POLISH-JAPANESE ACADEMY
OF INFORMATION TECHNOLOGY

Współdzielenie kodu w mikroserwisach

Artem Romanenko Numer albumu: S32237

Polish-Japanese Academy of Information Technology
Promotor: dr Krzysztof Barteczko, prof. PJATK

Warszawa, 10 maja 2025

Abstract. Rapid development and adoption of microservices architecture in last few years changed the world of Informatics. Microservices architecture has many advantages, such as flexibility, fast and comfortable deployment, and easy maintenance. However, this architecture also causes new challenges, such as code duplication and complicated code management. In my master's thesis, I've carried on deep and detailed analysis of all available approaches to manage shared code in system of microservices. In my work, I want to compare all available approaches of code share code in system of microservices using literature sources and prepared code laboratory, also to define cases in which it is better to choose one or another approach. In my practical part, I want to present my own approach to sharing code that will combine the advantages of all existing solutions and provide the best performance and code comfort for developer in code management. At the end of my work, I will place comparison of my solution with existing solutions using prepared performance tests.

Keywords: Microservices Architecture · Code Sharing · Performance.

Streszczenie Bez wątpienia mogę powiedzieć, że szybki rozwój i przyjęcie architektury mikroserwisowej w ostatnich latach zmieniły świat informatyki. Uważam, że architektura mikroserwisowa ma wiele zalet, takich jak na przykład elastyczność, prostota wdrożenia i utrzymania. Natomiast powoduje nowe wyzwania dla programistów, takie jak do przykładu problem duplikacji kodu. W pracy magisterskiej przeprowadziłem dokładną analizę metod i podejść współdzielenia kodu w systemach mikroserwisów, porównanie za pomocą źródeł literaturowych oraz przeprowadzonych badań dostępne podejścia do współdzielenia kodu, zdefiniować przypadki, w których warto wybrać takie lub inne podejście. W części praktycznej zaprezentowałem własne nowoczesne rozwiązanie, które połączyło zalety istniejących rozwiązań, wydajność oraz komfort zarządzania kodem. Na końcu umieściłem porównanie mojego rozwiązania z istniejącymi za pomocą przygotowanych testów wydajnościowych.

Keywords: Architektura mikrousług · Współdzielenie kodu · Wydajność.

Spis treści

1 Wstęp

1.1 Zakres pracy

W pracy magisterskiej przeprowadzono dokładną analizę metod i podejść do współdzielenia kodu w systemach mikroserwisów. Zakres tej pracy zawiera porównanie między architekturą monolitową a mikroserwisową, identyfikację typów obiektów wykorzystywanych w kodzie współczesnych aplikacji, analizę dostępnych metod i podejść do współdzielenia kodu, IDL, SDK oraz biblioteki, a także praktyczną implementację aplikacji wspierającej nowatorskie podejście do współdzielenia kodu Serverless.

1.2 Motywacja

Główną motywacją do napisania pracy było rosnące we współczesnych czasach zainteresowanie mikroserwisowym podejściem do architektury, które ma wiele zalet, takich jak elastyczność, łatwość wdrożenia i utrzymania na produkcji, możliwość łatwej i szybkiej naprawy problemów na produkcji, natomiast również powoduje nowe wyzwania, takie jak problem duplikacji kodu i zapotrzebowanie na współdzielenie kodu między mikroserwisami w systemie. W mojej pracy chcę przeanalizować, jakie metody współdzielenia kodu są dostępne na rynku, jakie są ich zalety i wady, jakie są najlepsze praktyki we współdzieleniu kodu w systemach mikroserwisów. Chcę porównać bardziej tradycyjne podejścia do współdzielenia kodu, takie jak IDL, SDK, biblioteki, z nowatorskim podejściem do współdzielenia kodu, takim jak Serverless, chcę stworzyć oprogramowanie, które połączy główne zalety tradycyjnych metod współdzielenia kodu z zaletami bardziej nowoczesnych podejść.

1.3 Zawartość pracy

Praca jest ustrukturyzowana w kilka rozdziałów, przedstawiam krótki opis rozdziałów parcy:

1. Wstęp - W tym rozdziale analizuję wady i zalety architektury mikroserwisowej oraz monolitycznej, też analizuję, dlaczego z architektury monolitycznej powstała w ostatich latach architektura mikroserwisowa, robię analizę przypadków w których lepiej użyć architektury monolitycznej a w których architekturę mikroserwisową. Opisuję dlaczego powstało zapotrzebowanie na współdzielenie kodu. Uzasadniam informację przykładami z literatury.
2. Architektura mikroserwisowa a Monolitowa - W tym rozdziale paracy porównuję mikroserwisy z podejściem monolitycznym, przedstawiam różnice w utrzymaniu i skalowalności, za pomocą źródeł literaturowych.
3. Analiza dziedziny problemowej - W tym rozdziale przedstawiam kategoryzuję typy obiektów które mogą potrzebować współdzielenia kodu w systemie mikroserwisów, analizuję możliwe wyzwania związane z współdzieleniem poszczególnych typów obiektów, przedstawiam uzasadnienia z literatury.
4. Analiza metod współdzielenia kodu - W tym rozdziale kompleksowa analiza istniejących metod współdzielenia kodu w systemach mikroserwisów za pomocą źródeł literaturowych kryterium ocenia.
5. Opis części praktycznej - Zawiera opis przygotowanej w ramach pracy aplikacji która wspomaga nowoczesne rozwiązanie do współdzielenia kodu w systemach mikroserwisów - Server Less rozszerzając możliwości tego rozwiązania.

2 Ogólna definicja współdzielenia kodu

Współdzielenie kodu – to praktyka w programowaniu polegająca na udostępnianiu fragmentów kodu lub całych funkcji, metod albo modułów – w zależności od architektury i użytych technologii – między różnymi projektami, zespołami, a czasem również między firmami lub społecznościami open source.

Celem współdzielenia kodu jest:

1. Unikanie duplikacji – możliwość wielokrotnego użycia fragmentu kodu w różnych miejscach.
2. Zwiększenie produktywności – możliwość szybkiego korzystania z gotowych rozwiązań.
3. Utrzymanie spójności – pozwala na stosowanie jednolitych reguł wewnątrz aplikacji oraz między aplikacjami, takich jak na przykład reguły walidacji, formatowania danych czy też obsługi błędów.
4. Ułatwienie współpracy zespołowej – ułatwia korzystanie z wyników pracy jednego programisty przez zespół oraz między zespołami.
5. Lepsze testowanie i jakość – ze względu na utrzymanie spójnych reguł w aplikacji kod jest łatwiejszy do testowania, co zmniejsza ryzyko błędów i poprawia ogólną jakość kodu.

W przypadku architektury mikroserwisowej współdzielenie kodu to praktyka udostępniania fragmentów kodu – na przykład bibliotek, modułów, metod lub funkcji – pomiędzy mikroserwisami (elementami systemu mikroserwisowego), w celu osiągnięcia korzyści przedstawionych powyżej. W przypadku architektury monolitycznej współdzielenie kodu również odgrywa ważną rolę w utrzymaniu jakości kodu i aplikacji.

3 Architektura mikroserwisowa a monolitowa

Mikroserwisowa infrastruktura, szybko rozwijająca się w ostatnich latach, przyniosła wiele korzyści w świat informatyki. Pozwoliła na tworzenie skalowalnych i elastycznych systemów, ułatwiła utrzymanie takich systemów. Natomiast musimy również uważać przy wyborze architektury na to, że w przypadku mikroserwisów musimy poradzić sobie z kwestią współdzielenia kodu w takim systemie mikroserwisów.

3.1 Architektura mikroserwisowa

Na podstawie mojego doświadczenia, podział aplikacji na mniejsze, niezależne usługi przynosi liczne korzyści. Każdy mikroserwis można skalować osobno, co pozwala na poprawę wydajności całego systemu. W sytuacji, gdy pojawia się problem, naprawa jest prostsza, ponieważ wystarczy skupić się na jednej części aplikacji, a nie na całym systemie. Mikrousługi wpływają również na organizację pracy zespołowej. Deweloperzy mogą skupić się na rozwoju pojedynczych mikroserwisów, co ułatwia zarządzanie projektem i zmniejsza ryzyko wystąpienia błędów w innych częściach aplikacji. Ponadto, gdy pojawia się problem, jest on łatwiejszy do zidentyfikowania, a proces naprawy staje się bardziej efektywny. Kolejną zaletą mikrousług, moim zdaniem, jest ich elastyczność pod względem doboru technologii. Dzięki temu, że każdą usługę można tworzyć i zarządzać nią z użyciem różnych narzędzi, istnieje większa swoboda w doborze najlepszych rozwiązań technologicznych do każdej części systemu. Mikrousługi wspierają również skuteczne zarządzanie projektem, ponieważ umożliwiają szybsze wdrażanie nowych funkcji i reagowanie na zmiany. Taki system jest bardziej odporny na awarie, a cały proces utrzymania aplikacji jest prostszy.

> "Microservices are small, autonomous services that work together. Let's break that definition down a bit and consider the characteristics that make microservices different." [12, p. 2]

Architektura mikrousług przedstawia następujące cechy:

1. Modułowość - W przypadku mikroserwisów możemy wdrażać pojedyncze elementy systemu bez wpływu na pozostałe.
2. Niezawodność - Mikroserwisy są niezawodne, ponieważ błąd w jednej części systemu niekoniecznie będzie miał wpływ na cały system.
3. Interoperacyjność — Komunikacja za pomocą wymiany danych przez zasoby sieciowe daje nam możliwość połączenia mikroserwisów napisanych w różnych językach i technologiach. W tym przypadku możemy połączyć w jednym systemie kilka technologii i wykorzystać mocne strony każdej z nich.

4. Równoległy rozwój - Zespoły programistów mogą pracować nad osobnymi mikroserwisami równolegle, bez konfliktów zmian w kodzie.

3.2 Architektura monolitowa

Charakterystyka architektury monolitycznej:

1. Wdrożnie - Cała aplikacja jest wdrażana jako całość, niepodzielna jednostka. Wszustkie aktualizacje wymagają wdrożenia całej aplikacji.
2. Kontrola przepływu kodu - Centralny moduł kontroli w aplikacji koordynuje sekwencyjne wykonywanie kodu.
3. Powiązanie - Poszczególne elementy aplikacji są ścisłe powiązane.
4. Współdzielona pamięć - Wszystkie komponenty oprogramowania mają bezpośredni dostęp do zasobów, co sprzyja ich integracji, natomiast taka konfiguracja może powodować konflikty elementów kodu na poziomie sprzętowym i trudności ze skalowaniem. [18]

3.3 Porównanie

> "I should call out that microservices are no free lunch or silver bullet, and make for a bad choice as a golden hammer. They have all the associated complexities of distributed systems." [12, p. 11]

Zalety architektury monolitycznej:

- Czas napisania – Dla małych i średnich projektów umożliwia szybsze napisanie kodu bez potrzeby zarządzania komunikacją między mikroserwisami w systemie.
- Łatwość wdrożenia – Wdrożenia kodu w przypadku architektury monolitycznej są mniej skomplikowane.
- Proste testowanie – O wiele łatwiejsze jest testowanie aplikacji monolitycznej dzięki ścisłej integracji wszystkich komponentów. Testy jednostkowe i integracyjne można w prosty sposób przeprowadzać w ramach pojedynczej bazy kodu.
- Skalowalność – Zapewnia odpowiednią skalowalność dla niewielkich rozwiązań poprzez replikację całej aplikacji.

Zalety architektury mikroserwisowej:

1. Skalowalność - Mikroserwisy umożliwiają niezależne skalowanie, optymalizując wykorzystanie zasobów i zapewniając wydajność systemu.
2. Równoległa praca - Architektura mikroserwisowa pozwala na równoległy rozwój kilku elementów systemu bez konfliktów, co prowadzi do szybszego rozwoju.
3. Elastyczność - Każdy mikroserwis może być napisany w języku najbardziej pasującym do jego konkretnej funkcji, daje to możliwość użycia mocnych stron każdej z wielu dostępnych technologii.
4. Debugowanie - Rozwiązywanie problemów w mikroserwisach jest prostsze, każdy osobny mikroserwis jest niewielki i prosty w analizie.
5. Bezpieczeństwo - Mikroserwisy ułatwiają ochronę danych. Każda usługa ma swoją bazę danych, co utrudnia hakerom dostanie się do danych. [18]

Wybór między architekturą monolityczną a mikroserwisową zależy od wielu faktorów, takich jak rozmiar projektu, zapotrzebowanie na ciągły rozwój, wymagania dotyczące skalowalności oraz kompetencje i doświadczenia zespołu. Architektura monolityczna lepiej pasuje dla małych i mniej skomplikowanych projektów natomiast mikroserwisowa oferuje benefity w przypadku wykorzystania w większych, bardziej skomplikowanych systemach.

4 Analiza dziedziny problemowej

Celem niniejszej pracy magisterskiej jest zgłębienie tematu współdzielenia obiektów w systemach mikroserwisów oraz zrozumienie wyzwań i możliwości związanych z tym zagadnieniem. Praca ma na celu analizę różnych strategii, narzędzi i metodyk, które mogą pomóc w efektywnym i bezpiecznym współdzieleniu danych i obiektów w skali mikroserwisowej architektury.

4.1 Typy obiektów w aplikacjach mikroserwisowych

Postanowiłem rozpocząć analizę dziedziny problemowej od tego że za pomocą źródeł literaturowych zidentyfikuję typy obiektów oraz dokonać analizy tego, które z nich mogą wymagać współdzielenia w systemach mikroserwisów.

1. DTO (Data Transfer Object) - Takie obiekty są specjalnie przeznaczone do obsługi przesyłania danych między elementami systemu mikroserwisów, są lekkie i nie zawierają logiki biznesowej, dlatego współdzielenie takich obiektów jest jak najbardziej zalecane ponieważ zapewniają spójność danych po obu stronach komunikacji oraz zmniejsza ilość błędów w przypadku rozwoju lub utrzymania aplikacji.
2. Model (lub Encja) - Najczęściej są logicznie powiązane z tabelami w bazie danych, ze względu na to że we współczesnych czasach za dobrą praktykę jest uważane podejście w którym jeden mikroserwis jest powiązany z maksymalnie jedną bazą danych, współdzielenie tego typu obiektów nie jest zalecane.
3. Wyjątek (Exception) - Zalecanym jest współdzielenie informacji o wyjątkach w systemie mikroserwisów tym samym tworząc jednolitą obsługę błędów w systemie. Logika związana z obsługą błędów, w przypadku podobności w kilku serwisach też jest dobrym kandydatem do współdzielenia. Taka standaryzacja w systemie mikroserwisów sprawia że łatwiej później znaleźć źródło błędu i naprawić go.
4. Walidatory - W przypadku współdzielenia walidatorów w systemie mikroserwisów można zapewnić integralność danych w całym systemie oraz zmniejszyć duplikację kodu, również standaryzacja

walidacji obiektów w systemie zmniejsza ilość ewentualnych problemów w trakcie ewolucji i rozwoju mikroserwisów w przyszłości, współdzielenie takich obiektów jest dobrą praktyką i jest zalecane.

5. Serwisy - W przypadku logiki która powtarza się w dwóch i więcej serwisach zalecanym jest współdzielenie takiego serwisu za pomocą najlepiej w tym przypadku pasującej metody, zmniejsza to duplikację kodu natomiast zwiększa czytelność i spójność kodu w systemie mikroserwisów, zmniejsza ilość ewentualnych problemów w trakcie ewolucji i rozwoju kodu.

Musimy zdawać sobie sprawę, że mimo tego, iż współdzielenie kodu daje nam wiele korzyści, nadmiarowe współdzielenie kodu może spowodować problemy i pogorszyć jakość oraz zrozumiałość naszego kodu, utrudnić testowanie, spowodować powstanie błędów kaskadowych, czyli sytuacji, kiedy błąd w jednym źródle powoduje problemy od razu w wielu miejscach, oraz utrudnić utrzymanie systemu. Należy zawsze używać współdzielenia ostrożnie i podejmować decyzję w każdym konkretnym przypadku współdzielenia.

4.2 Kryteria porównania i analizy metod współdzielenia kodu

Po przeprowadzeniu dogłębnej analizy tematu i między innymi źródeł literaturowych, takich jak [12], [14], [13], [7] mogę zdefiniować kryteria porównania metod współdzielenia kodu w systemach mikroserwisów jako następujące:

1. Komunikacja - Metoda współdzielenia obiektów powinna uwzględniać efektywną komunikację między serwisami. Ważne jest, aby obiekty były dostępne w sposób, który minimalizuje opóźnienia i obciążenie sieci. Dobre rozwiązania mogą obejmować użycie asynchronicznej komunikacji, takiej jak kolejki wiadomości, czy też bezpośrednie zapytania między serwisami.
2. Izolacja - Metoda współdzielenia obiektów powinna zapewniać odpowiednią izolację między serwisami. Każdy serwis powinien mieć kontrolę nad swoimi własnymi obiektami i nie powinien być zależny od innych serwisów. To pozwala na większą elastyczność i umożliwia niezależny rozwój i skalowanie serwisów.
3. Bezpieczeństwo - W przypadku współdzielenia obiektów, istotne jest zapewnienie odpowiedniego poziomu bezpieczeństwa. Dostęp do obiektów powinien być kontrolowany i zabezpieczony w sposób, który zapobiega nieautoryzowanemu dostępowi. Mechanizmy uwierzytelniania i autoryzacji powinny być odpowiednio wdrożone, aby zapewnić bezpieczne współdzielenie obiektów.
4. Skalowalność - Metoda współdzielenia obiektów powinna być skalowalna. System powinien być w stanie efektywnie obsługiwać rosnącą liczbę żądań i zapewniać odpowiednią wydajność. Współdzielenie obiektów powinno być projektowane w taki sposób, aby możliwe było łatwe skalowanie poszczególnych serwisów bez wpływu na cały system.
5. Wersjonowanie - Ważne jest również odpowiednie zarządzanie wersjami obiektów, szczególnie w środowisku mikroserwisów, gdzie różne serwisy mogą używać różnych wersji obiektów. Metoda współdzielenia obiektów powinna uwzględniać zarządzanie wersjami i umożliwiać aktualizacje w sposób kontrolowany i bezpieczny.

5 Analiza metod współdzielenia kodu

We współczesnych czasach współdzielenie kodu jest niezbędne w skomplikowanych systemach mikroserwisów, w tym rozdziale pracy porównuję istniejące podejścia do współdzielenia kodu oraz porównam je za pomocą zdefiniowanych w poprzednim rozdziale kryteriów. Głównym celem jest zdefiniowanie najlepszych praktyk współdzielenia kodu, zdefiniować, które podejście najbardziej dopasowane do współdzielenia konkretnych typów obiektów, zdefiniowanych powyżej w tej pracy oraz porównanie wydajności i możliwości skalowania w przypadku każdego z podejść za pomocą przygotowanych programistycznych testów wydajnościowych.

