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

Streszczenie w języku polskim.

Abstract

Streszczenie w języku angielskim.

Spis treści

Wstęp **1**

Słownik pojęć i akronimów **5**

1 Podstawy działania AWS Lambda **7**
1.1 Model bezserwerowy . . . 7
1.2 Funkcja jako usługa . . . 8
1.3 AWS Lambda . . . 10

2 Przegląd literatury **15**
2.1 Cel przeglądu . . . 15
2.2 Metodyka przeglądu literatury . . . 15
2.3 Proces przeglądu . . . 16
2.3.1 Omówienie pytań badawczych . . . 16
2.3.2 Przeszukiwane zasoby . . . 17
2.3.3 Wyszukiwane terminy . . . 17
2.3.4 Selekcja literatury . . . 20
2.3.5 Ocena jakości . . . 21
2.4 Wyniki przeglądu . . . 21
2.4.1 Czynniki wpływające na wydajność funkcji . . . 22
2.4.2 Metody optymalizacji funkcji w ekosystemie Java . . . 26
2.4.3 Cechy rozwoju aplikacji w AWS Lambda . . . 30
2.4.4 Podsumowanie wyników przeglądu . . . 33

3 Wybrane metody optymalizacji **37**
3.1 SnapStart . . . 37
3.2 GraalVM . . . 39
3.3 Kotlin . . . 41
3.4 Kotlin/JS . . . 44
3.5 Kotlin/Native . . . 46

4 Opis badania **49**
4.1 Cel i metodologia badań . . . 49
4.2 Implementacja funkcji . . . 52
4.3 Wdrożenie funkcji . . . 55
4.4 Środowisko badawcze . . . 56

5 Wyniki **59**
5.1 Ciepły start 59
5.2 Zimny start 61
5.3 Współczynnik wydajności funkcji 64
5.4 Koszt funkcji 65
5.5 Wpływ na rozwój oprogramowania 66

6 Wnioski i dyskusja **69**
6.1 Odpowiedzi na pytania badawcze 69
6.2 Zagrożenia trafności wyników 72

Zakończenie **75**

Wstęp

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 jej maszyne wirtualną. Według raportu Datadog „The State of Serverless" około 10% wywołań funkcji AWS Lambda jest obsługiwane przez język Java [28]. Języki oparte o maszynę wirtualną Java są jednak wykorzystywane o wiele częściej. Według ankiety StackOverflow w 2024 roku, około 30% profesjonlanych programistów zadeklarowało, że korzysta z tego Javy. Dodatkowo prawie 9.9% z nich oznajmiło korzystanie z Kotlina, 3.8% z Groovy, a 2.9% z języka Scala.

Wydajność jest jednym z najbardziej istotnych aspektów usług bezserwerowych. Wynika to z specyficznego modelu tych usług, gdzie opłacany jest wyłącznie czas realnego korzystania z infrastruktury (tzw. „płać za użycie", ang. „pay-per-use") [31]. Ze względu na jego wyjątkowe cechy i bezpośredni wpływ na koszt działania, opis tego modelu zostanie także zawarty w pracy. Istotę omawianego aspektu podkreśla także fakt, że znaczącym czynnikiem w wyborze usług bezserwerowych jest właśnie redukcja kosztów [31], która jest z bezpośrednio powiązana z efektywnością. Wspomniany raport Datadog „The State of Serverless" [28] porusza problemy wydajnościowe w konkteście ekosystemu Java dwukrotnie. Po pierwsze podkreśla znacznie dłuższe zimne starty w porównaniu z innymi językami programowania, a także alokację większych rozmiarów pamięci dla funkcji AWS Lambda opartych o Javę. Z tego powodu redukcja wpływu tych problemów może znacznie ułatwić korzystanie z AWS Lambda w ekosystemie Java, a co za tym idzie większą popularyzację tego języka programowania jako narzędzia dla usług bezserwerowych.

Jednymi z metod optymalizacji wydajności Javy w ramach AWS Lambda mogą być usługa SnapStart oraz kompilacja GraalVM. W ramach przeglądu literatury odnaleziono jedynie pojedyncze prace opisujące ich użycie w kontekście AWS Lambda [54] [63]. Dodatkowo poruszają one efektywność tych metod jedynie poprzez proste metryki jak czas wykonania ciepłych i zimnych startów, a także nie badają ich skuteczności pod względem rozmiaru pamięci funkcji. Co więcej, wśród znalezionych prac proponujących metody optymalizacji wydajności dla AWS Lambda [58] [22] [27] [34] [19] [54] [63], jedynie pojedyncza praca poruszała temat wpływu na jakość rozwoju oprogramowania [22]. Jest to jednak istotny element, który powinien być wzięty pod uwagę podzas wyboru odpowiedniego sposobu poprawy efektywności funkcji.

Nie odnaleziono także prac badających wydajność użycia w AWS Lambda, alternatywnych języków programowania opartych o maszynę wirtualną Javy. Istotną motywacją do badań w tym zakresie jest popularność języka Kotlin, którego użycie w ankiecie StackOverflow [65] zadeklarowało 9.9% profesjonalnych programistów. Sam Kotlin zapewnia szeroki wybór narzędzi oraz bibliotek, a także w ramach języka rozwijany jest projekt Kotlin Multiplatform, który może zostać potencjalnie wykorzystany w ramach AWS Lambda. Zatem język ten stanowi szeroki obszar do badań w kontekście wydajności w usłudze AWS Lambda.

