JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 31 (2024) 124–131
Received: 11 April 2024
Accepted: 2 June 2024

Investigating the impact of microservice-oriented platform configurations on application performance

Badanie wpływu konfiguracji platform do tworzenia mikrousług na wydajność aplikacji

Bartosz Biegajło[a,*], Dariusz Czerwiński[b]

[a] *Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland*

[b] *Department of Applied Computer Science, Lublin University of Technology, Nadbystrzycka 38, 20-618 Lublin, Poland*

Abstract

Effective management of containerized applications is crucial to ensuring their performance and reliability. The aim of this work was to indicate which configuration settings of the Kubernetes orchestrator have the greatest impact on microservice application performance under conditions of increased load. For each of the established configuration variants, the throughput and response time of the test application based on the microservices paradigm were measured. Research findings indicate that excessive horizontal scaling degrades application performance and that memory usage settings may play a greater role in optimizing system performance than CPU usage.

Keywords: Kubernetes; application performance; vertical scaling; horizontal scaling

Streszczenie

Efektywne zarządzanie skonteneryzowanymi aplikacjami jest kluczowe dla zapewnienia ich wydajności i niezawodności. Celem niniejszej pracy było wskazanie jakie ustawienia konfiguracyjne orkiestratora Kubernetes mają największy wpływ na wydajność aplikacji mikrousługowej w warunkach wzmożonego obciążenia. Dla każdego z ustalonych wariantów konfiguracji zmierzono przepustowość oraz czas odpowiedzi testowej aplikacji opartej o paradygmat mikrousług. Wyniki badań wskazują, że nadmierne skalowanie horyzontalne pogarsza wydajność aplikacji oraz że ustawienia zużycia pamięci operacyjnej mogą odgrywać większą rolę w optymalizacji wydajności systemu niż zużycie procesora.

Słowa kluczowe: Kubernetes; wydajność aplikacji; skalowanie wertykalne; skalowanie horyzontalne

*Corresponding author

Email address: **bartosz.biegajlo@pollub.edu.pl** (B. Biegajło)

1. Wstęp

W dobie cyfryzacji i rosnącej zależności od technologii, współczesna informatyka stoi przed wyzwaniem szybkiego rozwijania oprogramowania, zapewnienia jego wydajności, efektywnego wdrażania i zarządzania nim w złożonych środowiskach technologicznych. Optymalizacja procesów związanych z dostarczaniem wydajnych aplikacji to nie tylko kwestia techniczna, ale także biznesowa. Wysoka dostępność i niezawodność systemów są kluczowe dla utrzymania ciągłości działania usług oraz zapewnienia pozytywnego doświadczenia użytkowników.

W tradycyjnych środowiskach informatycznych aplikacje były często rozwijane zgodnie z architekturą monolityczną, gdzie cała funkcjonalność systemu była zintegrowana w jednym, niepodzielnym bloku oprogramowania. Jednym z głównych wyzwań związanych z rozwojem aplikacji w ten sposób jest jej rosnąca złożoność. Dodawanie nowych funkcjonalności lub modyfikowanie istniejących części systemu staje się coraz trudniejsze i czasochłonne [1]. Współczesne podejścia, takie jak konteneryzacja i architektura mikrousługowa, radykalnie zmieniają sposób tworzenia i dostarczania oprogramowania. Konteneryzacja, reprezentowana przez technologie takie jak Docker, umożliwia pakowanie aplikacji wraz z jej wszystkimi zależnościami i środowiskiem wykonawczym w lekkie, samowystarczalne jednostki (kontenery), co ułatwia ich wdrażanie na różne środowiska oraz skalowanie [2]. Architektura mikrousługowa natomiast polega na rozwoju systemu w formie stosunkowo małych, niezależnych usług, które komunikują się ze sobą za pośrednictwem „lekkich" mechanizmów, takich jak protokół HTTP i REST (ang. *Representational State Transfer*) [3]. W tym podejściu każda usługa jest odpowiedzialna za pojedynczą funkcjonalność lub obejmuję jedną, spójną domenę biznesową, co przekłada się na większą modularność i elastyczność systemu [4]. Umożliwia to niezależne wdrażanie i skalowanie poszczególnych mikrousług, szybsze wprowadzania zmian i aktualizacji, minimalizuje ryzyko wprowadzania błędów w innych częściach systemu oraz daje możliwość elastyczniejszego zarządzania zasobami [5].

Na tle wspomnianych podejść wyłaniają się istotne narzędzia jakimi są orkiestratory kontenerów. Pozwalają one na automatyzację wielu procesów związanych z tworzeniem i dostarczaniem do użytku skonteneryzowanych aplikacji. Kubernetes, jako jeden z najbardziej popularnych orkiestratorów, pozwala między innymi na zautomatyzowane wdrażanie i skalowanie wielu niezależnych usług, które razem

mogą tworzyć kompleksowe systemy [6]. Aby w pełni wykorzystać możliwości, jakie dają nam narzędzia do orkiestracji kontenerów, niezbędna jest analiza i zrozumienie dostępnych opcji ich konfiguracji oraz zbadanie jej wpływu na wydajność wdrażanej aplikacji.

Bieżąca praca skupia się na tych kwestiach poprzez eksperymentalne porównywanie wpływu konfiguracji orkiestratora Kubernetes na kluczowe wskaźniki wydajności aplikacji, która została przygotowana w sposób umożliwiający wdrażanie jej w postaci wielu skonteneryzowanych, komunikujących się ze sobą mikrousług. Ponadto, zadaniem niniejszej pracy jest sprawdzenie, jakie ustawienia konfiguracyjne mają największy wpływ na parametry wydajności systemu.

