JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 25 (2022) 393–400
Received: 5 September 2022
Accepted: 29 September 2022

Analysis of selected features of application based on monolithic and microservice architecture

Analiza wybranych cech aplikacji opartych na architekturze monolitycznej i mikrousługowej

Kamil Jaskot*, Sławomir Przyłucki

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

Abstract
The article describes the performance of applications built in monolithic and microservice architectures. The base of research includes application supporting prescription management developed with the use of Spring Framework technology and implemented in the Docker Swarm test environment. The tested applications were subjected to various loads in the form of sending HTTP requests that simulated user behaviour. The research has proven that an application created based on microservices architecture offers better traffic handling in case of high load. Scaling a microservice application allows for greater gains in performance measured as quantity served client requests per unit of time than scaling a monolithic application under the same conditions scaling.

Keywords: microservices; monolith-software architecture; scaling services; spring framework

Streszczenie
Artykuł przedstawia porównanie wydajności aplikacji utworzonych w architekturze monolitycznej i mikrousługowej. Zakres badań obejmuje aplikacje wspomagające zarządzanie receptami, utworzone przy wykorzystaniu technologii Spring Framework i wdrożone w środowisku testowym Docker Swarm. Aplikacje poddano różnym obciążeniom w postaci wysyłania zapytań HTTP, które symulowały zachowanie użytkowników. Przeprowadzone badania dowiodły, że aplikacja utworzona w oparciu o architekturę mikrousług lepiej radzi sobie z obsługą ruchu w przypadku dużego obciążenia. Skalowanie aplikacji mikrousługowej pozwala na uzyskanie większego przyrostu wydajności mierzonej jako liczba obsłużonych żądań klientów w jednostce czasu niż skalowanie aplikacji monolitycznej przy tych samych warunkach skalowania.

Słowa kluczowe: mikrousługi; architektura monolityczna oprogramowania; skalowanie usług; spring framework

*Corresponding author
Email address: kamil.jaskot@pollub.edu.pl (K. Jaskot)

1. Wstęp

Wiele istniejących już aplikacji często jest zaimplementowanych przy wykorzystaniu architektury monolitycznej. Stale rosnący kod aplikacji wprowadza większą złożoność i skomplikowanie co ostatecznie doprowadza do ujawnienia technicznych i organizacyjnych ograniczeń aplikacji. Często takie aplikacje stają się ciężkie w utrzymaniu i za duże aby mógł je w pełni zrozumieć jakikolwiek programista. W rezultacie naprawienie błędów i wdrażanie nowych funkcjonalności jest coraz bardziej trudne i czasochłonne. Rozwiązaniem pozwalającym rozwiązać te ograniczenia jest przebudowanie aplikacji do architektury wykorzystującej mikrousługi. W ogólnym ujęciu, każda aplikacja posiada dwie kategorie wymagań. Pierwsza obejmuje wymagania funkcjonalne określające co aplikacja ma robić i zazwyczaj architektura ma niewiele z nią wspólnego. Można zaimplementować zestaw przypadków w niemal dowolnej architekturze. Architektura ma znaczenie ponieważ umożliwia aplikacji spełnienie drugiej kategorii wymagań dotyczących jakości usług [1].

Przetwarzaniem w chmurze nazywamy model, który umożliwia dostęp na żądanie do zasobów obliczeniowych takich jak m.in. przestrzeń dyskowa, serwery, aplikacje czy usługi [2]. Coraz więcej firm decyduje się na wykorzystanie technologii chmurowych ze względu na ich skalowalność i elastyczność kosztów widząc w nich strategię biznesową, która pozwala na konkurencję z innymi firmami i realizację celów biznesowych [3, 4]. Wykorzystanie zalet oferowanych przez rozwiązania chmurowe i DevOps (ang. Development And Operations) wymaga aby podczas budowania aplikacja spełniała ich założenia. Niezależne wdrażanie wymaga posiadania systemu składającego się z modułów, które można niezależnie od siebie zaprojektować, zaimplementować, przetestować i wdrożyć. Dlatego w ostatnich latach architektura mikrousług stała się niezbędnym elementem rozwoju aplikacji wdrażanych w środowiskach chmurowych [5, 6].

Nie ma złotego środka dlatego bardzo ważnym jest aby odpowiednio dobrać architekturę do konkretnej aplikacji biorąc pod uwagę to co chcemy osiągnąć. Niekiedy dekompozycja aplikacji monolitycznej w mikrousługową może przynieść odwrotne skutki do zamierzonych. W tym artykule zostanie przedstawiona analiza porównawcza cech funkcjonalnych architektury monolitycznej i mikrousługowej na przykładzie aplikacji wykorzystującej technologie Spring Framework.

1.1. Przegląd literatury

W pracy [7] autorzy przeprowadzili badania zastosowania architektury monolitycznej i mikrousług do analizy systemu do zarządzania dystrybucją DBM (ang. Distribution Management System). System DBM optymalizuje przepływ mocy i zapobiega przeciążeniom. Przedstawiona w artykule aplikacja służy do analizy rozproszonej topologii modelu sieci elektroenergetycznej. Testy zostały przeprowadzone dla różnych przypadków testowych, a otrzymane wyniki zostały porównane pomiędzy aplikacją monolityczną i aplikacją mikrousług. Według otrzymanych wyników podział usługi na wiele mikrousług przyczynia się do przyspieszenia obliczeń i uzyskiwania wyników takich jak m.in. faza, moc czy hierarchia obwodów.

Aby dokonać oceny procesu migracji aplikacji o architekturze monolitycznej do mikroserwisowej oraz sprawdzić czy aplikacja wciąż spełnia określone wymagania należy wykonać testy wydajności. Z badań przeprowadzonych w pracy [8] wynika, że zaproponowany system wykorzystujący architekturę mikroserwisową zapewnia niemal zbliżoną wydajność do monolitu. Jednak może być skalowany nawet dziesiątki razy, a dzięki modułowości, możliwe jest zwiększenie liczby instancji każdego komponentu w celu uzyskania lepszych czasów odpowiedzi.

