Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: **Informatyka Stosowana (IST)**
Specjalność: **Projektowanie systemów informatycznych (PSI)**

PRACA DYPLOMOWA MAGISTERSKA

Metody optymalizacji wydajności rozwiązań ekosystemu Java w ramach AWS Lambda

Piotr Puchała

Opiekun pracy
dr inż. Michał Szczepanik

Słowa kluczowe: AWS Lambda, optymalizacja wydajności, Java, Kotlin, Kotlin Multiplatform, GraalVM

WROCŁAW 2025

Streszczenie

Dodaj streszczenie pracy w języku polskim. Staraj się uwzględnić wymienione na stronie tytułowej słowa kluczowe. Uwaga przedstawiony rekomendowany szablon dotyczy pracy dyplomowej pisanej w języku angielskim. W przeciwnym wypadku, student powinien samodzielnie zmienić nazwy „Chapter" na „Rodział" itp stosując odpowiednie pakiety systemu LaTeXoraz ustawienia w pliku *latex-settings.tex*.

Abstract

Streszczenie w języku angielskim.

Spis treści

Wstęp

TODO: Do napisania na końcu

Problem badawczy

Usługa AWS Lambda jest jednym z kluczowych serwisów oferowanych przez chmurę Amazon Web Services (AWS) w architekturze bezserwerowej. Wraz z rosnącą popularnością tego rodzaju architektur, pojawia się potrzeba ciągłej poprawy ich działania. Jednym z pierwotnie dostępnych języków programowania w AWS Lambda jest Java (oraz inne języki oparte o Java Virtual Machine), która jednak ze względu na swoją specyfikę (w tym wpływ na responsywność funkcji) nie jest najczęściej wybieraną opcją implementacji w tym serwisie.

Ważnym elementem pracy z AWS Lambda jest wydajność tworzonych funkcji, która przekłada się bezpośrednio na koszt usługi. Wpływ na wydajność mają takie czynniki jak na przykład tzw. zimne i ciepłe starty, czy czas działania samej funkcji, a niewystarczająco szybkie mogą powodować trudności dla programistów.

Mimo rozwoju wielu różnych technologii, ekosystem Java dalej cieszy się dużą popularnością wśród zespołów programistycznych. Poprawa wydajności funkcji AWS Lambda w tym ekosystemie ułatwi programistom decyzję o wyborze tego rozwiązania oraz pozwoli na lepszą pracę z już znanym językiem. Z tego powodu istnieje potrzeba analizy i zaproponowania metod optymalizacji wydajności dla funkcji AWS Lambda w ekosystemie Java.

Cel pracy

Celem pracy jest zaproponowanie nowych metod poprawy wydajności funkcji AWS Lambda w ekosystemie Java oraz analiza ich wpływu na czas działania funkcji i inne wybrane czynniki, które mogą wpłynąć na jakość pracy programistów.

Pytania badawcze

- PB1: Które metody optymalizacji pozwalają na najlepszą poprawę czasu wykonania funkcji AWS Lambda w ekosystemie Java?
- PB2: W jakim stopniu wybrane metody optymalizacji redukują czas zimnego startu funkcji Java w AWS Lambda?
- PB3: Jakie kompromisy w procesie rozwoju oprogramowania wiążą się implementacją poszczególnych metod optymalizacji wydajności funkcji Java w AWS Lambda?

Zakres pracy

Cel pracy zostanie zrealizowany poprzez następujące działania stanowiące zasadniczy wkład pracy:

1. Systematyczny przegląd literatury
2. Identyfikacja i zaproponowanie metod
3. Analiza wpływu

Struktura pracy

TODO: Do napisania na końcu

Słownik pojęć

Function as a Service (FaaS) -
Serverless -
AWS -
AWS Lambda -

1. Opis działania modelu FaaS

1.1. Model serverless

Jednym z dynamicznie rozwijających się obszarów chmur obliczeniowych są usługi serverless. W 2025 roku rynek usług opartych o architekturę bezserwerową jest wart 17,88 miliarda dolarów amerykańskich, a według prognoz jego wartość wzrośnie do 41,14 miliarda w roku 2029 [22]. Badania wykonane na otwartoźródłowych projektach serverless wykazały, że architektura ta używana jest ze względu na niższe koszty, uproszczenie procesów operacyjnych (jak wdrażanie, skalowanie i monitorowanie) oraz bardzo wysoką skalowalność [7]. Cechy te są osiągalne ze względu na wyjątkowe założenia tego modelu.

