JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 29 (2023) 383–390
Received: 3 July 2023
Accepted: 17 September 2023

Comparison of application container orchestration platforms

Porównanie platform do orkiestracji kontenerów aplikacji

Adam Pankowski*, Paweł Powroźnik

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract

This article presents a comparative analysis of three well-known container orchestration platforms: Docker Swarm, Kubernetes and Apache Mesos, focusing on the deployment of a test application and measuring parameters such as deployment time, CPU, memory and disk utilization, application response time and the time to restore a replica of the application using an auto-recovery mechanism. The aim of the research is to verify the performance and efficiency of the analyzed platforms, facilitating informed decisions while choosing an orchestrator for containerized applications. Two research hypotheses have been stated. The first one assumes that the time required to launch an application using the Docker Swarm tool is the shortest among the analyzed platforms. The second hypothesis is that Kubernetes provides the most efficient results in terms of load scheduling and application scaling. The analysis performed on the Jenkins application showed the superiority of the Docker Swarm platform over the other studied tools in terms of performance.

Keywords: contenerization; Docker Swarm; Kubernetes; Apache Mesos

Streszczenie

Niniejszy artykuł przedstawia analizę porównawczą trzech znanych platform orkiestracji kontenerów: Docker Swarm, Kubernetes i Apache Mesos, koncentrując się na wdrażaniu aplikacji testowej oraz pomiaru takich parametrów jak: czas wdrożenia, obciążenie procesora, wykorzystanie pamięci i dysku, czas odpowiedzi aplikacji oraz czas przywrócenia repliki aplikacji przy użyciu mechanizmu autoregeneracji. Celem badań jest weryfikacja wydajności i efektywności analizowanych platform, ułatwiając podjęcie świadomych decyzji przy wyborze orkiestratora dla skonteneryzowanych aplikacji. Zostały postawione dwie hipotezy badawcze. Pierwsza z nich zakłada, że czas potrzebny na uruchomienie aplikacji przy użyciu narzędzia Docker Swarm jest najkrótszy spośród analizowanych platform. Druga hipoteza zakłada, że Kubernetes zapewnia najwydajniejsze wyniki pod względem planowania obciążenia i skalowania aplikacji. Analiza przeprowadzona na podstawie wykonanych badań na obrazie aplikacji Jenkins wykazała przewagę platformy Docker Swarm nad pozostałymi badanymi narzędziami pod względem wydajnościowym.

Słowa kluczowe: konteneryzacja; Docker Swarm; Kubernetes; Apache Mesos

*Corresponding author

Email address: adam.pankowski@pollub.edu.pl (A. Pankowski)

1. Wstęp

Coraz więcej firm i organizacji decyduje się na zastosowanie systemu złożonego z mikroserwisów zamiast tradycyjnej architektury monolitycznej [1]. Budowę takiego systemu można zdefiniować w kontekście architektury zorientowanej na usługi jako kompozycję małych, rozproszonych i luźno powiązanych ze sobą bloków konstrukcyjnych, stanowiących komponenty oprogramowania [2]. Struktura ta polega na budowaniu autonomicznych komponentów, które używane są jako bloki do konstruowania złożonych systemów. Każda aplikacja składa się z niezależnych usług, które współdziałają ze sobą poprzez interfejsy programowania aplikacji (API). Wynika to z szybkiego tempa rozwoju technologii we współczesnym świecie. Z drugiej strony, wirtualizacja umożliwia budowanie maszyn wirtualnych, co pozwala na uruchamianie wielu systemów operacyjnych na jednym fizycznym hoście. Konteneryzacja to nie tylko technologie ułatwiające pracę administratorom i programistom, ale także sposób na zwiększenie produktywności i oszczędności dla firm. Rozwiązania te ułatwiają migrację aplikacji pomiędzy środowiskami oraz umożliwiają szybsze skalowanie i uruchamianie aplikacji, ponieważ konteneryzacja pozwala na oddzielenie aplikacji od zależności systemu operacyjnego. Konteneryzacja odgrywa znaczącą rolę w świecie komercyjnym. Skonteneryzowane aplikacje mogą być skalowane w górę lub w dół w zależności od potrzeb. Dzięki elastyczności i krótkiego czasu reakcji na potrzeby rynku kontenery zrewolucjonizowały sposób wdrażania i zarządzania poprzez redukcję złożoności aplikacji i jej zależności, jak również możliwości zbudowania na jej podstawie pojedynczego obrazu kontenera. Dzięki czemu programiści mogą w łatwy sposób dystrybuować swoje programy i uruchamiać je w dowolnym środowisku. Podejmowanie decyzji o wyborze narzędzi i technologii do wykorzystania w różnych kontekstach wymaga ciągłej oceny i analizy możliwości, które są do dyspozycji. Celem niniejszego artykułu jest zbadanie trzech znanych i stosowanych rozwiązań konteneryzacji: Kubernetes, Docker Swarm i Apache Mesos. Analiza dotyczy takich parametrów jak: czas wdrożenia, wykorzystanie pamięci, procesora i dysku, czasu odpowiedzi aplikacji oraz czasu przywrócenia repliki aplikacji przy użyciu mechanizmu autoregeneracji.

W pracy postawiono następujące hipotezy badawcze:

H1. Czas potrzebny na uruchomienie aplikacji za pomocą narzędzia Docker Swarm jest najkrótszy.

H2. Kubernetes zapewnia najlepsze wyniki pod względem planowania obciążenia aplikacji oraz skalowania.

