Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: **Informatyka techniczna (ITE)**
Specjalność: **Inżynieria systemów informatycznych (INS)**

PRACA DYPLOMOWA MAGISTERSKA

Ocena wydajności skalowalnych systemów webowych typu komunikator zaprojektowanych z wykorzystaniem technologii języka Go

Nikita Stepanenko

Opiekun pracy
Dr inż. Robert Wójcik, K30W04ND03

Słowa kluczowe: systemy webowe, język Go, równoważenie obciążenia, ocena wydajności

WROCŁAW, 2022

Streszczenie

Niniejsza praca zawiera ocenę wydajnościowo-obciążeniową oraz analizę porównawczą skalowalnych systemów webowych typu komunikator zaimplementowanych w języku Go. Rozpatrywane były różne architektury systemów ze skalowalnością poziomą (zmienna liczba serwerów aplikacji) oraz skalowalnością pionową (zmienna liczba rdzeni CPU serwerów aplikacji). W ramach analizy literaturowej przedstawiono metody oraz problemy związane ze skalowalnością i równoważeniem obciążenia systemów webowych. Aspekt inżynierski pracy obejmuje projekt i implementację wybranych wariantów architektonicznych systemu webowego ze skalowalnością, realizującego operacje przetwarzania wiadomości typu CRUD przez aplikację typu komunikator, a także opracowanie i skonfigurowanie środowiska wykorzystywanego do testów wydajnościowych i obciążeniowych systemu. Przeprowadzone badania polegały na porównaniu wydajności operacji – średnich czasów odpowiedzi aplikacji serwerowych na zapytania generowane przez program JMeter przez zadany okres, dla każdego z analizowanych wariantów architektonicznych systemu ze skalowalnością (uwzględnianie zmiennej liczby rdzeni CPU serwerów, zmiennej liczby instancji serwerów aplikacji, a także mechanizmu równoważenia obciążenia typu Least Connections). Testy polegały na ocenie wydajności działania i stabilności systemów przy różnych obciążeniach. Otrzymane wyniki przedstawiono w postaci tabelarycznej oraz na wykresach, a następnie zinterpretowano i poddano analizie w celu określenia ich istotności statystycznej. Na podstawie uzyskanych rezultatów oraz przeprowadzonej analizy stanu badań w dziedzinie problemu, zaprezentowano wnioski dotyczące oceny wydajności i użyteczności opracowanych wariantów systemu webowego typu komunikator ze skalowalnością.

Abstract

This thesis provides a performance and load evaluation and comparative analysis of scalable instant messaging web systems implemented in Go. Different system architectures with horizontal scalability (variable number of application servers) and vertical scalability (variable number of application server CPU cores) were considered. The literature analysis presents methods and problems related to web-based systems' scalability and load balancing. The engineering aspect of the work includes designing and implementing selected architectural variants of a web system with scalability, implementing CRUD-type message processing operations by a messenger-type application, and the development and configuration of the environment used for performance and load testing of the system. The tests carried out consisted in comparing the performance of operations - the average response times of server applications to queries generated by the JMeter program for a given period, for each of the analyzed architectural variants of the system with scalability (taking into account the variable number of server CPU cores, the variable number of application server instances, as well as the Least Connections type load balancing mechanism). The tests evaluated the performance of the operation and stability of the systems under different loads. The obtained results were presented in tabular form and on graphs and then interpreted and analyzed to determine their statistical significance. Based on the results obtained and the analysis of the state of research in the problem field, the conclusions on evaluating the performance and usability of the developed variants of the messenger- type web system with scalability were presented.

Spis treści

Spis rysunków

Spis tabel

Spis listingów

API (ang. *Application Programming Interface*) – interfejs programowania aplikcji.

BSON (ang. *Binary JavaScript Object Notation*) – binarnA postać formatu JSON.

JSON (ang. *JavaScript Object Notation*) – tekstowy, liegki format wykorzystujący się do wymiany danych, którego podstawą jest Java Scrypt.

HTTP (ang. *HyperText Transfer Protocol*) – protokół do wymiany danymi (najczęściej kolejności hipertekstu).

JWT (ang. *JSON Web Token*) – otwarty standard tworzenia tokenów dostępu. Jest oparty na formacie JSON.

NoSQL (ang. *Not only SQL*) – szereg podejść, których celem jest tworzenie systemów zarządzania bazami danych, które mają dużą różnicę w porównaniu do modeli tradycyjnych relacyjnych baz danych.

Rest (ang. *Representational state transfer*) – styl architektury oprogramowania przeznaczony do systemów rozproszonych. Zwykle używany do budowy usług internetowych.

SQL (ang. *Structured Query Language*) – język programowania strukturalnych zapytań. Najczęściej używa się go do skutecznego zapisywania danych, wyszukiwania, zmiany, pobierania oraz usuwania danych z bazy.

URL (ang. *Uniform Resource Locator*) – standardem zapisu linków do obiektów w Internecie.

UUID (ang. *universally unique identifier*) – standard identyfikacji stosowany w tworzeniu oprogramowania.

YAML (ang. *Yet Another Markup Language*) – język znaczników.

CRUD – skrót od Create, Read, Update oraz Delete. Odnosi się do podstawowych funkcji manipulowania danymi w systemie zarządzania Bazą Danych lub RESTful API. Cztery operacje CRUD służą do zarządzania danymi poprzez tworzenie nowych rekordów, pobieranie istniejących rekordów, aktualizowanie istniejących rekordów i usuwanie.

Rozdział 1

Wstęp

1.1. Cel pracy

Celem pracy dyplomowej jest ocena efektywności działania i analiza porównawcza zaprojektowanych wariantów architektonicznych skalowalnych systemów webowych typu komunikator zintegrowanych z bazą danych. W ramach badań uwzględniany będzie wpływ różnych rodzajów skalowalności (pozioma, pionowa) strony serwerowej (back-endu), zastosowanego mechanizmu load balancingu oraz zmiennego obciążenia aplikacji serwerowej zapytaniami od użytkowników (ocena wydajnościowo / obciążeniowa) na wydajność przetwarzania danych. Przeprowadzona zostanie analiza wybranych parametrów aplikacji webowych, np. czasu odpowiedzi, poziomu wykorzystania zasobów serwerów aplikacji, liczby odrzucanych zapytań w zależności od architektury systemu, rodzaju operacji CRUD [34], a także obciążenia systemu zapytaniami użytkowników i zastosowanego mechanizmu load balancingu.

1.1.1. Aspekt inżynierski

Aspekt inżynierski obejmuje: zaprojektowanie oraz realizację wariantów architektonicznych skalowalnego systemu webowego z uwzględnieniem mechanizmu load balancingu oraz różnych rodzajów skalowalności (pozioma, pionowa); implementację elementów systemu: części backendowej wraz z operacjami przetwarzania danych typu CRUD (proste, złożone) z wykorzystaniem języka Go, części bazodanowej z użyciem wybranych systemów baz danych (np. PostgreSQL [16], Redis [11]), oraz mechanizmu load balancingu i skalowalności (np. narzędzie NGINX [32]); weryfikację poprawności działania aplikacji (testy funkcjonalne, np. manualne, automatyczne); przeprowadzenie testów wydajnościowo / obciążeniowych opracowanych wariantów i konfiguracji systemu z wykorzystaniem odpowiednich narzędzi, realizujących operacje frontendu (np. Postman [21], JMeter [8]); graficzne monitorowanie stanu systemu (np. Grafana [19]).

1.1.2. Aspekt badawczy

Aspekt badawczy obejmuję: ocenę efektywności działania skalowalnych aplikacji webowych typu komunikator o wybranej architekturze zrealizowanych z wykorzystaniem technologii języka Go oraz zintegrowanych z bazą danych; analizę wpływu skalowalności poziomej systemu (liczba serwerów aplikacji) oraz skalowalności pionowej (np. pamięć RAM, procesor) na wydajność operacji przetwarzania danych po stronie back-endu; uwzględnienie w ramach badań zapytań CRUD, zmiennego obciążenia serwera (serwerów) aplikacji zapytaniami użytkowników (ocena wydajnościowo / obciążeniowa), a także zastosowanej architektury systemu; wykorzystanie do przeprowadzenie badań wydajnościowych opracowanych wariantów systemu z

wykorzystaniem odpowiednich narzędzi, zastępujących aplikację front-endową w generowaniu zapytań (np. Postman, JMeter); analizę i ocenę parametrów komunikacji z aplikacją webową, np. czasu odpowiedzi, liczby odrzucanych transakcji w zależności od architektury systemu webowego oraz obciążenia zapytaniami użytkowników; ocenę statystyczną, interpretację, analizę porównawcza opracowanych wariantów systemów webowych.

1.2. Zakres i układ pracy

Zakres pracy
Zakres pracy dyplomowej obejmuje: wstęp; sformułowanie problemu oceny efektywności działania i analizy porównawczej rozwiązań architektonicznych skalowalnych systemów webowych typu komunikator zintegrowanych z bazą danych; analizę literaturową stanu badań; sformułowanie wymagań funkcjonalnych i niefunkcjonalnych aplikacji; projekt oraz implementację z wykorzystaniem technologii języka Go wariantów systemów webowych z uwzględnieniem różnych rodzajów skalowalności (pozioma, pionowa) strony serwerowej (back-endu) oraz zastosowanego mechanizmu load balancingu, a także wybranych operacji przetwarzania danych (CRUD); testy poprawności działania systemów; konfigurowanie środowiska badawczego; realizację części badawczej w postaci testów wydajnościowo/obciążeniowych opracowanych wariantów systemu; przeprowadzenie analizy i oceny wybranych parametrów skalowalnych aplikacji webowych, np. czasu odpowiedzi, poziomu wykorzystania zasobów serwerów aplikacji, liczby odrzucanych zapytań w zależności od architektury systemu (mechanizmu skalowalności), rodzaju operacji CRUD, a także obciążenia systemu zapytaniami użytkowników i zastosowanego mechanizmu load balancingu; interpretację i analizę porównawczą wyników z uwzględnieniem wybranych kryteriów oceny; prezentację wniosków z badań oraz ich odniesienie do wyników przedstawionych w literaturze; podsumowanie pracy (zakończenie) oraz prezentację literatury (bibliografia).

Układ pracy
We wstępie zawarto cel pracy oraz opis aspektu badawczego oraz inżynierskiego.

Rozdział drugi został poświęcony przeglądowi literatury. Zostały przedstawione problemy, które powstają podczas konstruowania skalowalnych systemów webowych oraz metody ich rozwiązywania. Ponadto, w rozdziale zawarto analizę stanu badań oraz opis sposobów oceny wydajności systemów webowych.

W rozdziale trzecim przedstawiono analizę wymagań aplikacji webowej typu komunikator. Wymagania te będą dotyczyły systemu webowego realizowanego w ramach części inżynierskiej pracy i zostaną następnie wykorzystane do oceny wydajności analizowanych skalowalnych wariantów architektonicznych systemu. Rozdział ten zawiera opis narzędzi i technologie wykorzystywanych do ich projektowania, a także architektury systemu spełniającej opisane wymagania oraz kryteriów, według których zostaną porównywane zaimplementowane konfiguracje systemu.

Czwarty rozdział obejmuje projekt i implementację części backendowej systemu. Opisano w nim modele danych użyte w części serwerowej i bazach danych oraz najbardziej istotne elementy badań części serwerowej zaprojektowanej w języku Go. Ponadto opisano konfiguracje mechanizmów zarządzania obciążeniem.

Rozdział piąty dotyczy testowania poprawności zaimplementowanego systemu.

W rozdziale szóstym przedstawiono sposób prowadzenia testów wydajnościowo-obciążeniowych oraz sposób monitorowania zasobów i wyników testów. Dodatkowo tabelarycznie i graficznie zaprezentowano wyniki badań, a także przedstawiono ich analizę i interpretację.

Ostatni rozdział zawiera podsumowania pracy oraz wnioski końcowe.

Rozdział 2

Sformułowanie problemu i analiza literaturowa

W tym rozdziale opisano metody skalowania aplikacji webowych, najczęściej używane mechanizmu równoważenia obciążenia oraz metody oceniania wydajności systemu webowego.

2.1. Sformułowanie problemu

Gwałtowny wzrost popularności smartfonów, tabletów i komputerów osobistych prowadzi do rozwoju sieci webowej, a liczba użytkowników usług internetowych rośnie. Zwiększone obciążenie powoduje zauważalny wzrost czasu reakcji użytkowników. Oczekuje się, że aplikacja internetowa będzie stale dostępna, aby osiągnąć te cele, aplikacja internetowa musi być obsługiwana przez niezawodną, dynamiczną i elastyczną usługę internetową. Dla serwisu jest bardzo ważne aby jak najszybciej dostarczyć dane do użytkownika, ponieważ od tego zależy doświadczenie klienta i w konsekwencji zysk firmy. Uważa się, za dobry czas odpowiedzi czas mniejszy niż 200 milisekund, jako dopuszczalny poniżej 1 sekundy i niedopuszczalny powyżej [5].

2.1.1. Problem konstrukcji skalowalnego systemu webowego

Skalowalność opisuje zdolność systemu do dostosowywania się do zmian i wymagań. Dobra skalowalność chroni przed przyszłymi przestojami i gwarantuje jakość usług. Sukces strony, usługi lub aplikacji internetowej zależy od ilości otrzymywanego ruchu sieciowego. Im więcej użytkowników korzysta z serwisu tym lepiej. Istnieje kilka podejść do skalowania: pionowe i poziome. Oba podejścia zakładają dodanie zasobów do infrastruktury obliczeniowej systemu [4, 3].

Skalowanie pionowe oznacza zwiększenie mocy obliczeniowej pojedynczego węzła w istniejącej infrastrukturze poprzez zwiększenie parametrów takich jak procesor, pamięć RAM lub miejsce na dysku.

Skalowanie poziome oznacza dodawanie większej liczby komputerów do puli zasobów, a nie tylko dodawanie zasobów poprzez skalowanie pionowe. Jednak system powinien być odpowiednio skonstruowany aby możliwa była wymiana danych między węzłami oraz obsługa dodatkowego obciążenia infrastruktury, na przykład z powodu dodatkowej transmisji danych między węzłami.

Innym poważnym problemem w aplikacjach internetowych jest niezawodność. Obliczenia odporne na awarie oznaczają, że zadanie może być kontynuowane, aby wykonać pracę w przypadku awarii sprzętu lub oprogramowania. Aby aplikacje internetowe były dostępne, one muszą być obsługiwane w niezawodny, dynamiczny i elastyczny sposób. Jest to możliwe, jeśli system

jest zaprojektowany z uwzględnieniem możliwości użycia więcej niż jednego serwera do obsługi żądań od użytkowników [26].

2.1.2. Problem równoważenia obciążenia aplikacji serwerowych

Równoważenie obciążenia (load balancing) to metoda dystrybucji zadań między wieloma urządzeniami (węzłami) sieciowymi w celu optymalizacji wykorzystania zasobów, skrócenia czasu obsługi żądań, skalowania poziomego systemu i zapewnienia jego odporności na awarie [6].

Load balancer przekieruje żądania klientów na serwery zdolne do realizacji tych żądań w sposób, który maksymalizuje szybkość i wykorzystanie pojemności serwisu oraz zapewnia, że serwery nie zatykają się, co może obniżyć wydajność systemu oraz nie są bezczynne [2]. Load balancing pomaga rozwiązać następujące problemy: skraca czas przestoju, zwiększa zdolność systemu do skalowalności, redundancji, elastyczności oraz potencjalnie zwiększa wydajność.

Jednak architektura systemu powinna być skonstruowana w odpowiedni sposób, aby zapewnić nie przerywalność działań użytkowników. Przykładami rozwiązań tego problemu może być bezstanowy serwer (Rest [35]), gdzie informacja o sesji jest przechowywana po stronie klienta i zapytania zawierają całą niezbędną informację do działania. W przypadku przypiętych sesji, dane mogą być udostępnione poprzez pamięć zewnętrzną (np. Redis [hit]) lub wysyłane od tego samego klienta na ten sam serwer.

Równoważenie obciążenia może zachodzić na różnych poziomach modelu OSI/ISO i wykorzystywać różne technologie dla przekazywania danych. Na poziomie sieci loadbalancer wybiera serwer i przekierowuje zapytania użytkownika bezpośrednio do niego (często używane jest równoważenie sieci na podstawie terytorialnej). Na poziome transportu przedziela się serwer z pewnej puli według wybranego algorytmu. Na tym poziomie używa się proksowanie, czyli loadbalancer przekazuje dane serwerowi, otrzymuje od niego odpowiedź i przekazuje do użytkownika. Na poziomie aplikacji może być balansowanie według rodzaju zapytań (na przykład między backendem i frontendem) [9].

Najczęściej używane algorytmy do równoważenia obciążenia.

Round Robin. Najprostszym sposobem na zmniejszenie obciążenia serwera jest wysyłanie każdego żądania naprzemiennie do serwera. Po zakończeniu listy pętla zaczyna się od początku. Zalety podejścia: prostota, taniość, wydajność. Głównym problemem tego podejścia jest irracjonalna alokacja zasobów. Nawet jeśli wszystkie urządzenia mają podobną wydajność, rzeczywiste obciążenie może znacznie się różnić.

Weighted Round Robin. Ten algorytm jest podobny do Round Robin, ale dodatkowo bierze pod uwagę wydajność serwerów za pomocą ich priorytetyzacji. Im wydajniejsze urządzenie, tym więcej żądań będzie przekierowanych do tego węzła.

Least Connections. Istotą tego algorytmu jest to, że każde kolejne żądanie jest wysyłane do serwera z najmniejszą liczbą obsługiwanych połączeń. To rozwiązanie pozwala na odpowiednie rozłożenie obciążenia na serwery o podobnych parametrach.

Sticky Sessions. W tym algorytmie żądania są przydzielane na podstawie adresu IP użytkownika. Sticky Sessions zakłada, że sprawy od jednego klienta będą kierowane do tego samego serwera. Klient zmieni serwer tylko wtedy, gdy poprzednio używany nie jest już dostępny. Algorytm ten najczęściej używa się dla długich sesji.

2.1.3. Ocena wydajności wybranych operacji przetwarzania danych

W aplikacjach webowych użytkownicy wysyłają swoje żądania na serwer, które przetwarza zapytania i zwraca się informacja. Dla aplikacji webowej jest ważne przetwarzać dane od jak największej liczby użytkowników. Liczba użytkowników jest ściśle powiązana z szybkością dzia-

łania serwisu, bo serwis, który nie jest komfortowy do użytkowania nie będzie popularny. To prowadzi do niezbędności obserwacji następujących metryk:

- liczba żądań w jednostce czasu;
- średni czas odpowiedzi;
- liczby błędnych odpowiedzi.

Operacje wywoływane przez użytkowników działają na danych. Dane można stworzyć, odczytać, zmienić lub usunąć (operacje CRUD). Operacje mogą być proste (składać się z jednej operacji) i złożone (więcej niż jedna operacja). Jedno żądanie do serwera może wywoływać sekwencję operacji, które mogą wywoływać obróbkę danych w różnych częściach systemu. Z tego powstaje następna metryka: obciążenie urządzenia (obciążenie procesora i zużycie pamięci RAM). Te metryki pozwalą obserwować jakie części systemu są wąskim gardłem, stabilność systemu i jakość działania bilansowania obciążenia.

2.2. Stan badań

Do tej pory prowadzono sporo badań dotyczących narzędzi i algorytmów równoważenia obciążenia. W tym dla różnych typów zapytań [26]. Istnieje duża ilość informacji dotyczącej wad i zalet skalowania poziomego i pionowego aplikacji webowych. Większość artykułów zgadza się z tym, że trzeba kombinować różne typy skalowalności. Skalowalność pionowa nie potrzebuje zmian w architekturze i jest łatwiejsza w realizacji, jednak jest droższa dla zwiększenia mocy systemu i jest mniej niezawodna. Skalowalność pozioma wymaga odpowiedniej architekturę systemu, która pozwala na rozproszoną obróbkę danych i wymaga równoważenia obciążenia [4, 3, 1]. W analizowanych pracach nie znaleziono informacji dotyczących porównania wpływu wariantów architektonicznych skalowalności poziomej i pionowej oraz użytych zasobów systemu na efektywność jego działania, a w szczególności na wydajność skalowalnych aplikacji webowych typu komunikator skonstruowanych z wykorzystaniem technologii języka Go.

Rozdział 3

Analiza wymagań projektowych

W rozdziale opisano mechanizm działania i elementy komunikatora webowego. Zamieszczono w nim również wynik przeprowadzonej analizy wymagań funkcjonalnych i niefunkcjonalnych dla mającej powstać aplikacji. Zaproponowano ponadto przewidywany schemat architektoniczny komunikatora webowego, kryteria oceny jego działania oraz technologie wybrane do jego realizacji.

3.1. Opis działania systemu webowego – komunikatora internetowego

Komunikator internetowy to system pozwalający na przesyłanie natychmiastowych komunikatów pomiędzy dwoma lub więcej urządzeniami poprzez sieć komputerową [23].

Systemy typu komunikator ułatwiają komunikację między użytkownikami. Mogą być samodzielnymi aplikacjami lub zintegrowane z inną platformą. System może również umożliwiać komunikację w czatach, składających się z wielu użytkowników. W zależności od protokołu komunikatora, architektura systemu może być peer-to-peer lub klient-serwer. Struktura peer-to-peer oznacza bezpośrednie przekazywanie wiadomości od nadawcy do odbiorcy. Architektura klient-serwer stanowi większość. Umożliwia przekazywanie wiadomości od nadawcy do urządzenia komunikacyjnego. Aplikacje do przesyłania wiadomości mogą przechowywać wiadomości w lokalnej pamięci urządzenia lub w chmurze serwera. Wiadomości mogą zawierać różne typy informacji, na przykład tekstową, graficzną, dźwiękową.

Dzięki komunikatorom proces wymiany wiadomościami może odbywać się w czasie rzeczywistym.

Komponenty systemu:

- System identyfikacji (adresacji) klientów. Ten system powinien rozwiązywać problem dostarczania wiadomości do właściwego odbiorcy.
- System dostarczania wiadomości. System przesyłający wiadomości między użytkownikami systemu.
- System rejestrowania wiadomości. Pozwala na przeglądanie historii wiadomości oraz prowadzenie dialogu bez obecności obu użytkowników online w jednej chwili.
- Informacja o użytkownikach. Przegląd informacji o innym użytkowniku pozwala na weryfikację odbiorcy przez nadawcę.
- System ewidencji stanu użytkowników. Obecność tego systemu pozwala an wyświetlanie statusu użytkownika. To może być przydatne podczas komunikacji w czasie rzeczywistym.

3.2. Wymagania funkcjonalne

Wymagania funkcjonale definiują oczekiwany zakres funkcji dostarczanych przez system oraz jego zachowania. Wymagania te mogą być zdefiniowane w języku formalnym oraz naturalnym. W niniejszej pracy wymagania funkcjonalne przedstawiono na diagramie przypadków użycia (patrz rysunek 3.1) oraz opisano tabelarycznie cel ich wdrożenia (patrz tabela 3.1).

Tab. 3.1: Wymagania funkcjonalne

№	Przypadek	Cel
1	Rejestracja	Tworzenie nowego konta (w bazie danych pojawia się nowy użytkownik). To otwiera możliwość logowania się do systemu.
2	Logowanie	Logowanie się użytkownika do własnego konta. Tylko autoryzowany użytkownik może prowadzić działania w aplikacji (nie odnosi się do rejestracji i logowania).
3	Przeglądanie danych użytkownika	Użytkownik przegląda dozwolone dane. To umożliwia weryfikacje odbiorcy przez nadawcę.
4	Tworzenie czatu	Użytkownik tworzy czat dla wymiany wiadomościami z innym użytkownikiem lub grupą użytkowników.
5	Przeglądanie informacji dotyczącej czatu	Wczytanie danych wybranego czatu z bazy danych oraz wyświetlenie ich użytkownikowi.
6	Dodawanie użytkownika do czatu	Użytkownik, który już należy do czatu, dodaje nową osobę do czatu.
7	Opuszczanie czatu	Użytkownik zostaje usunięty z czatu przez dowolną osobę, która znajduje się w czacie.
8	Wysyłanie wiadomości	Użytkownik wysyła wiadomość do wybranego czatu.
9	Odebranie wiadomości	Użytkownik odbiera wiadomość z odpowiedniego czatu.
10	Przeglądanie wiadomości	Przeglądanie wcześniejszych wiadomości z odpowiedniego czatu.

3.2.1. Diagram przypadków użycia

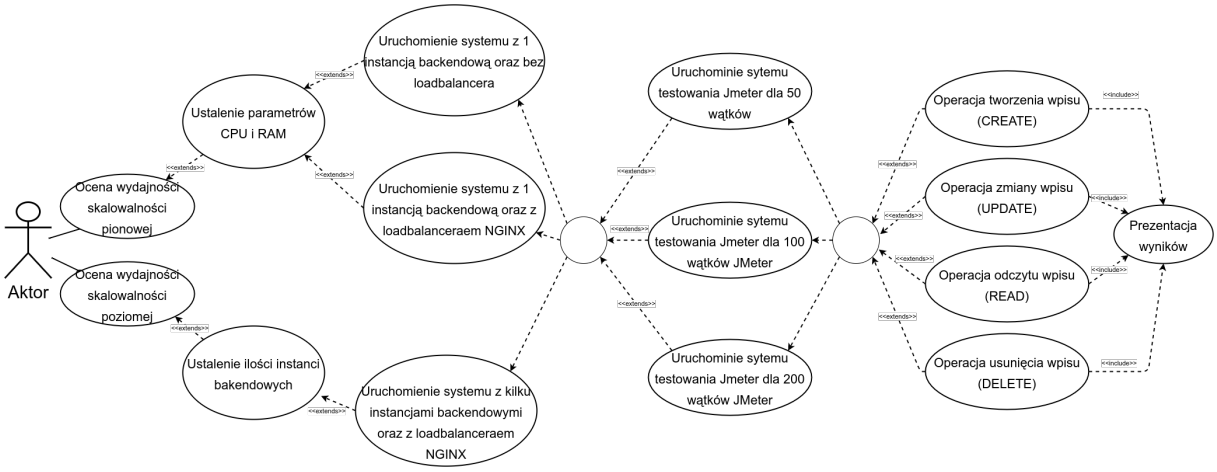
Diagram przypadków użycia (rysunek 3.1) graficznie przedstawia możliwości współdziałania użytkownika z aplikacją typy komunikator, konstruowanej w niniejszej pracy. W aplikacji istnieje tylko jeden typ użytkownika.

3.2.2. Scenariusze przypadków testowych

Scenariusze przypadków testowych stanowią specyfikację przepływu zdarzeń między aktorem a systemem testowym, przeznaczonym do badań wpływu skalowalności na działanie systemu. Na rysunku 3.2 zaprezentowano diagram przypadków testowych dla opracowanego systemu z uwzględnieniem przeznaczenia aplikacji typu komunikator dla zapewnienia potrzeb biznesowych. To oznacza, że żądania do systemu mogą mieć nieporównywalne między sobą złożoności. Testy powinny pokazywać wpływ konfiguracji systemu na jego działanie. Przedstawione przypadki zostały opisane poniżej.

