Praca Dyplomowa Magisterska

Patrycja Kozak
200831

Analiza porównawcza architektury monolitycznej oraz mikroserwisów na przykładzie aplikacji webowej

Comparative Analysis of Monolithic Architecture and Microservices Architecture Based on a Web Application Example

Praca dyplomowa na kierunku:
Informatyka

Praca wykonana pod kierunkiem
dr inż. Artur Krupa
Instytut Informatyki Technicznej
Katedra Sztucznej Inteligencji

Warszawa, 2024

SZKOŁA GŁÓWNA GOSPODARSTWA WIEJSKIEGO

Wydział Zastosowań Informatyki i Matematyki

Oświadczenie Promotora pracy

Oświadczam, że niniejsza praca została przygotowana pod moim kierunkiem i stwierdzam, że spełnia warunki do przedstawienia tej pracy w postępowaniu o nadanie tytułu zawodowego.

Data ..... Podpis promotora .....

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej, w tym odpowiedzialności karnej za złożenie fałszywego oświadczenia, oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami prawa, w szczególności ustawą z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz.U. 2019 poz. 1231 z późn. zm.).

Oświadczam, że przedstawiona praca nie była wcześniej podstawą żadnej procedury związanej z nadaniem dyplomu lub uzyskaniem tytułu zawodowego.

Oświadczam, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną. Przyjmuję do wiadomości, że praca dyplomowa poddana zostanie procedurze antyplagiatowej.

Data ..... Podpis autora pracy .....

Streszczenie

Analiza porównawcza architektury monolitycznej oraz mikroserwisów na przykładzie aplikacji webowej

Praca dyplomowa poświęcona jest analizie i porównaniu dwóch popularnych architektur systemowych – monolitycznej oraz mikroserwisowej – pod kątem ich implementacji, wydajności oraz kosztów wdrożenia. W pierwszej części pracy przedstawiono przegląd literatury, charakteryzując obie architektury, ich cechy, zalety i wady, a także dokonano ich porównania.

Następnie opisano wykorzystane rozwiązania technologiczne, w tym języki programowania, frameworki, bazy danych, narzędzia do konteneryzacji . W części praktycznej stworzono dwa systemy: pierwszy w architekturze monolitycznej, drugi w mikroserwisowej. Szczegółowo omówiono proces projektowania, implementacji, komunikacji między komponentami oraz konfiguracji systemów.

W ramach analizy porównano metryki wydajnościowe obu rozwiązań oraz oszacowano koszty wdrożenia aplikacji w środowisku chmurowy. Ostatni rozdział zawiera wnioski końcowej oraz rekomendacje.

Słowa kluczowe – monolit, mikroserwisy, analiza porównawcza, wydajność, architektura monolityczna, architektura mikroserwisowa, konteneryzacja

Summary

Comparative Analysis of Monolithic Architecture and Microservices Architecture Based on a Web Application Example

This thesis is dedicated to the analysis and comparison of two popular system architectures – monolithic and microservices – in terms of their implementation, performance, and deployment costs. In the first part of the thesis, a literature review is presented, characterizing both architectures, their features, advantages, and disadvantages, as well as providing a comparison between them. Next, the utilized technological solutions are described, including programming languages, frameworks, databases, and containerization tools.

In the practical part, two systems were developed: the first using the monolithic architecture, and the second using the microservices architecture. The process of designing, implementing, component communication, and configuring the systems is discussed in detail.

As part of the analysis, the performance metrics of both solutions were compared, and the costs of deploying the applications in the Microsoft Azure cloud environment were estimated. The final chapter includes conclusions and recommendations.

Keywords – monolith, microservices, comparative analysis, performance metrics, monolithic architecture, microservices architecture, containerization

Spis treści

Wstęp

W ciągu ostatnich lat na znaczeniu zyskały głównie dwie koncepcje architektoniczne: architektura monolityczna oraz mikroserwisowa. Wybór odpowiedniego podejścia może mieć kluczowe znaczenie dla sukcesu aplikacji, wpływając na elastyczność, skalowalność oraz łatwość utrzymania systemu [1].

Architektura monolityczna, będąca tradycyjnym podejściem do projektowania aplikacji, zakłada, że wszystkie komponenty systemu są zintegrowane w jedną całość [2]. Architektura mikroserwisowa proponuje podział systemu na niezależne, współpracujące ze sobą serwisy. Każdy mikroserwis realizuje określoną funkcjonalność i może być rozwijany, wdrażany oraz skalowany niezależnie od innych [3].

Wiele znanych firm rozpoczynało tworzenie swoich systemów od architektury monolitycznej, co w początkowych etapach rozwoju było rozwiązaniem prostym, logicznym oraz dającym zadowalający efekt. Przykładami takich firm są Netflix, Amazon, GitHub, Atlassian czy Basecamp [1]. Firma Netflix była jednym z pierwszych przedsiębiorstw, które zdało sobie sprawę, że architektura monolityczna może być niewystarczająca, i może nie być najlepszym rozwiązaniem w przypadku złożonych aplikacji. Architekci Netflixa, przez pojedynczy błąd z 2008 roku, który spowodował masową utratę danych i doprowadził do kilkudniowego przestoju, zdecydowali się przenieść swoją aplikację z architektury monolitycznej do mikroserwisów opartych na chmurze AWS[2]. Uznano, że migracja architektury, usprawni dostępność oraz skalowalność systemu. Podzielenie systemu na 700 mikroserwisów, z których każdy odpowiedzialny był za jedną funkcjonalność, umożliwiało dokonywanie zmian w dowolnej części systemu, mając pewność, że aplikacja nie zostanie wyłączona. Pomogło to w zminimalizowaniu braków dostępności do serwisu.

Wraz z dynamicznym rozwojem technologii oraz rosnącymi wymaganiami i potrzebami współczesnych aplikacji, pojawia się konieczność przeanalizowania, a także porównania architektury mikroserwisów oraz monolitycznej. Tradycyjne już

[1] „https://medium.com/@sadhnajuhirdeen/what-are-the-big-companies-still-uses-monolithic-architecture-2555a93345f1," [Online]. [Data uzyskania dostępu: 1 05 2024].
[2] „https://www.techtarget.com/searchaws/definition/Amazon-Web-Services," [Online]. [Data uzyskania dostępu: 05 2024]

podejście monolityczne, mimo swojej prostoty, często może nie sprostać wyzwaniom jakie może mieć aplikacja. Może ona okazać się niewystarczająca pod względem skalowalności, elastyczności czy szybkości wdrażania nowych, potrzebnych funkcji. Użycie architektury mikroserwisów, która pozwala na dekompozycję aplikacji na niezależnie rozwijalne komponenty, wymaga dokładnego zrozumienia, kiedy i dlaczego warto użyć właśnie jej, jako rozwiązania. Analiza ta może być szczególnie istotna dla podmiotów stojących przed decyzją o modernizacji swojej architektury. Porównanie tych dwóch podejść pozwoli również zidentyfikować najlepsze praktyki oraz potencjalne pułapki, co jest kluczowe dla skutecznego wdrożenia nowoczesnych rozwiązań technologicznych.

Celem niniejszej pracy magisterskiej jest przeprowadzenie analizy porównawczej architektury monolitycznej oraz mikroserwisowej w kontekście ich odpowiedniości w różnych scenariuszach wdrożenia aplikacji. Proces badawczy będzie obejmował aspekty techniczne oraz funkcjonalne obu podejść, wraz z uwzględnieniem ich wpływu na skalowalność, wydajność, elastyczność, łatwość zarządzania oraz utrzymania systemu. Dodatkowymi aspektami porównawczymi będzie utylizacja zasobów. Praca ta będzie również skupiać się na identyfikacji sytuacji, w których każda z analizowanych architektur może być preferowana. Dodatkowo praca ta ma na celu dostarczenie praktycznych wskazówek dla specjalistów zajmującymi się architekturą oprogramowania w celu wyboru tej odpowiedniej w zależności od kontekstu projektu oraz jego wymagań.

Praca składa się z pięciu głównych rozdziałów, które szczegółowo analizują implementację systemu w dwóch architekturach: monolitycznej oraz mikroserwisowej. Rozdział 1 to przegląd literatury, w którym scharakteryzowano obie architektury oraz dokonano ich porównania. Rozdział 2 przedstawia wykorzystane rozwiązania technologiczne, takie jak języki programowania, frameworki, bazy danych, systemy messaging, narzędzia do konteneryzacji i API Gateway. Rozdział 3 skupia się na implementacji systemu w obu architekturach, omawiając szczegóły projektowe i wnioski wynikające z ich realizacji. W Rozdziale 4 przeprowadzono analizę obu podejść, porównując je pod względem wydajności oraz kosztów wdrożenia w chmurze Azure. Całość podsumowano w Rozdziale 5, gdzie przedstawiono wnioski końcowe dotyczące obu architektur.

1 Przegląd literatury

Rosnące tempo rozwoju informatyki sprawia, że właściwy dobór architektury systemu staje się krytycznym elementem determinującym sukces projektu. W ramach niniejszego przeglądu literatury, skoncentrowano się na dwóch podejściach architektonicznych: tradycyjnej architekturze monolitycznej oraz nowszej architekturze mikroserwisowej. Analiza obejmuje charakterystykę obu rozwiązań, uwzględniając ich zalety, ograniczenia oraz typowe przypadki zastosowań. Dodatkowo, przedstawione zostanie porównanie obu architektur, co pozwoli na lepsze zrozumienie kontekstów, w których dane podejście może okazać się bardziej korzystne.

1.1 Charakterystyka architektury monolitycznej

Architektura monolityczna jest tradycyjnym podejściem do projektowania i budowania systemów informatycznych, w którym cała aplikacja funkcjonuje jako jednolity i nierozerwalny blok kodu [4]. W tego rodzaju architekturze wszystkie moduły systemu, takie jak interfejs użytkownika, logika biznesowa i warstwa danych, są ze sobą ściśle powiązane i uruchamiane wspólnie w ramach jednego procesu. Chris Richardson w swojej książce [5] wskazuje, że ta cecha wynika z fundamentalnego założenia monolitu - wszystkie funkcjonalności są kompilowane do jednego wykonywalnego pliku lub pakietu. Zintegrowana struktura monolitu sprawia, że aplikacja jest budowana, testowana i wdrażana jako jedna całość, co przekłada się na prostotę w początkowej fazie projektu.

Jedną z kluczowych zalet architektury monolitycznej jest jej prostota koncepcyjna i implementacyjna. Vaughn Vernon podkreśla, że cała aplikacja funkcjonująca w ramach jednego środowiska znacząco upraszcza procesy projektowania i implementacji, szczególnie dla małych zespołów programistycznych. Programiści mogą skupić się na rozwoju funkcjonalności bez konieczności zarządzania złożoną infrastrukturą czy komunikacją między komponentami [6].

Architektura monolityczna ma jednak swoje ograniczenia, które stają się widoczne wraz ze wzrostem złożoności aplikacji. Jej skalowanie jest trudne i kosztowne, ponieważ wymaga skalowania całej aplikacji, nawet jeśli tylko jedna jej część wymaga większej wydajności. Utrzymanie staje się coraz bardziej problematyczne, ponieważ zależności między modułami sprawiają, że każda zmiana może wywołać nieprzewidziane problemy

w innych częściach systemu [7]. Dodatkowo, awaria jednego modułu może unieruchomić całą aplikację, co obniża niezawodność rozwiązania. Architektura monolityczna najlepiej sprawdza się w mniejszych projektach, które nie wymagają dużej skalowalności, takich jak wewnętrzne systemy ERP [8] czy proste aplikacje e-commerce.

1.2 Charakterystyka architektury mikroserwisu

Architektura mikroserwisowa to nowoczesne podejście do projektowania aplikacji, które zakłada podział systemu na wiele niezależnych usług, z których każda odpowiada za realizację jednej konkretnej funkcjonalności. Mikroserwisy działają jako odrębne jednostki, które mogą być rozwijane, wdrażane i skalowane niezależnie od siebie. Komunikacja między nimi odbywa się za pomocą lekkich protokołów, takich jak HTTP, gRPC czy komunikaty w kolejkach wiadomości [9].

Główną zaletą architektury mikroserwisowej jest jej elastyczność i skalowalność. Dzięki niezależności poszczególnych usług można je skalować w zależności od obciążenia, co pozwala na efektywniejsze wykorzystanie zasobów. Dodatkowo, możliwość niezależnego wdrażania poszczególnych mikroserwisów przyspiesza wprowadzanie zmian i ułatwia dodawanie nowych funkcjonalności bez ryzyka destabilizacji całego systemu. Awaria jednego mikroserwisu zazwyczaj nie wpływa na działanie pozostałych, co zwiększa niezawodność aplikacji jako całości.

Architektura mikroserwisowa wymaga jednak większych nakładów na zarządzanie i utrzymanie. Rozproszona natura systemu sprawia, że niezbędne są zaawansowane narzędzia do monitorowania, orkiestracji (np. Kubernetes) i zarządzania komunikacją między usługami. Wdrożenie mikroserwisów wymaga także bardziej złożonej infrastruktury, co wiąże się z wyższymi kosztami początkowymi. Kolejnym wyzwaniem jest testowanie aplikacji, ponieważ integracja niezależnych usług może generować nieoczekiwane problemy [10].

Architektura mikroserwisowa znajduje zastosowanie głównie w dużych, złożonych systemach, które wymagają skalowalności, niezawodności i elastyczności. Jest to popularne rozwiązanie w platformach streamingowych, takich jak Netflix, czy dużych serwisach e-commerce, takich jak Amazon, które muszą obsługiwać miliony

użytkowników w sposób płynny i niezawodny[3]. Pozwala ono na niezależne skalowanie poszczególnych usług w odpowiedzi na zmieniające się obciążenie, co jest kluczowe dla utrzymania płynności działania przy takiej ilości użytkowników. Dodatkowo, ułatwia to wdrażanie nowych funkcji oraz aktualizacji, co pozwala tym firmom utrzymywać przewagę konkurencyjną na rynku.

Wybór architektury mikroserwisowej pozwala na większą adaptacyjność do dynamicznych wymagań, jednak wiąże się z wyższymi kosztami i większą złożonością zarządzania.

1.3 Porównanie architektury monolitycznej oraz mikroserwisów

W artykule[4] autor zgłębia się w długo toczącą się debatę pomiędzy architekturami monolitów oraz mikroserwisów. Omówione zostają powszechnie błędne przekonania na temat architektury mikroserwisów, twierdząc że nie są one cudownym środkiem ale raczej narzędziem do zarządzania złożonością nowoczesnych aplikacji. Autor podkreśla, że wszystkie aplikacje nieuchronnie stają się złożone, a mikroserwisy oferują sposób na złagodzenie zaistniałej złożoności poprzez możliwość podzielenia jej na mniejsze, oraz łatwiejsze do zarządzania fragmenty. Jednym z kluczowych argumentów autor uznał to, że rozmiar mikroserwisu ma mniejsze znaczenie niż jego skuteczność w zaspokajaniu konkretnych potrzeb aplikacji.

Zamiast skupiać wyłącznie na tworzeniu jak najmniejszych usług, sugeruje on tworzenie serwisów które zapewniają równowagę pomiędzy złożonością a łatwością zarządzania. Ponadto w artykule podważono koncepcję dychotomii[5] pomiędzy monolitami a mikroserwisami, przez co zaproponowany został model hybrydowy, łączący elementy obu architektur. Autor sugeruje, że taki model odzwierciedla rzeczywistość wielu aplikacji, w których współistnieje mieszanka komponentów opartych na podejściu monolitycznym oraz mikroserwisowym. Omówiona została także ewolucja mikroserwisów na przestrzeni czasu, gdzie wywnioskowane zostało, że postęp w narzędziach i platformach sprawił, że ten rodzaj architektury stał się dużo bardziej dostępny i przyjazny.

[3] „https://keen.io/blog/architecture-of-giants-data-stacks-at-facebook-netflix-airbnb-and-pinterest/," [Online]. [Data uzyskania dostępu: 05 12 2024].
[4] https://www.infoq.com/articles/monolith-versus-microservices/
[5] Dychotomia - podział jakiejś całości na dwie różniące się zasadniczo części; też: ta różnica

W ramach pracy [11] autorzy porównali wydajność samodzielnie opracowanego systemu opartego na mikroserwisach z jego monolitycznym odpowiednikiem, zbadali także wpływ konteneryzacji na wydajność systemu. Korzystając z JMeter[6] do symulacji użytkowników, stwierdzono, że system monolityczny miał krótszy czas reakcji oraz większą przepustowość, zużywając mniej pamięci RAM oraz procesora w porównaniu do systemu mikroserwisów. Konteneryzacja systemu, przy użyciu Dockera [12] nie zmniejszyła wydajności, wręcz wykazała poprawę zużycia pamięci RAM, procesora, przepustowości oraz zauważono mniejsze opóźnienia. Wykorzystanie Dockera na systemie opartym o mikroserwisy skutkowało podobnymi opóźnieniami, ale zwiększyło przepustowość nawet o 15% przy dużym obciążeniu systemu, jednakże kosztem użycia większej ilości pamięci RAM i procesora.

Kolejny artykuł [13] ocenia takie wskaźniki jak czas reakcji oraz przepustowość zarówno przy wysokim jak i niskim obciążeniu. Dodatkowo autorzy porównali skuteczność różnych technologii wykrywania usług (ang. Service Discovery echnologies)[7] Eureka oraz Consul, dla architektur mikroserwisowych. Ich ustalenia wskazują, że przy typowych obciążeniach obie te architektury wykazują podobny poziom wydajności. Jednakże różnice stają się widoczne wraz ze wzrostem obciążenia, gdzie aplikacje monolityczne wykazują tendencję do wykazywania większej wydajności. W artykule tym podkreślono, że użycie architektury monolitycznej jest zalecane w przypadku, gdy celem programisty jest to by aplikacja obsługiwała żądania szybciej. Warto zauważyć, że pod względem przepustowości oraz liczby obsługiwanych żądań na sekundę, mikroserwisy wykorzystujące Consul do wykrywania usług, były o 4% lepsze niż te używające Eureki.

Autorzy [14] porównując architketury monolitów oraz mikroserwisów w swojej pracy zauważają, że architektura monolityczna zazwyczaj oferuje lepszą wydajność na pojedynczej maszynie niż mikroserwisy. Wynika to przede wszystkim z braku potrzeby komunikacji między komponentami poprzez sieć, co w przypadku mikroserwisów generuje dodatkowe opóźnienia oraz narzut związany z serializacją, deserializacją i przesyłaniem danych.