5.1 Metody współdzielenia kodu z opisem

Za pomocą źródeł literatury oraz własnego doświadczenia zdefiniowałem dostępne na dzień dzisiejszy podejścia:

1. Interface definition languages - do IDL należą wiele popularnych technologii, Protocol buffers, Avro IDL, Open API. [20]
2. Biblioteki kodu - najbardziej oczywiste podejście, które daje możliwość współdzielenia wszystkich rodzajów obiektów w aplikacji.
3. Biblioteki klienckie - podejście które polega na udostępnieniu za pomocą biblioteki kodu zestawu obiektów potrzybnych do kmonikacji z narzedziem zewnętrznym, takim jak, na przykład konsola AWS.
4. Wyniesienie kodu do osobnego REST serwisu - podejście polega na przechwytywaniu i udostępnieniu wspólnej logiki poprzez utworzenie osobnego mikroserwisu.
5. Serverless - nowoczesne podejście do pisania kodu i przetwarzania informacji wykorzystywane w chmurach obliczeniowych. Pozwala na uruchamianie kawałków kodu, metod i funkcji niezależnie, bez warstwy zarządzania aplikacją, a także na uruchamianie kodu bez konieczności zarządzania podstawową infrastrukturą. [6]

Pryncypia działania poszczególnych metod współdzielenia kodu:

Interface definition languages – są powszechnie używane w oprogramowaniu zdalnych wywołań procedur. W takich przypadkach

maszyny po obu stronach łącza mogą używać różnych systemów operacyjnych i języków programowania. IDL oferują most pomiędzy dwoma różnymi systemami. Natomiast również mogą być użyte dla generacji obiektów lub kodów w systemach mikroserwisów. Każdy system IDL posiada określony przez twórców język IDL oraz interpretator języka IDL. Interpretator języka IDL przygotowany i dostarczony przez twórców potrafi na podstawie udostępnionych reguł wygenerować kod używając przygotowane pliki IDL. Używając języka IDL możemy przygotować obiekty lub kod zapisany za pomocą języka IDL, a później udostępnić przygotowane pliki IDL nieograniczonej ilości mikroserwisów i na podstawie udostępnionych plików wygenerować w każdym z mikroserwisów kod lub obiekty, które zostaną użyte przez specyficzną logikę konkretnego serwisu dla osiągnięcia konkretnego celu biznesowego. Tworzenie kodu na podstawie plików IDL jest łatwo zautomatyzowane za pomocą narzędzi do budowania aplikacji, takich jak Gradle czy Maven, dlatego możemy używać IDL jako metodę współdzielenia kodu. W trakcie pracy zamierzam sprawdzić skuteczność tej metody, problemy związane z jej użyciem oraz określić przypadki, w których dobrze się nadaje, jak również przypadki, w których lepiej jej nie stosować.

Libraries – biblioteki to w odpowiedni sposób przygotowany kod, który my za pomocą odpowiednich narzędzi możemy łatwo importować i używać jako część innego programu. Program importujący bibliotekę może używać kodu biblioteki tak, jakby to był własny kod programu. Istnieje wiele narzędzi, które wspomagają łatwe i szybkie importowanie i zarządzanie bibliotekami kodu, takie jak Maven czy Gradle. W współczesnych systemach mikroserwisów współdzielenie kodu za pomocą bibliotek kodu odbywa się za pomocą serwisów do przechowywania artefaktów i plików binarnych, takich jak Nexus. [8, 5] Na pierwszym etapie narzędzie do budowania projektu przygotowuje bibliotekę i wysyła ją na serwer. Dalej kod może być przechowywany nieograniczoną ilość czasu na serwerze. Aplikacje, które mają dostęp do serwera, mogą pobrać bibliotekę z kodem i przechowywać w lokalnym systemie plików, używając jako część kodu źródłowego. Biblioteka może być udostępniona nieograniczonej liczbie projektów. Organizacja może ograniczać dostęp do serwera.

SDK – mechanizm działania podobny do bibliotek, różni się jedynie podejściem. W przypadku kodu SDK na serwerze udostępnia-

jącym dependencje przechowywane są jedynie obiekty, które muszą być użyte do komunikacji z innym programem lub kodem. Logika biznesowa nie została udostępniona w takim przypadku i zostaje ukryta oraz nie może zostać zmieniona. W przypadku udostępnienia SDK możemy łatwo zapewnić bezpieczeństwo kodu (użytkownicy wciągający dependencje nie widzą logiki biznesowej) oraz chronimy się przed przypadkowymi oraz niepożądanymi zmianami kodu. Pozwala to zaoszczędzić na testach manualnych oraz automatycznych kodu. Przykład, w którym możemy użyć współdzielenia kodu SDK to – mamy mikroserwis, który udostępnia API. API przyjmuje obiekty JSON, które mogą być opisane za pomocą obiektów Java. Obiekty, które pozostałe mikroserwisy w systemie mikroserwisów mogą użyć do wysłania żądań do określonego wcześniej mikroserwisu udostępniającego API, możemy udostępnić dla naszego systemu mikroserwisów jako bibliotekę. Każdy serwis, który chce wysyłać żądania do serwisu REST-owego, może pobrać bibliotekę w postaci dependencji za pomocą narzędzia do budowania i użyć przygotowane obiekty do komunikacji. Natomiast kod serwisu nie zostanie udostępniony. W ramach prac nad serwisem korzystającym z API REST-owego nie musimy testować kodu API, bo on nie został udostępniony i dlatego nie mógł ulec zmianie w trakcie pisania kodu serwisu.

REST API - Representational State Transfer Application Programming Interface jeden z najpopularniejszych podejść do komunikacji między mikroserwisami i dla współdzielenia kodu w systemach mikroserwisów. Przykładami współdzielenia kodu za pomocą REST mogą być mikroserwis do uwierzytelniania użytkowników, który może być wykorzystany przez każdy serwis w systemie mikroserwisów, tym samym logika związana z uwierzytelnieniem użytkowników jest współdzielona między elementami systemu. REST API pozwala na kompletne odseparowanie współdzielonego kawałka logiki od implementacji aplikacji, tym samym redukując problemy związane z zarządzaniem wersjami, które występują w przypadku bibliotek i SDK. Również współdzielenie kodu za pomocą REST API nie powoduje zależności i sztywnych powiązań między mikroserwisami, co sprawia, że system jest bardziej elastyczny, natomiast trudniej w takim przypadku utrzymać spójność API kontraktów w systemie. Współdzielenie kodu za pośrednictwem interfejsu API REST obejmuje udostępnianie danych lub funkcji za pomocą standardowych metod HTTP

(GET, POST, PUT, DELETE) i wymianę informacji w formatach takich jak JSON lub XML. Do zalet podejścia można odnieść dobrą skalowalność, nowe mikroserwisy mogą być łatwo dodawane i usuwane bez wpływu na cały system, również REST API pozwala na lepszy i bardziej zrozumiały podział odpowiedzialności w systemie, co prowadzi do zmniejszenia duplikacji kodu. Do wad tego podejścia możemy odnieść trudności w utrzymaniu, w przeciwieństwie do bibliotek i SDK zmiany w API nie są automatycznie wykrywane przez klientów, również różni klienci mogą równocześnie używać różnych wersji API, co wymaga mechanizmów kontroli wersji.

Serverless - Nowoczesne podejście, które pozwala na uruchamianie kodu bez bezpośredniego zarządzania infrastrukturą i sprzętem. W przypadku tego podejścia chmura obliczeniowa zarządza przydzieleniem odpowiednich zasobów, zarządza skalowaniem i utrzymaniem serwera, dając możliwość programiście skupić się na napisaniu kodu. W kontekście współdzielenia kodu, serverless pozwala na stworzenie małych, niezależnych funkcji, które mogą być łatwo wykorzystane do współdzielenia kodu w systemach mikroserwisów. Takie podejście dobrze pasuje do współdzielenia małych, często powtarzających się kawałków kodu. Technologia serverless również pozwala usprawnić współpracę między zespołami, zmniejsza duplikację kodu i upraszcza konserwację. Funkcje współdzielone za pomocą serverless mogą być udostępniane za pomocą technologii REST, co wiąże się z wszystkimi zaletami i wadami tego podejścia, albo za pomocą SDK udostępnionego przez dostawcę chmury.

5.2 Analiza metod współdzielenia kodu dla różnych typów obiektów

Po przeprowadzeniu analizy źródeł literaturowych oraz własnego doświadczenia dokonałem analizy tego, jakie metody współdzielenia mogą zostać wykorzystane dla poszczególnych typów obiektów.

Obiekty DTO (Interface Definition Languages) mogą być współdzielone za pomocą IDL, generowanie obiektów na podstawie plików IDL pozwala na standaryzację kontraktów komunikacji między mikroserwisami. Również obiekty DTO mogą być współdzielone za pomocą bibliotek, takie obiekty mogą albo bezpośrednio w bibliotekach, albo pliki IDL, na podstawie których później będą wygenerowane klasy

DTO, mogą być udostępniane za pomocą bibliotek. Takie podejście jest często wykorzystywane w procesach CI/CD. SDK też jest dobrze dopasowanym podejściem wykorzystywanym razem z obiektami DTO, udostępnienie obiektów DTO za pomocą SDK bez udostępniania bezpośrednio logiki biznesowej jest dobrą praktyką w programowaniu. Współdzielenie obiektów DTO może zapewnić to, że zakłada usługa będzie wykorzystywać spójne struktury danych, ułatwiając komunikację, zmniejszając tym samym ryzyko niespójności.

Obiekty Model, współdzielenie takich obiektów zgodnie z wcześniejszą analizą nie jest zalecane, ze względu na to, że zalecane jest wykorzystywanie nie więcej niż jednej bazy dla każdego mikroserwisu. Ze względu na to, nie można znaleźć najlepszej opcji do współdzielenia takich obiektów. Teoretycznie obiekty Model mogą być współdzielone za pomocą bibliotek, ale jak zaprezentowano w [11], może to powodować później problemy z zarządzaniem kodem i środowiskami.

Walidatory, jako obiekty zawierające w ograniczonej skali logikę programu, mogą być współdzielone za pomocą bibliotek, REST API oraz technologii Serverless. Współdzielenie walidatorów jest dobrą praktyką i jest zalecane, ponieważ pozwala na utrzymanie spójnych reguł walidacji w systemie mikroserwisów, co poprawia jakość danych, które wymieniają się elementy systemu. W przypadku współdzielenia za pomocą bibliotek, tracimy możliwość połączenia kilku mikroserwisów napisanych w różnych językach programowania w jeden system mikroserwisów, co natomiast możemy zrobić w przypadku współdzielenia za pomocą REST oraz Serverless. Serverless jest najlepszą opcją współdzielenia walidatorów, ze względu na to, że walidatory zawierają zwykle niewielką ilość logiki, która mieści się w jednej funkcji, którą idealnie nadają się do hostowania za pomocą Serverless. Również w tym przypadku nie tracimy możliwości połączenia kilku mikroserwisów napisanych w różnych językach programowania w jeden system mikroserwisów.

Serwisy, serwisami są głównym miejscem przechowywania, serwisy mogą być współdzielone za pomocą bibliotek, SDK, REST oraz Serverless. W przypadku wyboru podejścia do współdzielenia mikroserwisów należy dokonać analizy i zastanowić się. Biblioteki kodu są najbardziej uniwersalnym podejściem w przypadku serwisów, pasują do wykorzystania w przypadku współdzielenia małych metod zawierających małą ilość kodu, jak i tych większych. Podejś-

cie REST też dobrze pasuje do wszystkich form współdzielonej logiki, ale natomiast w wysoko obciążonych systemach może powodować obniżenie wydajności ze względu na rozproszenie zasobów na komunikację sieciową. Podejście Serverless dobrze się sprawdza w przypadku współdzielenia mniejszych metod, zarządzanie skomplikowaną logiką za pomocą Serverless może być skomplikowane, jak w przypadku REST, w wysoko obciążonych systemach podejście raczej trzeba zmienić na bibliotekę lub SDK.

5.3 Definicja kryterium oceny i porównania metod współdzielenia kodu

Na podstawie źródeł literaturowych, własnego doświadczenia oraz przygotowanych tstów wydajnościowych dokonałem analizy przedstawionych powyżej podejść do wspówdzieleina kodu w systemach mikroserwisów.

5.3.1 Definicja kryterium oceny pod kątem izolacji w przypadku IDL Przedstawiam kryteria które wybrałem w rezultacie analizy źródeł internetowych oraz książek takich jak [7] w których w znalezłem opis tego jak musi wyglądać architektura mikro serwisowa, wady i zalety różnych rozwiązań.

1. Conflict Rate - wskaźnik tego, jak często zmiany w plikach IDL powodują merge konflikty. Wyzwania dotyczące pracy z IDL i podejścia do ich pokonania są dobrze opisane w książce [7]. W szczegółach są opisane metody pracy z schematami IDL, między innymi narzędziem Protocol Buffer dostrzeganego przez Alphabet oraz Avro. To oznacza, że stare i nowe wersje kodu oraz stare i nowe formaty danych mogą potencjalnie współistnieć w systemie w tym samym czasie. Aby system działał płynnie, musimy zachować kompatybilność w obu kierunkach: Kompatybilność wsteczna i kompatybilność w przód. Musimy uważać na to, jaką ilość konfliktów generuje stosowanie wybranego podejścia. Podobne konflikty często są czasochłonne i nawet w przypadku nietrudnych przypadków potrafią spowolnić zespół deweloperski oraz spowodować inne, bardziej skomplikowane błędy. [7, p. 112]

2. Version Drift Occurrence - wskaźnik tego, jak często rozwiązanie powoduje użycie niezgodnych wersji IDL w różnych zespołach. Kiedy zespoły programistów pracujące nad jednym systemem zaczynają używać niezgodną wersję plików IDL, mikroserwisy w systemie mikroserwisów przestają komunikować się poprawnie, co powoduje błędy w runtime, które są bardzo trudne do wyłapania i naprawy, nieoczekiwane zachowanie systemu oraz uszkodzenie danych. W przypadku wykorzystania podejścia, które nie może zapewnić odpowiedniego poziomu kontroli nad wersjami i izolacji, rozwój i wdrażanie zmian w systemie mogą być problematyczne. W przypadku braku kompatybilności nie jest możliwe poprawne działanie systemu. [7, p. 123]
3. Build Failure Rate - wskaźnik tego, jak często zmiany w plikach IDL powodują problemy w trakcie budowania aplikacji. Dlatego w przypadku problemów z schematem wynikającym często z nieprawidłowo wybranego podejścia do zarządzania plikami IDL. [7, p. 123]
4. Deployment Rollback Frequency - wskaźnik tego, jak często zmiany w plikach IDL powodują problemy w trakcie wdrożenia aplikacji na środowiska w systemie mikroserwisów. Częste rollbacki usług w trakcie wdrożeń mogą spowodować spowolnienie wdrożenia nowych zmian i opóźnienie projektu, dlatego w trakcie wyboru podejścia do zarządzania plikami IDL musimy brać to pod uwagę. Musimy w trakcie analizy i planowania projektu wybrać podejście, które zapewni najmniejsze ryzyko błędów w trakcie próbnego wdrożenia i wycofania zmian. [7, p. 92]

5.3.2 Definicja kryterium oceny pod kątem izolacji w przypadku bibliotek, SDK, REST, serverless

W kontekście bibliotek, SDK, REST i serverless znaczenie pojęcia izolacji różni się od izolacji w kontekście IDL. W tym przypadku izolacja to zdolność systemu do odseparowania komponentów systemu i zminimalizowania wpływu zmian w jednym serwisie na pozostałe, co daje podejściu mikroserwisowemu jego wygodność i elastyczność.

W przypadku izolacji na podstawie analzy źródeł literaturowych oraz własnego doświadcznia zdefiniowałem kryteria porównania jako następujące:

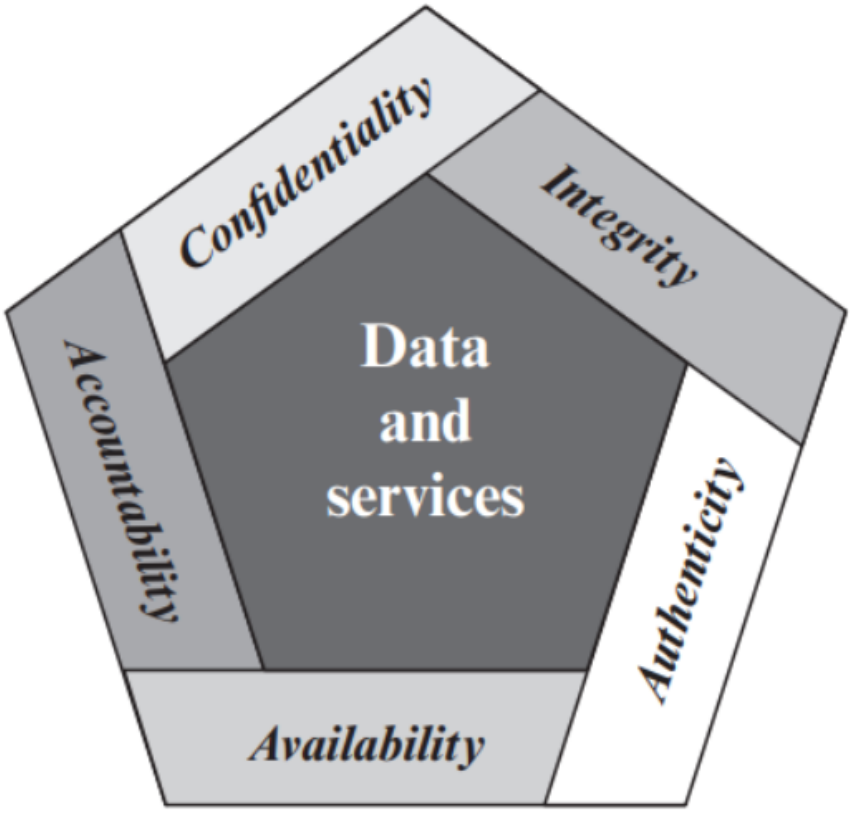
1. Izolacja interfejsów - elementy w systemie mikroserwisów muszą zapewnić jasno określone interfejsy, które pozwolą na łatwą komunikację. Zmiany w jednym elemencie systemu nie powinny wpływać na pozostałe. Za pomocą tego cytatu możemy wyjaśnić, w jaki sposób dobrze zaprojektowane interfejsy umożliwiają zmiany bez wpływu na inne części systemu. [?, p. 75]
2. Izolacja wersji - w przypadku aktualizacji, nowa wersja powinna być kompatybilna wstecznie z poprzednimi wersjami kodu. Izolacja wersji zapewnia, że stare wersje, dostarczone przez biblioteki, SDK, REST czy serverless, mogą współistnieć z nowszymi wersjami, umożliwiając stopniową aktualizację systemu. Powyższy cytat pokazuje, jak ważnym jest zachowanie kompatybilności wstecznej w przypadku współdzielenia kodu w systemach mikroserwisów. [4, p. 172]
3. Izolacja zależności - w przypadku SDK, bibliotek, REST czy serverless powinniśmy minimalizować wpływ zewnętrznych zależności na naszą aplikację. Powyższy cytat pokazuje, jak ważnym jest odseparowanie i izolacja zależności dla zachowania elastyczności. [9, p. 218]

5.3.3 Definicja kryterium oceny pod kątem bezpieczeństwa

1. Poufność (Confidentiality) - Zapewnienie tego, że poufne dane nie będą udostępnione osobom lub systemom nieupoważnionym. Zachowanie poufności jest kluczowe dla ochrony danych osobowych i korporacyjnych. [19, p. 22]
2. Integralność danych (Integrity) - Integralność danych koncentruje się na dokładności i wiarygodności informacji, chroniąc przed nieautoryzowanymi modyfikacjami, zapewniając, że dane są zarówno niezawodne, jak i spójne. [19, p. 22]
3. Dostępność (Availability) - Zapewnia to, że dane w systemie są dostępne, a różnego rodzaju zakłócenia dostępu do danych są zminimalizowane. [19, p. 22]
4. Autentyczność (Authenticity) - Gwarancja tego, że autoryzowanym użytkownikom można ufać, zapobieganie manipulacjom. [19, p. 23]
5. Odpowiedzialność (Accountability) - Definiuje zapotrzebowanie na zapisywanie w celach monitoringu działań użytkowników w

systemie w celu późniejszej analizy. W trakcie analizy można wykryć podejrzane zachowania lub naruszenia przepisów bezpieczeństwa. Pomaga w zapewnieniu tego, że podmioty naruszające przepisy prawne oraz przepisy bezpieczeństwa poniosą odpowiedzialność oraz zapobiegnie podobnym sytuacjom w przyszłości. [19, p. 23]

23



Rysunek 1. Essential Network and Computer Security Requirements.