- Przypadek testowy: Ocena wydajności skalowalności pionowej
  Scenariusz główny:
  1. Ustalenie parametrów liczby rdzeni CPU i RAM (na przykład 1 rdzeń CPU i 1Gb RAM)
  2. Uruchomienie aplikacji typu komunikator zawierającej jedną instancję backendową z ustalonymi parametrami CPU i RAM bez narzędzia do równoważenia obciążenia

Rys. 3.1: Diagram przypadków użycia



Rys. 3.2: Diagram przypadków testowych

3. Uruchomienie systemu do testowania wykorzystującego JMeter do przeprowadzenia testów oraz innych narzędzi do gromadzenia, przechowywania i wyświetlania danych otrzymanych w wyniku prowadzenia testów.
4. Operacja tworzenia wpisu w systemie aplikacji typu komunikator.
5. Prezentacja wyników.

Alternatywy dla scenariusza:
W kroku 1: Ustalenie innych parametrów: liczby rdzeni CPU i RAM.
W kroku 3: Uruchomienie systemu aplikacji typu komunikator zawierającej jedną instancję backendową z ustalonymi parametrami CPU i RAM z narzędziem do równoważenia obciążenia NGINX.
W kroku 5: Operacja tworzenia wpisu w systemie aplikacji typu komunikator.
W kroku 5: Operacja edycji wpisu w systemie aplikacji typu komunikator.
W kroku 5: Operacja odczytu wpisu w systemie aplikacji typu komunikator.
W kroku 5: Operacja usunięcia wpisu w systemie aplikacji typu komunikator.

- Przypadek testowy: Ocena wydajności skalowalności poziomej
  Scenariusz główny:
  1. Ustalenie liczby instancji backendowych.

2. Uruchomienie systemu aplikacji typu komunikator zawierającej kilka instancji backendowych z narzędziem do równoważenia obciążenia NGINX.
3. Uruchomienie systemu do testowania zawierającej JMeter do przeprowadzenia testów oraz innych narzędzi dla zboru, przechowywania i wyświetlania danych otrzymanych w wyniku prowadzenia testów.
4. Operacja tworzenia wpisu w systemie aplikacji typu komunikator.
5. Prezentacja wyników

Alternatywy dla scenariusza:
W kroku 1: Ustalenie innych parametrów: liczby instancji backendowych.
W kroku 5: Operacja tworzenia wpisu w systemie aplikacji typu komunikator.
W kroku 5: Operacja edycji wpisu w systemie aplikacji typu komunikator.
W kroku 5: Operacja odczytu wpisu w systemie aplikacji typu komunikator.
W kroku 5: Operacja usunięcia wpisu w systemie aplikacji typu komunikator.

3.3. Wymagania niefunkcjonalne

W tabeli 3.2 przedstawiono wymagania niefunkcjonalne aplikacji: krótki opis wymagania, typ (jakiej dziedziny dotyczy) oraz uwagi do wymagania.

Tab. 3.2: Wymagania niefunkcjonalne

№	typ	Opis	Uwagi
1	Bezpieczeństwo	Każdy użytkownik powinien być autoryzowanym.	Wszystkie działania dozwolone tylko autoryzowanym użytkownikom.
2	Bezpieczeństwo	Hasła powinny być przechowywane w bezpiecznej postaci.	Hasła przechowają się formie haszowanej.
3	Bezpieczeństwo	Zmniejszenie ryzyka otrzymanie dostępu do konta użytkownika przez osoby trzecie.	Autoryzacja użytkownika na serwerze powinna zachodzić za pomocą tokenów.
4	Niezawodność dostarczania wiadomości	Dostarczanie zaległych wiadomości.	Jeśli użytkownik został wylogowany z systemu, po ponownym zalogowaniu wszystkie wiadomości, które były wysłane do niego, muszą być dostarczone.
5	Skierowalność wiadomości	Wiadomości są skierowane od jednego użytkownika do innych konkretnych użytkowników.	
6	Informatywność	System powinien powiadamiać o sukcesach i błędach.	System informuje użytkowników o wyniku jego działania.
7	Wielozadaniowość	Jednoczesny dostęp kilku użytkowników.	Kilku użytkowników może działać jednocześnie, nie oczekując na zakończenie działania innego użytkownika.
8	Dostęp	Użytkownicy nie mają bezpośredniego dostępu do bazy danych.	Wszystkie zapytania muszą być opracowanie przez część serwerową aplikacji.
9	Skalowalność	System powinien mieć zdolność do skalowania.	System może być łatwo skalowalny poziomo oraz pionowo, bez konieczności ponownej kompilacji elementów systemu.
10	Przenoszalność	Wdrożenie systemu powinno być szybkie i łatwe.	System powinien być gotowy do wdrożenia na różnych urządzeniach bądź systemach operacyjnych.
11	Konfigurowalność	Zmiana ustaleń bez konieczności ponownej kompilacji elementów systemu.	Możliwość zmiany ustawień systemu (na przykład adres bazy danych) musi być możliwa za pomocą pliku konfiguracyjnego.

3.3.1. Technologie i narzędzia

Do stworzenia środowiska testowego, mechanizmu równoważenia obciążenia, części serwerowej oraz bazy danych wybrano następujące narzędzia i technologie:

- Visual Studio Code,
- Go,
- JWT,
- REST API,
- HTTP 1.1,
- PostgreSQL,
- Redis,
- Docker,
- Docker compose,
- NGINX,
- JMeter,
- Postman,
- Influxdb,
- Telegraf,
- Grafana.

Visual Studio Code – to edytor kodu o otwartym kodzie źródłowym, wyprodukowany przez firmę Microsoft. Wspiera on różne języki programowania i syntaks dzięki możliwości instalacji rozszerzeń za pomocą mechanizmu plug-inów. Wśród jego funkcji można wymienić: podświetlanie składni, debugowanie, refaktoryzacja i inne [17].

Go (golang) – jest kompilowalnym językiem programowania o statycznej typizacji, stworzonym przez firmę Google. Syntaktycznie jest podobny od języka C, ale zawiera wiele różnic: odśmiecanie pamięci (ang. *garbage collection*), bezpieczeństwo pamięci, system wielowątkowości bazujący na goroutinach. Język jest przeznaczony do budowania serwisów webowych działających efektywnie pod wysokim obciążeniem w środowisku rozproszonym oraz pracy z wielowątkowymi procesorami [12, 18, 31, 28].

REST API (ang. *Representational State Transfer Application Programming Interface*) – podejście stosowane do tworzenia interfejsów programistycznych aplikacji webowych. Interfejs napisany zgodnie z zasadami: komunikacja elementów w architekturze klient-serwer (serwer nasłuchuje na żądania wysyłane przez klienta oraz udziela na nie odpowiedzi), brak stanu (serwer nie przechowuje stanu klienta, wszystkie niezbędne dane są wysyłane wraz z żądaniem), jednorodność interfejsu (stosowanie się do jednego stylu), wielowarstwowość systemu (komponenty systemu mają bezpośredni dostęp tylko do sąsiednich warstw), łatwość rozszerzenia funkcjonalności. [35, 27].

HTTP 1.1 (ang. *Hypertext Transfer Protocol*) – protokół transmisji danych. Wykorzystywany w sieciach komputerowych [20].

JWT (ang. *JSON Web Token*) – standard wykorzystywany do tworzenia tokenów dostępu. JWT pozwalają sprawdzić, czy dane zostały wysłane przez autoryzowane źródło. Token składa się z trzech części: nagłówka (informacja o sposobie szyfrowania tokena), danych (dane w postaci zaszyfrowanej) oraz sygnatury (suma kontrolna niepozwalająca na podmianę JWT). Tokeny mają określony okres ważności. JWT token nie może być oznaczony jako nieważny, dopóki ten okres się nie zakończy [22].

PostgreSQL – otwarta relacyjna baza danych. Podtrzymuje wysokowydajne i niezawodne mechanizmy transakcji i replikacji. Posiada wbudowaną obsługę słabo strukturyzowanych danych typu JSON [16].

Redis – baza danych typu NoSQL, operująca na strukturach typu „klucz-wartość”. Najczęściej używa się jej do implementacji baz danych w pamięci podręcznej, głównie brokerów wiadomości lub wspólnej pamięci do cachowania danych [11].

JMeter jest narzędziem do testowania obciążenia, opracowanym przez Apache Software Foundation. Zostało zrealizowane w języku Java. Wykorzystuje się go do testowania web-aplikacji. Podtrzymuje ono system plug-inów, co pozwala prowadzać różnego rodzaju testy aplikacji webowych (REST API, gRPC, WebSocket, GraphQL, i inne) [8].

Postman jest narzędziem do testowania różnego rodzaju API web-aplikacji [21].

NGINX – darmowe narzędzie webowe pracujące jako wysokowydajny serwer HTTP typu open source i/lub odwrotny proxy serwer i/lub proxy serwer typu IMAP/POP3. Funkcje proksowania pozwalają na równoważenie obciążenia [32].

Docker jest oprogramowaniem przeznaczonym do automatyzacji wdrażania serwisu i jego zarządzania. Aplikacje uruchamiają się i działają w kontenerach. Docker pozwala na upakowanie aplikacji/serwisu oraz wszystkich zależności w niezależny od podstawowego systemu obraz, który może być łatwo przeniesiony na inne urządzenie. Ponadto ułatwia skalowanie i zarządzanie aplikacjami [33, 25].

Telegraf to serwerowy agent napisany w języku Go. Jest przeznaczony do zbierania i wysyłania metryk i zdarzeń z baz danych, systemów i czujników IT [30].

Influxdb – baza danych o otwartym kodzie źródłowym, przechowująca dane szeregowe. Najczęściej używana do pobierania i przechowywania danych szeregowych z obszaru różnego rodzaju metryk, danych czujników oraz analizy danych w czasie rzeczywistym. Napisana w języku Go [29].

Grafana – system wizualizacji danych, który jest zorientowany na dane z monitorowania systemów IT. Pozwala na obserwowanie danych systemu w postaci wykresów i tabel w czasie rzeczywistym oraz zarządzane alertami [19].

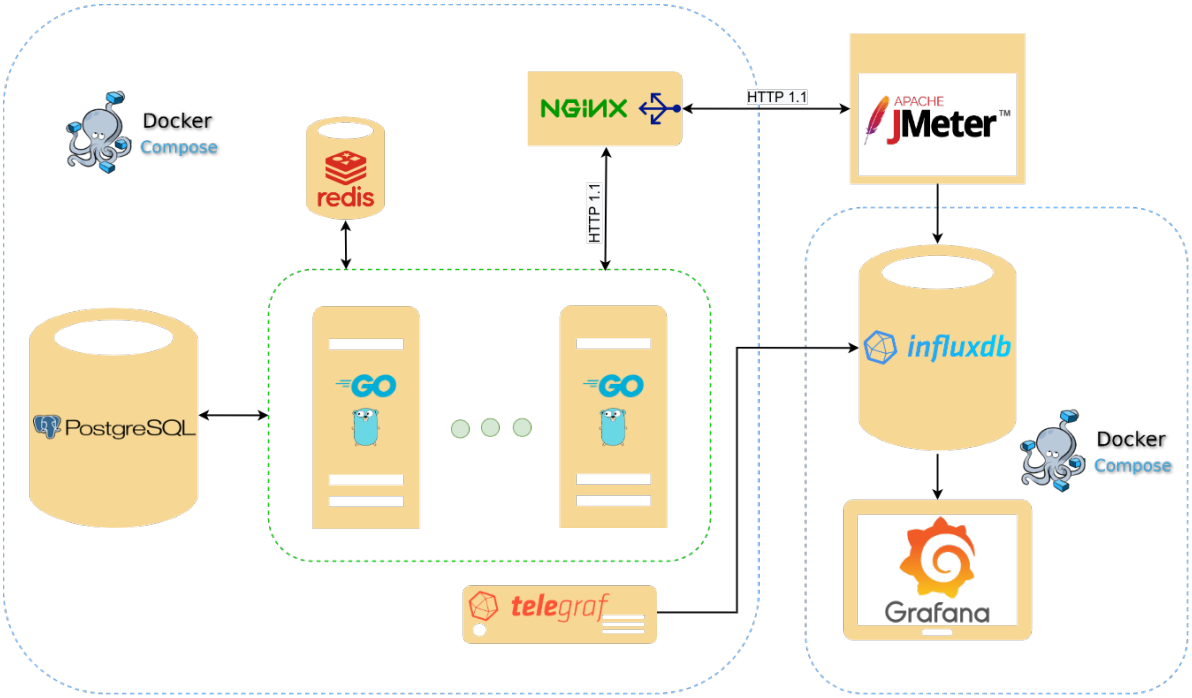
Diagrams.net – otwarta platforma pozwalająca na tworzenie diagramów [10]. Została wykorzystana do stworzenia diagramów w niniejszej pracy.

3.3.2. Architektura systemu uwzględniająca jego skalowalność

W ramach niniejszej pracy testowane mechanizmy skalowalności przeznaczone dla strony serwerowej. W celu symulacji działań użytkowników zostaną użyte narzędzia do testowania obciążeniowego web serwisów.

Na rysunku 3.3 przedstawiono schemat działania systemu. System można zrealizować w architekturze składającej się z następujących elementów:

- PostgreSQL – baza danych do ciągłego przechowywania danych systemu. Przewidywana możliwość skalowania;
- Redis – baza danych, która jest wykorzystywana jako forma pamięci podręcznej do przechowywania JWT tokenów zalogowanych użytkowników;
- Go – skalowalna (poziomo i pionowo) część serwerowa napisana w języku Go. Umożliwia komunikację między interfejsem użytkownika a bazą danych. Przeznaczona do autentykacji użytkowników, dostarczania wiadomości oraz obliczeń biznesowych;
- NGINX – jedno narzędzi do równoważenia obciążenia zapytań skierowanych do części serwerowej. Używa mechanizmu proksowania zapytań.
- JMeter – narzędzie do testowania obciążeniowego web serwisów. Wykonuję rolę interfejsu użytkownika;
- Docker Compose – kontenery do wdrażania elementów systemu;
- Telegraf – agent serwerowy do zbierania i wysyłania metryk (CPU, RAM) kontenerów w systemie;
- Influxdb – baza danych do przechowywania metryk;
- Grafana – wizualizacja metryk zebranych w Unfluxdb z Telegraf i JMeter.



Rys. 3.3: Zarys architektury systemu

3.3.3. Wykorzystywane mechanizmy zarządzania obciążeniem

W ramach niniejszej pracy w celu zmniejszenia wpływu narzędzi na wyniki zostało wybranych kilka narzędzi dla równoważenia obciążenia między serwerami backendowymi. W przypadku pracy jednej instancji przetwarzającej żądanie loadbalancer nie jest wymagany, jednak w ramach pracy są rozpatrywane warianty z nim i bez. NGINX pozwala równoważyć obciążenie między serwerami, używając mechanizmu odwrotnego proxy serwera. To podejście działa na poziomie transportu warstwy ISO/OSI, t.j żądanie od klienta trafia do proxy serwera gdzie jest przekierowywane się do serwera do analizy. Opracowana odpowiedź trafia do proxy serwera, a następnie wysyłana jest do klienta. Algorytmem wybranym dla danego systemu został Least Connections, który uwzględnia istniejące obciążenia serwerów poprzez liczbę aktywnych połączeń i przekieruje żądania do najmniej obciążonego.

3.3.4. Kryteria oceny wydajnościowej i obciążeniowej

Głównymi kryteriami oceny wydajności zaimplementowanych, skalowalnych wariantów architektonicznych systemu typu komunikator są czas odpowiedzi oraz procent błędnych odpowiedzi. Dodatkowym kryterium jest liczba odpowiedzi na sekundę. Porównując czasy odpowiedzi uzyskane dla wybranych konfiguracji systemów i procent błędów, można określić która implementacja jest lepsza.

Pomiar zużycia zasobów CPU i pamięci RAM pozwoli porównać efektywność zużycia zasobów dla różnych konfiguracji systemów oraz znaleźć miejsca i parametry istotne z punktu widzenia efektywności działania systemu.

Rozdział 4

Projekt i implementacja komunikatora internetowego

W niniejszym rozdziale zaprezentowano opis struktur danych znajdujących się w bazach danych. A także interfejsy programistyczne, modele danych oraz zasady działania części serwerowej. Przedstawiono operacje części serwerowej, które podlegają testowaniu w niniejszej pracy oraz najważniejsze funkcje tych operacji. Opisano również konfiguracje narzędzi wykorzystywanych do równoważenia obciążenia między backendami.

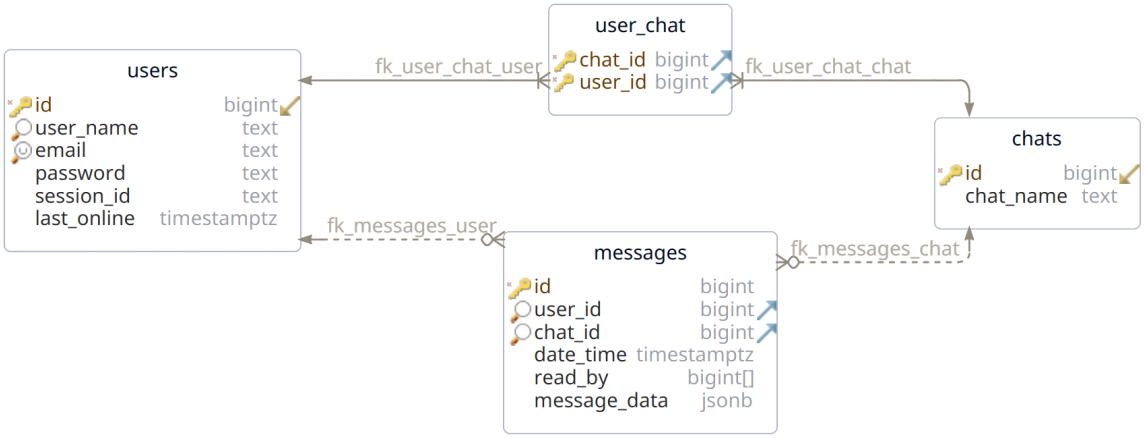
4.1. System bazodanowy

System bazodanowy w części backendowej jest zintegrowany z systemami PostgreSQL i Redis. Relacyjna baza danych PostgreSQL jest przeznaczona do przechowywania danych stałych. Nierelacyja baza danych Redis wykorzystywania dla przechowywania tokenów JWT.

4.1.1. Baza danych PostgreSQL

4.1.1.1. Model fizyczny bazy danych

W aplikacji stworzono trzy encje zapewniające potrzeby biznesowe oraz jedną pomocniczą dla stworzenia relacji many-to-many. Do pierwszych odnoszą się: `users`, `chats`, `messages`. Encja `user_chat` jest pomocniczą. Na rysunku 4.1 przedstawiono model fizyczny bazy danych.



Rys. 4.1: Model fizyczny bazy danych PostgreSQL

Encja `users` jest przeznaczona do przechowywania danych użytkowników. Opis kolumn encji znajduje się w tabeli 4.1.

Tab. 4.1: tabela users

kolumna	Opis	Właściwości
id	id wpisu	big integer, primary key
user_name	nazwa użytkownika	text, NOT NULL, index
email	e-mail użytkownika	text, unique, index
passwor	hasło do konta użytkownika	text, NOT NULL
session_id	numer sesji użytkownika	text
last_online	czas ostatniego on-line	timestamptz

Encja `chats` jest przeznaczona do przechowywania danych chatu. Opis kolumn encji znajduje się w tabeli 4.2.

Tab. 4.2: Tabela chats

kolumna	Opis	Właściwości
id	id wpisu	big integer, primary key
chat_name	nazwa czatu	text, NOT NULL

Encja `user_chat` jest przeznaczona do stworzenia relacji many-to-many. Użytkownik może znajdować się w kilku czatach jednocześnie, natomiast w czacie może znajdować się kilka użytkowników. Opis kolumn encji znajduje się w tabeli 4.3.

Tab. 4.3: Tabela user_chat

kolumna	Opis	Właściwości
chat_id	id czatu, z którym powiązany wpis	big integer, primary key, foreign key do tabeli `chats` pola `id`
user_id	id użytkownika, z którym powiązany wpis	big integer, primary key, foreign key do tabeli `users` pola `id`

Encja `messages` jest przeznaczona dla przechowywania danych wiadomości wysłanej od użytkownika do chatu. Opis kolumn encji znajduje się w tabeli 4.4.

Tab. 4.4: Tabela messages

kolumna	Opis	Właściwości
`id`	id wpisu	big integer, primary key
user_id	id użytkownika, który wysłał wiadomość	text, NOT NULL, index, foreign key do tabeli `chats` pola `id`, ON UPDATE CASCADE, ON DELETE SET NULL
chat_id	id czatu, do którego wysłana wysłana wiadomość	text, NOT NULL, index , foreign key do tabeli `users` pola `id`, ON UPDATE CASCADE, ON DELETE CASCADE
date_time	czas wysłania wiadomości	timestamptz
read_by	lista indeksów użytkowników, które już przeczytały wiadomość	tablica big integer
message_data	zawartość wiadomości	jsonb

4.1.2. Baza danych Redis

Baza danych Redis jest wykorzystana tylko do przechowywania aktywnych tokenów użytkowników, ponieważ jedną z wad JWT tokenów jest to, że wygenerowany token nie można określić jako niedziałający, dopóki nie skończy się określony czas jego działania. Trzymający dane w pamięci podręcznej system bazodanowy Redis eliminuje ten problem. Po zalogowaniu użytkownika zostaje stworzony odpowiedni wpis. Dane przechowują się jako para „klucz - wartość", przez pewien czas. Po upływie tego czasu zapis automatycznie jest usuwany. Dla szybkiego wyszukiwania encja (rekord) wygląda w następujący sposób: `user_id`, `JWTtoken`.

4.2. Realizacja części serwerowej aplikacji z użyciem technologii języka Go

4.2.1. Struktura części backendowej

4.2.1.1. Wykorzystywane biblioteki

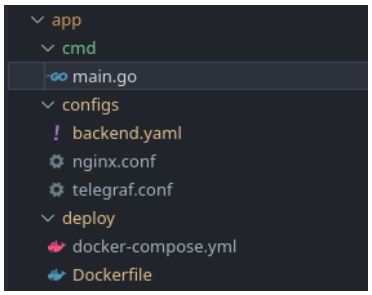
Do stworzenia serwerowej części aplikacji typu komunikator użyto następujących technologii:

- gorilla/mux.v1 - implementacja routera, który mapuje przychodzące żądania na listę zarejestrowanych tras i wywołuje moduł obsługi tego żądania, który odpowiada adresowi URL (ang. *Uniform Resource Locator*);
- gorilla/schema - biblioteka parsująca parametry z linku URL do odpowiedniej struktury;
- stretchr/testify - biblioteka przeznaczona do prowadzenia testów jednostkowych;
- bitcomplete/sqltestutil - biblioteka przeznaczona do prowadzenia testów integracyjnych z bazą danych PostgreSQL;
- sirupsen/logrus - rejestrator strukturalny;
- gorm - system ORM (ang. *Object-Relational Mapping*) dla języka Go;
- gorm.io/driver/postgres - sterownikdo połączenia z bazą danych PostgreSQL;
- go-redis/redis - współpraca z bazą danych Redis z poziomu języka Go;
- dgrijalva/jwt-go - praca z JWT w języku Go;
- go.uber.org/fx - system wstrzykiwania zależności w języku Go;
- go-ozzo/ozzo-validation - pakiet wspomagający walidację danych;
- yaml.v2 - implementuje obsługę YAML (ang. *Yet Another Markup Language*).

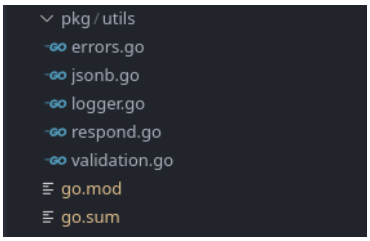
4.2.1.2. Struktura plików

Na rysunkach 4.2 i 4.3 została przedstawiona struktura plików części backendowej niniejszej pracy znajdującej się w katalogu `app`. Ten katalog zawiera pliki z kodem źródłowym, testami jednostkowymi, plikami konfiguracyjnymi oraz plikami przeznaczonymi do kompilacji i uruchomienia (`dockerfile, docker-compose.yaml`).

a)



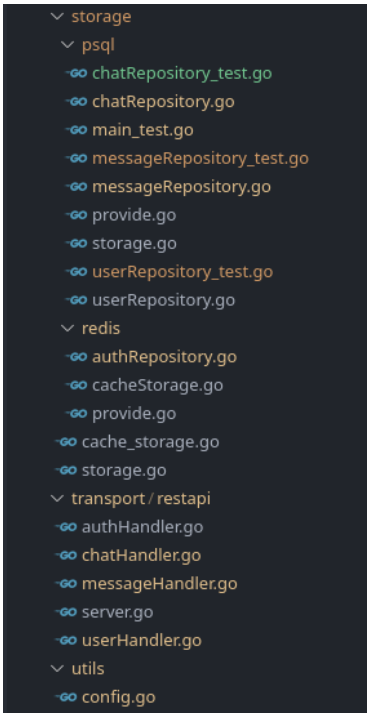
b)



Rys. 4.2: Struktura plików: a) pliki wejściowe, b) katalog pkg

a)

b)



Rys. 4.3: Struktura plików: a) pliki wewnętrzne część 1, b) pliki wewnętrzne część 2

Katalog `cmd` (Rys. 4.2a) zawiera plik `main.go`, który jest punktem wejściowym aplikacji.

W katalogu `configs` (Rys. 4.2a) znajdują się pliki konfiguracyjne systemu. Dla plików wdrożeniowych jest przeznaczony katalog `deploy` (Rys. 4.2a).

W katalogu `internal` (Rys. 4.3a) znajdują się zasoby, które mogą być wykorzystywane wyłącznie przez niniejszą aplikację, czyli nie mogą być użyte jako biblioteka zewnętrza.

Modele danych znajdują się w katalogu `interal/model` (Rys. 4.3a).

Funkcje przechowywane w katalogu `internal/services` (Rys. 4.3) prowadzą biznes logikę (obróbkę danych). Warstwa serwisów komunikuje się z warstwą `DAL` (ang. *Data Access Layer*).

Współpraca z bazą danych zachodzi w warstwie `DAL` znajdującej się w katalogu `interna/storage` (Rys. 4.3b). Katalog `psql` współdziała z PostgreSQL, natomiast `redis` jest przeznaczony do pracy z bazą danych Redis.

Katalog `internal/transport/testapi` (Rys. 4.3b) zawiera funkcje obróbki wchodzących zapytań oraz mapowanie typów tych zapytań na wspomniane funkcje.

