JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 34 (2025) 81–88
Received: 21 October 2024
Accepted: 2 November 2024

Comparative analysis of selected aspects of web application architectures

Analiza porównawcza wybranych aspektów architektur aplikacji internetowych

Łukasz Krzysztoń*, Konrad Paweł Łatwiński, Małgorzata Plechawska-Wójcik

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract
The article presents a comparative analysis of two web application architectures: microservices and modular monoliths. The research focuses on evaluating the impact of infrastructure, inter-component communication, and horizontal scaling on system performance. For the purpose of the study, a dedicated system was implemented in two versions: as microservices and as a modular monolith. This allowed for a direct comparison of both architectures under identical experimental conditions. The application underwent a series of performance tests, including simulations of various load levels and analysis of response times in different operational scenarios. The collected data was presented in the form of statistics and graphs, facilitating result interpretation and allowing for the verification of the research hypotheses.

Keywords: microservices; modular monolith; application architecture; application performance

Streszczenie
Artykuł przedstawia analizę porównawczą dwóch architektur aplikacji internetowych: mikroserwisów i modularnego monolitu. Badania koncentrują się na ocenie wpływu infrastruktury, komunikacji między-komponentowej oraz horyzontalnego skalowania na wydajność systemu. Na potrzeby badań zaimplementowano dedykowany system, który został opracowany w dwóch wersjach: jako mikroserwisy oraz jako modularny monolit. Pozwoliło to na bezpośrednie porównanie obu architektur w identycznych warunkach eksperymentalnych. Aplikacja została poddana serii testów wydajnościowych, w tym symulacji różnych poziomów obciążenia oraz analizy czasu odpowiedzi w różnych scenariuszach operacyjnych. Otrzymane dane zostały przedstawione w formie statystyk i wykresów, ułatwiających interpretację wyników oraz pozwoliły na weryfikację postawionych hipotez badawczych.

Słowa kluczowe: mikroserwisy; modularny monolit; architektura aplikacji; wydajność aplikacji

*Corresponding author
Email address: lukasz.krzyszton@pollub.edu.pl (Ł. Krzysztoń)

1. Wstęp

Współczesne środowisko informatyczne dynamicznie ewoluuje, napotykając nowe wyzwania i wymagania. Wraz z rosnącym zapotrzebowaniem na skalowalność, elastyczność i łatwość utrzymania systemów informatycznych, architekci oprogramowania zwracają uwagę na różne strategie projektowe. Architektura ma bardzo duży wpływ na koszty związane z infrastrukturą oraz zasobami ludzkimi, dobrze dobrana może pozwolić zredukować koszty operacyjne oraz zapewnić optymalną wydajność i pozwolić efektywnie zarządzać zasobami systemu [1]. Wszystko w architekturze jest kompromisem [2], dlatego nie da się jednoznacznie powiedzieć, jaka architektura jest najlepsza, wybór zależy od konkretnego przypadku i potrzeby użycia. Dwie z najbardziej prominentnych koncepcji architektonicznych, które wyróżniają się w dzisiejszym krajobrazie informatycznym, to architektura monolityczna oraz architektura mikroserwisów.

Architektura monolityczna jest klasycznym podejściem, które zdobyło popularność w minionych dekadach. W ciągu ostatnich lat jednak zdarza się, że pojęcie monolitu niesie ze sobą negatywne konotacje w kontekście architektury [3]. Oparta na jednym, zintegrowanym kodzie źródłowym, skupia się na budowie, spójnego systemu, w którym poszczególne funkcjonalności również mogą być zorganizowane w formie modułów. Taka struktura monolitu nosi nazwę architektury modularnego monolitu.

Architektura mikroserwisów prezentuje innowacyjną perspektywę, rozkładając system na niezależne, samodzielne serwisy, które komunikują się ze sobą poprzez interfejsy programistyczne. Oznacza to, że każdy mikroserwis jest autonomiczny, czyli może być odrębnie rozwijany, wdrażany oraz skalowany. Zaletą tego rozwiązania jest to, że każda z mikrousług może być zaimplementowana w dowolnym języku, dostosowanym do specyfikacji danego serwisu, a jedynym warunkiem jest to, aby odpowiednio dostosować punkty końcowe i protokoły komunikacyjne [4].

Poprzez przeprowadzenie analizy porównawczej architektury modularnego monolitu i mikroserwisów, praca ma na celu uwidocznić ich zalety i wady, zidentyfikować ich zastosowanie, wzbogacić dyskusję na temat wyboru architektonicznego oraz przekazać wskazówki dla decydentów i architektów.

2. Analiza literatury

Zapoznanie się z literaturą dotyczącą zbliżonych tematów do tematu porównania architektury modularnego monolitu i architektury mikrousług, umożliwia zrozumienie dotychczasowego stanu wiedzy na temat badanych zagadnień.

Vaughn Vernon i Tomasz Jaskula [5] podkreślają, że bardzo ważnym elementem architektury jest możliwość jej rozbudowy, bez ingerencji w nią samą. Ważne, jest aby była odporna na zmiany, które w perspektywie czasu są nieuniknione a zastępowanie i wymiana komponentów powinny być możliwe bez dużego nakładu pracy i ponoszenia zbędnych kosztów. Autorzy podkreślają, że dobrze zaprojektowana architektura monolityczna również umożliwia odizolowanie komponentów poszczególnych podsystemów oraz zapewnienie komunikacji między nimi. Istnieją również podejścia, które zakładają wykorzystanie architektury monolitycznej, podzielonej na moduły, na początkowym etapie prac, która docelowo ma zostać przekształcona do architektury mikroserwisowej. W pewnym stopniu to może wskazywać na pewne podobieństwo między tymi dwiema koncepcjami architektonicznymi.

