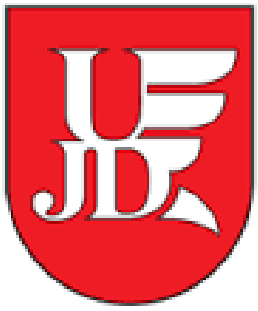
UNIWERSYTET JANA DŁUGOSZA W CZĘSTOCHOWIE



Wydział Nauk Ścisłych, Przyrodniczych i Technicznych

Kierunek: **Informatyka**
Specjalność: **Programowanie gier komputerowych**

**Przemysław Kamiński**
Nr albumu: **88751**

# Sposoby optymalizacji aplikacji internetowych. Ways to optimize web applications

Praca magisterska
przygotowana pod kierunkiem
dr hab. Bożeny Woźnej-Szcześniak, prof. UJD

Częstochowa, 2025

## Streszczenie

Celem niniejszej pracy jest identyfikacja i ocena sposobów optymalizacji aplikacji webowych poprzez analizę wydajności protokołów komunikacyjnych oraz formatów wymiany danych. Badania obejmują protokoły HTTP/1.1, HTTP/2, GraphQL, gRPC, SOAP, WebSocket Secure oraz WebSocket, a także formaty danych takie jak JSON, XML, Protocol Buffers i MessagePack. Analiza koncentruje się na porównaniu rozmiaru przesyłanych danych, czasu przetwarzania oraz kompatybilności pomiędzy systemami w kontekście możliwości optymalizacyjnych.

Dodatkowo w pracy oceniono wpływ zastosowania technologii WebAssembly na wydajność przetwarzania danych po stronie klienta w porównaniu z tradycyjnym podejściem opartym na języku JavaScript jako potencjalną metodę optymalizacji aplikacji. Eksperymenty zostały przeprowadzone z wykorzystaniem dedykowanej aplikacji testowej, umożliwiającej rzetelny pomiar parametrów wydajnościowych w kontrolowanym środowisku.

Uzyskane wyniki pozwalają określić zależności pomiędzy wyborem protokołów komunikacyjnych, formatów danych oraz technologii wykonawczych a osiąganą wydajnością systemu. Na ich podstawie sformułowano praktyczne wnioski i rekomendacje dotyczące optymalnego doboru rozwiązań technologicznych, które mogą wspierać proces optymalizacji nowoczesnych aplikacji webowych oraz architektur rozproszonych.



## Abstract

The aim of this thesis is to identify and evaluate methods for optimizing web applications through performance analysis of communication protocols and data exchange formats. The study covers protocols such as HTTP/1.1, HTTP/2, GraphQL, gRPC, SOAP, and WebSocket, as well as data formats including JSON, XML, Protocol Buffers, and MessagePack. The analysis focuses on data size, processing time, and cross-system compatibility in the context of optimization opportunities.

Additionally, the thesis evaluates the impact of using WebAssembly on client-side data processing performance compared to the traditional JavaScript-based approach as a potential application optimization method. The experiments were conducted using a dedicated test application that enabled reliable performance measurements in a controlled environment.

The obtained results allow identifying relationships between the choice of communication protocols, data formats, and execution technologies and the resulting system performance. Based on these findings, practical conclusions and recommendations regarding optimal technology selection are formulated to support the optimization of modern web applications and distributed architectures.



**Słowa kluczowe:**Rust, WebAssembly, JavaScript, Sieci komputerowe, Protokoły komunikacyjne, Optymalizacja, Serializacja, Deserializacja, Klient–serwer

# Spis treści

# Wstęp

Dynamiczny rozwój aplikacji webowych oraz rosnące wymagania dotyczące ich skalowalności, responsywności i niezawodności sprawiają, że optymalizacja wydajności komunikacji sieciowej staje się jednym z kluczowych aspektów projektowania nowoczesnych systemów informatycznych. Współczesne aplikacje coraz częściej opierają się na architekturach rozproszonych, w których wymiana danych pomiędzy klientem a serwerem, a także poszczególnymi usługami, odbywa się z wykorzystaniem różnych protokołów komunikacyjnych oraz formatów danych. Świadomy wybór i optymalne wykorzystanie dostępnych technologii w tym zakresie ma bezpośredni wpływ na czas odpowiedzi systemu, obciążenie sieci oraz koszty przetwarzania po obu stronach komunikacji.

Celem niniejszej pracy magisterskiej jest identyfikacja i ocena skutecznych metod optymalizacji aplikacji webowych poprzez analizę i porównanie wydajności wybranych protokołów komunikacyjnych oraz formatów wymiany danych. W pracy szczególną uwagę poświęcono porównaniu protokołów HTTP/1.1, HTTP/2[34], GraphQL[26], gRPC[32], SOAP[44] oraz WebSocket[49], które reprezentują różne podejścia do komunikacji w środowiskach sieciowych i oferują odmienne możliwości optymalizacyjne. Analizie poddano również popularne formaty danych, takie jak JSON[37], XML[31], Protocol Buffers[6] oraz MessagePack[20], uwzględniając ich rozmiar, czas serializacji i deserializacji oraz kompatybilność pomiędzy różnymi systemami jako potencjalne obszary optymalizacji.

Dodatkowym celem pracy jest zbadanie wpływu zastosowania technologii WebAssembly[27] jako metody optymalizacji przetwarzania danych po stronie klienta w porównaniu z tradycyjnym podejściem opartym na języku JavaScript. W tym kontekście przeprowadzono eksperymenty mające na celu ocenę różnic w czasie wykonywania operacji, zużyciu zasobów oraz potencjalnych korzyści wydajnościowych wynikających z wykorzystania WebAssembly w aplikacjach webowych.

Podsumowując, praca ma na celu dostarczenie praktycznych i eksperymentalnie potwierdzonych strategii optymalizacyjnych oraz rekomendacji, które mogą stanowić wsparcie przy wyborze i optymalizacji protokołów komunikacyjnych, formatów danych oraz technologii wykonawczych w projektowaniu wydajnych i nowoczesnych aplikacji webowych.

# Rozdział 1

# Wprowadzenie do problematyki doboru technologii

## 1.1 Znaczenie doboru technologii w aplikacjach webowych

Współczesne aplikacje webowe stanowią podstawę funkcjonowania wielu systemów informatycznych wykorzystywanych w biznesie, administracji publicznej oraz życiu codziennym. Rosnąca liczba użytkowników, potrzeba obsługi dużych wolumenów danych oraz wymagania dotyczące niskich czasów odpowiedzi powodują, że zagadnienia związane z wydajnością i skalowalnością systemów stają się kluczowe już na etapie projektowania architektury. Jednym z najważniejszych czynników wpływających na te cechy jest dobór odpowiednich technologii komunikacyjnych oraz formatów wymiany danych.

W architekturach rozproszonych komunikacja pomiędzy komponentami systemu odbywa się za pośrednictwem sieci komputerowej, która wprowadza dodatkowe opóźnienia oraz ograniczenia przepustowości. Niewłaściwy wybór protokołu komunikacyjnego lub formatu danych może prowadzić do nadmiernego obciążenia sieci, zwiększonego zużycia zasobów obliczeniowych oraz pogorszenia doświadczeń użytkownika końcowego. Z tego względu świadomy dobór technologii powinien być oparty nie tylko na ich popularności, lecz przede wszystkim na analizie wymagań danego problemu.

## 1.2 Charakterystyka współczesnych systemów rozproszonych

Nowoczesne systemy informatyczne coraz częściej projektowane są w oparciu o architektury rozproszone[46], takie jak architektura klient–serwer, mikroserwisy[39] czy systemy oparte na komunikacji zdarzeniowej[38]. W tego typu rozwiązaniach poszczególne komponenty systemu działają niezależnie i komunikują się ze sobą za pomocą jasno zdefiniowanych interfejsów.

Taki model projektowy umożliwia łatwiejszą skalowalność oraz rozwój systemu, jednak jednocześnie zwiększa znaczenie wydajnej i niezawodnej komunikacji. Każde wywołanie zdalne wiąże się z kosztami transmisji danych, serializacji[43] i deserializacji[29] komunikatów oraz przetwarzania po obu stronach połączenia. W praktyce oznacza to, że nawet niewielkie różnice w zastosowanych technologiach mogą prowadzić do zauważalnych różnic w wydajności całego systemu.

## 1.3 Protokoły komunikacyjne jako fundament wymiany danych

Protokoły komunikacyjne definiują sposób, w jaki dane są przesyłane pomiędzy uczestnikami komunikacji. W aplikacjach webowych powszechnie wykorzystywane są protokoły oparte na rodzinie HTTP, jednak wraz z rozwojem technologii pojawiły się również alternatywne rozwiązania, takie jak gRPC, WebSocket, GraphQL. Każdy z tych protokołów oferuje inne właściwości w zakresie wydajności, sposobu zestawiania połączeń, obsługi strumieniowania danych oraz kompatybilności z istniejącą infrastrukturą.

Dobór protokołu komunikacyjnego powinien uwzględniać charakter wymiany danych, częstotliwość komunikacji, wymagania dotyczące opóźnień oraz środowisko uruchomieniowe aplikacji. Przykładowo, protokoły oparte na długotrwałych połączeniach mogą być bardziej efektywne w aplikacjach czasu rzeczywistego, natomiast klasyczne podejście żądanie–odpowiedź sprawdza się w prostszych scenariuszach komunikacyjnych.

## 1.4 Uzasadnienie podjęcia badań

Różnorodność dostępnych protokołów komunikacyjnych, formatów danych oraz technologii wykonawczych sprawia, że projektanci systemów informatycznych stają przed trudnym zadaniem wyboru najbardziej odpowiednich rozwiązań. Brak jednoznacznych odpowiedzi oraz silne uzależnienie wyników od kontekstu zastosowania powodują, że decyzje te często podejmowane są intuicyjnie.

Celem niniejszej pracy jest dostarczenie empirycznych danych oraz praktycznych wniosków, które mogą wspierać proces podejmowania decyzji technologicznych. Przeprowadzone analizy i eksperymenty pozwalają lepiej zrozumieć zależności pomiędzy wyborem technologii a wydajnością aplikacji webowych.

## 1.5 Słownik pojęć i skrótów

**API (*Application Programming Interface*)** – Zbiór reguł, definicji oraz mechanizmów umożliwiających komunikację pomiędzy różnymi komponentami oprogramowania[35].

**HTTP (*Hypertext Transfer Protocol*)** – Protokół komunikacyjny wykorzystywany do przesyłania danych w sieci WWW w modelu klient–serwer.

**WebSocket** – Protokół umożliwiający utrzymywanie stałego, dwukierunkowego połączenia pomiędzy klientem a serwerem, wykorzystywany do komunikacji w czasie rzeczywistym.

**gRPC** – Wysokowydajny framework komunikacyjny typu RPC, oparty na protokole HTTP/2 oraz binarnym formacie *Protocol Buffers*.

**RPC (*Remote Procedure Call*)** – Mechanizm komunikacji międzyprocesowej umożliwiający wykonywanie procedur lub funkcji w zdalnym systemie komputerowym tak, jakby były wywoływane lokalnie[42].

**Protocol Buffers (*Protobuf*)** – Binarny format serializacji danych opracowany przez firmę Google, charakteryzujący się kompaktową reprezentacją oraz efektywnym przetwarzaniem, wykorzystywany m.in. w komunikacji gRPC.

**SOAP (*Simple Object Access Protocol*)** – Protokół komunikacyjny oparty na wymianie komunikatów XML, stosowany w architekturach usług sieciowych.

**MQTT (*Message Queuing Telemetry Transport*)** – Lekki protokół komunikacyjny typu publish–subscribe, przeznaczony do systemów o ograniczonych zasobach oraz komunikacji w sieciach o niskiej przepustowości[13].

**JSON (*JavaScript Object Notation*)** – Tekstowy format wymiany danych oparty na składni JavaScript, charakteryzujący się czytelnością dla człowieka oraz łatwością parsowania przez maszyny.

**MessagePack** – Binarny format serializacji danych zaprojektowany jako wydajniejsza alternatywa dla JSON, oferujący mniejszy rozmiar oraz szybsze przetwarzanie przy zachowaniu podobnej elastyczności.

**Apache Avro** – System serializacji danych opracowany w ramach projektu Apache Hadoop, wykorzystujący schematy JSON do definiowania struktury danych oraz zapewniający kompaktową reprezentację binarną[25].

**BSON (*Binary JSON*)** – Binarny format serializacji dokumentów wzorowany na JSON, wykorzystywany m.in. w bazie danych MongoDB, rozszerzający JSON o dodatkowe typy danych oraz umożliwiający efektywne przechowywanie i przetwarzanie[28].

**Serializacja** – Proces przekształcania struktury danych do postaci umożliwiającej jej zapis lub transmisję pomiędzy systemami[43].

**Deserializacja** – Proces odtwarzania pierwotnej struktury danych na podstawie jej zserializowanej reprezentacji[29].

**XML (*eXtensible Markup Language*)** – Rozszerzalny język znaczników wykorzystywany do opisu i strukturyzacji danych tekstowych.

**Mikroserwis** – Architektoniczny wzorzec projektowy, w którym aplikacja składa się z wielu niezależnych, luźno powiązanych usług, z których każda realizuje konkretną funkcjonalność biznesową i może być rozwijana oraz wdrażana niezależnie.

**Systemy rozproszone** – Zbiór niezależnych komponentów obliczeniowych połączonych siecią, które współpracują ze sobą w celu realizacji wspólnego zadania, prezentując się użytkownikowi jako jeden spójny system.

**Komunikacja zdarzeniowa (*Event-Driven Architecture*)** – Architektura oprogramowania, w której komponenty systemu komunikują się poprzez generowanie, wykrywanie i reagowanie na zdarzenia, umożliwiając luźne powiązanie oraz asynchroniczną wymianę informacji.

**Strumieniowanie danych** – Technika przetwarzania i przesyłania danych w sposób ciągły, bez konieczności oczekiwania na kompletny zbiór danych[45].

**Head-of-Line Blocking (*HOL*)** – Zjawisko polegające na blokowaniu przetwarzania kolejnych żądań przez opóźnienie jednego z nich w ramach tego samego połączenia[33].

**WebAssembly (*Wasm*)** – Binarny format wykonywalnego kodu umożliwiający uruchamianie aplikacji o wysokiej wydajności w środowisku przeglądarek internetowych.

**TLS (*Transport Layer Security*)** – Protokół kryptograficzny zapewniający poufność, integralność oraz uwierzytelnianie danych przesyłanych w sieci komputerowej[47].

**Architektura klient–serwer** – Model architektoniczny, w którym klient inicjuje żądania, a serwer przetwarza je i zwraca odpowiedzi[40].

**Silnik JavaScript V8** – Wysokowydajny silnik JavaScript opracowany przez firmę Google, wykorzystywany m.in. w przeglądarce Google Chrome oraz środowisku Node.js[5].

**DOM (*Document Object Model*)** – Model obiektowy dokumentu HTML lub XML, który reprezentuje jego strukturę w postaci drzewa obiektów, umożliwiając programom (np. skryptom JavaScript) dynamiczny dostęp do zawartości, struktury oraz stylów strony internetowej[30].

**Web Worker** – Mechanizm platformy webowej umożliwiający uruchamianie skryptów JavaScript w tle, w osobnym wątku względem głównego wątku interfejsu użytkownika, co pozwala na wykonywanie kosztownych obliczeń bez blokowania renderowania strony oraz interakcji użytkownika. Web Worker nie ma bezpośredniego dostępu do drzewa DOM, a komunikacja z głównym wątkiem odbywa się za pomocą asynchronicznego przesyłania komunikatów[48].

# Rozdział 2

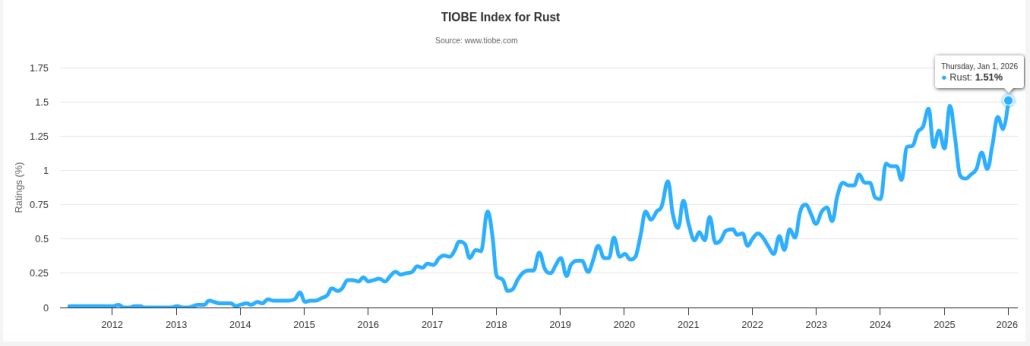
# Technologie wykorzystane w projekcie

## 2.1 Rust

**Rust** jest nowoczesnym językiem programowania systemowego, który kładzie duży nacisk na bezpieczeństwo pamięci, wydajność oraz wielowątkowość. Dzięki mechanizmowi własności (ownership) oraz statycznej analizie błędów w czasie kompilacji, Rust minimalizuje ryzyko występowania błędów takich jak wycieki pamięci czy dereferencja pustych wskaźników. Rust zdobył popularność również dzięki nowoczesnemu systemowi typów i możliwości pisania kodu niskopoziomowego bez rezygnacji z bezpieczeństwa[19].

Wybór języka Rust był podyktowany potrzebą uzyskania niskich czasów odpowiedzi oraz stabilności działania aplikacji przy dużej liczbie równoległych zapytań. Dodatkowo paczki wykorzystane do badań nie posiadały zbędnych funkcjonalności, co umożliwiło skupienie się na analizie wyników i minimalnej implementacji rozwiązań.

Według Tiobe Index, Rust zyskał największą dotychczas popularność 1 stycznia 2026 roku.




Rysunek 2.1: Pozycja języka Rust w rankingu Tiobe Index.

Jak widać na Rysunku 2.1, Rust zyskuje coraz większą popularność wśród programistów, co potwierdza rosnące zainteresowanie nim w przemyśle i projektach open source.

