

# Architektura mikroserwisów z wykorzystaniem Spring Cloud

Łukasz Andrzejewski

# Podstawowe definicje związane logiką

- **Reguły biznesowe** definiują pojęcia i polityki niezbędne dla działania biznesu
- **Proces biznesowy** to seria powtarzalnych kroków, wykonywanych przez organizację, w celu uzyskania pożądanego efektu (celu biznesowego)
- **Logika biznesowa** to część aplikacji, odpowiedzialna za realizację przyjętych reguł biznesowych

# Poprawna implementacja logiki biznesowej

- Opiera się o stosowanie praktyk, prowadzących do czystego kodu oraz czystej architektury
- Większość z nich daje się uogólnić do działań zapewniających **niskie sprzężenie (low coupling)** i **wysoką spójność (high cohesion)**

# Interfejs / API

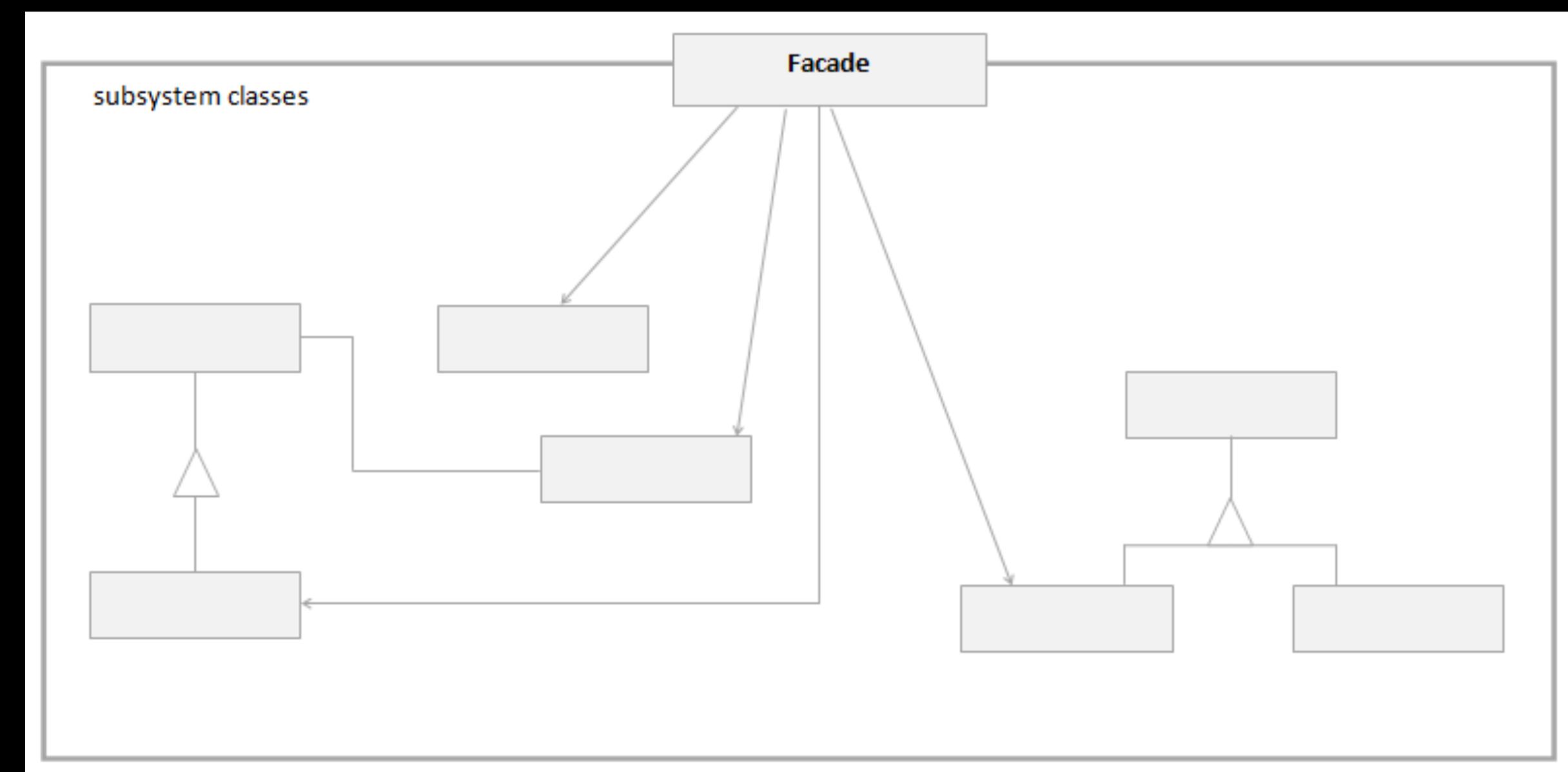
- Abstrakcja definiująca kontrakt między współpracującymi elementami (obiektami, komponentami, modułami, usługami, mikroserwisami)
- Określa możliwe sposoby interakcji
- Ukrywa szczegóły implementacyjne

# Wzorzec projektowy

- Szablon najlepszego rozwiązania, odkrywany na drodze doświadczenia
- Koncepcja pochodzi od architekta (Christopher Alexaner), który zaproponował wykorzystanie języka wzorców w klasycznej architekturze / budownictwie
- Wzorce projektowe związane z programowaniem obiektowym zostały opisane w „[Design Patterns Elements of Reusable Object-Oriented Software](#)” przez Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides tzw. bandę czworga (GOF)

# Fasada jako szczególny przypadek interfejsu

- Oferuje wysokopoziomowy interfejs dostępowy do zbioru obiektów, komponentów, podsystemów
- Redukuje sprzężenie i ukrywa szczegóły implementacyjne



# Inwersja kontroli (IoC)

- Odwrócenie sterowania wykonania programu / przeniesienie na zewnątrz odpowiedzialności za kontrolę wykonania

# Wstrzykiwanie zależności

- Tworzenie, konfigurowanie i podawanie zależności „z zewnątrz”
- Zmniejsza sprzężenie między elementami aplikacji, co m.in. daje swobodę wymiany implementacji i ułatwia testowanie
- Przykłady realizacji:
  - Ręczne podawanie zależności np. przez konstruktor, metody
  - Wzorce projektowe np. [Factory Method](#)
  - Kontener zarządzający cyklem życia komponentów ([Spring](#), [CDI](#))

# Programowanie aspektowe

- Uzupełnia paradymat programowania obiektowego
- Umożliwia oddzielenie logiki biznesowej od dodatkowych zadań pobocznych, takich jak: transakcje, logowanie, bezpieczeństwo
- Często realizowane z wykorzystaniem wzorca [Proxy](#)

# Programowanie przez zdarzenia

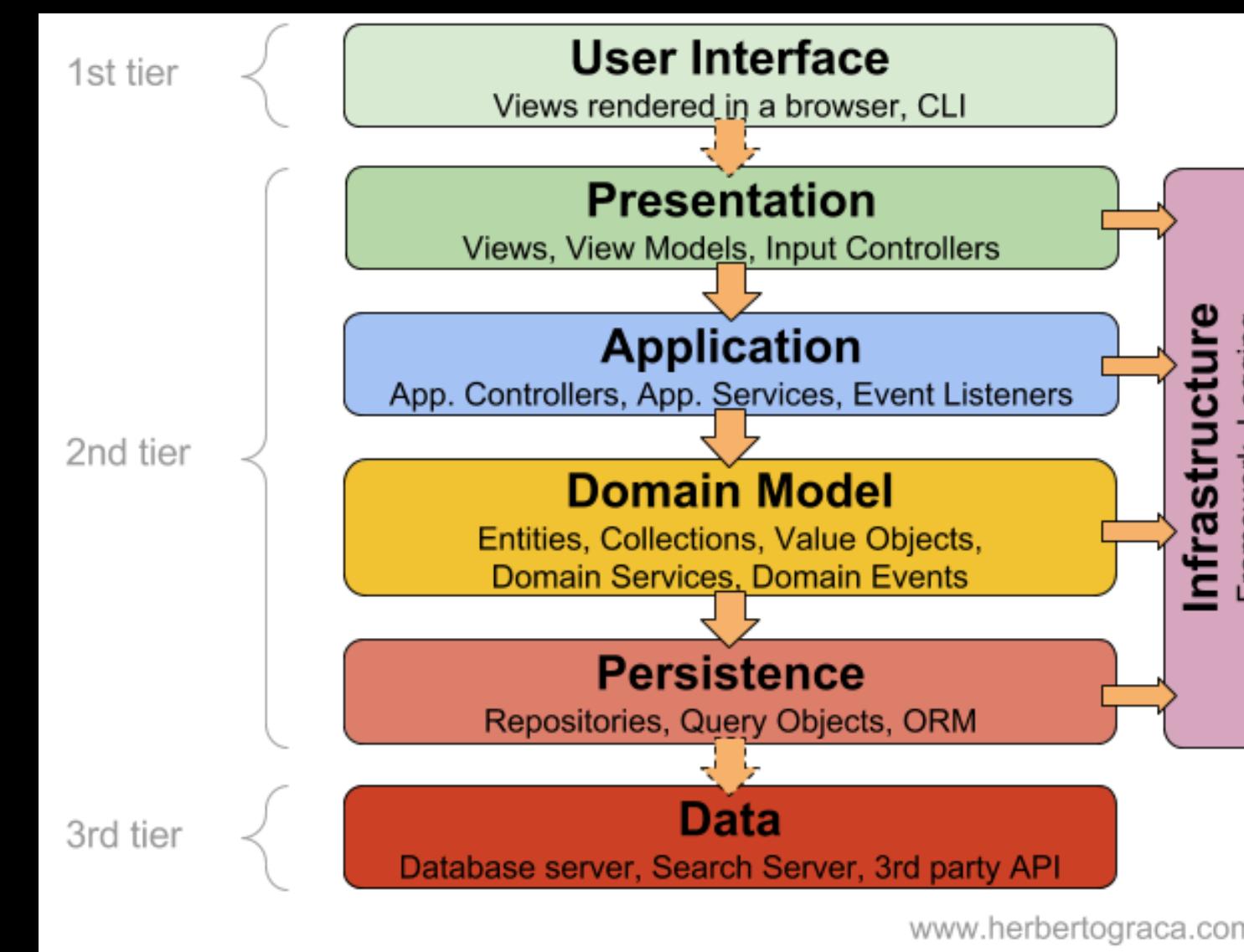
- Umożliwia rozluźnienie powiązań - komponenty mogą się komunikować mimo, że niewiele o sobie wiedzą
- Opiera się o wzorzec **Observer**

# Architektura

- Definiuje najważniejsze komponenty rozwiązania, zakres ich odpowiedzialności, a także wzajemne relacje i sposób interakcji
- Stanowi szablon rozwiązania
- Może być zdefiniowana na różnym poziomie np. aplikacja, system, infrastruktura

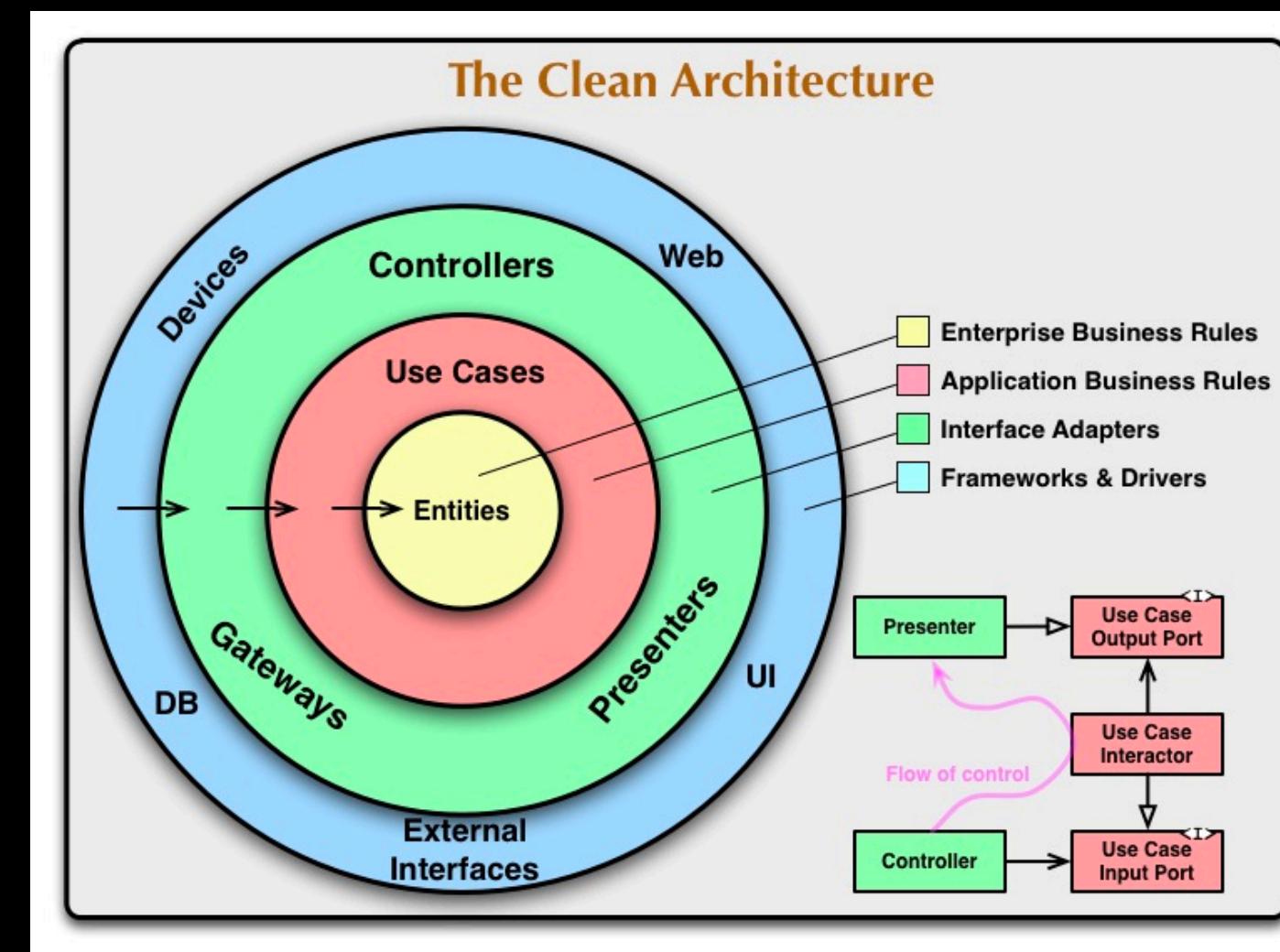
# Architektura warstwowa (n-tier pattern)

- Komponenty aplikacji organizowane są w warstwy pełniące określoną rolę np. prezentacja, logika, utrwalanie danych (podział techniczny)
- Komunikacja między warstwami odbywa się zwykle w określonym kierunku, w oparciu o zdefiniowany kontrakt