Chris Richardson w swojej książce [6] opisuje natomiast wady architektury modularnego monolitu, a jako główny defekt przedstawia problemy ze skalowalnością serwerową aplikacji. Różne moduły mogą mieć sprzeczne wymagania dotyczące zasobów, a ponieważ są one częścią jednej aplikacji, podczas konfiguracji serwera trzeba znaleźć złoty środek i kompromis w dostosowaniu środowiska. W architekturze mikrousług ten problem nie występuje ponieważ instalacja poszczególnych elementów systemu może być rozłożona na różne serwery, co pozwala przygotować zasoby zgodnie z potrzebami konkretnych modułów.

Przeprowadzone badania w artykule [7] dowodzą, że to architektura mikroserwisowa pod dużym obciążeniem działa lepiej od architektury monolitycznej, pozwalając na lepszą filtrację oraz obsługę ruchu. Jest to spowodowane, między innymi, dużą elastycznością oraz rozbudowaną opcją skalowania. Jednak w przypadku scenariuszy testowych charakteryzujących się niskim obciążeniem to architektura monolityczna osiąga lepsze rezultaty. Takie wyniki pozwalają wysnuć wnioski, że decyzja dotycząca wyboru pomiędzy tymi architekturami jest zależna od konkretnego przypadku użycia oraz od przyszłego zastosowania aplikacji.

W książce [8] przedstawione zostały zalety i wady systemów napisanych w architekturach monolitycznych – rozproszonych i modułowych oraz mikrousług. Każda z koncepcji architektonicznych ma mocne i słabe strony. Jako korzyści mikrousług przedstawiono między innymi niezależność instalacji, możliwość łączenia różnych technologii oraz opcję równoległej pracy nad całym systemem przez większą grupę programistów. Jako widoczną wadę autor wskazał problem z komunikacją przez sieć, gdy poszczególne części całego systemu znajdują się na różnych komputerach, zwracając również uwagę na opóźnienia, które przekraczają te występujące w operacjach wewnątrz-procesowych. Zawarte w książce wnioski ujawniają, że opóźnienia mogą być zmienne a przez to działanie systemu nieprzewidywalne. W opisie systemów monolitycznych jako zalety zostały przedstawione proste modele instalacyjne oraz duża szansa na reużywalność kodu. Nie ma takiego problemu jak w systemie rozproszonym, gdzie trzeba decydować czy kopiować kod, wydzielać go do oddzielnych bibliotek lub też przenosić funkcje do usług. Jako główny problem systemów monolitycznych (czy to rozproszonych, czy też jednoprocesowych) została zauważona tendencja do jednoczesnej niespójności oraz ścisłego powiązania. Często zdarza się tak, że razem umieszczane są fragmenty kodu niezwiązane ze sobą, co zaburza idę dążenia do spójności i łączenia elementów modyfikowalnych łącznie.

Wpis na blogu zespołu inżynierów Amazon Prime Video [9] pokazuje, że architektura mikroserwisowa nie zawsze przewyższa architekturę monolityczną. Opisano tam narzędzie do monitorowania transmisji na żywo, którego głównym celem jest identyfikowanie problemów z jakością transmisji oraz uruchamianie procesów naprawczych. Na etapie początkowym narzędzie zostało zbudowane w architekturze mikroserwisowej, jednak znaczące koszty utrzymania oraz pojawiające się problemy ze skalowaniem związanym z obsługą dużej liczby strumieni, były przyczyną przejścia na architekturę monolityczną. Według autorów wpisu, działanie to wyeliminowało wyżej wspomniane problemy oraz pozwoliło zredukować koszty utrzymania o 90%.

W literaturze anglojęzycznej i polskiej zauważalna jest niewielka liczba publikacji porównawczych pomiędzy architekturą modularnego monolitu a mikrousług. Wiele prac koncentruje swoją uwagę na klasycznej architekturze monolitycznej, która jednak znacząco różni się od architektury mikroserwisów. Modularny monolit w porównaniu do klasycznego monolitu oferuje elastyczność i modularność, które mogą okazać się kluczowe dla aktualnych potrzeb firm informatycznych i mogą przypominać pewne cechy mikrousług. Dlatego bardziej adekwatne jest porównywanie architektury mikroserwisowej z architekturą monolityczną, podzieloną na moduły, ponieważ obie z tych koncepcji oferują z jednej strony podobną a z drugiej strony inną organizację złożonych systemów.

3. Cel i zakres badań

Głównym celem niniejszego artykułu jest przeprowadzenie analizy porównawczej wybranych aspektów architektur aplikacji internetowych. W tym celu zostaną zestawione ze sobą dwie popularne architektury – architektura mikroserwisów oraz architektura monolityczna. Porównania dokonane zostaną poprzez analizę wydajności na podstawie porównania narzutu infrastrukturalnego, skalowalności horyzontalnej oraz komunikacji pomiędzy komponentami dwóch tożsamych systemów, przygotowanych na potrzeby badań. Jeden z nich został przygotowany w oparciu o architekturę mikroserwisów, a drugi w oparciu o architekturę modularnego monolitu. Wyniki badań dostarczą informacji, które mogą zostać wykorzystane do podejmowania decyzji dotyczących wyboru odpowiedniej architektury dla konkretnych zastosowań. Na potrzeby prowadzonej analizy porównawczej postawiono następujące hitpotezy:

H1: Aplikacje stworzone w architekturze mikroserwisów, przy założeniu pracy na pojedynczych instancjach bez skalowania, mają dłuższy czas odpowiedzi na

operacje CRUD, niż aplikacje stworzone w architekturze modularnego monolitu, czego powodem jest dodatkowy narzut infrastrukturalny taki jak bramka czy mechanizm odkrywania usług.

H2: Kolejki w pamięci, z których mogą korzystać aplikacje monolityczne szybciej przetwarzają dane i generują mniejsze opóźnienia niż zewnętrzne brokery wiadomości.

H3: W architekturze mikroserwisów zwiększenie dostępności całego systemu można osiągnąć poprzez skalowanie horyzontalne tylko tych usług, które są najbardziej obciążone, gdzie w architekturze modularnego monolitu wymagane jest równoczesne skalowanie całego systemu.