[6] „https://jmeter.apache.org/
[7] https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/ [Data uzyskania dostępu: 05 2024].

W pracy [15] wykonano porównanie obu architektur przy użyciu narzędzia do testowania obciążenia Gatling[8]. Badania te zostały wykonane na maszynie z systemem operacyjnym Ubuntu 18.04.2 LTS, a aplikacje wdrożone zostały przy pomocy Dockera. Praca ta porównuje te architektury pod względem takim parametrów jak: wydajność i czas odpowiedzi. Testy obciążeniowe wykazały ze mikroserwisy są bardziej wydajne, gdy aplikacja musi przetworzyć więcej requestów. Natomiast monolity mogą być dobrym rozwiązaniem, w przypadku aplikacji o mniejszym obciążeniu, przy czym dodatkowo są prostsze do stworzenia. Autorzy wspominają, że wybór architektury powinien być zdefiniowany przez wymagania biznesowe, tak aby spełniało ono wymagania inwestorów.

[8] „https://docs.gatling.io/," [Online]. [Data uzyskania dostępu: 05 2024].

2 Wykorzystane rozwiązania

Wybór opisanych poniżej technologii został podyktowany zarówno zainteresowaniami autora, jak i jego doświadczeniem w pracy z nimi. W celu zapewnienia wysokiej jakości kodu, opierano się na wskazówkach i rekomendacjach zawartych w książkach [16] [17].

2.1 Języki programowania

.Net oraz C#

Przed wyborem C# rozważano inne popularne języki programowania, takie jak Java, Python czy PHP które również cieszą się dużą popularnością w środowisku deweloperskim. Java, jako obiektowy i wieloplatformowy język programowania [18], jest szeroko stosowana w aplikacjach korporacyjnych oraz systemach mobilnych, zwłaszcza na Androida, dzięki bogatej bibliotece narzędzi i ogromnej społeczności. Jednak z perspektywy integracji z .NET oraz nowoczesnych funkcji Java w niektórych aspektach ustępuje C#. Python natomiast zdobył uznanie dzięki swojej prostocie, wszechstronności i zastosowaniu w analizie danych czy uczeniu maszynowym [19], ale jako język interpretowany może być mniej wydajny niż C#. PHP, będący językiem szeroko stosowanym w backendzie aplikacji webowych, oferuje prostotę i szybki czas wdrożenia [20], jednak jego starsze wersje cechują się mniejszą nowoczesnością i niższą wydajnością w porównaniu z C#. Wybór .Net oraz C# wydawał się jako najlepszy kompromis pomiędzy wydajnością, elastycznością i nowoczesnością. Dodatkowo, autor posiada największe doświadczenie w pracy z językiem C#.

Platforma .NET to darmowe, otwarte środowisko programistyczne, które umożliwia tworzenie i uruchamianie aplikacji na różnych, popularnych platformach. momencie rozpoczęcia pracy nad aplikacją najnowszą dostępną wersją .NET była wersja 8 [21]. Język C# jest nowoczesnym, obiektowym językiem programowania, w którym kod jest kompilowany do CIL (ang. Common Intermediate Language).

Typescript

Typescript to silnie typowany język programowania, który rozszerza możliwości JavaScript poprzez wprowadzenie statycznych typów. Kompiluje się do standardowego JavaScript, dzięki czemu może być używany we wszystkich nowoczesnych przeglądarkach oraz środowiskach uruchomieniowych [22]. Był on zastosowany to utworzenia UI aplikacji.

2.2 Frameworki oraz biblioteki

MassTransit

MassTransit[9] jest oprogramowaniem typu open-source dla platformy .NET, służącym do implementacji wzorców komunikacji [23] między usługami. Framework ten umożliwia asynchroniczną komunikację oraz integrację między systemami przy użyciu wiadomości. Rozwiązanie to jest stosowanie w tworzeniu rozwiązań opartych na mikrousługach [24].

Vue.js 3

Vue to progresywny framework JavaScript, który służy do budowania interfejsów użytkownika oraz aplikacji jednostronicowych (SPA – ang. Single Page Application). Jego głównymi zaletami są wydajność, prostota oraz elastyczność, co sprawia, że jest chętnie wybierany zarówno przez początkujących, jak i doświadczonych programistów. Vue oferuje modularną strukturę, która pozwala na stopniowe rozwijanie aplikacji oraz łatwą integrację z innymi bibliotekami i narzędziami [25].

2.3 Bazy danych i messaging

Microsoft SQL Server

Microsoft SQL Server relacyjny system zarządzania relacyjną bazą danych stworzony przez Microsoft. Charakteryzuje się wysoką wydajnością, niezawodnością i skalowalnością. Wykorzystuje język T-SQL [26] do zarządzania danymi i zapytań. SQL Server doskonale integruje się z technologiami Microsoft, co czyni go popularnym rozwiązaniem w aplikacjach biznesowych i korporacyjnych [27] [28].

[9] https://masstransit.io/introduction," [Online]. [Data uzyskania dostępu: 05 12 2024]

RabbitMQ

RabbitMQ jest systemem kolejkowania wiadomości (message broker) opartym na protokole AMQP (ang, Advanced Message Queuing Protocol), czyli standardowym protokole komunikacyjnym, służącym do niezawodnego przesyłania wiadomości między systemami[10]. Umożliwia asynchroniczną komunikację między usługami, zapewniając niezawodne przesyłanie, kolejkowanie i odbieranie wiadomości. Charakteryzuje się wysoką wydajnością, skalowalnością i wsparciem dla wielu języków programowania, dzięki czemu jest często stosowany w systemach rozproszonych i mikroserwisach [29].

2.4 Konteneryzacja i orkiestracja

Docker

Docker jest platformą do tworzenia, uruchamiania i zarządzania kontenerami, które izolują aplikacje wraz ze wszystkimi ich zależnościami. Dzięki temu zapewnia spójność działania aplikacji w różnych środowiskach, takich jak lokalne maszyny, serwery czy chmura. Docker jest lekki, szybki i umożliwia efektywne zarządzanie zasobami [30].

Docker Compose

Docker Compose jest narzędziem umożliwiającym definiowanie i uruchamianie aplikacji składających się z wielu kontenerów za pomocą jednego pliku konfiguracyjnego YAML [31]. Ułatwia to zarządzanie skomplikowanymi środowiskami, w których poszczególne usługi działają w oddzielnych kontenerach. Narzędzie to pozwala na szybkie wdrażanie, skalowanie i testowanie aplikacji wielokontenerowych[11] .

2.5 API Gateway

API Gateway jest pośrednikiem zarządzającym ruchem między klientami, a usługami w systemach opartych na mikroserwisach. Służy jako pojedynczy punkt wejścia dla żądań API, umożliwiając tym przekierowanie do odpowiednich usług [32]. Dodatkowo oferuje takie funkcje jak autoryzacja, czy limitowanie liczby żądań (ang. - rate limiting). Dzięki temu odciąża poszczególne mikroserwisy od zadań związanych z zarządzaniem ruchem, poprawiając ich wydajność i upraszczając architekturę systemu.

[10] https://boringowl.io/blog/wprowadzenie-do-amqp-advanced-message-queuing-protocol
[11] https://docs.docker.com/compose/

Ocelot

Ocelot jest to API Gateway dla platformy .NET, który służy jako brama wejściowa do mikroserwisów, zarządzając routingiem żądań, agregacją, loadbalancingiem oraz dostarczając funkcje takie jak cache'owanie czy rate limiting. Jest to narzędzie open-source, które pozwala na łatwe konfigurowanie tras za pomocą pliku JSON, dzięki czemu można w prosty sposób przekierowywać ruch z jednego punktu wejścia do różnych mikroserwisów [33].

3 Implementacja rozwiązania

Implementacja została zrealizowana etapowo. Pierwszym krokiem została stworzona wersja monolityczna, która pozwoliła na szybkie opracowanie podstawowej funkcjonalności oraz przetestowanie głównych założeń systemu. Model ten umożliwił na łatwiejsze zrozumienie logiki biznesowej oraz eliminację potencjalnych błędów na wczesnym etapie projektu. Następnie, zgodnie z podejściem rekomendowanym w literaturze [34], [35] przeprowadzona została dekompozycja monolitu na niezależne mikroserwisy. Proces dekompozycji polegał na identyfikacji kluczowych komponentów systemu, które następnie zostały wydzielone jako odrębne serwisy.

3.1 System w architekturze monolitycznej

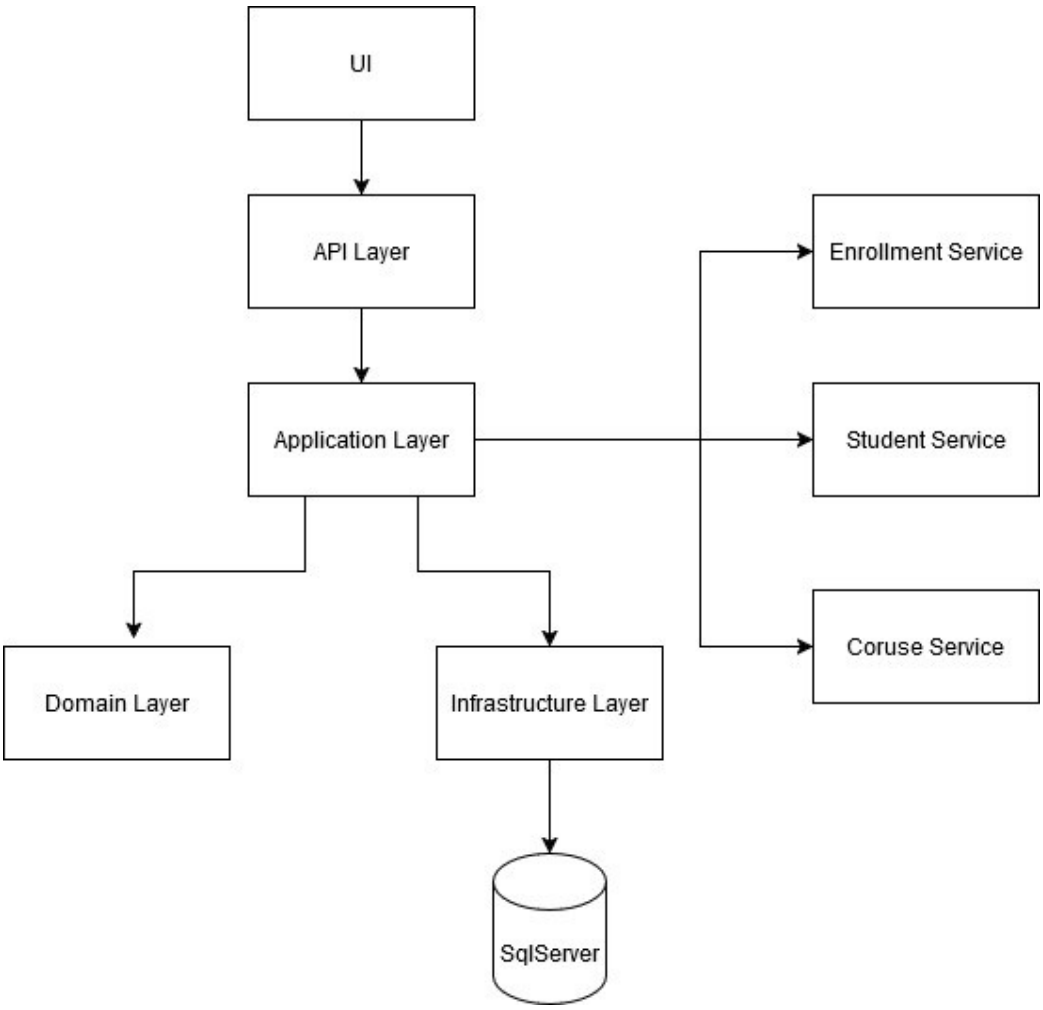
Wersja monolityczna systemu stanowiła pierwszy etap implementacji, umożliwiając szybkie stworzenie podstawowej funkcjonalności oraz weryfikację założeń projektu [36]. Na etapie projektu systemu wszystkie komponenty zostały zintegrowane w jednej strukturze, co uprościło planowanie oraz zrozumienie logiki biznesowej. W ramach szczegółów implementacji architektura monolityczna pozwoliła na efektywne tworzenie, testowanie i analizowanie funkcji w jednym środowisku, co sprzyjało eliminacji potencjalnych błędów. W ostatnim podrozdziale, omówione zostały najważniejsze obserwacje i doświadczenia związane z realizacją systemu w architekturze monolitycznej oraz jej wpływ na dalsze etapy rozwoju projektu.

3.1.1 Projekt systemu

Projekt systemu w architekturze monolitycznej został zrealizowany w technologii .NET 8.0, wykorzystując wzorzec warstwowy [37] (Rys. 1). System składa się z czterech głównych projektów:

- **LMS.Core** - zawiera modele domenowe i interfejsy
- **LMS.Infrastructure** - zawiera implementację dostępu do danych
- **LMS.Application** - zawiera logikę biznesową i serwisy
- **LMS.API** - warstwa prezentacji, kontrolery API

Każda warstwa pełni określoną rolę i ma dostęp tylko do warstw niższego poziomu. Taki podział pozwala na zachowanie separacji zagadnień i ułatwia utrzymanie kodu.



Rys. 1 Architektura aplikacji monolitycznej

3.1.2 Szczegóły Implementacji

Warstwa dostępu do danych

Przy projektowaniu systemu przyjęto podejście "database-first", gdzie najpierw zaprojektowano strukturę bazy danych zawierającą tabele:

- Courses - przechowującą informacje o kursach
- Students - zawierającą dane studentów
- Enrollments - przechowującą zapisy studentów na kursy
- CoursePrerequisites - definiującą wymagania wstępne dla kursów

W ramach implementacji warstwy dostępu do danych wykorzystano Entity Framework Core [38] jako ORM (ang. Object-Relational Mapping). Zdefiniowano kontekst bazy danych *ApplicationContext*, który zawiera konfigurację wszystkich encji (Kod 1) oraz ich konfigurację ich relacji w metodzie *OnModelCreating()* (Kod 2).

```
1.         public ApplicationContext(DbContextOptions<ApplicationContext> options) :
base(options){}
2.         public DbSet<Course> Courses { get; set; }
3.         public DbSet<Student> Students { get; set; }
4.         public DbSet<Enrollment> Enrollments { get; set; }
5.         public DbSet<CoursePrerequisite> CoursePrerequisites { get; set; }
```

Kod 1 Definicja encji w ApplicationContext

```
 1.  protected override void OnModelCreating(ModelBuilder modelBuilder)
 2.          {
 3.              // Course Configuration
 4.              modelBuilder.Entity<Course>(entity =>
 5.              {
 6.                  entity.HasKey(e => e.Id);
 7.                  entity.Property(e => e.Title).IsRequired().HasMaxLength(100);
 8.                  entity.Property(e => e.Description).HasMaxLength(500);
 9.                  entity.Property(e => e.MaxEnrollment).IsRequired();
10.                  entity.Property(e => e.CurrentEnrollment).IsRequired();
11.
12.                  // Course - Prerequisites relationship
13.                  entity.HasMany(e => e.Prerequisites)
14.                      .WithOne(p => p.Course)
15.                      .HasForeignKey(p => p.CourseId)
16.                      .OnDelete(DeleteBehavior.Restrict);
17.              });
18.
19.              // Student Configuration
20.              modelBuilder.Entity<Student>(entity =>
21.              {
22.                  entity.HasKey(e => e.Id);
23.                  entity.Property(e => e.FirstName).IsRequired().HasMaxLength(50);
24.                  entity.Property(e => e.LastName).IsRequired().HasMaxLength(50);
25.                  entity.Property(e => e.Email).IsRequired().HasMaxLength(100);
26.                  entity.HasIndex(e => e.Email).IsUnique();
27.              });
28. // Dalsza czesc kodu
29. }
30.
```

Kod 2 Przykład konfiguracji relacji pomiędzy encjami w ApplicationContext

Szczególną uwagę poświęcono mapowaniu relacji między kursami a ich wymaganiami wstępnymi, gdzie zastosowano relację wiele-do-wielu z dodatkową tabelą *CoursePrerequisite* (Kod 3). Zaimplementowano również mechanizm migracji, który pozwala na wersjonowanie schematu bazy danych.

```
1.      public class CoursePrerequisite
2.      {
3.          public int Id { get; set; }
4.          public int CourseId { get; set; }
5.          public int PrerequisiteCourseId { get; set; }
6.          public bool IsMandatory { get; set; }
7.          public Course Course { get; set; }
8.          public Course PrerequisiteCourse { get; set; }
9.      }
10.
```

Kod 3 Klasa CoursePrerequisite

Modele domenowe

Zdefiniowano kluczowe modele systemu: Course, Student oraz Enrollment. Każdy z nich posiada własne właściwości oraz logikę biznesową. W modelu Course znajdują się informacje na temat kursów (Kod 4). Model Student (Kod 5) zawiera podstawowe informacje o uczestnikach kursów, natomiast Enrollment (Kod 6) reprezentuje zapis na kurs wraz ze statusem i datą zapisu.

```
1. public class Course
2.      {
3.          public int Id { get; set; }
4.          public string Title { get; set; }
5.          public string Description { get; set; }
6.          public int MaxEnrollment { get; set; }
7.          public int CurrentEnrollment { get; set; }
8.          public DateTime CreatedDate { get; set; }
9.          public DateTime? UpdatedDate { get; set; }
10.         public ICollection<CoursePrerequisite> Prerequisites { get; set; }
11.         public ICollection<Enrollment> Enrollments { get; set; }
12.     }
13.
```

Kod 4 Klasa Course

```
1.      public class Student
2.      {
3.          public int Id { get; set; }
4.          public string FirstName { get; set; }
5.          public string LastName { get; set; }
6.          public string Email { get; set; }
7.          public DateTime DateOfBirth { get; set; }
8.          public DateTime CreatedDate { get; set; }
9.          public DateTime? UpdatedDate { get; set; }
10.         public ICollection<Enrollment> Enrollments { get; set; }
11.     }
12.
```