Źródło: William Stallings, *Cryptography and Network Security*, 7. wydanie.

5.3.4 Definicja kryterium oceny pod kątem skalowalności

Czym jest skalowalność

> "Scalability – the ability to work well when the load or the number of users increases – failure handling, concurrency of components, transparency and providing quality of service." [3, p. 21]

W rozdziale 1 autorzy definiują pojęcie skalowalności jako 4 główne aspekty systemu: Controlling the cost of physical resources, Controlling the performance loss, Preventing software resources running out, Avoiding performance bottlenecks, strony 20-21, czyli Kontrola kosztów zasobów fizycznych, Kontrola utraty wydajności, Zapobieganie wyczerpywaniu się zasobów oprogramowania, Unikanie wąskich gardeł wydajności, dlatego mogę zdefiniować kryterium oceny pod kątem skalowalności jako:

1. Wpływ na koszty zasobów fizycznych - to jak wzrostu obciążenia wpływa na koszty utrzymania sprzętu.
2. Wpływ na wydajność - wydajność systemu w miarę wzrostu obciążenia.
3. Wpływ na zasoby serwera - jak program wykorzystuje zasoby serwera w miarę wzrostu obciążenia.
4. Unikanie wąskich gardeł wydajności - wiarygodność powstania wąskich gardeł w miarę wzrostu obciążenia systemu.

5.3.5 Definicja kryterium oceny pod kątem wersjonowania

Za pomocą źródeł literaturowych oraz własnego doświadczenia zdefiniowałem kryteria oceny pod kątem wersjonowania jako:

1. **Kompatybilność wsteczna** - Zapewnienie, że starsze wersje systemu lub komponentów nadal będą działały poprawnie, mimo wprowadzania nowych wersji. Kompatybilność wsteczna pomaga zapewnić funkcjonowanie istniejących systemów mikroserwisów w trakcie wprowadzania zmian i rozwoju systemu, pozwala przejść z kosztownych i bardzo skomplikowanych upgrade'ów systemu w całości za raz, a podzielić prace na mniejsze, łatwiejsze do zarządzania kawałki. [?, p. 75]

2. **Łatwość migracji** - Prostota przejścia między wersjami, umożliwienie łatwego przechodzenia między różnymi wersjami plików lub interfejsów, aby dostosować się do zmian bez wprowadzania zakłóceń w pracy systemu. Możliwość zmniejszenia przestojów aplikacji w trakcie prac serwisowych i możliwość płynnego przejścia na nową wersję. Proste migracje przyspieszają proces rozwoju aplikacji i zmniejszają ilość błędów. [**?**, p. 78]
3. **Zarządzanie historią wersji** - Możliwość przechowywania i uzyskiwania dostępu do poprzednich wersji plików. Historia wersji pozwala programistom szybko i łatwo przywrócić poprzednią wersję w przypadku powstania błędu w nowej wersji. Pozwala na zmniejszenie utraty danych i niespójności w trakcie rozwoju aplikacji. Posiadanie historii wersji zapewnia to, że projekty pozostają łatwe w utrzymaniu i skalowalne. [17, p. 150]
4. **Śledzenie zmian** - Zdolność systemu do rejestrowania i monitorowania wszystkich zmian. Możliwość śledzenia zmian zapewnia to, że każda zmiana w systemie będzie zapisana, wspierając tym programistów w zrozumieniu problemów i błędów albo pozwalając cofnąć zmiany. Poprawia współpracę między zespołami, wykazując kto, kiedy i dlaczego wprowadzał zmiany, co zwiększa również odpowiedzialność. W branżach regulowanych prawem taka audytowalność systemu może być wymagana przez prawo. [17, p. 150]

5.4 Ocena i porównanie metod współdzielenia kodu

5.4.1 Ocena wspówdzieleina kodu za pomocą Interface definition languages

Ocena pod kątem komunikacji – zapewnia potrzebny poziom komunikacji, obiekty są generowane na podstawie języka definicji interfejsu (IDL) na etapie uruchamiania aplikacji. Dalej, w trakcie wykonywania logiki programu, takie obiekty mogą być używane przez program bez obciążenia wydajnościowego. Obiekty wygenerowane na podstawie plików IDL mogą być wykorzystane do komunikacji asynchronicznej, takich jak brokery wiadomości, jak również do wysyłania bezpośrednich zapytań między serwisami.

Metody zarządzania plikami idl w systemach :

1. SVN systemy takie jak, na przykład GIT - Rozwiązania posiada następujące korzyści: łatwe osiągnięcie wstecznej kompaty-

bilności, ułatwienie osiągnięcia jednego źródła prawdy w systemie mikroserwisów, łatwiejsze zapewnienie tego, że wszystkie usługi korzystają z najnowszej wersji IDL. Do minusów rozwiązania możemy odnieść to, że system SVN może szybko stać wąskim gardłem w przypadku wykorzystania przez wiele zespołów, wymaga zarządzania i utrzymania.

2. Dystrybucja za pomocą bibliotek - Kompilowanie plików IDL do współdzielonych bibliotek, i publikacja w współdzielonym repozytorium pakietów, na przykład Nexus. Do korzyści tego rozwiązania możemy odnieść łatwą integrację z procesami CI/CD, możliwość korzystania z wersjonowanych artefaktów bez bezpośredniej interakcji z surowymi plikami IDL oraz to, że różne języki programowania mogą wymagać różnych bibliotek, co zwiększa złożoność systemu.
3. Podejście Monorepo - Przechowywanie wszystkich mikroserwisów i powiązanych z nimi plików IDL w jednym repozytorium, co zapewni, że wszystkie zespoły będą pracować na tej samej bazie kodu. Do korzyści tego rozwiązania możemy odnieść to, że pliki IDL w implementacjach usług są zawsze zsynchronizowane oraz to, że takie podejście upraszcza zarządzanie zależnościami, ponieważ cały kod przechowywany jest w jednym miejscu. Do wad podejścia mogę odnieść problemy ze skalowalnością, zwiększenie ilości kodu oraz to, że często wymaga narzędzi do zarządzania dużymi bazami kodu.
4. Repozytoria z automatyczną synchronizacją - Repozytoria z automatyczną synchronizacją to specyficzny rodzaj repozytoriów, w których zmiany wprowadzone w głównym repozytorium są automatycznie propagowane do pozostałych subrepozytoriów. Dzięki temu każdy uczestnik projektu ma dostęp do wszystkich wersji kodu. Do zalet tego podejścia możemy zaliczyć możliwość scentralizowanego wymuszania wersjonowania oraz łatwy dostęp do wszystkich wersji plików. Do minusów możemy zapisać dość trudne utrzymanie oraz złożoność w zarządzaniu synchronizacją i konfliktami w repozytorium.
5. IDL as a Service (IDLaS) - Przygotowana wcześniej usługa, która obsługuje pliki IDL na żądanie. Ta usługa może wersjonować, weryfikować i dostarczać pliki IDL. Do korzyści tego podejścia możemy odnieść dostęp na żądanie do plików IDL, możliwość dy-

namicznego wersjonowania. Do minusów możemy doliczyć to, że takie podejście wymaga zbudowania i utrzymania dodatkowej usługi w systemie oraz to, że komunikacja sieciowa może powodować opóźnienia w dostarczaniu plików IDL.

Ocena pod kątem izolacji - ocena pod kątem izolacji zależy od implementacji zarządzania plikami IDL, dla każdego z powyżej opisanego podejścia dokonałem analizy.

Ze względu na to, że mamy wiele kryteriów oceny i wiele możliwych podejść do współdzielenia plików IDL, postanowiłem stworzyć tabelę porównawczą, która pomoże w czytelnym zaprezentowaniu wyników analizy.

Dane do porównania zostały zebrane z źródeł literaturowych. [12] , [7]

Tablica 1. Porównanie podejść do współdzielenia kodu w systemach mikroserwisów

Kryteria	**SVN/Git Systemy**	**Biblioteki**	**Podejście Monorepo**	**Repo z Auto-Synchro.**	**IDL as a Service (IDLaS)**
Conflict Rate	Wysoki: Merge konflikty są powszechne w przypadku wykorzystania przez kilkoma zespołami jednocześnie. Rozwiązanie takich konfliktów może być zasobochłonne	Umiarkowany: Konflikty są mniej powszechne, ale wciąż mogą wystąpić.	Niski: Zespoły pracują na tej samej bazie kodu, dlatego konflikty zdarzają się rzadko, mogą wystąpić w trakcie pracy na tej samej usłudze.	Umiarkowany: Konflikty mogą wystąpić, dlatego w przypadku gdy synchronizacja między usługami nie jest dobrze zarządzana.	Niski: Konflikty zdarzają się rzadko, dlatego, że IDL są dostarczane na żądanie a wersjonowanie jest wymuszane.
Version Drift Occurrence	Umiarkowana: Może wystąpić jeśli zespoły programistów nie aktualizują się do nowszej wersji regularnie.	Niski: Wiarygodność jest minimalna, ponieważ biblioteki są wersjonowane i dystrybuowane za pomocą odpowiednich narzędzi.	Niski: Bazy kodu w tym przypadku baza kodu jest synchronizowana co minimalizuje problemy z wersjonowaniem.	Umiarkowany: Problemy mogą wystąpić, jeśli synchronizacja nie jest dobrze utrzymana.	Bardzo Niski: Dynamiczne zarządzanie wersjami po stronie usługi zmniejsza wiarygodność problemów z wersjonowaniem.
Build Failure Rate	Wysoki: Błędy w trakcie kompilacji mogą wystąpić w przypadku niekompatybilnych usług.	Umiarkowana: Awarie zdarzają się rzadko, możliwe są z przypadku nieprawidłowych wersji zawartych w bibliotece.	Umiarkowana: Błędy w trakcie kompilacji zdarzają się rzadko, ze względu na synchronizację kodu.	Umiarkowana: Błędy w trakcie kompilacji zdarzają się rzadko, możliwe w przypadku niedysynchronizacji repozytorium.	Niska: Minimalne ryzyko błędów w trakcie kompilacji ze względu na wymuszaną synchronizację.
Deployment Rollback Frequency	Wysoka: Częste zmiany w plikach IDL mogą spowodować częste wycofanie zmian w trakcie wdrożenia.	Umiarkowana: Wycofania zmian w trakcie wdrożenia mogą wystąpić, jeśli wystąpią problemy z kompatybilnością wsteczną.	Umiarkowana: Wycofania zmian w trakcie wdrożenia są rzadsze ze względu na synchronizację, ale problemy mogą wystąpić.	Niska: Wycofania zmian w trakcie wdrożenia są rzadkie, automatyczna synchronizacja minimalizuje potrzebę wycofywania zmian.	Niska: Wycofania zmian w trakcie wdrożenia są rzadkie. Zarządzanie IDL na żądanie zmniejsza częstotliwość wycofywania.

Ocena pod kątem bezpieczeństwa:

Tablica 2: Porównanie SVN, bibliotek, Monorepo i IDLaS pod kątem bezpieczeństwa.

Kryteria	**Zalety**	**Wyzwania**
SVN		
Poufność	Zaawansowane mechanizmy kontroli dostępu wspierają w zapewnieniu poufności	Wymaga odpowiedniej dodatkowej konfiguracji
Integralność	Pełna i przejrzysta historia zmian zapewnia integralność danych	Problemy z integralnością mogą wystąpić w przypadku, gdy zmiany nie zostaną prawidłowo zsynchronizowane
Dostępność	Scentralizowane repozytorium dostępne z wielu lokalizacji	Awaria serwera centralnego może ograniczyć dostęp
Autentyczność	Uwierzytelnianie użytkowników kontroluje dostęp do repozytorium	Nieprawidłowa konfiguracja uwierzytelniania może prowadzić do problemów z bezpieczeństwem
Odpowiedzialność	Za pomocą uwierzytelniania można kontrolować dostęp do repozytorium	Proces przeprowadzenia audytu w systemie musi być prawidłowo skonfigurowany
Dystrybucja za pomocą bibliotek (Nexus)		
Poufność	Uwierzytelnianie i mechanizmy kontroli dostępu pomagają chronić pliki IDL	Nieprawidłowa konfiguracja może prowadzić do nieautoryzowanego udostępniania

Integralność	Kontrola wersji artefaktów zapewnia integralność	Brak wersjonowania może spowodować problemy
Dostępność	Scentralne przechowywanie artefaktów zapewnia dla nich dostępność	Wymaga dodatkowej konfiguracji do prawidłowego działania
Autentyczność	Mechanizmy podpisywania artefaktów pomagają w znalezieniu zaufanych źródeł	Wymaga prawidłowej konfiguracji
Odpowiedzialność	Nie wszystkie rozwiązania zapewniają audyt i monitorowanie działań użytkowników	Właściwe logowanie zewnętrzne jest niezbędne do rozliczalności
Podejście Monorepo		
Poufność	Scentralizowana kontrola dostępu pomaga poprawić bezpieczeństwo	Zarządzanie dostępem w dużych repozytoriach potrafi być skomplikowane
Integralność	Scentralizowane zarządzanie zmianami pomaga w zapewnieniu spójności	Są różne wyzwania związane z synchronizacją w dużych zespołach
Dostępność	Scentralizowane repozytorium umożliwia łatwy dostęp do treści repozytorium	Wymaga solidnie postawionej infrastruktury do użytku na dużą skalę
Autentyczność	Kontrola dostępu na wyższym poziomie w organizacji pomaga w zapewnieniu bezpieczeństwa	Zarządzanie uprawnieniami w dużych zespołach jest trudne i może doprowadzić do błędów

Odpowiedzialność	Scentralizowane procesy biznesowe umożliwiają pełne śledzenie zmian	W dużych organizacjach trudniej znaleźć odpowiedzialnego za błąd w rezultacie audytu
IDL as a Service (IDLaS)		
Poufność	Bezpieczne punkty końcowe API z uwierzytelnianiem i szyfrowaniem zapewniają poufność	Wymaga konfiguracji bezpieczeństwa i stałego monitorowania dostępu
Integralność	Systemy kontroli wersji i automatyczna dokumentacja mogą pomóc w zapewnieniu integralności zmian	Nieprawidłowe zarządzanie wersjami może prowadzić do problemów z integralnością
Dostępność	Zapewnia wysoką dostępność dzięki automatycznemu zarządzaniu zasobami	Zależność od stabilnego połączenia sieciowego
Autentyczność	Mechanizmy uwierzytelniania, autoryzacji i certyfikatów cyfrowych wspomagają uwierzytelnienie	Wymaga regularnego monitorowania systemu bezpieczeństwa
Odpowiedzialność	Szczegółowe logi umożliwiają precyzyjne śledzenie zmian	Integracja z systemami zarządzania logami wymaga konfiguracji

Ocena pod kątem wersjonowania:

Na podstawie analizy źródeł literaturowych oraz własnych doświadczeń dokonałem analizy podejścia do współdzielenia kodu za pomocą plików IDL. W tym przypadku ocena pod kątem wersjonowania różni się w zależności od przyjętego podejścia do zarządzania plikami IDL, dlatego przeprowadziłem porównanie dla każdego z tych podejść.

Tablica 3: Porównanie podejść do zarządzania kodem, wersjonowania i repozytoriami: Git, biblioteki, Monorepo i IDLaS.

Zaleta	Opis
1. SVN Systemy (Git)	
śledzienie zmian	– Każda zmiana jest rejestrowana w postaci commitu z timestampem i informacjami wskazującymi autora. – Rozgałęzianie i tagowanie umożliwiają izolowane przygotowanie paczki zmian i później oznaczanie stabilnych wersji. – Pełna historia commitów umożliwia przeprowadzani audytów.
Rozgałęzianie i tagowanie	– Rozgałęzienie pozwala zespołom na przygowtoanie nowych wersji oprogramowania w izolacji, nie prxerywając pracy innych zespołów. – Tagi dają możliwości oznaczania stabilnych wersji, zapewniając łatwą identyfikację wersji gotowych do użytku.
Audytowalność	– Pełna historia commitów jest zawsze przechowywana, co umożliwia audyty kodu. – Zapewnia odpowiedzialnosc za wprowadzone zmniany i usprawnia współpracę w zespołów.
2. Biblioteki	
Wersjonowanie semantyczne	– Artefaktom przypisywane są numery. – Do każdej wersji dołączone są "Release notes". – Repozytoria przechowują pełną historię wersji.
Dzienniki zmian	– Do każdego wydania dołączona szczegółowa dokumentacja. – Takie changelogi zmian zapewniają uporządkowaną historię modyfikacji,. – Takie podejście zapewnia to przejrzystość śledzenia zmian.