Katalog `utils` (Rys. 4.3b) przechowuje wewnętrzne dane pomocnicze.

W katalogu `pkg` (Rys. 4.2b) znajdują się resursy, które mogą być wykorzystywane wyłącznie jako biblioteki zewnętrze dla innych aplikacji, a mianowicie jako dane pomocnicze (katalog `pkg/utils`).

4.2.1.3. Konfigurowanie serwera

Do zapobiegania ponownej kompilacji w przypadku zmiany konfiguracji serwera backendowego, na przykład: portu, na którym działa serwer lub adresów baz danych, została utworzona struktura `Config` 4.2 (plik app/utils/config.go), która przy uruchomieniu aplikacji wczytuje dane z pliku, którego lokalizacja zostanie podana jako parametr wejściowy przy uruchomieniu aplikacji. Plik ten musi być typu `yaml`. Po wczytywaniu pliku (listing 4.1), biblioteka `yaml.v2` parsuje (listing 4.3) zawartość do obiektu struktury `Config` 4.2. Przykład pliku:

Listing 4.1: Przykład pliku konfiguracyjnego części backendowej

```
postgresql:
  master:
    host: psql
    port: 5432
    user: gorm
    password: gorm
    dbname: webmesk
    sslmode:  disable
    timeZone: Europe/Warsaw
    autoMigration: true
  slave:
rest_api_server:
  port: 8081
jwt:
  token_expires: 2592000s
  jwt_key: test_key
redis:
  addr: redis:6379
```

Listing 4.2: Struktura konfiguracyjna części serwerowej

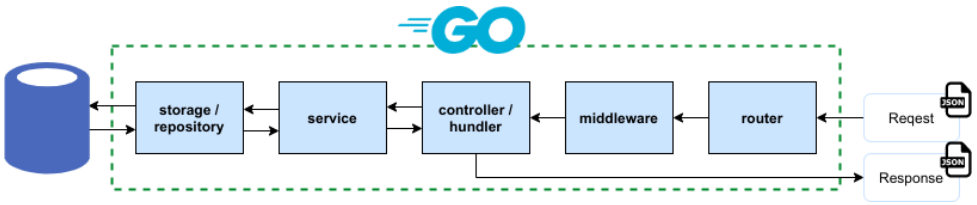
```
type Config struct {
    PostgreSQL    *PostgreSQLConfig    `yaml:"postgresql,omitempty"`
    RestAPIServer *RestAPIServerConfig `yaml:"rest_api_server,omitempty"`
    RedisConfig   *RedisConfig         `yaml:"redis,omitempty"`
    JWTConfig     *JWTConfig           `yaml:"jwt,omitempty"`
}

type RestAPIServerConfig struct {
    Port string `yaml:"port"`
}

type RedisConfig struct {
    Addr string `yaml:"addr"`
}

type JWTConfig struct {
    TknExpires time.Duration `yaml:"token_expires"`
    JWTKey     string        `yaml:"jwt_key"`
}

type PostgreSQLConfig struct {
    Master struct {
            Host          string `yaml:"host"`
            Port          string `yaml:"port"`
            User          string `yaml:"user"`
            DBName        string `yaml:"dbname"`
            Password      string `yaml:"password"`
            SSLMode       string `yaml:"sslmode"`
            TimeZone      string `yaml:"timeZone"`
            AutoMigration bool   `yaml:"autoMigration"`
    } `yaml:"master"`
    Slave []struct {
            Host string `yaml:"host"`
            Port string `yaml:"port"`
    } `yaml:"slave"`
}
```

Listing 4.3: Funkcja wczytująca plik konfiguracyjny części serwerowej

```
func LoadConfig() (*Config, error) {
    configFile, err := ioutil.ReadFile(ConfigPath)
    if err != nil {
            return nil, err
    }
    config := &Config{}
    err = yaml.Unmarshal(configFile, config)
    if err != nil {
            return nil, err
    }
    return config, nil
}
```

4.2.1.4. Przepływ danych

Na rysunku 4.4 został przedstawiony schemat przetwarzania i przepływu danych podczas obróbki żądania przychodzącego do części serwerowej aplikacji.



Rys. 4.4: Przepływ wchodzącego żądania do aplikacji serwerowej

4.2.1.5. Punkty końcowe

Część serwerowa niniejszej aplikacji jest napisana zgodnie ze stylem architektonicznym Rest API. Wymiana danych zachodzi za pomocą następujących standardów: HTTP (ang. *HyperText Transfer Protocol*), URL, JSON. W tabeli 4.5 został przedstawiony spis endpointów (punktów końcowych) razem z metodą ich wysyłania oraz krótkim opisem.

Listing 4.4 przedstawia implementacja punktów końcowych. Do rejestracji danych wchodzących połączeń zostały użyte metody medialne: `pkgutils.SetRequestId` (nadaje id każdemu połączeniu), oraz `logger.LogRequest`, do wypisywania danych tego żądania do konsoli. Większość endpointów przetwarzających dane wymaga aby użytkownik został autoryzowany. Wymaganie to sprawdza się za pomocą metody `authHandler . ValidateToken`. Mechanizm jej działania został wyjaśniony w sekcji 4.2.2.

Listing 4.4: Implementacja punktów końcowych

```
func RegisterHundlers(
    router *mux.Router,
    logger *pkgutils.Logger,
    userHandler *UserHandler,
    messageHandler *MessageHandler,
    chatHandler *ChatHandler,
    authHandler *AuthHandler,
) {
    pfx := "restapi"
    pfx = "/" + pfx
    router.Schemes("http")
    router.Use(pkgutils.SetRequestId)
    router.Use(logger.LogRequest)

    router.HandleFunc(pfx+"/signup", authHandler.SignUp()).Methods("POST")
    router.HandleFunc(pfx+"/signin", authHandler.SignIn()).Methods("PUT")

    authorizedRouter := router.NewRoute().Subrouter()
    authorizedRouter.Use(authHandler.ValidateToken)

    authorizedRouter.HandleFunc(pfx+"/logout", authHandler.Logout()).Methods("GET")

    usrRouter := authorizedRouter.PathPrefix(pfx + "/users").Subrouter()
    usrRouter.HandleFunc("", userHandler.CreateUser()).Methods("POST")
    usrRouter.HandleFunc("/filter", userHandler.FilterUsers()).Methods("GET")
    usrRouter.HandleFunc("/{id:[0-9]+}", userHandler.FindUserById()).Methods("GET")
    usrRouter.HandleFunc("", userHandler.UpdateUserByID()).Methods("PUT")
    usrRouter.HandleFunc("", userHandler.DeleteUserByID()).Methods("DELETE")

    chatRouter := authorizedRouter.PathPrefix(pfx + "/chats").Subrouter()
    chatRouter.HandleFunc("", chatHandler.CreateChat()).Methods("POST")
    chatRouter.HandleFunc("/filter", chatHandler.FilterChats()).Methods("GET")
    chatRouter.HandleFunc("/{id:[0-9]+}", chatHandler.FindChatById()).Methods("GET")
    chatRouter.HandleFunc("/{id:[0-9]+}", chatHandler.UpdateChatByID()).Methods("PUT")
    chatRouter.HandleFunc("/{id:[0-9]+}", chatHandler.DeleteChatByID()).Methods("DELETE")
```

Tab. 4.5: Lista Endpointów części serwerowej

№	Metoda	Endpoint	Opis
1	POST	/restapi/signup	Rejestracja nowego użytkownika.
2	PUT	/restapi/signin	Zalogowanie się użytkownika.
3	GET	/restapi/logout	Wylogowanie się użytkownika.
4	POST	/restapi/users	Tworzenie użytkownika (funkcja dla debugowania).
5	GET	/restapi/users/id	Wczytywanie danych jednego użytkownika według jego id.
6	GET	/restapi/users/filter?skip=""&limit="" &sessionId=""&email=""&user_name=""	Wczytywanie danych lisy użytkowników według parametrów filtracji.
7	PUT	/restapi/users/id	Edycja użytkownika.
8	DELETE	/restapi/users/id	Usuwanie użytkownika.
9	POST	/restapi/chats	Tworzenie nowego czatu dla wymiany wiadomościami między grupą użytkowników.
10	GET	/restapi/chats/id	Wczytywanie danych jednego czatu według jego id.
11	GET	/restapi/chats/filter?skip=""&limit="" &user_id=""&chat_name=""	Wczytywanie danych lisy czatów według parametrów filtracji.
12	PUT	/restapi/chats/id	Edycja czatu.
13	DELETE	/restapi/chats/id	Usuwanie czatu.
14	POST	/restapi/messages	Tworzenie nowego czatu dla wymiany wiadomościami między grupą użytkowników.
15	GET	/restapi/messages/id	Wczytywanie jednej wiadomości według jej id.
16	GET	/restapi/messages/filter?skip=""&limit="" &user_id=""&chat_id=""&date_time=""&date_time_comparation=""&unread_only=""&owner_only	Wczytywanie lisy wiadomości według parametrów filtracji.
17	PUT	/restapi/messages/id	Edycja wiadomości.
18	DELETE	/restapi/messages/id	Usuwanie wiadomości.
19	PUT	/restapi/messages/markasread/id	Oznaczenie wiadomości jako przeczytanej.

```
    messageRouter := authorizedRouter.PathPrefix(pfx + "/messages").Subrouter()
    messageRouter.HandleFunc("", messageHandler.CreateMessage()).Methods("POST")
    messageRouter.HandleFunc("/filter", messageHandler.FilterMessages()).Methods("GET")
    messageRouter.HandleFunc("/{id:[0-9]+}", messageHandler.FindMessageByID()).Methods("
        ↪ GET")
    messageRouter.HandleFunc("/{id:[0-9]+}", messageHandler.UpdateMessageByID()).Methods("
        ↪ PUT")
    messageRouter.HandleFunc("/{id:[0-9]+}", messageHandler.DeleteMessageByID()).Methods("
        ↪ DELETE")
    messageRouter.HandleFunc("/markasread/{id:[0-9]+}", messageHandler.MarkAsRead()).
        ↪ Methods("PUT")
}
```

4.2.1.6. Modele danych

W celu odebrania i wysyłania danych do części klienta oraz współdziałania z bazą danych PostgreSQL zostały zaimplementowane następujące struktury: `Chat` (listing4.5), `User` (listing 4.6), `Message` (listing4.7).

Listing 4.5: Struktura Chat

```
type Chat struct {
    ID         uint     `json:"id,omitempty" gorm:"primaryKey"`
    ChatName   string   `json:"chat_name,omitempty" gorm:"not null"`
    MemberList []*User `json:"member_list,omitempty" gorm:"many2many:user_chat"`
}
```

Listing 4.6: Struktura User

```
type User struct {
    ID         uint      `json:"id,omitempty" gorm:"primaryKey"`
    UserName   string    `json:"user_name,omitempty" gorm:"not null;index"`
    Email      string    `json:"email,omitempty" gorm:"index:,unique"`
    Password   string    `json:"password,omitempty" gorm:"not null"`
    SessionId  string    `json:"sessionId,omitempty"`
    LastOnline time.Time `json:"last_online,omitempty"`
}
```

Listing 4.7: Struktura Message

```
type Message struct {
    ID          uint           `json:"id,omitempty" gorm:"primaryKey"`
    UserID      uint           `json:"user_id,omitempty" gorm:"index"`
    User        *User          `json:"-" gorm:"constraint:OnUpdate:CASCADE,OnDelete:SET
        ↪ NULL;"`
    ChatID      uint           `json:"chat_id,omitempty" gorm:"index"`
    Chat        *Chat          `json:"-" gorm:"constraint:OnUpdate:CASCADE,OnDelete:CASCADE
        ↪ ;"`
    DateTime    time.Time      `json:"date_time,omitempty"`
    ReadBy      pq.Int64Array `json:"read_by,omitempty" gorm:"type:bigint[]"`
    MessageData utils.JSONB    `json:"message_data,omitempty" gorm:"type:jsonb"`
}
```

Endpointy pozwalające na filtrację danych uwzględniają obecność w URL żądania odpowiednich parametrów, nazywanych filtrami: `UserFilter` (listing4.8), `ChatFilter` (listing4.9), `MessageFilter` (listing4.10).

Listing 4.8: Struktura UserFilter

```
type UserFilter struct {
    UserName  string `schema:"user_name,omitempty"`
    Email     string `schema:"email,omitempty"`
    SessionId string `schema:"sessionId,omitempty"`
    Skip      uint   `schema:"skip,omitempty"`
    Limit     uint   `schema:"limit,omitempty"`
}
```

Listing 4.9: Struktura ChatFilter

```
type ChatFilter struct {
    UserID   uint   `schema:"user_id,omitempty"`
    ChatName string `schema:"chat_name,omitempty"`
    Skip     uint   `schema:"skip,omitempty"`
    Limit    uint   `schema:"limit,omitempty"`
}
```

Listing 4.10: Struktura MessageFilter

```
type MessageFilter struct {
    UserID                   uint      `schema:"user_id,omitempty"`
    ChatID                   uint      `schema:"chat_id,omitempty"`
    DateTime                 time.Time `schema:"date_time,omitempty"`
    DateTimeComparationType string    `schema:"date_time_comparation_type,omitempty"`
    UnreadOnly               bool      `schema:"unread_only,omitempty"`
    OwnerOnly                bool      `schema:"owner_only,omitempty"`
    Skip                     uint      `schema:"skip,omitempty"`
    Limit                    uint      `schema:"limit,omitempty"`
}
```

Struktura `UserAuth` (listing4.11) jest używana do przechowywania danych autoryzacji (JWT token) w bazie danych Redis oraz sprawdzenia ważności tokenu.

Listing 4.11: Struktura UserAuth

```
type UserAuth struct {
    ID        string
    SessionId string
    JWTToken  string
}
```

4.2.2. Realizacja operacji przetwarzania danych

4.2.2.1. Uwierzytelnianie użytkowników

Uwierzytelnianie użytkowników jest bardzo ważnym elementem systemu. Każda operacja (oprócz logowania się i rejestracji) wymaga sprawdzenia tokenów autoryzacji oraz uprawnień dostępu do danych. Uwierzytelnienie użytkownika zachodzi w funkcji pośredniczącej `ValidateToken` (listing 4.12) poprzez walidację tokena (funkcja `ValidAuthorization` (listing 4.13)) oraz sprawdzenie jego ważności w bazie danych Redis (funkcja `FindById` zaprezentowana w listingu 4.14).

Listing 4.12: Finkcja pośrednicza przyznaczona do walidacji JWT

```
func (ah *AuthHandler) ValidateToken(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                tokenString := r.Header.Get("Authorization")
                if len(tokenString) == 0 {
                        utils.Error(w, r, http.StatusUnauthorized, errors.New("missing
                            ↪ authorization header"))
                        return
                }
                tokenString = strings.Replace(tokenString, "Bearer ", "", 1)
                ctx, err := ah.service.Auth().ValidAuthorization(r.Context(), tokenString)
                r = r.WithContext(ctx)
                if err != nil {
                        utils.Error(w, r, http.StatusUnauthorized, err)
                        return
                }
                next.ServeHTTP(w, r)
        })
}
```

Listing 4.13: Walidacja JWT tokena

```
func (as *AtuthService) ValidAuthorization(ctx context.Context, jwtToken string) (context.
    ↪ Context, error) {
        claims := &jwt.StandardClaims{}
        tkn, err := jwt.ParseWithClaims(jwtToken, claims, func(token *jwt.Token) (interface{},
            ↪  error) {
                return []byte(as.jwtConfig.JWTKey), nil
        })
        if err != nil {
                return ctx, err
        }
        if !tkn.Valid {
                return ctx, jwt.ErrSignatureInvalid
        }
        redisUser, err := as.authRepository.FindById(ctx, claims.Id)
        if err != nil {
                return ctx, err
        }
        if redisUser.ID == claims.Id && redisUser.JWTToken == jwtToken {
                return context.WithValue(ctx, "user_id", claims.Id), nil
        }
        return ctx, putils.ErrUnauthorized
}
```

Listing 4.14: Pobranie JWT tokenu z bazy danych Redis

```
func (ar *AuthRepository) FindById(ctx context.Context, id string) (*model.UserAuth, error) {
        res, err := ar.storage.redisClient.Get(id).Result()
        if err != nil {
```

```
                return nil, err
        }
        if res == "" {
                return nil, utils.ErrRecordNotFound
        }
        token := &model.UserAuth{}
        if err := token.UnmarshalBinary([]byte(res)); err != nil {
                return nil, err
        }
        return token, nil
}
```

4.2.2.2. Zapisywanie danych do bazy

Część backendowa zaimplementowanego systemu typu komunikator zawiera cztery endpointy przeznaczone do stworzenia nowych wpisów w głównej bazie danych PostgreSQL: `/restapi/signup`, `/restapi/users`, `/restapi/chats`, `/restapi/messages`. Punkty końcowe wymagają wysłania zapytania typu `POST` zawierającego odpowiedni obiekt typu `json`.

Tworzenie wiadomości

Z punktu widzenia biznesowego endpoint `/restapi/messages` jest przeznaczony do wysłania wiadomości od jednego użytkownika do innych użytkowników znajdujących się we wspólnym czacie. Model danych opisano w listingu 4.7. Wchodzące żądanie metodą `POST` w funkcji obróbczej (listing 4.15) najpierw prowadzi autentykację użytkownika (opisano w części 4.2.2.1). W warstwie serwisu (listing 4.16) dodawana jest dodatkowa informacja, na przykład czas tworzenia (funkcja `message.BeforeCreate`), sprawdzane są uprawnienia użytkownika do pisania do danego czatu (funkcja `chat.CheckPermissions`) i usuwane dane wrażliwe (funkcja `message.Sanitize`). W funkcji warstwy DAO (listing 4.17) zachodzi tworzenie jednego wpisu w tabeli `messages`.

Listing 4.15: Obróbka tworzenia wiadomości

```
func (mh *MessageHandler) CreateMessage() http.HandlerFunc {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                message := &model.Message{}
                err := json.NewDecoder(r.Body).Decode(message)
                if err != nil {
                        utils.Error(w, r, http.StatusBadRequest, err)
                        return
                }
                userId, err := strconv.ParseUint(r.Context().Value("user_id").(string), 10,
                    ↪ 64)
                if err != nil {
                        utils.Error(w, r, http.StatusBadRequest, utils.ErrUserNotFound)
                        return
                }
                message.UserID = uint(userId)
                err = message.Validate()
                if err != nil {
                        utils.Error(w, r, http.StatusBadRequest, err)
                        return
                }
                message, err = mh.service.Message().Create(r.Context(), message)
                if err != nil {
                        if err == utils.ErrRecordAlreadyExists {
                                utils.Error(w, r, http.StatusBadRequest, err)
                                return
                        }
                        utils.Error(w, r, http.StatusInternalServerError, err)
                        return
                }
                utils.Respond(w, r, http.StatusOK, message)
        })
}
```

Listing 4.16: Warstwa serwisu tworzenia wiadomości

```
func (ms *MessageService) Create(ctx context.Context, message *model.Message) (*model.Message,
    ↪ error) {
        if err := message.BeforeCreate(); err != nil {
                return message, err
        }
        chat, err := ms.storage.Chat().FindById(ctx, message.ChatID)
        if err != nil {
                return message, err
        }
        if !chat.CheckPermissions(message.UserID) {
                return message, utils.ErrNoPermissions
        }
        message, err = ms.storage.Message().Create(ctx, message)
        if err != nil {
                return message, err
        }
        message.Sanitize()
        return message, nil
}
```

Listing 4.17: Warstwa DAO tworzenia wiadomości

```
func (mr *MessageRepository) Create(ctx context.Context, message *model.Message) (*model.
    ↪ Message, error) {
        res := mr.storage.DB.WithContext(ctx).Create(message)
        if res.RowsAffected != 1 {
                return message, utils.ErrRowsNumberAffected(int(res.RowsAffected))
        }
        return message, nil
}
```

4.2.2.3. Pobieranie danych z bazy

Następujące punkty końcowe przeznaczone są do odczytu danych z głównej bazy danych PostgreSQL: `/restapi/users/id`, `/restapi/chats/id`, `/restapi/messages/id`, `/restapi/users/filter?skip=&limit=&sessionId=&email=&user_name=`, `/restapi/chats/filter?skip=&limit=&user_id=&chat_name=`, `restapi/messages/filter?skip=&limit=&user_id=&chat_id=&date_time=&date_time_comparation=&unread_only=&owner_only=`. Punkty końcowe wymagają dodatkowych danych umieszczonych w URL żądań typu `GET`: `id` wpisów lub danych do filtracji (np. `email`, `date_time`).

Pobranie listy wiadomości

Pobranie listy wiadomości może być użyte na przykład do otrzymania dużej liczby wiadomości filtrowanych według: użytkownika, czatu, czasu napisania (np. napisany do lub po jakimś czasie), jeszcze nie przeczytane według użytkownika, tylko napisane oraz ustalić limity liczby zwróconych wiadomości. Do tego służy endpoint `/restapi/messages/filter?skip=""&limit=""&user_id=""&chat_id=""&date_time=""&date_time_comparation=""&unread_only=""&owner_only=""`.
W tym celu została zaimplementowana odpowiednia struktura filtru (listing 4.10). Ustalenie parametrów filtracji zachodzi za pomocą parametrów zapytania w URL adresie. Parametry umieszczone w zapytaniu od autentykowanego użytkownika dekodowane są za pomocą biblioteki `gorilla/schema` (listing 4.18) do struktury przeznaczonej do filtracji. W warstwie serwisu (listing 4.19) wywołuje się warstwa DAO oraz zachodzi usunięcie danych wrażliwych. W zależności od danych umieszczonych w filtrze jest odpowiednio budowane zapytanie do bazy danych PostgreSQL (listing 4.20). Zapytanie to może tyczyć się tylko tabeli `messages` (w przypadku filtracji po czasie lub statusie odczytu przez użytkownika) lub współpracować z tabelami `chats` i `user_chat` (na przykład w przypadku wyszukiwania wiadomości, które nie należą do użytkownika, który je utworzył).

Listing 4.18: Obróbka pobierania listy wiadomości

```
func (mh *MessageHandler) FilterMessages() http.HandlerFunc {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                filter := &model.MessageFilter{}
                err := decoder.Decode(filter, r.URL.Query())
                if err != nil {
                        utils.Error(w, r, http.StatusBadRequest, err)
                        return
                }
                userId, err := strconv.ParseUint(r.Context().Value("user_id").(string), 10,
                    ↪ 64)
                if err != nil {
                        utils.Error(w, r, http.StatusBadRequest, utils.ErrUserNotFound)
                        return
                }
                filter.UserID = uint(userId)
                messages, err := mh.service.Message().FilterMessage(r.Context(), filter)
                if err != nil {
                        utils.Error(w, r, http.StatusNotFound, err)
                        return
                }
                utils.Respond(w, r, http.StatusOK, messages)
        })
}
```

Listing 4.19: Warstwa serwisu pobierania listy wiadomości

```
func (ms *MessageService) FilterMessage(ctx context.Context, messageFilter *model.
    ↪ MessageFilter) ([]model.Message, error) {
        messages, err := ms.storage.Message().FilterMessage(ctx, messageFilter)
        if err != nil {
                return nil, err
        }
        for i, _ := range messages {
                messages[i].Sanitize()
        }
        return messages, nil
}
```

Listing 4.20: Warstwa DAO pobierania listy wiadomości

```
func (mr *MessageRepository) FilterMessage(ctx context.Context, messageFilter *model.
    ↪ MessageFilter) ([]model.Message, error) {
        query := mr.storage.DB.WithContext(ctx)
        if !messageFilter.DateTime.IsZero() && messageFilter.DateTimeComparationType != "" {
                filter := "date_time " + messageFilter.DateTimeComparationType + " ?"
                query = query.Where(filter, messageFilter.DateTime)
        }
        if messageFilter.UserID != 0 {
                if messageFilter.OwnerOnly {
                        query = query.Where("messages.user_id = ?", messageFilter.UserID)
                } else {
                        query = query.Joins("JOIN chats ON chats.id = messages.chat_id").Joins
                            ↪ ("JOIN user_chat ON chats.id = user_chat.chat_id").Where("
                            ↪ user_chat.user_id = ?", messageFilter.UserID)
                }
        }
        if messageFilter.UnreadOnly && messageFilter.UserID > 0 {
                query = query.Where("NOT (? = ANY(read_by))", messageFilter.UserID)
        }
        if messageFilter.ChatID != 0 {
                query = query.Where("chat_id = ?", messageFilter.ChatID)
        }
        query = query.Offset(int(messageFilter.Skip))
        if messageFilter.Limit != 0 {
                query = query.Limit(int(messageFilter.Limit))
        }
        messages := []model.Message{}
        res := query.Find(&messages)
        return messages, res.Error
}
```

4.2.2.4. Aktualizowanie danych w bazie

Część backendowa zawiera 4 zaimplementowane endpointy do zmiany istniejących wpisów w bazie danych: `/restapi/users/id` , `/restapi/chats/id`, `/restapi/messages/id`, `/restapi/messages/markasread/id`. Endpointy te wymagają wysłania żądania metodą PUT oraz niektóre wymagają podania nowych, aktualnych obiektów w postaci `json`.

Aktualizowanie wiadomości

Aby aktualizować wiadomość, na przykład dla zmiany jej treści, wysyła się PUT żądanie na endpoint `/restapi/messages/id`, gdzie `id` jest identyfikatorem wiadomości. Po autoryzacji użytkownika i pobraniu jego `id` z JWT tokenu (listing 4.21) oraz przeprowadzeniu operacji Update w bazie danych w warstwie serwisu usuwane są dane wrażliwe (listing 4.22). Zapytanie do bazy danych odwołuje się do tabeli `messages` (listing 4.20).