Przepustowość aplikacji mierzona jako liczba żądań na sekundę jest podstawowym atrybutem przy obliczaniu wydajności. W ramach badań [9] autorzy przeprowadzili testy obciążenia aplikacji w celu uzyskania danych związanych z wydajnością i dostępnością dla poszczególnych implementacji metod komunikacji pomiędzy usługami (gRPC, RabbitMQ, REST API). W celu pomiaru wydajności zrealizowano trzy przypadki testowe przy wykorzystaniu narzędzia JMeter [10]. Każdy z przypadków miał na celu zbadanie przepustowości każdej z metod IPC (ang. Interprocess Communication). Czas trwania, każdego testu wynosił 180 sekund i polegał na ciągłym wysyłaniu wcześniej ustalonych żądań do systemu i czekaniu na odpowiedź przez wirtualnych użytkowników.

Analiza oprogramowania stworzonego za pomocą architektury monolitycznej i mikrousługowej może zostać przeprowadzona z wykorzystaniem testów wydajności. Porównując ten sam system w dwóch rożnych architekturach uwagę należy skupić na liczbie wysyłanych zapytań, czasie obsłużenia zapytania, ilości obsłużonych zapytań w zależności od liczby wysyłanych zapytań na jedną usługę lub ilości wątków [11].

W artykule [12] zostały scharakteryzowane różne wzorce projektowe odnoszące się do architektury mikrousług oraz omówione zostały zasady, które pozwalają na ich poprawną klasyfikację. Przeprowadzono systematyczne badanie, aby zidentyfikować zgłoszone użycie mikrousług i na podstawie przypadków użycia wyodrębnić wspólne wzorce i zasady. W efekcie wyróżniono trzy wzorce orkiestracji i przechowywania danych, które wydają się być powszechnie stosowane podczas wdrażania systemów opartych na mikrousługach. Niektóre wzorce takie jak np. wzorzec hybrydowy (ang. Hybrid Pattern) lub wzorzec rejestracji i udostępniania usług (ang. Service Registry Pattern) były używane głównie w celu migracji istniejących aplikacji monolitycznych i SOA (ang. Service-Oriented Architecture).

Z kolei badania przeprowadzone przez autorów artykułu [13] polegały na systematycznym przeglądzie literatury, z którego wybrano 8 istotnych artykułów. W przypadku wzorców projektowych przegląd wyłonił 44 wzorce dla mikroserwisów, co pozwoliło zaproponować opartą na nich taksonomię (wzorce: Front-End, Back-End, DevOps, IOT, migracji i orkiestracji).

Na podstawie porównań narzędzi Locust, Gatling i JMeter można stwierdzić, że każde z nich wykonuje swoją pracę doskonale. Locust [14] to oparte na języku Python narzędzie do testowania obciążenia aplikacji w postaci wysyłanych żądań, uważane za proste w użyciu i używane do testowania rozproszonego. Zyskuje przewagę, jeżeli chodzi o wydajność. Gatling [15] to platforma o otwartym kodzie źródłowym oparta na Netty. Bazuje na języku Scala oraz narzędziach Akka. Zapewnia doskonałą obsługę protokołu HTTP (ang. Hypertext Transfer Protocol) oraz umożliwia obsługę innych protokołów. Jego przewaga nad pozostałymi narzędziami to dostarczane wizualizacje wyników. Apache JMeter to kolejne narzędzie dostępne jako aplikacja komputerowa z przyjaznym dla użytkownika graficznym interfejsem użytkownika. Wspiera wykorzystanie systemu dedykowanych wtyczek, a jego popularność opiera się m.in. na różnorodności obsługiwanych protokołów i łatwości użytkowania [16].

Docker wykorzystując Docker Compose, tworzy odpowiedni ekosystem dla aplikacji opartych na architekturze mikroserwisów. Docker Compose tworzy uporządkowane grupy kontenerów na podstawie stworzonego pliku konfiguracyjnego YAML (ang. YAML Ain't Markup Language). Wirtualizacja oparta na kontenerach oferuje doskonałe narzędzia do obniżenia kosztów wdrażania aplikacji i zapewnia pełny wgląd w różne problemy związane z mikrousługami, które mogą się pojawić na etapie wdrażania jak i działania [17].

1.2. Charakterystyka badanych architektur

Architektura monolityczna jest tradycyjnym podejściem do projektowania systemów, w którym wszystkie moduły lub usługi są zawarte w jednej aplikacji, a wszystkie funkcje systemu muszą być wdrożone razem. Najpopularniejszym i najczęściej spotykanym przykładem jest monolit jednoprocesowy. Charakteryzuje się tym, że cały kod systemu wdrażany jest jako pojedynczy proces. Tego typu systemy mogą być prostymi systemami rozproszonymi, a ich funkcjonowanie sprowadza się najczęściej do operacja odczytu i zapisu danych z bazy danych i prezentacji ich w aplikacjach webowych. Wraz ze wzrostem systemu rozrasta się również monolit co może doprowadzić do powstania monolitu modułowego. Monolit modułowy jest rozszerzeniem monolitu jednoprocesowego, w którym pojedynczy proces składa się z wielu modułów. Umożliwia to niezależną i równoległą pracę nad każdym z modułów jednak podczas

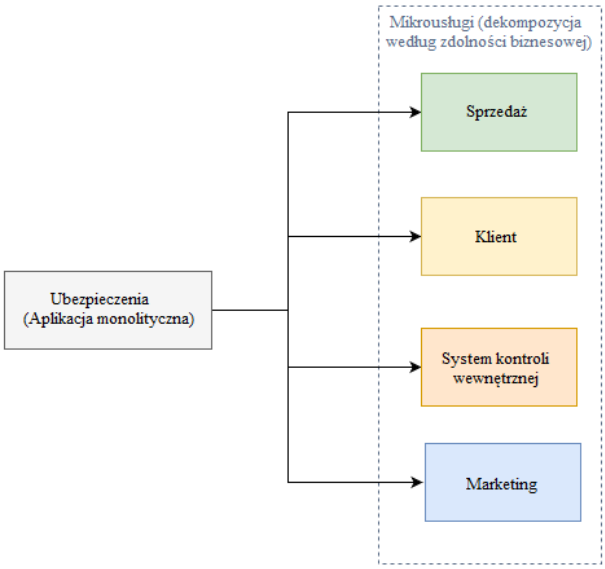
procesu wdrażania muszą one być ze sobą połączone. Jednym z największych problemów jakie wynikają z monolitu modułowego jest dekompozycja bazy danych dlatego czasami podejmowane są próby rozdzielenia bazy danych według poszczególnych modułów. Monolit rozproszony podobnie jak monolit modułowy jest podzielony, ale już nie na moduły lecz na usługi. Swoją architekturą bardzo często przypomina SOA, a nawet mikrousługi jednak w rzeczywistości usługi są silnie ze sobą powiązane przez co cały system musi być wdrożony razem.