Zaleta	opis
Centralne repozytorium artefaktów	– Centralne repozytorium przechowuje kompletną historię wersji. – Dzięki temu zespoły mogą, w razie potrzeby, pobierać, porównywać i przywracać wcześniejsze stabilne wersje.
3. Podejście Monorepo	
Zunifikowany dziennik zmian	– Wszystkie mikrserwisy w systemie i powiązane z nimi pliki IDL znajdują się w jednym repozytorium. – Historia commitów zapewnia całościowy widok zmian.
Spójność	– Wszystkie zmiany są wprowadzane do jednego repozytorium, co zapewnia synchronizację. – Takie podejście minimalizuje ryzyko różnic w wersjach pomiędzy usługami. – Spójne wersjonowanie poprawia stabilność systemu.
Łatwość cofanięcia	– Scentralizowana historia ułatwia powrót do poprzedniego stabilnego stanu. – Takie podejście ułatwia debugowanie i zwiększa stabilność systemu.
4. IDL as a Service (IDLaS)	
Zintegrowana kontrola wersji	– Dostarczana usługa obejmuje wbudowaną kontrolę wersji. – Dostarczana usługa daję dostęp do histoii wersji za pomocą API.
Automatyczna dokumentacja	– IDLaS często udostępniają interfejsy webowe. – Ze względu na to, że IDL jest udostępniane jako usługa, każdy użytkownik może uzyskać dostęp do historii wersji za pomocą API

5.4.2 Ocena współdzieleina kodu za pomocą bibliotek Ocena pod kątem komunikacji:

Pod kątem komunikacji podejście zapewnia potrzebny poziom komunikacji, obiekty są dodawane do projektu przez narzędzie do budowania projektów na etapie uruchamiania aplikacji, dalej w trakcie wykonania logiki programu takie obiekty mogą być użyte przez program bez żadnych obciążeń wydajnościowych. Obiekty wciągnięte z biblioteki mogą być użyte dla komunikacji asynchronicznej, takich jak brokery wiadomości, jak również dla wysłania bezpośrednich zapytań między serwisami.

Ocena pod kątem izolacji:

Zgodnie z wcześniej zaprezentowanymi kryteriami oceny, w przypadku oceny izolacji bibliotek, kryteria oceny są następujące: ocena pod kątem izolacji interfejsów, izolacji wersji oraz izolacji zależności.

Izolacja interfejsów: Dobrze ustrukturyzowana biblioteka izoluje skomplikowane elementy logiki od swoich użytkowników. Gdy są użyte prawidłowo, biblioteki pozwalają na zmianę lub aktualizację kodu przy minimalnym wpływie na kod wykorzystujący bibliotekę. [2, p. 75]

Izolacja wersji: Biblioteki zazwyczaj używają semantycznego wersjonowania za pomocą nazwy. Przejrzysta historia wersji umożliwia deweloperom odpowiednie zarządzanie zależnościami. Przy odpowiedniej izolacji wersji wiele wersji biblioteki może współistnieć i być używane jednocześnie, ułatwia to proces stopniowej migracji. [4, p. 172]

Izolacja zależności: Biblioteki często zależą od innych bibliotek, w przypadku gdy zależności nie są prawidłowo izolowane mogą prowadzić do konfliktów i innych problemów. Dobre praktyki zarządzania zależnościami pomagają zapewnić, że zależności biblioteki nie będą miały negatywnego wpływu na aplikację. [9, p. 218]

Ocena pod kątem bezpieczeństwa:

Zgodnie z wcześniej zaprezentowanymi kryteriami oceny, w przypadku oceny bezpieczeństwa bibliotek kryteria oceny są następujące: Poufność, Integralność danych, Dostępność, Autentyczność, Odpowiedzialność.

Poufność: Biblioteki, które są wykorzystywane do przetważania poufnych danych muszą pochodzić z sprawdzonych źródeł i ciągle aktualizowane, aby zapobiec utracie danych. [2, p. 82]

Integralność danych: Biblioteki często korzystają z sum kontrolnych, certyfikatów, podpisów cyfrowych aby mieć pewność, że przetwarzane dane pozostają bezpieczne. [2, p. 84]

Dostępność: Dostępność może zostać naruszona, jeśli biblioteki zewnętrzne będą niedostępne, lub wewnętrzne w przypadku awarii narzędzia do zarządzania bibliotekami. Dlatego ważne jest wykorzystanie menedżerów zależności które są zaufane. [2, p. 85]

Autentyczność: W przypadku bibliotek jest zapewniana za pomocą sprawdzenia jej pochodzenia za pomocą podpisów cyfrowych i sum kontrolnych dostarczonych przez wydawcę. Zapobiega to ryzyku włączenia wykonania złośliwego kodu po stronie klienta biblioteki. [2, p. 87]

Odpowiedzialność: W przypadku bibliotek przejrzysta historia może zostać zapewniona za pomocą menedżerów pakietów, które umożliwiają śledzenie pochodzenia zmian. Wspomaga to w zapewnieniu odpowiedzialności w przypadku problemów. [2, p. 88]

Ocena pod kątem wersjonowania:

> "When a data format or schema changes, ... careful management of schema evolution is required to ensure that old and new versions can coexist. This is best achieved by maintaining a structured version history that records every change in the schema in a clear, traceable manner." [7, p. 111]

5.4.3 Ocena współdzieleina kodu za pomocą SDK Ocena pod kątem komunikacji: zapewnia potrzebny poziom komunikacji, mechanizm współdzielenia jest podobny do biblioteki, tak samo jak w przypadku biblioteki obiekty są dodawane do projektu przez narzędzie do budowania projektów na etapie uruchamiania aplikacji, dalej w trakcie wykonania logiki programu takie obiekty mogą być użyte przez program bez żadnych obciążeń wydajnościowych. Obiekty wciągnięte z biblioteki mogą być użyte dla komunikacji asynchronicznej takich jak brokery wiadomości jak również dla wysłania bezpośrednich zapytań między serwisami.

Ocena pod kątem izolacji:

Zgodnie z wcześniej zaprezentowanymi kryteriami oceny, w przypadku oceny izolacji SDK kryteria oceny są następujące: ocena pod

kątem izolacji interfejsów, izolacja wersji, izolacja zależności. Jednak są one trudniejsze w projektowaniu i wymagają ostrożności w trakcie zarządzania. [5, p. 75]

Izolacja wersji: W przypadku SDK, które agregują kilka bibliotek i narzędzi, muszą zarządzać wersjonowaniem wszystkich zawartych komponentów. W idealnym przypadku zestaw SDK jest zaprojektowany tak, aby nowe dodane wersje były kompatybilne wstecz, umożliwiając klientom działanie bez natychmiastowych zmian.

Izolacja zależności: SDK często zawierają wiele bibliotek w sposób, co oznacza, że często zarządzają dużą ilością zależności. Izolowanie tych zależności jest krytyczne, aby uniknąć konfliktów. [9, p. 218]

Ocena pod kątem bezpieczeństwa:

Zgodnie z wcieśniej zaprezentowanymi kryteriami oceny, w przypadku oceny bezpieczeństwa bibliotek kryteria oceny są następujące: Poufność, Integralność danych, Dostępność, Autentyczność, Odpowiedzialność

Poufność: SDK muszą zapewnić, że wszystkie zawarte komponenty przestrzegają ścisłych zasad kontroli dostępu zmniejąc to ryzyko nieautoryzowanego dostępu do danych. [10, para 4]

Integralność danych: SDK wymaga, aby były one kompleksowo testowane, aby zapewnić, że zmiany nie naruszą integralności danych. Różne frameworki do testowania bezpieczeńswa SDK pomagają w wykrywaniu wszelkich podatności po modyfikacji. [10, para 5]

Dostępność: SDK muszą być zaprojektowane tak, aby działały zawsze, nawet przy dużym obciążeniu. [10, para 6]

Autentyczność: Autentyczność w przypadku SDK może zostać zapewniona za pomocą weryfikacji dostawców i stosowania środków kryptograficznych do walidacji ich komponentów. Warto dokonywać takich działań, aby zapobiec wykonaniu złośliwego kodu w naszym systemie. [10, para 7]

Odpowiedzialność: SDK zazwyczaj posiadają różne mechanizmy zarządzania wersjami. Takie metody zarządzania wersjami umożliwiają deweloperom śledzenie pochodzenia wszelkich zmian lub problemów. Pomaga to zapewnić odpowiedzialność programistów za przygotowane zmiany. [10, para 7]

Ocena pod kątem wersjonowania:

SDK mogą zawierać wiele bibliotek i narzędzi, z których każde może mieć własną historię wersji. Wersjonowanie SDK potrafi być

złożone, ponieważ musimy uwzględniać zmiany we wszystkich wewnętrznych komponentach, prowadzi to do bardziej złożonej historii wersji, która wymaga dobrej dokumentacji. [10, para. 3]

Przywracanie kodu do stabilnej wersji: Wycofywanie zmian w przypadku SDK może być trudne, ponieważ SDK może integrować wiele komponentów. Ta złożoność może sprawić, że wycofywanie jest bardziej czasochłonne i podatne na błędy, jeśli nie jest zarządzane prawidłowo. [10, para. 3]

Utrata danych i niespójność: SDK są narażone na większe ryzyko wewnętrznych niespójności i problemów w przypadku, gdy moduły są aktualizowane asynchronicznie. Może to doprowadzić do błędów, które mogą spowodować utratę danych lub uszkodzenie danych, jeśli nie są w odpowiedni sposób zarządzane. [10, para. 3]

5.4.4 Ocena współdzieleina kodu za pomocą API Ocena pod kątem komunikacji:

zapewnia podstawowy poziom komunikacji poprzez standardowy protokół HTTP/HTTPS, mechanizm współdzielenia opiera się na wywoływaniu endpointów API poprzez żądania HTTP. Komunikacja odbywa się poprzez wymianę dokumentów JSON/XML między serwisami, możliwe są zarówno wywołania synchroniczne (bezpośrednie żądania) jak i asynchroniczne (poprzez kolejki i brokery wiadomości). Wymaga dodatkowej konfiguracji i obsługi serializacji/deserializacji danych.

Ocena pod kątem izolacji:

Zgodnie z wcześniej zaprezentowanymi kryteriami oceny, w przypadku oceny izolacji REST kryteria oceny są następujące: ocena pod kątem izolacji interfejsów, izolacja wersji, izolacja zależności.

Izolacja interfejsów: REST API zapewnia jasno zdefiniowane punkty końcowe i kontrakty. Interfejsy są wersjonowane poprzez URL lub nagłówki HTTP, zmiana implementacji wewnętrznej nie wpływa na kontrakt API. Wymagane jest dokładne dokumentowanie API (np. poprzez Swagger/OpenAPI). [15, p. 124]

Izolacja wersji:

Możliwość utrzymywania wielu wersji API równolegle. Wersjonowanie może być realizowane poprzez URL (np. /api/v1/, /api/v2/). Konieczność obsługi kompatybilności wstecznej między wersjami. Złożoność zarządzania rośnie z każdą nową wersją API. [15, p. 89]

Izolacja zależności:

Serwisy są luźno powiązane, komunikują się tylko poprzez zdefiniowane interfejsy REST. Każdy serwis może być rozwijany i wdrażany niezależnie. Wymagane jest zarządzanie dostępnością i odpornością na błędy. [12, p. 156]

Ocena pod kątem bezpieczeństwa:

Zgodnie z wcześniej zaprezentowanymi kryteriami oceny, w przypadku oceny bezpieczeństwa REST API kryteria oceny są następujące: Poufność, Integralność danych, Dostępność, Autentyczność, Odpowiedzialność.

Poufność:

Konieczność implementacji mechanizmów autentykacji (np. JWT, OAuth). Wymagane szyfrowanie komunikacji poprzez HTTPS. Możliwość definiowania różnych poziomów dostępu do zasobów. [12, p. 245]

Integralność danych:

Walidacja danych wejściowych na poziomie API. Konieczność weryfikacji poprawności przesyłanych danych. Możliwość zastosowania podpisów cyfrowych dla krytycznych operacji. [12, p. 267]

Dostępność:

Konieczność implementacji mechanizmów retry i circuit breaker. Wymagane monitorowanie dostępności usług. Możliwość skalowania poziomego dla zwiększenia dostępności. [15, p. 178]

Autentyczność:

Implementacja mechanizmów uwierzytelniania i autoryzacji. Możliwość wykorzystania certyfikatów SSL/TLS. Konieczność weryfikacji źródła żądań. [12, p. 289]

Odpowiedzialność:

Możliwość śledzenia wszystkich operacji poprzez logi. Konieczność implementacji mechanizmów audytu. Identyfikacja źródła każdego żądania. [12, p. 312]

Ocena pod kątem wersjonowania:

REST API wymaga precyzyjnego zarządzania wersjami interfejsów. Konieczność utrzymywania kompatybilności wstecznej. Możliwość równoległego działania wielu wersji API. Wymagane jasne zasady deprecjacji starych wersji. Dokumentacja zmian między wersjami musi być dokładna i aktualna. [15, p. 134]

5.4.5 Ocena współdzieleina kodu za pomocą ServerLess

Ocena pod kątem komunikacji:

Zapewnia wysoki poziom komunikacji poprzez zdarzenia (events) i wywołania funkcji. Mechanizm współdzielenia opiera się na wywoływaniu funkcji bezserwerowych poprzez różne wyzwalacze (triggers), takie jak żądania HTTP, kolejki wiadomości, czy harmonogramy czasowe. Komunikacja odbywa się poprzez przekazywanie zdarzeń i danych między funkcjami, które są automatycznie skalowane w zależności od obciążenia. [16, p. 45]

Ocena pod kątem izolacji:

Zgodnie z wcześniej zaprezentowanymi kryteriami oceny, w przypadku oceny izolacji architektury serverless kryteria oceny są następujące: ocena pod kątem izolacji interfejsów, izolacja wersji, izolacja zależności.

Izolacja interfejsów:

Funkcje serverless są naturalnie izolowane, każda funkcja posiada własny, jasno zdefiniowany interfejs wejścia/wyjścia. Interfejsy są definiowane poprzez konfigurację zdarzeń i wyzwalaczy. Zmiana implementacji jednej funkcji nie wpływa na inne, o ile zachowany jest kontrakt interfejsu. [16, p. 78]

Izolacja wersji: Platformy serverless oferują wbudowane mechanizmy wersjonowania funkcji. Możliwe jest równoległe uruchamianie różnych wersji tej samej funkcji, co ułatwia wdrażanie zmian i testowanie. System aliasów i etapów (stages) pozwala na kontrolowane przełączanie między wersjami. [16, p. 112]

Izolacja zależności:

Każda funkcja serverless jest całkowicie niezależna i może mieć własne zależności. Pakiety i biblioteki są izolowane na poziomie funkcji, co eliminuje konflikty zależności. Wymaga to jednak zarządzania duplikacją kodu i zależności między funkcjami. [16, p. 156]

Ocena pod kątem bezpieczeństwa:

Zgodnie z wcześniej zaprezentowanymi kryteriami oceny, w przypadku oceny bezpieczeństwa architektury serverless kryteria oceny są następujące: Poufność, Integralność danych, Dostępność, Autentyczność, Odpowiedzialność.

Poufność:

Platformy serverless zapewniają wbudowane mechanizmy bezpieczeństwa i izolacji. Każda funkcja działa w izolowanym środowisku

(sandbox). Dostęp do zasobów jest kontrolowany przez system uprawnień i ról. [16, p. 167]

Integralność danych:

Funkcje serverless mogą wykorzystywać wbudowane mechanizmy walidacji i transformacji danych. Platformy oferują narzędzia do monitorowania i logowania wszystkich operacji. Możliwe jest wykorzystanie managed services do przechowywania i przetwarzania danych. [16, p. 189]

Dostępność:

Architektura serverless zapewnia automatyczne skalowanie i wysoką dostępność. Platformy zarządzają infrastrukturą i gwarantują określony poziom SLA. Funkcje są automatycznie replikowane w różnych strefach dostępności. [16, p. 198]

Autentyczność:

Platformy serverless oferują wbudowane mechanizmy uwierzytelniania i autoryzacji. Możliwa jest integracja z różnymi dostawcami tożsamości. Każde wywołanie funkcji może być weryfikowane pod kątem uprawnień. [16, p. 223]

Odpowiedzialność:

Platformy zapewniają szczegółowe logowanie i monitorowanie wszystkich wywołań funkcji. Dostępne są narzędzia do analizy wydajności i kosztów. Możliwe jest śledzenie całego łańcucha wywołań między funkcjami. [16, p. 245]

Ocena pod kątem wersjonowania:

Wersjonowanie w architekturze serverless jest wspierane przez wbudowane mechanizmy platform. Każda funkcja może mieć wiele wersji, które są zarządzane niezależnie. System aliasów pozwala na kierowanie ruchu do określonych wersji funkcji. Możliwe jest stopniowe wdrażanie zmian poprzez techniki takie jak canary deployments. [16, p. 267]

Przywracanie kodu do stabilnej wersji:

W przypadku problemów możliwe jest szybkie przełączenie na poprzednią stabilną wersję funkcji. Platformy serverless przechowują historię wersji i umożliwiają łatwe przełączanie między nimi. [16, p. 289]

Utrata danych i niespójność:

Ryzyko utraty danych jest minimalizowane przez wykorzystanie managed services do przechowywania stanu. Transakcyjność może

być zapewniona przez odpowiednie projektowanie przepływu danych między funkcjami. [16, p. 312]

5.4.6 Ocena współdzieleina pod kątem skalowalności Ocena podejścia IDL pod kątem skalowalności:

Systemy bazujące na IDL, takie jak Protocol Buffers oraz Avro zazwyczaj serializują dane do kompaktowych formatów binarnych, to znaczy, że ilość danych, które trzeba przesłać przez sieć i przechowywać na dysku, jest minimalna. Dzięki temu koszt zasobów fizycznych jest minimalny i ma tendencję do pozostania niskim, nawet w przypadku wzrostu obciążenia. Wydajność systemu korzystającego z współdzielenia za pomocą IDL nie zależy od metody zarządzania plikami IDL, jak w powyższych przypadkach.

