

Badanie AI Testing Lab, edycja druga

AI NIE WYSTARCZY

ROLA OPERATORA W ZAPEWNIENIU JAKOŚCI TESTÓW

Spis treści

Executive summary 3

Wprowadzenie 5

Projekt badania 5

Wyniki 8

Dlaczego jedni znajdowali błędy, a inni nie 12

Model Operatora AI 19

Od wyników do wdrożenia 20

Dyskusja i implikacje 22

Wnioski końcowe 23

Aneksy 25



1. EXECUTIVE SUMMARY

Problem

W pierwszej edycji badania Sii Testing Lab sprawdziliśmy, czy AI pomaga testerom automatyzującym. Pomaga, i to wyraźnie: więcej działających testów w tym samym czasie, bez spadku średniej jakości.

Pozostała jednak kwestia, która znacznie mocniej wpływa na ryzyko. Więcej testów chroni produkcję dopiero wtedy, gdy te testy potrafią złapać istotny błąd. Można zbudować rozbudowany framework, setki scenariuszy i efektowny raport, a mimo to nie wykryć defektu, który mieści się w jednej odpowiedzi API.

Druga edycja sprawdza więc, czy większa liczba testów faktycznie oznacza lepszą ochronę produkcji.

NAJWAŻNIEJSZE WYNIKI

Badanie objęło szesnastu inżynierów testujących z pomocą narzędzi AI system, w którym celowo umieściliśmy 19 defektów.

- **AI nadal znacząco zwiększa produktywność.** Wniosek z pierwszej edycji potwierdził się w całości: modele przyspieszają tworzenie infrastruktury testowej.
- **Produkcję chroni celność testów, a nie ich liczba.** Najefektywniejsze zgłoszenie potrzebowało 7,5 przypadku testowego na wykryty defekt. Zgłoszenie z 196 przypadkami wykryło jeden defekt.
- **O wyniku decydował sposób pracy człowieka.** Warsztat inżynierski był wyrównany, a narzędzia te same, więc różnice leżały w warstwie decyzji: co testować i czym mierzyć poprawność.
- **Dwóch zasianych defektów nie potwierdził nikt.** Oba należały do warstwy przekrojowej (współbieżność, decyzje), czyli tam, gdzie awaria na produkcji kosztuje najwięcej.
- **W typowym rozwiązaniu wykryto około 6 z 19 zasianych defektów.** Skuteczność wykrywania nie nadążyła za produktywnością.
- **Im wyższy koszt defektu na produkcji, tym rzadziej go wykrywano.** Prosty błąd kontraktu złapało trzynaście zespołów z szesnastu, defektu współbieżności nie wykrył żaden.

AI wspiera wykonanie i eksplorację.

Odpowiedzialność za jakość pozostaje po stronie testera.



CO Z TEGO WYNIKA DLA ORGANIZACJI

Mierz skuteczność wykrywania, nie liczbę testów.

Liczba przypadków jest metryką pomocniczą i nie mówi o ryzyku, które zostaje na produkcji.

Wdróż standard pracy operatora.

Pięć bramek jakości to szkielet, a piętnaście praktycznych zasad to instrukcja wykonawcza.

Wbuduj bramki jakości (ang. quality gates) w proces.

Każdy test powinien mieć niezależne źródło oczekiwanego wyniku, a nie wartość przepisaną z kodu, który testuje. Trzeba też sprawdzić, czy test w ogóle potrafi nie przejść (np. po odwróceniu oczekiwanej wartości). Oba wymogi wpisz do Definition of Done i do procesu przeglądu kodu (ang. code review).

Zmierz zdolność zespołu.

Ocena (ang. assessment) w środowisku z celowo wprowadzonymi defektami pokazuje, ile rzeczywistych błędów zespół wychwytuje i w których warstwach ma luki.

2. Wprowadzenie

W pierwszej edycji badania Sii Testing Lab sprawdziliśmy, czy AI pomaga testerom automatyzującym. Pomaga, i to wielokrotnie. Porównaliśmy zespoły pracujące z AI z zespołami pracującymi bez AI i różnica była nie do przeoczenia: praca z modelem daje kilkukrotnie więcej działających testów w tym samym czasie, bez spadku średniej jakości. Wyniki pierwszej edycji pokazały wyraźny wzrost produktywności.

Zamknęliśmy tamten raport zapowiedzią, że kolejna edycja pójdzie o krok dalej i odpowie na pytanie trudniejsze, a biznesowo ważniejsze: jak używać AI, żeby maksymalizować efekty bez kompromisów. Powód był praktyczny. Więcej testów przekłada się na lepszą ochronę produkcji dopiero pod pewnymi warunkami. Można wygenerować rozbudowany framework, setki scenariuszy i efektowny raport,
a mimo to przejść obok błędu, który mieści się w jednej odpowiedzi API. Pytanie, które faktycznie decyduje o ryzyku, brzmi: czy te testy w ogóle potrafią złapać błąd, który boli.

Zmieniliśmy w związku z tym punkt wyjścia.
W pierwszej edycji badaliśmy, czy AI pomaga, więc zmienną było narzędzie. W drugiej wszystkie szesnaście zespołów pracowało z modelem, na tym samym systemie i w tym samym czasie.

W drugiej edycji rozstrzygającą zmienną stał się człowiek i sposób, w jaki prowadzi model przez pracę.



3. Projekt badania

CEL

W tej edycji wyszliśmy poza samo mierzenie tego, ile testów da się napisać z pomocą AI. Chcieliśmy zweryfikować, jak testerzy i narzędzia AI radzą sobie w realistycznym środowisku domenowym: takim, w którym trzeba przeczytać dokumentację, zrozumieć proces biznesowy, wybrać strategię, zbudować framework, napisać testy, uruchomić je, zinterpretować wyniki i odróżnić prawdziwy defekt od mylnego tropu. Nasza uwaga skupiła się na poszukiwaniu takich elementów rozwiązania, które zapewniały najlepsze rozłożenie ciężaru testów na różne obszary aplikacji oraz zapewniały efektywność kosztową i operacyjną.

HIPOTEZA

Założyliśmy, że przy wyrównanym warsztacie inżynierskim i tych samych narzędziach o wyniku decyduje sposób pracy człowieka z modelem. Różnice miały ujawnić się przede wszystkim w warstwach defektów wymagających rozumowania domenowego oraz analizy stanu, a nie w jakości samego kodu testów.

ŚRODOWISKO: LOANFLOW QA

Na potrzeby badania zbudowaliśmy własne środowisko: LoanFlow QA, aplikację do obsługi wniosków kredytowych przygotowaną jako pełny system testowy. Uczestnicy otrzymali frontend, backend API, bazę danych, mock

zewnętrznego scoringu, role użytkowników (klienta, analityka i administratora), workflow, dokumentację wymagań, specyfikację OpenAPI oraz konta i dane testowe. Było to więc realistyczne środowisko, dalekie od prostego formularza czy wyizolowanego ćwiczenia technicznego (code kata). LoanFlow zaprojektowaliśmy tak, aby samo powierzchowne testowanie nie wystarczało.

ZASIANE DEFEKTY

W systemie ukryliśmy pulę 19 celowo zasianych defektów, rozłożonych na cztery warstwy trudności: od tych widocznych po jednym zapytaniu po błędy wymagające myślenia o konsekwencjach biznesowych.

