



Warsaw, March 9, 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** Szybki rozwój i powszechne przyjęcie architektury mikroservisowej w ostatnich latach zmienił świat informatyki. Architektura mikroservisowa ma wiele zalet, takich jak elastyczność, łatwe wdrożenia i prostota w utrzymaniu. Natomiast powoduje nowe wyzwania, takie jak, na przykład, problem duplikacji kodu i zarządzanie kodem. W mojej pracy magisterskiej przeprowadziłem dokładną analizę metod i podejść współdzielenia kodu w systemach mikroservisów. W mojej pracy chcę porównać za pomocą źródeł literaturowych oraz przeprowadzonych badań dostępne metody współdzielenia kodu, zdefiniować przypadki w których warto wybrać takie lub inne podejście. Oraz w części praktycznej chcę zaproponować własne nowoczesne podejście do współdzielenia kodu które połączy zalety istniejących rozwiązań i zapewni wydajność oraz komfort w zarządzaniu kodem. Na końcu pracy znajduje się porównanie mojego rozwiązania z istniejącymi za pomocą przygotowanych testów wydajnościowych.

**Keywords:** Architektura mikroservisów · Współdzielenie kodu  
· Wydajność.

# Spis treści

1	Wstęp.....	6
1.1	Zakres pracy .....	6
1.2	Motywacja .....	6
1.3	Zawartość pracy .....	7
2	Architektura mikroserwisowa a monolitowa.....	8
2.1	Architektura mikroserwisowa .....	8
2.2	Architektura monolitowa .....	9
2.3	Porównanie .....	9
3	Analiza dziedziny problemowej .....	12
3.1	Typy obiektów w aplikacjach mikroserwisowych .....	12
3.2	Kryteria porównania i analizy metod współdzielenia kodu .....	14
4	Analiza metod współdzielenia kodu .....	15
4.1	Metody współdzielenia kodu z opisem .....	15
4.2	Analiza metod współdzielenia kodu dla różnych typów obiektów .....	18
4.3	Definicja kryterium oceny i porównania metod współdzielenia kodu .....	20
4.3.1	Definicja kryterium oceny pod kątem izolacji w przypadku IDL .....	20
4.3.2	Definicja kryterium oceny pod kątem izolacji w przypadku bibliotek, sdk, REST, serverless ...	22
4.3.3	Definicja kryterium oceny pod kątem bezpieczeństwa .....	24
4.3.4	Definicja kryterium oceny pod kątem skalowalności .....	25
4.3.5	Definicja kryterium oceny pod kątem wersjonowania .....	27
4.4	Ocena i porównanie metod współdzielenia kodu .....	29
4.4.1	Ocena współdzielenia kodu za pomocą Interface definition languages .....	29
4.4.2	Ocena współdzielenia kodu za pomocą bibliotek .....	37
4.4.3	Ocena współdzielenia kodu za pomocą SDK .....	41
4.4.4	Ocena współdzielenia kodu za pomocą API .....	44
4.4.5	Ocena współdzielenia kodu za pomocą ServerLess .....	44

4.5	Opis części praktycznej .....	44
4.6	Podsumowanie .....	44
5	Appendices .....	45
5.1	Sample Code .....	45
5.2	Sample Data .....	45

# 1 Wstęp

## 1.1 Zakres pracy

W pracy magisterskiej przeprowadzono dokładną analizę metod i podejść do współdzielenia kodu w systemach mikroservisów, zakres tej pracy zawiera porównanie między architekturą monolitową a mikroservisową, identyfikację typów obiektów wykorzystywanych w kodzie współczesnych aplikacji, analizę dostępnych metod i podejść do współdzielenia kodu IDL, SDK oraz biblioteki, praktyczna implementacja aplikacji wspierającej nowatorskie podejście do współdzielenia kodu ServerLess.

## 1.2 Motywacja

Główną motywacją do napisania pracy było rosnące w współczesnych czasach zainteresowanie mikroservisowym podejściem do architektury, które ma wiele zalet, takich jak elastyczność, łatwość wdrożenia i utrzymania na produkcji, możliwość łatwej i szybkiej naprawy problemów na produkcji ale natomiast ma również powoduje nowe wyzwania, takie jak problem duplikacji kodu i zapotrzebowanie na współdzielenie kodu między mikroservisami 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 w współdzieleniu kodu w systemach mikroservisów. Porównać bardziej tradycyjne podejścia do współdzielenia kodu, takie jak IDL, SDK, biblioteki z nowatorskim podejściem do współdzielenia kodu takich 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, które wprowadzają czytelnika najpierw w szczegóły teoretyczne a później w praktyczne aspekty współdzielenia kodu w systemach mikroservisów. Krótki opis rozdziałów znajduje się poniżej:

1. Wstęp - W tym rozdziale analizuję wady i zalety architektury mikroservisowej, monolitycznej, porównuję dlaczego z architektury monolitycznej powstała architektura mikroservisowa, analizuję przypadki w których lepiej użyć architektury monolitycznej a w której architekturę mikroservisową. Opisuję dlaczego powstało zapotrzebowanie na współdzielenie kodu. Wyjaśniam logikę leżącą u podstaw eksploracji wielu metod współdzielenia kodu. Uzasadniam informację przykładami z literatury.
2. Architektura mikroservisowa a Monolitowa - W tym rozdziale porównuję architekturę mikroservisową z monolityczną, przedstawiam różnice w utrzymaniu i skalowalności, za pomocą źródeł literaturowych prezentuję.
3. Analiza dziedziny problemowej - W tym rozdziale pracy kategoryzuję typy obiektów które mogą potrzebować współdzielenia kodu w systemie mikroservisów, 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 mikroservisów 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 mikroservisów - Server Less rozszerzając możliwości tego rozwiązania.

## 2 Architektura mikroservisowa a monolitowa

Rozwój architektury mikroservisowej w ostatnich latach przyniósł wiele korzyści, takich jak łatwość skalowania, niezależność wdrożenia i elastyczność. Jednak, wraz z tą elastycznością, pojawiają się również nowe wyzwania, związane między innymi z odpowiednim współdzieleniem obiektów pomiędzy mikroservisami.

