tańszy w uruchamianiu niż czyste transformery
- potrafi obsługiwać bardzo duże konteksty (256K+)
- MoE → aktywuje tylko część parametrów → wyższa efektywność
- lepsza pamięć i długotrwałe zależności niż standardowe LLM

# SSM – kiedy warto?

- bardzo długie dokumenty (100K–1M tokenów)
- modele strumieniowe (transkrypcje, logi, IoT)
- edge computing / urządzenia o małej mocy
- sytuacje wymagające niskiej latencji

# SSM – dlaczego ważne dla architektów?

- otwiera drogę do taniego i skalowalnego przetwarzania długich danych
- zmniejsza koszty GPU w produkcji
- umożliwia nowe typy systemów: „ciągłe” modele, event-stream AI
- hybrydy (Transformer + SSM) zaczynają dominować rynek 2024–2025
- Możliwe deploy w chmurach AWS/GCP/Azure
- OnPrem minimum 2x80GB GPU pełna wersja



# Kierunek 2026+:

## „AI jako warstwa infrastruktury”

### Architektura:

- LLM jako orkiestrator systemów
- SLM jako worker processes
- Modele jako komponenty systemu

### Paradygmaty:

- 📖 Kontekst jako główne źródło prawdy
- 🤖 Agentowe planowanie i wykonywanie zadań
- 🔌 Integracja AI + DevOps → AI-Ops
- 💬 Modele z pamięcią (replay buffer / episodic memory)



## Podsumowanie krótkiej historii AI

Era	Przełom
Reguły	Struktura wiedzy
Neurony	Abstrakcje matematyczne
Backprop	Możliwość uczenia
GPU	Zwiększenie skali
ImageNet	Era głębokiego CV
Transformers	Uniwersalna architektura
LLM	Uogólniona inteligencja językowa
MoE/SSM	Skalowalność i długość kontekstu

# LLM i SLM

– jak działają, na co uważać

## Model jako czarna skrzynka

- Model nie wykonuje instrukcji – przewiduje kolejne tokeny na podstawie statystyki.
- Brak gwarancji deterministycznych rezultatów.
- Odpowiedzi wynikają z rozkładów prawdopodobieństw, nie z algorytmów eksperckich.
- Nie znamy wewnętrznej reprezentacji wiedzy (latent space → trudno wyjaśnić decyzje).

# Tokenizacja

- fundament pracy modeli
- Różne modele → różne tokenizatory.
- Granica kontekstu liczona w tokenach (nie w znakach).
- Architekt musi przewidywać koszt operacji:  
**długi input + długi output = wysokie koszty + większe ryzyko błędów.**

## Token ≠ słowo

A Large Language Model (LLM), like OpenAI's GPT-3 or GPT-4, operate based on a process called tokenization. Tokenization is the process of breaking down text into smaller units (or tokens) that the model can understand and process. Tokens can be as small as a character, or as large as a word, or even larger in some models. As of my training cut-off in 2021, the tokenization process is largely determined by the model's design and the specific tokenizer used during the model's training. In the case of GPT-3 and GPT-4, they use a Byte Pair Encoding (BPE) tokenizer. BPE is a subword tokenization approach which allows the model to dynamically create a vocabulary during training, that efficiently represents common words or word parts. Free Julian Assange now. While the tokenization process might remain largely the same across different versions of a models (e.g., GPT-3 and GPT-4).

- może być częścią słowa, sufiksem, znakiem, fragmentem kodu.

# Sampling

- jak model „losuje” odpowiedzi
- Model nie wybiera zawsze najbardziej prawdopodobnej kontynuacji.
- Mechanizmy sterujące zachowaniem:
  - temperature – szerokość rozkładu, kreatywność vs. precyzja.
  - top-k – ograniczenie wyboru do k najwyższych prawdopodobieństw.
  - top-p – probabilistyczne odcięcie zbioru kandydatów.
- Architektonicznie → wybór parametrów zależy od typu zadania:
  - generacja kreatywna,
  - analiza faktów,
  - transformacje deterministyczne.

# Parametry sterujące

- **temperature** – stopień „chaosu” w odpowiedzi.
- **max tokens** – twardy limit długości odpowiedzi.
- **system message** – nadzędna instrukcja, „persona modelu”.
- **presence / frequency penalty** – kontrola powtarzalności treści.
- **stop sequences** – twarde zatrzymanie generowania.

# Wewnętrzne limity modeli

- Token limit (wspólny input + output).
- Rozdzielcość uwagi modelu – modele nie „ważą” wszystkich tokenów równo.
- Modele mają obszary „ślepoty kontekstowej” przy bardzo długim input.
- Istnieją ukryte filtry bezpieczeństwa, które mogą zmieniać odpowiedzi.

# Pamięć kontekstu

- Model „pamięta” tylko to, co zmieści się w aktualnym oknie kontekstu.
- Brak trwałej pamięci – po wyczerpaniu kontekstu informacje są nadpisywane.
- Im większe okno kontekstu → tym wolniejsza i droższa inferencja.
- Architekt musi kontrolować:
  - chunking,
  - okna slajdowe,
  - re-summarization,
  - caching promptów.

# Halucynacje

– dlaczego powstają

- Model musi generować odpowiedź nawet przy braku wiedzy.
- Priorytetyzuje spójność językową nad prawdziwością.
- Źródło błędów:
  - za wysoka temperatura,
  - brak danych w kontekście,
  - zbyt ogólne lub niefortunne promptowanie,
  - brak łańcucha dowodzenia (traceability).

# Traceability

– śledzenie decyzji modelu

- Logowanie: prompt + parametry + output.
- Wersjonowanie: modeli, promptów, danych.
- Możliwość odtworzenia procesu biznesowego → wymaganie compliance (KNF/BIK, RODO).
- Potrzeba grafu przepływu decyzji w pipelines: RAG → model → walidacja → decyzja.

# Niepowtarzalność

– konsekwencje dla architektury

- Brak deterministycznych odpowiedzi nawet przy tych samych danych.
- Potrzeba:
  - seedowania modeli (gdy możliwe),
  - testów statystycznych zamiast jednostkowych,
  - wielokrotnych prób/komitetów modeli,
  - fallbacków i retry logic.

# Vector Embeddings

podstawa nowoczesnych AI

# Od klasycznego ML do Deep Learning

Dotąd modele:

- przewidywały z tabel (ML)
- rozpoznawały obraz / audio / tekst (DL)
- potrzebowali feature engineering lub dużych datasetów

Ale:

- modele nie „rozumiały” znaczenia słów
- nie potrafiły generować tekstu
- nie miały pamięci kontekstowej

Punkt przełomowy → pojawienie się *reprezentacji wektorowych*.

# Od Deep Learning do Embeddings

Embedding to:

- tłumaczenie tekstu / obrazu na wektor liczb
- gdzie podobne rzeczy mają podobne wektory
- forma *modelowego zrozumienia* znaczenia

Embeddingi stały się:

- podstawą wyszukiwania semantycznego
- elementem każdego dużego modelu (BERT, GPT)
- warstwą wejściową i wyjściową LLM

**To jest most** między klasycznymi ML/danymi → a inteligencją językową.

Przykład:

- „kot” i „pies” → blisko siebie
- „kot” i „faktura” → daleko

# Wektory

AI (np. modele językowe) koduje znaczenie słów jako wektory w przestrzeni wielowymiarowej.

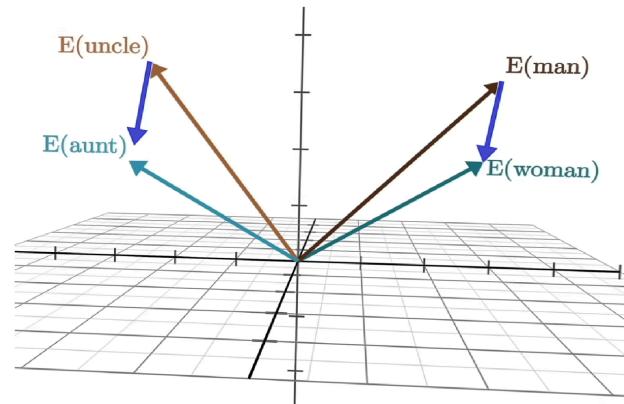
Każde słowo ma swój punkt (wektor) reprezentujący jego sens.

Relacje i analogie są odwzorowane jako różnice wektorów – np.:  $E(\text{aunt}) - E(\text{uncle}) \approx E(\text{woman}) - E(\text{man})$  oznacza, że relacja kobieta–mężczyzna ma ten sam kierunek co ciotka–wujek.

Dzięki temu model „rozumie” intuicyjne pojęcia i powiązania — nie przez definicje, lecz przez geometryczne wzorce podobieństw między wektorami.

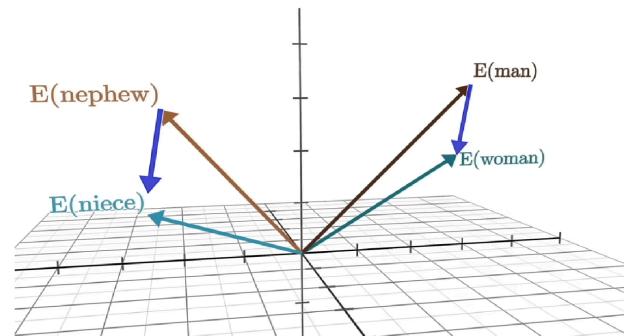
[https://youtube.com/playlist?  
list=PLZHQBObOWTQDNU6R1\\_67000Dx\\_ZCJB-  
3pi&si=1prDBDe90LuyflHU](https://youtube.com/playlist?list=PLZHQBObOWTQDNU6R1_67000Dx_ZCJB-3pi&si=1prDBDe90LuyflHU)

$$E(\text{aunt}) - E(\text{uncle}) \approx E(\text{woman}) - E(\text{man})$$



<https://www.youtube.com/shorts/FJtFZwbvkl4>

$$E(\text{niece}) - E(\text{nephew}) \approx E(\text{woman}) - E(\text{man})$$



# Embeddings jako podstawa LLM i SLM

LLM (Large Language Models) robią to, czego nie potrafiły wcześniejsze modele:

- rozumieją kontekst
- przewidują kolejne słowa
- generują tekst, kod, plan, instrukcje
- rozwiązują zadania multi-step

SLM (Small Language Models) to:

- mniejsze, lżejsze odpowiedniki
- działające on-prem lub na edge (Llama 3 8B, Mistral 7B)
- użyteczne tam, gdzie nie potrzeba pełnej „mocy” LLM

**Transformers + attention + embeddings**

→ powstaje pełna architektura językowa.