Przetwarzanie bezserwerowe możemy zdefiniować jako „formę przetwarzania w chmurze, która umożliwia użytkownikom uruchamianie aplikacji sterowanych zdarzeniami i rozliczanych granularnie bez konieczności zarządzania logiką operacyjną" [29]. W definicji tej znajdują się dwa ważne aspekty działania modelu bezserwerowego:

1. „Przetwarzania w chmurze, która umożliwia użytkownikom uruchamianie aplikacji (...) bez konieczności zarządzania logiką operacyjną."

2. „Aplikacji sterowanych zdarzeniami i rozliczanych granularnie."

Pierwszy punkt odnosi się do zwiększenia zakresu odpowiedzialności dostawcy chmurowego w porównaniu do klasycznych usług (np. Amazon Elastic Compute Cloud). W usługach tych fizyczne serwery są utrzymywane przez dostawcę chmurowego, a użytkownik jedynie wynajmuje jednostki obliczeniowe. Posiada on dalej kontrolę nad konfiguracją wielu aspektów infrastruktury, co pozwala na większą wydajność rozwoju oprogramowania w porównaniu z środowiskami niechmurowymi. Mimo to, dalej wymaga to poświęcenia czasu i środków na skonfigurowanie oraz zabezpieczenie aplikacji. Architektury bezserwerowe mają na celu uproszczenie tych procesów. Podczas tworzenia aplikacji w usługach bezserwerowych zespoły programistyczne nie muszą zarządzać wdrożeniem, a następnie utrzymaniem serwerów (nawet w formie jednostek jak AWS EC2). Rolą twórcy oprogramowania jest dostarczenie kodu aplikacji lub obrazu Docker [26] [25], które zostaną uruchomione w utrzymywanym przez dostawcę chmurowego środowisku. Dzięki temu inżynierowie mogą skupić się w większym stopniu na logice aplikacji. Pozwala to na zmniejszenie liczby obowiązków, a co za tym idzie kosztów zespołu [30].

Drugi punkt skupia się na charakterystycznym modelu płatności oraz sposobie działania architektur bezserwerowych, który umożliwia taki rodzaj rozliczeń. W przypadku klasycznych usług jak AWS EC2 płatność dokonywana jest za czas działania instancji, niezależnie od tego czy jest ona używana [24]. Model ten

2. Przegląd literatury

2.1. Cel przeglądu

2.2. Metodyka przeglądu literatury

2.3. Proces przeglądu

2.3.1. Omówienie pytań badawczych

2.3.2. Przeszukiwane zasoby

2.3.3. Wyszukiwane terminy

2.3.4. Selekcja literatury

2.3.5. Ocena jakości

2.4. Wyniki przeglądu

W ramach przeglądu wybrano X prac badawczych. Na bazie prac rozpatrzono postawione pytania badawcze. Odpowiedzi na nie zostały zawarte w kolejnych podrozdziałach.

2.4.1. Czynniki wpływające na wydajność funkcji

Pierwszym pytaniem badawczym postawionym do przeglądu literatury jest: „Jakie są główne czynniki wpływające na wydajność funkcji AWS Lambda?". Identyfikacja czynników wpływających na wydajność jest kluczowa w kontekście jej optymalizacji. Pozwoli to następnie na zrozumienie na które z czynników ma także wpływ twórca funkcji AWS Lambda, co ułatwi dalszą analizę metod poprawy ich wydajności.

Wielkosć pamięci funkcji

Kelly, Glavin i Barrett [12] zauważają, że wielkość pamięci funkcji oprócz bezpośredniego wpływu na całkowity czas działania, wywiera także wpływ na inne czynniki jak użycie procesora czy wydajność I/O dysku. W ramach badania wykonano pomiary dla funkcji bezserwerowych oferowanych przez wielu dostawców chmurowych, w tym Amazon Web Services, w celu zrozumienia infrastruktury i jej zarządzania, co domyślnie jest ukryte dla użytkownika. Poprzez analizę maszyn wirtualnych, w ramach których uruchamiany jest kod funkcji, możliwe było otrzymanie wartości parametrów, które nie są domyślnie konfigurowalne podczas wdrażania funkcji przez programistę.