Proste defekty

Walidacja, statusy HTTP, konfiguracja, kształt odpowiedzi API.

Błędy domenowe

Wymagają zrozumienia obliczeń finansowych i procesu kredytowego, na przykład obliczania wysokości raty.

Defekty cross-functional

Stan, sesje, współbieżność, zmiany danych w czasie.

Niespójności dokumentacji

Różnice między wymaganiami, OpenAPI a rzeczywistym zachowaniem systemu.

To rozłożenie było celowe. W realnym projekcie część błędów widać po jednym curlu, ale część wymaga myślenia o konsekwencjach biznesowych. Jedno pole może istnieć, ale mieć złą wartość.
Endpoint może zwracać poprawny status, ale łamać regułę scoringu. Dokumentacja może wyglądać wiarygodnie, ale opisywać produkt albo endpoint, którego system w ogóle nie ma. Błędy oprócz swoich warstw trudności były równomiernie rozłożone według kategorii oraz części funkcjonalności. Taki sposób rozłożenia defektów miał w założeniu zagwarantować istnienie korelacji pomiędzy liczbą wykrytych defektów, a jakością pokrycia testami.

Opis	Kategoria	Część aplikacji
Słaba walidacja e-mail (akceptuje 'user@')	Autoryzacja i konto	API - kontrakt
POST zwraca 200 zamiast 201	Wnioski kredytowe (API/CRUD)	API - kontrakt
Paginacja bez tiebreakera (duplikaty)	Wnioski kredytowe (API/CRUD)	API - kontrakt
Submit aktywny przy niepoprawnym formularzu	Frontend/UI	UI
password_hash w odpowiedzi profilu	Autoryzacja i konto	API - kontrakt
/admin/stats przed CORS	Admin i produkty	API - kontrakt
DTI od brutto zamiast netto	Kalkulacje finansowe	UI
Akceptacja mimo rekomendacji REJECT	Proces analityka	API - proces
Prowizja ignoruje próg obniżki	Kalkulacje finansowe	UI
Math.floor na latach zatrudnienia	Wnioski kredytowe (API/CRUD)	API - kontrakt
Rata dla termMonths + 1 okresów	Kalkulacje finansowe	UI
Anulowane wnioski na liście analityka	Proces analityka	UI
Brak SELECT FOR UPDATE (race)	Wnioski kredytowe (API/CRUD)	API - proces
Brak przeliczenia po edycji produktu	Admin i produkty	API - proces
Stary token ważny po zmianie hasła	Autoryzacja i konto	API - proces
Limit próśb o dokumenty per analityk	Proces analityka	API - proces
Spec: employmentYears string vs number	Dokumentacja	Dokumentacja
Widmowy 'Kredyt Konsolidacyjny'	Dokumentacja	Dokumentacja

UCZESTNICY I ZAKRES OCENY

Oceniono 16 zgłoszeń. Każde zawierało zwykle własny framework testowy, a w nim: testy, raporty, zgłoszenia błędów, opis strategii oraz ślady pracy z AI. Te ostatnie były szczególnie ważne dla rzetelności analizy, bo interesował nas zarówno finalny kod, jak i proces, który doprowadził do jego powstania.

Każdy raport błędu został zweryfikowany względem kodu LoanFlow, dokumentacji uczestnika, OpenAPI i wewnętrznego answer key. Łącznie sprawdzono około 350 plików z raportami błędów. Podobnie jak w poprzednim badaniu, ocena miała dwa wymiary. Dlatego w szablonie dla uczestników znalazły się także katalogi na ślady pracy z AI oraz post-mortem.

To rozdzielenie okazało się kluczowe. Sam ranking błędów mówiłby nam, kto znalazł więcej defektów. Analiza procesu rozszerza ten wymiar, mówiąc nam dodatkowo, dlaczego.



EFEKT

Ile zasianych błędów udało się znaleźć, jakiej były trudności i czy zgłoszenia były poprawne.



PROCES

Jak uczestnik pracował z AI: czy budował kontekst, prowadził model przez specyfikację, weryfikował odpowiedzi, odrzucał błędne sugestie i raportował defekty w sposób powtarzalny.

OGRANICZENIA BADANIA

Granice wniosków nazywamy wprost: szesnaście zgłoszeń, jedna domena, mieszane narzędzia. Opisane dalej praktyki to wzorce konsekwentnie obserwowane u skutecznych zespołów i nieobecne u pozostałych. Traktujemy je jako korelacje, bez roszczenia do dowodu przyczynowego. Dlatego raportujemy mechanizmy i sceny, a nie współczynniki.

Wstępny ranking i późniejsza szczegółowa weryfikacja nie były identyczne. To normalne w takim badaniu: część raportów łączyła kilka zjawisk, część dotyczyła realnych defektów niezasianych, a część wyglądała podobnie do defektu z klucza odpowiedzi, ale po inspekcji okazywała się innym problemem. Dobrym przykładem był błąd przejmowania wniosku przez drugiego analityka. To był realny, często zgłaszany defekt, ale nie był tym samym co zasiany race condition na równoległej decyzji analityka. Taka różnica ma znaczenie, bo pokazuje, że ewaluacja bug huntingu też wymaga precyzyjnego klucza odpowiedzi.

Jednocześnie uczestnicy znaleźli wiele realnych, nieplanowanych defektów: rozbieżności między dokumentacją a implementacją, błędy walidacji, problemy autoryzacji dokumentów, niespójności UI, przypadki zapisu niepoprawnych wartości przed walidacją czy endpointy zwracające błąd 500 przy niepoprawnym parametrze. To potwierdziło, że środowisko było wystarczająco realistyczne. Uczestnicy nie tylko trafiali w nasze zasiane punkty kontrolne, ale odkrywali też realne skutki decyzji implementacyjnych.

4. Wyniki

Produktywność została potwierdzona

Większość uczestników zbudowała pełnoprawny framework testów automatycznych: najczęściej Playwright z TypeScriptem, czasem Java, C# lub pytest.

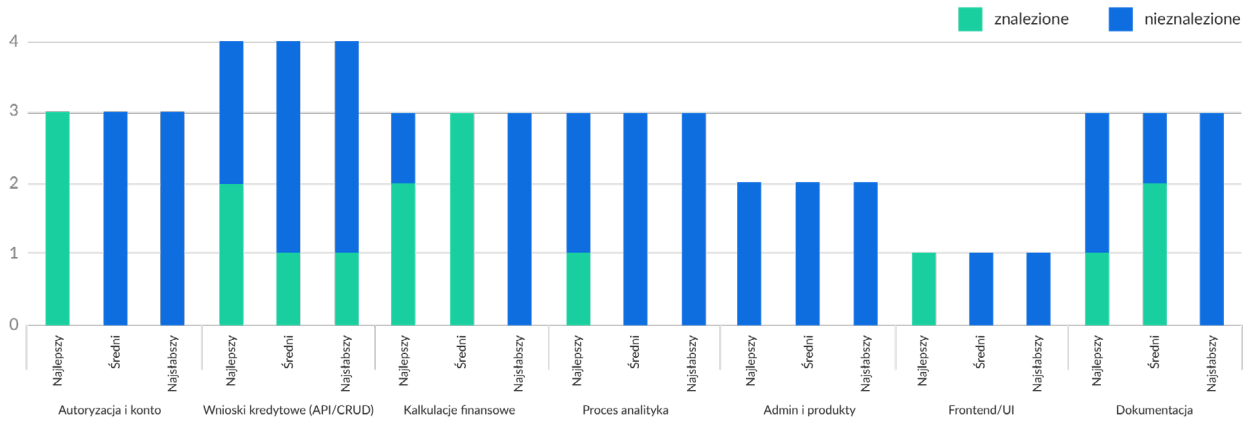
Pojawiały się page objecty, fixture'y, helpery, wrappery API, raporty HTML, pliki kontekstowe dla AI i skrypty uruchamiane jednym poleceniem. Pod względem inżynieryjnym większość repozytoriów spełniała standardy projektu produkcyjnego, a mediana jakości procesu była dość wysoka. Uczestnicy wykraczali poza generowanie pojedynczych testów i potrafili zbudować strukturę projektu. To potwierdza wniosek z pierwszej edycji: AI znacząco przyspiesza tworzenie infrastruktury testowej.