Problem badawczy

Usługa AWS Lambda oferuje możliwość implementacji funkcji w językach z ekosystemu Java. Zespoły programistyczne korzystające z tych rozwiązań mogą jednak doświadczać problemów wydajnościowych, które mogą być kluczowe w modelu bezserwerowym. Niska wydajność funkcji może mieć wpływ na koszt infrastruktury oraz zadowolenie użytkowników systemu, którzy oczekują niskich czasów odpowiedzi na zapytania. Istnieje potrzeba analizy i zaproponowania metod, które pozwolą zespołom programistycznym na poprawę wydajności tworzonych funkcji.

Przypadki użycia systemów opartych o funkcje AWS Lambda są bardzo róznorodne. Z tego powodu twórcy takich systemów potrzebują odpowiednich informacji na temat metod, które potencjalnie zastosują. Rodzi to zapotrzebowanie na badanie metod pod kątem różnych typów funkcji, a także ich potencjalnego wpływu na proces rozwoju oprogramowania. Zastosowanie nieodpowiedniej metody może prowadzić do niespodziewanych ograniczeń w rozwoju i utrzymaniu funkcji AWS Lambda.

Cel pracy

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

Pytania badawcze

- PB1: Które metody optymalizacji pozwalają na najlepszą poprawę ogólnej wydajności 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. Wykonanie systematycznego przeglądu literatury w konkteście wydajności i tworzenia funkcji AWS Lambda. Przegląd ten pozwoli na zrozumienie aktualnego stanu wiedzy oraz rozpoznanie obszarów, który nie zostały jeszcze wystarczająco zbadane.
2. Identyfikacja i zaproponowanie metod optymalizacji wydajności funkcji AWS Lambda w ekosystemie Java. Propozycja nowych metod będzie znacząco opierać się na wykonanym

przeglądzie literatury. Jest to kluczowy element pracy, który następnie pozwoli na analizę tych metod w celu odpowiedzenia na postawione pytania badawcze.

3. Przygotowanie i wykonanie badania poprzez pomiar wydajności funkcji korzystających z wybranych metod. Etap ten będzie opierał się o nowo zaproponowane metody oraz te wybrane z przeglądu literatury. Zbadanie obu rodzajów sposobów pozwoli na bardziej rzetelną ocenę nowo wybranych metod optymalizacji. Przygotowane badanie zostanie dokładnie opisane, co pozwoli na jego odtworzenie.

4. Analiza wyników badania i wyciągnięcie wniosków, co pozwoli odpowiedzieć na sformułowane w pracy pytania badawcze.

Struktura pracy

Praca składa się ze wstępu, słownika pojęć i akronimów, sześciu rozdziałów, zakończenia oraz bibliografii. W ramach wstępu zdefiniowano problem badawczy, cel pracy, pytania badawcze oraz zakres pracy. W pierwszym rozdziale opisano teoretyczne podstawy działania AWS Lambda, a także znaczenie modeli bezserwerowych i funkcji jako usług (na których opiera się AWS Lambda). W rozdziale drugim dokonano systematycznego przeglądu literatury, na bazie którego opisano aktualny stan wiedzy. Odpowiedziano na pytania badawcze do przeglądu oraz zlokalizowano luki w literaturze, co zostało wykorzystane w następnym rozdziale. W trzecim rozdziale zawarto opis wybranych metod optymalizacji, w tym nowo zaproponowanych. W rozdziale czwartym dokładnie opisano przygotowanie badania oraz jego przebieg, co pozwoli na jego odtworzenie. W piątym rozdziale przedstawiono wyniki wykonanego badania oraz dokonano obiektywnej analizy poszczególnych metryk. W rozdziale szóstym dokonano szerszej analizy wyników badania, wyciągnięto wnioski oraz przeprowadzono ich dyskusję w celu odpowiedzi na sformułowane pytania badawcze. Zakończenie zostało wykonane jako podsumowanie całej pracy oraz wyznaczenie dalszych perspektyw rozwoju. Pracę zakończono bibliografią, w tym wykazem cytowanej literatury oraz spisami rysunków i tabel.

Słownik pojęć

AWS

AWS Lambda

Ciepły start

Ekosystem Java

FaaS

JVM

Serverless

Zimny start

SDK

1. Podstawy działania AWS Lambda

1.1. Model bezserwerowy

Jednym z dynamicznie rozwijających się obszarów chmur obliczeniowych są usługi bezserwerowe. W 2025 roku rynek usług opartych o architekturę serverless jest wart 17,88 miliarda dolarów amerykańskich, a według prognoz jego wartość wzrośnie do 41,14 miliarda w roku 2029 [62]. 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 [31]. 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" [69]. 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 [13] [12], 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 [71].

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 [11]. Model ten może być nieefektywny kosztowo dla systemów o zmiennym lub niewielkim obciążeniu. Często powoduje to konieczność tworzenia rozwiązań skalowania mocy obliczeniowej w zależności od ruchu w aplikacji, co wymaga znaczących nakładów pracy.