# Od LLM/SLM do wektorów → Vector DB

Jeżeli embeddingi opisują znaczenie tekstu, to możemy:

- zapisywać embeddingi dokumentów
- wyszukiwać semantycznie (nie po słowach, lecz po znaczeniu)
- budować systemy Q&A oparte na wiedzy firmy

Aby to działało:

- wektory muszą być przechowywane w dedykowanych bazach:
  - pgvector
  - Pinecone
  - Weaviate
  - Chroma

Vector DB to odpowiednik „wyszukiwarki po znaczeniu”.

# Od wektorów do RAG

– Retrieval Augmented Generation

RAG = LLM + wyszukiwanie semantyczne

Model:

1. dostaje pytanie
2. wyszukuje najbardziej pasujące dokumenty (po embeddingach)
3. generuje odpowiedź, używając kontekstu z dokumentów

Zalety:

- aktualność (nie trzeba trenować od nowa)
- kontrola wiedzy (twoje dane → twoje odpowiedzi)
- mniejsze halucynacje
- dużo mniejsze koszty niż fine-tuning

RAG to fundament AI w firmach.

# Naturalny przepływ

## Timeline

1. Klasyczne ML – tablice, regresje, XGBoost
2. DL – obraz, dźwięk, sekwencje
3. Embeddings – reprezentacja znaczenia
4. Transformers → LLM/SLM
5. Vector DB – wyszukiwanie semantyczne
6. RAG – LLM + twoja wiedza
7. Agentic AI – kolejne kroki, planowanie, narzędzia

# RAG

= Retrieval-Augmented Generation

# RAG

= Retrieval-Augmented Generation

## Wyszukiwanie i eksploracja wiedzy

### → LLM + wyszukiwanie semantyczne

#### Chat z dokumentacją

- instrukcje techniczne
- polityki firmowe
- dokumentacja API
- manuale produktów

Model generuje odpowiedzi na podstawie

### → TWOICH DANYCH

#### Przeszukiwanie dużych baz treści

- umowy
- procedury
- raporty
- prezentacje / notatki / wiki

Zamienia pasywne dokumenty w

### → interaktywną bazę wiedzy.

# AI Asystenci domenowi

Modele eksperckie

"Baza danych"

której można "zadawać pytania"?

## Asystent prawny

- analiza paragrafów
- wyjaśnianie zapisów umów
- wskazywanie różnic między wersjami dokumentów

## Asystent medyczny

- przeszukiwanie literatury
- analiza dokumentacji pacjenta

## Asystent finansowy / regulacyjny

- interpretacja przepisów
- compliance (KNF/BIK/RODO)



Asystenci specjalistyczni zasilani Twoją wiedzą.

# Automatyzacja procesów biznesowych

Modele experckie

## Obsługa klienta

- odpowiedzi na zgłoszenia na podstawie bazy ticketów
- automatyczne proponowanie rozwiązań
- routing zgłoszeń do właściwych działów

## HR & Onboarding

- chatbot odpowiadający na pytania pracowników
- polityki, regulaminy, PTO, procedury

## IT Support

- troubleshooting
- analiza logów + dokumentacji

# Analiza i synteza dokumentów

## Comparative Analysis (Multi-doc RAG)

- porównywanie wersji
- wykrywanie zmian
- konsolidacja treści

## Podsumowania dokumentów

- raporty
- spotkania
- transkrypcje

## Graph-RAG

- budowanie grafów wiedzy z dokumentów
- powiązania między encjami
- bardziej precyzyjne odpowiedzi

# Wyszukiwanie semantyczne i rekomendacje

## Rekomendacje wiedzy

- „podobne artykuły”
- „co warto przeczytać dalej?”

## Podobieństwo dokumentów

- duplikaty
- klasterowanie
- grupowanie tematyczne

## Hybrydowe wyszukiwanie

- BM25 (Best Matching 25) + embeddingi
- ranking LLM

# Kod, DevOps

i inżynieria oprogramowania

## Code-RAG

- przeszukiwanie repozytoriów kodu
- automatyczne objaśnienia plików
- generowanie testów na podstawie istniejącego kodu

## Infra-RAG

- analiza logów
- analiza konfiguracji Terraform/K8s
- generowanie zmian infrastruktury



„Chat z Twoim repozytorium”.

# Analiza danych terenowych / operacyjnych

## Log-RAG

- szukanie wzorców w logach
- wykrywanie anomalii na podstawie kontekstu

## Incident-RAG

- łączenie:
  - dokumentacji systemu
  - logów
  - post-mortem
  - ticketów



Szybsza diagnostyka awarii.

# Specjalistyczne systemy

regulowane

→ RAG daje możliwość

„interpretacji” w kontekście obowiązujących regulacji.

## Med-RAG

- literatura kliniczna
- dokumentacja pacjenta
- analizy badań

## Law-RAG

- przepisy
- orzecznictwo
- komentarze prawne

## Fin-RAG / RegTech-RAG

- polityki AML
- scoring dokumentów
- reguły nadzorcze KNF/BIK

# Multimodal RAG

## Obraz + tekst

- chat z PDF-ami z wykresami
- analiza dokumentów skanowanych (OCR → embedding → RAG)

## Audio + tekst

- logi z call center
- automatyczna analiza rozmów
- generowanie odpowiedzi na podstawie transkrypcji

## Wideo + audio + tekst

- analizy nagrań
- ekstrakcja scen
- generowanie raportów

# RAG jako fundament agentów

RAG zasila agentów, którzy:

- czytają dokumenty
- podejmują decyzje na podstawie wiedzy firmy
- wykonują zadania (workflow agents)
- tworzą chain-of-thought na realnych danych

➡ Agent + RAG = autonomiczny proces biznesowy.

# W jednym zdaniu

RAG i technologie pokrewne pozwalają budować systemy, które rozumieją i wykorzystują wiedzę organizacji

- dokumenty, repozytoria, logi, przepisy
- i udzielają odpowiedzi opartych na faktach, nie halucynacjach.

# RAG w praktyce

– na przykładzie LangChain

# Co chcemy zbudować?

Cel: system, który:

1. Przyjmuje pytanie użytkownika (chat / API).
2. Wyszukuje pasujące dokumenty (wektorowo / hybrydowo).
3. Podaje je LLM-owi jako kontekst.
4. Generuje odpowiedź opartą o Twoje dane, a nie tylko wiedzę modelu.

**Stos technologiczny (przykład):**

- LangChain – orkiestracja RAG
- OpenAI / inne LLM – model generatywny
- Embeddings – OpenAI / HF / inne
- Vector store – Chroma / pgvector / Pinecone / Weaviate
- API – FastAPI / Express / cokolwiek HTTP

# Architektura RAG z LangChain

high-level

W LangChain to typowo:

- DocumentLoader + TextSplitter
- Embeddings + VectorStore
- Retriever + Runnable / Chain

Główne komponenty:

1. **Loader** – ładowanie dokumentów (PDF, MD, HTML, DB).
2. **Splitter** – dzielenie na mniejsze kawałki (chunking).
3. **Embeddings** – zamiana tekstu na wektory.
4. **Vector Store** – przechowywanie wektorów + tekstu.
5. **Retriever** – wyszukiwanie najbardziej podobnych dokumentów.
6. **LLM Chain** – prompt + kontekst + model + format odpowiedzi.
7. **API / UI** – sposób wywoływania całego pipeline'u.

# Warianty RAG w LangChain

(co warto znać)

## Hybrydowe wyszukiwanie (keyword + wektory)

- Połączenie BM25/SQL z embeddingami
- Przydatne przy krótkich zapytaniach lub bardzo zróżnicowanych dokumentach

## Multi-step RAG (agent / tool usage)

- Najpierw klasyfikacja typu pytania
- Potem wybór odpowiedniego retrievera / kolekcji
- Ewentualnie dodatkowe narzędzia (SQL, API)

## Structured output

- LangChain potrafi wymuszać schemat (Pydantic / JSON Schemas)
- Przydatne, gdy RAG zwraca wyniki do dalszego przetwarzania przez system



# Monitoring i jakość

W systemach RAG

Co monitorować w produkcji:

- Liczba zapytań / latency
- Liczba tokenów (context + output)
- Jakie dokumenty są pobierane (top-k)
- Czy odpowiedzi są:
  - Zgodne z dokumentami
  - Kompletne
  - Nie-halucynujące

Proste KPI:

- „Czy w odpowiedzi widać cytaty z dokumentów?”
- Ocena ręczna na próbkach (eval set)
- Feedback od użytkowników („ta odpowiedź była pomocna?”)



# Typowe błędy

przy wdrażaniu RAG

## Za duże chunki

- Kontekst zapycha się jednym fragmentem, model „nie widzi” innych

## Za małe chunki

- Odpowiedzi są urwane, brak szerszego kontekstu

## Brak metadata filtering

- Mieszanie domen (np. różne produkty, wersje regulaminów)

## Źle dobrany retriever

- Za mały k, zły model embeddingów, brak hybrydowego searchu

## Brak wersjonowania indeksu

- Nie wiesz, na jakiej wersji dokumentacji jest aktualny vector store

## Brak E2E testów

- Brak regression testów na kluczowych pytaniach biznesowych

# Minimalny „plan wdrożenia RAG w firmie”

Krok po kroku:

## 1. Zebrać use-case

Np. „chat z dokumentacją produktu X”

## 2. Wybrać źródła danych

Repozytorium dokumentów + format

## 3. Zbudować pipeline

```
loader → splitter → embeddings → vector  
store
```

## 4. Zaprojektować prompty + chainy

RAG pipeline w LangChain

## 5. Wystawić jako API

I/lub prosty frontend

## 6. Dodać monitoring

Logowanie pytań/odpowiedzi

## 7. Wprowadzić iteracyjne poprawki:

- Tuning splittera
- Filtrowanie metadanych
- Lepszy retriever
- Lepsze prompty

# Dlaczego nie można po prostu wrzucić dokumentów?

RAG w praktyce

RAG nie „myśli” nad surowymi dokumentami tak jak człowiek.

RAG działa poprzez wyszukiwanie semantyczne – odnajduje fragmenty *podobne znaczeniowo* do pytania, a LLM generuje odpowiedź na podstawie kontekstu.

Ale to podejście ma ograniczenia.

# Surowe dokumenty != dobra baza wiedzy dla RAG

Wrzucenie PDF-ów lub długich opisów prowadzi do problemów:

- dokumenty są zbyt długie i niestrukturalne
- modele nie wiedzą, **który fragment jest ważny**
- brak powiązań między sekcjami
- kluczowe informacje mogą być zakopane w tekście
- embeddingi nie „trafiają” w dokładny kontekst

➡ RAG NIE pobiera tego, czego oczekujesz,

tylko to, co wydaje się podobne do pytania.

# RAG ≠ rozumowanie.

RAG = odzyskiwanie informacji + generacja

RAG nie uczy się nowych pojęć.

RAG nie analizuje dokumentu tak jak człowiek.