## 2.2 JavaScript

**JavaScript** jest językiem skryptowym powszechnie wykorzystywanym do tworzenia aplikacji webowych[36]. W projekcie został użyty głównie jako punkt odniesienia dla tradycyjnych interfejsów użytkownika w porównaniu z technologią WebAssembly. Dzięki swojej uniwersal-

ności i dużemu ekosystemowi bibliotek, JavaScript nadal pozostaje podstawowym językiem front-endowym.

## 2.3 TypeScript

**TypeScript** jest statycznie typowanym nadzbiorem języka JavaScript, rozwijanym przez firmę Microsoft. Rozszerza on JavaScript o system typów oraz mechanizmy weryfikacji poprawności kodu na etapie kompilacji, co znacząco ułatwia tworzenie, utrzymanie i rozwój większych aplikacji[11].

## 2.4 Framework Actix-web

**Actix-web** jest asynchronicznym frameworkiem webowym dla języka Rust, opartym na modelu aktorów. Charakteryzuje się wysoką wydajnością oraz niskim narzutem czasowym, co sprawia, że jest jedną z najszybszych opcji w ekosystemie[24].

Framework Actix-web został wybrany ze względu na:

- wysoką wydajność potwierdzoną testami benchmarkowymi,
- dobrą integrację z ekosystemem Rust,
- wsparcie dla programowania asynchronicznego.

| Framework | Requests per second (higher is better) | Latency (lower is better) | Stability (higher is better) | Memory (lower is better) |
|---|---|---|---|---|
| Rust / Actix | 21965 | 1.4 ms | 66.60 % | 16.6 MB |
| Rust / Axum | 21030 | 1.6 ms | 71.96 % | 8.5 MB |
| C# (.NET) / ASP.NET Core | 14707 | 1.2 ms | 2.64 % | 136.5 MB |
| JavaScript (Node.js) / Express | 5766 | 5.5 ms | 64.50 % | 82.5 MB |
| Go / Gin | 3546 | 1.0 ms | 1.14 % | 16.7 MB |
| Python / FastAPI (Uvicorn) | 1185 | 21.0 ms | 21.21 % | 41.2 MB |
| Java (Temurin) / Spring Boot (MVC) | 1105 | 1.7 ms | 1.67 % | 596.8 MB |
| Python / Flask (Gunicorn) | 1092 | 7.7 ms | 9.22 % | 90.3 MB |
| Python / Django (Gunicorn) | 950 | 8.8 ms | 10.33 % | 130.1 MB |
| PHP / Laravel | 299 | 101.7 ms | 56.47 % | 84.2 MB |

Rysunek 2.2: Framework Actix-web w porównaniu z innymi frameworkami.

Rysunek 2.2 ilustruje doskonałe wyniki wydajnościowe frameworka Actix w zakresie obsługi intensywnego ruchu – cecha szczególnie istotna w aplikacjach wymagających minimalnych czasów odpowiedzi. REST API zaimplementowano zgodnie z nowoczesnymi wzorcami programowania opisanymi przez Luca Palmieriego[15].

## 2.5 Framework Tonic

**Tonic** jest frameworkiem do implementacji gRPC w języku Rust, opartym na bibliotece `tokio` oraz protokole HTTP/2[23]. W projekcie został wykorzystany do:

- implementacji serwera gRPC,

- definiowania kontraktów komunikacyjnych w postaci plików `.proto`,
- realizacji wydajnej komunikacji klient–serwer.

Framework Tonic umożliwił stworzenie stabilnej i szybkiej komunikacji typu RPC, co było kluczowe dla testów wydajnościowych i porównawczych z GraphQL.

## 2.6 tungstenite-rs

**tungstenite-rs** jest biblioteką w Rust umożliwiającą obsługę protokołu WebSocket. Została wykorzystana do[21]:

- obsługi komunikacji w czasie rzeczywistym,
- przesyłania danych bez konieczności inicjowania nowych połączeń HTTP,
- testów alternatywnych modeli komunikacji.

## 2.7 h2load

`h2load` jest narzędziem służącym do testowania wydajności aplikacji korzystających z protokołu HTTP/2[22]. W projekcie zostało wykorzystane do:

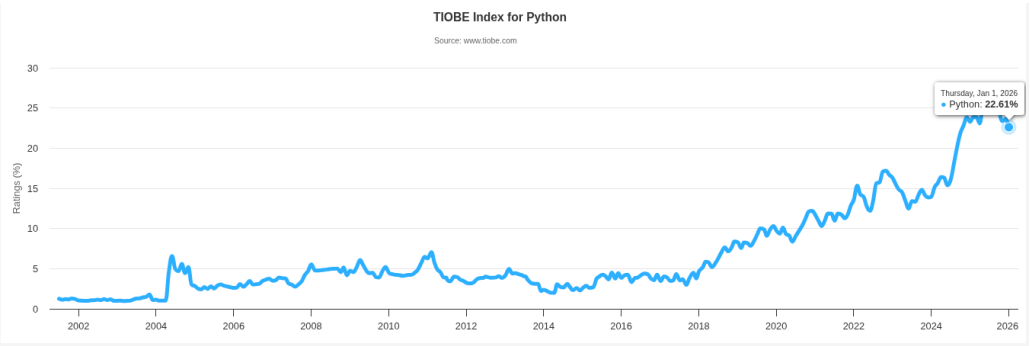
- generowania dużej liczby równoległych zapytań,
- pomiaru czasów odpowiedzi serwera,
- analizy zachowania systemu pod obciążeniem.

## 2.8 Python

**Python** jest językiem wysokiego poziomu, który w projekcie został użyty głównie pomocniczo, do:

- analizy wyników testów wydajnościowych,
- generowania wykresów porównawczych,
- przetwarzania danych pomiarowych.

Python pozostaje jednym z najpopularniejszych języków programowania[41], co obrazuje Rysunek 2.3.




Rysunek 2.3: Pozycja języka Python w rankingu Tiobe Index.

## 2.9 GraphQL

**GraphQL** jest językiem zapytań do API, pozwalającym klientowi precyzyjnie określić, jakie dane są potrzebne. W projekcie został wykorzystany do:

- realizacji zapytań o złożone struktury danych,
- testów wydajnościowych i porównawczych.

## 2.10 gRPC

**gRPC** jest mechanizmem komunikacji RPC opartym na protokole HTTP/2 oraz formacie binarnym Protocol Buffers. W projekcie zostało wykorzystane do:

- komunikacji między komponentami systemu,
- przeprowadzania testów wydajnościowych.

## 2.11 WebAssembly

**WebAssembly** (Wasm) to binarny format wykonywalny, który umożliwia natywne uruchamianie kodu Rust bezpośrednio w przeglądarce internetowej. Proces integracji WebAssembly z JavaScriptem został przeprowadzony zgodnie z metodologią przedstawioną w publikacji[7].

Technologia ta została wykorzystana w projekcie do realizacji następujących celów:

- delegowania wybranych elementów logiki aplikacyjnej do wykonania po stronie klienta,
- przeprowadzenia eksperymentalnej analizy komparatywnej wydajności w odniesieniu do tradycyjnego JavaScriptu,
- optymalizacji parametrów wydajnościowych aplikacji webowej.

## 2.12 rumqttc

**rumqttc** jest asynchroniczną biblioteką w języku Rust służącą do obsługi protokołu MQTT (Message Queuing Telemetry Transport). Umożliwia ona implementację klientów MQTT, oferując niskie opóźnienia oraz wysoką wydajność komunikacji[16].

W projekcie biblioteka `rumqttc` została wykorzystana do:

- testów komunikacji typu publish–subscribe,
- analizy wydajności lekkich protokołów komunikacyjnych,
- porównania MQTT z innymi mechanizmami wymiany danych.

## 2.13 wasm-pack

**wasm-pack** jest narzędziem wspierającym proces kompilacji projektów napisanych w języku Rust do formatu WebAssembly. Automatyzuje on generowanie pakietów WASM, integrację z ekosystemem JavaScript oraz publikację modułów.

W projekcie `wasm-pack` został wykorzystany do:

- kompilacji modułów Rust do WebAssembly,
- przygotowania interfejsów JavaScript do komunikacji z WASM,
- budowy aplikacji demonstracyjnych uruchamianych w przeglądarce.

## 2.14 Juniper

**Juniper** jest biblioteką w języku Rust służącą do implementacji serwerów GraphQL. Umożliwia definiowanie schematów, zapytań oraz mutacji w sposób silnie typowany, zgodny z paradygmatami języka Rust[9].

W projekcie biblioteka `Juniper` została użyta do:

- implementacji API GraphQL,
- realizacji zapytań o złożone struktury danych,
- testów wydajnościowych komunikacji GraphQL.

## 2.15 Apache Avro

**Apache** Avro jest binarnym formatem serializacji danych opartym na schematach, wykorzystywanym głównie w systemach przetwarzania danych i architekturach rozproszonych. Zapewnia kompaktową reprezentację danych oraz możliwość ewolucji schematów.

W projekcie Apache Avro został wykorzystany do:

- testów wydajności serializacji i deserializacji danych,
- porównania z innymi formatami binarnymi i tekstowymi,
- analizy rozmiaru wynikowych struktur danych.

## 2.16 BSON

**BSON** (Binary JSON) jest binarną reprezentacją formatu JSON, zaprojektowaną z myślą o efektywnej serializacji danych. Jest powszechnie stosowany m.in. w bazach danych MongoDB.

W projekcie format BSON został wykorzystany do:

- testów wydajności serializacji danych,
- porównania binarnych i tekstowych formatów wymiany danych,
- analizy narzutu pamięciowego.

## 2.17 Serde

**Serde** jest biblioteką w języku Rust służącą do serializacji i deserializacji struktur danych[3]. Zapewnia zunifikowany interfejs obsługujący wiele formatów, takich jak JSON, BSON, XML czy MessagePack.

W projekcie `Serde` została wykorzystana jako:

- warstwa abstrakcji dla procesów serializacji danych,
- wspólna baza dla testów różnych formatów danych,
- narzędzie upraszczające implementację testów porównawczych.

## 2.18 Prost

**Prost** jest biblioteką w języku Rust służącą do obsługi formatu Protocol Buffers. Umożliwia generowanie kodu Rust na podstawie plików `.proto` oraz efektywną serializację danych binarnych[2].

W projekcie `Prost` została wykorzystana do:

- obsługi komunikacji gRPC,
- serializacji danych w formacie Protocol Buffers,
- testów wydajności komunikacji RPC.

## 2.19 quick-xml

**quick-xml** jest szybką biblioteką w języku Rust przeznaczoną do parsowania i generowania dokumentów XML. Charakteryzuje się niskim narzutem pamięciowym oraz wysoką wydajnością[8].

W projekcie biblioteka `quick-xml` została użyta do:

- testów serializacji i deserializacji danych w formacie XML,
- porównania XML z innymi formatami wymiany danych,
- analizy wydajności przetwarzania danych tekstowych.

# Rozdział 3

# Użyte narzędzia programistyczne

## 3.1 Git

**Git** jest rozproszonym systemem kontroli wersji, którego głównym celem jest zarządzanie historią zmian w kodzie źródłowym projektu programistycznego[4]. Narzędzie to umożliwia rejestrowanie kolejnych wersji plików, analizę wprowadzonych modyfikacji oraz powrót do wcześniejszych stanów projektu. Dzięki rozproszonej architekturze każdy użytkownik posiada pełną kopię repozytorium wraz z całą historią zmian, co zwiększa niezawodność oraz elastyczność pracy zespołowej.

Git oferuje mechanizmy takie jak gałęzie (ang. *branches*) oraz scalanie zmian (ang. *merge*), które pozwalają na równoległą pracę nad różnymi funkcjonalnościami bez ingerencji w główną wersję projektu. System ten jest szeroko stosowany zarówno w małych projektach indywidualnych, jak i w dużych przedsięwzięciach komercyjnych oraz otwartoźródłowych.

## 3.2 Zed

**Zed** jest nowoczesnym edytorem kodu źródłowego zaprojektowanym z myślą o wysokiej wydajności, niskich opóźnieniach oraz współczesnych potrzebach programistów[50]. Narzędzie to zostało stworzone przy wykorzystaniu języka `Rust`, co przekłada się na bezpieczeństwo pamięci oraz wysoką responsywność interfejsu użytkownika.

Edytor oferuje wsparcie dla wielu języków programowania, inteligentne podpowiedzi kodu, integrację z systemami kontroli wersji oraz możliwość pracy zespołowej w czasie rzeczywistym. Zed kładzie duży nacisk na minimalizm interfejsu oraz płynność pracy, co czyni go atrakcyjną alternatywą dla bardziej rozbudowanych środowisk programistycznych.

## 3.3 Visual Studio Code

**Visual Studio Code** jest wieloplatformowym edytorem kodu źródłowego rozwijanym przez firmę Microsoft[12]. Narzędzie to łączy w sobie prostotę edytora tekstu z funkcjonalnością zintegrowanego środowiska programistycznego (`IDE`). Dzięki rozbudowanemu systemowi rozszerzeń możliwe jest dostosowanie edytora do pracy z niemal dowolnym językiem programowania oraz frameworkiem.

Visual Studio Code oferuje funkcje takie jak podświetlanie składni, inteligentne uzupełnianie kodu, debugowanie aplikacji oraz integrację z systemem `Git`. Jego popularność wynika z

dużej elastyczności, aktywnej społeczności oraz regularnych aktualizacji, co czyni go jednym z najczęściej wybieranych narzędzi programistycznych.

## 3.4 Bun

**Bun** jest nowoczesnym środowiskiem uruchomieniowym dla języka `JavaScript`, `TypeScript` oraz `WebAssembly`, pełniącym jednocześnie rolę menedżera pakietów, narzędzia do budowania aplikacji oraz serwera uruchomieniowego[1]. Jego głównym celem jest zwiększenie wydajności i uproszczenie procesu tworzenia aplikacji webowych poprzez integrację funkcjonalności, które w tradycyjnym ekosystemie `Node.js`[14] realizowane są przez wiele odrębnych narzędzi.

Jednym z kluczowych elementów środowiska Bun jest wbudowany menedżer pakietów, stanowiący alternatywę dla NPM. Umożliwia on instalowanie, aktualizowanie oraz usuwanie zależności projektowych, a także korzystanie z pakietów dostępnych w rejestrze NPM, przy jednoczesnym zachowaniu znacznie krótszych czasów instalacji. Konfiguracja projektu opiera się na pliku `package.json`, analogicznie do standardowego ekosystemu JavaScript, natomiast dodatkowym elementem jest plik `bun.lock`, zapewniający deterministyczność i powtarzalność procesu budowania aplikacji.

Bun obsługuje również natywne uruchamianie kodu `TypeScript` bez konieczności wcześniejszej transpilacji oraz oferuje wbudowany system budowania i bundlowania zasobów. Dzięki temu środowisko to może pełnić rolę kompleksowej platformy deweloperskiej, szczególnie przydatnej w aplikacjach frontendowych, backendowych oraz projektach wykorzystujących technologię `WebAssembly`.

## 3.5 Cargo

**Cargo** jest oficjalnym narzędziem do zarządzania projektami w języku `Rust`[18]. Odpowiada ono za proces kompilacji kodu źródłowego, pobieranie i zarządzanie zależnościami oraz uruchamianie testów jednostkowych. Cargo integruje się bezpośrednio z kompilatorem Rust, zapewniając spójność i automatyzację procesu budowania aplikacji.

Konfiguracja projektu realizowana jest za pomocą pliku `Cargo.toml`, który zawiera informacje o projekcie, wersjach bibliotek oraz ustawieniach kompilacji. Dzięki Cargo proces tworzenia aplikacji w języku Rust jest uproszczony i ustandaryzowany, co sprzyja utrzymaniu wysokiej jakości kodu oraz jego skalowalności.

# Rozdział 4

# Budowa środowiska testowego

W niniejszym rozdziale przedstawiono kluczowe elementy środowiska testowego. W pierwszej kolejności zaprezentowano szczegółową strukturę katalogów projektu wraz z opisem zawartości poszczególnych folderów i plików. Następnie omówiono działanie wybranych aplikacji testowych, których implementację oraz wyniki pomiarów przedstawiono w kolejnym rozdziale.

## 4.1 Struktura plików aplikacji

Architektura przygotowanego środowiska testowego została zobrazowana na Rysunku 4.1 (str. 19). Struktura projektu obejmuje pięć głównych katalogów, z których każdy jest dedykowany do realizacji odrębnego zakresu badań. Architektura projektowa aplikacji została zaprojektowana w oparciu o koncepcje systemów rozproszonych i organizacji aplikacji opisane w publikacji Martina Kleppmanna[10].