2. Przegląd literatury

Metody konteneryzacji stale się rozwijają, szukając bardziej wydajnych rozwiązań. Dokonano analizy artykułów skupiających się na porównaniu wybranych narzędzi pod względem wdrażania oraz zarządzania kontenerami, a także te, prezentujące innowacyjne rozwiązania tym zakresie.

W artykule [3] badano orkiestratory Docker Swarm, Kubernetes, Apache Mesos oraz Cattle pod względem czasu dostarczenia aplikacji o różniej złożoności oraz skalowalności kontenerów. Do badań wybrane zostały aplikacje: Jenkins na jednym kontenerze, WordPress z dwoma kontenerami i GitLab składający się z czterech kontenerów. Eksperyment wykonano 33 razy. Zebrane wyniki eksperymentalne pokazują, że Kubernetes przewyższa swoje odpowiedniki w przypadku bardzo złożonych wdrożeń aplikacji, podczas gdy inne orkiestratory mogą być lepszym wyborem dla prostszych rozwiązań.

Artykuł [4] przedstawił analizę Kubernetes, Docker Swarm, Mesos i Redhat OpenShift pod kątem różnych parametrów bezpieczeństwa, wdrożenia, stabilności, skalowalności, instalacji klastra oraz czasu nauki obsługi. Zaobserwowano, że Kubernetes ma najlepsze funkcje planowania, podczas gdy Docker Swarm jest łatwy w użyciu. Stwierdzono również dobrą skalowalność orkiestratora Mesos, podczas gdy OpenShift jest wysoce bezpiecznym narzędziem orkiestracji.

W kolejnym artykule [5] skupiono się na ocenie kosztów wydajności za pomocą znanych narzędzi porównawczych. Podjęto szereg badań z wykorzystaniem dobrze znanych narzędzi takich jak Phoronix Test Suite i LiDAR Data Benchmarks, służących do testów porównawczych. Przeprowadzono analizę wydajności narzędzi do orkiestracji kontenerów biorąc pod uwagę ich wady i zalety. Wyniki pokazują, że wydajność Kubernetes jest nieco gorsza niż Docker w trybie Swarm. Jednak Docker w trybie Swarm nie jest tak elastyczny i wydajny jak Kubernetes w bardziej złożonych środowiskach.

Artykuł [6] zawiera analizę porównawczą implementacji „Container Storage Interface" (CSI) typu „bare-metal", która pokazuje zalety i wady istniejących implementacji CSI, na podstawie, której uważa się, że podejście „bare-metal" jest najbardziej wydajne. Posiada ono jednak wiele błędów, które wymagają dalszej pracy nad rozwiązaniem. Istnieje więc potrzeba rozszerzenia CSI na „bare-metal storage provisioning", aby uniknąć kosztów wirtualizacji i chmury oraz zminimalizować ręczne operacje zarządzania pamięcią masową.

W artykule [7] autorzy przedstawili, że obecnie Docker Swarm posiada trzy podstawowe strategie harmonogramowania (spread, binpack i random), każda z nich uruchamia kontener z ustaloną liczbą zasobów. Nowość strategii prezentowanej w artykule polega jednak na wykorzystaniu klasy „Service Level Agreement" (SLA) użytkownika do dostarczenia zasobów kontenera, który musi wykonać usługę, na podstawie dynamicznego obliczenia liczby rdzeni CPU, które muszą być przydzielone kontenerowi zgodnie z klasą SLA użytkownika i obciążeniem maszyn równoległych w infrastrukturze. Testy nowej strategii zostały przeprowadzone, poprzez emulację, na różnych częściach ogólnego szkieletu i pokazują potencjał podejścia do dalszego rozwoju.

Technologia Apache Mesos jest bliżej przedstawiona w artykule [8]. Ma ona na celu zapewnienie wysokiego wykorzystania klastra poprzez ziarnisty podział i sprawiedliwość przyznawania zasobów wśród wielu użytkowników poprzez alokację opartą na „Dominant Resource Fairness" (DRF). Technologia ta bierze pod uwagę różne typy zasobów, procesor, pamięć czy dysk, wymagane przez każdą aplikację i określa udział każdego zasobu klastra, który może być przydzielony aplikacjom. Mesos przyjął dwupoziomową politykę szeregowania: DRF do przydzielania zasobów między konkurencyjnymi strukturami aplikacyjnymi oraz szeregowanie na poziomie zadań przez każdą strukturę aplikacyjną dla zasobów przydzielonych w poprzednim kroku.

W artykule [9] proponowany jest system pośrednictwa „Workload aware Energy Efficient Container" (WEEC), aby zaoszczędzić zużycie energii spowodowane przez uruchomione aplikacje kontenerowe, jednocześnie gwarantując akceptowalny poziom wydajności. Dodatkowo, przedstawiono heterogeniczność mocy i wydajność kilku serwerów kontenerowych Docker na podstawie wyników eksperymentalnych uzyskanych z urządzeń do pomiaru mocy. W ten sposób zidentyfikowano korzyści wynikające z wydajności proponowanego systemu.

Na podstawie przedstawionego przeglądu literatury można zauważyć, że brakuje szczegółowej analizy dotyczącej wydajności orkiestratorów uwzględniającej takie parametry jak: czas wdrożenia, obciążenie procesora, wykorzystanie pamięci, czas odpowiedzi aplikacji oraz czas przywrócenia repliki aplikacji przy użyciu mechanizmu autoregeneracji. Ponadto badania przeprowadzone w wymienionych artykułach zostały przeprowadzone na starszych wersjach oprogramowania.