# Czysta architektura

- Centralnym elementem aplikacji jest logika biznesowa, zaimplementowana w sposób niezależny od bibliotek, frameworków czy użytej infrastruktury
- Wokół logiki biznesowej tworzone są kolejne, bardziej wysokopoziomowe warstwy m.in. warstwa adapterów umożliwiająca komunikację ze światem zewnętrznym



<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

# Domain Driven Design

- Podejście do tworzenia oprogramowania, które kładzie nacisk na to, aby obiekty i ich zachowania wiernie odzwierciedlały rzeczywistość / domenę problemu

# Bounded Context

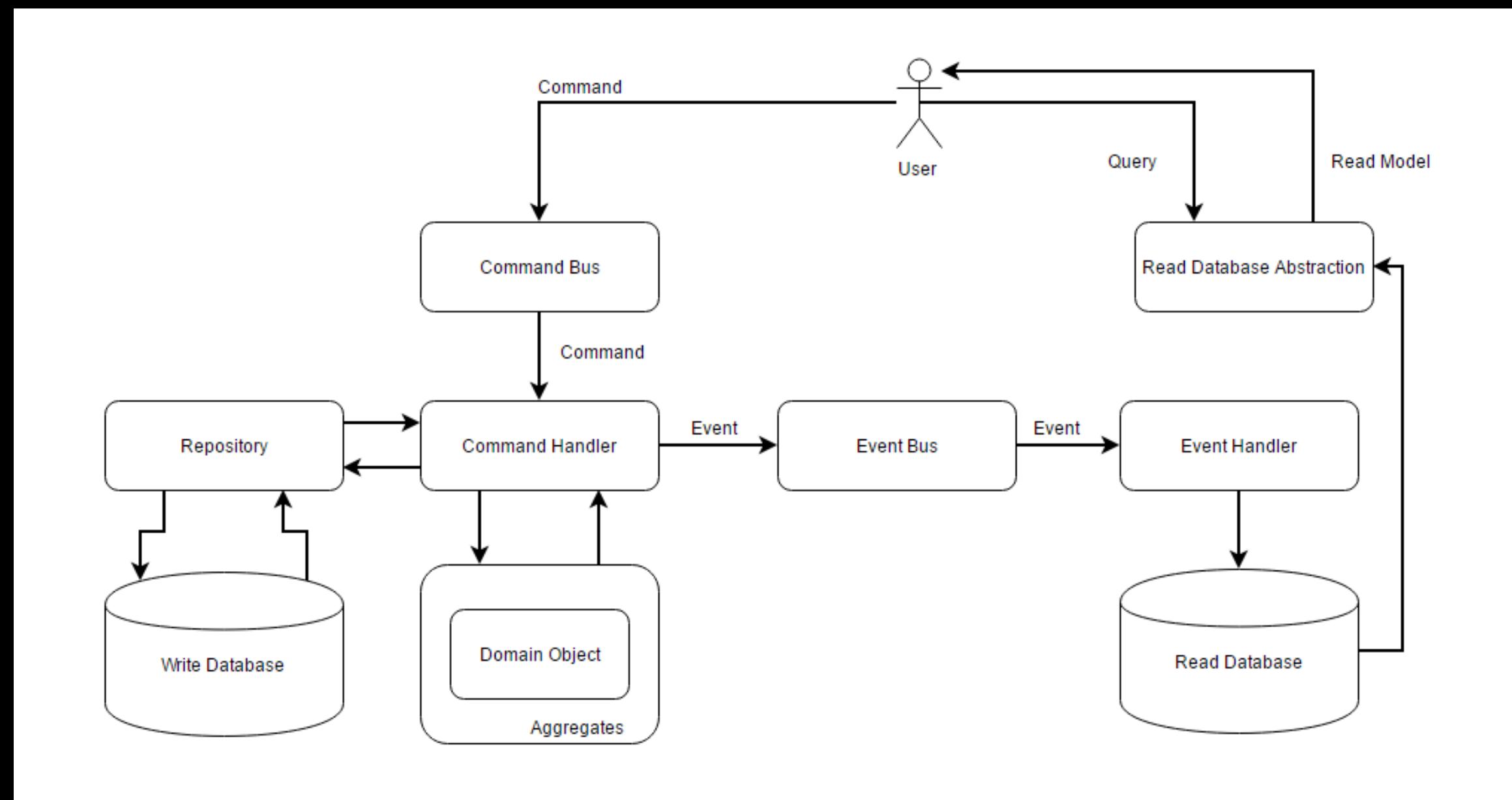
- Koncepcja zakładająca podział aplikacji na konteksty, definiujące własny model dziedzinowy, odwzorowujące konkretne potrzeby, warunki i procesy np. płatności, wyszukiwanie produktów, realizacja zamówień
- Komunikacja między kontekstami odbywa się za pośrednictwem dobrze określonego interfejsu, a zmiany modelu wewnętrznego lub logiki nie powinny mieć na nią wpływu

# Command Query Separation

- Zasada przedstawiona przez w 1986 roku przez Bertranda Meyera, mówiąca o tym, że każda metoda powinna być zaklasyfikowana do jednej z grup:
  - **Command** - metody, które zmieniają stan aplikacji i nic nie zwracają
  - **Query** - metody, które coś zwracają, ale nie zmieniają stanu aplikacji

# CQRS - Command Query Responsibility Segregation

- Przeniesienie pomysłu CQS na poziom klas / komponentów / całego systemu, zaproponowane przez Greg Young oraz Udi Dahan



# CQRS - elementy składowe

- **Command** - obiekt reprezentujący intencję użytkownika systemu np. `UpdateItemQuantityCommand`
- **Command Bus** - kolejka dla przychodzących komend, „router” przekazujący zadania do odpowiedniego handlera
- **Command Handler** - waliduje otrzymane komendy, tworzy lub zmienia stan obiektu domenowego, utrwała zmiany z pomocą repozytorium (write database) i przekazuje ewentualne zdarzenia do Event Bus
- **Domain objects** i **Aggregates** - model domenowy i logika biznesowa, zmiany na obiektach tego typu generują zdarzenia

# CQRS - elementy składowe

- **Event** - obiekt reprezentujący zmiany, które zaszły w domenie  
np. ItemQuantityUpdatedEvent
- **Event Bus** - kolejka dla generowanych zdarzeń, „router” przekazujący zdarzenia do odpowiedniego event handlera
- **Event Handler** - utrwalają zmiany stanu w bazie do odczytu (read database)
- **Read Database Abstraction** - abstrakcja / kontrakt umożliwiająca odczyt danych / stanu domeny

# Event Sourcing

- W CQRS zdarzenia są używane jako mechanizm synchronizacji dwóch baz danych (read i write)
- Event Sourcing pozwala na odtworzenie aktualnego stanu aplikacji (obiektów domenowych) na podstawie zdarzeń składowanych w magazynie danych zwanym Event Store

# Monolit

- Aplikacja rozwijana, testowana i wdrażana jako całość (single artifact)
- Zalety (do pewnego rozmiaru projektu)
  - Łatwa implementacja nowych funkcjonalności biznesowych i pobocznych
  - Mała złożoność infrastrukturalna
- Wyzwania (od pewnego rozmiaru projektu)
  - Trudność utrzymania i rozwoju ze względu na rosnącą złożoność, zakres funkcjonalności i rozmiar aplikacji
  - Ograniczona skalowalność
  - Przywiązanie do określonych rozwiązań i technologii

# Modularny monolit

- Aplikacja rozwijana, testowana i wdrażana jako zbiór modułów, które:
  - są od siebie niezależne
  - mają dobrze zdefiniowaną odpowiedzialność
  - posiadają publiczny interfejs / kontrakt
  - mogą być reużywane w innych aplikacjach

# Architektura zorientowana na usługi (SOA)

- Rozwiązanie w postaci rozproszonych usług / komponentów, które są:
  - mocno reużywalne - mogą być wykorzystane do realizacji różnych celów biznesowych, często współdzielą dane
  - wytwarzane i wdrażane zgodnie z przyjętymi standardami organizacji
  - komponowane / integrowane z wykorzystaniem ESB

# Architektura oparta o mikrosierwisy

- Rozwiązanie w postaci rozproszonych mikro usług (mini aplikacji), które:
  - są od siebie niezależne
  - mają dobrze zdefiniowaną odpowiedzialność (najczęściej w oparciu o domenę biznesową)
  - posiadają publiczny interfejs / kontrakt
  - nie współdzielą danych i komunikują się wyłącznie przez publiczne API

# Architektura na poziomie micro i macro

- Mikroarchitektura - obejmuje decyzje dotyczące poszczególnych usług, np. wybór technologii, języka programowania, bazy danych
- Makroarchitektura - obejmuje decyzje dotyczące wszystkich usług, np. wybór protokołu komunikacyjnego lub mechanizmu bezpieczeństwa
- Niektóre decyzje wpływają głównie na poziom operacyjny, np. dostarczenie konfiguracji, monitorowanie, analiza logów lub wdrożenie

# Mikroserwis

- Samowystarczalna usługa, zamodelowana wokół dobrze określonej logiki biznesowej, niezależnie rozwijana i wdrażana

# Mikroservisy - zalety

- Modularność
- Reużywalność
- Skalowalność
- Optymalne zużycie zasobów
- Niezawodność i wysoka dostępność
- Łatwość tworzenia / utrzymania na poziomie pojedynczych usług

# Mikroservisy - wyzwania

- Złożoność na poziomie makro (infrastruktura, utrzymanie, implementacja pobocznych funkcjonalności, synchronizacja stanu / procesów)
- Koszty (zasoby ludzkie, sprzęt, utrzymanie)
- Mniejsza wydajność i większa podatność na awarie (komunikacja przez sieć)

# Mikroservisy - przypadki użycia

- Duże, złożone systemy, obsługujące wielu użytkowników, pracujące pod dużym obciążeniem, wymagające elastyczności w zakresie skalowania czy zastosowanej technologii np. usługa przesyłania strumieniowego, rozwiązania e-commerce
- Powinniśmy pomyśleć o migracji, jeśli możemy osiągnąć cele (zyski), na które nie pozwala obecna architektura / podejście

# Mikroservisy - pytania zanim zastosujemy

- Co chcemy osiągnąć?
- Jakie alternatywy są dostępne?
- Skąd będziemy wiedzieć, czy migracja przynosi pożądane rezultaty?

# Mikroservisy - kiedy migracja jest ryzykowna?

- Niejasna lub nieznana domena, w której występuje problem
- Projekty start-upowe
- Hurra optymizm / podążanie za najnowszymi trendami

# Mikroservisy - jak implementować?

- Powinieneś zacząć od dobrej znajomości i zrozumienia domeny (kontekstu)
- Stopniowo i iteracyjnie wyodrębniaj kolejne moduły, a następnie usługi (aby uniknąć wysokich kosztów należy podejmować małe kroki i nie bać się popełnienia błędów)
- Proces ekstrakcji powinien obejmować wdrożenie / wykorzystanie danej mikrousługi na produkcji (dopiero wtedy pojawiają się rzeczywiste problemy)
- Wymagane jest utworzenie / reorganizacja zespołów (kompetencje interdyscyplinarne)
- Dobrą praktyką jest wykonywanie retrospekcji i wyciąganie wniosków

# Podział monolitu na mikroservisy

- Domenowy / procesowy - ze względu na funkcjonalność / zaimplementowany proces, np. składanie zamówienia
- Techniczny - ze względów technicznych, np. wydajność lub skalowalność, np. wyszukiwanie produktów według kategorii i wyszukiwanie pełnotekstowe
- Ostateczny podział zależy od wymagań systemowych i zaimplementowanej funkcjonalności
- Usługi powinny być projektowane z perspektywy interfejsu / API, a nie implementacji

# Kontekst ma znaczenie

- Każdy problem, proces, firma jest inna, dlatego za każdym razem należy do dostosować podejście - jeśli coś działa w jednym przypadku, nie oznacza to, że zadziała w innym

# Mikroservisy - aspekty techniczne

- Niezależność / autonomia
- Automatyczna konfiguracja
- Konteneryzacja i niezależność od środowiska wykonawczego
- Odkrywanie usług i komunikacja między nimi
- Dostęp do danych i transakcje
- Monitorowanie stanu, śledzenie ruchu, analiza dzienników, diagnostyka
- Bezpieczeństwo
- CI/CD
- Dostęp z poziomu klienta i integracja z technologiami frontendowymi

# Dostęp do danych

- Każda usługa powinna zarządzać swoimi danymi i jako jedyna mieć do nich bezpośredni dostęp
- Jeśli jedna usługa potrzebuje dostępu do danych przechowywanych przez inną usługę, powinna poprosić o to za pośrednictwem uzgodnionego interfejsu (usługi decydują o tym, co jest publiczne, a także ukrywają szczegóły implementacji, co zmniejsza potencjalne sprzężenie)
- Transakcje są lokalne

# ISA (Independent Systems Architecture)

- System należy podzielić na moduły, które komunikują się ze sobą tylko poprzez kontrakt / interfejs (niezależność od szczegółów implementacyjnych)
- System powinien posiadać dwa poziomy architektury
  - a) Macro - decyzje dotyczą wszystkich modułów
  - b) Micro - decyzje dotyczą indywidualnych modułów
- Moduły powinny być niezależnymi procesami, kontenerami lub maszynami wirtualnymi, aby zapewnić odpowiedni poziom izolacji
- Wybór metod komunikacji i integracji powinien być zawężony i ustandaryzowany