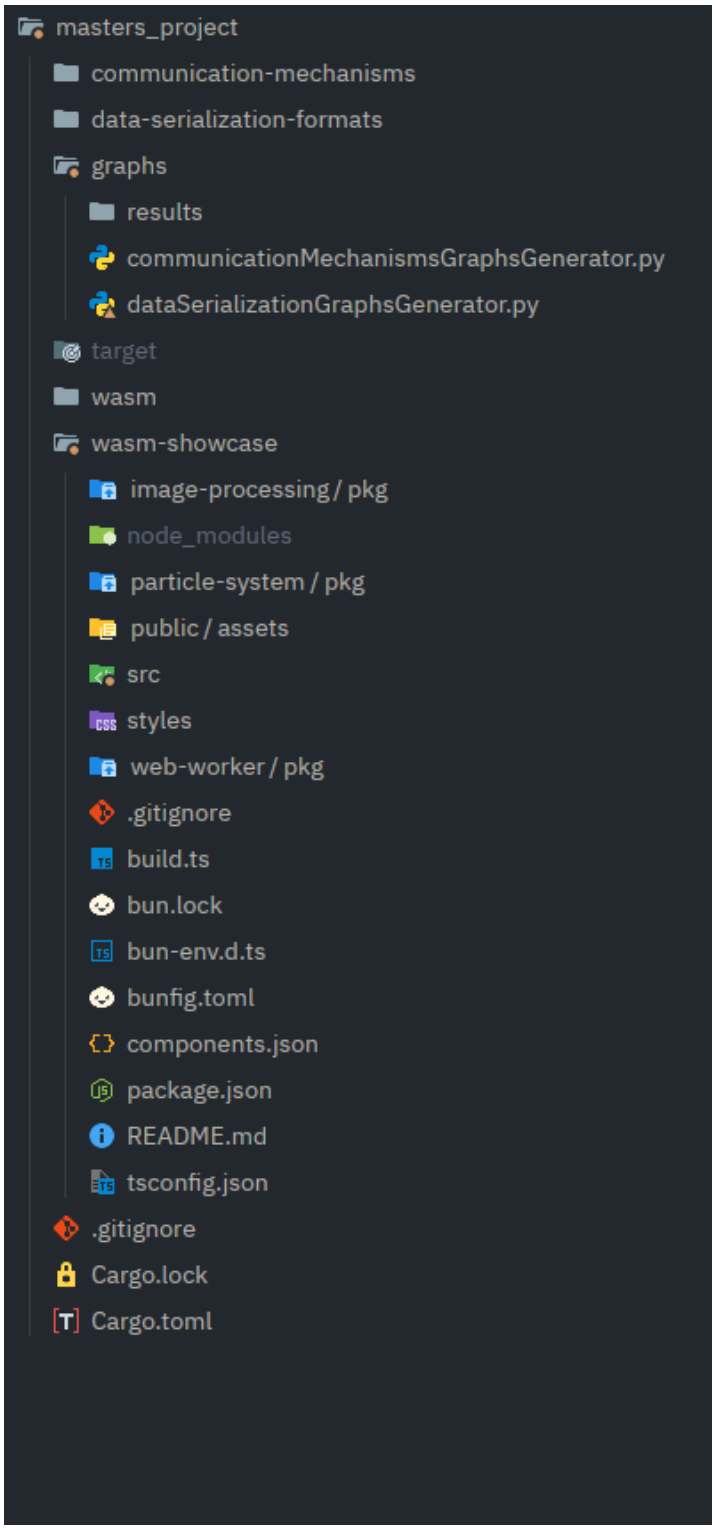
### 4.1.1 Katalog communication-mechanisms

Katalog ten zawiera implementacje aplikacji testujących wydajność różnych protokołów komunikacji. Każda aplikacja mierzy czasy odpowiedzi oraz przepustowość dla konkretnego mechanizmu wymiany danych. Wspólna struktura podkatalogów dla wszystkich implementacji obejmuje:

- **src** – katalog zawierający kod źródłowy aplikacji serwerowej i klienckiej dla danego protokołu,
- **cert.pem** – lokalny certyfikat SSL/TLS niezbędny do nawiązania bezpiecznego połączenia z wykorzystaniem protokołu HTTP/2,
- **key.pem** – prywatny klucz kryptograficzny odpowiadający certyfikatowi, wymagany do obsługi szyfrowanej komunikacji,
- **Cargo.toml** – manifest pakietu w formacie TOML, zawierający metadane projektu, listę zależności oraz parametry kompilacji niezbędne do zbudowania aplikacji w języku Rust.

W katalogu communication-mechanisms znajdują się następujące implementacje protokołów:

- **graphql** – implementacja serwera GraphQL umożliwiającego elastyczne zapytania o dane ze zdefiniowanego schematu,




Rysunek 4.1: Struktura plików i katalogów w aplikacji.

- **grpc** – implementacja serwera gRPC opartego na Protocol Buffers, wykorzystującego HTTP/2 do efektywnej komunikacji,

- **REST** – standardowa implementacja architektury REST API z wykorzystaniem metod HTTP (GET, POST, PUT, DELETE),

- **soap** – implementacja protokołu SOAP (Simple Object Access Protocol) opartego na wymianie komunikatów XML,
- **websocket** – implementacja serwera WebSocket umożliwiającego dwukierunkową komunikację w czasie rzeczywistym,
- **websocket_secure** – implementacja protokołu WebSocket Secure (WSS) z szyfrowaniem SSL/TLS.

### 4.1.2 Katalog data-serialization-formats

Katalog zawiera aplikacje służące do porównania wydajności procesów serializacji i deserializacji danych w różnych formatach. Każda implementacja mierzy czas potrzebny na przekształcenie struktur danych oraz rozmiar wynikowych reprezentacji. W katalogu znajdują się następujące moduły testowe:

- **avro** – implementacja testów dla formatu Apache Avro, binarnego systemu serializacji z wbudowanym wsparciem dla schematów danych,
- **bson_benchmark** – testy wydajności formatu BSON (Binary JSON), wykorzystywanego między innymi w bazach danych MongoDB,
- **json** – implementacja testów dla formatu JSON (JavaScript Object Notation), popularnego tekstowego formatu wymiany danych,
- **message-pack** – testy formatu MessagePack, będącego binarną alternatywą dla JSON o mniejszym rozmiarze danych,
- **protobuf** – implementacja testów dla Protocol Buffers (Protobuf), binarnego formatu serializacji opracowanego przez Google,
- **xml** – testy wydajności formatu XML (eXtensible Markup Language), rozszerzalnego języka znaczników.

### 4.1.3 Katalog graphs

Katalog zawiera narzędzia do wizualizacji wyników przeprowadzonych eksperymentów oraz wygenerowane wykresy. Struktura obejmuje:

- **communicationMechanismsGraphsGenerator.py** – skrypt w języku Python generujący wykresy porównawcze dla testów mechanizmów komunikacji, wizualizujący czasy odpowiedzi oraz przepustowość poszczególnych protokołów,
- **dataSerializationGraphsGenerator.py** – skrypt w języku Python tworzący wykresy przedstawiające wyniki testów serializacji danych, w tym czasy operacji oraz rozmiary zserializowanych struktur,
- **results** – katalog przechowujący wygenerowane wykresy w formatach graficznych (PNG, SVG) gotowe do wykorzystania w dokumentacji.

### 4.1.4 Katalog target

Katalog target stanowi standardowy folder roboczy kompilatora Rust, zawierający pliki binarne, biblioteki oraz artefakty pośrednie powstałe w procesie budowania projektu. Zawartość tego katalogu jest automatycznie zarządzana przez narzędzie Cargo i nie wymaga ręcznej modyfikacji.

### 4.1.5 Katalog wasm

Katalog zawiera programy napisane w języku `Rust`, które są kompilowane do formatu `WebAssembly` (WASM). Powstałe moduły binarne są następnie wykorzystywane w środowisku przeglądarki internetowej, umożliwiając uruchomienie kodu o wysokiej wydajności po stronie klienta.

latex

### 4.1.6 Katalog wasm-showcase

Katalog `wasm-showcase` reprezentuje w pełni funkcjonalną aplikację webową o charakterze demonstracyjnym, ilustrującą możliwości oraz zalety technologii `WebAssembly`. Fundamenty architektoniczne oraz implementowane wzorce programistyczne czerpią z koncepcji zaprezentowanych w publikacji[17]. Organizacja projektu opiera się na następujących elementach składowych:

- **image-processing** – katalog zawierający skompilowane do WebAssembly moduły służące do przetwarzania obrazów. Pliki WASM oraz pomocnicze skrypty JavaScript pozwalają na wykonywanie operacji graficznych bezpośrednio w przeglądarce,
- **particle-system** – katalog z modułami WASM implementującymi system cząsteczek, wykorzystywany do demonstracji wydajności obliczeń graficznych w środowisku webowym,
- **web-worker** – katalog zawierający skompilowane moduły WASM przeznaczone do uruchomienia w Web Workers, umożliwiające wielowątkowe przetwarzanie danych bez blokowania głównego wątku interfejsu użytkownika,
- **styles** – katalog ze stylami CSS definiującymi wygląd aplikacji webowej,
- **build.ts** – natywny skrypt TypeScript odpowiedzialny za proces budowania aplikacji, zawierający logikę kompilacji i optymalizacji zasobów,
- **bun.lock** – plik blokady rejestrujący dokładne wersje wszystkich zależności projektu oraz ich podzależności, zapewniający powtarzalność budowania aplikacji,
- **bunfig.toml** – plik konfiguracyjny środowiska uruchomieniowego Bun, zawierający ustawienia dotyczące kompilacji i wykonania projektu,
- **components.json** – plik konfiguracyjny definiujący strukturę importów komponentów oraz integrację z biblioteką Tailwind CSS,
- **package.json** – manifest projektu Node.js zawierający listę zależności npm wraz z określonymi wersjami, skrypty budowania oraz metadane aplikacji,

- **README.md** – plik dokumentacji zawierający szczegółowy opis procesu instalacji, konfiguracji oraz uruchomienia aplikacji,
- **tsconfig.json** – plik konfiguracyjny kompilatora TypeScript, definiujący opcje transpilacji, ścieżki modułów oraz poziom zgodności ze standardem ECMAScript,
- **bun-env.d.ts** – plik deklaracji typów TypeScript dla środowiska Bun, zapewniający wsparcie IntelliSense i kontrolę typów,
- **src** – katalog zawierający pliki źródłowe aplikacji, w tym komponenty React, logikę biznesową oraz pomocnicze moduły,
- **public** – katalog z zasobami statycznymi (obrazy, ikony, manifesty) udostępnianymi bezpośrednio przez serwer webowy,
- **node_modules** – katalog zawierający zainstalowane biblioteki i moduły npm, automatycznie zarządzany przez menedżer pakietów.

### 4.1.7 Plik .gitignore

Plik konfiguracyjny systemu kontroli wersji `Git`, zawierający listę plików i katalogów wykluczonych z repozytorium. Typowo obejmuje katalogi tymczasowe (target, node_modules), pliki konfiguracyjne środowiska oraz artefakty kompilacji, które nie powinny być śledzone w historii zmian projektu.

## 4.2 Testy wydajnościowe

W ramach przygotowanego środowiska testowego przeprowadzono szereg eksperymentów mających na celu obiektywną ocenę wydajności różnych mechanizmów komunikacji oraz formatów serializacji danych. Poniżej przedstawiono szczegółowy opis metodologii testowania poszczególnych grup aplikacji.

### 4.2.1 Aplikacje zawarte w katalogu communication-mechanisms

Dla każdego zaimplementowanego protokołu komunikacji przeprowadzono cztery kategorie testów wydajnościowych, pozwalających na kompleksową ocenę charakterystyk przesyłania danych:

1. **Transmisja małych pakietów danych** – test polegający na cyklicznym odpytywaniu serwera o dane o rozmiarze `1 KB`. Pomiar ten pozwala ocenić narzut protokołu oraz czas odpowiedzi dla typowych żądań zawierających niewielką ilość informacji.
2. **Transmisja dużych pakietów danych** – test analogiczny do poprzedniego, z tą różnicą, że rozmiar przesyłanych danych wynosi `1 MB`. Eksperyment ten umożliwia ocenę wydajności protokołów w scenariuszu transferu większych zasobów.
3. **Head-of-Line Blocking** – test weryfikujący występowanie zjawiska blokowania głowy kolejki, w którym opóźnienie w przetwarzaniu jednego żądania wpływa na obsługę kolejnych żądań w ramach tego samego połączenia. Testom został poodany tylko protokół `HTTP/1.1` oraz `HTTP/2`.

4. **Strumieniowanie danych** – test przeprowadzany wyłącznie dla protokołów wspierających tryb strumieniowy (streaming). Ocenie podlega efektywność przesyłania danych w sposób ciągły, bez konieczności oczekiwania na kompletną odpowiedź serwera. Testom został poodany tylko protokół `HTTP/1.1` oraz `HTTP/2`.

### 4.2.2 Aplikacje zawarte w katalogu data-serialization-formats

Dla każdej z zaimplementowanych bibliotek serializacji danych przeprowadzono pomiary wydajności na podstawie ustandaryzowanej struktury danych. W celu zapewnienia porównywalności wyników, wszystkie testy wykorzystują identyczny zestaw danych testowych składający się z kolekcji 1000 obiektów użytkowników wraz z metadanymi.

Struktura danych testowych jest generowana w sposób pokazany na listunku 4.1:

```
    let count = 1000;
    let users: Vec<User> = (0..count)
    .map(|i| User {
        id: i as i64,
        name: format!("User {}", i),
        email: format!("user{}@example.com", i),
        age: 20 + (i % 50) as i32,
        is_active: i % 2 == 0,
        tags: vec![
        "tag1".to_string(),
        "tag2".to_string(),
        "tag3".to_string(),
        ],
    })
    .collect();

    UserCollection {
        users,
        metadata: Metadata {
            version: "1.0.0".to_string(),
            created_at: "2025-01-23T00:00:00Z".to_string(),
            total_count: count,
        },
    }
```

Listing 4.1: Struktura danych wykorzystana w testach serializacji.

Każdy obiekt użytkownika zawiera kompletny zestaw atrybutów: identyfikator liczbowy, nazwę, adres email, wiek, status aktywności oraz listę trzech tagów. Dodatkowo, kolekcja jest wzbogacona o metadane zawierające wersję struktury danych, znacznik czasu utworzenia oraz łączną liczbę elementów. Taka konstrukcja zapewnia reprezentatywny zestaw różnorodnych typów danych (liczby całkowite, ciągi znaków, wartości logiczne, kolekcje) występujących w rzeczywistych zastosowaniach.

Dla każdego formatu serializacji mierzono następujące parametry wydajnościowe:

1. **Czas serializacji** – czas potrzebny na przekształcenie struktury danych z reprezentacji obiektowej języka Rust do formatu serializowanego. Pomiar wyrażony w milisekundach (`ms`) i uśredniony na podstawie wielokrotnych iteracji.

2. **Czas deserializacji** – czas wymagany do odtworzenia struktury obiektowej z danych w formacie serializowanym. Pomiar wyrażony w milisekundach (`ms`) i uśredniony na podstawie wielokrotnych iteracji.

3. **Rozmiar zserializowanych danych** – całkowita wielkość danych po procesie serializacji, wyrażona w bajtach (`B`). Parametr ten pozwala ocenić efektywność kompresji oraz narzut protokołu dla różnych formatów wymiany danych.

Wykorzystanie identycznego zestawu danych testowych dla wszystkich formatów serializacji umożliwia obiektywne porównanie ich wydajności oraz charakterystyk w zakresie rozmiaru wynikowych struktur danych.

### 4.2.3 Aplikacje zawarte w katalogu wasm oraz wasm-showcase

Katalogi *wasm* oraz *wasm-showcase* obejmują zbiór aplikacji eksperymentalnych napisanych w języku `Rust`, które zostały skompilowane do formatu `WebAssembly` (WASM) z wykorzystaniem narzędzia *wasm-pack*. Celem tych aplikacji jest analiza wpływu wykorzystania technologii WebAssembly na wydajność wykonywania obliczeń po stronie klienta w środowisku przeglądarki internetowej.

W ramach przeprowadzonych eksperymentów porównywany jest czas wykonania wybranych operacji matematycznych realizowanych bezpośrednio w języku JavaScript oraz analogicznych operacji wykonywanych przez skompilowane moduły `WebAssembly`. Takie zestawienie umożliwia ocenę potencjalnych korzyści wynikających z wykorzystania kodu natywnego kompilowanego do WASM w kontekście obliczeń o zwiększonej złożoności.

Testy zostały przeprowadzone z wykorzystaniem dwóch interfejsów programistycznych dostępnych w środowisku webowym:

- **Canvas API** – wykorzystywanego do wykonywania obliczeń związanych z przetwarzaniem grafiki oraz renderowaniem dynamicznych scen, co pozwala ocenić wydajność obliczeń realizowanych w głównym wątku aplikacji,

- **Web Workers API** – umożliwiającego uruchamianie modułów WebAssembly w osobnych wątkach roboczych, co pozwala na analizę wpływu przetwarzania równoległego na czas wykonania obliczeń oraz responsywność interfejsu użytkownika.

Zastosowanie zarówno `Canvas API`, jak i `Web Workers` pozwala na kompleksową ocenę wydajności modułów WebAssembly w różnych scenariuszach użycia, obejmujących zarówno obciążenie głównego wątku aplikacji, jak i przetwarzanie asynchroniczne. Wyniki uzyskane w ramach tych testów stanowią podstawę do dalszej analizy porównawczej przedstawionej w kolejnym rozdziale pracy.

# Rozdział 5

# Przeprowadzanie testów

Niniejszy rozdział prezentuje metodologię przeprowadzonych badań eksperymentalnych oraz ich wyniki ilościowe.

## 5.1 Metodologia testowania protokołów komunikacyjnych

Wszystkie eksperymenty zostały przeprowadzone z wykorzystaniem ujednoliconych parametrów testowych, dostosowanych do specyfiki każdego badanego protokołu:

- Liczba współbieżnych operacji: `100`
- Liczba zapytań na wątek: `1000`
- Rozmiar standardowego ładunku danych: 1024 bajty (`1 KB`)
- Rozmiar rozszerzonego ładunku danych: 1 048 576 bajtów (`1 MB`)

## 5.2 Architektura REST API

Implementacja architektury `REST` została zrealizowana przy użyciu biblioteki Actix-web – jednej z najpopularniejszych wysokopoziomowych platform wspierających natywnie strumieniowanie danych oraz protokół `HTTP/2` w połączeniu z certyfikatami TLS. W ramach analizy zbadano również problem blokowania na początku kolejki (ang. *Head-of-Line blocking*, HOL) wraz z proponowanym rozwiązaniem.

### 5.2.1 Wyniki dla protokołu HTTP/1.1

Listing 5.1 przedstawia implementację punktów końcowych (ang. *endpoints*) obsługujących zapytania ze standardowym oraz rozszerzonym ładunkiem danych. Funkcja `test()` zwraca odpowiedź JSON zawierającą 1 KB danych (1024 znaki), natomiast `test_high_payload()` generuje ładunek o rozmiarze 1 MB (1 048 576 znaków). Obie implementacje wykorzystują asynchroniczny model przetwarzania oraz mechanizm serializacji `serde_json`.

```
#[get("/test")]
async fn test() -> HttpResponse {
    HttpResponse::Ok().json(serde_json::json!({
        "message": "x".repeat(1024)
    }))
```

```
}

#[get("/test_high_payload")]
async fn test_high_payload() -> HttpResponse {
    HttpResponse::Ok().json(serde_json::json!({
        "message": "x".repeat(1024 * 1024)
    }))
}
```

Listing 5.1: Implementacja zapytań dla ładunków 1KB oraz 1MB dla protokołu HTTP/1.1 oraz HTTP/2 dla architektury REST API

**Wyniki pomiarów dla standardowego ładunku (1 KB):** Przepustowość wyniosła 764 076 żądań na sekundę, co stanowi najwyższą wartość spośród wszystkich testowanych konfiguracji dla małych pakietów danych.

