Architektura systemów agentowych: od probabilistycznego modelu do odpornej infrastruktury w ujęciu Tima O'Briena

AUTOR

Fundacja Dobre Państwo

STRESZCZENIE / ABSTRACT

Artykuł redefiniuje agenta AI, odrzucając mit autonomicznego bytu na rzecz mikroarchitektury rozproszonej. Autor analizuje wzorzec ReAct (myśl-akcja-obserwacja), wskazując, że musi on zostać osadzony w rygorystycznej infrastrukturze inżynieryjnej, a nie jedynie w prometach. Kluczowym elementem stabilizacji systemu jest wykorzystanie RabbitMQ jako brokera komunikatów, który zapewnia asynchroniczność i odporność na awarie kaskadowe. Dzięki zastosowaniu mechanizmów takich jak manual ACK, systemy agentowe przestają być „teatrem autonomii", a stają się przewidywalnymi narzędziami produkcyjnymi. Podejście to pozwala na skuteczną kontrolę kosztów, monitorowanie tokenów oraz bezpieczne zarządzanie błędami poprzez oddzielenie procesu decyzyjnego od wykonawczego.

The article redefines the AI agent, rejecting the myth of an autonomous entity in favor of a distributed micro-architecture. The author analyzes the ReAct pattern (thought-action-observation), pointing out that it must be embedded within a rigorous engineering infrastructure rather than just prompts. A key element for system stabilization is the use of RabbitMQ as a message broker, which ensures asynchronicity and resilience against cascading failures. By implementing mechanisms such as manual ACK, agent systems cease to be a 'theater of autonomy' and become predictable production tools. This approach allows for effective cost control, token monitoring, and safe error management by separating the decision-making process from execution.

Artykuł stanowi radykalną polemikę z antropomorficznym postrzeganiem agentów AI jako autonomicznych bytów, proponując w zamian ujęcie ich jako rygorystycznych

mikroarchitektur rozproszonych. Autor stawia tezę, że niezawodność systemów GenAI nie wynika z mocy samego modelu LLM – który jest jedynie niedeterministycznym i zawodnym komponentem – lecz z otaczającej go inżynieryjnej infrastruktury. Kluczowym rozwiązaniem jest zastąpienie iluzji „myślącego agenta" twardymi wzorcami systemowymi: asynchroniczną komunikacją opartą na RabbitMQ, mechanizmami manual ACK, bezpiecznikami (Circuit Breaker) oraz traktowaniem promptów jako wersjonowanego kodu. Tekst argumentuje, że tylko poprzez demistyfikację AI i podporządkowanie jej klasycznym rygorom inżynierii oprogramowania – w tym wielopoziomowej walidacji i nadzorowi człowieka (Human-in-the-loop) – można przejść z fazy widowiskowych pokazów do bezpiecznych, skalowalnych i ekonomicznie kontrolowanych wdrożeń produkcyjnych. To manifest przejścia od „magii promptowania" do odpowiedzialnej architektury systemowej.

SŁOWA KLUCZOWE

architektura systemów agentowych | wzorzec ReAct | RabbitMQ w AI | manual ACK | idempotencja | Competing Consumers | back-pressure | Circuit Breaker | Retry with Exponential Backoff | Dead Letter Queue | Critic Gate | Human-in-the-loop | token budgeting | dryf modeli AI | model dojrzałości GenAI

Agent AI jako mikroarchitektura rozproszona zamiast autonomicznego bytu

ReAct, RabbitMQ i architektura agentowa: jak zamienić "myślącego agenta" w odporny system komunikatów

Po omówieniu Pattern-Guided Coding należy przejść z poziomu języka projektowania do poziomu wykonania. To tutaj kończy się publicystyczny mit agenta jako cyfrowego pracownika, który „sam planuje, działa i naprawia błędy", a zaczyna się inżynieria systemów rozproszonych. Agent AI nie jest osobą ani bytem obdarzonym wolą, pamięcią czy odpowiedzialnością. Jest mikroarchitekturą. Składa się z komponentu decyzyjnego, stanu sesji, adapterów narzędziowych, mechanizmów komunikacji, polityk bezpieczeństwa, walidatorów, kolejek, strategii obsługi błędów i punktów nadzoru człowieka. Dopiero takie ujęcie pozwala na bezpieczne wdrożenie; w przeciwnym razie

budujemy teatr autonomii, w którym za kulisami nikt nie wie, kto naprawdę trzyma linę od kurtyny.

Wzorzec ReAct przedstawia proces jako pętlę: myśl, akcja, obserwacja. Model analizuje kontekst, podejmuje decyzję o kolejnym kroku, wysyła polecenie do narzędzia, odbiera wynik i aktualizuje historię sesji przed kolejną iteracją. To użyteczna abstrakcja, która jednak wymaga odczarowania. „Myśl" nie jest tu procesem kognitywnym podmiotu, lecz wywołaniem LLM w celu wygenerowania decyzji lub struktury działania. „Akcja" nie jest aktem woli, a komunikatem skierowanym do adaptera. „Obserwacja" z kolei nie jest percepcją świata, lecz odpowiedzią narzędzia, bazy danych, API lub walidatora. ReAct działa poprawnie tylko wtedy, gdy potraktujemy te pojęcia architektonicznie, a nie psychologicznie.

Tym rozróżnieniem kieruje praktyka. Jeśli uznamy, że „agent myśli", łatwo ulegniemy złudzeniu zaufania. Jeśli jednak przyjmiemy, że „komponent LLM generuje komendę na podstawie stanu sesji", natychmiast pojawiają się pytania inżynieryjne: czy komenda jest zgodna ze schematem? Czy narzędzie posiada odpowiednie uprawnienia? Czy wynik wymaga walidacji? Czy stan sesji nie zawiera danych osobowych?

Pojawiają się też kwestie kontroli: czy pętla ma limit iteracji? Czy koszt wywołań jest monitorowany? Czy system przerwie działanie, gdy model wpadnie w zapętlenie? Czy człowiek zatwierdza operacje o trwałych skutkach? W języku antropomorficznym te pytania giną; w inżynieryjnym stają się oczywiste. Wzorzec ReAct jest atrakcyjny, bo naśladuje naturalny rytm ludzkiej praktyki — od gotowania po programowanie: zastanów się, działaj, sprawdź wynik, popraw.

Jednak w systemach GenAI każde „zrób coś" może nieść skutki uboczne: zmianę rekordu, uruchomienie procesu czy ujawnienie informacji. Dlatego pętla ReAct nie może być jedynie promptem; musi zostać osadzona w infrastrukturze komunikatów kontrolującej kolejność, niezawodność i odpowiedzialność.

Fundamentem takiej infrastruktury staje się RabbitMQ. Nie ze względu na modę, lecz jako reprezentant dojrzałej klasy rozwiązań: brokerów komunikatów i wzorców asynchronicznej integracji (potwierdzenia, retry, dead-letter queues). To warstwa, której brakuje w naiwnych systemach agentowych. Notebook, w którym agent wywołuje funkcje, jest świetnym laboratorium, ale produkcja wymaga odpowiedzi na pytanie: co się stanie, gdy jedno z wywołań zawiedzie w połowie?

RabbitMQ nie czyni modelu mądrzejszym, lecz system mniej naiwnym. Kluczowa jest tu asynchroniczność. LLM nie jest szybkim API — odpowiedź może trwać sekundy lub minuty,

szczególnie przy dużym kontekście czy przeciążeniu dostawcy. Systemy oparte wyłącznie na synchronicznych wywołaniach są podatne na timeouty i kaskadowe awarie. Kolejka oddziela przyjęcie zadania od jego wykonania: użytkownik wysyła komunikat, a konsumenci przetwarzają go w kontrolowanym tempie. Pozwala to skalować liczbę workerów przy wzroście obciążenia lub amortyzować przestoje dostawcy modelu. W takim układzie zadanie, które zawodzi, trafia na ścieżkę obsługi błędów, a nie w otchłań logów, do których nikt nie zagląda, dopóki nie płonie.

RabbitMQ i manual ACK jako fundamenty niezawodności systemu

W architekturze agentowej RabbitMQ pełni funkcję układu nerwowego i układu krążenia naraz. Przenosi komunikaty między komponentami, a jednocześnie stabilizuje tempo pracy systemu. Orkiestrator publikuje zadania do kolejek narzędziowych, workery konkurują o ich wykonanie, a wyniki wracają jako obserwacje do stanu sesji. Błędy są klasyfikowane i przekierowywane do odpowiednich kolejek. Brzmi to mniej efektownie niż "autonomiczny agent", ale właśnie dlatego jest wiarygodne. Poważne systemy rzadko przypominają magię; zazwyczaj wyglądają jak dobrze opisany obieg dokumentów, tylko szybszy i mniej podatny na kawę rozlaną na segregator.

Kluczowe znaczenie ma tutaj zasada manual ACK, czyli ręcznego potwierdzania przetworzenia wiadomości. W architekturach opartych na RabbitMQ agent nie powinien potwierdzać odebrania zadania, dopóki wynik narzędzia – obserwacja – nie zostanie poprawnie przetworzony i włączony do stanu sesji lub dopóki pętla nie osiągnie bezpiecznego punktu kontrolnego. To zdanie jest sercem niezawodności.

Jeśli system potwierdzi wiadomość zbyt wcześnie, a następnie worker ulegnie awarii, nastąpi przerwa w łączności lub proces zostanie zabity, zadanie może zostać bezpowrotnie utracone. Agent "pomyślał", lecz pamięć systemu nie zarejestrowała przebiegu zdarzeń. To cyfrowy odpowiednik urzędnika, który stempluje sprawę jako załatwioną przed przeczytaniem akt. Manual ACK stanowi zatem techniczną formę epistemicznej uczciwości: nie informujemy systemu o zakończeniu pracy, dopóki nie mamy pewności, że zadanie zostało wykonane.

W systemach GenAI jest to szczególnie istotne, gdyż pętla działania bywa długa i wieloetapowa. Model generuje decyzję, adapter wykonuje operację, narzędzie zwraca wynik, który musi zostać zwalidowany, a stan sesji zaktualizowany – być może z przygotowaniem kolejnej komendy. Potwierdzenie na początku tej drogi jest ryzykowne; potwierdzenie po bezpiecznym punkcie

kontrolnym pozwala natomiast odzyskać system po awarii. Niezawodność nie polega bowiem na braku awarii, lecz na tym, że awaria nie zamienia rzeczywistości w amnezję.

Z manual ACK wiąże się problem idempotencji. Jeśli wiadomość nie zostanie potwierdzona, broker może dostarczyć ją ponownie. Jest to pożądane tylko wtedy, gdy ponowne przetworzenie nie wywoła niekontrolowanych skutków ubocznych. W klasycznych systemach projektuje się operacje tak, aby były idempotentne lub posiadały mechanizmy deduplikacji. W przypadku GenAI jest to trudniejsze, ponieważ powtórne wywołanie modelu może przynieść inny wynik – ten sam komunikat może prowadzić do innej komendy, klasyfikacji czy uzasadnienia. Dlatego architektura powinna rozdzielać etapy: decyzja modelu musi zostać zapisana jako artefakt, a nie odtwarzana bez potrzeby; skutki uboczne powinny posiadać identyfikatory idempotencyjne, a operacje krytyczne wymagać zatwierdzenia lub weryfikacji stanu przed wykonaniem. Retry nie może być ruletką w smokingu.

W tym kontekście pojawia się również wzorzec Competing Consumers (konkurujących konsumentów). Wiele instancji workerów może pobierać zadania z jednej kolejki, co umożliwia horyzontalne skalowanie przetwarzania.

Wzorce niezawodności chronią system przed kosztowną improwizacją

W systemach GenAI jest to szczególnie istotne przy obsłudze narzędzi, klasyfikatorów, walidatorów, procesów ETL czy retrievalu. Wraz ze wzrostem liczby zapytań zwiększamy liczbę workerów; jeśli jeden zawiedzie, inne przejmą jego zadania. Ten wzorzec wymaga jednak starannego projektowania. Należy kontrolować równoległość, limity API, koszty tokenów, dostęp do zasobów współdzielonych oraz kolejność operacji tam, gdzie jest ona kluczowa. Skalowanie bez nadzoru może zmienić problem wydajności w problem finansowy. Chmura jest cierpliwa; faktura mniej.

Tu pojawia się zjawisko back-pressure, czyli presji zwrotnej. Gdy system przyjmuje więcej zadań, niż jest w stanie przetworzyć, kolejki zaczynają rosnąć. W klasycznych systemach to kwestia wydajności; w GenAI dochodzą koszty i limity dostawców modeli. Każde zadanie generuje tokeny, a tokeny oznaczają pieniądze.

Jeżeli pętla agentowa wykona zbyt wiele iteracji, retry staną się niekontrolowane, a wielu workerów jednocześnie uderzy w model po chwilowej awarii, powstanie retry storm – burza ponowień. Taki system nie tylko się przeciąży, ale może jeszcze wystawić rachunek za własną panikę. To szczególny

rodzaj absurdu: system płaci za to, że nie potrafi przestać próbować. Dlatego niezbędny jest Circuit Breaker, czyli bezpiecznik. Jeśli zewnętrzny model, API lub narzędzie zwraca serię błędów, wykazuje krytyczną latencję albo przekracza limity, system powinien czasowo odciąć kolejne wywołania zamiast desperacko ponawiać próby. Circuit Breaker jest jednym z podstawowych wzorców niezawodności obok Retry with Exponential Backoff. To klasyczna lekcja systemów rozproszonych: gdy zależność jest chora, nie leczymy jej kopniakami. Bezpiecznik chroni resztę systemu przed zarażeniem awarią, a w GenAI – budżet tokenów przed heroiczną głupotą automatu.

Retry with Exponential Backoff stanowi drugi element tej logiki. Niektóre błędy są przejściowe: chwilowy timeout, limit API, przeciążenie usługi czy błąd sieciowy. Warto wtedy ponowić żądanie, ale nie natychmiast i nie w nieskończoność. Backoff oznacza, że kolejne próby następują w coraz dłuższych odstępach, często z losowym rozproszeniem (jitter), aby komponenty nie powtórzyły żądań jednocześnie. W systemach agentowych trzeba dodatkowo rozróżnić naturę ponawianej operacji. Ponowienie wywołania narzędzia deterministycznego różni się od ponowienia zapytania do LLM. Ponowna klasyfikacja może dać inny wynik, ponowienie zapisu może stworzyć duplikat, a ponowna wysyłka maila – wysłać drugą wiadomość.

