JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 14 (2020) 31-36
Wysłane: 2019-10-14
Przyjęte: 2019-10-18

Wpływ języka programowania aplikacji chmurowej na wydajność jej implementacji w wybranych środowiskach serverless

Krzysztof Bezrąk*, Sławomir Przyłucki
Politechnika Lubelska, Katedra Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. Ostatnie lata rozwoju technologii chmurowych przyniosły gwałtowny wzrost zainteresowania rozwiązaniami określanymi jako systemy bezserwerowe. Ich wydajność, a tym samym przydatność w potencjalnych zastosowaniach, jest silnie uzależniona od sposobu implementacji programowej konkretnych zadań. W artykule poddano analizie wpływ wybranych, obecnie najpopularniejszych, języków programowania na wydajność testowej infrastruktury bezserwerowej uruchomionej w środowisku zarządzanym przez system Kubernetes. Zgromadzone dane posłużyły do sformułowania wniosków dotyczących przydatności poszczególnych języków w warunkach zróżnicowanych obciążeń systemu bezserwerowego.

Słowa kluczowe: Kubernetes; serverless; Kubeless

*Autor do korespondencji.

Adres e-mail: krzysztof.bezrak@pollub.edu.pl

Impact of the cloud application programming language on the performance of its implementation in selected serverless environments

Krzysztof Bezrąk*, Sławomir Przyłucki
Deaprtment of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. Recent years of cloud technology development have brought a sharp increase in interest in solutions known as serverless systems. Their performance, and thus usefulness in potential applications, strongly depends on the method of program implementation of specific tasks. The article analyzes the impact of selected, currently the most popular, programming languages on the performance of the serverless test infrastructure running in an environment managed by the Kubernetes system. The collected data were used to formulate conclusions regarding the suitability of individual languages in the conditions of varying serverless system loads.

Keywords: Kubernetes; serverless; Kubeless

*Corresponding author.

E-mail address/addresses: krzysztof.bezrak@pollub.edu.pl

1. Wstęp

Systemy chmurowe stały się nieodłącznym komponentem współczesnych systemów informatycznych i są tym samym powszechnie wykorzystywane jako środowiska programistyczne zarówno w dużych korporacjach, jak i małych firmach informatycznych. Dla tych drugich jednak często koszty rozwiązań chmurowych oraz trudność ich wdrożenia stanowiły ogromną barierę, co było jednym z powodów dla których powstała technologia systemów bezserwerowych (ang. serverless). Dostarcza ona skalowalne środowiska uruchomieniowe aplikacji, w których płaci się tylko za wykorzystane przez nią zasoby i czas jej działania. Czynnikiem bezpośrednio wpływającymi na powyższe parametry jest sposób implementacji programowej danego zadania. Ponieważ w chwili obecnej wybór języków jest uzależniony od wielu czynników i nie ogranicza się do jednego, dedykowanego rozwiązania, celowym jest identyfikacja wpływu wyboru języka na wydajność systemów serverless w różnych scenariuszach ich użycia.

2. Cel i teza

Celem artykułu jest zbadanie wpływu języka programowania danej aplikacji napisanej w technologii serverless na jej wydajność. Na jego potrzeby zostały przygotowane zbliżone do siebie aplikacje wykonujące podobny algorytm w wybranych językach programowania. W ramach badania zostały przeprowadzone testy polegające na uruchamianiu tychże aplikacji i pomiarze czasów ich odpowiedzi przy różnych scenariuszach obciążenia. Analiza wyników powyższych testów w powiązaniu z wiedzą teoretyczną na temat języków programowania i środowisk uruchomieniowych aplikacji umożliwiła wyciągnięcie i uogólnienie wniosków w obszarze zdefiniowanym celem badawczym.