Kod 5 Klasa Student

```
    public class Enrollment
    {
        public int Id { get; set; }
        public int StudentId { get; set; }
        public int CourseId { get; set; }
        public DateTime EnrollmentDate { get; set; }
        public EnrollmentStatus Status { get; set; }
        public Student Student { get; set; }
        public Course Course { get; set; }
    }

```

Kod 6 Klasa Enrollment

Serwisy aplikacyjne

Warstwa serwisów zawiera główną logikę biznesową systemu. CourseService (Kod 7) odpowiada za zarządzanie kursami, StudentService (Kod 9) obsługuje operacje związane ze studentami, a EnrollmentService (Kod 8) implementuje złożoną logikę zapisów na kursy, włącznie z weryfikacją wymagań wstępnych i dostępności miejsc. W serwisach zaimplementowano obsługę transakcji, zapewniając spójność danych przy operacjach modyfikujących wiele encji.

```
public class CourseService : ICourseService
{
        private readonly ApplicationContext _context;
        private readonly IMapper _mapper;

        public CourseService(ApplicationContext context, IMapper mapper)
        {
            _context = context;
            _mapper = mapper;
        }
// Wybrane metody w CourseService
public async Task<CourseDto> CreateCourseAsync(CreateCourseDto courseDto)
 {
     var course = _mapper.Map<Course>(courseDto);
     _context.Courses.Add(course);
     await _context.SaveChangesAsync();
     return _mapper.Map<CourseDto>(course);
 }
public async Task<IEnumerable<CourseDto>> GetAllCoursesAsync()
{
    var courses = await _context.Courses.ToListAsync();
    return _mapper.Map<IEnumerable<CourseDto>>(courses);
}
public async Task<IEnumerable<PrerequisiteDto>> GetPrerequisitesAsync(int courseId)
 {
     return await _context.CoursePrerequisites
         .Where(p => p.CourseId == courseId)
         .Select(p => new PrerequisiteDto
         {
             CourseId = p.CourseId,
             PrerequisiteCourseId = p.PrerequisiteCourseId,
             PrerequisiteCourseName = p.PrerequisiteCourse.Title,
             IsMandatory = p.IsMandatory
         })
         .ToListAsync();
 }
}
```

Kod 7 Wybrane metody w CourseService

```
 1. public class EnrollmentService : IEnrollmentService
 2.  {
 3.      private readonly ApplicationContext _context;
 4.      private readonly IMapper _mapper;
 5.
 6.      public EnrollmentService(ApplicationContext context, IMapper mapper)
 7.      {
 8.          _context = context;
 9.          _mapper = mapper;
10.      }
11.
12.
13.      public async Task<EnrollmentDto> CreateEnrollmentAsync(CreateEnrollmentDto
enrollmentDto)
14.      {
15.          var prerequisites = await _context.CoursePrerequisites
16.              .Where(p => p.CourseId == enrollmentDto.CourseId)
17.              .ToListAsync();
18.
19.          var studentEnrollments = await _context.Enrollments
20.              .Where(e => e.StudentId == enrollmentDto.StudentId && e.Status ==
EnrollmentStatus.Completed)
21.              .Select(e => e.CourseId)
22.              .ToListAsync();
23.
24.          foreach (var prerequisite in prerequisites.Where(p => p.IsMandatory))
25.          {
26.              if (!studentEnrollments.Contains(prerequisite.PrerequisiteCourseId))
27.              {
28.                  throw new ApplicationException("Prerequisites not met");
29.              }
30.          }
31.
32.          var enrollment = new Enrollment
33.          {
34.              StudentId = enrollmentDto.StudentId,
35.              CourseId = enrollmentDto.CourseId,
36.              EnrollmentDate = DateTime.UtcNow,
37.              Status = EnrollmentStatus.Active
38.          };
39.
40.          _context.Enrollments.Add(enrollment);
41.          await _context.SaveChangesAsync();
42.
43.          return await GetEnrollmentAsync(enrollment.Id);
44.      }
45.
46.      public async Task<IEnumerable<EnrollmentDto>> GetAllEnrollmentsAsync()
47.      {
48.          var enrollments = await _context.Enrollments.ToListAsync();
49.
50.          return _mapper.Map<IEnumerable<EnrollmentDto>>(enrollments);
51.      }
52. }
53.
```

Kod 8 Wybrane metody EnrollmentService

```
1.   public class StudentService : IStudentService
2.   {
3.       private readonly ApplicationContext _context;
4.       private readonly IMapper _mapper;
5.       public StudentService(ApplicationContext context, IMapper mapper)
6.       {
7.           _context = context;
8.           _mapper = mapper;
9.       }
10.      public async Task<StudentDto> CreateStudentAsync(CreateStudentDto studentDto)
11.      {
12.          var student = _mapper.Map<Student>(studentDto);
13.
14.          _context.Students.Add(student);
15.          await _context.SaveChangesAsync();
16.
17.          return _mapper.Map<StudentDto>(student);
18.      }
19.      public async Task<StudentDto> GetStudentByIdAsync(int id)
20.      {
21.          var student = await _context.Students.FindAsync(id);
22.          return _mapper.Map<StudentDto>(student);
23.      }
24.
25.      public async Task<IEnumerable<EnrollmentDto>> GetStudentEnrollmentsAsync(int
studentId)
26.      {
27.          var enrollments = await _context.Enrollments
28.              .Include(e => e.Course)  // Include course details
29.              .Include(e => e.Student) // Include student details
30.              .Where(e => e.StudentId == studentId)
31.              .Select(e => new EnrollmentDto
32.              {
33.                  Id = e.Id,
34.                  StudentId = e.StudentId,
35.                  CourseId = e.CourseId,
36.                  StudentName = $"{e.Student.FirstName} {e.Student.LastName}",
37.                  CourseName = e.Course.Title,
38.                  EnrollmentDate = e.EnrollmentDate,
39.                  Status = e.Status.ToString()
40.              })
41.              .ToListAsync();
42.
43.          return enrollments;
44.      }
45.  }
46.
```

Kod 9 Wybrane metody StudentService

Kontrolery API

Zaimplementowano kontrolery REST API dla każdego głównego modułu systemu. Kontrolery odpowiadają za przyjmowanie żądań HTTP, walidację danych wejściowych oraz przekazywanie ich do odpowiednich serwisów. Dla przykładu na figurze Kod 10 znajduje się implementacja kontrolera dla kursów.

```
 1. [Route("api/[controller]")]
 2.     [ApiController]
 3.     public class CourseController : ControllerBase
 4.     {
 5.         private readonly ICourseService _courseService;
 6.         private readonly ILogger<CourseController> _logger;
 7.         public CourseController(ICourseService courseService,
ILogger<CourseController> logger)
 8.         {
 9.             _courseService = courseService;
10.             _logger = logger;
11.         }
12.         [HttpGet]
13.         public async Task<ActionResult<IEnumerable<CourseDto>>> GetCourses()
14.         {
15.             var courses = await _courseService.GetAllCoursesAsync();
16.             return Ok(courses);
17.         }
18.         [HttpGet("{id}")]
19.         public async Task<ActionResult<CourseDto>> GetCourse(int id)
20.         {
21.             var course = await _courseService.GetCourseByIdAsync(id);
22.             if (course == null) return NotFound();
23.             return Ok(course);
24.         }
25.         [HttpPost]
26.         public async Task<ActionResult<CourseDto>> CreateCourse(CreateCourseDto
courseDto)
27.         {
28.             try
29.             {
30.                 var course = await _courseService.CreateCourseAsync(courseDto);
31.                 return CreatedAtAction(nameof(GetCourse), new { id = course.Id },
course);
32.             }
33.             catch (Exception ex)
34.             {
35.                 _logger.LogError(ex, "Error creating course");
36.                 return BadRequest(ex.Message);
37.             }
38.         }
39.         [HttpGet("{id}/prerequisites")]
40.         public async Task<ActionResult<IEnumerable<PrerequisiteDto>>>
GetPrerequisites(int id)
41.         {
42.             var prerequisites = await _courseService.GetPrerequisitesAsync(id);
43.             return Ok(prerequisites);
44.         }
45.         [HttpPost("{id}/prerequisites")]
46.         public async Task<ActionResult<PrerequisiteDto>> AddPrerequisite(int id,
AddPrerequisiteDto prerequisiteDto)
47.         {
48.             try
49.             {
50.                 var prerequisite = await _courseService.AddPrerequisiteAsync(id,
prerequisiteDto);
51.                 return Ok(prerequisite);
52.             }
53.             catch (Exception ex)
54.             {
55.                 _logger.LogError(ex, "Error adding prerequisite");
56.                 return BadRequest(ex.Message);
57.             }
58.         }
59.     }
```

Kod 10 Implementacja kontrolera CourseController

Zastosowano wzorzec DTO (Data Transfer Objects) do bezpiecznej komunikacji między klientem a serwerem, aby ograniczyć ilość przesyłanych danych oraz ukryć wewnętrzną strukturę obiektów domenowych. Przykładowy model DTO znajduje się na Kod 11.

```
1.     public class CourseDto
2.     {
3.         public int Id { get; set; }
4.         public string Title { get; set; }
5.         public string Description { get; set; }
6.         public int MaxEnrollment { get; set; }
7.         public int CurrentEnrollment { get; set; }
8.     }
9.
```

Kod 11 CourseDTO

Konfiguracja aplikacji

W ramach konfiguracji aplikacji, skonfigurowane zostało DI (ang. Dependency Injection), co umożliwia zarządzanie zależnościami oraz wstrzykiwanie ich w różnych częściach systemu. Na początku, w głównej konfiguracji aplikacji, wywoływane są metody rozszerzające (Kod 12).

```
1. builder.Services.AddInfrastructure(builder.Configuration);
2. builder.Services.AddApplication();
```

Kod 12 Rejestracja zależności DI

Metoda *AddApplication(),* rejestruje nam *AutoMapper* oraz za pomocą metody AddScoped rejetruje konkretne serwisy (Kod 13).

```
 1.   public static IServiceCollection AddApplication(this IServiceCollection services)
 2.         {
 3.             services.AddAutoMapper(typeof(MappingProfile).Assembly);
 4.             services.AddScoped<ICourseService, CourseService>();
 5.             services.AddScoped<IStudentService, StudentService>();
 6.             services.AddScoped<IEnrollmentService, EnrollmentService>();
 7.
 8.             return services;
 9.         }
10.
```

Kod 13 Rejestracja zależnosci DI serwisów aplikacji

Skonfigurowano middleware pipeline. Polega to na dodaniu komponentów middleware do przetwarzania żądań (Kod 14).

```
 1. // Configure the HTTP request pipeline.
 2. if (app.Environment.IsDevelopment())
 3. {
 4.     app.UseSwagger();
 5.     app.UseSwaggerUI();
 6. }
 7. app.UseCors("AllowVueApp");
 8. app.UseHttpsRedirection();
 9. app.UseAuthorization();
10. app.MapControllers();
```

Kod 14 Konfiguracja middleware pipeline

Do ustawienia połączenia z bazą danych zastosowano wzorzec *Options Pattern* do zarządzania konfiguracją aplikacji, Pozwala on na wygodne mapowanie ustawień z plików konfiguracyjnych (Kod 15, Kod 16)

```
 1. public static IServiceCollection AddInfrastructure(this IServiceCollection
services,IConfiguration configuration)
 2.  {
 3.      services.AddDbContext<ApplicationContext>(options =>
 4.          options.UseSqlServer(
 5.              configuration.GetConnectionString("DefaultConnection"),
 6.              b => b.MigrationsAssembly(typeof(ApplicationContext).Assembly.FullName)
 7.          )
 8.      );
 9.
10.      services.AddScoped<DataSeeder>();
11.      return services;
12.  }
13.
```

Kod 15 Wstrzykniecie zależności bazodanowej

```
1.   "ConnectionStrings": {
2.     "DefaultConnection":
"Server=(localdb)\\MSSQLLocalDB;Database=LMS.Monolith;Trusted_Connection=True;MultipleActi
veResultSets=true"
3.   },
4.
```

Kod 16 Fragment konfiguracji do bazy danych

3.1.3 Wnioski z implementacji systemu w architekturze monolitycznej

Prosta implementacja i zarządzanie

W architekturze monolitycznej cała logika biznesowa, interfejs użytkownika i warstwa danych są zawarte w jednej, spójnej aplikacji. To znacząco upraszcza proces tworzenia, testowania i wdrażania. Brak konieczności koordynowania komunikacji między wieloma usługami sprawia, że monitorowanie i debugowanie błędów są prostsze i bardziej bezpośrednie. Wymagania dotyczące wdrożenia są znacznie mniejsze – najczęściej wystarczy pojedynczy serwer lub maszyna wirtualna, co redukuje koszty związane z infrastrukturą.

Spójność danych

Monolityczne systemy korzystają z jednej, centralnej bazy danych, co ułatwia zarządzanie transakcjami. Brak potrzeby synchronizacji danych między rozproszonymi bazami redukuje ryzyko niespójności. a także eliminuje problemy związane z replikacją danych. Centralna baza danych ułatwia również monitorowanie, utrzymanie i optymalizację wydajności, ponieważ całość operacji odbywa się w jednym, spójnym środowisku.

Szybsze wdrażanie początkowej wersji

W architekturze monolitycznej zespoły programistyczne mogą szybciej stworzyć prototyp systemu i wdrożyć pierwszą wersję aplikacji, ponieważ wszystkie komponenty są zintegrowane w jednej aplikacji. Brak potrzeby koordynacji pracy między wieloma serwisami sprzyja krótszemu czasowi dostarczenia.

Wydajność w przypadku mniejszych aplikacji

Monolit często może oferować lepszą wydajność w przypadku mniejszych systemów, ponieważ komunikacja między komponentami odbywa się lokalnie (w pamięci) i nie wymaga dodatkowych protokołów sieciowych. Brak narzutu związanego z sieciową komunikacją między usługami sprzyja szybszemu przetwarzaniu operacji.

Złożoność i wyzwania związane z rozwojem dużych systemów

W przypadku dużych aplikacji monolitycznych skalowanie systemu staje się trudne, ponieważ cała aplikacja musi być uruchomiona w całości. Skalowanie pojedynczych komponentów nie jest możliwe. W miarę rozwoju aplikacji kod źródłowy staje się coraz większy i bardziej skomplikowany, co może prowadzić do powstawania "Big Ball of Mud"[12] – chaotycznej, trudnej do zrozumienia i rozwijania struktury kodu. W monolicie wszystkie komponenty są związane z jedną technologią i środowiskiem, co ogranicza możliwość wprowadzania nowych narzędzi czy rozwiązań. Każda zmiana w kodzie wymaga ponownego wdrożenia całej aplikacji, co może wydłużyć proces deploymentu i zwiększyć ryzyko awarii.

Mniejsza odporność na błędy

Ponieważ wszystkie komponenty działają w ramach jednej aplikacji, awaria jednego modułu może wpłynąć na działanie całego systemu, co zmniejsza odporność na błędy. W przypadku dużych aplikacji trudno jest także odizolować problematyczny fragment systemu i wdrożyć szybkie poprawki.

[12] https://blog.codinghorror.com/the-big-ball-of-mud-and-other-architectural-disasters/," [Online]. [Data uzyskania dostępu: 12 12 2024]

3.2 System w architekturze mikroserwisów

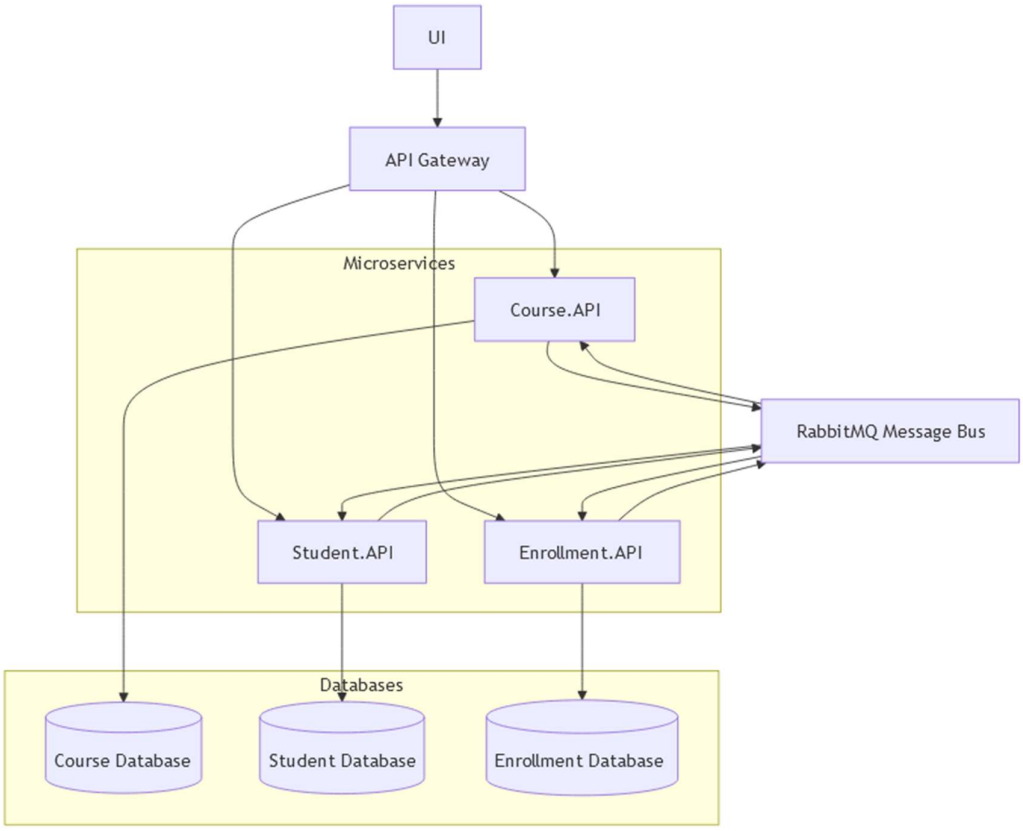
Po początkowej implementacji systemu jako monolitu, dokonano jego podziału na niezależne mikroserwisy. W pierwszego podrozdziału zidentyfikowane zostały kluczowe elementy systemu, które zostały wyodrębnione jako samodzielne jednostki, umożliwiając ich niezależny rozwój i wdrożenie. W kolejnym etapie zaprojektowano oraz omówiono komunikację między serwisami, która opierała się na protokołach, takich jak HTTP/REST oraz komunikatach asynchronicznych, co zapewniło efektywną wymianę danych między komponentami systemu. W ostatnim podrozdziale, zostały przedstawione szczegółowe obserwacje i rezultaty wynikające z przejścia na architekturę mikroserwisową, w tym jej wpływ na rozwój i utrzymanie systemu.