Zakres badań obejmuje:

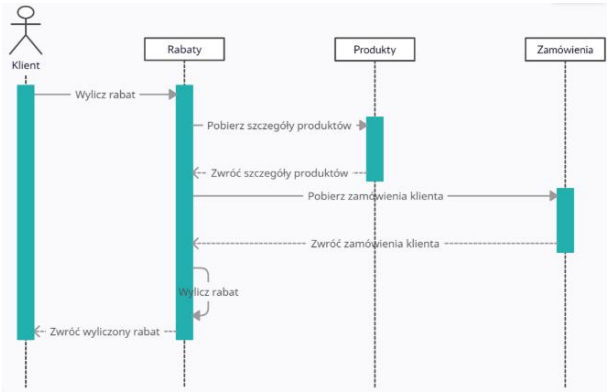
1. Implementację aplikacji na potrzeby badań.
2. Opracowanie scenariuszy badawczych.
3. Przeprowadzenie testów według przygotowanych scenariuszy.
4. Analizę wyników uzyskanych z testów i przedstawienie ich w formie ułatwiającej interpretację danych.

4. Plan badań

Badania zostały oparte na różnych scenariuszach badawczych, aby przetestować wybrane aspekty architektur aplikacji. W celu realizacji badań skupiono się na dwóch kluczowych funkcjonalnościach systemu, by móc następnie podczas wykonywania ich, zmierzyć wydajność aplikacji i porównać obie architektury.

4.1. Zadanie obliczania rabatu

Jedną z funkcjonalności dostępnych w aplikacji jest obliczanie rabatu dla produktów umieszczonych w koszyku użytkownika. Operacja ta angażuje wszystkie moduły systemu, które wymieniają między sobą informacje, co zapewnia spójność i prawidłowość obliczeń. Do wymiany informacji pomiędzy modułami wykorzystano kolejki komunikatów, co pozwala na asynchroniczne przetwarzanie operacji. Sekwencja, w jakiej realizowana jest ta funkcjonalność, została przedstawiona na Rysunku 1.



Rysunek 1: Diagram sekwencji operacji obliczania rabatu.

4.2. Zadanie CRUD (Create, Read, Update, Delete) produktów

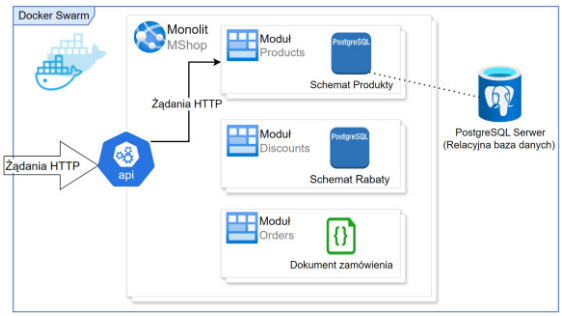
Kolejnym przygotowanym zadaniem są operacje CRUD (*Create*, *Read*, *Update*, *Delete*) wykonywane w module *Products*, które odpowiadają takim akcjom jak:

- utworzenie produktu;
- pobranie wszystkich produktów;
- zaktualizowanie wybranego produktu po id;
- usunięcie wybranego produktu po id.

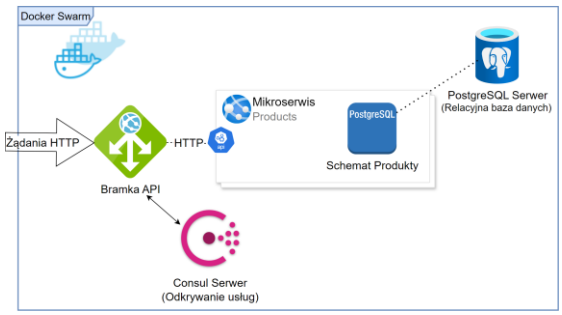
Całość zadania odbywa się przy wykorzystaniu protokołu http i nie jest angażowana komunikacja pomiędzy komponentami aplikacji.

4.3. Scenariusz testów wpływu infrastruktury na wydajność systemu

W pierwszym scenariuszu badawczym porównywany jest czas odpowiedzi aplikacji uruchomionej w dwóch różnych architekturach, na operacje CRUD (*Create*, *Read*, *Update*, *Delete*) opisane w Rozdziale 4.4. Aplikacja uruchomiona w architekturze modularnego monolitu może przyjąć żądanie http bezpośrednio na swój interfejs API, od razu przekierowując wspomniane żądanie na odpowiedni moduł co obrazuje Rysunek 2, natomiast aplikacja uruchomiona w architekturze mikroserwisowej, aby mogła odnaleźć moduł z odpowiednim interfejsem API i przekierować do niego żądanie http, posiada dodatkową infrastrukturę w postaci bramki API (ang. *api gateway*) oraz odkrywania serwisów (ang. *service discovery*) co przedstawia Rysunek 3. Kod wykonywany przez interfejs API aplikacji uruchamianej w obu architekturach jest identyczny i to właśnie dodatkowa infrastruktura w aplikacji uruchamianej w architekturze mikroserwisowej jest przedmiotem badania.



Rysunek 2: Schemat prezentujący uproszczoną architekturę modularnego monolitu na potrzeby badania wpływu infrastruktury na wydajność systemu.



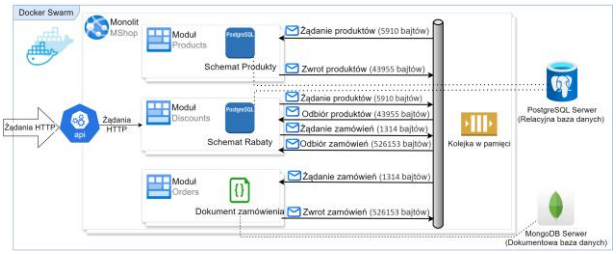
Rysunek 3: Schemat prezentujący uproszczoną architekturę mikroserwisów na potrzeby badania wpływu infrastruktury na wydajność systemu.