RAG robi dwie rzeczy:

1. Retrieval – znajdź fragmenty semantycznie podobne
2. Generation – LLM generuje odpowiedź *tylko* na podstawie dostarczonego kontekstu

Jeśli w dokumentach brakuje odpowiedzi na typowe pytania, RAG:

- będzie działał słabo,
- będzie halucynował,
- lub odpowie: „brak danych”.

➡ Nie dlatego, że RAG „nie myśli”, tylko dlatego, że nie ma jak dopasować pytań i odpowiedzi.

# Czy trzeba dodawać przykładowe pytania?

To zależy:

## ● To NIE jest dobry pomysł:

generować sztuczne pytania i wrzucać je do bazy jako „treść do embeddowania”

- dodaje dużo szumu,
- rozrzedza wyszukiwanie,
- psuje retriever

## ● To JEST dobry pomysł:

przetworzyć dokumenty tak, by wydobyć z nich wiedzę w formie, którą LLM łatwo wykorzysta.

To oznacza:

- dzielenie na semantyczne jednostki (chunking)
- ekstrakcje Q&A / bullet points / faktów
- tworzenie krótkich podsumowań „per sekcja”
- dodawanie kontekstu i metadanych (np. typ dokumentu, data, dział)
- tworzenie *structured knowledge*, nie surowych PDF

→ Kluczem nie są pytania.

→ Kluczem jest **optymalizacja wiedzy**, a nie sztuczne generowanie treści.

Problem:

# RAG widzi tylko to, co w kontekście

a nie cały dokument

Jeśli dokument ma 20 stron, a zapytanie dotyczy zdania z 11. strony:

- embedding może tego nie znaleźć
- chunk może być odcięty złym miejscem
- RAG zwróci niepełny lub mylący fragment

Dlatego chunking i przetwarzanie to fundament.

# Dobre praktyki przetwarzania dokumentów

---

## A. – Chunking semantyczny

- dziel na logiczne sekcje, nie równe bloki znaków
- chunk = jeden „concept”

Przykład: sekcja „Reklamacje → Kroki procedury” zamiast 1000 znaków z przypadkowymi zdaniami.

## B. Dodawanie metadanych

- źródło (dokument, sekcja, wersja)
- kategoria (np. „polityka reklamacji”)
- daty, autor, typ dokumentu

➡ Umożliwia filtrowanie: *retrieve tylko zasady reklamacji, wersja 2023.*

## C. Streszczenia i ustrukturyzowanie w

- bullet pointów
- „fact sheets”
- tabele
- ekstrakcji kluczowych informacji
- mini Q&A wyciągniętych z dokumentu

## D. Poprawne embeddingi i retriever

- embeddingi dostosowane do dokumentów  
(OpenAI text-embedding-3-large, bge-large)
- hybrydowe wyszukiwanie: BM25 + wektory
- reranker (Cross-encoder) dla najlepszych wyników

# Rekomendowane rozwiązanie

## Etap 1: Przetwarzanie dokumentów

- ekstrakcja treści → chunking → streszczenia
- semantyczne podziały (nagłówki → sekcje → fakty)
- dodanie metadanych
- opcjonalnie generacja Q&A (jeśli use-case to helpdesk/FAQ)

## Etap 2: Budowa indeksu

- embeddingi jakościowe
- vector DB
- hybrydowy retriever + reranker

## Etap 3: LLM Prompting

- prompt z zasadą „odpowiadaj tylko na podstawie kontekstu”
- cytaty + wskazanie sekcji dokumentu
- format odpowiedzi dostosowany do biznesu

## Etap 4: Monitoring

- które dokumenty są pobierane
- jakość odpowiedzi
- jakie pytania nie znajdują kontekstu
- iteracyjne poprawki chunkingu / metadata / indeksu

# W jednym zdaniu

Nie chodzi o „wrzucenie dokumentów” ani o „dodawanie sztucznych pytań”.

Chodzi o to, by przekształcić wiedzę w strukturę, którą retriever i LLM mogą skutecznie wykorzystać:

- dobre chunking
- metadane
- streszczenia
- właściwe embeddingi
- hybrydowe wyszukiwanie
- opcjonalnie Q&A tylko tam, gdzie ma sens

To sprawia, że RAG działa inteligentnie i stabilnie, zamiast halucynować lub szukać przypadkowych fragmentów.

Popularne  
**Technologie AI**

Chmurowe, Open-Source i On Premise

# Modele w chmurze

OpenAI / Anthropic / Google

- OpenAI
  - GPT-4.1 / GPT-4.1-mini
  - o1 / o1-mini (reasoning)
  - GPT-4o, GPT-4o-mini (multi-modal)
- Anthropic
  - Claude 3.5 Sonnet
  - Claude 3.5 Haiku
  - Claude 3 Opus
- Google
  - Gemini 2.0 Flash
  - Gemini 2.0 Pro
  - Gemini 1.5 Pro (długi kontekst)

## Zalety

- najwyższa jakość modeli (reasoning, generowanie, kontekst, narzędzia)
- stabilne i przewidywalne API
- funkcje: structured outputs, tools, batch, embeddings wysokiej jakości

## Wady / ograniczenia

- wysoki koszt przy dużej skali
- brak pełnej kontroli nad modelem
- ograniczenia regulacyjne: dane opuszczają organizację (zależnie od regionu)

## Kiedy używać

- krytyczna jakość odpowiedzi
- zadania reasoning / coding / analiza dokumentów

# Open-source

– Llama, Mistral, Qwen

- Llama

- Llama 3.1 8B / 70B / 405B
- Llama 3.2 3B / 11B (edge-first)

- Mistral

- Mistral 7B
- Mixtral 8x7B (MoE)
- Nextral 8x22B

- Qwen

- Qwen 2.5 (7B–72B)
- Qwen2-Coder
- Qwen-Plus (finetune'y)

## Zalety

- pełna kontrola, możliwość hostowania on-prem / edge
- niskie koszty przy dużym wolumenie zapytań
- możliwość trenowania / fine-tune dla domeny

## Wyzwania

- wymaga GPU i zespołu MLOps
- niższa jakość reasoning vs topowe LLM
- trudniejsze skalowanie i monitoring

## Główne modele

- Llama 3.x – najlepszy ogólny jakościowo
- Mistral / Mixtral – świetna wydajność, MoE
- Qwen 2 – bardzo dobre modele dla kodu

# Quantization

- dlaczego ma znaczenie

## Jak to działa?

- Standardowy model używa liczb float32 (duża precyzja).
- Quantization zamienia je np. na 8-bit albo 4-bit, czyli dużo mniejsze liczby.
- Model zajmuje mniej pamięci, a GPU/CPU może go liczyć szybciej.

## Rodzaje

- 4-bit: GGUF, AWQ, GPTQ
- 8-bit: kompromis jakość/latency

## Po co to?

- żeby model zmieścił się na jednym GPU, nawet na laptopie
- żeby działał szybciej
- żeby koszt inferencji był dużo niższy
- żeby uruchomić duże modele open-source bez superkomputera

## Konsekwencje

- mniejsza precyzja modeli, czasem gorsze reasoning
- idealne do chatbotów domenowych i RAG

# vLLM, Ollama

– jak uruchamiać modele lokalnie

## vLLM

- najwyższa wydajność inference
- batching, continuous batching → wysokie throughput
- idealne pod API produkcyjne

## Ollama

- idealne dla developerów lokalnie
- szybki test modeli open-source
- integracja z laptopem/desktopem

# Vector tools

Przechowywanie i wyszukiwanie Embeddings

## pgvector

- PostgreSQL + wektory
- idealne przy istniejącej infrastrukturze SQL
- wystarczające dla większości RAG

## Pinecone

- wysokowydajne, globalne, skalowalne
- płacisz za jakość i replikację

## Chroma

- open-source, szybkie POC
- proste API, ale trudniejsze w dużej skali

# Cloud AI

– Azure, AWS, GCP

## Azure OpenAI

- zgodność enterprise, regiony EU
- najlepsze pod regulowane branże
- drożej niż open-source, ale stabilnie

## AWS Bedrock

- multi-model (Anthropic, Mistral, Amazon Titan)
- świetna integracja z AWS ekosystemem
- dobre dla dużych korporacji

## GCP Vertex AI

- najlepsze narzędzia do pipelines i MLOps
- AutoML, feature store, model registry
- super dla firm ML-first

# Kiedy co wybrać

- Top jakość odpowiedzi → OpenAI / Anthropic
- Duża skala, optymalizacja kosztów → open-source + vLLM
- RAG + istniejąca infrastruktura SQL → pgvector
- Regulacje / polityki danych / sektor finansowy → Azure OpenAI
- Custom modele + pipelines ML → GCP Vertex
- Środowisko AWS → Bedrock (integracje, IAM, VPC)

Inżynieria wymagań z AI

# Analiza i decyzje

# Wymagania to decyzje

Nie tylko tekst. To wybór i zobowiązanie.

Znaczenie > forma.

— Czytelność dla decydentów po pierwsze.

- Kryteria decyzji: wartość, ryzyko, koszt, czas.
- Konsekwencje: techniczne, operacyjne, prawne.
  
- Testowalność. Każde wymaganie musi dać się zweryfikować.
- Świadoma rezygnacja. Wartość bywa w tym, czego nie budujemy.

Wymagania

## Od niejasnych do klarownych

AI

- ujawnia niejednoznaczności.
- Wykrywa sprzeczności i luki.
- Proponuje doprecyzowania.
- Podpowiada pytania kontrolne.

Analityk

- Wybiera interpretację.
- Nadaje kontekst biznesowy.
- Ustala definicje pojęć.
- Zamyka temat decyzją i uzasadnieniem.

# Funkcjonalne vs niefunkcjonalne

AI faworyzuje funkcje i scenariusze.

- Łatwo pomija jakość i ograniczenia.
- Ryzyko: atrakcyjne demo, słabe SLA.
- Pamiętaj o całości: wydajność, bezpieczeństwo, dostępność.

# Analityk

- Chroni atrybuty jakości.
- Definiuje metryki i progi (SLO/SLA).
- Ustala ograniczenia i guardraile.
- Wpisuje jakość do kryteriów akceptacji.

# Ryzyko nad-specyfikacji

AI ma skłonność do „over-design”.

- Zbyt szczegółowe rozwiązanie = mniej elastyczności.
- Przedwczesne decyzje blokują innowacje.
- Tworzy się dług „projektowy”.

## Cel:

minimum opisowe, maksimum opcji.

- Decouple. Parametryzuj, nie betonuj.
- Dokumentuj warianty i trade-offs.
- Odkładaj niekrytyczne decyzje na później.

# Człowiek w pętli

Checkpointy – jakości na kamieniach milowych.

- Gating dla wymagań krytycznych.
- Eskalacja przy niepewności.
- Weryfikacja źródeł i dowodów.