3. Narzędzia orkiestracji kontenerów

Docker Swarm to pierwsze omawiane narzędzie do zarządzania kontenerami. Jedną z głównych cech wyróżniających tę technologię jest jej prostota konfiguracji i uruchamiania. Dzięki temu, że jest zintegrowana z Docker, nie wymaga dodatkowych instalacji ani zewnętrznych narzędzi. Tworzenie i zarządzanie klastrami za pośrednictwem Docker Swarm jest najprostsze spośród badanych orkiestratorów. Zapewnia zautomatyzowane równoważenie obciążenia w klastrach Docker, podczas gdy inne narzędzia do orkiestracji kontenerów wymagają działań manualnych [10]. Docker Swarm oferuje elastyczność w zależności od potrzeb użytkownika. Może działać zarówno w trybie jednego węzła,

gdzie wszystkie kontenery uruchamiane są na pojedynczej maszynie, jak i w trybie wielu węzłów, gdzie kontenery są rozproszone na różnych maszynach, tworząc klaster. Możliwe jest wtedy skalowanie aplikacji i dostosowywanie klastra do potrzeb organizacji. Narzędzie to jest kompatybilne z systemem Docker, co oznacza, że korzysta z tych samych narzędzi, interfejsów i obrazów kontenerowych. Dzięki temu użytkownicy pracujący z Dockerem mogą łatwo przenieść swoje aplikacje do klastra Swarm. Zapobiega on awariom dzięki współistnieniu wielu węzłów zarządzających w klastrze, aby odzyskać sprawność po awarii bez żadnych przestojów. Gdy istniejący lider nie działa lub nie jest dostępny, grupa wybierze nowego do prowadzenia zadań orkiestracji. Wiele firm, które korzystają z technologii kontenerowej, nadal z powodzeniem stosuje Docker Swarm do wdrażania i zarządzania swoimi aplikacjami [4]. Docker Swarm jest szeroko stosowany w małych i średnich przedsiębiorstwach oraz w zastosowaniach, gdzie prostota i łatwość konfiguracji są kluczowymi czynnikami, a ekosystem nie jest zaawansowany do tego stopnia, żeby potrzebne były bardziej wyspecjalizowane narzędzia.

Drugim analizowanym narzędziem jest Kubernetes będący jednym z najpopularniejszych i najdynamiczniej rozwijających się projektów w dziedzinie konteneryzacji i orkiestracji aplikacji [11]. Od momentu, gdy został udostępniony jako otwarte oprogramowanie zyskał ogromną popularność i zdobył wsparcie wielu wiodących firm technologicznych na całym świecie. Wielu gigantów branży technologicznej, takich jak Google, Microsoft, Amazon, IBM, Intel, Cisco czy Red Hat aktywnie wykorzystuje i rozwija tę technologię. Firmy te wnoszą znaczący wkład w rozwój projektu, udostępniając narzędzia, usługi chmurowe, integracje oraz innowacje. Kubernetes, znany również jako „K8s" jest systemem zarządzania kontenerami, który został opracowany przez Google na podstawie ich wewnętrznego systemu zarządzania kontenerami, znanego jako Borg. Jest to platforma do automatycznego wdrażania, skalowania i zarządzania aplikacjami kontenerowymi. Kubernetes integruje technologie i narzędzia, takie jak kontenery, automatyzację operacji i zarządzanie zasobami. Pozwala na scentralizowane i automatyczne wdrażanie aplikacji w kontenerach, skalowanie aplikacji w zależności od obciążenia oraz zapewnia jednolite środowisko uruchomieniowe niezależnie od infrastruktury. Podobnie jak Docker Swarm, Kubernetes również został opracowany w języku Go, a wdrażanie kontenerów również opiera się na wykorzystaniu języka serializacji YAML [12]. Współpraca wielu gigantów na rynku z Kubernetes świadczy o szerokim zainteresowaniu i uznaniu dla tej technologii, która stała się standardem przemysłowym w dziedzinie orkiestracji kontenerów. Dzięki temu Kubernetes nie tylko rozwija się intensywnie, ale także posiada bogaty ekosystem i wsparcie ze strony wiodących graczy na rynku technologicznym.

Apache Mesos to otwarte oprogramowanie, które ułatwia orkiestrację zasobów w środowiskach rozproszonych. Posiada system zarządzania, który może efektywnie wykorzystywać i udostępniać zasoby obliczeniowe w klastrze. Jedną z najważniejszych cech Mesos jest możliwość współdzielenia zasobów pomiędzy różnymi aplikacjami. Pozwala to na uruchamianie różnych typów aplikacji na tym samym klastrze, poprawiając wykorzystanie zasobów i wydajność. Mesos podobnie jak inne platformy do zarządzania kontenerami umożliwia efektywne wykorzystanie dostępnych zasobów i automatyczne skalowanie aplikacji w zależności od potrzeb. Działa on w modelu „master-service", w którym węzeł główny pełni rolę koordynatora, a węzły podrzędne zapewniają zasoby do wykonywania zadań. Może obsługiwać aplikacje kontenerowe, takie jak Docker, a także aplikacje na wirtualnych maszynach lub te działające bezpośrednio na systemie operacyjnym hosta. Rozwiązania z użyciem klastra Mesos są często wspomagane szkieletem zarządzającym Marathon. Marathon współdziała z komponentem głównym, zapewniając orkiestrację dla całego klastra Mesos. Tak więc, w przypadku błędu węzła podrzędnego, Marathon uruchamia nową instancję, aby zagwarantować tolerancję błędów [4]. Apache Mesos jest szeroko stosowany w dużych i złożonych środowiskach, w których konieczne jest zarządzanie dużymi zasobami obliczeniowymi. Jest używany przez wiele znanych firm i organizacji, takich jak Twitter, Airbnb, Apple i Netflix do skalowania i zarządzania ich infrastrukturą.