Badanie przeprowadzono wykorzystując narzędzie JMeter. Aplikacja JMeter został skonfigurowana tak, aby wysyłała żądania z pięciu wątków jednocześnie. Scenariusz testu wygląda następująco:

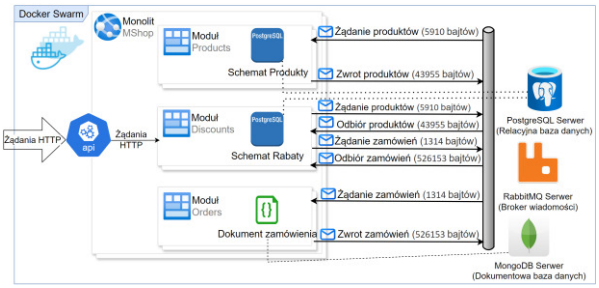
1. Wysłanie 1000, 10000, 50000, 10000 żądań http POST do aplikacji w architekturze mikroserwisowej.
2. Wysłanie 1000, 10000, 50000, 10000 żądań http GET do aplikacji w architekturze mikroserwisowej.
3. Wysłanie 1000, 10000, 50000, 10000 żądań http PUT do aplikacji w architekturze mikroserwisowej.
4. Wysłanie 1000, 10000, 50000, 10000 żądań http DELETE do aplikacji w architekturze mikroserwisowej.
5. Wysłanie 1000, 10000, 50000, 10000 żądań http POST do aplikacji w architekturze modularnego monolitu
6. Wysłanie 1000, 10000, 50000, 10000 żądań http GET do aplikacji w architekturze modularnego monolitu
7. Wysłanie 1000, 10000, 50000, 10000 żądań http PUT do aplikacji w architekturze modularnego monolitu
8. Wysłanie 1000, 10000, 50000, 10000 żądań http DELETE do aplikacji w architekturze modularnego monolitu

4.4. Scenariusz testów wpływu komunikacji między-komponentowej na wydajność systemu

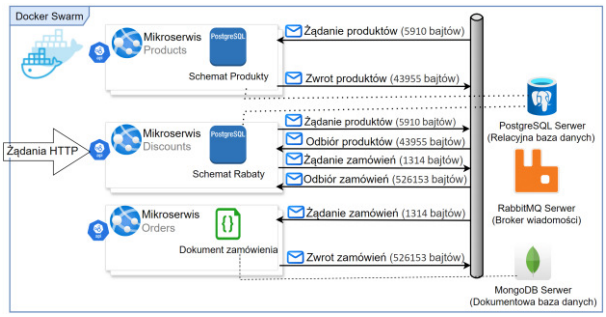
W kolejnym scenariuszu badawczym porównywany jest wpływ mechanizmów zarządzania komunikacją na czas odpowiedzi aplikacji. W architekturze modularnego monolitu przygotowane zostały dwie różne konfiguracje aplikacji, które różnią się zarządzaniem wiadomościami. Pierwsza konfiguracja aplikacji uruchomionej w architekturze modularnego monolitu to użycie kolejki w pamięci (Rysunek 4), natomiast druga konfiguracja, to wykorzystanie zewnętrznego brokera wiadomości – RabbitMQ (Rysunek 5). W aplikacji uruchomionej w architekturze mikroserwisowej użyty został również broker wiadomości – RabbitMQ (Rysunek 6). Aby wyniki nie zostały zakłócone poprzez opóźnienia spowodowane narzutem infrastruktury, w systemie składającym się z mikroserwisów została wyłączona bramka (ang. *API Gateway*) oraz mechanizm odkrywania serwisów (ang. *service discovery*), a ruch został skierowany bezpośrednio na mikroserwis *Discounts*.



Rysunek 4: Schemat architektury monolitycznej wykorzystującej kolejkę w pamięci do obsługi wiadomości.



Rysunek 5: Schemat architektury monolitycznej wykorzystującej RabbitMQ do obsługi wiadomości.



Rysunek 6: Schemat architektury mikroserwisów wykorzystującej RabbitMQ do obsługi wiadomości.

Badanie przeprowadzono wykorzystując narzędzie JMeter, wykonując operację obliczania rabatu opisaną w rozdziale 4.1 Scenariusz badawczy wygląda następująco:

- 10, 100, 200 i 400 użytkowników wysyła 250 żądań do aplikacji uruchomionej w architekturze mikroserwisowej.
- 10, 100, 200 i 400 użytkowników wysyła 250 żądań do aplikacji uruchomionej w architekturze modularnego monolitu przy wykorzystaniu kolejki RabbitMQ.
- 10, 100, 200 i 400 użytkowników wysyła 250 żądań do aplikacji uruchomionej w architekturze modularnego monolitu przy wykorzystaniu kolejki w pamięci.

4.5. Scenariusz testów wpływu selektywnego skalowania horyzontalnego mikroserwisów na wydajność systemu w porównaniu do skalowania całego monolitu

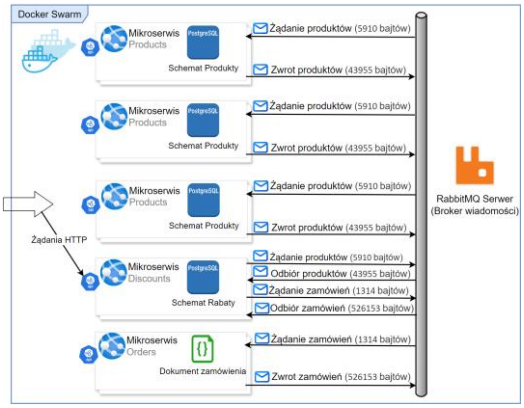
W ostatnim scenariuszu badawczym badany jest wpływ skalowania horyzontalnego wybranych komponentów systemu uruchomionego w architekturze mikroserwisów na wydajność całego systemu, w porównaniu do systemu uruchomionego w architekturze modularnego monolitu, gdzie wymagane jest skalowanie całej aplikacji.