**Wyniki pomiarów dla rozszerzonego ładunku (1 MB):** Odnotowano przepustowość na poziomie 2 759 żądań na sekundę, co stanowi spadek o 99,6% w porównaniu ze standardowym ładunkiem. Ta znacząca degradacja wynika z narzutu związanego z nawiązywaniem nowego połączenia TCP dla każdego żądania.

### Transmisja strumieniowa

Listing 5.2 przedstawia implementację strumieniowania danych, gdzie odpowiedź jest wysyłana w postaci sekwencji 100 fragmentów (ang. *chunks*). Funkcja `stream_handler()` wykorzystuje mechanizm `stream::unfold` do generowania kolejnych porcji danych w sposób asynchroniczny, eliminując konieczność przygotowania całej odpowiedzi w pamięci przed rozpoczęciem transmisji.

```
#[get("/stream")]
async fn stream_handler() -> HttpResponse {
    let response_stream = stream::unfold(0, |count| async move {
        if count >= 100 {
            None
        } else {
            let chunk = format!("chunk {}\n", count);
            Some((Ok::<Bytes, actix_web::Error>(Bytes::from(chunk)),
                count + 1))
        }
    });

    HttpResponse::Ok()
    .content_type("text/plain")
    .streaming(response_stream)
}
```

Listing 5.2: Implementacja strumieniowania dla protokołu HTTP/1.1 oraz HTTP/2 dla architektury REST API

Transmisja strumieniowa składająca się ze 100 fragmentów osiągnęła przepustowość 239 056 żądań na sekundę, co stanowi wartość pośrednią między scenariuszami małych i dużych pakietów.

**Problem Head-of-Line Blocking**

Listing 5.3 przedstawia implementację testową symulującą problem HOL, w którym opóźnienie jednego żądania blokuje przetwarzanie kolejnych zapytań w ramach tego samego połączenia. Funkcja `hol()` wprowadza sztucznie opóźnienie 50 ms przed zwróceniem odpowiedzi, co pozwala zaobserwować wpływ zablokowanego żądania na całkowitą przepustowość systemu.

```
#[get("/hol")]
async fn hol() -> HttpResponse {
    sleep(Duration::from_millis(50)).await;
    HttpResponse::Ok()
    .content_type("text/plain")
    .body("ok")
}
```

Listing 5.3: Implementacja symulacji problemu HOL dla protokołu HTTP/1.1 oraz HTTP/2 dla architektury REST API

W warunkach występowania problemu Head-of-Line blocking zmierzono przepustowość wynoszącą 1 954 żądania na sekundę, demonstrując znaczącą degradację wydajności względem normalnego strumieniowania.

### 5.2.2 Wyniki dla protokołu HTTP/2

Protokół HTTP/2 wykorzystuje ten sam kod implementacyjny co HTTP/1.1. Jedyną różnicą jest dodanie warstwy TLS, która zapewnia bezpieczną komunikację poprzez szyfrowanie danych, integralność oraz uwierzytelnianie serwerów [47].

**Wyniki pomiarów – standardowy ładunek:** Implementacja z wykorzystaniem HTTP/2 osiągnęła przepustowość 198 272 żądań na sekundę dla standardowego ładunku – wartość o 74% niższą niż w przypadku HTTP/1.1. Degradacja ta wynika z narzutu związanego z negocjacją TLS oraz multipleksowaniem strumieni.

**Wyniki pomiarów – rozszerzony ładunek:** Dla dużych pakietów danych zaobserwowano przepustowość 1 690 żądań na sekundę, co stanowi spadek o 39% względem HTTP/1.1. Mimo niższej wartości bezwzględnej, proporcjonalnie mniejsza degradacja wskazuje na lepszą optymalizację HTTP/2 dla transferu dużych wolumenów danych.

**Wyniki pomiarów – transmisja strumieniowa:** Transmisja strumieniowa w protokole HTTP/2 uzyskała przepustowość 91 687 żądań na sekundę, wykazując degradację o 62% w stosunku do HTTP/1.1.

**Wyniki pomiarów – HOL blocking:** W scenariuszu testowym HOL blocking protokół HTTP/2 osiągnął przepustowość 135 372 żądań na sekundę, demonstrując 69-krotną poprawę względem HTTP/1.1. Ten wynik potwierdza efektywność mechanizmu multipleksowania strumieni w mitigacji problemu blokowania kolejki.

## 5.3 Architektura GraphQL

Implementację GraphQL zrealizowano przy użyciu biblioteki Juniper – zaawansowanego frameworka wspierającego natywnie strumieniowanie oraz HTTP/2 z wykorzystaniem TLS.

Listing 5.4 przedstawia implementację schematu GraphQL wraz z funkcjami resolver, które są wywoływane w celu uzyskania żądanych danych. Funkcja `test_payload()` zwraca obiekt zawierający 1 KB danych, natomiast `large_payload()` generuje ładunek o rozmiarze 1 MB, umożliwiając porównanie wydajności GraphQL dla różnych rozmiarów przesyłanych danych.

```rust
#[juniper::graphql_object]
impl Query {
    fn test_payload() -> Payload {
        Payload {
            message: "x".repeat(1024),
        }
    }

    fn large_payload() -> Payload {
        const PAYLOAD_SIZE: usize = 1024 * 1024;
        Payload {
            message: "x".repeat(PAYLOAD_SIZE),
        }
    }
}
```

Listing 5.4: Implementacja zapytania GraphQL

### 5.3.1 Wyniki dla protokołu HTTP/1.1

GraphQL w połączeniu z HTTP/1.1 osiągnął przepustowość 179 760 żądań na sekundę dla standardowego ładunku, co stanowi 76% wydajności REST API w analogicznej konfiguracji.

Dla rozszerzonego ładunku zmierzono przepustowość 2 418 żądań na sekundę, wykazując nieznaczną przewagę (12%) nad REST API. Lepsza wydajność dla dużych pakietów może wynikać z bardziej efektywnej serializacji JSON w bibliotece Juniper.

### 5.3.2 Wyniki dla protokołu HTTP/2

Konfiguracja z HTTP/2 osiągnęła przepustowość 195 877 żądań na sekundę, demonstrując 9% poprawę względem HTTP/1.1 oraz porównywalne wyniki z REST HTTP/2.

Dla dużych pakietów danych zaobserwowano przepustowość 2 097 żądań na sekundę, co stanowi 24% wzrost względem REST HTTP/2, potwierdzając optymalizacje GraphQL dla złożonych zapytań z dużymi zbiorami danych.

## 5.4 Architektura gRPC

Implementacja gRPC została zrealizowana przy użyciu biblioteki Tonic – wysoko wydajnego frameworka natywnie wspierającego HTTP/2, strumieniowanie danych oraz szyfrowanie TLS. Ze względu na optymalizację wydajności zrezygnowano z implementacji mechanizmu refleksji API. Protokół gRPC automatycznie wykorzystuje protokół HTTP/2.

### 5.4.1 Definicja protokołu

Listing 5.5 przedstawia zawartość pliku definicji protokołu (ang. *Protocol Buffers*), na podstawie którego generowany jest automatycznie kod oraz typy danych dla języka Rust.

```
syntax = "proto3";

package benchmark;
```

```
4
5  service BenchmarkService {
6      rpc TestPayload (TestPayloadRequest) returns (TestPayloadResponse);
7      rpc TestHighPayload (TestHighPayloadRequest) returns (
           TestHighPayloadResponse);
8  }
9
10 message TestPayloadRequest {}
11
12 message TestPayloadResponse {
13     string message = 1;
14 }
15
16 message TestHighPayloadRequest {}
17
18 message TestHighPayloadResponse {
19     string message = 1;
20 }
```

Listing 5.5: Zawartość pliku .proto, do którego tworzony jest automatycznie kod oraz typy

### 5.4.2 Wyniki dla protokołu HTTP/2

Listing 5.6 przedstawia implementację funkcji obsługujących zapytania gRPC. Funkcja `test_payload()` zwraca odpowiedź zawierającą 1 KB danych, natomiast `test_high_payload()` generuje ładunek o rozmiarze 1 MB. Typy żądań i odpowiedzi zostały automatycznie wygenerowane z definicji Protocol Buffers, wymagając jedynie implementacji logiki zwracającej dane.

```
1  async fn test_payload(
2  &self,
3  _request: Request<TestPayloadRequest>,
4  ) -> Result<Response<TestPayloadResponse>, Status> {
5      let payload = "x".repeat(1024);
6      Ok(Response::new(TestPayloadResponse {
7          message: payload,
8      }))
9  }
10
11 async fn test_high_payload(
12 &self,
13 _request: Request<TestHighPayloadRequest>,
14 ) -> Result<Response<TestHighPayloadResponse>, Status> {
15     let payload = "x".repeat(1024 * 1024);
16     Ok(Response::new(TestHighPayloadResponse {
17         message: payload,
18     }))
19 }
```

Listing 5.6: Funkcje wykorzystywane w testach gRPC

**Wyniki pomiarów – standardowy ładunek:** gRPC osiągnął przepustowość 168 060 żądań na sekundę dla standardowego ładunku, co stanowi 15% spadek względem GraphQL HTTP/2. Niższa wartość może wynikać z narzutu związanego z serializacją Protocol Buffers oraz dodatkowych nagłówków gRPC.

**Wyniki pomiarów – rozszerzony ładunek:** Dla rozszerzonego ładunku zmierzono przepustowość 2 243 żądań na sekundę, wykazując 7% przewagę nad GraphQL oraz 33% nad REST w konfiguracji HTTP/2. Wynik ten potwierdza optymalizację gRPC dla transferu dużych wolumenów danych binarnych.

## 5.5 Architektura WebSocket

Do implementacji architektury WebSocket wykorzystano bibliotekę `tungstenite-rs` – popularny framework wysokiego poziomu dedykowany komunikacji dwukierunkowej w czasie rzeczywistym.

Listing 5.7 przedstawia implementację logiki obsługi wiadomości w serwerze WebSocket. Kod znajduje się w głównej pętli obsługującej zdarzenia, gdzie na podstawie odebranej komendy tekstowej serwer zwraca odpowiedni typ odpowiedzi: standardowy ładunek (1 KB), rozszerzony ładunek (1 MB).

```
match text.as_str() {
    "test" => {
        let payload = "x".repeat(PAYLOAD_SIZE);
        if let Err(e) = websocket.send(Message::Text(payload.into())).
           await {
            eprintln!("Connection {} write error: {}", conn_id, e);
            break;
        }
    }
    "test_high_payload" => {
        let payload = "x".repeat(HIGH_PAYLOAD_SIZE);
        if let Err(e) = websocket.send(Message::Text(payload.into())).
           await {
            eprintln!("Connection {} write error: {}", conn_id, e);
            break;
        }
    }
    _ => {
        eprintln!("Connection {} unknown command: {}", conn_id, text);
    }
}
```

Listing 5.7: Implementacja obsługi wiadomości WebSocket

### 5.5.1 Wyniki dla protokołu WebSocket

**Wyniki pomiarów – standardowy ładunek:** Protokół WebSocket osiągnął przepustowość 122 431 żądań na sekundę, wykazując najniższą wartość spośród testowanych protokołów dla standardowego ładunku. Niższa wydajność wynika z narzutu związanego z utrzymywaniem trwałych połączeń oraz ramkowaniem wiadomości.

**Wyniki pomiarów – rozszerzony ładunek:** Dla dużych pakietów danych zaobserwowano przepustowość 3 187 żądań na sekundę – najwyższą wartość wśród wszystkich badanych protokołów, przewyższającą o 42% najbliższego konkurenta (gRPC). Wynik ten potwierdza optymalizację WebSocket dla transferu dużych wolumenów danych poprzez eliminację narzutu związanego z nawiązywaniem połączeń.

### 5.5.2 Wyniki dla protokołu WebSocket Secure (WSS)

Implementacja protokołu WebSocket Secure jest niemal identyczna z wersją niezabezpieczoną. Kluczową różnicą jest dodanie warstwy TLS, która zapewnia szyfrowanie komunikacji oraz weryfikację tożsamości serwera.

**Wyniki pomiarów – standardowy ładunek:** Wersja zabezpieczona (WSS) osiągnęła przepustowość 111 086 żądań na sekundę, co stanowi 9% degradację względem niezabezpieczonej implementacji. Spadek ten wynika z narzutu związanego z szyfrowaniem oraz weryfikacją certyfikatów.

**Wyniki pomiarów – rozszerzony ładunek:** Dla rozszerzonego ładunku zmierzono przepustowość 2 932 żądań na sekundę, wykazując 8% spadek względem wersji niezabezpieczonej. Proporcjonalnie mniejsza degradacja dla dużych pakietów wskazuje, że koszt TLS handshake jest amortyzowany w długotrwałych transferach.

## 5.6 Architektura SOAP

Implementacja protokołu SOAP została przetestowana w obu wersjach HTTP w celu porównania charakterystyk wydajnościowych. Implementacja opiera się na bibliotece Actix-web, identycznej jak w przypadku REST API.

Listing 5.8 przedstawia implementację punktów końcowych SOAP generujących odpowiedzi w formacie XML zgodnym ze specyfikacją SOAP 1.2. Funkcja `soap_handler()` zwraca komunikat SOAP zawierający 1 KB danych, natomiast `soap_large_payload_handler()` generuje ładunek o rozmiarze 1 MB. Obie funkcje konstruują pełną strukturę koperty SOAP (Envelope) z odpowiednią deklaracją przestrzeni nazw oraz zagnieżdżonym elementem Body zawierającym dane testowe.

```
#[post("/test")]
async fn soap_handler() -> HttpResponse {
    let payload = "x".repeat(1024);

    let response = format!(
    r#"
    <?xml version="1.0" encoding="utf-8"?>
    <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope
       /"
    xmlns:test="urn:benchmark">
    <soap:Body>
    <test:TestPayloadResponse>
    <test:message>{}</test:message>
    </test:TestPayloadResponse>
    </soap:Body>
    </soap:Envelope>
    "#,
```

```
    payload
    );

    HttpResponse::Ok()
    .content_type("text/xml; charset=utf-8")
    .body(response)
}

#[post("/test_large_payload")]
async fn soap_large_payload_handler() -> HttpResponse {
    let payload = "x".repeat(1024 * 1024);
    let response = format!(
    r#"
    <?xml version="1.0" encoding="utf-8"?>
    <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope
       /"
    xmlns:test="urn:benchmark">
    <soap:Body>
    <test:TestPayloadResponse>
    <test:message>{}</test:message>
    </test:TestPayloadResponse>
    </soap:Body>
    </soap:Envelope>
    "#,
    payload
    );

    HttpResponse::Ok()
    .content_type("text/xml; charset=utf-8")
    .body(response)
}
```

Listing 5.8: Implementacja endpointów SOAP

### 5.6.1 Wyniki dla protokołu HTTP/1.1

**Wyniki pomiarów – standardowy ładunek:** SOAP z HTTP/1.1 osiągnął przepustowość 198 232 żądań na sekundę, demonstrując porównywalne wyniki z REST HTTP/2. Nieoczekiwanie wysoka wydajność wynika z efektywnej implementacji generowania XML oraz braku konieczności parsowania złożonych struktur danych.

**Wyniki pomiarów – rozszerzony ładunek:** Dla rozszerzonego ładunku zmierzono przepustowość 2 892 żądań na sekundę – najwyższą wartość wśród protokołów wykorzystujących HTTP/1.1, przewyższającą REST o 4,8%.

### 5.6.2 Wyniki dla protokołu HTTP/2

Protokół SOAP z TLS automatycznie wykorzystuje HTTP/2, jeśli klient i serwer wspierają negocjację ALPN (Application-Layer Protocol Negotiation).

**Wyniki pomiarów – standardowy ładunek:** Implementacja z HTTP/2 osiągnęła przepustowość 213 973 żądań na sekundę, wykazując 8% poprawę względem HTTP/1.1 oraz naj-

wyższy wynik spośród wszystkich protokołów HTTP/2 dla małych pakietów.

**Wyniki pomiarów – rozszerzony ładunek:** Dla dużych pakietów danych zaobserwowano przepustowość 2 478 żądań na sekundę, co stanowi 14% spadek względem HTTP/1.1. Degradacja ta jest nietypowa i może wynikać z narzutu multipleksowania strumieni dla pojedynczych, dużych odpowiedzi XML.

## 5.7 Protokół MQTT

MQTT (Message Queuing Telemetry Transport) to lekki protokół publikacji/subskrypcji zaprojektowany dla środowisk z ograniczonymi zasobami oraz niestabilnymi połączeniami sieciowymi.