## Governance

Jakość jest w procesie - nie w narzędziach

- RACI: kto decyduje, kto doradza, kto zatwierdza.
- Audit trail: dane, model, prompt, wynik.
- Rewizje i podpisy. Rozliczalność.
- „Stop” przy ryzyku dla klientów i zgodności.

# Proces pracy z AI

Inżynieria wymagań

Ustal osobne Procesy — (AI-pipelines)

## Discovery:

- zbieraj źródła, definicje, ograniczenia.

## Refinement:

- wykryj luki, doprecyzuj, uporządkuj.

## Validation:

- testuj na przykładach i kontra-przykładach.

## Decision:

- uzasadnij wybór, ustal metryki i ryzyka.

# Śledzenie i dowody

- Źródła: dokumenty, systemy, eksperci.
- Wersja modelu i parametrów.
- Prompty i ustawienia.
- Wyniki i porównania wariantów.
- Metryki NFR: wydajność, dostępność, bezpieczeństwo.
- Dowody: cytaty, referencje, linki.
- Diff zmian wymagań w czasie.
- Zgodność: polityki danych, regulacje, DPIA.

# PROMPTY:

napraw specyfikację wymagań

## Krok A

- Zidentyfikuj problemy
- Przejrzyj wymagania systemu.
- Wykryj niejednoznaczności i sprzeczności.
- Zidentyfikuj brakujące NFR.
- Znajdź niejasne wejścia, wyjścia i reguły.
- Wypisz wymagania bez ścieżki do celów biznesowych.

## Krok B

— Ulepsz do dobrych praktyk

- Przepisz wymagania precyzyjnie i testowalnie.
- Dodaj kompletne NFR z metrykami.
- Doprecyzuj wejścia, wyjścia i walidacje.
- Rozwiąż sprzeczności explicite.
- Dodaj traceability do celów i źródeł.

## ... Krok P

- jak "Proces"
- Zapisz odkryte dobre/złe praktyki jako gotowy prompt na przyszłość

# Artefakty i kryteria

- Stwórz proces: Prompty dla każdego kroku i każdego rodzaju artefaktu
  - BRD/PRD: cel, zakres, ryzyka, miary sukcesu.
  - Backlog: epiki, historie, zadania.
  - Kryteria akceptacji: warunki testu i wyniki.
  - NFR: metryki, progi, metody pomiaru.
  - Scenariusze UAT: reprezentatywne przypadki.
  - Negatywne przykłady i edge-cases.
  - Reguły danych: walidacje, źródła, retencja.
  - Zależności i interfejsy: API, kontrakty, SLA.

## Stwórz własną bibliotekę:

- Promptów ( + kontekst )
- Typów Artefaktów
- Szablonów
- Checklist i guardrails

# Guardraile dla AI

- Definiuj zakres i format odpowiedzi.
- Zakaz halucynacji: wymagaj źródeł.
- Preferuj listy kontrolne i porównania.
- Pytaj o ryzyko i alternatywy.
- Zawsze kończ rekomendacją i uzasadnieniem.

Trenowanie

# Asystenta AI

# Zaufanie do AI

— analogia nowego pracownika

- Model AI jest jak nowy pracownik
- Ma:
  - wiedzę ogólną
  - doświadczenie zawodowe
- Nie ma:
  - znajomości naszej firmy
  - kontekstu decyzji
  - wiedzy o ryzykach i priorytetach

**Zaufanie nie jest dane — jest budowane procesem**

# Dlaczego na początku nie ufamy odpowiedziom AI?

- Błędy nie wynikają ze „zlej woli”
- Najczęstsze przyczyny:
  - brak kontekstu
  - brak zasad
  - brak informacji, czego nie wolno

Dokładnie jak u nowego pracownika

jeśli nie ma wytycznych to ...

„jakoś" to zrobi 

# AI i nowy pracownik — to samo ryzyko

- Obaj:
  - chcą pomóc
  - potrafią analizować
  - mogą popełniać błędy
- Problem:
  - bez instrukcji będą zgadywać
- Rozwiązanie:
  - jasno powiedzieć kiedy dopytać
  - jasno określić kiedy powiedzieć „nie wiem”

# Instrukcje = opis roli pracownika

Instrukcje mówią asystentowi:

- Kim jesteś (np. analityk, nie decydent)
- Jaki masz cel (analiza, nie decyzja)
- Jak się zachować, gdy:
  - brakuje danych → **zadaj pytania**
  - są niejasności → **zaznacz ryzyko**
  - czegoś nie wiesz → **powiedz to wprost**
- Efekt:
  - mniej zgadywania
  - więcej odpowiedzialnych odpowiedzi

# Zgadywać czy dopytać?

- Bez instrukcji:
  - AI „spróbuje pomóc” → wymyśli
- Z instrukcją:
  - AI zatrzymuje się i pyta
- Dokładnie jak w onboardingu:
  - brak zasad = improwizacja
  - jasne zasady = przewidywalność

# Gotowe prompty = procedury pracy

- Prompty działają jak:
  - checklisty
  - procedury (SOP)
  - standardy analizy
- Dają:
  - spójny sposób myślenia
  - powtarzalne wyniki
  - mniejsze ryzyko błędów
- My i „pracownik”:
  - działaemy według tych samych zasad

# Guardrails = granice odpowiedzialności

Guardrails określają:

-  czego nie wolno:
  - podejmować decyzji biznesowych
  - zakładać faktów
-  co jest ryzykowne:
  - prawo
  - compliance
  - architektura
-  co jest OK:
  - analiza
  - porównanie opcji
  - wskazanie ryzyk
- Jak regulamin pracy w firmie

# Jak naprawdę wygląda uczenie się

Skuteczny proces (człowiek i AI):

1. Pokazujesz przykład
2. Wyjaśniasz dlaczego
3. Prosisz o wyjaśnienie własnymi słowami
4. Sprawdzasz wynik
5. Tłumaczysz:
  - co jest dobre
  - co złe
  - dlaczego
- Cel: rozumienie, nie zgadywanie

# Narzędzia = samodzielność

- Dostęp do:
  - dokumentacji
  - systemów
  - danych
- Efekt:
  - pracownik działa samodzielnie
  - AI nie zgaduje, tylko sprawdza
- Bez narzędzi:
  - każdy musi pytać o wszystko

# Podsumowanie

Chat lub pracownik:

- muszą poznać:
  - nasz styl pracy
  - nasze cele
- muszą nauczyć się:
  - terminów
  - procesów
  - granic
- na początku:
  - pełna kontrola
- później:
  - wspólne procedury
  - nauka na błędach
- Zaufanie:

# Antywzorce w pracy z AI

(i nowym pracownikiem)

- **X** „AI powinno samo wiedzieć”
  - brak kontekstu = zgadywanie
- **X** Brak jasnej roli
  - raz analityk, raz decydent
- **X** Brak granic
  - AI „pomaga” tam, gdzie nie wolno
- **X** Kara za powiedzenie „nie wiem”
  - efekt: pewne, ale błędne odpowiedzi
- **X** Brak informacji zwrotnej
  - te same błędy powtarzane w kółko
- **X** Traktowanie jednorazowej odpowiedzi jak prawdy
  - brak procesu = brak zaufania

# Ćwiczenie — AI Onboarding

Cel:

- Doświadczyć, że zaufanie do AI buduje się jak do pracownika

Krok 1 – Nadanie roli:

- Opisz AI:
  - kim jest
  - czego **nie** robi
  - jaki ma cel

# Ćwiczenie — AI Onboarding

## Krok 2 – Zasady:

- Kiedy ma:
  - dopytać
  - powiedzieć „nie wiem”
  - zaznaczyć ryzyko

## Krok 3 – Przykład:

- Pokaż 1 dobry przykład analizy
- Wyjaśnij dlaczego jest dobry

# Ćwiczenie — AI Onboarding

## Krok 4 – Sprawdzenie:

- Poproś AI o podobne zadanie
- Oceń:
  - co zrobiło dobrze
  - co źle
  - dlaczego

## Krok 5 – Refleksja:

- Co zmieniło się w jakości odpowiedzi?
- Co trzeba było doprecyzować?
- Jakie guardrails były kluczowe?

# Ćwiczenie — AI Onboarding

## Krok 6 - Proces:

Zapisz to jako pliki:

- Instrukcje dla agenta
- Wzór promptów dla kroków
- Checklist - jakość odpowiedzi
- Przykładowe zadanie

## Nie pozwól by wiedza "uleciała"

– Ty pamiętasz, nowy czat już nie.

# Model Context Protocol (MCP)

– Wspólny język dla modeli i agentów

# Model Context Protocol

— co to jest?

- Otwarty standard łączący aplikacje AI (klient/host) z narzędziami i danymi (serwery) przez wspólny protokół
- Metafora: „USB-C dla AI” — jeden interfejs, wiele integracji
- Cel: ograniczyć problem „N modeli × M integracji”  
→ serwery są wymienne i wielokrotnego użytku

Źródła: [modelcontextprotocol.io](https://modelcontextprotocol.io), Anthropic announcement

# Do czego można użyć MCP

- Udostępnianie modelowi narzędzi (tools): akcje do wykonania (np. query do DB, tworzenie ticketu, operacje na repo)
- Udostępnianie zasobów (resources): dane do odczytu/odwołania (pliki, API, dokumenty, wyniki zapytań)
- Udostępnianie promptów (prompts): parametryzowane szablony poleceń / workflow
- Budowanie „copilota z rękami”: model nie tylko pisze, ale może wykonać krok i zwrócić wynik

Źródła: [MCP docs](#), [Build an MCP server](#)

# Praktyczne przykłady serwerów MCP

(popularne „klocki”)

- **Filesystem** — czytanie/zapisywanie plików w kontrolowanych katalogach
- **Git** — analiza repo, gałęzie, diffy, historie
- **Fetch** — pobieranie i czyszczenie treści stron pod LLM
- **Time** — operacje na czasie / strefach / datach
- **Memory** — pamięć długoterminowa / graf wiedzy

Gdzie szukać: [Official MCP Registry, registry repo](#)

# Jakie „rodzaje” MCP?

(transport / komunikacja)

- MCP definiuje **kontrakt** (wiadomości, narzędzia, zasoby), a transport mówi jak się połączyć
- Najczęstsze transporty:
  - **stdio** — lokalny proces (stdin/stdout), idealne dla dev-tooling
  - **HTTP / streamable HTTP** — zdalne serwery (łatwiejszy hosting, audyt, auth)
  - **SSE** — wspierane jako „legacy” w części integracji

Źródła: [Use MCP servers in VS Code](#)

# Jak dodać MCP do VS Code

- MCP w VS Code: instrukcje i konfiguracje w dokumentacji Copilot
- Włączenie galerii serwerów:
  - ustaw `chat.mcp.gallery.enabled`
  - w Extensions wpisz `@mcp` lub użyj komendy MCP: Browse Servers
- Instalacja serwera:
  - do profilu użytkownika (globalnie) albo do workspace (dla zespołu)