Równoważenie obciążenia użytkowników odbywa się poprzez wbudowany wewnętrznie Ingress w Docker Swarm. Docker Swarm przypisuje dodatkowo każdej usłudze odpowiedni wpis w DNS. Dzięki powyższym funkcjonalnościom podczas odpytywania konkretnej usługi po jej adresie, Docker Swarm równoważy obciążenie przy wykorzystaniu algorytmu karuzelowego (ang. *round robin*) [10]. W RabbitMQ podczas skalowania serwisów, każda z instancji usługi subskrybuje się do kolejki nowym konsumentem, który następnie odbiera

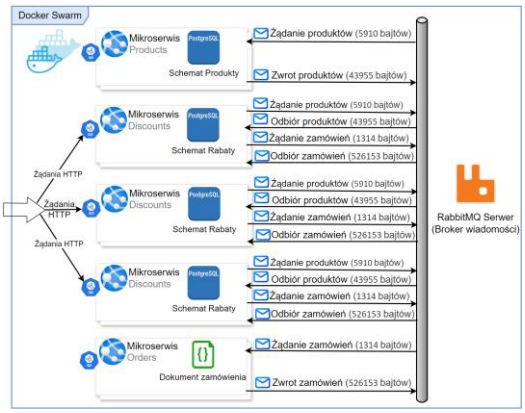
część wiadomości dostępnych w kolejce przy wykorzystaniu wcześniej wspomnianego algorytmu karuzelowego [11].

Aby wyniki nie zostały zakłócone poprzez opóźnienia spowodowane narzutem infrastruktury, w systemie składającym się z mikroserwisów została wyłączona bramka (ang. *API Gateway*) oraz mechanizm odkrywania serwisów (ang. *service discovery*). W badaniu zostały przeprowadzone testy obciążeń przy wykorzystaniu 400 wątków, z których każdy wysyła 250 żądań obliczenia rabatu opisanego w rozdziale 4.1. Testy obciążeń zostały przeprowadzone dla poniższych konfiguracji:

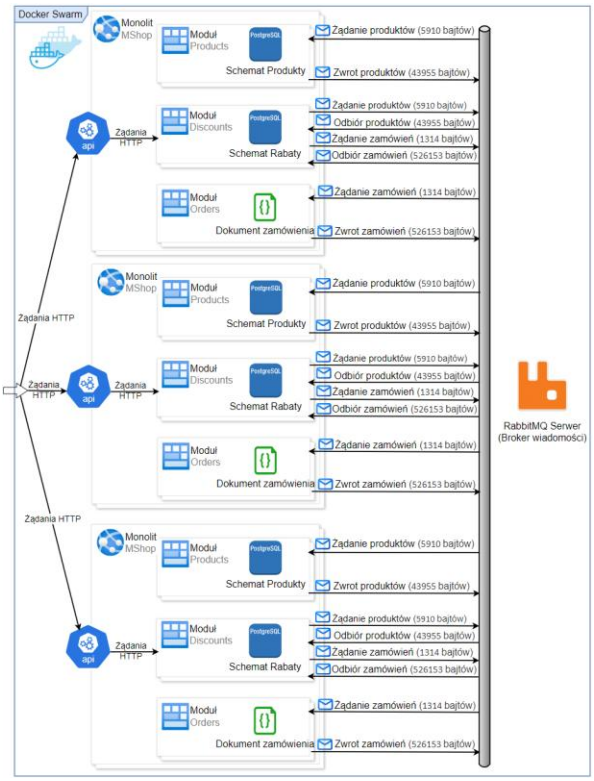
- brak skalowanego żadnego z mikroserwisów - Rysunek 6;
- brak skalowanej aplikacji uruchamianej w architekturze modularnego monolitu - Rysunek 5;
- trzy instancje mikroserwisu *Products*, reszta mikroserwisów pojedyncze instancje - Rysunek 7;
- trzy instancje mikroserwisu *Discounts*, reszta mikroserwisów pojedyncze instancje - Rysunek 8;
- trzy instancje modularnego monolitu - Rysunek 9;
- sześć instancji mikroserwisu *Products*, reszta mikroserwisów pojedyncze instancje;
- sześć instancji mikroserwisu *Discounts*, reszta mikroserwisów pojedyncze instancje;
- sześć instancji aplikacji uruchomionej w architekturze modularnego monolitu.



Rysunek 7: Schemat prezentujący architekturę mikroserwisów z trzykrotnie zeskalowanym mikroserwisem Products.



Rysunek 8: Schemat prezentujący architekturę mikroserwisów z trzykrotnie zeskalowanym mikroserwisem Discounts.



Rysunek 9: Schemat prezentujący architekturę modularnego monolitu z trzykrotnie zeskalowanym modularnym monolitem.

4.6. Środowisko testowe

W Tabelach 1, 2 i 3 zaprezentowano środowisko testowe, użyte do przeprowadzonych badań. Poza parametrami komputera, tabele zawierają informacje o zastosowanych narzędziach wraz z ich wersjami oraz technologiach użytych do napisania aplikacji testowej.

Tabela 1: Parametry sprzętowe środowiska testowego

Sprzęt	
Procesor	AMD Ryzen 9 7945HX with Radeon Graphics
Pamięć RAM	40 GB DDR5
System operacyjny	Windows 11
Dysk	SSD

Tabela 2: Aplikacje użyte do przeprowadzenia badań

Aplikacja	**Technologia**
Mikroserwis Orders	.NET 8
Mikroserwis Discounts	.NET 8
Mikroserwis Products	.NET 8
Modularny monolit	.NET 8