### 2.1 Architektura mikroservisowa

Podział aplikacji na mniejsze, niezależne serwisy umożliwia elastyczne skalowanie poszczególnych komponentów, co przekłada się na lepszą wydajność i dostępność systemu. Ponadto, rozbudowa i utrzymanie aplikacji w oparciu o mikroservisy jest znacznie prostsze, ponieważ każdy serwis może być rozwijany niezależnie. Taka modułowa struktura pozwala na szybsze wprowadzanie nowych funkcjonalności oraz łatwiejsze naprawianie błędów. Istotną korzyścią jest również możliwość korzystania z różnych technologii w poszczególnych serwisach, co daje większą elastyczność i umożliwia wykorzystanie najlepszych narzędzi dla każdej części aplikacji. W rezultacie, architektura mikroservisowa umożliwia bardziej efektywne zarządzanie projektem, lepszą skalowalność zespołu oraz izolację błędów, co przyczynia się do wyższej jakości i niezawodności systemu.

"Microservices are small, autonomous services that work together. Let's break that definition down a bit and consider the characteristics that make microservices different." [11, p. 2]

Charakterystyka architektury mikrosług: Modułowość - Usługi są modułowe, co umożliwia łatwiejszy rozwój, konserwację i skalowalność, ponieważ każda usługa kontroluje określoną funkcję lub cechę. Solidność - Mikrosługi zwiększają ogólną solidność systemu, ponieważ błędy w jednej usłudze nie wpływają na całą aplikację, zapewniając tolerancję błędów i niezawodność systemu. Interoperacyjność - Różne mikrosługi komunikują się za pośrednictwem dobrze zdefiniowanych interfejsów API, zapewniając bezproblemową integrację i interakcję usług. Równoległy rozwój - Oddzielne zespoły programistyczne mogą pracować nad różnymi mikrosługami jednocześnie,



co przyspiesza rozwój i funkcje. Elastyczność w zakresie elastycznych technologii - Różne mikrousługi można budować przy użyciu różnych technologii, co pozwala na wykorzystanie najskuteczniejszych narzędzi dla każdej przydzielonej funkcji/cechy. [15]

## 2.2 Architektura monolitowa

Architektura monolityczna, która opiera się na jednym spójnym kodzie źródłowym, oferuje pewne korzyści w kontekście prostoty zarządzania i łatwości wdrożenia. Wszystkie komponenty aplikacji są ze sobą ściśle powiązane, co ułatwia debugowanie i śledzenie błędów. Ponadto, brak konieczności konfigurowania i zarządzania infrastrukturą dla wielu usług upraszcza proces wdrażania. W przypadku mniejszych projektów o prostszych wymaganiach, architektura monolityczna może być bardziej efektywna i wydajna, eliminując niepotrzebną złożoność komunikacji między usługami. Jednak warto pamiętać, że architektura monolityczna może napotkać trudności w skalowaniu i utrzymaniu w przypadku większych, bardziej złożonych systemów.

Charakterystyka architektury monolitycznej: Pojedyncza jednostka wdrożenia - Cała aplikacja jest wdrażana jako pojedyncza, niepodzielna jednostka. Wszelkie uaktualnienia, ulepszenia lub modyfikacje wymagają wdrożenia całej aplikacji, w tym wszystkich jej komponentów. Scentralizowany przepływ kontroli - Centralny moduł lub funkcja podstawowa nadzoruje przepływ kontroli w aplikacji, koordynując sekwencyjny postęp wykonywania od jednego komponentu do następnego. Ścisłe sprzężenie - Komponenty i moduły w aplikacji są silnie powiązane i zależne od siebie. Współdzielona pamięć - Wszystkie komponenty oprogramowania mają bezpośredni dostęp do zasobów pamięci, co sprzyja ścisłej integracji. Jednak taka konfiguracja może również powodować konflikty zasobów i trudności ze skalowaniem aplikacji. [15]

## 2.3 Porównanie

W współczesnym świecie programowania, zawsze wybieramy między podejściem monolitycznym i mikroserwisowym, to wiąże się z kompromisem między prostotą, główną cechą podejścia monolitycznego

a elastycznością w przypadku podejścia mikroserwisowego. Podejście monolitowe, które polega na przechowywaniu wszystkich komponentów systemu w jednym miejscu, oferuje łatwość w rozwoju i zarządzaniu kodem ale natomiast może stać wąskim gardłem kiedy aplikacja urośnie. W przeciwieństwie do architektury monolitowej, architektura mikroserwisowa dekomponuje aplikacje w małe, niezależnie zarządzane i wdrażane mikrousługi, umożliwiające granularną skalowalność i niezależne aktualizacje, zmniejszając w ten sposób ryzyko tego, że zmiana w jednym module spowoduje awarię innych. Natomiast takie podejście dodaje skomplikowości projektom poprzez zapotrzebowanie na obsługę komunikacji między mikroserwisami, wersjonowaniem, współdzieleniem kodu.

"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." [11, p. 11]

Zalety architektury monolitycznej: Czas napisania aplikacji - W przypadku małych i średnich aplikacji budowanie aplikacji z architekturą monolityczną jest łatwiejsze i szybsze. Zespół programistów może pracować efektywniej z ujednoliconą bazą kodu bez konfigurowania i zarządzania komunikacją między usługami. Łatwe wdrożenie - Wdrożenie architektury monolitycznej obejmuje wdrożenie pojedynczej jednostki, co jest mniej złożone i wymaga mniejszej liczby katalogów konfiguracyjnych niż systemy rozproszone. Uproszczone testowanie i debugowanie - O wiele łatwiejszym jest testowanie aplikacji monolitycznej. Ze względu na ścisłą integrację wszystkich komponentów, testy jednostkowe i integracyjne można przeprowadzać w ramach pojedynczej bazy kodu, co upraszcza proces testowania. Skalowalność - Architektura monolityczna, w przypadku małych i średnich aplikacji, zapewnia odpowiednią skalowalność poprzez replikację całej aplikacji. [15]

Zalety architektury mikroserwisowej: Skalowalność - Mikrousługi umożliwiają niezależne skalowanie różnych usług w oparciu o ich zapotrzebowanie, optymalizując wykorzystanie zasobów i zapewniając wydajną wydajność nawet podczas szczytów ruchu. Utrzymanie kodu - Mikrousługi umożliwiają równoległy rozwój przez różne zespoły, co prowadzi do szybszego rozwoju funkcji i szybszego wdrażania.