W celu przygotowania analizy pod kątem skalowalności, opracowałem system, którego celem jest testowanie wydajności przykładowej aplikacji wykorzystującej podejście IDL do współdzielenia kodu oraz obserwacja, w jaki sposób to podejście wpływa na wydajność i skalowalność systemu.

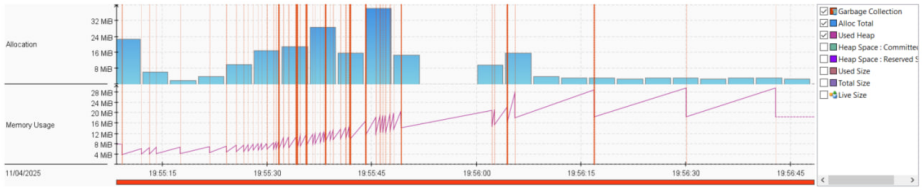
Przygotowane aplikacje przykładowe oraz kod systemu, który powstał w celu sprawdzenia i porównania wydajności, zostały dołączone do materiałów pracy.

Testowany system mikroserwisów składa się z dwóch mikroserwisów napisanych z użyciem technologii Java oraz Spring. Pierwszy mikroserwis produkuje dane, drugi je konsumuje. Oba serwisy uruchamiane są za pomocą narzędzia Docker. Kontenery Dockera mają ustawione limity zasobów, które mogą wykorzystywać.

Po uruchomieniu aplikacji w Dockerze, automatycznie uruchamiane są testy wydajnościowe przygotowane za pomocą narzędzia Gatling. Projekt został stworzony przy użyciu Javy, Springa oraz Gatlinga i znajduje się w materiałach dołączonych do pracy.

W trakcie testów wydajnościowych zbierane są metryki dotyczące użycia pamięci, liczby wątków, zużycia CPU, garbage collection i inne. Dane te są zapisywane do pliku za pomocą narzędzia Java Flight Recorder. Uruchomienie JFR odbywa się również automatycznie w momencie startu aplikacji ca poomocą skrypu, kod skryptu również znajdije sie w materiałach dołączonych do pracy.

Aplikacja testująca wydajność aplikacji przykładowej posiada trzy tryby testowania: małe obciążenie, średnie oraz duże. Zostało to zrealizowane w celu obserwacji działania aplikacji przy różnych poziomach obciążenia.



Rysunek 2. Wykres przedstawiający zużycie pamięci RAM aplikacji wykorzystującej podejście IDL pod wpływem małego obciażenia.

Źródło: opracowanie własne.

Wyniki obserwacji:
Alokacja (górny wykres — niebieskie słupki)

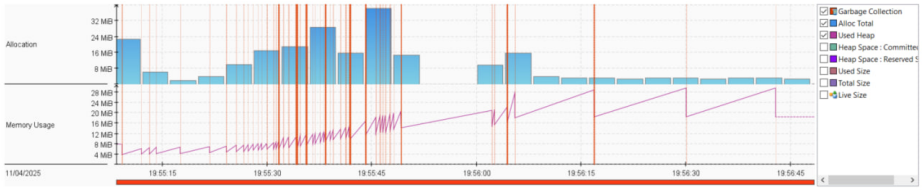
- Szczyty alokacji są bardziej okresowe i zazwyczaj utrzymują się poniżej 32 MiB.
- Występuje więcej przerw między zdarzeniami alokacji, co wskazuje na rzadsze tworzenie nowych obiektów.

Garbage collection (pionowe pomarańczowe linie)

- Ogólnie mniej zdarzeń GC.
- GC wydaje się być wyzwalane rzadziej ze względu na mniejsze obciążenie pamięci.

Wykorzystanie pamięci (dolny wykres – fioletowa linia)

- Wykorzystanie sterty rośnie powoli i konsekwentnie, osiągając maksymalnie około 24 MiB.
- Stopniowy wzrost z regularnymi małymi spadkami (GC), pokazujący stabilne wykorzystanie pamięci.



Rysunek 3. Wykres przedstawiający zużycie pamięci RAM aplikacji wykorzystującej podejście IDL pod wpływem średnigo obciażenia.

Źródło: opracowanie własne.

Wyniki obserwacji:
Alokacja (górny wykres — niebieskie słupki)

- Aktywność alokacji jest większa i bardziej spójna.
- Maksymalne wartości osiągają 48 MiB, co wskazuje na znaczny wzrost alokacji obiektów ze względu na większą liczbę żądań.
- Alokacje są utrzymywane przez dłuższy okres z krótszym czasem bezczynności między słupkami.

Garbage collection (pionowe pomarańczowe linie)

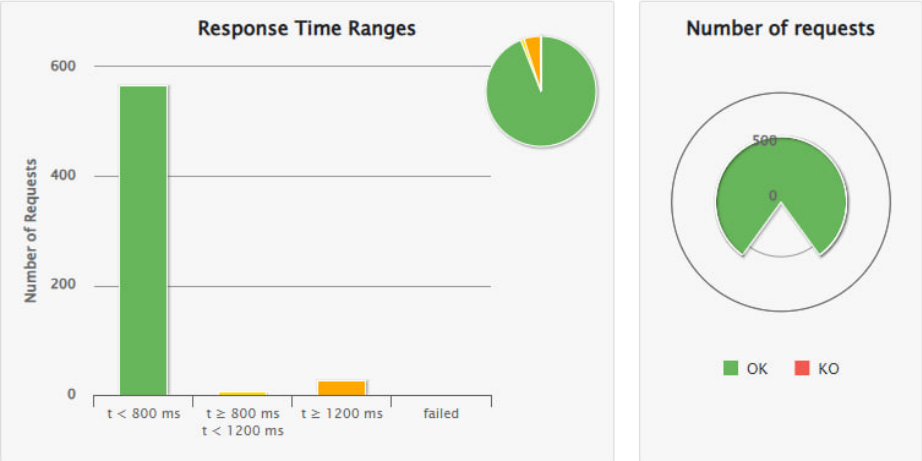
- Znacznie częstsze zdarzenia GC.
- Wskazuje na zwiększoną utratę pamięci — częstsze tworzenie i czyszczenie obiektów ze względu na dużą liczbę żądań.

Wykorzystanie pamięci (dolny wykres – fioletowa linia)

- Wykorzystanie sterty wzrasta do 64 MiB i okresowo spada.
- Wzór pokazuje częste cykle alokacji i GC, sugerując agresywne wykorzystanie pamięci i czyszczenie.

W przypadku duzego obciążenia nie było mozliwe wygenerowanie raportu ze wzgledu na Memory Error.

Raporty wygenerowane przez narzędzie Gatling:

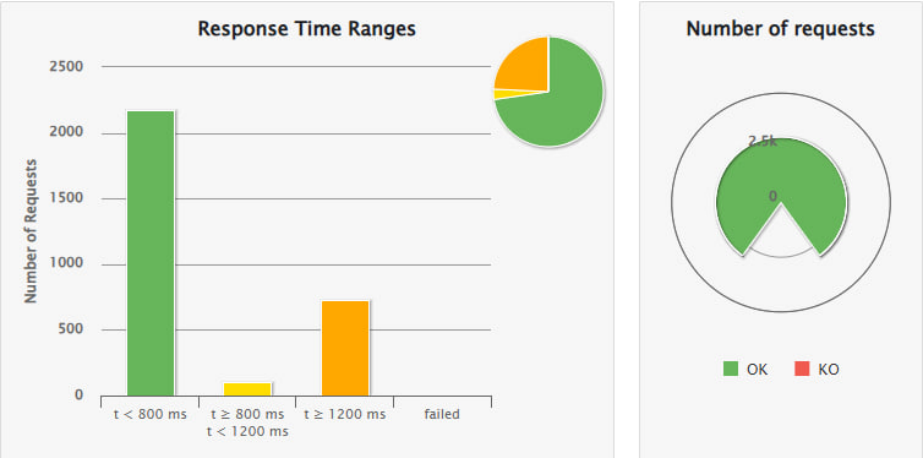


Rysunek 4. Raport wygenerowany w narzędziu Galing dla aplikacji działającej w oparciu o podejście IDL, testowanej przy niskim obciążeniu.

Źródło: opracowanie własne.

Na rysunku widać że:

- Czas odpowiedzi krótszy niż 800 ms: Zdecydowana większość żądań (około 580), oznaczona kolorem zielonym.
- Czas odpowiedzi od 800 ms do poniżej 1200 ms: Niewielka liczba żądań, przedstawiona kolorem żółtym.
- Czas odpowiedzi równy lub przekraczający 1200 ms: Również niewielka liczba żądań.
- Żądania nieudane: Brak nieudanych żądań.



Rysunek 5. Raport wygenerowany w narzędziu Galing dla aplikacji działającej w oparciu o podejście IDL, testowanej przy średnim obciążeniu.

Źródło: opracowanie własne.

Na rysunku widać że:

- Czas odpowiedzi krótszy niż 800 ms: Duża liczba żądań (około 2300), jednak proporcjonalnie mniej niż w przypadku niskiego obciążenia.
- Czas odpowiedzi od 800 ms do poniżej 1200 ms: Niewielki wzrost liczby żądań w porównaniu do niskiego obciążenia.
- Czas odpowiedzi równy lub przekraczający 1200 ms: Wyraźny wzrost – znacznie więcej żądań z długim czasem odpowiedzi.
- Żądania nieudane: Nadal brak nieudanych żądań.

Na przedstawionych wykresach widać, że zarówno w przypadku małego, jak i średniego obciążenia aplikacja zachowuje się stabilnie. Przy średnim obciążeniu zauważalne jest jednak nieznaczne wydłużenie czasu oczekiwania na odpowiedź.

Wraz ze zwiększaniem obciążenia wzrasta czas oczekiwania na odpowiedź. Porównując czas oczekiwania w przypadku różnych podejść do współdzielenia kodu, możemy ocenić efektywność poszczególnych narzędzi.

Ocena współdzielenia kodu za pomocą SDK pod kątem skalowalności:

W celu przygotowania analizy pod kątem skalowalności, tak samo jak w poprzednim przypadku opracowałem system, którego celem jest testowanie wydajności przykładowej aplikacji wykorzystującej SDK do współdzielenia kodu oraz obserwacja, w jaki sposób to podejście wpływa na wydajność i skalowalność systemu.

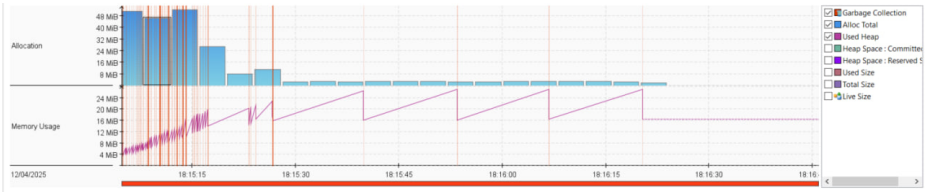
Przygotowane aplikacje przykładowe oraz kod systemu, który powstał w celu sprawdzenia i porównania wydajności, zostały dołączone do materiałów pracy.

Testowany system mikroserwisów składa się z dwóch mikroserwisów napisanych z użyciem technologii Java oraz Spring. Pierwszy mikroserwis produkuje dane, drugi je konsumuje. Oba serwisy uruchamiane są za pomocą narzędzia Docker. Kontenery Dockera mają ustawione limity zasobów, które mogą wykorzystywać.

Po uruchomieniu aplikacji w Dockerze, automatycznie uruchamiane są testy wydajnościowe przygotowane za pomocą narzędzia Gatling. Projekt został stworzony przy użyciu Javy, Springa oraz Gatlinga i znajduje się w materiałach dołączonych do pracy.

W trakcie testów wydajnościowych zbierane są metryki dotyczące użycia pamięci, liczby wątków, zużycia CPU, garbage collection i inne. Dane te są zapisywane do pliku za pomocą narzędzia Java Flight Recorder. Uruchomienie JFR odbywa się również automatycznie w momencie startu aplikacji.

Aplikacja testująca wydajność aplikacji przykładowej posiada trzy tryby testowania: małe obciążenie, średnie oraz duże. Zostało to zrealizowane w celu obserwacji działania aplikacji przy różnych poziomach obciążenia.

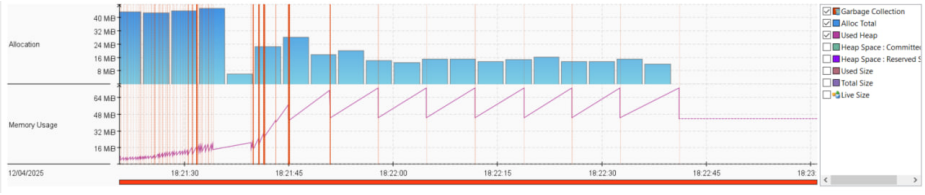


Rysunek 6. Wykres przedstawiający zużycie pamięci RAM aplikacji wykorzystującej SDK wpływem małego obciażenia.

Źródło: opracowanie własne.

Wyniki obserwacji:

- Przegląd: Ten wykres pokazuje stabilny i stosunkowo niski poziom zużycia pamięci.
- Wykorzystanie pamięci: Wykorzystanie pamięci osiąga szczyt na poziomie około 200 MB, przy średnim wykorzystaniu około 150 MB. Wykorzystanie pozostaje stale poniżej 250 MB, z niewielkimi skokami dochodzącymi do 220 MB.
- Garbage collection: Wykres wskazuje 2-3 zdarzenia zbierania śmieci w całym okresie monitorowania, oznaczone rzadkimi fioletowymi liniami. Sugeruje to wydajne zarządzanie zasobami, ponieważ aplikacja nie wykorzystuje ich intensywnie.



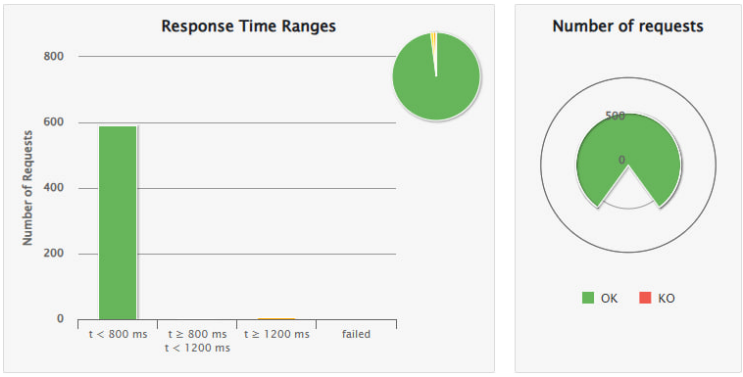
Rysunek 7. Wykres przedstawiający zużycie pamięci RAM aplikacji wykorzystującej SDK wpływem średniego obciażenia.

Źródło: opracowanie własne.

- Przegląd: Ten wykres przedstawia znaczny wzrost zużycia pamięci w miarę wzrostu obciążenia aplikacji.
- Wykorzystanie pamięci: Wykorzystanie pamięci osiąga szczyt na poziomie około 600 MB, przy średnim wykorzystaniu około 450 MB. Wykorzystanie często przekracza 500 MB w godzinach szczytu, ilustrując reakcję aplikacji na większe zapotrzebowanie.
- Garbage collection: Wykres pokazuje około 5-7 zdarzeń zbierania śmieci, oznaczonych bardziej wyraźnymi fioletowymi liniami, odzwierciedlającymi potrzebę aplikacji aktywnego zarządzania pamięcią przy zwiększonym obciążeniu.

W przypadku duzego obciążenia nie było mozliwe wygenerowanie raportu ze wzgledu na Memory Error. Wykres niskiego obciążenia sugeruje optymalną wydajność przy minimalnym obciążeniu zasobów, co jest idealne w scenariuszach, w których wydajność jest kluczowa. Wykres średniego obciążenia, pokazujący wyższe zużycie, wskazuje, że aplikacja skaluje się, aby sprostać zwiększonym wymaganiom, chociaż może to prowadzić do potencjalnych problemów z opóźnieniami, jeśli nie zostanie odpowiednio zarządzane.

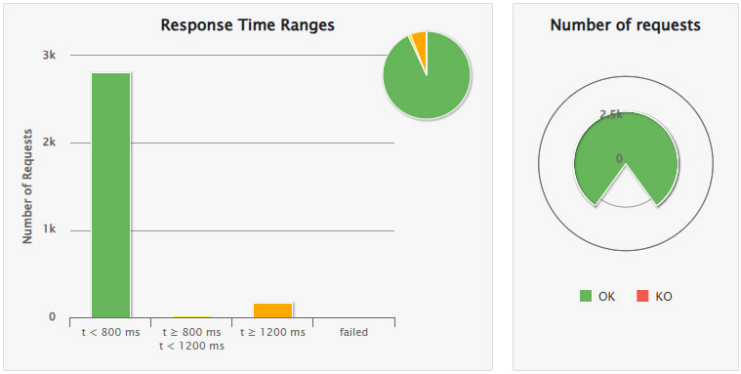
Raporty wygenerowane przez narzędzie Gatling:



Rysunek 8. Raport wygenerowany w narzędziu Gatling dla aplikacji działającej w oparciu o podejście SDK, testowanej przy niskim obciążeniu.

Źródło: opracowanie własne.

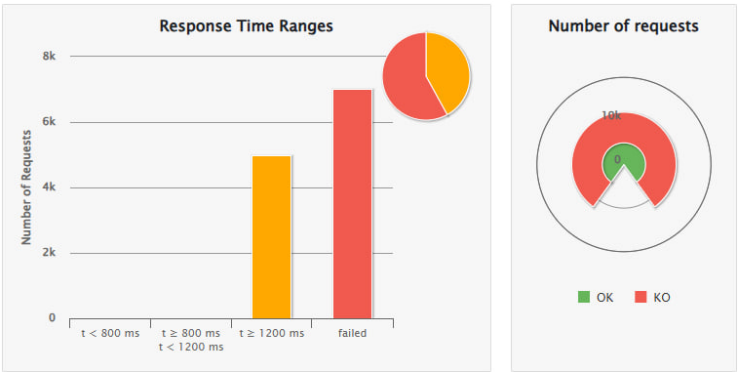
- Czas reakcji poniżej 800 ms: Około 600 żądań – bardzo dobra wydajność, większość operacji wykonywana szybko.
- Czas reakcji od 800 ms do poniżej 1200 ms: Pojedyncze żądania – praktycznie brak wpływu na ogólną wydajność.
- Czas reakcji 1200 ms i więcej: Kilka żądań – sporadyczne opóźnienia, nieistotne z punktu widzenia użytkownika.
- Nieudane żądania: Brak – aplikacja działa niezawodnie.