### 5.7.1 Implementacja serwera

Listing 5.9 przedstawia główną pętlę zdarzeń (ang. *event loop*) odbiorcy MQTT, która przetwarza przychodzące publikacje i wysyła odpowiedzi na dedykowane tematy. Pętla nasłuchuje na zdarzenia przy użyciu metody `poll()`, a po odebraniu wiadomości na temat `benchmark/request` lub `benchmark/high_payload` publikuje otrzymany ładunek na odpowiadający temat odpowiedzi (`benchmark/response` lub **benchmark/high_payload_response**). Komunikacja odbywa się z poziomem Quality of Service ustawionym na `AtLeastOnce`, zapewniając dostarczenie wiadomości co najmniej raz. Implementacja zawiera również podstawową obsługę błędów oraz logowanie liczby przetworzonych żądań co 1000 operacji.

```
loop {
    match eventloop.poll().await {
        Ok(Event::Incoming(Incoming::Publish(p))) => {
            request_count += 1;

            let response_topic = match p.topic.as_str() {
                "benchmark/request" => "benchmark/response",
                "benchmark/high_payload" => "benchmark/
                   high_payload_response",
                _ => continue,
            };

            if let Err(e) = client
            .publish(
            response_topic,
            QoS::AtLeastOnce,
            false,
            p.payload,
            )
            .await
            {
                eprintln!("Failed to publish response: {:?}", e);
            }

            if request_count % 1000 == 0 {
                println!("Processed {} requests", request_count);
```

```
            }
        }
        Ok(_) => {}
        Err(e) => {
            eprintln!("Server error: {:?}", e);
            tokio::time::sleep(Duration::from_secs(1)).await;
        }
    }
}
```

Listing 5.9: Główna pętla zdarzeń odbiorcy MQTT

### 5.7.2 Implementacja klienta

Listing 5.10 przedstawia logikę generowania ładunku danych w kliencie MQTT wykorzystywanym w testach wydajnościowych. W zależności od parametru konfiguracyjnego `high_payload()`, funkcja tworzy obiekt JSON zawierający identyfikator wątku roboczego oraz dane testowe o rozmiarze 1 MB (dla `high_payload = true`) lub o rozmiarze zdefiniowanym w konfiguracji (dla `high_payload = false`). Wygenerowana struktura jest następnie serializowana do formatu tekstowego JSON i przesyłana jako payload wiadomości MQTT.

```
let payload = if config.high_payload {
    serde_json::json!({
        "id": format!("worker_{}", worker_id),
        "payload": "x".repeat(1024 * 1024)
    })
    .to_string()
} else {
    serde_json::json!({
        "id": format!("worker_{}", worker_id),
        "payload": "x".repeat(config.payload_size)
    })
    .to_string()
};
```

Listing 5.10: Logika generowania ładunku danych w kliencie MQTT

### 5.7.3 Wyniki pomiarów

**Wyniki pomiarów – standardowy ładunek:** Protokół MQTT osiągnął przepustowość 112 126 żądań na sekundę dla standardowego ładunku, co lokuje go w dolnej części spektrum wydajnościowego dla małych pakietów.

**Wyniki pomiarów – rozszerzony ładunek:** Dla rozszerzonego ładunku zmierzono przepustowość 2 167 żądań na sekundę, lokując MQTT w środkowym przedziale wydajnościowym. Wynik ten potwierdza, że protokół jest zoptymalizowany pod kątem niezawodności i niskiego zużycia zasobów, nie zaś maksymalnej przepustowości.

## 5.8 Analiza porównawcza protokołów komunikacyjnych

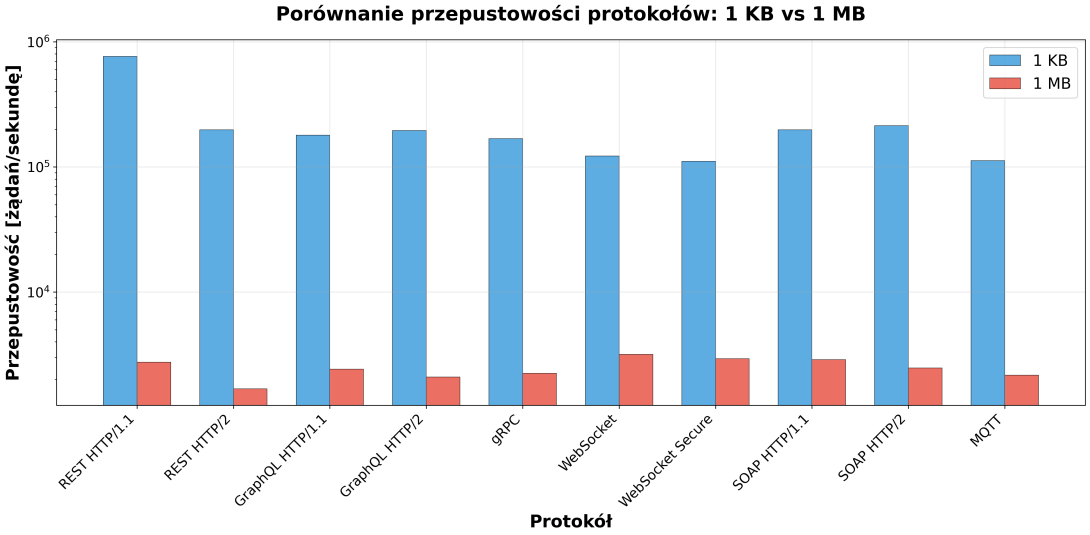
Przeprowadzone badania umożliwiły kompleksową ocenę wydajności protokołów komunikacyjnych w różnych scenariuszach użycia. Poniższa analiza syntetyzuje kluczowe wnioski z testów.

### 5.8.1 Wpływ rozmiaru ładunku na wydajność

Tabela 5.1: Porównanie przepustowości protokołów komunikacyjnych

| Protokół | 1 KB [req/s] | 1 MB [req/s] | Degradacja |
|---|---|---|---|
| REST HTTP/1.1 | 764 076 | 2 759 | 276,9× |
| REST HTTP/2 | 198 272 | 1 690 | 117,3× |
| GraphQL HTTP/1.1 | 179 760 | 2 418 | 74,3× |
| GraphQL HTTP/2 | 195 877 | 2 097 | 93,4× |
| gRPC | 168 060 | 2 243 | 74,9× |
| WebSocket | 122 431 | 3 187 | 38,4× |
| WebSocket Secure | 111 086 | 2 932 | 37,9× |
| SOAP HTTP/1.1 | 198 232 | 2 892 | 68,5× |
| SOAP HTTP/2 | 213 973 | 2 478 | 86,3× |
| MQTT | 112 126 | 2 167 | 51,7× |

Analiza współczynnika degradacji (tabela 5.1) ujawnia fundamentalne różnice w charakterystykach wydajnościowych protokołów. WebSocket Secure wykazał najniższą degradację (37,9×), co potwierdza jego optymalizację dla transmisji dużych wolumenów danych poprzez utrzymywanie stałego połączenia. W przeciwieństwie do tego, REST HTTP/1.1 odnotował najwyższą degradację (276,9×), co jest konsekwencją narzutu związanego z nawiązywaniem nowego połączenia TCP dla każdego żądania oraz brakiem multipleksowania.




Rysunek 5.1: Porównanie przepustowości protokołów dla ładunku 1 KB i 1 MB (skala logarytmiczna).

Wykres 5.1 ilustruje wyraźną dychotomię między wydajnością dla małych i dużych ładunków. Protokoły oparte na HTTP/1.1 dominują w scenariuszach małych pakietów, podczas gdy

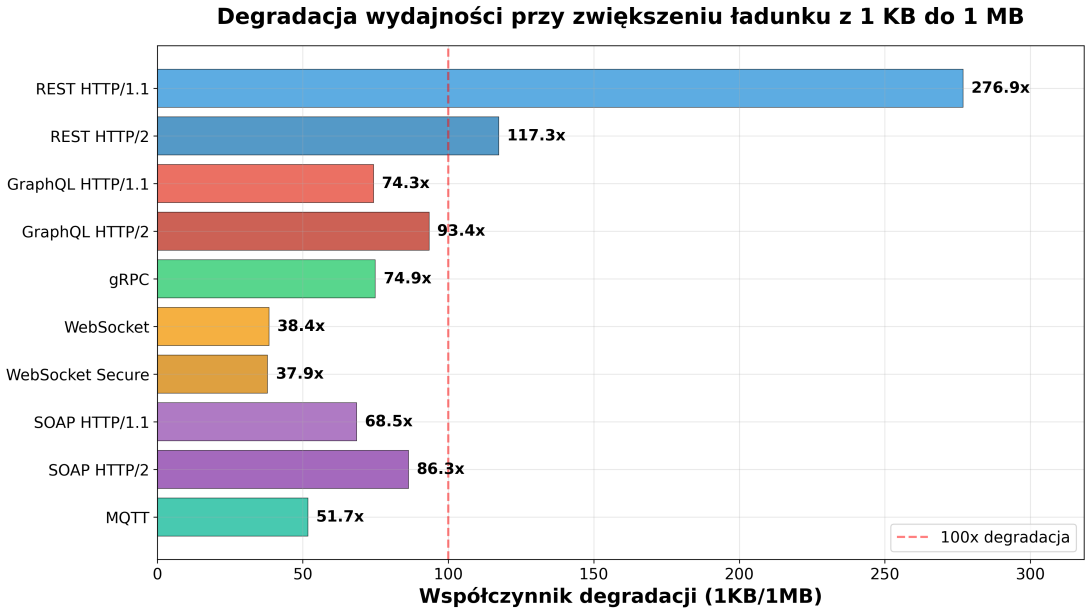
protokoły z trwałymi połączeniami (WebSocket, WSS) wykazują przewagę dla dużych transferów danych.

### 5.8.2 Wpływ TLS na wydajność

Bezpieczeństwo transmisji za pomocą TLS wprowadza dodatkowy narzut wydajnościowy, jednak jest niezbędne w aplikacjach produkcyjnych. Kluczowe aspekty TLS handshake:

- **Czas inicjalizacji:** TLS handshake typowo dodaje 50–100 ms opóźnienia dla pierwszego połączenia
- **Narzut obliczeniowy:** Szyfrowanie symetryczne (AES) ma minimalny wpływ na przepustowość (<5%)
- **Optymalizacja:** HTTP/2 i WebSocket amortyzują koszt TLS poprzez długotrwałe połączenia
- **Session resumption:** TLS 1.3 redukuje czas ponownego handshake'a do ∼1 RTT

Porównanie WebSocket (122 431 req/s) vs WebSocket Secure (111 086 req/s) pokazuje 9% degradację – znacząco niższą niż REST HTTP/1.1 vs HTTP/2 (74%), co wskazuje, że jednorazowy koszt TLS handshake jest amortyzowany w długotrwałych połączeniach.




Rysunek 5.2: Współczynnik degradacji wydajności przy zwiększeniu ładunku z 1 KB do 1 MB.

### 5.8.3 Rekomendacje wyboru protokołu

Na podstawie przeprowadzonych badań, dobór protokołu powinien uwzględniać specyficzne wymagania aplikacji:

1. **REST API (HTTP/1.1 lub HTTP/2)**
   - Zastosowanie: Aplikacje webowe, API publiczne, architektury mikroserwisowe

- Zalety: Najlepsza skalowalność, cache'owalność, powszechne wsparcie
- Wybór HTTP/2: Gdy priorytetem jest mitigacja HOL blocking i bezpieczeństwo

2. **GraphQL**
   - Zastosowanie: Frontend-heavy aplikacje, złożone modele danych
   - Zalety: Eliminacja over-fetchingu, redukcja liczby zapytań
   - Kompromis: ∼1–2% niższa wydajność niż REST przy znaczącej poprawie developer experience
3. **gRPC**
   - Zastosowanie: Komunikacja service-to-service, high-performance backends
   - Zalety: Najlepsza wydajność dla dużych ładunków w HTTP/2, silne kontrakty
   - Ograniczenie: Słabsze wsparcie w przeglądarkach (wymaga gRPC-web)
4. **WebSocket/WSS**
   - Zastosowanie: Aplikacje real-time (czaty, giełdy, gaming)
   - Zalety: Najniższa latencja, najlepsza wydajność dla dużych ładunków
   - Kompromis: Zwiększona złożoność skalowania (sticky sessions)
5. **SOAP**
   - Zastosowanie: Systemy enterprise, integracje legacy, transakcje finansowe
   - Zalety: Formalne standardy (WS-Security, WS-Transaction), compliance
   - Ograniczenie: Narzut XML, większa złożoność implementacji
6. **MQTT**
   - Zastosowanie: IoT, urządzenia wbudowane, sieci z ograniczonym pasmem
   - Zalety: Minimalny narzut protokołu, kolejkowanie, QoS levels
   - Optymalne: Dla rozproszonych sieci sensorów z intermittent connectivity

### 5.8.4 Kluczowe wnioski

Przeprowadzone badania prowadzą do następujących konkluzji:

1. **Brak uniwersalnego lidera:** Każdy protokół wykazuje przewagi w specyficznych scenariuszach użycia. REST HTTP/1.1 dominuje dla małych zapytań, WebSocket dla dużych transferów, gRPC dla komunikacji backend-to-backend.
2. **TLS jest akceptowalnym kosztem:** Degradacja wydajności 5–15% (w zależności od protokołu) jest w pełni uzasadniona przez krytyczne wymagania bezpieczeństwa. Nowoczesne implementacje TLS 1.3 dodatkowo minimalizują ten narzut.
3. **Długotrwałe połączenia amortyzują koszty:** Protokoły z persistent connections (WebSocket, HTTP/2) wykazują niższą degradację przy wzroście ładunku, co czyni je preferowanymi dla high-throughput scenarios.

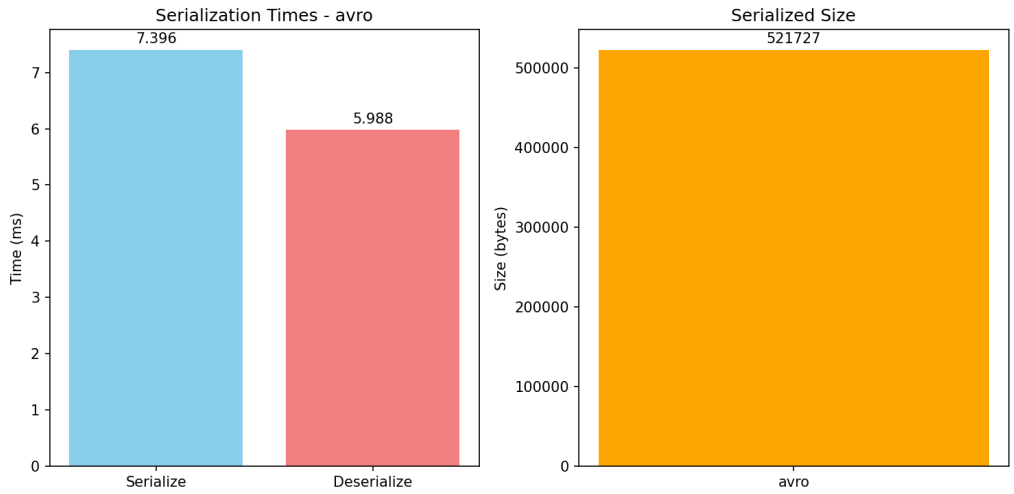
4. **Multipleksowanie rozwiązuje HOL blocking:** HTTP/2 osiągnęło 69-krotną poprawę w testach HOL, potwierdzając efektywność multipleksowania strumieni.

5. **Specjalizacja protokołów IoT:** MQTT, mimo umiarkowanej przepustowości, pozostaje optymalnym wyborem dla IoT dzięki mechanizmom QoS i niskiej konsumpcji zasobów.

## 5.9 Analiza porównawcza formatów serializacji danych

Przeprowadzono testy wydajnościowe dla sześciu popularnych formatów serializacji, mierząc czas serializacji, deserializacji oraz rozmiar wynikowych danych. Testowy zbiór składał się z 1000 obiektów użytkownika wraz z metadanymi.

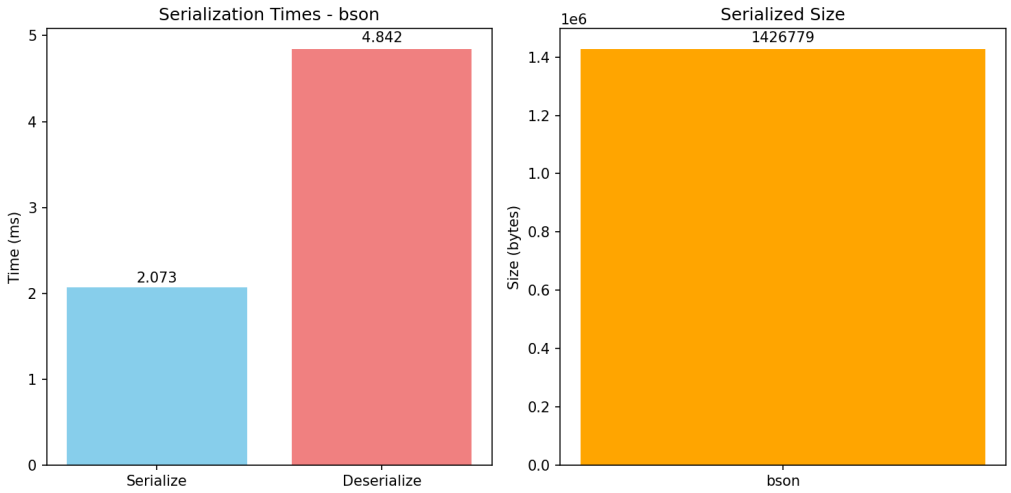
### 5.9.1 Apache Avro




Rysunek 5.3: Metryki wydajności Apache Avro.

Analiza metryk przedstawionych na Rysunku 5.3 wskazuje, że Apache Avro osiągnął czas serializacji na poziomie 7,396 ms oraz deserializacji wynoszący 5,988 ms, generując przy tym dane o rozmiarze 521 727 bajtów (509,5 KB).

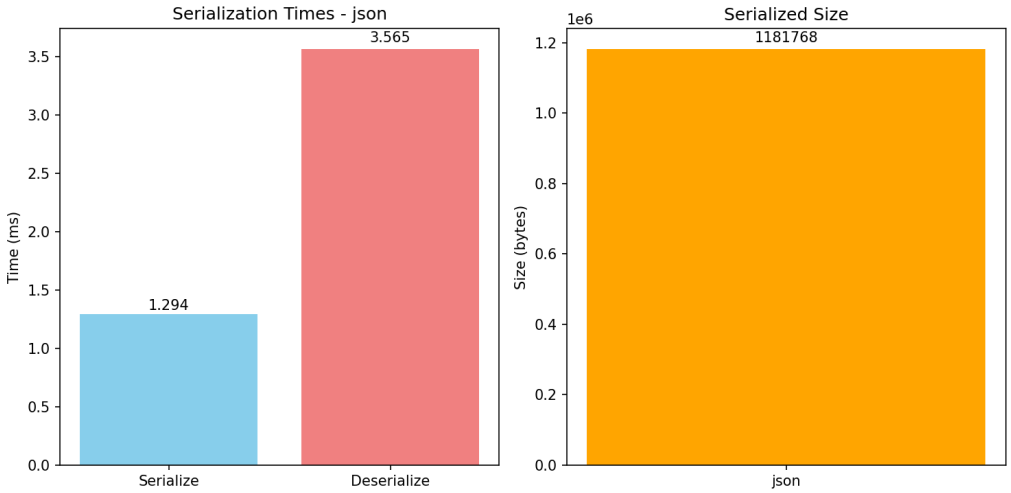
### 5.9.2 Binary JSON (BSON)




Rysunek 5.4: Metryki wydajności Binary JSON.