nia. Elastyczność w trakcie wyboru technologii - Każda mikrousluga może być rozwijana przy użyciu najbardziej odpowiedniego stosu technologicznego dla jej konkretnej funkcji, co promuje elastyczność i adaptowalność do różnych technologii w ramach tej samej aplikacji. Łatwe debugowanie - Lokalizowanie i rozwiązywanie błędów w poszczególnych usługach jest proste. Bezpieczeństwo - Mikrouslugi ułatwiają separację danych. Każda usługa ma swoją bazę danych, co utrudnia hakerom próbę naruszenia aplikacji. [15]

Wybór między architekturą monolityczną a mikroservisową 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 mikroservisowa oferuje benefity w przypadku wykorzystania w większych, bardziej skomplikowanych systemach.

### 3 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.

#### 3.1 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.

Zawsze należy pamiętać w trakcie developmentu aplikacji, że współdzielenie kodu ma wiele zalet, natomiast nadmierne współdzielenie kodu może spowodować nadmierne powiązania między mikroserwisami i utrudnić niezależną ewolucję poszczególnych serwisów oraz zniweczyć zalety infrastruktury mikroserwisowej. Również nadmierne współdzielenie kodu utrudnia testowanie aplikacji co może spowodować zmniejszenie jakości kodu. Również nadmierne współdzielenie kodu może spowodować błędy kaskadowe, czyli błędy które pojawiają się w kilku miejscach w systemie jednocześnie spowodowane współdzielonym kawałkiem kodu, utrudnia utrzymanie aplikacji również są powodem na to, żeby podchodzić do współdzielenia kodu ostrożnie podejmować przemyślaną decyzję w każdym konkretnym przypadku.

### 3.2 Kryteria porównania i analizy metod współdzielenia kodu

Po przeprowadzeniu dogłębnej analizy tematu i między innymi źródeł literaturowych, takich jak [11], [13], [12], [6] mogą 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.

## 4 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.

### 4.1 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. [17]
2. Libraries - najbardziej oczywiste podejście, które daje możliwość współdzielenia wszystkich rodzajów obiektów w aplikacji.
3. Client Libraries -
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. [5]

Pryncypia działania poszczególnych metod współdzielenia kodu:

Interface definition languages – są powszechnie używane w oprogramowaniu zdalnych wywołań procedur. W takich przypadkach maszyny po obu stronach łączy 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 mikroservisów i na podstawie udostępnionych plików wygenerować w każdym z mikroservisów 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 nada, 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 mikroservisów współdzielenie kodu za pomocą bibliotek kodu odbywa się za pomocą serwisów do przechowywania artefaktów i plików binarnych, takich jak Nexus. [7, 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 udostępniającym 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 mikroservisów. 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.

## **4.2 Analiza metod współdzielenia kodu dla różnych typów obiektów**

Po przeprowadzeniu analizy źródeł literaturowych oraz własnego doświadczenia dokanełem analizę 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 mikroservisami. 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

udostępnienia bezpośrednio logiki biznesowej jest dobrą praktyką w programowaniu. Współdzielenie obiektów DTO może zapewnić to, że każda usługa będzie wykorzystywać spójne struktury danych ułatwiając komunikację zmniejszając tym samym ryzyko niesójności.

Obiekty Model, współdzielenie takich obiektów zgodnie z wcześniejszą analizą nie jest zalecane, ze względu na to, że zalecany jest wykorzystywanie nie więcej niż jednej bazy dla każdego mikroservisu. Ze względu na to, nie można zleć najlepszej opcji do współdzielenia takich obiektów. Teoretycznie obiekty Model mogą być współdzielone za pomocą bibliotek ale jak zaprezentowano w [10] 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 systemach mikroservisów co poprawia jakość danych które one wyminiają się elementami systemu. W przypadku współdzielenia za pomocą bibliotek, tracimy możliwość połączenia kilku mikroservisów napisanych w różnych językach programowania w jeden system mikroservisów 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 mikroservisów napisanych w różnych językach programowania w jeden system mikroservisów.

Servisy, w których są głównym miejscem przechowywania, serwisy mogą być współdzielone za pomocą bibliotek, SDK, REST oraz serverless. W przypadku wyboru podejścia do współdzielenia mikroservisów 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. Podejście REST też dobrze pasuje do wszystkich zmierzających do współdzielenia logiki, ale natomiast w wysoko obciążonych systemach może powodować obniżenie wydajności ze względu na rozproszenie zasobów na komunikację sieciową. podejście serverless dobrze się sprawdzi w przypadku

współdzielienia mniejszych metod, zarządnie skomplikowaną logiką za pomocą serwerles może skomplikowanym, jak w przypadku REST w wysoko obciążonych systemach podejście raczej trzeba zminić na bibliotekę lub sdk.

Wyjątki, do współdzielenia wyjątków możemy zastosować SDK oraz IDL w przypadku dyjatków nie zawierających logiki związanej z obsługą błędów. Stosowanie współdzielenia za pomocą SDK w systemie mikroserwisów wspomaga stworzenie ustandaryzowanego podejścia do obsługi błędów w systemie co powoduje lepsze logowanie i raportowanie błędów oraz szybsze ich naprawienie i zniejszenie ilości porlpemów na przestrzeni czasu.

### 4.3 Definicja kryterium oceny i porównania metod współdzielenia kodu

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

**4.3.1 Definicja kryterium oceny pod kątem izolacji w przypadku IDL** Przedstawiam kryteria które wybrałem w rezultacie analizy źródeł internetowych oraz książek takich jak [6] w których w znalazłem opis tego jak musi wyglądać architektura mikro serwisowa, wady i zalety różnych rozwiązań.

1. Conflict Rate - wskaźnik tego, jak często zmiany w plikach IDL powoduje merge konflikty. Wyzwania dotyczące pracy z IDL i podejścia do ich pokonania są dobrze opisane w książce [6]. W szczegółach są opisane metody pracy z schematami idl między innymi narzędziem Protocol Buffer dostrzanego przez Alphabet oraz Avro. Można posłużyć się cytatem z zosdziłu 4 żeby zaprezentować że musimy uwzględniać ten aspekt w analizie.