# ISA (Independent Systems Architecture)

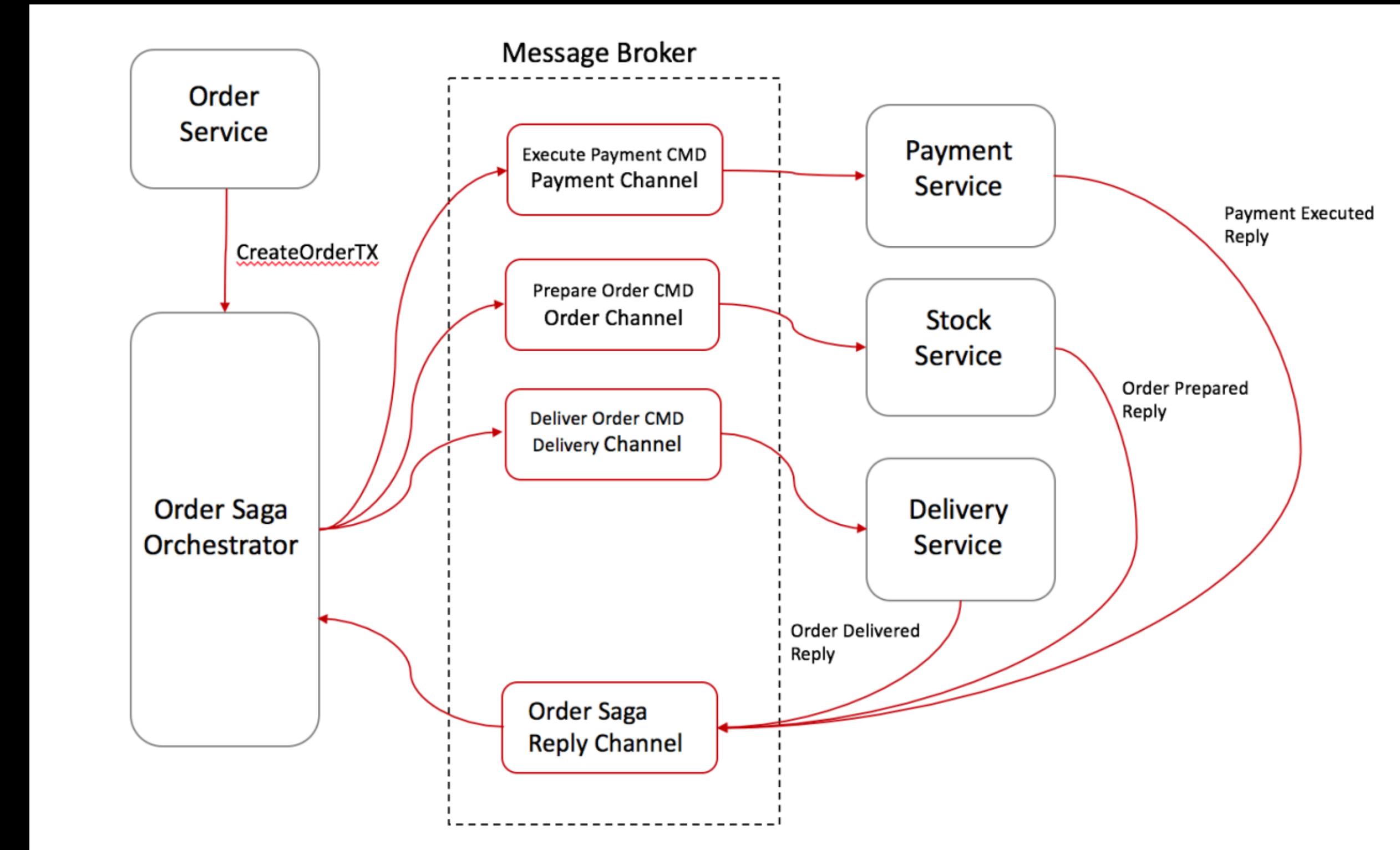
- Metadane transferowane między modułami powinny być ustandaryzowane
- Operacje obejmujące konfigurację, monitoring, tracing czy wdrażanie powinny być ustandaryzowane
- Każdy moduł powinien mieć własny / niezależny pipeline
- Moduły powinny zapewniać ciągłość działania usług - funkcjonować niezależnie od działania pozostałych modułów lub braku komunikacji z nimi

<https://isa-principles.org>

# Logika zdalna

- Logika dostępna przez sieć, realizowana w formie rozproszonego systemu
- Komunikacja może odbywać się na różne sposoby i z użyciem różnych protokołów np.:
  - protokoły tekstowe i binarne
  - synchronicznie i asynchronicznie
  - komunikacja oparta o zdarzenia / wiadomości, rpc lub transferowanie stanu zasobów

# Wzorzec Saga i orkiestracja logiki



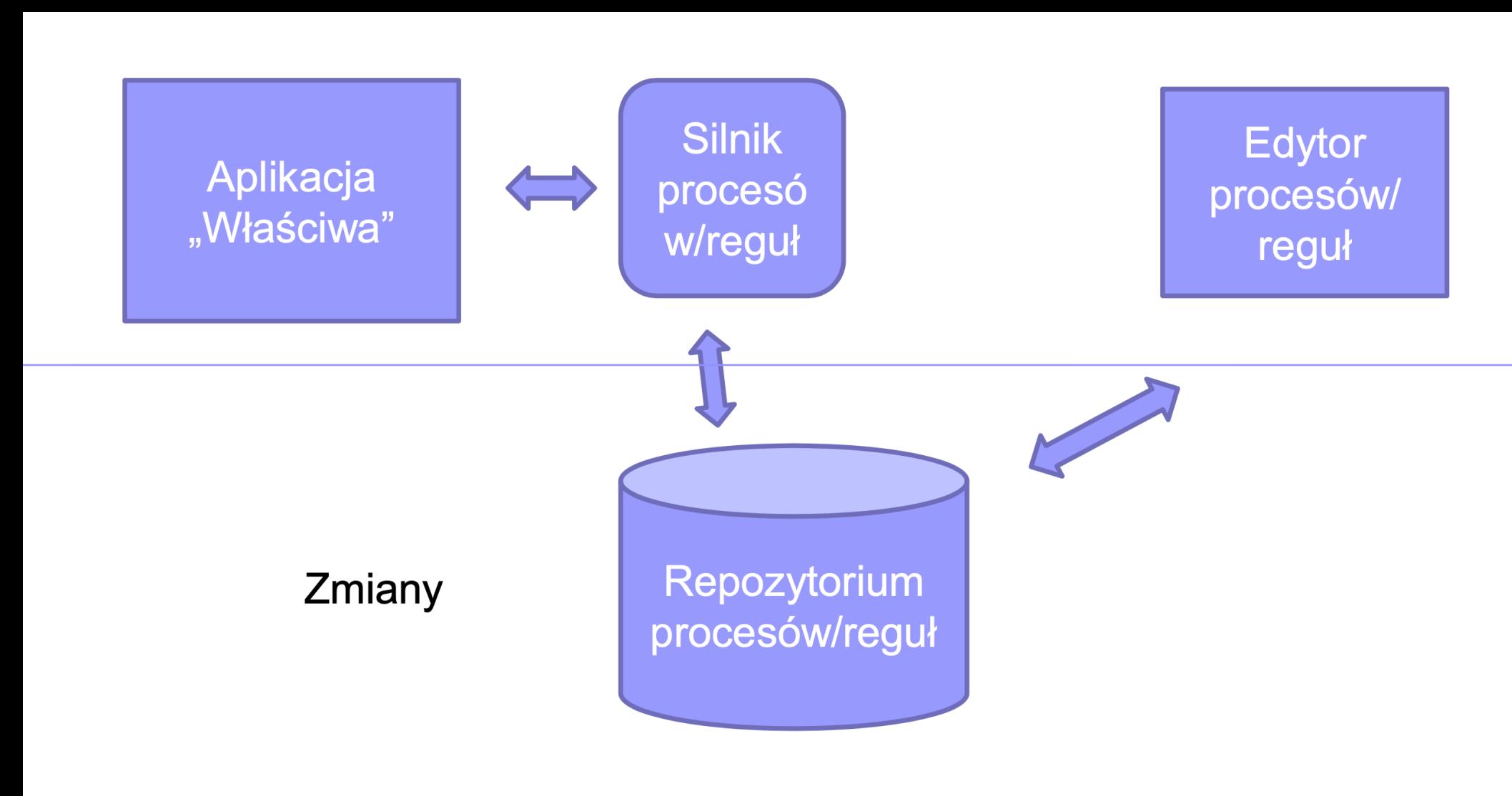
<https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part-2>

# Wyzwania związane z logiką biznesową

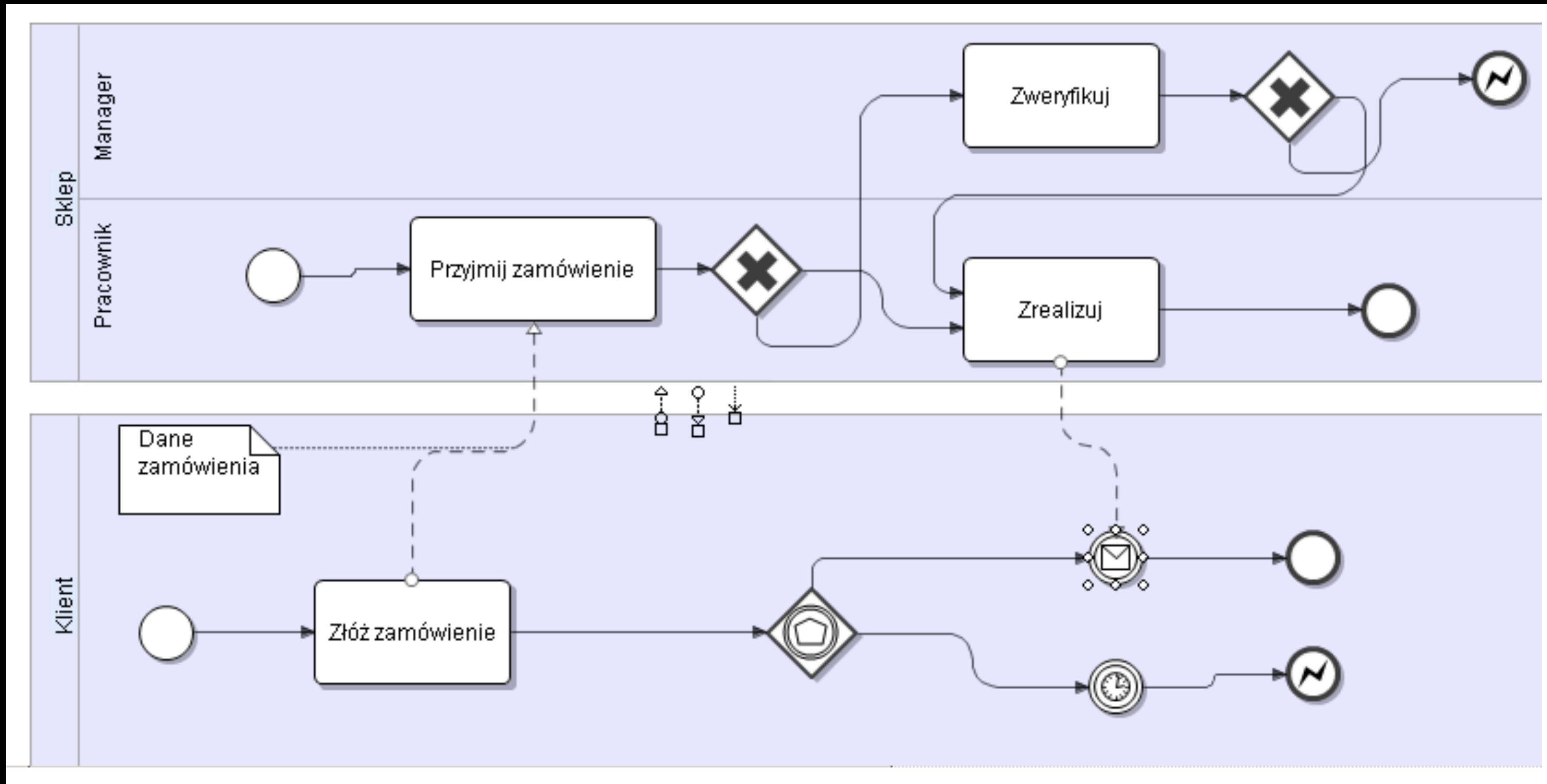
- Jak sprawić aby oprogramowanie nadążało za zmianami niezbędnymi dla funkcjonowania biznesu?
- Jak precyzyjnie określić kiedy wprowadzane zmiany wchodzą w życie?
- Jak aplikować zmiany tylko dla wybranych klientów?
- Jak określić kto i co może zmieniać oraz zachować historię zmian?
- ...

# Silniki reguł i procesów biznesowych

- Oprogramowanie umożliwiające wykonanie i monitorowanie reguł wyrażonych formalnym językiem
- Pozwala wyodrębnić z kodu aplikacji procesy i reguły, które mogą być tworzone i zarządzane z zewnątrz



# Proces BPMN



# Drools

```
rule "Reject Young"
when
    a:Application( age < 18 , status == Application.PENDING)
then
    a.setStatus(Application.REJECTED);
    update(a);
end
```

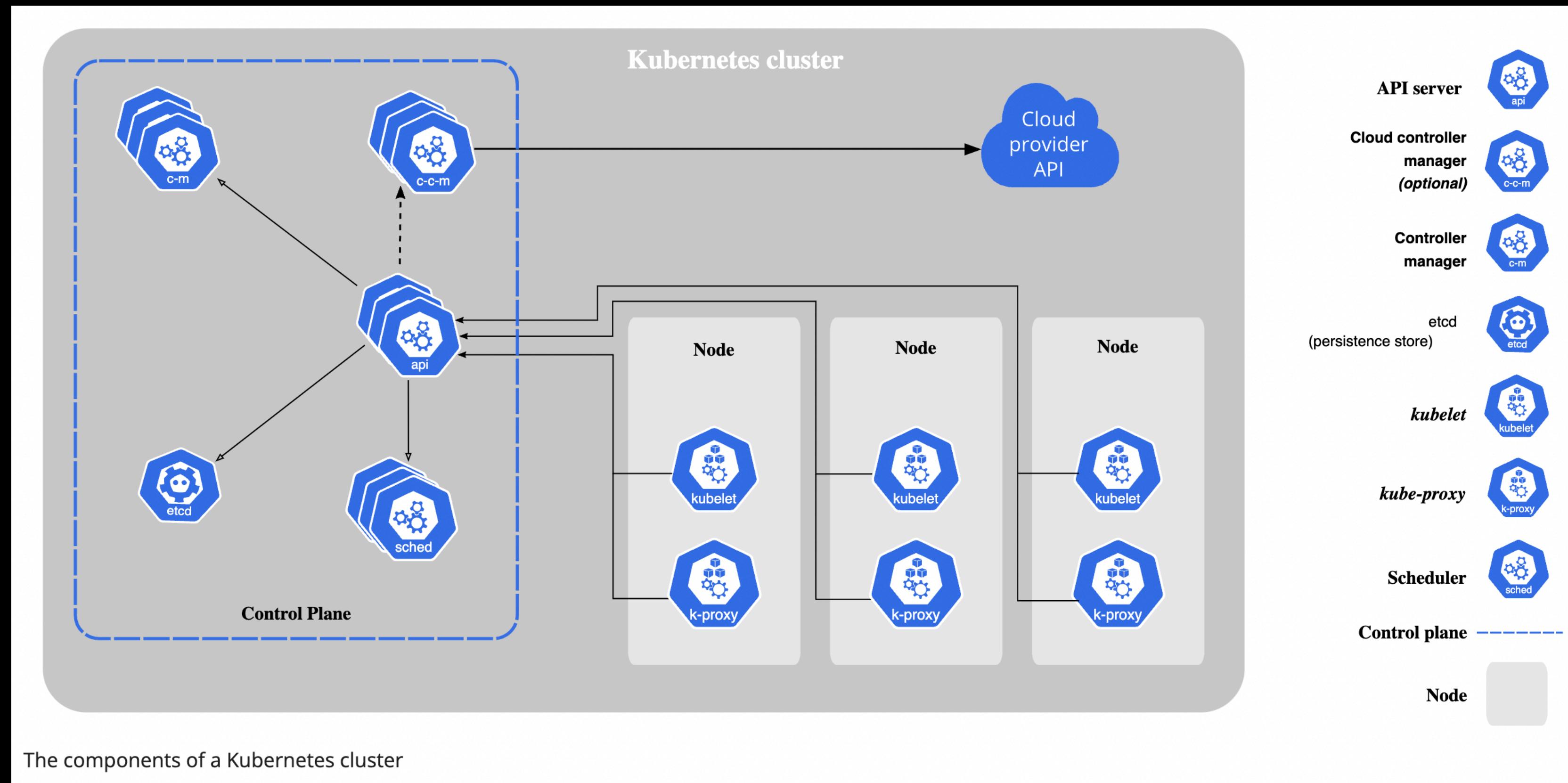
# Wyzwania związane z integracją systemów