Analiza metryk przedstawionych na Rysunku 5.4 wskazuje, że Format BSON wykazał czas serializacji 2,073 ms, deserializacji 4,842 ms, generując dane o rozmiarze 1 426 779 bajtów (1 393,3 KB) – największym spośród testowanych formatów.

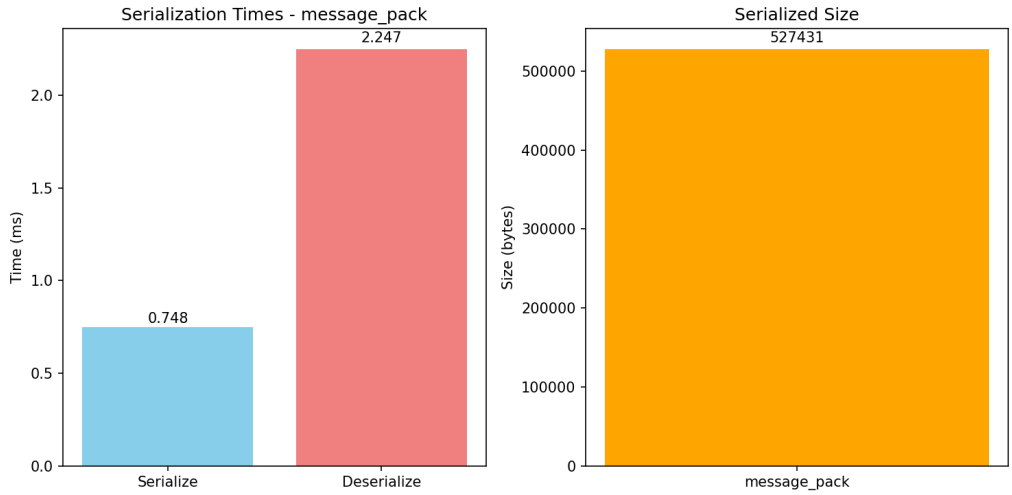
### 5.9.3 JSON




Rysunek 5.5: Metryki wydajności JSON.

Metryki zawarte na Rysunku 5.5 wykazują, że format JSON osiągnął czas serializacji 1,294 ms oraz deserializacji 3,565 ms przy rozmiarze 1 181 768 bajtów (1 154,1 KB), oferując najszybszą deserializację wśród formatów tekstowych.

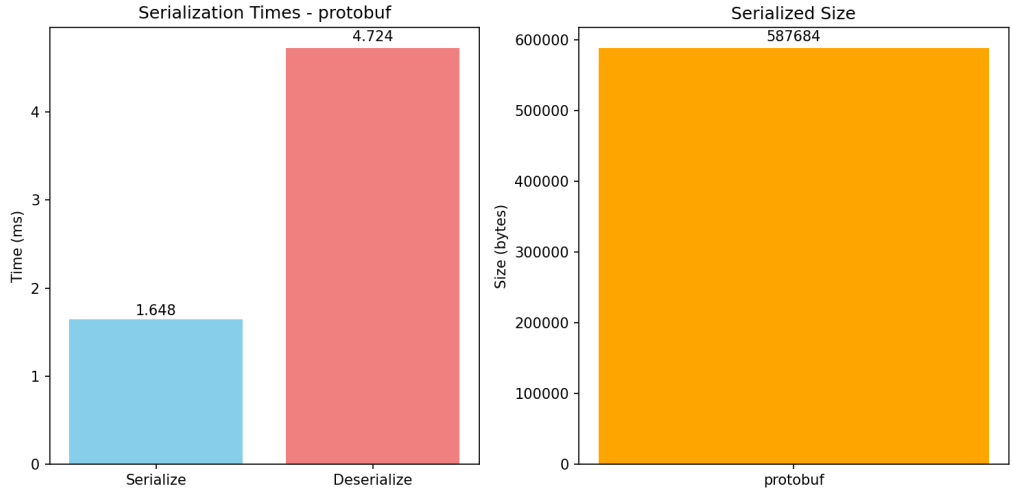
### 5.9.4 MessagePack




Rysunek 5.6: Metryki wydajności MessagePack.

Metryki zawarte na Rysunku 5.6 wykazują, że format MessagePack osiągnął najlepszą wydajność czasową z serializacją 0,748 ms i deserializacją 2,247 ms, przy kompaktowym rozmiarze 527 431 bajtów (515,1 KB).

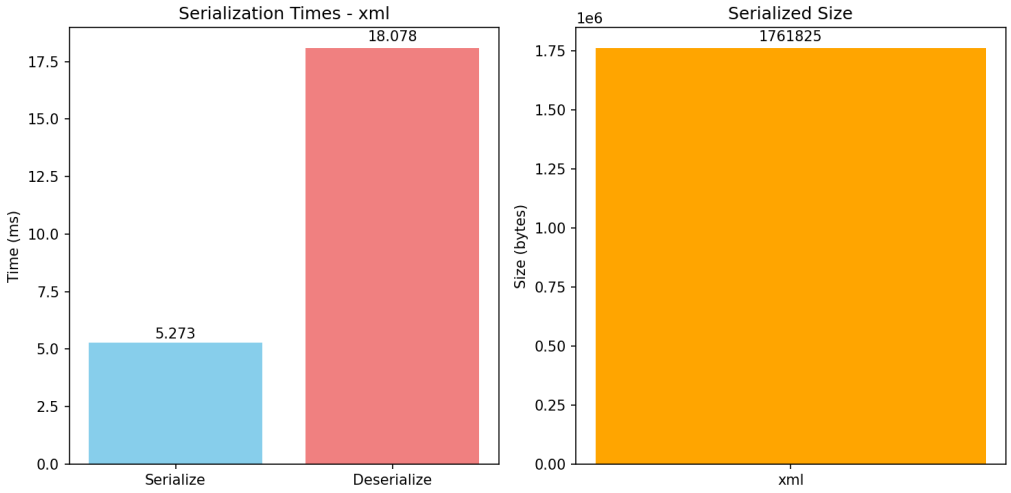
### 5.9.5 Protocol Buffers




Rysunek 5.7: Metryki wydajności Protocol Buffers.

Metryki zawarte na Rysunku 5.7 wykazują, że format Protocol Buffers osiągnął czas serializacji 1,648 ms, deserializacji 4,724 ms, generując dane o rozmiarze 587 684 bajtów (573,9 KB).

### 5.9.6 XML




Rysunek 5.8: Metryki wydajności XML.

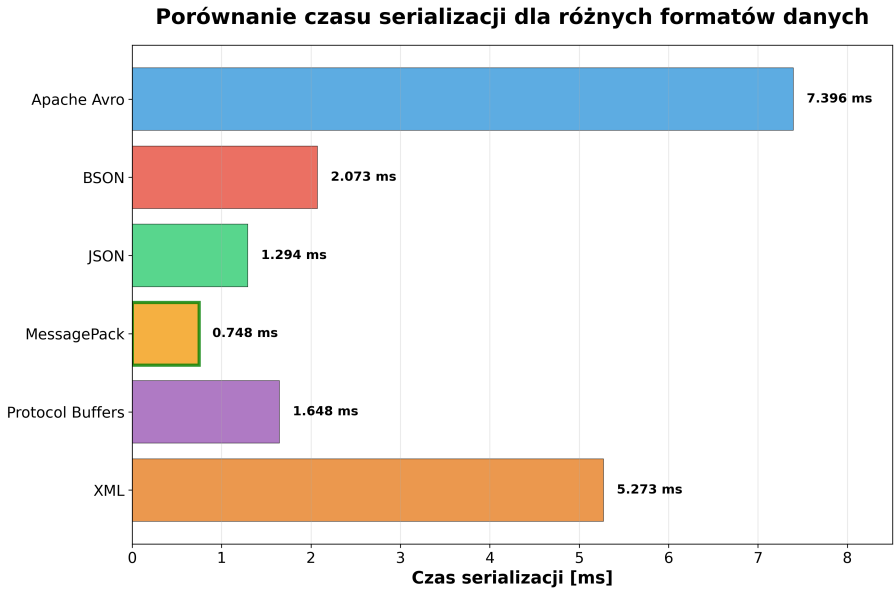
Metryki zawarte na Rysunku 5.8 wykazuje, że format XML wykazał najdłuższe czasy przetwarzania – serializację 5,273 ms i deserializację 18,078 ms – przy największym rozmiarze danych 1 761 825 bajtów (1 720,5 KB), co stanowi 3,4-krotność najbardziej efektywnych formatów binarnych.

### 5.9.7 Analiza porównawcza formatów serializacji

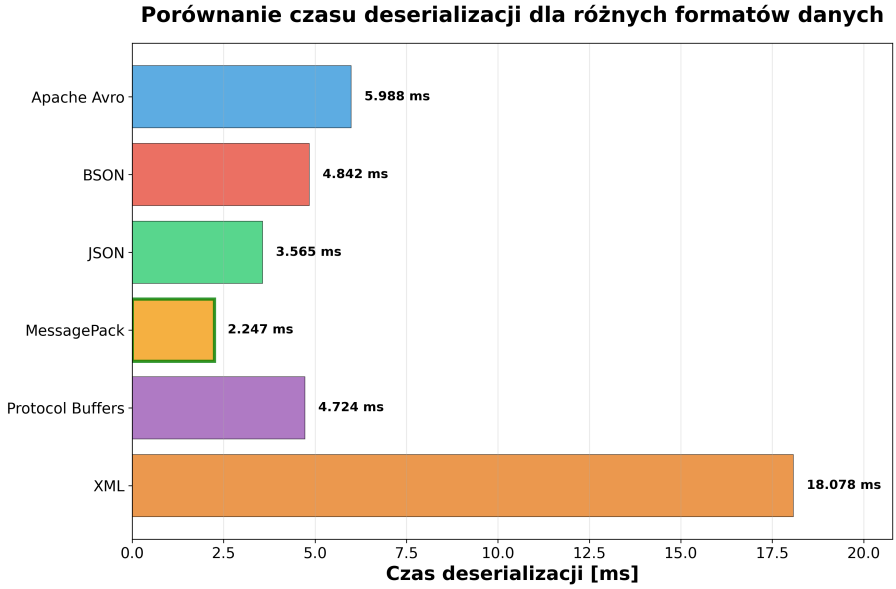
Kompleksowa analiza wydajności formatów serializacji wymaga uwzględnienia trzech kluczowych metryk: czasu serializacji, czasu deserializacji oraz rozmiaru wynikowych danych.

Tabela 5.2: Porównanie formatów serializacji danych

| **Format** | **Serializacja [ms]** | **Deserializacja [ms]** | **Rozmiar [KB]** |
|---|---|---|---|
| Apache Avro | 7,396 | 5,988 | 509,5 |
| BSON | 2,073 | 4,842 | 1393,3 |
| JSON | 1,294 | 3,565 | 1154,1 |
| MessagePack | 0,748 | 2,247 | 515,1 |
| Protocol Buffers | 1,648 | 4,724 | 573,9 |
| XML | 5,273 | 18,078 | 1720,5 |




Rysunek 5.9: Porównanie czasu serializacji różnych formatów.




Rysunek 5.10: Porównanie czasu deserializacji formatowania danych.

### Wydajność czasowa

`MessagePack` wykazał najlepszą wydajność czasową we wszystkich kategoriach:

- Najszybsza serializacja: 0,748 ms (6,0× szybciej niż XML, 9,9× szybciej niż Apache Avro)
- Najszybsza deserializacja: 2,247 ms (8,0× szybciej niż XML, 2,7× szybciej niż Apache Avro)
- Najniższy całkowity czas przetwarzania: 2,995 ms

JSON i texttttProtocol Buffers osiągnęły porównywalne wyniki czasowe (odpowiednio 4,859 ms i 6,372 ms łącznie), oferując dobry kompromis między wydajnością a czytelnością (JSON) lub efektywnością kompresji (Protobuf).

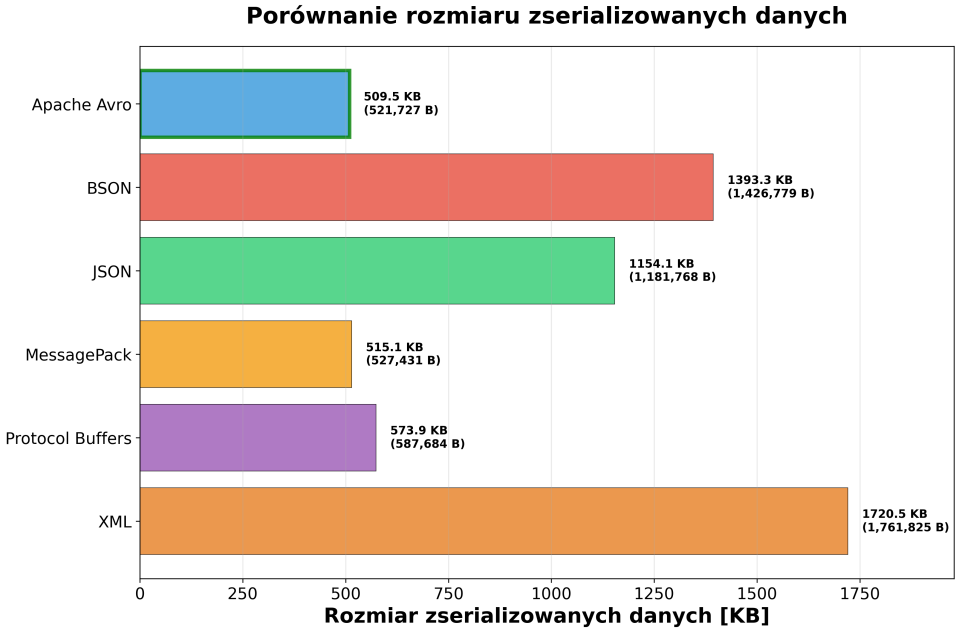
XML wykazał najgorsze charakterystyki czasowe z deserializacją trwającą 18,078 ms – ponad 8× dłużej niż najbliższy konkurent (Apache Avro). Ta degradacja wynika z parsowania tekstowej struktury hierarchicznej oraz narzutu związanego z walidacją schematu.

### Efektywność kompresji

Apache Avro osiągnął najlepszą kompresję danych z rozmiarem 521 727 bajtów (509,5 KB), co stanowi:

- 2,2× mniejszy rozmiar niż JSON
- 2,7× mniejszy rozmiar niż BSON
- 3,4× mniejszy rozmiar niż XML

MessagePack wykazał rozmiar 527 431 bajtów (515,1 KB) – zaledwie 1,1% większy niż Avro, co w połączeniu z najlepszą wydajnością czasową czyni go najbardziej zrównoważonym formatem.




Rysunek 5.11: Porównanie rozmiaru zserializowanych danych dla różnych formatów.

XML i BSON wykazały najbardziej nieefektywną kompresję, przekraczając 1,4 MB dla tego samego zestawu danych. W przypadku XML nadmiarowy rozmiar wynika z tagów otwierających i zamykających oraz metadanych, podczas gdy BSON przechowuje dodatkowe informacje typów dla każdego pola.

### Charakterystyki formatów i rekomendacje

**MessagePack** – optymalny wybór dla aplikacji wymagających maksymalnej wydajności:

- Najszybszy całkowity czas przetwarzania (2,995 ms)
- Bardzo dobra kompresja (515 KB, +1% względem najlepszego)
- Idealne zastosowanie: real-time gaming, high-frequency trading, cache'owanie
- Ograniczenie: brak natywnej czytelności dla człowieka

**Apache Avro** – najlepsza kompresja dla big data:

- Najmniejszy rozmiar danych (509,5 KB)
- Schema evolution – wsparcie dla zmian struktury danych w czasie
- Idealne zastosowanie: Apache Kafka, Hadoop, długoterminowe przechowywanie danych
- Ograniczenie: wolniejsza serializacja (7,4 ms) – nie nadaje się dla ultra-low-latency

**JSON** – balans między wydajnością a developer experience:

- Szybka serializacja (1,294 ms), umiarkowana deserializacja (3,565 ms)
- Natywna czytelność, debugowalność, wsparcie w przeglądarkach
- Idealne zastosowanie: REST APIs, konfigurace, komunikacja frontend-backend
- Kompromis: 2,2× większy rozmiar niż formaty binarne

**Protocol Buffers** – wydajność dla kontraktów API:

- Dobra równowaga: średni czas (6,4 ms), średni rozmiar (574 KB)
- Silne typowanie, backward/forward compatibility
- Idealne zastosowanie: gRPC, komunikacja mikroserwisowa, mobile APIs
- Wymaganie: generowanie kodu z plików .proto

**BSON** – specjalizacja dla baz dokumentowych:

- Szybka serializacja (2,073 ms), dedykowana dla MongoDB
- Wsparcie dla typów binarnych (ObjectId, DateTime, Binary)
- Idealne zastosowanie: MongoDB storage/queries, document databases
- Ograniczenie: 2,7× większy rozmiar niż formaty binarne

**XML** – legacy i compliance:

- Wsparcie dla złożonych schematów (XSD), przestrzeni nazw, transformacji (XSLT)
- Najgorsze metryki wydajnościowe: 23,4 ms łącznie, 1721 KB
- Idealne zastosowanie: systemy enterprise, SOAP, compliance (HIPAA, financial regulations)
- Uzasadnienie: wymagania regulacyjne często przeważają nad wydajnością

**Wnioski**

Przeprowadzone badania potwierdzają, że nie istnieje uniwersalnie optymalny format serializacji – wybór powinien być determinowany przez specyficzne wymagania aplikacji:

1. **Dla maksymalnej wydajności:** MessagePack oferuje najlepszy czas przetwarzania przy akceptowalnym rozmiarze
2. **Dla minimalnego rozmiaru:** Apache Avro zapewnia najlepszą kompresję, idealny dla storage i network bandwidth
3. **Dla developer experience:** JSON oferuje czytelność i debugowalność kosztem rozmiaru
4. **Dla typowanych API:** Protocol Buffers zapewnia kontrakt i kompatybilność wsteczną
5. **Dla compliance:** XML pozostaje niezbędny w regulowanych środowiskach enterprise

Kluczowym odkryciem jest, że różnica między najszybszym (MessagePack: 2,995 ms) a najwolniejszym (XML: 23,351 ms) formatem wynosi 7,8×, podczas gdy różnica w rozmiarze wynosi 3,4×. To wskazuje, że wydajność czasowa wykazuje większą wariancję niż efektywność kompresji, czyniąc ją krytycznym czynnikiem decyzyjnym w aplikacjach real-time.

