

Lista kontrolna długu technicznego

Siedem obszarów, 42 pytania i prosta skala 0–21. W jedno popołudnie, razem z ludźmi, którzy najlepiej znają system, ocenicie, ile długu technicznego się w nim nazbierało.

Jak korzystać z listy

1. Zarezerwujcie jedno popołudnie dla dwóch–trzech osób, które dobrze znają system: tech leada, doświadczonego programisty i kogoś, kto utrzymuje system na produkcji.
2. Na każde pytanie odpowiedzcie „tak” albo „nie”. Wszystkie pytania są tak sformułowane, że „tak” oznacza stan zdrowy, a „nie” wskazuje na dług. „Nie wiemy” liczcie jako „nie” — brak odpowiedzi to już ustalenie.
3. Każdy obszar oceńcie w skali 0–3 według liczby odpowiedzi „nie”: żadnej = 0, jedna lub dwie = 1, trzy lub cztery = 2, pięć lub sześć = 3. Zero oznacza brak problemu, trójka — poważne ryzyko.
4. Zsumujcie wyniki siedmiu obszarów (od 0 do 21) i odczytajcie odpowiedni przedział poniżej. Każdy obszar z wynikiem 3 wymaga uwagi niezależnie od sumy. Przedziały to praktyczna wskazówka na początek rozmowy, a nie branżowy standard. Jeśli asystenci AI współtworzą Wasz kod, odpowiedzcie dodatkowo na pytania o kod generowany przez AI.
5. Powtarzajcie to ćwiczenie co kwartał lub dwa. Trend powie Wam więcej niż pojedynczy wynik.

Arkusz wyników

Po odpowiedzi na pytania danego obszaru wpiszcie jego wynik w odpowiednim wierszu.

OBSZAR	WYNIK (0–3)	UWAGI
1. Jakość kodu		
2. Architektura		
3. Testy		
4. Zależności i platformy		
5. Bezpieczeństwo		
6. Dostarczanie i utrzymanie		
7. Zespół, wiedza i proces		
Razem	/ 21	

OBSZAR 1/7

Jakość kodu

Jak łatwo czytać, zmieniać i rozumieć kod, którego zespół dotyka co tydzień.

PYTANIA — ODPOWIEDZCIE „TAK” ALBO „NIE”

- ☐ Czy nowa osoba w zespole jest w stanie już w pierwszym tygodniu wprowadzić małą, bezpieczną zmianę w najczęściej modyfikowanym module?
- ☐ Czy wiecie, które pliki zmieniają się najczęściej i jak bardzo są złożone (tzw. hotspoty)?
- ☐ Czy nowe uwagi z analizy statycznej (linter, sprawdzanie typów, skaner jakości kodu) są naprawiane, zanim pull request zostanie zmergowany?
- ☐ Czy typowe rzeczy, takie jak obsługa błędów, logowanie i dostęp do danych, robi się w jeden uzgodniony sposób, a nie na kilka konkurencyjnych?
- ☐ Gdy zmienia się reguła biznesowa, czy poprawiacie ją w jednym miejscu, a nie w kilku kopiach tej samej logiki?
- ☐ Czy zakomentowany i martwy kod oraz nieużywane flagi funkcji są regularnie usuwane?

Liczba odpowiedzi „nie”: ____ → wynik obszaru (0–3): ____

SYGNAŁY OSTRZEGAWCZE

- „Tego modułu odważy się dotknąć tylko jedna osoba”.
- Wyceny drobnych zmian w niektórych częściach kodu są rutynowo zawyżane, „bo to ten moduł”.
- Ostrzeżenia lintera albo analizatora liczone w tysiącach, na które nikt już nie patrzy.

PRZYKŁADOWE MIARY

- Przykład: częstotliwość zmian × złożoność dla każdego pliku (hotspoty), wyliczona z historii repozytorium
- Przykład: liczba plików, których dotyka typowa mała zmiana
- Przykład: trend liczby uwag z analizy statycznej między wydaniem (kierunek jest ważniejszy niż wartość bezwzględna)

OBSZAR 2/7

Architektura

Czy struktura systemu wciąż odpowiada temu, czego potrzebuje od niego biznes.