Badanie zostało przeprowadzone na serwerze z zainstalowanym systemem Kubernetes, w którym zostanie uruchomione środowisko Kubeless. System Kubernetes został wybrany ze względu na wiodącą rolę w zakresie zarządzania skonteneryzowanymi aplikacjami, a takimi są aplikacje serverless i rosnącą popularność [1]. Kubeless jest jednym z rozwiązań umożliwiających obsługę aplikacji serverless w systemie Kubernetes, jego wybór uwarunkowała największa

ilość wspieranych języków programowania w porównaniu z konkurencją, a także wsparcie komercyjne zapewniane przez firmę Bitnami. Serwer w miarę możliwości nie będzie zawierał dodatkowego oprogramowania poza tym, które będzie potrzebne dla systemu Kubernetes aby wyeliminować czynniki mogące potencjalnie wpłynąć na wyniki pomiarów. Serwer będzie podłączony do Internetu i do sieci lokalnej.

Na powyższym serwerze umieszczone zostaną aplikacje serverless napisane w poniższych językach programowania:

- Golang,
- Java,
- JavaScript,
- Python.

Języki te zostały wybrane ze względu na popularność w ostatnim czasie, szczególnie w platformach serverless. Różnią się też od siebie pod względem architektury - JavaScript i Python to języki skryptowe, interpretowalne, Golang kompiluje się statycznie do postaci kodu maszynowego, a Java kompiluje się do kodu bajtowego wykonywanego przez maszynę wirtualną (JVM). Wszystkie są też wspierane przez wybraną platformę serverless – Kubeless.

W tej samej sieci lokalnej co serwer będzie podłączony drugi komputer z którego będą wykonywane pomiary. Wynika to z potrzeby zminimalizowania ewentualnego, negatywnego wpływu innych aktywnych urządzeń i powiązanych z nimi usług na rezultaty pomiarów. Pomiary będą wykonywane przy pomocy narzędzia Apache Benchmark (ab), które służy do mierzenia wydajności aplikacji komunikujących się poprzez protokół HTTP. O jego wyborze zadecydowała możliwość definiowania ilości równoczesnych połączeń z serwerem a także sposób prezentowania rezultatów pomiaru.

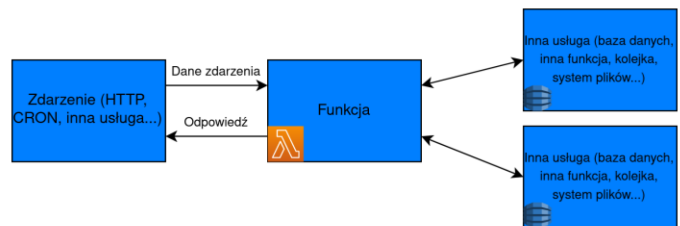
3. Systemy bezserwerowe

3.1. Cechy charakterystyczne systemów bezserwerowych

Systemy bezserwerowe dostarczają usługi serwerowe rozliczane na podstawie ich rzeczywistego zużycia, uruchamiane na serwerach dostawcy tychże usług. Cechują się one niskim nakładem pracy związanej z zarządzaniem infrastrukturą, wysoką skalowalnością i dostępnością. Takie podejście tworzy alternatywę dla standardowych wdrożeń aplikacji, w których poza napisaniem kodu należy także stworzyć i utrzymać środowisko na którym będzie ona uruchomiona. Przeważnie dostawcy takich usług rozliczają klienta bazując na pojedynczych uruchomieniach aplikacji, dzięki czemu nie trzeba płacić za ciągłe utrzymywanie serwerów gdy nie są wykorzystywane. Systemy serverless mają zastosowanie szczególnie w procesach, które są uruchamiane na podstawie określonego wyzwalacza, np. wysłanie powiadomienia na e-mail po wysyłce formularza kontaktowego, czy cotygodniowe usuwanie nieaktywnych użytkowników systemu [2].

3.2. Architektura systemów bezserwerowych

Typowa architektura systemu bezserwerowego opiera się na komponentach wykorzystywanych w innych usługach chmurowych, najczęściej od tego samego dostawcy, takimi jak baza danych, kolejka wydarzeń czy system plików. Do każdego wywołania funkcji napisanej w architekturze bezserwerowej jest przekazywany obiekt zawierający dane przekazane przez jej wyzwalacz, którym może być żądanie HTTP czy inna usługa (np. dodanie nowego użytkownika do bazy danych może wywołać funkcję wysyłki maila powitalnego). Przykład takiej architektury przedstawiony jest na rysunku 1.