## 5.10 Analiza porównawcza WebAssembly i JavaScript

Do stworzenia modułów WebAssembly wykorzystano narzędzie `wasm-pack`. Jest to gotowe rozwiązanie, które kompiluje kod z języka Rust na język JavaScript, gdzie część kodu jest napisana w WebAssembly. Takie podejście umożliwia przeniesienie wybranych fragmentów kodu do WebAssembly, co pozwala na przyśpieszenie działania aplikacji.

### 5.10.1 Web Workers

Technologia Web Workers umożliwia przerzucenie części kodu, który wykonuje się na wątku głównym, na wątek poboczny. Stworzone zostały dwa przykładowe workery zawierające kod napisany w języku Rust (z równoważnym kodem napisanym w JavaScript) – jeden z małym obciążeniem obliczeniowym i drugi z dużym obciążeniem obliczeniowym.

**Implementacja workerów o niskim obciążeniu obliczeniowym**

Listing 5.11 przedstawia implementację Web Workera w środowisku WebAssembly z wykorzystaniem biblioteki wasm-bindgen. Funkcja `worker_entry()` stanowi punkt wejścia workera i inicjalizuje obsługę komunikatów przychodzących z głównego wątku. Odebrane dane są interpretowane jako tablica liczb zmiennoprzecinkowych typu Float64Array, na której wykonywane są operacje obliczeniowe polegające na sumowaniu iloczynów pierwiastków kwadratowych oraz logarytmów naturalnych kolejnych elementów. Czas wykonania obliczeń mierzony jest przy użyciu interfejsu Performance, a uzyskany wynik czasowy przesyłany jest z powrotem do głównego wątku za pomocą mechanizmu postMessage, co umożliwia ocenę kosztu obliczeń realizowanych w osobnym wątku.

```
#[wasm_bindgen(start)]
pub fn worker_entry() {
    let global: DedicatedWorkerGlobalScope =
```

```
    js_sys::global().unchecked_into();

    let global_for_closure = global.clone();

    let onmessage = Closure::<dyn FnMut(_)>::new(move |event: web_sys::
        MessageEvent| {
        let buffer = event.data();
        let data = Float64Array::new(&buffer);

        let performance = global_for_closure.performance().unwrap();
        let t0 = performance.now();

        let mut sum = 0.0;
        for i in 0..data.length() {
            let x = data.get_index(i);
            sum += x.sqrt() * (x + 1.0).ln();
        }

        let t1 = performance.now();

        let response = js_sys::Object::new();
        js_sys::Reflect::set(&response, &"time".into(), &(t1 - t0).into
            ()).unwrap();

        global_for_closure.post_message(&response).unwrap();
    });

    global.set_onmessage(Some(onmessage.as_ref().unchecked_ref()));
    onmessage.forget();
}
```

Listing 5.11: Implementacja Web Workera z prostymi operacjami obliczeniowymi

### Implementacja workerów o wysokim obciążeniu obliczeniowym

Listing 5.12 pokazuję implementację Web Workera wykonanego w bardzo podobny sposób do poprzedniej implementacji (Listing 5.11), z tym że wykonuje złożone operacje obliczeniowe z wykorzystaniem mapy chaotycznej (ang. *chaotic map*). Algorytm ten symuluje zachowanie chaotyczne poprzez iteracyjne obliczenia wykorzystujące mapowanie logistyczne.

```
#[wasm_bindgen(start)]
pub fn worker_entry() {
    let global: DedicatedWorkerGlobalScope =
    js_sys::global().unchecked_into();

    let global_for_closure = global.clone();

    let onmessage = Closure::<dyn FnMut(_)>::new(move |event: web_sys::
        MessageEvent| {
        let buffer = event.data();
        let data = Float64Array::new(&buffer);

```

```
        let perf = global_for_closure.performance().unwrap();
        let t0 = perf.now();

        let mut acc = 0.0;
        let mut seed = 0.123456789_f64;

        for i in 0..data.length() {
            let v = data.get_index(i);

            // Mapowanie chaotyczne (logistic map)
            seed = seed * 3.999999 - seed * seed;
            let x = seed + v;

            seed = seed * 3.999999 - seed * seed;
            let y = seed + v;

            acc += (x * x + y * y).sqrt().ln();
        }

        let t1 = perf.now();

        let result = js_sys::Object::new();
        js_sys::Reflect::set(&result, &"value".into(), &acc.into()).
           unwrap();
        js_sys::Reflect::set(&result, &"time".into(), &(t1 - t0).into()
           ).unwrap();

        global_for_closure.post_message(&result).unwrap();
    });

    global.set_onmessage(Some(onmessage.as_ref().unchecked_ref()));
    onmessage.forget();
}
```

Listing 5.12: Implementacja Web Workera z zaawansowanymi operacjami obliczeniowymi

### Wyniki pomiarów – Web Workers

Tabela 5.3 przedstawia porównanie czasów wykonania dla obu implementacji (JavaScript i WebAssembly) w zależności od złożoności zadania obliczeniowego.

Tabela 5.3: Porównanie wydajności Web Workers – JavaScript vs. WebAssembly

| **Typ zadania** | **JavaScript [ms]** | **WebAssembly [ms]** |
|---|---|---|
| Małe zadanie obliczeniowe | 9,6 | 43,1 |
| Duże zadanie obliczeniowe | 113,3 | 91,69 |

Wyniki wskazują, że dla prostych operacji JavaScript osiąga lepszą wydajność (9,6 ms vs. 43,1 ms), prawdopodobnie ze względu na narzut związany z inicjalizacją modułu WebAssembly oraz przekazywaniem danych między środowiskami. Natomiast w przypadku złożonych obliczeń WebAssembly wykazuje przewagę wydajnościową (91,69 ms vs. 113,3 ms), co potwierdza jego przydatność w scenariuszach wymagających intensywnych operacji numerycznych.

### 5.10.2 Operacje na Canvas

Drugi zestaw testów dotyczył operacji przetwarzania obrazu na elemencie Canvas. Wszystkie operacje zostały wykonane na obrazie w rozdzielczości 4K (3840 × 2160 pikseli), co odpowiada przetwarzaniu 8 294 400 pikseli (każdy reprezentowany przez 4 bajty w formacie RGBA).

#### Implementacja filtrów graficznych

Listing 5.13 zawiera implementację czterech podstawowych operacji przetwarzania obrazu: konwersji do skali szarości, regulacji jasności, kontrastu oraz rozmycia (blur).

```
#[wasm_bindgen]
pub fn grayscale(data: &overmut [u8]) {
    for i in (0..data.len()).step_by(4) {
        let r = data[i] as u32;
        let g = data[i + 1] as u32;
        let b = data[i + 2] as u32;
        // Wzor uwzgledniajacy percepcje ludzkiego oka
        let gray = ((299 * r + 587 * g + 114 * b) / 1000) as u8;
        data[i] = gray;
        data[i + 1] = gray;
        data[i + 2] = gray;
    }
}

#[wasm_bindgen]
pub fn brightness(data: &mut [u8], amount: i32) {
    for i in (0..data.len()).step_by(4) {
        data[i] = (data[i] as i32 + amount).clamp(0, 255) as u8;
        data[i + 1] = (data[i + 1] as i32 + amount).clamp(0, 255) as u8
            ;
        data[i + 2] = (data[i + 2] as i32 + amount).clamp(0, 255) as u8
            ;
    }
}

#[wasm_bindgen]
pub fn contrast(data: &mut [u8], factor: f32) {
    let f = (259.0 * (factor + 255.0)) / (255.0 * (259.0 - factor));
    for i in (0..data.len()).step_by(4) {
        data[i] = (f * (data[i] as f32 - 128.0) + 128.0).clamp(0.0,
            255.0) as u8;
        data[i + 1] = (f * (data[i + 1] as f32 - 128.0) + 128.0).clamp
            (0.0, 255.0) as u8;
        data[i + 2] = (f * (data[i + 2] as f32 - 128.0) + 128.0).clamp
            (0.0, 255.0) as u8;
    }
}

#[wasm_bindgen]
pub fn blur(data: &mut [u8], width: usize, height: usize, radius: usize
    ) {
```

```
    let temp = data.to_vec();
    let radius_i = radius as i32;

    for y in 0..height {
        for x in 0..width {
            let mut r_sum = 0u32;
            let mut g_sum = 0u32;
            let mut b_sum = 0u32;
            let mut count = 0u32;

            let y_start = (y as i32 - radius_i).max(0) as usize;
            let y_end = (y + radius + 1).min(height);
            let x_start = (x as i32 - radius_i).max(0) as usize;
            let x_end = (x + radius + 1).min(width);

            for ny in y_start..y_end {
                for nx in x_start..x_end {
                    let idx = (ny * width + nx) * 4;
                    r_sum += temp[idx] as u32;
                    g_sum += temp[idx + 1] as u32;
                    b_sum += temp[idx + 2] as u32;
                    count += 1;
                }
            }

            let idx = (y * width + x) * 4;
            data[idx] = (r_sum / count) as u8;
            data[idx + 1] = (g_sum / count) as u8;
            data[idx + 2] = (b_sum / count) as u8;
        }
    }
}
```

Listing 5.13: Implementacja filtrów graficznych dla operacji Canvas

### Wyniki pomiarów – operacje Canvas

Tabela 5.4 przedstawia szczegółowe porównanie czasów wykonania dla każdej z czterech operacji graficznych, wraz z wyliczonym współczynnikiem przyspieszenia.

Tabela 5.4: Porównanie wydajności operacji Canvas – JavaScript vs. WebAssembly

| **Operacja** | **JavaScript [ms]** | **WebAssembly [ms]** | **Współczynnik** |
|---|---|---|---|
| Grayscale | 23,6 | 26,3 | 0,90× |
| Brightness | 22,1 | 34,3 | 0,64× |
| Contrast | 28,4 | 51,1 | 0,56× |
| Blur | 1308,5 | 371,3 | 3,52× |

Analiza wyników ujawnia interesujący wzorzec: dla prostych operacji pikselowych (grayscale, brightness, contrast) JavaScript przewyższa WebAssembly pod względem wydajności. Natomiast w przypadku operacji rozmycia (blur), która charakteryzuje się znacznie wyższą

złożonością obliczeniową ($O(n \cdot r^2)$, gdzie $n$ to liczba pikseli, a $r$ to promień rozmycia), WebAssembly osiąga ponad 3,5-krotne przyspieszenie (371,3 ms kontra 1308,5 ms).

### 5.10.3 Wnioski

Przeprowadzone testy wydajnościowe pozwalają na sformułowanie następujących wniosków:

1. **Złożoność obliczeniowa jako czynnik decydujący** – WebAssembly wykazuje przewagę wydajnościową głównie w scenariuszach wymagających intensywnych obliczeń o wysokiej złożoności algorytmicznej. Dla operacji prostych narzut związany z inicjalizacją i komunikacją między środowiskami przewyższa potencjalne korzyści.

2. **Optymalizacje specyficzne dla przeglądarki** – nowoczesne silniki JavaScript, takie jak V8, zawierają rozbudowane optymalizacje dla typowych operacji webowych, co może przeważać nad wydajnością WebAssembly w przypadku prostych transformacji danych.

3. **Rekomendacje praktyczne** – WebAssembly stanowi optymalny wybór dla:

   - algorytmów o złożoności $O(n^2)$ lub wyższej,
   - przetwarzania dużych zbiorów danych,
   - operacji wymagających przewidywalnej wydajności,
   - portowania istniejących bibliotek C/C++/Rust.

   Z kolei JavaScript pozostaje preferowany dla operacji o niskiej złożoności, szybkiego prototypowania oraz scenariuszy wymagających częstej interakcji z DOM.

Podsumowując, wybór między WebAssembly a JavaScript powinien być uzależniony od specyfiki problemu – optymalne rozwiązanie często stanowi architektura hybrydowa, łącząca elastyczność JavaScript z wydajnością WebAssembly.

# Zakończenie

Niniejsza praca magisterska poświęcona była identyfikacji i ocenie metod optymalizacji aplikacji webowych poprzez kompleksową analizę wydajności protokołów komunikacyjnych, formatów serializacji danych oraz technologii wykonawczych. Przeprowadzone badania eksperymentalne dostarczyły cennych danych ilościowych oraz pozwoliły na sformułowanie praktycznych strategii optymalizacyjnych dla architektów systemów i programistów.

## Podsumowanie wyników badań

Analiza protokołów komunikacyjnych wykazała, że optymalizacja aplikacji webowych wymaga doboru rozwiązań dostosowanych do konkretnych scenariuszy użycia. REST API z protokołem HTTP/1.1 osiągnął najwyższą przepustowość dla małych pakietów danych (764 076 żądań na sekundę), co pozwala efektywnie optymalizować mikrousługi operujące na niewielkich zbiorach danych niewymagających bezpiecznego połączenia. Z kolei WebSocket wykazał najlepszą wydajność dla transferu dużych ładunków (3 187 żądań na sekundę) oraz najniższy współczynnik degradacji (38,4×), potwierdzając, że zastosowanie protokołów z trwałymi połączeniami stanowi kluczową metodę optymalizacji w scenariuszach wymagających intensywnej wymiany danych.

Szczególnie istotnym odkryciem dla optymalizacji aplikacji była efektywność protokołu HTTP/2 w mitigacji problemu Head-of-Line blocking. Osiągnięta 69-krotna poprawa przepustowości w porównaniu z HTTP/1.1 (135 372 vs. 1 954 żądań na sekundę) wskazuje, że migracja na HTTP/2 stanowi jedną z najskuteczniejszych metod optymalizacji nowoczesnych aplikacji webowych. Dodatkowym aspektem wartym podkreślenia jest stosunkowo niewielki narzut wydajnościowy związany z warstwą TLS – degradacja rzędu 5–15% jest w pełni akceptowalna w kontekście krytycznych wymagań bezpieczeństwa współczesnych systemów.

W zakresie formatów serializacji zidentyfikowano konkretne możliwości optymalizacyjne. MessagePack osiągnął najkrótszy całkowity czas przetwarzania (2,995 ms), podczas gdy Apache Avro zapewnił najlepszą kompresję danych (509,5 KB). Wybór formatu binarnego zamiast JSON może przynieść 2,2-krotną redukcję rozmiaru przesyłanych danych, co stanowi istotną optymalizację dla aplikacji obsługujących duże wolumeny transferu. Z kolei XML, choć wykazał najgorsze metryki wydajnościowe (23,4 ms łącznie, 1721 KB), zachowuje znaczenie w środowiskach enterprise ze względu na wymagania regulacyjne oraz wsparcie dla złożonych schematów walidacji.

Analiza technologii WebAssembly dostarczyła kluczowych wskazówek dotyczących optymalizacji wydajności po stronie klienta. Złożoność obliczeniowa stanowi czynnik decydujący o opłacalności implementacji w WebAssembly jako metodzie optymalizacji. Podczas gdy proste operacje (grayscale, brightness, contrast) wykazały lepszą wydajność w JavaScript, złożone algorytmy – reprezentowane przez operację blur – osiągnęły 3,52-krotne przyspieszenie w WebAssembly (371,3 ms vs. 1308,5 ms). Ten wzorzec wskazuje, że selektywne zastosowa-

nie WebAssembly dla operacji o wysokiej złożoności obliczeniowej stanowi efektywną strategię optymalizacyjną, podczas gdy JavaScript pozostaje preferowany dla operacji o niskiej złożoności oraz częstej interakcji z DOM.

## Wkład naukowy i praktyczny

Wartość niniejszej pracy przejawia się w kilku kluczowych aspektach. Po pierwsze, przeprowadzone badania opierają się na rzeczywistych implementacjach oraz kontrolowanych eksperymentach, dostarczając konkretnych metryk umożliwiających podejmowanie świadomych decyzji optymalizacyjnych. Ujednolicona metodologia testowania (100 współbieżnych operacji, 1000 zapytań na wątek) zapewnia porównywalność wyników między różnymi protokołami i formatami.

Po drugie, praca dostarcza konkretnych, kwantyfikowalnych strategii optymalizacyjnych, które mogą stanowić bezpośrednie wsparcie w procesie poprawy wydajności aplikacji. Zamiast ogólnych stwierdzeń typu "protokół X jest szybszy od Y", prezentowane są precyzyjne dane numeryczne wraz z analizą kontekstów, w których dana technologia umożliwia osiągnięcie optymalnej wydajności.

Po trzecie, sformułowane rekomendacje optymalizacyjne uwzględniają wielowymiarowy charakter wydajności systemu – nie ograniczają się wyłącznie do przepustowości, ale uwzględniają również aspekty takie jak skalowalność, kompatybilność wsteczna, developer experience oraz wymagania bezpieczeństwa i compliance.

## Ograniczenia badań

Przy interpretacji wyników należy uwzględnić pewne ograniczenia przeprowadzonych badań. Testy wykonano w kontrolowanym środowisku testowym, które nie w pełni odzwierciedla złożoność środowisk produkcyjnych, charakteryzujących się zmiennym obciążeniem sieciowym, różnorodną infrastrukturą oraz nieprzewidywalnymi wzorcami użytkowania.

Dodatkowo, implementacje testowe zostały zoptymalizowane pod kątem maksymalnej wydajności dla poszczególnych protokołów, co może nie odzwierciedlać typowych wdrożeń produkcyjnych, w których priorytetem jest często równowaga między wydajnością a możliwością utrzymania kodu oraz zgodnością z istniejącą architekturą systemu.