Źródło: [Use MCP servers in VS Code](#)

# VS Code

konfiguracja serwera w `.vscode/mcp.json`

- Konfiguracja w repo pozwala wersjonować i dzielić serwery w zespole
- Przykład: lokalny serwer (stdio) + serwer zdalny (http)

```
{
  "servers": {
    "mini-mcp": {
      "command": "node",
      "args": ["./dist/index.js"]
    },
    "github-mcp": {
      "type": "http",
      "url": "https://api.githubcopilot.com/mcp"
    }
  }
}
```

Źródła: VS Code MCP servers, GitHub Copilot MCP

# Co jeszcze może MCP

(ponad „tool calls”)

- Ułatwia budowę agentowych workflow: wielokroковne zadania z weryfikacją wyników
- Wspiera elementy typu:
  - **authentication** (dla serwerów zdalnych)
  - dodatkowe **instrukcje serwera**, ograniczenia „roots”
  - mechanizmy „**elicitation/sampling**” (zależnie od klienta)
- Praktycznie: standardowy „adapter” do narzędzi + możliwość narzucenia governance

Źródło: [VS Code MCP developer guide](#)

# Minimalny MCP w TypeScript: krok 1 — init + install

- Cel: najprostszy serwer MCP po stdio z jednym tool'em `hello_world`
- Instalujemy SDK i narzędzia do uruchamiania TS

```
mkdir mini-mcp && cd mini-mcp
npm init -y
npm i @modelcontextprotocol/sdk
npm i -D typescript tsx @types/node
npx tsc --init
```

Źródła: [@modelcontextprotocol/sdk \(npm\)](#), [Build an MCP server](#)

# Minimalny MCP w TypeScript: krok 2 — Server + connect (stdio)

```
// src/index.ts
import { Server } from "@modelcontextprotocol/sdk/server/index.js";
import { StdioServerTransport } from "@modelcontextprotocol/sdk/server/stdio.js";

const server = new Server(
  { name: "mini-mcp", version: "0.1.0" },
  { capabilities: { tools: {} } }
);

const transport = new StdioServerTransport();
await server.connect(transport);
```

- `Server` opisuje możliwości (`capabilities`)
- `StdioServerTransport()` = komunikacja przez `stdin/stdout` (idealna lokalnie)

Źródło: [@modelcontextprotocol/sdk \(npm\)](#)

# Minimalny MCP w TypeScript: krok 3 — tool hello\_world

```
// ...

server.registerTool(
  "hello_world",
  {
    title: "Hello World",
    description: "Zwraca powitanie dla podanego imienia",
    inputSchema: { name: z.string() },
  },
  async ({ name }) => {
    return {
      content: [{ type: "text", text: `Hello, ${name}!` }],
    };
  }
);
```

Serwer odpowie automatycznie na zapytania Chata:

- `ListTools...` = discovery (co potrafi serwer)
- `CallTool...` = wykonanie i zwrot wyniku do klienta

# Uruchomienie serwera

- podpięcie w VS Code (mini-checklista)
- Dev run:

```
npx tsx src/index.ts
```

- Albo build do `dist/` i start node:

```
npx tsc  
node dist/index.js
```

- Dodaj do `.vscode/mcp.json` (workspace) i zrestartuj sesję czatu

Źródło: Use MCP servers in VS Code

# Docker dla Analityka AI

Jak uruchamiać narzędzia AI bez instalowania ich ręcznie

# Docker dla Analityka AI

Jak uruchamiać narzędzia AI bez instalowania ich ręcznie

- Docker = kontener na gotowe narzędzie
- Bez instalacji Javy, Node, Pythona
- Jedna komenda → działa
- Idealny do:
  - serwerów AI
  - MCP
  - narzędzi analitycznych
  - PoC i warsztatów

# Dlaczego Docker jest ważny dla Analityka?

- Analityk nie konfiguruje środowisk
- Analityk uruchamia gotowe narzędzia
- Docker daje:
  - powtarzalność
  - brak „u mnie działa”
  - szybki start PoC
- AI + Docker = lokalne, kontrolowane środowisko

# Podstawowe pojęcia Dockera (intuicyjnie)

- **Obraz (image)** – „zestaw instalacyjny” aplikacji
- **Kontener (container)** – uruchomiona aplikacja
- **Port** – jak wchodzisz do aplikacji przez przeglądarkę
- **Volume** – dane, które nie znikają po restarcie

 *obraz = przepis, kontener = gotowe danie*

# Jak wygląda uruchamianie aplikacji w Dockerze?

- Bez Dockera:
  - instalacja
  - konfiguracja
  - konflikty wersji
- Z Dockerem:
  - `docker run ...`
  - aplikacja działa od razu
- Dla analityka:
  - nie interesuje Cię jak
  - interesuje Cię że działa

# Porty – jak „wejść” do aplikacji

- Aplikacje w Dockerze są „zamknięte”
- Port = drzwi do środka
- Przykład:
  - aplikacja słucha na porcie 8080
  - mapujesz ją na localhost:8080
- Efekt:
  - otwierasz przeglądarkę
  - widzisz UI / API / MCP

# Zmienne środowiskowe (ENV) – ustawienia bez kodu

- ENV = konfiguracja aplikacji
- Przykłady:
  - klucz API do LLM
  - tryb DEV / PROD
  - adres bazy danych
- Zaleta:
  - nie zmieniasz obrazu
  - tylko ustawienia

# Docker Hub – sklep z gotowymi narzędziami

- Repozytorium gotowych obrazów
- Bazy danych, AI, MCP, narzędzia
- Szukasz → uruchamiasz → testujesz

# Docker vs Docker Compose

## Docker

- Jedna aplikacja
- Jedna komenda

## Docker Compose

- Zestaw aplikacji
- Jeden plik `docker-compose.yaml`
- Idealne do AI + MCP

# docker-compose.yaml – serce konfiguracji

- Jeden plik = cała konfiguracja
- Obrazy, porty, ENV, wolumeny
- Zero kodu

# MCP + Docker

- MCP = serwer narzędzi dla AI
- Docker = szybkie i bezpieczne uruchamianie
- Idealne środowisko dla analityka

# Oficjalny katalog MCP w Dockerze

<https://docs.docker.com/ai/mcp-catalog-and-toolkit/>

---

- Gotowe serwery MCP
- Instrukcje Compose
- Punkt startowy bez kodowania

# Mentalny model Analityka

- Docker = START
- Compose = scenariusz
- MCP = rozszerzenie AI
- AI = asystent
- Ty = decyzyjny użytkownik

# AI – Koncepcje vs Narzędzia

(dla Analityków)

# Model vs Narzędzie (produkt)

## MODEL (konsepcja)

- Matematyczny / statystyczny mechanizm generujący odpowiedzi
- „Silnik” rozumowania językowego
- Sam w sobie nie ma UI

## NARZĘDZIE (produkt)

- Konkretna aplikacja używająca modelu
- Ma interfejs, ustawienia i integracje

## Przykłady

- Model: LLM
- Narzędzie: GPT-5.2, Claude, Gemini

Perspektywa analityka Model = kompetencja

Narzędzie = pracownik

# Chat vs Model

## CHAT

- Interfejs rozmowy
- Przechowuje kontekst rozmowy (czasowo)

## MODEL

- Generuje odpowiedzi statystycznie
- Nie pamięta poza kontekstem

## Przykłady

- ChatGPT
- Gemini Chat

# Copilot vs Chat

## COPilot

- AI wbudowana w narzędzie pracy
- Dostęp do plików, backlogu, dokumentów

## CHAT

- Brak kontekstu systemowego

## Przykłady

- GitHub Copilot
- Microsoft Copilot

# Asystent vs Agent

## ASYSTENT

- Reaguje na polecenia
- Nie działa samodzielnie

## AGENT

- Ma cel i kroki działania
- Może używać narzędzi

## Przykłady

- Asystent: ChatGPT
- Agent: AutoGPT, MCP Agent

# Prompt vs Instrukcja

## PROMPT

- Jednorazowe polecenie

## INSTRUKCJA

- Stałe zasady zachowania

# Token vs Kontekst vs Okno kontekstu

## TOKEN

- Jednostka tekstu

## KONTEKST

- Wszystko co model widzi

## OKNO KONTEKSTU

- Maksymalny rozmiar kontekstu

# Projekt (Custom GPT) vs Chat

## PROJEKT

- Zapisana konfiguracja
- Instrukcje, pliki, narzędzia

## CHAT

- Rozmowa ad-hoc

## Przykłady

- Custom GPT
- Gemini Gem

# Notebook vs Dokument

## NOTEBOOK

- Proces analizy krok po kroku

## DOKUMENT

- Statyczny artefakt

## Przykłady

- Copilot Notebook
- Jupyter Notebook

# RAG vs Chat z plikami

## RAG

- AI + wyszukiwanie w wiedzy

## CHAT Z PLIKAMI

- Jednorazowy upload

# Model ≠ Wiedza ≠ Prawda

- Model generuje odpowiedzi
- Wiedza pochodzi z danych
- Prawda wymaga walidacji

# Mentalny model analityka

- Model = silnik
- Chat = interfejs
- Copilot = pracownik
- Agent = wykonawca
- Prompt = pytanie
- Instrukcja = proces
- Projekt = rola
- RAG = pamięć organizacji

# Prompt Engineering

jako element integracji systemów z AI

# Prompty jako interfejsy

- Prompt = API kontraktowe między systemem a modelem
- Wejście:
  - formatowane dane,
  - kontekst,
  - rola,
  - ograniczenia
- Wyjście:
  - przewidywalna struktura
  - JSON, listy, schematy

Prompty są częścią architektury systemu, nie tylko dodatkiem

## Hermetyzacja promptów:

- dedykowane moduły promptów
- spójne konwencje, nazewnictwo, parametry modeli

## Prompty jako komponenty:

- „prompt-per-task” → separacja odpowiedzialności
- miks warstw: system → developer → user

# Standardy, testowanie, wersjonowanie

## Wersjonowanie promptów:

- repozytorium kodu
- PR review → porównanie jakości odpowiedzi
- tagowanie:

```
/prompts/v1/analysis/system-  
architecture.md
```

## Testowanie promptów:

- A/B testy jakości odpowiedzi
- regression tests: te same dane  
→ stabilność wyjścia
- testy „adversarial”  
→ sprawdzanie odporności na injection

## Operacjonalizacja:

- SLO: czas odpowiedzi, spójność, topical precision
- monitoring driftu danych i efektu zmian promptów
- walidacja schematu odpowiedzi (JSON Schema)
- limity długości i precyzji odpowiedzi
- obserwacja tokenów → koszty i czas

# Szablony

Prompt Templates

**query:**

„Stwórz ADR dotyczący wyboru CQRS w mikroserwisie płatności.”

**context:**

„System posiada dużo operacji odczytu, ale ma być zgodny z PSD2 i audytowalny.”