Rysunek 9. Raport wygenerowany w narzędziu Gatling dla aplikacji działającej w oparciu o podejście SDK, testowanej przy średnim obciążeniu.

Źródło: opracowanie własne.

- Czas reakcji poniżej 800 ms: Około 2800 żądań – wyraźna poprawa w porównaniu do testu średniego.
- Czas reakcji od 800 ms do poniżej 1200 ms: Około 100 żądań – minimalna liczba, pozytywny sygnał stabilizacji.
- Czas reakcji 1200 ms i więcej: Około 300 żądań – niewielkie opóźnienia nadal obecne, ale nie wpływają znacząco na ogólną ocenę.
- Nieudane żądania: Brak – aplikacja odzyskała stabilność i działa niezawodnie.



Rysunek 10. Raport wygenerowany w narzędziu Gatling dla aplikacji działającej w oparciu o podejście SDK, testowanej przy wysokim obciążeniu.

Źródło: opracowanie własne.

- Czas reakcji poniżej 800 ms: 0 żądań
- Czas reakcji od 800 ms do poniżej 1200 ms: 0 żądań
- Czas reakcji 1200 ms i więcej: Ponad 5000 żądań – dominująca część, silny wskaźnik przeciążenia systemu.
- Nieudane żądania: Około 7000 – aplikacja nie radzi sobie z obsługą dużej liczby żądań, występują błędy.

Porównanie wydajności:

- Aplikacja działa optymalnie przy niskim obciążeniu — szybkie odpowiedzi, brak błędów i wysoka responsywność.
- Przy średnim obciążeniu wydajność ulega znacznemu pogorszeniu — dominują opóźnienia powyżej 1200 ms oraz wysoka liczba błędów.
- Wysokie obciążenie pokazuje dalsze pogorszenie — jeszcze więcej żądań ma bardzo długi czas odpowiedzi, a liczba nieudanych żądań sięga około 8000, co świadczy o przeciążeniu systemu.

Ocena współdzielenia kodu za pomocą bibliotek pod kątem skalowalności:

Biblioteki są zazwyczaj zoptymalizowane pod kątem konkretnego zadania w jednym języku. Jednak jeśli biblioteka nie jest zaprojektowana do pracy w środowiskach o dużej współbieżności, może wymagać dodatkowych zasobów sprzętowych w miarę wzrostu obciążenia. Na przykład intensywne przetwarzanie danych w bibliotekach może powodować większe zużycie pamięci lub CPU podczas pracy z dużymi zbiorami danych. Wspierająca myśl: Badania nad wydajnością obliczeniową często zauważają, że nieoptymalizowane biblioteki mogą stać się wąskimi gardłami pod dużym obciążeniem, prowadząc do zwiększenia kosztów zasobów fizycznych. [4, p. 192]

W celu przygotowania analizy pod kątem skalowalności, tak samo0 jak w przypadku opracowałem system, którego celem jest testowanie wydajności przykładowej aplikacji wykorzystującej biblioteki do współdzielenia kodu oraz obserwacja, w jaki sposób to podejście wpływa na wydajność i skalowalność systemu.

Przygotowane aplikacje przykładowe oraz kod systemu, który powstał w celu sprawdzenia i porównania wydajności, zostały dołączone do materiałów pracy.

Testowany system mikroserwisów składa się z dwóch mikroserwisów napisanych z użyciem technologii Java oraz Spring. Pierwszy mikroserwis produkuje dane, drugi je konsumuje. Oba serwisy uruchamiane są za pomocą narzędzia Docker. Kontenery Dockera mają ustawione limity zasobów, które mogą wykorzystywać.

Po uruchomieniu aplikacji w Dockerze, automatycznie uruchamiane są testy wydajnościowe przygotowane za pomocą narzędzia Gatling. Projekt został stworzony przy użyciu Javy, Springa oraz Gatlinga i znajduje się w materiałach dołączonych do pracy.

W trakcie testów wydajnościowych zbierane są metryki dotyczące użycia pamięci, liczby wątków, zużycia CPU, garbage collection i inne. Dane te są zapisywane do pliku za pomocą narzędzia Java Flight Recorder. Uruchomienie JFR odbywa się również automatycznie w momencie startu aplikacji.

Aplikacja testująca wydajność aplikacji przykładowej posiada trzy tryby testowania: małe obciążenie, średnie oraz duże. Zostało to zrealizowane w celu obserwacji działania aplikacji przy różnych poziomach obciążenia.

Rysunek 11. Wykres przedstawiający zużycie pamięci RAM aplikacji wykorzystującej biblioteki pod wpływem małego obciążenia.

Źródło: opracowanie własne.

Wyniki obserwacji:
Alokacja górny wykres — niebieskie słupki

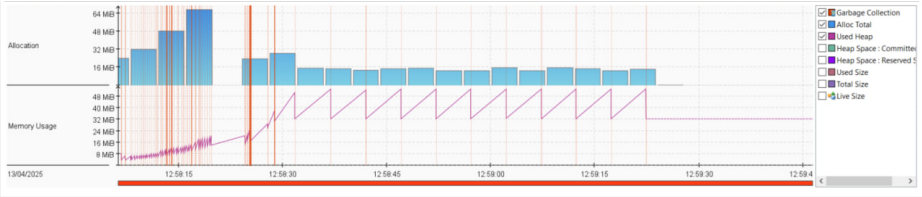
- Szczyty alokacji występują okresowo i zwykle nie przekraczają wartości 32 MiB.
- Widoczne są dłuższe przerwy pomiędzy zdarzeniami alokacji, co sugeruje rzadsze tworzenie obiektów i mniejsze zapotrzebowanie na pamięć.
- Po początkowej fazie obciążenia alokacja szybko stabilizuje się na niskim poziomie.

Garbage Collection – pionowe pomarańczowe linie

- Liczba zdarzeń GC (Garbage Collection) jest stosunkowo niewielka.
- Mechanizm odśmiecania uruchamiany jest sporadycznie, co wynika z niskiego poziomu zużycia pamięci.

Wykorzystanie pamięci, dolny wykres – fioletowa linia

- Wzrost zużycia sterty jest powolny i przewidywalny, osiągając maksymalnie około 24 MiB.
- Charakterystyczny jest łagodny wzrost z regularnymi, niewielkimi spadkami, które odpowiadają działaniu GC — co świadczy o stabilnym zarządzaniu pamięcią.



Rysunek 12. Wykres przedstawiający zużycie pamięci RAM aplikacji wykorzystującej biblioteki pod wpływem średnigo obciażenia.

Źródło: opracowanie własne.

W przypadku duzego obciążenia nie było mozliwe wygenerowanie raportu ze wzgledu na Memory Error.

1. Alokacja (górny wykres — niebieskie słupki)

Alokacja górny wykres — niebieskie słupki

- Alokacja jest wyraźnie intensywniejsza i bardziej regularna.
- Maksymalne wartości alokacji dochodzą do 48 MiB, co wskazuje na wzmożone tworzenie obiektów w wyniku większej liczby żądań.
- Alokacje są utrzymywane przez dłuższy czas, a okresy bezczynności między kolejnymi słupkami są znacznie krótsze.

Garbage Collection – pionowe pomarańczowe linie

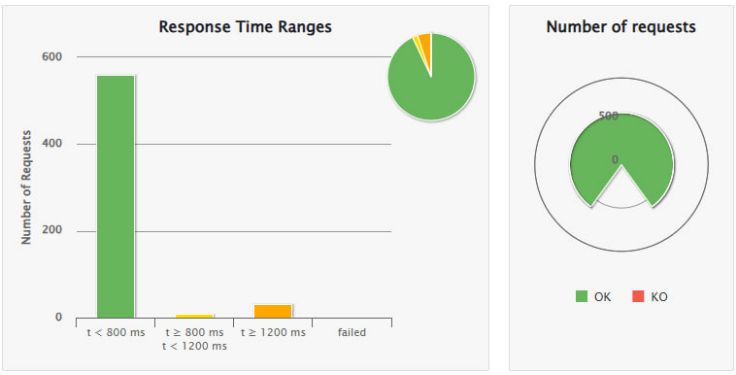
- Zdarzenia GC są znacznie częstsze i bardziej cykliczne.
- Taki wzorzec odzwierciedla wzmożone wykorzystanie pamięci — więcej obiektów jest tworzonych i usuwanych, co jest typowe przy zwiększonym ruchu w aplikacji.

Wykorzystanie pamięci, dolny wykres – fioletowa linia

- Użycie sterty wzrasta dynamiczniej i osiąga wartości nawet do 64 MiB.
- Fioletowa linia ukazuje klasyczny wzór zęba piły, który sugeruje intensywne cykle alokacji i czyszczenia pamięci.
- System zarządzania pamięcią reaguje odpowiednio na zwiększone obciążenie, ale wymaga częstszych interwencji GC.

Podsumowując, aplikacja wykazuje stabilne i efektywne zarządzanie pamięcią przy niskim i średnim obciążeniu, jednak załamuje się przy wysokim obciążeniu, co może oznaczać ograniczenia skalowalności.

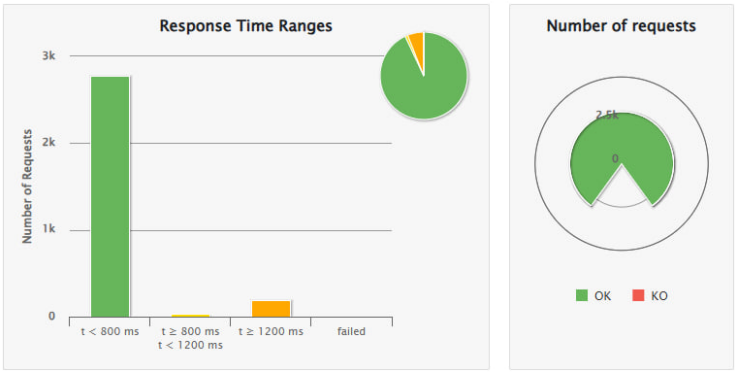
Raporty wygenerowane przez narzędzie Gatling:



Rysunek 13. Raport wygenerowany w narzędziu Gatling dla aplikacji działającej w oparciu o biblioteki testowanej przy niskim obciążeniu.

Źródło: opracowanie własne.

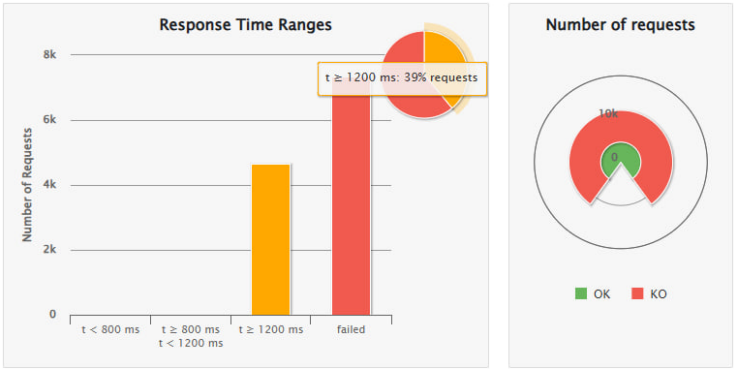
- Czas reakcji poniżej 800 ms: Około 570 żądań – dobra wydajność, większość operacji wykonywana szybko.
- Czas reakcji od 800 ms do poniżej 1200 ms: Pojedyncze żądania – praktycznie brak wpływu na ogólną wydajność.
- Czas reakcji 1200 ms i więcej: Kilka żądań – sporadyczne opóźnienia, nieistotne z punktu widzenia użytkownika.
- Nieudane żądania: Brak – aplikacja działa niezawodnie.



Rysunek 14. Raport wygenerowany w narzędziu Gatling dla aplikacji działającej w oparciu o biblioteki testowanej przy średnim obciążeniu.

Źródło: opracowanie własne.

- Czas reakcji poniżej 800 ms: Około 2700 żądań – wyraźna poprawa w porównaniu do testu średniego.
- Czas reakcji od 800 ms do poniżej 1200 ms: Około 10 żądań – minimalna liczba, pozytywny sygnał stabilizacji.
- Czas reakcji 1200 ms i więcej: Około 300 żądań – niewielkie opóźnienia nadal obecne, ale nie wpływają znacząco na ogólną ocenę.
- Nieudane żądania: Brak – aplikacja odzyskała stabilność i działa niezawodnie.



Rysunek 15. Raport wygenerowany w narzędziu Gatling dla aplikacji działającej w oparciu o biblioteki, testowanej przy wysokim obciążeniu.

Źródło: opracowanie własne.

- Czas reakcji poniżej 800 ms: Około 0 żądań – spadek w porównaniu do testu przy niskim obciążeniu.
- Czas reakcji od 800 ms do poniżej 1200 ms: Około 0 żądań – znaczny spadek, wskazujący na pogorszenie responsywności.
- Czas reakcji 1200 ms i więcej: Ponad 5000 żądań – dominująca część, silny wskaźnik przeciążenia systemu.
- Nieudane żądania: Około 7000 – aplikacja nie radzi sobie z obsługą dużej liczby żądań, występują błędy.

Wnioski dotyczące skalowalności:

- Dobre zachowanie przy niskim i średnim obciążeniu
- Problemy ze skalowalnością przy wysokim obciążeniu
- Widoczna granica wydajności około 2500-3000 żądań
- Potrzeba optymalizacji zarządzania pamięcią przy dużym obciążeniu

Podsumowanie: Aplikacja wykorzystująca biblioteki wykazuje wysoką efektywność przy niskim i średnim obciążeniu, jednak przy wysokim obciążeniu ujawnia problemy z zarządzaniem pamięcią.

Ocena współdzielenia kodu za pomocą REST pod kątem skalowalności:

W celu przygotowania analizy pod kątem skalowalności, tak samo jak w poprzednim przypadku opracowałem system, którego celem jest testowanie wydajności przykładowej aplikacji wykorzystującej biblioteki do współdzielenia kodu oraz obserwacja, w jaki sposób to podejście wpływa na wydajność i skalowalność systemu.

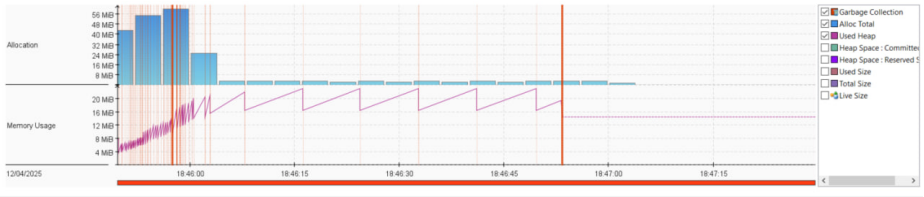
Przygotowane aplikacje przykładowe oraz kod systemu, który powstał w celu sprawdzenia i porównania wydajności, zostały dołączone do materiałów pracy.

Testowany system mikroserwisów składa się z dwóch mikroserwisów napisanych z użyciem technologii Java oraz Spring. Pierwszy mikroserwis produkuje dane, drugi je konsumuje. Oba serwisy uruchamiane są za pomocą narzędzia Docker. Kontenery Dockera mają ustawione limity zasobów, które mogą wykorzystywać.

Po uruchomieniu aplikacji w Dockerze, automatycznie uruchamiane są testy wydajnościowe przygotowane za pomocą narzędzia Gatling. Projekt został stworzony przy użyciu Javy, Springa oraz Gatlinga i znajduje się w materiałach dołączonych do pracy.

W trakcie testów wydajnościowych zbierane są metryki dotyczące użycia pamięci, liczby wątków, zużycia CPU, garbage collection i inne. Dane te są zapisywane do pliku za pomocą narzędzia Java Flight Recorder. Uruchomienie JFR odbywa się również automatycznie w momencie startu aplikacji.

Aplikacja testująca wydajność aplikacji przykładowej posiada trzy tryby testowania: małe obciążenie, średnie oraz duże. Zostało to zrealizowane w celu obserwacji działania aplikacji przy różnych poziomach obciążenia.



Rysunek 16. Wykres przedstawiający zużycie pamięci RAM aplikacji wykorzystującej podejście REST pod wpływem małego obciażenia

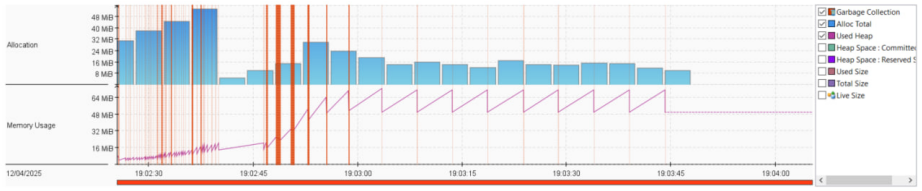
Źródło: opracowanie własne.

Wyniki obserwacji:
Alokacja górny wykres — niebieskie słupki

- Szczyty alokacji są okresowe i zazwyczaj utrzymują się poniżej 32 MiB, co wskazuje na rzadkie tworzenie obiektów.
- Dłuższe okresy bezczynności pomiędzy zdarzeniami alokacji sugerują mniejszą liczbę żądań lub lżejsze przetwarzanie.

Garbage collection - pionowe pomarańczowe linie

- Mniej zdarzeń GC, występujących sporadycznie z powodu niskiego obciążenia pamięci.
- Cykl GC prawdopodobnie prewencyjny, utrzymujący stabilną kondycję pamięci.



Rysunek 17. Wykres przedstawiający zużycie pamięci RAM aplikacji wykorzystującej podejście REST pod wpływem średnigo obciażenia

Źródło: opracowanie własne.

W przypadku duzego obciążenia nie było mozliwe wygenerowanie raportu ze wzgledu na Memory Error.

Wyniki obserwacji:

Alokacja górny wykres — niebieskie słupki

- Aktywność alokacji jest bardziej spójna, a szczyty sięgają 48 MiB.
- Krótsze przerwy między alokacjami wskazują na ciągłe tworzenie obiektów przy zwiększonej liczbie żądań.
- Brak najwyższego poziomu alokacji 64 MiB sugeruje umiarkowane, ale nie ekstremalne obciążenie.

- Powolny, stabilny wzrost pamięci sterty do około 24 MiB, z niewielkimi spadkami po GC.
- Odzwierciedla przewidywalne zapotrzebowanie na pamięć i efektywne zarządzanie obiektami.