Usługi serverless wprowadzają kompletnie nowy sposób rozliczeń, często określany jako model „pay-per-use" (ang. płać za użycie) [31]. Obejmuje on głównie dwa elementy: liczbę wywołań (zapytań lub zdarzeń uruchamiających funkcję) oraz łączny czas obliczeń wykorzystany przez wszystkie wykonania. Czas ten jest zazwyczaj mierzony z wysoką precyzją (np. co do milisekundy lub 100 milisekund) i powiązany z ilością przydzielonej pamięci (np. w jednostkach GB-sekund) [13]. Jest to możliwe dzięki sterowanej zdarzeniami naturze usług bezserwerowych.

W usługach serverless zasoby obliczeniowe nie są stale aktywne. W momencie pojawienia się zdarzenia (np. żądania HTTP, wiadomości w kolejce, zmiany w bazie danych) dostawca chmurowy alokuje część swojej infrastruktury dla naszych obliczeń. Dostawca chmurowy zarządza pulą zasobów i mogą być one zwolnione po zakończeniu obliczeń. Dzięki temu użytkownicy płacą wyłącznie za używaną w danym momencie infrastrukturę, co eliminuje koszty związane z jej bezczynnością. Pozwala to także na osiągnięcie bardzo wysokiej skalowalności, którą zarządza platforma chmurowa, a nie programista aplikacji.

Warto zaznaczyć, że model bezserwerowy nie ma jeszcze ogólnoprzyjętej przez społeczność terminologii [69]. Powyższa definicja została przyjęta na potrzeby niniejszej pracy, jednak istnieją także alternatywne opisy formy serverless. Jedną z nich zaprezentował Ben Kehoe [44], który definuje serverless jako pewien stan umysłu (ang. state of mind). W swoim artykule Kehoe przekazuje, że „serverless to sposób na skupienie się na wartości biznesowej" [44]. Jednocześnie odrzuca on skupianie się na elementach modeli bezserwerowych jak na przykład kod, funkcje, serwisy zarządzane czy koszt takiej architektury.

Inne podejście prezentuje znany w społeczności Yan Cui [26], który dochodzi do wniosku, że powinniśmy utożsamiać serverless z FaaS (ang. function-as-a-service). Wynika to częściowo z niedokładności innych definicji, a sam Cui proponuje skupienie się na prostej definicji. Według autora pozwoliłoby to na łatwiejsze zrozumienie modeli bezserwerowych przez początkujących.

Niezależnie od przyjętej definicji praca w modelu bezserwerowym wymaga często użycia innych schematów niż te już znane przez programistów. Interesującym przypadkiem jest migracja do modelu bezserwerowego firmy PostNL, opisana przez Donkersgoed [68]. Jako jeden z problemów dla takiej tranzycji wskazuje on często wąskie specjalizacje inżynierów (np. aplikacje monolityczne Java czy aplikacje Kubernetes). Model serverless łączy się z użyciem różnych technologii i podejść. Części z nich można nauczyć się w ramach pracy, jednak część podejść wymaga pewnego „oduczenia się" starych praktyk. W przypadku PostNL problem ten wymagał utworzenia dedykowanej ścieżki nauki i rozwoju, we współpracy z AWS.

1.2. Funkcja jako usługa

Funkcja jako usługa (ang. Function as a Service, FaaS) to jeden z kluczowych komponentów modelu bezserwerowego. W ramach rozdziału serwis ten zostanie opisany w kontekście jego działania i cech. Celem rozdziału jest scharakteryzowanie funkcji jako ogólnego konceptu, niezależnie od dostawcy chmurowego. Konkretna implementacja modelu (AWS Lambda) zostanie zostanie w następnym rozdziale.

Jedna z definicji funkcji chmurowych została przedstawiona przez Eismanna i innych autorów: „funkcja chmurowa to mała, bezstanowa usługa na żądanie z pojedynczą odpowiedzialnością funkcjonalną" [69]. Stwierdzają oni, że konceptualnie funkcja chmurowa to typ

mikroserwisu, który posiada wyłącznie jeden punkt końcowy HTTP lub pojedynczą odpowiedzialność. Usługa jako serwis to techniczna implementacja tego modelu, definiowana jako „forma przetwarzania bezserwerowego, która zarządza zasobami, cyklem życia i sterowanym zdarzeniami wykonywaniem funkcji chmurowej dostarczanej przez użytkownika." [69].

Pierwszą istotną cechą poruszoną w definicji funkcji jako usługi jest zarządzanie zasobami. Oznacza to, że dostawca usługi przejmuje pełną odpowiedzialność za przydzielanie, konfigurację oraz utrzymanie jednostek niezbędnych do uruchomienia funkcji chmurowej. Mogą to być serwery, maszyny wirtualne czy kontenery. Użytkownik funkcji nie musi być jednak tego świadomy, a może skupić się wyłącznie na działeniu dostarczanej funkcji [30].