PYTANIA — ODPOWIEDZCIE „TAK” ALBO „NIE”

- ☐ Czy istnieje aktualny opis lub diagram głównych komponentów i sposobu, w jaki się komunikują, a zespół zgadza się, że odpowiada on rzeczywistości?
- ☐ Czy da się zmienić jeden obszar biznesowy (np. rozliczenia) bez skoordynowanych zmian w kilku innych?
- ☐ Czy granice modułów lub usług są egzekwowane przez strukturę kodu albo narzędzia, a nie wyłącznie przez umowę w zespole?
- ☐ Czy wiecie, gdzie system najpierw osiągnie swoje granice, jeśli ruch albo wolumen danych wzrośnie dwukrotnie?
- ☐ Czy kluczowe decyzje architektoniczne są zapisane razem z uzasadnieniem (np. jako architecture decision records)?
- ☐ Czy każdy kluczowy zbiór danych ma jedno źródło prawdy, a nie kopie synchronizowane zadaniami cyklicznymi lub ręcznymi poprawkami?

Liczba odpowiedzi „nie”: ____ → wynik obszaru (0–3): ____

SYGNAŁY OSTRZEGAWCZE

- Prawie każda funkcja wymaga „drobnej zmiany” we wspólnym schemacie bazy albo we wspólnym module, od którego zależy kilka zespołów.
- Usługi, które da się wdrożyć tylko razem i w ściśle określonej kolejności.
- Obejścia tłumaczone zdaniem „architektura na to nie pozwala”.

PRZYKŁADOWE MIARY

- Przykład: odsetek zmian, które dotyczą więcej niż jednego modułu lub usługi
- Przykład: liczba komponentów, które trzeba wydać razem przy typowej zmianie

OBSZAR 3/7

Testy

Czy testy automatyczne dają zespołowi dość pewności, by zmieniać kod bez obaw.

PYTANIA — ODPOWIEDZCIE „TAK” ALBO „NIE”

- ☐ Czy testy automatyczne uruchamiają się przy każdym pull requestie i blokują merge, gdy nie przechodzą?
- ☐ Czy najważniejsze ścieżki biznesowe (płatności, rejestracja, kluczowe wyliczenia) są pokryte testami automatycznymi, którym zespół ufa?
- ☐ Czy zestaw testów jest na tyle szybki, że programiści uruchamiają go przed wystaniem zmian do repozytorium, zamiast czekać na wynik z CI?
- ☐ Czy niestabilne testy (flaky) są naprawiane albo izolowane w ciągu kilku dni, a nie puszczane ponownie aż do skutku?
- ☐ Czy błąd z produkcji da się odtworzyć w środowisku testowym na realistycznych, zanonimizowanych danych?
- ☐ Czy po naprawie błędu dopisujecie test, który by go wychwycił?

Liczba odpowiedzi „nie”: ____ → wynik obszaru (0–3): ____

SYGNAŁY OSTRZEGAWCZE

- „Puść pipeline jeszcze raz, ten test czasem nie przechodzi”.
- Ręczne testy regresji przed każdym wydaniem, które trwają kilka dni.
- Wysoki wskaźnik pokrycia, a mimo to regresje trafiają na produkcję w „pokrytych” miejscach.

PRZYKŁADOWE MIARY

- Przykład: czas wykonania pełnego zestawu testów i jego zmiana w czasie
- Przykład: odsetek przebiegów pipeline’u, które kończą się błędem z powodów niezwiązanych ze zmianą (niestabilność)
- Przykład: liczba błędów produkcyjnych w obszarach z testami automatycznymi w porównaniu z obszarami bez nich

OBSZAR 4/7

Zależności i platformy

Biblioteki, frameworki, środowiska uruchomieniowe i platformy, na których opiera się system — oraz to, jak bardzo są nieaktualne.

PYTANIA — ODPOWIEDZCIE „TAK” ALBO „NIE”

- ☐ Czy macie aktualny spis zależności bezpośrednich i przechodnich (np. software bill of materials)?
- ☐ Czy wszystkie elementy podstawowego stosu (środowisko uruchomieniowe języka, framework, baza danych) są w wersjach, które wciąż otrzymują poprawki bezpieczeństwa?
- ☐ Czy aktualizacje zależności to rutyna — np. automatyczne pull requesty przeglądane w każdym sprincie — a nie rzadkie, duże projekty?
- ☐ Czy wiadomo, kto decyduje o wprowadzeniu, aktualizacji albo wymianie kluczowej biblioteki?
- ☐ Czy znacie licencje open source swoich zależności i sprawdzacie je pod kątem sposobu, w jaki dystrybuujecie produkt?
- ☐ Czy dałoby się dziś zbudować i wdrożyć system od zera, bez archiwalnej paczki, nieutrzymywanego prywatnego mirrora czy laptopa jednej osoby?

Liczba odpowiedzi „nie”: ____ → wynik obszaru (0–3): ____

SYGNAŁY OSTRZEGAWCZE

- Aktualizacja frameworka do nowej wersji głównej, „zaplanowana na przyszły kwartał” od ponad roku.
- Przypięte wersje z komentarzami w rodzaju „nie aktualizować — psuje X”.
- Komponenty po końcu wsparcia (end of life), które wciąż działają na produkcji.