2. Przegląd literatury

W epoce przetwarzania w chmurze obliczeniowej (ang. *cloud computing*), konteneryzacja i mikrousługi stały się elementami powszechnie wykorzystywanymi do wytwarzania i dostarczania oprogramowania. Chmura obliczeniowa opiera się na technologiach wirtualizacji, które umożliwiają użytkownikom „wypożyczanie" wirtualnych zasobów od dostawców usług chmurowych [7]. W klasycznym podejściu, w różnych modelach udostępniania usług chmurowych używana jest tzw. pełna wirtualizacja w postaci wirtualnych maszyn zarządzanych przez hipernadzorce (ang. *hypervisor*). Alternatywą dla klasycznej wirtualizacji jest konteneryzacja. Kontenery zużywają mniej zasobów niż maszyny wirtualne, ponieważ uruchomione na jednej maszynie dzielą ten sam system operacyjny. Dzięki temu zajmują mniej miejsca w pamięci i efektywniej wykorzystują procesor oraz inne zasoby. Ponadto są one dobrze odizolowane od siebie i reszty środowiska, a także mogą być szybciej uruchamiane i zatrzymywane [8].

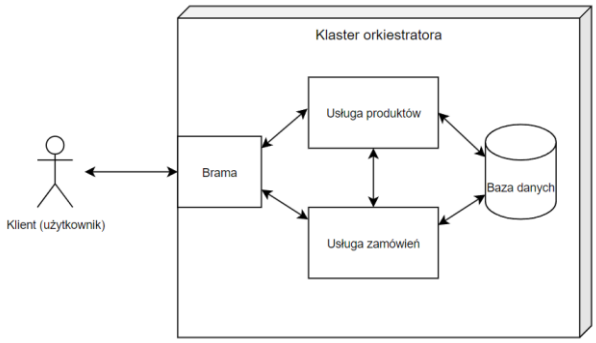
Jedną z kluczowych zalet konteneryzacji jest przenośność, czyli możliwość tworzenia jednostek gotowych do uruchomienia w różnych środowiskach. Technologie takie jak Docker pozwalają na tworzenie pliku zwanego obrazem kontenera, który składa się z „warstw" danych dotyczących spakowanej w nim aplikacji wraz z zależnościami. Plik ten może być następnie użyty do uruchomienia kontenera [9]. Obraz niesie ze sobą dane aplikacji oraz wszystkich elementów zewnętrznych, które są niezbędne do jej prawidłowego działania, takich jak biblioteki i inne pakiety oprogramowania. Dzięki temu możliwe jest łatwe wdrożenie takiej aplikacji w formie kontenera na różnorodnych środowiskach wspierających wirtualizacje na poziomie systemu operacyjnego [10]. Wdrażanie aplikacji w formie wyizolowanych kontenerów dobrze wpasowuje się w założenia mikrousługowej architektury oprogramowania. Konteneryzacja upraszcza procesy niezależnego rozwijania, testowania, wdrażania i skalowania każdej usługi składowej systemu. Dodatkowo, powszechnie wykorzystywane są możliwości integracji narzędzi konteneryzacyjnych z narzędziami CICD (ang. *Continuous Integration, Continuous Delivery*), aby zautomatyzować wspomniane procesy [11]. W odniesieniu do sposobu wdrażania usług, badania wskazują na to, że optymalnym jest pakowanie jednej usługi w jednym kontenerze. Pomiary dokonane w jednej z chmur publicznych wykazały, że przy wdrażaniu usług w ten sposób możliwa jest znacząca redukcja zużycia zasobów sieciowych [12]. Wytwarzanie i działanie systemów składających się z wielu oddzielnych usług jest złożone, co stanowi wyzwanie wobec ich projektowania, monitorowania i testowania, ponieważ nie ma do tego standardowych, dedykowanych rozwiązań [13]. Zauważono, że podczas monitorowania takich systemów przeważnie znaczenie ma zużycie zasobów oraz zrównoważenie obciążenia. Orkiestracja kontenerów staje się kluczowym procesem wspomagającym specjalistów we wspomnianych kwestiach. Polega ona na automatyzacji wielu działań takich jak przygotowanie do uruchomienia (ang. *provisioning*), wdrażanie, skalowanie, tworzenie sieci, udostępnianie i zarządzanie cyklem życia kontenerów [14]. Aktualnie najpopularniejszą platformą do orkiestracji kontenerów jest Kubernetes. Konfiguracja orkiestracji kontenerów przy użyciu tego narzędzia jest możliwa w sposób deklaratywny. Na podstawie pliku konfiguracyjnego orkiestrator dąży do osiągnięcia zadeklarowanego stanu, wykonując wdrożenie i automatycznie zarządzając różnymi ustawieniami.

Możliwości związane z zarządzaniem systemu mikrousługowego są szeroko poruszane jako temat różnych badań naukowych. Zauważono, że skalowalność jako główna zaleta architektury mikrousługowej, stanowi jednocześnie powszechne wyzwanie, z którym trzeba mierzyć się przy wytwarzaniu tego typu oprogramowania [15]. Istnieją strategie skalowania aplikacji, takie jak skalowanie horyzontalne oraz wertykalne. W odniesieniu do skonteneryzowanych usług skalowanie horyzontalne polega na zwiększaniu liczby ich instancji (replikacja). Skalowaniem wertykalnym nazwiemy przydzielenie konkretnym maszynom lub instancjom usług większej ilości mocy obliczeniowej lub pamięci operacyjnej [16]. Okazuje się, że w różnych okolicznościach, te same strategie skalowania dają różne rezultaty. W jednym z przeprowadzonych eksperymentów wykazano, że w pewnych warunkach dla automatycznie skalowalnych środowisk chmurowych, zwiększanie mocy obliczeniowej nie było efektywną techniką podnoszenia wydajności systemu mikrousługowego [17]. Innym razem, w przypadku środowiska chmurowego Azure, skalowanie wertykalne jawiło się jako bardziej efektywne niż skalowanie horyzontalne [18]. Ponadto badacze wykazali, że skalowanie aplikacji ponad pewną określoną liczbę instancji pogarsza wydajność aplikacji.