Skuteczność wykrywania błędów

Skuteczność wykrywania błędów nie rosła jednak proporcjonalnie do liczby testów. Typowe zgłoszenie znajdowało około 6 z 19 zasianych defektów. Najczęściej były to błędy kontraktowe i obliczeniowe: niepoprawny status HTTP, zła rata, prowizja liczona według niewłaściwej stawki, brak produktu opisanego w wymaganiach albo endpoint zadeklarowany w OpenAPI, którego backend nie implementował.

W trudniejszych warstwach wykrywalność była znacznie niższa. Dotyczy to współbieżności, zmian stanu, sesji po zmianie hasła, zależności decyzji analityka od scoringu, limitów dokumentów po zmianie analityka oraz stabilności paginacji przy identycznych timestampach.

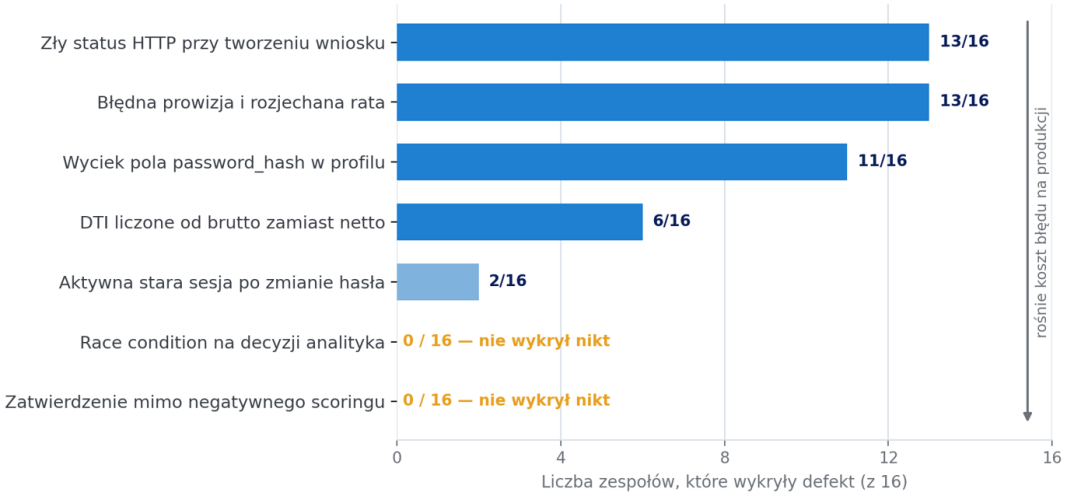
Dwóch zasianych błędów nie potwierdził w raportach nikt. To były dokładnie te obszary, których nie łapie zwykłe pokrycie samych ścieżek pozytywnych (ang. happy path) ani test sprawdzający tylko kształt odpowiedzi.



Wykres 1. Pokrycie testami w formie ilości znalezionych błędów do wszystkich błędów.

Krzywa ryzyka

Wykrywalność spadała według wyraźnego wzorca: dokładnie tam, gdzie rośnie cena błędu na produkcji. Zły status HTTP przy tworzeniu wniosku złapało trzynastu z szesnastu inżynierów. Błędną prowizję i rozjechaną ratę wykryło również trzynastu. DTI liczone od dochodu brutto zamiast netto wykryło już tylko sześciu. Aktywną starą sesję po zmianie hasła zauważyło dwóch. Race condition na równoległej decyzji analityka oraz zatwierdzenia wniosku wbrew negatywnemu scoringowi nie wykrył nikt.



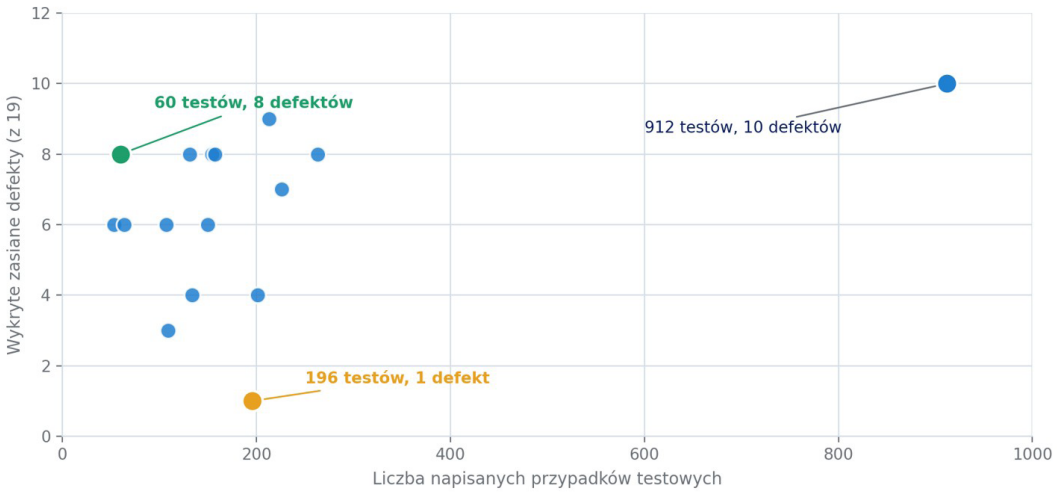
Wykres 2. Wykrywalność zasianych defektów w 16 zespołach.

Proste błędy kontraktowe były wykrywane przez większość zespołów. Znacznie większe różnice pojawiały się przy bardziej złożonych defektach, które kosztują najwięcej, gdy trafią na produkcję, a które wykrywają tylko nieliczni. To był pierwszy mocny sygnał:

AI pomaga zbudować testy, ale nie gwarantuje, że będą one testowały właściwe rzeczy.

Efektywność pokrycia

Po analizie zestawów testowych pojawił się ciekawy wniosek: większość uczestników nie napisała za dużo testów w rozumieniu pustego wypełniacza. Stosunek testów realnie potrzebnych do całkowitej liczby przypadków był w wielu repozytoriach wysoki. Daleko więc było do historii o setkach kompletnie losowych testów wygenerowanych przez AI. Znaczenie miało nie tylko to, ile testów powstało, ale przede wszystkim jakie obszary i warunki sprawdzały. Słabsze repozytoria potrafiły mieć dużo testów, ale testowały statusy, widoczność elementów i kształt JSON-a, nie wartości biznesowe. Nie sprawdzały obliczeń. Nie asertowały braku pól zabronionych. Nie budowały scenariuszy wieloaktorskich. Nie próbowały równoległości. Nie wchodziły w momenty, w których dane zmieniają się po utworzeniu wniosku.