PRZYKŁADOWE MIARY

- Przykład: liczba zależności opóźnionych o więcej niż jedną wersję główną
- Przykład: wiek najstarszej znanej, niezaktualizowanej podatności w zależnościach
- Przykład: czas od wydania poprawki bezpieczeństwa biblioteki do jej wdrożenia na produkcji

OBSZAR 5/7

Bezpieczeństwo

Słabości wbudowane w kod i proces — ta część długu technicznego, która z dnia na dzień może zamienić się w incydent.

PYTANIA — ODPOWIEDZCIE „TAK” ALBO „NIE”

- ☐ Czy repozytorium — łącznie z całą historią zmian — jest wolne od sekretów, takich jak klucze API, hasła i tokeny?
- ☐ Czy skany zależności i statyczne skany bezpieczeństwa uruchamiają się automatycznie, a ich wyniki ktoś konkretny weryfikuje?
- ☐ Czy uwierzytelnianie i autoryzacja są obsługiwane w jednym spójnym, zrozumiałym miejscu, a nie implementowane od nowa przy każdym endpointzie?
- ☐ Czy wiecie, gdzie przechowywane są dane osobowe i wrażliwe, a dostęp do nich jest ograniczony i logowany?
- ☐ Czy macie przećwiczoną procedurę rotacji danych dostępowych i reagowania na incydent bezpieczeństwa?
- ☐ Czy w ciągu ostatnich 12 miesięcy ktoś spoza zespołu ocenił bezpieczeństwo systemu (przegląd kodu, test penetracyjny), a ustalenia zostały zamknięte?

Liczba odpowiedzi „nie”: ____ → wynik obszaru (0–3): ____

SYGNAŁY OSTRZEGAWCZE

- Wspólne konta administracyjne albo hasła, które zna „cały zespół”.
- Dane dostępowe wklejane do czatu, zgłoszeń albo plików konfiguracyjnych „tylko na chwilę”.
- Ustalenia bezpieczeństwa ze statusem „nie naprawiamy” bez udokumentowanego powodu.
- Nikt nie potrafi szybko powiedzieć, które usługi mają dostęp do danych klientów.

OBSZAR 6/7

Dostarczanie i utrzymanie

Jak kod trafia z komputera programisty na produkcję i ile tarcia dokłada po drodze dług.

PYTANIA — ODPOWIEDZCIE „TAK” ALBO „NIE”

- ☐ Czy wdrożenie na produkcję jest zautomatyzowane i powtarzalne, bez ręcznych kroków, które znają tylko wybrane osoby?
- ☐ Czy małą zmianę możecie wydać w dowolny dzień roboczy, a nie tylko w zaplanowanym oknie wydawniczym?
- ☐ Czy nieudane wydanie da się w ciągu kilku minut wycofać albo wyłączyć flagą funkcji?
- ☐ Czy środowiska (testowe, stagingowe, produkcyjne) są zdefiniowane jako kod i spójne ze sobą?
- ☐ Czy monitoring informuje Was o incydentach, zanim zrobią to klienci?
- ☐ Czy czas na spłatę długu technicznego jest jawnie zaplanowany w roadmapie, a nie upychany „jak będzie chwila”?

Liczba odpowiedzi „nie”: ____ → wynik obszaru (0–3): ____

SYGNAŁY OSTRZEGAWCZE

- Wydania, które wymagają weekendu, długiej checklisty i jednej konkretnej osoby.
- Incydenty typu „na stagingu działało”, wynikające z różnic między środowiskami.
- Czas realizacji drobnych zmian rośnie z kwartału na kwartał, choć zespół ma tę samą wielkość.

PRZYKŁADOWE MIARY

- Przykład: czas od commita do wdrożenia na produkcji (lead time for changes)
- Przykład: częstotliwość wdrożeń
- Przykład: odsetek wdrożeń kończących się awarią i czas przywrócenia działania

Zespół, wiedza i proces

Wiedza, odpowiedzialność i nawyki — dług, który siedzi w ludziach i procesach, a nie tylko w kodzie.