- Mnogość protokołów
- Mnogość formatów danych
- Ewolucja / zamiany API w czasie

# Kubernetes

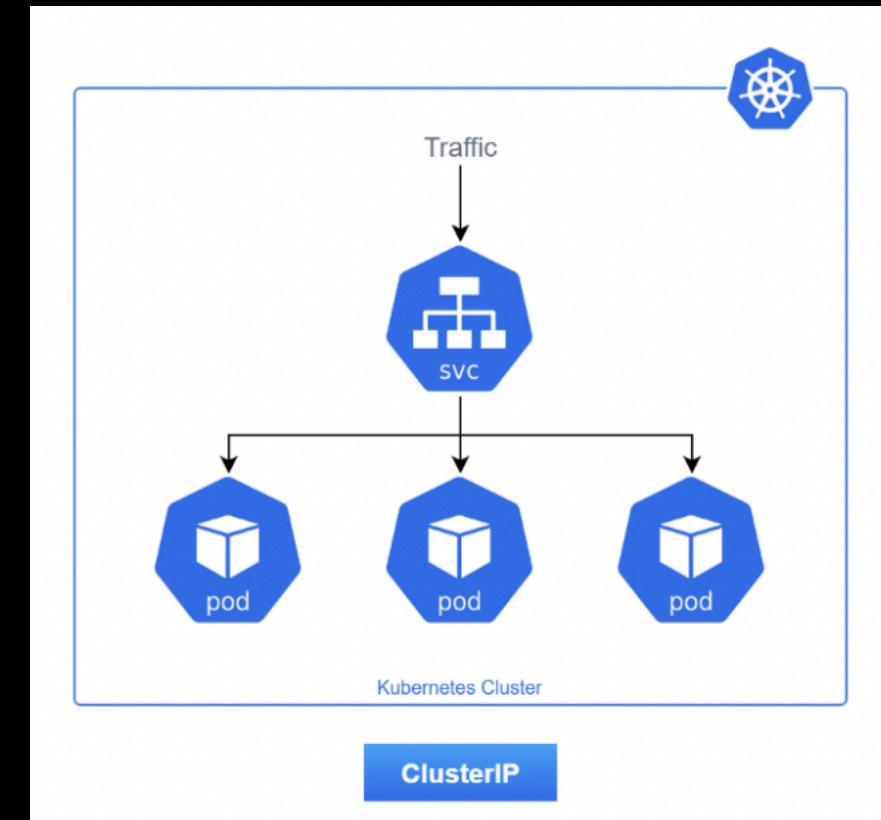
- Niezwykle popularna, otwarta, platforma do zarządzania, automatyzacji i skalowania aplikacji kontenerowych

# Architektura Kubernetes



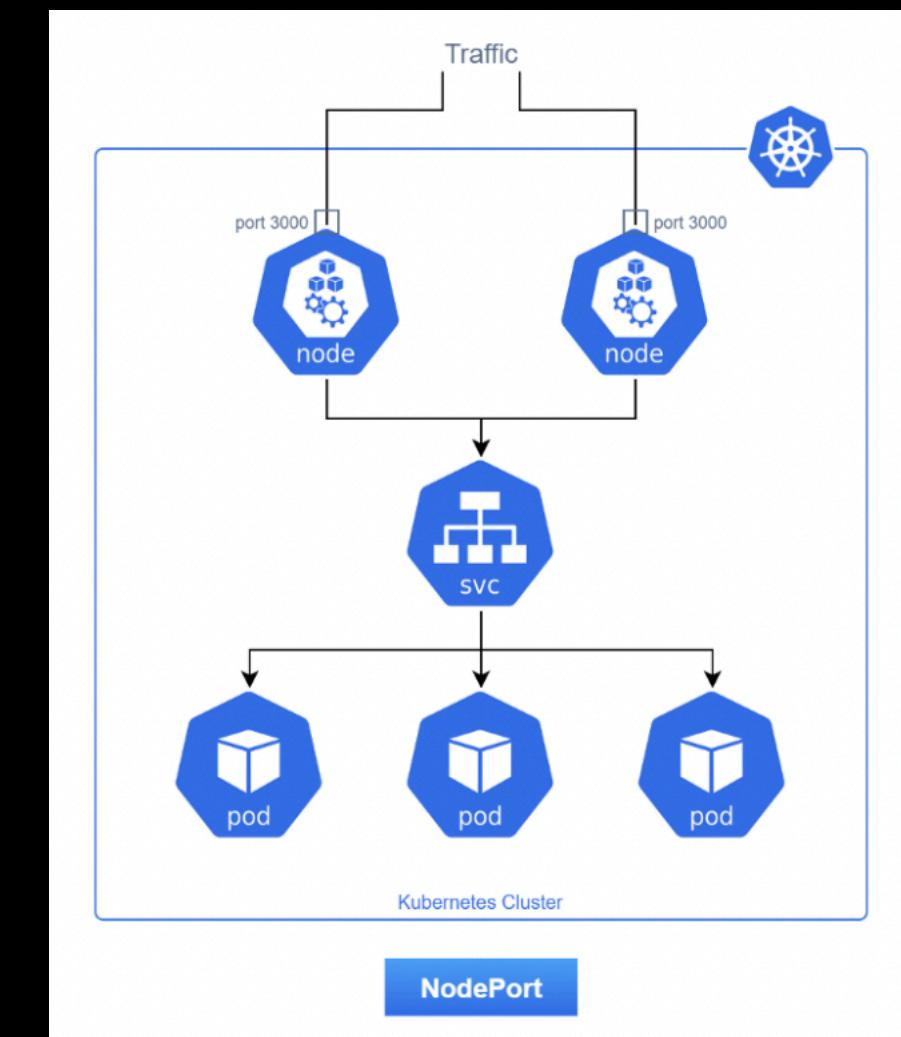
# Cluster IP Service

- Domyślny i najpopularniejszy typ usługi
- Definiuje wyłącznie wewnętrzny adres IP (zapewnia dostęp z wnętrza klastra)



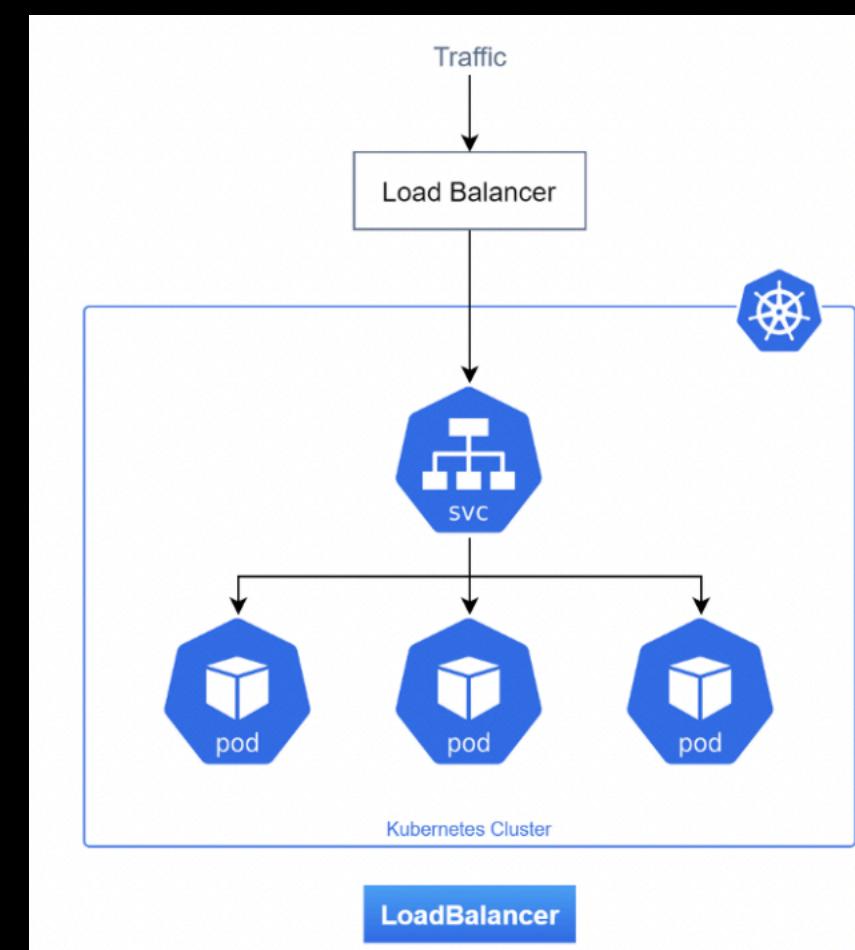
# NodePort Service

- Rozszerzenie wariantu ClusterIP
- Definiuje wewnętrzny adres IP oraz port z przedziału 30000–32767 na poziomie maszyn klastra umożliwiając dodatkowy dostęp z zewnątrz



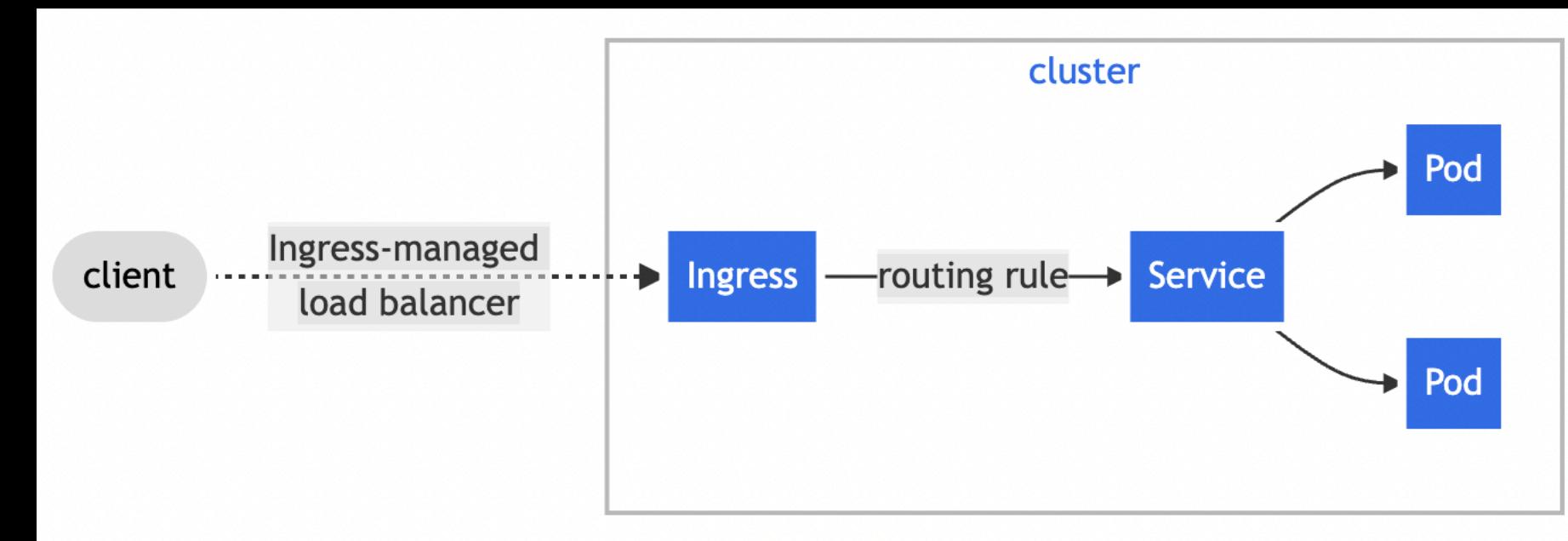
# LoadBalancer Service

- Rozszerzenie wariantu NodePort
- Zapewnia dodatkową integrację z rozwiązaniami typu load balancers, oferując dostęp do usług w postaci dedykowanego, zewnętrznego adresu IP



# Ingress

- Udostępnia, mapuje żądania http/https na serwisy wewnątrz klastra zgodnie z ustalonymi regułami routingu



# Kafka

- Pośrednik w komunikacji (MOM), stworzony przez LinkedIn, aktualnie otwarty
- Zapewnia bardzo wysoką wydajność / przepustowość, skalowalność, a także gwarantuje bezpieczeństwo (replikacja danych)
- Wykorzystanie: systemy wymiany wiadomości, śledzenie aktywności, zbieranie metryk, procesowanie strumieni danych, integracja ...