Architektura mikrousługowa charakteryzuje się podziałem systemu na mniejsze, niezależne i luźno powiązane ze sobą usługi, które mogą być wdrażane niezależnie od siebie. Taka modułowość jest niezbędna przede wszystkim podczas tworzenia dużych i złożonych aplikacji. Każda usługa implementuje własny zestaw funkcji biznesowych i udostępnia na zewnątrz tylko niezbędne informacje wymagane przez inne usługi w postaci API (ang. Application Programming Interface). Pozostałe informacje takie jak m.in. sposób implementacji funkcjonalności, użyte technologie czy modele danych są ukrywane co pozwala na swobodne i niezależne zarządzanie usługą, wprowadzanie zmian czy dalszy rozwój. Pozwala to na wyróżnienie i oddzielenie fragmentów, które mogą być dowolnie zmieniane od tych, których zmiana jest trudniejsza. Zmiany wprowadzane wewnątrz jednej mikrousługi nie powinny wpływać na inne usługi, które się z nią komunikują [18].

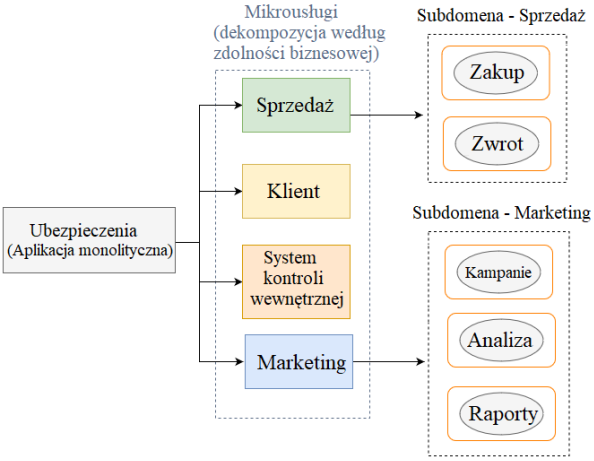
2. Metody dekompozycji aplikacji monolitycznych

W przypadku podjęcia decyzji o dekompozycji aplikacji monolitycznej do postaci struktury rozproszonej opartej o mikrousługi pojawiają się problemy związane z odpowiednim doborem wzorca dekompozycji i technik przejścia. Obecnie nie istnieje jedna metoda przebudowy struktury, która byłaby uniwersalna dla każdej aplikacji i każdego przypadku dokonując odpowiedniego podziału. Temat wyboru odpowiedniej metody według posiadanego modelu i celu jaki ma być osiągnięty jest przedmiotem wielu badań i propozycji.

Jednymi z popularnych zaproponowanych rozwiązań są dekompozycje według zdolności biznesowej i subdomeny. Dekompozycja według zdolności biznesowej przedstawiona na rysunku 1 polega na wyodrębnieniu usług odzwierciedlających jej działanie czyli to czym firma się zajmuje, aby pozyskiwać wartości np. sprzedaż, oferowanie usług czy kampania reklamowa. Natomiast dekompozycja według subdomeny, której przykład przedstawia rysunek 2 skupia się na budowaniu aplikacji wokół modelu domeny zorientowanego na obiekty. Rozwiązanie to wywodzi się z podejścia DDD (ang. Domain-Driven Design). Głównym założeniem jest utworzenie i orkiestracja usług wokół osobnych modeli domeny określających przestrzeń problemu aplikacji. Przykładowymi modelami domeny mogą być np. klient, zamówienie i sprzedaż [19]. Opóźnienia sieciowe są nieuniknioną częścią aplikacji utworzonej w architekturze mikrousługowej. Utworzone usługi mogą wymieniać między sobą bardzo dużą liczbę informacji. Aby temu zapobiec można pogrupować te usługi na podstawie transakcji. Pozwala to na uzyskanie szybszych czasów odpowiedzi i nie trzeba martwić się o spójność danych. Takie podejście nazywane jest dekompozycją według transakcji [20].



Rysunek 1: Dekompozycja aplikacji monolitycznej według zdolności biznesowej.



Rysunek 2: Dekompozycja aplikacji monolitycznej według subdomeny.

Głównym powodem podziału aplikacji monolitycznej do postaci mikrousług jest skalowanie. Możliwości skalowania aplikacji najczęściej definiuje się w postaci modelu skalowalności nazywanego sześcianem skalowania. Definiuje on trzy drogi skalowania według swoich osi: X, Y i Z. Skalowanie w osi Y polega na wykorzystaniu dekompozycji funkcjonalnej czyli podzieleniu monolitycznej aplikacja na zestaw usług według pewnych cech takich jak funkcjonalność. Otrzymane w ten sposób usługi mogą być skalowane niezależnie od siebie według osi X. Takie skalowanie polega na horyzontalnym duplikowaniu czyli powielaniu instancji. Skalowanie w osi Z czyli partycjonowanie danych polega na skalowaniu przez podział podobnych cech takich jak np.

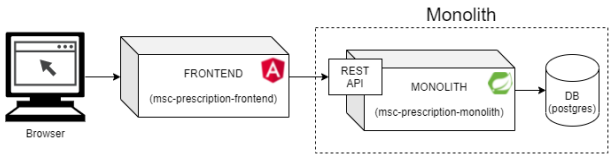
identyfikator zamówienia lub nazwisko klienta. W przeciwieństwie do skalowania w osi X, każda instancja usługi jest odpowiedzialna tylko za konkretny podzbiór cech np. obsługa klientów według przedziałów ustalonych na podstawie pierwszych liter nazwiska. Aby zdecydować do jakiej instancji trafi dane żądanie według przyjętego podziału wykorzystywany jest router pełniący rolę filtrów przychodzących żądań.