Rys. 1. Przykładowa architektura systemu bezserwerowego

W odróżnieniu od tradycyjnej architektury monolitycznej, łatwo jest określić źródło wywołujące daną funkcję, co może pomóc w określeniu procesów biznesowych jakie dana funkcja ma spełniać.

3.3. Przegląd języków programowania

W procesie tworzenia systemów bezserwerowych często wykorzystywane są języki interpretowalne, takie jak Python czy JavaScript, głównie ze względu na ich rosnącą popularność wśród programistów i szybkość wdrażania zmian . W przypadku potrzeby wydania nowej wersji funkcji nie jest wymagana kompilacja kodu źródłowego, co upraszcza proces uruchamiania danej funkcji.

Język Python jest często wykorzystywany do analizy danych i uczenia maszynowego. W przypadku takich aplikacji architektura systemów bezserwerowych może w znaczącym stopniu zmniejszyć koszty związane z zapotrzebowaniem na wysoką moc obliczeniową. Najpopularniejsze środowisko uruchomieniowe JavaScript, Node.js, jest natomiast wydajne w komunikacji asynchronicznej z zewnętrznymi zasobami, a w repozytorium pakietów można znaleźć dużą ilość gotowych integracji z popularnymi usługami chmurowymi, co również czyni go dobrym kandydatem do wykorzyatania w środowiska bezserwerowych.

Innym przykładem języka programowania stosowanego w systemach bezserwerowych jest Golang. Jest to język komplilowalny do postaci kodu maszynowego, dzięki czemu jest wydajny w zadaniach wymagających dużej mocy obliczeniowej. Czas wykonania jest w systemach bezserwerowych istotny, gdyż rozliczane są wykorzystane zasoby w trakcie wykonywania funkcji. Ze względu na stałą popularność w zastosowaniach serwerowych, w takich systemach popularna jest również Java. Jednym z powodów

jest łatwość przepisywania starych, monolitycznych systemów informatycznych na architekturę serverless.

4. System Kubernetes

4.1. Znaczenie i rola kontenerów programowych

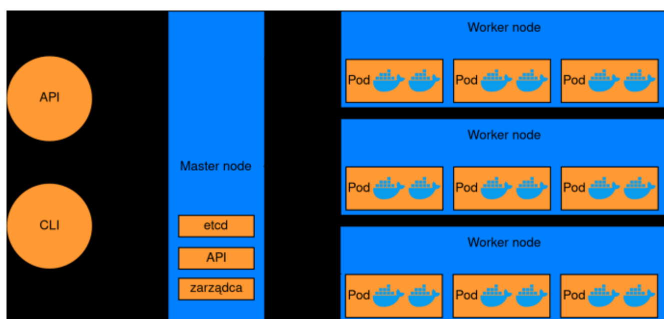
Kontenery programowe oferują izolację usług i aplikacji realizowaną na poziomie systemu operacyjnego. Zapewniają dzięki temu bezpieczeństwo i skalowalność, przy zachowaniu wydajności dorównującej systemowi gospodarza dzięki zastosowaniu lekkiej wirtualizacji.

Najpopularniejszą technologią wykorzystującą kontnerery programowe jest Docker, który cechuje się tworzeniem obrazów aplikacji zawierających ich środowisko uruchomieniowe, w tym system operacyjny, opisywanych za pomocą pliku tekstowego (Dockerfile). Za pomocą tego pliku można tworzyć odtwarzalne i przenośne kompilacje takich obrazów, które mogą być później uruchamiane i połączone ze sobą w wirtualnych sieciach. Dzięki zastosowanej architekturze pomiędzy kontenerami i systemem operacyjnym są współdzielone jedynie jądro systemu i zasoby sprzętowe, co pozwala na osiągnięcie wysokiej wydajności [3].

4.2. System Kubernetes

Najczęściej jako alternatywę do architektury monolitycznej stosuje się obecnie architekturę opartą o oddzielne mikroserwisy, które realizują pojedynczą potrzebę biznesową [4]. Są one wdrażane przez osobne kontenery, które muszą się ze sobą komunikować, co przy większej ilości kontenerów i potrzebie wykorzystania wielu fizycznych serwerów może stanowić kłopot.