# Topic

- Strumień niezmiennych danych, identyfikowany przez unikalną nazwę
- Dzielony na uporządkowane partycje (numerowane od 0 w górę)
- Wiadomości w ramach partycji otrzymują unikalne, inkrementowane id (offset)



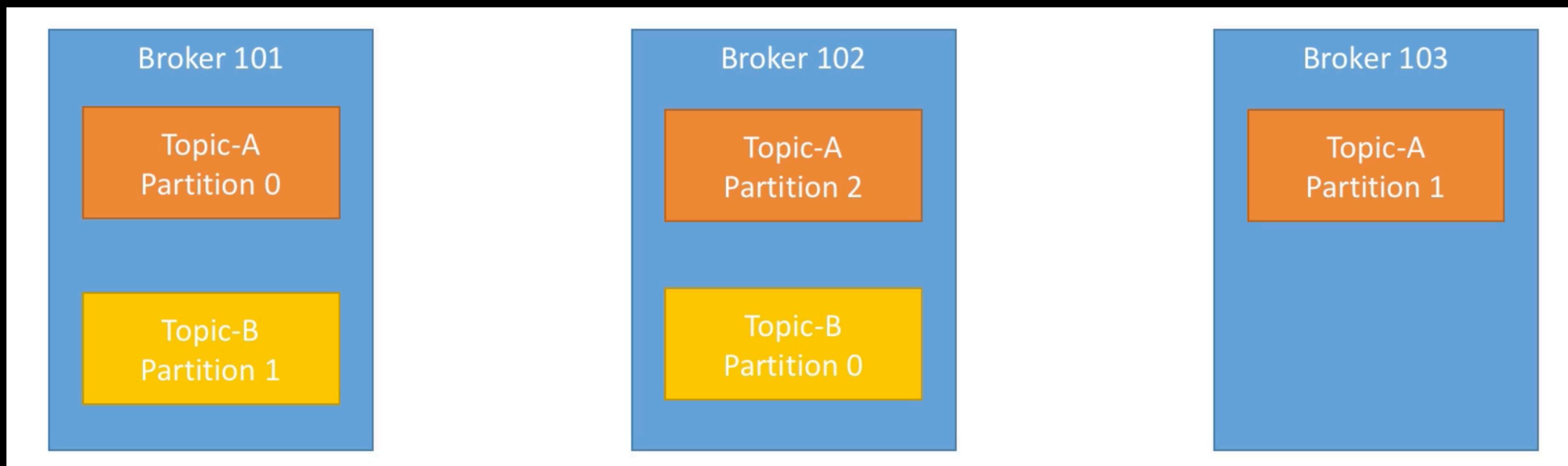
# Topic

- Kolejność wiadomości jest gwarantowana tylko w ramach partycji
- Dane są przechowywane przez określony czas (domyślnie tydzień)
- Rozdział na partycje odbywa się automatycznie



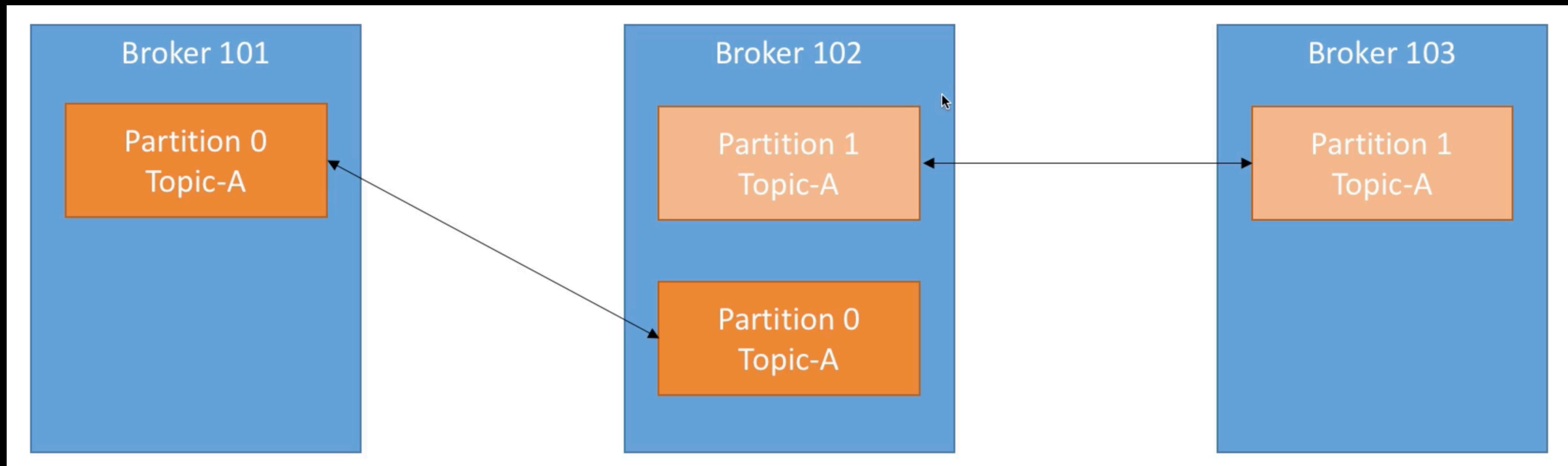
# Kafka cluster

- Klaster składa się z wielu brokerów (serwerów), z których każdy posiada unikalny identyfikator
- Każdy broker zawiera tylko wybrane partycje danego tematu
- Po połączeniu z jednym brokerem mamy dostęp do całego klastra



# Replication factor

- Określa poziom replikacji danych
- Przykład: 2 partycje z poziomem replikacji ustawionym na 2



# Partition leader

- Każda partycja posiada dokładnie jednego lidera, odpowiedzialnego za zapis i odczyt danych
- Pozostali brokerzy to tzw. in-sync replicas (tylko replikują stan lidera)
- W przypadku kiedy lider staje się niedostępny, wybierany jest nowy (z udziałem zookeepera)

# Producer

- Wysyła / zapisuje dane do tematu
- Rozwiązywanie brokera i partycji realizowane jest automatyczne (nie musimy ich wskazywać, ani martwić się ich dostępnością)
- Potwierdzenie zapisu wiadomości może odbywać się na 3 poziomach:
  - `acks=0` - producent nie czeka na potwierdzenie
  - `acks=1` - producent czeka na potwierdzenie od lidera (domyśle ustawienie)
  - `acks=all` - producent czeka na potwierdzenie od lidera i wszystkich replik

# Producer

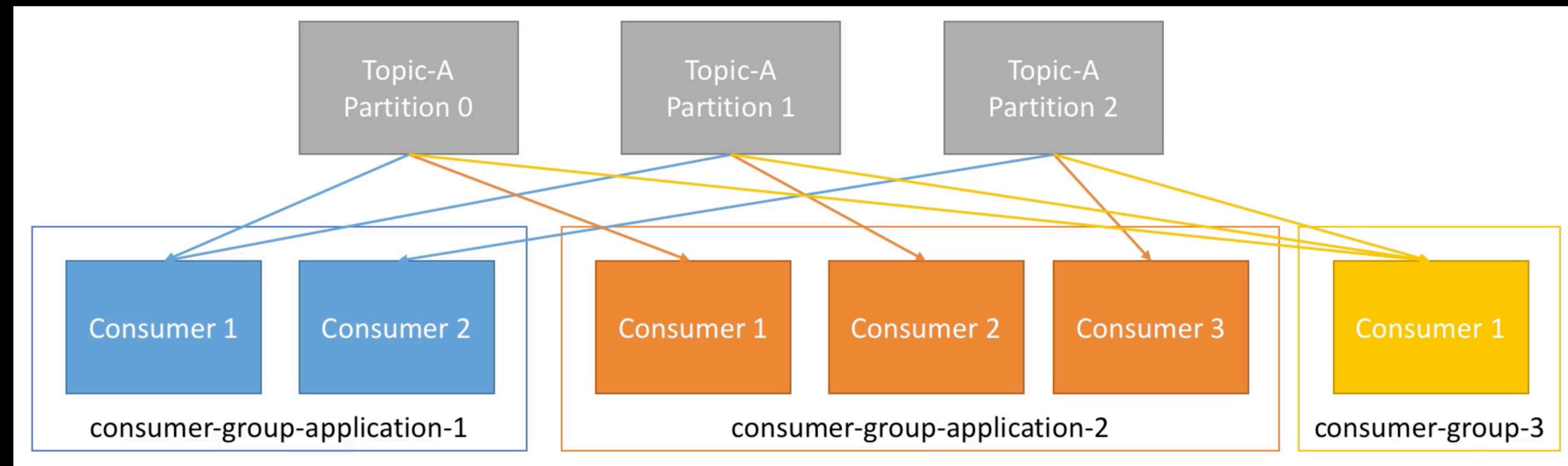
- Razem z wiadomością może zostać wysyłany klucz (tekst, liczba, ...)
- Jeśli klucz jest zdefiniowany, mamy zagwarantowane że wiadomość trafi zawsze na tą samą partycję
- Jeśli klucz nie jest zdefiniowany broker i partycja wybierany jest na podstawie mechanizmu round robin

# Consumer

- Czyta dane z tematu o danej nazwie
- Wybór brokera jest automatyczny, a jego niedostępność jest transparentna dla konsumenta
- Dane są czytane w kolejności tylko w ramach partycji
- Dane mogą być czytane z wielu partycji jednocześnie

# Consumer groups

- Konsumenti mogą być łączeni w grupy
- Każda partycja jest czytana tylko przez jednego konsumenta z grupy
- Jeśli konsumentów w grupie jest więcej niż partycji, będą oni nieaktywni



# Consumers offsets

- Kafka przechowuje max. odczytany offset, zapisywany w momencie potwierdzenia odbioru wiadomości przez konsumenta (temat `_consumer_offsets`)
- Offset umożliwia ponowne przetworzenie wiadomości w momencie awarii
- Poziomy potwierdzenia:
  - At most once - potwierdzenie zaraz po otrzymaniu wiadomości, jeśli procesowanie się nie powiedzie, wiadomość zostanie utracona
  - At least once - potwierdzenie po przeprocesowaniu wiadomości, może prowadzić do jej wielokrotnego przetworzenia
  - Exactly once - dla układu Kafka <-> Kafka z wykorzystaniem Kafka Streams API lub Kafka <-> idempotent systems

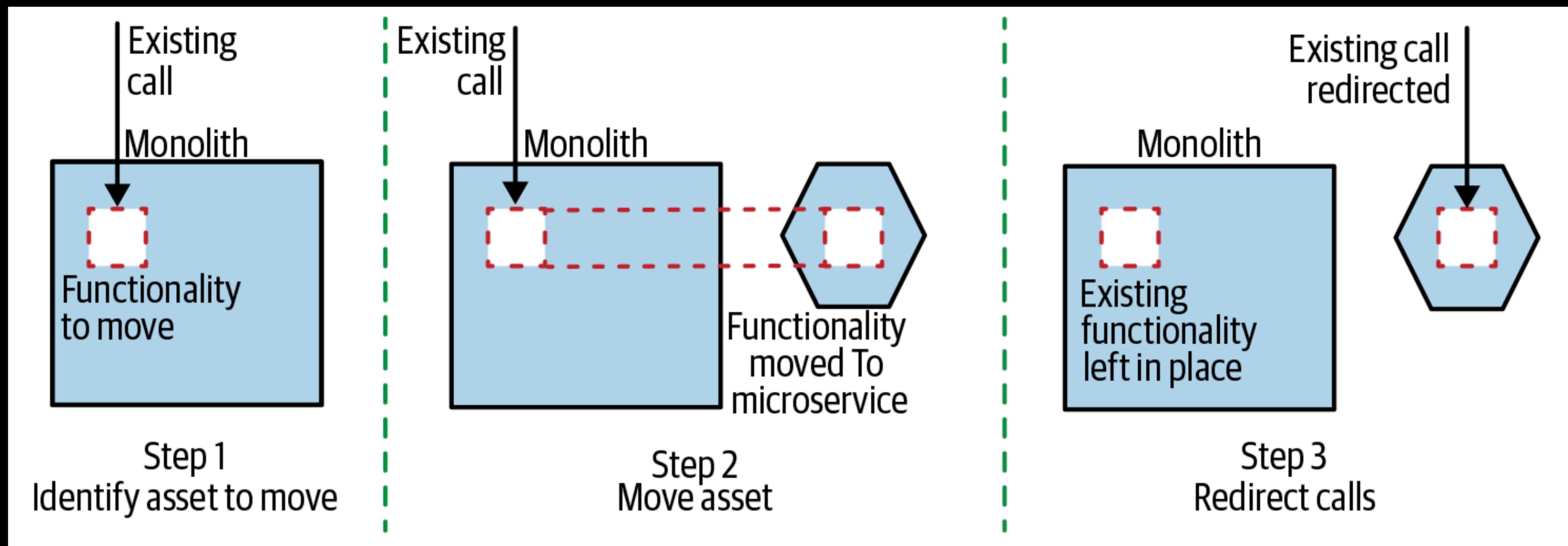
# Broker discovery

- Każdy broker wie o wszystkich pozostałych serwerach, a także o wszystkich tematach i partycjach
- Wystarczy, że podłączymy się do jednego brokera (tzw. bootstrap server) i mamy dostęp do wszystkich pozostałych