Listing 4.21: Obróbka aktualizacji wiadomości

```
func (mh *MessageHandler) UpdateMessageByID() http.HandlerFunc {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                vars := mux.Vars(r)
                sid, ok := vars["id"]
                if !ok {
                        utils.Error(w, r, http.StatusBadRequest, utils.ErrWrongRequest)
                        return
                }
                id, err := strconv.ParseUint(sid, 10, 64)
                if err != nil {
                        utils.Error(w, r, http.StatusBadRequest, err)
                        return
                }
                message := &model.Message{}
                err = json.NewDecoder(r.Body).Decode(message)
                if err != nil {
                        utils.Error(w, r, http.StatusBadRequest, err)
                        return
                }

                userId, err := strconv.ParseUint(r.Context().Value("user_id").(string), 10,
                    ↪ 64)
                if err != nil {
                        utils.Error(w, r, http.StatusBadRequest, utils.ErrUserNotFound)
                        return
                }
                message.ID = uint(id)
                message.UserID = uint(userId)
                message, err = mh.service.Message().Update(r.Context(), message)
                if err != nil {
                        utils.Error(w, r, http.StatusNotFound, err)
                        return
                }
                utils.Respond(w, r, http.StatusOK, message)
        })
}
```

Listing 4.22: Warstwa serwisu aktualizacji wiadomości

```
func (ms *MessageService) Update(ctx context.Context, message *model.Message) (*model.Message,
    ↪  error) {
        message, err := ms.storage.Message().Update(ctx, message)
        if err != nil {
                return nil, err
        }
        message.Sanitize()
        return message, nil
}
```

Listing 4.23: Warstwa DAO aktualizacji wiadomości

```
func (mr *MessageRepository) Update(ctx context.Context, message *model.Message) (*model.
    ↪ Message, error) {
```

```
        res := mr.storage.DB.WithContext(ctx).Where("id = ? AND user_id = ?", message.ID,
            ↪ message.UserID).Omit("date_time", "read_by", "user_id", "chat_id").Updates(
            ↪ message)
        if res.RowsAffected != 1 {
                return message, utils.ErrRowsNumberAffected(int(res.RowsAffected))
        }
        return message, res.Error
}
```

4.2.2.5. Usuwanie danych z bazy

W celu przeprowadzenia operacji usunięcia wpisów z bazy danych zaimplementowano następujące endpointy (metoda `DELETE`): `/restapi/users/id`, `/restapi/chats/id`, `/restapi/messages/id`, gdzie `id` jest identyfikatorem wpisu w bazie danych.

Usunięcie wiadomości

Endpoint `/restapi/chats/id` jest przeznaczony do usunięcia jednej wiadomości. Sprawdzenie tokenu autoryzacji i uprawnień użytkownika opisano w listingu 4.24. Z poziomu serwisu DAO usuwa się jeden wpis z tabeli `messages` (listing 4.25).

Listing 4.24: Obróbka usunięcia chata

```
func (mh *MessageHandler) DeleteMessageByID() http.HandlerFunc {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                vars := mux.Vars(r)
                sid, ok := vars["id"]
                if !ok {
                        utils.Error(w, r, http.StatusBadRequest, utils.ErrWrongRequest)
                        return
                }
                id, err := strconv.ParseUint(sid, 10, 64)
                if err != nil {
                        utils.Error(w, r, http.StatusBadRequest, err)
                        return
                }

                userId, err := strconv.ParseUint(r.Context().Value("user_id").(string), 10,
                    ↪ 64)
                if err != nil {
                        utils.Error(w, r, http.StatusBadRequest, utils.ErrUserNotFound)
                        return
                }
                message := &model.Message{
                        ID:     uint(id),
                        UserID: uint(userId),
                }

                err = mh.service.Message().Delete(r.Context(), message)
                if err != nil {
                        utils.Error(w, r, http.StatusInternalServerError, err)
                        return
                }
                utils.Respond(w, r, http.StatusOK, nil)
        })
}
```

Listing 4.25: Warstwa DAO usunięcia chata

```
func (mr *MessageRepository) Delete(ctx context.Context, message *model.Message) error {
        res := mr.storage.DB.WithContext(ctx).Delete(&model.Message{ID: message.ID, UserID:
            ↪ message.UserID})
        if res.RowsAffected != 1 {
                return utils.ErrRowsNumberAffected(int(res.RowsAffected))
        }
        return res.Error
}
```

4.3. Konfigurowanie mechanizmu zarządzania obciążeniem – loadbalancerów

Mechanizm równoważenia obciążenia jest przeznaczony do podziału napływu żądań między instancjami aplikacji części backendowej. W przypadku pracy jednej instancji loadbalancer nie jest wymagany. Zostały zrealizowane warianty z loadbalancerem i bez, gdzie żądania idą bezpośrednio do aplikacji backendowej. Do równoważenia obciążenia został użyty serwer NGINX.

W listingu 4.26 umieszczono przykład konfiguracji loadbalancera NGINX. Maksymalna ilość połączeń 10000. Blok `upstream` przeznaczony jest do konfigurowania metody równoważenia obciążenia oraz rejestracji instancji, na które przekierowywać żądania (`server golang-restapi-n`). Blok `server` zawiera ustalenia proxy serwera: port, na którym nasłuchuje NGINX oraz sposób dostarczana danych do aplikacji. Do równoważenia obciążenia użyta została metoda Least Connections.

Listing 4.26: Konfiguracja serwera NGINX

```
events {
    worker_connections   10000;
}
http {
    upstream api_servers {
        least_conn;
        server golang-restapi-1:8081;
        server golang-restapi-2:8081;
        server golang-restapi-3:8081;
        server golang-restapi-4:8081;
    }
    server {
        listen 8081;
        access_log off;
        location / {
            proxy_pass http://api_servers;
        }
    }
}
```

Rozdział 5

Testowanie aplikacji

Niniejszy rozdział zawiera opis testów sprawdzających poprawność działania zaimplementowanej części backendowej aplikacji typu komunikator: testy jednostkowe i funkcjonalne API.

5.1. Testowanie poprawności działania aplikacji

Testy poprawności systemu obejmują testy jednostkowe kodu aplikacji backendowej typu komunikator oraz testy sprawdzające działanie API poprzez wysyłanie odpowiednich żądań do zaimplementowanego systemu.

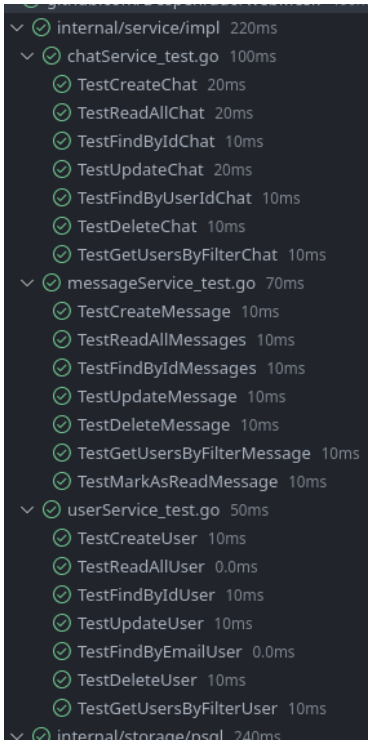
5.1.1. Testy jednostkowe opracowanych wariantów skalowalnej aplikacji webowej

Pliki o postaci `*_test.go`, które znajdują się obok plików z kodem źródłowym (Rys. 4.2 4.3) zawierają testy jednostkowe. Do testowania operacji wymagających obecność bazy danych jest wykorzystywana biblioteka `bitcomplete/sqltestutil`. Jej działanie polega na stworzeniu i sterowaniu specjalnym Docker kontenerem, zawierającym instancję bazy danych PostgreSQL, z którą łączy się aplikacja podczas prowadzenia testów. Przykład takiego testu umieszczono w listingu 5.1. W wymienionym przykładzie tworzą się nowe wiadomości w bazie danych, wyniki porównywane są z wartościami oczekiwanymi. Po zakończeniu testu wszystkie dane usuwane są z bazy danych. Lista wykonanych testów jednostkowych znajduje się na rysunkach 5.1a,b. Testy te zostały wykonane dla warstwy serwisów oraz warstwy DAO.

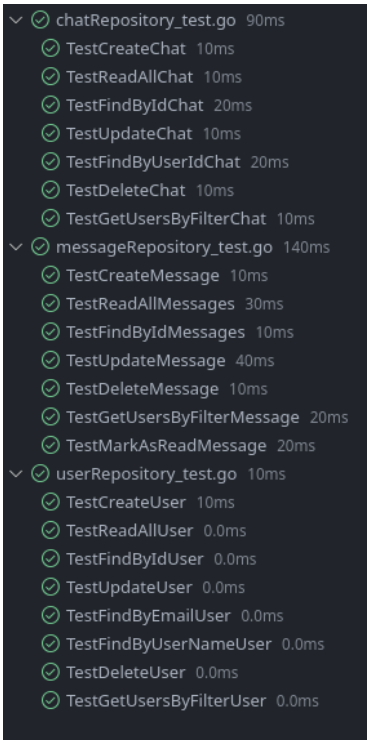
Listing 5.1: Przykład testu jednostkowego: tworzenie czatu.

```
func TestCreateMessage(t *testing.T) {
        defer func() {
                storageInt.DB.Exec("DELETE FROM user_chat")
                storageInt.DB.Exec("DELETE FROM chats")
                storageInt.DB.Exec("DELETE FROM users")
                storageInt.DB.Exec("DELETE FROM messages")
        }()
        mr := serviceInt.Message()
        testsData := prepareDataMessage(t)

        for _, testData := range testsData {
                msg, err := mr.Create(context.TODO(), testData.actual)
                assert.Nil(t, err)
                assert.NotNil(t, msg)
                msg.DateTime = time.Time{}
                testData.expected.ID = msg.ID
                assert.Equal(t, testData.expected, msg)
        }
}
```

a)



b)



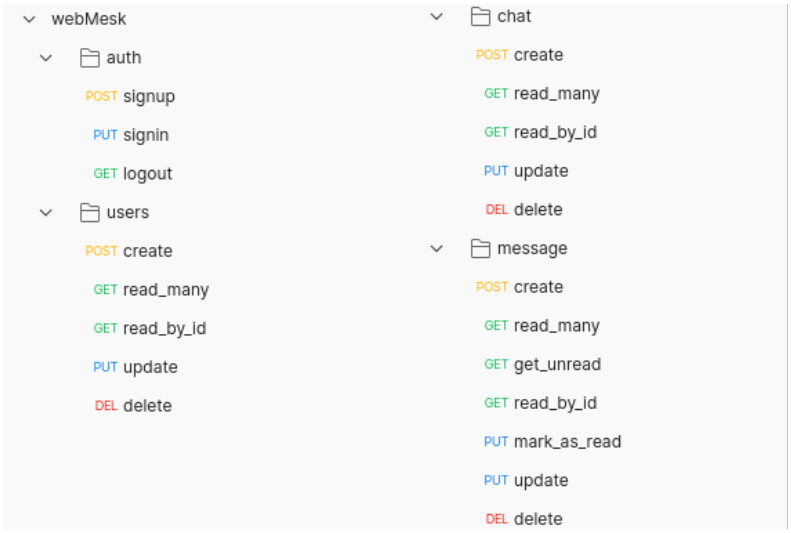
Rys. 5.1: Testy jednostkowe: a) warstwa serwisu, b) warstwa DAO

5.1.2. Testy opracowanych funkcjonalności API

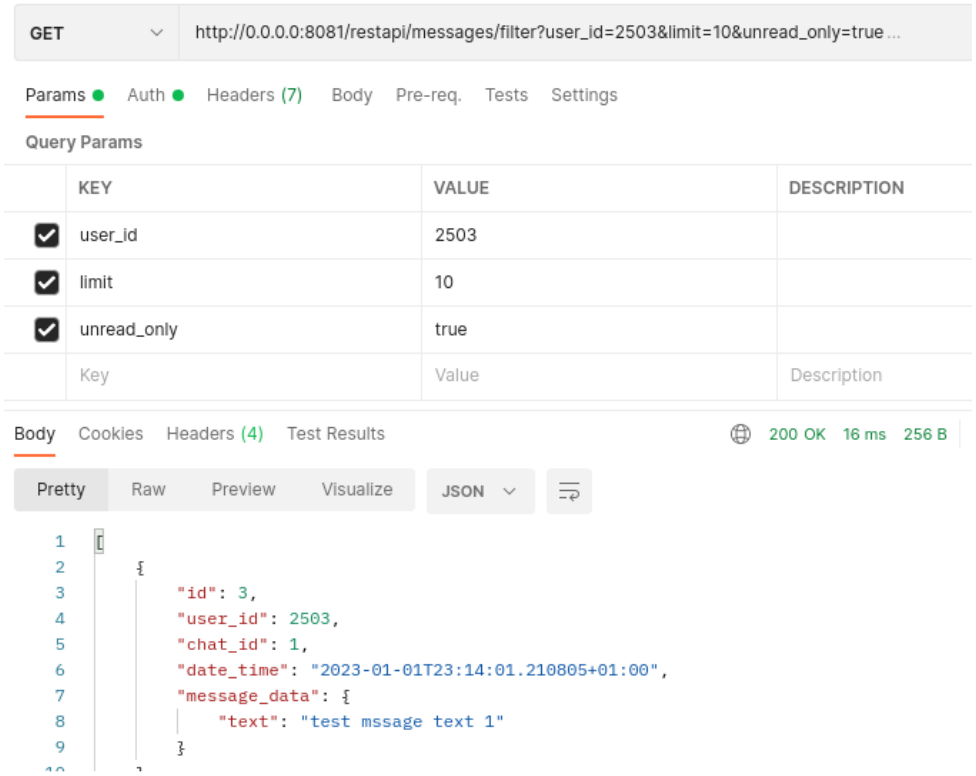
Sprawdzenie poprawności działania całej aplikacji zostało przeprowadzone za pomocą platformy API Postman. Zostały przetestowane wszystkie endpointy opracowanej aplikacji (Rys. 5.2). Na rysunkach 5.3 i 5.4 zaprezentowano przykłady wyników żądań wysłanych z Postmana do części backendowej zaimplementowanego systemu typu komunikator.

Rysunek 5.3 ilustruje przykład zapytania GET do części serwerowej, przeznaczonego do filtracji wiadomości. Zwrócone dane zawierają pierwsze 10 nieprzeczytanych wiadomości dla użytkownika z id równym 2503.

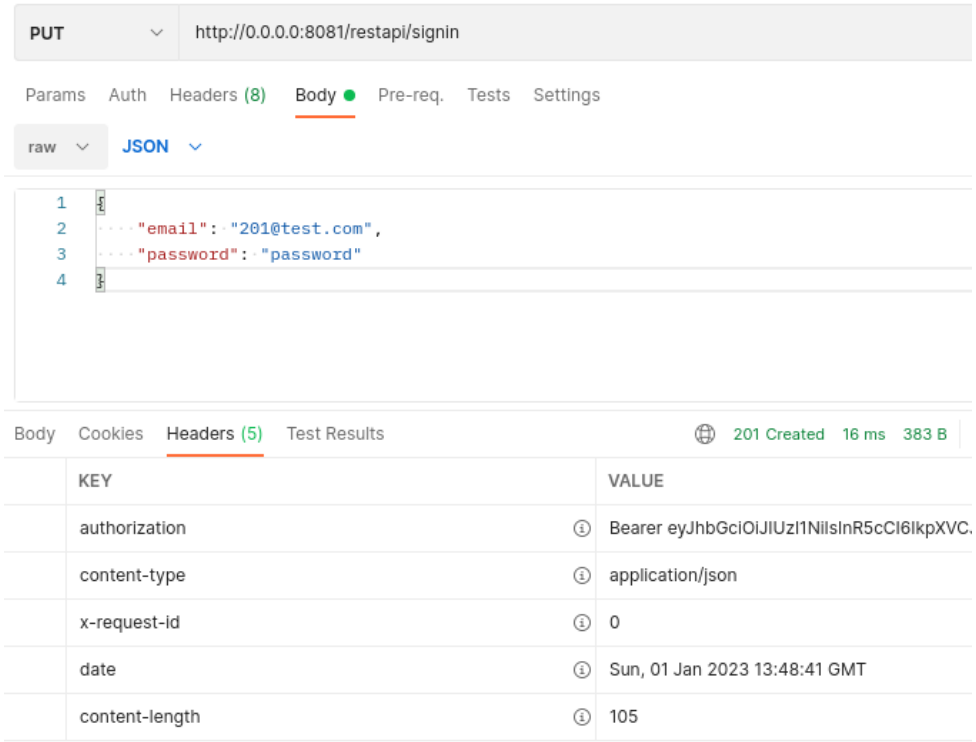
Na Rysunku 5.4 przedstawiono zapytanie przeznaczone do logowania użytkownika. Odpowiedź zawiera JWT token autoryzacji, który znajduje się w nagłówku w polu `authorization`.



Rys. 5.2: Lista testów punktów końcowych



Rys. 5.3: Przykład testowania punktu końcowego dla operacji filtrowania wiadomości



Rys. 5.4: Przykład testowania punktu końcowego dla operacji logowania się do systemu

5.1.3. Wnioski z testów

Przeprowadzenie testów jednostkowych umożliwiło znalezienie błędów w algorytmach i modułach części backendowej zaimplementowanego systemu. W efekcie działanie systemu zostało poprawione i wszystkie 44 testy zostały zakończone pozytywnie.

Testy funkcjonalne 18 endpointów API przeprowadzone za pomocą narzędzia Postman pozwoliły sprawdzić działanie nie tylko części przetwarzającej żądania, alew także i współdziałanie między sobą wszystkich elementów systemu oraz logiki biznesowej aplikacji.

Rozdział 6

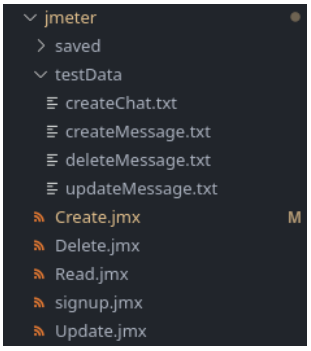
Ocena wydajności zaprojektowanych wariantów systemu webowego

W niniejszym rozdziale opisano sposób oceny wydajności opracowanych wariantów architektonicznych systemu webowego z wykorzystaniem wybranych narzędzi. Zawarto w nim również wyniki testów ich analiza oraz wnioski z badań.

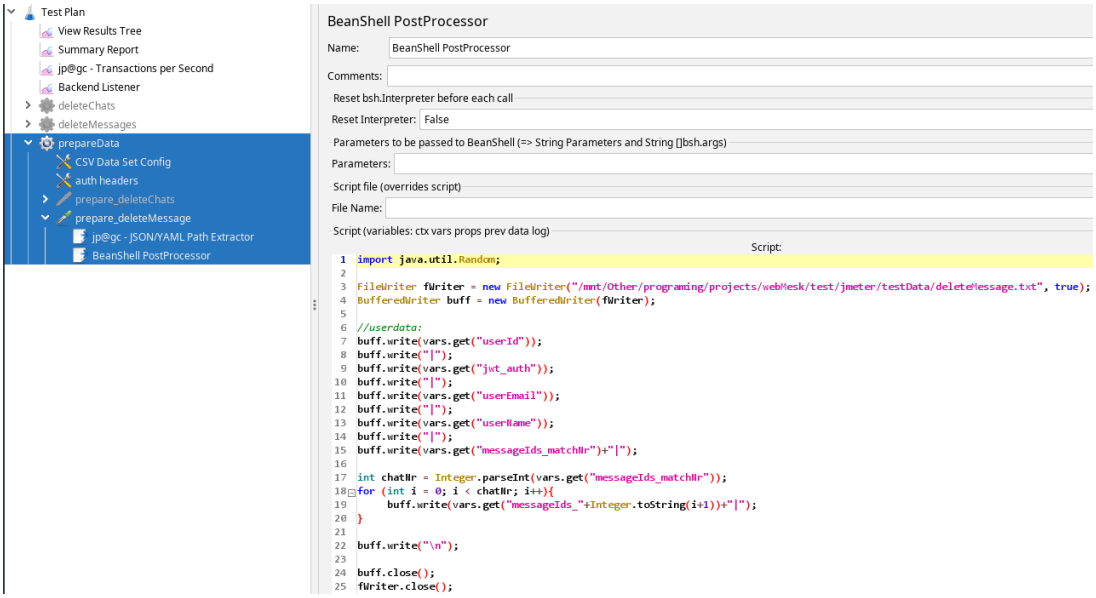
6.1. Monitorowanie i prowadzenie testów wydajnościowo-obciążeniowych

W celu przeprowadzenia testów wydajnościowo-obciążeniowych, zbierania danych statystycznych oraz monitorowania obciążenia zostały użyte następujące narzędzia: JMeter, Influxdb, Telegraf, Grafana.

JMeter
JMeter służy do prowadzenia testów wydajnościowo-obciążeniowych oraz zastępuje część frontendową aplikacji webowej. Katalog `JMeter` (Rys. 6.1) zawiera plany prowadzenia testów razem z wątkami, które przygotowują dane do testowania. Katalog `testData` i plik `users.csv` zawierają dane do prowadzenia testów, które zostały stworzone podczas uruchomienia przygotowawczych wątków (przykład na rysunku 6.2). Katalog `saved` zawiera dane dla różnych zbiorów danych. Wszystkie przypadki testowane w pracy zostały opisane w rozdziale 6.



Rys. 6.1: Pliki testowe JMeter



Rys. 6.2: Przygotowanie danych do testowania usuwania wiadomości

Telegraf

Telegraf służy do zbierania danych o obciążeniu elementów systemu. Skoro wszystkie elementy znajdują się w kontenerach Docker, wystarczy odczytać plik `/var/run/docker.sock`, który zawiera dane dotyczące użycia przez kontener CPU, RAM, sieci. Narzędzie to powinno mieć dostęp do wspomnianego pliku urządzenia na którym zainstalowano kontenery Docker. Plik konfiguracji (listing 6.1) zawiera ustawienia samego narzędzia (blok `agent`), metodę zbierania danych (blok `inputs.docke`), oraz miejscu zachowywania otrzymanych danych (blok `outputs.influxdb`).

Listing 6.1: Konfiguracja programu Telegraf

```
[agent]
interval = "3s"
round_interval = true
metric_batch_size = 1000
metric_buffer_limit = 10000
collection_jitter = "0s"
precision = "1s"
hostname = "192.168.0.63"
omit_hostname = false

[[inputs.docker]]
endpoint = "unix:///var/run/docker.sock"
gather_services = false
container_names = []
source_tag = false
container_name_include = []
container_name_exclude = []
timeout = "3s"
perdevice = true
total = false
docker_label_include = []
docker_label_exclude = []
tag_env = ["JAVA_HOME", "HEAP_SIZE"]

[[outputs.influxdb]]
urls = ["http://192.168.0.63:8086"]
database = "JMeter_telegraf"
timeout = "5s"
username = "telegraf"
password = "metricsmetricsmetricsmetrics"
```

Influxdb
Dane zebrane za pomocą programów JMeter i Telegraf trafiają do systemu bazy danych Influxdb. W niej stworzono bazę danych `JMeter_telegraf`. Która posiada kilka zdefiniowanych przez Telegraf i JMeter tabel. Dane zapisane w bazie danych zostały użyte do analizy wyników oraz obliczania wskaźników oceny wydajności systemów.

Grafana
Do wyświetlania danych zawartych w Influxdb w czasie rzeczywistym oraz obliczania danych została użyta Grafana. Narzędzie to pozwala na eksportowanie danych do plików formatu .csv, co ułatwia pracę z nimi.

Sposób wyświetlania i przetwarzania danych dotyczących wykorzystania CPU% i pamięci RAM w MB kontenerów zaprezentowano na rysunku 6.3. Zużycie 100% CPU oznacza zużycie 1 jądra, tj. jeśli kontener używa 230%, to oznacza, że używa kilka rdzeni procesora i obciążenie CPU jest sumą zasobów użytych na wszystkich przedzielonych rdzeniach. Zwiększenie użycia zasobów jest spowodowano prowadzeniem testu wydajnościowo-obciążeniowego. Najwięcej zasobów, jak procesora tat i pamięci RAM, używa system bazy danych PostgreSQL. Dwaj kontenery, zawierające instancje instancje backendowe używają mniej niż 100 MB pamięci i około 100% CPU.

Do przetwarzania danych zebranymi przez JMeter został użyty dashboard `Apache JMeter Dashboard using Core InfluxdbBackendListenerClient` (id 5496) Na rysunku 6.4 widoczne następujące dane: liczba zapytań, liczba błędnych odpowiedzi, procent błędnych odpowiedzi, ilość przesyłanych danych, wykresy pokazujące liczbę zapytań na sekundę, liczbę błędów, czasy odpowiedzi (średnia, minimum, maksimum i inne) i inne.

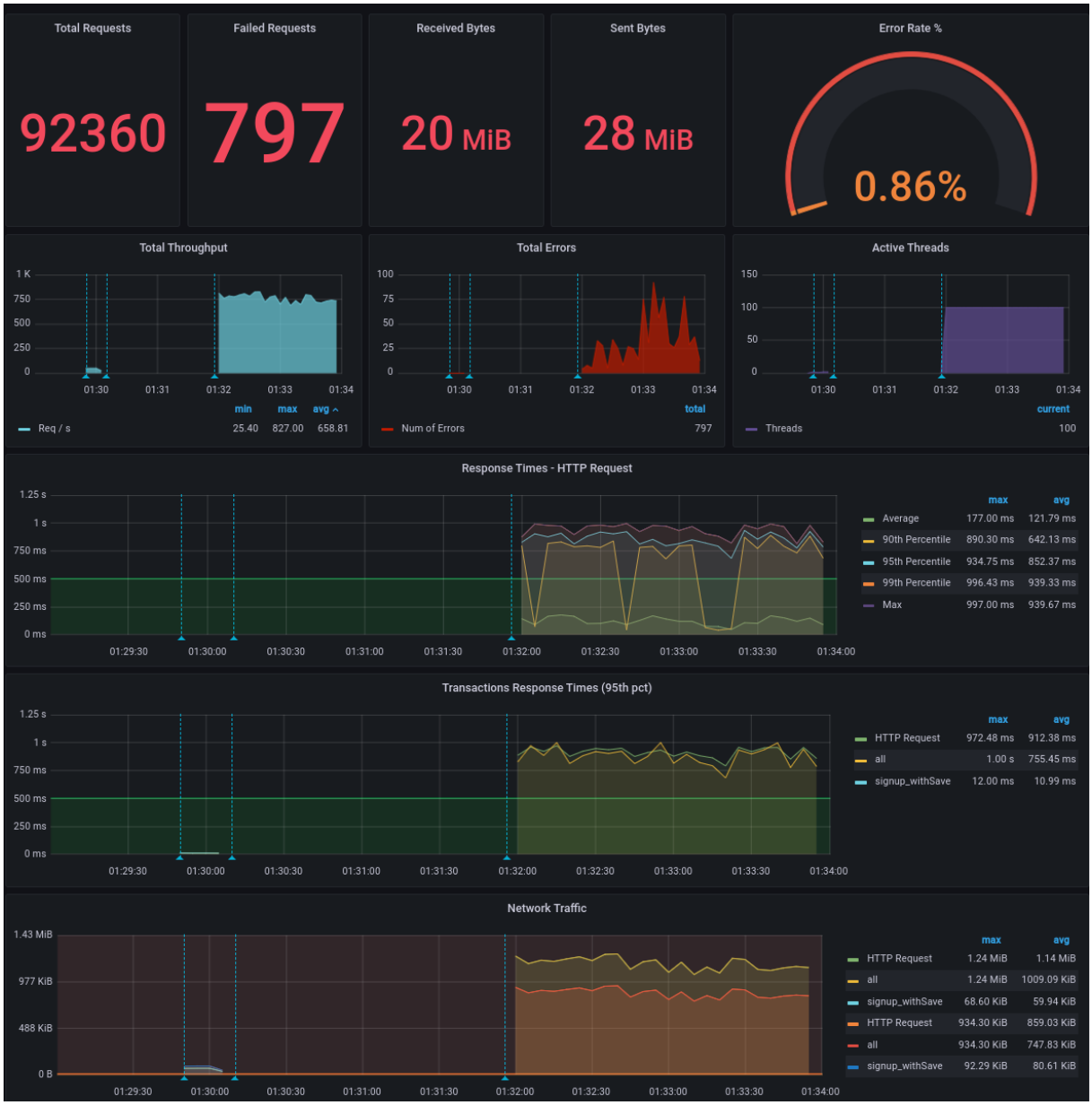
Rys. 6.3: Przykład wykresów Grafana generowanych podczas testowania wydajnościowo-obciążeniowego endpointu wczytującego listę wiadomości. Wyświetlane są dotyczące zużycia CPU i pamięci RAM dla każdego kontenera systemu

6.1.1. Wnioski z testów wstępnych

Wstępne testy obciążeniowe wykazały użyteczność zastosowanych narzędzi oraz możliwość działania zaimplementowanego systemu typu komunikator pod obciążeniem.