4. Metody badawcze

Celem badań było przeprowadzenie analizy porównawczej narzędzi do orkiestracji kontenerów aplikacji oraz zbadanie, który orkiestrator spełnia dane zadanie najlepiej. Badanie przeprowadzono dla narzędzi Docker Swarm, Kubernetes i Apache Mesos. Metryki obejmują czas potrzebny na uruchomienie aplikacji dla każdego z orkiestratorów. Następnie zbadano wykorzystanie zasobów sprzętowych z uruchomioną aplikacją testową, przepustowość dla nagłego wzrostu żądań oraz czas przywrócenia repliki po nieoczekiwanej awarii. Badania zostały przeprowadzone z użyciem oprogramowania o otwartym kodzie Jenkins.

4.1. Pomiar czasu uruchomienia aplikacji

Scenariusz badawczy dotyczący pomiaru czasu uruchomienia aplikacji zakłada monitorowanie procesu tworzenia kontenerów aplikacji oraz mierzenie czasu od rozpoczęcia tego procesu do momentu, kiedy aplikacja jest w pełni gotowa do działania. Badanie zostało przeprowadzone dla dwóch różnych przypadków, z uwzględnieniem jednej oraz pięciu replik aplikacji, aby zbadać różnicę w czasie uruchomienia w zależności od liczby replik. W celu dokładnego pomiaru, zastosowano analizę logów z poszczególnych kontenerów zawierających aplikację. Logi dostarczyły niezbędnych informacji dotyczących kolejnych etapów procesu uruchamiania, takich jak inicjalizacja kontenera, pobieranie obrazu aplikacji, konfiguracja środowiska, instalacja zależności, aż do momentu, gdy aplikacja jest w pełni gotowa do obsługi żądań. Poprzez analizę tych logów, możliwe było precyzyjne zarejestrowanie momentu, w

którym każda replika aplikacji osiągnęła stan gotowości. Czas uruchomienia został zdefiniowany jako różnica pomiędzy momentem rozpoczęcia tworzenia kontenera a momentem, kiedy aplikacja była w pełni gotowa do obsługi żądań. W celu zredukowania przypadkowości wyników, badanie powtórzono 33 razy.

4.2. Zbadanie obciążenia procesora i wykorzystanie pamięci.

Drugim scenariuszem badawczym było zbadanie obciążenia procesora i wykorzystanie pamięci podczas pracy orkiestratora. W celu dokładnego pomiaru, przeprowadzono badania dla dwóch różnych przypadków – z jedną i pięcioma replikami aplikacji. Aby przeprowadzić badania dla środowisk Docker Swarm Kubernetes oraz Apache Mesos, skorzystano z aplikacji Docker Desktop. Udostępnia ona metryki dotyczące wykorzystania pamięci, procesora i dysku bez konieczności dodatkowej instalacji lub konfiguracji narzędzi monitorujących. Dodatkowo, dla Apache Mesos, metryki dotyczące obciążenia procesora i wykorzystania pamięci zostały odczytane również z wbudowanego interfejsu Mesos Web UI [13]. W wyniku tego, można było monitorować obciążenie tych podzespołów w czasie rzeczywistym i zbierać dane do analizy. Badanie zostało wykonane 33 razy aby zmniejszyć losowość wyników. Aplikacja Jenkins z zainstalowanymi polecanymi wtyczkami zużywa znaczącą ilość pamięci podczas pracy. Podczas analizy wyników można łatwo zauważyć jak każdy orkiestrator radzi sobie z optymalizacją podczas replikacji aplikacji.

4.3. Zbadanie przepustowości i niezawodności.

Kolejnym scenariuszem badawczym było zbadanie przepustowości i niezawodności systemu poprzez testy wydajnościowe z użyciem pakietów danych. Metoda ta polegała na wysyłaniu dużej liczby pakietów danych przez system i mierzeniu ilości danych, które zostały prawidłowo przesłane. W tym celu użyto narzędzia Apache JMeter [14]. Umożliwia ono symulowanie dużej liczby użytkowników i generowanie intensywnego ruchu w celu przetestowania przepustowości systemu. W aplikacji JMeter zastosowano scenariusz testowy, który zawierał wejście na stronę powitalną badanej aplikacji oraz odczytanie tekstu z głównego widoku. W tym celu zastosowano 2000 hostów, które powtarzały operację 30 razy, co łącznie dawało 60000 operacji testowych. Liczba hostów została dobrana w celu symulacji dużego obciążenia systemu, często występującego w rzeczywistych warunkach użytkowania. Podczas przeprowadzania testów, zbierano dane dotyczące liczby prawidłowo przesłanych pakietów danych, jak i tych które nie powiodły się. Celem tego badania było zbadanie, jak system radzi sobie z dużą liczbą operacji i czy jest w stanie utrzymać odpowiednią przepustowość przy intensywnym ruchu.