→ model otrzyma spójny kontekst i dostarczy poprawny ADR.

**SYSTEM:**

You are an expert assistant.

Provide validated structured output only.

**USER QUERY:**

{query}

**CONTEXT:**

{context}

**INSTRUCTIONS:**

1. Combine query + context.
2. No hallucinations.
3. If context is irrelevant, ignore it.
4. Missing info → return "missing\_data".

# Po co stosować template prompt

## Standaryzacja pracy

- Każdy prompt ma tę samą strukturę → mniej błędów, spójne rezultaty.
- Łatwiej tworzyć procesy, checklisty i workflow dla zespołu.

## Powtarzalność i automatyzacja

- Jeden template może obsłużyć setki przypadków użycia.
- Można go wstrzykiwać do CI/CD, Confluence, Jira, Copilot, pipelines.

## Lepsza kontrola jakości

- Wymuszenie formatów (JSON, ADR, RFC, diagramy).
- Minimalizacja halucynacji dzięki kontraktowi wejście/wyjście.

# Po co stosować template prompt

## Łatwiejsze testowanie i wersjonowanie

- Prompt jako artefakt → można go testować, porównywać i versionować w Git.
- A/B testy promptów stają się proste.

## Większa odporność na błędne dane

- Template pozwala modelowi reagować na braki: zgłosić ryzyka, nie wymyślać.

## Hermetyzacja logiki

- Template oddziela *logikę zadania od zmiennych danych* (query, context).
- Tak jak w programowaniu: parametr zamiast „wklejania kodu”.

## Spójność komunikacji

- W zespołach architektonicznych, analitycznych, dev i business — każdy używa tych samych formatów, produktów wyjściowych i standardów.

OpenAI Chat API:

# Structured Output

w kontekście API i integracji systemów

# Structured output

model zwraca ścisłe zdefiniowaną strukturę danych

Prompt - język naturalny:

```
Przeanalizuj ryzyka dla modułu core service ...
```

Odpowiedź - struktura:

```
{  
  "risks": ["single point of failure"],  
  "components": ["core-service"],  
  "recommendations": ["add redundancy"]  
}
```

To oznacza:

- Model nie generuje dowolnego tekstu, tylko **konkretny obiekt JSON**
- Każde pole ma z góry określony typ (string, number, array, enum...)
- Model nie może wyjść poza strukturę – nie wolno mu dopisać zdań, komentarzy, itp.
- Odpowiedź staje się walidowalna i deterministyczna

Zazwyczaj w formacie JSON,  
zgodnie z podanym schematem (np. JSON Schema).

# Structured output

Dlaczego jest kluczowe w integracji?

- Minimalizuje halucynacje dzięki twardemu schematowi odpowiedzi
- Umożliwia walidację na poziomie API (schema-first)
- Upraszczają mapowanie do kontraktów domenowych
- Stabilizuje integracje
  - LLM zwraca dane, nie tekst
- Pozwala budować "*deterministyczne*" pipeline'y (AI jako krok transformacji)

Model jako

# Komponent API

LLM jako mikroserwis

→ musi mieć kontrakt wejścia/wyjścia

- Structured output pełni rolę **API specification**
- Możliwość wdrożenia:
  - typowanych DTO (TS/Java/Kotlin)
  - validatorów (Zod/Effect Schema/JSON Schema)
  - automatycznego generowania dokumentacji

```
{  // JSON Schema
  "response_format": {
    "type": "json_schema",
    "json_schema": {
      "name": "system_analysis",
      "strict": true,
      "schema": {
        "type": "object",
        "properties": {
          "risks": {
            "type": "array",
            "items": { "type": "string" }
          },
          "components": {
            "type": "array",
            "items": { "type": "string" }
          },
          "recommendations": {
            "type": "array",
            "items": { "type": "string" }
          }
        },
        "required": [
          "risks",
          "components",
          "recommendations"
        ]
      }
    }
  }
}
```

# Kluczowe parametry

ścisłego output Chat API

- `response_format:{type:"json_schema"}`
  - ścisły format odpowiedzi
- `max_output_tokens`
  - kontrola kosztu i bezpieczeństwa
- `strict:true` w schemacie
  - modeli nie wolno wyjść poza kontrakt
- `tool_choice:"required"`
  - forcer na zwrócenie struktury zamiast tekstu

messages - interfejs sterujący zachowaniem modelu

```
messages: [  
    {  
        "role": "system",  
        "content": "Return JSON only."  
    },  
    {  
        "role": "user",  
        "content": "Analyze simple CRUD application."  
    }  
]
```

# Typowy wzorzec architektoniczny

Backend → AI Layer → Validation → Domain → Integrations

1. Backend formuje twarde wymagania (schema)
2. Chat API generuje *draft* danych w strukturze JSON
3. Wálidacja:
  - JSON Schema
  - Zod / Effect Schema
4. Mapowanie do obiektów domenowych
5. Agregacje, workflow, systemy downstream, ...

# Tools & Function Calling

W kontekście integracji systemów

Czym są

# “Tools” i Function Calling?

Tools w OpenAI to sposób na wywoływanie zewnętrznych akcji przez model

- API
- bazy danych
- systemy wewnętrzne
- kalkulatory
- parsery
- generatory plików
- agenci itp.

Model nie „zgaduje” formatu

— generuje strukturalne wywołanie funkcji.

Twoja aplikacja decyduje, co dalej zrobić.

Funkcje są kontraktem → JSON Schema określa:

- nazwę funkcji
- parametry
- typy danych
- strukturę komunikacji

# Architektura integracji: Model ↔ Tool

## 1. Klient wysyła zapytanie

→ user prompt (np. „podaj pogodę w Warszawie”).

## 2. Model ocenia kontekst

→ czy sama odpowiedź wystarczy, czy musi wywołać „tool”?

## 3. Model zwraca JSON “function\_call”

```
{  
  "name": "getWeather",  
  "arguments": { "city": "Warsaw" }  
}
```

## 4. Backend wykonuje funkcję

→ Twój kod, microserwis, DB, API zewnętrzne.

## 5. Wynik wraca do modelu jako tool message

→ Model generuje finalną odpowiedź dla użytkownika.

# Perspektywa Architekta

## Kontrolowane, deterministyczne API

- Model nie buduje zapytań tekstowych, tylko strukturalny JSON.

## Idempotencja

- narzędzia muszą być projektowane jak API
- przewidywalne.

## Separacja odpowiedzialności

Model: rozumienie i decyzja *co* zrobić.  
System: faktyczne wykonanie *jak i gdzie*.

## Bezpieczeństwo

- brak swobodnych komend, sandboxing na poziomie kontraktów.
- walidacja JSON schema przed wykonaniem.

# Perspektywa Architekta

- **Orkiestracja workflow**  
(model podejmuje decyzje, a system wykonuje akcje)
- **Natural language → API**  
(zapytania biznesowe mapowane na funkcje)
- **Inteligentne asystenty developerów**  
(tool do parsowania kodu, AST)
- **RAG + Tooling**  
(wyszukaj → policz → zwróć)
- **Autonomiczne czynności, ale kontrolowane**  
(np. wystaw fakturę → dodaj do CRM → wyślij Slack)

## Projektowanie funkcji:

# zasady:

**1. Jedna funkcja = jedna odpowiedzialność**  
Żadnych “doAll()”.

**2. Płaskie schematy JSON**  
Model lepiej radzi sobie  
ze strukturami 2–3 poziomów.

**3. Deterministyczność**  
Funkcje muszą być przewidywalne i idempotentne.

**4. Walidacja**  
Każdy argument przechodzi JSON Schema  
Validation.

**5. Kontekst minimalny**  
Nie wkładaj logiki biznesowej do modelu  
— w funkcji robisz core logic.

# integracyjne Best practices

## Hermetyzacja promptów

- Parametry, role, kontrakty kontrolowane po stronie serwera.

## Observability

- Loguj każde wywołanie toola wraz z promptem wejściowym.

## Retry & kompensacje

- Tool wyrzuca błąd
- model może podjąć inną akcję.

## Versioning

- Nowe funkcje  
= nowa wersja API + nowy system prompt.

## Testy:

- Testujesz funkcje niezależnie od modelu
- Testy regresyjne promptów i schem JSON

# Anti-patterns

- Funkcja „zrób wszystko”
- Zagnieżdżone i mega-skomplikowane JSONy
- Przekazywanie logiki do modelu zamiast do narzędzia
- Brak walidacji odpowiedzi modelu
- Tools w stylu: `exec_shell` (zaproszenie do exploitów)
- Tool → API → model → tool → infinite loop (bez kontroli)

# Jak projektować warstwy architektury pod Tools?

## Warstwa 1 – Model Orchestrator

OpenAI Chat API z toolami.

## Warstwa 2 – Tool Adapters

Jedna klasa na tool.

Odpowiedzialne za weryfikację inputu i mapowanie JSON → funkcja.

## Warstwa 3 – Services / Domain Logic

Prawdziwa logika biznesowa poza modelem.

## Warstwa 4 – Infrastructure

DB, HTTP, chmura, storage — poza wiedzą modelu.

# Przykładowe kategorie narzędzi

- **CRUD tools** → odczyt / zapis danych
- **Computation tools** → kalkulatory, pricing engines
- **Search tools** → vector search, SQL search
- **Integration tools** → Slack, e-mail, Jira, GitHub
- **Automation tools** → workflow steps
- **Parsing tools** → ekstrakcja danych z  
PDF/HTML/kodu

# Jak połączyć to z RAG?

1. Model ocenia pytanie
2. Wywołuje `vectorSearch` tool
3. Z wynikami podejmuje decyzję o kolejnym narzędziu
4. Może ponownie odpalić kolejne tooly (multi-step)
5. Finalna odpowiedź tworzy się dopiero na końcu

Architektonicznie:

- RAG staje się jednym z „tools” w pipeline
- Nie mieszasz logiki biznesowej z promptem

# Kiedy NIE używać Tools?

- Gdy odpowiedź modelu jest wystarczająca
- Gdy system jest prosty i ma mało integracji
- Gdy funkcja może być wykonana *taniej i szybciej*  
po stronie klienta

Tools są najważniejsze w systemach AI, które:

- mają side-effects
- integrują wiele API
- wymagają kontrolowanej logiki biznesowej

# AI w projektowaniu systemów

— asystent architektów systemowych

# Myślenie o promptach jak o narzędziu inżynierskim

- Prompty ≠ magia  
→ narzędzie pracy, jak UML, ADR, RFC
- LLM traktujemy jak **komponent systemu**:  
ma własne ograniczenia i interfejs
- Każdy prompt powinien mieć:  
cel, zakres i kryteria jakości
- Architekt nie „pyta z ciekawości”,  
tylko świadomie projektuje zapytanie

# Struktura promptów

w analizie technicznej i projektowaniu architektur

Rola (system) - nadaje zakres, ton,  
odpowiedzialność:

- „Jesteś architektem systemów...”