Testy te pozwoliły odpowiednio skonfigurować dopuszczalną liczbę podłączeń zaimplementowanej aplikacji w języku Go do bazy danych PostgreSQL. Bez tych zmian wszystkie konfiguracje systemu szybko przepełniały dostępną pulę połączeń.

Nie wszystkie z możliwych konfiguracje działały stabilnie. Do stabilnych konfiguracji (pełna lista testowanych konfiguracji znajduje się w tabeli 6.2) należą: konfiguracje używające 1 instancję backendową z 1 i 2 rdzeniami procesora, niezależnie od zastosowania loadbalancera oraz instancje używające 2 i 4 instancji backendowych z 1 rdzeniem procesora. Do niestabilnych można zaliczyć konfiguracje używające 4 rdzenie procesora. Na rysunku 6.5(a) przedstawiono przykład stabilnego systemu, a na rysunku 6.5(b) niestabilnego systemu. Na wykresach dotyczących niestabilnego systemu na wykresach można zauważyć nie tylko błędy, ale także mniejsze zużycie zasobów z bazy danych PostgreSQL oraz korelację zmian wykresów (spadów i wzniesień). Przebiegi te mają charakter okresowy. Logi działania aplikacji wskazywały na cza-



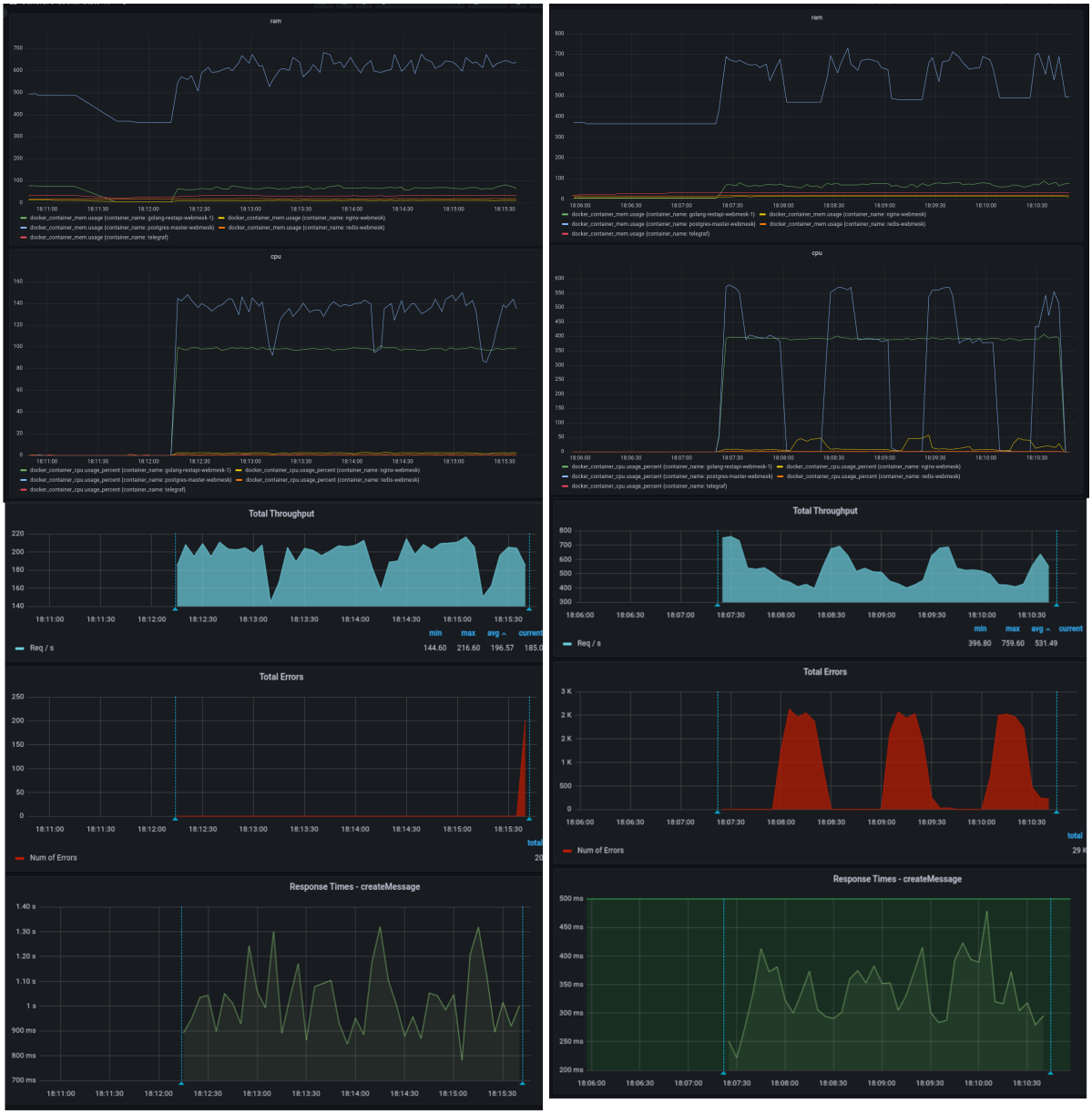
Rys. 6.4: Przykład wykresów Grafana generowanych podczas testowania wydajnościowo-obciążeniowego endpointu wczytującego listę wiadomości; dane dotyczące analizy testu prowadzonego przez JMeter

sowe odłączenie bazy danych PostgreSQL od sieci. Jednak to może być spowodowane również błędnym działaniem biblioteki `gorm`, przeznaczonej do współpracy z PostgreSQL.

6.2. Konfiguracja środowiska testowego

Stanowisko testowe składa się z dwóch urządzeń:

- Laptop `MSI Ge62-7RE`, procesor `i7-7700HQ (2,8GHz, 4 jądra, 8 wątków)`, 32GB RAM, z systemem operacyjnym ArchLinux 5.19.12-zen przeznaczony do wysłania żądań za pomocą narzędzia JMeter oraz gromadzenia i wyświetlania metryk za pomocą Influxdb i Grafana;
- Laptop `DELL Latitude 5521`, procesor `i7-11850H (2,5GHz, 8 jądra, 16 wątków)`, 32GB RAM, z systemem operacyjnym Ubuntu 22.04 5.15.0-56, przeznaczony do uruchomienia części wykonawczej: instancji części backendowej, baz danych PostgreSQL i Redis,



((a)) konfiguracja 1 urządzenia z 1 rdzeniem i z NGINX

((b)) konfiguracja 1 urządzenia z 4 rdzeni i z NGINX

Rys. 6.5: Monitorowanie systemu podczas tworzenia wpisów od konfiguracji systemu

loadbalancera NGINX oraz narzędzia Telegraf, zbierającego dane o działaniu wymienionych instancji telegraf.

Ograniczenia zasobów oraz wersji kontenerów wykorzystanych narzędzi i aplikacji opisano w tabeli 6.1. Wszystkie elementy, oprócz JMeter, uruchomiono jako docker kontenery.

Tab. 6.1: Tabela zasobów i wersji narzędzi wykorzystanych w środowisku testowym

Narzędzie	Wersja	Ograniczenie CPU (rdzenie)	Ograniczenie RAM	Liczba instancji
Go (zaimplementowana część backendowa)	go:1.18	ograniczenia scenariusza badań (tab 6.2)	ograniczenia scenariusza badań (tab 6.2)	ograniczenia scenariusza badań (tab 6.2)
PostgreSQL	postgresql:14	8	8GB	1
Redis	redis:7.0.3	1	1GB	1
NGINX	nginx:1.22	1	1GB	1
telegraf	telegraf:1.24.3	1	1GB	1
grafana	grafana/grafana:9.2.3	bez ograniczeń	bez ograniczeń	1
influxdb	influxdb:1.8	bez ograniczeń	bez ograniczeń	1
JMeter	JMeter 5.5	bez ograniczeń	bez ograniczeń	1

6.3. Scenariusze badań

Warianty skalowalności zawsze testowane osobno. Listę badanych wariantów architektonicznych systemu dla aplikacji ze skalowalnością poziomą i pionową, ich parametrów i aliasów (skrótów) zaprezentowano w tabeli 6.2. Wersje używające 1 instancji części backendowej oraz 1 rdzeni procesora i 1GB RAM są wersjami kontrolnymi: 1x1 i 1x1_n.

Tab. 6.2: Lista badanych wariantów systemu

Aliasy konfiguracji	Metoda skalowania	Loadbalancer	Liczba instancji	Ograniczenie CPU (rdzenie)	Ograniczenie RAM
1x1	brak	brak	1	1	1GB
1x2	pionowa	brak	1	2	2GB
1x4	pionowa	brak	1	4	4GB
1x1_n	brak	jest (NGINX)	1	1	1GB
1x2_n	pionowa	jest (NGINX)	1	2	2GB
1x4_n	pionowa	jest (NGINX)	1	4	4GB
2x1_n	pozioma	jest (NGINX)	2	1	1GB
4x1_n	pozioma	jest (NGINX)	4	1	1GB

Zakresy danych w PostgreSQL: 50000 użytkowników, 600000 czatów i 1000000 wiadomości. Każdy użytkownik jest właścicielem 6-iu czatów (3 - dwuosobowe, 2 - trzyosobowe i 1 - pięcioosobowy), w których komunikuje się z losowymi użytkownikami. W celu eliminacji wpływu wygenerowanych danych na wyniki, zostało wygenerowano 15 zbiorów danych, tj. zbiorów odpowiadających 15 próbom dla każdego testu. Czas trwania jednej próby testu jest 1 minuta.

6.4. Ocena wydajności

Testy wydajnościowo-obciążeniowe zostały wykonane dla operacji zapisywania, wczytywania, aktualizowania i usuwania danych. Każdy test był powtarzany 15 razy oraz trwał 1 minutę, uśrednione wyniki zostały zapisane w tabelach 6.4, 6.5, 6.6, 6.10, 6.11, 6.12, 6.16, 6.17, 6.18, 6.22, 6.23, 6.24. W tabelach zawarto następujące uśrednione dane dla 15 próbek w ramach każdego z testów:

- alias do konfiguracji z tabeli 6.2;
- średnia liczba zapytań;
- średni czas odpowiedzi na żądanie;

- minimalny czas odpowiedzi na żądanie;
- maksymalny czas odpowiedzi na żądanie;
- odchylenie standardowe dla pojedynczych żądań; dane wyliczane przez JMeter [7];
- odchylenie standardowe obliczone dla 15 średnich czasów odpowiedzi otrzymanych w ramach testu (obliczone za pomocą funkcji `STDEV.S` w tablicach google [13]);
- wariancja obliczone dla 15 średnich czasów odpowiedzi otrzymanych w ramach testu (obliczone za pomocą funkcji `VARA` w tablicach google [15]);
- średnia liczba żądań na sekundę;
- minimalna liczba żądań na sekundę;
- maksymalna liczba żądań na sekundę;
- średnia liczba błędnych odpowiedzi;
- minimalna liczba błędnych odpowiedzi;
- maksymalna liczba błędnych odpowiedzi;
- średni procent użycia CPU przez jedną instancję;
- średni procent użycia CPU przez bazę danych PostgreSQL;
- średni procent użycia CPU przez loadbalancer NGINX (jeśli jest uruchomiony);
- średni procent użycia CPU przez bazę danych Redis;
- średnia liczba MB użycia pamięci RAM przez jedną instancję;
- średnia liczba MB użycia pamięci RAM przez bazę danych PostgreSQL;
- średnia liczba MB użycia pamięci RAM przez loadbalancer NGINX (jeśli jest uruchomiony);
- średnia liczba MB użycia pamięci RAM przez bazę danych Redis.

Statystyka t-Studenta użyta do analizy różnic między średnimi czasami odpowiedzi prób testowych między konfiguracjami (tabela 6.2) skalowalnymi poziomo i pionowo systemu typu komunikator zrealizowanego w języku Go. Obliczenia mogą być prowadzone przy wykonaniu następujących założeń:

- rozkład danych w każdej z analizowanych grup ma rozkład normalny (otrzymane dane z prób mają wykres o postaci wykresu normalnego. Sprawdzono za pomocą `GIGAcalculator` [24]);
- porównywane grupy danych mają podobną liczebność (spełnione: 15 prób w ramach każdego testu systemy);
- wariancje w porównywanych grupach danych są do siebie podobne (spełnione: wariancja w tabelach 6.4, 6.5, 6.6, 6.10, 6.11, 6.12, 6.16, 6.17, 6.18, 6.22, 6.23, 6.24).

Przeprowadzono zostało obliczenie t-Statystyki za pomocą funkcji `T.TEST` w tablicach Google [14] dla każdych dwóch analizowanych grup danych oraz dla parametru istotności p=0,05.

6.4.1. Zapisywanie danych do bazy

W celu porównania wydajności skalowalnych systemów o różnej architekturze (tabela 6.2) w trybie zapisywania danych, zostały przeprowadzone testy wydajnościowo-obciążeniowe endpointa `/restapi/messages` z metodą POST dla 50, 100 i 200 wątków uruchomionych w programie JMeter.

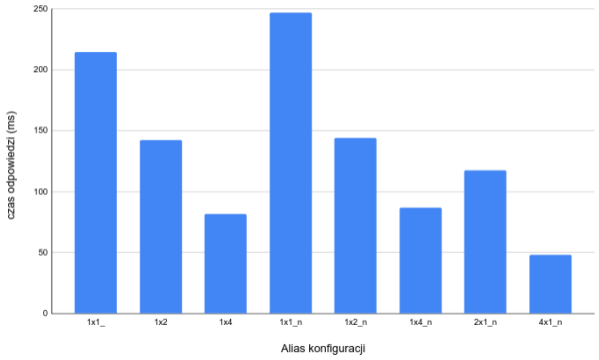
Wyniki testowania żądań tworzących nowe wiadomości w systemie, czyli symulacji wysłania nowych wiadomości do czatów:

- dla 50 wątków JMetera znajdują się w tabeli 6.4 na końcu rozdziału;
- dla 100 wątków JMetera znajdują się w tabeli 6.5 na końcu rozdziału;
- dla 200 wątków JMetera znajdują się w tabeli 6.6 na końcu rozdziału.

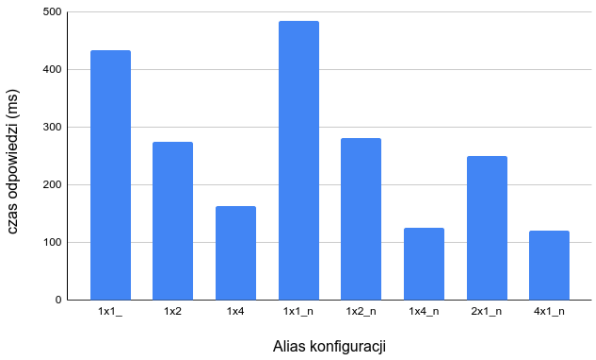
Na rysunkach przedstawiono porównanie średniego czasu odpowiedzi podczas zapisywania danych dla różnych konfiguracji systemów:

- dla 50 wątków JMetera wyniki pokazano na rysunku 6.6(a);

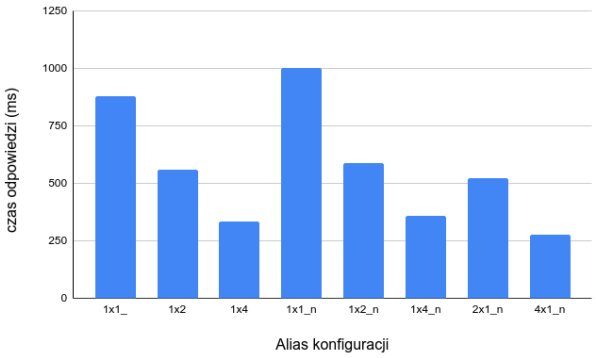
- dla 100 wątków JMetera wyniki pokazano się na rysunku 6.6(b);
- dla 200 wątków JMetera wyniki pokazano się na rysunku 6.6(c).



((a)) przy 50 wątkach JMetera



((b)) przy 100 wątkach JMetera



((c)) przy 200 wątkach JMetera

Rys. 6.6: Zależność czasu odpowiedzi tworzenia wpisów od konfiguracji systemu

Na rysunkach przedstawiono porównanie średniej liczby błędów dla różnych konfiguracji systemów:

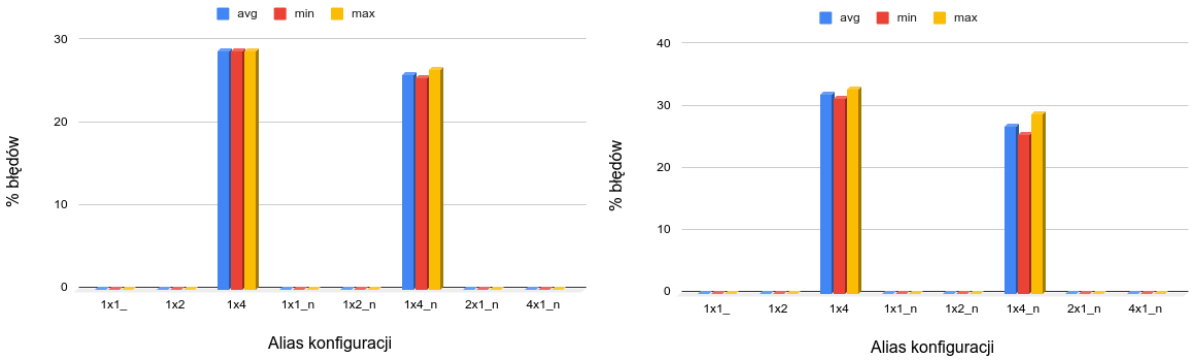
- dla 50 wątków JMetera wykres znajduje się na rysunku 6.7(a);
- dla 100 wątków JMetera wykres znajduje się na rysunku 6.7(b);
- dla 200 wątków JMetera wykres znajduje się na rysunku 6.7(c).

Wyniki testów pokazują, że wraz ze zwiększeniem mocy obliczeniowej części backedowej zarówno poprzez skalowanie poziomą jak i pionową, zmniejsza się najważniejszy parametr — czas odpowiedzi. To oznacza, że komfort pracy dla użytkowników się zwiększa oraz wzrasta liczba żądań, które mogą być przetworzone, o czym świadczy zwiększona liczba żądań oraz liczba żądań na sekundę. Dla 50 wątków JMetera wszystkie systemy pokazały dobre wyniki, ponieważ średnie czasy odpowiedzi nie przekroczyły 200 ms. W przypadku 100 wątków tylko systemy używające 4 rdzeni procesora lub 4 instancji mają wyniki dobre. Natomiast dla 200 wątków wszystkie systemy wykazują wyłącznie wyniki dopuszczalne. Żaden system nie zyskał wyników gorszych niż dopuszczalnie, czyli powyżej 1 sekundy.

Konfiguracje backendu, używające 4 rdzeni procesora, choć pokazały bardzo dobre wyniki czas odpowiedzi, jednak zostają nieużyteczne z powodu dużej liczby błędów, podczas gdy inne badane systemy pokazywały ich brak.

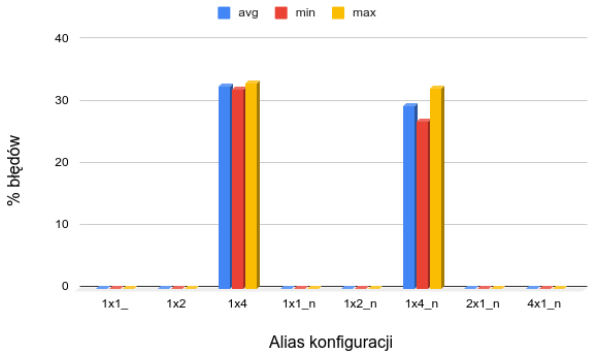
Porównując systemy o takich samych zasobach CPU i RAM, ale z loadbalancerem NGINX i bez można zauważyć, że loadbalancer negatywnie wpływa na czas reakcji systemu (zwiększa go).

Patrząc na wartości czasu odpowiedzi zasobów CPU i pamięci RAM w tabelach 6.4 6.5 6.6, można zauważyć, że skalowalność pozioma jest bardziej efektywna. Przy dwukrotnym zwiększeniu mocy jednego serwera uzyskano zmniejszenie czasu odpowiedzi mniejszą niż dwukrotną,



((a)) przy 50 wątkach JMetera

((b)) przy 100 wątkach JMetera



((c)) przy 200 wątkach JMetera

Rys. 6.7: Zależność liczby błędów tworzenia wpisów od konfiguracji systemu

natomiast podczas podwajania ilości instancji czasy odpowiedzi zmniejszają się nawet więcej niż dwukrotnie.

Zużycie zasobów procesora przez backend napisany w języku Go, świadczy o tym, że zostały zużyte wszystkie zasoby, które były dostępne. Obciążenie bazy danych PostgreSQL, Redis i loadbalancera NGINX korelują z liczbą żądań na sekundę. Narzędzia NGINX i Redis prawie nie używały przedzielonych zasobów. Baza danych PostgreSQL zużywała zawsze najwięcej zasobów ze wszystkich użytych narzędzi.

Przedzielone zasoby pamięci RAM były zbyt duże dla danego systemu. Żadna cześć systemu nie wykorzystywała nawet dziesiątej części dostępnej pamięci. Jednak zużycie zasobów rosło proporcjonalnie do liczby żądań na sekundę. Systemy skalowalne poziomo sumarycznie zużywają więcej pamięci, niż systemy skalowalne pionowo. W przypadku występowania błędów ilość pamięci użytej przez backend, często się zmniejszała (CPU % i RAM w tabelach 6.4 6.5 6.6).

Przeprowadzane testy t-Studenta (spełnione warunki: dane należą do rozkładu normalnego, jednakowa liczba próbek, podobne wariancje) wskazują na statystycznie znaczącą różnicę między systemami oraz na możliwość odrzucenia zerowej hipotezy (wartości dużo mniejsze od p=0,005) o braku wpływu skalowalności części backendowej na średni czas odpowiedzi:

- dla 50 wątków JMetera wyniki znajduje się w tabeli 6.7 na końcu rozdziału;
- dla 100 wątków JMetera wyniki znajduje się w tabeli 6.8 na końcu rozdziału;
- dla 200 wątków JMetera wyniki znajduje się w tabeli 6.9 na końcu rozdziału.

6.4.2. Pobieranie danych z bazy

W celu porównania wydajności skalowalnych systemów o różnej architekturze (tabela 6.2) w trybie odczytu danych, zostały przeprowadzone testy wydajnościowo-obciążeniowe endpointa `/restapi/messages/filtr` z metodą GET dla 50, 100 i 200 wątków programu JMeter. Te-

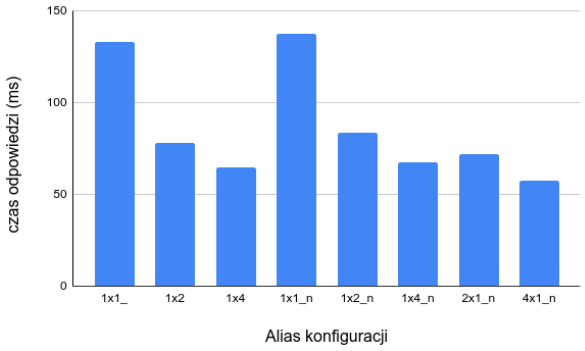
stowane żądanie miało jako parametry filtracji: ograniczenie liczby zwracanych danych do 1 wiadomości oraz odczytana wiadomość przeznaczona była dla ustalonego użytkownika. Przykład URL żądania: `/restapi/messages/filter?user_id=25888&limit=1`.

Wyniki testowania żądań pobierania wiadomości w systemie, czyli symulacji pobierania wiadomości dla podanego użytkownika:

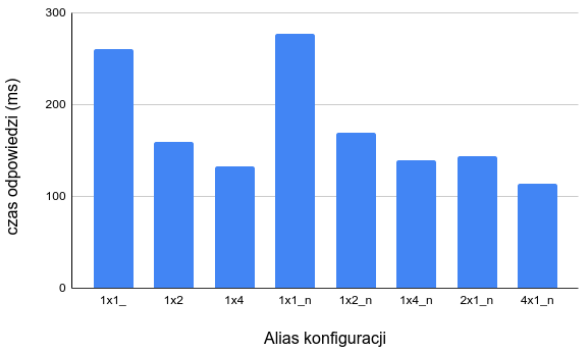
- dla 50 wątków JMetera wyniki znajdują się w tabeli 6.10 na końcu rozdziału;
- dla 100 wątków JMetera wyniki znajdują się w tabeli 6.11 na końcu rozdziału;
- dla 200 wątków JMetera wyniki znajdują się w tabeli 6.12 na końcu rozdziału.

Na rysunkach przedstawiono porównanie średniego czasu odpowiedzi podczas pobierania danych dla różnych konfiguracji systemów:

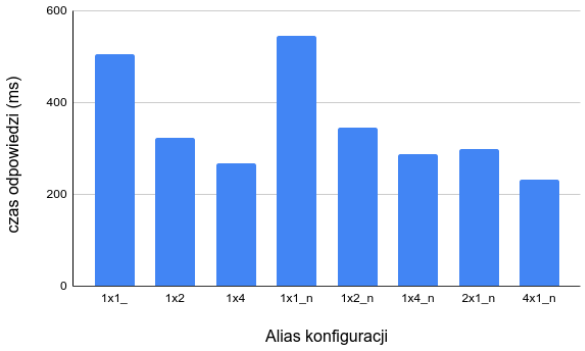
- dla 50 wątków JMetera wykres pokazano na rysunku 6.8(a);
- dla 100 wątków JMetera wykres pokazano na rysunku 6.8(b);
- dla 200 wątków JMeterawykres pokazano na rysunku 6.8(c).



