Migracja i ewolucja architektury

Strangler Fig · Branch by Abstraction · ACL · Feature Flags · 7 R-ów

mgr inż. Jakub Woźniak

Politechnika Poznańska · Instytut Informatyki · Semestr letni 2025/26

Plan wykładu

Agenda

- Identyfikacja bounded contexts w monolicie (nawiązanie do wykładu 2)
- Strangler Fig Pattern — inkrementalna migracja
- Branch by Abstraction — zamiana komponentu bez zmiany interfejsu
- Anti-Corruption Layer jako wzorzec migracyjny (rozwinięcie z wykładu 2)
- Parallel Run i Shadow Traffic
- Zarządzanie danymi — dual writes i Transactional Outbox
- Feature flags — progressive rollout i kill switch
- Migracja chmurowa — 7 R-ów
- ☁ Narzędzia migracyjne w chmurze

Diagnoza istniejącego systemu

Identyfikacja bounded contexts

Od wykładu 2 znamy bounded contexts — teraz: jak je znaleźć w monolicie?

Z wykładu 2: bounded context to granica, w której model domeny jest spójny. W monolicie te granice mogą być ukryte lub zatarte.

Techniki identyfikacji:

- **Event Storming** — mapowanie zdarzeń biznesowych, grupowanie w konteksty
- **Analiza języka uniwersalnego** — różne znaczenia tego samego terminu w różnych modułach
- **Analiza zależności kodu** — moduły, które rzadko się zmieniają razem = osobny kontekst

E. Evans — Domain-Driven Design, *rozdz. 14–15*

Sygnały, że system potrzebuje migracji

Kiedy monolit przestaje wystarczać?

- Jeden deploy = koordynacja 5+ zespołów
- Zmiana w jednym module powoduje regresję w niezwiązanym module
- Czas buildu > 30 minut
- Skalowanie całej aplikacji dla jednego wąskiego gardła

Lekarstwo gorsze od choroby?

Migracja do mikroserwisów bez zrozumienia domeny = distributed monolith (wykład 2).
Najpierw bounded contexts, potem extraction.

Strangler Fig Pattern

Idea — drzewo dusiciel

Strangler Fig (M. Fowler, 2004)

Wzorzec nazwany na cześć drzewa dusiciela — figowca, który rośnie na pniu gospodarza, stopniowo go oplatając, aż zastępuje całkowicie. Inkrementalnie zastępujemy funkcjonalność monolitu nowymi serwisami.

Kluczowa właściwość:

- Facade — warstwa trasująca ruch
- Nowa funkcjonalność trafia do nowych serwisów
- Stara funkcjonalność pozostaje w monolicie
- Monolit „usycha“ — aż można go wyłączyć

Big-bang cutover = ryzyko:

- Jeden moment przełączenia
- Brak możliwości rollbacku
- Pełny testing = iluzja

Mechanizm działania

Trzy komponenty Strangler Fig

1. **Routing proxy / API Gateway** — przyjmuje cały ruch, trasuje do monolitu lub nowych serwisów
2. **Nowe serwisy** — implementują wydzieloną funkcjonalność (bounded context)
3. **Legacy monolit** — traci endpointy jeden po drugim

Proces:

1. Postaw facade przed monolitem
2. Wydziel pierwszy bounded context ⟶ nowy serwis
3. Przekieruj ruch z facade do nowego serwisu
4. Powtarzaj, dopóki monolit obsługuje ruch
5. Wyłącz monolit

Strangler Fig — etapy na przykładzie

Etap	Ruch przez facade	Stan monolitu
0 — start	100% ⟶ monolit	Pełny monolit
1 — pierwszy serwis	90% ⟶ monolit	Usunięty endpoint zamówień
5 — piąty serwis	40% ⟶ monolit	Zredukowany do CRM i raportów
N — koniec	0% ⟶ monolit	Wyłączony

Feature Parity Trap

Nie kopiuj całej funkcjonalności starego serwisu — wydzielaj bounded context i dostarczaj nową wartość.
Rebuilding unused features = strata czasu.

Branch by Abstraction

Idea — warstwa abstrakcji jako szew

Branch by Abstraction (P. Hammant / S. Curl, 2007)

Technika inkrementalnej zamiany komponentu *wewnątrz* codebase'u. Nazwa: P. Hammant, koncepcja: S. Curl. Zamiast jednorazowego cięcia, tworzymy warstwę abstrakcji i stopniowo migrujemy klientów na nową implementację.