Często z dekompozycją aplikacji monolitycznej wiąże się konieczność podziału bazy danych. Idealnym rozwiązaniem jest utworzenie dla każdej z usług osobnej bazy danych o ile jest to wymagane i do której dostęp ma tylko i wyłącznie jedna usługa. Jednak istnieje jeszcze kilka sposobów zapewniających ukrywanie informacji o modelu danych i dostępu do nich. Można wykorzystać jedną bazę danych posiadającą zestaw tabel, do których dostęp mają wyłącznie konkretne usługi (do każdej z tabeli może odwoływać się tylko jedna usługa). Drugim podobnym rozwiązaniem jest utworzenie prywatnych schematów w bazie danych według usług. W pojedynczej bazie danych spójność danych zapewniały transakcje bazodanowe. Natomiast zachowanie spójności danych pomiędzy usługami obecnie stanowi spore wyzwanie i wymaga wykorzystania dodatkowych mechanizmów takich jak np. wzorzec saga [21].

3. Metody badawcze i plan eksperymentu

Do przeprowadzenia badań został wykorzystany zaprojektowany w tym celu system wspomagający zarządzanie receptami. Za stan początkowy przyjęto zapisane w systemie recepty wystawione przez lekarza. Głównym użytkownikiem systemu jest pacjent, który może zrealizować swoje recepty. W tym celu w pierwszej kolejności wybiera jedną z przypisanych do niego recept, a następnie wybiera leki razem z ich ilością do realizacji zamówienia. System zwraca listę aptek posortowaną malejąco według ilości posiadanych leków wskazanych przez pacjenta. W celu złożenia zamówienia użytkownik wybiera jedną z dostępnych aptek, a następnie uzupełnia dane niezbędne do jego realizacji np. adres zamówienia i po zatwierdzeniu danych w podsumowaniu zamówienie zostaje utworzone. Część dotyczącą płatności stanowi API dla zewnętrznych systemów i odpowiada za zmianę statusu zamówienia pozwalając na jego dalszą realizację. Kolejne funkcjonalności dotyczą pracownika apteki. Ma on wgląd we wszystkie zamówienia i możliwość ich realizacji, która polega na przygotowaniu wybranych przez pacjenta leków zmieniając ich status w systemie na gotowy. Następnie zamówienie jest przekazywane do wysyłki otrzymując status „gotowe do wysyłki” i kod dostawy. Proces związany z dostawą podobnie jak część związana z płatnością stanowi API dla zewnętrznych systemów dlatego obecnie skupia się jedynie na zmianach statusu za pomocą protokołu HTTP. W celu zachowania spójności systemu podczas implementowania go w dwóch różnych architekturach została utworzona aplikacja frontend w postaci strony internetowej wykorzystując takie technologie jak Angular i język TypeScript. Pozwoliło to na zachowanie spójności modelu pomiędzy aplikacjami.

W pierwszej kolejności system został zaimplementowany wykorzystując architekturę monolityczną według schematu przedstawionego na rysunku 3. Jest to monolit jednoprocesowy ponieważ cały kod systemu jest ze sobą ściśle powiązany i wymaga wdrożenia jako jeden proces. Cały system zawiera się w jednym serwisie posiadającym jedną bazę danych do którego odwołuje się aplikacja frontend. Do implementacji wykorzystano obiektowy język programowania Java w wersji 11 oraz Spring Framework, pozwalający na szeroki wybór możliwości podejścia do implementacji wszystkich zdefiniowanych funkcjonalności. Jako narzędzie wspomagające budowę i zarządzanie projektem wykorzystano Maven.

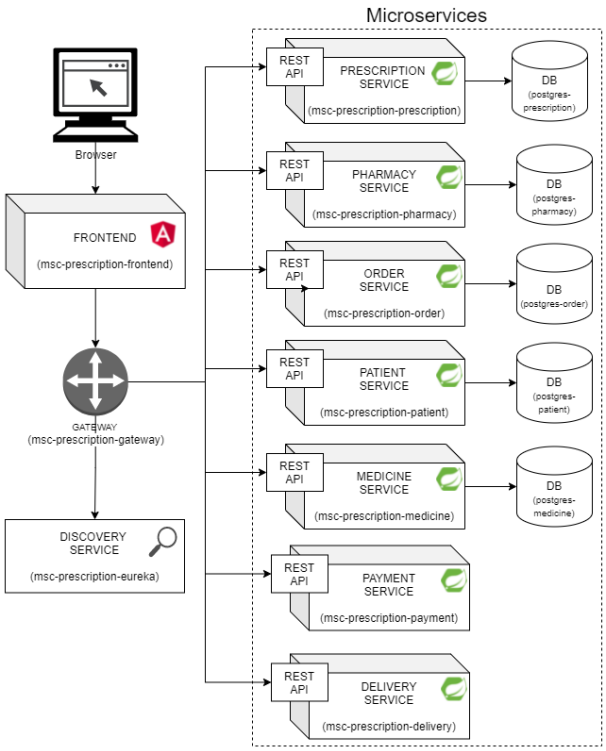


Rysunek 3: Architektura monolityczna zaimplementowanego systemu.

Do przebudowy aplikacji ze struktury monolitycznej do struktury opartej o mikrousługi została wybrana metoda podziału według funkcjonalności. Jest to idealne rozwiązanie pozwalające na skalowanie wyodrębnionych funkcjonalności np. ustawiając odpowiednią liczbę replikacji, a tym samym umożliwia łatwy dalszy rozwój systemu. Główną korzyści płynącą z podziału według funkcjonalności jest możliwość zmiany rozmiaru podstawowej infrastruktury, według wymagań niezbędnych do uruchomienia dla różnych obciążeń co zwiększa elastyczność w zakresie optymalizacji rozłożenia przychodzącego ruchu. Aby uzyskać pełne korzyści wybranej metody dekompozycji należy również dokonać podziału bazy danych tak aby każda usługa stanowiąca odrębną funkcjonalność ukrywała własny model danych i dostęp do niego jednocześnie udostępniając jedynie niezbędne operacje w postaci interfejsów.