Wykres 3. Liczba napisanych przypadków testowych a liczba wykrytych zasianych defektów (16 zgłoszeń).

Zgłoszenie z 60 przypadkami testowymi wykryło osiem defektów. Zgłoszenie z 196 przypadkami wykryło jeden. Mniejszy zestaw testów z precyzyjnymi asercjami wykrywał więcej defektów niż większe zestawy ze słabszymi asercjami.

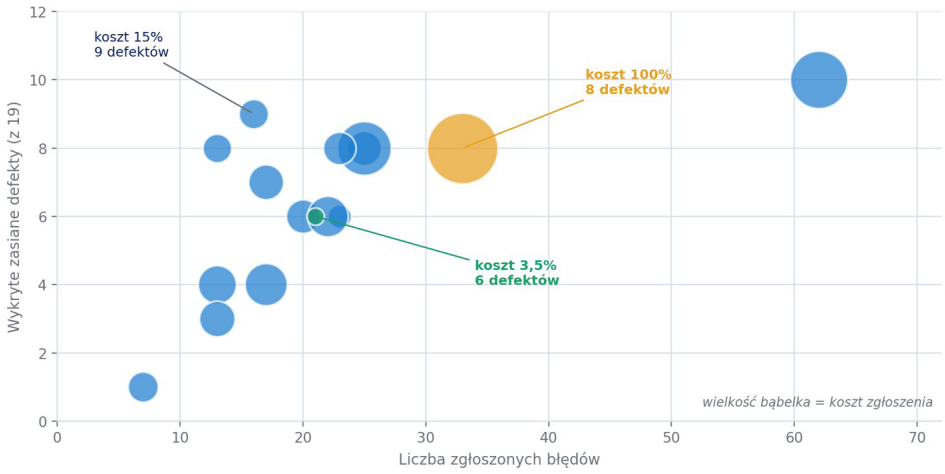
Profil zgłoszenia	Przypadków na 1 wykryty defekt	Pozycja w grupie wykrywalności
Najszczuplejsze („ostre asercje”)	7,5	blisko czołówki: 8 defektów wobec 10 u najlepszego zgłoszenia, przy ok. 1/15 przypadków testowych (60 wobec 912)
Typowe zgłoszenie	ok. 25–40	środek grupy
Najcięższe („dużo testów”)	od ok. 90 do 196	zróżnicowana: od 10 defektów (912 przypadków) do 1 (196 przypadków)

To ma bezpośrednie przełożenie na koszt i utrzymanie. Mały, precyzyjny zestaw testów jest tańszy w utrzymaniu i skuteczniejszy niż duży framework, który nie sprawdza właściwych hipotez. Sama liczba przypadków testowych nie jest dobrą miarą skuteczności testów.

Koszt a skuteczność

Trzecim wymiarem, obok liczby testów i wykrywalności, jest koszt zgłoszenia. Wielkość bąbelka na wykresie poniżej odpowiada kosztowi, w skali względnej, w której 100% oznacza zgłoszenie najdroższe w stawce. Koszt przedstawiony na wykresie został oszacowany na podstawie kosztu tokenów przy wykorzystaniu API Anthropic.

Wartości zostały znormalizowane do skali względnej, w której 100% oznacza najdroższe zgłoszenie. W praktyce rzeczywiste koszty uczestników były niższe, ponieważ korzystali oni głównie z planów abonamentowych (np. Claude Max czy ChatGPT), a nie z rozliczeń API.



Wykres 4. Zgłoszone błędy, wykryte defekty zasiane oraz koszt zgłoszenia (16 zgłoszeń).

Rozrzut kosztu okazał się szerszy niż rozrzut wyników. Pięć zespołów wykryło dokładnie osiem zasianych defektów, koszty tych zgłoszeń wyniosły kolejno 14,1%, 19,1%, 21,3%, 56,6% oraz 100%. Ten sam efekt kosztował więc siedem razy więcej albo siedem razy mniej, zależnie od sposobu pracy. Najtańsze zgłoszenie w stawce, o koszcie 3,5%, wykryło sześć defektów, czyli tyle, co typowe zgłoszenie w badaniu. Zgłoszenie o koszcie 15% wykryło dziewięć defektów i wypadło lepiej niż najdroższe w całym badaniu, które przy koszcie 100% zatrzymało się na ośmiu. Przeliczone na jeden wykryty defekt koszty różnią się między zgłoszeniami blisko trzydziestokrotnie.

Co naprawdę różnicowało wyniki

Warsztat był wyrównany. To, co różnicowało skuteczność, leżało w warstwie decyzji: co testować, czym mierzyć poprawność i jak prowadzić AI. Praca agentów AI wymaga zatem kontroli na poziomie całego procesu, a nie tylko oceny jakości i ilości napisanego kodu.

INTERPRETACJA **WYNIKÓW**

- Produktywność nie przekłada się automatycznie na skuteczność. Więcej testów nie oznaczało więcej złapanych błędów.
- Trudność wykrycia rośnie wraz z kosztem biznesowym defektu. Najdroższe błędy były wykrywane najrzadziej albo wcale.
- Różnice między zespołami nie wynikały z narzędzi ani z poziomu warsztatu, lecz ze sposobu pracy z modelem.
- Celność pokrycia ważyła więcej niż jego rozmiar.

5. Dlaczego jedni znajdowali błędy, a inni nie

Dobrze pokazuje to jeden z przypadków zaobserwowanych w badaniu. Niezależna wyrocznia jednej z uczestniczek wyliczyła ratę annuitetową 315,68, podczas gdy API zwracało 308,18. Model zaproponował dostosowanie oczekiwanej wartości do wyniku zwracanego przez aplikację.

Gdyby uczestniczka się zgodziła, rozbieżność zniknęłaby bez śladu, a wraz z nią zasiany defekt. Zamiast tego przeliczyła ratę niezależnie, trzecim narzędziem, potwierdziła, że to backend liczy źle, i zgłosiła błąd.



Kiedy wyrocznia i system się różnią,
nie wolno domyślnie zakładać, **że myli się wyrocznia.**

W tej jednej decyzji mieści się różnica między akceptowaniem wyników AI a operowaniem procesem jakości. Kiedy warsztat jest wyrównany, o wyniku decyduje sposób pracy. U zespołów skutecznych konsekwentnie wracało pięć praktyk, których u pozostałych brakowało. Każda jest bramką nałożoną na pracę modelu: punktem, w którym człowiek nie przyjmuje wyniku, tylko go sprawdza. To one są operacyjną definicją kompetencji, którą oferujemy klientom.

BRAMKA 1

Niezależna wyrocznia poprawności

PROBLEM

Test, który sprawdza obecność pola, potwierdza jedynie, że system coś zwrócił. Test z niezależną wyrocznią sprawdza, czy zwrócona wartość jest poprawna. Zespoły bez wyroczni potrafiły mieć rozbudowane frameworki, ale ich asercje kończyły się na statusie HTTP, widoczności elementu lub zgodności schematu. W takim modelu aplikacja może zwracać błędną ratę przez cały czas, a testy nadal będą zielone.