Różnica względem Strangler Fig:

- **Strangler Fig** — działa na granicy systemu (routing zewnętrzny)
- **Branch by Abstraction** — działa wewnątrz kodu (zastąpienie komponentu z zależnościami upstream)

5 kroków Branch by Abstraction

1. **Stwórz abstrakcję** — interfejs opisujący kontrakt między klientem a supplierem
2. **Zmigruj klientów** — przenieś kod wywołujący starego suppliera na abstrakcję
3. **Zbuduj nowego suppliera** — nowa implementacja za tą samą abstrakcją
4. **Przełącz** — użyj flagi lub konfiguracji, by wybrać nową implementację
5. **Usuń starego suppliera i abstrakcję** — po pełnym przejściu, wyczyść kod

Feature Toggles + Branch by Abstraction

Abstrakcja daje *szew*, flaga daje *przełącznik*. Używane razem: migracja bez downtime.

Strangler Fig vs Branch by Abstraction

Aspekt	Strangler Fig	Branch by Abstraction
Poziom działania	Granica systemu (API Gateway)	Wewnątrz kodu (interfejs)
Zakres	Cały serwis / endpoint	Pojedynczy komponent
Wymaga	Routing zewnętrzny	Abstrakcja w kodzie
Przełączenie	Zmiana trasy na poziomie Gateway	Feature toggle / konfiguracja
Kiedy stosować	Monolit $\longrightarrow$ mikroserwisy	Zamiana biblioteki / modułu

Anti-Corruption Layer

ACL — od relacji w Context Map do wzorca migracyjnego

Z wykładu 2 znamy ACL jako relację w Context Map

„Odbiorca tłumaczy model dostawcy na swój" — chroni bounded context przed przeciekiem obcego modelu.

ACL należy do kontekstu odbiorcy — każdy serwis może mieć własny ACL dostosowany do swojego modelu:

- Stary system: `FIRST_NM`, `DT_CREATED`, `IS_ACTV_FLAG`
- ACL tłumaczy: `FIRST_NM` ⟶ `firstName`, `DT_CREATED` ⟶ `createdAt`
- Nowy serwis widzi tylko czysty model domeny

Składniki ACL: Facade, Adapter, Translator

Trzy warstwy ACL

1. **Facade** — uproszczony interfejs do starego systemu (ukrywa złożoność)
2. **Adapter** — konwersja protokołów i formatów (REST ↔ SOAP, JSON ↔ XML)
3. **Translator** — konwersja modelu danych (legacy model → czysty model domeny)

Cykl życia ACL:

- **Migracja**: ACL jest aktywny — tłumaczy między starym a nowym
- **Pełne przejście**: stary system wyłączony → ACL można usunąć

Anti-pattern: ACL na zawsze

ACL to warstwa przejściowa. Jeśli zostaje na stałe — masz nowy distributed monolith z dodatkowym przeskokiem.

Parallel Run i Shadow Traffic

Dark Launch i Shadow Traffic

Dark Launch

Nowy serwis przetwarza rzeczywisty ruch, ale wyniki nie są zwracane użytkownikowi. Walidacja na produkcji bez ryzyka.

Shadow Traffic

Ruch produkcyjny kopiowany do nowego serwisu. Wyniki porównywane z legacy, ale użytkownik widzi tylko odpowiedź legacy.

Parallel Run

Obie implementacje obsługują ten sam ruch. Wyniki porównywane automatycznie (diff testing).

Narzędzia: Istio mirror (traffic mirroring), API Gateway canary weights, feature toggle shadow mode.

Porównanie strategii weryfikacji

Strategia	Użytkownik widzi	Ryzyko	Kiedy
Canary release	Nowy serwis	Średnie	Pewność co do funkcjonalności
Shadow traffic	Legacy	Niskie	Weryfikacja poprawności nowej implementacji
Parallel run	Legacy + diff	Niskie	Krytyczne systemy (płatności, finanse)
Big-bang cutover	Nowy serwis	Wysokie	Nigdy, jeśli masz wybór

Zarządzanie danymi w migracji

Problem dual writes

Dual write — prędzej czy później zawiedzie

Serwis zapisuje do bazy danych i publikuje zdarzenie do brokera. Nie ma transakcji rozproszonej między DB a Kafką.
Jeśli zapis do DB się uda, a publish nie — dane niespójne.
Jeśli publish się uda, a zapis do DB nie — dane niespójne.