Platforma FaaS oprócz zarządzania zasobami odpowiada również za cykl życia funkcji [70]. Deweloper zapewnia jedynie kod, który ma zostać wykonany z użyciem infrasktruktury chmurowej. Dostawca następnie wdraża dostarczony kod oraz inicjuje środowisko uruchomieniowe (co jest przyczyną tzw. „zimnych startów", omówionych w dalszej części rozdziału). Następnie monitoruje on obciążenie funkcji i w razie wzmożenia ruchu tworzy nowe instancje funkcji (np. maszyny wirtualne) lub usuwa te nadmiarowe. W razie potrzeby dostawca może także kompletnie wygasić funkcję poprzez zwolnienie wszystkich instancji. Oznacza to, że w przypadku braku ruchu użytkownik nie ponosi żadnych kosztów związanych z usługą.

Kluczową rolę w cyklu życia funkcji odgrywają zdarzenia. Model FaaS jest z natury sterowany zdarzeniami, co oznacza, że wykonanie funkcji chmurowej nie jest inicjowane bezpośrednio przez użytkownika w tradycyjny sposób. Następuje ono w odpowiedzi na wystąpienie określonego zdarzenia w systemie lub infrastrukturze chmurowej. Źródłem zdarzeń mogą być różne usługi jak żądanie HTTP, czy przesłanie pliku do usługi przechowywania obiektów (np. S3) [13]. Dzięki temu model ten pozwala na tworzenie elastycznych architektur. Funkcje w nich mogą reagować na zdarzenia w sposób zarówno synchroniczny, jak i asynchroniczny.

Kluczem funkcji jako usługi z perspektywy programisty jest kod. Jest on dostarczany przez dewelopera w momencie tworzenia nowej funkcji i jest niezbędny do jej utworzenia. Zgodnie z przytoczoną definicją jest to zazwyczaj niewielki fragment kodu (lub jego kolejnej wersji), który realizuje pojedynczą, dobrze zdefiniowaną odpowiedzialność. Istotną cechą FaaS jest wsparcie dla wielu języków programowania (np. Java, JavaScript, Python) oraz dostarczenie środowiska wykonawczego do ich uruchomienia. Dzięki temu użytkownik jest w stanie skupić się na logice biznesowej aplikacji [31].

Bardzo istotną cechą fukcji chmurowych jest ich bezstanowość, co znacząco wpływa na wcześniej poruszone elementy. Bezstanowość funkcji oznacza, że usluga nie przechowuje żadnych danych pomiędzy kolejnymi wywołaniami. Wszelkie potrzebne dane wejściowe są przekazywane w momencie wywołania (najczęściej jako część zdarzenia wywołującego uruchomienie kodu). Następnie, ewentualne wyniki muszą być bezpośrednio zwrócone z funkcji (w przypadku wywołań synchronicznych) lub zapisane w zewnętrznych usługach (w przypadku wywołań asynchronicznych). Usługami tymi mogą być na przykład bazy danych, czy systemy plików. Powoduje to, że usługi FaaS są bardzo często integrowane z zarządzanymi serwisami oferowanymi przez dostawcę chmurowego. Potwierdza to Eismann i inni autorzy, stwierdzając poprzez analizę aplikacji otwartoźródłowych, że jedynie 12% z nich nie korzysta z rozwiązań BaaS (ang. Backend as a Service), w kontekście serwisów zarządznych.

Oprócz bezstanowości, kolejną charakterystyczną właściwością funkcji chmurowych jest ich zazwyczaj krótki czas życia. Oznacza to, że pojedyncze wywołanie jest z natury krótkie. Analiza przeprowadzona przez Eismanna i innych autorów wykazała, że obecne platformy

bezserwerowe mogą nie być odpowiednie dla zadań długotrwałych [31]. Ich badanie 89 aplikacji serverless potwierdziło tę hipotezę, wskazując, że aż 75% zbadanych przypadków użycia miało czas wykonania mieszczący się w zakresie sekund. Wskazuje to, że funkcje FaaS są najczęściej wykorzystywane do realizacji zadań, które z natury są krótkie i nie wymagają długotrwałego przetwarzania w ramach pojedynczej instancji funkcji.

Rozwinięciem koncepcji zarządzania cyklem życia przez platformę FaaS jest zjawisko tzw. zimnych i ciepłych startów funkcji. Zimny start (ang. cold start) występuje, gdy funkcja wywołana zostanie w momencie, gdy żadna z jej instancji nie jest aktywna i gotowa do jego obsługi. W takim przypadku musi zostać wykonany cały proces inicjaliacji: przydzielenie zasobów, wdrożenie kodu oraz uruchomienie środowiska wykonawczego [69]. Ten proces wprowadza zauważalne opóźnienie, które jest szczególnie problematyczne dla aplikacji wrażliwych na czas odpowiedzi.

Przeciwieństwem zimnych startów są tzw. ciepłe starty (ang. warm starts). Występuje on, gdy żądanie może zostać wykonane przez już istniejącą (czyli „rozgrzaną") instancję funkcji [69]. Jest to możliwe dzięki temu, że dostawcy chmurowi pozostawiają instancje aktywne przez pewien czas po wywołaniu funkcji. Pozwala to następnie na jej ponowne wykorzystanie i uniknięcie kosztownego procesu inicjaliacji.

Model funkcji jako usług znajduje zastosowanie w wielu różnych przypadkach. Mogą to być warstwy serwerowe aplikacji mobilnych i społecznościowych, czy przetwarzanie multimediów, takich jak transkodowanie wideo i skalowanie obrazów [31]. FaaS wykorzystywany jest również do implementacji funkcji pomocniczych. Mogą to być elementy procesów automatycznej integracji i wdrażania, czy monitoringu systemów. Na bazie badań projektów otwartoźródłowych możemy stwierdzić, że FaaS służy zarówno do budowy kluczowych funkcjonalności aplikacji, zadań pomocniczych oraz obliczeń naukowych [31].