3. Aplikacja testowa

W ramach niniejszej pracy została przygotowana mikrousługowa aplikacja testowa, której założeniem było przybliżone odwzorowanie funkcjonowania systemu handlu elektronicznego (ang. *e-commerce*). Aplikacja składa się głównie z trzech rodzajów usług nadających się do wdrożenia jako oddzielne kontenery.

Pierwszy rodzaj usługi funkcjonuje w poddomenie produktów. Drugi rodzaj usługi obsługuje żądania i wykonuje operacje związane z przetwarzaniem zamówień na konkretne produkty (poddomena zamówień). Trzecim rodzajem usługi jest baza danych. Jest ona współdzielona przez dwa wcześniej opisane rodzaje usług. Użytkownik, który chce skorzystać z systemu, może w bezpośredni sposób komunikować się jedynie z bramą (ang. *gateway*), czyli punktem wejściowym systemu, który przekazuje żądania do odpowiedniej usługi. Rysunek 1 obrazuje ogólną architekturę aplikacji i przepływ komunikacji pomiędzy jej składowymi komponentami zgodnie z przytoczonym opisem.



Rysunek 1: Ogólna architektura aplikacji testowej.

Aplikacja została przygotowana w taki sposób, aby możliwa była jej orkiestracja. Jako narzędzie do orkiestracji wybrano Minikube w wersji 1.32.0, czyli odmianę Kubernetesa pozwalającą na szybkie uruchomienie klastra Kubernetes na lokalnej maszynie [19]. Do implementacji składowych usług systemu funkcjonujących w domenie produktów i zamówień wykorzystano szkielet programistyczny Spring Boot w wersji 3.2.1 oparty na języku Java (wersja 17). W przypadku implementacji usługi bazodanowej użyto systemu PostgreSQL w wersji 16.2. Usługa z bazą danych jako jedyna nie podlegała procesowi skalowania. Wynika to z istotnych ograniczeń związanych z powielaniem tego typu kontenerów. W środowiskach z wieloma kopiami baz danych, każda z nich musi być aktualizowana równocześnie, aby zapewnić spójność danych pomiędzy replikami. Wymaga to implementacji złożonych mechanizmów synchronizacji i replikacji danych [20].

4. Stanowisko badawcze

Badania przeprowadzono przy wykorzystaniu komputera stacjonarnego posiadającego procesor AMD Ryzen 5 3600 o taktowaniu 3600 MHz, pamięć RAM o pojemności 32 GB, system Microsoft Windows w wersji 10 Home oraz kartę graficzną Nvidia GeForce GTX 1050 Ti. Wykorzystana implementacja klastra Kubernetes bazowała na środowisku wykonawczym containerd. Realizacja komunikacji została skonfigurowana poprzez domyślny most Docker (Docker w wersji 24.0.7) w trybie L2. W roli bramy wykorzystano kontroler Ingress NGINX w wersji 1.9.4.

5. Metoda badań

Do przeprowadzenia eksperymentów wydajnościowych, dla każdego scenariusza badawczego przewidziana została ogólna procedura składająca się z następujących kroków:

1. Zdefiniowanie wejściowej konfiguracji wdrożeniowej.
2. Wdrożenie usług w określonej konfiguracji orkiestratora.
3. Symulacja obciążenia z równoczesnym wykonywaniem i zapisywaniem pomiarów.
4. Analiza wyników pomiarów.

Głównym wskaźnikiem wydajności mierzonym podczas symulacji obciążenia była osiągana przez aplikacje przepustowość, rozumiana jako liczba w pełni obsłużonych żądań użytkowników w czasie jednej sekundy. Dodatkowo wzięty pod uwagę został czas odpowiedzi, czyli średni czas, w którym użytkownik dostaje odpowiedź na żądanie, mierzony od momentu wykonania zapytania do momentu uzyskania odpowiedzi. Do przeprowadzania samej symulacji obciążenia wykorzystane zostało otwarto-źródłowe narzędzie k6 oparte na języku Go [21]. W tym przypadku generowanie obciążenia polega na tworzeniu wielu działających równolegle „wirtualnych użytkowników” (ang. *Virtual Users*). Każdy z nich wysyła zapytania do systemu będącego przedmiotem eksperymentu. Podczas wykonywania zapytań przez wirtualnych użytkowników, narzędzie testowe na bieżąco dokonuje pomiarów. W uruchomionym systemie, poprzez konfiguracje, instancjom usług były przydzielane konkretne ilości zasobów. Przydzielanymi zasobami były pamięć RAM liczona w MiB oraz moc obliczeniowa mierzona w CPU, gdzie 1 CPU jest równowartością mocy obliczeniowej jednego rdzenia procesora maszyny, na której przeprowadzano eksperymenty. Pod z bazą danych posiadał stały limit zasobów w postaci 1 CPU oraz 4096 MiB pamięci operacyjnej.

6. Opis eksperymentów

Na samym początku zostały wykonane podstawowe testy wydajności, aby zrozumieć, w jaki sposób pojedyncze repliki usług radzą sobie z obciążeniem. W tym celu, w sposób iteracyjny, dopasowywano warunki obciążenia dla systemu uruchomionego w konfiguracji, która dopuszcza maksymalnie jedną instancje usługi każdego typu. Określano bazowe obciążenie (maksymalną liczbę wirtualnych użytkowników), dla którego zmieniano ilość zasobów przydzielonych dla każdej instancji. Dla konkretnych ilości przydzielonych zasobów zostały wykonane symulacje, na bazie których wyznaczano przepustowość i czasy odpowiedzi. Następnie warunki obciążenia ulegały zmianie, a procedura była wykonywana od początku.