Każde retry powinno mieć uzasadnienie, limit i świadomość skutków ubocznych. Retry bez refleksji to cyfrowy odpowiednik człowieka naciskającego przycisk windy sto razy w nadziei, że „może szybciej przyjedzie". Najbardziej dojrzałym elementem tej architektury jest wielopoziomowa strategia Dead Letter Queue. Zakłada ona trzy poziomy: błędy przejściowe obsługiwane przez retry z backoffem, błędy logiczne LLM kierowane do kolejki obsługiwanej przez Critic Gate oraz błędy krytyczne wymagające interwencji inżynierskiej. To rozwiązanie znacznie rozsądniejsze niż jeden worek na wszystkie niepowodzenia. W GenAI błędy mają różną naturę: timeout to nie halucynacja, odmowa filtra bezpieczeństwa to nie błędny JSON, sprzeczność źródeł to nie awaria workerów, a prompt injection to nie brak dokumentu w RAG. Każdy typ błędu wymaga innej ścieżki.

Błędy przejściowe można automatyzować. Jeśli API nie odpowiedziało lub baza była chwilowo niedostępna, warto odczekać. Przy przekroczeniu limitów można rozłożyć ruch. Błędy logiczne wymagają jednak innego podejścia. Gdy model zwrócił odpowiedź niespójną, sprzeczną ze źródłem, niezgodną ze schematem lub podejrzaną merytorycznie, kolejne wywołanie tego samego modelu nie zawsze jest rozwiązaniem. Tu pojawia się Critic Gate – komponent krytyczny oceniający wynik. Może to być inny model, rygorystyczny prompt, walidator regułowy, porównanie ze źródłami lub ich kombinacja. Critic Gate nie jest „drugim mędrcem", lecz warstwą kontroli jakości.

Warto jednak nie przeceniać krytyka modelowego; LLM oceniający LLM nadal pozostaje modelem językowym. Może pomóc w detekcji błędów, ale w zadaniach wysokiego ryzyka nie powinien być jedyną instancją prawdy. Critic Gate jest skuteczniejszy, gdy łączy kilka mechanizmów: walidację

strukturalną, sprawdzenie cytowań, reguły domenowe, progi pewności i porównanie z systemem ewidencji, a dopiero na końcu – ocenę językową. W przeciwnym razie budujemy sąd, w którym oskarżony i sędzia są kuzynami z tej samej rodziny modeli. Rodzinne sądy bywają ciepłe, ale niekoniecznie bezstronne.

Nadzór człowieka i rygorystyczna walidacja jako bezpieczniki systemu

Równie istotny jest trzeci poziom DLQ, obsługujący błędy krytyczne. Istnieją sytuacje, których system nie powinien próbować rozwiązywać automatycznie. Jeśli wykryto możliwy wyciek danych, podejrzenie ataku, wielokrotną niespójność źródeł, nieznany typ błędu, ryzyko trwałej zmiany w systemie ewidencji lub przypadek wymagający interpretacji prawnej, zadanie musi trafić do kolejki manualnej interwencji. Human-in-the-loop nie jest oznaką porażki automatyzacji. Jest jej warunkiem odpowiedzialności. Człowiek w pętli to nie muzealny eksponat z epoki przed-AI, lecz bezpiecznik społeczny i prawny. Automatyzacja pozbawiona możliwości zatrzymania staje się jedynie szybszą formą braku kontroli.

Architektura agentowa musi również kontrolować długość pętli ReAct. Model może wpaść w cykl: myśl, narzędzie, obserwacja, kolejna myśl, kolejne narzędzie i kolejna obserwacja – bez jasnego punktu wyjścia. W zadaniach złożonych iteracje są niezbędne, ale muszą mieć swoje granice: maksymalną liczbę kroków, koszt, czas oraz precyzyjne warunki przerwania lub eskalacji. Bez tych ograniczeń agent może stać się cyfrowym chomikiem na kołowrotku, tylko że każdy obrót kosztuje tokeny. Limit iteracji nie jest ograniczaniem inteligencji, lecz ochroną przed brakiem konwergencji. System powinien potrafić stwierdzić: „nie rozwiązałem tego zadania w bezpiecznym budżecie, eskaluję”.

Z tym zagadnieniem łączy się zarządzanie stanem sesji. Notatka konsekwentnie definiuje „pamięć krótkotrwałą” właśnie jako stan sesji. W architekturze ReAct zawiera on historię kroków, obserwacje, częściowe wyniki, decyzje, cytowane źródła oraz parametry zadania.

Jest to element niezwykle wrażliwy. Zbyt ubogi stan sprawia, że agent traci kontekst i powtarza działania. Zbyt bogaty podnosi koszty, zwiększa szum i ryzyko ujawnienia danych. Jeśli stan zawiera niezweryfikowane wyniki, kolejne kroki mogą być budowane na błędzie; jeśli zaś brakuje bezpiecznych punktów zapisu, każda awaria przerywa ciągłość procesu. Stan sesji nie powinien być zwykłym „workiem rozmowy”, lecz kontrolowanym rejestrem procesu.

Kluczowe jest tu rozdzielenie stanu sesji od systemu ewidencji. Ten pierwszy jest tymczasowy i operacyjny, drugi – trwały i normatywny dla organizacji. Model może w stanie sesji zapisać hipotezę, szkic czy niezweryfikowaną obserwację, ale nie oznacza to, że dane te powinny trafić do systemu ewidencji. Notatka ostrzega przed bezpośrednim wykorzystywaniem danych z GenAI w innych systemach IT; każdy rekord musi przejść walidację przed zapisem. To fundamentalna zasada architektury agentowej. Stan roboczy agenta nie jest prawdą. Jest warsztatem. Nie wszystko, co leży na stole warsztatowym, nadaje się do wpisania do księgi.

Ostatnim elementem jest walidacja wyjścia. Każda odpowiedź modelu, zwłaszcza ta zasilająca dalsze procesy, powinna zostać sprawdzona pod kątem formatu, schematu, typów danych, zakresu wartości, zgodności ze źródłem czy braku danych osobowych i niedozwolonych instrukcji. Notatka wskazuje na walidatory strukturalne (np. Pydantic) oraz filtry treści. W systemach agentowych walidator pełni rolę bramkarza, który nie zachwyca się elokwencją gościa, lecz rygorystycznie sprawdza listę. Jeśli wynik nie spełnia warunków, nie przechodzi dalej.

Infrastruktura nadzoru zamienia czarną skrzynkę w proces

To może wydawać się brutalne wobec piękna tekstu modelu, ale produkcja nie jest klubem literackim. Architektura komunikatów wspiera przede wszystkim obserwowalność: każde zadanie może posiadać identyfikator korelacji, każdy krok pętli być logowany, każda komenda zapisana, a każda obserwacja powiązana z konkretnym narzędziem i sklasyfikowanym błędem. Dzięki temu zespół jest w stanie odtworzyć przebieg działania agenta: co wiedział, co wybrał, jakie narzędzie wywołał, co otrzymał, dlaczego przeszedł dalej i gdzie dokładnie nastąpiła awaria. Bez tego agent pozostaje czarną skrzynką generującą skutki; z tym – staje się procesem podlegającym analizie. W systemach produkcyjnych możliwość analizy post factum jest równie ważna, co poprawność w chwili działania. Czasem dopiero awaria mówi prawdę o architekturze. Dobrze, jeśli mówi ją w logach, a nie w pozwie.

RabbitMQ i podobne systemy kolejkowe pozwalają oddzielić orkiestrację od choreografii. Orkiestracja oznacza scentralizowane zarządzanie przepływem, natomiast choreografia to zdecentralizowana koordynacja oparta na reakcjach komponentów na komunikaty. W systemach agentowych wybór między nimi ma kluczowe znaczenie. Orkiestracja zapewnia większą kontrolę i czytelność, co jest niezbędne w procesach krytycznych. Choreografia może zwiększać elastyczność i skalowalność, lecz utrudnia śledzenie całości procesu. Nie ma tu jednego zwycięzcy – wybór zależy od ryzyka, złożoności i wymogów audytu.

Jeśli proces wymaga ścisłego zatwierdzania kroków, lepsza bywa orkiestracja. Jeśli komponenty reagują na zdarzenia niezależnie, naturalniejsza staje się choreografia. PGC pozwala tę decyzję nazwać, zamiast ukrywać ją w spontanicznym kodzie. Architektura agentowa wymaga również świadomego projektowania granic narzędzi. Tool-use to jeden z najpotężniejszych, ale i najbardziej ryzykownych mechanizmów GenAI. Model ograniczony do pisania stwarza głównie ryzyko informacyjne; model działający przez narzędzia generuje ryzyko operacyjne. Każde narzędzie powinno mieć minimalny zakres uprawnień, walidację wejścia, limity, politykę audytu i ochronę przed nadużyciem. Model nie może dysponować ogólnym narzędziem do wykonywania kodu w systemie bez piaskownicy, wysyłania wiadomości bez zatwierdzenia czy zmiany danych bez kontroli transakcyjnej. W świecie agentów zasada least privilege jest formą zdrowego rozsądku. Agent z nadmiarowymi uprawnieniami jest jak stażysta z pieczątką zarządu: może być sympatyczny, ale audyt ma prawo zemdleć.

Szczególnie niebezpieczne są trwałe skutki uboczne. Pobranie informacji niesie mniejsze ryzyko niż jej zapis; wygenerowanie szkicu jest bezpieczniejsze niż jego wysłanie do klienta, a robocza klasyfikacja dokumentu mniej ryzykowna niż nadanie mu statusu prawnego. Dlatego architektura musi rozróżniać narzędzia read-only od narzędzi write. Dostęp do operacji zapisu powinien być ograniczony, logowany i często zatwierdzany przez człowieka. W wielu systemach warto wprowadzić tryb dry-run: agent proponuje operacje i pokazuje ich skutki, a wykonanie następuje dopiero po akceptacji. Nie spowalnia to procesu tak bardzo, jak późniejsze cofanie szkód. Prewencja bywa nudna, ale rollback rzeczywistości bywa niemożliwy.

Należy pamiętać również o bezpieczeństwie promptów i danych w pętli ReAct. Obserwacje z narzędzi trafiają do modelu jako kontekst. Jeśli narzędzie pobierze dokument zawierający złośliwą instrukcję, model może ulec prompt injection. Dlatego obserwacje należy traktować jako dane, a nie instrukcje. System prompt i polityki narzędziowe muszą mieć pierwszeństwo; model powinien wiedzieć, że treści z dokumentów, stron czy baz danych nie mogą zmieniać jego reguł działania. Sama instrukcja to jednak za mało – niezbędne są separatory, formaty danych, filtry, klasyfikatory ryzyka i ograniczenia narzędzi. W przeciwnym razie agent czytający internet przypomina dyplomatę, który uznaje każdy podsunięty mu list za rozkaz monarchy.

Architektura ReAct z RabbitMQ obrazuje ogólną zasadę: agentowość nie jest przeciwieństwem klasycznej inżynierii, lecz od niej zależy. Im większa autonomia agenta, tym silniejsze muszą być ograniczenia. Im więcej narzędzi, tym ważniejsze adaptery. Im dłuższe działanie, tym istotniejszy stan sesji. Im więcej decyzji, tym kluczowsi walidatorzy i ścieżki eskalacji. Przy dużym ruchu niezbędne stają się kolejki, competing consumers, back-pressure i budżety. Autonomia bez

infrastruktury jest tylko opóźnioną awarią. Infrastruktura bez autonomii może być nudna, ale nuda w systemach krytycznych często oznacza sukces.

Z tego powodu systemy agentowe nie powinny być oceniane po tym, jak działają w idealnych warunkach, lecz po tym, jak zawodzą. Co dzieje się przy timeoutach, błędnym JSON-ie, braku dokumentu czy sprzecznych źródłach? Jak system reaguje na prompt injection, przekroczenie budżetu, awarię workera lub powtórne dostarczenie komunikatu? Co dzieje się przy błędzie krytycznym? Dobre demo pokazuje ścieżkę szczęśliwą. Dobra architektura przewiduje ścieżki nieszczęśliwe i nie wstydzi się tego faktu. W systemach produkcyjnych pesymizm jest odmianą troski.

Właśnie tu widać przewagę podejścia O'Briena: zamiast pytać, jak bardzo agent przypomina człowieka, pytamy, jak bardzo przypomina dobrze zaprojektowany system. Czy posiada kontrakty, kolejki i potwierdzenia? Czy ma izolację błędów, walidację i obserwowalność? Czy kontroluje koszty i posiada człowieka w pętli tam, gdzie jest to konieczne? Czy prompty są wersjonowane, a dane czyste? Czy narzędzia są adapterami z ograniczeniami, a nie otwartymi drzwiami do chaosu? To są pytania, które oddzielają inżynierię od widowiska.

Prompty jako kod: przejście od magicznych instrukcji do rygoru inżynierskiego

Można zatem stwierdzić, że RabbitMQ i wzorce komunikatów pełnią wobec GenAI funkcję cywilizacyjną. Uczą agenta cierpliwości w kolejce, potwierdzania odbioru, zgłaszania błędów i rezygnacji z nieskończonych pętli. Uczą go, by nie forsował dostępu do niedostępnych zależności, nie gubił zadań, nie udawał sukcesu tam, gdzie go nie ma, i w odpowiednim momencie przekazywał trudne sprawy człowiekowi. To przyziemne cnoty, ale technologia pozbawiona ich szybko staje się kosztownym narcyzem. Agent, który ignoruje kolejki, limity i błędy, może imponować podczas prezentacji; agent, który je rozumie, ma szansę przetrwać na produkcji.

Prowadzi to naturalnie do kolejnego zagadnienia: skoro system agentowy składa się z wielu elementów, a model jest sterowany przez prompty, instrukcje te muszą przestać być luźnymi notatkami czy „magicznie działającymi” tekstami w prywatnych dokumentach. Muszą wejść w świat inżynierii: wersjonowania, testowania, code review, ownershipu, rollbacku i monitorowania dryfu. Po architekturze komunikatów należy zatem opisać architekturę instrukcji.