Przypis	Autorzy	Rok publikacji
[12]	TODO	TODO
[21]	TODO	TODO
[8]	TODO	TODO
[19]	TODO	TODO
[3]	TODO	TODO
[14]	TODO	TODO
[12]	TODO	TODO
[16]	TODO	TODO
[6]	TODO	TODO
[10]	TODO	TODO
[4]	TODO	TODO
[27]	TODO	TODO
[20]	TODO	TODO
[5]	TODO	TODO
[15]	TODO	TODO
[11]	TODO	TODO
[9]	TODO	TODO
[1]	TODO	TODO
[23]	TODO	TODO
[17]	TODO	TODO
[18]	TODO	TODO
[28]	TODO	TODO
[13]	TODO	TODO
[2]	TODO	TODO
[?]	TODO	TODO
[?]	TODO	TODO
[?]	TODO	TODO
[?]	TODO	TODO
[?]	TODO	TODO
[?]	TODO	TODO

Tabela 2.1: Prace wybrane w ramach przeglądu literatury

Pomiary pokrywały wiele parametrów funkcji, m. in. łączny czas wykonania, czas inicjalizacji, użycie procesora, wydajność I/O dysku oraz liczbę utworzonych maszyn wirtualnych (co wpływa na częstość zimnych startów). W badaniu uwzględniono predefiniowane wielkości pamięci, które mogą być wybrane przez programistę (128MB, 256MB, 512MB, 1024MB oraz 2048MB). Wraz z wzrostem pamięci funkcji, parametry te poprawiały się.

Autorzy podkreślili ważność odpowiedniego doboru wielkości pamięci podczas tworzenia funkcji. Dodatkowo, wykazali, że w przypadku kolejnych wywołań funkcji, platforma AWS ogranicza ponowne użycie wykorzystanych wcześniej maszyn wirtualnych, co powoduje częstsze zimne starty. Wykazano zatem, że zarówno wielkość pamięci, jak i zimne starty znacząco wpływają na ogólną wydajność funkcji AWS Lambda.

Innym aspektem optymalizacji pamięci jest odpowiednie jej wykorzystanie. W nowoczesnych językach programowania, takich jak Java, powszechne jest użycie różnych implementacji odśmiecania pamięci (ang. garbage collection). Quaresma, Fireman i Pereira [21] przeanalizowali wpływ odśmiecania pamięci w środowisku wykonawczym Java i AWS Lambda. Po pierwsze, wykazali oni, że użycie odśmiecania pamięci może negatywnie wpłynąć na wydajność funkcji. Następnie, poprzez użycie techniki „Garbage Collector Control Interceptor", złagodzili negatywny wpływ GC (ang. Garbace Collector), co przyspieszyło czas odpowiedzi o około 10% oraz zmniejszyło koszt działania o 7%.

Wybór odpowiedniej wielkości pamięci funkcji jest bardzo ważnym elementem wdrożenia także ze względu na bezpośredni jej wpływ na koszty. Elgamal i inni autorzy [8] zaproponowali model optymalizacji kosztów funkcji, w którym jednym z czynników była pamięć AWS Lambda. Zwrócili uwagę na to, że nawet niewielkie wartości (z 128 MB do 256 MB) było wstanie poprawić szybkość wykonania algorytmu o 10%, przy jednoczesnym obniżeniu kosztów o 6%.

Pawlik, Figiela i Malawski [19] zwrócili uwagę na wpływ pamięci FaaS dla konkretnych zastosowań naukowych. Dokonali równoczesnej ewaluacji 5120 zadań z użyciem serwisów różnych dostawców (m. in. AWS). Duża liczba zadań wynikała z chęci przetestowania przekroczenia limitów współbieżności oferowanych przez dostawców. Wykazali, że wraz z wzrostem pamięci rośnie wydajność funkcji, mierzona za pomocą wskaźnika GFlops (ang. Giga Floating Point Operations Per Second). Interesującym szczegółem jest niewielka różnica wydajności pomiędzy 2048 MB i 3008 MB (około 0.8%). Autorzy wskazują, że może to być spowodowane taką samą konfiguracją limitów procesora, gdyż wielkość 2048 do niedawna była największą oferowaną przez AWS.

Architektura procesora

Amazon Web Services oferuje możliwość wyboru architektury procesora spośród X86_64 oraz ARM64. Architektura ARM64 jest wspierana poprzez procesory AWS Graviton2 rozwijane przez AWS.

Chen, Hung, Cordingly oraz Lloyd [3] zwrócili uwagę na znaczące różnice wydajności między obiema dostępnymi architekturami procesora. Autorzy przeprowadzili testy wydajnościowe 18 funkcji AWS Lambda i działających na obu rodzajach procesorów (Intel Xeon dla X86_64 oraz AWS Graviton2 dla ARM64).