# Zookeeper

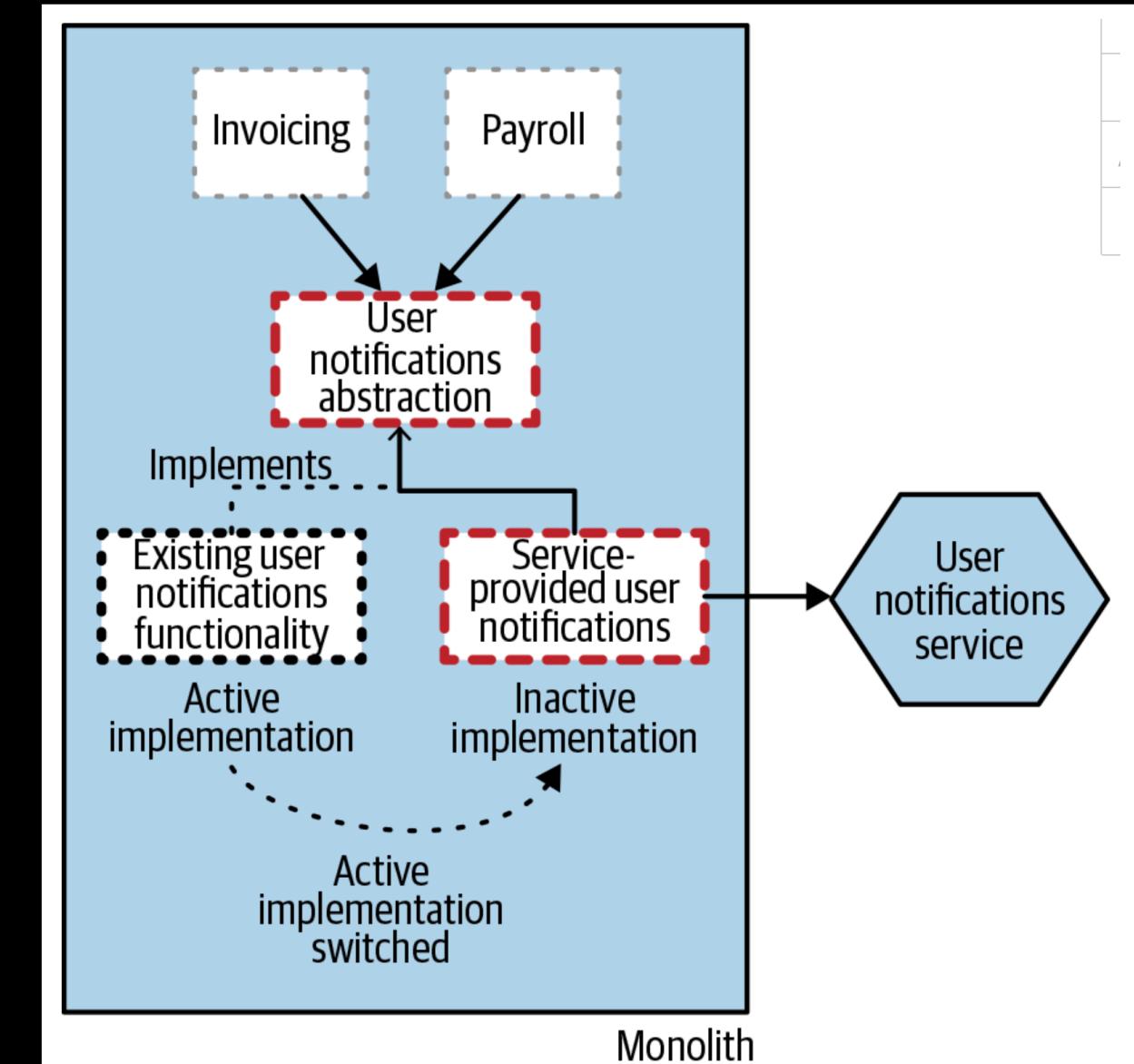
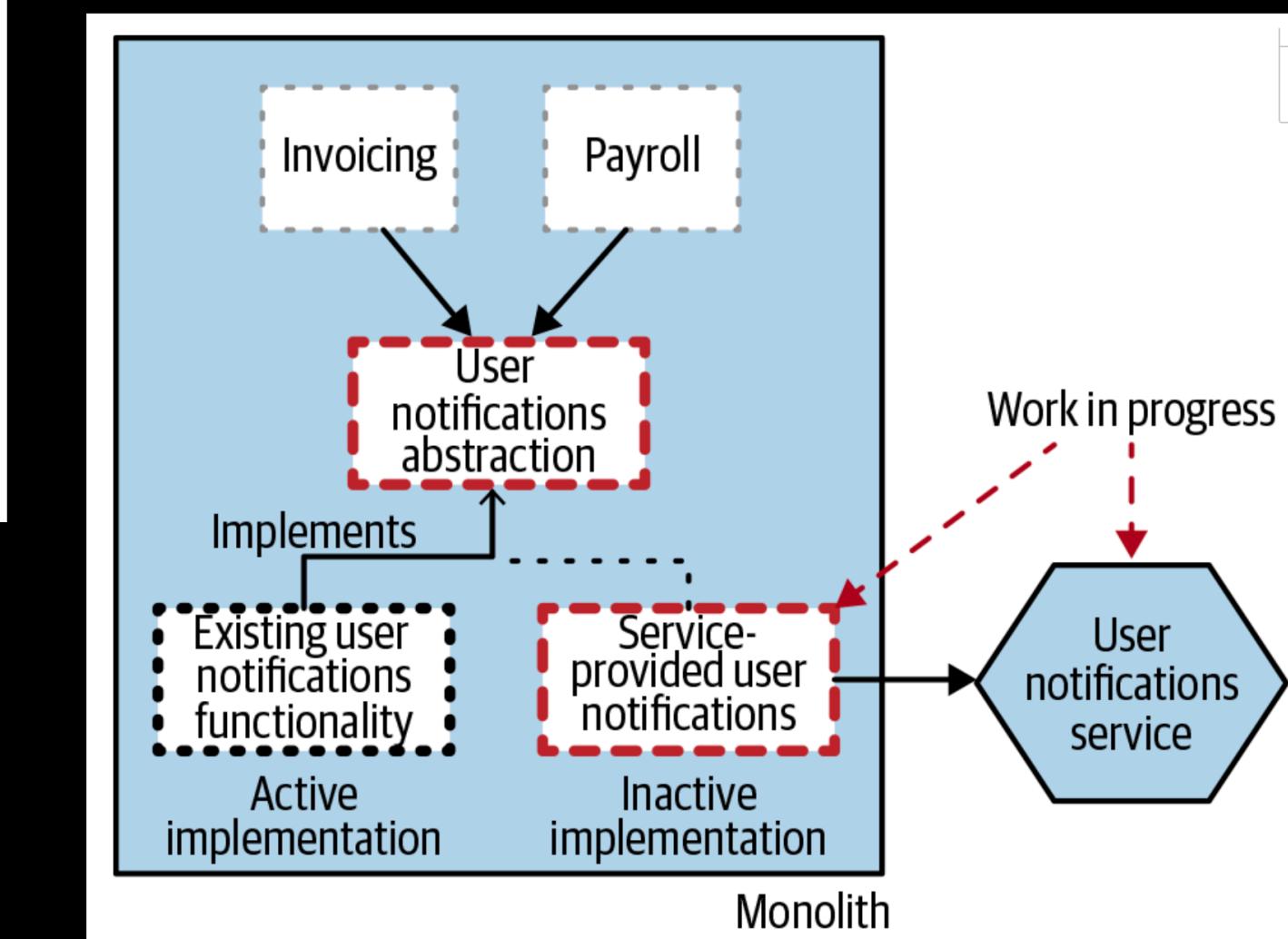
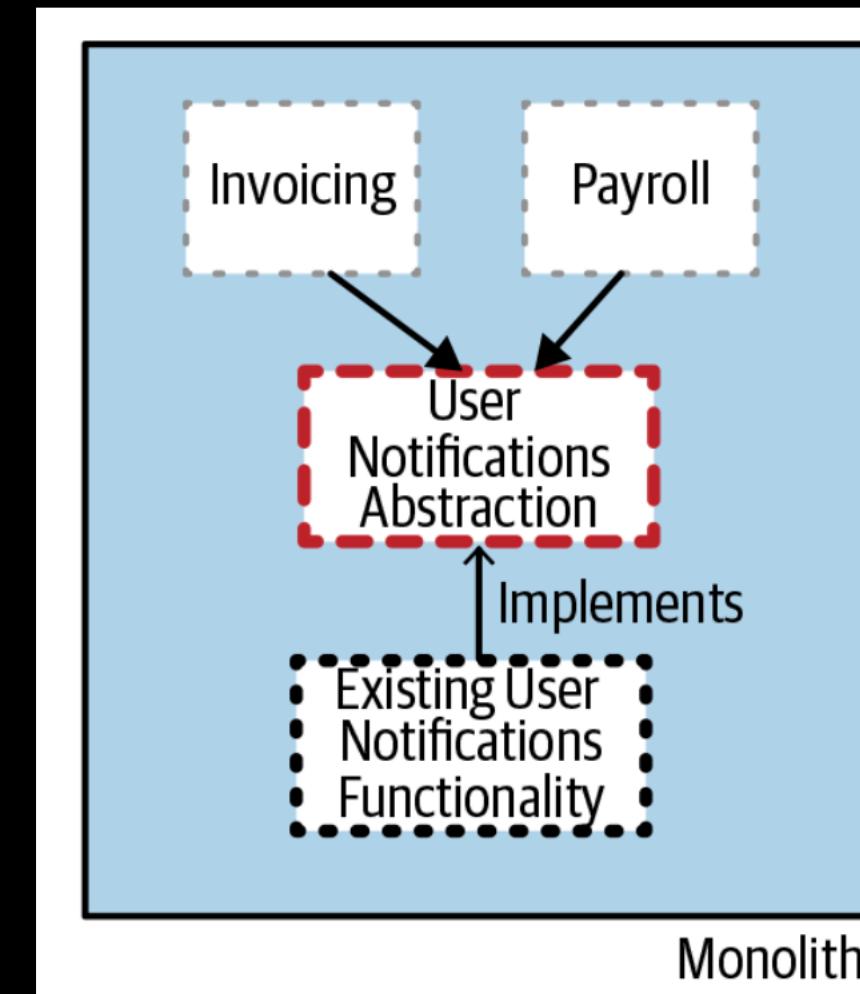
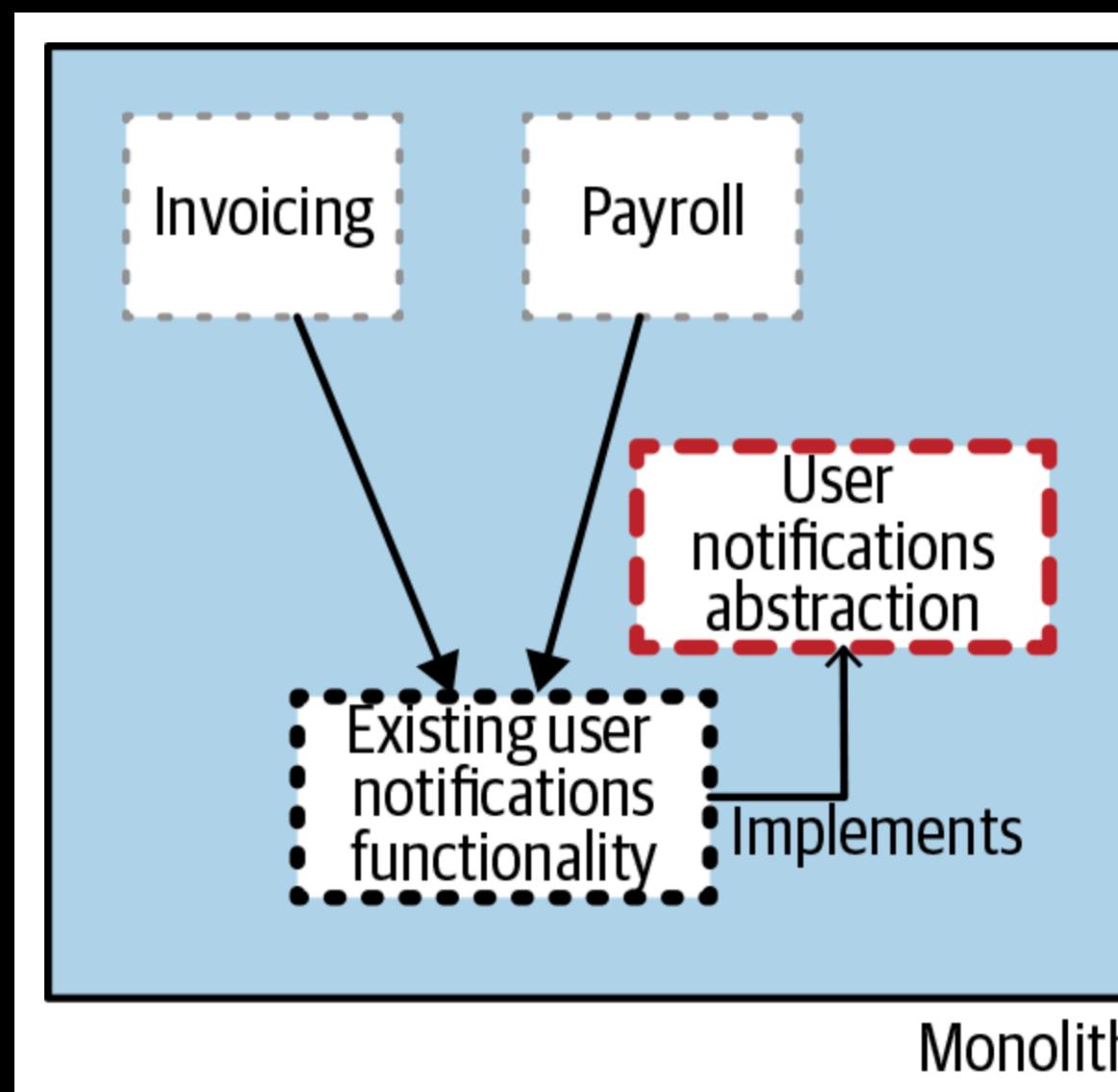
- Zarządza brokerami
- Pomaga przy wyborze lidera
- Powiadamia o zmianach (utworzenie tematu, dodanie brokera, śmierć brokera...)
- Powinien być wdrażany w nieparzystej liczbie (3, 5, 7...)
- Posiada lidera (zapis) i repliki (odczyt)

# Pattern: Strangler Fig Application



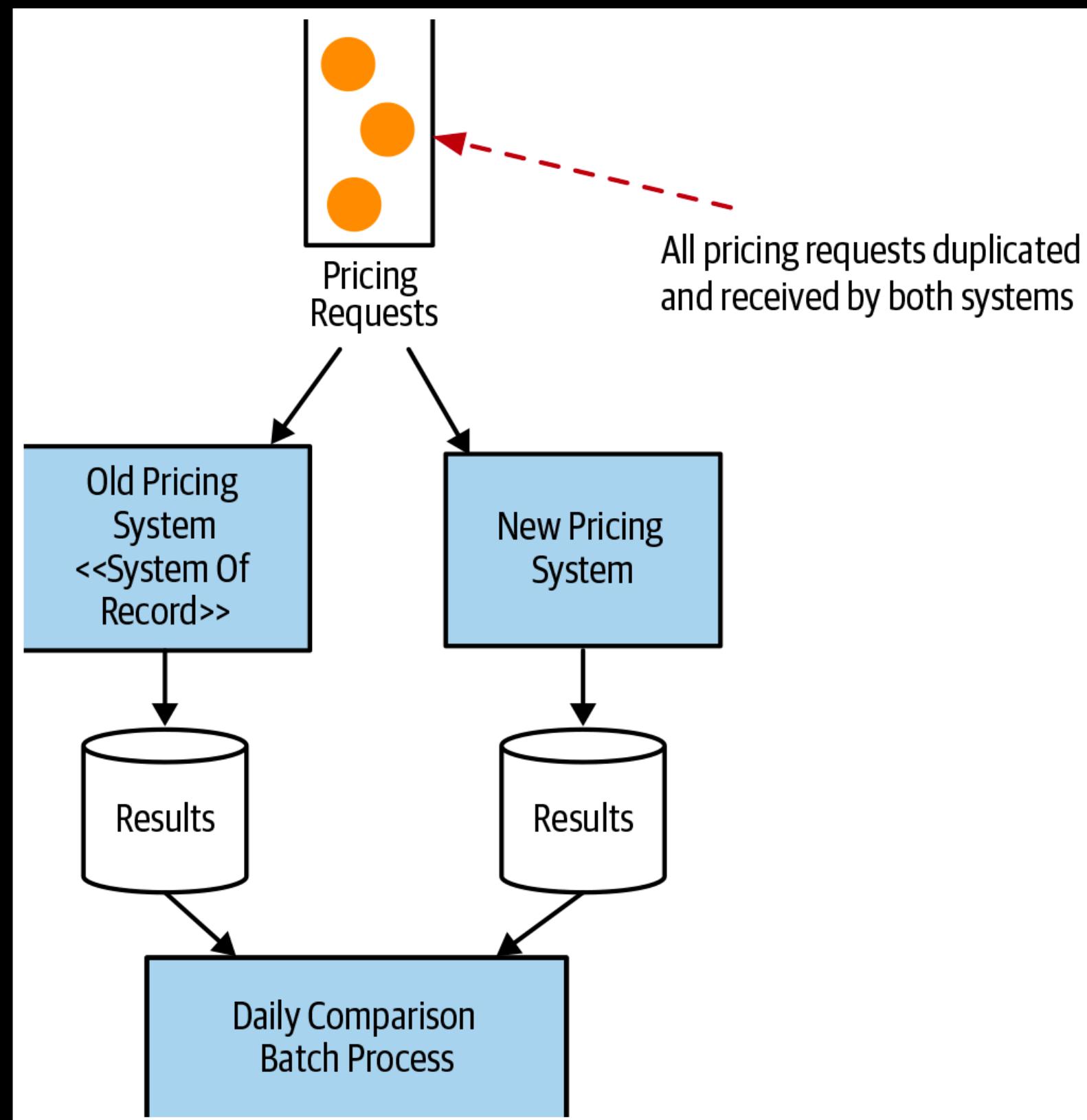
<http://shop.oreilly.com/product/0636920233169.do>