Kontekst

- architektura istniejąca
- ograniczenia biznesowe, NFR-y, ryzyka
- założenia sektorowe (finanse, telco, medyczne)

Format odpowiedzi

- JSON, listy, schematy, tabelaryczne porównania

Zadanie analityczne

- „przeanalizuj ryzyka”,  
„stwórz warianty architektury”
- „porównaj EDA vs SOA w tym przypadku”

Kryteria jakości:

- prompting strukturalny (sekcje, checklisty)
- few-shot - przykłady
- chain-of-thought kontrolowany:
  - „pokaż tylko plan / wnioski”
  - „krótko”, „z przykładami”,  
„z wyjaśnieniem trade-offów”

# Meta-prompts

- Prompty definiujące **jak** model ma myśleć, a nie **co** ma wygenerować
- Ustalają **proces**, styl, poziom szczegółowości, ograniczenia
- Używane do kontroli jakości i powtarzalności

## Przykłady meta-promptów

- „Zachowuj się jak senior system architect. Zanim odpowiesz — zapytaj o brakujące dane.”
- „Oceń jakość mojej specyfikacji, znajdź ryzyka, wskazówki usprawnień.”
- „Przeanalizuj problem jak ekspert DDD i zaproponuj model domenowy z wariantami.”

# Step-by-Step

## Prompts

- Wymuszają proces rozumowania krok po kroku
- Minimalizują halucynacje i błędy logiczne
- Idealne przy: analizie architektury, planowaniu migracji, refaktoryzacjach

## Struktura

- Krok 1: Zbierz wymagania
- Krok 2: Oceń kontekst i ograniczenia
- Krok 3: Porównaj opcje (wzorce, architektury)
- Krok 4: Zaproponuj rekomendację
- Krok 5: Dodaj ryzyka i testy weryfikujące

**Przykład** „Rozwiąż ten problem w 5 krokach: analiza → opcje → decyzja → ryzyka → metryki do monitoringu.”

# Prompty „Ask-Me-Questions”

Najważniejsze narzędzie architekta: model musi dopytać o brakujące dane

- Redukuje błędne założenia
- Tworzy proces podobny do realnych warsztatów architektonicznych

## Przykład

- „Zanim zaproponujesz architekturę, zadaj do 10 pytań, które są konieczne, aby uniknąć złych założeń.”

## Typowe obszary pytań

- RPO/RTO
- skala i throughput
- integracje zewnętrzne
- dane wrażliwe / compliance
- cykl życia danych
- krytyczność biznesowa
- ograniczenia budżetowe

## Przykład

- „Zanim wygenerujesz model domenowy, zapytaj o scenariusze, zdarzenia, bounded contexts.”
- „Najpierw sprawdź: czy kontekst jest kompletny? Jeśli nie — zapytaj.”

# Meta-prompty

Jak wykorzystać AI jako eksperta od Prompt  
Engineeringu

## Rola AI jako „prompt co-pilota”

- AI pełni rolę eksperta-asystenta, który pomaga:
  - doprecyzować cele,
  - strukturyzować prompty,
  - eliminować niejednoznaczności,
  - ustalić formaty wyjścia,
  - wykonać sanity-check promptu przed użyciem.

Instrukcja:

- „Najpierw oceń mój prompt → listuj problemy.”
- „Potem zaproponuj ulepszoną wersję.”
- „Na końcu wygeneruj wersję produkcyjną zgodną z moimi ograniczeniami.”

# Meta-prompt

„Tworzenie promptów wielorundowych”

AI jako architekt promptów:

- projektuje scenariusz dialogu,
- definiuje kolejne kroki,
- dodaje instrukcje typu:
  - „Zawsze zadawaj pytania, jeśli czegoś brakuje”
  - „Nie wykonuj zadania, dopóki nie potwierdzę specyfikacji”
  - „Użyj chain-of-thought, ale nie pokazuj go w output”

Prompty jako

# Narzędzie Architekta

# Wspomaganie projektowania architektury systemów

AI jako narzędzie augmentujące,  
a nie zastępujące architekta

- przyspiesza analizę,  
nie zastępuje decyzji
- świetna w eksploracji opcji,  
słabsza w ocenie trade-offów biznesowych
- warto traktować model jako  
**sparring partnera dla architekta**

# Generowanie propozycji wzorców architektonicznych

AI jako „katalog wzorców na żądanie”

- dobór wzorca do kontekstu domenowego
  - event-driven,
  - CQRS,
  - microservices,
  - SOA,
  - modular monolith
- omówienie plusów/minusów i typowych pułapek
  - generowanie wariantów architektury pod konkretne ograniczenia
    - latency,
    - throughput,
    - RODO/PII,
    - multi-tenant,
    - zero-downtime
  - analiza dopasowania:
    - czy wzorzec jest adekwatny do skali i ryzyka?
    - typowe antywzorce:
      - „microservices for everything”,
      - „CQRS bez potrzeby”

Analiza

# zależności i ryzyk technicznych

LLM jako narzędzie do *risk-spotting*

analiza zależności między usługami na podstawie kodu / opisów

- wykrywanie cykli, niejawnych couplingów
- szacowanie skutków zmian (blast radius)

identyfikacja miejsc podatnych na awarie:

- single point of failure
- brak timeouts / retry / circuit breaker

ocena gotowości na skalowanie:

- wąskie gardła I/O
- ograniczenia GPU/LLM, cache miss rate, cold starts

analiza zgodności z wymaganiami niefunkcjonalnymi

- SLO, RTO/RPO, polityki danych, compliance

# Dokumentowanie architektury

AI jako szybki generator dokumentacji,  
a nie „źródło prawdy”

tworzenie z AI:

- ADR (Architecture Decision Records)
- RFC / technical proposals
- standardów integracyjnych

automatyczne streszczenia  
dokumentów projektowych

porządkowanie istniejącej dokumentacji:

- deduplikacja, scalanie, porównywanie wersji
- generowanie checklist technicznych

# Generowanie diagramów architektury

Współpraca AI ↔ narzędzia diagramowe

aktualizacja diagramów na podstawie zmian w projekcie

- AI → tekst → automatyczna transformacja → diagram

generowanie szkiców diagramów z opisu tekstowego

- C4 (Context / Container / Component / Code)
- sequence diagrams
- flowcharts

utrzymywanie spójności między kodem, dokumentacją i diagramami

# Integracja AI w procesach projektowych

Jak realnie używa AI w architekturze?

AI jako „reviewer” decyzji projektowych

- ocena konsekwencji wyborów
- wskazanie brakujących założeń

AI jako wsparcie w warsztatach architektonicznych

- generowanie pytań i scenariuszy „what if”

AI w integracji systemów

- generowanie kontraktów API  
(OpenAPI/AsyncAPI)
- transformacje danych  
(mappings, validation)
- testy integracyjne i scenariusze E2E

AI w integracji danych

- propozycje schematów, rozwiązywanie konfliktów
- generowanie reguł ETL/ELT,  
walidacja jakości danych

# Ograniczenia i pułapki

– które architekt MUSI znać

- halucynacje → konieczna walidacja techniczna
- brak pełnej wiedzy o kontekście organizacji
- modele nie rozumieją kosztów operacyjnych i polityk bezpieczeństwa
- potrzeba audytowalności:
  - trace logi promptów
  - wersjonowanie promptów i artefaktów

ryzyka:

- prompt injection w dokumentacji
- błędne wzorce wynikające z niepełnych danych
- przeszacowanie możliwości modeli

# Best practices dla architektów

Jak pracować efektywnie z AI?

- używaj **structured prompting**:  
role → kontekst → cel → format
- utrzymuj źródło prawdy poza modelem (repo, ADR, UML/C4)
- generuj wiele wariantów i porównuj
- twórz szablony promptów dla:
  - analizy ryzyk
  - decyzji architektonicznych
  - generowania diagramów
- stosuj podejście: „AI jako junior architect + senior review”

# Podsumowanie

AI sam nie projektuje architektury  
— pomaga ją zaprojektować LEPiej

- szybciej eksploruje opcje
- wykrywa zależności i ryzyka
- przyspiesza dokumentowanie
- automatyzuje powtarzalne elementy procesu

■ ...

- pozostawia decyzje strategiczne architektowi

# AI w procesie deweloperskim

hype a rzeczywistość

# Vibe coding

"Kodowanie intuicyjne"

W praktyce vibe coding przyjmuje dwie formy:

## Czysty vibe coding

– pełne zaufanie do wyników AI.

Sprawdza się przy szybkich eksperymentach lub „weekendowych projektach do wyrzucenia”, gdzie liczy się tempo, a nie perfekcja.

## Odpowiedzialne programowanie z asystą AI

– AI działa jak inteligentny współprogramista.

Ty prowadzisz, weryfikujesz, testujesz i rozumiesz wygenerowany kod. To Ty zachowujesz kontrolę i pełną odpowiedzialność za efekt końcowy.

# Vibe coding

"Kodowanie intuicyjne"

Najpopularniejsze narzędzia do „ai - codingu”:

- GitHub Copilot – asystent AI w IDE, podpowiada kontekstowo kod. Obsługa kontekstu, MCP, itd.
- Cursor – (fork VS Code) z generowaniem i debugowaniem kodu.
- WindSurf – edytor AI do automatyzacji DevOps i pracy w chmurze.
- Replit – chmurowe IDE z funkcją AI „Get Unstuck”.
- Cody (Sourcegraph) – asystent AI z dostępem do repozytorium.
- Bolt (StackBlitz) – generuje aplikacje full-stack.
- v0 (Vercel) – generuje UI (React + Tailwind)
- ChatGPT – uniwersalny asystent AI do generowania i debugowania kodu.

# Vibe Coding

– skrócona checklista dobrych praktyk

- Najpierw Wizja i projekt

- Wiedz, co i po co budujesz.
- Myśl z perspektywy użytkownika i produktu.
- Najpierw UI/UX i spójny system komponentów.

- Środowisko i stack

- Używaj popularnych, dobrze wspieranych technologii (AI zna je najlepiej)
- Folder /instructions z przykładami i zasadami
- Czyste repozytorium GIT i często commity .

- Praca z AI

- Pisz konkretne, jednoznaczne prompty.
- Dziel duże funkcje na małe kroki.
- Restartuj chat, gdy AI zaczyna się gubić
- Podawaj tylko istotne pliki i komponenty, odwołuj się do przykładów.
- Audytuj kod, Waliduj dane, ukrywaj sekrety, weryfikuj uprawnienia i błędy

- Mindset

- Jasność i struktura ponad szybkość.
- Krótkie pętle, szybka weryfikacja.
- Buduj, testuj, poprawiaj — vibe z dyscyplina.