Kolejny scenariusz eksperymentów obejmował testy obciążeniowe aplikacji przy manualnym zwiększaniu docelowej liczby replik usług (skalowaniu horyzontalnym). Dla różnych warunków obciążenia wykonywano serie symulacji. Po każdej symulacji z serii

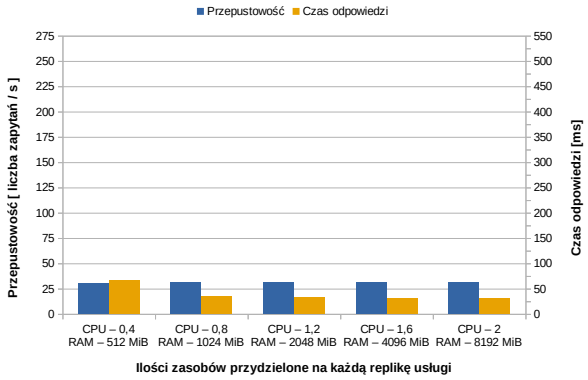
zwiększano liczbę instancji usług. Na jedną replikę usługi zawsze przypadała stała ilość zasobów. Przy skalowaniu nie powielano kontenerów z bazą danych. Następnie przeprowadzono eksperymenty stosując automatyczne skalowanie horyzontalne. Zdefiniowano bazową konfigurację mechanizmu HPA (ang. Horizontal Pod Autoscaler) i uruchomiono symulację zmiennego obciążenia. W kolejnych krokach zmieniane były parametry HPA, takie jak średnie docelowe wykorzystanie mocy obliczeniowej dostępnej dla pojedynczej repliki. Po zmianie parametrów konfiguracyjnych symulacja była ponawiana.

7. Wyniki

Niniejszy rozdział prezentuje efekty przeprowadzonych eksperymentów oraz ich analizę w kontekście założonych celów pracy.

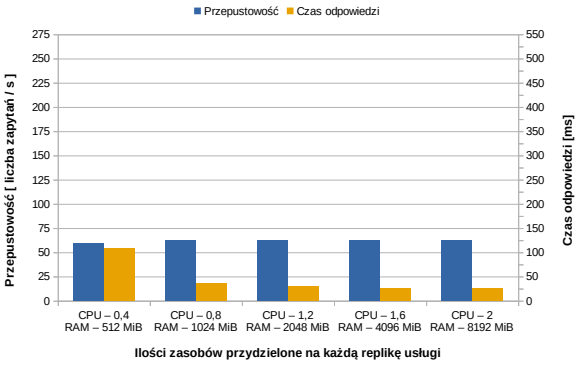
7.1. Wstępne testy obciążeniowe

Podstawowe badania wydajnościowe wykonano pod obciążeniem aplikacji w postaci maksymalnie 50, 100, 200, 400 oraz 800 wirtualnych użytkowników (VU). Rysunki 2 – 6 przedstawiają wyniki tych testów. System uruchamiano przy maksymalnie jednej replice usługi każdego typu. Dla każdego wariantu obciążenia zmieniano ilość zasobów przydzielanych na każdą instancje usługi.



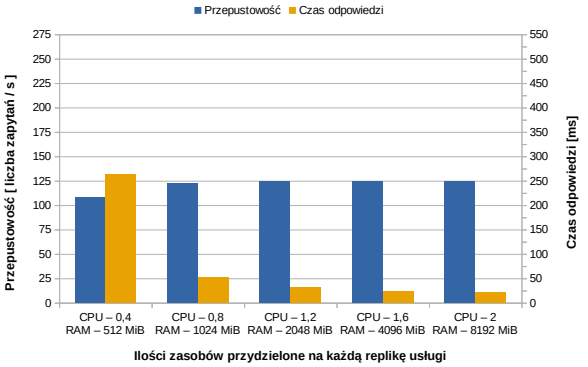
Rysunek 2: Przepustowość i czas odpowiedzi aplikacji przy obciążeniu maksymalnym 50 VU.

Jak widać na Rysunku 2 badany system bez problemu obsługuje ruch w postaci maksymalnie 50 wirtualnych użytkowników. W każdym z wariantów konfiguracyjnych uśredniona przepustowość mieści się w okolicach 30 zapytań na sekundę. Średni czas odpowiedzi również jest stabilny (około 32 ms), jednak z wartością zauważalnie większą dla pierwszego wariantu (66 ms). Rysunek 3 bardziej dosadnie obrazuje tę różnicę. Nastąpił dwukrotny wzrost przepustowości, co oznacza, że aplikacja adekwatnie reaguje na również dwukrotny wzrost obciążenia.



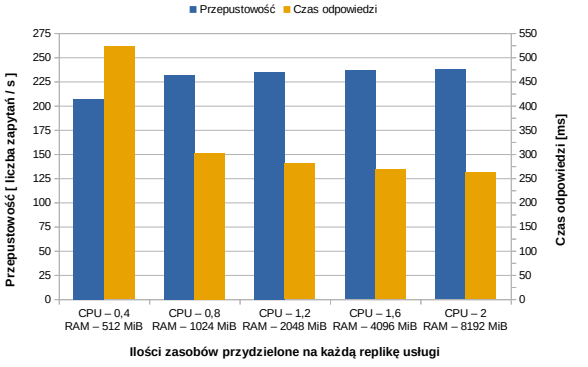
Rysunek 3: Przepustowość i czas odpowiedzi aplikacji przy obciążeniu maksymalnym 100 VU.

Na Rysunku 4 widzimy jeszcze większą przepaść między pierwszą, a pozostałymi opcjami konfiguracyjnymi. Zaczyna być również zauważalna różnica w przepustowości.



Rysunek 4: Przepustowość i czas odpowiedzi aplikacji przy obciążeniu maksymalnym 200 VU.