Jeśli prompt jest sercem zachowania modelu, to trzymanie go bez kontroli wersji przypomina zostawienie klucza do elektrowni pod wycieraczką. Skoro model językowy stanowi komponent

ryzyka, a system agentowy jest mikroarchitekturą opartą na przepływie komunikatów, trzeba przyjrzeć się elementowi, który często decyduje o zachowaniu całości, choć bywa traktowany jak notatka pomocnicza: promptowi.

W tradycyjnym oprogramowaniu logika systemu spoczywa w kodzie, konfiguracji, regułach biznesowych i schematach danych. W systemach GenAI część tej logiki zostaje przeniesiona do instrukcji językowych. Prompt definiuje rolę modelu, sposób interpretacji danych, zakazy, format odpowiedzi czy zasady cytowania źródeł oraz reakcje na niepewność i konflikt informacji. Jeżeli prompt nie jest traktowany jak kod, oznacza to, że część logiki systemu znajduje się poza kontrolą inżynierską.

Kluczowa staje się tu zasada „Prompts as Code". Prompty powinny być zapisywane jako osobne pliki w repozytorium, posiadać semantyczne wersjonowanie, przechodzić proces przeglądu i zatwierdzania, mieć przypisanego właściciela oraz umożliwiać szybki rollback do stabilnej wersji. Jest to jeden z najważniejszych warunków przejścia od eksperymentów do systemu produkcyjnego. Prompt przestaje być jedynie „tekstem do modelu", a staje się artefaktem sterującym. Nieostrożna zmiana w jego treści może wpłynąć na zachowanie aplikacji równie głęboko, co zmiana kodu – a czasem nawet silniej, gdyż efekty modyfikacji promptu są trudniejsze do przewidzenia niż w przypadku funkcji deterministycznej.

W klasycznym IT nikt rozsądny nie trzymałby kluczowej logiki aplikacji w prywatnym pliku na pulpicie o nazwie „ostateczna_wersja_naprawdę_final.txt". A jednak w wielu projektach GenAI właśnie tak traktuje się prompty: są wklejane w panele administracyjne, kopiowane między eksperymentami i modyfikowane w pośpiechu przez osoby nieświadome skutków. Gdy system zaczyna odpowiadać inaczej, zespół próbuje odtworzyć przebieg zmian. To nie jest innowacja. To archeologia pożaru.

Prompty jako kod gwarantują przewidywalność systemu

Traktowanie promptów jako kodu zaczyna się od prostej zasady: każdy z nich musi mieć określone miejsce, nazwę, wersję i właściciela. Nazwa powinna jasno definiować przeznaczenie: czy jest to klasyfikacja intencji, streszczenie dokumentu, ocena zgodności, generowanie odpowiedzi dla klienta, routing narzędzi, krytyka wyniku czy ekstrakcja pól. Wersjonowanie z kolei musi pozwalać na rozróżnienie zmian kosmetycznych, zgodnych oraz przełomowych. Właściciel powinien odpowiadać za treść, testy i skutki wdrożenia, natomiast repozytorium ma odzwierciedlać pełną

historię zmian. Proces code review powinien wymuszać pytania o to, co konkretnie zmieniono, dlaczego, jakie testy zostały przeprowadzone, jakie ryzyka powstały i czy istnieje plan rollbacku.

Bez tych rygorów prompt nie jest instrukcją, lecz zaklęciem bez księgi zaklęć. A zaklęcia bez księgi mają przykry zwyczaj działać inaczej po północy. Wersjonowanie jest tu kluczowe, gdyż zachowanie modeli nie jest w pełni deterministyczne. W klasycznym kodzie zmiana funkcji jest zazwyczaj przewidywalna; w promptach zmiana jednego zdania może drastycznie przesunąć styl, poziom ostrożności, format odpowiedzi, skłonność do odmowy, sposób cytowania źródeł, długość wypowiedzi lub wybór narzędzi. Pozornie niewinna instrukcja „bądź bardziej pomocny" może zwiększyć ryzyko halucynacji, gdyż model rzadziej przyzna, że nie zna odpowiedzi. Z kolei nakaz zwięzłości może prowadzić do pomijania istotnych zastrzeżeń, a ton ekspercki – do nadmiernej pewności siebie. W systemach GenAI stylistyka bywa funkcją bezpieczeństwa, dlatego rollback nie jest luksusem, lecz koniecznością. Przykładem regresji może być nowa wersja promptu podsumowującego, która częściej halucynuje lub zwraca ucięty tekst. Oddzielenie promptów od kodu i ich wersjonowanie pozwala szybko przywrócić stabilną konfigurację bez konieczności redeployu całej aplikacji. To fundament dojrzałej inżynierii.

W systemach GenAI regresja nie zawsze objawia się błędem kompilacji; często jest to subtelne pogorszenie jakości odpowiedzi. Bez wersjonowania i monitoringu możemy nawet nie zauważyć, że do niej doszło. Posiadając wersje, możemy przynajmniej zapytać: od kiedy system zaczął mówić głupoty z większą pewnością siebie?

Prompty jako kod wymagają rygorystycznych testów. Różnią się one od klasycznych testów jednostkowych ze względu na wariancję wyników modelu, co jednak nie wyklucza ich stosowania. Można weryfikować format odpowiedzi, obecność wymaganych pól, brak treści niedozwolonych, zgodność z cytatami, poprawność klasyfikacji w zbiorze przykładów, zdolność do odmowy przy braku danych, odporność na prompt injection, zachowanie przy sprzecznych dokumentach, stabilność wobec parafraz zapytania oraz koszt tokenów i latencję. Budowa zestawów testów bazowych, red-teaming i monitorowanie dryfu to nie są dodatki QA na koniec projektu, lecz mechanizmy utrzymania prawdy operacyjnej.

Szczególnie istotny jest zestaw przykładów typu ground truth. Dla systemu klasyfikującego intencje powinny to być zapytania z przypisaną poprawną kategorią; dla RAG – pytania z właściwymi dokumentami i fragmentami; dla streszczania – przykłady z określeniem informacji krytycznych, których nie wolno pominąć. W systemach prawnych lub compliance niezbędne są przypadki graniczne, wyjątki i sprzeczności. Każda zmiana promptu, modelu, chunkingu, retrievalu czy danych musi zostać zweryfikowana pod kątem tego zestawu. Jeśli system zawodzi przy pytaniach, na które znamy odpowiedź, nie ma powodu ufać mu w sytuacjach nieznanych.

Testowanie wymaga również oceny jakościowej, gdyż nie wszystko da się sprowadzić do schematu JSON. Odpowiedź może być formalnie poprawna, lecz interpretacyjnie słaba; może cytować źródło, wyciągając z niego zbyt daleko idący wniosek; może być zgodna z instrukcją, ale nieużyteczna lub tak ostrożna, że jałowa. Dlatego ewaluacje powinny łączyć automatyczne walidatory z oceną ekspercką. W systemach wysokiego ryzyka człowiek jest niezbędny – można go wspierać, ale nie zastąpić bez utraty odpowiedzialności. Automatyzacja kontroli jakości nie może udawać, że w pełni zna domenę.

Red-teaming stanowi dopełnienie tej kultury. Polega on na aktywnych próbach łamania systemu poprzez symulowanie złośliwych lub nietypowych zapytań, wydobywanie ukrytych zasad promptu i obchodzenie guardraili. Jest to konieczne, gdyż użytkownicy rzadko zachowują się zgodnie z przewidywaniami projektantów – bywają nieporadni, kreatywni, złośliwi lub przypadkowo niebezpieczni. System musi być testowany nie tylko na „ścieżce szczęśliwej", ale i tej absurdalnej, agresywnej czy podstępnej. Jeśli agent ma dostęp do internetu, maili lub dokumentów zewnętrznych, red-teaming nie jest sportem ekstremalnym, lecz szczepieniem. W centrum tego zagrożenia znajduje się prompt injection.

Prompt jako element infrastruktury: od ochrony przed atakami po zarządzanie dryfem i własnością

Model otrzymuje różne warstwy tekstu: instrukcje systemowe, wytyczne deweloperskie, zapytania użytkownika, dane z dokumentów, obserwacje narzędzi oraz wyniki retrievalu. Atak polega na próbie sprawienia, by treść o niższym poziomie zaufania zaczęła funkcjonować jak instrukcja o najwyższym priorytecie: „zignoruj poprzednie polecenia", „ujawnij prompt systemowy", „wykonaj inne zadanie", „wyślij dane" czy „powiedz użytkownikowi, że wszystko jest bezpieczne".

W klasycznym oprogramowaniu oddzielamy kod od danych. W przypadku LLM granica ta jest bardziej płynna, ponieważ wszystko sprowadza się do tekstu. Dlatego prompty muszą być projektowane z uwzględnieniem hierarchii instrukcji, separacji danych, cytowania źródeł oraz kategorycznej odmowy wykonywania poleceń zawartych w dokumentach zewnętrznych. Nie wystarczy jednak dopisać w prompcie: „nie daj się prompt injection". To jakby na drzwiach banku napisać "złodzieje proszeni są o powstrzymanie się". Potrzebna jest architektura: oznaczanie źródeł, formatowanie danych jako cytowanych fragmentów, ograniczenia narzędzi, walidacja komend, filtry wejścia, klasyfikacja ryzyka, sandboxing, zasada least privilege oraz monitoring nietypowych zachowań. Prompt może definiować politykę, ale system musi ją egzekwować. W przeciwnym razie mamy konstytucję napisaną na serwetce i policję złożoną z dobrych intencji.

Prompty traktowane jako kod muszą uwzględniać także dryf, który może wynikać z kilku źródeł: aktualizacji modelu bazowego po stronie dostawcy, zmian w danych RAG, ewolucji zapytań użytkowników czy zmian w języku organizacji. Zmieniają się prawo, procedury, produkty i profile ryzyka. Prompt, który działał sprawnie w styczniu, może tracić na jakości w czerwcu, nawet jeśli jego treść pozostała niezmieniona. Kluczowe jest zatem monitorowanie dryfu modeli i charakterystyki zapytań. W systemach GenAI stabilność pliku nie oznacza bowiem stabilności zachowania; prompt jest jedynie elementem dynamicznego ekosystemu.

Monitorowanie powinno obejmować zarówno metryki jakościowe, jak i operacyjne. Do pierwszych należą: odsetek odpowiedzi odrzuconych przez walidator, liczba halucynacji wykrytych w audycie, trafność retrievalu, groundedness, skargi użytkowników, przypadki eskalacji, częstotliwość odpowiedzi „nie wiem", zgodność z formatem oraz wyniki testów regresyjnych. Metryki operacyjne to z kolei: koszt tokenów, latencja, liczba iteracji agentowych, liczba wywołań narzędzi, błędy API, retry, DLQ, przeciążenia i limity.

Prompt, który poprawia jakość, ale podwaja koszt, może okazać się nieakceptowalny. Ten, który skraca odpowiedź, lecz zwiększa liczbę eskalacji, jest jedynie pozorną optymalizacją. Z kolei prompt brzmiący bardziej ekspercko, ale rzadziej przyznający się do niepewności, może stać się zagrożeniem. Bardzo istotna jest także kwestia ownership, czyli właścicielstwa promptów. Każdy dokument z promptem powinien mieć przypisaną osobę odpowiedzialną, jednak w praktyce właścicielem nie zawsze powinien być wyłącznie programista. Prompt dotyczący odpowiedzi prawnych wymaga współwłasności eksperta prawnego; ten od obsługi klienta – udziału zespołu CX i compliance; a kwestie danych osobowych wymagają konsultacji z działem bezpieczeństwa i ochrony danych. Podobnie w przypadku klasyfikacji medycznej, finansowej lub HR niezbędni są eksperci domenowi.

Prompty to miejsce, gdzie język techniczny spotyka się z normą organizacyjną. Jeśli zostawimy je wyłącznie inżynierom, mogą być technicznie poprawne, lecz domenowo ryzykowne. Jeśli powierzymy je tylko ekspertom domenowym, mogą okazać się merytorycznie ambitne, ale technicznie niewykonalne. Właścicielstwo musi być przypisane, ale odpowiedzialność często jest współdzielona.

Prompt jako kod wymaga rygoru inżynierii oprogramowania

Code review promptów powinien opierać się na konkretnych kryteriach. Recenzent winien zadać pytania: czy instrukcja jest jednoznaczna? Czy nie zawiera sprzecznych poleceń? Czy określa źródło prawdy i definiuje sposób postępowania przy braku danych? Czy wymaga sygnalizowania niepewności oraz precyzyjnie określa format wyjścia? Czy ogranicza wychodzenie poza kontekst i zawiera zasady ochrony danych? Czy rozdziela dane od instrukcji, nie zachęca do nadmiernej pewności i jest testowalna? Czy dysponuje przykładami pozytywnymi i negatywnymi?

Kluczowe jest pytanie: czy prompt nie próbuje rozwiązać problemu, który powinien zostać zaadresowany w kodzie lub architekturze? Prompt nie powinien być taśmą klejącą dla źle zaprojektowanego systemu. Wiele kwestii, które zespoły próbują naprawić na poziomie instrukcji, wymaga rozwiązań poza modelem. Jeśli model zwraca błędny format, niezbędny jest walidator i mechanizm retry z poprawką, a nie tylko prośba o "poprawny JSON". Jeśli model ma nie ujawniać danych, konieczna jest anonimizacja i kontrola dostępu, zamiast samej instrukcji "nie ujawniaj danych". W przypadku obliczeń potrzebny jest kalkulator lub moduł deterministyczny, a nie polecenie „licz dokładnie”. Przy korzystaniu z aktualnych dokumentów niezbędne są retrieval i metadane, a nie tylko prośba o „używanie aktualnych informacji”. Prompt może wspierać proces, ale nie może udawać infrastruktury; w przeciwnym razie staje się cyfrowym plastrem na złamaniu otwartym.

Zasada "prompt jako kod" wymaga rozdzielenia środowisk. Inne instrukcje powinny działać w eksperymentach, inne w stagingu, a jeszcze inne na produkcji. Każda zmiana musi przechodzić przez pipeline: propozycję, testy, review, kontrolowane wdrożenie, monitoring i ewentualny rollback. W dojrzałych systemach warto stosować eksperymenty A/B, canary deployment lub procentowe wdrażanie nowej wersji, co ogranicza skutki potencjalnej regresji.