Szerokie możliwości użycia omawianych funkcji znacząco wpływają na jej popularność. Sam koncept nie jest związany z konkretną chmurą obliczeniową, jednak konkretne implementacje modelu są oferowane przez większość dużych dostawców. Pierwszą znaczącą usługą była AWS Lambda (zaprezentowana w 2015 roku), oferowana przez Amazon Web Services [32]. Jednak także platformy jak Azure czy Google Cloud Platform oferują swoje alternatywy (odpowiednio Azure Function i Google Cloud Functions). Model funkcji jako usługi dostarczany jest także przez mniej popularne chmury obliczeniowe, jak Vercel, czy Netlify.

1.3. AWS Lambda

AWS Lambda to konkretna implementacja funkcji jako usługi oferowana przez Amazon Web Services od 2015 roku. W ramach rozdziału opisano działanie usługi, wraz z jej najważniejszymi z perspektywy programisty cechami. Skupiono się na konkretnym użyciu języka Java, który jest dostępny w serwisie.

AWS Lambda była pierwszą platformą FaaS oferowaną przez dużego dostawcę chmurowego. Znacząco przyczyniło się to do popularyzacji paradygmatu bezserwerowego. Usługa ta implementuje założenia modelu funkcji jako usługi, oferując programistom możliwość uruchamiania kodu bez konieczności zarządzania infrasktrukturą. AWS Lambda przejmuje odpowiedzialność za przydzielenie zasobów, skalowanie, monitorowanie oraz zarządzanie cyklem życia wykonywanych funkcji. Umożliwia to twórcom oprogramowania skupienie się na logice biznesowej

aplikacji, rozliczając ich jedynie za faktyczny czas obliczeń i liczbę wywołań.

```java
public class MyHandler implements RequestHandler<MyHandler.Request, String> {
  public record Request(String message) {}

  public record Response() {}

  @Override
  public Response handleRequest(Request event, Context context) {
    // logika biznesowa aplikacji
    return Response()
  }
}
```

Rysunek 1.1: Przykładowa implementacja funkcji AWS Lambda w języku Java [źródło: opracowanie własne]

Pierwszym krokiem wdrożenia funkcji AWS Lambda jest napisanie uruchamianego kodu. Przykładowa implementacja została przedstawiona na Rysunku 1.1. Odpowiedni interfejs, który wykorzystuje programista, jest dostarczany przez AWS. Jego implementacja w klasie MyHandler pozwala na nadpisanie metody handleRequest, która jest wywoływana w momencie obsługi zdarzenia. W ciele metody umieszczana jest konkretna logika biznesowa funkcji chmurowej, na której skupia się użytkownik.

Aby kod został uruchomiony w chmurze należy utworzyć nową funkcję AWS Lambda. Może to być wykonane poprzez konsolę Amazon Web Services, AWS CLI (ang. Command Line Interface) lub narzędzia infrasktruktury jako kod (np. AWS CDK, Terraform) [13]. Niezbędne do tego jest przygotowanie zasobów do wdrożenia. Może to być wykonane na dwa sposoby: poprzez pliki ZIP lub JAR (istnieje także możliwość wdrożenia obrazów Dockerowych, jednak w ramach podrozdziału skupiamy się na środowisku uruchomieniowym Java) [13]. Tak przygotowany plik jest gotowy do wdrożenia podczas uruchamiania funkcji.

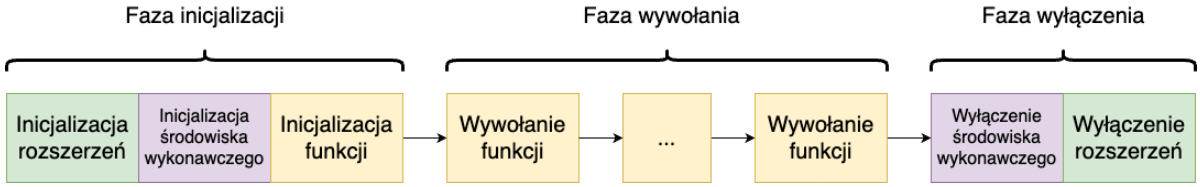
Kolejnym krokiem jest skonfigurowanie funkcji w usłudze AWS Lambda, co obejmuje zdefiniowanie kluczowych parametrów [13]. Jednym z najważniejszych jest pamięć funkcji, która odpowiada ilości pamięci RAM w megabajtach. Innym parametrem jest limit czasu wykonania, który definiuje maksymalny czas, przez jaki pojedyncze wywołanie funkcji może być przetwarzane, zanim zostanie automatycznie zakończone przez platformę. Dodatkowo, programista ma możliwość zdefiniowania zmiennych środowiskowych, które pozwalają na przekazywanie konfiguracji do kodu funkcji w czasie jej działania.