Wykazali oni podobne zużycie procesora dla obu architektur. Wiele funkcji wykorzystywało wyłącznie jeden z dostępnych rdzeni procesora, co wskazuje na możliwość optymalizacji w kierunku zrównoleglania obliczeń. Mimo podobnego zużycia, sam czas działania funkcji był zróżnicowany. 7 z 18 funkcji działało szybciej na ARM64 (4 były ponad 10% szybsze), podczas gdy 6 działało znacznie wolniej (o ponad 10%). Funkcje ARM64 były bardziej opłacalne dla większości przypadków. 15 z 18 funkcji miało niższe szacunkowe koszty działania na ARM64 w porównaniu do X86 (jednak znaczący wpływ na to miała zniżka oferowana przez dostawcę chmurowego).

Lambion i inni autorzy [14] przeprowadzili analizę użycia algorytmów przetwarzania języka naturalnego z użyciem obu architektur procesora. W ramach badania przygotowali oni składający się z kilku etapów pipeline, który używał wspomniancyh algorytmów. Funkcje zostały wdrożone z użyciem obu architektur oraz w różnych regionach AWS.

Wydajność obu architektur różniła się w zależności od etapu pipeline'u. Dla regionu us-

east-2 ARM64 był szybszy w przypadku funkcji przetwarzania wstępnego (o 7,3%) i zapytań (o 8,9%), podczas gdy X86_64 był znacznie szybszy (o 23,6%) w przypadku funkcji treningowej. W ujęciu globalnym funkcje ARM64 były średnio o 1.7% szybsze niż funkcje X86_64.

Działanie funkcji różniło się także w zależności od regionu. Funkcje w architekturze X86_64 działały najszybciej w regionie eu-central-1, a najwolniej w us-west-2. W przypadku ARM64, region us-west-2 był najszybszy, a us-east-2 najwolniejszy. Funkcje wykazały także tendencję do bycia szybszymi poza typowymi godzinami pracy (np. funkcje w godzinach 6:00-8:00 działały o 6% szybciej niż funkcje w godzinach 10:00-12:00).

Zimne starty

Specyficznym zjawiskiem dla usług FaaS są tzw. zimne starty. Polegają one na dłuższym czasie inicjalizacji funkcji, co wynika z konieczności przygotowania infrastruktury w postaci maszyny wirtualnej i środowiska wykonawczego. Jest to ważny czynnik wpływający na serwisy jak AWS Lambda, w szczególności w przypadku aplikacji skierowanych do użytkowników.

Zimne starty występują często na platformie AWS, co stwierdzili Kelly, Glavin i Barrett [12] podczas analizy infrastruktury obsługującej AWS Lambda. Podczas badań z użyciem powtarzających się co godzinę wywołań doświadczyli oni bardzo częstych zimnych startów funkcji (aż około 89% uruchomień). Podkreśla to wielkość problemu zimnych startów w przypadku rzadko używanych funkcji. Zimne start na platformie AWS były jednak znacząco krótsze niż w usługach innych dostawców. Dla funkcji o pamięci 128 MB było to maksymalnie około 350 milisekund. W przypadku funkcji o większym rozmiarze pamięci opóźnienia były zbliżone. Na bazie porównania różnych dostawców chmurowych autorzy podkreślili, że infrastruktura AWS Lambda ma tendencję do utrzymywania gotowych maszyn wirtualnych krócej niż inni dostawcy, co prowadzi do częstszych zimnych startów, jednak z mniejszymi opóźnieniami.

Manner i inni autorzy [16] skupili się na występowaniu zimnych startów i wpływu różnych czynników na nie. Wykazali oni różnice w czasie wykonania funkcji w przypadku zimnych i ciepłych startów. Udowodnili, że bezpośredni wpływ na nie ma wybrany język programowania, wielkość pamięci i rodzaj wdrożenia artefaktu (ZIP lub Docker). Pokazuje to skomplikowanie pojęcia jakim są zimne starty.

Na istotę zimnych startów wskazali także Ebrahimi, Ghoabaei-Arani i Saboohi [6], poprzez dokonanie przeglądu literatury w zakresie metod ich optymalizacji. Jedną z najczęściej omawianych platform w literaturze jest właśnie AWS Lambda. Zimne starty są mierzone poprzez m. in. opóźnienie, liczbę wystąpień, użycie pamięci i całkowity czas odpowiedzi funkcji. Użycie odpowiednich metryk pozwala następnie na ocenę jakości konkretnych metod optymalizacji zimnych startów.