"This means that old and new versions of the code, and old and new data formats, may potentially all coexist in the system at the same time. In order for the system to continue running smoothly, we need to maintain compatibility in both directions: Backward compatibility and Forward compatibility." [6, p. 112]

Również w tym rozdziale autor prezentuje dlaczego takie problemy powstają, możemy posłużyć się cytatem.

"When a data format or schema changes, a corresponding change to application code often needs to happen (for example, you add a new field to a record, and the application code starts reading and writing that field). However, in a large application, code changes often cannot happen instantaneously: old and new versions of the code, and old and new data formats, may potentially all coexist in the system at the same time." [6, p. 111]

Nusimy uważać na tożaką ilość konfliktów generuje stosowanie wybranego podejścia. Podobne konflikty często są czasozerne i nawet w trzypadku nietrudnych przypadku potrafią spowolnić zespół deweloperski oraz spowodować inne, bardziej skomplikowane błędy.

2. Version Drift Occurrence - wskaźnik tego, jak często rozwizanie powoduje użycie niezgodnych wersji IDL w różnych zespołach. Kiedy zespoły programistów pracujące nad jednym systemem zaczynają używać niezgodną wersję plików IDL mikroserwisów w systemie mikroserwisów przestają komunikować poprawnie co powoduje błędy w runtime które są bardzo trudne do wyłapania i naprawy, nieoczekiwanego zachowania systemu oraz uszkodzenia danych. W przypadku wykorzystania podejścia które nie może zapewnić odpowiedni poziom kontroli nad wersjami i izolacji rozwój i wdrażania zmian w systemie mogą być problematyczne. Możemy posłużyć się cytatem:

"With Avro, when an application wants to encode some data (to write it to a file or database, to send it over the network, etc.), it encodes the data using whatever version of the schema it knows about... When an application wants to decode some data (read it from a file or database, receive it from the network, etc.), it is expecting the data to be in some schema, which is known as the reader's schema... The key idea with Avro is that the writer's schema and the reader's schema don't have to be the same—they only need to be compatible." [6, p. 123]

W przypadku braku kompatybilności nie jest możliwe poprawnie działanie systemu.

3. Build Failure Rate - wskaźnik tego, jak często zmiany w plikach IDL powodują problemy w trakcie budowania aplikacji.

"When an application wants to decode some data (read it from a file or database, receive it from the network, etc.), it is expecting the data to be in some schema, which is known as the reader's schema. That is the schema the application code is relying on—code may have been generated from that schema during the application's build process." [6, p. 123]

Dlatego w przypadku problemów z schematem wynikającym często z nieprawidłowo wybranego podejścia do zarządzania plikami IDL.

4. Deployment Rollback Frequency - wskaźnik tego, jak często zmiany w plikach IDL powodują problemy w trakcie wdrożenia aplikacji na źródowiska w systemie mikroservisów.

"With server-side applications you may want to perform a rolling upgrade (also known as a staged rollout), deploying the new version to a few nodes at a time, checking whether the new version is running smoothly, and gradually working your way through all the nodes. This allows new versions to be deployed without service downtime, and thus encourages more frequent releases and better evolvability." [6, p. 92]

Częste reolbaki 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 planowanie projektu wybrać podejście które zapełni najmniejsze ryzyko błędów w trakcie próbnego wdrożenia i wycofania zmian.

**4.3.2 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 zminimalizować wpływ zmian w jednym serwisie na pozostałe co i daje podejściu mikroservisowemu jego wygodność i elastyczność.

W przypadku izolacji na podstawie analizy źródeł literaturowych oraz własnego doświadczenia zdefiniowałem kryteria porównania jako następujące: |

1. Izolacja interfejsów - elementy w systemie mikroserwisów muszą zapełnić jasno określone interfejsy, które pozwolą na łatwą komunikację komponentów bez wprowadzania nieposzadanych zależności i nadmiernych powiązań. Zmiany w jednym elemencie systemu nie powinny wpływać na pozostałe elementy systemu.

"A well-designed library or SDK isolates its implementation details from the user, allowing developers to interact with the API without needing to understand its inner workings. This reduces the risk of breaking changes and simplifies maintenance." [?, p. 75]

Za pomocą tego cytatu możemy wyjaśnić, w jaki sposób jasno dobrze zaprojektowane interfejsy pomagają oddzielić wewnętrzne implementacje od zewnętrznego interfejsu, umożliwiając zmiany bez wpływu na inne części systemu.

2. Izolacja wersji - w przypadku aktualizacji, nowa wersja utworzona to wprowadzeniu zmian powinna być kompatybilna wstecznie, umożliwiając migrację bez zakłóceń. Izolacja wersji zapewnia nam, że stare wersje, dostarczone przez bibliotek, sdk, REST czy serverless mogą współistnieć z nowszymi wersjami, umożliwiając tym samym – W przypadku aktualizacji SDK, nowa wersja powinna być w pełni kompatybilna wstecz, umożliwiając migrację bez zakłóceń. Izolacja wersji zapewnia, że starsze wersje SDK mogą współistnieć z nowszymi, umożliwiając płynne przejście między nimi bez wymuszania natychmiastowych zmian w istniejącej aplikacji.

"Versioning APIs and libraries carefully ensures that changes can be introduced gradually, providing compatibility for both old and new clients. This allows legacy systems to function alongside the updated ones without breaking the user experience or requiring immediate changes." [4, p. 172]

Powyższy cytat pokazuje, jak ważne jest zachowanie kompatybilności wstecznej w przypadku współdzielenia kodu w systemach mikroserwisów.

3. Izolacja zależności - w przypadku SDK, bibliotek, REST czy serverless powinniśmy minimalizować wpływ zewnętrznych zależności na aplikację końcową, pozwala to na kontrolowanie wersji zależności oraz odseparowanie od podstawowych komponentów aplikacji.