Tabela 3: Narzędzia wraz z wersjami wykorzystywanymi w badaniach

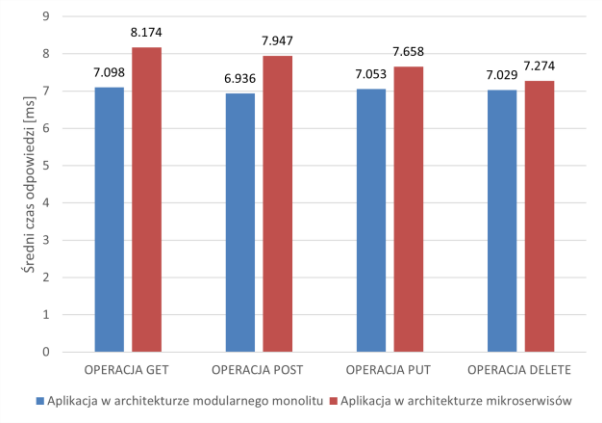
Narzędzia	**Wersja**
.NET	8.0.303
RabbitMQ	3.13.3
Apache JMeter	5.6.3
Docker	26.1.4
PostgreSQL	16.3
MongoDB	6.0.7
Consul	1.19.2

5. Analiza wyników badań

W poniższym rozdziale przedstawione zostały uzyskane wyniki dla poszczególnych badań.

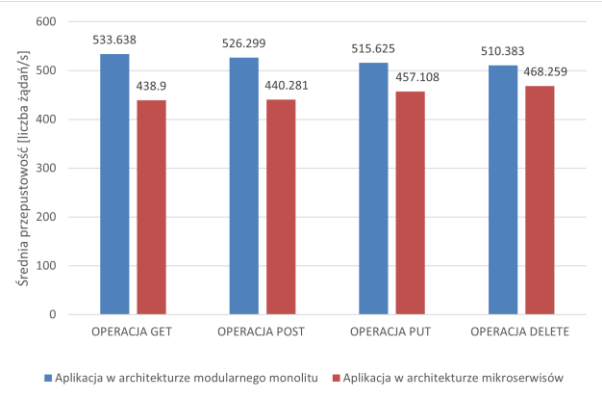
5.1. Wyniki badań wydajności

W tym podrozdziale przedstawione zostały średnie czasy odpowiedzi dla operacji http takich jak *GET*, *POST*, *PUT*, *DELETE*. Aby zminimalizować wpływ przypadkowych zmian systemowych oraz zapewnić większą dokładność wyników, symulacje zostały przeprowadzone wielokrotnie z różną liczbą powtórzeń. Średni czas odpowiedzi, na podstawie wszystkich prób został przedstawiony na Rysunku 10. To co można zauważyć na wspomnianym rysunku, to dłuższy średni czas odpowiedzi dla każdej operacji CRUD w przypadku architektury mikroserwisowej. Dla operacji *GET* średnia różnica wynosiła 1,076 ms, dla operacji *POST* różnica to 1,009ms, dla operacji *PUT* było to 0,605ms, natomiast dla operacji *DELETE* różnica wynosiła 0,245 ms.



Rysunek 10: Wykres średnich czasów odpowiedzi dla żądań http w architekturze monolitu i architekturze mikroserwisów.

Różnice w średnim czasie odpowiedzi mają wyraźny wpływ na wydajność aplikacji. Aby lepiej zilustrować spadek wydajności, na Rysunku 11 przedstawiono wykres średniej przepustowości aplikacji. Widać na nim, że aplikacja oparta na architekturze mikroserwisowej jest w stanie obsłużyć znacznie mniej wiadomości na sekundę w porównaniu do aplikacji monolitycznej.



Rysunek 11: Wykres średniej przepustowości aplikacji dla żądań http w architekturze monolitu i architekturze mikroserwisów.

5.2. Wyniki badań wpływu komunikacji międzykomponentowej na wydajność systemu

W Tabeli 4 przedstawiono wyniki testów wydajności dla systemu uruchomionego w architekturze modularnego monolitu z użytą kolejką w pamięci do komunikacji pomiędzy modułami. Można zauważyć, że wraz ze wzrostem liczby wątków wykonujących zapytania, systematycznie wzrastał też średni czas odpowiedzi. Przy najniższym obciążeniu tj. 10 wątków, średni czas wykonywania operacji był najniższy, co wskazuje na największą efektywność przy małym ruchu. W miarę zwiększania liczby wątków do 100, 200 i 400 obserwowano wyraźny wzrost średnich czasów odpowiedzi, z jednoczesnym wzrostem odchylenia standardowego.

Tabela 4: Wyniki wykonania testu dla różnych obciążeń aplikacji w architekturze modularnego monolitu z komunikacją wykorzystującą kolejkę w pamięci

Liczba wątków	Średni czas wykonywania operacji obliczania rabatu	Współczynnik zmienności	Przepustowość systemu
	[ms]	[%]	[liczba żądań/s]
10	31 ± 10	33	277
100	215 ± 62	28	436
200	438 ± 129	29	440
400	866 ± 265	31	449

W Tabeli 5 przedstawiono wyniki testów wydajności dla systemu uruchomionego w architekturze modularnego monolitu z użytym zewnętrznym brokerem wiadomości RabbitMQ do komunikacji pomiędzy modułami. Można zauważyć, że wraz ze wzrostem liczby wątków wykonujących zapytania, systematycznie wzrastał też średni czas odpowiedzi. Przy najniższym obciążeniu tj. 10 wątków, średni czas wykonywania operacji był najniższy, co wskazuje na największą efektywność przy małym ruchu. W miarę zwiększania liczby wątków do 100, 200 i 400 obserwowano wyraźny wzrost średnich czasów odpowiedzi, z jednoczesnym wzrostem odchylenia standardowego.

Tabela 5: Wyniki wykonania testu dla różnych obciążeń aplikacji w architekturze modularnego monolitu z komunikacją wykorzystującą zewnętrzny broker wiadomości – RabbitMQ

Liczba wątków	Średni czas wykonywania operacji obliczania rabatu	Współczynnik zmienności	Przepustowość systemu
	[ms]	[%]	[liczba żądań/s]
10	66 ± 17	26	132
100	661 ± 273	41	146
200	1333 ± 449	34	147
400	2680 ± 83	31	147

Wyniki testów wydajności dla systemu uruchomionego w architekturze mikroserwisów, wykorzystującego do komunikacji RabbitMQ, przedstawione w Tabeli 6 wykazują, że wraz ze wzrostem obciążenia wzrasta również średni czas odpowiedzi. Przepustowość dla 100, 200, 400 wątków jest bardzo zbliżona do siebie.