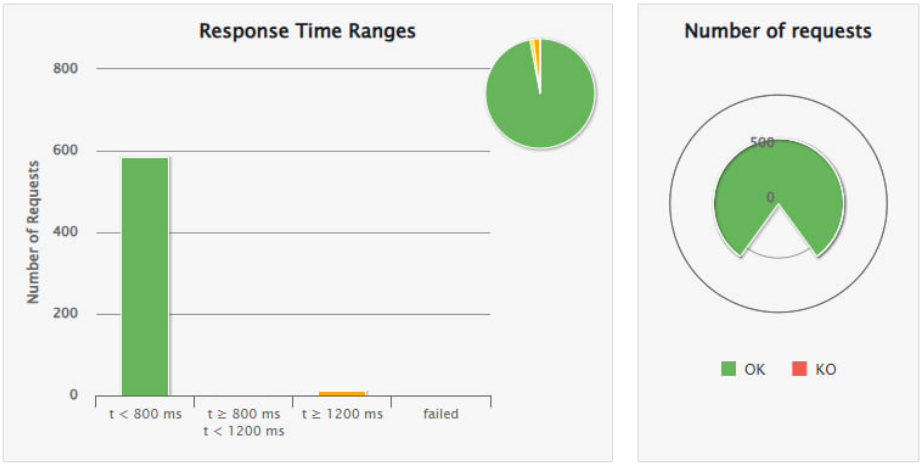
Garbage collection - pionowe pomarańczowe linie

- Częstsze zdarzenia GC, skorelowane ze zwiększoną liczbą alokacji.
- Wskazuje na większą rotację obiektów tworzenie/niszczenie, choć nie tak intensywną jak przy wysokim obciążeniu.

Wykorzystanie pamięci dolny wykres — fioletowa linia

- Szybszy wzrost do około 48 MiB, z regularnymi spadkami wynikającymi z agresywniejszych cykli GC.
- Wzorzec „wspinaczka i zrzut” wskazuje na cykliczne, ale kontrolowane zużycie pamięci.

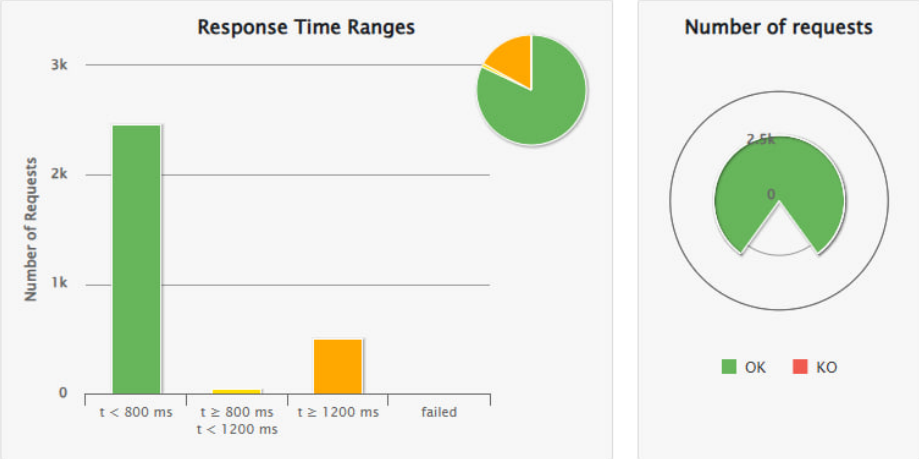
Raporty wygenerowane przez narzędzie Gatling:



Rysunek 18. Raport wygenerowany w narzędziu Gatling dla aplikacji działającej w oparciu o podejście REST, testowanej przy niskim obciążeniu.

Źródło: opracowanie własne.

– Analiza przy niskim obciążeniu:
  - Dominujący czas odpowiedzi poniżej 800 ms (około 595 żądań)
  - 0 żądań w przedziale 800-1200 ms
  - Minimalna liczba żądań powyżej 1200 ms
  - Brak żądań nieudanych (KO = 0)



Rysunek 19. Raport wygenerowany w narzędziu Gatling dla aplikacji działającej w oparciu o podejście REST, testowanej przy średnim obciążeniu.

Źródło: opracowanie własne.

- Analiza przy średnim obciążeniu:
  - Około 2400 żądań poniżej 800 ms - zauważalny spadek wydajności
  - Mała żądań w przedziale 800-1200 ms
  - Pojawienie się około 500 żądań powyżej 1200 ms
  - Minimalna liczba nieudanych żądań, system zachowuje stabilność

Rysunek 20. Raport wygenerowany w narzędziu Gatling dla aplikacji działającej w oparciu o podejście REST, testowanej przy wysokim obciążeniu.

Źródło: opracowanie własne.

– Analiza przy wysokim obciążeniu:
  - Brak odpowiedzi poniżej 800 ms
  - 0 żądań w przedziale 800-1200 ms
  - Około 89 procentów żądań (7500-10000) powyżej 1200 ms
  - Znacząca liczba nieudanych żądań wskazująca na przeciążenie systemu

Podsumowując, podejście REST wykazuje dobrą wydajność przy niskim obciążeniu, ale znacząco traci na efektywności wraz ze wzrostem liczby żądań. Przy wysokim obciążeniu system ujawnia poważne problemy z wydajnością, co manifestuje się poprzez długie czasy odpowiedzi i występowanie błędów.

Ocena współdzielenia kodu za pomocą Serverles pod kątem skalowalności:

W celu przygotowania analizy pod kątem skalowalności, tak samo jak w poprzednim przypadku opracowałem system, którego celem jest testowanie wydajności przykładowej aplikacji wykorzystującej biblioteki do współdzielenia kodu oraz obserwacja, w jaki sposób to podejście wpływa na wydajność i skalowalność systemu.

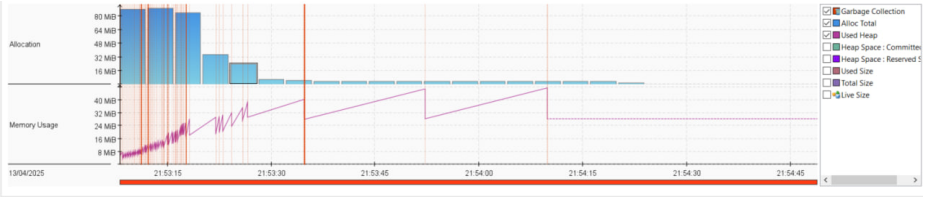
Przygotowane aplikacje przykładowe oraz kod systemu, który powstał w celu sprawdzenia i porównania wydajności, zostały dołączone do materiałów pracy.

Testowany system mikroserwisów składa się z dwóch mikroserwisów napisanych z użyciem technologii Java oraz Spring. Pierwszy mikroserwis produkuje dane, drugi je konsumuje. Oba serwisy uruchamiane są za pomocą narzędzia Docker. Kontenery Dockera mają ustawione limity zasobów, które mogą wykorzystywać.

Po uruchomieniu aplikacji w Dockerze, automatycznie uruchamiane są testy wydajnościowe przygotowane za pomocą narzędzia Gatling. Projekt został stworzony przy użyciu Javy, Springa oraz Gatlinga i znajduje się w materiałach dołączonych do pracy.

W trakcie testów wydajnościowych zbierane są metryki dotyczące użycia pamięci, liczby wątków, zużycia CPU, garbage collection i inne. Dane te są zapisywane do pliku za pomocą narzędzia Java Flight Recorder. Uruchomienie JFR odbywa się również automatycznie w momencie startu aplikacji.

Aplikacja testująca wydajność aplikacji przykładowej posiada trzy tryby testowania: małe obciążenie, średnie oraz duże. Zostało to zrealizowane w celu obserwacji działania aplikacji przy różnych poziomach obciążenia.



Rysunek 21. Wykres przedstawiający zużycie pamięci RAM aplikacji wykorzystującej podejście Serwerless pod wpływem małego obciażenia

Źródło: opracowanie własne.

Wyniki obserwacji:
Alokacja pamięci (niebieskie słupki)

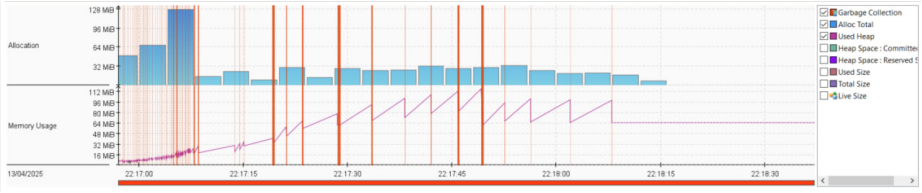
- Szczyty alokacji są bardzo niskie, utrzymujące się w okolicach 9 MB.
- Brak wyraźnych okresowych wzorców, co sugeruje sporadyczne i minimalne tworzenie obiektów.
- Wykres wskazuje na bardzo małe obciążenie systemu, z rzadkimi zdarzeniami alokacji.

Zbieranie śmieci (pionowe pomarańczowe linie)

- Bardzo rzadkie zdarzenia GC, co jest spodziewane przy minimalnym wykorzystaniu pamięci.
- Brak wyraźnych sygnałów GC na wykresie, co potwierdza niską aktywność systemu.

Wykorzystanie pamięci (linie fioletowe)

- Wykorzystanie pamięci jest stabilne i bardzo niskie, oscylujące wokół 9 MB.
- Brak znaczących wahań, co wskazuje na brak presji na pamięć.



Rysunek 22. Wykres przedstawiający zużycie pamięci RAM aplikacji wykorzystującej podejście Serverless pod wpływem średnigo obciażenia

Źródło: opracowanie własne.

W przypadku duzego obciążenia nie było mozliwe wygenerowanie raportu ze wzgledu na Memory Error.

Wyniki obserwacji:

Alokacja pamięci (niebieskie słupki)

- Aktywność alokacji jest znacznie wyższa, z wartościami sięgającymi 12 MB.
- Występuje bardziej regularny wzorzec alokacji, wskazujący na ciągłe przetwarzanie żądań.
- Zwiększona częstotliwość alokacji sugeruje większe obciążenie systemu w porównaniu do scenariusza niskiego obciążenia.

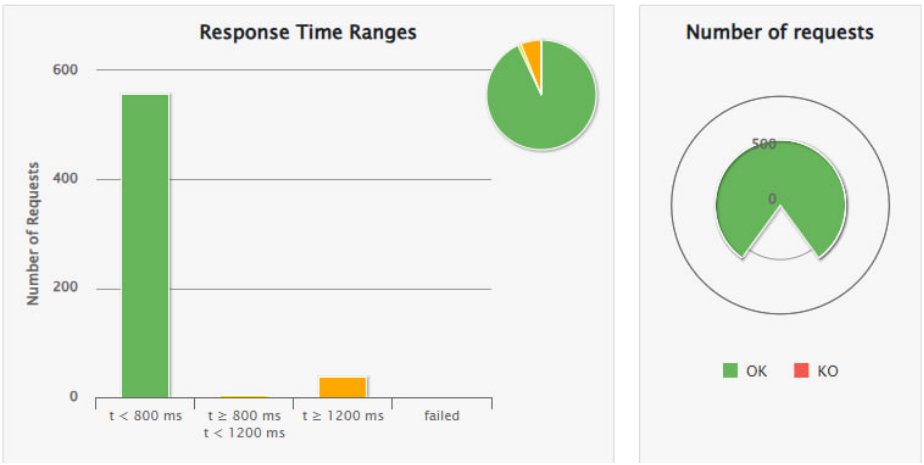
Zbieranie śmieci (pionowe pomarańczowe linie)

- Częstsze zdarzenia GC, choć nie są one bardzo intensywne.
- Występowanie GC jest związane ze zwiększoną alokacją obiektów, ale nie osiąga poziomu agresywnego czyszczenia.

Wykorzystanie pamięci (linie fioletowe)

- Wykorzystanie pamięci jest wyższe, osiągając wartości do 128 MB, ale z częstymi spadkami.
- Wykres pokazuje okresowe wzrosty i spadki, co sugeruje cykle alokacji i zwalniania pamięci.
- System radzi sobie z obciążeniem, ale wymaga częstszego odśmiecania.

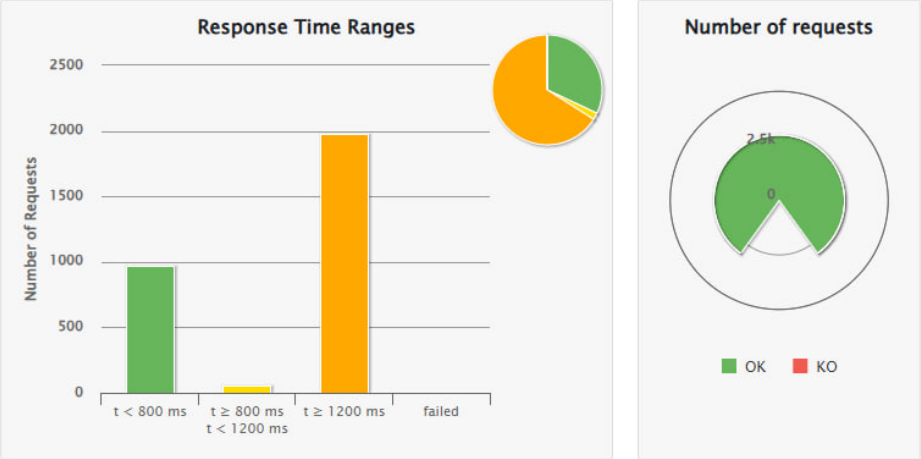
Raporty wygenerowane przez narzędzie Gatling:



Rysunek 23. Raport wygenerowany w narzędziu Gatling dla aplikacji działającej w oparciu o podejście Serverless, testowanej przy niskim obciążeniu.

Źródło: opracowanie własne.

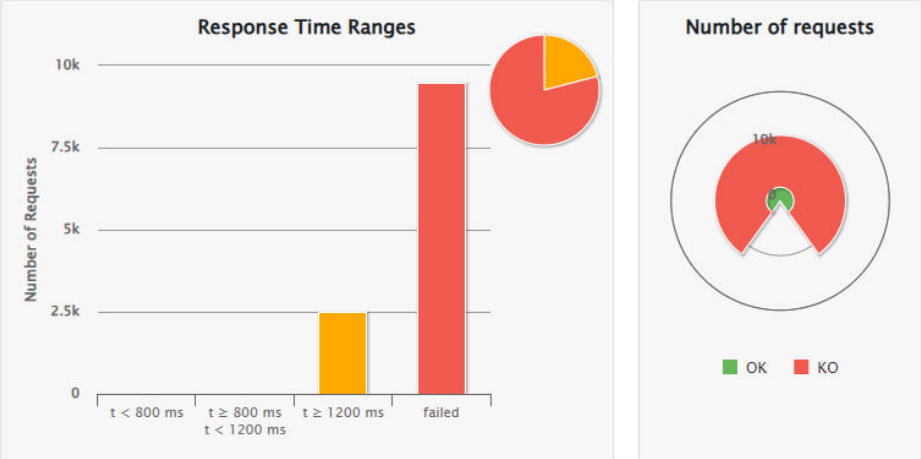
– Analiza przy niskim obciążeniu:
  - Dominująca liczba żądań (580) z czasem odpowiedzi poniżej 800 ms
  - Minimalna liczba żądań w przedziale 800-1200 ms
  - Mała liczba żądań powyżej 1200 ms
  - Brak żądań nieudanych - pełna niezawodność



Rysunek 24. Raport wygenerowany w narzędziu Gatling dla aplikacji działającej w oparciu o podejście Serverless, testowanej przy średnim obciążeniu.

Źródło: opracowanie własne.

– Analiza przy średnim obciążeniu:
  - Około 1000 żądań poniżej 800 ms - poprawa względem poprzedniego testu
  - Minimalna liczba żądań w przedziale 800-1200 ms
  - 2000 ządań powyżej 1200 ms
  - Utrzymana pełna niezawodność - brak żądań nieudanych



Rysunek 25. Raport wygenerowany w narzędziu Gatling dla aplikacji działającej w oparciu o podejście Serverless, testowanej przy wysokim obciążeniu.

Źródło: opracowanie własne.

- Analiza przy wysokim obciążeniu:
  - 0 żądań poniżej 800 ms
  - 0 żądań w przedziale 800-1200 ms
  - 39 procentów żądań powyżej 1200 ms
  - Bardzo duża ilość żądań nieudanych
- Podsumowanie wyników:
  - Niskie obciążenie: Optymalna wydajność z szybkim czasem reakcji
  - Średnie obciążenie: Stabilna wydajność z niewielkimi opóźnieniami
  - Wysokie obciążenie: Znaczące pogorszenie wydajności i stabilności

Obserwowane pogorszenie wydajności, szczególnie przy większych obciążeniach, prawdopodobnie wynika z opóźnienia sieciowego wprowadzonego przez architekturę serverless. W porównaniu z innymi podejściami do współdzielenia kodu.

Podsumowanie wyników testowania rozwiązań pod kątem wydajności i skalowalności.

Ranking rozwiązań pod względem wydajności i skalowalności który stowżywem na podstawie przeprowadzonych testów i analiz:

1. Serverless
   - Wydajność:
     - Bardzo wydajne wykorzystanie pamięci (9MB przy niskim obciążeniu)
     - Spójna wydajność z większością żądań poniżej 800ms
   - Skalowalność:
     - Automatyczne skalowanie w zależności od obciążenia
     - Efektywne zarządzanie zasobami przy zmiennym obciążeniu
     - Najlepsza elastyczność przy nagłych skokach ruchu
2. IDL (Interface Definition Language)
   - Wydajność:
     - Najlepsza gospodarka pamięcią (szczyty około 32 MiB)
     - Najbardziej spójne czasy odpowiedzi poniżej 800ms
     - Doskonałe wzorce garbage collection
   - Skalowalność:
     - Dobra skalowalność horyzontalna
     - Wymaga ręcznej konfiguracji skalowania
     - Stabilna wydajność przy zwiększającym się obciążeniu
3. Biblioteki
   - Wydajność:
     - Umiarkowane wykorzystanie pamięci (150-200MB)
     - Dobra wydajność przy niskim obciążeniu
   - Skalowalność:
     - Ograniczona skalowalność automatyczna
     - Wymaga dodatkowej infrastruktury do skalowania
     - Stabilna przy przewidywalnym obciążeniu
4. SDK
   - Wydajność:
     - Wyższe zużycie pamięci
     - Częstsze zdarzenia garbage collection
   - Skalowalność:
     - Trudniejsza w implementacji skalowalność

    - Wyższe koszty skalowania
    - Znacząca degradacja przy wysokim obciążeniu
5. REST
   - Wydajność:
     - Najwyższe zużycie pamięci
     - Najwyższy procent wolnych odpowiedzi (>1200ms)
   - Skalowalność:
     - Najbardziej problematyczna skalowalność
     - Wysokie ryzyko błędów przy skalowaniu
     - Znaczące koszty infrastruktury przy skalowaniu

Wnioski końcowe

Na podstawie przeprowadzonej analizy można stwierdzić, że wybór optymalnego rozwiązania zależy od specyficznych wymagań projektu, jednak biorąc pod uwagę zarówno wydajność jak i skalowalność, architektura Serverless oraz IDL wykazują najlepsze właściwości.