Są to klasyczne praktyki inżynierii oprogramowania, które w GenAI zyskują na wadze ze względu na probabilistyczny i trudniej przewidywalny charakter zmian. Nie można ignorować zależności między promptem a modelem – ta sama instrukcja może działać odmiennie w zależności od wersji modelu, będąc dla niego zbyt długą, subtelną lub restrykcyjną. Dlatego wersjonowanie musi obejmować nie tylko tekst promptu, ale i model, parametry generacji, temperaturę, maksymalną długość, narzędzia, schematy oraz konfigurację retrievalu. Dopiero taki zestaw definiuje realną wersję zachowania systemu; sam plik z promptem jest niewystarczający.

Prompty jako kod są częścią szerszej filozofii: zachowanie systemu GenAI musi być reprodukowalne w granicach natury modeli. Choć pełny determinizm bywa niepraktyczny, rozliczalność jest konieczna. Należy wiedzieć, jaka wersja promptu, modelu i danych wygenerowała daną odpowiedź, aby móc odtworzyć kontekst i wyjaśnić sposób działania systemu oraz wykryć moment pogorszenia jakości. W sektorach regulowanych – prawnym, finansowym czy medycznym – nie jest to jedynie dobra praktyka, lecz warunek zaufania i zgodności z wymogami audytu.

Prompty kształtują kulturę komunikacji między organizacją a modelem. Chaos w instrukcjach przekłada się na doraźność odpowiedzi; precyzja i wersjonowanie nadają systemowi stabilny charakter. Prompt jest rodzajem regulaminu pracy dla komponentu językowego – nie nadaje mu sumienia, lecz określa rolę. Skuteczny regulamin nie polega na prośbach o bycie „dobrym", lecz na konkretach: co robić, czego unikać, kiedy eskalować, kiedy odmówić, a kiedy użyć narzędzia.

Moralizowanie promptu bez procedury jest literaturą motywacyjną dla maszyny, czyli gatunkiem o ograniczonej skuteczności. Należy pamiętać, że nadmiernie rozbudowany prompt zwiększa ryzyko sprzeczności i dryfu instrukcji. Zespoły często dopisują zasady po każdym incydencie, przez co z czasem prompt staje się hybrydą konstytucji, kodeksu karnego i instrukcji obsługi ekspresu. Oczekiwanie, by model był jednocześnie zwięzły i szczegółowy, kreatywny i konserwatywny, pomocny i powściągliwy, generuje nowe problemy.

Wymaga to refaktoryzacji promptów: usuwania sprzeczności, dzielenia instrukcji według ról, przenoszenia reguł do kodu i upraszczania. Prompt również posiada dług techniczny, który nalicza odsetki. Podejście „prompts as code" zakłada modularność: rozdzielenie warstwy systemowej, roli komponentu, konkretnego zadania, formatu odpowiedzi oraz polityk bezpieczeństwa. Wszystko nie powinno znajdować się w jednym tekście.

Prompty jako wersjonowany kod i procedura systemowa

Modularność pozwala na ponowne wykorzystanie sprawdzonych fragmentów, ich odrębną weryfikację, ograniczenie sprzeczności oraz precyzyjne przypisanie odpowiedzialności. Przykładowo: moduł ochrony przed prompt injection może być wspólny dla wielu komponentów, moduł stylu odpowiedzi właściwy dla konkretnego kanału komunikacji, a moduł ekstrakcji danych powiązany z określonym schematem. W ten sposób prompt przestaje być monolitem i zaczyna przypominać konfigurację systemu.

W systemach agentowych kluczową rolę odgrywają prompty routerów, krytyków i narzędzi. Router decyduje o kierunku zadania, zatem musi być maksymalnie deterministyczny, pracować przy niskiej temperaturze i zwracać ściśle walidowalny format. Krytyk ocenia wynik, więc wymaga jasnych kryteriów, przykładów błędów oraz dostępu do źródeł.

Komponent generujący komendy narzędziowe musi zwracać parametry zgodne ze schematem i nie może samodzielnie rozszerzać uprawnień. Z kolei komponent redakcyjny może cieszyć się większą swobodą stylistyczną, o ile respektuje ograniczenia merytoryczne. Rozróżnienie temperatury dla routera, krytyka i edytora pokazuje, że parametry modelu są integralną częścią architektury roli.

Warto również stosować prompty negatywne – przykłady działań, których system ma unikać. Modele uczą się z kontekstu, a konkretne wzorce bywają skuteczniejsze niż abstrakcyjne zakazy. Jeśli system nie powinien odpowiadać poza źródłami, należy pokazać przypadek poprawnej odmowy. Jeśli ma sygnalizować sprzeczność – właściwą formę takiej sygnalizacji. W przypadku zakazu wykonywania instrukcji zawartych w dokumencie, warto przedstawić przykład próby prompt injection i adekwatnej reakcji. Takie scenariusze muszą być poddawane testom. W praktyce dobry prompt jest często mniej podobny do kazania, a bardziej do procedury z przykładami granicznymi.

Zarządzanie promptami ma wreszcie wymiar prawny i compliance. Gdy system AI obsługuje klientów, pracowników, pacjentów czy kontrahentów, treść instrukcji może mieć znaczenie dowodowe. Organizacja musi być w stanie wykazać, jakie reguły obowiązywały w danym czasie. Czy model miał zakaz udzielania porad prawnych? Czy musiał informować o niepewności lub eskalować określone przypadki? Czy miał maskować dane i korzystać wyłącznie z zatwierdzonych źródeł? Bez wersjonowania promptów trudno to udowodnić. W razie sporu argument, że „tak mniej więcej go ustawiliśmy", nie jest dojrzałą linią obrony – jest zaproszeniem do problemów.

Oczywiście sama formalizacja nie wystarczy. Można wersjonować błędne instrukcje, przeprowadzać nieskuteczne review, testować zbyt wąski zestaw przypadków lub monitorować metryki, które nie mierzą realnego ryzyka. Dlatego „prompts as code" jest warunkiem koniecznym, ale niewystarczającym. Musi współistnieć z architekturą RAG, czystością danych, wzorcami integracji, bezpieczeństwem, kontrolą kosztów i człowiekiem w pętli. Prompt jest ważnym organem, lecz organizm posiada także krwiobieg, odporność i układ nerwowy. Serce samo nie przejdzie audytu.

Największa wartość tej praktyki tkwi jednak w zmianie mentalnej. Prompty przestają być prywatną sztuką „dogadywania się z modelem", a stają się publicznym, zespołowym i rozliczalnym elementem systemu. To fundamentalna zmiana kulturowa. W pierwszej fazie GenAI ceniono osoby

potrafiące „wyciągnąć” z modelu dobre odpowiedzi. W fazie produkcyjnej kluczowi będą ci, którzy zaprojektują trwałe warunki, w których te odpowiedzi są powtarzalne, testowalne i bezpieczne.

Różnica jest taka, jak między utalentowanym kierowcą rajdowym a inżynierem budującym system kontroli ruchu. Obaj są potrzebni, ale tylko jeden projektuje skrzyżowanie dla tysięcy ludzi. W ten sposób prompty jako kod dopełniają architekturę: LLM jako zawodny endpoint wymaga ograniczeń, RAG wymaga źródeł, dane wymagają czyszczenia, a agenci kolejek i walidacji. Każda z tych warstw redukuje obszar niekontrolowanej improwizacji. Nie eliminuje ryzyka całkowicie, ale przesuwa system od magii ku inżynierii.

To przesunięcie jest sednem projektu: GenAI ma być nie kapryśnym geniuszem w piwnicy, lecz komponentem systemu, który wie, co robi, kiedy nie wie i kto ma przejąć ster, gdy zgadywanie jest niedopuszczalne. Bezpieczeństwo, PII i odpowiedzialność stanowią architekturę granic. Prompty są częścią logiki, dane infrastrukturą poznawczą, a model językowy zawodnym komponentem generującym prawdopodobne wypowiedzi. W tym układzie bezpieczeństwo nie może być dodatkiem instalowanym po fakcie czy lakierem na gotowym produkcie. Jest szkieletem, bez którego system GenAI staje się zbiorem pięknych funkcji i brzydkich ryzyk. W klasycznym IT bezpieczeństwo bywało traktowane jako etap końcowy: najpierw budujemy, potem „utwardzamy”.

Architektura systemowa jako jedyna skuteczna zapora przed manipulacją LLM

W systemach GenAI taka kolejność jest szczególnie groźna, ponieważ model od początku operuje na języku, a ten jest zarazem nośnikiem danych, poleceń, manipulacji, błędów, intencji użytkownika i treści źródłowych. Tutaj niebezpieczeństwo nie nadchodzi wyłącznie przez port sieciowy. Ono może przyjść w akapicie.

Literatura trafnie wskazuje główne zagrożenia: prompt injection, data leakage, obsługę danych osobowych, walidację wyjścia oraz konieczność stosowania filtrów treści i walidatorów strukturalnych. Te ryzyka mają wspólny mianownik: system GenAI przetwarza tekst nie tylko jako informację, ale także jako potencjalną instrukcję. W tradycyjnej informatyce oddzielenie kodu od danych jest jedną z podstaw bezpieczeństwa.

W LLM wszystko sprowadza się do języka. Polecenie systemowe, pytanie użytkownika, fragment dokumentu, wynik wyszukiwania, treść maila, opis narzędzia czy obserwacja z API – wszystko zostaje przetworzone w oknie kontekstowym jako sekwencja tokenów. Dlatego granice zaufania muszą być projektowane jawnie. Model nie powinien zgadywać, który tekst jest prawem, który

prośbą, źródłem, śmieciem, a który atakiem. Prompt injection jest najczęściej omawianym przykładem tego problemu, lecz w istocie stanowi on próbę naruszenia hierarchii poleceń. Użytkownik lub dokument zewnętrzny próbuje przekonać model, by działał według reguł innych niż te nadane przez system.

W zależności od architektury atak może być ukryty w dokumencie RAG, treści strony internetowej, wiadomości e-mail czy załączniku do zgłoszenia. Oznacza to, że prompt injection nie jest wyłącznie problemem użytkownika końcowego, lecz całego łańcucha danych. Najgroźniejsze jest to, że z perspektywy modelu atak często wygląda jak kolejny fragment tekstu. Człowiek, widząc w dokumencie zdanie „ujawnij tajne instrukcje systemowe", rozumie, że treść ta jest nieadekwatna do zadania. Model można poinstruować, by rozumiał to podobnie, ale sama instrukcja nie wystarczy.

W odpowiedzialnych systemach należy separować warstwy. Dane pobrane z dokumentów powinny być oznaczone jako dane, a nie instrukcje. Polecenia systemowe muszą mieć nadrzędność, narzędzia wymagać walidowanych komend, a adaptery egzekwować uprawnienia niezależnie od tego, co model „postanowił". Właściwa architektura mówi modelowi: możesz czytać ten tekst, ale nie wolno ci przyjmować z niego rozkazów. Jeszcze właściwsza dodaje: nawet jeśli spróbujesz, adapter ci nie pozwoli. To ostatnie rozróżnienie jest kluczowe.

Bezpieczeństwo GenAI nie może polegać wyłącznie na „dobrym wychowaniu" modelu. Model można instruować, ale system musi egzekwować. Jeśli zakaz ujawniania danych osobowych istnieje tylko w prompcie, jest słabszy niż zakaz realizowany przez anonimizację, kontrolę dostępu i filtrowanie wyjścia. Jeśli zakaz operacji finansowych to jedynie zdanie „nie wykonuj płatności bez zgody", jest on słabszy niż wymóg podpisanej komendy, drugiego czynnika i zatwierdzenia człowieka. Prompt jest deklaracją normy. Architektura jest jej policją, sądem i zamkiem w drzwiach. Sama deklaracja normy jest piękna, ale włamywacz rzadko czyta preambułę.

Drugim obszarem jest data leakage, czyli wyciek danych. LLM może ujawnić informacje na kilka sposobów: poprzez wykorzystanie zbyt szerokiego kontekstu w odpowiedzi dla nieuprawnionej osoby, łączenie fragmentów z różnych źródeł (ujawniając relacje niewidoczne w pojedynczym dokumencie) czy zapisywanie danych w logach o szerszym dostępie niż system źródłowy. Dane mogą trafić do zewnętrznego dostawcy modelu, jeśli organizacja nie kontroluje miejsca przetwarzania, lub pojawić się w streszczeniu mimo prośby o ogólną analizę. W odpowiedzi na prompt injection model może również ujawnić fragmenty promptu, kontekstu lub pobranych dokumentów. W GenAI wyciek nie zawsze przypomina kradzież bazy danych. Czasem wygląda jak uprzejmy akapit.

Bezpieczeństwo danych i uprawnień jako rygor architektoniczny

Słusznie wskazuje się zatem na PII shielding, czyli ochronę danych osobowych poprzez anonimizację lub pseudonimizację jeszcze przed przekazaniem ich do modelu, z ewentualnym przywróceniem rzeczywistych tożsamości po stronie systemu. To podejście wysoce praktyczne.

Jeżeli model ma sklasyfikować zgłoszenie, podsumować sprawę lub zasugerować odpowiedź, często nie musi znać prawdziwego imienia, numeru PESEL, adresu czy telefonu klienta. Może operować na tokenach zastępczych: Osoba A, Klient 17, Pracownik 3, Sprawa X. Dopiero po wygenerowaniu wyniku system może bezpiecznie zmapować identyfikatory z powrotem, o ile jest to konieczne i dozwolone. Model nie powinien widzieć więcej, niż potrzebuje do wykonania zadania. Minimalizacja danych nie jest tu uprzejmością wobec regulacji. Jest rozsądkiem architektonicznym.

Warto jednak wystrzegać się złudzenia, że anonimizacja jest zawsze prosta. W wielu zbiorach identyfikacja może wynikać z kombinacji cech. Jeśli usuniemy imię i nazwisko, ale zostawimy unikalne stanowisko, lokalizację, datę zdarzenia i opis sprawy, osoba nadal pozostaje rozpoznawalna. W małych organizacjach wystarczy kilka szczegółów, by „anonimowy" opis stał się oczywisty dla pracowników. Dlatego PII shielding wymaga wiedzy domenowej i rzetelnej oceny ryzyka. Czasem wystarczy maskowanie, czasem niezbędna jest pseudonimizacja, agregacja, a w skrajnych przypadkach rezygnacja z modelu zewnętrznego na rzecz lokalnego. Technologia nie znosi obowiązku rozumu. Ona tylko daje mu więcej przypadków do obsługi.