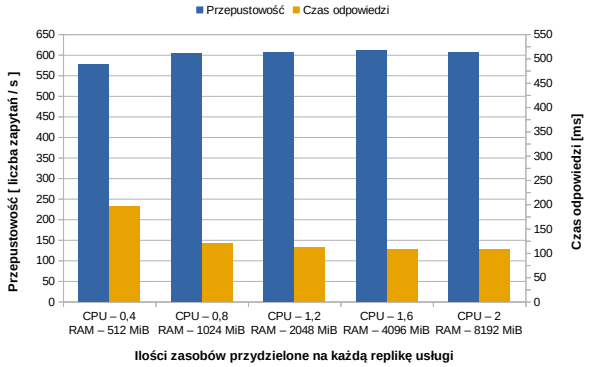
W przypadku obciążenia maksymalnie 400 VU (Rysunek 5) utrzymuje się charakter wspomnianych różnic. Można zauważyć, że czas odpowiedzi wyraźnie zwiększył się dla wszystkich opcji. Dodatkowo 34% wszystkich wysłanych zapytań nie powiodło się.



Rysunek 5: Przepustowość i czas odpowiedzi aplikacji przy obciążeniu maksymalnym 400 VU.

Przy obciążeniu w postaci 800 wirtualnych użytkowników, odsetek zapytań wzrósł do 74%, natomiast poprawiły się inne parametry, co pokazane jest na Rysunku 6. Przepustowość znów wzrosła, ale czas odpowiedzi znacząco zmalał dla każdej opcji konfiguracyjnej. Jest to prawdopodobnie spowodowane faktem, że w dużej części przypadków system zwracał

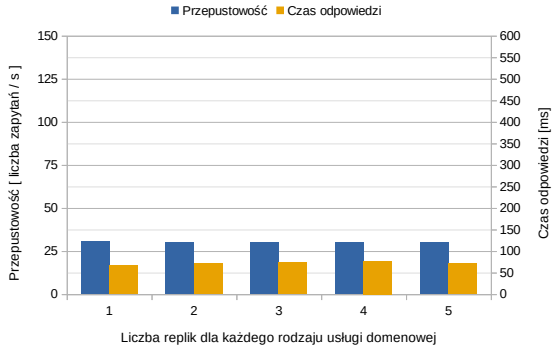
odpowiedź informującą użytkownika o braku możliwości przetworzenia jego żądania.



Rysunek 6: Przepustowość i czas odpowiedzi aplikacji przy obciążeniu maksymalnym 800 VU.

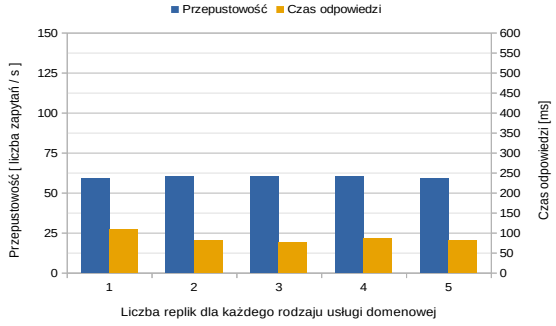
7.2. Manualne skalowanie horyzontalne

Po wstępnym rozpoznaniu możliwości badanego systemu zbadano zachowanie aplikacji w sytuacji manualnego skalowania horyzontalnego. Symulacje z założonym obciążeniem wykonywano dla każdej, z góry ustalonej liczby replik usług. Dla najniższego obciążenia (50 VU) zwiększanie liczby uruchamianych replik usług nie powodowało znaczących rezultatów, co obrazuje Rysunek 7.



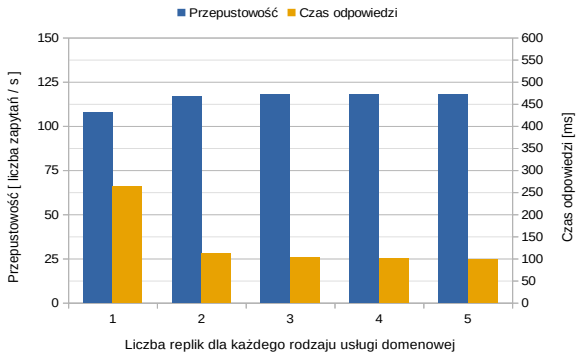
Rysunek 7: Przepustowość i czas odpowiedzi aplikacji przy powielaniu instancji usług dla obciążenia 50 VU.

Przy obciążeniu 100 VU wystąpił naturalny wzrost przepustowości w związku z większą liczbą odbieranych i obsługiwanych przez aplikacje żądań (Rysunek 8).



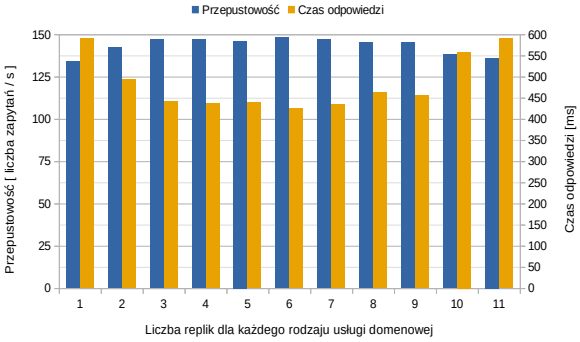
Rysunek 8: Przepustowość i czas odpowiedzi aplikacji przy powielaniu instancji usług dla obciążenia 100 VU.

Rysunek 9 pokazuje wyniki kolejnej serii symulacji przy dwukrotnie większym obciążeniu.



Rysunek 9: Przepustowość i czas odpowiedzi aplikacji przy powielaniu instancji usług dla obciążenia 200 VU.

Dla ostatniego przypadku, tj. obciążenia w postaci 300 VU, zdecydowano się na wykonanie większej liczby symulacji. Maksymalna liczba replik dla każdego typu usług domenowych wyniosła 11. Jak widać na Rysunku 10, powtarza się trend poprawy wydajności w pierwszych etapach skalowania. Dla pojedynczych replik czas odpowiedzi wynosi około 590 ms, a przepustowość – ok. 135 żądań na sekundę. Dla sześciu replik czas odpowiedzi jest już równy około 420 ms a przepustowość wynosi niemal 150 zapytań na sekundę. Dalsze zwiększanie liczby replik powoduje stopniowe pogorszenie wydajności aplikacji. Dla 11 instancji wartości parametrów wydajnościowych są zbliżone do wartości początkowych.