3.2.1 Dekompozycja na mikroserwisy

Proces dekompozycji aplikacji monolitycznej na aplikacje mikroserwisową stanowił kluczowy etap w ewolucji systemu. Dekompozycja ta wymagała dokładnej analizy istniejącego monolitu oraz zaplanowania strategii podziału na niezależne komponenty [39]. Głównym celem było przekształcenie pojedynczej aplikacji, na zestaw autonomicznych serwisów, z których każdy odpowiada za konkretną domenę biznesową [40].

Analiza dekompozycji, poza aspektami technicznymi, obejmowała przede wszystkim logikę biznesową oraz wzajemne zależności między komponentami. Każdy wydzielony serwis został zaprojektowany tak, aby mógł działać niezależnie, posiadając własną bazę danych oraz jasno zdefiniowany interfejs do komunikacji [41].

Efektem tego procesu było wydzielenie trzech niezależnych serwisów: Course Serivce, Student Service oraz Enrollment Service. Każdy z tych serwisów otrzymał własną bazę danych. W celu komunikacji między serwisami zastosowano RabbitMQ, aby serwisy te były w stanie informować się o zdarzeniach wzajemnie na siebie wpływających. Diagram aplikacji mikroserwisowej został przedstawiony na Rys. 2



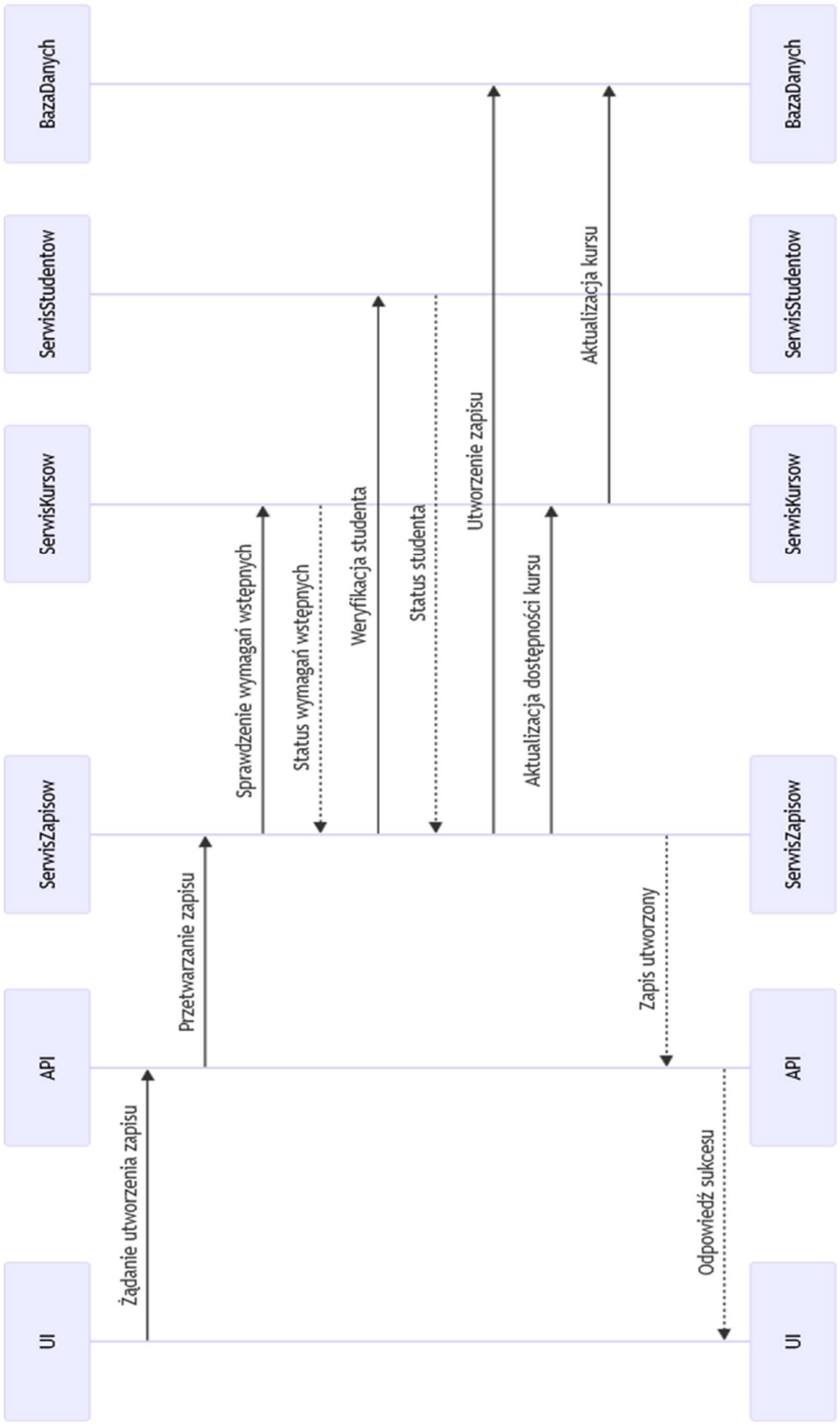
Rys. 2 Diagram aplikacji mikroserwisowej

Komunikacja miedzy serwisami

Komunikacja między komponentami w monolicie była prosta oraz bezpośrednia przede wszystkim dzięki wspólnej bazie danych. W tym przypadku każdy z serwisów miał bezpośredni dostęp do całej bazy, przez co istniała możliwość tworzenia złożonych zapytań łączących wiele tabel (Kod 17). Przykładem tego jest proces zapisu studenta na kurs przedstawiony na Rys. 3.

```
1. public async Task<CourseDetails> GetCourseDetails(int courseId) {
2.     return await _context.Courses .Include(c => c.Students) .Include(c =>
c.Enrollments) .FirstOrDefaultAsync(c => c.Id == courseId);
3.  }
4.
```

Kod 17 Przykład zapytania łączonego wiele tabel



Rys. 3 Przepływ danych dla procesu zapisu studenta na kurs w aplikacji monolitycznej

Diagram przedstawia proces zapisu studenta na kurs w ramach aplikacji monolitycznej, ukazując interakcje pomiędzy klientem, API, poszczególnymi warstwami serwisowymi oraz bazą danych.

1. **Żądanie od UI**

   Proces rozpoczyna się od wysłania przez UI żądania zapisu studenta na kurs do warstwy API. Żądanie to zawiera dane dotyczące studenta oraz kursu, na który ma być zapisany.

2. **Przetwarzanie zapisu w serwisie zapisów**

   API deleguje żądanie do Serwisu Zapisów (EnrollmentService), który pełni główną rolę w zarządzaniu logiką zapisu.

3. **Sprawdzenie wymagań wstępnych**

   Serwis Zapisów komunikuje się z Serwisem Kursów (CourseService), aby sprawdzić, czy student spełnia wymagania wstępne dla wybranego kursu. Serwis Kursów zwraca status wymagań wstępnych do Serwisu Zapisów

4. **Weryfikacja studenta**

   Jeśli wymagania wstępne są spełnione, Serwis Zapisów odwołuje się do Serwisu Studentów (StudentService) w celu weryfikacji statusu studenta (np. czy student istnieje i ma uprawnienia). Serwis Studentów zwraca wynik weryfikacji do Serwisu Zapisów.

5. **Utworzenie zapisu w bazie danych:**

   Po pozytywnej weryfikacji studentów i wymagań wstępnych, Serwis Zapisów tworzy nowy zapis w Bazie Danych. Informacje o zapisach są tam zapisywane jako nowy rekord.

6. **Aktualizacja dostępności kursu**

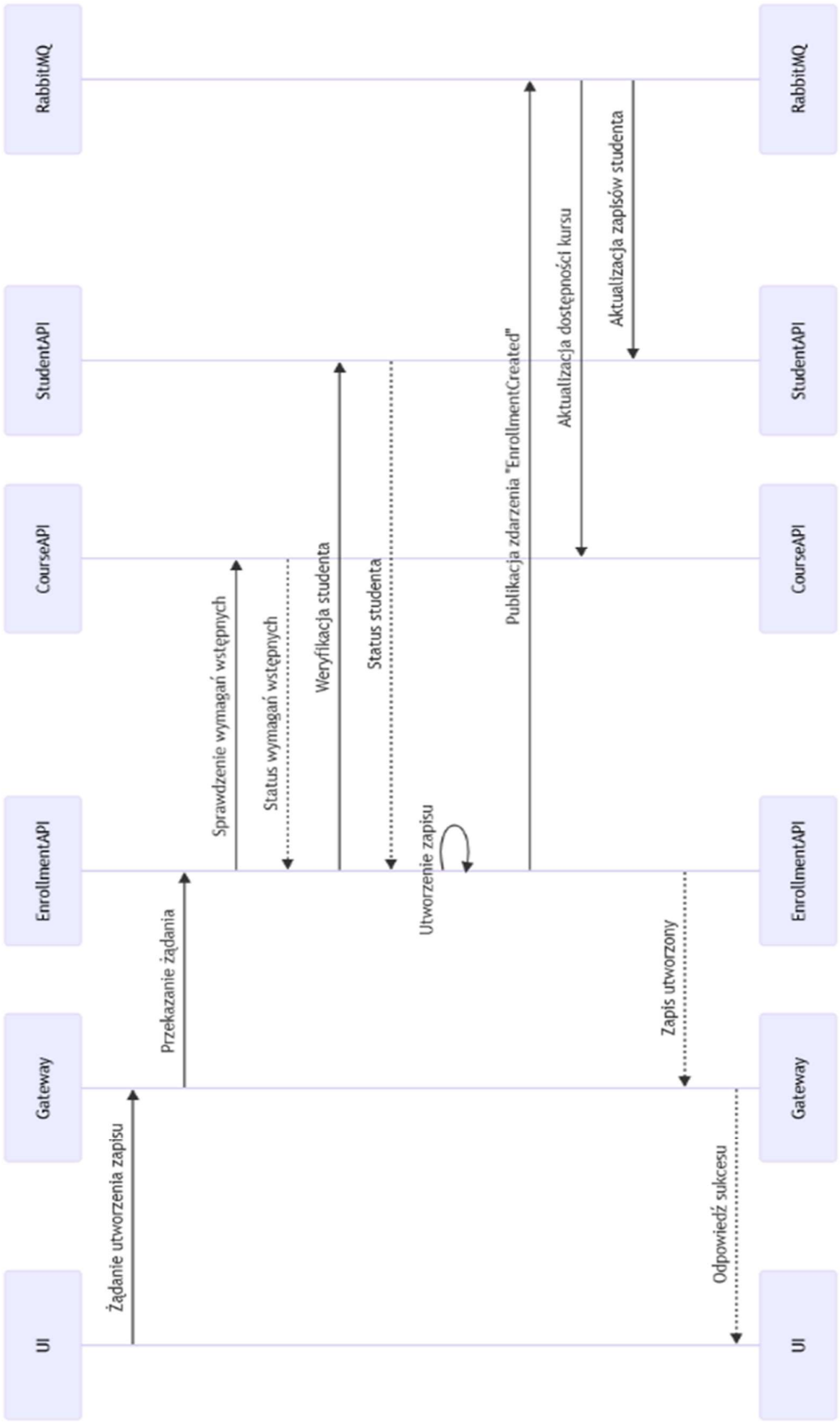
   Następnie Serwis Zapisów informuje Serwis Kursów o konieczności zmniejszenia dostępnej liczby miejsc na kursie.

   Serwis Kursów aktualizuje te dane w Bazie Danych, zmniejszając liczbę dostępnych miejsc.

7. **Odpowiedź**

   Gdy proces zakończy się Serwis Zapisów wysyła potwierdzenie do API, które następnie przekazuje odpowiedź sukcesu do Klienta

Proces ten musiał znacznie się zmienić gdy aplikacja została zmieniona na architekturę mikroserwisową. W tym podejściu każdy mikroserwis stał się autonomiczną jednostką zarządzającą własną konfiguracją oraz połączeniami z bazą danych (Rys. 4).



Rys. 4 Przepływ danych zapisu studenta na kurs w aplikacji mikroserwisowej

1. **Żądanie utworzenia zapisu przez klienta**

   Proces rozpoczyna się, gdy Klient (interfejs użytkownika) wysyła żądanie utworzenia nowego zapisu na kurs do Gateway. Gateway pełni rolę punktu wejściowego (API Gateway), który odbiera żądanie i przekazuje je do odpowiedniego serwisu backendowego.
2. **Przekazanie żądania do EnrollmentAPI**

   Gateway przekazuje żądanie do mikroserwisu EnrollmentAPI. Serwis ten jest odpowiedzialny za logikę zapisu studenta na kursSprawdzenie wymagań wstępnych.
3. **Sprawdzenie wymagań wstępnych**

   EnrollmentAPI komunikuje się z CourseAPI za pomocą HTTP, aby sprawdzić, czy student spełnia wymagania wstępne dla wybranego kursu. CourseAPI zwraca status wymagań wstępnych do EnrollmentAPI.
4. **Weryfikacja studenta:**

   Po uzyskaniu statusu wymagań, EnrollmentAPI wysyła zapytanie do StudentAPI w celu weryfikacji studenta (czy student istnieje w systemie i ma prawo do zapisu).
5. **Utworzenie zapisu:**

   Gdy wszystkie weryfikacje przebiegną pomyślnie (wymagania i status studenta są poprawne), EnrollmentAPI tworzy zapis we własnym kontekście
6. **Publikacja zdarzenia do RabbitMQ**

   EnrollmentAPI publikuje zdarzenie EnrollmentCreated do systemu kolejkowego RabbitMQ. RabbitMQ umożliwia asynchroniczną komunikację między mikroserwisami, zapewniając luźne powiązania między nimi
7. **Aktualizacja stanu w innych serwisach**

   RabbitMQ przesyła zdarzenie do CourseAPI, który aktualizuje dostępność miejsc na kursie (zmniejsza liczbę dostępnych miejsc). Przesłane jest także zdarzenie do StudentAPI, który aktualizuje informacje o zapisach studenta.
8. **Odpowiedz sukcesu**

   Po zakończeniu wszystkich operacji, EnrollmentAPI przesyła odpowiedź do Gateway, który przekazuje komunikat sukcesu z powrotem do UI.

Kluczową zmianą był sposób informowania innych serwisów o zmianach. Poprzednio z powodu jednej bazy danych, aktualizacja informacji była bezpośrednia. W obecnym

rozwiązaniu wykorzystano dwa sposoby komunikowania się z innymi mikroserwisami: komunikacja synchroniczna (REST API) oraz RabbitMQ.

Kod 18 ilustruje komunikację między mikroserwisami przy użyciu protokołu HTTP. W architekturze mikroserwisowej usługi takie jak Course.API i Student.API działają jako niezależne serwisy, które udostępniają swoje funkcjonalności za pośrednictwem endpointów HTTP. Metoda CheckPrerequisites sprawdza, czy student spełnia wymagania wstępne dla danego kursu, korzystając z dwóch zewnętrznych API: Course.API i Student.API. Za pomocą klienta HTTP metoda wysyła żądanie GET do Course.API i pobiera listę wymagań wstępnych kursu. Kolejne żądanie GET do Student.API zwraca listę identyfikatorów kursów ukończonych przez studenta.

Przy użyciu LINQ sprawdzane jest, czy wszystkie obowiązkowe wymagania wstępne znajdują się na liście ukończonych kursów. W przypadku problemów z komunikacją z API rzucany jest wyjątek ServiceUnavailableException. Metoda zwraca true, jeśli wymagania są spełnione, lub false, jeśli nie.

```
1. private async Task<bool> CheckPrerequisites(int studentId, int courseId)
2.  {
3.      try
4.      {
5.          // Get prerequisites from Course.API
6.          var courseClient = _clientFactory.CreateClient("CourseAPI");
7.          var prerequisitesResponse = await courseClient
8.              .GetAsync($"/api/course/{courseId}/prerequisites");
9.
10.         if (!prerequisitesResponse.IsSuccessStatusCode)
11.             throw new ServiceUnavailableException("Could not get prerequisites");
12.
13.         var prerequisites = await prerequisitesResponse.Content
14.             .ReadFromJsonAsync<List<PrerequisiteDto>>();
15.
16.         // Get completed courses from Student.API
17.         var studentClient = _clientFactory.CreateClient("StudentAPI");
18.         var completedCoursesResponse = await studentClient
19.             .GetAsync($"/api/student/{studentId}/completed-courses");
20.
21.         if (!completedCoursesResponse.IsSuccessStatusCode)
22.             throw new ServiceUnavailableException("Could not get completed courses");
23.
24.         var completedCourses = await completedCoursesResponse.Content
25.             .ReadFromJsonAsync<List<int>>();
26.
27.         // Check if all mandatory prerequisites are met
28.         return prerequisites
29.             .Where(p => p.IsMandatory)
30.             .All(p => completedCourses.Contains(p.PrerequisiteCourseId));
31.     }
32.     catch (Exception ex)
33.     {
34.         _logger.LogError(ex, "Error checking prerequisites");
35.         throw;
36.     }
37. }
```

Kod 18 Przykład wywołania innych serwisów za pomocą HTTP

W mikroserwisowej wersji systemu LMS, wysyłanie zdarzeń do RabbitMQ jest realizowane poprzez publikowanie oraz konsumpcję zdarzeń. Zdarzenia są zdefiniowane w postaci obiektu z potrzebnymi danymi (Kod 19)

```
1.      public class EnrollmentCreatedEvent
2.      {
3.          public int Id { get; set; }
4.          public int StudentId { get; set; }
5.          public int CourseId { get; set; }
6.          public DateTime EnrollmentDate { get; set; }
7.      }
8.
```