Warto również zaznaczyć, że wyniki dotyczące WebAssembly odnoszą się do konkretnych implementacji algorytmów i mogą różnić się w zależności od charakterystyki przetwarzanych danych oraz specyfiki wykorzystywanych bibliotek.

## Kierunki dalszych badań

Przeprowadzone badania otwierają kilka perspektywicznych kierunków dalszych eksploracji w zakresie optymalizacji aplikacji webowych. Pierwszym z nich jest analiza możliwości optymalizacyjnych w środowiskach rozproszonych charakteryzujących się wysokim opóźnieniem sieciowym oraz niestabilnymi połączeniami, co jest szczególnie istotne w kontekście aplikacji mobilnych oraz systemów IoT.

Interesującym kierunkiem jest również badanie potencjału optymalizacyjnego nowych wersji protokołów – w szczególności HTTP/3 z protokołem QUIC. HTTP/3, eliminujący problem Head-of-Line blocking na poziomie warstwy transportowej, może wprowadzić nowe możliwości optymalizacji wydajności aplikacji webowych.

Wreszcie, w kontekście rosnącej popularności architektur serverless oraz edge computing, wartościowe byłoby zbadanie strategii optymalizacyjnych specyficznych dla środowisk o dynamicznej skalowalności i geograficznie rozproszonych węzłach obliczeniowych.

# Wnioski końcowe

Głównym przesłaniem niniejszej pracy jest przekonanie, że efektywna optymalizacja aplikacji webowych wymaga świadomych i opartych na danych empirycznych decyzji architektonicznych. Wybór protokołu komunikacyjnego, formatu serializacji oraz technologii wykonawczej powinien być determinowany przez specyficzne wymagania aplikacji, uwzględniając takie czynniki jak charakterystyka przetwarzanych danych, oczekiwana latencja, wymagania bezpieczeństwa oraz ograniczenia infrastrukturalne.

Przeprowadzone badania potwierdzają, że nowoczesne aplikacje webowe mogą osiągać wysoką wydajność poprzez inteligentne wykorzystanie dostępnych technologii i świadome stosowanie metod optymalizacyjnych. REST API pozostaje doskonałym wyborem dla większości aplikacji webowych dzięki skalowalności i uniwersalności, WebSocket okazuje się optymalny dla komunikacji w czasie rzeczywistym, gRPC przewyższa konkurencję w komunikacji backend-to-backend, a WebAssembly stanowi wartościowe narzędzie optymalizacji w scenariuszach wymagających intensywnych obliczeń.

Równocześnie praca podkreśla, że optymalizacja wydajności nie powinna być traktowana jako wyłączny czynnik decyzyjny – aspekty takie jak możliwość utrzymania kodu, ekosystem narzędzi, wsparcie społeczności oraz kompatybilność z istniejącymi systemami często przeważają nad przewagą wydajnościową rzędu kilku czy kilkunastu procent. Skuteczna optymalizacja wymaga holistycznego podejścia uwzględniającego zarówno metryki wydajnościowe, jak i długoterminowe koszty utrzymania systemu.

Podsumowując, niniejsza praca dostarcza praktycznych strategii optymalizacyjnych oraz danych empirycznych, które mogą wspierać proces projektowania i poprawy wydajności nowoczesnych aplikacji webowych. Mam nadzieję, że przedstawione wyniki i rekomendacje okażą się wartościowe dla architektów systemów, programistów oraz badaczy zajmujących się optymalizacją aplikacji internetowych.

# Bibliografia


[1] Anthropic. bun documentation. `https://bun.com//`. [dostęp: 2 luty 2026].

[2] Dan Burkert. Prost. `https://github.com/tokio-rs/prost`. [dostęp: 2 luty 2026].

[3] David Tolnay, Erick Tryzelaar, Oli Scherer. Serde. `https://serde.rs/`. [dostęp: 2 luty 2026].

[4] Git Project. Git. `https://git-scm.com/`. [dostęp: 21 stycznia 2026].

[5] Google. Javascript v8 engine. `https://v8.dev/`. [dostęp: 2 luty 2026].

[6] Google. Protocol buffers. `https://protobuf.dev/`. [dostęp: 2 luty 2026].

[7] Kevin Hoffman. *Programming WebAssembly with Rust, Unified Development for Web, Mobile, and Embedded Applications*. Pragmatic Bookshelf, 2019.

[8] Johann Tuffe, Daniel Alley, Mingun. quck-xml. `https://crates.io/crates/quick-xml/0.24.1`. [dostęp: 2 luty 2026].

[9] Kai Ren, Magnus Hallin, Christoph Herzog, Christian Legnitto. Juniper graphql server library for rust. `https://crates.io/crates/juniper`. [dostęp: 2 luty 2026].

[10] Martin Kleppmann. *Designing Data-Intensive Applications. The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017.

[11] Microsoft. Typescript. `https://www.typescriptlang.org/`. [dostęp: 2 luty 2026].

[12] Microsoft. Visual studio code. `https://code.visualstudio.com/`. [dostęp: 21 stycznia 2026].

[13] MQTT.org. Message queuing telemetry transport. `https://mqtt.org/`. [dostęp: 2 luty 2026].

[14] OpenJS Foundation oraz kontrybutorzy Node.js. Node.js. `https://nodejs.org/en`. [dostęp: 9 luty 2026].

[15] Luca Palmieri. *Zero to Production In Rust: An introduction to backend development in Rust*. 2022.

[16] Ravi Teja. rumqttc. `https://crates.io/crates/rumqttc`. [dostęp: 2 luty 2026].

[17] Carlos Santana Roldán. *React 18 Design Patterns and Best Practices. Design, build, and deploy production-ready web applications with React by leveraging industry-best practices - Fourth Edition*. Packt Publishing, 2023.

[18] Rust Project. Cargo – the rust package manager. https://doc.rust-lang.org/cargo/. [dostęp: 21 stycznia 2026].

[19] Rust Team. Rust language. https://rust-lang.org/. [dostęp: 2 luty 2026].

[20] Sadayuki Furuhashi. Messagepack serialization format. https://msgpack.org/. [dostęp: 9 luty 2026].

[21] Snapview GmbH. tungstenite-rs. https://github.com/snapview/tungstenite-rs. [dostęp: 2 luty 2026].

[22] Tatsuhiro Tsujikawa. h2load. https://nghttp2.org/documentation/h2load-howto.html#. [dostęp: 2 luty 2026].

[23] Team hyperium tonic. Tonic framework. https://crates.io/crates/tonic. [dostęp: 2 luty 2026].

[24] @The Actix Team. Actix-web. https://actix.rs/. [dostęp: 2 luty 2026].

[25] The Apache Software Foundation. Apache avro. https://avro.apache.org/docs/. [dostęp: 2 luty 2026].

[26] The GraphQL Foundation. Graphql. https://graphql.org/. [dostęp: 2 luty 2026].

[27] Webassembly.org. Webassembly. https://webassembly.org/. [dostęp: 21 stycznia 2026].

[28] Wikipedia. Bson. https://en.wikipedia.org/wiki/BSON. [dostęp: 2 luty 2026].

[29] Wikipedia. Deserializacja. https://en.wikipedia.org/wiki/Marshalling_(computer_science)#Unmarshalling. [dostęp: 21 stycznia 2026].

[30] Wikipedia. Document object model. https://pl.wikipedia.org/wiki/Obiektowy_model_dokumentu. [dostęp: 2 luty 2026].

[31] Wikipedia. Extensible markup language. https://pl.wikipedia.org/wiki/XML. [dostęp: 2 luty 2026].

[32] Wikipedia. google remote procedure call. https://en.wikipedia.org/wiki/GRPC. [dostęp: 21 stycznia 2026].

[33] Wikipedia. Head-of-line-blocking. https://en.wikipedia.org/wiki/Head-of-line_blocking. [dostęp: 2 luty 2026].

[34] Wikipedia. Hypertext transfer protocol. https://pl.wikipedia.org/wiki/HTTP. [dostęp: 21 stycznia 2026].

[35] Wikipedia. Interfejs programowania aplikacji. https://pl.wikipedia.org/wiki/Interfejs_programowania_aplikacji. [dostęp: 21 stycznia 2026].

[36] Wikipedia. Javascript. https://pl.wikipedia.org/wiki/JavaScript. [dostęp: 2 luty 2026].

[37] Wikipedia. Javascript object notation. https://en.wikipedia.org/wiki/JSON. [dostęp: 9 luty 2026].

[38] Wikipedia. Komunikacja zdarzeniowa. https://en.wikipedia.org/wiki/Event-driven_architecture. [dostęp: 9 luty 2026].

[39] Wikipedia. Mikroserwisy. https://en.wikipedia.org/wiki/Microservices. [dostęp: 9 luty 2026].

[40] Wikipedia. Model klient serwer. https://en.wikipedia.org/wiki/Client%E2%80%93server_model. [dostęp: 21 stycznia 2026].

[41] Wikipedia. Python. https://en.wikipedia.org/wiki/Python_(programming_language). [dostęp: 2 luty 2026].

[42] Wikipedia. Remote procedure call. https://en.wikipedia.org/wiki/Remote_procedure_call. [dostęp: 09 lutego 2026].

[43] Wikipedia. Serializacja. https://en.wikipedia.org/wiki/Serialization. [dostęp: 21 stycznia 2026].

[44] Wikipedia. Simple object access protocol. https://en.wikipedia.org/wiki/SOAP. [dostęp: 2 luty 2026].

[45] Wikipedia. Strumieniowanie. https://en.wikipedia.org/wiki/Stream_(computing). [dostęp: 2 luty 2026].

[46] Wikipedia. Systemy rozproszone. https://pl.wikipedia.org/wiki/System_rozproszony. [dostęp: 9 luty 2026].

[47] Wikipedia. Tls - transport layer security. https://pl.wikipedia.org/wiki/Transport_Layer_Security. [dostęp: 2 luty 2026].

[48] Wikipedia. Web worker. https://en.wikipedia.org/wiki/Web_worker. [dostęp: 9 luty 2026].

[49] Wikipedia. Websocket. https://en.wikipedia.org/wiki/WebSocket. [dostęp: 21 stycznia 2026].

[50] Zed Industries. Zed editor. https://zed.dev/. [dostęp: 21 stycznia 2026].

# Spis rysunków

# Spis listingów

# Dodatek A

# Dodatkowy kod źródłowy

Poniżej znajduje się autorski kod wykorzystany w badaniach.

## A.1 Konfiguracja TLS 1.3

```rust
use std::{fs::File, io::BufReader};

use rustls::ServerConfig;

pub fn tls_config() -> ServerConfig {
    rustls::crypto::aws_lc_rs::default_provider()
    .install_default()
    .unwrap();

    let mut certs_file = BufReader::new(File::open("cert.pem").unwrap()
       );
    let mut key_file = BufReader::new(File::open("key.pem").unwrap());

    let tls_certs = rustls_pemfile::certs(&mut certs_file)
    .collect::<Result<Vec<_>, _>>()
    .unwrap();
    let tls_key = rustls_pemfile::pkcs8_private_keys(&mut key_file)
    .next()
    .unwrap()
    .unwrap();

    rustls::ServerConfig::builder()
    .with_no_client_auth()
    .with_single_cert(
    tls_certs,
    rustls::pki_types::PrivateKeyDer::Pkcs8(tls_key),
    )
    .unwrap()
}
```

Listing A.1: Konfiguracja serwera TLS 1.3 z wykorzystaniem biblioteki rustls

## A.2 Implementacja funkcji, która wykonywała zapytania do brokera MQTT

```rust
async fn run_worker(
worker_id: usize,
config: BenchmarkConfig,
semaphore: Arc<Semaphore>,
) -> Result<Vec<Duration>, Box<dyn std::error::Error + Send + Sync>> {
    let mut latencies = Vec::with_capacity(config.requests_per_worker);

    // Create unique client ID for this worker
    let client_id = format!("benchmark_worker_{}", worker_id);
    let mut opts = MqttOptions::new(client_id, "127.0.0.1", 1883);
    opts.set_keep_alive(Duration::from_secs(30));

    // Increase max packet size to handle 1MB+ payloads
    opts.set_max_packet_size(10 * 1024 * 1024, 10 * 1024 * 1024); // 10
        MB max

    let (client, mut eventloop) = AsyncClient::new(opts, 1000);

    let (response_tx, mut response_rx) = mpsc::unbounded_channel();

    // Spawn event loop handler
    let response_topic = if config.high_payload {
        "benchmark/high_payload_response"
    } else {
        "benchmark/response"
    };

    tokio::spawn(async move {
        loop {
            match eventloop.poll().await {
                Ok(Event::Incoming(Incoming::Publish(p))) => {
                    if p.topic == response_topic {
                        let _ = response_tx.send(());
                    }
                }
                Ok(_) => {}
                Err(e) => {
                    eprintln!("Worker {} eventloop error: {:?}",
                       worker_id, e);
                    break;
                }
            }
        }
    });

    // Subscribe to response topic
    client
```

```
    .subscribe(response_topic, QoS::AtLeastOnce)
    .await?;

    // Wait a bit for subscription to be established
    tokio::time::sleep(Duration::from_millis(100)).await;

    let request_topic = if config.high_payload {
        "benchmark/high_payload"
    } else {
        "benchmark/request"
    };

    let payload = if config.high_payload {
        serde_json::json!({
            "id": format!("worker_{}", worker_id),
            "payload": "x".repeat(1024 * 1024) // 1 MB payload
        })
        .to_string()
    } else {
        serde_json::json!({
            "id": format!("worker_{}", worker_id),
            "payload": "x".repeat(config.payload_size)
        })
        .to_string()
    };

    for i in 0..config.requests_per_worker {
        // Acquire semaphore permit to control concurrency
        let _permit = semaphore.acquire().await?;

        let start = Instant::now();

        // Send request
        client
        .publish(request_topic, QoS::AtLeastOnce, false, payload.clone
           ())
        .await?;

        // Wait for response with timeout
        match timeout(Duration::from_secs(10), response_rx.recv()).
           await {
            Ok(Some(_)) => {
                let latency = start.elapsed();
                latencies.push(latency);
            }
            Ok(None) => {
                eprintln!("Worker {} request {}: Channel closed",
                   worker_id, i);
                break;
            }
            Err(_) => {
```

```
                eprintln!("Worker {} request {}: Timeout", worker_id, i
                    );
            }
        }

        drop(_permit);
    }

    Ok(latencies)
}
```

Listing A.2: Funkcja, która wykonywała zapytania do brokera MQTT

## A.3 Implementacja serwera websocket

```
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let server = TcpListener::bind("127.0.0.1:9001").await?;
    println!("WebSocket server listening on ws://127.0.0.1:9001");

    let connection_count = Arc::new(AtomicUsize::new(0));

    loop {
        let (stream, _) = server.accept().await?;
        let conn_id = connection_count.fetch_add(1, Ordering::SeqCst);

        tokio::spawn(async move {
            let mut websocket = match accept_async(stream).await {
                Ok(ws) => ws,
                Err(e) => {
                    eprintln!("WebSocket handshake failed: {}", e);
                    return;
                }
            };

            println!("Connection {} established", conn_id);

            let mut request_count = 0;

            while let Some(msg) = websocket.next().await {
                match msg {
                    Ok(Message::Text(text)) => {
                        request_count += 1;

                        match text.as_str() {
                            "test" => {
                                let payload = "x".repeat(PAYLOAD_SIZE);
                                if let Err(e) = websocket.send(Message
                                    ::Text(payload.into())).await {
```

```
                                eprintln!("Connection {} write error: {}", conn_id, e);
                                break;
                            }
                        }
                        "test_high_payload" => {
                            let payload = "x".repeat(HIGH_PAYLOAD_SIZE);
                            if let Err(e) = websocket.send(Message::Text(payload.into())).await {
                                eprintln!("Connection {} write error: {}", conn_id, e);
                                break;
                            }
                        }
                        "stream" => {
                            for count in 0..100 {
                                let chunk = format!("chunk {}\n", count);
                                if let Err(e) = websocket.send(Message::Text(chunk.into())).await {
                                    eprintln!("Connection {} write error at chunk {}: {}", conn_id, count, e);
                                    break;
                                }
                            }
                        }
                        _ => {
                            eprintln!("Connection {} unknown command: {}", conn_id, text);
                        }
                    }

                    if request_count % 100 == 0 {
                        println!("Connection {} processed {} requests", conn_id, request_count);
                    }
                }
                Ok(Message::Close(_)) => {
                    println!("Connection {} closed gracefully after {} requests", conn_id, request_count);
                    let _ = websocket.send(Message::Close(None)).await;
                    break;
                }
                Err(e) => {
                    eprintln!("Connection {} error: {}", conn_id, e);
                    break;
```

```
                        }
                        _ => {}
                    }
                }

                println!("Connection {} handler exiting", conn_id);
            });
        }
    }
```

Listing A.3: Implementacja serwera websocket

## A.4 Implementacja funkcji do testowania rozmiaru, czasu serializacji oraz deserializacji formatu JSON

```
    #[tokio::main]
fn benchmark_json(data: &UserCollection) {
    let start = Instant::now();
    let serialized = serialize_json(data).expect("Serialization failed");
    let serialize_time = start.elapsed();

    let size_bytes = serialized.len();

    let start = Instant::now();
    let deserialized = deserialize_json(&serialized).expect("Deserialization failed");
    let deserialize_time = start.elapsed();

    println!("\n{:-^80}", " Summary ");
    println!("Records:          {}", data.users.len());
    println!("Serialize time:   {:.4} ms", serialize_time.as_secs_f64() * 1000.0);
    println!("Deserialize time: {:.4} ms", deserialize_time.as_secs_f64() * 1000.0);
    println!("Total time:       {:.4} ms", (serialize_time + deserialize_time).as_secs_f64() * 1000.0);
    println!("Size:             {} bytes ({:.2} KB)", size_bytes, size_bytes as f64 / 1024.0);
    println!("Bytes per record: {:.2}", size_bytes as f64 / data.users.len() as f64);
}
```

Listing A.4: Implementacja funkcji do testowania rozmiaru, czasu serializacji oraz deserializacji formatu JSON