Język programowania

Znaczące różnice w wydajności pomiędzy językami w AWS Lambda podkreślili Jackson i Clynch [10]. Przeanalizowali oni ich wpływ na zarówno ciepłe i zimne starty oraz koszty funkcji. Testowane funkcje zostały napisane w językach .NET 2, JavaScript (dla NodeJS), Java 8, Go oraz Python 3.

W przypadku ciepłych startów najszybszymi testowanymi językami były Python (średnio 6,13 ms) oraz .NET 2 (średnio 6,32 ms). Java uplasowała się na trzecim miejscu (średnio 11,33 ms), a najgorszy okazał się Go (średnio 19,21 ms). Wzorce wydajności zmieniają się jednak diametralnie w przypadku zimnych startów: najszybszy dalej jest Python (średnio 2,94 ms). Java wykazała znacząco większe opóźnienia (391,91 ms) w porównaniu z ciepłym startem (ponad 3-krotny wzrost). Interesującym faktem jest około 40-krotny wzrost opóźnienia funkcji .NET 2 dla zimnych startów (średnio 2.5 sekundy). Czas wykonania wpłynął bezpośrednio na koszty, które różniły się nawet 13-krotnie. Badania nie zostały niestety wykonane na różnych wielkościach pamięci, która jest jednym z głównych składowych kosztu działania.

Cordingly i inni autorzy [4] zwracają uwagę na potrzebę odpowiedniego doboru języka programowania do konkretnej funkcji AWS Lambda. Poprzez przygotowanie procesu Transform-Load-Query, składającego się z kilku komponentów, byli w stanie dokładnie przeanalizować wpływ języków na poszczególne jego etapy. Badane języki mogą być podzielone na dwie grupy: kompilowane (Java, Go) oraz interpretowane (Python, Javascript). Wykazano, że żaden z języków nie był najlepszego dla każdego etapu procesu. Poprzez przygotowanie hybrydowego pipeline'u możliwe było osiągnięcie znaczącej poprawy opóźnień (17%-129% szybciej). Wybrany język ma także wpływ na czas inicjalizacji funkcji. W przypadku Javy wymagana jest inicjalizacja JVM, jednak nie powodowało to znacząco dłuższych zimnych startów w porównaniu z Python i Node.js. Go prezentowało jednak około 20% dłuższe zimne starty.

Podobne porównanie zostało wykonane przez Shrestha [27], który przeanalizował natywnie wspierane języki w AWS Lambda. Języki interpretowane (Javascript, Python, Ruby) cechowały się szybszymi zimnymi startami niż kompilowane (Java, C#, Go), choć w przypadku ciepłych startów Java oferowała wysoką wydajność. Shrestha zwraca jednak uwagę, że istnieje wiele innych czynników wpływających na wydajność. Podkreśla on, że w wyborze języka programowania ważnym elementem powinny być także osobiste preferencje programisty co do niego, a nie tylko wydajność.

2.4.2. Metody optymalizacji funkcji w ekosystemie Java

TODO: wstep dla podrozdziału

Redukcja rozmiaru artefaktu

Podczas rozwoju aplikacji w ekosystemie Java ważnym etapem jest odpowiednie utworzenie artefaktu, który zawiera wszystkie zależności potrzebne do uruchomienia. Wynikiem tego są pliki JAR, które podczas użycia w AWS Lambda muszą zostać pobrane do używanej maszyny wirtualnej, co wpływa na czas inicjalizacji.

Puripunpinyo i Samadzadeh [20] zwrócili uwagę, że klasyczne narzędzia budowy artefaków tworzą często artefakty o dużym rozmiarze, gdzie część kodu jest niepotrzebna. Zademonstrowali, że optymalizacja tych artefaktów pozwoli na poprawę wydajności, w tym zmniejszenie efektu zimnych startów. Dodatkowo, problemem może być przekroczenie limitu wielkości artefaktu dla AWS Lambda, który wynosi 50 MB.

Autorzy zaproponowali kilka technik optymalizacji, jak odpowiedni wybór wersji danej zależności, czy użycie zewnętrznego oprogramowania (jak ProGuard), co pozwoliło na redukcję rozmiaru. Wśród zaproponowanych metod bardzą ważną dla kontekstu FaaS jest odpowiednie

grupowanie artekfatów i funkcji. W pracy przedyskutowano głównie dwa podejścia: grupowanie ze względu na serwis oraz na rozmiar.