SCENA Z BADANIA

W LoanFlow było to szczególnie widoczne w obliczeniach finansowych. Wymagania opisywały formułę raty annuitetowej, sposób liczenia DTI oraz reguły prowizji. Dobre testy szły dalej niż samo sprawdzenie, czy pole monthlyInstallment istnieje: same liczyły oczekiwaną ratę, prowizję albo wskaźnik DTI, a następnie porównywały wynik z odpowiedzią API. Właśnie dlatego zespoły z niezależną wyrocznią znacznie częściej znajdowały błędy finansowe.

MECHANIZM

Wyrocznia poprawności (correctness oracle) wylicza oczekiwany wynik na podstawie specyfikacji, a nie na podstawie zachowania aplikacji. Pełna formuła nie zawsze była przy tym konieczna. W niektórych przypadkach wystarczał metamorphic invariant, czyli prosta własność, która musi być prawdziwa niezależnie od szczegółów implementacji. Przykład: przy dodatnim oprocentowaniu miesięczna rata powinna być większa niż kwota kredytu podzielona przez liczbę miesięcy. Taki test obywa się bez pełnej formuły annuitetowej, a mimo to wykrywa klasę błędów, której zwykły test UI nie zauważy.

REKOMENDACJA

Jeśli wartość ma znaczenie biznesowe, policz ją niezależnie, zamiast pytać aplikację, jaka powinna być.
To wymaganie procesowe, obowiązujące niezależnie od technologii.

BRAMKA 2

Asercje na właściwej warstwie: API przed UI

PROBLEM

UI jest ważny, ale pieniądze, statusy, uprawnienia i decyzje systemowe najczęściej żyją w API. Jeśli testy zatrzymują się na ekranie, mogą nie zobaczyć tego, co naprawdę decyduje o ryzyku. Pokrycie UI bez sondowania API daje fałszywy komfort dokładnie tam, gdzie ryzyko jest największe.

SCENA Z BADANIA

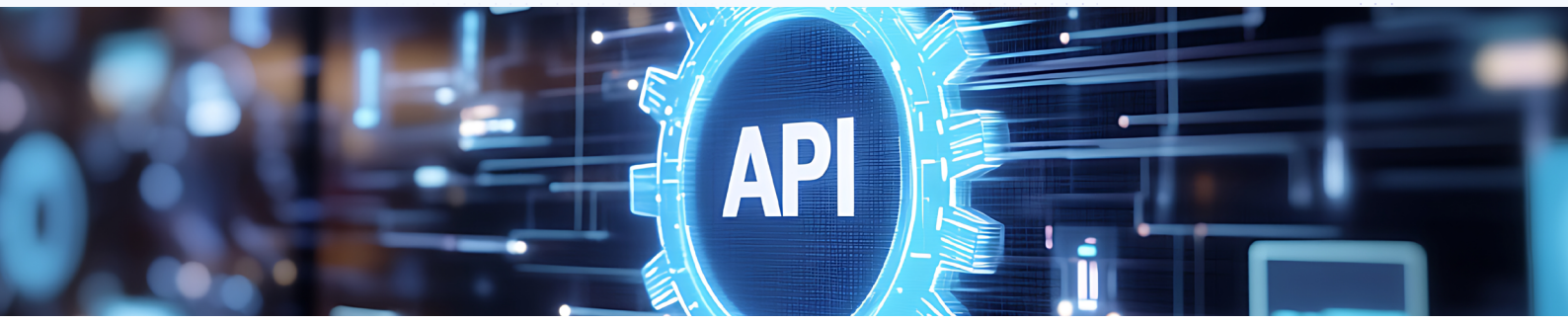
W LoanFlow frontend w niektórych miejscach celowo liczył wartości poprawnie, ale backend zwracał wartości błędne. Tester, który sprawdzał tylko UI, mógł więc dostać fałszywe poczucie bezpieczeństwa. Asercja przechodziła, bo sprawdzała warstwę, która akurat była poprawna. Błąd żył niżej, w API.

MECHANIZM

Dobre zgłoszenia często zaczynały od sondy API: zaloguj użytkownika, wywołaj endpoint, zobacz odpowiedź, porównaj z dokumentacją, dopiero potem zamknij to w teście automatycznym. Ten tryb pracy miał dwie zalety. Po pierwsze, szybko oddzielał błąd aplikacji od błędu w założeniu testu. Po drugie, zmuszał do sprawdzenia danych, a nie tylko przepływu ekranów.

REKOMENDACJA

Dla wartości biznesowych asercja powinna sięgać warstwy API. Test interfejsu weryfikuje prezentację, nie poprawność wyliczenia.



BRAMKA 3

Osobny przebieg adwersarialny

PROBLEM

Wiele zespołów robiło naturalną dla nich rzecz: czytało wymagania, pisało testy pokrywające funkcje, a błędy znajdowało przy okazji. To dawało solidny baseline, ale szybko osiągało sufit. Testy pokrycia zwykle potwierdzają, że system działa w zakładany sposób. Bug hunting, będący praktycznym przejawem testerskiego myślenia, wymaga innego nastawienia: trzeba aktywnie szukać sposobu, w jaki system może się złamać.

SCENA Z BADANIA

Najlepsze strategie oddzielały te dwa tryby. Najpierw budowano podstawowe pokrycie: logowanie, tworzenie wniosku, workflow klienta, analityka i administratora, upload dokumentów, walidacje. Potem uruchamiano osobny adversarial pass, którego celem było zaatakowanie konkretnych ryzyk:

- Ryzyko „scoring może nie blokować decyzji" prowadzi do scenariusza: utwórz wniosek z negatywnym scoringiem, przypisz analityka, spróbuj zatwierdzić, oczekuj blokady.
- Ryzyko „zmiana produktu może nie wpływać na istniejące wnioski" prowadzi do scenariusza cross-role: klient tworzy wniosek, administrator zmienia parametry produktu, analityk próbuje podjąć decyzję.
- Ryzyko „limit żądania dokumentów może być źle liczony" prowadzi do scenariusza z dwoma analitykami, zmianą przypisania i ponownym żądaniem dokumentów.

MECHANIZM

Rejestr ryzyk jest wejściem do tej bramki. To dobra i potrzebna praktyka, bo daje przebiegowi adwersaryjnemu listę celów i porządkuje myślenie o tym, gdzie system może się złamać. Jego wartość jest jednak warunkowa: rejestr chroni produkcję dopiero wtedy, gdy jest rozliczany do zera, czyli każde nazwane ryzyko domyka się scenariuszem z asercją zdolną je obalić i przechodzącą „próbę zębów". Rejestr z pozycjami bez powiązanych, failowalnych testów pozostaje spisem życzeń. Dlatego traktujemy identyfikowalność: ryzyko – scenariusz – asercja jako element definicji ukończenia, a nie jako dobrowolny dodatek.

Jak łatwo o złudzenie, że nazwanie ryzyka wystarczy, pokazuje wynik jednego z uczestników, u którego spośród sześciu zasianych defektów cztery niewykryte leżały w obszarach wprost wskazanych w jego własnym rejestrze ryzyk. Mechanizm był za każdym razem ten sam: ryzyko nazwane, test napisany, test nawet dotykał właściwego obszaru, ale przechodził na zielono, bo nigdy nie asertował samego naruszenia. Oba defekty, których nie znalazł nikt, leżały dokładnie tu.