"Dependencies should be isolated to minimize their impact on the main application, allowing the code to remain flexible. One of the key principles of dependency management is to ensure that dependencies can be replaced without affecting the core business logic." [8, p. 218]

Powyższy cytat pokazuje, jak ważne jest odseparowanie i izolacja zależności, pomoże to w zachowaniu elastyczności i odseparowania bazy kodu aplikacji, co jest kluczowe dla uniknięcia niepożądanych efektów ubocznych i uproszczenia konserwacji.

#### 4.3.3 Definicja kryterium oceny pod kątem bezpieczeństwa

1. Poufność - (Confidentiality) Zapewnienie tego, że poufne dane nie będą udostępnione osobom lub systemom nieupoważnionym do tego. Zachowanie poufności jest kluczowe dla ochrony danych osobowych i korporacyjnych.

"Preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information. A loss of confidentiality is the unauthorized disclosure of information." [16, p. 22]

2. Integralność danych - (Integrity) Integralność danych koncentruje się na dokładności i wiarygodności informacji, chroniąc je przed nieautoryzowanymi modyfikacjami, zapewnia, że dane są zarówno niezawodne, jak i spójne.

"Guarding against improper information modification or destruction, including ensuring information nonrepudiation and authenticity. A loss of integrity is the unauthorized modification or destruction of information." [16, p. 22]

3. Dostępność - (Availability) Zapewnia to, że dane w systemie są dostępne, a różnego rodzaju zakłócenia dostępu do danych są zminimalizowane.



"Ensuring timely and reliable access to and use of information. A loss of availability is the disruption of access to or use of information or an information system." [16, p. 22]

4. Autentyczność - (Authenticity) Gwarancja tego, że autoryzowanym użytkownikom można zaufać, co zapobiega podszywaniu się lub manipulacjom. Polega to na weryfikacji tożsamości użytkowników i potwierdzeniu źródeł danych w celu zapewnienia autentyczności.

"The property of being genuine and being able to be verified and trusted; confidence in the validity of a transmission, a message, or message originator. This means verifying that users are who they say they are and that each input arriving at the system came from a trusted source." [16, p. 23]

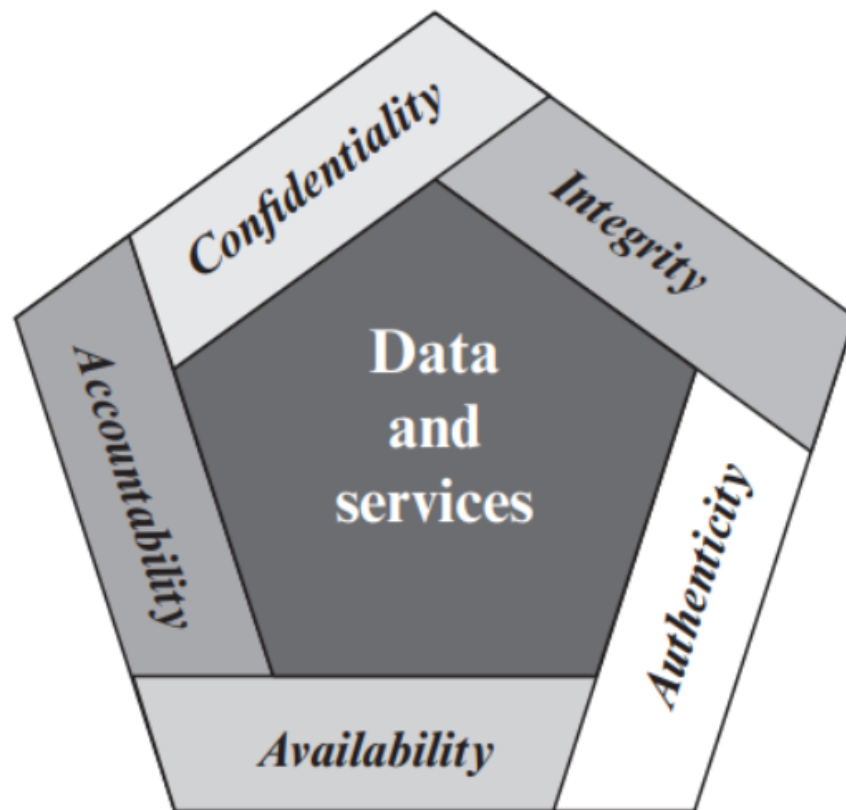
5. Odpowiedzialność - (Accountability) Definiuje zapotrzebowanie na zapisywanie w celach monitoringu działań użytkowników w systemie w celu późniejszej analizy. W trakcie analizy można wykryć podejrzanе zachowania, naruszenia przepisów bezpieczeństwa. Pomaga w zapewnieniu tego, że podmioty naruszające przepisy prawne oraz przepisy bezpieczeństwa poniosą odpowiedzialność oraz zapobiec podobnym sytuacjom w przyszłości.

"The security goal that generates the requirement for actions of an entity to be traced uniquely to that entity. This supports nonrepudiation, deterrence, fault isolation, intrusion detection and prevention, and afteraction recovery and legal action. Because truly secure systems are not yet an achievable goal, we must be able to trace a security breach to a responsible party. Systems must keep records of their activities to permit later forensic analysis to trace security breaches or to aid in transaction disputes." [16, p. 23]

1

#### 4.3.4 Definicja kryterium oceny pod kątem skalowalności

Czym jest skalowalność



**Rysunek 1.** Essential Network and Computer Security Requirements.

"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 mogą zdefiniować kryterium oceny pod kątem skalowalności jako:

Wpływ na koszty zasobów fizycznych - ocena tego, jak wybrane podejście wpływa na wykorzystaniem zasobów sprzętowych w miarę wzrostu obciążenia.

Wpływ na wydajność - ocena tego, jak wybrane podejście wpływa na wydajność systemu w miarę wzrostu obciążenia.

Wpływ na zasoby serwera - ocena tego, jak wybrane podejście wpływa na jakość wykorzystania zasobów serwera w miarę wzrostu obciążenia.

Unikanie wąskich gardeł wydajności - ocena tego, czy wybrane podejście powoduje powstanie wąskich gardeł w miarę wzrostu obciążenia systemu.

**4.3.5 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 zapełnić funkcjonowanie istniejących systemów mikroservisów w trakcie wprowadzenia 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.