Kod 19 Zdarzenie stworzenia zapisu

Po tworzenia zapisu oraz po uzyskaniu oraz zweryfikowaniu wstępnych informacji od innych serwisów, metoda *CreateEnrollmentAsync* (Kod 20) publikuje zdarzenie *EnrollmentCreatedEvent* za pomocą *_publishEndpoints.Publish*(), która wysyła to zdarzenie do kolejki RabbitMQ. Wzorzec ten nazywa się Architektura Sterowana Zdarzeniami (ang. Event Driven Architecture) [42], umożliwia on innym mikroserwisom asynchroniczne reagowanie na to zdarzenie.

```
1.      public async Task<EnrollmentDto> CreateEnrollmentAsync(CreateEnrollmentDto dto)
 2.      {
 3.          _logger.LogInformation("Creating enrollment for student {StudentId} in course
{CourseId}",
 4.              dto.StudentId, dto.CourseId);
 5.          // 1. Check prerequisites via HTTP (need immediate response)
 6.          var prerequisitesMet = await CheckPrerequisites(dto.StudentId, dto.CourseId);
 7.          if (!prerequisitesMet)
 8.              throw new PrerequisitesNotMetException("Not all prerequisites are met");
 9.
10.          // 2. Create enrollment
11.          var enrollment = new Models.Enrollment
12.          {
13.              StudentId = dto.StudentId,
14.              CourseId = dto.CourseId,
15.              EnrollmentDate = DateTime.UtcNow,
16.              Status = EnrollmentStatus.Pending
17.          };
18.
19.          _context.Enrollments.Add(enrollment);
20.          await _context.SaveChangesAsync();
21.
22.          _logger.LogInformation("Creating enrollment for student {StudentId} in course
{CourseId}",
23.              enrollment.StudentId, enrollment.CourseId);
24.          await _publishEndpoint.Publish(new EnrollmentCreatedEvent
25.          {
26.              Id = enrollment.Id,
27.              StudentId = enrollment.StudentId,
28.              CourseId = enrollment.CourseId,
29.              EnrollmentDate = enrollment.EnrollmentDate
30.          });
31.          _logger.LogInformation("Published EnrollmentCreated event for enrollment
{EnrollmentId}", enrollment.Id);
32.
33.          return _mapper.Map<EnrollmentDto>(enrollment);
34.      }
```

Kod 20 Implementacja publikacji eventu zapisu przez MassTransit

Konsument *EnrollmentCreatedConsumer* nasłuchuje na kolejce RabbitMQ zdarzeń typu *EnrollmentCreatedEvent* i przetwarza je asynchronicznie. Po otrzymaniu zdarzenia, sprawdza w bazie danych, czy istnieje student o podanym StudentId, a następnie dodaje wpis o nowym zapisie studenta na kurs do tabeli StudentEnrollments. Proces kończy się zapisaniem zmian w bazie danych i zalogowaniem informacji o przetworzeniu zdarzenia. Proces ten został przedstawiony w Kod 21.

```
 1. public class EnrollmentCreatedConsumer : IConsumer<EnrollmentCreatedEvent>
 2. {
 3.     private readonly StudentContext _context;
 4.     private readonly ILogger<EnrollmentCreatedConsumer> _logger;
 5.     public EnrollmentCreatedConsumer(StudentContext context,
ILogger<EnrollmentCreatedConsumer> logger)
 6.     {
 7.         _context = context;
 8.         _logger = logger;
 9.     }
10.     public async Task Consume(ConsumeContext<EnrollmentCreatedEvent> context)
11.     {
12.         _logger.LogInformation("Consuming EnrollmentCreated event: StudentId
{StudentId}", context.Message.StudentId);
13.         var student = await _context.Students.FindAsync(context.Message.StudentId);
14.         if (student != null)
15.         {
16.             var enrollment = new StudentEnrollment
17.             {
18.                 StudentId = context.Message.StudentId,
19.                 CourseId = context.Message.CourseId,
20.                 EnrollmentDate = context.Message.EnrollmentDate,
21.                 Status = EnrollmentStatus.Active
22.             };
23.             _context.StudentEnrollments.Add(enrollment);
24.             await _context.SaveChangesAsync();
25.             _logger.LogInformation("Added enrollment for student {StudentId} in course
{CourseId}",
26.                 student.Id, context.Message.CourseId);
27.         }
28.     }
29. }
30.
```

Kod 21 Implementacja konsumowania zdarzenia zapisu przez StudentService

Dzięki użyciu RabbitMQ i biblioteki MassTransit, konsument automatycznie reaguje na pojawienie się nowego zdarzenia w kolejce poprzez odpowiednią konfigurację (Kod 22).

```
 1. builder.Services.AddMassTransit(x =>
 2. {
 3.     // Add consumers
 4.     x.AddConsumer<EnrollmentCreatedConsumer>();
 5.     x.AddConsumer<EnrollmentCancelledConsumer>();
 6.
 7.     x.UsingRabbitMq((context, cfg) =>
 8.     {
 9.         cfg.Host(builder.Configuration["EventBusSettings:HostAddress"]);
10.
11.         // Configure consumers
12.         cfg.ConfigureEndpoints(context);
13.     });
14. });
```

Kod 22 Konfiguracja MassTransit

API Gateway

W architekturze monolitycznej API Gateway nie było potrzebne, ponieważ cała logika i dane były obsługiwane w ramach jednego procesu, a wszystkie żądania trafiały bezpośrednio do jednego punktu wejścia. W przypadku mikroserwisów Gateway pełni rolę pośrednika między klientem (UI) a mikroserwisami i jest zrealizowane za pomocą biblioteki Ocelot w .NET (Kod 23).

```
1. // Add Ocelot configuration file
2. builder.Configuration.SetBasePath(builder.Environment.ContentRootPath)
3.     .AddJsonFile("ocelot.json", optional: false, reloadOnChange: true)
4.     .AddEnvironmentVariables();
5.
6. // Add Ocelot
7. builder.Services.AddOcelot(builder.Configuration);
8.
```

Kod 23 Konfiguracjia Ocelot w .Net

W przypadku mikroserwisów mamy wiele niezależnych usług, które realizują różne funkcjonalności systemu. Bez API Gateway klient musiałby:

- Znać adresy poszczególnych mikroserwisów (CourseAPI, StudentAPI, EnrollmentAPI).
- Wysyłać żądania bezpośrednio do każdego serwisu.
- Obsługiwać różne formaty odpowiedzi

API Gateway rozwiązuje te problemy, pełniąc funkcję centralnego punktu wejścia do systemu mikroserwisowego. API Gateway odbiera żądania od klienta i przekazuje je do odpowiedniego mikroserwisu na podstawie zdefiniowanych reguł w pliku *ocelot.json* (Kod 24). Wartość *DownstreamPathTemplate* definiuje ścieżkę, na którą zostanie przekazane żądanie. Wartość */api/{everything}* wskazuje, że dowolna podścieżka, gdzie *{everything}* jest symbolem zastępczym, zostanie przekazana dalej. *DownstreamScheme* ustawia protokół komunikacji (HTTP). Klucz *DownstreamHostAndPorts* wskazuje mikroserwis docelowy. W tym przypadku *Host course.api* jest nazwą hosta usługi, a *Port: 80* wskazuje port, na którym działa dana usługa. *UpstreamPathTemplate* definiuje ścieżkę, jaką użytkownik końcowy będzie wywoływał w API Gateway. W polu *UpstreamHttpMethod* definiowana jest lista obsługiwanych metod HTTP.

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/{everything}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "course.api",
          "Port": 80
        }
      ],
      "UpstreamPathTemplate": "/course-api/{everything}",
      "UpstreamHttpMethod": ["Get", "Post", "Put", "Delete"]
    },
    {
      "DownstreamPathTemplate": "/api/{everything}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "student.api",
          "Port": 80
        }
      ],
      "UpstreamPathTemplate": "/student-api/{everything}",
      "UpstreamHttpMethod": ["Get", "Post", "Put", "Delete"]
    },
    {
      "DownstreamPathTemplate": "/api/{everything}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "enrollment.api",
          "Port": 80
        }
      ],
      "UpstreamPathTemplate": "/enrollment-api/{everything}",
      "UpstreamHttpMethod": ["Get", "Post", "Put", "Delete"]
    }
  ],
  "GlobalConfiguration": {
    "BaseUrl": "http://gateway.api"
  }
}

```

Kod 24 Plik ocelot.json

3.2.2 Wnioski implementacji systemu w architekturze mikroserwisowej

Niezależność komponentów

Każdy mikroserwis działa jako niezależna jednostka, która realizuje konkretne zadanie lub odpowiada za określoną funkcjonalność systemu. Dzięki temu możliwa jest izolacja błędów – awaria jednego mikroserwisu nie musi prowadzić do całkowitego zatrzymania systemu. Skalowalność staje się bardziej elastyczna – tylko te komponenty, które wymagają większej wydajności, mogą być skalowane niezależnie.

Wygoda pisania i wdrażania

Mikroserwisy można tworzyć przy użyciu różnych technologii i języków programowania, co pozwala zespołom programistycznym wybierać najbardziej odpowiednie narzędzia dla danego komponentu. Zespół odpowiedzialny za dany mikroserwis może rozwijać, testować i wdrażać swoje rozwiązanie niezależnie od innych zespołów, co skraca czas dostarczania funkcjonalności.

Łatwiejsza konserwacja i rozwój

Mikroserwisy można aktualizować niezależnie, bez potrzeby ingerencji w cały system. Dzięki modularnej budowie łatwiej jest zarządzać kodem oraz utrzymywać i rozwijać poszczególne komponenty. Każdy mikroserwis ma jasno zdefiniowaną granicę odpowiedzialności, co ułatwia zarówno debugowanie, jak i dodawanie nowych funkcji.

Przejrzystość i podział odpowiedzialności

W architekturze mikroserwisowej system jest podzielony na mniejsze, łatwiejsze do zrozumienia komponenty, co sprawia, że złożone systemy są bardziej przejrzyste. Wysoka spójność i niski stopień powiązań między komponentami wspierają zasadę SRP (ang. Single Responsibility Principle)**.**

Zwiększona elastyczność biznesowa

Umożliwia szybkie wdrażanie nowych funkcjonalności i łatwe testowanie eksperymentalnych zmian na małych fragmentach systemu. Mikroserwisy pozwalają na szybszą adaptację do zmieniających się wymagań biznesowych i rynkowych.

Wyzwania

Należy jednak pamiętać o potencjalnych wyzwaniach związanych z mikroserwisami. W architekturze mikroserwisowej system składa się z wielu małych, niezależnych komponentów. Każdy mikroserwis wymaga zarządzania cyklem życia, wdrażania, konfiguracji oraz aktualizacji. W miarę wzrostu liczby usług złożoność zarządzania znacząco wzrasta. Mikroserwisy komunikują się ze sobą za pomocą sieci (najczęściej REST API lub komunikatów przez brokery wiadomości, np. RabbitMQ).

Zarządzanie komunikacją, szczególnie w przypadku błędów sieciowych, opóźnień czy przeciążeń, staje się bardziej wymagające. Dodatkowo, konieczność zapewnienia spójności danych w rozproszonym systemie mikroserwisów może prowadzić do wyzwań związanych z implementacją transakcji rozproszonych oraz zarządzaniem ich niezawodnością.

Choć mikroserwisy są niezależne, niektóre funkcje wymagają współpracy kilku usług. Koordynowanie tych zależności wymaga odpowiedniego projektowania i synchronizacji, aby uniknąć tzw. efektu domina w przypadku awarii jednej z usług. W architekturze mikroserwisów każdy komponent często posiada własną bazę danych. Choć to sprzyja niezależności, utrzymanie spójności danych staje się wyzwaniem, szczególnie w scenariuszach wymagających transakcji między różnymi usługami.

4 Porównanie obu architektur

Celem analizy jest zbadanie, jak poszczególne elementy systemu, w tym usługi (API) i cała infrastruktura, radzą sobie w warunkach obciążenia oraz jakie wyzwania pojawiają się podczas testów wydajnościowych [43]. Przeprowadzone testy obciążeniowe pozwalają ocenić zużycie zasobów, takich jak CPU, pamięć oraz sieć, dla poszczególnych komponentów systemu (Course API, Student API, Enrollment API) przy symulowanej liczbie użytkowników (ang. VU – Virtual Users)[13]. W szczególności skupiono się na:

- Identyfikacji wąskich gardeł i problemów wydajnościowych.
- Analizie zużycia zasobów w kontekście skalowalności systemu.
- Porównaniu zużycia CPU, pamięci oraz Network I/O dla poszczególnych usług.
- Oceny kosztów związanych z utrzymaniem zasobów dla wybranej architektury

4.1 Wydajność

W celu przeprowadzenia analizy wydajności obu architektur – monolitycznej oraz mikroserwisowej – aplikacje zostały skonteneryzowane przy użyciu Dockera. Skonteneryzowanie aplikacji pozwala na stworzenie izolowanych środowisk, które gwarantują spójność działania niezależnie od platformy, na której są uruchamiane. Dla obu przypadków został stworzony plik *docker-compose.yml* (Kod 25, Kod 26), który umożliwia zdefiniowanie, konfigurację oraz zarządzanie wielokontenerowymi środowiskami.

```
1.  services:
 2.    course.api:
 3.      image: lms-course-api
 4.      container_name: lms.course.api
 5.      build:
 6.        context: .
 7.        dockerfile: Course.API/Dockerfile
 8.      depends_on:
 9.        rabbitmq:
10.          condition: service_healthy
11.        sqlserver:
12.          condition: service_healthy
13.      networks:
14.        - lms-micro-network
15.
16.
```

Kod 25 Fragment docker-compose.yaml dla mikroserwisów

[13] https://medium.com/@novyludek/virtual-users-vs-rps-77627b384127," [Online]. [Data uzyskania dostępu: 12 12 2024

```
  lms-monolith:
    image: ${DOCKER_REGISTRY:-}lms-api
    container_name: lms.monolith
    build:
      context: .
      dockerfile: LMS.Monolith/Dockerfile
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ConnectionStrings__DefaultConnection=Server=sqlserver-
monolith;Database=LmsDb;User Id=sa;Password=LMS123!@#Strong;TrustServerCertificate=True
      - ASPNETCORE_URLS=http://+:80
    ports:
      - "5110:80"
    depends_on:
      - sqlserver-monolith
    networks:
      - lms-monolith-network