# Pattern: Branch by Abstraction



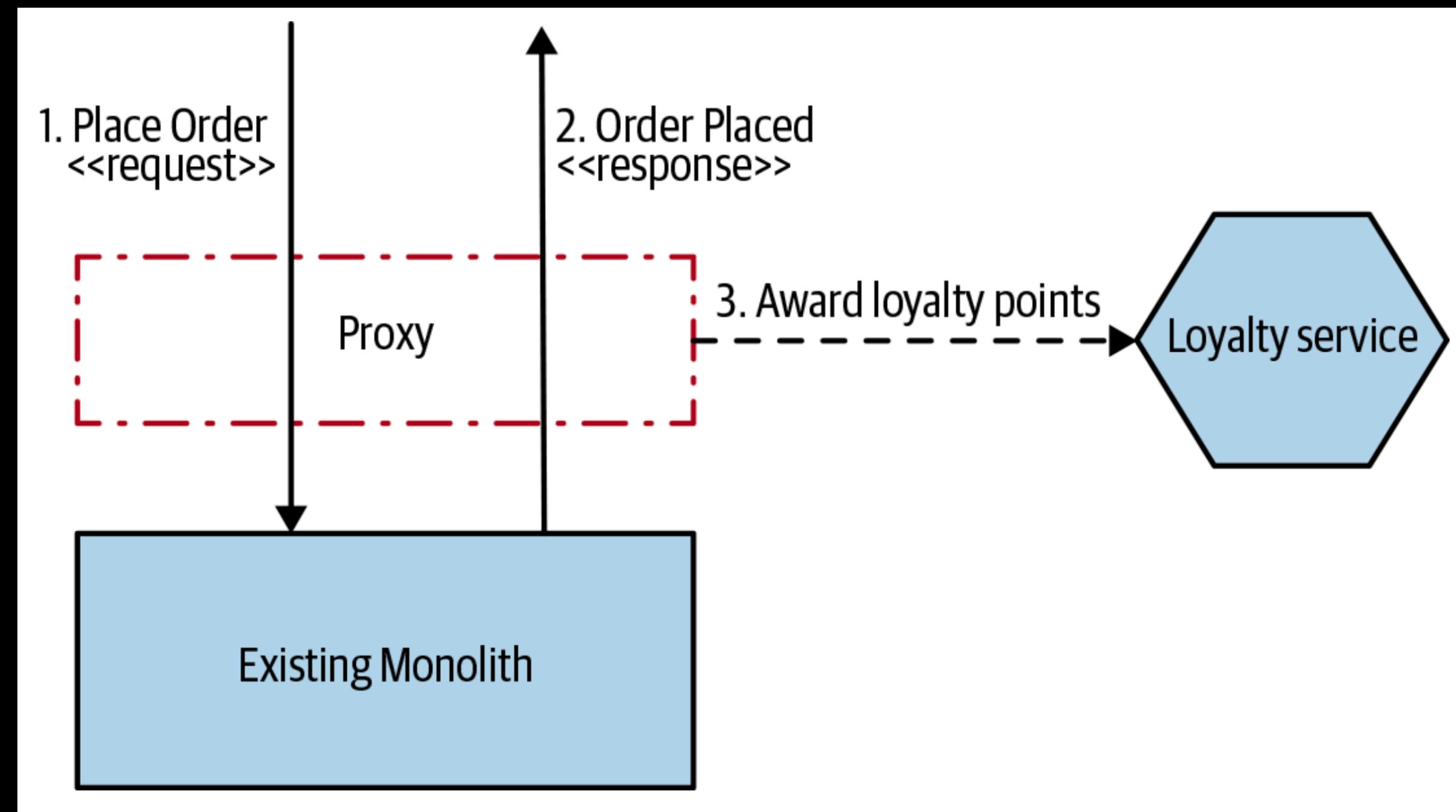
<http://shop.oreilly.com/product/0636920233169.do>

# Pattern: Parallel Run



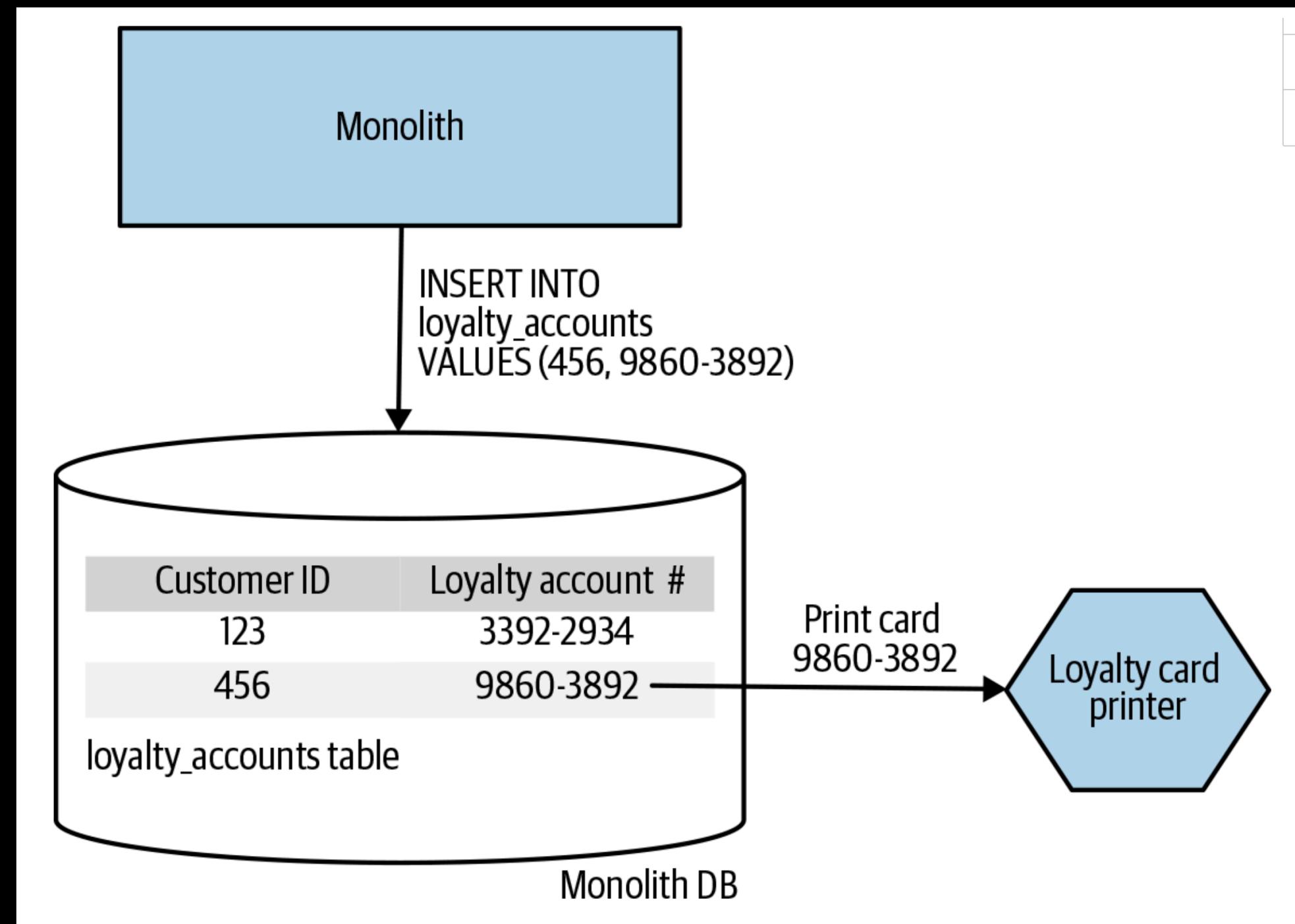
<http://shop.oreilly.com/product/0636920233169.do>

# Pattern: Decorating Collaborator



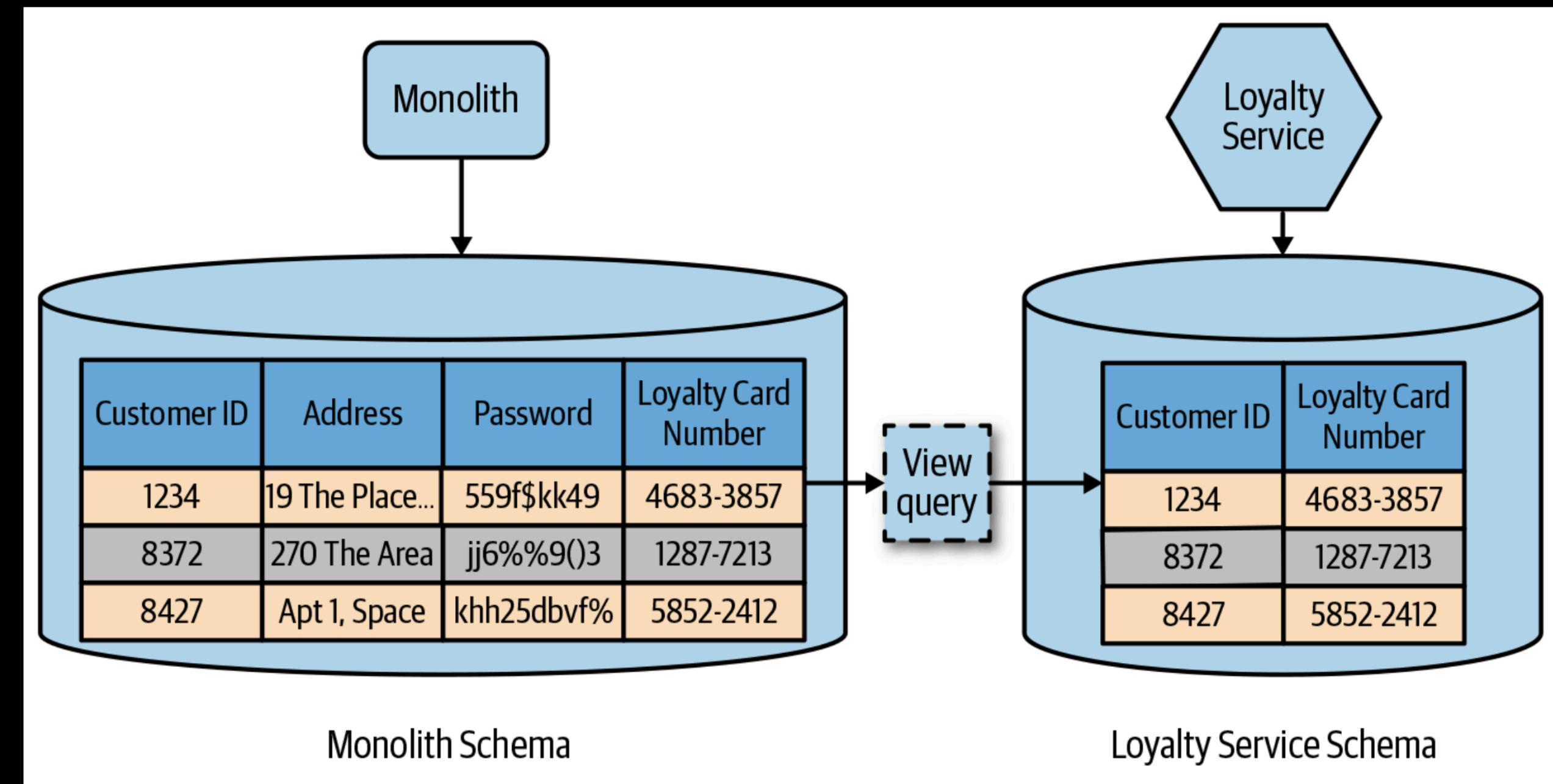
<http://shop.oreilly.com/product/0636920233169.do>

# Pattern: Change Data Capture



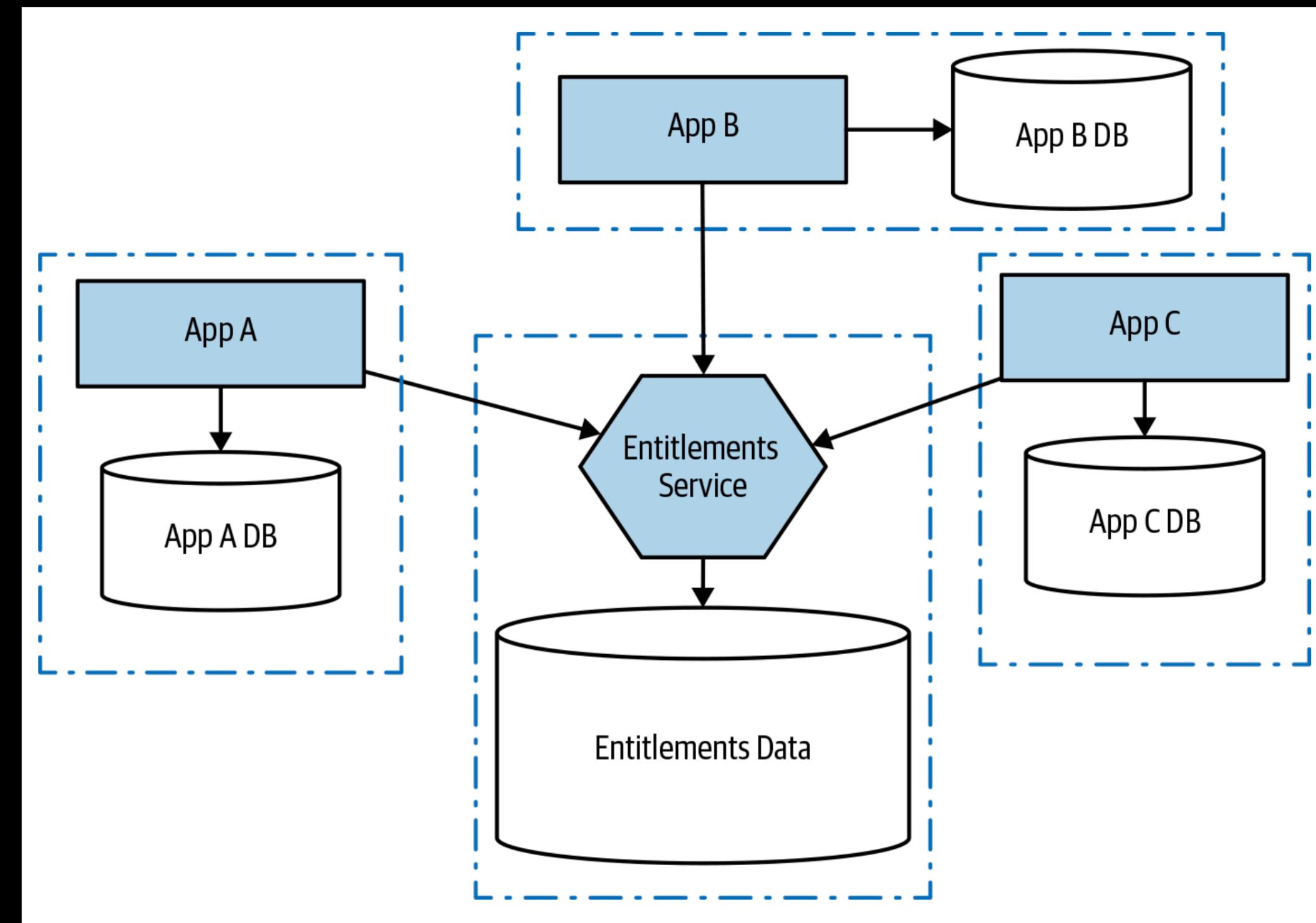
<http://shop.oreilly.com/product/0636920233169.do>