4.4. Pomiar skuteczności mechanizmów autoregeneracji narzędzi.

Ostatnim scenariuszem badawczym jest pomiar skuteczności mechanizmów autoregeneracji orkiestratorów. Celem badania było sprawdzenie, jak każdy z orkiestratorów radzi sobie z błędem krytycznym, polegającym na zatrzymaniu jednej z replik aplikacji. Obserwowano, jak szybko orkiestrator podejmuje działania w celu przywrócenia usługi. Czas, który upłynął od momentu zatrzymania repliki do momentu, kiedy nowa replika została uruchomiona i aplikacja była ponownie w pełni dostępna został odczytany i zarejestrowany. Pomiar został wykonany za pomocą odczytania dokładnego czasu inicjalizacji nowej repliki w logach Docker, przeprowadzony został 33 razy w celu zredukowania błędów.

4.5. Środowisko testowe

Tabela 1: Środowisko testowe

Środowisko	
Procesor	Intel Core i7-13700KF
Ilość rdzeni	8
Pamięć RAM	32 GB
System	Ubuntu 20.04
Dysk	1 TB SSD
Karta sieciowa	Realtek GbE Network Adapter

Tabela 2: Narzędzia testowe

Narzędzie	wersja
Docker Swarm	23.0.5
Kubernetes	1.25.9
Apache Mesos	1.11.0
Apache JMeter	5.5
Docker Desktop	4.19.0

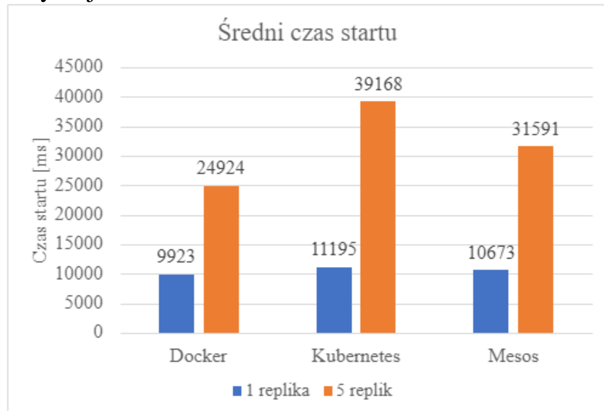
W tabelach 1 i 2 przedstawiono informacje dotyczące środowiska testowego, takie jak specyfikacje sprzętowe, wersje systemu operacyjnego oraz wykorzystane narzędzia. Wszystkie testowane narzędzia orkiestracji kontenerów: Docker Swarm, Kubernetes oraz Apache Mesos korzystały z demonów dockerd, które pełniły rolę silników kontenerów.

5. Wyniki badań

5.1. Pomiar czasu startu

Na rysunku 1 przedstawiono średnie czasy uruchomienia aplikacji dla każdego z orkiestratorów. Celem tego badania było zbadanie wydajności i efektywności trzech popularnych orkiestratorów: Docker Swarm, Kubernetes i Apache Mesos. Badanie przeprowadzono dla jednej oraz pięciu replik 33 razy. W celu prezentacji dokładnych pomiarów czas został podany w milisekundach (Rysunek 1). Na podstawie wyników z Rysunku 1 moż-

na zauważyć, że Docker osiągnął najkrótszy czas startu aplikacji w porównaniu do Kubernetes i Mesos dla pojedynczej repliki, co potwierdza hipotezę H1. Wraz ze wzrostem liczby replik, wszystkie trzy orkiestratory miały wydłużony czas startu aplikacji. Jednak nadal Kubernetes i Mesos wykazywały dłuższe czasy uruchamiania niż Docker. Spowodowane jest to natywną naturą tej technologii dla zastosowanego silnika konteneryzacji.



Rysunek 1: Średnie czasy startu aplikacji.

Z tabel 3 i 4 można odczytać mediany oraz odchylenia standardowe czasu startu aplikacji zarówno dla jednego jak i pięciu replik. Mediana każdego pomiaru jest zbliżona do wartości średniej co potwierdza stabilność narzędzi dla przeprowadzonych testów. Odchylenia standardowe z niskimi wartościami pokazują, że wyniki scenariuszy były do siebie zbliżone.

Tabela 3: Mediana i odchylenie standardowe czasu startu aplikacji dla jednej repliki

	Mediana [ms]	Odchylenie standardowe [ms]
Docker	9786,00	442,45
Kubernetes	11258,00	219,76
Mesos	10589,50	407,84

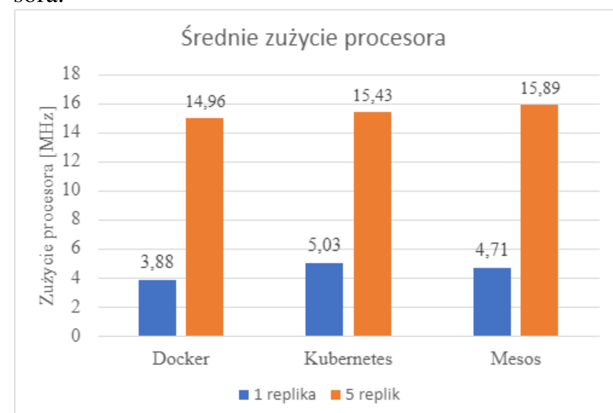
Tabela 4: Mediana i odchylenie standardowe czasu startu aplikacji dla pięciu replik

	Mediana [ms]	Odchylenie standardowe [ms]
Docker	22493,50	1367,21
Kubernetes	39721,00	1441,33
Mesos	30493,50	1167,21

5.2. Pomiar obciążenia podzespołów

Następne badanie skupiało się na pomiarze obciążenia podzespołów takich jak procesor i pamięć RAM przez każdy z orkiestratorów z uruchomioną aplikacją testową. Obciążenie procesora zostało podane w MHz, aby otrzymać precyzyjne dane na temat obciążenia podczas działania aplikacji testowej. Zużycie pamięci RAM ze względu na specyfikację testowanej aplikacji podano w GB. Na podstawie badania można było ocenić, jak dużo zasobów potrzebował zaalokować każdy orkiestrator, aby obsłużyć aplikację w sposób efektywny. Dzięki przeprowadzeniu testu 33 razy dla każdego scenariusza, możliwe było uzyskanie optymalnych wyników oraz zapewniło wiarygodność obserwacji.