Podejście	Problem
Zapis do DB + publish	Brak atomowości — dual write problem
CDC (Debezium)	Atomowe — odczyt z transaction log (WAL, binlog) ⟶ patrz wykład 8
Outbox Pattern	Atomowe — zapis zdarzenia w tej samej transakcji DB

Nawiązanie do wykładu 5

CDC omówiliśmy jako mechanizm propagacji zmian między serwisami. Szczegóły pipeline’u (Debezium + Kafka Connect) — na wykładzie 8.

Transactional Outbox Pattern

Jak to działa

1. Serwis zapisuje dane biznesowe i zdarzenie do *tej samej bazy* (jedna transakcja ACID)
2. Zdarzenie ląduje w tabeli `outbox` wewnątrz tej samej bazy
3. Osobny proces (relay) czyta z `outbox` i publikuje do Kafki
4. Po potwierdzeniu publish — usuwa wpis z `outbox`

Zalety:

- Atomowość — ta sama transakcja DB
- Brak dual write
- Latencja: 500ms–2s (polling)

Wady:

- Opóźnienie między zapisem a publishem
- Relay to dodatkowy komponent
- Konsumenci muszą być idempotentni (at-least-once delivery)

Produkcyjna praktyka: Outbox + CDC

Zamiast polling z latencją 500ms–2s, Debezium czyta tabelę `outbox` z transaction log i publikuje natychmiast. Łączy atomowość Outbox z niską latencją CDC.

Feature Flags

Progressive Rollout i Kill Switch

Progressive Rollout

Kod wdrożony „wyłączony“ $\longrightarrow$ włączany stopniowo: 0% $\longrightarrow$ wewnętrzni $\longrightarrow$ 1% $\longrightarrow$ 5% $\longrightarrow$ 25% $\longrightarrow$ 50% $\longrightarrow$ 100%

Kill Switch — ręczny wyłącznik awaryjny

Boolean w konfiguracji wyłączający funkcjonalność w sekundy.
Nawiązanie do wykładu 6 — kill switch uzupełnia circuit breaker: CB działa automatycznie na poziomie infrastruktury, kill switch to ręczna decyzja na poziomie biznesowym.

Typowe narzędzia: LaunchDarkly, Split.io, Unleash, AWS AppConfig.