PYTANIA — ODPOWIEDZCIE „TAK” ALBO „NIE”

- ☐ Czy każdy krytyczny komponent rozumieją co najmniej dwie osoby, które wciąż są w zespole?
- ☐ Czy dokumentacja wystarcza, żeby nowy programista uruchomił projekt i wdrożył pierwszą zmianę bez ciągłego dopytywania?
- ☐ Czy dług techniczny jest widoczny w backlogu jako konkretne zadania z oszacowaniem kosztu albo wpływu?
- ☐ Czy code review dotyczy projektu rozwiązania i ryzyka, a nie tylko stylu?
- ☐ Czy Wasi inżynierowie potrafią wytłumaczyć dług product ownerom i zarządowi językiem biznesu: opóźnieniem, ryzykiem i kosztem?
- ☐ Gdyby w przyszłym miesiącu odszedł Wasz główny dostawca albo kluczowy programista, czy wiedzielibyście dokładnie, co przejmiecie?

Liczba odpowiedzi „nie”: ____ → wynik obszaru (0–3): ____

SYGNAŁY OSTRZEGAWCZE

- Wdrożenie nowej osoby trwa miesiącami i zależy od dostępności jednego seniora.
- O długu technicznym rozmawia się dopiero po incydencie albo przekroczonym terminie.
- Przy tych samych systemach dyżurują zawsze te same osoby.
- Zmiany, które trafiają na produkcję, zanim przeczyta je ktokolwiek poza autorem.

PRZYKŁADOWE MIARY

- Przykład: liczba komponentów, które naprawdę rozumie tylko jedna osoba
- Przykład: czas, po którym nowy programista wdraża pierwszą zmianę na produkcję
- Przykład: odsetek czasu sprintu zużywanego na pracę nieplanowaną

NIE WLICZA SIĘ DO WYNIKU 0–21

Kod generowany przez AI: cztery dodatkowe pytania

- ☐ Czy wiecie, które części kodu powstały w dużej mierze z pomocą asystentów AI?
- ☐ Czy kod wygenerowany przez AI przechodzi takie same przeglądy i testy jak kod pisany ręcznie, a testy sprawdzają faktyczne zachowanie, zamiast powielać implementację?
- ☐ Czy weryfikujecie zależności podpowiedziane przez asystenta, zanim je dodacie (czy istnieją, są utrzymywane, mają odpowiednią licencję), i skanujecie wygenerowany kod pod kątem zaszytych sekretów?
- ☐ Czy zespół potrafi wyjaśnić i utrzymywać kod wygenerowany przez AI bez generowania go od nowa?

Jak czytać łączny wynik

To praktyczna wskazówka na początek rozmowy, a nie standard branżowy.

0–5 Pod kontrolą

Dług jest, jak w każdym systemie, ale widzicie go i nim zarządzacie. Utrzymajcie rytm: regularne aktualizacje zależności, zadania długu w backlogu i przegląd raz na kwartał.

6–10 Odczuwalny

W konkretnych miejscach dług już spowalnia pracę. Wybierzcie jeden lub dwa obszary z najwyższym wynikiem i zaplanujcie tam konkretne prace, zanim problem się rozleje.

11–15 Poważny

Dług zaczyna dyktować roadmapę: zmiany trwają dłużej, niż powinny, a ryzyko narasta. To dobry moment na uporządkowany przegląd i plan naprawczy z jasnymi priorytetami.

16–21 Krytyczny

System niesie ryzyko, które może się ujawnić jako incydent, nieudane wdrożenie albo zablokowana roadmapa. Zanim oprzecie na nim kolejne zobowiązania, zlećcie niezależną ocenę.

Co dalej

Kiedy wystarczy przegląd wewnętrzny

Wewnętrzny przegląd zwykle wystarczy, gdy łączny wynik jest niski albo wysokie oceny skupiają się w jednym–dwóch obszarach, które zespół dobrze rozumie, gdy osoby, które budowały system, wciąż są na pokładzie, a głównym celem jest zaplanowanie prac naprawczych. Zamieńcie każde „nie” w zadanie w backlogu z przypisanym właścicielem i wróćcie do listy w kolejnym kwartale.

Kiedy warto zlecić audyt zewnętrzny

Zewnętrzny audyt ma sens, gdy stawka albo niepewność są wysokie: przed inwestycją lub przejęciem, gdy przejmujecie cudzy kod albo odchodzi dostawca, gdy duża część kodu powstała z pomocą asystentów AI, gdy wiele odpowiedzi brzmi „nie wiemy” albo gdy zespół i zarząd różnie oceniają powagę sytuacji. Niezależny audytor czyta kod bez historii i założeń jego autorów, a ustalenia zamienia w priorytety, na podstawie których decydenci mogą działać.

Wersja 2026-10-20 · ARDURA Consulting · ardura.pl/lista-kontrolna-dlugu-technicznego/

Potrzebujecie spojrzenia z zewnątrz?

Niezależny audyt kodu i technical due diligence: ardura.pl/audyt-kodu-due-diligence/