Aplikacja mikrousługowa została utworzona przy wykorzystaniu takich samych technologii jak aplikacja monolityczna korzystając z narzędzi udostępnionych w ramach Spring Cloud będącego lekkim frameworkiem pozwalającym na implementację architektury mikroserwisowej. Oferuje on m.in. takie rozwiązania jak routing, rejestracje i odkrywanie usług, połączenia pomiędzy usługami, równoważenie ruchu, bezpiecznik czy rozproszone wiadomości [22]. Przyjęto podział według następujących funkcjonalności: prescription, pharmacy, order, patient, medicine, payment i delivery według założenia, że każda z usług powinna być wyodrębniona według zasady pojedynczej odpowiedzialności SRP (ang. Single Responsibility Principle) i zbioru realizowanych funkcjonalności skupiając się wokół obiektu domenowego (Domain-Driven Design). Podział bazy danych został wykonany według wzoru DPS (ang. Database Per Service) zachowując spójność modelu danych [21]. W rezultacie otrzymano pięć osobnych baz danych: postgres-prescription, postgres-pharmacy, post-

gres-order, postgres-patient, postgres-medicine. Architekturę zaimplementowanego systemu przedstawia rysunek 4.



Rysunek 4: Architektura mikrousługowa zaimplementowanego systemu.

Środowisko badawcze stanowią aplikacje utworzone w architekturze monolitycznej i mikrousługowej opisane powyżej. Zostały one wdrożone przy wykorzystaniu narzędzia wirtualizacji Docker Swarm na laptopie:

- procesor: Intel core i7-8750H,
- pamięć RAM: 16GB DDR4,
- system operacyjny: Windows 10,
- Docker Engine: v20.10.7,

W badaniach wykorzystywane są dwie konfiguracje platformy sprzętowej w formie maszyny wirtualnej zdefiniowane jako:

- VMB (Virtual Machine – Base) – wirtualna maszyna podstawowa o ograniczonych zasobach (vCPU = 4, RAM = 6 GB),
- VME (Virtual Machine – Extended) – wirtualna maszyna rozszerzona o nieograniczonych zasobach tj. wykorzystująca dostępne zasoby laptopa (vCPU=12, RAM=16GB),

W celu wdrażania aplikacji na środowisku testowym Docker Swarm dla każdej z aplikacji utworzono plik Dockerfile na podstawie którego zostały utworzone obrazy, które następnie umieszczono w repozytorium Docker Hub. Proces budowy i umieszczania obrazów w repozytorium odbywał się przy wykorzystaniu narzędzia CI/CD GitHub Actions. Wykorzystano obrazy apache maven w wersji 3.8.4-jdk-11-slim oraz środowiska JRE (ang. Java Runtime Environment) w wersji openjdk11:jre-11.0.6_10-alpine.

Dane testowe zostały wygenerowane za pomocą kodu napisanego w języku Java w aplikacji testowej wykorzystując Framework Hibernate i zapisane do bazy danych PostgreSQL. Na ich podstawie utworzono skrypty SQL tworzące testowe bazy danych dla aplikacji monolitycznej i mikroserwisowej oraz wypełniające je zawsze tymi samymi rekordami:

- Liczba lekarstwa: 10 000,
- Liczba aptek: 1 000,
- Liczba sztuk leku w aptece: 500,
- Liczba pacjentów: 2 000,
- Każdy pacjent posiada 10 recept i do każdej z nich przypisanych jest 6 leków po 5 sztuk do wybrania,

Do przeprowadzenia testów aplikacji wykorzystano narzędzie Gatling w wersji 3.7, które jest darmowym narzędziem pozwalającym na przeprowadzanie testów wydajnościowych. Wybrano je ze względu na możliwość definiowania scenariuszy testowych opartych o komunikację za pomocą protokołu HTTP. Oferuje funkcjonalności umożliwiające szczegółowe zdefiniowanie scenariuszy jak np. budowanie i przetwarzanie wysyłanych żądań i odpowiedzi w formacie JSON (ang. JavaScript Object Notation) odpowiadającym strukturze obiektów w aplikacjach testowych, sprawdzanie statusów odpowiedzi, wykorzystywanie instrukcji warunkowych oraz pętli, tworzenie zmiennych zapisywanych w sesji użytkownika oraz wykorzystanie funkcjonalności języka Java [15]. Aplikację do testowania stworzono wykorzystując narzędzie Maven, konwersje pomiędzy obiektami Java oraz ich reprezentacją w formacie JSON wykonano przy użyciu biblioteki Gson w wersji 2.9.0. Symulacja składa się z dwóch scenariuszy odpowiadającym korzystaniu z aplikacji przez użytkowników. W celu jak najdokładniejszego odzwierciedlenia tych procesów została utworzona aplikacja webowa stanowiąca część frontend dla testowanych aplikacji pozwalająca na realizację zamówienia od strony użytkowników systemu. Pozwoliło to na zdefiniowane niezbędnych żądań HTTP, sposobu przetwarzania danych po stronie użytkownika oraz zasymulowanie korzystania ze strony internetowej.

3.1. Pierwszy scenariusz badawczy

Pierwszy scenariusz symulacji stanowi odwzorowanie przejścia użytkownika przez aplikację od momentu wczytania danych poprzez złożenie zamówienia, aż po moment jego dostarczenia. Składa się z kroków, których implementację przedstawiono na listingu 1:

1. Pobranie wszystkich zamówień pacjenta.
2. Pobranie wszystkich recept pacjenta i wykonanie dla każdej z nich poniższych czynności.
3. Pobranie recepty.
4. Wyszukanie i pobranie listy aptek według dostępnych leków z recepty.
5. Utworzenie zamówienia wybierając pierwszą aptekę.
6. Pacjent płaci za zamówienie.
7. Pobranie zamówień dla apteki w której zostało złożone powyższe zamówienie.
8. Pobranie zamówienia.

9. Przygotowanie leku przez pracownika apteki dla każdej pozycji z zamówienia.
10. Pracownik apteki inicjalizuje wysyłkę. Firma kurierska jest powiadamiana o przesyłce do doręczenia i generowany jest kod wysyłki.
11. Wysyłka zamówienia.
12. Dostarczenie przesyłki.