Typy flag i pułapki

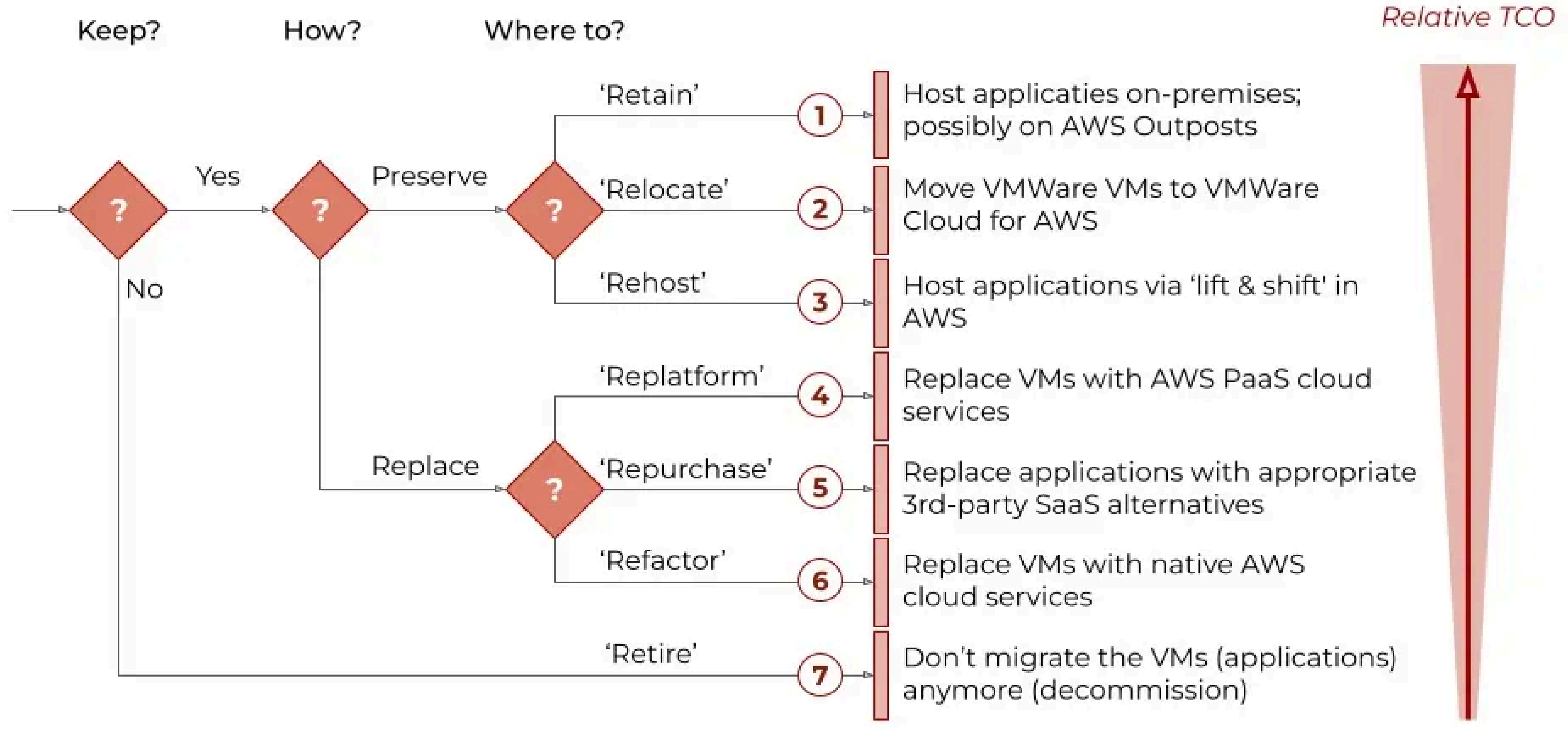
Typ flagi	Cykl życia	Przykład
Release flag	Dni–tygodnie	Strangler Fig: nowy serwis włączony na 5% ruchu
Ops flag	Stała	Kill switch dla dostawcy płatności
Experiment flag	Tygodnie	A/B test layoutu koszyka
Permission flag	Stała	Premium features dla planu Enterprise

Flag Debt

Stare flagi = martwy kod w produkcji. Każda release flaga musi mieć **datę usunięcia**.
Reguła: jeśli flaga na 100% przez 2 tygodnie → usuń ją i stary kod path.

Migracja chmurowa — 7 R-ów

7 R-ów migracji do chmury (AWS)



AWS Community — 7 Rs of Cloud Migration

7 R-ów — szczegółowe porównanie

Strategia	Opis	Nakład	Ryzyko
Rehost	Przenieś „jak jest“ na VM/kontener	Niski	Niskie
Replatform	Minorne optymalizacje chmurowe (managed DB)	Średni	Średnie
Re-architect	Przeprojektuj na cloud-native (mikroserwisy, serverless)	Wysoki	Wysokie
Repurchase	Zmień na SaaS (np. CRM ⟶ Salesforce)	Średni	Średnie
Retire	Wyłącz nieużywane aplikacje	Niski	Niskie
Retain	Zostaw on-prem (np. compliance, legacy hardware)	Żaden	Niskie
Relocate	Migracja na poziomie hypervisora (VMware ⟶ AWS)	Niski	Niskie

Zasada AWS

Rehost first, optimize later. Największe programy migracyjne zaczynają od Rehost — potem Replatform i Re-architect na kolejnych falach.

Decyzja: która strategia i kiedy

Sygnał	Strategia
Aplikacja działa, brak zmian biznesowych	**Retire** — może nie jest potrzebna
Stabilny monolit, potrzeba szybko wejść do chmury	**Rehost** — lift & shift
Chcesz managed DB i autoscaling, bez rewrite	**Replatform** — lift, tinker & shift
Domena dojrzała, mikroserwisy mają sens	**Re-architect** — re-architektura
Komercyjny SaaS robi to lepiej	**Repurchase** — drop & shop
Compliance wymusza on-prem	**Retain** — revisited later

Case study: Monzo Bank

Od kilku serwisów do 2800+

Monzo Bank — ewolucja architektury

2015: Kilka serwisów w Go — można odpalić wszystko na laptopie.