Rysunek 10: Przepustowość i czas odpowiedzi aplikacji przy powielaniu instancji usług dla obciążenia 300 VU.

Na podstawie przytoczonych wyników można zauważyć, że do pewnego momentu, im wyższe obciążenie aplikacji, tym większy zysk wydajnościowy spowodowany skalowaniem horyzontalnym. Widoczny jest również punkt, w którym po przekroczeniu optymalnej liczby replik aplikacja wolniej obsługuje nadchodzący ruch.

7.3. Automatyczne skalowanie horyzontalne

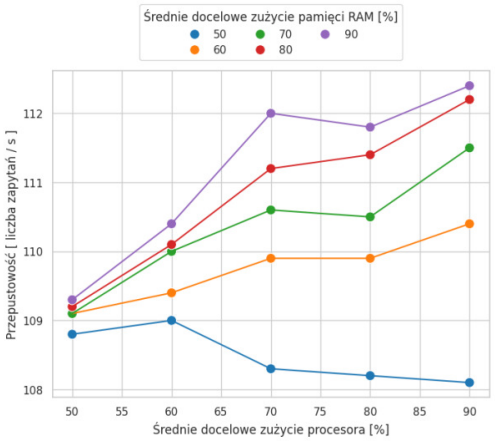
Zgodnie z założeniami przystąpiono do eksperymentów dotyczących wydajności aplikacji w warunkach automatycznego skalowania horyzontalnego. Przy użyciu ustawień HPA zadeklarowano minimalną i maksymalną dozwoloną liczbę replik usług każdego typu, wynoszącą odpowiednio 1 i 15. Dla każdej instancji zdefiniowano również limity dotyczące maksymalnego zużycia zasobów w postaci 0,4 CPU oraz 512 MiB

pamięci RAM. Następnie wykonano serię testów obciążeniowych, za każdym razem zmieniając docelowe średnie zużycie procesora lub pamięci RAM. Zbiorcze zestawienie wyników tych testów przedstawia tabela 1. Na podstawie ustawień średniego zużycia zasobów algorytm wykorzystany przez HPA dopasowywał liczbę uruchomionych podów. Przykładowo, parametr średniego docelowego wykorzystania pamięci RAM ustawiony na 50% oznacza, że wszystkie pody, których dotyczy ustawienie, powinny zużywać średnio 50% maksymalnie dozwolonej ilości pamięci RAM. Badane parametry posiadały takie same wartości dla obu typów usług domenowych.

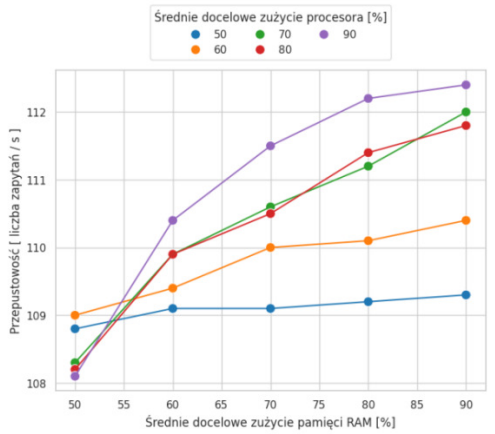
Tabela 1: Przepustowość i czas odpowiedzi przy zmianie docelowego zużycia zasobów

Średnie docelowe zużycie procesora [%]	Średnie docelowe zużycie pamięci RAM [%]	Czas odpowiedzi [ms]	Przepustowość [liczba zapytań / s]
50	50	212	108,8
50	60	210	109,1
50	70	209	109,1
50	80	207	109,2
50	90	205	109,3
60	50	213	109,0
60	60	203	109,4
60	70	195	110,0
60	80	191	110,1
60	90	185	110,4
70	50	213	108,3
70	60	194	109,9
70	70	170	110,6
70	80	167	111,2
70	90	165	112,0
80	50	215	108,2
80	60	190	109,9
80	70	179	110,5
80	80	164	111,4
80	90	159	111,8
90	50	219	108,1
90	60	183	110,4
90	70	165	111,5
90	80	149	112,2
90	90	146	112,4

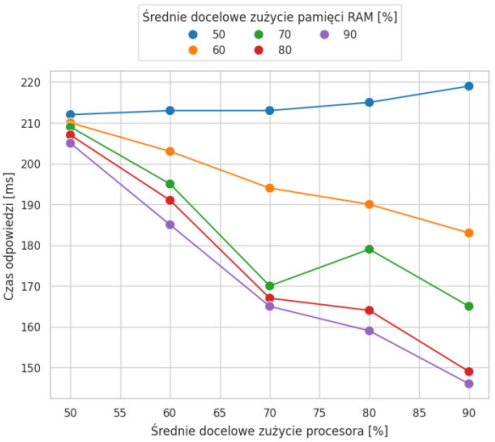
Wizualizacja przytoczonych wyników (Rysunki 11 – 14) pozwala lepiej zrozumieć wpływ obu ustawień na wydajność aplikacji. Jak widać na Rysunkach 11 i 12, zwiększanie docelowego zużycia zasobów powoduje większą przepustowość systemu. Szczególnie interesujące są przypadki skrajne. Gdy średnie docelowe zużycie pamięci RAM jest najniższe (50%), zwiększanie ustawienia zużycia procesora nie przynosi żadnych korzyści, a wręcz powoduje niższą przepustowość. Z kolei w sytuacji, gdy średnie docelowe zużycie procesora jest najwyższe, wraz ze zwiększaniem ustawienia zużycia pamięci przyrosty przepustowości zdają się coraz bardziej maleć.



Rysunek 11: Przepustowość a średnie docelowe zużycie procesora przy różnych poziomach średniego docelowego zużycia pamięci.