((a)) przy 50 wątkach JMetera



((b)) przy 100 wątkach JMetera

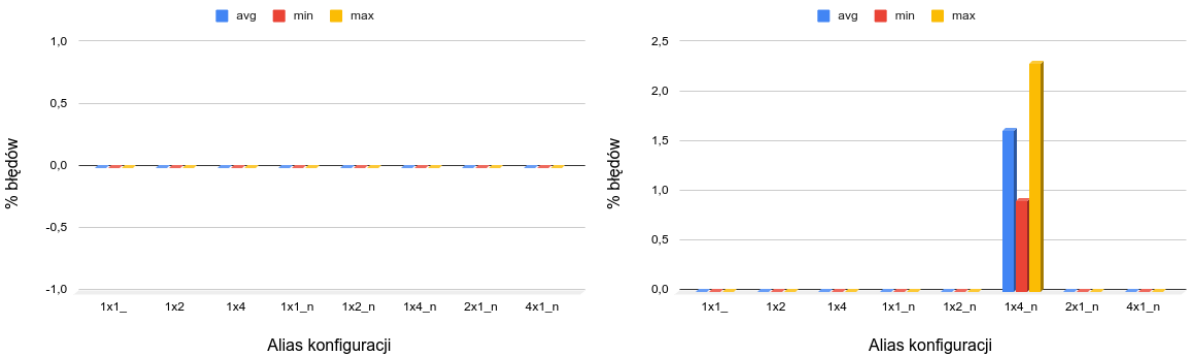


((c)) przy 200 wątkach JMetera

Rys. 6.8: Zależność czasu odpowiedzi pobierania danych od konfiguracji systemu

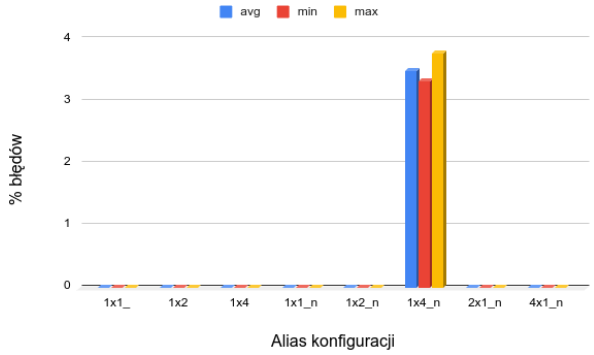
Na rysunkach przedstawiono porównanie średni procent błędów podczas pobierania danych dla różnych konfiguracji systemów:

- dla 50 wątków JMetera wykres znajduje się na rysunku 6.9(a);
- dla 100 wątków JMetera wykres znajduje się na rysunku 6.9(b);
- dla 200 wątków JMetera wykres znajduje się na rysunku 6.9(c).



((a)) przy 50 wątkach JMetera

((b)) przy 100 wątkach JMetera



((c)) przy 200 wątkach JMetera

Rys. 6.9: Zależność liczby błędów pobierania danych od konfiguracji systemu

Wyniki testów pobierania pokazują, że wraz ze zwiększeniem mocy części backedowej zarówno dla skalowalności poziomej jaki i pionowej, zmniejsza się najważniejszy parametr — czas odpowiedzi. To oznacza, że komfort pracy dla użytkowników się zwiększa oraz wzrasta liczba żądań, które mogą być opracowane, o czym świadczy zwiększona liczba żądań oraz liczba żądań na sekundę. Dla 50 wątków JMetera wszystkie systemy pokazały dobre wyniki, czyli średnie czasy odpowiedzi nie przekroczyły 200 ms. W przypadku 100 wątków tylko systemy używające 1 instancję backendową o 1 rdzeniu CPU mają wartość czasu odpowiedzi dopuszczalną, a nie dobrą. Natomiast dla 200 wątków wszystkie systemy mają wyłącznie czasy dopuszczalne, jednak konfiguracja z 4 instancjami jest zbliżona do poziomu dobrego. Żaden system nie pokazał wyniki gorsze niż dopuszczalnie, czyli powyżej 1 sekundy (rys. 6.8).

Konfiguracje backendu, używające 4 rdzenia procesora z użyciem loadbalancera, choć pokazały dobre wyniki w czasie odpowiedzi, jednak w odróżnieniu od innych konfiguracji systemu, dla 100 i 200 wątków JMetera wygenerowano błędne odpowiedzi (rys. 6.9).

Patrząc na średni czas odpowiedzi (tabeli 6.10, 6.11, 6.12), systemy o takich samych zasobach CPU i RAM, ale z loadbalancerem NGINX i bez można zauważyć, że loadbalancer negatywnie wpływa na czas reakcji systemu (zwiększa go).

Podczas porównania czasów odpowiedzi różnych metod skalowalności (poziomej i pionowej), można zauważyć, że skalowalność pionowa jest bardziej efektywna. Przy zwiększaniu liczby instancji backendowych z 1 do 2, jest zauważalne większe zwiększanie wydajności niż przy zwiększaniu liczby rdzeni z 1 do 2. Wydajność systemu zwiększa się i podczas zwiększania ilości rdzeń z 2 do 4 i podczas dodawania instancji (z 2 do 4).

Zużycie zasobów procesora przez backend napisany w języku Go świadczy o tym, że zostały zużyte wszystkie zasoby, które były dostępne. Wyjątkiem jest konfiguracja systemu skalowalna poziomo używająca 4 instancje backendów. Prawdopodobnie jest to spowodowano zbliżaniem się do limitu zasobów CPU bazy danych PostgreSQL. Obciążenia bazy danych PostgreSQL, Redis i loadbalancera NGINX korelują z liczbą żądań na sekundę. Narzędzia NGINX i Redis prawie nie używały przedzielonych zasobów. Baza danych PostgreSQL zużywała zawsze najwię-

cej zasobów ze wszystkich użytych narzędzi i w niektórych przypadkach (1 instancja 4 rdzenie bez loadbalancera, 1 instancja 4 rdzenie z loadbalancerem NGINX, 2 instancje 1 rdzeń z loadbalancerem NGINX, 4 instancje 1 rdzeń z loadbalancerem NGINX) zostają prawie w 100% wykorzystywane wszystkie 8 rdzenie CPU (łącznie zbliża się do 800%).

Przedzielone zasoby pamięci RAM były zbyt duże dla danego systemu. Żaden element systemu nie wykorzystywała nawet dziesiątej części z nich. Jednak zużycie zasobów rosło proporcjonalnie do liczby żądań na sekundę. Systemy skalowalne poziomo sumarycznie zużywają więcej pamięci, niż systemy skalowalne pionowo. W przypadku występowania błędów ilość pamięci użytej przez backend, często się zmniejszała.

Przeprowadzane testy t-Studenta (spełnione warunki: dane należą do rozkładu normalnego, jednakowa liczba próbek, podobne wariancje) wskazują na statystycznie znaczącą różnicę między systemami oraz na możliwość odrzucenia zerowej hipotezy (wartości dużo mniejsze od p=0,005) o braku wpływu skalowalności części backendowej na średni czas odpowiedzi:

- dla 50 wątków JMetera wyniki znajdują się w tabeli 6.13 na końcu rozdziału;
- dla 100 wątków JMetera wyniki znajdują się w tabeli 6.14 na końcu rozdziału;
- dla 200 wątków JMetera wyniki znajdują się w tabeli 6.15 na końcu rozdziału.

6.4.3. Aktualizowanie danych w bazie

W celu porównania wydajności skalowalnych systemów o różnej architekturze (tabela 6.2) w trybie zmiany istniejących danych, zostały przeprowadzone testy wydajnościowo-obciążeniowe endpointa `/restapi/messages` z metodą PUT dla 50, 100 i 200 wątków programu JMeter.

Wyniki testowania żądań aktualizacji wiadomości w systemie:

- dla 50 wątków JMetera wyniki znajdują się w tabeli 6.16 na końcu rozdziału;
- dla 100 wątków JMetera wyniki znajdują się w tabeli 6.17 na końcu rozdziału;
- dla 200 wątków JMetera wyniki znajdują się w tabeli 6.18 na końcu rozdziału.

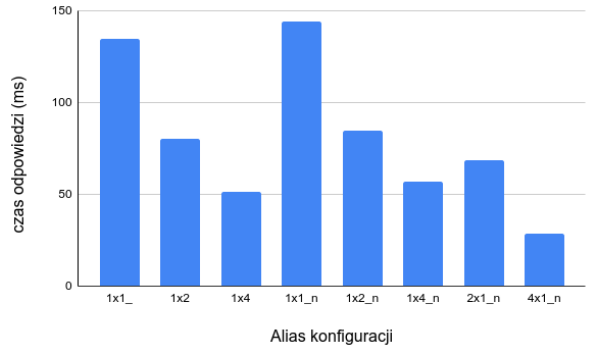
Na rysunkach przedstawiono porównanie średniego czasu odpowiedzi podczas operacji aktualizowania danych dla różnych konfiguracji systemów:

- dla 50 wątków JMetera wykres pokazano na rysunku 6.10(a);
- dla 100 wątków JMetera wykres pokazano się na rysunku 6.10(b);
- dla 200 wątków JMetera wykres pokazano się na rysunku 6.10(c).

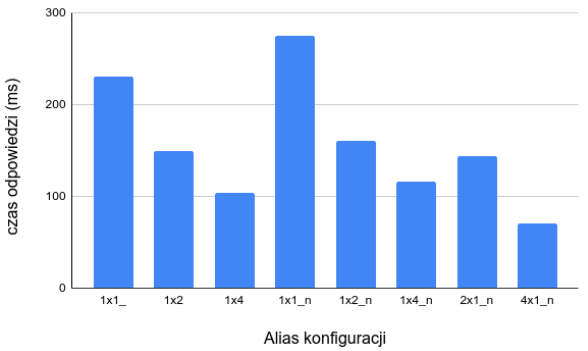
Na rysunkach przedstawiono porównanie średniej liczby błędów podczas operacji aktualizowania danych dla różnych konfiguracji systemów:

- dla 50 wątków JMetera wykres znajduje się na rysunku 6.11(a);
- dla 100 wątków JMetera wykres znajduje się na rysunku 6.11(b);
- dla 200 wątków JMetera wykres znajduje się na rysunku 6.11(c).

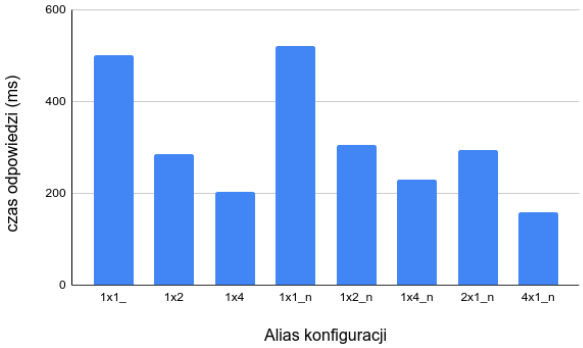
Wyniki testów aktualizacji pokazują, że wraz ze zwiększeniem mocy części backedowej zarówno dla skalowalności poziomej jak i pionowej, zmniejsza się najważniejszy parametr — czas odpowiedzi. To oznacza, że komfort pracy dla użytkowników się zwiększa oraz wzrasta liczba żądań, które mogą być opracowane, o czym świadczy zwiększona liczba żądań oraz liczba żądań w sekundę. Dla 50 wątków JMetera wszystkie systemy pokazały dobre wyniki, czyli średnie czasy odpowiedzi nie przekroczyły 200 ms. W przypadku 100 wątków tylko systemy używające 1 instancję backendową o 1 rdzeniu CPU mają wartość czasu odpowiedzi dopuszczalną, a nie dobrą. Natomiast dla 200 wątków jedynie 4 instancje o 1 rdzeniu procesora nie przekroczyły czasu 200 ms. Do tej wartości zbliżone są również konfiguracje systemu używające 4 rdzenie. Żaden system nie uzyskał wyników gorszych niż dopuszczalne, czyli powyżej 1 sekundy.



((a)) przy 50 wątkach JMetera

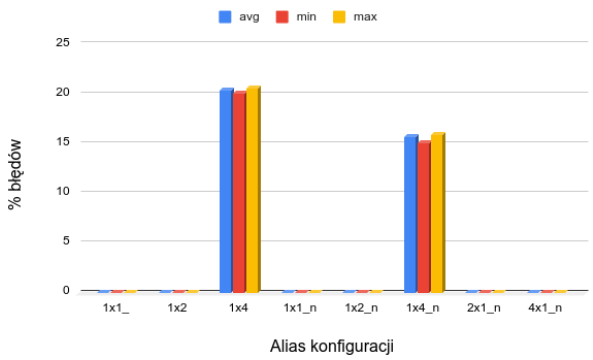


((b)) przy 100 wątkach JMetera

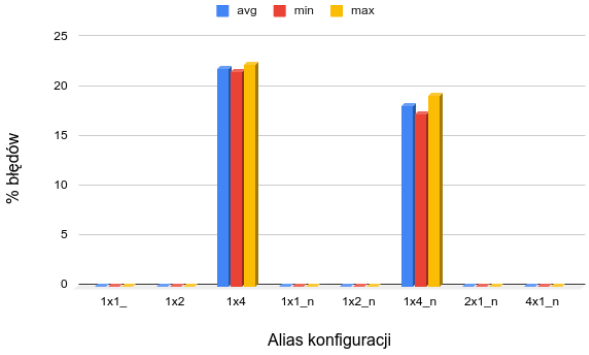


((c)) przy 200 wątkach JMetera

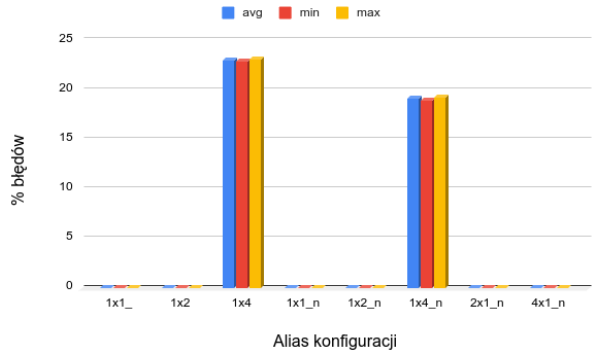
Rys. 6.10: Zależność czasu odpowiedzi aktualizowania danych od konfiguracji systemu



((a)) przy 50 wątkach JMetera



((b)) przy 100 wątkach JMetera



((c)) przy 200 wątkach JMetera

Rys. 6.11: Zależność liczby błędów aktualizowania danych od konfiguracji systemu

Konfiguracje backendu, używające 4 rdzenie procesora, choć pokazały lepsze wyniki czasów odpowiedzi, to jednak są nieużyteczne z powodu dużej liczby błędów, gdy inne systemy pokazywały ich brak (rysunki 6.11).

Porównując systemy o takich samych zasobach CPU i RAM, ale z loadbalancerem NGINX i bez niego można zauważyć, że loadbalancer negatywnie wpływa na czas reakcji systemu (zwiększa go).

Podczas porównania wpływu metody skalowalności poziomej i pionowej, można zauważyć (rysunki 6.10), że skalowalność pionowa jest bardziej efektywna. Przy dwukrotnym zwiększeniu mocy jednego serwera uzyskano zmniejszenie czasu odpowiedzi mniejszą niż dwukrotną, natomiast podczas podwajania liczby instancji serwerów (skalowalnośc pozioma) czasy odpowiedzi zmniejsza się nawet więcej niż dwukrotnie.

Zużycie zasobów procesora przez backend napisany w języku Go, świadczy o tym, że zostały zużyte wszystkie zasoby, które były dostępne. Obciążenie bazy danych PostgreSQL, Redis i loadbalancera NGINX korelują z liczbą żądań na sekundę. Narzędzia NGINX i Redis nie wykorzystały przedzielonych zasobów. Baza danych PostgreSQL zużywała zawsze najwięcej zasobów ze wszystkich użytych narzędzi.

Przedzielone zasoby pamięci RAM były zbyt duże dla danego systemu. Żaden element systemu nie wykorzystywał nawet dziesiątej części z nich. Jednak zużycie zasobów rosło odpowiednio do do żądań na sekundę. Systemy skalowalne poziomo sumarycznie zużywają więcej pamięci, niż systemy skalowalne pionowo. W przypadku występowania błędów ilość pamięci użytej przez backend, często się zmniejszała.

Przeprowadzane testy t-Studenta (spełnione warunki: dane należą do rozkładu normalnego, jednakowa liczba próbek, podobne wariancje) wskazują na statystycznie znaczącą różnicę między systemami oraz na możliwość odrzucenia zerowej hipotezy (wartości dużo mniejsze od p=0,005) o braku wpływu skalowalności części backendowej na średni czas odpowiedzi:

- dla 50 wątków JMetera wyniki znajdują się w tabeli 6.19 na końcu rozdziału;
- dla 100 wątków JMetera wyniki znajdują się w tabeli 6.20 na końcu rozdziału;
- dla 200 wątków JMetera wyniki znajdują się w tabeli 6.21 na końcu rozdziału.

6.4.4. Usuwanie danych z bazy

W celu porównania wydajności skalowalnych systemów o różnej architekturze (tabela 6.2) w trybie usunięcia danych, zostały przeprowadzone testy wydajnościowo-obciążeniowe endpointa `/restapi/messages` z metodą DELETE dla 50, 100 i 200 wątków JMetera.

Wyniki testowania żądań usuwania wiadomości w systemie, czyli symulacji usuwania danych:

- dla 50 wątków JMetera wyniki znajdują się w tabeli 6.22 na końcu rozdziału;
- dla 100 wątków JMetera wyniki znajdują się w tabeli 6.23 na końcu rozdziału;
- dla 200 wątków JMetera wyniki znajdują się w tabeli 6.24 na końcu rozdziału.

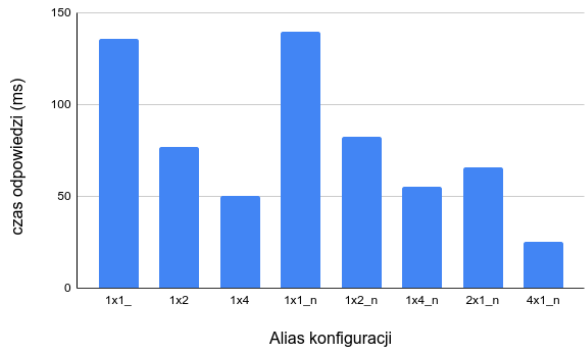
Na rysunkach przedstawiono porównanie średniego czasu odpowiedzi podczas operacji usuwania danych dla różnych konfiguracji systemów:

- dla 50 wątków JMetera wyniki pokazano na rysunku 6.12(a);
- dla 100 wątków JMetera wyniki pokazano na rysunku 6.12(b);
- dla 200 wątków JMetera wyniki pokazano na rysunku 6.12(c).

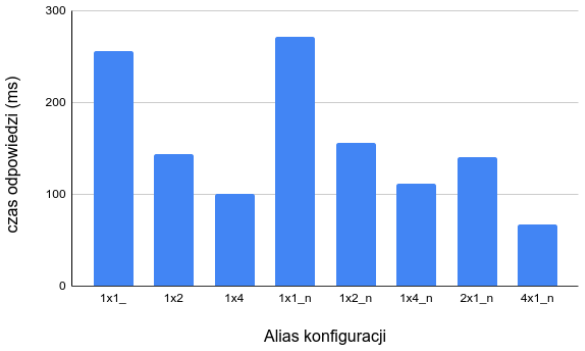
Na rysunkach przedstawiono porównanie średniej liczby błędów podczas operacji usuwania danych dla różnych konfiguracji systemów:

- dla 50 wątków JMetera wykres znajduje się na rysunku 6.13(a);
- dla 100 wątków JMetera wykres znajduje się na rysunku 6.13(b);
- dla 200 wątków JMetera wykres znajduje się na rysunku 6.13(c).

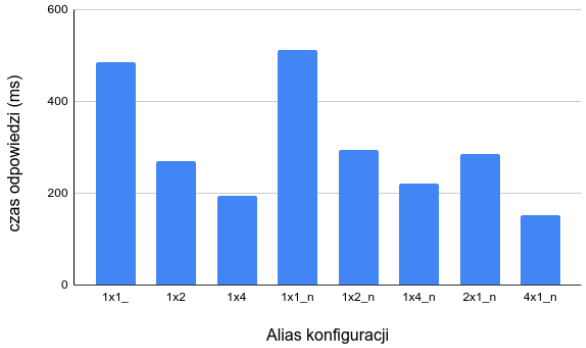
Wyniki testów usunięcia pokazują, że wraz ze zwiększeniem mocy części backedowej zarówno dla skalowalności poziomej jak i pionowej, zmniejsza się najważniejszy parametr — czas



((a)) przy 50 wątkach JMetera

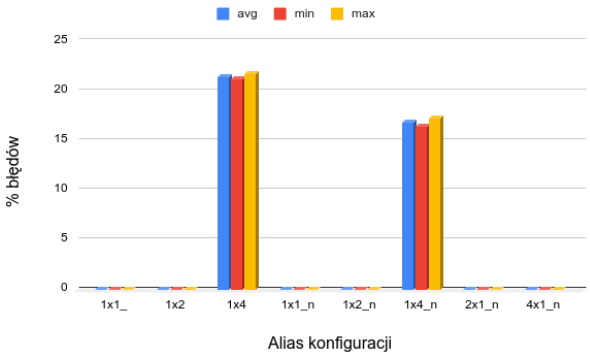


((b)) przy 100 wątkach JMetera

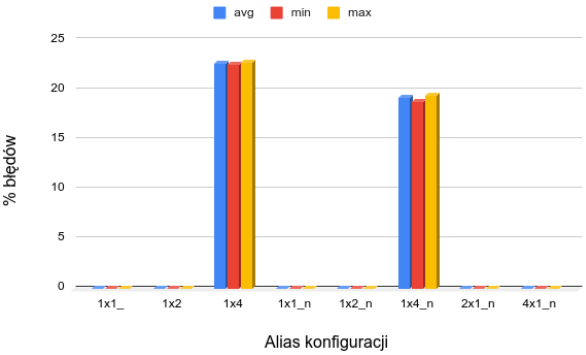


((c)) przy 200 wątkach JMetera

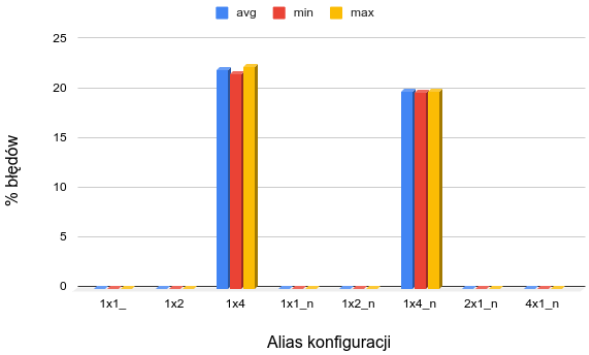
Rys. 6.12: Zależność czasu odpowiedzi usunięcia danych od konfiguracji systemu



((a)) przy 50 wątkach JMetera



((b)) przy 100 wątkach JMetera



((c)) przy 200 wątkach JMetera

Rys. 6.13: Zależność liczby błędów usunięcia danych od konfiguracji systemu

odpowiedzi. To oznacza, że komfort pracy dla użytkowników się zwiększa oraz wzrasta liczba żądań, które mogą być opracowane, o czym świadczy zwiększona liczba żądań oraz liczba żą-

dań w sekundę. Dla 50 wątków JMeter wszystkie systemy pokazały dobre wyniki, czyli średnie czasy odpowiedzi nie przekroczyły 200 ms. W przypadku 100 wątków tylko systemy używające 1 instancję backendową o 1 rdzeniu CPU mają wartość dopuszczalną czasu odpowiedzi, a nie dobrą. Natomiast dla 200 wątków dobry czas odpowiedzi pokazały następujące konfiguracje: 1 instancja o 4 rdzeniach CPU bez loadbalancer oraz 4 instancje o 1 rdzeniu CPU z loadbalancerem NGINX. Konfiguracja 1 instancji o 4 rdzeniach CPU z loadbalancerem NGINX pokazała wyniki dopuszczalne, czyli czas odpowiedzi przekroczył 200 ms. Żaden system nie pokazał wyników gorsze niż dopuszczalne, czyli powyżej 1 sekundy.

Konfiguracje backendu, używające 4 rdzenie procesora, choć pokazały lepsze wyniki czasów odpowiedzi, to jednak są nieużyteczne z powodu dużej liczby błędów, podczas gdy inne systemy pokazywały ich brak (rysunki 6.13).

Porównując systemy o takich samych zasobach CPU i RAM, ale z loadbalancerem NGINX i bez można zauważyć, że loadbalancer negatywnie wpływa na czas reakcji systemu (rysunki 6.12).

Pozostałe wnioski wpływu skalowalności poziomej oraz pionowej na wydajność systemów są analogiczne jak w punkcie poprzednim (aktualizacja danych).

Przeprowadzane testy t-Studenta (spełnione warunki: dane należą do rozkładu normalnego, jednakowa liczba próbek, podobne wariancje) wskazują na statystycznie znaczącą różnicę między systemami oraz na możliwość odrzucenia zerowej hipotezy (wartości dużo mniejsze od p=0,005) o braku wpływu skalowalności części backendowej na średni czas odpowiedzi:

- dla 50 wątków JMetera wyniki znajdują się w tabeli 6.25 na końcu rozdziału;
- dla 100 wątków JMetera wyniki znajdują się w tabeli 6.26 na końcu rozdziału;
- dla 200 wątków JMetera wyniki znajdują się w tabeli 6.27 na końcu rozdziału.

6.4.5. Interpretacja wyników oraz wnioski z badań

Przedstawione wyniki badań pokazały, na ile różnią się zastosowane metody skalowalności oraz architektury współdziałania elementów systemu. Wyniki należały do rozkładu normalnego, miały jednakową liczebność oraz podobne wariancje. Co pozwoliło zastosować do porównania średnich czasów odpowiedzi uzyskanych dla analizowanych wariantów systemu ze skalowaniem poziomym oraz pionowym, a także bez skalowania (tabele 6.7, 6.8, 6.9, 6.13, 6.14, 6.15, 6.19, 6.20, 6.21, 6.25, 6.26, 6.27) testu t-Studenta. Podczas porównywania systemów parami test t-Studenta wykazał istotne statystycznie różnice pomiędzy analizowanymi grupami wyników dotyczących średnich czasów odpowiedzi.