networks:
  lms-monolith-network:
    name: lms-monolith-network
    driver: bridge

```

Kod 26 Fragment docker-compose.yaml dla monolitu

Podczas testów wydajnościowych wykorzystano narzędzie *docker stats*, które jest wbudowanym narzędziem Dockera do monitorowania zasobów systemowych zużywanych przez działające kontenery.

Dzięki skonteneryzowaniu aplikacji i użyciu *docker-compose.yml* oraz *docker stats*, możliwe było stworzenie kontrolowanego środowiska testowego oraz zebranie szczegółowych statystyk wydajnościowych, które posłużyły do przeprowadzenia analizy porównawczej obu architektur.

Przedstawione w dalszej części wykresy obrazują zmiany wydajności systemu w czasie. Znajdują się na nich trzy główne parametry: użycie CPU (%), zużycie pamięci (MiB) oraz przepustowość sieci (Network I/O w MB/s). Skala osi X to czas, natomiast skala osi Y różni się w zależności od parametru: dla CPU jest to procentowe użycie procesora, dla pamięci jednostką są mebibajty (MiB), a dla sieci megabajty (MB). Wyniki przedstawione w tabelach będą prezentowane zgodnie z tymi jednostkami.

Warto zwrócić uwagę na różnicę między MiB (mebibajt) a MB (megabajt). Mebibajt (1 MiB = 1024^2 bajtów) to jednostka binarna, wykorzystywana głównie w systemach komputerowych, gdzie opiera się na potęgach liczby dwa. Natomiast megabajt (1 MB = 1000^2 bajtów) jest jednostką dziesiętną, stosowaną głównie w kontekście transmisji danych oraz zgodnie z międzynarodowym układem SI. Różnica ta, ma znaczenie, ponieważ 1 MiB odpowiada około 1,048 MB[14].

[14] https://www.computernetworkingnotes.com/networking-tutorials/megabytes-mb-v-s-mebibytes-mib-differences-explained.html

4.1.1 Wydajność mikroserwisów

W ramach przeprowadzonych badań, przetestowano wydajność mikroserwisów przy różnym obciążeniu użytkowników wirtualnych. Testy przeprowadzono dla trzech scenariuszy obciążenia.

Ogólna wydajność przy 2 VU

- Course Serwis (Tab 1, Rys. 5):
    - Największe zużycie zasobów (CPU i Network I/O).
    - CPU osiąga 33.8%, a transfer danych wynosi nawet 51.8 MB.
    - Sugeruje to, że Course API jest obciążone operacjami wymagającymi przetwarzania i przesyłu dużych ilości danych.
- Student Serwis (Rys. 6, Tab 2):
    - Zużycie CPU jest umiarkowane (4.56% średnio, maksymalnie 51.3%), a Network I/O wynosi średnio 8.52 MB.
    - Wzrost CPU do 51.3% wskazuje, że istnieje wrażliwość na większy ruch w krótkim czasie.
- Enrollment Serwis (Rys. 7, Tab 3):
    - CPU jest na stałym poziomie (4.78% średnio) z maksymalnym skokiem 32.68%, a transfer sieciowy jest zbliżony do Student API (8.52 MB).

Metryka	Średnia	Min	Max
CPU %	4.5592592592592585	0.42	51.3
Memory Usage (MiB)	235.73703703703708	110.0	319.1
Network I/O (MB)	8.516239872685185	0.16298828125	21.4

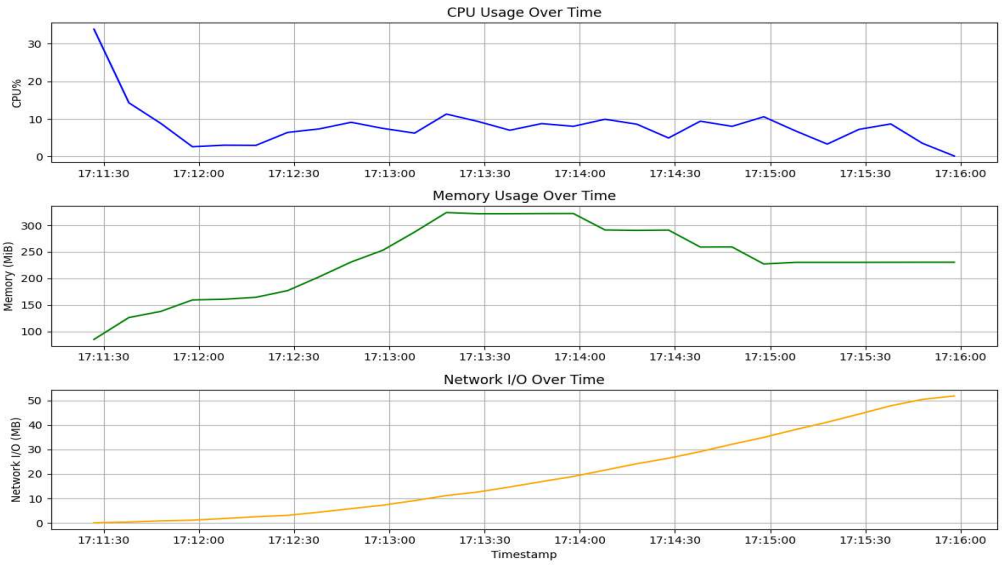
Tab 1 Tabela podsumowująca wyniki dla Course Serwisu dla 2VU

Metryka	Średnia	Min	Max
CPU %	4.5592592592592585	0.42	51.3
Memory Usage (MiB)	235.73703703703708	110.0	319.1
Network I/O (MB)	8.516239872685185	0.16298828125	21.4

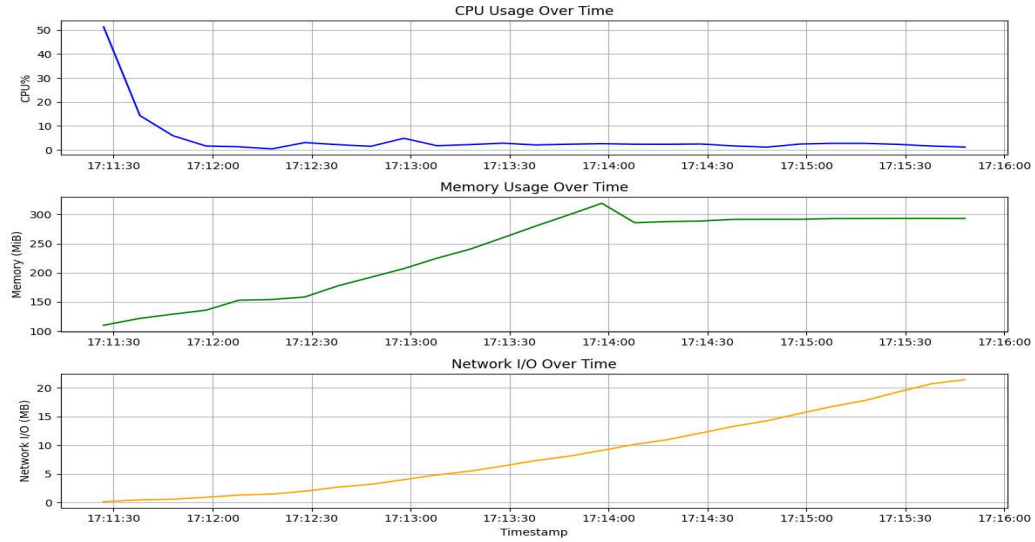
Tab 2 Tabela podsumowująca wyniki dla Course Serwisu dla 2VU

Metryka	Średnia	Min	Max
CPU %	4.780370370370371	0.93	32.68
Memory Usage (MiB)	220.58	107.4	309.3
Network I/O (MB)	8.52	0.12	21.94

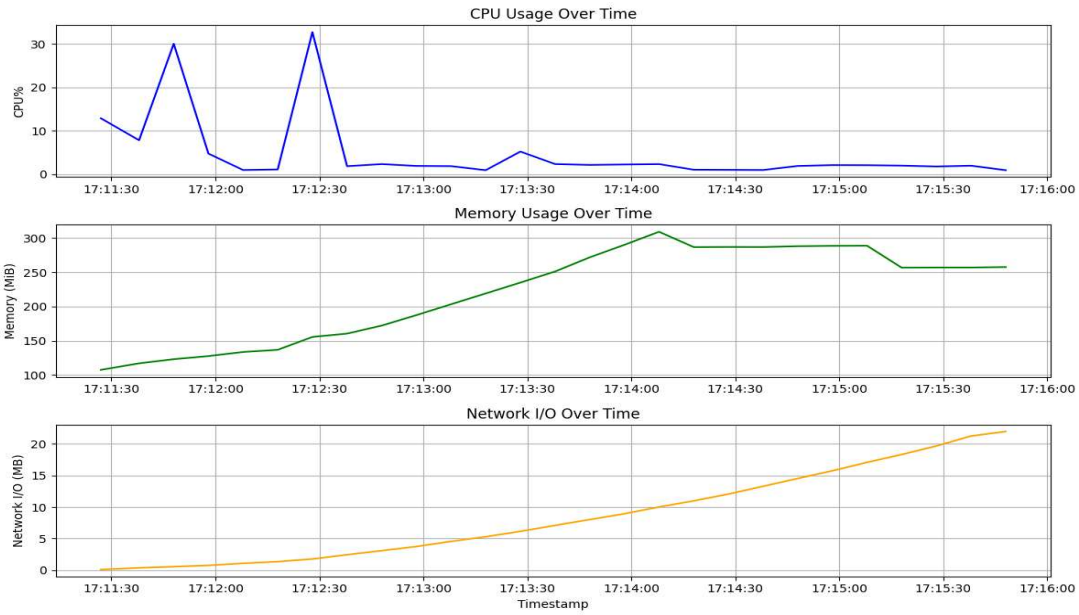
Tab 3 Tabela podsumowująca wyniki dla Enrollment Serwice dla 2VU



Rys. 5 Wykresy zużycia metryk na przestrzeni czasu testów dla Course Serwis 2VU



Rys. 6 Wykresy zużycia metryk na przestrzeni czasu testów dla Student Serwis 2VU



Rys. 7 Wykresy zużycia metryk na przestrzeni czasu dla testów dla Enrollment Service 2VU

Ogólna wydajność przy 100 VU:

- Course Serwis (Tab 4, Rys. 8):
  - Wysokie zużycie CPU: Średnie zużycie CPU na poziomie 130.16%, a maksymalne skoki sięgające 270.63%, wskazują na przeciążenie procesora w szczytowych momentach.
  - Intensywny transfer sieciowy osiąga maksymalnie 6512.64 MB, co sugeruje przesyłanie dużych ilości danych.
  - Wzrost zużycia pamięci: Średnie zużycie pamięci wynosi 807.29 MiB, a maksymalna wartość sięga 1576.96 MiB.
- Student Serwis (Rys. 9, Tab 5):
  - Stabilne zużycie CPU: Wzrost obciążenia procesora będzie proporcjonalny do liczby użytkowników, ale na podstawie wcześniejszych testów CPU jest mniej przeciążone niż w Course Serwis.
  - Umiarkowane zużycie pamięci: Student Serwis charakteryzuje się niższym zużyciem pamięci w porównaniu do Course Serwis. Średnie zużycie nie powinno przekraczać 1–1.5 GB przy 100 VU.
  - Niższy Network I/O: Transfer sieciowy jest mniejszy niż w Course API, co sugeruje mniej intensywną komunikację z klientami i mniejszy rozmiar przesyłanych danych.

- Enrollment Serwis (Rys. 10, Tab 6)
    - Stabilne obciążenie CPU: Zużycie procesora jest umiarkowane i stabilne, choć maksymalne wartości mogą sięgać 70-80% przy 100 VU. API jest mniej podatne na przeciążenie niż Course Serwis.
    - Efektywne zarządzanie pamięcią: Zużycie pamięci jest stabilne i oscyluje wokół średnich wartości, które nie powinny przekroczyć 1.5–2 GB przy 100 VU.
    - Zrównoważony Network I/O: Transfer sieciowy jest porównywalny ze Student Serwis, co wskazuje na umiarkowane zapotrzebowanie na przepustowość sieci i dobrze zoptymalizowane odpowiedzi.

Metryka	Średnia	Min	Max
CPU%	130.1603125	0.01	270.63
Memory Usage (MiB)	807.2910625	85.25	1576.96
Network I/O (MB)	2162.3246148681637	0.07265625	6512.639999999999

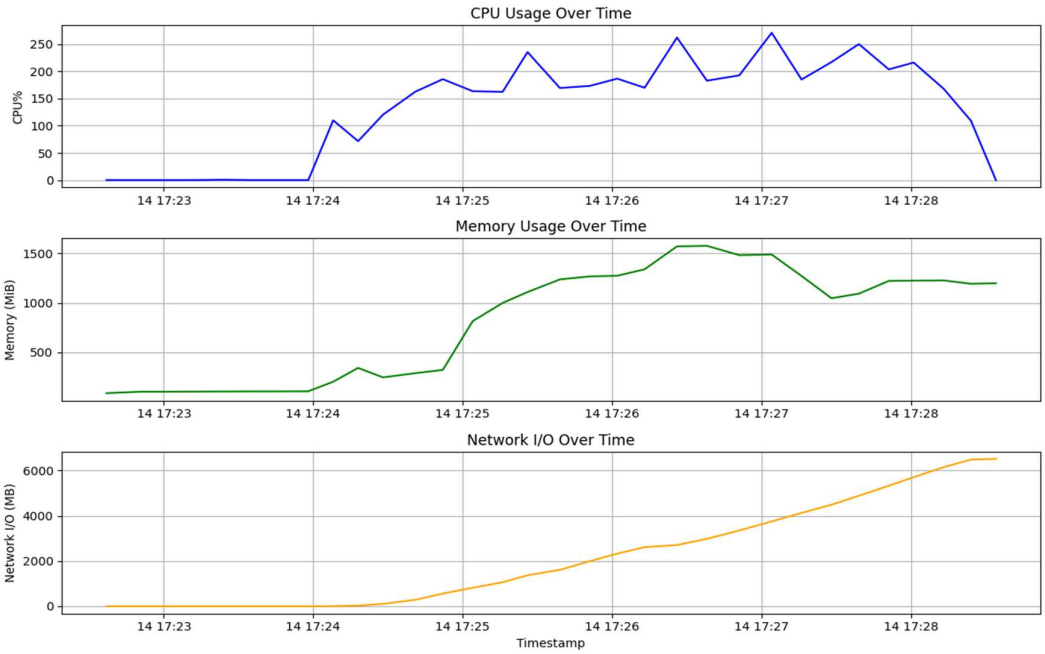
Tab 4 Tabela podsumowująca wyniki dla Course Serwice dla 100VU

Metryka	Średnia	Min	Max
CPU%	37.47064516129033	0.01	85.96
Memory Usage (MiB)	190.30935483870968	91.55	300.2
Network I/O (MB)	60.092318548387105	0.11025390625	149.2

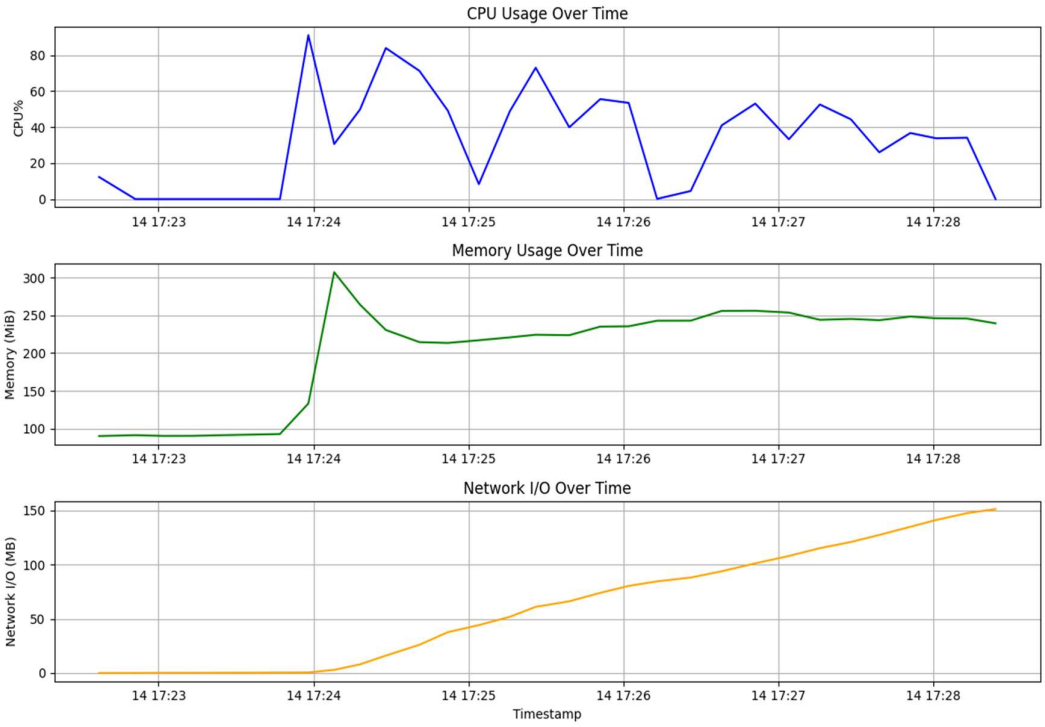
Tab 5 Tabela podsumowująca wyniki dla Student Serwice dla 100VU

Metryka	Średnia	Min	Max
CPU%	33.13387096774194	0.01	91.13
Memory Usage (MiB)	203.98903225806447	90.03	307.4
Network I/O (MB)	60.81540511592743	0.06826171875	151.2

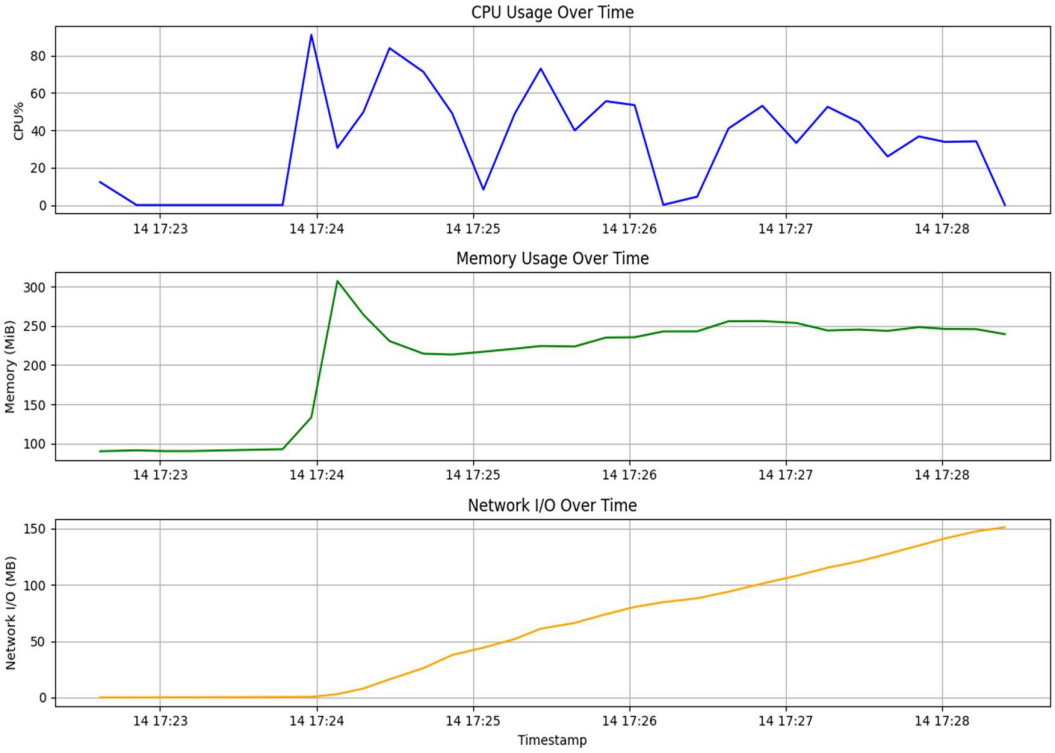
Tab 6 Tabela podsumowująca wyniki dla Enrollment Serwice dla 100VU



Rys. 8 Wykresy zużycia metryk na przestrzeni czasu testów dla Course Serwis 10OVU



Rys. 9 Wykresy zużycia metryk na przestrzeni czasu testów dla Student Serwis 100VU



Rys. 10 Wykresy zużycia metryk na przestrzeni czasu testów dla Enrollment Serwis 100VU

Ogólna wydajność przy 500 VU:

- Course Serwis (Rys. 11, Tab 7):
  - CPU: Średnie zużycie CPU wynosi 160.42%, a maksymalne osiąga 636.24%, co wskazuje na przeciążenie procesora i konieczność skalowania poziomego.
  - Pamięć: Średnie zużycie pamięci wynosi 933.03 MiB, a maksymalna wartość sięga 1609.73 MiB, co sugeruje potrzebę monitorowania i optymalizacji zarządzania pamięcią.
  - Network I/O: Średni transfer danych osiąga 3152.67 MB, a maksymalny 10.6 GB, co wskazuje na bardzo duży ruch sieciowy wymagający kompresji i optymalizacji payloadów.
- Student Serwis (Rys. 12, Tab 8):
  - CPU: Średnie zużycie CPU wynosi 41.42%, ale maksymalne skoki do 240.42% wskazują na intensywne momenty przetwarzania, co może prowadzić do opóźnień.
  - Pamięć: Użycie pamięci osiąga średnio 224.96 MiB, z maksymalnymi

wartościami na poziomie 484.8 MiB, co jest stabilne, ale wymaga monitorowania.

- Network I/O: Transfer danych wynosi średnio 64.54 MB, z maksymalnymi wartościami 167.4 MB, co wskazuje na umiarkowany ruch sieciowy.

- Enrollment Serwis (Rys. 13, Tab 9);
  - CPU: Średnie zużycie CPU wynosi 47.91%, a maksymalne skoki sięgają 386.47%, co sugeruje okresowe intensywne obciążenie procesora.
  - Pamięć: Średnie zużycie pamięci to 214.40 MiB, z maksymalnym zużyciem na poziomie 404.2 MiB, co wskazuje na stabilne zarządzanie zasobami.
  - Network I/O: Transfer danych osiąga średnio 63.38 MB, a maksymalna wartość to 165.3 MB, co sugeruje przewidywalne obciążenie sieciowe.

Metryka	Średnia	Min	Max
CPU%	160.41642857142858	0.02	636.24
Memory Usage (MiB)	933.0260476190477	85.05	1609.728
Network I/O (MB)	3152.668628162202	0.069921875	10598.4

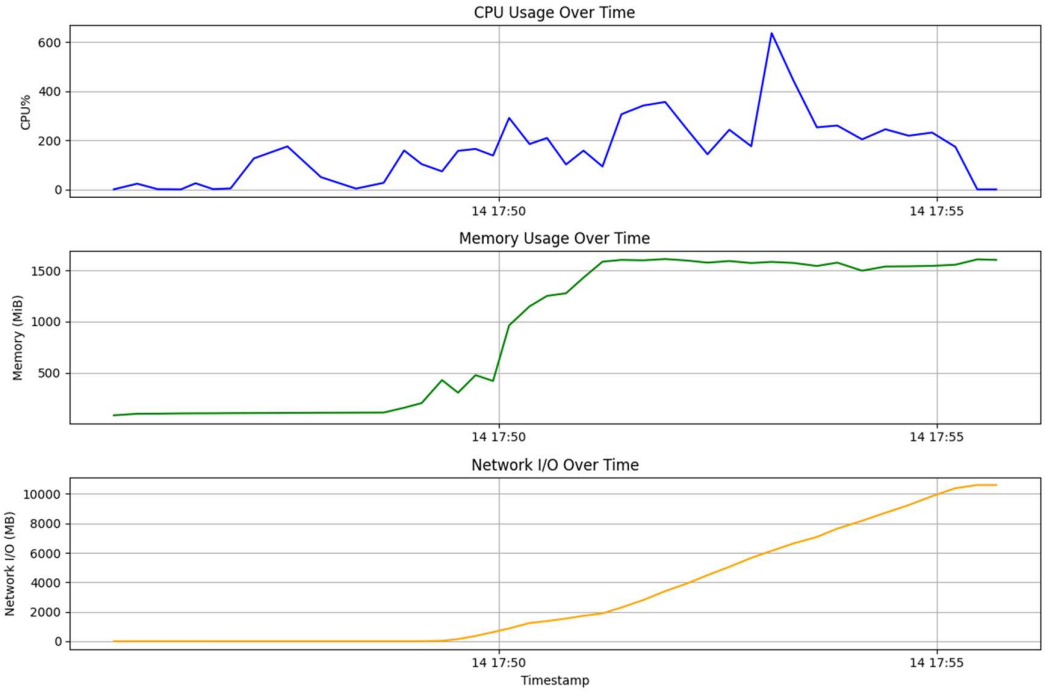
Tab 7 Tabela podsumowująca wyniki dla Course Serwice dla 500VU

Metryka	Średnia	Min	Max
CPU%	41.42309523809523	0.01	240.42
Memory Usage (MiB)	224.96190476190475	79.72	484.8
Network I/O (MB)	64.54400623139881	0.0904296875	167.4

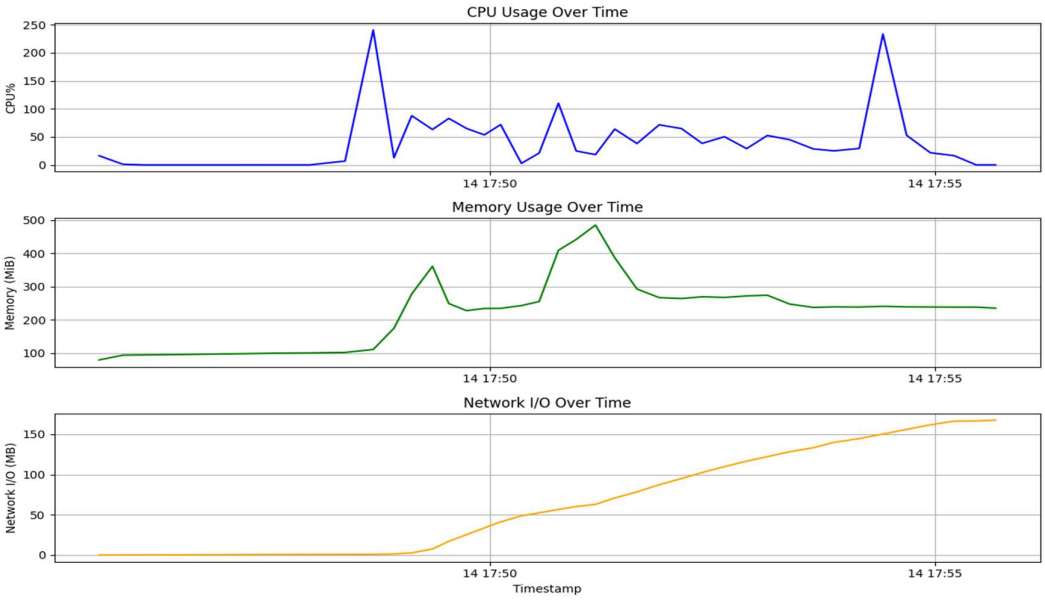
Tab 8 Tabela podsumowująca wyniki dla Student Serwice dla 500VU

Metryka	Średnia	Min	Max
CPU%	47.91309523809524	0.01	386.47
Memory Usage (MiB)	214.3983333333333	88.03	404.2
Network I/O (MB)	63.38004882812499	0.08525390625	165.3

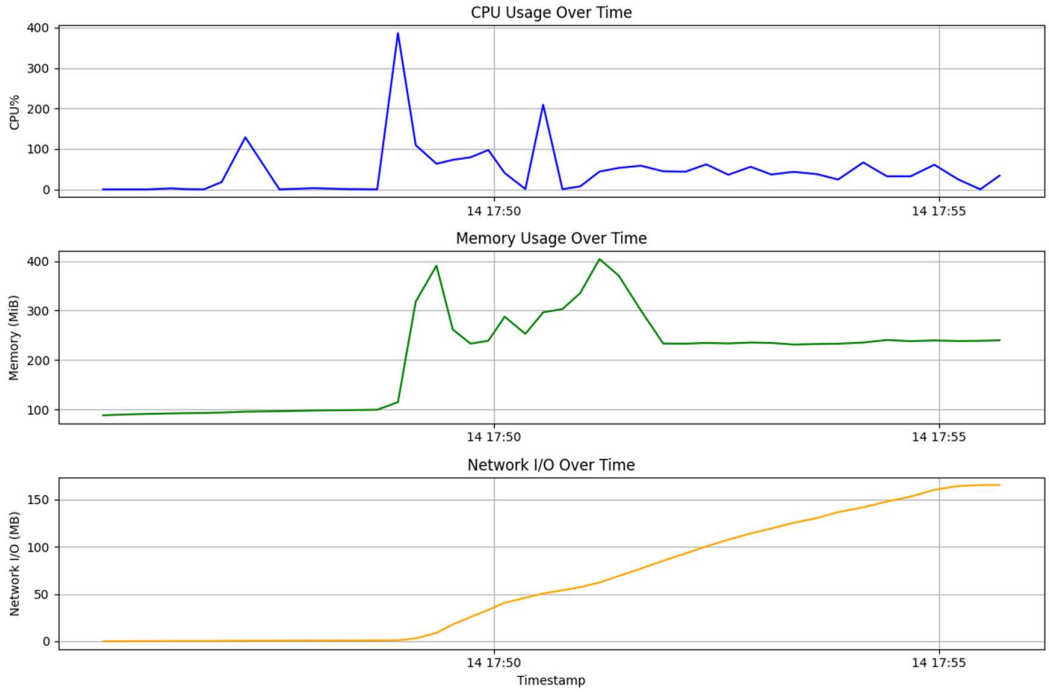
Tab 9 Tabela podsumowująca wyniki dla Enrollment Serwice dla 500VU



Rys. 11 Wykresy zużycia metryk na przestrzeni czasu testów dla Course Serwis 50OVU



Rys. 12 Wykresy zużycia metryk na przestrzeni czasu testów dla Student Serwis 50OVU



Rys. 13 Wykresy zużycia metryk na przestrzeni czasu testów dla Enrollment Serwis 50OVU

4.1.2 Wydajność monolitu

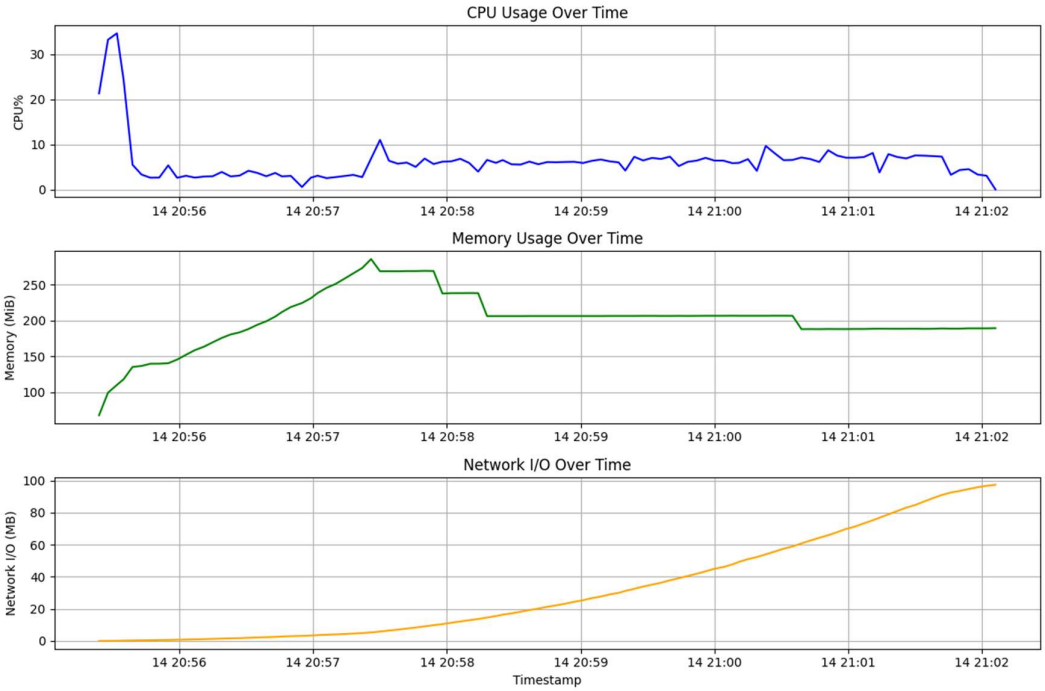
W ramach badań przeprowadzono również testy wydajnościowe dla architektury monolitycznej. Celem było porównanie jej zachowania z mikroserwisami przy różnych poziomach obciążenia. W badaniach uwzględniono te same scenariusze obciążenia, co w przypadku mikroserwisów.