Aby funkcja mogła zostać zintegrowana z tworzonym systemem należy skonfigurować źródła zdarzeń. Zgodnie z naturą FaaS, funkcje Lambda są projektowane do uruchamiania w odpowiedzi na zdarzenia pochodzące z szerokiego ekosystemu usług AWS lub źródeł zewnętrznych. Do typowych usług integrujących się z Lambda jako wyzwalacze należą m.in. [13]:

- Amazon S3 (np. przy tworzeniu nowego obiektu),

- Amazon API Gateway (dla żądań HTTP),
- Amazon DynamoDB Streams (reakcja na zmiany w tabelach),
- Amazon SQS (nowe wiadomości w kolejce),
- Amazon SNS (powiadomienia tematyczne),
- Amazon EventBridge (dla zdarzeń harmonogramowych oraz zdarzeń z innych usług i aplikacji).

Dzięki temu programista określa, w jakich okolicznościach i z jakimi danymi wejściowymi funkcja Lambda ma zostać automatycznie uruchomiona. Poprzez skonfigurowane wcześniej zdarzenia funkcja może zostać uruchomiona, co zachodzi w ramach określonego cyklu życia. Składa się on z trzech etapów: inicjalizacji, wywołania i wyłączenia [13], przedstawionych na Rysunku 1.2.



Rysunek 1.2: Cykl życia funkcji AWS Lambda [źródło: opracowanie własne na bazie dokumentacji AWS Lambda [13]]

Pierwszym etapem uruchomienia funkcji jest faza inicjalizacji, która wykonuje się gdy zasoby nie są jeszcze przydzielone. Składa się ona z trzech mniejszych etapów [13]:

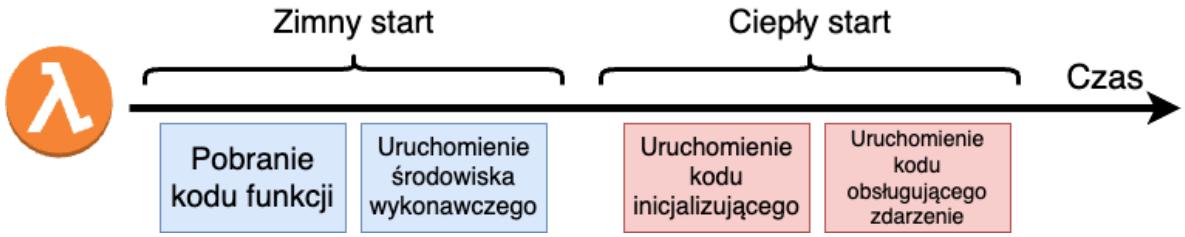
- Inicjalizacji rozszerzeń (np. narzędzia do monitorowania)
- Inicjalizacji środowiska wykonawczego (np. maszyny wirtualnej Java)
- Inicjalizacji funkcji (uruchomienie kodu statycznego funkcji)

Po fazie inicjalizacji, gdy środowisko wykonawcze jest już przygotowane, następuje właściwa faza wywołania funkcji. Rozpoczyna się ona, gdy platforma Lambda przesyła zdarzenie do przygotowanego środowiska uruchomieniowego. W ten sposób inicjowane jest wykonanie kodu funkcji. Całkowity czas trwania tej fazy jest ściśle ograniczony przez skonfigurowany wcześniej limit czasu wykonania. Faza wywołania kończy się, gdy środowisko uruchomieniowe zakończy swoje zadania i zasygnalizuje gotowość do obsługi kolejnego żądania poprzez wysłanie komunikatu do usługi Lambda.

Ostatnim etapem w cyklu życia środowiska wykonawczego AWS Lambda jest faza wyłączenia. Następuje on gdy usługa decyduje o zakończeniu działania konkretnego środowiska uruchomieniowego (np. z powodu zmniejszenia aktywności). Po tej fazie zasoby dostawcy chmurowego zostają zwolnione.

Istotne dla działania AWS Lambda są tak zwane zimne i ciepłe starty, które wynikają bezpośrednio z cyklu życia funkcji. Aby umożliwić wywołanie, platforma AWS Lambda musi

pobrać kod użytkownika, a następnie utworzyć nowe środowiska wykonawcze. Fazy te są określane jako tzw. zimne starty (ang. cold starts) [13], co zostało przedstawione na Rysunku 1.3. Dostawca chmurowy nie pobiera opłaty za czas trwania tych etapów, jednak wpływają one na ogólną wydajność funkcji. Po przygotowaniu środowiska może zostać uruchomiony kod (inicjujący oraz obsługujący funkcje). Następnie, środowisko wykonawcze funkcji jest zatrzymywane, jednak nie usuwane. Pozwala to następnie na jego ponowne użycie (przez pewien określony czas), co pozwala na szybsze uruchomienie funkcji. Zjawisko to nazywane jest ciepłym startem (ang. warm start) [13].



Rysunek 1.3: Proces uruchomienia AWS Lambda w kontekście zimnych i ciepłych startów [źródło: opracowanie własne na bazie dokumentacji AWS Lambda [13]]

2. Przegląd literatury

Usługa AWS Lambda oraz podejście bezserwerowe stale zyskuje na popularności. Amazon Web Services co roku tworzy nowe możliwości dla funkcji AWS Lambda, a projektowanie aplikacji o nie opartych jest stałym elementem dyskusji w społeczności programistycznej. Ze względu na dynamikę rozwoju, przegląd literatury stanowi bardzo ważny element pracy. Pozwoli to na zrozumienie aktualnego stanu wiedzy na temat optymalizacji wydajności AWS Lambda oraz samej usługi. Dodatkowo, umożliwi to identyfikację obszarów, które wymagają dalszych badań.