Grupowanie ze względu na serwis wynikało z chęci zmniejszenia rozmiaru artefaktu, który jak wykazali autorzy wpływa na czas zimnych startów. Odpowiedni podział artefaktów może pozwolić na otrzymanie rozmiaru, który pozwoli na szybsze wykonanie funkcji. Może to jednak powodować konieczność wysłania żądania do innej funkcji, w której znajduje się potrzebny kod. Grupowanie ze względu na serwis pozwala zastąpić zapytania między funkcjami zapytaniami natywnymi w obrębie jednej funkcji, które z natury są szybsze. Strategia ta jednak prowadzi do większych artefaktów.

Metodą na zmiejszenie rozmiaru może być także użycie mniejszej liczby bibliotek. Problemem dla funkcji AWS Lambda może być za duża liczba zależności, co podkreślili Nupponen i Taibi [18]. Z przedstawionych przez nich problemów, które dotyczą funkcji AWS Lambda wynika, że odpowiednia budowa artefaktu może być kluczowa dla wydajności funkcji. Zbyt duży ich podział prowadzi do zapytań między funkcjami, podczas gdy zapytania te mogą być wolne oraz trudne do debuggowania. Jednocześnie, zbyt duży ilość kodu współdzielonego między funkcjami może prowadzić do zwiększenia rozmiaru funkcji i opóźnień, zatem zalecane jest stosowanie się do zasady pojedynczej odpowiedzialności funkcji.

Problem optymalizacji artefaktów dla Javy w kontekście AWS Lambda podjęli również Chatley i Allerton w ramach prac nad frameworkiem Nimbus [2]. Podkreślili, że klasyczne narzędzia (jak Maven Shade) łączą wszystkie zależności w jeden artefakt, nawet jeśli funkcja wykorzystuje niewielką ich część. Aby temu zaradzić ich framework Nimbus wprowadza mechanizm budowy artefaktów, które zawierają wyłącznie te klasy, które są potrzebne do uruchomienia funkcji. Autorzy wykazali, że takie podejście pozwala zmniejszyć rozmiar plików JAR, co przyczynia się do redukcji czasu zimnych startów. Nimbus potrafi także wykryć i wdrożyć tylko zmienione funkcje, co dodatkowo ogranicza liczbę niepotrzebnych zimnych startów.

Wybór rodzaju wdrożenia

AWS oferuje dwie strategie wdrożenia funkcji Lambda, opierające się o pliki ZIP lub obrazy Docker. Świadomy ich wybór ma znaczący wpływ na wydajność, szczególnie na czas zimnych startów. Dantas, Khazaei i Litoiu [5] przeprowadzili szczegółowe badania porównujące obie opcje dla wybranych języków programowania.

Wyniki różniły się w zależności od wybranego języka. Dla Pythona oraz Javascriptu obie opcje działały podobnie, lub z korzyścią dla obrazów Dockerowych. Co ciekawe, dla języka Java tendencja była odwrotna - wdrożenie oparte o pliki ZIP zapewniało krótszy czas zimnego startu w porównaniu do wdrożenia kontenerowego. Wyniki te były spójne niezależnie od testowanego rozmiaru aplikacji czy ilości przydzielonej pamięci, a przewaga wdrożeń ZIP była szczególnie widoczna dla większych aplikacji.

Pingowanie

Po wykonaniu zapytania do funkcji AWS Lambda maszyny wirtualne, które zostały wykorzystane, pozostają aktywne w oczekiwaniu na kolejne uruchomienia. Trwa to przeważnie kilka minut [12], gdy uruchomiony już kod dalej jest gotowy do działania. Taktyką z tym

związaną jest regularne uruchamianie funkcji zaproponowane przez [15] Lloyd i innych autorów. Funkcje AWS Lambda były uruchamiane przez specjalne instancje EC2 lub usługę CloudWatch, co pozwoliło utrzymać funkcje aktywne nawet do 24 godzin. Podejście to pozwoliło na redukcję zimnych startów i przyspieszenie funkcji około czterokrotnie. W porównaniu z klasyczną infrastrukturą (opartą o np. kontenery Docker) funkcje Lambda były około 10% wolniejsze, jednak użycie funkcji bezserwerowych pozwoliło na około 18-krotne zmniejszenie kosztów.