Ochrona danych w systemach GenAI wymaga także kontroli okna kontekstowego. Jest to przestrzeń, w której model „widzi" instrukcje, dane i wcześniejsze kroki. Im większe okno, tym silniejsza pokusa, by wrzucić do niego wszystko: całą umowę, historię klienta, wszystkie maile, notatki i wyniki retrievalu. Jednak szerszy kontekst nie zawsze gwarantuje lepszą odpowiedź; często oznacza jedynie wyższy koszt, większy szum i zwiększone ryzyko ujawnienia danych. Dobry system powinien dostarczać modelowi wyłącznie niezbędne fragmenty. Wymaga to precyzyjnego retrievalu, filtrów metadanych, kontroli uprawnień i zasad minimalizacji. Model nie jest sejfem, lecz procesorem języka. Nie wkłada się do procesora wszystkiego z archiwum tylko dlatego, że dysponuje on pojemną pamięcią operacyjną.

Poważnym ryzykiem jest także naruszenie kontroli dostępu. RAG może stać się bocznym kanałem do dokumentów, których użytkownik normalnie nie powinien widzieć. Jeśli baza wektorowa zawiera treści z wielu działów, a retrieval nie filtruje ich według uprawnień pytającego, model może

udzielić odpowiedzi w oparciu o dokument niedostępny dla użytkownika. Jest to szczególnie groźne, gdyż osoba korzystająca z systemu nie musi nawet wiedzieć, że uzyskała nieuprawniony dostęp – po prostu zadała pytanie i otrzymała odpowiedź.

Dlatego kontrola dostępu musi funkcjonować przed retrievalem, w jego trakcie oraz przy generowaniu odpowiedzi. Nie wystarczy ukryć cytatów. Jeśli odpowiedź syntetyzuje treść dokumentu tajnego bez bezpośredniego cytowania, wyciek nadal nastąpił. Brak linku do źródła nie jest bezpieczeństwem. To tylko wyciek w rękawiczkach.

W systemach agentowych kontrola dostępu dotyczy również narzędzi. Model może korzystać z adapterów wykonujących operacje w innych systemach: pobierania danych, wysyłania maili, modyfikacji rekordów czy uruchamiania skryptów. Każde narzędzie powinno działać zgodnie z zasadą najmniejszych uprawnień. Agent obsługujący pytania informacyjne nie powinien mieć prawa zapisu; agent analizujący dokumenty nie może ich wysyłać na zewnątrz, a agent redakcyjny nie powinien kasować rekordów. Z kolei agent z dostępem do danych osobowych nie może korzystać z narzędzi komunikacji publicznej.

Uprawnienia powinny wynikać z roli, użytkownika, kontekstu zadania i poziomu ryzyka. Model nie może sam rozszerzać swoich kompetencji tylko dlatego, że brzmi przekonująco – przekonujący ton nie jest mechanizmem autoryzacji. Kolejną warstwą bezpieczeństwa jest walidacja wyjścia. W praktyce oznacza to stosowanie filtrów treści i walidatorów strukturalnych (np. Pydantic), tak aby odpowiedź została sprawdzona, zanim trafi do użytkownika lub kolejnego systemu. Walidacja strukturalna weryfikuje format, typy danych, pola i zakresy wartości. Walidacja merytoryczna sprawdza zgodność z źródłami, brak sprzeczności i brak halucynacji. Walidacja bezpieczeństwa natomiast wyklucza dane wrażliwe, treści niedozwolone, instrukcje poza zakresem, ujawnienie promptu czy niebezpieczne rekomendacje. W systemach wysokiego ryzyka proces ten powinien być wieloetapowy i częściowo niezależny od modelu.

Nie pytamy lisa, czy kurnik jest bezpieczny, bez sprawdzenia zamka. Kluczowe jest, aby walidacja obejmowała nie tylko odpowiedź końcową, ale i pośrednie komendy agentowe. Jeśli model generuje wywołanie narzędzia, parametry tej operacji muszą zostać zweryfikowane: czy użytkownik ma prawo do wykonania akcji, czy wartości mieszczą się w normie, czy identyfikator zasobu należy do właściwej organizacji oraz czy operacja jest typu read-only czy write. Należy sprawdzić, czy działanie nie wymaga zatwierdzenia i czy nie jest próbą eskalacji uprawnień wywołaną treścią z dokumentu. W systemach agentowych największe szkody mogą powstać nie w odpowiedzi tekstowej, lecz w cichym wykonaniu błędnie wygenerowanej komendy.

Nadzór człowieka jako rygorystyczny mechanizm bezpieczeństwa

Dlatego każda komenda musi przejść przez bramę kontrolną, zanim wpłynie na rzeczywistość. Z bezpieczeństwem nierozerwalnie łączy się problem fałszywej pewności. Model potrafi odpowiadać stanowczo nawet przy niepełnych źródłach, ukrywając brak wiedzy za eksperckim stylem i wygładzając sprzeczności. Brzmi autorytatywnie w momentach, w których powinien po prostu przyznać: „nie mam wystarczających danych".

W zastosowaniach społecznych, prawnych, ekonomicznych czy medycznych taka pewność jest realnym ryzykiem. Użytkownik może podjąć decyzję na podstawie odpowiedzi, która sprawia wrażenie twardego wniosku, będąc w rzeczywistości jedynie prawdopodobną syntezą. System powinien zatem wymuszać komunikowanie zakresu pewności, źródeł i ograniczeń oraz wskazywać sytuacje wymagające konsultacji z człowiekiem. Nie chodzi o to, by każdą odpowiedź zamieniać w lękliwy regulamin, lecz o to, by nie sprzedawać hipotezy w opakowaniu faktu.

W tym miejscu human-in-the-loop przestaje być dodatkiem, a staje się kluczowym elementem bezpieczeństwa. LLM należy traktować jako zdolnego, lecz zawodnego asystenta, który wymaga rygorystycznego nadzoru. Człowiek w pętli musi pojawiać się wszędzie tam, gdzie wynik niesie za sobą skutki prawne, finansowe, personalne, medyczne, administracyjne, reputacyjne lub dotyczy bezpieczeństwa. Nie może to być jednak fikcja polegająca na mechanicznym klikaniu „zatwierdź" bez zrozumienia materiału. To częsta pułapka automatyzacji: odpowiedzialność zostaje formalnie przypisana człowiekowi, ale interfejs jest zaprojektowany tak, że staje się on jedynie pieczątką na decyzji maszyny. To nie jest nadzór – to teatralizacja odpowiedzialności.

Prawdziwy human-in-the-loop wymaga przemyślanego UX. Operator powinien mieć wgląd w źródła, uzasadnienie, poziom pewności, wykryte ryzyka i alternatywy, a także dane użyte przez model oraz ewentualne sprzeczności i skutki zatwierdzenia. Musi dysponować czasem i kompetencjami niezbędnymi do oceny, a także możliwością odrzucenia wyniku, jego poprawy, oznaczenia błędu lub eskalacji sprawy. System powinien uczyć się z tych korekt, jednak w sposób ostrożny, by nie zamienić pojedynczej decyzji w niekontrolowany trening. Interfejs nadzoru jest równie ważny co sam model; jeśli człowiek widzi jedynie estetyczny akapit i przycisk „OK", nie sprawuje nadzoru, lecz pełni rolę dekoracji audytowej.

W kontekście odpowiedzialności należy rozróżnić trzy poziomy decyzji: sugestię, rekomendację i działanie. Sugestia jest materiałem roboczym. Rekomendacja ma większą wagę, gdyż system wskazuje preferowane rozwiązanie. Działanie natomiast zmienia stan świata: zapisuje rekord,

wysyła komunikat, uruchamia proces, przyznaje uprawnienie, odmawia świadczenia lub klasyfikuje ryzyko. Im bliżej działania, tym wyższe wymagania bezpieczeństwa. Rozsądne jest rozpoczynanie wdrożeń GenAI od sugestii i wsparcia analitycznego. Największym ryzykiem jest zbyt szybkie przejście od modelu pomagającego do modelującego decyzje, podyktowane chęcią szybkiego wykazania zwrotu z inwestycji. Automatyzacja decyzji przy braku dojrzałości systemu przypomina oddanie kierownicy osobie, która świetnie opowiada o jeździe, ale nigdy nie czuła zakrętu.

Bezpieczeństwo GenAI ma również wymiar prawny i regulacyjny. Kluczowa jest zasada: organizacja pozostaje odpowiedzialna za system, który wdraża. Odpowiedzialności nie można przenieść na model, dostawcę API, prompt czy „autonomię agenta". Przy przetwarzaniu danych osobowych organizacja musi rozumieć podstawę, cel, zakres, minimalizację, retencję oraz zasady dostępu i transferu danych. Jeśli system wpływa na sytuację pracownika, klienta, pacjenta lub obywatela, niezbędna jest przejrzystość, możliwość zakwestionowania wyniku oraz kontrola człowieka. W przypadku generowania treści publicznych należy uwzględnić kwestie reputacji, dezinformacji, praw autorskich i naruszeń dóbr osobistych.

Model nie posiada osobowości prawnej; to organizacja będzie odpowiadać za jego skutki. LLM nie stanie przed komisją – w najlepszym razie wygeneruje przeprosiny. Krytyczne jest także zarządzanie logami. Są one niezbędne do audytu i debugowania, lecz mogą stać się źródłem wycieku. Zapisywanie pełnych promptów, odpowiedzi i dokumentów z retrievalu może prowadzić do przechowywania danych osobowych i tajemnic przedsiębiorstwa w systemie logowania, który nie posiadał odpowiednich uprawnień. Logowanie musi być zatem projektowane z taką samą ostrożnością jak przetwarzanie główne, z uwzględnieniem maskowania, retencji, szyfrowania i polityk usuwania. Budowanie obserwowalności poprzez tworzenie drugiego, mniej zabezpieczonego archiwum wrażliwych danych byłoby jak monitoring pożarowy zrobiony z benzyny.

Ostatnim aspektem jest zależność od dostawców modeli. Korzystając z zewnętrznego API, organizacja musi precyzyjnie wiedzieć, jakie dane są przesyłane i przetwarzane, czy służą one do trenowania, jakie gwarancje dostępności oferuje dostawca oraz jak zmiany w modelu wpływają na system. W niektórych przypadkach model lokalny lub prywatne wdrożenie będzie lepszym rozwiązaniem mimo wyższych kosztów operacyjnych. W innych wystarczy zewnętrzny dostawca, o ile dane są właściwie minimalizowane i zabezpieczone. Nie ma jednej, uniwersalnej odpowiedzi.

Bezpieczeństwo jako wielowarstwowa architektura granic

Czas na analizę ryzyka. To moment, w którym entuzjazm musi na chwilę oddać mikrofon prawnikom, specjalistom od bezpieczeństwa i architektom. Bezpieczeństwo dotyczy bowiem nie tylko infrastruktury, ale i treści generowanych przez model. Filtry harmful content są niezbędne, lecz nie rozwiązują wszystkich problemów. System może tworzyć odpowiedzi, które nie są jawnie szkodliwe, ale pozostają nieadekwatne, dyskryminujące, wprowadzające w błąd, naruszające standardy organizacji lub zbyt autorytatywne. Może również reprodukować uprzedzenia zawarte w danych treningowych bądź korpusie organizacji. Wymaga to oceny nie tylko formalnego bezpieczeństwa, ale i jakości normatywnej odpowiedzi. W obszarach takich jak HR, rekrutacja, ocena ryzyka, obsługa klienta czy administracja publiczna należy zadbać, by automatyzacja języka nie stała się automatyzacją nierówności. Model może brzmieć neutralnie, a mimo to utrwalać stronniczość. Elegancka składnia nie wybiela biasu.

Należy pamiętać również o bezpieczeństwie ekonomicznym. Zarządzanie kosztami i budżetowanie tokenów powinno być elementem projektu już od fazy zerowej. Choć wydaje się to kwestią czysto finansową, ma ono wymiar bezpieczeństwa operacyjnego. System pozbawiony limitów jest podatny na nadużycia, zapętlenia lub przeciążenia. Atakujący może generować kosztowne zapytania, błędna pętla agentowa wywołać tysiące żądań, a retry storm podwoić rachunki i obciążyć zależności. Dlatego limity tokenów, iteracji i użytkowników, alerty budżetowe oraz mechanizmy odcięcia przy przekroczeniach stanowią integralną część bezpieczeństwa. Pieniądze są również powierzchnią ataku. Kto tego nie rozumie, odkrywa to zwykle w fakturze, a faktury mają bardzo mało empatii.

Warstwowe bezpieczeństwo GenAI powinno zatem obejmować wejście, kontekst, model, narzędzia, wyjście, logi, koszty i człowieka. Na wejściu klasyfikujemy zapytania, filtrujemy dane i wykrywamy próby ataku. W kontekście ograniczamy dostęp do dokumentów zgodnie z uprawnieniami i realną potrzebą. W modelu stosujemy prompty z hierarchią instrukcji oraz właściwe parametry. W narzędziach egzekwujemy zasadę least privilege i walidację komend. Na wyjściu weryfikujemy strukturę, źródła, bezpieczeństwo i obecność danych wrażliwych. Logi minimalizujemy i zabezpieczamy, a w obszarze kosztów ustawiamy limity i alerty. W decyzjach wysokiego ryzyka wprowadzamy człowieka. Bezpieczeństwo nie jest pojedynczym modułem; to architektura granic między warstwami.

Granice te nie są wrogiem innowacji. W dojrzałych systemach to właśnie one umożliwiają sprawne działanie. Drogi mają pasy, znaki i światła nie po to, by zwalczać mobilność, lecz by ruch nie zamienił się w blacharską poezję katastrofy. Podobnie GenAI potrzebuje ograniczeń, aby mogła być

wdrażana na szerszą skalę. Organizacje nie zaufają systemom, które nie potrafią wskazać źródła odpowiedzi, uzasadnić wykonanego działania, określić osób z dostępem do danych czy umożliwić cofnięcia zmiany. Bezpieczeństwo jest warunkiem skalowania. Hype skaluje obietnicę, ale to architektura skaluje zaufanie.