Serverless zajmuje pierwsze miejsce głównie ze względu na:

- Automatyczną skalowalność bez potrzeby ręcznej konfiguracji
- Najniższe zużycie zasobów przy zmiennym obciążeniu
- Optymalne zarządzanie kosztami w zależności od rzeczywistego wykorzystania

IDL plasuje się na drugim miejscu, oferując:

- Najlepszą wydajność przy stałym obciążeniu
- Przewidywalną skalowalność
- Niższe koszty operacyjne przy stałym, wysokim obciążeniu

Biblioteki stanowią rozsądny kompromis dla projektów o średniej skali, podczas gdy SDK i REST wykazują największe ograniczenia w kontekście skalowalności i wydajności przy większym obciążeniu.

Rekomendacje

- Dla projektów z dynamicznym obciążeniem: Serverless
- Dla projektów ze stałym, wysokim obciążeniem: IDL
- Dla projektów średniej wielkości z przewidywalnym obciążeniem: Biblioteki
- Unikać REST i SDK w przypadku wymagań dotyczących wysokiej skalowalności

Należy jednak pamiętać, że błędy pamięci przy wysokim obciążeniu były obecne we wszystkich podejściach, co podkreśla znaczenie właściwego planowania pojemności i implementacji strategii skalowania, niezależnie od wybranego rozwiązania.

5.5 Opis części praktycznej

Jaką część praktyczną mojej pracy magisterskiej przygotowałem aplikację webową, która: Składa się z dwóch części, backendowej i frontendowej. Głównym celem napisania aplikacji jest wprowadzenie nowego podejścia do współdzielenie kodu – współdzielenie za pomocą technologii serverless. Aplikacja wspiera deweloperów korzystających z technologii serverless, zmniejsza wady tego rozwiązania, jednocześnie dając możliwość skorzystania z mocnych stron tej technologii.

Czym jest technologia Serverless?

Serverless to nowoczesny model wykonywania obliczeń w chmurze, w którym dostawcy chmury automatycznie obsługują aspekty infrastruktury, takie jak zarządzanie serwerem oraz skalowanie zasobów. Zamiast zarządzać serwerami fizycznymi lub wirtualnymi, programiści mogą skupić się wyłącznie na pisaniu kodu. Termin „serverless" nie oznacza, że nie ma fizycznych serwerów, ale że zarządzanie serwerem jest oderwane od użytkowników. W architekturach serverless wdrażamy do chmury funkcje zamiast całych aplikacji, a dostawca chmury jest odpowiedzialny za zarządzanie kodem, skalowanie i utrzymywanie tych funkcji. Są one też często nazywane Function-as-a-Service (FaaS).

Przykłady platform server less dostępnych w 2025 roku:

1. AWS Lambda
2. Azure Functions
3. Google Cloud Functions
4. IBM Cloud Functions

5.5.1 Ogulny opis aplikacji Za pomocą przygotowanej przeze mnie aplikacji webowej wykonuję następujące czynności:

1. Akceptuję i sprawdzam kod, który chcesz podzielić za pomocą technologii serverless. Jeśli coś nie gra w czasie sprawdzania, powiem ci, co masz zrobić, żeby skończyć to sukcesem.

2. Automatycznie przygotowuję kod do wrzucenia na Amazon Lambda (w przyszłości może dać ci opcję wyboru innej platformy, np. Azure Functions).
3. Automatycznie wrzucam kod na platformę Serverless, po chwili będzie od razu dostępny.
4. Zapisuję kod w bazie danych, żebyś mógł go użyć później; będziesz miał widok na listę wszystkich wrzutek kodu.
5. Możesz edytować i ponownie wrzucić kod z tej listy.
6. Możesz wyszukiwać kod po opisie.
7. Dodaję wersjonowanie kodu.
8. Automatycznie przygotowuję pliki IDL, które są przydatne do wykorzystania, gdy kod jest wywoływany na IDL.
9. Przechowuję te pliki IDL i możesz je udostępnić, żeby inni członkowie zespołu mogli współdzielić ten kod.

5.5.2 **Opis aplikacji back-end** Używane technologie

1. Java - Język do napisania backendu.
2. Spring Boot - Narzędzie, które ułatwia tworzenie aplikacji w Javie.
3. Maven - Automation template, które automatyzuje tworzenie procedur śledzenia zależności projektu, kompilacji projektów w inteligentny sposób.
4. AWS Lambda - To jest usługa obliczeniowa, która nie wymaga używania serwerów, ale jednocześnie dostawcy mogą tu pokazywać i uruchomić kod bez konieczności wdrażania lub zarządzania serwerami.
5. Protobuf - Jest to język IDL do szybkiego kodowania i dekodowania obiektów DTO.
6. Jackson - Jeden z najsłynniejszych narzędzi do parsowania JSONów, jak również konwertowania wejść/wyjść na obiekty DTO itp. w drugą stronę.

Ogólny opis

Ta aplikacja jest częścią aplikacji webowej, która odpowiada za następujące działania:

1. Odbiera kod jako dane wejściowe użytkownika - Akceptuje kod od użytkownika za pośrednictwem żądań HTTP.

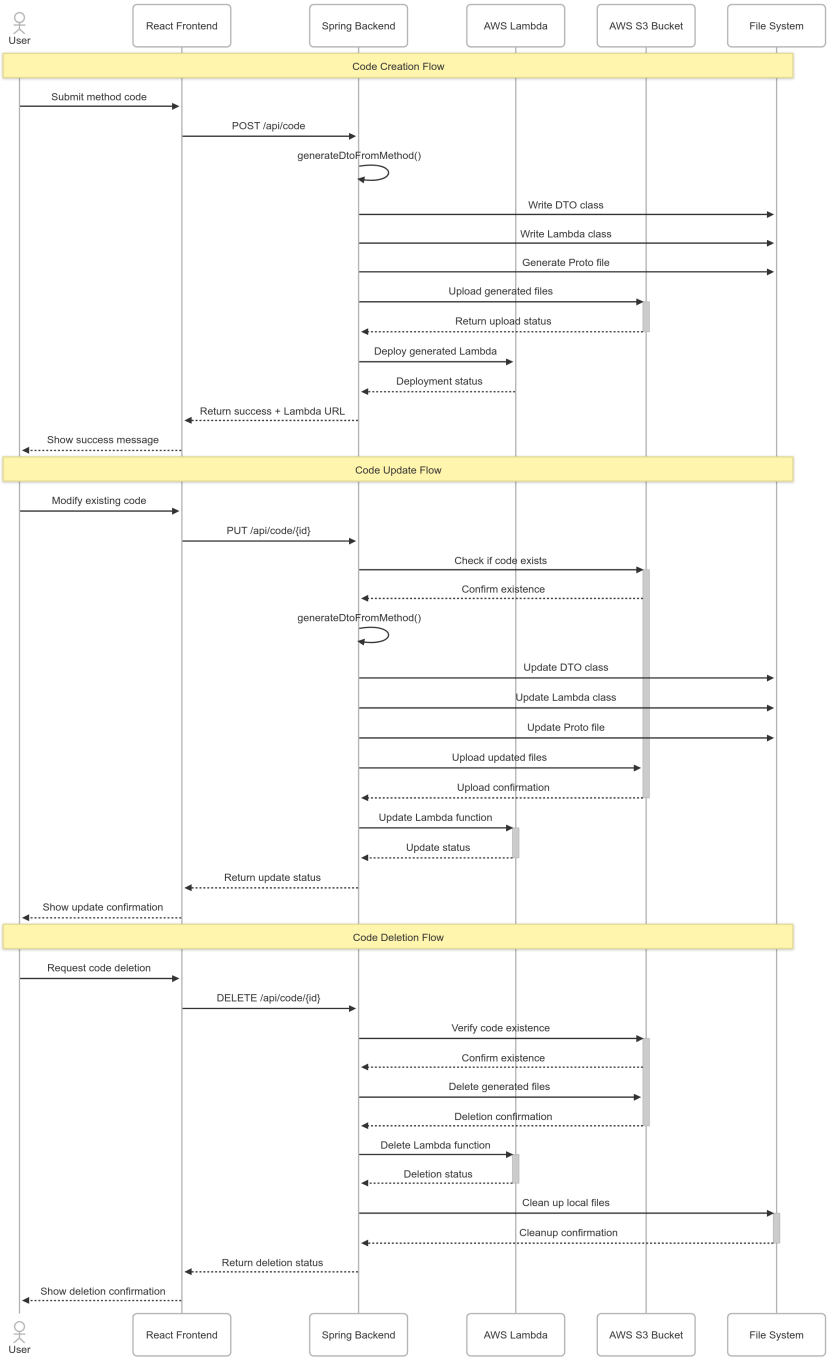
2. Weryfikuje kod - Sprawdza, że otrzymany kod spełnia wymagane standardy i format.
3. Automatycznie wdraża kod jako funkcję Lambda - Wdraża sprawdzony kod jako funkcję AWS Lambda.
4. Generuje pliki .proto - Tworzy pliki .proto na podstawie kodu wejściowego, które mogą być używane do generowania obiektów DTO do komunikacji z wdrożoną funkcją Lambda.

5.5.3 Opis aplikacji front-end Aplikację front-endową przygotowałem za pomocą JavaScript, React i npm. React to biblioteka używana do tworzenia aplikacji frontendowych. Npm służy do zarządzania zależnościami projektu.

Ogólny opis:

Ta aplikacja jest częścią aplikacji webowej, część front-endowa jest odpowiedzialna za następujące działania:

1. Przyjmowanie kodu, który użytkownik chce współdzielić, a następnie wysyłanie go do serwera back-endowego.
2. Weryfikacja tego, czy kod spełnia wymagane standardy i formatowanie.
3. Wyświetlanie wyników walidacji.
4. Możliwość przeglądania, edytowania i usuwania kodu, który został wcześniej zapisany i wdrożony w AWS Lambda.
5. Możliwość wyszukiwania wdrożonego kodu, używając jego opisu i nazwy.
6. Wsparcie wersjonowania kodu w celu śledzenia zmian lub przywracania poprzednich wersji.
7. Udostępnianie plików .proto: aplikacja front-end udostępnia użytkownikom wygenerowane pliki .proto, których użytkownicy mogą użyć do generowania obiektów DTO dla komunikacji z wdrożoną funkcją Lambda.



Rysunek 26. Diagram sekwencyjny aplikacji praktycznej

Źródło: opracowanie własne.

5.6 Prezdsawienie ważnych fragmentów kodu

Chcę przedstawić kluczowe fragmenty kodu mojego projektu praktycznego wraz z krótkim opisem ich działania.

Poniżej przedstawiono podstawową konfigurację usługi Lambda przy użyciu klienta AWS.

Klasa inicjalizuje klienta usług AWS — w moim przypadku są to: S3Client, LambdaClient oraz IamClient, które służą odpowiednio do zarządzania funkcjami Lambda, przechowywaniem danych w S3 oraz rolami IAM.

```
@Service
public class LambdaService {
    private final String bucketName = "lambda-jars";
    private final String roleName = "lambda-execution-role";
    private final String jarFilePath = "path/to/jar";

    private final S3Client s3Client;
    private final LambdaClient lambdaClient;
    private final IamClient iamClient;

    public LambdaService(S3Client s3Client, LambdaClient lambdaClient,
                         IamClient iamClient) {
        this.s3Client = s3Client;
        this.lambdaClient = lambdaClient;
        this.iamClient = iamClient;
    }
}
```

Poniższy kod wykonuje budowanie pliku JAR za pomocą Mavena, a następnie przesyła go do S3 oraz tworzy nową funkcję Lambda z odpowiednią rolą IAM.

```
public void prepareJarForLambda(String id) {
    System.setProperty("maven.multiModuleProjectDirectory", "path/to/project")
    System.setProperty("maven.home", "path/to/maven");
    System.setProperty("JAVA_HOME", "path/to/java");

    InvocationRequest request = new DefaultInvocationRequest();
```

```
    request.setPomFile(new File("path/to/pom.xml"));
    request.setGoals(Collections.singletonList("clean package"));

    Invoker invoker = new DefaultInvoker();
    try {
        invoker.execute(request);
    } catch (Exception e) {
        throw new RuntimeException("Maven build failed", e);
    }
    uploadJarToS3(id);
    createLambdaFunction(ensureIamRole(), id);
}
```

W poniższym fragmencie kodu przedstawiam, w jaki sposób tworzę funkcję Lambda w AWS za pomocą języka Java oraz SDK dostarczonego przez AWS, na podstawie wcześniej przygotowanego pliku JAR. Proces przygotowania tego pliku został pokazany w poprzednim fragmencie kodu.

```
private void createLambdaFunction(String roleArn, String id) {
    FunctionCode functionCode = FunctionCode.builder()
            .s3Bucket(bucketName)
            .s3Key(id + ".jar")
            .build();

    CreateFunctionRequest functionRequest = CreateFunctionRequest.builder()
            .functionName(id)
            .runtime(Runtime.JAVA21)
            .role(roleArn)
            .handler("org.example.MyLambdaFunction::handleRequest")
            .code(functionCode)
            .build();

    lambdaClient.createFunction(functionRequest);
}
```

W poniższym fragmencie kodu przedstawiam sposób, w jaki używam usługi AWS S3 — serwisu, który służy do przechowywania dołączonych plików. Robię to ze względu na fakt, że funkcje Lambda

w sposób programistyczny możemy tworzyć wyłącznie na podstawie pliku JAR umieszczonego w S3. W przypadku konsoli webowej można to zrobić bezpośrednio, ale z ograniczeniem dotyczącym rozmiaru pliku JAR. W kodzie poniżej pokazuję, w jaki sposób aplikacja dodaje pliki do S3, pobiera je oraz usuwa.

```
public class S3Operations {
    private final S3Client s3Client;
    private final String bucketName;

    public void uploadJarToS3(String id, Path jarPath) {
        s3Client.putObject(PutObjectRequest.builder()
                .bucket(bucketName)
                .key(id + ".jar")
                .build(),
            RequestBody.fromFile(jarPath));
    }

    public void deleteJarFromS3(String id) {
        s3Client.deleteObject(DeleteObjectRequest.builder()
                .bucket(bucketName)
                .key(id + ".jar")
                .build());
    }

    public byte[] downloadJar(String id) {
        ResponseBytes<GetObjectResponse> objectBytes = s3Client.getObjectAsByt
            GetObjectRequest.builder()
                .bucket(bucketName)
                .key(id + ".jar")
                .build());
        return objectBytes.asByteArray();
    }
}
```

W następnym fragmencie kodu pokazuję, jak wygląda obsługa błędów w mojej aplikacji. Używam powszechnie stosowanego i popularnego we współczesnych aplikacjach podejścia @ControllerAdvice

```
@ControllerAdvice
public class LambdaExceptionHandler {
    @ExceptionHandler(LambdaException.class)
    public ResponseEntity<ErrorResponse> handleLambdaException(
            LambdaException ex) {
        ErrorResponse error = new ErrorResponse(
            HttpStatus.BAD_REQUEST.value(),
            ex.getMessage(),
            System.currentTimeMillis()
        );
        return new ResponseEntity<>(error, HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(S3Exception.class)
    public ResponseEntity<ErrorResponse> handleS3Exception(
            S3Exception ex) {
        ErrorResponse error = new ErrorResponse(
            HttpStatus.INTERNAL_SERVER_ERROR.value(),
            "S3 Operation failed: " + ex.getMessage(),
            System.currentTimeMillis()
        );
        return new ResponseEntity<>(error, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Poniżej pokazuję fragment kodu, który przedstawia kontroler REST aplikacji backendowej. Kontroler udostępnia aplikacji front-endowej podstawowe operacje dla funkcji Lambda, takie jak: tworzenie nowej funkcji, sprawdzanie statusu, modyfikacja oraz usunięcie.

```
@RestController
@RequestMapping("/api/lambda")
public class LambdaController {
    private final LambdaService lambdaService;

    @PostMapping("/deploy")
    public ResponseEntity<DeploymentResponse> deployFunction(
            @RequestBody DeploymentRequest request) {
        String functionId = lambdaService.prepareJarForLambda(request.getCode(
```

```
        return ResponseEntity.ok(new DeploymentResponse(functionId));
    }

    @DeleteMapping("/{functionId}")
    public ResponseEntity<Void> deleteFunction(
            @PathVariable String functionId) {
        lambdaService.deleteLambdaFunction(functionId);
        return ResponseEntity.noContent().build();
    }

    @GetMapping("/{functionId}/status")
    public ResponseEntity<FunctionStatus> getFunctionStatus(
            @PathVariable String functionId) {
        return ResponseEntity.ok(
            lambdaService.getFunctionStatus(functionId));
    }
}
```

5.7 Podsumowanie

Spis tablic

Spis rysunków

Literatura

1. O. A. Ahmed et al. Impact of third-party sdks on mobile application performance. In *Proceedings of the IEEE International Conference on Mobile Software Engineering and Systems*, 2019.
2. Joshua Bloch. *Effective Java*. Addison-Wesley Professional, 2018.
3. George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Pearson Education, 5th edition, 2011.
4. Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2012.
5. Ian Gorton. *Essential Software Architecture*. Springer, 2011.

6. IBM. What is serverless computing?, 2023.
7. Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, Inc., 2017.
8. Mohamed Labouardy. *Pipeline as Code: Continuous Delivery with Jenkins, Kubernetes, and Terraform*. Manning Publications, 2021.
9. Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.
10. Microsoft Azure. Recommendations for optimizing code and infrastructure, 2020. Retrieved from Microsoft Azure Optimization Guide.
11. Ronnie Mitra, Irakli Nadareishvili, Matt McLarty, and Mike Amundsen. *Microservices: Up and Running: A Step-by-Step Guide to Building a Microservice Architecture*. O'Reilly Media, Inc., 2020.
12. Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., 2015.
13. Michael T. Nygard. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2008.
14. Chris Richardson. *Microservices Patterns: With examples in Java*. Manning Publications, 2018.
15. Leonard Richardson, Mike Amundsen, and Sam Ruby. *RESTful Web APIs*. O'Reilly Media, 2013.
16. John Gilbert Roberts. *Cloud Native Development Patterns and Best Practices*. Packt Publishing, 2018.
17. Kenneth S. Rubin. *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Addison-Wesley, 2012. See p. 150.
18. Raj Sharma. *Monolithic to Microservices: Evolution of Software Architecture*. Tech Publications, 2023.
19. William Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson, 2017.
20. Wikipedia. Interface description language, 2023.

6 Appendices