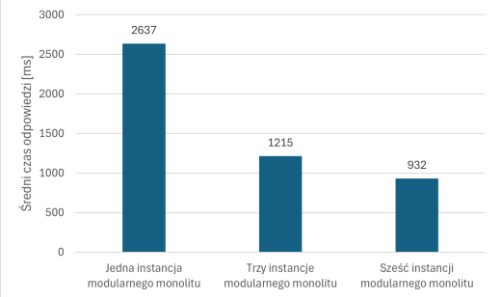
Tabela 6: Wyniki wykonania testu dla różnych obciążeń aplikacji w architekturze mikroserwisów z komunikacją wykorzystującą broker wiadomości – RabbitMQ

Liczba wątków	Średni czas wykonywania operacji obliczania rabatu	Współczynnik zmienności	Przepustowość systemu
	[ms]	[%]	[liczba żądań/s]
10	64 ± 15	23	140
100	662 ± 245	37	147
200	1343 ± 377	28	146
400	2678 ± 723	27	147

Na podstawie powyższych wyników można zauważyć, że mikroserwisy jak i modularny monolit z zewnętrznym brokerem RabbitMQ posiadają wydajność na podobnym poziomie. Dużo lepszą wydajność można zaobserwować natomiast w przypadku modularnego monolitu z kolejką w pamięci. Powyższy wynik potwierdza, że przyczyną spadku wydajności jest infrastruktura związana z zewnętrznym brokerem RabbitMQ, co sugeruje opóźnienia spowodowane warunkami sieciowymi i architekturą komunikacji, podczas gdy kolejka w pamięci pomija sieć i nie powoduje takich opóźnień.

5.3. Wyniki badań wpływu selektywnego skalowania horyzontalnego mikroserwisów na wydajność systemu w porównaniu do skalowania całego monolitu

Rysunek 12 przedstawia wpływ liczby uruchomionych instancji modularnego monolitu na średni czas odpowiedzi systemu. Wyniki przedstawione na wykresie pokazują wyraźną zależność pomiędzy liczbą instancji, a wydajnością systemu. System działający z jedną instancją osiągał średni czas odpowiedzi na poziomie 2637 ms, co stanowiło najwyższą wartość spośród testowanych konfiguracji. Uruchomienie trzech instancji znacząco poprawiło wydajność, redukując średni czas odpowiedzi do 1215 ms, co było znaczącą poprawą względem jednej instancji. Najlepszy wynik uzyskano przy sześciu instancjach, gdzie czas odpowiedzi został zredukowany do 902 ms. Wnioski z tych analiz sugerują, że zwiększenie liczby instancji modularnego monolitu prowadzi do znaczącego obniżenia średniego czasu odpowiedzi. Zwiększenie liczby instancji z jednej do sześciu skutkowało poprawą wydajności, co wskazuje, że systemy oparte na tej architekturze mogą skutecznie skalować się poprzez dodawanie nowych instancji, co korzystnie wpływa na ich ogólną wydajność.



Rysunek 12: Średni czas odpowiedzi na żądanie w zależności od liczby instancji modularnego monolitu.

Wyniki przedstawione na Rysunku 13 obrazują średni czas odpowiedzi w systemie mikroserwisów w zależności od liczby instancji uruchomionych dla różnych komponentów, przy czym mikroserwis *Products* jest mniej obciążony, a *Discounts* bardziej obciążony. Dla konfiguracji z jedną instancją każdego z serwisów (*Orders*, *Discounts*, *Products*), średni czas odpowiedzi wynosił 2679 ms. Przy wzroście liczby instancji serwisu *Products* do trzech, średni czas odpowiedzi wzrósł do 2704 ms, co pokazuje, że dodatkowe instancje mniej obciążonego serwisu nie przyniosły poprawy wydajności. Podobny wynik, 2704 ms, zaobserwowano przy sześciu instancjach *Products*, co potwierdza ten trend.

Z kolei w konfiguracji, gdzie zwiększono liczbę instancji serwisu *Discounts* (bardziej obciążonego mikroserwisu) do trzech, średni czas odpowiedzi uległ znaczącej poprawie, spadając do 1233 ms. Najlepszy wynik uzyskano, gdy liczba instancji serwisu *Discounts* została zwiększona do sześciu, co obniżyło średni czas odpowiedzi do 1023 ms.

Największą poprawę wydajności uzyskuje się poprzez skalowanie bardziej obciążonych mikroserwisów, takich jak *Discounts*. Skalowanie mniej obciążonych mikroserwisów, jak *Products*, nie przynosi widocznych korzyści w zakresie redukcji czasu odpowiedzi.

Rysunek 13: Średni czas odpowiedzi na żądanie dla różnych konfiguracji instancji mikroserwisów.

Jak widać w wynikach przeprowadzonego badania skalowanie modularnego monolitu przyniosło korzyści w zakresie redukcji średniego czasu odpowiedzi na żądania obliczenia rabatu. Jednak pomimo zauważalnych efektów należy podkreślić, że jakiekolwiek skalowanie horyzontalne monolitu wymaga zwiększenia liczby instancji całej aplikacji, bez względu na to czy faktycznie wszystkie komponenty są obciążone czy nie. Pomimo tego, że aplikacja jest podzielona na moduły, to nadal jest to jedna i ta sama struktura wdrożeniowa co właśnie powoduje, że w sytuacji gdy jeden z modułów nie jest w stanie obsłużyć skierowanego do niego ruchu i staje się wąskim gardłem całego systemu, skalowanie obciążonego elementu pociąga za sobą skalowanie innych elementów. To prowadzi do nieefektywnego gospodarowania mocą obliczeniową i innymi zasobami, które są alokowane tam, gdzie nie są potrzebne.