Ogólna Wydajność przy 2 VU

Średnie obciążenie procesora wynosi 6.28%, co sugeruje, że monolit nie jest intensywnie obciążony przy tak małej liczbie użytkowników. Maksymalne skoki CPU do poziomu 34.61% wskazują na momenty większej aktywności obliczeniowej. Średnie zużycie pamięci wynosi 201.24 MiB, natomiast wartość maksymalna osiągnęła 285.4 MiB. Wzrost ten sugeruje, że aplikacja alokuje coraz więcej zasobów w trakcie działania, co może być efektem przetwarzania żądań lub krótkotrwałych operacji pamięciowych. Średni transfer danych wynosi 31.79 MB, a wartość maksymalna sięga 97.3 MB. Obciążenie sieci stopniowo rośnie w miarę trwania testu, co wskazuje na stopniowe przetwarzanie i przesyłanie danych. Przy małym ruchu jest to akceptowalne. (Rys. 14, Tab 10).

Metryka	Średnia	Min	Max
CPU%	6.278155339805826	0.01	34.61
Memory Usage (MiB)	201.23572815533984	67.91	285.4
Network I/O (MB)	31.787575280643207	0.030664062500000002	97.3

Tab 10 Tabela podsumowująca wyniki monolitu dla 2VU



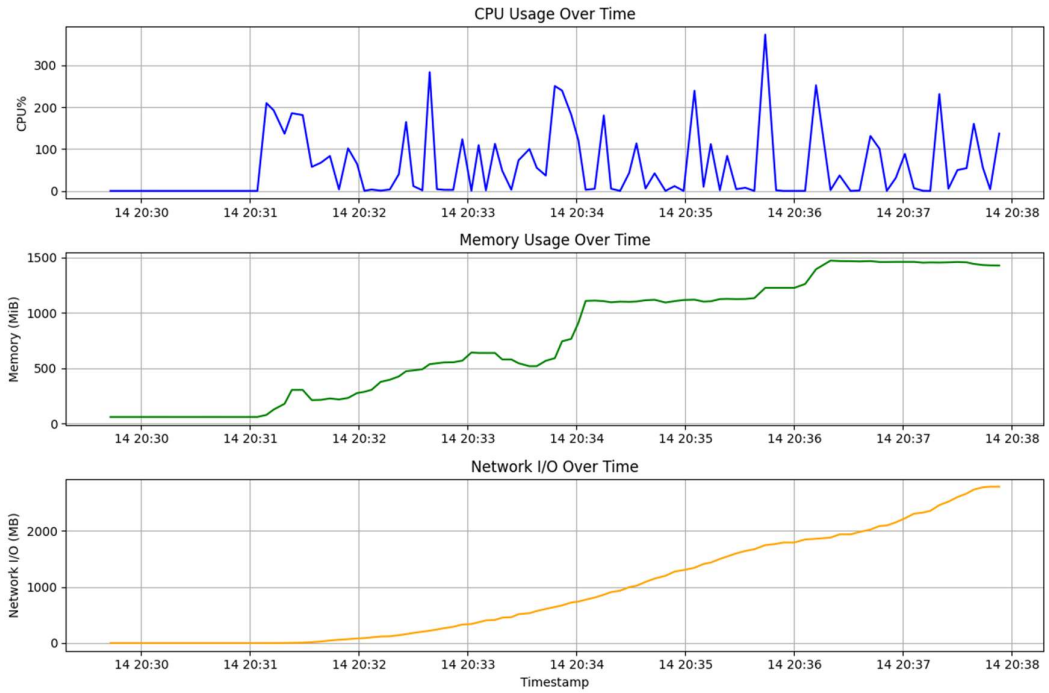
Rys. 14 Wykresy zużycia metryk na przestrzeni czasu testów dla aplikacji monolitowej dla 2VU

Ogólna Wydajność przy 100 VU

Średnie zużycie CPU na poziomie 57.95% sugeruje, że procesor jest już znacznie obciążony przy obecnym ruchu. Skoki CPU do wartości maksymalnej 373.78% wskazują na intensywne operacje obliczeniowe w określonych momentach, prawdopodobnie związane z większymi żądaniami lub niewydajnym przetwarzaniem. Średnie zużycie pamięci wynosi 764.37 MiB, a maksymalne osiąga aż 1471.49 MiB. To ponad siedmiokrotny wzrost względem minimalnego zużycia (61.27 MiB) i sugeruje dynamiczną alokację pamięci w trakcie przetwarzania żądań. Średni transfer danych wynosi 911.10 MB, a maksymalna wartość sięga 2785.28 MB, co jest bardzo wysokim obciążeniem sieci jak na 100 użytkowników. Wzrost transferu sugeruje, że aplikacja przesyła duże payloady lub przetwarza żądania, które wymagają intensywnej komunikacji. (Rys. 15, Tab 11)

Metryka	Średnia	Min	Max
CPU%	57.950693069306915	0.01	373.78
Memory Usage (MiB)	764.3734257425741	61.27	1471.488
Network I/O (MB)	911.0996691290222	0.02890625	2785.2799999999997

Tab 11 Tabela podsumowująca wyniki monolitu dla 100VU



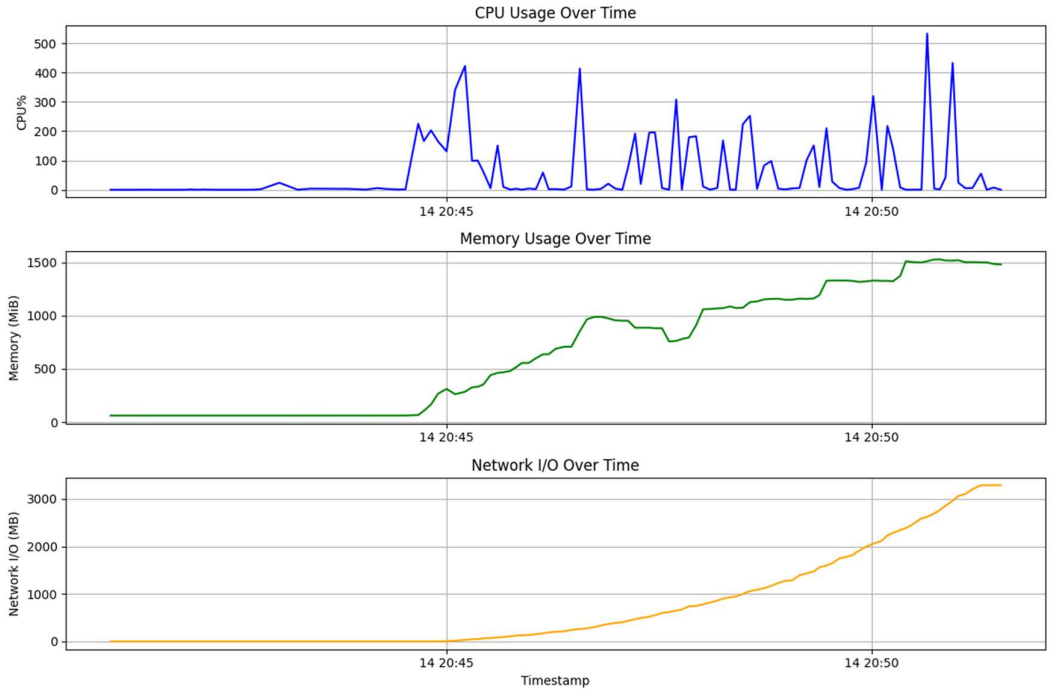
Rys. 15 Wykresy zużycia metryk na przestrzeni czasu testów dla aplikacji monolitowej dla 100VU

Ogólna Wydajność przy 500 VU

Średnia wartość CPU wynosi 61.11%, co wskazuje na bardzo intensywną pracę procesora. Maksymalne skoki CPU osiągają 533.57%, co świadczy o przeciążeniu systemu i braku efektywnego rozłożenia obciążenia. Takie wartości mogą prowadzić do opóźnień w odpowiedziach oraz zwiększonego ryzyka niestabilności systemu przy długotrwałym obciążeniu. Średnia wartość pamięci wynosi 729.08 MiB, a maksymalne zużycie sięga 1527.80 MiB. Wysokie zużycie pamięci wskazuje na dużą alokację zasobów, co może prowadzić do przekroczenia limitów przy dalszym wzroście obciążenia. Zarządzanie pamięcią wymaga monitorowania pod kątem wycieków oraz optymalizacji alokacji zasobów. Średni transfer danych wynosi 829.01 MB, a maksymalne wartości osiągają 3287.04 MB. Wysoki poziom ruchu sieciowego świadczy o dużej liczbie żądań oraz przesyłaniu dużych ilości danych.(Tab 12, Rys. 16)

Metryka	Średnia	Min	Max
CPU%	61.11134453781513	0.01	533.57
Memory Usage (MiB)	729.0815630252099	61.46	1527.808
Network I/O (MB)	829.0122442883401	0.026074218750000003	3287.04

Tab 12 Tabela podsumowująca wyniki monolitu dla 500VU



Rys. 16 Wykresy zużycia metryk na przestrzeni czasu testów dla aplikacji monolitowej dla 500VU

4.1.3 Wnioski

Przeprowadzone testy wydajnościowe dla mikroserwisów i monolitu przy różnych poziomach obciążenia (od 2 VU do 500 VU) ukazują istotne różnice w zarządzaniu zasobami, skalowalności oraz ogólnej stabilności systemu.

Obciążenie CPU

Monolit wykazuje stopniowy wzrost obciążenia CPU w miarę zwiększania liczby użytkowników. Przy 500 VU średnie zużycie procesora wyniosło 61.11%, a maksymalne skoki osiągnęły 533.57%, co wskazuje na brak izolacji operacji oraz przeciążenie w szczytowych momentach. CPU staje się wąskim gardłem, co wpływa na całą aplikację.

W architekturze mikroserwisowej obciążenie CPU jest bardziej rozproszone pomiędzy poszczególne usługi.

- Course API wykazuje największe obciążenie (do 636% przy 500 VU), co

sugeruje, że jest to usługa stanowiąca główny punkt przeciążenia.

- Student API i Enrollment API pozostają stabilniejsze, z mniejszym obciążeniem CPU na poziomie 50-70%.Rozdzielenie obciążenia między usługi pozwala lepiej zarządzać zasobami i skalować tylko te komponenty, które tego wymagają.

Mikroserwisy oferują lepszą elastyczność w zarządzaniu obciążeniem CPU dzięki izolacji usług, podczas gdy monolit cierpi z powodu skumulowanego obciążenia.

Zużycie Pamięci

Monolit zużywa znaczną ilość pamięci przy wzroście liczby użytkowników. Przy 500 VU średnie zużycie pamięci wynosi 729.08 MiB, a maksymalne osiąga 1527.80 MiB. Duża alokacja zasobów może prowadzić do ryzyka przekroczenia limitów i niestabilności systemu. W mikroserwisach zużycie pamięci jest bardziej równomiernie rozłożone między poszczególne usługi:

- Course API: Największe zużycie pamięci przy większym obciążeniu (1.6 GB przy 500 VU).
- Student API i Enrollment API zużywają średnio 700–800 MiB. Dzięki izolacji usług wzrost obciążenia jednej usługi nie wpływa bezpośrednio na inne komponenty systemu.

Mikroserwisy zapewniają większą kontrolę nad zasobami pamięci, pozwalając na skalowanie pamięci tylko dla konkretnych, najbardziej obciążonych usług. Monolit natomiast zużywa zasoby jednorazowo dla całego systemu, co oznacza, że nawet jeśli tylko jedna część aplikacji wymaga większej ilości pamięci lub mocy obliczeniowej, konieczne jest skalowanie całego systemu

Obciążenie Sieci

Monolit generuje wysoki transfer danych, co jest wynikiem obsługi dużej liczby żądań oraz braku optymalizacji komunikacji. Przy 500 VU średni transfer wynosi 829 MB, a maksymalny sięga 3287 MB. Brak separacji prowadzi do przeciążenia sieci, zwłaszcza gdy aplikacja przetwarza duże payloady. W architekturze mikroserwisowej ruch sieciowy jest podzielony pomiędzy usługi co pozwala na bardziej efektywne zarządzanie przepustowością oraz ogranicza ryzyko wystąpienia wąskich gardeł w komunikacji. W komponentach mononlitu obciążenie wygląda następująco:

- Course API generuje największy ruch (do 10 GB) przy dużym obciążeniu, co sugeruje intensywne przetwarzanie danych.
- Student API i Enrollment API utrzymują umiarkowane wartości (2–3 GB). Dzięki podziałowi ruchu sieciowego poszczególne usługi można optymalizować niezależnie, co zmniejsza ryzyko przeciążenia całego systemu.

Mikroserwisy lepiej zarządzają ruchem sieciowym poprzez podział komunikacji między różne usługi, podczas gdy monolit generuje skumulowane obciążenie sieciowe. Z tych danych można wywnioskować, że mikroserwisy są znacznie bardziej skalowalne, ponieważ obciążenie CPU, pamięci i sieci jest rozdzielone między poszczególne usługi. W monolicie wzrost obciążenia wpływa na cały system jednocześnie, co prowadzi do szybkiego przeciążenia zasobów. Mikroserwisy, zdecyduowanie oferują większą elastyczność w zarządzaniu zasobami, pozwalając na optymalizację i skalowanie wyłącznie tych komponentów, które tego wymagają. W monolicie zasoby są współdzielone, co utrudnia kontrolę nad ich wykorzystaniem. Należy też zauważyć, że monolit przy wyższych poziomach obciążenia staje się niestabilny z powodu skumulowanego zużycia CPU, pamięci i sieci. Mikroserwisy, dzięki izolacji, zachowują większą stabilność i odporność na przeciążenia.

4.2 Koszt wdrożenia aplikacji w chmurze Azure

Z pomocą narzędzia Azure Pricing Calculator[15] wyestymowane zostały koszty wdrożenia aplikacji w chmurze. Aby to zrobić należało przeprowadzić analizę potrzebnych zasobów oraz zrozumieć różnice pomiędzy architekturami.

Wspólne serwisy zostały wybrane jako kluczowe elementy infrastruktury, aby zapewnić podstawowe działanie aplikacji. Obejmują one:

- Azure App Service na poziomie Basic Tier (B1) z 1 rdzeniem, 1,75 GB RAM i 10 GB Storage przy pełnym obciążeniu 730 godzin miesięcznie.
- Azure SQL Database w modelu DTU[16] Purchase Model na poziomie Basic Tier (B), z 5 DTUs oraz 2 GB wbudowanej pamięci dla pojedynczej bazy danych działającej przez 730 godzin miesięcznie.

W przypadku architektury mikroserwisowej, oprócz powyższych serwisów, dodano również Service Bus – usługę komunikacyjną umożliwiającą wymianę wiadomości między niezależnymi mikroserwisami. Usługa ta zapewnia niezawodną komunikację i usprawnia przepływ danych pomiędzy komponentami aplikacji, co jest kluczowe dla aplikacji opartej na mikroserwisach. Podobnie jak w przypadku innych usług, wybrano Basic Tier, aby zoptymalizować koszty.

Monolit

W przypadku aplikacji monolitycznej, aplikacja rozwijana i wdrażana jest jako jedność, co prowadzi do mniejszej liczby zasobów oraz prostszej konfiguracji infrastruktury.

Kategoria Serwisu	**Rodzaj Serwisu**	**Region**	**Opis**	**Koszt**
Przetwarzanie	App Service	Zachodnie Stany Zjednoczone	Basic Tier; 1 B1	€52,23
Bazy danych	Azure SQL Database	Wschodnie Stany Zjednoczone	Single Database, DTU Purchase Model	€4,67