Na rysunku 2 przedstawiono diagram średniego obciążenia procesora podczas pracy aplikacji. W przypadku jednej repliki podobnie jak czas startu aplikacji Docker Swarm uzyskał najmniejsze zużycie procesora dzięki bliskiej współpracy z silnikiem konteneryzacji Docker. Kubernetes i Mesos uzyskały wyniki zbliżone do siebie, lecz już nakład infrastruktury wymagany przez te platformy jest bardziej widoczny. W przypadku pięciu replik wszystkie trzy narzędzia uzyskały bardzo podobne wyniki. Biorąc pod uwagę przebieg badań jednej repliki można stwierdzić, że Kubernetes najlepiej poradził sobie ze skalowaniem replik aplikacji co potwierdza hipotezę H2. Niemniej jednak to Swarm po raz kolejny uzyskał najmniejszą wartość obciążenia procesora.



Rysunek 2: Średnie zużycie procesora.

Tabela 5 i 6 zawiera miarę median i odchylenia standardowe dla wykorzystania pamięci przez testowaną aplikację dla jednej i pięciu replik.

Tabela 5: Mediana i odchylenie standardowe obciążenia procesora dla jednej repliki

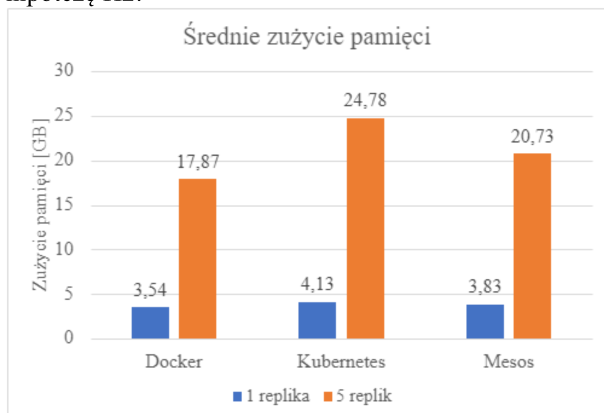
	Mediana [MHz]	Odchylenie standardowe [MHz]
Docker	3,40	1,16
Kubernetes	4,93	1,23
Mesos	4,38	0,98

Mediana posiada podobną wartość do średniego zużycia pokazanego na Rysunku 2 dla każdego narzędzia. Odchylenie standardowe oscyluje wokół 1 MHz dla jednej oraz 2,4 MHz dla pięciu replik. Są to stosunkowo niskie wartości, które wskazują na stabilne zużycie zasobów.

Tabela 6: Mediana i odchylenie standardowe obciążenia procesora dla pięciu replik

	Mediana [MHz]	Odchylenie standardowe [MHz]
Docker	14,28	2,62
Kubernetes	14,28	2,48
Mesos	15,98	2,01

Wyniki średniego wykorzystania pamięci RAM przedstawione na Rysunku 3 po raz kolejny wskazują na Docker Swarm jako najlepiej zoptymalizowane narzędzie dla małych systemów. Pomimo użycia tego samego obrazu aplikacji z identycznymi wtyczkami, poradził sobie najlepiej z optymalizacją aż o ponad 0,5GB dla jednej repliki w stosunku do narzędzia Kubernetes. Mesos prezentuje się pobodnie ze średnią wartością zajętości pamięci większą o około 0,3GB. W przypadku pięciu replik każdy orkiestrator uzyskał proporcjonalne wyniki około 5 razy większe od pojedynczej repliki. Jedynie Kubernetes uzyskał większą wartość na poziomie 6 krotności swojej pojedynczej repliki, co obala hipotezę H2.



Rysunek 3: Średnie zużycie pamięci RAM.

W tabelach 7 i 8 przedstawiono wartości mediany oraz odchylenia standardowego dla badań wykorzystania pamięci dla RAM jednej i pięciu replik aplikacji. Wartości mediany są po raz kolejny bardzo zbliżone do wartości średnich, co wskazuje na równomierność uzyskanych wyników. Dodatkowo, odchylenia standardowe są minimalne, co oznacza, że zmienność w zajętości pamięci jest niewielka. Gwarantuje to stabilny poziom zużycia pamięci RAM podczas przebiegu testów.

Tabela 7: Mediana i odchylenie standardowe wykorzystania pamięci RAM dla jednej repliki

	Mediana [GB]	Odchylenie standardowe [GB]
Docker	3,58	0,148
Kubernetes	4,12	0,073
Mesos	3,83	0,077

Tabela 8: Mediana i odchylenie standardowe wykorzystania pamięci RAM dla pięciu replik

	Mediana [GB]	Odchylenie standardowe [GB]
Docker	17,75	0,054
Kubernetes	24,78	0,011
Mesos	20,88	0,071

5.3. Pomiar przepustowości

Kolejny pomiarem jest badanie przepustowości i niezawodności orkiestratorów. Celem badania było zbadanie wydajności każdego orkiestratora w przypadku nagłego wzrostu obciążenia aplikacji. Zarządzanie ruchem dla każdego orkiestratora zostało przeprowadzone przez narzędzie równoważenia obciążenia. Dla Docker Swarm użyto wbudowany Ingress, Kubernetes korzystał z zadeklarowanego serwisu, w przypadku Mesos był to Nginx. Automatyczne skalowanie jest ważną cechą, która powinna być prawidłowo zaimplementowana w narzędziu.