W przypadku mikroserwisów istnieje możliwość skalowania tylko najbardziej obciążonej usługi i właśnie po skalowaniu mikroserwisu *Discounts* nastąpiła widoczna redukcja średniego czasu odpowiedzi na żądania. Poprawa ta została osiągnięta bez potrzeby skalowania całego systemu jak jest to konieczne w przypadku aplikacji

monolitycznej, a jedynie poprzez skalowanie komponentu stanowiącego wąskie gardło. Zasoby zostały skierowane tam gdzie były najbardziej potrzebne i zostały dobrze spożytkowane.

6. Wnioski

Celem artykułu było przeprowadzenie analizy porównawczej wybranych aspektów architektur aplikacji internetowych.

W pierwszym badaniu skupiono uwagę na porównaniu wpływu narzutu infrastrukturalnego w architekturze mikroserwisów w porównaniu do architektury modularnego monolitu na wydajność systemu. Wyniki badania jednoznacznie wykazały, że w każdej z badanych operacji CRUD (*GET*, *POST*, *PUT*, *DELETE*) aplikacja w architekturze mikroserwisowej uzyskała dłuższe czasy odpowiedzi niż aplikacja w architekturze modularnego monolitu. Największe różnice średniego czasu wystąpiły dla operacji *GET* i *POST*, wynoszące odpowiednio 1,076 ms i 1,011 ms, natomiast dla operacji *PUT* i *DELETE* były mniejsze i wyniosły odpowiednio 0,605 ms i 0,245 ms. Wyniki te jednoznacznie pokazują, że dodatkowy narzut infrastrukturalny obecny w architekturze mikroserwisów powoduje wydłużenie czasu odpowiedzi na operacje, co potwierdza postawioną hipotezę H1.

W drugim badaniu skupiono uwagę na porównaniu wydajności komunikacji miedzy-komponentowej w obu podejściach architektonicznych o różnych konfiguracjach. Najbardziej wydajnym rozwiązaniem okazała się aplikacja monolityczna w której zastosowano kolejkę w pamięci. To rozwiązanie umożliwiło eliminację potrzeby komunikacji sieciowej oraz brak potrzeby korzystania z zewnętrznego brokera wiadomości, takiego jak RabbitMQ. Jako, że spadek wydajności został zaobserwowany zarówno w przypadku aplikacji monolitycznej korzystającej z RabbitMQ, jak i w aplikacji mikroserwisowej, a obie aplikacje wykonują identyczny kod logiki biznesowej, a spadki wydajności w obu przypadkach były bardzo zbliżone, można stwierdzić, że problem nie wynika bezpośrednio z samej architektury ale z użycia zewnętrznego brokera wiadomości. Uzyskane wyniki pokazują, że kolejki w pamięci z których mogą korzystać aplikacje monolityczne szybciej przetwarzają dane co potwierdza postawioną hipotezę H2.

W trzecim przeprowadzonym badaniu skupiono uwagę na horyzontalnym skalowaniu aplikacji. Skalowanie aplikacji w architekturze modularnego monolitu, poprzez zwiększenie liczby jego instancji, przyniosło pozytywne efekty poprawiając wydajność systemu. Jednak dla osiągnięcia takiego rezultatu konieczne było skalowanie każdego z modułów wchodzących w skład systemu, bez możliwości wyboru konkretnych komponentów. W przypadku aplikacji w architekturze mikroserwisów możliwe było zwiększenie liczby instancji tylko wybranych usług. Wykonane skalowanie mniej obciążonego mikroserwisu *Products*, który przetwarzał niewielką ilość danych, nie przyniosło istotnych zmian w wydajności systemu. Udowodniło, to, że skalowanie niektórych elementów może być zbędne i powodować nieefektywne zarządzanie zasobami. Natomiast skalowanie mikroserwisu *Discounts*, który był znacznie bardziej obciążony, przyniosło wyraźną poprawę wydajności, porównywalną z efektami uzyskanymi przy skalowaniu całej aplikacji monolitycznej. To potwierdziło, że w przypadku architektury mikroserwisów poprawa wydajności jest możliwa poprzez precyzyjne skalowanie wybranych części aplikacji, bez konieczności skalowania wszystkich komponentów, gdzie w architekturze monolitycznej aby poprawić wydajność konieczne jest równoczesne skalowanie całej aplikacji. Uzyskane wyniki potwierdzają, więc postawioną hipotezę H3.

Literatura

[1] A. M. Abd-Elwahab, A. G. Mohamed, E. M. Shaaban, MicroServices-driven enterprise architecture model for infrastructure optimization, Future Business Journal 9 (2023) 90-104, https://doi.org/10.1186/s43093-023-00268-3.

[2] M. Richards, N. Ford, Fundamentals of Software Architecture: An Engineering Approach, O'Reilly Media, Sebastopol, 2020.

[3] S. Whitesell, R. Richardson, M. D. Groves, Pro Microservices in .NET 6, Apress, New York, 2022.

[4] S. Fowler, Production-Ready Microservices, O'Reilly Media, Sebastopol, 2017.

[5] V. Vernon, T. Jaskula, Strategic Monoliths and Microservices: Driving Innovation Using Purposeful Architecture, Addison-Wesley Publishing, Boston, 2022.

[6] Ch. Richardson, Mikroserwisy, Wydawnictwo Naukowe PWN, Warszawa, 2020.

[7] K. Jaskot, S. Przyłucki, Analysis of selected features of application based on monolithic and microservice architecture, Journal of Computer Science Institute 25 (2022) 393-400, https://doi.org/10.35784/jcsi.3061.

[8] S. Newman, Monolith To Microservices, O'Reilly Media, Sebastopol, 2020.

[9] Prime Video Tech, https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90, [16.06.2024].

[10] Docker Documentation, https://docs.docker.com/, [15.10.2024].

[11] RabbitMQ Documentation, https://www.rabbitmq.com, [06.08.2024].