REKOMENDACJA

Recepta jest czysto mechaniczna: ryzyko tłumaczymy na scenariusz naruszenia z asertowaną odmową, zanim trafi do modelu, a gotowy test poddajemy próbie zębów, czyli odwracamy oczekiwaną wartość i sprawdzamy, czy zakończy się wynikiem negatywnym. Jeżeli zmiana oczekiwanej wartości nie powoduje niepowodzenia testu, jego asercje są niewystarczające. Część defektów pozostała nieznaleziona właśnie dlatego, że wymagała lateralnego myślenia: równoległych requestów, cross-origin probe'a, dwóch aktorów, zmiany stanu w czasie albo testu stabilności paginacji. Takie testy trzeba wymyślić, a to najczęściej wynika z doświadczenia samego testera.



BRAMKA 4

Krótki sweep bezpieczeństwa i higieny odpowiedzi

PROBLEM

Strategia skoncentrowana na wymaganiach biznesowych, przepływach i kontrakcie pozytywnym pomija całą klasę błędów tanich do wykrycia, a jednocześnie krytycznych wizerunkowo i prawnie.

SCENA Z BADANIA

Jednym z najbardziej pouczających przykładów był wyciek password_hash w odpowiedzi profilu użytkownika. Technicznie to bardzo tani błąd do wykrycia: wystarczy zalogować się, wywołać /auth/profile i sprawdzić, czy odpowiedź nie zawiera pola z hashem. A jednak pominęło go pięć z szesnastu zespołów. Powodem był kształt strategii, która sprawdzała, czy profil działa, ale nie zawierała przeglądu odpowiedzi pod kątem pól, których nigdy nie powinno tam być.

MECHANIZM

Response-hygiene sweep uzupełnia testy domenowe i pozostaje od nich niezależny. Właśnie dlatego łapie błędy, które łatwo ominąć, gdy cały proces skupia się na business flow. Po podstawowym pokryciu warto poświęcić osobny, krótki przebieg na pytania:

- czy odpowiedzi API nie wyciekają pól technicznych lub wrażliwych,
- czy endpointy chronione naprawdę wymagają autoryzacji,
- czy CORS zachowuje się spójnie,
- czy walidacja nie przepuszcza oczywistych śmieci,
- czy role użytkowników nie widzą danych spoza swojego zakresu.

REKOMENDACJA

To jedna z najtańszych praktyk do powielenia. Zaplanuj ją jako osobny przebieg po zbudowaniu podstawowego pokrycia.

BRAMKA 5

Zielony test nie jest dowodem

PROBLEM

Najważniejsze pojęcie, które wracało w analizie, to greenwashing. Greenwashing w testach oznacza sytuację, w której testy świecą na zielono, ale nie dowodzą niczego istotnego. Czasem dlatego, że asercja jest zbyt słaba. Czasem dlatego, że test sprawdza niewłaściwą warstwę. Czasem dlatego, że AI naprawiło test, dopasowując oczekiwania do błędnego zachowania aplikacji. Model bardzo dobrze poprawia testy tak, żeby przechodziły. Problem w tym, że „żeby przechodziły" i „żeby były prawdziwe" to dwa różne cele.

SCENA Z BADANIA

W zgłoszeniach widzieliśmy kilka powtarzalnych wzorców:

- test tworzenia wniosku akceptował zarówno 200, jak i 201, więc maskował błąd statusu HTTP,
- test profilu sprawdzał tylko kilka oczekiwanych pól i nie dokonywał asercji braku password_hash,
- test zmiany hasła sprawdzał, czy stare hasło już nie działa, ale nie sprawdzał, czy stary token JWT został unieważniony,
- test DTI porównywał dane z innym wynikiem aplikacji, zamiast z niezależną wartością oczekiwaną,
- część czerwonych testów była opisywana jako „flaky", zanim sprawdzono, czy przypadkiem nie wskazują realnego błędu.

MECHANIZM

Najlepsze procesy miały zabezpieczenia przeciw temu zjawisku. Pojawiały się red-by-design repros, czyli testy opisujące poprawne zachowanie i oznaczone jako oczekiwane niepowodzenie do czasu naprawy buga. Pojawiały się xfail, Ignore, osobne katalogi bug-huntingowe, lintery asercji, mutation-checki, reguły w plikach kontekstowych typu „nie osłabiaj asercji tylko po to, żeby test przeszedł" oraz niezależna ponowna weryfikacja przez drugiego agenta albo człowieka.

Greenwashing ma też lustrzane odbicie, które nazwaliśmy red-washingiem: wycofanie realnego defektu jako niereprodukowalnego bez wykonania weryfikacji. W badaniu jeden z zespołów poprawnie wykrył krytyczną rozbieżność kontraktu, czyli endpoint istniejący w specyfikacji, a zakazany w wymaganiach, po czym zamknął zgłoszenie na podstawie zakładanego, nigdy nieuruchomionego wyniku. Wycofany prawdziwy błąd kosztuje dokładnie tyle samo co ukryty.

REKOMENDACJA

Dobry test powinien być restrykcyjny. Powinien natychmiast failować, kiedy system łamie zdefiniowane wymaganie. Czerwony wynik to hipoteza o błędzie, dopóki weryfikacja nie wykaże, że test jest flaky. Model nie może osłabiać asercji, żeby test przeszedł, a „nie reprodukuje się" to zadanie do wykonania.

BOOKLET OPERATORA AI

Każdą z tych pięciu praktyk sprowadziliśmy do konkretnych zasad wykonawczych, wyprowadzonych wprost z repozytoriów uczestników.

Pełna checklista 15 zasad jest dostępna do pobrania.



6. Model Operatora AI

Najważniejsza teza tej edycji brzmi tak: AI potrzebuje od człowieka czegoś więcej niż akceptacji. Rola akceptującego, sprowadzająca się do tego, by poprosić o testy, odebrać kod, uruchomić go i ucieszyć się zielonym raportem, jest wobec modelu redundantna, bo model potrafi wypełnić ją sam. W trybie autopilota LLM jest jednocześnie wykonawcą i akceptującym: generuje rozwiązanie i sam przyjmuje je za dobre. Człowiek, który dokłada do tego wyłącznie własną akceptację, powiela pracę, którą model wykonuje sam, a przecież zielony wynik testów nie potwierdza jeszcze ich skuteczności.

Dlatego rola człowieka nie powinna ograniczać się do akceptowania wyników modelu. Potrzebuje operatora: kogoś, kto prowadzi model i weryfikuje jego wyniki, zamiast ufać mu domyślnie. Operator definiuje wyrocznię poprawności, wybiera ryzyka i projektuje ataki, pilnuje, żeby asercje miały zęby, wymusza pełną reprodukcję
i oddziela realny defekt od szumu, czyli robi dokładnie to, czego model sam z siebie nie wniesie.
Te dwa podejścia różnią się przede wszystkim zakresem odpowiedzialności człowieka:

Akceptujący	Operator
Prosi model o testy i przyjmuje kod	Prowadzi model przez domenę, ryzyka i specyfikację
Cieszy się zielonym raportem	Traktuje zielony wynik jako twierdzenie do sfalsyfikowania
Poprawia błędy techniczne	Wymaga wyroczni, pilnuje asercji, wymusza pełną reprodukcję
Czerwony test opisuje jako „flaky”	Czerwony test traktuje jako hipotezę o błędzie
Miarą sukcesu jest liczba testów	Miarą sukcesu jest zdolność testów do wykrywania rzeczywistych defektów

Lewa kolumna to zadania, które model wykona bez nas. Prawa to kompetencja, którą w tym badaniu zdefiniowaliśmy i nauczyliśmy się mierzyć, odrębna od umiejętności promptowania. Druga część badania bardzo mocno potwierdziła, że praca z AI w testach jest osobną kompetencją. Skuteczny tester w świecie AI jest przede wszystkim operatorem procesu jakości, a samo umiejętne promptowanie to za mało.