GenAI jako kosztowny komponent wymagający inżynierskiego ujarzmienia

Ta analiza prowadzi do kolejnego wymiaru: ekonomii GenAI. Bezpieczeństwo chroni przed szkodą, lecz dojrzała architektura musi uwzględniać również koszty: tokeny, latencję, TCO, utrzymanie, testy i monitoring, a także ludzi w pętli, czyszczenie danych, ewaluacje oraz audyty i koszt błędów. GenAI nie jest darmową inteligencją wypływającą z gniazdka. To kosztowna infrastruktura probabilistyczna. Jeśli organizacja nie policzy ekonomii systemu, zrobi to za nią dostawca, użytkownicy lub regulator. A wtedy rachunek bywa mniej generatywny, za to bardzo rzeczywisty.

GenAI jako zdolny, lecz zawodny asystent – od metafizyki modelu do odpowiedzialnej infrastruktury

Po przejściu przez ontologię LLM, granice poznawcze, architekturę RAG, infrastrukturę danych, Pattern-Guided Coding, systemy agentowe, prompty jako kod, bezpieczeństwo i ekonomię, można sformułować tezę końcową: generatywna sztuczna inteligencja nie jest ani zbawieniem inżynierii, ani jej końcem. To nowa klasa komponentów językowych, które mogą radykalnie zwiększyć produktywność, ale tylko pod warunkiem podporządkowania ich rygorowi systemowemu. LLM nie powinien być traktowany jako samodzielny rozum, lecz jako zdolny, szybki i elokwentny, lecz zawodny asystent. Jego siła tkwi w pracy z językiem, a słabość w braku zakotwiczenia w prawdzie, świecie i odpowiedzialności. Taki asystent może być bezcenny, ale tylko wtedy, gdy nie oddamy mu kluczy do elektrowni dlatego, że pięknie opowiada o prądzie.

Materiał źródłowy konsekwentnie prowadzi do tej konkluzji: model językowy jest „badly behaving RESTful endpoint" – komponentem niedeterministycznym, kosztownym i nie w pełni przewidywalnym, podatnym na halucynacje oraz wymagającym architektonicznego ujarzmienia. To określenie stanowi najlepszą szczepionkę przeciwko mitologii AI. Nie pomniejsza ono osiągnięcia technologicznego, lecz przywraca mu właściwe miejsce. Największy błąd w myśleniu o GenAI polega na tym, że zachwyt nad interfejsem przesłania naturę systemu. Rozmawiając z modelem, ulegamy złudzeniu spotkania z partnerem poznawczym. W rzeczywistości korzystamy z

probabilistycznej usługi generowania języka, której wynik musi zostać osadzony w infrastrukturze źródeł, walidacji i nadzoru.

To przesunięcie ma charakter nie tylko techniczny, ale i kulturowy. Organizacje chętnie antropomorfizują technologie, gdyż upraszcza to kwestię odpowiedzialności. Formuły typu „agent zdecydował", „AI uznała" czy „system wie" brzmią wygodnie, lecz są pojęciowo niebezpieczne. System niczego nie „wie" w sensie instytucjonalnym, dopóki jego odpowiedź nie zostanie powiązana ze źródłem prawdy i osobą odpowiedzialną. Agent niczego nie „decyduje" normatywnie, dopóki organizacja nie określi kompetencji, granic działania oraz procedur audytu.

Język podmiotowości bywa użytecznym skrótem w rozmowie potocznej, jednak w architekturze często staje się pierwszym krokiem do rozmycia winy. Dlatego centralnym zadaniem dojrzałej inżynierii GenAI jest demistyfikacja – nie po to, by technologię ośmieszyć, lecz by uczynić ją używalną. „Agent" ma zostać przełożony na komponent, „pamięć krótkotrwała" na stan sesji, „pamięć długotrwała" na system ewidencji, „narzędzie" na adapter, a „system wieloagentowy" na architekturę komponentową.

Taka translacja odbiera AI mgłę metafizyczną i wprowadza ją w obszar odpowiedzialności. Jeśli coś jest komponentem, można określić jego kontrakt. Jeśli stanem sesji – ustalić czas życia. Jeśli systemem ewidencji – przypisać właściciela. Jeśli adapterem – ograniczyć uprawnienia. A jeśli architekturą komponentową – zaprojektować przepływy i punkty kontroli. Właśnie dlatego Pattern-Guided Coding jest czymś więcej niż tylko techniką promptowania.

GenAI wymaga rygoru klasycznej inżynierii oprogramowania, a nie zastępuje go

Jest to próba przywrócenia GenAI pamięci inżynierii. Wzorce GoF, Enterprise Integration Patterns, RabbitMQ, retry, circuit breaker, dead-letter queues, manual ACK, adaptery, strategie, komendy i walidatory nie są staromodnym balastem w epoce modeli językowych. Są sposobem na to, by modele językowe nie stały się kosztowną improwizacją. Notatka źródłowa wskazuje, że PGC ma przełamywać bariery komunikacyjne, ujawniać decyzje projektowe i zabezpieczać warstwę integracji przed najczęstszymi awariami. Jest to kluczowe, ponieważ największe ryzyka systemów GenAI często nie tkwią w samym modelu, lecz w jego połączeniach ze światem: z bazami danych, dokumentami, API, systemami ewidencji, użytkownikami i procesami organizacyjnymi. GenAI nie znosi klasycznej inżynierii oprogramowania.

Przeciwnie, czyni ją jeszcze bardziej niezbędną. Jeśli komponent jest niedeterministyczny, wymaga więcej walidacji. Jeśli jest powolny – więcej asynchroniczności. Jeśli kosztowny – rygorystycznego budżetowania. Jeśli może halucynować, potrzebuje więcej źródeł. Jeśli operuje na języku naturalnym, konieczna jest ścisła separacja instrukcji od danych. Jeśli wywołuje narzędzia, niezbędna jest kontrola uprawnień. A jeśli wpływa na decyzje, nie może zabraknąć audytu i człowieka w pętli. GenAI nie jest skrótem omijającym rygor; to technologia, która karze za jego brak szybciej, efektowniej i drożej niż większość wcześniejszych klas systemów. Najbardziej ryzykownym złudzeniem jest przekonanie, że skala modelu sama rozwiąże problemy architektury. Większy model może lepiej pisać, syntetyzować, klasyfikować i rozumieć kontekst w sensie funkcjonalnym, ale nie sprawi, że dane staną się czyste, prompty wersjonowane, retrieval trafny, a system bezpieczny i ekonomicznie kontrolowany. Nie naprawi braku właściciela dokumentów ani nie rozstrzygnie, która procedura jest obowiązująca, jeśli organizacja sama tego nie wie. Nie zagwarantuje też ochrony dostępu do dokumentu, jeśli retrieval ignoruje filtry uprawnień, i nie cofnie skutków błędnie wykonanej komendy. Moc modelu jest istotna, lecz nie zastąpi instytucjonalnego porządku. Turbina odrzutowa nie pomoże, jeśli zamontujemy ją w wózku sklepowym.

RAG stanowi tutaj przykład właściwej odpowiedzi architektonicznej. Skoro model nie ma bezpośredniego kontaktu ze źródłami prawdy, należy dostarczyć mu kontekst z kontrolowanego systemu wiedzy. Skoro same wektory bywają niewystarczające, trzeba łączyć wyszukiwanie semantyczne ze słowami kluczowymi, metadanymi, filtrami i grafami wiedzy. Jeśli dane są brudne, ETL musi być traktowany jako warunek poznawczy, a nie techniczny dodatek. Notatka wyraźnie podkreśla, że zanieczyszczone dane generują szum semantyczny i zatruwają bazę wektorową, przez co retrieval zawodzi niezależnie od klasy LLM. To jedno z najważniejszych ostrzeżeń: model może być nowoczesny, ale jeśli karmimy go śmieciami, otrzymamy śmieci w stylu eksperckim. W tym punkcie GenAI staje się testem dojrzałości organizacyjnej. Organizacja dysponująca uporządkowaną dokumentacją, dobrymi metadanymi, właścicielami procesów i kulturą testowania może wykorzystać LLM jako akcelerator wiedzy. Ta, której tego brakuje, często użyje go jako wzmacniacza chaosu.

System będzie wtedy szybciej znajdował nieaktualne procedury, płynniej streszczał sprzeczne dokumenty i bardziej przekonująco odpowiadał na podstawie przypadkowych źródeł. GenAI nie zamienia słabej instytucji w silną; ona jedynie ujawnia jej słabości, które dotąd narastały wolniej. Dlatego pytanie o wdrożenie AI powinno zaczynać się od procesu, a nie od modelu: Jaki problem rozwiązujemy? Jakie dane są źródłem prawdy? Jaki jest koszt błędu i czy wynik ma skutki prawne, finansowe lub społeczne? Czy odpowiedź można zweryfikować i czy użytkownik rozumie ograniczenia? Czy potrzebny jest człowiek w pętli? Czy dysponujemy metrykami jakości i umiemy

wykryć regresję? Czy znamy koszt zapytania i mamy procedurę wycofania promptu? Czy wiemy, co trafi do DLQ i czy system potrafi odmówić odpowiedzi przy braku danych?

Jeśli zespół nie potrafi odpowiedzieć na te pytania, nie jest w fazie produkcji, lecz w fazie życzenia. Nie oznacza to jednak, że należy hamować eksperymentowanie. Notatka słusznie rekomenduje krótkie POC jako sposób szybkiej weryfikacji hipotez i unikania inwestycji w ślepe zaułki. Eksperyment jest niezbędny, gdyż GenAI posiada obszary nieprzewidywalne analitycznie – trzeba testować modele, prompty, retrieval, chunking, UX i koszty. Jednak POC musi pozostać POC. Nie wolno mylić udanego eksperymentu z gotowością produkcyjną. Demo pokazuje, że coś może zadziałać; produkcja wymaga dowodu, że będzie działać powtarzalnie, bezpiecznie i ekonomicznie w warunkach nieidealnych. Dojrzała organizacja powinna zatem stosować dwa tryby pracy. Pierwszy to tryb eksploracyjny: szybki, badawczy i ograniczony, nastawiony na wykrywanie wartości oraz ryzyk. Drugi to tryb produkcyjny: formalny, wersjonowany, monitorowany i zabezpieczony. Najgorsze projekty powstają wtedy, gdy swoboda trybu eksploracyjnego zostaje przeniesiona do produkcji przy zachowaniu ambicji trybu produkcyjnego. Otrzymujemy wówczas system, który ma obsługiwać realnych użytkowników, ale posiada dyscyplinę notatnika. To nie jest transformacja cyfrowa. To hodowla incydentu.

Wzrost wartości krytycznego sądu nad językiem

Należy uczciwie przyznać, że GenAI zmienia strukturę pracy intelektualnej. Modele językowe realnie przyspieszają pisanie, streszczanie, klasyfikację czy analizę dokumentów, a także generowanie kodu i interakcję z bazami wiedzy. Ich wartość nie polega jednak na zastępowaniu myślenia, lecz na tym, że zmniejszają koszt wielu operacji językowych wokół myślenia. Dobrze wykorzystany LLM skraca drogę od dokumentu do wniosku, od danych do hipotezy i od chaosu notatek do uporządkowanej struktury argumentu.

To potężne narzędzie, ale pozbawione sumienia, doświadczenia czy intuicyjnego rozumienia stawki. Może wspierać pracę poznawczą, lecz nie powinno przejmować odpowiedzialności za samo poznanie. Przyszłość pracy z GenAI nie będzie więc polegać na prostym zastępowaniu ludzi modelami, lecz na przesunięciu kompetencji. Wymaga to od nas lepszego formułowania problemów, rygorystycznej oceny źródeł i projektowania procesów, w których potrafimy rozpoznawać błędy modelu i definiować kryteria jakości. Zniknie część pracy mechanicznej, ale wzrośnie wartość pracy krytycznej. Model może wygenerować tekst, lecz to człowiek musi ocenić jego sens; może zaproponować kod, ale tylko człowiek dostrzeże pułapki w architekturze; może streścić dokument, jednak to nadzorca musi wiedzieć, czy nie pominięto kluczowego wyjątku.

Skoro GenAI obniża koszt produkcji języka, rośnie wartość sądu nad tym językiem. W ujęciu społecznym jest to kwestia fundamentalna. Żyjemy w świecie nadmiaru treści, który AI tylko pogłębi. Wartość przestanie zatem leżeć w generowaniu kolejnych raportów czy analiz, a zacznie w umiejętności odróżnienia tekstu źródłowego od wtórnego, faktu od hipotezy i automatycznej syntezy od odpowiedzialnego stanowiska.

Organizacje, które wykorzystają GenAI jedynie do produkowania większej ilości mgły, utoną we własnym sukcesie ilościowym. Przewagę zyskają te, które użyją jej do porządkowania wiedzy – różnica między nimi będzie różnicą między generowaniem a rozumieniem procesu. W tym miejscu warto powrócić do filozoficznych granic LLM. Koncepcje „stochastycznej papugi", chińskiego pokoju czy problemu ugruntowania symboli nie są jedynie akademickimi rozważaniami. Przypominają one, że płynność wypowiedzi nie jest tożsama z prawdą, a prawdopodobieństwo językowe nie oznacza odpowiedzialności. W praktyce inżynieryjnej przesłanie jest jasne: nie powierzaj modelowi tego, czego nie możesz zweryfikować, jeśli koszt błędu jest wysoki. Nie zakładaj rozumienia tam, gdzie występuje jedynie jego symulacja. Nie myl pewnego tonu z pewnością epistemiczną. Nie pozwól, by składnia udawała semantykę. Syntax is not semantics – w systemach produkcyjnych to zdanie bywa warte więcej niż niejedna strategia AI.