		Razem		**€56,90**

Tab 13 Tabela cen wdrożenia aplikacji monolitycznej na platformę Azure

Mikroserwisy

[15] „https://azure.microsoft.com/en-us/pricing/calculator/," [Online]. [Data uzyskania dostępu: 10 12 2024].
[16] DTU – Jednostka transakcji bazy danych. Reprezentuje połączoną miarę procesora, pamięci, odczytów i zapisów.

Architektura mikroserwisów zakłada podział aplikacji na niezależne, mniejsze usługi, które mogą działać autonomicznie. Wymaga to rozbudowania infrastruktury, ponieważ każdy Mikroserwis musi być hostowany oraz zarządzany osobno. Wpływa to na wzrost kosztów związany z wykorzystaniem wielu instancji maszyn wirtualnych, kontenerów oraz wszelkich innych usług.

Kategoria Serwisu	**Rodzaj Serwisu**	**Region**	**Opis**	**Koszt**
Przetwarzanie	App Service	Zachodnie Stany Zjednoczone	Basic Tier; 1 B1	€52,23
Przetwarzanie	App Service	Zachodnie Stany Zjednoczone	Basic Tier; 1 B1	€52,23
Przetwarzanie	App Service	Zachodnie Stany Zjednoczone	Basic Tier; 1 B1	€52,23
Przetwarzanie	App Service	Zachodnie Stany Zjednoczone	Basic Tier; 1 B1	€52,23
Bazy danych	Azure SQL Database	Wschodnie Stany Zjednoczone	Single Database, DTU Purchase Model, Basic Tier, B:	€4,67
Bazy danych	Azure SQL Database	Wschodnie Stany Zjednoczone	Single Database, DTU Purchase Model, Basic Tier, B:	€4,67
Bazy danych	Azure SQL Database	Wschodnie Stany Zjednoczone	Single Database, DTU Purchase Model, Basic Tier, B	€4,67
Integracja	Service Bus	Wschodnie Stany Zjednoczone	Basic tier: 10 million messaging operations	€0.48
		Razem		**€ 223.42**

Tab 14 Tabela cen wdrożenia aplikacji mikroserwisowej na platforme Azure

Podsumowanie

Wdrożenie aplikacji monolitycznej charakteryzuje się znacznie niższymi kosztami wdrożenia w wysokości €56,90 miesięcznie (Tab 13). Wynika to z dużo prostszej infrastruktury oraz ograniczonej liczby wymaganych zasobów. Aplikacja mikroserwisowa generuje znacznie wyższe koszty - €223,42 miesięcznie (Tab 14).

Wzrost ten wynika z konieczności wykorzystania wielu instancji Azure App Service oraz Azure SQL Database, a także dodatkowych usług, takich jak Service Bus, które umożliwiają komunikację pomiędzy mikroserwisami.

Koszt wdrożenia jednego mikroserwisu został oszacowany na €57,02 miesięcznie. Obejmuje on:

- Azure App Service: €52,23
- Azure SQL Database: €4,67
- Proporcjonalny koszt Service Bus: €0,12

W związku z tym, wdrożenie każdego kolejnego mikroserwisu będzie najprawdopodobniej zwiększać miesięczne koszty o około €57, co w dłuższym okresie czasu znacząco wpłynie na koszt projektu.

5 Wnioski

Celem niniejszej pracy było przeprowadzenie analizy porównawczej architektury monolitycznej oraz mikroserwisowej w kontekście ich odpowiedniości w różnych scenariuszach wdrożenia aplikacji. Badania i testy zostały zrealizowane na jednej aplikacji, co pozwoliło na szczegółowe zbadanie wydajności oraz utylizacji zasobów dla obu podejść architektonicznych przy różnym poziomie obciążenia systemu.

Wyniki testów jednoznacznie pokazały, że architektura mikroserwisowa lepiej radzi sobie w kontekście skalowalności i wydajności, szczególnie w środowiskach o dużym obciążeniu. Podział aplikacji na niezależne usługi pozwolił na lepsze rozłożenie obciążenia, umożliwiając efektywniejsze zarządzanie zasobami CPU, pamięci oraz sieci. Dzięki temu każda z usług mogła być optymalizowana niezależnie, co minimalizowało ryzyko przeciążenia całego systemu. W przypadku architektury monolitycznej, testy wykazały, że wraz ze wzrostem liczby użytkowników, obciążenie zasobów, takich jak CPU, pamięć i sieć, staje się skumulowane, prowadząc do przeciążenia systemu i ograniczenia jego wydajności. Monolit okazał się bardziej odpowiedni dla mniejszego obciążenia, gdzie prosta struktura i łatwość zarządzania stanowią główne zalety tego podejścia.

Choć testy zostały przeprowadzone na jednej, niezłożonej aplikacji, pozwoliły one na wyciągnięcie istotnych wniosków dotyczących ogólnych różnic pomiędzy monolitem a mikroserwisami. Monolit okazał się prostszy i szybszy do wdrożenia oraz zarządzania w przypadku małych projektów, które nie posiadają skomplikowanej logiki biznesowej i nie wymagają wysokiej skalowalności. W takich sytuacjach monolit stanowi efektywne rozwiązanie ze względu na niższy poziom złożoności, co pozwala na szybszy rozwój i wdrożenie aplikacji. Jednakże w przypadku dużych i złożonych systemów przewaga mikroserwisów jest wyraźna. Architektura mikroserwisowa oferuje lepszą wydajność, skalowalność oraz elastyczność, szczególnie pod względem pracy wielu zespołów nad różnymi częściami aplikacji jednocześnie. Dzięki izolacji poszczególnych usług możliwe jest nie tylko efektywne zarządzanie zasobami, ale także niezależne rozwijanie, wdrażanie i optymalizowanie poszczególnych komponentów systemu.

Bibliografia

[1] N. F. Mark Richards, Fundamentals of Software Architecture: An Engineering Approach, 2020.

[2] S. K. M. N. Mayank Mishra, Cracking the monolith: challenges in data transitioning to cloud native architectures.

[3] S. A. T. H. P. M. T.-O. A. f. D. Architectures, Neal Ford, Mark Richards, Pramod Sadalage, Zhamak Dehghani, 2021.

[4] G. Blokdyk, Monolithic application The Ultimate Step-By-Step Guide, 2021.

[5] C. Richardson, Microservices Pattern ith examples in Java.

[6] V. Vernon, Implementing Domain-Driven Design.

[7] V. V. T. Jaskula, Strategic Monoliths and Microservices: Driving Innovation Using Purposeful Architecture.

[8] T. Gospodarek, „Systemy ERP. Modelowanie, projektowanie, wdrażanie," 2015.

[9] F. Wan, „Modelling Distributed Systems".

[10] B. Brendan, Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services.

[11] R. Flygare i A. Holmqvist, „Performance characteristics between monolithic and microservice-based systems".

[12] S. K. M. Karl, Docker. Niezawodne kontenery produkcyjne. Praktyczne zastosowania, 2024.

[13] O. Al-Debagy i P. Martinek, „A Comparative Review of Microservices and Monolithic Architectures".

[14] A. O. A. P. G. Blinowski, „https://www.semanticscholar.org/paper/Monolithic-vs.-Microservice-Architecture%3A-A-and-Blinowski-Ojdowska/31a9d5d7286b24d5d2a99af005dca7a814640aec," w *Monolithic vs. Microservice Architecture:*.

[15] W. Z. Konrad Gos, „The Comparison of Microservice and Monolithic Architecture," w *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, 2020.

[16] R. C. Martin, „Czysty kod. Podręcznik dobrego programisty".

[17] R. C. Martin, „Czysta architektura".

[18] B. B. T. G. Kathy Sierra, Head First Java: A Brain-Friendly Guide, 2022.

[19] L. Ramalho, Fluent Python: Clear, Concise, and Effective Programming, 2022.

[20] Z. Matt, PHP 8. Obiekty, wzorce, narzędzia, 2024.

[21] P. M. J, C# 12 i .NET 8 dla programistów aplikacji wieloplatformowych. Twórz aplikacje, witryny WWW oraz serwisy sieciowe za pomocą ASP.NET Core 8, Blazor i EF Core 8, 2024.

[22] J. Goldberg, Poznaj TypeScript, 2022.

[23] J. Read, Wzorce komunikacji. Przewodnik dla programistów i architektów, 2024.

[24] C. Gammelgaard, Microservices in .NET, Second Edition, 2022.

[25] H. R. Ribeiro, Vue.js 3 Cookbook, 2020.

[26] B.-G. Itzik, Podstawy języka T-SQL. Microsoft SQL Server 2022 i Azure SQL Database.

[27] P. Adam, MS SQL Server. Zaawansowane metody programowania. Wydanie II, 2021.

[28] D. Korotkevitch, SQL Server. Zaawansowane techniki rozwiązywania problemów i

poprawiania wydajności.

[29] P. Morlion, Ultimate Microservices with RabbitMQ.

[30] B.-M. R. S. M. D. A. K., Docker dla programistów. Rozwijanie aplikacji i narzędzia ciągłego dostarczania DevOps, 2021.

[31] E. Gkatziouras, A Developer's Essential Guide to Docker Compose: Simplify the development and orchestration of multi-container applications, 2022.

[32] A. R. H. S. Pethuru Raj, Architectural Patterns.

[33] B. W. M. R. Cesar de la Torre, .NET Microservices: Architecture for Containerized .NET Applications.

[34] Ś. M. K. Rybiński, Inżynieria oprogramowania w praktyce. Od wymagań do kodu z językiem UML.

[35] N. S. Saurabh Shrivastava, Podręcznik architekta rozwiązań. Wydanie II, 2023.

[36] N. T. Scott Millett, Patterns, Principles, and Practices of Domain-Driven Design, 2015.

[37] D. Farley, Nowoczesna inżynieria oprogramowania. Stosowanie skutecznych technik szybszego rozwoju oprogramowania wyższej jakości, 2023.

[38] J. P. Smith, Entity Framework Core in Action, Second Edition, 2021.

[39] S. Newman, „Building Microservices: Designing Fine-Grained Systems".

[40] N. D. S. D. S. T. L. a. M. M. ". M. t. M. A. E. R. f. t. B. D. A. Bucchiarone, „From Monolithic to Microservices: An Experience Report from the Banking Domain," *IEEE Software,* Tomy %1 z %2vol. 35,, nr no. 3, pp. 50-55, May/June 2018.

[41] M. R. P. S. Z. D. Neal Ford, Software Architecture: The Hard Parts: Modern Trade-Off Analyses for Distributed Architectures, 2021.

[42] B. Adam, Building Event-Driven Microservices. Leveraging Organizational Data at Scale.

[43] G. Mohan, Testowanie full stack. Praktyczny przewodnik dostarczania oprogramowania wysokiej jakości, 2023.

Wyrażam zgodę na udostępnienie mojej pracy w czytelniach Biblioteki SGGW w tym w Archiwum Prac Dyplomowych SGGW

.....

(czytelny podpis autora pracy)