Kubernetes powstał w celu rozwiązania tych problemów. Stanowi on rolę nadzorcy nad kontenerami tworząc dla nich izolowane, wirtualne sieci, niezależnie od serwerów na których są uruchamiane. Zarządza ich cyklem życia, monitoruje zużycie zasobów i pozwala na wzajemną komunikację, co czyni go kompleksowym narzędziem do zarządzania nowoczesnymi systemami informatycznymi jak systemy mikroserwisowe czy bezserwerowe [5].

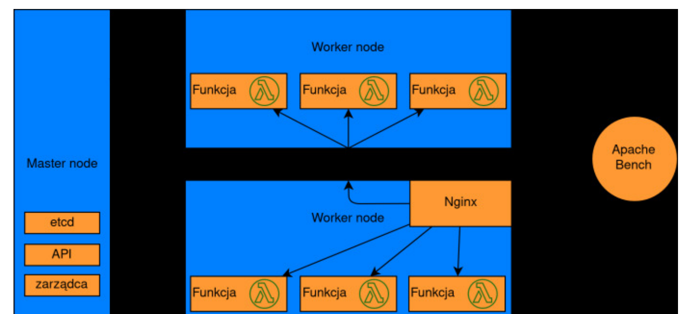


Rys. 2. Architektura systemu Kubernetes

Na rysunku 2 przedstawiona jest typowa architektura systemu Kubernetes. Podstawowym jej elementem jest węzeł główny (ang. master node), który jest zarządcą całego systemu. Zawiera on między innymi moduł etcd, który przechowuje dane konfiguracyjne systemu, interfejs REST API do komunikacji zewnętrznej i wewnętrznej oraz zarządcę przydzielającego kontenery do poszczególnych serwerów. Do niego podłączone są węzły robocze (ang. worker node), na których uruchomione są kontenery ulokowane w jednostki nazwane podami.

5. Testy wydajności systemu bezserwerowego

Na komputerze stacjonarnym z zainstalowanym systemem Ubuntu 19.04 zostało utworzone środowisko testowe. Składa się ono z klastra Kubernetes uruchomionego w oparciu o narzędzie kind. Klaster ten zawiera jeden węzeł główny i dwa węzły robocze. Na nim zostało uruchomione środowisko serverless w postaci frameworka Kubeless, serwer HTTP Nginx udostępniający funkcje na zewnątrz klastra, oraz moduł metrics-server służący do pomiaru zużyciu zasobów na potrzeby autoskalowania. Do pomiaru wydajności został użyty program Apache Benchmark, uruchamiany na systemie gospodarza. Rysunek 3 ilustruje architekturę środowiska testowego.



Rys. 3. Przykładowa architektura systemu bezserwerowego

5.1. Aplikacja testowa

Aplikacja testowa ma za zadanie wyznaczenie *n*-tego wyrazu ciągu Fibonacciego. Ciąg Fibonacciego jest wyrażony następującym wzorem:

$$F_n = \begin{cases} 0 & \text{dla } n=0 \\ 1 & \text{dla } n=1 \\ F_{n-1}+F_{n-2} & \text{dla } n>1 \end{cases} \qquad (1)$$

Na potrzeby tego zadania został wyznaczony algorytm, który w sposób rekurencyjny znajduje podany wyraz ciągu. Schemat algorytmu został przedstawiony poniżej:

Przykład 1. Algorytm wyliczania n-tego wyrazu ciągu Fibonacciego

```
procedure Fibonacci(n)
  if n > 1 then return Fibonacci(n − 1) + Fibonacci(n − 2)
  else if n = 1 then return 1
  return 0
```

Algorytm ten zostanie zaimplementowany we wszystkich wspomnianych poprzednio językach programowania. Aplikacje będą otrzymywać parametr *n* za pomocą zdarzenia („event") jako parametr *n*, a jako wynik będą zwracać rezultat działania algorytmu.