W najlepszych przykładach człowiek robił rzeczy, których model nie gwarantuje sam z siebie:

- zauważał, że AI próbuje dopasować wyrocznię do błędnego wyniku aplikacji,
- zatrzymywał zgłoszenia false positives,
- wymuszał odtworzenie dokładnej ścieżki użytkownika, a nie tylko syntetycznego requestu,
- obniżał lub podnosił severity na podstawie wpływu biznesowego,
- decydował, kiedy porzucić rabbit-hole i zmienić strategię.

Zmienia to rolę testera automatyzującego. Coraz mniej chodzi o ręczne produkowanie kodu, a coraz bardziej o projektowanie procesu pozwalającego dowieść osiągnięcia oczekiwanych poziomów jakości: jakie hipotezy sprawdzamy, czym mierzymy poprawność, jak bronić się przed fałszywą zielenią i gdzie wykorzystanie modelu daje największy zwrot.

AI jest wtedy wykonawcą i partnerem eksploracji.
Właścicielem procesu pozostaje tester.



7. Od wyników do wdrożenia

Wnioski przełożyliśmy na trzy elementy gotowe do użycia w projektach.

Standard pracy operatora

Standard pracy operatora przekłada Model Operatora AI z sekcji 6 na praktykę. Szkielet to 5 bramek, a instrukcja wykonawcza to 15 praktycznych zasad. Każda zasada wynika z konkretnego wzorca zaobserwowanego w repozytoriach uczestników. Zespół dostaje sprawdzony sposób pracy, który można wdrożyć od pierwszego dnia.

Bramki jako quality gates w procesie

Każdą z pięciu bramek można osadzić w definicji ukończenia zadania testowego i w code review: od wymogu niezależnej wyroczni dla wartości biznesowych po próbę zębów, czyli odwrócenie oczekiwanej wartości i sprawdzenie, czy test potrafi sfailować. Jakość przestaje wtedy zależeć od dobrej woli pojedynczego inżyniera i staje się warunkiem, który trzeba spełnić.

Assessment istniejących zespołów

Metodologia badania, czyli środowisko z zasianymi defektami, klucz odpowiedzi i weryfikacja raportów względem kodu, pozwala zmierzyć w zespole klienta to samo, co zmierzyliśmy tutaj: zdolność wykrywania błędów, które kosztują. Wynikiem jest konkretna liczba: ile realnych defektów zespół wychwytuje i w których warstwach trudności ma luki.

Te trzy elementy działają razem: standard mówi, jak pracować, punkty kontrolne pilnują, żeby tak pracować, a pomiar pokazuje, czy to działa. Dzięki temu skuteczność testów można zweryfikować, zamiast zakładać ją na podstawie samej liczby przypadków.

Co to znaczy dla budżetu

Bramki jakości zmieniają przede wszystkim strukturę kosztu, a łączny koszt pozostaje zbliżony. Dane z sekcji 4.5 pokazują to wprost: pięć zespołów wykryło po osiem defektów przy koszcie od 14,1% do 100%, a przeliczone na jeden wykryty defekt koszty różniły się blisko trzydziestokrotnie. Wyższy koszt nie kupował lepszej wykrywalności.

Mniejszy, lepiej dobrany zestaw testów jest też tańszy w utrzymaniu: po zmianach interfejsu mniej testów wymaga poprawek, a rzeczywiste błędy łatwiej zauważyć wśród pozytywnych wyników.

Najrzadziej wykrywane defekty z Aneksu A, takie jak race condition na równoległych decyzjach analityka i zatwierdzenia wbrew negatywnemu scoringowi, razem z wyciekiem danych użytkowników, na produkcji kosztują najwięcej: błędne rozliczenia z klientami, decyzje kredytowe łamiące własne reguły oraz incydenty bezpieczeństwa wraz z konsekwencjami prawnymi. Bramki mają za zadanie je pokryć.

BOOKLET OPERATORA AI

Standard pracy operatora spisaliśmy jako piętnaście zasad gotowych do wdrożenia, pogrupowanych według pięciu bramek, w formie checklisty do wydruku.

Pobierz booklet operatora AI

8. Dyskusja i implikacje

Pozytywny pesymizm, czyli mindset testera stosowany bez przerwy

Dobry tester zamiast „czy to działa?" pyta „jak to zepsuć?". To zawodowa deformacja, którą w QA pielęgnujemy od zawsze: system jest winny, dopóki nie udowodni niewinności. Nasze badanie pokazało, że w pracy z AI takie podejście powinno stać się stałym elementem procesu.

LLM ma tendencję do generowania rozwiązań zgodnych z przyjętymi przez siebie założeniami i do potwierdzania wcześniejszych odpowiedzi. Wygeneruje testy, które przechodzą, raport, który wygląda kompletnie, i asercje, które niczego nie ryzykują. Jeśli operator zaufa tym zbyt optymistycznym wynikom, może uzyskać wysoki odsetek przechodzących testów bez odpowiadającego mu wzrostu wykrywalności błędów. Zespoły, które wykrywały realne błędy, robiły coś odwrotnego: traktowały każdy wynik z modelu jak hipotezę do obalenia. Zielony test był dla nich twierdzeniem, które dopiero trzeba sfalsyfikować.

Pesymizm w tej wersji jest pozytywny z dwóch powodów. Po pierwsze, prowadzi do wprowadzenia dodatkowych mechanizmów weryfikacji wyników AI, dzięki którym można jej ufać warunkowo i delegować coraz więcej. Po drugie, paradoksalnie to właśnie nieufność pozwala wycisnąć z modelu najwięcej: operator, który systematycznie kwestionuje wyniki, zmusza LLM do pracy w pętli poprawek, zamiast akceptować pierwszą wiarygodnie wyglądającą odpowiedź.

Pięć bramek to pięć form nieufności

Nie jest przypadkiem, że nasza lista pięciu najważniejszych praktyk to w gruncie rzeczy pięć bramek jakościowych nałożonych na pracę modelu. Niezależna wyrocznia specyfikacji, bo nie wierzymy, że kod, który model napisał, model uczciwie zweryfikuje. Osobny przebieg adwersaryjny po osiągnięciu pokrycia, bo nie wierzymy, że pokrycie oznacza wykrywanie. Przegląd higieny odpowiedzi i bezpieczeństwa, bo nie wierzymy, że model sam z siebie zajrzy tam, gdzie nikt nie kazał. Jawne sondowanie współbieżności i stanu, bo najpoważniejsze błędy w naszym badaniu, czyli wyścig na decide i limit dokumentów, leżały dokładnie tam, gdzie nie sięgnął żaden zespół.