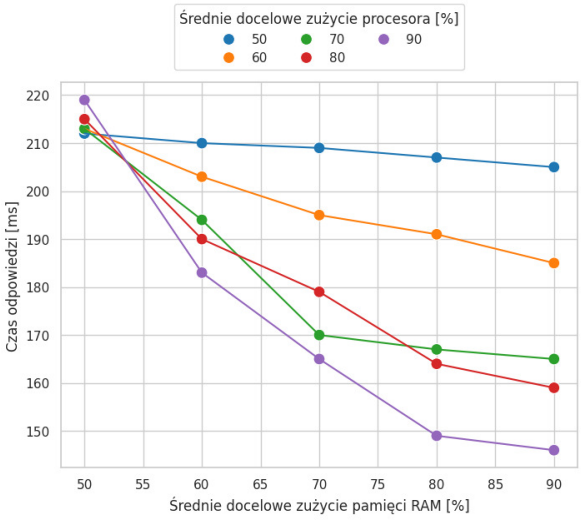
Rysunek 12: Przepustowość a średnie docelowe zużycie pamięci przy różnych poziomach średniego docelowego zużycia procesora.



Rysunek 13: Czas odpowiedzi a średnie docelowe zużycie procesora przy różnych poziomach średniego docelowego zużycia pamięci.

Badając drugi parametr wydajności, jakim jest czas odpowiedzi można dostrzec podobne zależności. Zwiększanie docelowych poziomów zużycia zasobów generalnie powoduje mniejszy czas odpowiedzi aplikacji. Dla minimalnego docelowego zużycia pamięci RAM zwiększanie zużycia procesora wydaje się być antyproduktywne. Gdy średnie docelowe zużycie

procesora wynosi 90% a zwiększane jest ustawienie zużycia pamięci RAM, czas odpowiedzi maleje w coraz mniejszym stopniu.



Rysunek 14: Czas odpowiedzi a średnie docelowe zużycie pamięci przy różnych poziomach średniego docelowego zużycia procesora.

Zgodnie z wynikami, podnoszenie zarówno docelowego średniego zużycia procesora jak i pamięci RAM, związane jest ze zmniejszaniem się czasu odpowiedzi oraz zwiększaniem przepustowości. Zaobserwowano jednak, że badane ustawienia mają różny wpływ na wymienione parametry wydajności. Aby dokładniej określić te zależności obliczono współczynniki korelacji Pearsona przedstawione w Tabeli 2.

Tabela 2: Współczynniki korelacji pomiędzy ustawieniami konfiguracyjnymi a parametrami wydajności

Badana zależność	Współczynnik korelacji
Przepustowość a docelowe średnie zużycie procesora	0,48
Przepustowość a docelowe średnie zużycie pamięci RAM	0,73
Czas odpowiedzi a docelowe średnie zużycie procesora	-0,57
Czas odpowiedzi a docelowe średnie zużycie pamięci RAM	-0,68

Z obliczeń wynika, że korelacja pomiędzy średnim docelowym zużyciem pamięci a parametrami wydajności jest silniejsza niż zależność między tymi parametrami a średnim docelowym zużyciem procesora. Oznacza to, że przy ustalaniu pożądanej konfiguracji systemu, większe znaczenie może mieć optymalizacja zużycia pamięci operacyjnej.

8. Wnioski

Przeprowadzone eksperymenty pozwoliły na zgłębienie relacji między konfiguracją systemu a jego wydajnością. Możliwe było także dokonanie dodatkowych obserwacji na temat zachowania aplikacji mikrousługowej pod obciążeniem. Z wykonanych badań wynika, że skalowanie wertykalne generalnie podnosi wydajność systemu. Niestety, przy zastosowaniu tego podejścia musimy liczyć się z ograniczeniami sprzętowymi i fundamentalnymi barierami technologicznymi. W trakcie eksperymentów okazało się też, że niski czas odpowiedzi nie zawsze wiąże się z wysoką wydajnością systemu. Może to być efektem na przykład odbijania żądań przez system z powodu przeciążeń. System reaguje wtedy szybciej, dając użytkownikowi informacje zwrotną, ale tylko dlatego, że nie jest w stanie prawidłowo przetworzyć kolejnego żądania.

Na bazie doświadczeń związanych z manualnym skalowaniem horyzontalnym zauważono, że przy niskim obciążeniu aplikacji, skalowanie horyzontalne nie przyniosło zauważalnych korzyści. Dopiero przy bardziej wymagającym obłożeniu żądaniami widoczna była poprawa wydajności. Taka obserwacja sugeruje, że przy niewielkim ruchu sieciowym i niskiej liczbie żądań, pojedyncza instancja usługi jest w stanie sprostać obsłudze zapytań bez wyraźnych opóźnień, a wprowadzanie dodatkowych replik nie generuje dodatkowej wartości. W okolicznościach większego ruchu zaczynały pojawiać się opóźnienia w reakcji pojedynczych replik. Prawdopodobnie wtedy, zdolność systemu do równoległego przetwarzania żądań poprzez dystrybucję ruchu między większą liczbą instancji pozwalała na efektywniejsze wykorzystanie zasobów. Potwierdzenie znalazł również fakt, że zbyt intensywne skalowanie horyzontalne może prowadzić do pogorszenia wydajności aplikacji.

Wyniki eksperymentów dotyczących skalowania horyzontalnego sugerują, że ustawienie konfiguracyjne, jakim jest docelowe średnie zużycie pamięci RAM miało większy wpływ na wydajność aplikacji niż docelowe średnie zużycie procesora. Przy podwyższaniu docelowego zużycia pamięci operacyjnej, na ogół aplikacja osiągała większe przepustowości i krótsze czasy odpowiedzi. Wynika stąd, że przy optymalizacji wydajności systemu, skupienie się na tym ustawieniu może przynieść więcej korzyści. Dalsze badania obejmujące dokładniejsze monitorowanie zużycia zasobów mogłyby pomóc w określeniu „wąskich gardeł" aplikacji, czyli specyficznych obszarów, które stają się przyczyną opóźnień lub ograniczeń przepustowości. Identyfikacja i uwzględnienie tych obszarów podniosłoby obiektywność przy ustalaniu warunków wyjściowych do testów obciążeniowych. Potrzebna jest również analiza większego zestawu ustawień konfiguracyjnych, na przykład różnych sposobów równoważenia obciążenia.