Listing 1: Implementacja pierwszego scenariusza symulacji w narzędziu Gatling

```
ScenarioBuilder patientOrderScenario =
scenario("patientOrderScenario")
  .feed(feeder).tryMax(2)
  .on(exec(getOrdersByPatientId)) .tryMax(2)
  .on(exec(getPrescriptionsByPatientId))
  .exitHereIfFailed().foreach("#{prescriptionIds}",
  "prescriptionId" ).on(getPrescriptionById
  .exitHereIfFailed()
  .exec(getPharmaciesWithAvailableMedicines)
  .exitHereIfFailed()
  .exec(createOrder).exitHereIfFailed()
  .exec(payOrderById).exitHereIfFailed()
  .exec(getOrdersByPharmacyId).exitHereIfFailed()
  .exec(getOrderById).exitHereIfFailed()
  .foreach("#{orderItemIds}","orderItemId")
  .on(preparedOrderItem)
  .exec(readyToSend).exitHereIfFailed()
  .exec(deliveryOrderSent).exitHereIfFailed()
  .exec(deliveryOrderDelivered)
  .exitHereIfFailed());
```

3.2. Drugi scenariusz badawczy

Drugi scenariusz symulacji został utworzony analogicznie do pierwszego tak aby odzwierciedlał zachowanie użytkowników, którzy przeglądają swoje recepty. Umożliwia zasymulowanie procesów w systemie, które są uruchamiane bez związku z zadaniem generacji zamówienia, dlatego został nazwany jako „obciążenie tła". Na listingu 2 przedstawiono jego implementację:

1. Pobranie wszystkich recept pacjenta.
2. Pobranie wszystkich zamówień pacjenta.
3. Każda z recept pacjenta zostaje pobrana po kolei w celu wyświetlenia szczegółowych informacji.
4. Pobranie wszystkich recept i zamówień pacjenta, powrót do okna wyboru kolejnej recepty.

Listing 2: Implementacja drugiego scenariusza symulacji w narzędziu Gatling

```
ScenarioBuilder backgroundUser =
  scenario("backgroundUserScenario")
  .feed(feederBackgroundUser)
  .exec(getPrescriptionsByPatientId)
  .exec(getOrdersByPatientId).exitHereIfFailed()
  .foreach("#{prescriptionIds}", "prescriptionId")
  .on(getPrescriptionById
  .exec(getPrescriptionsByPatientIdWithoutSaveIds)
  .exec(getOrdersByPatientId));
```

3.3. Przebieg eksperymentu

Proces testowania polegał na wdrożeniu kolejno aplikacji monolitycznej, mikroserwisowej bez replikacji i mikroserwisowej z replikacjami na konkretnym środowisku testowym, a następnie dla każdej z nich przeprowadzenie symulacji według przyjętych scenariuszy. Kroki przeprowadzenia symulacji dla każdego scenariusza wyglądały następująco:

1. Wdrożenie aplikacji na środowisko testowe (Docker Swarm) za pomocą pliku docker-stack.
2. Przeprowadzenie symulacji ustawiając liczbę użytkowników oraz obciążenie tła według obecnie realizowanego scenariusza.
3. Zapis wyników.
4. Restart środowiska testowego.

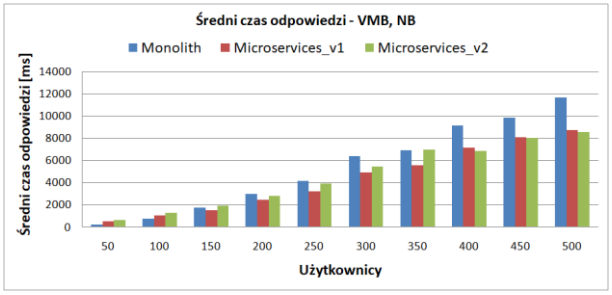
Po wykonaniu wszystkich pomiarów dane eksperymentalne w postaci czasów odpowiedzi serwera oraz parametry związane z obciążeniem aplikacji na wysłane żądania zostały zebrane i pogrupowane według scenriuszy. Testowane aplikacje oznaczono w następujący sposób:

- Monolith – aplikacja monolityczna,
- Microservices_v1 – aplikacja mikroserwisowa bez replikacji (wszystkie komponenty z ustawioną ilością replik na 1),
- Microservices_v2 – aplikacja mikroserwisowa z replikacjami dla VMB,
- Microservices_v3 – aplikacja mikroserwisowa z replikacjami dla VME,

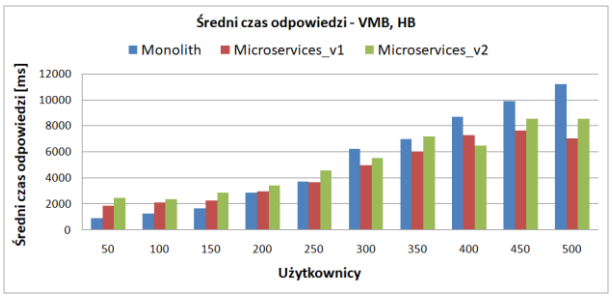
4. Wyniki badań

Wyniki każdej symulacji zostały zebrane, przetworzone i prezentowane przez narzędzie Gatling w formie strony internetowej, która oferuje wiele danych zebranych w postaci interaktywnych wykresów i tabel z których można odczytać szczegółowe dane dotyczące symulacji np. jak zmieniała się liczba aktywnych uczestników w jednostce czasu, liczbę żądań i odwiedzi na sekundę, dystrybuantę czasu odpowiedzi. Uwagę skupiono głównie na zakładce „GLOBAL", w której znajdują się najważniejsze dane z wyników symulacji zebrane w postaci wykresu „Indicators", który przedstawia rozkład czasów odpowiedzi w standardowych zakresach, oraz tabeli „STATISTICS" ze standardowymi danymi statystycznymi takimi jak m.in. minimalny, maksymalny i średni czas odpowiedzi, odchylenie standardowe, percentyle i liczba żądań na sekundę. Stanowi to przejrzystą formę wizualizacji danych ułatwiając ich odczyt i interpretację. W celu porównanie wydajności aplikacji według przyjętych kryteriów wyniki otrzymane z narzędzia Gatling zostały pogrupowane i zebrane w formie tabel na podstawie, których zostały wygenerowane wykresy.