# Kilka praktycznych wskazówek

- Najpierw opowiedz mi swój plan — nie pisz kodu.
- Podaj kilka opcji, zaczynając od najprostszej — bez kodowania.
- Pomyśl tyle, ile trzeba, i zadawaj pytania, jeśli potrzebujesz więcej informacji.
- Wyszukaj powtórzenia lub zbędny kod i wypisz je.
- Skup się na zrozumieniu, jak działa kod, a nie na nauce jego pisania.

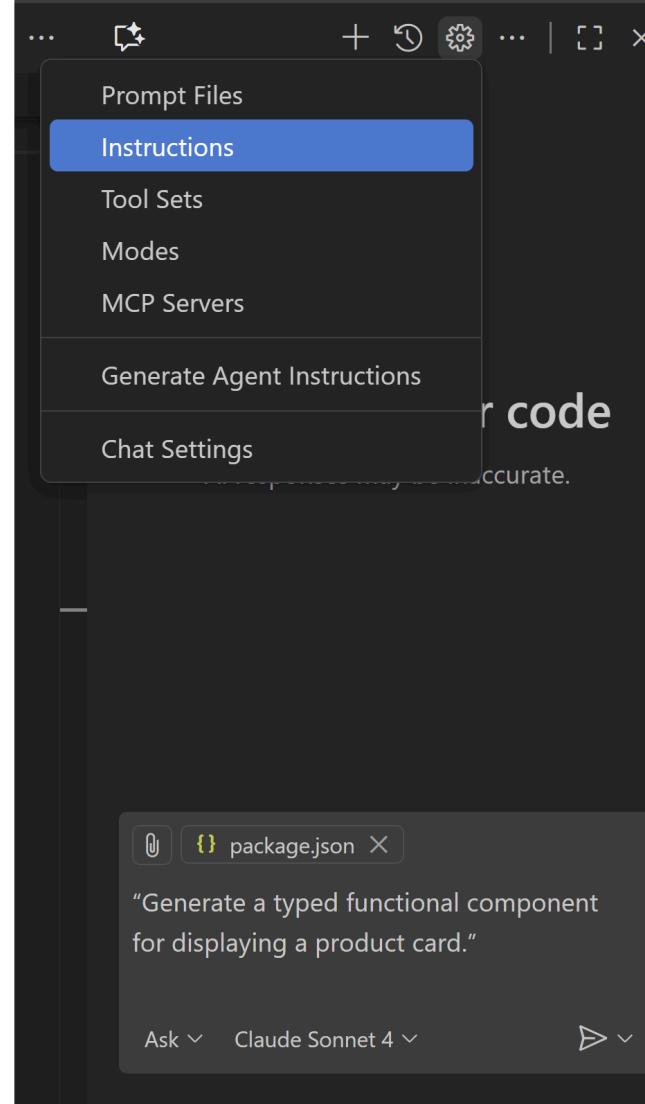
# Copilot Chat

Okno Copilot Chat (panel boczny lub inline)

- /explain — wyjaśnij zaznaczony kod
- /fix — popraw błędy lub linter issues
- /tests — wygeneruj testy jednostkowe
- /docs — stwórz dokumentację lub komentarz
- /commit — zaproponuj opis do commita

**Inline Chat (Ctrl+I)** - Chat kontekstowy  
otwierany w edytorze obok zaznaczonego  
fragmentu kodu.

- wyjaśnienie działania
- refaktoryzację
- konwersję do TypeScript
- dodanie testów lub komentarzy



## Konteksty Copilot Chat

- Open Editors – aktualnie otwarte pliki.
- Files & Folders – wybrane pliki lub katalogi.
- Search Results – wyniki wyszukiwania.
- Related Files – powiązane testy, style, typy.
- Instructions – plik zasad i stylu (copilot-instructions.md).
- Screenshot Window – zrzut ekranu edytora.
- Source Control – zmiany z Git (diff, commity).
- Problems – błędy i ostrzeżenia z panelu Problems.
- Symbols – funkcje, klasy i komponenty z kodu.
- Tools – integracje z narzędziami (npm, eslint, git).

Search for files and context to add to your request

-  Open Editors
-  Files & Folders...
-  Search Results
-  Related Files
-  Instructions...
-  Screenshot Window
-  Source Control...
-  Problems...
-  Symbols...
-  Tools...
-  package.json
-  copilot-instructions.md .github
-  mcp.json .vscode



# Process AI-driven

draft → refine → verify → integrate

# Wprowadzenie

do programowania z AI

## Czym jest proces AI-driven

- Systematyczny sposób pracy z Copilotem
- Oparty na cyklu iteracyjnym,  
nie pojedynczym promptcie
- Łączy kreatywność AI z kontrolą programisty

## Dlaczego potrzebny jest proces

- Pojedynczy prompt = losowy wynik
- Proces = przewidywalne rezultaty i wyższa jakość
- Każdy etap ma inne pytania i metody

## Rola Copilota

- Partner, nie generator
- Pomaga tworzyć, tłumaczyć, testować i dokumentować
- Wymaga kontekstu i kierunku, nie zgadywania

# Draft

Uzyskaj pierwszą wersję kodu lub struktury, szybko i bez oceniania

Prompty:

- “Create base React component with placeholder data”
- “Generate folder structure for a X feature using A + B”
- “Bootstrap simple layout with header and sidebar using Z”

Kroki:

- Poproś Copilota o wygenerowanie pojedynczego komponentu
- Sprawdź, co stworzył, bez poprawiania
- Zapisz wersję jako `draft`

# Refine

Uporządkuj, doprecyzuj i popraw jakość kodu

Prompty:

- “Refactor into smaller components and add typings”
- “Improve readability and follow our style rules”
- “Add abstract types and interfaces to this component”
- “Extract data fetching logic into custom service”

Kroki:

Ulepsz Komponent dodając typy i logikę stanu

# Verify

Sprawdź poprawność, stabilność i jakość

Prompty:

- “Generate test for this component”
- “Check this file for performance or security issues”
- “List potential runtime errors and fix them”

Kroki:

- Wygeneruj test dla komponentu
- Uruchom npm test
- Popraw błędy wskazane przez Copilot Chat

# Integrate

Połącz dopracowane elementy w spójny system

Prompty:

- “Summarize this diff as a commit message”
- “Document public API for this module”
- “Integrate components and services into complete view”
- “Explain how state flows through this component”

Kroki:

- Zintegruj komponent i serwis
- Wygeneruj opis commita przez Copilota
- Sprawdź integrację aplikacji

# Copilot w Pull Requestach

Code-review na "naszych zasadach"

# Zasada ogólna

Copilot “czyta” tylko to, co ma w kontekście:

- pliki repozytorium
- diff PR
- opis PR
- komentarze w PR

➡ Twoje zasady muszą:

- istnieć w repo
- pojawiać się w kontekście PR.

# Zasady w repo – źródło prawdy

Umieść wytyczne jako krótkie pliki:

- docs/architecture
  - architecture-guidelines.md
  - bounded-contexts.md
- docs/coding-style
  - typescript-style-guide.md
- docs/testing
  - testing-strategy.md
- docs/security
  - secure-coding-guide.md

Dlaczego działa?

Copilot widzi pliki repo i potrafi je wykorzystać przy generowaniu zmian.

# Włącz zasady do szablonu PR

PULL\_REQUEST\_TEMPLATE.md jako mini-prompt:

```
## Cel zmian

<!-- Krótko opisz problem i rozwiązanie. --&gt;

## Kontekst architektoniczny

- [ ] Zgodne z naszym stylem architektonicznym (`docs/architecture/architecture-guidelines.md`)
- [ ] Zgodne z zasadami warstw (`docs/architecture/layers.md`)
- [ ] Zgodne z zasadami DDD (`docs/architecture/ddd-rules.md`)

## Checklist

- [ ] Zastosowano konwencje nazewnicze z `docs/coding-style/typescript-style-guide.md`
- [ ] Dodano testy zgodnie z `docs/testing/testing-strategy.md`
- [ ] Zwrócono uwagę na bezpieczeństwo (`docs/security/secure-coding-guide.md`)</pre>
```

Komentarze w PR jako

# “mini system prompts”

Używaj komentarzy typu:

- „Zrefaktoryzuj zgodnie z `ddd-rules.md`”
- „Proponuj zmiany zgodne z `typescript-style-guide.md`”
- „Zastosuj nasze zasady warstwowania (patrz `layers.md`)”

Copilot dodaje te pliki do kontekstu.

# Połącz Copilota z twardymi regułami

Copilot to asystent, nie policjant.

Egzekwuj zasady narzędziowo:

- ESLint / Prettier
- ArchUnit, dependency rules
- testy architektoniczne
- skanery bezpieczeństwa

Copilot pomaga „naprawić”, ale to CI wymusza zgodność.

Narzędzia do statycznej analizy kodu są:

- przewidywalne
- konfigurowalne
- wytłumaczalne
- ścisłe

- ...
- Al nie jest.

# Copilot Enterprise + baza wiedzy

Jeśli organizacja używa Copilot Enterprise:

- dodaj tam ADR-y, guideline'y, playbooki
- Copilot w PR będzie mógł się do nich odwołać
- zwiększa trafność refaktoryzacji architektonicznych

# Standaryzowane prompty dla reviewerów

Przykłady komentarzy:

- „Przeanalizuj PR pod kątem warstw i DDD (patrz `architecture/layers.md` ).”
- „Sprawdź zgodność ze style guide + zaproponuj brakujące testy.”
- „Zidentyfikuj naruszenia zasad bezpieczeństwa.”

Stosowane konsekwentnie → Copilot zaczyna trafiać częściej.

# CODEOWNERS

AI review gate

## Wzmocnij proces PR:

- CODEOWNERS dla kluczowych katalogów
- CI z linterami, arch-tests, security scan
- Copilot wspiera autora i architekta w przejściu review

# CODEOWNERS

```
# This is a comment.  
  
*      @username1  
  
/docs  @username2  
  
*.js   @dev-team
```

Gwiazdka `*` reprezentuje wszystkie pliki w repozytorium, a `@username1` jest przypisany jako właściciel kodu.

`/docs` określa, że `@username2` będzie właścicielem kodu wszystkich plików w katalogu `docs`.

`*.js` wskazuje, że wszelkie pliki JavaScript w repozytorium należą do `@dev-team`.

# Plan wdrożenia

w organizacji:

- Stwórz krótkie zasady w `docs/`
- Dodaj szablon PR z checklistami
- Ustaw linters + arch-tests
- Przygotuj zestaw komentarzy-promptów
- Przeszkol zespół (1h)
- (Opcjonalnie) Połącz z Copilot Enterprise

# Efekt

Copilot zaczyna proponować zmiany:

- zgodne z architekturą
- spójne ze stylem organizacji
- zgodne z zasadami DDD, testów i bezpieczeństwa



PR stają się szybsze, czytelniejsze i bardziej spójne.