Testy pokazały we wszystkich rodzajach (CRUD) żądań statystycznie znacząco lepsze wyniki dla systemów skalowanych poziomo niż pionowo. W tabeli 6.3 przedstawiono średni czas odpowiedzi (porównanie czasów odpowiedzi między parami typów) w odniesieniu do konfiguracji kontrolnych (bazowych), czyli bez skalowania (1 instancja 1 rdzeń z loadbalancerem (1x1_n) i bez loadbalancera (1x1)). Jak widać, średnio, w żadnym z przypadków nie ma dwukrotnego zmniejszenia czasu odpowiedzi: ani dla skalowalności poziomej ani pionowej, mimo podwajania ilości zasobów części backendowej. Oprócz tego można zauważyć, że obecność narzędzia do równoważenia obciążenia negatywnie wpływa na czas odpowiedzi (zwiększa go). Przykłady obliczenia (tabela 6.4 i kolejne):

- Dla `1x1`: (214.33 + 433.33 + 878.67 + 133 + 261 + 506.33 + 135 + 231 + 501 + 135.67 + 256.33 + 486.33) / 12 = 347.67 ms
- Dla `1x1_n`: (247 + 483.67 1002.67 + 136.67 + 277 + 546.67 + 275.33 + 522 + 84.67 + 140 + 271.67 + 512.33) / 12 = 379,91 ms
- Wynik: 1x1 / 1x1_n * 100% = 347.67 / 379.91 * 100% = 91,51%

Tab. 6.3: Porównanie metod skalowalności odnośnie konfiguracji kontrolnych

konfiguracja	1x1	1x1_n
1x1	100%	91,51%
1x1_n	109,27%	100%
1x2	60,99%	55,82%
1x4	41,94%	38,38%
1x2_n	64.70%	59,18%
1x4_n	45,65%	41,77%
2x1_n	57,60%	52,71%
4x1_n	32,41%	29,66%

Oprócz tego, że skalowanie pionowe jest mniej wydajne niż skalowania poziome, to dla zaimplementowanych systemów zbyt duża moc pojedynczego serwera prowadzi do błędów.

Do wad skalowalności poziomej można zaliczyć większe potrzeby dotyczące pamięci RAM (choć bardzo małe) oraz niezbędność korzystania z takich systemów jak loadbalancer, które też wymagają dostępu do odpowiednich zasobów środowiska produkcyjnego, choć niewielkich w porównaniu do aplikacji.

Najbardziej obciążonymi elementami systemu podczas testów były aplikacja backendowa i baza danych PostgreSQL. Podczas każdego testu użyte zasoby CPU przez część backendową zbliżały się do przedzielonego jej maksimum. Jedynym wyjątkiem była metoda GET przy czterech instancjach, gdzie "wąskim gardłem"systemu były zasoby CPU przedzielone dla bazy danych PostgreSQL. Natomiast inne elementy systemu podczas prowadzenia testów wykorzystywały mniej zasobów (CPU i RAM), niż im przedzielono.

Tab. 6.4: Wyniki testów tworzenia wpisów dla różnych konfiguracji systemu przy obciążeniu 50 wątkami JMetera

Alias konfiguracji	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
liczba żądań	13083	21175	36710	12232,67	20879	34456,33	25678	61777,33
czas odpowiedzi (ms)	214,33	142,33	81,67	247	144,33	87	117,33	48,33
min czas odp. (ms)	4	6	5	6,33	5,67	4,33	5	4,33
max czas odp. (ms)	915	810,33	942,67	904	536,33	571,67	1001	1764,33
odchylenie standardowe wyliczone przez JMeter	125,86	78,1	47,66	112,88	255,5	48,11	61,74	35,63
odchylenie standardowe testów	3,06	0,58	0,58	2	0,58	1	0,58	0,58
wariancja testów	9,33	0,33	0,33	4	0,33	1	0,33	0,33
średnia liczba żądań w sekundę	213,63	345,63	600,93	199,93	341,7	564,13	420,2	1001,37
min liczba żądań w sekundę	58,33	173,67	317	94	133,67	206,67	115	339,33
max liczba żądań w sekundę	321,67	426,33	784	215,33	418	793,33	424,33	1085,33
średnia liczba błędów	0	0	28,9	0	0	26	0	0
min liczba błędów	0	0	28,9	0	0	25,61	0	0
max liczba błędów	0	0	28,9	0	0	26,67	0	0
backend CPU (%)	98,63	197,67	395,33	98,53	198	394,33	99	99,3
postgresql CPU (%)	129,67	214,33	259,67	123,67	209,33	240,33	242,33	488,33
nginx CPU (%)	brak	brak	brak	2,17	3,69	9,01	4,59	13,1
redis CPU (%)	0,61	0,97	1,4	0,58	0,98	1,49	1,11	2,81
backend RAM (MB)	28,80	30,37	31,30	28,57	29,37	30,43	17,9	13,9
postgresql RAM (MB)	452,33	310,2	487	477,33	493	484,67	502	528
nginx RAM (MB)	brak	brak	brak	7,79	8,43	9,86	8,39	11,26
redis RAM (MB)	17,73	17,87	17,83	17,73	17,77	17,8	17,8	17,8

Tab. 6.5: Wyniki testów tworzenia wpisów dla różnych konfiguracji systemu przy obciążeniu 100 wątkami JMetera

Alias konfiguracji	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
liczba żądań	12968	22026	36917	12541	21435,33	34354,33	24201	50036,67
czas odpowiedzi (ms)	433,33	274,67	163	483,67	281,67	125,29	250	120,33
min czas odp. (ms)	4	6,33	5,33	6,67	8	5,05	7	6
min czas odp. (ms)	1908,33	1354,33	1094,67	1861,33	1270,33	832,95	1190	1221,67
odchylenie standardowe wyliczone przez JMeter	251,06	143,82	90,84	233,65	144,14	67,66	115,43	66,13
odchylenie standardowe testów	4,04	1,53	1	3,21	0,58	2,08	1	0,58
wariancja testów	16,33	2,33	1	10,33	0,33	1	1	0,33
liczba żądań w sekundę	211	352,72	603,63	204,1	350	562,09	394,8	819,2
min liczba żądań w sekundę	105	121,33	253,6	85	186,33	230,10	80,33	491,67
max liczba żądań w sekundę	234,67	426,33	807,33	214,67	426,33	792,90	415,33	834
średnia liczba błędów	0	0	32,19	0	0	27,09	0	0
min liczba błędów	0	0	31,55	0	0	25,61	0	0
max liczba błędów	0	0	33,01	0	0	29,09	0	0
backend CPU (%)	98,73	197,67	395	98,57	198	394,48	98,8	98,5
postgresql CPU (%)	135,67	228,33	276,67	133	224,33	246,76	265	501,67
nginx CPU (%)	brak	brak	brak	2,28	3,95	12,03	4,19	10,53
redis CPU (%)	0,64	1,017	1,67	0,639	1,03	1,52	1,15	2,45
backend RAM (MB)	45,30	47,83	48,33	44,17	45,27	36,78	27,4	17,5
postgresql RAM (MB)	529	542,67	520,67	522,67	507,33	498,38	558,33	581
nginx RAM (MB)	brak	brak	brak	8,74	9,73	10,43	9,37	11,03
RAM (MB)	17,8	18,03	18,1	17,8	17,9	17,89	18,2	18,13

Tab. 6.6: Wyniki testów tworzenia wpisów dla różnych konfiguracji systemu przy obciążeniu 200 wątkami JMetera

Alias konfiguracji	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
liczba żądań	12837	21697.33	36114.67	12155	20614	34038.33	23205.33	43390.67
czas odpowiedzi (ms)	878.67	559.67	335	1002.67	588.67	358.33	522.67	278.33
min czas odp. (ms)	4.67	7	6.33	8.33	6.67	8	7.67	5.33
min czas odp. (ms)	4728	2332.67	1894.33	4611.67	2891.67	2050.33	2157	1487.33
odchylenie standardowe wyliczone przez JMeter	548.50	317.52	206.37	529.06	333.03	204.74	246.03	135.14
odchylenie standardowe testów	3,51	5,86	1	5,86	5,51	5,03	4,04	2,08
wariancja testów	12,3	34,3	1	34,3	30,33	25,33	16,33	4,33
liczba żądań w sekundę	207.90	352.83	589.43	196.40	335.37	550.63	377	705
min liczba żądań w sekundę	97.67	165	318	82	189.67	278.67	203.33	356
max liczba żądań w sekundę	223	424.67	755	215	414.67	740.33	398.54	739.33
średnia liczba błędów	0	0	32.74	0	0	29.51	0	0
min liczba błędów	0	0	32.11	0	0	27.09	0	0
max liczba błędów	0	0	33.09	0	0	32.31	0	0
backend CPU (%)	98.73	198	395	98.57	197.67	394.64	98,9	99,5
postgresql CPU (%)	136.33	233.33	259.67	129.67	223.67	283	272.67	545.33
nginx CPU (%)	brak	brak	brak	2.18	3.87	15.70	4.22	8.92
redis CPU (%)	0.66	1.05	1.67	0.62	0.97	1.54	1.12	2.18
backend RAM (MB)	68.93	72.73	68.93	66.13	54.90	65.57	45,5	25,3
postgresql RAM (MB)	592.33	611.33	565	582.67	609	558.33	644.67	705.67
nginx RAM (MB)	brak	brak	brak	10.20	11.43	12.87	11.03	16.77
RAM (MB)	18	18.33	18.50	18.23	18.30	18.43	18.30	18.70

Tab. 6.7: Test t-Studenta dla tworzenia wpisów przy 50 wątkach JMetera

	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
1x1	1,00E+00	6,82E-25	5,10E-28	6,15E-26	9,50E-25	5,56E-30	2,04E-26	3,64E-29
1x2	6,82E-25	1,00E+00	2,34E-48	1,53E-30	1,48E-16	1,14E-37	6,83E-40	1,53E-52
1x4	5,10E-28	2,25E-48	1,00E+00	4,36E-33	1,14E-48	6,13E-20	2,77E-43	1,22E-42
1x1_n	6,15E-26	4,65E-09	4,36E-33	1,00E+00	1,96E-30	2,52E-38	9,82E-32	4,14E-34
1x2_n	9,50E-25	3,40E-10	1,14E-48	1,96E-30	1,00E+00	6,09E-38	1,26E-40	9,64E-53
1x4_n	5,56E-30	5,33E-46	6,13E-20	2,52E-38	6,09E-38	1,00E+00	4,44E-33	6,22E-35
2x1_n	2,04E-26	8,49E-37	2,77E-43	9,82E-32	1,26E-40	4,44E-33	1,00E+00	1,38E-49
4x1_n	3,64E-29	2,00E-12	1,22E-42	4,14E-34	9,64E-53	6,22E-35	1,38E-49	1,00E+00

Tab. 6.8: Test t-Studenta dla tworzenia wpisów przy 100 wątkach JMetera

	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
1x1	1,00E+00	1,23E-30	3,88E-31	3,43E-28	8,38E-27	4,11E-37	4,68E-29	2,10E-30
1x2	1,23E-30	1,00E+00	1,01E-41	3,30E-36	1,31E-17	1,67E-39	2,63E-29	2,04E-36
1x4	3,88E-31	1,01E-41	1,00E+00	1,21E-34	1,68E-43	1,69E-20	1,49E-46	1,10E-35
1x1_n	3,43E-28	3,30E-36	1,21E-34	1,00E+00	8,69E-30	9,33E-44	7,66E-33	9,00E-33
1x2_n	8,38E-27	1,31E-17	1,68E-43	8,69E-30	1,00E+00	3,77E-30	2,08E-33	1,06E-57
1x4_n	4,11E-37	1,67E-39	1,69E-20	9,33E-44	3,77E-30	1,00E+00	3,61E-32	1,13E-26
2x1_n	4,68E-29	2,63E-29	1,49E-46	7,66E-33	2,08E-33	3,61E-32	1,00E+00	3,53E-44
4x1_n	2,10E-30	2,04E-36	1,10E-35	9,00E-33	1,06E-57	1,13E-26	3,53E-44	1,00E+00

Tab. 6.9: Test t-Studenta dla tworzenia wpisów przy 200 wątkach JMetera

	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
1x1	1,00E+00	4,54E-38	1,74E-36	1,44E-23	1,15E-38	1,30E-45	1,26E-46	3,04E-41
1x2	4,54E-38	1,00E+00	3,52E-27	8,98E-37	8,27E-20	8,95E-38	7,31E-22	1,90E-29
1x4	1,74E-36	3,52E-27	1,00E+00	1,88E-29	3,13E-28	1,13E-16	3,51E-29	3,89E-46
1x1_n	1,44E-23	8,98E-37	1,88E-29	1,00E+00	1,90E-35	1,69E-37	6,45E-33	2,57E-30
1x2_n	1,15E-38	8,27E-20	3,13E-28	1,90E-35	1,00E+00	9,49E-40	1,59E-27	1,35E-30
1x4_n	1,30E-45	8,95E-38	1,13E-16	1,69E-37	9,49E-40	1,00E+00	5,66E-37	5,40E-24
2x1_n	1,26E-46	7,31E-22	3,51E-29	6,45E-33	1,59E-27	5,66E-37	1,00E+00	4,35E-33
4x1_n	3,04E-41	1,90E-29	3,89E-46	2,57E-30	1,35E-30	5,40E-24	4,35E-33	1,00E+00

Tab. 6.10: Wyniki testów pobierania danych dla różnych konfiguracji systemu przy obciążeniu 50 wątkami JMetera

Alias konfiguracji	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
liczba żądań	22563.67	38213.33	46092.67	21869.33	35794.67	44419	43584	52218.33
czas odpowiedzi (ms)	133	78.33	64.67	137.67	83.67	67.33	71,67	57.33
min czas odp. (ms)	3.33	2.67	2.33	2.67	2.33	2.67	2.33	2.67
min czas odp. (ms)	574.33	931.67	988.67	995.33	980.33	752.33	1024,33	2101
odchylenie standardowe wyliczone przez JMeter	73.75	51.54	44.12	73.67	54.60	39.60	46,32	54.90
odchylenie standardowe testów	2	0,58	0,58	0,58	1,15	0,58	3	0,58
wariancja testów	4	0,33	0,33	0,33	1,33	0,33	9	0,33
liczba żądań w sekundę	369.07	626.47	754.37	357.97	583.07	727.37	708,8	838.67
min liczba żądań w sekundę	202.77	242.67	457.17	152.67	265	313	277,33	291.33
max liczba żądań w sekundę	393.67	759	805.33	374	723.67	751,33	755	887.33
średnia liczba błędów	0	0	0	0	0	0	0	0
min liczba błędów	0	0	0	0	0	0	0	0
max liczba błędów	0	0	0	0	0	0	0	0
backend CPU (%)	97.50	197	230.67	98.67	197,33	289	98,5	44,9
postgresql CPU (%)	341	632.33	729.61	316.33	566,67	773.33	695	786
nginx CPU (%)	brak	brak	brak	3.73	6,79	25.23	8.70	11.23
redis CPU (%)	0.93	1.65	1.92	0.90	1.52	2.06	1.90	2.42
backend RAM (MB)	25.67	26.40	43.74	25.07	25,93	24.33	17,4	12,9
postgresql RAM (MB)	542	539.67	602.67	523.67	537,67	540.33	545.67	567.67
nginx RAM (MB)	brak	brak	brak	8.23	9,56	10.50	9.62	11.87
RAM (MB)	18	17.87	18	17.77	17,8	17.75	17.87	14.80

Tab. 6.11: Wyniki testów pobierania danych dla różnych konfiguracji systemu przy obciążeniu 100 wątkami JMetera

Alias konfiguracji	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
liczba żądań	23162	37700.67	45310.67	21823.33	35556	46886.33	42158	52840
czas odpowiedzi (ms)	261	159.67	132.67	277	169.67	139.33	144	113.67
min czas odp. (ms)	3.33	2.67	3	4	4.33	3.33	3.33	2.67
min czas odp. (ms)	1459	1222.33	1002.33	1295.67	1114	833	1384.67	1107
odchylenie standardowe wyliczone przez JMeter	148.91	98.59	95.27	142.17	96.25	85.97	87.42	106.43
odchylenie standardowe testów	1,73	1,53	1,53	1	1,53	0,58	1	0,58
wariancja testów	3	2,33	2,33	1	2,33	0,33	1	0,33
liczba żądań w sekundę	377.70	614.03	741.20	355.77	578.60	707.03	679.67	864.80
min liczba żądań w sekundę	167.67	254	451.67	110.67	234.33	282.67	324.33	420.67
max liczba żądań w sekundę	403.67	730	795	373.67	690.67	768	708	914
średnia liczba błędów	0	0	0	0	0	1.63	0	0
min liczba błędów	0	0	0	0	0	0.92	0	0
max liczba błędów	0	0	0	0	0	2.30	0	0
backend CPU (%)	98.90	196.67	267	98.58	198	290.67	98,2	45,1
postgresql CPU (%)	383.33	669	789	329.89	624	771.22	712	780.17
nginx CPU (%)	brak	brak	brak	3.71	7.72	26.66	8.51	11.32
redis CPU (%)	1.01	1.71	2.15	0.92	1.62	2.02	1.93	2.46
backend RAM (MB)	38.40	40.17	37.30	31.99	39.97	29.87	24,3	15,9
postgresql RAM (MB)	570.67	593.67	623.33	544.28	584	570.22	578.28	612.94
nginx RAM (MB)	brak	brak	brak	8.81	10.87	14.10	10.28	12.21
RAM (MB)	18	18.13	17.90	17.89	21.27	17.75	17.94	15.68

Tab. 6.12: Wyniki testów pobierania danych dla różnych konfiguracji systemu przy obciążeniu 200 wątkami JMetera

Alias konfiguracji	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
liczba żądań	23964.67	37790.33	44981.33	22215	35030	41994	40411.33	52348.33
czas odpowiedzi (ms)	506.33	323	268.67	545.67	345.67	288	299.33	231.33
min czas odp. (ms)	4	3.33	2.33	4.33	4.33	3.33	3.33	3
min czas odp. (ms)	3031	1966	1970	2387	1731.67	1949	1543	2418.67
odchylenie standardowe wyliczone przez JMeter	336.45	214.50	207.13	306.44	202.47	187.10	180.73	240.40
odchylenie standardowe testów	2,52	5,56	3,05	2,52	1,15	1	1,154	4,04
wariancja testów	6,33	31	9,33	6,33	1,33	1	1,33	16,33
liczba żądań w sekundę	388.57	614.23	733.50	359.97	533.50	684.33	651.50	854.13
min liczba żądań w sekundę	221.33	339.23	372	167.67	223.67	293.67	246.67	448.67
max liczba żądań w sekundę	418.67	728	775	388.33	680.33	766	685.67	893
średnia liczba błędów	0	0	0	0	0	3.50	0	0
min liczba błędów	0	0	0	0	0	3.33	0	0
max liczba błędów	0	0	0	0	0	3.79	0	0
backend CPU (%)	99.03	196	260	98.27	197	281.33	98,5	43,1
postgresql CPU (%)	411.67	670.67	788.67	373.67	636.33	761	744	786.67
nginx CPU (%)	brak	brak	brak	3.90	11.87	27.77	7.62	11.43
redis CPU (%)	1.06	1.73	2.11	0.99	1.58	1.89	1.83	2.47
backend RAM (MB)	62.23	61.33	55.43	57.97	60.03	53.97	36,5	20,8
postgresql RAM (MB)	621.33	662	718.33	604	652.67	692	706.67	759.67
nginx RAM (MB)	brak	brak	brak	11.07	12.80	13.63	12.57	14.60
RAM (MB)	18.70	18.30	19.07	18.63	18.60	18.27	18.53	17.87

Tab. 6.13: Test t-Studenta dla pobierania danych przy 50 wątkach JMetera

	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
1x1	1,00E+00	6,30E-27	3,61E-28	1,56E-13	1,68E-31	6,01E-28	2,08E-21	9,78E-29
1x2	6,30E-27	1,00E+00	3,90E-34	3,81E-48	1,91E-18	4,52E-32	5,89E-10	3,15E-38
1x4	3,61E-28	1,04E-26	1,00E+00	3,99E-50	3,16E-27	5,66E-19	4,00E-10	3,11E-28
1x1_n	1,56E-13	1,28E-08	3,99E-50	1,00E+00	1,49E-34	9,04E-50	3,34E-20	4,86E-51
1x2_n	1,68E-31	6,36E-10	3,16E-27	1,49E-34	1,00E+00	3,61E-26	2,48E-12	1,63E-29
1x4_n	6,01E-28	2,82E-33	5,66E-19	9,04E-50	3,61E-26	1,00E+00	1,12E-08	3,62E-31
2x1_n	2,08E-21	5,32E-22	4,00E-10	3,34E-20	2,48E-12	1,12E-08	1,00E+00	5,25E-13
4x1_n	9,78E-29	1,19E-11	3,11E-28	4,86E-51	1,63E-29	3,62E-31	5,25E-13	1,00E+00

Tab. 6.14: Test t-Studenta dla pobierania danych do bazy przy 100 wątkach JMetera

	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
1x1	1,00E+00	7,54E-43	4,53E-45	4,89E-24	7,15E-42	3,88E-33	3,40E-39	2,98E-34
1x2	7,54E-43	1,00E+00	2,33E-31	3,93E-42	4,47E-22	4,91E-24	1,34E-25	5,12E-29
1x4	4,53E-45	2,33E-31	1,00E+00	7,75E-44	2,36E-34	2,53E-17	5,60E-23	1,27E-23
1x1_n	4,89E-24	3,93E-42	7,75E-44	1,00E+00	2,13E-41	1,23E-44	1,31E-50	6,07E-46
1x2_n	7,15E-42	4,47E-22	2,36E-34	2,13E-41	1,00E+00	1,79E-26	1,24E-29	3,22E-30
1x4_n	3,88E-33	4,91E-24	2,53E-17	1,23E-44	1,79E-26	1,00E+00	5,80E-19	3,83E-40
2x1_n	3,40E-39	1,34E-25	5,60E-23	1,31E-50	1,24E-29	5,80E-19	1,00E+00	4,44E-33
4x1_n	2,98E-34	5,12E-29	1,27E-23	6,07E-46	3,22E-30	3,83E-40	4,44E-33	1,00E+00

Tab. 6.15: Test t-Studenta dla pobierania danych do bazy przy 200 wątkach JMetera

200	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
1x1	1,00E+00	4,51E-31	4,08E-45	3,41E-30	1,18E-35	5,35E-36	2,37E-37	1,25E-56
1x2	4,51E-31	1,00E+00	3,86E-24	2,30E-32	4,66E-16	4,08E-18	2,83E-16	8,31E-28
1x4	4,08E-45	3,86E-24	1,00E+00	1,58E-46	6,26E-28	4,07E-19	2,55E-22	1,02E-30
1x1_n	3,41E-30	2,30E-32	1,58E-46	1,00E+00	4,04E-37	4,93E-37	1,62E-38	3,24E-58
1x2_n	1,18E-35	4,66E-16	6,26E-28	4,04E-37	1,00E+00	2,68E-41	3,64E-39	5,18E-41
1x4_n	5,35E-36	4,08E-18	4,07E-19	4,93E-37	2,68E-41	1,00E+00	3,75E-26	1,92E-33
2x1_n	2,37E-37	2,83E-16	2,55E-22	1,62E-38	3,64E-39	3,75E-26	1,00E+00	3,57E-36
4x1_n	1,25E-56	8,31E-28	1,02E-30	3,24E-58	5,18E-41	1,92E-33	3,57E-36	1,00E+00

Tab. 6.16: Wyniki testów aktualizowania danych dla różnych konfiguracji systemu przy obciążeniu 50 wątkami JMetera

Alias konfiguracji	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
liczba żądań	22816.33	37351.67	57637.67	20932.33	35246	52609.67	43433.33	102961.67
czas odpowiedzi (ms)	135	80.33	51.33	144	84.67	56.67	68.67	28.67
min czas odp. (ms)	4	4	3.67	4	4.67	4	4	4
min czas odp. (ms)	964	704.33	830.67	633.67	877	711	744.67	2443.33
odchylenie standardowe wyliczone przez JMeter	75.72	51.26	42.79	73.22	52.55	45.63	41.39	24.47
odchylenie standardowe testów	3,36	0,58	0,58	1,73	0,58	0,58	0,58	0,58
wariancja testów	19	0,33	0,33	3	0,33	0,33	0,33	0,33
liczba żądań w sekundę	362.07	611.63	946.33	342.10	574.17	861.77	711.10	1673
min liczba żądań w sekundę	74.93	247.33	351	91	228.67	148.67	298.67	566.33
max liczba żądań w sekundę	406.67	765	1510	366.67	728.33	5766.67	751.67	1153.92
średnia liczba błędów	0	0	20.41	0	0	15.70	0	0
min liczba błędów	0	0	20.12	0	0	15.12	0	0
max liczba błędów	0	0	20.62	0	0	16	0	0
backend CPU (%)	97.70	199.67	401	99.27	200	400	99.5	99.5
postgresql CPU (%)	103.33	177.67	201	105.67	167	215.33	200.67	399.33
nginx CPU (%)	brak	brak	brak	3.25	5.51	23.23	6.93	17.70
redis CPU (%)	0.81	1.28	2.01	0.80	1.26	1.95	1.52	3.72
backend RAM (MB)	25.13	27.13	30.20	25.10	25.77	28.13	16.9	13.4
postgresql RAM (MB)	467.33	498	507.67	463	496.33	504.67	489	524.67
nginx RAM (MB)	brak	brak	brak	8.19	8.89	16.27	8.58	10.70
RAM (MB)	18	17.93	17.97	17.83	17.87	17.80	17.83	17.90

Tab. 6.17: Wyniki testów aktualizowania danych dla różnych konfiguracji systemu przy obciążeniu 100 wątkami JMetera

Alias konfiguracji	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
liczba żądań	23312	40221	58036.33	21974.67	37491	51937	41713	84736.67
czas odpowiedzi (ms)	231	150	103.67	275.33	160.67	115.67	143.67	70.33
min czas odp. (ms)	4.67	4.33	3.67	5	5.67	4.67	5.33	5.67
min czas odp. (ms)	1244.67	733.33	824.67	1317.33	897	991.33	1037.33	1180.67
odchylenie standardowe wyliczone przez JMeter	137.77	86.70	79.90	146.13	87.72	83.50	74.87	41.77
odchylenie standardowe testów	4,16	1	1,53	4,16	1,15	1,15	0,58	0,58
wariancja testów	17,33	1	2,33	17,33	1,33	1,33	0,33	0,33
liczba żądań w sekundę	380.30	657.17	949.33	358.47	613.17	848.80	681.70	1383.57
min liczba żądań w sekundę	184.40	278.67	344.33	170	235	277	245.33	658
max liczba żądań w sekundę	416.67	780.67	1506.67	395	765	1410	730	1450
średnia liczba błędów	0	0	22.00	0	0	18.24	0	0
min liczba błędów	0	0	21.71	0	0	17.40	0	0
max liczba błędów	0	0	22.36	0	0	19.29	0	0
backend CPU (%)	98.63	199.33	397.67	98.77	199.33	398.33	99.4	99.3
postgresql CPU (%)	111.67	194	234	109.33	187	228.33	219.67	419
nginx CPU (%)	brak	brak	brak	3.35	8.78	36.10	6.88	15.83
redis CPU (%)	0.95	1.51	2.35	0.86	1.48	2.09	1.59	3.36
backend RAM (MB)	36.70	40.40	43.33	37.87	40.23	40.17	24.9	17.1
postgresql RAM (MB)	501.67	539.67	528.33	495	526.67	524.67	540.33	571
nginx RAM (MB)	brak	brak	brak	9.51	10.90	11.87	10.20	11.80
RAM (MB)	18	18.27	18.40	18.13	18.37	18.33	18.33	18.50