Algorytm ten został wybrany ze względu na złożoność obliczeniową proporcjonalną do wartości parametru *n*, która pozwoli zarówno na zbadanie działania aplikacji przy minimalnej ilości obliczeń (np. dla *n = 0*), jak również przy wysokim obciążeniu procesora (np. dla *n = 40*). Jest on także niezłożony i prosty w implementacji w każdym z badanych w ramach tej pracy języków programowania.

Przykład implementacji tego algorytmu w aplikacji serverless dla języka JavaScript można zobaczyć poniżej.

Przykład 2. Aplikacja testowa w języku JavaScript

```
function fib(n) {
  if (n < 2) {
    return n;
  } else {
    return fib(n - 1) + fib(n - 2);
  }
}

module.exports = {
  handler: (event, context) => {
    var n = parseInt(event.data.n, 10);
    return fib(n);
  },
};
```

W powyższej aplikacji można zwrócić uwagę na dwie części: najpierw jest zdefiniowana funkcja implementująca opisywany wcześniej algorytm. Następnie mamy obsługę zdarzenia („eventu”) wywołanego za pomocą wyzwalacza („triggera”). Ze zdarzenia jest wyznaczany parametr *n*, który jest konwertowany do typu liczbowego metodą *parseInt* (na wypadek gdyby był zdefiniowany jako ciąg znaków). Następnie zostaje on przekazany do funkcji obliczającej *n*-ty wyraz ciągu Fibonacciego, której wynik jest zwracany jako odpowiedź aplikacji na dane zdarzenie.

5.2. Scenariusze testów

W ramach badań zostały wykonane trzy scenariusze testowe skupiające się na różnych obszarach analizy działania środowiska serverless. W ramach każdego z nich uruchomiono funkcje napisane w zdefiniowanych wcześniej językach poddawanych analizie i rejestrowano czas ich odpowiedzi. Funkcje były wywoływane poprzez protokół HTTP za pomocą narzędzia Apache Benchmark. W ramach testów zmieniano ilość replik funkcji oraz liczbę N dla której jest wyliczany wyraz ciągu Fibonacciego. Apache Benchmark wykonywał zapytania 1000 razy i ze zgromadzonych wyników została wyliczona średnia arytmetyczna. Każdy z testów był powtarzany trzykrotnie ze zmienną liczbą równoczesnych zapytań do serwera - odpowiednio z jednym, pięcioma i dwudziestoma pięcioma zapytaniami.

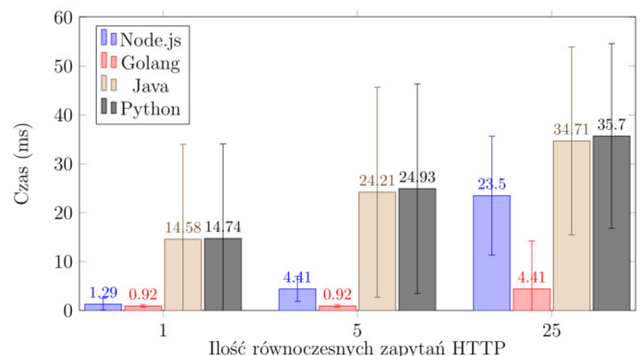
Pierwszy test był przeprowadzony dla jednej repliki każdej z funkcji i parametru N = 1. Zgodnie z zaimplementowanym algorytmem wyliczania n-tego wyrazu ciągu Fibonacciego nie wymagał on rekurencyjnych wywołań funkcji, a w rezultacie funkcje powinny zwrócić 1 bez wykonywania obliczeń. Dzięki temu testowi określono wpływ samego środowiska w którym uruchamiana jest funkcja na całkowity czas jej wykonywania, (ponieważ funkcja nie wykonywała żadnych obliczeń).

Drugi test również wykorzystywał jedną replikę funkcji, natomiast parametr N był równy 25. Pozwoliło to zaobserwować działanie funkcji przy dużym obciążeniu procesora, ponieważ w ramach jednego wywołania niezbędne było rekurencyjne, wielokrotne odwołanie do funkcji, co bezpośrednio wynika z zastosowanego algorytmu. Parametr N dla tego testu został wyznaczony manualnie, weryfikując czy funkcje obciążają procesor w wystarczająco dużym stopniu.