# Pattern: Database View



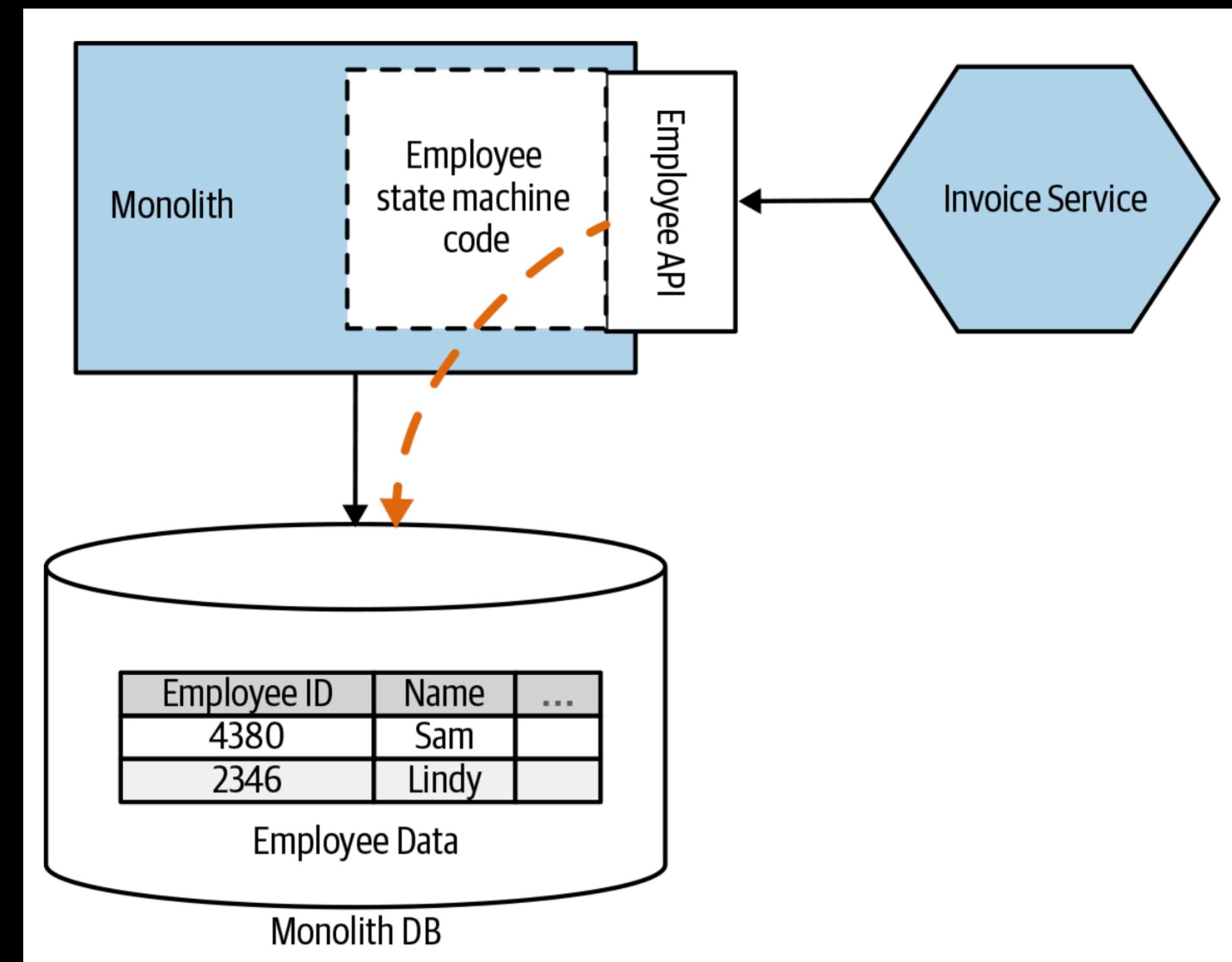
<http://shop.oreilly.com/product/0636920233169.do>

# Pattern: Database Wrapping Service



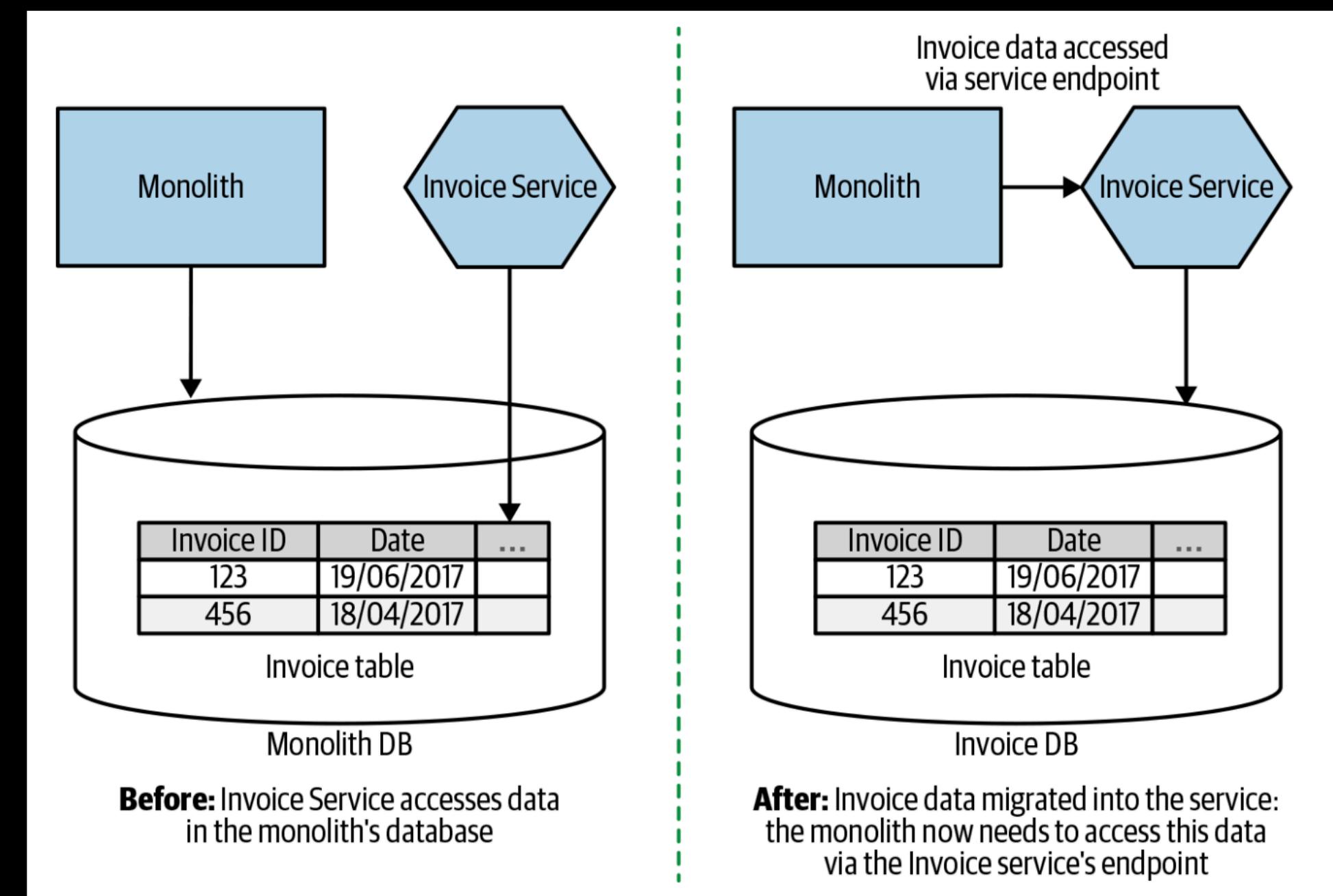
<http://shop.oreilly.com/product/0636920233169.do>

# Pattern: Aggregate Exposing Monolith



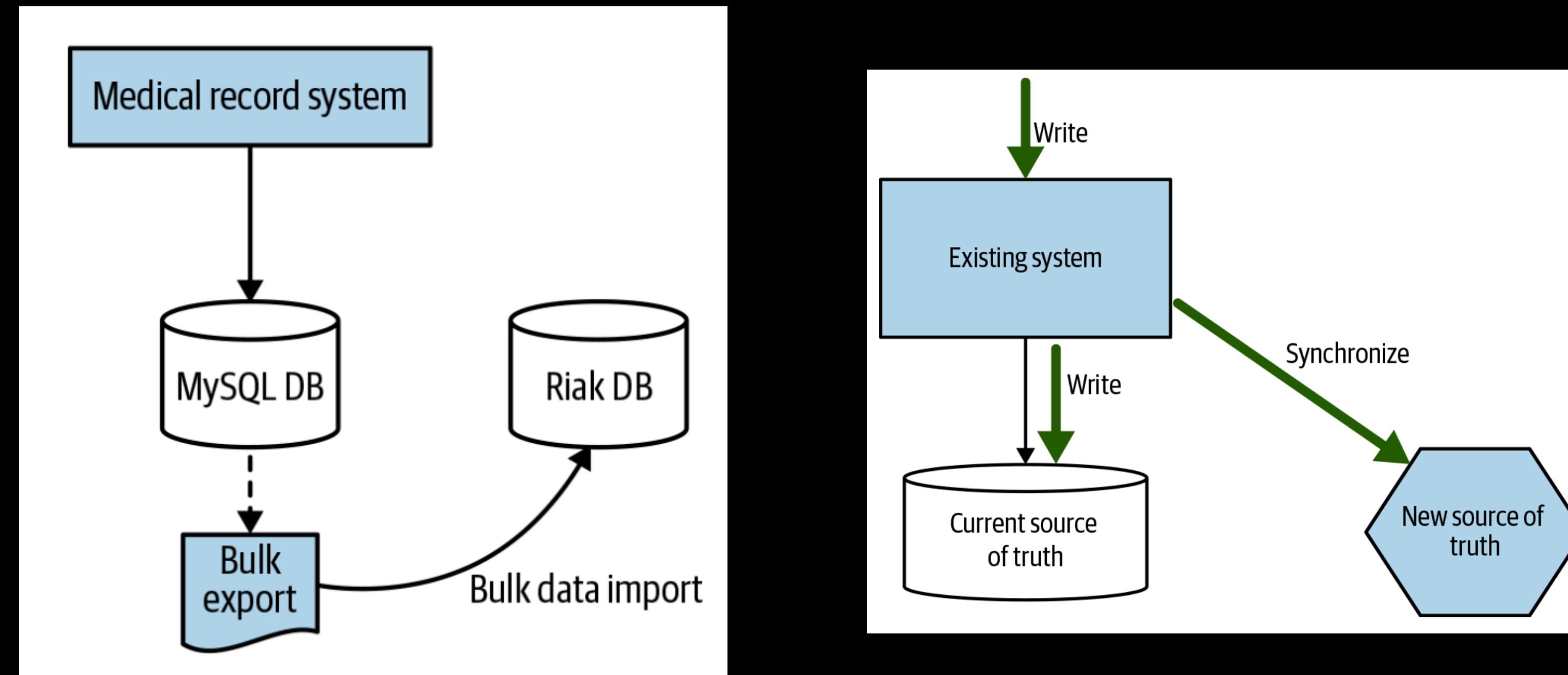
<http://shop.oreilly.com/product/0636920233169.do>

# Pattern: Change Data Ownership



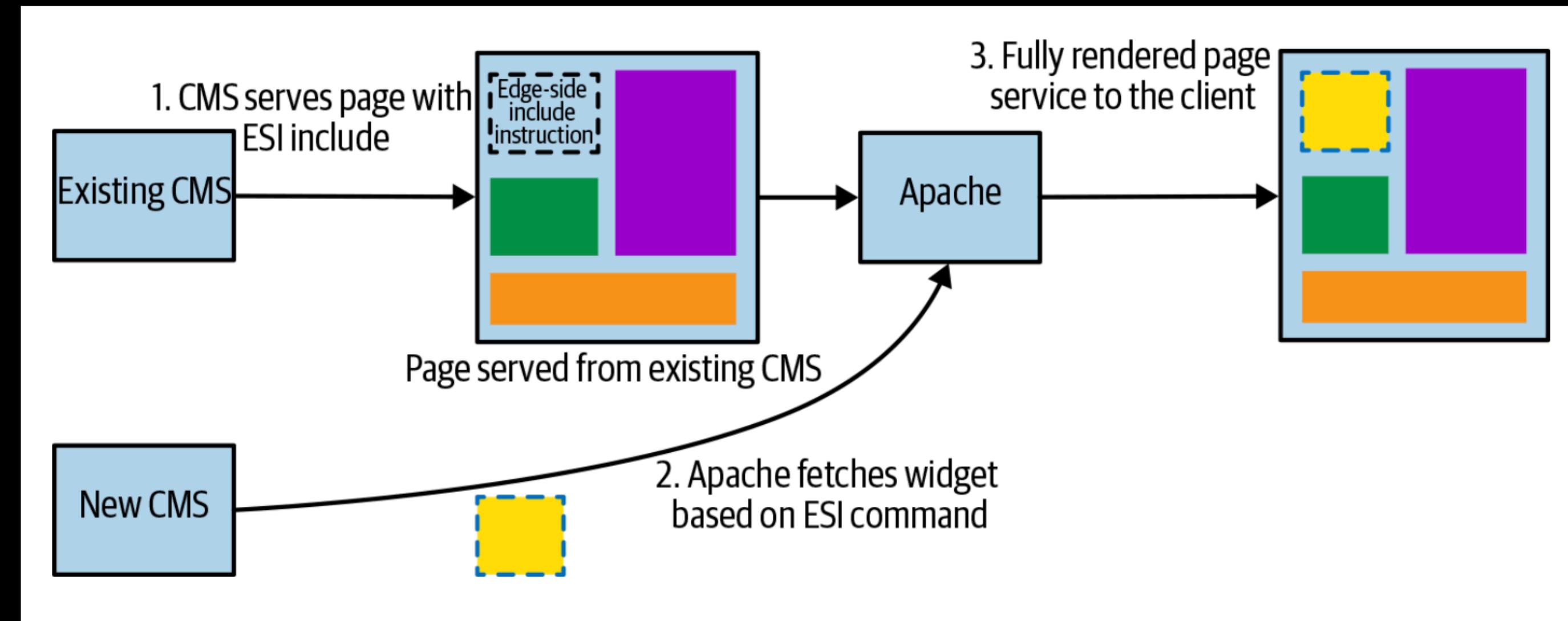
<http://shop.oreilly.com/product/0636920233169.do>

# Pattern: Synchronize Data Application



<http://shop.oreilly.com/product/0636920233169.do>

# Pattern: UI Composition



<http://shop.oreilly.com/product/0636920233169.do>

# Microservices anti-patterns

- <https://dzone.com/articles/microservices-anti-patterns>
- <https://www.oreilly.com/content/microservices-antipatterns-and-pitfalls>
- <https://www.infoq.com/articles/seven-uservices-antipatterns>