Label	Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received KB/s	Sent KB/s
HomePage	60000	604	261	1659	2373	3746	3	9455	1.41%	2027.36	43280.67	226.43
TOTAL	60000	604	261	1659	2373	3746	3	9455	1.41%	2027.36	43280.67	226.43

Rysunek 4: Tabela raportu wygenerowanego dla testów obciążeniowych narzędzia JMeter dla Docker Swarm.

Label	Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received KB/s	Sent KB/s
HomePage	60000	892	78	3780	4367	5150	4	7441	0.41%	1420.7	31299.83	161.66
TOTAL	60000	892	78	3780	4367	5150	4	7441	0.41%	1420.7	31299.83	161.66

Rysunek 5: Tabela raportu wygenerowanego dla testów obciążeniowych narzędzia JMeter dla Kubernetes.

Label	Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received KB/s	Sent KB/s
HomePage	60000	742	143	2741	3439	4671	4	8647	0.71%	1824.65	38532.74	170.75
TOTAL	60000	742	143	2741	3439	4671	4	8647	0.71%	1824.65	38532.74	170.75

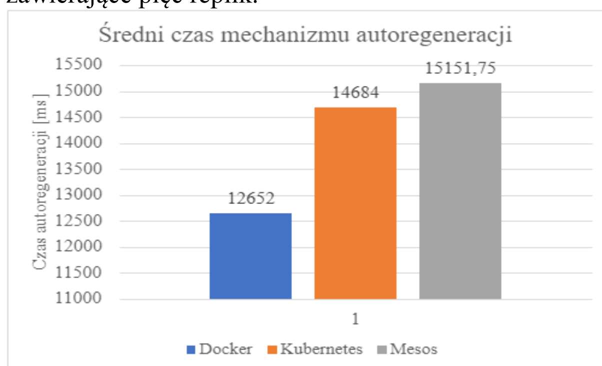
Rysunek 6: Tabela raportu wygenerowanego dla testów obciążeniowych narzędzia JMeter dla Apache Mesos.

Przeprowadzono badania, których celem było ocenienie wydajności różnych narzędzi w kontekście działania aplikacji na 2000 hostach. Testy opierały się na scenariuszu polegającym na uruchomieniu aplikacji i odczytaniu tekstu powitalnego "Welcome to Jenkins!". Całkowita liczba żądań wyniosła 60000, przy czym każdy host przeprowadzał test 30 razy, z 10-sekundowym czasem rozruchu. Wyniki testów zostały przedstawione na Rysunkach 4, 5 i 6, zawierają one informacje dotyczące sumy żądań, wartości minimalnej, maksymalnej, średniej i mediany czasu odpowiedzi, ilości wysłanych i przyjętych danych oraz poziomu przepustowości. Ważnym aspektem są także percentyle o wartościach 90%, 95% i 99%. Określają one wartości, poniżej których znajduje się odpowiedni procent wyników. Z analizy wynika, że Docker Swarm uzyskał najkrótszy i najdłuższy czas odpowiedzi, a także najlepszą wydajność przesyłu danych spośród badanych narzędzi. Jednocześnie, miał on również największą liczbę błędów na poziomie 1.41%, co oznacza, że 1.41% żądań nie powiodło się z powodu braku odpowiedzi. Najlepsze rezultaty w przypadku odporności na błędy zyskał Kubernetes, gdzie wskaźnik niepowodzeń wynosił około 0.40%. Pozostałe badane narzędzia wykazały podobne

wyniki pod względem czasu odpowiedzi i wydajności w obliczu nagłego wzrostu ruchu w aplikacji. Podsumowując, wyniki badań wskazują, że Docker Swarm wykazał się najlepszą ogólną wydajnością w zależności od mierzonej metryki. Pozostałe narzędzia radziły sobie podobnie w zakresie obsługi wzrostu ruchu, jednak Kubernetes osiągnął najniższy wskaźnik niepowodzeń.

5.4. Pomiar skuteczności mechanizmów autoregeneracji

Ostatnim z przeprowadzonych badań był pomiar mechanizmów autoregeneracji testowanych platform orkiestracji. Pomiar wykonano umyślnie wywołując usterkę, która zatrzyma działanie aplikacji oraz odczytano czas od zatrzymania do w pełni przywróconego kontenera z aplikacją. W tym celu zastosowano środowiska testowe zawierające pięć replik.



Rysunek 7: Średni czas regeneracji kontenera.

Na rysunku 7 przedstawiono średni czas, jaki potrzebował każdy z badanych orkiestratorów na przywrócenie repliki z aplikacją do stanu gotowości. Wynik został przedstawiony w milisekundach. Stanowczo najlepszy wynik, lepszy aż o 2 sekundy od Kuberntesa oraz o 3 sekundy od Mesosa zyskał Docker Swarm. Po raz kolejny narzędzie natywne do silnika konteneryzacji Docker radzi sobie najlepiej z małymi klastrami (do kilkudziesięciu podów). Kubernetes oraz Mesos uzyskały wynik zbliżony do siebie.