"APIs, like diamonds, are forever. Once you publish an API, you are more or less stuck with it, so it is critically

important to design it well. When evolving an API, preserving backward compatibility is crucial to avoid breaking existing clients." [?, p. 75]

2. Łatwość migracji - Prostota przescia 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.

"When evolving an API, it is essential to provide a smooth migration path for clients. Deprecating old functionality rather than removing it immediately allows developers to transition gradually without breaking their systems." [?, 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 uraty danych i niespójności w trakcie rozwoju aplikacji. Posiadanie historii wersji zapewnia to, że projekty pozostają łatwe w utrzymaniu i skalowalne.

"Effective version management is crucial for tracking changes over time, allowing teams to understand what was modified, when, and by whom. A well-maintained version history ensures accountability and facilitates rollback if necessary." [14, 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 wsierając tym programistów w zrozwiązywaniu problemów i bagów albo pozwalając wyscofać zminy. Poprawia spówpracę między zespołami wykazując kto kiedy i dlaczego wprowadzał zminay co ziękasza również opowiedzialność. W brażach regulowanych prawem taka audutowalność systemu może być wymagana przez prawo.

"Tracking changes in source code is essential for understanding the evolution of a system. Without a proper

change history, debugging, auditing, and maintaining software become significantly more difficult." [14, p. 150]

#### 4.4 Ocena i porównanie metod współdzielenia kodu

**4.4.1 Ocena współdzielenia kodu za pomocą Interface definition languages** Ocena pod kątem komunikacji – zapewnia potrzebny poziom komunikacji, obiekty są generowane na podstawie definition language na etapie uruchamiania aplikacji, dalej w trakcie wykonania logiki programu takie obiekty mogą być użyte przez program bez żadnych obciążeń wydajnościowych. Obiekty wygenerowane z plików IDL mogą być wykorzystane dla komunikacji asynchronicznej takich jak brokery wiadomości jak również dla wysłania bezpośrednich zapytań między serwisami.

Metody zarządzania plikami idl w systemach :

1. SVN systemy takie jak, na przykład GIT - Rozwiązania posiadają następujące korzyści: łatwe osiągnięcie wstecznej kompatybilności, ualenie osiągnięcia jednego źródła prawdy w systemie mikroservisów, łatwie zapełnienie tego że wysłukiwe 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 osób, 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 mikroservisów i powiązane z nimi pliki 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 implementacji usług są zawsze zsynchronizowane oraz to, że takie podejście upraszcza zarządzanie zależnościami, ponieważ

cału kod przechowywany w jednym miejscu. Do wad podejścia mogą odnieść problemy ze skalowalnością zwiększenia 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 sybrepozytoriów. 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 utrzymywanie oraz złożoność w zarządzaniu synchronizacją i konfliktami w repozytorium.
5. IDL as a Service (IDLaS) - Przygotowania wcześniejszej usługi, 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, możliwości dynamicznego 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 dostarczeniu plików IDL.

Ocena pod kątem izolacji - ocena pod kątem izolacji zależy od implementacji, dla każdej z powyżej opisanych implementacji dokonałem analizy.

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

Dane do porównania zostały zebrane z źródeł literaturowych [11], [6]

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)
<b>Conflict Rate</b>	Wysoki: Merge konflikty są powszechne w przypadku wykorzystania przez kilkoma zespołami jednocześnie. Rozwiązanie takich konfliktów może być zasochłonne	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 trybie 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.
<b>Version Drift Occurrence</b>	Umiarkowana: Może wystąpić jeśli zespoły programistów nie aktualizują się do nowszej wersji regularnie.	Niski: Warygodność 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 Warygodność problemów z wersjonowaniem.
<b>Build Failure Rate</b>	Wysoki: Błędy w trakcie kompilacji mogą wystąpić w przypadku niekompatybilnych usług.	Umiarkowana: Awaryjne zdarzają się rzadko, możliwe są z przyczyną 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 dysynchronizacji repozytorium.	Niska: Minimalne ryzyko błędów w trakcie kompilacji ze względu na wymuszoną synchronizację.
<b>Deployment Rollback Frequency</b>	Wysoka: Zęste 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 ID-LaS pod kątem bezpieczeństwa.

Kryteria	Zalety	Wyzwania
<b>SVN</b>		
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
<b>Dystrybucja za pomocą bibliotek (Nexus)</b>		
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
<b>Podejście Monorepo</b>		
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
<b>IDL as a Service (IDLaS)</b>		
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 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.

"With efficient binary serialization formats, IDLs allow rapid encoding and decoding of messages. This efficiency is crucial for high-throughput systems where low latency is required. IDLs also facilitate cross-language interoperability without incurring significant performance penalties." [6, p. 123]

Ocena pod kątem wersjonowania:

Na podstawie analizy źródeł literaturowych oraz doświadczenia własnego dokonałem analizy podejścia do współdzielenia kodu za pomocą plików IDL. W tym trybaku ocena pod kątem wersjonowania różni się w zależności od podejścia do zarządzania plikami IDL, dlatego dokonałem porównania dla każdego z podejść.

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

Zalety	Opis
<b>1. SVN Systemy (Git)</b>	
śledzenie zmian	<ul style="list-style-type: none"> <li>– Każda zmiana jest rejestrowana w postaci commitu z timestampem i informacjami wskazującymi autora.</li> <li>– Rozgałęzianie i tagowanie umożliwiają izolowane przygotowanie paczki zmian i później oznaczanie stabilnych wersji.</li> <li>– Pełna historia commitów umożliwia przeprowadzanie audytów.</li> </ul>
Rozgałęzianie i tagowanie	<ul style="list-style-type: none"> <li>– Rozgałęzienie pozwala zespołom na przygotowanie nowych wersji oprogramowania w izolacji, nie przekierowując pracy innych zespołów.</li> <li>– Tagi dają możliwości oznaczania stabilnych wersji, zapewniając łatwą identyfikację wersji gotowych do użytku.</li> </ul>