Aplikacja monolityczna charakteryzuje się krótszym czasem odpowiedzi dla scenariuszy z niskim obciążeniem w postaci liczby użytkowników jednak wraz ze wzrostem obciążenia, a tym samym liczby żądań różnica w czasach odpowiedzi zwiększa się na korzyść aplikacji mikroserwisowej i wynosi coraz więcej. Dla podstawowej maszyny wirtualnej zmiana ta następuje dla ok. 200 – 300 użytkowników w zależności od obciążenia dodatkowego i replikacji aplikacji mikrousługowej co przedstawiają wykresy na rysunkach 5 i 6.

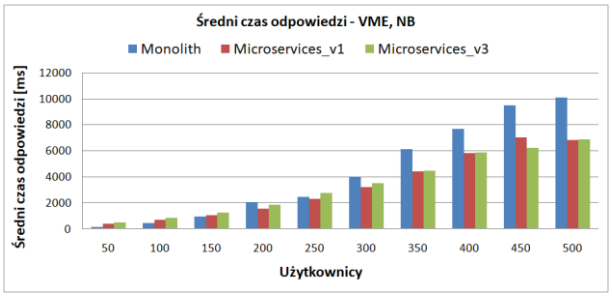


Rysunek 5: Średni czas odpowiedzi, brak dodatkowego obciążenia, podstawowa maszyna wirtualna.

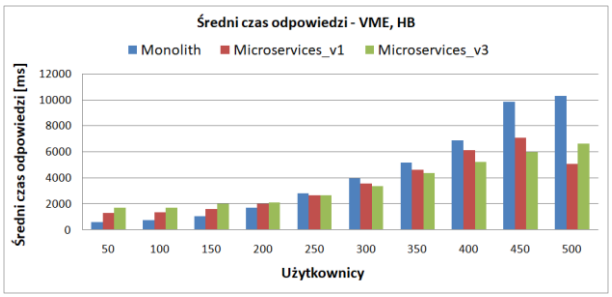


Rysunek 6: Średni czas odpowiedzi, silne obciążenie dodatkowe, podstawowa maszyna wirtualna.

Zmiana na korzyść aplikacji mikrousługowej dla rozszerzonej maszyny wirtualnej zachodzi podobnie w granicach scenariuszy dla 200 – 300 użytkowników i jest widoczna na wykresach przedstawionych na rysunkach 7 i 8.

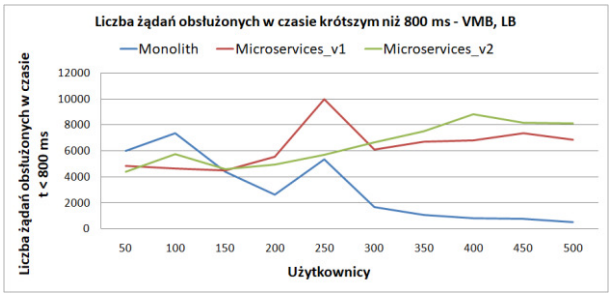


Rysunek 7: Średni czas odpowiedzi, brak dodatkowego obciążenia, rozszerzona maszyna wirtualna.

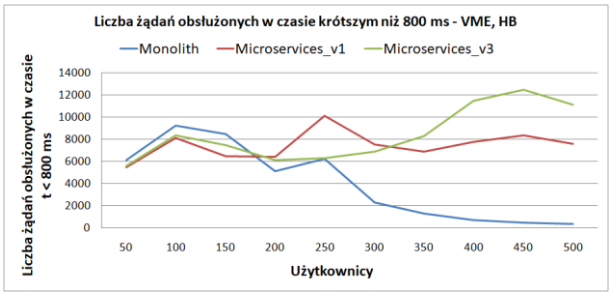


Rysunek 8: Średni czas odpowiedzi, silne obciążenie dodatkowe, rozszerzona maszyna wirtualna.

Rysunki 9 i 10 przedstawiają wykresy liczby żądań obsłużonych w czasie krótszym niż 800 ms. Jest to domyślna wartość ustawiona w wykorzystywanym narzędziu Gatling i określa górną granicę czasu odpowiedzi, do której odpowiedź na żądanie uznawana jest za zrealizowaną w krótkim czasie. Do ok. 150 użytkowników dla podstawowej maszyny wirtualnej i ok. 200 dla rozszerzonej maszyny wirtualnej lepiej radzi sobie z obsługą żądań aplikacja monolityczna, ale później wraz ze wzrostem obciążenia różnica diametralnie się zmienia na korzyść aplikacji mikroserwisowej. Wraz ze wzrostem obciążenia liczba żądań obsłużonych w czasie $t < 800$ ms dla aplikacji monolitycznej maleje podczas gdy dla aplikacji mikroserwisowej rośnie w zależności od scenariusza do 400 lub 450 użytkowników. Na wykresach widoczne jest, że dla aplikacji monolitycznej i mikroserwisowej bez replikacji następuje gwałtowny wzrost wartości dla obciążenia w postaci od 200 do 250 użytkowników oraz znaczny spadek wartości dla scenariuszy od 250 do 300 użytkowników.



Rysunek 9: Liczba żądań obsłużonych w czasie krótszym niż 800 ms, słabe obciążenie dodatkowe, podstawowa maszyna wirtualna.



Rysunek 10: Liczba żądań obsłużonych w czasie krótszym niż 800 ms, silne obciążenie dodatkowe, rozszerzona maszyna wirtualna.