Liczba testów jako metryka pomocnicza

Dyskusja o AI w testach często skręca w stronę liczby przypadków. „AI wygenerowała 200 testów" brzmi imponująco, ale jest metryką pomocniczą. Pytanie brzmi: ile z tych testów może obalić system? Nasze dane odpowiadają na nie jednoznacznie, bo zgłoszenie z 60 przypadkami wykryło osiem defektów, a zgłoszenie z 196 przypadkami tylko jeden. Dlatego liczba scenariuszy nie powinna być podstawową miarą skuteczności dostawcy.

Weryfikacja wyników AI

AI zmieniła narzędzia; epistemologia testowania pozostała ta sama, zmienił się tylko jej adresat. Kiedyś nieufność kierowaliśmy wyłącznie w stronę systemu pod testem. Dziś kierujemy ją równocześnie w dwie strony: w stronę systemu i w stronę asystenta, który pomaga nam go badać.

W testowaniu krytyczne podejście do wyników pozostaje kluczowe. Nasza nieufność jest dokładnie tym wkładem, którego model sam nie wniesie.



9. **WNIOSKI** KOŃCOWE

CO UDOWODNILIŚMY

Pierwsza edycja pokazała, że AI realnie i znacząco zwiększa produktywność. Druga to potwierdziła: modele przyspieszają budowę frameworków, testów kontraktowych i raportów, a poziom rzemiosła inżynierskiego pozostaje wysoki.

CO ODKRYLIŚMY

Ta produktywność zamienia się w skuteczne testowanie tylko wtedy, gdy człowiek prowadzi model przez właściwy proces. AI potrafi szybko zbudować framework, testy kontraktowe i raporty. Nie potrafi samodzielnie określić, które zachowanie jest poprawne z punktu widzenia reguł biznesowych. Nie odróżnia sama z siebie czerwonego testu, który jest artefaktem, od czerwonego testu, który właśnie znalazł błąd. I potrafi bardzo przekonująco osłabić asercję, jeśli celem rozmowy stanie się „żeby test przeszedł". Wykrywalność spadała dokładnie tam, gdzie rósł koszt produkcyjny defektu, a dwóch najdroższych błędów nie znalazł nikt.

Co powinny zrobić organizacje

- Wdrożyć standard pracy operatora. Pięć bramek jako szkielet, piętnaście zasad jako instrukcja wykonawcza.
- Mierzyć skuteczność, nie objętość. Rozliczać zespoły i dostawców z wykrytych, potwierdzonych ryzyk, a nie z liczby scenariuszy.
- Projektować proces wokół quality gates. Wymóg wyroczni, identyfikowalność od ryzyka przez scenariusz do asercji oraz próba zębów w definicji ukończenia.

AI pozwala pisać szybciej. Dobry operator sprawia, że szybciej piszemy właściwe testy. Wiemy, jak ten proces wygląda, umiemy go zmierzyć i umiemy go wdrożyć.

Ile z Waszych „zielonych" testów **wykryłoby błąd, który trafi do produkcji?**

Przeprowadzimy analizę na Waszej aplikacji z wykorzystaniem tej samej metodologii, którą zastosowaliśmy w badaniu AI Testing Lab. Wynik pokaże, które obszary ryzyka są skutecznie zabezpieczone przez testy, a gdzie istnieją luki wymagające uwagi.



Remigiusz Bednarczyk
Test Manager w Sii Polska.



Tomasz Kuran
Solution Architect w Sii Polska.



Krzysztof Bednarski
Solution Architect w Sii Polska.

Masz pytania?

Iuliia Toporova
Business Development Manager



Skontaktuj się z ekspertami

10. Aneksy

Aneks A. Rozkład wykrywalności zasianych defektów

Warstwa trudności	Przykładowy zasiany defekt	Wykryło zespołów
Kontrakt / obliczenia	Zły status HTTP (200 zamiast 201) przy tworzeniu wniosku	13 / 16
Kontrakt / obliczenia	Błędna prowizja i rozbieżna rata annuitetowa	13 / 16
Kontrakt / dokumentacja	Brak „Kredytu Konsolidacyjnego” opisanego w wymaganiach	13 / 16
Bezpieczeństwo	Wyciek pola password_hash w odpowiedzi profilu	11 / 16
Kontrakt	Fantomowy endpoint DELETE zadeklarowany w OpenAPI	11 / 16
Domenowy	DTI liczone od dochodu brutto zamiast netto	6 / 16
Cross-functional	Stara sesja aktywna po zmianie hasła	2 / 16
Cross-functional	Zmiana produktu wpływająca na istniejące wnioski	2 / 16
Cross-functional	Race condition na równoległej decyzji analityka	0 / 16
Cross-functional	Zatwierdzenie mimo negatywnego scoringu	0 / 16

Aneks B. Liczba testów a liczba wykrytych defektów

Dane źródłowe do wykresu 3. Każdy wiersz odpowiada jednemu zgłoszeniu.

Napisane przypadki testowe	Wykryte zasiane defekty	Przypadków na 1 defekt
60	8	7,5
53	6	8,8
64	6	10,7
131	8	16,4
107	6	17,8
154	8	19,2
157	8	19,6
213	9	23,7
150	6	25,0
226	7	32,3
263	8	32,9
134	4	33,5
109	3	36,3
201	4	50,2
912	10	91,2
196	1	196,0

Aneks C. Zgłoszone błędy, wykryte defekty i koszt zgłoszenia

Dane źródłowe do wykresu 4. Koszt podany w skali względnej, w której 100% oznacza zgłoszenie najdroższe w stawce. Ostatnia kolumna przelicza koszt na jeden wykryty defekt zasiany.

Zgłoszone błędy	Wykryte zasiane defekty	Koszt	Koszt na 1 defekt
21	6	3,5%	0,58
23	6	8,5%	1,42
16	9	15,0%	1,67
13	8	14,1%	1,76
23	8	19,1%	2,39
25	8	21,3%	2,66
17	7	22,3%	3,19
20	6	20,4%	3,40
22	6	31,2%	5,20
62	10	65,0%	6,50
13	4	27,1%	6,78
25	8	56,6%	7,08
13	3	23,3%	7,77
17	4	33,9%	8,47
33	8	100,0%	12,50
7	1	16,0%	16,00

Aneks D. Zakres weryfikacji i stack uczestników

- 16 zgłoszeń, każde z własnym frameworkiem testowym.
- 19 celowo zasianych defektów w 4 warstwach trudności.
- Około 350 plików z raportami błędów sprawdzonych ręcznie.
- Każdy raport konfrontowany z kodem LoanFlow, dokumentacją uczestnika, specyfikacją OpenAPI i wewnętrznym answer key.
- Rozbieżności między wstępnym rankingiem a weryfikacją rozstrzygane inspekcją kodu.

Zgłoszenia powstawały w różnych technologiach, co pozwala traktować opisane praktyki jako niezależne od stacku. Najczęściej wybierano Playwright z TypeScriptem, pojawiały się także rozwiązania w Javie, C# oraz pytest. W repozytoriach powtarzały się page objecty, fixture'y, helpery, wrappery API, raporty HTML, pliki kontekstowe dla AI oraz skrypty uruchamiające zestaw jednym poleceniem.