<b>Zalety</b>	<b>opis</b>
Audytowalność	<ul style="list-style-type: none"> <li>– Pełna historia commitów jest zawsze przechowywana, co umożliwia audyty kodu.</li> <li>– Zapewnia odpowiedzialność za wprowadzone zmiany i usprawnia współpracę w zespołach.</li> </ul>
<b>2. Biblioteki</b>	
Wersjonowanie semantyczne	<ul style="list-style-type: none"> <li>– Artefaktom przypisywane są numery.</li> <li>– Do każdej wersji dołączone są "Release notes".</li> <li>– Repozytoria przechowują pełną historię wersji.</li> </ul>
Dzienniki zmian	<ul style="list-style-type: none"> <li>– Do każdego wydania dołączona szczegółowa dokumentacja.</li> <li>– Takie changelogi zmian zapewniają uporządkowaną historię modyfikacji.</li> <li>– Takie podejście zapewnia to przejrzystość śledzenia zmian.</li> </ul>
Centralne repozytorium artefaktów	<ul style="list-style-type: none"> <li>– Centralne repozytorium przechowuje kompletną historię wersji.</li> <li>– Dzięki temu zespoły mogą, w razie potrzeby, pobierać, porównywać i przywracać wcześniejsze stabilne wersje.</li> </ul>
<b>3. Podejście Monorepo</b>	
Zunifikowany dziennik zmian	<ul style="list-style-type: none"> <li>– Wszystkie mikroserwisy w systemie i powiązane z nimi pliki IDL znajdują się w jednym repozytorium.</li> <li>– Historia commitów zapewnia całościowy widok zmian.</li> </ul>
Spójność	<ul style="list-style-type: none"> <li>– Wszystkie zmiany są wprowadzane do jednego repozytorium, co zapewnia synchronizację.</li> <li>– Takie podejście minimalizuje ryzyko różnic w wersjach pomiędzy usługami.</li> <li>– Spójne wersjonowanie poprawia stabilność systemu.</li> </ul>

Zalety	opis
Łatwość cofnięcia	<ul style="list-style-type: none"> <li>– Scentralizowana historia ułatwia powrót do poprzedniego stabilnego stanu.</li> <li>– Takie podejście ułatwia debugowanie i zwiększa stabilność systemu.</li> </ul>
<b>4. IDL as a Service (IDLaS)</b>	
Zintegrowana kontrola wersji	<ul style="list-style-type: none"> <li>– Dostarczana usługa obejmuje wbudowaną kontrolę wersji.</li> <li>– Dostarczana usługa daje dostęp do historii wersji za pomocą API.</li> </ul>
Automatyczna dokumentacja	<ul style="list-style-type: none"> <li>– IDLaS często udostępniają interfejsy webowe.</li> <li>– Ze względu na to, że IDL jest udostępniane jako usługa, każdy użytkownik może uzyskać dostęp do historii wersji za pomocą API</li> </ul>

#### 4.4.2 Ocena współdzielina 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 wcześniejszymi kryteriami oceny, w przypadku oceny izolacji bibliotek kryteria oceny są następujące: ocena pod kątem izolacji interfejsów, izolacja wersji, izolacja zależności.

Izolacja interfejsów: Biblioteki, zwykle udostępniają zdefiniowany interfejs. Dobrze ustrukturyzowana biblioteka izoluje skomplikowane elementy logiki od swoich użytkowników. Gdy są użyte prawidłowo, pozwalają na zmianę lub aktualizację kodu przy minimalnym wpływie na kod wykorzystujący bibliotekę.

"A well-designed library or SDK isolates its implementation details from the user." [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.

"Versioning APIs and libraries carefully ensures that changes can be introduced gradually." [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.

"Dependencies should be isolated to minimize their impact on the main application." [8, p. 218]

Ocena pod kątem bezpieczeństwa:

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

Poufność: Biblioteki które są wykorzystywane dla preteżania poufnych danych powinny pochodzić z sprawdzonych źródeł i ciągle aktualizowane aby zapobiec utracę danych. Ich integracja powinna wymuszać ścisłe kontrole dostępu.

"Ensuring that libraries handling sensitive information are sourced from reputable channels and regularly updated is critical to maintaining confidentiality throughout an application." [2, p. 82]

Integralność danych: Biblioteki często korzystają z sum kontrolnych, certyfikatów, podpisów cyfrowych przeprowadzają obszerne testy, aby mieć pewność, że przetwarzane dane pozostają bezpieczne.

"The integrity of a library is maintained through rigorous testing and the use of cryptographic measures that detect any unauthorized changes to the code or data it processes." [2, p. 84]

**Dostępność:** Dostępność może zostać naruszona, jeśli biblioteki zewnętrzne staną się nieobsługiwane lub niedostępne lub wewnętrzne w przypadku awarii narzędzia do zarządzania bibliotekami. Dlatego jest ważnym wykożystanie managarów zależności które są zaufane i w tym przupadku możemy pomaga zapewnić, że aplikacje pozostaną dostępne nawet w miarę ewolucji bibliotek.

"By leveraging modern dependency management tools and regularly updating libraries, developers can mitigate risks to availability due to deprecated or unsupported components." [2, p. 85]

**Autentyczność:** W przypadku bibliotek jest zapewniania za pomocą sprawdnie jej pochodzenia za pomocą podpisów cyfrowych i sum kontrolnych dostarczonych przez wydawcę. Zapobiega to ryzyku włączenia wykonania złośliwego kodu po stronie klienta biblioteki.

"Authenticating libraries through digital signatures and secure distribution channels is essential to ensuring that only verified, unmodified code is used in production environments." [2, p. 87]

**Odpowiedzialność:** W przypadku bibliotek przejrzysta historia mnoże zostać zapełniona za pomocą menedżerów pakietów które umożliwiają śledzenie pochodzenia zmian. Wspomaga to w zabezpieczeniu odpowiedzialności w przypadku problemów.

"A well-documented version history is indispensable for accountability, as it enables developers to trace and audit every modification made to a library over time." [2, p. 88]

**Ocena pod kątem skalowalności:**

Libraries are generally optimized for specific tasks within one language. However, if a library is not designed for high-concurrency environments, it may require additional hardware resources as the

load increases. For example, intensive data processing libraries may consume more memory or CPU when dealing with large datasets. Supporting Idea: Studies on computational efficiency often note that non-optimized libraries can become bottlenecks under heavy load, leading to increased physical resource costs.