Trzeci test łączył dwa poprzednie testy, a zmieniana była ilość replik do 3 i 10. Rezultatem tego testu są 4 wykresy, czyli powtórzenie pierwszego testu dla trzech i dziesięciu replik, oraz analogiczne powtórzenie drugiego testu dla trzech i dziesięciu replik. Dzięki temu określony został wpływ skalowania ilości replik funkcji na ich wydajność wyrażoną poprzez pełen czas odpowiedzi serwera.

5.3. Wyniki testów

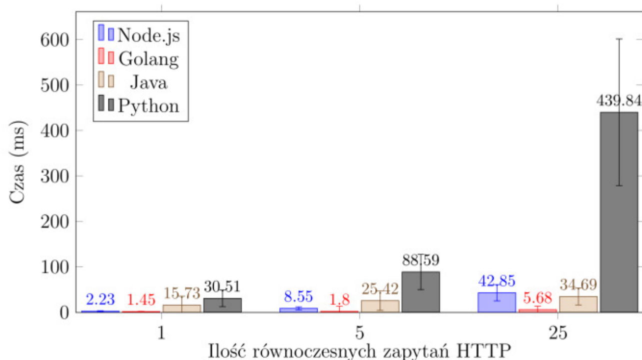
W pierwszym teście dla każdego przypadku testowego wszystkie zapytania HTTP wykonane przez Apache Benchmark się powiodły. Test opiera się na wykonaniu najprostszego możliwego scenariusza dla jednej repliki każdej z funkcji. Oczekiwanym rezultatem były średnie czasy odpowiedzi o wartościach dużo niższych niż w teście drugim. Na rysunku 4 przedstawione są rezultaty pierwszego testu.



Rys. 4. Rezultaty testów dla jednej repliki funkcji i niskiego obciążenia procesora

Otrzymane wyniki wskazują, że ilość równoczesnych zapytań HTTP ma proporcjonalny wpływ na średni czas wykonania zapytania w przypadku Node.js i Golang, ale dla funkcji w języku Java i Python wyniki są istotnie gorsze, co sugeruje negatywny wpływ tych języków na wydajność testowego systemu.

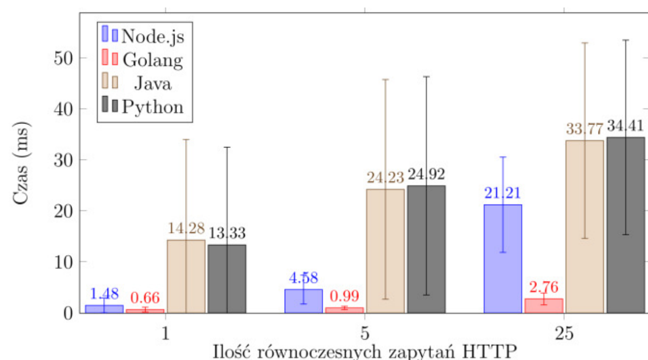
Drugi test został wykonywany w analogiczny sposób co pierwszy. Zmieniła się tylko liczba N przekazywana do funkcji, co powinno mieć wpływ na obciążenie procesora, a tym samym na czasy odpowiedzi funkcji. Ten test również się powiódł i jego wyniki są pokazane na rysunku 5.



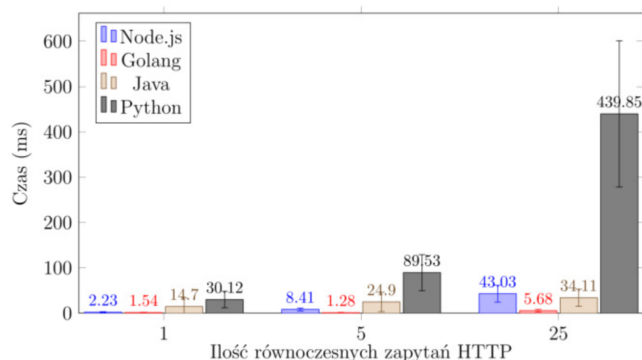
Rys. 5. Rezultaty testów dla jednej repliki funkcji i wysokiego obciążenia procesora