5. Podsumowanie

Tabela 1 przedstawia zestawienie ze sobą badanych aplikacji pod względem otrzymanych wartości parametrów rejestrowanych w poszczególnych scenariuszach testowych. Wyróżniono dwie kategorie obciążenia aplikacji, małe i duże na podstawie liczby wysyłanych żądań związanych z liczbą wirtualnych użytkowników. Dodatkowo w tabeli uwzględniono podział według konfiguracji platformy sprzętowej na VMB i VME. Kolorami odpowiadającymi poszczególnym rodzajom aplikacji zaznaczono, które z nich prezentują najlepsze wyniki dla danego scenariusza i kryterium.
Oznaczenia kolorów:

- niebieski – aplikacja oparta o architekturę monolityczną,
- pomarańczowy – aplikacja wykorzystująca mikrousługi,
- zielony – aplikacja wykorzystująca mikrousługi ze skalowaniem,
- szary – wyniki porównywalne dla wszystkich aplikacji,

Tabela 1: Zestawienie badanych aplikacji pod względem otrzymanych wartości parametrów rejestrowanych w poszczególnych scenariuszach testowych

Badane kryterium	Małe obciążenie		Duże obciążenie	
	VMB	VME	VMB	VME
Liczba poprawnie obsłużonych żądań				
Liczba przetworzonych żądań na sekundę				
Różnica liczby obsłużonych żądań w jednostce czasu pomiędzy aplikacją mikrousługową i monolityczną				
Średni czas odpowiedzi				
Odchylenie standardowe czasu odpowiedzi				
Liczba żądań obsłużonych w czasie krótszym niż 800 ms				
Procentowy udział żądań z czasem odpowiedzi krótszym niż 800 ms spośród poprawnie obsłużonych żądań				

Z tabeli wynika, że dla małego obciążenia zarówno dla środowiska VMB jak i VME niemal dla każdego badanego kryterium lepsza jest aplikacja oparta o architekturę monolityczną. Jedynie w przypadku liczby poprawnie obsłużonych żądań wyniki są porównywalne dla wszystkich aplikacji, a podczas dużego obciążenia dla podstawowej maszyny wirtualnej widać przewagę aplikacji monolitycznej. Natomiast dla VME najlepsze wyniki osiąga aplikacja mikrousługowa ze skalowaniem. Aplikacja wykorzystująca mikrousługi bez skalowania najlepiej sprawdza się w przypadku dużego obciążenia dla podstawowej maszyny wirtualnej. Osiąga ona najlepsze rezultaty dla kryteriów związanych z liczbą przetworzonych żądań na sekundę, średnim czasem odpowiedzi oraz odchyleniem standardowym, które wskazuje na jej stabilność podczas zmiany obciążenia. Podczas dużego obciążenia dla rozszerzonej maszyny wirtualnej najlepsze wyniki dla wszystkich kryteriów osiąga aplikacja wykorzystująca mikrousługi ze skalowaniem. Wynika to z większych zasobów sprzętowych wpływających na wydajność uruchamianych replik. Skalowanie aplikacji mikrousługowej umożliwia przede wszystkim poprawę wydajności związanej z szybszą obsługą większej liczby żądań na co wskazują kryteria związane z obsługą żądań z czasem odpowiedzi krótszym niż 800 ms.

Literatura

[1] C. Richardson, Mikroserwisy: Wzorce z przykładami w języku Java, PWN, 2020.

[2] P. Mell, T. Grance, et al. The NIST definition of cloud computing. National Institute of Standards and Technology Special Publication 800-145, Gaithersburg (2011) 1-7.

[3] V. Andrikopoulos, T. Binz, F. Leymann, S. Strauch, How to adapt applications for the cloud environment. Challenges and solutions in migrating applications to the cloud, Computing 95(6) (2013) 493–535.

[4] P. Jamshidi, A. Ahmad, C. Pahl, Cloud migration research: A systematic review, IEEE Transactions on Cloud Computing 1(2) (2013) 142–157.

[5] A. Balalaie, A. Heydarnoori, P. Jamshidi, Migrating to Cloud-Native Architectures Using Microservices. An Experience Report, European Conference on Service-Oriented and Cloud Computing (2015) 201-215.

[6] L. Bass, I. Weber, L. Zhu, DevOps: A Software Architect's Perspective, O'Reilly, 2019.

[7] S. Stoja, S. Vukmirovic, N. Dalcekovic, D. Capko, Accelerating Performance in Critical Topology Analysis of Distribution Management System Process by Switching from Monolithic to Microservices, Revue Roumaine des Sciences Techniques Serie Electrotechnique et Energetique 63 (2018) 338-343.

[8] K. Cebeci, Ö. Korçak, Design of an Enterprise Level Architecture Based on Microservice, Bilişim Teknolojileri Dergisi 13 (2020) 357-371.

[9] B. Shafabakhsh, R. Lagerström, S. Hacks, Evaluating the Impact of Inter Process Communication in Microservice Architectures, International Workshop on Quantitative Approaches to Software Quality 2767 (2020) 55–63.

[10] Strona główna Apache JMeter, https://jmeter.apache.org/, [27.05.2022].

[11] V. Adamescu, Analysing monolithic and microservices software architecture for SME web services/applications, (2020) https://www.researchgate.net/publication/341353952_Analysing_monolithic_and_microservices_software_architecture_for_SME_web_servicesapplications

[12] D. Taibi, V. Lenarduzzi, P, Claus, Architectural Patterns for Microservices: A Systematic Mapping Study, Closer (2018) https://hdl.handle.net/10863/5599

[13] F. Vera-Rivera, H. Astudillo, M. Gaona, Desarrollo de aplicaciones basadas en microservicios: tendencias y desafíos de investigación, Revista Iberica de Sistemas e Tecnologias de Informacao E23 (2019) 107 - 120.

[14] Strona główna narzędzia testowego Locust, https://locust.io/, [27.05.2022].

[15] Dokumentacja narzędzia Gatling, https://gatling.io/docs/gatling/reference/3.7/, [27.05.2022].

[16] S. Shrivastava, S. B. Prapulla, Comprehensive Review of Load Testing Tools, IRJET (2020) 3392-3395.

[17] A. Raj, K. Jasmine, Building Microservices with Docker Compose, The International journal of analytical and experimental modal analysis XIII (2021) 1215- 1219.

[18] S. Newman, Budowanie mikrousług. Projektowanie drobnoziarnistych systemów, Helion, 2022.

[19] S. Newman, Monolith to Microservices. Evolutionary Patterns to Transform Your Monolith, O'Reilly, 2019.

[20] Decompose by transactions, https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-decomposing-monoliths/decompose-transactions.html, [27.05.2022].

[21] Wzorce projektowe architektury mikrousługowej, https://microservices.io/patterns/index.html, [27.05.2022].

[22] Dokumentacja Spring Cloud, https://spring.io/projects/spring-cloud, [27.05.2022].