2.1. Cel przeglądu

Celem przeglądu jest poznanie obecnego stanu wiedzy oraz wykorzystanie jej jako wsparcia w odpowiedzi na postawione w pracy pytania badawcza. Zamierzeniem przeglądu jest analiza trzech głównych obszarów:

1. Czynników wpływających na wydajność AWS Lambda.
2. Istniejących metod optymalizacji wydajności AWS Lambda dla ekosystemu Java.
3. Charakterystyki rozwoju funkcji AWS Lambda z perspektywy pracy programisty.

Obszary te zostały wybrane na podstawie pytań badawczych. Ich zrozumienie pozwoli na identyfikację zagadnień, które nie zostały wystarczająco zbadane, a mogą zawierać potencjalne metody usprawnienia wydajności funkcji AWS Lambda w ramach ekosystemu Java. Dodatkowo, pozwoli to na przygotowanie badań, które będą lepiej odzwierciedlać rzeczywistą praktykę aplikacji bezserwerowych.

2.2. Metodyka przeglądu literatury

Do wykonania przeglądu literatury wybrano metodykę szybkiego przeglądu (ang. rapid review). Szybkie przeglądy to metoda badań wtórnych stosowana w inżynierii oprogramowania, której celem jest szybka synteza wyników badań. Ich głównym zadaniem jest dostarczanie aktualnych informacji opartych na dowodach. Pomaga to praktykom podejmować decyzje dotyczące specyficznych problemów w ramach ich kontekstu pracy i ograniczeń czasowych. Metoda ta często wykonywana jest przez jedną osobę, przy jednoczesnym użyciu bardziej restrykcyjnych kryteriów. Jest to jednak świadoma strategia mająca na celu redukcję czasu i wysiłku [20]. Proces szybkiego przeglądu składa się z trzech etapów, które zostały zrealizowane w pracy:

1. Zaplanowanie (określenie potrzeby przeglądu, problemu i pytań badawczych).

2. Przeprowadzenie (stworzenie i wykonanie strategi wyszukiwania, procedur selekcji, oceny jakości, ekstrakcji i syntezy).

3. Raportowanie (omówienie wyników przeglądu i odpowiedź na pytania badawcze).

2.3. Proces przeglądu

2.3.1. Omówienie pytań badawczych

Formułowanie pytań badawczych to istotna część przeglądu [46]. Pytania te, oparte na celu pracy, wskazują kierunek przy opracowywaniu i wdrażaniu kryteriów przeglądu. W ramach wykonanego przeglądu literatury postawiono konkretne pytania, mające na celu ukierunkowanie analizy, co umożliwi odpowiedź na główne pytania badawcze całej pracy. Do każdego z nich dołączono krótkie wyjaśnienie i motywację.

- PB1: Jakie są główne czynniki wpływające na wydajność funkcji AWS Lambda?

  Odpowiedź na postawione pytanie ma na celu zrozumienie sposobu działania AWS Lambda pod kątem wydajności. Powyższe pytanie nie odnosi się wyłącznie do technologii z środowiska Java, co pozwoli na poszerzenie analizy. Dostarczone odpowiedzi będą wsparciem dla znalezienia nowych metod optymalizacji wydajności, ze względu na zrozumienie na jakie czynniki owe metody mogą wpływać. Dodatkowo, zidentyfikowane czynniki pozwolą na przygotowanie bardziej jakościowych badań. Badania będą mogły być realizowane w ramach różnych scenariuszy, które będą wykonane dla różnych wartości znalezionych parametrów.

- PB2: Jakie są istniejące metody optymalizacji wydajności funkcji AWS Lambda działających w ekosystemie Java?

  Pytanie pozwoli ustalić jaki jest istniejący stan wiedzy dla metod optymalizacji wydajności funkcji AWS Lambda w ramach środowiska Java. W ramach tego pytania analiza skupi się na szeroko pojętym ekosystemie Java, czyli różnych językach wywodzących się z Javy, bibliotekach, frameworkach i środowiskach wykonawczych. Analiza pozwoli na wskazanie obszarów, które nie zostały jeszcze zbadane lub zostały zbadane niewystarczająco. Tak wyznaczone zagadnienia będą potencjalnym miejscem poszukiwania nowych metod. Stanowi to znaczącą pomoc w realizacji celu pracy.

- PB3: Jakie są cechy rozwoju aplikacji w architekturze bezserwerowej AWS Lambda?

  Ostatnie pytanie skupia się na perspektywie programisty tworzącego aplikacje opierające się o funkcje bezserwerowe oferowane przez AWS. Zdecydowano się na przegląd literatury w tym zakresie ze względu na specyficzne podejście do tworzenia takich aplikacji. Rozpatrzenie tego pozwoli następnie na analizę zaproponowanych metod optymalizacji pod względem ich wpływu na proces wytwarzania oprogramowania. Przewiduje się, że analiza ta może być skomplikowana z powodu zróżnicowanych praktyk budowy wspomnianych systemów.