Z otrzymanych rezultatów tego testu wynika, że funkcja napisana w języku Python przy większym obciążeniu procesora jest nawet dwunastokrotnie wolniejsza od funkcji napisanej w języku Java. W przypadku Go natomiast nie zaobserwowano tak dużego wpływu dodatkowych obliczeń spowodowanych większym parametrem N w porównaniu do innych języków. Jest to prawdopodobnie spowodowane specyfiką tego języka, gdyż jest kompilowany do kodu w postaci binarnej.

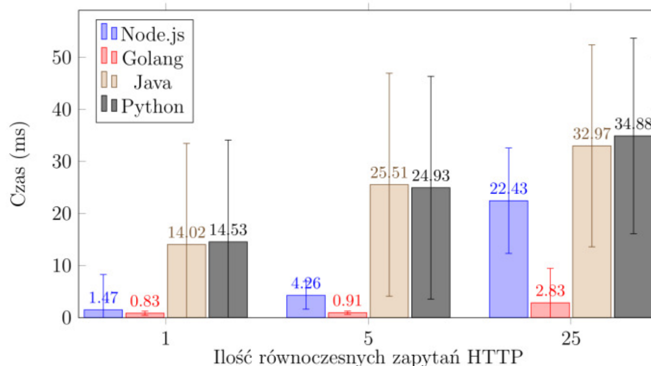
W przypadku trzeciego testu uwaga została poświęcona skalowalności funkcji w przypadku uruchamiania wielu jej instancji. Test wykonał się pomyślnie, a jego rezultaty są przedstawione na rysunkach od 6 do 9.



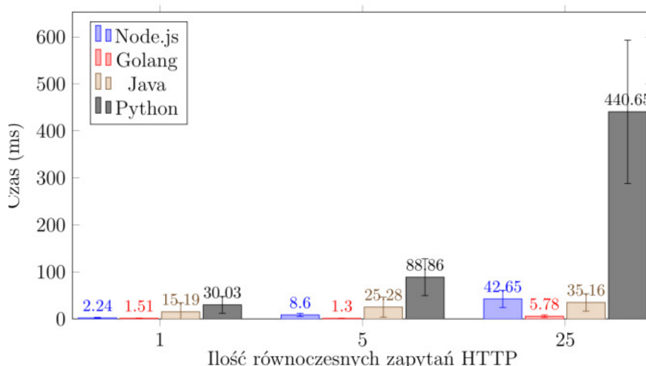
Rys. 6. Rezultaty testów dla trzech replik funkcji i niskiego obciążenia procesora



Rys. 7. Rezultaty testów dla trzech replik funkcji i wysokiego obciążenia procesora



Rys. 8. Rezultaty testów dla dziesięciu replik funkcji i niskiego obciążenia procesora



Rys. 9. Rezultaty testów dla dziesięciu replik funkcji i wysokiego obciążenia procesora

Na podstawie powyższych rezultatów można stwierdzić, że we wszystkich przypadkach zarejestrowano największe wartości czasów potrzebnych na wykonanie funkcji w języku Python. W przypadku Node.js i Java ilość replik miała bardzo mały wpływ na czas odpowiedzi, natomiast dla Go widać krótsze czasy odpowiedzi przy zwiększaniu ilości replik.

6. Wnioski

W celu końcowej weryfikacji opracowane funkcje zostały uruchomione na tym samym komputerze, na którym zaimplementowano środowisko testowe, ale tym razem bez środowiska serverless. Pozwoliło to na porównanie ich wydajności z pominięciem wszystkich czynników powiązanych z systemem chmurowym. Funkcja napisana w języku Golang skompilowana do postaci binarnej wykonywała się najszybciej. Z kolei, o 25% wolniej od niej wykonywała się ta funkcja w języku Java skompilowana do kodu bajtowego i uruchamiana w maszynie wirtualnej JVM. Funkcja w języku JavaScript wykonywana w środowisku Node.js była 3x wolniejsza od Golang. Python okazał się najwolniejszym ze wszystkich funkcji wykonując kod 35x wolniej niż Golang. Te porównawcze wyniki pokrywają się z charakterystyką testowanych języków i odzwierciedlają ich wydajność.