Użycie Javascript

Kaplunovich [11] proponuje interesujące podejście do migracji monolitycznych aplikacji napisanych w technologii Java do AWS Lambda. Migracja ta prowadzona jest z użyciem narzędzia ToLambda, które jest w stanie transformować kod Java na JavaScript, który jest następnie uruchamiany w AWS Lambda jako funkcje NodeJS. Autor motywuje wybór JavaScriptu jako docelowego języka ze względu na trendy wskazujące lepszą wydajność i popularność tego języka. Dodatkowo, podkreśla takie zalety jak mniejsza szczegółowość kodu w porównaniu do Javy, brak konieczności kompilacji, co przyspiesza wdrożenia oraz dobra integracja z usługami AWS.

Z użyciem zaproponowanego narzędzia ToLambda użytkownik jest w stanie przekształcić wybraną funkcję publiczną Javy w niezależne funkcje Lambda. Dla każdej funkcji transformowane są także wszystkie wymagane do uruchomienia zależności (jak klasy), wraz z zachowaniem ich właściowości (np. poziom dostępu do pól). Autor zwraca jednak uwagę na skomplikowanie Javy i jej specyficznych konstrukcji (jak polimorfizm czy hierarchia konstruktorów), które stanowią wyzwanie w trakcie transformacji.

Choć głównym celem pracy jest automatyzacja migracji monolitycznych aplikacji do architektury bezserwerowej, autor prezentuje interesujące podejście do użycia Java w AWS Lambda. W celu uniknięcia wyzwań jak zimne starty czy czasochłonność budowy aplikacji, pośrednie użycie innego języka może być skuteczną metodą optymalizacji.

Podobny trend został także zaprezentowany w pracy Dos Santosa i innych autorów [9]. Jako rozwiązanie problemu zimnych startów funkcji AWS Lambda o niskiej częstotliwości użycia, autorzy zaproponowali użycie NodeJS jako alternatywy dla Javy. W ramach badania wykazano, że Node.js oferuje znaczącą redukcję czasu zimnego startu w porównaniu do Javy (nawet o 82%). Zaobserwowano, że funkcje z pamięcią 512 MB stawały się nieaktywne już po 6-7 minutach, co czyni problem zimnego startu szczególnie istotnym dla rzadziej używanych aplikacji.

Autorzy zasugerowali zastosowanie NodeJS jako łatwiejszą w implementacji alternatywę dla bardziej złożonych technik optymalizacji Javy w środowisku AWS Lambda. Choć Java oferuje lepszą wydajność przy ciepłych startach, w przypadku systemów, w których wywołania funkcji następują w odstępach kilku minut, NodeJS może skutecznie ograniczyć opóźnienia związane ze startem funkcji.

Mimo interesującego podejścia, użycie JavaScript w miejsce Javy może być jednak nieoptymalne dla zespołów programistycznych z umiejętnościami w ekosystemie Java. Wymaga to kompletnej zmiany używanych narzędzi, co może być kosztowne. Dlatego metody te zostały zawarte w wynikach przeglądu jako prezentacja alternatywy, jednak nie jako pełne rozwiązanie problemu wydajności.

JIT

[1]

GraalVM

[17] [23]

2.4.3. Cechy rozwoju aplikacji w AWS Lambda

TODO: wstep dla podrozdziału
[18] - mocno podzielone, małe funkcje, różne technologie itp (takie mikro-mikroserwisy)
[28] - w sumie to co wyżej ale lepiej opisane, opiera się na ankietach
[13] - https://gemini.google.com/gem/180d518dd570/9ba1bf85bf544314
[2] - gemini.google.com/gem/180d518dd570/d2c2aaa642e9fcd1

2.4.4. Podsumowanie wyników przeglądu

TODO: napisac wnioski

Bibliografia

[1] J. Carreira, S. Kohli, R. Bruno, and P. Fonseca. From warm to hot starts: leveraging runtimes for the serverless era. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 58–64, New York, NY, USA, 2021. Association for Computing Machinery.

[2] R. Chatley and T. Allerton. Nimbus: improving the developer experience for serverless applications. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE '20, page 85–88, New York, NY, USA, 2020. Association for Computing Machinery.

[3] X. Chen, L.-H. Hung, R. Cordingly, and W. Lloyd. X86 vs. arm64: An investigation of factors influencing serverless performance. In *Proceedings of the 9th International Workshop on Serverless Computing*, WoSC '23, page 7–12, New York, NY, USA, 2023. Association for Computing Machinery.