Precyzyjne dostosowanie ustawień orkiestratora kontenerów do specyficznych wymagań aplikacji może prowadzić do znaczących korzyści, zarówno pod względem wydajnościowym, jak i ekonomicznym. W kontekście dynamicznie rozwijających się technologii oraz stale zmieniających się wymagań biznesowych, ciągłe badanie funkcjonowania aplikacji i adaptacja ustawień konfiguracyjnych to czynności niezbędne dla zapewnienia optymalnej wydajności, skalowalności i

niezawodności nowoczesnych systemów mikrousługowych.

Literatura

[1] Z. Mushtaq, N. Saher, F. Shazad, S. Iqbal, A. Qasim, A Review on Transformation of Monolithic Applications towards Microservices Environment, International Journal of Innovations in Science & Technology 4 (2022) 1–18, https://doi.org/10.33411/ijist/2022040101.

[2] Y. Zhang, B. Vasilescu, H. Wang, V. Filkov, One size does not fit all: an empirical study of containerized continuous deployment workflows, In 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) (2018) 295–306, https://doi.org/10.1145/3236024.3236033.

[3] M. Fowler, J. Lewis, Microservices a definition of this new architectural term, https://martinfowler.com/articles/microservices.html, [17.01.2024].

[4] X. Larrucea, I. Santamaria, R. Colomo-Palacios, C. Ebert, Microservices, IEEE Software 35 (2018) 96–100, http://doi.org/10.1109/MS.2018.2141030.

[5] I. K. Aksakalli, T. Celik, A. B. Can, B. Tekinerdogan, Systematic Approach for Generation of Feasible Deployment Alternatives for Microservices, IEEE Access 9 (2021) 29505–29529, https://doi.org/10.1109/ACCESS.2021.3057582.

[6] Dokumentacja orkiestratora Kubernetes, https://kubernetes.io/docs/home/, [18.01.2024].

[7] PaaS vs. IaaS vs. SaaS vs. CaaS: How are they different? https://cloud.google.com/learn/paas-vs-iaas-vs-saas, [20.01.2024].

[8] PJ. Maenhaut, B. Volckaert, V. Ongenae, F. De Turck, Resource Management in a Containerized Cloud: Status and Challenges, Journal of Network and Systems Management 28 (2019) 197–246, https://doi.org/10.1007/s10922-019-09504-0.

[9] Y. Zhang, G. Yin, T. Wang, Y. Yu, H. Wang, An Insight Into the Impact of Dockerfile Evolutionary Trajectories on Quality and Latency, In 42nd IEEE Annual Computer Software and Applications Conference (COMPSAC) (2018) 138–143, http://doi.org/10.1109/COMPSAC.2018.00026.

[10] D. Boxley, Containers Vs. Virtual Machines: Why the Paradigm Shift Matters, https://cloudnativenow.com/topics/cloudnativedevelopment/containers-vs-virtual-machines-why-the-paradigm-shift-matters/, [30.01.2024].

[11] S. P. Sinde, B. Thakkalapally, M. Ramidi, S. Veeramalla, Continuous Integration and Deployment Automation in AWS Cloud Infrastructure, International Journal for Research in Applied Science and Engineering Technology 10 (2022) 1305–1309, https://doi.org/10.22214/ijraset.2022.44106.

[12] F. H. L. Buzato, A. Goldman, D. Batista, Efficient Resources Utilization by Different Microservices Deployment Models, In 17th IEEE International Symposium on Network Computing and Applications (NCA) (2018) 1–4, https://doi.org/10.1109/NCA.2018.8548346.

[13] M. Waseem, P. Liang, M. Shahin, A. Di Salle, G. Márquez, Design, Monitoring, and testing of microservices systems: The practitioners' perspective, Journal of Systems and Software 182 (2021) 111061–111105, https://doi.org/10.1016/j.jss.2021.111061.

[14] What is container orchestration, https://www.ibm.com/topics/container-orchestration, [28.01.2024].

[15] S. Li, H. Zhang, Z. Jia, C. Zhong, C. Zhang, Z. Shan, J. Shen, M. A. Babar, Understanding and addressing quality attributes of microservices architecture: A Systematic literature review, Information and Software Technology 131 (2021) 106449–106472, https://doi.org/10.1016/j.infsof.2020.106449.

[16] Horizontal vs Vertical scaling: An in-depth Guide, https://www.nops.io/blog/horizontal-vs-vertical-scaling/, [29.01.2024].

[17] A. Avritzer, V. Ferme, A. Janes, B. Russo, A. Hoorn, H. van Schulz, D. Menasché, V. Rufino, Scalability Assessment of Microservice Architecture Deployment Configurations: A Domain-based Approach Leveraging Operational Profiles and Load Tests, Journal of Systems and Software 165 (2020) 110564–110580, https://doi.org/10.1016/j.jss.2020.110564.

[18] G. Blinowski, A. Ojdowska, A. Przybyłek, Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation, IEEE Access 10 (2022) 20357–20374, https://doi.org/10.1109/ACCESS.2022.3152803.

[19] Dokumentacja Minikube, https://minikube.sigs.k8s.io/docs/, [19.02.2024].

[20] J. Gray, R. Helland, R. O'Neil, D. Shasha, The dangers of replication and a solution, ACM SIGMOD 25 (1996) 173–182, https://doi.org/10.1145/235968.233330.

[21] Dokukmentacja k6, https://grafana.com/docs/k6/latest/, [09.03.2024].