2018: Setki serwisów — lokalne dev zaczyna boleć. Stworzono „Orchestra“.

2022: 1500 serwisów — Orchestra nie wystarcza. Stworzono „Devproxy” + wirtualizacja serwisów.

2024: 2800+ serwisów — centralnie sterowane migracje jednym zespołem.

Kluczowe techniki:

- **Monorepo** — wszystkie serwisy w jednym repo, masowe refaktoryzacje jednym commitem
- **Standaryzacja** — ta sama struktura folderów, te same wersje bibliotek
- **Centralnie sterowane migracje** — jeden zespół pcha zmianę przez 2800 serwisów

Monzo — lekcje

Co poszło dobrze

- Standaryzacja = tajna broń
- Monorepo + generator = spójność
- Kill switch na każdy krytyczny przepływ
- Centralny zespół migracji zamiast 50 zespołów

Na co uważać

- Skala lokalnego dev → wymaga specjalistycznych narzędzi
- 2800 serwisów = 2800 potencjalnych punktów awarii
- Migracja biblioteki = wdrożenie do wszystkich serwisów
- Staging drift — współdzielone środowisko rozjeżdża się z produkcją

S. Patel — „Banking on Thousands of Microservices“ (QCon London 2023)

Narzędzia migracyjne w chmurze

☁ Migracja do chmury — usługi i narzędzia

Dostawca	Usługa	Funkcja
AWS	AWS DMS	Migracja bazy danych (homogeneous i heterogeneous)
AWS	AWS MGN	Rehost — block-level replikacja serwerów
AWS	AWS Application Discovery	Inwentaryzacja i zależności on-prem
GCP	Migration Center	Ocena portfolio, zalecenia migracji
GCP	Database Migration Service	Migracja DB do Cloud SQL / Spanner
Azure	Azure Migrate	Ocena, Rehost, Replatform w jednym narzędziu
Azure	Azure Database Migration	Online migracja do Cosmos DB / PostgreSQL

Macie 5-letni monolit w Rails z 2 mln LoC i 20 zespołami.

Zespół płatności chce wydzielić się jako pierwszy mikroserwis.

Który wzorzec wybierzecie — Strangler Fig czy Branch by Abstraction?

Jakie są pierwsze 3 kroki migracji danych?

Podsumowanie

Kluczowe wnioski

1. **Poznaj domenę przed dekompozycją** — bounded contexts (W02) wyznaczają granice cięcia
2. **Strangler Fig** — inkrementalna migracja na granicy systemu, bez big-bang cutover
3. **Branch by Abstraction** — zamiana komponentu wewnątrz kodu, bez zmiany interfejsu
4. **ACL chroni nowy model** przed przeciekiem starego (rozwinięcie z W02)
5. **Dual writes nie działają** — Outbox Pattern lub CDC (⟶ wykład 8)
6. **Feature flags** — progressive rollout, kill switch, ale flag debt to realny problem
7. **7 R-ów** — rehost na start, re-architect na później; nie wszystko musi do chmury
8. ☁ **Managed migration tools** — DMS, Migrate, Migration Center

Źródła i lektury

- M. Fowler — „Strangler Fig Application“ (martinfowler.com, 2004/2024)
- M. Fowler — „Branch By Abstraction“ (martinfowler.com, 2014)
- I. Cartwright, R. Horn, J. Lewis — „Patterns of Legacy Displacement“ (martinfowler.com, 2024)
- S. Newman — *Monolith to Microservices* (O’Reilly, 2020)
- M. Nygard — *Release It!*, 2nd ed. (2018)
- AWS — 7 Rs of Migration (docs.aws.amazon.com)
- Monzo Engineering Blog — „How we run migrations across 2,800 microservices“ (2024)
- P. Hodgson — „Feature Toggles“ (martinfowler.com, 2017)

Następny wykład

Wykład 8: Przetwarzanie danych w skali

Batch · Streaming · Kafka jako szkielet · CDC w praktyce