[4] R. Cordingly, H. Yu, V. Hoang, D. Perez, D. Foster, Z. Sadeghi, R. Hatchett, and W. J. Lloyd. Implications of programming language selection for serverless data processing pipelines. page 704 – 711, 2020. Cited by: 21.

[5] J. Dantas, H. Khazaei, and M. Litoiu. Application deployment strategies for reducing the cold start delay of aws lambda. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 1–10, 2022.

[6] A. Ebrahimi, M. Ghobaei-Arani, and H. Saboohi. Cold start latency mitigation mechanisms in serverless computing: Taxonomy, review, and future directions. *Journal of Systems Architecture*, 151:103115, 2024.

[7] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup. Serverless applications: Why, when, and how? *IEEE Software*, 38(1):32–39, 2021.

[8] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha. Costless: Optimizing cost of serverless computing through function fusion and placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 300–312, 2018.

[9] P. O. Ferreira Dos Santos, H. Jorge De Moura Costa, V. R. Q. Leithardt, and P. Jorge Silveira Ferreira. An alternative to faas cold start latency of low request frequency applications. 2023. Cited by: 0.

[10] D. Jackson and G. Clynch. An investigation of the impact of language runtime on the performance and cost of serverless functions. In *2018 IEEE/ACM International*

Conference on Utility and Cloud Computing Companion (UCC Companion), pages 154–160, 2018.

[11] A. Kaplunovich. Tolambda–automatic path to serverless architectures. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 1–8, 2019.

[12] D. Kelly, F. Glavin, and E. Barrett. Serverless computing: Behind the scenes of major platforms. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 304–312, 2020.

[13] K. Kritikos and P. Skrzypek. A review of serverless frameworks. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 161–168, 2018.

[14] D. Lambion, R. Schmitz, R. Cordingly, N. Heydari, and W. Lloyd. Characterizing x86 and arm serverless performance variation: A natural language processing case study. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, ICPE '22, page 69–75, New York, NY, USA, 2022. Association for Computing Machinery.

[15] W. Lloyd, M. Vu, B. Zhang, O. David, and G. Leavesley. Improving application migration to serverless computing platforms: Latency mitigation with keep-alive workloads. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 195–200, 2018.

[16] J. Manner, M. Endreß, T. Heckel, and G. Wirtz. Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 181–188, 2018.

[17] J. M. Menéndez and M. Bartlett. Performance best practices using java and aws lambda, 2023.

[18] J. Nupponen and D. Taibi. Serverless: What it is, what to do and what not to do. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 49–50, 2020.

[19] M. Pawlik, K. Figiela, and M. Malawski. Performance considerations on execution of large scale workflow applications on cloud functions, 2019.

[20] H. Puripunpinyo and M. Samadzadeh. Effect of optimizing java deployment artifacts on aws lambda. In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 438–443, 2017.

[21] D. Quaresma, D. Fireman, and T. E. Pereira. Controlling garbage collection and request admission to improve performance of faas applications. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 175–182, 2020.

[22] Research and Markets. Serverless architecture market report 2025, (2025). Dostęp 20.03.2025 z https://www.researchandmarkets.com/reports/5953253/serverless-architecture-market-report.

[23] R. Ritzal. Optimizing java for serverless applications. Master's thesis, University of Applied Sciences FH Campus Wien, 2020.

[24] A. W. Services. Amazon ec2 documentation, 2025. Dostęp 21.03.2025 z https://docs.aws.amazon.com/ec2/.

[25] A. W. Services. Amazon ecs developer guide, 2025. Dostęp 21.03.2025 z https://docs.aws.amazon.com/AmazonECS/latest/developerguide.

[26] A. W. Services. Aws lambda documentation, 2025. Dostęp 21.03.2025 z https://docs.aws.amazon.com/lambda/latest/dg/.

[27] S. Shrestha. Comparing programming languages used in aws lambda for serverless architecture. Bachelor's thesis, Theseus, 2019.

[28] D. Taibi, B. Kehoe, and D. Poccia. Serverless: From bad practices to good solutions. In *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 85–92, 2022.

[29] E. van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann. A spec rg cloud group's vision on the performance challenges of faas cloud architectures. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, page 21–24, New York, NY, USA, 2018. Association for Computing Machinery.

[30] J. Wen, Z. Chen, X. Jin, and X. Liu. Rise of the planet of serverless computing: A systematic review. *ACM Trans. Softw. Eng. Methodol.*, 32(5), 2023.