Tab. 6.18: Wyniki testów aktualizowania danych dla różnych konfiguracji systemu przy obciążeniu 200 wątkami JMetera

Alias konfiguracji	1x1 avg	1x2 avg	1x4 avg	1x1_n avg	1x2_n avg	1x4_n avg	2x1_n avg	4x1_n avg
liczba żądań	24228	42188	59251.67	23198.67	39549.33	52723.33	41187.33	75619
czas odpowiedzi (ms)	501	285.67	203.67	522	305.67	229	293.67	159
min czas odp. (ms)	6	5.33	3.67	6.67	5.33	5.67	6	5.67
min czas odp. (ms)	2877	1517.33	1857	2676	1527.33	1987.33	1570.33	1500.33
odchylenie standardowe wyliczone przez JMeter	307.86	182.96	175.79	285.53	172.43	178.06	147.10	81.46
odchylenie standardowe testów	5,20	0,58	2,08	4,00	3,05	1,00	0,58	1,73
wariancja testów	27	0,33	4,33	16	9,33	1	0,33	3
liczba żądań w sekundę	393.07	688.53	968.33	378.10	645.07	860.80	671.07	1227
min liczba żądań w sekundę	169.20	274.33	343	157.33	244.67	193.33	305.67	603
max liczba żądań w sekundę	436.33	811	1039.67	413	776.33	947.12	724.33	1280
średnia liczba błędów	0	0	23.01	0	0	19.15	0	0
min liczba błędów	0	0	22.94	0	0	18.97	0	0
max liczba błędów	0	0	23.10	0	0	19.28	0	0
backend CPU (%)	98.70	199.67	398	98.87	199.67	395	99	9.,9
postgresql CPU (%)	113.33	198.33	219	112	192	231	226	463.33
nginx CPU (%)	brak	brak	brak	3.57	16.27	41.90	6.71	14
redis CPU (%)	0.94	1.68	2.43	0.96	1.58	2.09	1.64	3.17
backend RAM (MB)	56.10	59.53	64.67	53.67	55.67	56.43	38.4	23.9
postgresql RAM (MB)	555.67	604.33	563.67	539.67	722.33	549.33	620	679
nginx RAM (MB)	brak	brak	brak	11.50	14.87	14.27	12.37	13.60
RAM (MB)	18.77	18.63	18.93	18.70	18.73	18.77	18.90	19

Tab. 6.19: Test t-Studenta dla aktualizowania danych przy 50 wątkach JMetera

	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
1x1	1,00E+00	2,66E-21	2,11E-23	3,67E-13	6,78E-21	4,46E-23	2,95E-22	1,38E-24
1x2	2,66E-21	1,00E+00	2,62E-41	2,29E-29	2,51E-23	2,28E-39	1,25E-32	7,99E-47
1x4	2,11E-23	1,88E-41	1,00E+00	1,50E-31	1,22E-42	3,01E-25	2,12E-36	5,88E-39
1x1_n	3,67E-13	9,95E-10	1,50E-31	1,00E+00	5,90E-29	3,31E-31	2,40E-30	7,95E-33
1x2_n	6,78E-21	6,16E-10	1,22E-42	5,90E-29	1,00E+00	5,66E-41	1,23E-35	1,36E-47
1x4_n	4,46E-23	1,15E-41	3,01E-25	3,31E-31	5,66E-41	1,00E+00	6,73E-33	5,66E-41
2x1_n	2,95E-22	4,05E-51	2,12E-36	2,40E-30	1,23E-35	6,73E-33	1,00E+00	2,22E-44
4x1_n	1,38E-24	1,07E-12	5,88E-39	7,95E-33	1,36E-47	5,66E-41	2,22E-44	1,00E+00

Tab. 6.20: Test t-Studenta dla aktualizowania danych przy 100 wątkach JMetera

	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
1x1	1,00E+00	6,50E-14	5,20E-16	2,17E-11	2,76E-13	1,55E-15	3,01E-14	4,50E-17
1x2	6,50E-14	1,00E+00	1,75E-34	9,22E-27	1,35E-25	1,89E-36	3,29E-21	1,86E-40
1x4	5,20E-16	1,75E-34	1,00E+00	1,05E-30	1,02E-37	5,71E-24	3,66E-28	4,75E-27
1x1_n	2,17E-11	9,22E-27	1,05E-30	1,00E+00	8,42E-27	1,27E-28	6,43E-26	4,10E-28
1x2_n	2,76E-13	1,35E-25	1,02E-37	8,42E-27	1,00E+00	6,91E-39	1,89E-26	3,62E-38
1x4_n	1,55E-15	1,89E-36	5,71E-24	1,27E-28	6,91E-39	1,00E+00	6,05E-30	2,51E-33
2x1_n	3,01E-14	3,29E-21	3,66E-28	6,43E-26	1,89E-26	6,05E-30	1,00E+00	3,61E-50
4x1_n	4,50E-17	1,86E-40	4,75E-27	4,10E-28	3,62E-38	2,51E-33	3,61E-50	1,00E+00

Tab. 6.21: Test t-Studenta dla aktualizowania danych przy 200 wątkach JMetera

	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
1x1	1,00E+00	4,87E-27	1,78E-33	2,31E-18	6,11E-35	5,10E-29	7,46E-27	7,13E-31
1x2	4,87E-27	1,00E+00	9,03E-29	4,49E-29	2,29E-18	7,48E-38	4,69E-29	3,34E-52
1x4	1,78E-33	9,03E-29	1,00E+00	7,26E-39	1,02E-35	7,46E-25	2,77E-29	2,46E-31
1x1_n	2,31E-18	4,49E-29	7,26E-39	1,00E+00	4,04E-41	1,14E-31	6,66E-29	4,60E-34
1x2_n	6,11E-35	2,29E-18	1,02E-35	4,04E-41	1,00E+00	4,78E-27	8,23E-16	1,44E-32
1x4_n	5,10E-29	7,48E-38	7,46E-25	1,14E-31	4,78E-27	1,00E+00	7,32E-39	4,98E-51
2x1_n	7,46E-27	4,69E-29	2,77E-29	6,66E-29	8,23E-16	7,32E-39	1,00E+00	8,20E-53
4x1_n	7,13E-31	3,34E-52	2,46E-31	4,60E-34	1,44E-32	4,98E-51	8,20E-53	1,00E+00

Tab. 6.22: Wyniki testów aktualizowania danych dla różnych konfiguracji systemu przy obciążeniu 50 wątkami JMetera

Alias konfiguracji	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
liczba żądań	22062	42832	57645.33	21337	35926.33	53344	44743	106428.33
czas odpowiedzi (ms)	135.67	77	50.33	140	82.33	55	65.67	25.33
min czas odp. (ms)	4	4	3.67	4	3.67	4	3.67	3.67
min czas odp. (ms)	1295.67	548.33	1049	623	376.33	745	600.67	1165
odchylenie standardowe wyliczone przez JMeter	72.85	48.50	41.07	72.53	51.89	43.94	41.04	22.74
odchylenie standardowe testów	3,21	2,65	0,58	2	1,53	1,00	0,58	0,58
wariancja testów	10,33	7	0,33	4	2,33	1,00	0,33	0,33
liczba żądań w sekundę	354.77	663.03	945.00	349.07	521.57	873.53	732.27	1772.93
min liczba żądań w sekundę	165.13	320	241.67	151.67	166.67	267	316.33	817.67
max liczba żądań w sekundę	405.80	780.33	1566.67	379.33	751.33	1475	774.67	1903.33
średnia liczba błędów	0	0	21.51	0	0	16.85	0	0
min liczba błędów	0	0	21.21	0	0	16.45	0	0
max liczba błędów	0	0	21.72	0	0	17.25	0	0
backend CPU (%)	98.97	197	394	98.20	198	398.33	99.6	99.5
postgresql CPU (%)	99.93	163.33	179.33	68.23	155.33	172	191.50	379.33
nginx CPU (%)	brak	brak	brak	3.31	5.51	92.03	7.60	18.73
redis CPU (%)	0.77	1.31	1.93	0.80	1.26	1.86	1.54	3.88
backend RAM (MB)	22.73	25.33	27.70	23.47	25.03	27.83	16.4	13.5
postgresql RAM (MB)	404.67	403	393.67	405.33	404	397.33	401	418
nginx RAM (MB)	brak	brak	brak	8.09	8.72	10.37	16.20	11.07
RAM (MB)	17.97	17.97	17.97	17.80	17.90	17.83	17.90	17.90

Tab. 6.23: Wyniki testów usunięcia danych dla różnych konfiguracji systemu przy obciążeniu 100 wątkami JMetera

Alias konfiguracji	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
liczba żądań	23403.33	44902.67	58778.67	22140.33	38120	53079.33	42459.33	86685.67
czas odpowiedzi (ms)	256.33	143.67	100.33	271.67	156.33	112	140.67	67.33
min czas odp. (ms)	5	4	3	5.33	4.33	5	4.67	4.67
min czas odp. (ms)	1296	642	968	1103.33	968	1262.33	857.67	977.33
odchylenie standardowe wyliczone przez JMeter	137.09	85.08	79.25	135.26	84.74	80.43	74.30	43.89
odchylenie standardowe testów	6,35	0,58	0,58	1,15	0,58	1,15	0,58	1,15
wariancja testów	40,33	0,33	0,33	1,33	0,33	1,33	0,33	3,33
liczba żądań w sekundę	382	652.10	957.73	361.40	623.43	4144	693.40	1419.13
min liczba żądań w sekundę	187.83	370.33	348.37	174.67	242.67	292	295.67	304.33
max liczba żądań w sekundę	418	828.33	1600	392.33	761	1435	752.33	1453.33
średnia liczba błędów	0	0	22.71	0	0	19.23	0	0
min liczba błędów	0	0	22.57	0	0	18.88	0	0
max liczba błędów	0	0	22.84	0	0	19.52	0	0
backend CPU (%)	98.60	197	393.67	98.03	197.67	398	99.3	98.9
postgresql CPU (%)	107	180	212.33	102.67	174.67	177.33	207.67	395
nginx CPU (%)	brak	brak	brak	3.33	8.20	38.60	7.19	16.07
redis CPU (%)	0.86	1.52	2.29	0.86	1.45	1.93	1.57	3.37
backend RAM (MB)	35.70	37.27	41.13	35.90	36.83	39.53	23.8	16.9
postgresql RAM (MB)	443.33	446.67	420.67	440.33	439	411	454	456.67
nginx RAM (MB)	brak	brak	brak	9.48	10.87	12.13	15074.71	11.77
RAM (MB)	18.20	18.13	18.30	18.17	18.17	18.23	18.33	18.40

Tab. 6.24: Wyniki testów usunięcia danych dla różnych konfiguracji systemu przy obciążeniu 200 wątkami JMetera

Alias konfiguracji	1x1 avg	1x2 avg	1x4 avg	1x1_n avg	1x2_n avg	1x4_n avg	2x1_n avg	4x1_n avg
liczba żądań	24734.33	44353	60885.67	23610.67	40796.33	53866	42084.67	78112.67
czas odpowiedzi (ms)	486.33	270.33	194.67	512.33	294.67	221.33	285.67	152
min czas odp. (ms)	5	4.33	4	6.33	5	5.33	5.33	5
min czas odp. (ms)	2574.67	1523.33	1955.67	2504.67	1343	2247.33	1341.33	1195.33
odchylenie standardowe wyliczone przez JMeter	306.84	176.82	174.42	282.03	168.07	172.53	147.80	79.96
odchylenie standardowe testów	3,79	0,58	1,15	2,31	2,31	0,58	1,15	1,73
wariancja testów	14,33	0,33	1,33	5,33	5,33	0,33	1,33	3,00
liczba żądań w sekundę	319.87	723.43	996.03	384.07	665.93	869.63	685.73	1276.13
min liczba żądań w sekundę	206.67	345	394	195.33	218	161.33	316.67	620
max liczba żądań w sekundę	451	862.67	1556.67	423.67	813	1465	731.33	1323.33
średnia liczba błędów	0	0	22.08	0	0	19.86	0	0
min liczba błędów	0	0	21.70	0	0	19.77	0	0
max liczba błędów	0	0	22.41	0	0	19.91	0	0
backend CPU (%)	98.73	197	393	98.30	198	398	98.9	98.9
postgresql CPU (%)	106.33	182.67	191.33	104	176.67	210	212.33	432.67
nginx CPU (%)	brak	brak	brak	3.53	16.97	42.37	6.81	14.37
redis CPU (%)	0.96	1.69	2.39	0.92	1.55	2.10	1.62	3.19
backend RAM (MB)	54.97	57.87	60.87	51.63	53.53	56.13	38.4	23.8
postgresql RAM (MB)	484	510.33	456	487.33	492.67	437.33	518	563
nginx RAM (MB)	brak	brak	brak	11.10	13.23	14.40	12.63	13.77
RAM (MB)	18.73	18.80	18.93	18.53	18.70	18.67	18.83	18.73

Tab. 6.25: Test t-Studenta dla usunięcia danych przy 50 wątkach JMetera

	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
1x1	1,00E+00	8,98E-32	2,09E-25	4,64E-11	6,86E-27	7,36E-27	2,12E-24	1,03E-26
1x2	8,98E-32	1,00E+00	9,05E-21	9,93E-34	2,76E-13	1,90E-21	2,33E-16	3,22E-24
1x4	2,09E-25	9,94E-23	1,00E+00	1,11E-29	8,43E-27	3,10E-17	3,13E-35	6,83E-40
1x1_n	4,64E-11	5,88E-10	1,11E-29	1,00E+00	1,66E-35	5,77E-34	1,23E-28	4,75E-31
1x2_n	6,86E-27	7,73E-10	8,43E-27	1,66E-35	1,00E+00	1,92E-30	7,96E-23	2,51E-30
1x4_n	7,36E-27	2,33E-41	3,10E-17	5,77E-34	1,92E-30	1,00E+00	8,38E-26	1,20E-32
2x1_n	2,12E-24	1,05E-50	3,13E-35	1,23E-28	7,96E-23	8,38E-26	1,00E+00	1,85E-44
4x1_n	1,03E-26	7,38E-13	6,83E-40	4,75E-31	2,51E-30	1,20E-32	1,85E-44	1,00E+00

Tab. 6.26: Test t-Studenta dla usunięcia danych przy 100 wątkach JMetera

	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
1x1	1,00E+00	8,61E-23	2,27E-24	1,90E-13	3,26E-22	1,16E-24	6,42E-23	5,20E-26
1x2	8,61E-23	1,00E+00	3,82E-45	1,29E-40	2,06E-33	5,92E-31	5,27E-20	5,51E-37
1x4	2,27E-24	3,82E-45	1,00E+00	1,15E-42	1,36E-47	2,06E-23	1,85E-44	4,26E-31
1x1_n	1,90E-13	1,29E-40	1,15E-42	1,00E+00	6,95E-40	5,08E-51	8,86E-41	2,45E-53
1x2_n	3,26E-22	2,06E-33	1,36E-47	6,95E-40	1,00E+00	2,83E-33	1,95E-35	4,60E-38
1x4_n	1,16E-24	5,92E-31	2,06E-23	5,08E-51	2,83E-33	1,00E+00	2,85E-30	1,13E-38
2x1_n	6,42E-23	5,27E-20	1,85E-44	8,86E-41	1,95E-35	2,85E-30	1,00E+00	1,05E-36
4x1_n	5,20E-26	5,51E-37	4,26E-31	2,45E-53	4,60E-38	1,13E-38	1,05E-36	1,00E+00

Tab. 6.27: Test t-Studenta dla usunięcia danych przy 200 wątkach JMetera

	1x1	1x2	1x4	1x1_n	1x2_n	1x4_n	2x1_n	4x1_n
1x1	1,00E+00	5,52E-29	4,85E-33	4,63E-22	9,63E-38	5,25E-30	6,34E-31	2,52E-34
1x2	5,52E-29	1,00E+00	6,35E-37	1,36E-33	2,87E-21	2,56E-46	9,97E-26	1,59E-51
1x4	4,85E-33	6,35E-37	1,00E+00	3,93E-42	5,16E-34	1,33E-29	1,31E-45	9,43E-44
1x1_n	4,63E-22	1,36E-33	3,93E-42	1,00E+00	2,55E-47	1,39E-34	9,22E-40	7,26E-45
1x2_n	9,63E-38	2,87E-21	5,16E-34	2,55E-47	1,00E+00	3,51E-27	2,59E-17	5,78E-38
1x4_n	5,25E-30	2,56E-46	1,33E-29	1,39E-34	3,51E-27	1,00E+00	8,76E-36	3,38E-46
2x1_n	6,34E-31	9,97E-26	1,31E-45	9,22E-40	2,59E-17	8,76E-36	1,00E+00	4,22E-57
4x1_n	2,52E-34	1,59E-51	9,43E-44	7,26E-45	5,78E-38	3,38E-46	4,22E-57	1,00E+00

Rozdział 7

Podsumowanie

Celem pracy była ocena i analiza porównawcza wydajności skalowalnych systemów webowych typu komunikator implementowanych w środowisku rozproszonym. W ramach pracy zbadano wpływ architektury systemu, tj. metod skalowalności poziomej (parametrem zmiennym była m.in. liczba instancji aplikacji z równoważeniem obciążenia w oparciu o algorytm Least Connections) oraz metod skalowalności pionowej (parametrem zmiennym była m. in. liczba rdzeni CPU serwerów aplikacji) na wydajność operacji przetwarzania wiadomości przez komunikator internetowy zrealizowany z użyciem technologii języka Go. W szczególności, oceniono wydajność średnich czasów odpowiedzi dla wybranych operacji CRUD w zależności od konfiguracji systemu ze skalowalnością poziomą (konfiguracje oznaczane jako a x b_n) oraz pionową (konfiguracje oznaczane jako a x b). Wybrane operacje CRUD mają swoją logikę biznesową odpowiadającą potrzebom systemu typu komunikator. Wyniki otrzymane dla analizowanych wariantów rozwiązań architektonicznych porównano pomiędzy sobą na podstawie czasów odpowiedzi aplikacji uzyskanych z zapytań generowanych losowo przez narzędzie JMeter w ustalonym przedziale czasu. Rezultaty odniesiono również do wyników uzyskanych dla konfiguracji bazowych z jednym rdzeniem CPU i bez loadbalancera (konfiguracja 1x1) oraz z jednym rdzeniem CPU i z loadbalancerem (konfiguracja 1x1_n). Wnioski wyciągnięto na podstawie parametrów statystycznych wygenerowanych podczas symulacji komputerowych za pomocą programu JMeter, a także na podstawie testów statystycznych t-Studenta potwierdzających wiarygodność wyników dla zadanego poziomu istotności p = 0.05, przeprowadzonych parami dla analizowanych wariantów architektonicznych systemu komunikatora webowego. Przedstawione wyniki rozszerzają stan badań w dziedzinie oceny wpływu skalowalności na wydajność rozproszonych aplikacji typu komunikator internetowy opracowanych z użyciem technologii języka Go, dla których analogicznych wyników niewiele jest znanych w literaturze.

Testy porównawcze wykazały, że zastosowanie obu typów skalowalności znacząco polepsza wyniki dotyczące średnich czasów odpowiedzi systemu, chociaż skalowalność pozioma prowadziła do nieco lepszych wyników.

Skalowalność pionowa nie wymaga dodatkowych elementów systemu, np. loadbalancera, przez co architektura ta jest łatwiejsza w implementacji. Jednak taki rodzaj skalowalności wymaga możliwości zwiększenia zasobów (np. liczby rdzeni CPU) jednego serwera. Z kolei skalowalność pozioma oprócz zwiększenia wydajności może powodować większą odporność na awarie i większą elastyczność systemu. Jednak odbywa się to kosztem rozbudowanej struktury systemu (np. użycia loadbalancera) oraz wykorzystywania dodatkowych zasobów na jej utrzymanie. Natomiast architektura przeznaczona dla skalowalności pionowej nie wymaga loadblancera, który nieco zmniejsza wydajność systemu. Z drugiej strony architektura pionowa bez modułu równoważenia obciążenia jest mniej odporna na awarie i ma pojedynczy punkt awarii. Taki system będzie miał również ograniczoną skalowalność, tj. dostępna jest tylko skalowalność pionowa. System równoważenia obciążenia może także pomóc w poprawie dostępności i skalowal-

ności systemu, ale zwiększa za to złożoność architektury. Innym ważnym aspektem jest to, że dzięki skalowalności poziomej oraz loadbalancerowi, który stanowi element takich systemów, stają się one bardziej odporne na awarie, ponieważ w przypadku awarii jednego serwera aplikacji system równoważenia obciążenia może automatycznie przekierować ruch na inne serwery.

Z powodu ograniczoności zasobów urządzeń, na których zostały zainstalowane badane systemy, nie da się jednoznacznie stwierdzić jaka jest wydajność rzeczywistych systemów, których używają miliony użytkowników. Oprócz tego, użyta biblioteka do komunikacji z bazą danych PostgreSQL z poziomu języka Go ma swoje ograniczenia, które nie dało się obejść w ramach niniejszej pracy. Podsumowując, w pracy wykazano, że zastosowanie skalowania systemu poziomo bądź pionowo zwiększa jego wydajność i jest to wpływ istotny statystycznie, jak pokazują porównania średnich czasów odpowiedzi pomiędzy wybranymi rozwiązaniami oraz rozwiązaniami bazowymi bez skalowalności. Wnioski te potwierdzają, że cel pracy został osiągnięty.

Literatura

[1] Scalability in Cloud Computing: Horizontal vs. Vertical Scaling. https://www.stormit.cloud/blog/scalability-in-cloud-computing-horizontal-vs-vertical-scaling/ [dostęp dnia 20 listopada 2022].

[2] What Is Load Balancing? https://www.nginx.com/resources/glossary/load-balancing/ [dostęp dnia 20 listopada 2022].

[3] Scaling Horizontally vs. Scaling Vertically, Lip. 2020. https://www.section.io/blog/scaling-horizontally-vs-vertically/ [dostęp dnia 19 listopada 2022].

[4] Horizontal Vs. Vertical Scaling: How Do They Compare?, Czerw. 2021. https://www.cloudzero.com/blog/horizontal-vs-vertical-scaling [dostęp dnia 19 listopada 2022].

[5] 8 Ways to Effectively Reduce Server Response Time, Sty. 2022. https://datadome.co/learning-center/how-to-reduce-server-response-time/ [dostęp dnia 19 listopada 2022].

[6] Issues Related to Load Balancing in Distributed System, Czerw. 2022. https://www.geeksforgeeks.org/issues-related-to-load-balancing-in-distributed-system/ [dostęp dnia 20 listopada 2022].

[7] Apache JMeter - 23. Glossary. https://jmeter.apache.org/usermanual/glossary.html [dostęp dnia 20 stycznia 2023].

[8] Apache JMeter™. https://jmeter.apache.org [dostęp dnia 28 listopada 2022].

[9] A. Berezin. Algorithms and methods of load distribution on the server, Kwi. 2020. https://timeweb.com/ru/community/articles/algoritmy-i-metody-raspredeleniya-nagruzki-na-server [dostęp dnia 20 listopada 2022].

[10] diagrams.net. https://www.diagrams.net/about.html [Online; accessed 30-November-2020].

[11] Documentation. https://redis.io/documentation dostęp dnia 26 listopada 2022].

[12] Documentation. https://golang.org/doc/ [dostęp dnia 26 listopada 2022].

[13] Edytory dokumentów Google - Pomoc: odchylenie standardowe. https://support.google.com/docs/answer/3094064?hl=pl&ref_topic=3105600 [dostęp dnia 29 stycznia 2023].

[14] Edytory dokumentów Google - Pomoc: prawdopodobieństwo związane z rozkładem t. https://support.google.com/docs/answer/6055837?hl=pl [dostęp dnia 29 stycznia 2023].

[15] Edytory dokumentów Google - Pomoc: wariancja. https://support.google.com/docs/answer/3094054?hl=pl [dostęp dnia 29 stycznia 2023].

[16] Getting Started. https://code.visualstudio.com/docs [dostęp dnia 28 listopada 2022].

[17] Getting Started. https://code.visualstudio.com/docs [dostęp dnia 26 listopada 2022].

[18] Go by Example. https://gobyexample.com [dostęp dnia 26 listopada 2022].

[19] Grafana OSS. https://grafana.com/docs/grafana/latest/introduction/ [dostęp dnia 28 listopada 2022].

[20] Hypertext Transfer Protocol – HTTP/1.1. https://www.w3.org/Protocols/rfc2616/rfc2616.html [dostęp dnia 28 listopada 2022].

[21] Introduction. https://learning.postman.com/docs/getting-started/introduction/ [dostęp dnia 28 listopada 2022].

[22] Introduction to JSON Web Tokens. https://jwt.io/introduction/ [dostęp dnia 26 listopada 2022].

[23] Komunikator internetowy, Maj 2020. https://mfiles.pl/pl/index.php/Komunikator_internetowy [dostęp dnia 26 listopada 2022].

[24] Normality Calculator. https://www.gigacalculator.com/calculators/normality-test-calculator.php [dostęp dnia 29 stycznia 2023].

[25] Orientation and setup. https://docs.docker.com/get-started/ [dostęp dnia 26 listopada 2022].

[26] L. H. Pramono, R. Cokro. Round-robin algorithm in haproxy and nginx load balancing performance evaluation: a review. *2018 International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*, 2018. https://www.researchgate.net/publication/336561317_Round-robin_Algorithm_in_HAProxy_and_Nginx_Load_Balancing_Performance_Evaluation_a_Review.

[27] RESTful API Design: 13 Best Practices to Make Your Users Happy, Sier. 2018. https://florimond.dev/blog/articles/2018/08/restful-api-design-13-best-practices-to-make-your-users-happy/ [dostęp dnia 26 listopada 2022].

[28] Search for Go Packages. https://godoc.org [dostęp dnia 26 listopada 2022].

[29] Simplicity matters. Build once. https://www.influxdata.com/products/ [dostęp dnia 28 listopada 2022].

[30] Telegraf. https://www.influxdata.com/time-series-platform/telegraf/ [dostęp dnia 28 listopada 2022].

[31] The Go Project. https://golang.org/project/ [dostęp dnia 26 listopada 2022].

[32] Welcome to NGINX Wiki! https://www.nginx.com/resources/wiki/ [dostęp dnia 28 listopada 2022].

[33] What is a Container? https://www.docker.com/resources/what-container [dostęp dnia 26 listopada 2022].

[34] What is CRUD? https://www.codecademy.com/article/what-is-crud [dostęp dnia 29 stycznia 2023].

[35] What is REST. https://restfulapi.net [dostęp dnia 26 listopada 2022].