Tabela 9: Mediana i odchylenie standardowe czasu autoregeneracji kontenera

	Mediana [ms]	Odchylenie standardowe [ms]
Docker	12979	929,36
Kubernetes	14684	713,47
Mesos	15234	599,21

W Tabeli 9 przedstawione są mediana oraz odchylenie standardowe czasu autoregeneracji kontenera z aplikacją dla każdego narzędzia. Niskie wartości odchyleń standardowych potwierdzają stabilność przeprowadzonych badań. Mediana pokrywa się ze średnią wartością a odchylenie standardowe nie przekracza 1 sekundy. Z analizy uzyskanych danych wynika, że mechanizmy przywracania repliki po awarii działają przewidywalnie oraz bez większych zakłóceń.

6. Wnioski

Niniejszy artykuł przedstawia analizę narzędzi orkiestracji kontenerów, porównując ich funkcje i usługi. Opracowano kompleksowy zestaw wskaźników do oceny wydajności tych narzędzi pod względem planowania i zarządzania usługami. Wybrano trzy reprezentatywne orkiestratory, a mianowicie Docker Swarm, Kubernetes i Apache Mesos, i przeprowadzono szczegółowe badanie i ocenę każdego z nich. Pod względem wydajności, badanie wykazało, że Docker Swarm jest obecnie jednym z najwydajniejszych dostępnych orkiestratorów (Rysunek 1-7), co tłumaczy jego popularność wśród praktyków. Złożona architektura Kubernetes oraz Apache Mesos może czasami powodować znaczne obciążenie, co może wpływać na ich wydajność. Przodują one pod względem funkcjonalności. Wyniki tego badania dostarczyły cennych spostrzeżeń i służą jako podstawa do przyszłych badań. W celu głębszego zbadania narzędzi należy użyć rozbudowanego klastra z wieloma powiązanymi elementami. Przeprowadzenie badań na takim środowisku dostarczy szczegółowych danych na temat cech każdego orkiestratora. Wyniki przeprowadzonego badania (Rysunek 1) wyraźnie potwierdzają hipotezę H1. Docker Swarm potrzebował najmniej czasu na uruchomienie aplikacji testowej. Wyniki badania zużycia procesora potwierdzają hipotezę badawczą H2. Dla pięciu replik aplikacji platforma Kubernetes uzyskała najlepszą efektywność skalowania (Rysunek 2).

Przeprowadzone badania są zgodne z wynikami przytoczonych artykułów. Analiza skupiała się na małym środowisku kontenerowym oraz zbadaniu wydajności platform w zarządzaniu nim. Potwierdzono najlepszą pozycję orkiestratora Docker Swarm dla tego typu wdrożeń.

Literatura

[1] Mikrousługi a architektura monolityczna, https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith, [07.06.2023].

[2] J. Stubbs, W. Moreira, R. Dooley, Distributed systems of microservices using Docker and Serfnode, 7th International Workshop on Science Gateways (2015) 34–39, https://doi.org/10.1109/iwsg.2015.16.

[3] I. M. A. Jawarneh et al., Container Orchestration Engines: A Thorough Functional and Performance Comparison, ICC 2019 - 2019 IEEE International Conference on Communications (2019) 1-6, https://doi.org/10.1109/ICC.2019.8762053.

[4] A. Malviya, R. K. Dwivedi, A Comparative Analysis of Container Orchestration Tools in Cloud Computing, 9th International Conference on Computing for Sustainable Global Development (2022) 698-703, https://doi.org/10.23919/INDIACom54597.2022.9763171.

[5] Y. Pan, I. Chen, F. Brasileiro, G. Jayaputera, R. Sinnott, A Performance Comparison of Cloud-Based Container

Orchestration Tools, IEEE International Conference on Big Knowledge (2019) 191-198, https://doi.org/10.1109/ICBK.2019.00033.

[6] A. Shemyakinskaya, I. Nikiforov, Disk Space Management Automation with CSI and Kubernetes. Proceedings of Seventh International Congress on Information and Communication Technology. Lecture Notes in Networks and Systems 447 (2023) 171-179, https://doi.org/10.1007/978-981-19-1607-6_15.

[7] C. Cérin, T. Menouer, W. Saad, W. B. Abdallah, A New Docker Swarm Scheduling Strategy, IEEE 7th International Symposium on Cloud and Service Computing (2017) 112-117, https://doi.org/10.1109/SC2.2017.24.

[8] P. Saha, A. Beltre, M. Govindaraju, Exploring the Fairness and Resource Distribution in an Apache Mesos Environment, IEEE 11th International Conference on Cloud Computing (2018) 434-441, https://doi.org/10.1109/CLOUD.2018.00061.

[9] D. K. Kang, G. B. Choi, S. H. Kim, I. S. Hwang, C. H. Youn, Workload-aware resource management for energy efficient heterogeneous Docker containers, IEEE Region 10 Conference (2016) 2428-2431, https://doi.org/10.1109/TENCON.2016.7848467.

[10] Porównanie Docker Swarm i Kubernetes, https://circleci.com/blog/docker-swarm-vs-kubernetes/, [07.06.2023].

[11] Ankieta CNCF 2022, https://www.cncf.io/reports/cncf-annual-survey-2022/, [07.06.2023].

[12] Porównanie Kubernetes, Mesos oraz Docker Swarm, https://www.sumologic.com/insight/kubernetes-vs-mesos-vs-swarm/, [07.06.2023].

[13] Dokumentacja Apache Mesos, https://mesos.apache.org/documentation/lates, [07.06.2023].

[14] Dokumentacja Apache JMeter, https://jmeter.apache.org, [07.06.2023].