"Non-optimized libraries can quickly become bottlenecks in high-concurrency environments, forcing systems to consume additional CPU and memory resources as load increases." [4, p. 192]

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." [6, p. 111]

Ustrukturyzowana historia wersji:

"APIs, like diamonds, are forever. Once you publish an API, you are more or less stuck with it; hence, maintaining a clear, structured version history through semantic versioning is critical for ensuring that developers know exactly when breaking changes occur." [2, p. 75]

text

"When an API change turns out to be problematic, having a stable version available through semantic versioning allows developers to quickly roll back to a previous release and restore system stability." [2, p. 75]

text

"Inadequate versioning can lead to subtle incompatibilities, resulting in data inconsistencies that are difficult to debug. Hence, maintaining strict backward compatibility is crucial to avoid such pitfalls." [2, p. 75]



**4.4.3 Ocena współdziałania kodu za pomocą SDK** Ocena pod kątem komunikacji: zapewnia potrzebny poziom komunikacji, mechanizm współdziałania 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 wcześniejszymi 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 staranności w trakcie trzymania.

"SDK isolates its implementation details from the user, allowing developers to interact with the API without needing to understand its inner workings." [2, 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 projektowany 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 wielką ilością zależności. Izolowanie tych zależności jest krytyczne, aby uniknąć konfliktów.

"...ensuring that dependencies can be replaced without affecting the core business logic" [8, p. 218]

Ocena pod kątem bezpieczeństwa:

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

Poufność: SDK w przypadku połączenia wielu narzędzi i bibliotek, muszą zapewnić, że wszystkie zawarte komponenty przestrzegają ścisłych zasad kontroli dostępu. Zmniejsza to ryzyko nieautoryzowanego ujawnienia danych.

"SDKs, when integrated with stringent access controls and privacy policies, help shield sensitive data from unauthorized access, thus preserving confidentiality across all integrated modules." [9, para 4]

Integralność danych: SDK wymaga, aby były one kompleksowo testowane, aby zapewnić, że zmiany nie naruszają integralności danych. Różne frameworki do testowania bezpieczeństwa SDK pomagają w wykrywaniu wszelkich podatności po modyfikacji.

"Robust testing frameworks within SDKs are critical for maintaining integrity, ensuring that updates do not inadvertently alter data structures or application behavior." [9, para 5]

Dostępność: SDK muszą być zaprojektowane tak, aby działały zawsze, nawet przy dużym obciążeniu.

"The modular and redundant design of well-architected SDKs contributes significantly to system availability, ensuring that critical services remain accessible even under peak loads." [9, para 6]

Autentyczność: Autentyczność w przypadku z SDK może zostać zapewniona za pomocą weryfikacji dostawców i stosowanie środków kryptograficznych do walidacji ich komponentów. Warto dokonywać takich działań aby zapobiec wykonaniu złośliwego kodu w naszym systemie.

"Verifying the authenticity of an SDK via cryptographic signatures and secure distribution channels is vital to ensure that the components used are genuine and have not been tampered with." [9, para 7]

Odpowiedzialność: SDK zazwyczaj posiadają przeróżne mechanizmy zarządzania wersjami. Takie metody zarządzania wersjami

umożliwia deweloperom śledzenie pochodzenia wszelkich zmian lub problemów. Pomaga to zapłacić odpowiedzialność programistów za przygotowane zmiany.

"Comprehensive logging and version tracking in SDKs provide the necessary audit trails that hold developers accountable for every change made, thereby ensuring that any issues can be traced back to their origin." [9, para 7]

Ocena pod kątem skalowalności:

SDK zazwyczaj zawierają wiele narzędzi czy komponentów. Ta kompleksowość oznacza, że upraszczają dewelopment, mają tendencję do wyższego zużycia zasobów pamięci i procesora w porównaniu z, na przykład, IDL. Różne raporty dotyczące skalowania aplikacji w chmurach podkreślają, że SDK mogą powodować dodatkowe obciążenie na środowiskach z dużym obciążeniem.

"SDKs, though beneficial for rapid development by bundling multiple tools, tend to impose a higher baseline resource consumption, especially under high-load production scenarios." [9, para 3]

"Third-party SDKs can significantly increase memory and CPU usage, becoming a notable overhead in environments with high concurrent demands." [1, p. 4]

Ocena pod kątem wersjonowania:

SDK mogą zawierać wiele bibliotek i narzędzi, z których każde może mieć własną historię wersji. Wersjonowanie SDK może być złożone, ponieważ musi odzwierciedlać zmiany we wszystkich wewnętrznych komponentach. Prowadzi to do bardziej złożonej historii wersji, która wymaga dobrej dokumentacji i koordynacji.

"SDKs encapsulate multiple interdependent modules. As a result, maintaining a structured version history across all components is challenging but essential in order to manage compatibility and coordinate upgrades effectively." [9, para. 3]

Przywracanie kodu do stabilnej wersji: Wycofywanie zmian w przypadku SDK może być trudnym, ponieważ SDK może integrować

wiele komponentów. Ta złożoność może sprawić, że wycofywanie jest bardziej czasochłonne i podatny na błędy, jeśli nie jest zarządzany prawidłowo.

"SDKs, by virtue of bundling several interdependent tools and libraries, require a coordinated rollback mechanism to ensure that all components revert to a stable, compatible state. This adds complexity compared to single-module libraries or IDL files." [9, 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.

"The interdependencies within an SDK mean that even small version mismatches can lead to significant inconsistencies or data loss, making rigorous testing and coordinated versioning critical." [9, para. 3]

**4.4.4 Ocena współdzielena kodu za pomocą API** Ocena pod kątem komunikacji:

**4.4.5 Ocena współdzielena kodu za pomocą ServerLess**

**4.5 Opis części praktycznej**

**4.6 Podsumowanie**

## Spis tablic

1	Porównanie podejść do współdzielenia kodu w systemach mikroserwisów .....	31
2	Porównanie SVN, bibliotek, Monorepo i IDLaS pod kątem bezpieczeństwa.....	32
3	Porównanie podejść do zarządzania kodem, wersjonowania i repozytoriami: Git, biblioteki, Monorepo i IDLaS.....	35

## Spis rysunków

1	Essential Network and Computer Security Requirements. .	26
---	---	----

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

## 5 Appendices

### 5.1 Sample Code

### 5.2 Sample Data