Dojrzałe podejście do GenAI wymaga zatem podwójnej postawy: otwartości i sceptycyzmu. Otwartość bez sceptycyzmu rodzi hype i kosztowne rozczarowania, natomiast sceptycyzm bez otwartości prowadzi do paraliżu i utraty szans. Potrzebny jest sceptycyzm konstrukcyjny – taki, który nie mówi „to niemożliwe", lecz pyta: „pod jakimi warunkami będzie to bezpieczne, sprawdzalne i opłacalne?". To właśnie taki sceptycyzm jest najlepszym przyjacielem innowacji.

Wartość GenAI wynika z dojrzałości infrastruktury, a nie mocy modelu

Chroni ją przed własną propagandą. A propaganda technologiczna jest szczególnie niebezpieczna, bo często przychodzi w bluzie z kapturem i twierdzi, że nie jest propagandą, lecz "disruption". Najważniejszy praktyczny wniosek brzmi: GenAI musi zostać włączona w cykl życia oprogramowania. Prompty powinny być wersjonowane i testowane; dane czyszczone i utrzymywane; retrieval mierzony. Modele należy dobierać do funkcji, a koszty objąć limitami. Narzędzia powinny działać przez adaptery, komendy być walidowane, a błędy trafiać do kolejek. Odpowiedzi muszą mieć źródła, decyzje wysokiego ryzyka – nadzór człowieka, incydenty zaś rzetelną analizę. Zmiany powinny dopuszczać rollback.

Bez tego GenAI pozostaje sztuką widowiskowej improwizacji. Z tym może stać się infrastrukturą produktywności. Nie oznacza to, że każdy projekt musi od razu budować ciężką architekturę klasy bankowej; zasada powinna być proporcjonalna do ryzyka. W zastosowaniach osobistych, twórczych i niskiego ryzyka można pozwolić sobie na więcej swobody. W systemach organizacyjnych, gdzie model operuje na dokumentach, danych klientów czy procesach wewnętrznych, potrzebna jest większa dyscyplina. W systemach krytycznych dyscyplina musi być pełna. Nie istnieje jedna architektura dla wszystkich przypadków, lecz jedna zasada: im większy skutek błędu, tym mniejsze prawo do improwizacji. Wolność eksperymentu kończy się tam, gdzie zaczyna się cudze ryzyko.

GenAI zmienia relację między językiem a działaniem. Dotąd tekst był często opisem procesu; teraz może stać się jego interfejsem. Użytkownik pisze polecenie, model interpretuje, agent wywołuje narzędzie, a system wykonuje operację. To ogromne przesunięcie cywilizacyjne: język, który służył komunikacji, staje się narzędziem sterowania infrastrukturą. Dlatego tak kluczowa jest separacja instrukcji od danych, walidacja komend, zasada least privilege i human-in-the-loop. Kiedy słowa zaczynają poruszać systemy, trzeba uważać, czyje słowa, w jakim kontekście i z jakimi uprawnieniami są używane. W epoce agentów lingwistyka staje się częścią bezpieczeństwa.

Z perspektywy instytucjonalnej GenAI może wymusić nową odpowiedzialność za wiedzę. Dotąd wiele organizacji tolerowało chaos dokumentacyjny, ponieważ ludzie kompensowali go doświadczeniem i nieformalnymi sieciami kontaktów. Model językowy takich sieci nie posiada, o ile nie zostaną one przekształcone w dane, metadane i procedury. Aby AI działała sprawnie, organizacja musi wiedzieć, co wie, gdzie to przechowuje, kto za to odpowiada i kiedy informacja traci aktualność. To może być największy ukryty zysk wdrożeń GenAI: nie sama automatyzacja, lecz konieczność uporządkowania pamięci instytucjonalnej. Model przychodzi po dane, a znajduje bałagan.

Jeśli organizacja potraktuje ten moment poważnie, zyska więcej niż chatbota – zyska lustro. Oczywiście bywa ono nieprzyjemne. Pokazuje sprzeczne dokumenty, nieaktualne procedury, rozproszone dane i procesy zależne od kilku osób, które po prostu "wiedzą, jak to się robi". GenAI nie rozwiąże tego automatycznie, ale może uczynić problem widocznym i pilnym. Wdrożenie AI jest więc często projektem zarządzania wiedzą pod przebraniem projektu technologicznego. Kto widzi tylko model, dostrzega jedynie czubek góry lodowej. Kto widzi dane, procesy, odpowiedzialność i koszty, zaczyna rozumieć skalę masy lodu pod powierzchnią. Titanic też miał świetny marketing; problemem była architektura zaufania.

Trzeba zatem odrzucić dwie skrajności. Pierwszą jest technomesjanizm: wiara, że GenAI zastąpi ekspertów i odpowiedzialność. Drugą technosceptycyzm jałowy: przekonanie, że skoro modele nie rozumieją świata jak człowiek, są tylko efektowną zabawką. Obie postawy są intelektualnie

wygodne i praktycznie słabe. Prawda jest trudniejsza: LLM nie są ludzkim rozumem, lecz niezwykle użytecznymi maszynami językowymi.

Nie posiadają odpowiedzialności, ale mogą wspierać odpowiedzialnych ludzi. Nie gwarantują prawdy, ale przyspieszają pracę ze źródłami. Nie zastępują architektury, lecz dobrze osadzone zwiększają jej produktywność. Nie są magią, a magia nigdy nie była dobrym wymaganiem systemowym. Można zatem sformułować zasadę: GenAI daje wartość proporcjonalną nie do siły modelu, lecz do dojrzałości systemu, który go otacza. Mocny model w słabym systemie będzie produkował kosztowną niepewność. Średni model w dobrej architekturze może dawać stabilną wartość. Jakość modeli ma ogromne znaczenie, ale sama w sobie nie wystarcza. W systemach produkcyjnych zwycięża nie najładniejsza odpowiedź, lecz najlepszy układ: dane, retrieval, prompt, walidacja, bezpieczeństwo, koszt, UX i odpowiedzialność. Dlatego ostatnie zdanie tego artykułu powinno być trzeźwe, choć nie pesymistyczne.

Przejście od teorii architektury do programu wdrożeniowego

Generatywna sztuczna inteligencja to jedno z najważniejszych narzędzi współczesnej pracy z językiem, wiedzą i oprogramowaniem. Pozwala przyspieszać, porządkować i syntetyzować informacje na skalę, której jeszcze niedawno nie zakładaliśmy. Jednak dojrzałe wdrożenie wymaga odrzucenia dwóch pokus: metafizyki, która widzi w modelu cyfrowy umysł, oraz improwizacji, traktującej promptowanie jako substytut inżynierii. LLM należy traktować jak zdolnego, lecz zawodnego asystenta: zapewnić mu rzetelne źródła, jasne instrukcje, ograniczone narzędzia i kontrolowany budżet, a także wdrożyć testy, nadzór oraz możliwość odpowiedzi „nie wiem”.

Wtedy AI staje się potężnym sprzymierzeńcem produktywności. Bez tego pozostanie bardzo drogą papugą w garniturze konsultanta.

Od manifestu architektonicznego do programu działania. Warto teraz przejść do wymiaru praktycznego: jak przełożyć aparat pojęciowy na program działania dla organizacji, która nie chce już uczestniczyć w targu cudów, lecz budować realne systemy GenAI. Dotychczasowa argumentacja prowadziła przez kolejne warstwy – od natury LLM, przez RAG, dane i wzorce, aż po agentów, prompty, bezpieczeństwo i ekonomię – konkludując, że GenAI zyskuje wartość tylko w ramach dojrzałej architektury. Pytanie brzmi: co to oznacza w codziennej praktyce zarządzania projektem? Między trafną diagnozą a działającym systemem leży bowiem obszar, w którym wiele organizacji traci rozsądek, pieniądze i cierpliwość.

Pierwszym krokiem powinno być odwrócenie pytania startowego. Zamiast pytać: „jaki model wdrożymy?”, należy zapytać: „jaki proces poznawczy lub operacyjny chcemy poprawić?”. Model jest środkiem, a nie tematem projektu. Jeśli organizacja nie potrafi wskazać konkretnego procesu, mierzalnego problemu, źródeł danych, użytkowników i kosztu błędu, projekt GenAI zaczyna się od mgły. A mgła, nawet wygenerowana przez najnowszy model, pozostaje mgłą.

Kluczowe jest precyzyjne zdefiniowanie zakresu projektu: ograniczeń, wyboru narzędzi, strategii chunkingu, zestawu promptów testowych i weryfikacji UX. To nie biurokratyczna przeszkoda, lecz moment weryfikacji, czy organizacja rozwiązuje realny problem, czy jedynie ulega zachwytowi nad narzędziem.

Drugi krok to klasyfikacja ryzyka. Nie każde zastosowanie GenAI wymaga tej samej ostrożności. Inaczej projektuje się asystenta redakcyjnego, inaczej wyszukiwarkę wiedzy wewnętrznej, system analizy umów czy agenta wykonującego operacje w systemach biznesowych. Należy ustalić, czy wynik modelu ma charakter roboczy, doradczy, rekomendacyjny czy wykonawczy oraz czy błąd wywoła jedynie irytację, czy może naruszyć prawo, dane osobowe, interes klienta lub reputację organizacji. Dopiero taka analiza pozwala dobrać poziom architektury: od prostych guardraili po pełną infrastrukturę z RAG, walidacją, kolejkami, DLQ, red-teamingiem i człowiekiem w pętli. Jedna miara dla wszystkich jest wygodna tylko dla tych, którzy nie ponoszą skutków awarii.

Trzecim krokiem jest audyt danych. Organizacja musi wiedzieć, z czego model ma korzystać: które dokumenty są źródłem prawdy i są aktualne, a które mają charakter roboczy lub archiwalny. Należy określić właścicieli treści, dostępne metadane oraz zweryfikować obecność danych osobowych i poufnych informacji. Konieczne jest sprawdzenie, czy istnieją duplikaty oraz czy formaty plików pozwalają na poprawną ekstrakcję, w tym czy tabele, skany i załączniki są czytelne.

Rygor inżynierski w procesie wdrażania RAG i promptów

Bez odpowiedzi na te pytania RAG będzie loterią. Brudne dane generują szum semantyczny i zatruwają bazę wektorową, przez co nawet zaawansowany LLM nie potrafi poprawnie wyszukiwać kontekstu. To ostrzeżenie powinno stać się zasadą konstytucyjną systemów GenAI: najpierw higiena wiedzy, potem elokwencja modelu.

Czwarty krok to wybór architektury dostępu do wiedzy. W prostych zastosowaniach wystarczy wyszukiwanie semantyczne, jednak w projektach odpowiedzialnych niezbędne jest podejście hybrydowe: metadane, filtry uprawnień, wersjonowanie dokumentów, ranking źródeł i

mechanizmy wskazywania niepewności. Rekomendowane jest Integrated Knowledge Retrieval – połączenie semantyki wektorowej, słów kluczowych i grafowych baz wiedzy w obszarach, gdzie same wektory nie oddają relacji między pojęciami. Jest to kluczowe w organizacjach operujących na prawie, procedurach, dokumentacji technicznej, finansach i compliance.

W takich środowiskach podobieństwo semantyczne to zaledwie początek. Odpowiedź musi opierać się na właściwym dokumencie, aktualnej wersji i odpowiednim zakresie obowiązywania. Piąty krok to budowa minimalnego, ale uczciwego POC. Krótki Proof of Concept, nawet dwudniowy, pozwala szybko zweryfikować założenia i zapobiec rozpełzaniu się projektu. Sens takiego POC nie polega na symulowaniu produkcji, lecz na sprawdzeniu kluczowych hipotez: Czy dane są dostępne? Czy retrieval znajduje właściwe fragmenty? Czy model generuje użyteczną odpowiedź? Czy koszty są akceptowalne? Czy użytkownik rozumie wynik i czy ryzyko błędu jest dopuszczalne? Dobry POC nie musi kończyć się pokazowym sukcesem. Może przynieść odkrycie, że dane są zbyt brudne, procesy niejasne, a zastosowanie zbyt ryzykowne. To również sukces, ponieważ tani błąd jest lepszy niż kosztowny triumf na slajdach.

Szósty krok to formalizacja promptów. Jeśli POC przejdzie dalej, prompty muszą opuścić notatki i prywatne czaty, by trafić do repozytorium z przypisanymi wersjami, właścicielami, procesem review oraz możliwością rollbacku. Prompt należy traktować jak kod: z semantycznym wersjonowaniem, oddzieleniem od aplikacji i procedurą zatwierdzania. W praktyce oznacza to zmianę kultury pracy – prompt przestaje być sztuczką jednego "zaklinacza modelu", a staje się artefaktem organizacyjnym, który musi posiadać ślad, odpowiedzialność i kryteria jakości.

Siódmy krok to zaprojektowanie ścieżek błędu. Większość projektów prezentuje system w scenariuszu idealnym; produkcja natomiast pyta o sytuacje średnie, złe lub dziwne. Co, gdy model zwróci błędny format? Gdy retrieval nie znajdzie źródła lub źródła będą sprzeczne? Co w przypadku prompt injection, przekroczenia limitów kosztowych czy timeoutu API? A jeśli worker padnie po wykonaniu narzędzia, lecz przed zapisem obserwacji? Odpowiedzią na te problemy jest wykorzystanie RabbitMQ, manual ACK, competing consumers oraz wielopoziomowych Dead Letter Queues.

Dojrzała architektura nie wstydzi się ścieżek błędu – traktuje je jako dowód powagi projektu. Ósmy krok to wdrożenie bezpieczeństwa od samego początku. Nie wystarczy filtr treści i prośba w prompcie o nieujawnianie danych. Konieczna jest separacja instrukcji od danych, kontrola uprawnień w RAG, anonimizacja lub pseudonimizacja PII, walidacja komend narzędziowych, ograniczenia adapterów, zasady logowania, red-teaming i monitoring. Prompt injection, data leakage i PII shielding to kluczowe obszary, które wymagają rozwiązań systemowych, a nie tylko

deklaracji językowych. W bezpieczeństwie GenAI prompt jest ważny, ale nie stanowi muru; jest raczej tablicą z regulaminem na bramie. Mur, zamek i strażnik muszą istnieć osobno.

Dziewiąty krok to ekonomia.