Patrząc na rezultaty testów widać, że najszybciej wykonywały się zawsze funkcje napisane w języku Golang. Zmiana parametrów miała też w przypadku tego języka

proporcjonalny wpływ na rezultaty, a ich odchylenie standardowe było małe. Oznacza to, że jego zachowanie w środowisku serverless jest stabilne i przewidywalne, jednocześnie będąc zdecydowanie szybsze od pozostałych testowanych języków. Język ten był również najszybszy w testach wykonanych bez środowiska serverless.

Node.js w wynikach testów zwykle znajdował się na drugiej pozycji, za wyjątkiem testów z parametrem N=25 i ilością równoczesnych zapytań równą 25, w których spadał na trzecią pozycję. Można z tego wywnioskować, że nie jest najlepszym wyborem do implementacji funkcji pod dużym obciążeniem, które wykonują intensywne obliczenia. W pozostałych testach nie odstawał on w znacznym stopniu od języka Golang. Środowisko serverless może mieć znaczący wpływ na wyniki pod obciążeniem, zważywszy na fakt, że rezultaty testów porównawczych bez środowiska serverless nie wykazywały tak dużych różnic. Wyniki czasów odpowiedzi funkcji zaimplementowanej w tym języku również nie charakteryzowały się dużym odchyleniem standardowym.

W przypadku funkcji napisanej w języku Java zaobserwowano, że pomiędzy wynikami z testów pierwszego i drugiego nie ma dużych różnic, w odróżnieniu od pozostałych języków. Jedną z możliwych przyczyn takiego stanu rzeczy jest długi czas potrzebny na samo uruchomienie funkcji w środowisku serverless, przy jednoczesnym szybkim wykonywaniu obliczeń, których czas w rezultacie jest niezauważalny. Potwierdzają to także rezultaty porównawczych wyników testów bez środowiska serverless. Każde zapytanie przy jednym równoczesnym połączeniu zajmowało średnio minimum 15 milisekund, rosnąc razem z liczbą równoczesnych zapytań, co może sugerować, że Java w środowisku serverless może mieć zastosowanie dla rzadko wykonywanych funkcji które wykonują zaawansowane obliczenia. Uzyskane wyniki testów cechują się też w tym przypadku dużym odchyleniem standardowym.

Funkcja napisana w języku Python była najwolniejsza w niemal wszystkich przypadkach testowych. Przy małej ilości obliczeń czasy są zbliżone do Javy, ale przy dużym obciążeniu procesora Python jest wielokrotnie wolniejszy od reszty badanych języków. Rezultaty otrzymane pod dużym obciążeniem pokrywają się z rezultatami porównawczych wyników testów bez środowiska serverless. Odchylenie standardowe w przypadku aplikacji w języku Python, podobnie jak w języku Java, było bardzo duże, co podkreśla niestabilność zachowania w środowisku serverless. Biorąc zatem pod uwagę czas wykonywania funkcji można dojść do wniosku, że w gronie badanych języków jest on najgorszym wyborem dla środowiska serverless.

Literatura

[1] Everything You Need to Know about Containers, Part III: Orchestration with Kubernetes, https://www.linuxjournal.com/content/everything-you-need-know-about-containers-part-iii-orchestration-kubernetes [14.10.2019]

[2] What Is Serverless Computing, https://www.cloudflare.com/learning/serverless/what-is-serverless/ [10.10.2019]

[3] Jarosław Krochmalski, Docker: projektowanie i wdrażanie aplikacji, Wydawnictwo Helion (2017), 16-18

[4] Łukasz Wróbel, Trzy powody, dla których warto przejść z architektury monolitycznej na mikroserwisową, PC World Komp-Wiadomości, 2018.

[5] Jonathan Baier, Getting Started with Kubernetes, Packt Publishing (2015), 6-7