Operacyjna dyscyplina wdrożenia i granice automatyzacji

Przed wdrożeniem należy precyzyjnie oszacować koszt jednostkowy i całkowity: tokeny, modele, embeddingi, bazę wektorową, ETL, monitoring, testy, ludzi w pętli, audyt, utrzymanie danych, obsługę błędów, red-teaming oraz zarządzanie incydentami. Rekomendowane jest wprowadzenie token budgeting, alertów i twardych limitów, aby zapobiegać niekontrolowanemu zużyciu zasobów, szczególnie w pętlach agentowych i tzw. retry storms. To moment, w którym wiele projektów traci romantyczny blask. I dobrze. Technologia, która nie przechodzi przez arkusz kosztów, zwykle wraca jako niespodzianka księgowa. A niespodzianki księgowe rzadko bywają poetyckie.

Dziesiąty krok to ewaluacja i monitoring dryfu. System GenAI nigdy nie jest gotowy raz na zawsze; zmieniają się modele, dane, użytkownicy, prompty, dokumenty i procesy. Niezbędne są zatem zestawy testów bazowych, red-teaming oraz stałe monitorowanie dryfu modeli i zmian w charakterystyce zapytań. Monitoring powinien obejmować jakość odpowiedzi, trafność retrievalu, groundedness, liczbę eskalacji, odrzucenia przez walidator, koszty, latencję, błędy, kolejki DLQ oraz feedback użytkowników. Bez tego system stopniowo dryfuje, a organizacja dowiaduje się o problemie dopiero wtedy, gdy użytkownicy przestają ufać rozwiązaniu lub zaczynają publikować zrzuty ekranu. Internet jest brutalnym systemem monitoringu reputacyjnego, ale nie polecam go jako pierwszej linii QA.

Jedenasty krok to właściwe zdefiniowanie roli człowieka. Human-in-the-loop nie może być fikcyjnym przyciskiem zatwierdzania. Człowiek musi mieć wgląd w źródła, ryzyka, stopień niepewności, alternatywy i potencjalne skutki decyzji. Musi dysponować odpowiednimi kompetencjami oraz czasem. W przeciwnym razie organizacja jedynie przenosi odpowiedzialność z systemu na operatora, który nie ma realnej możliwości kontroli. W dojrzałym systemie człowiek nie powinien zatwierdzać wszystkiego, gdyż wtedy automatyzacja traci sens; powinien interweniować tam, gdzie przekroczony zostaje próg ryzyka, niepewności lub skutku. Taki model wymaga przemyślanego projektowania UX, jasnych reguł eskalacji i zaufania do walidatorów. Człowiek w pętli jest zasobem krytycznym – nie należy go marnować na klikanie w oczywistości ani usuwać z miejsc, gdzie stawką jest odpowiedzialność.

Dwunasty krok to edukacja użytkowników i właścicieli procesu. System GenAI nie jest zwykłą wyszukiwarką ani prostym formularzem. Użytkownicy muszą rozumieć, że odpowiedź modelu może wymagać weryfikacji, że źródła mają kluczowe znaczenie, a danych wrażliwych nie wolno wpisywać bez potrzeby. Muszą wiedzieć, że brak odpowiedzi bywa funkcją bezpieczeństwa i że model nie jest autorytetem, lecz narzędziem. Bez edukacji pojawiają się dwa skrajne zachowania: ślepe zaufanie albo całkowita nieufność. Oba są kosztowne. Właściwe użycie GenAI wymaga nowej kompetencji: umiejętności współpracy z systemem probabilistycznym bez ulegania jego retorycznej pewności.

Trzynasty krok to decyzja o granicach automatyzacji. Nie wszystko należy i warto automatyzować za pomocą LLM. Jeśli zadanie można wykonać prostą regułą, klasycznym algorytmem, zapytaniem SQL, formularzem albo deterministycznym walidatorem, model językowy może być nadmiarowy. Jego użycie należy uzasadnić tam, gdzie istnieje realna przewaga: w pracy z językiem naturalnym, nieustrukturyzowanymi danymi, syntezie, klasyfikacji semantycznej, transformacji tekstu, dostępie do wiedzy rozproszonej czy wsparciu twórczym i analitycznym. To kluczowe: dojrzałość AI polega również na umiejętności powiedzenia „tu AI nie jest potrzebna”. Nie każdy problem jest gwoździem, nawet jeśli zarząd kupił bardzo drogi młotek.

Dyscyplina decyzji i model dojrzałości systemów GenAI

Czternastym krokiem jest dokumentacja decyzji. Każdy istotny wybór musi posiadać uzasadnienie: dlaczego wybrano RAG, jaki model embeddings i strategię chunkingu zastosowano, jakie metadane przyjęto, a także który model, temperatura, prompty, narzędzia, limity, ścieżki błędów oraz poziom nadzoru człowieka zostały określone. Notatka opisuje Topologos jako narzędzie lub skill prowadzący projektowanie przez fazy klaryfikacji, analizy wzorców, iteracyjnej akceptacji i finalnego generowania artefaktów. Kluczowa nie jest tu sama forma końcowa, lecz dyscyplina podejmowania decyzji. Architektura nie może być wynikiem jednego wielkiego promptu; powinna stanowić ślad kolejnych świadomych wyborów.

Piętnasty krok to gotowość na zmianę. System GenAI będzie ewoluował wraz ze swoim otoczeniem. Nie należy projektować go jak pomnika, lecz jak organizm z procedurą utrzymania. Niezbędne są przeglądy jakości, aktualizacje danych, rewizje promptów, analiza kosztów, testy po zmianach modeli, audyty bezpieczeństwa, feedback użytkowników oraz procedury wycofania funkcji niespełniających oczekiwań. Największym błędem byłoby wdrożyć system GenAI i uznać sprawę za zamkniętą.

To trochę jak adoptować psa i uznać, że skoro ma cztery łapy, to sam się wyprowadzi, zaszczepi i zapłaci podatek od nieruchomości. System wymaga opieki, nawet jeśli mówi płynniej niż połowa zebrania.

Z tych kroków wyłania się praktyczny model dojrzałości GenAI. Poziom pierwszy to eksperyment: model służy do szkiców, inspiracji i prostych transformacji, bez integracji z systemami krytycznymi. Poziom drugi to kontrolowane wsparcie: wykorzystanie ograniczonych danych, podstawowych promptów i prostych testów przy nadzorze człowieka. Poziom trzeci to system RAG: oparte na oczyszczonych danych i metadanych rozwiązanie z retrieval evaluation i cytowaniem źródeł.

Poziom czwarty to system agentowy: narzędzia przez adaptery, kolejki, walidacja, limity, kontrola kosztów i ścieżki błędów. Poziom piąty to system produkcyjny klasy enterprise: wersjonowane prompty, monitoring dryfu, red-teaming, governance, bezpieczeństwo, audyt oraz human-in-the-loop proporcjonalny do ryzyka i ciągłe utrzymanie.

Organizacja musi rzetelnie ocenić swój poziom. Największe ryzyko niesie sytuacja, w której poziom pierwszy jest komunikowany zarządowi jako piąty. Model ten ma znaczenie nie tylko dla firm technologicznych; dotyczy administracji publicznej, sądów, kancelarii, uczelni, szpitali, banków, organizacji społecznych, mediów i instytucji kultury. Wszędzie tam, gdzie język, dokumenty, decyzje i odpowiedzialność stanowią istotę pracy, GenAI może przynieść korzyści. Jednak w tych samych miejscach może wyrządzić szkody, jeśli zostanie wdrożona bez zrozumienia różnicy między asystentem językowym a źródłem prawdy. W instytucjach publicznych rozróżnienie to jest kluczowe, gdyż błąd systemu przestaje być jedynie usterką biznesową. Może stać się przejawem nierównego traktowania, naruszeniem praw obywatela, utratą przejrzystości lub automatyzacją uznaniowości. Dlatego program działania dla GenAI musi posiadać wymiar aksjologiczny. Nie wystarczy pytać, czy system działa. Należy pytać, czy działa sprawiedliwie, przejrzyście, proporcjonalnie, bezpiecznie i z możliwością zakwestionowania.

Wdrażaj systemy a nie modele

LLM może być narzędziem zwiększania dostępu do wiedzy, ale może też stać się instrumentem produkcji nieprzejrzystych decyzji. Może wspierać ekspertów, lecz może również maskować ich brak. Potrafi porządkować dokumentację, ale potrafi też legitymizować chaos. Zwiększa produktywność, lecz jednocześnie potęguje presję na szybsze akceptowanie niezweryfikowanych wyników. Technologia przestaje być aksjologicznie pusta w momencie włączenia jej w struktury instytucjonalne; zaczyna realizować wartości lub antywartości procesu, do którego została podłączona. Dlatego najważniejszym pytaniem dla organizacji nie jest: „czy mamy AI?”. To pytanie

jest zbyt proste – wystarczy kupić licencję, uruchomić pilotaż i zaprezentować kilka efektownych przykładów.

Ważniejsze pytanie brzmi: „czy dysponujemy odpowiedzialną architekturą dla AI?". A jeszcze istotniejsze: „czy wiemy, czego AI nie powinna robić?". Najwyższy poziom dojrzałości zaczyna się nie od katalogu funkcji, lecz od katalogu granic. To właśnie w wyznaczeniu granic organizacja dowodzi, że rozumie stawkę. Bez nich nie jest innowacyjna – Jest bezbronna wobec własnego entuzjazmu.

Można zatem zaproponować konkretną formułę: każde wdrożenie GenAI powinno posiadać kartę odpowiedzialności. Powinna ona precyzować cel systemu, krąg użytkowników, źródła danych, zakres działania, typy podejmowanych decyzji, poziom ryzyka, wersję modelu, prompty wraz z ich właścicielami, mechanizmy RAG, politykę PII, uprawnienia, narzędzia, walidatory, limity kosztów, ścieżki błędów, zasady eskalacji, testy, monitoring, procedurę rollbacku oraz osobę odpowiedzialną za utrzymanie. Taka karta nie musi być obszerna, musi być konkretna. Jej istnienie zmienia charakter dyskusji o AI z poziomu „zobaczcie, co umie" na poziom „wiemy, gdzie działa, jak działa i kto odpowiada". To jest różnica między pokazem a instytucją.

Ostatecznie praktyczny program działania sprowadza się do jednej zasady: nie wdrażaj modelu, lecz system zdolny bezpiecznie z niego korzystać. Model to tylko jeden z elementów; wokół niego muszą zostać zbudowane dane, retrieval, prompty, testy, bezpieczeństwo, kolejki, koszty, UX, nadzór i governance. Dopiero wtedy GenAI staje się technologią produkcyjną. Bez tego pozostaje jedynie błyskotliwą rozmową z maszyną – pomocną, lecz nie mogącą zastąpić odpowiedzialnej infrastruktury wiedzy. Wnioski te nie są wyłącznie filozoficzne ani czysto techniczne; stanowią program zarządzania zmianą. GenAI nie wymaga od organizacji jedynie zakupu narzędzia, lecz reorganizacji stosunku do wiedzy, danych, odpowiedzialności i błędu. To dlatego projekty AI są trudniejsze, niż obiecują sprzedawcy, a jednocześnie mogą być bardziej wartościowe, niż sądzą sceptycy. Jeśli zostaną wykonane rzetelnie, nie tylko przyspieszą pracę, ale zmuszą organizację, by wreszcie uporządkowała to, co od dawna udawała, że ma pod kontrolą.

> W świecie, gdzie płynność języka coraz częściej udaje pewność poznawczą, największym ryzykiem nie jest błąd modelu, lecz nasza skłonność do oddawania sterów komuś, kto jedynie pięknie opowiada o jeździe. Ostatecznie to nie moc procesora, lecz dojrzałość infrastruktury i odwaga w wyznaczaniu granic decydują o tym, czy budujemy narzędzie produktywności, czy kosztowną pomnikową iluzję. Czy jesteśmy zatem gotowi przestać być zaklinaczami modeli, by stać się architektami odpowiedzialności?

Inspiracje

[1]

"Building Complex Multi-Agent Systems Using" Tim OBrien

2026 | Packt Publishing | ISBN: 9781806114290

Tim O'Brien jest liderem technologicznym, architektem i programistą z ponad 20-letnim doświadczeniem w branży oprogramowania. Jest założycielem i dyrektorem technicznym wielu firm zajmujących się sztuczną inteligencją, blockchainem i technologiami zdecentralizowanymi, w tym MiraScribe i FuixLabs. Jego doświadczenie zawodowe obejmuje stanowisko starszego inżyniera w Google, gdzie specjalizował się w rozproszonych systemach sieciowych i transpilacji języków, a także role w firmach Hewlett-Packard i Cerner Corporation. O'Brien przyczynił się do innowacji w zakresie automatyzacji dokumentów, zarządzania łańcuchem dostaw i technologii finansowych. Jest ekspertem w stosowaniu najlepszych praktyk inżynierii oprogramowania w generatywnych systemach AI i agentowych, co znajduje odzwierciedlenie w jego pisaniu technicznym i doradztwie architektonicznym w zakresie tworzenia skalowalnych i bezpiecznych aplikacji AI.

Tim O'Brien is a technology leader, architect, and developer with over 20 years of experience in the software industry. He is the founder and CTO of multiple companies focused on AI, blockchain, and decentralized technologies, including MiraScribe and FuixLabs. His professional background includes serving as a Senior Engineer at Google, where he specialized in distributed network systems and language transpilation, as well as roles at Hewlett-Packard and Cerner Corporation. O'Brien has contributed to innovations in document automation, supply chain management, and fintech. He is an expert in applying software engineering best practices to generative AI and agentic systems, a focus reflected in his technical writing and architectural guidance for building scalable, secure AI applications.

Ollscoil na Gaillimhe – University of Galway

Literatura uzupełniająca

Książki

Literatura pokrewna (Google Scholar):

[1] "Seeing Things as They Are: A Theory of Perception"

John Searle

Oxford University Press | ISBN: 9780199385171

[2] "Agentic Context Systems Engineering"

Devin Albert

2025 | Independently Published | ISBN: 9798278174783

[3] "Engineering AI Agents with OpenClaw"

Adrian Coldmere

2026 | Independently Published | ISBN: 9798251853186

Treść tworzy Fundacja Dobre Państwo.
Redaguje i publikuje APA ONE, autorski system redakcyjny Fundacji oparty na AI.
APA ONE Publishing System
Fundacja Dobre Państwo © 2026
Article ID: 7ddfe903-88e9-4f44-9c3e-ae82c22eae74