2.3.2. Przeszukiwane zasoby

Biblioteki cyfrowe

Głównymi źródłami informacji, które zostały przeszukane były biblioteki cyfrowe IEEE oraz Scopus. Wybór bibliotek był podyktowany dostępnością w e-zasobach Politechniki Wrocławskiej. Kluczową cechą wybranych bibliotek była możliwość użycia operatorów logicznych i wyszukiwania słów kluczowych w konkretnych elementach pracy. W ramach pierwszych iteracji stwierdzono potrzebę ich użycia ze względu na obszerność tematów bezserwerowych, co opisano w dalszej części pracy.

Szara literatura

W celu uwzględnienia w przeglądzie szarej literatury zdecydowano się na użycie wyszukiwarki Google Scholar. W procesie użyto zapytań odpowiednio dostosowanych do wyszukiwarki, gdzie przeszukiwano tytuł, abstrakt oraz pełny tekst pracy.

Rekomendacje w zakresie włączania szarej literatury do systematycznych przeglądów wyznaczyli Kitchenham, Madeyski i Budgen [47]. Podkreślają oni jendak, aby wyraźnie rozróżnić informacje uzyskane z źródeł zgodnych z definicją „Prague" a innych zasobów internetowych (jak np. blogi). W ramach przeglądu przeszukiwano dodatkowe materiały internetowe, które posłużyły jako wsparcie wyjaśnienia poszczególnych metod optymalizacji. Nie zostały one jednak włączone bezpośrednio do wyników przeglądu.

2.3.3. Wyszukiwane terminy

Istotnym elementem strategii wyszukiwania było stworzenie odpowiednich zapytań dla wyszukiwarek. Zostały one przygotowane na bazie postawionych pytań badawczych do przeglądu literatury. Każde zapytanie zostało przygotowane według poniższego procesu:

1. Wyznaczenie słów kluczowych na bazie pytania badawczego, gdzie słowa kluczowe definiują odpowiednie domeny [72]. Pozwoliło to na wykonanie pierwszych prób wyszukiwania, co pozwoliło na późniejsze dostosowanie zapytań i użycie odpowiednich filtrów oferowanych przez wyszukiwarki. Wszystkie słowa kluczowe zostały przedstawione w języku polskim i angielskim.

2. Dla każdego słowa kluczowego określono synonimy i wyrazy bliskoznaczne. Tam gdzie to możliwe terminy zostały zawarte zarówno w liczbie mnogiej jak i pojedynczej.

3. Połączono terminy odpowiednimi operatorami logicznymi. Synonimy i wyrazy bliskoznaczne zostały połączone operatorami OR, a poszczególne ich grupy AND.

4. Dostosowano wyszukiwania fraz aby obejmowały elementy pracy, jak: słowa kluczowe, tytuł, abstrakt oraz pełny tekst (w miarę dostępności dla wyszukiwarki).

Ustalenie słów kluczowych

Dla każdego pytania badawczego ustalono zbiór pytań kluczowych:

- PB1: „Jakie są główne czynniki wpływające na wydajność funkcji AWS Lambda?”
  - W języku polskim: czynniki, wydajność, AWS Lambda
  - W języku angielskim: factors, performance, AWS Lambda
- PB2: „Jakie są istniejące metody optymalizacji wydajności funkcji AWS Lambda działających w ekosystemie Java?”
  - W języku polskim: optymalizacja, wydajność, AWS Lambda, Java
  - W języku angielskim: optimization, performance, AWS Lambda, Java
- PB3: „Jakie są cechy rozwoju aplikacji w architekturze bezserwerowej AWS Lambda?”
  - W języku polskim: rozwój oprogramowania, AWS Lambda, model bezserwerowy
  - W języku angielskim: software development, AWS Lambda, serverless

Ustalenie synonimów i wyrazów bliskoznacznych

Dla każdego słowa kluczowego ustalono synonimy, wyrazy bliskoznaczne i powiązane terminy. Zostały one określone poprzez iteracyjne testowanie kolejnych terminów. Słowo kluczowe „AWS Lambda” zostało rozdzielone na dwa pojęcia: AWS oraz Lambda. Wynika to z różnych użyć tych terminów w wyszukiwanych pracach.

Opracowane słowa kluczowe z powiązanymi terminami:

- PB1:
  - **factors**: factor, factors, aspect, aspects, component, components, parameter, parameters, influence, influences, consideration, considerations
  - **performance**: performance, efficiency, effectiveness, speed, throughput, responsiveness, latency, execution time, processing time, productivity, computational efficiency, optimization
  - **AWS**: AWS, Amazon
  - **Lambda**: Lambda, FaaS, Function-as-a-Service, serverless function, serverless functions, cloud function, cloud functions
- PB2:
  - **optimization**: optimization, optimisation, optimizing, improve, improvement, improving, method, methods, strategy, strategies, approach, approaches, technique, techniques, procedure, procedures, practice, practices, mechanism, mechanisms, solution, solutions, pattern, patterns
  - **performance**: performance, efficiency, effectiveness, speed, throughput, responsiveness, latency, execution time, processing time, delay, warm start, cold start
  - **Java**: Java, JVM, Java Virtual Machine, JDK, Kotlin, Scala, GraalVM, Spring, SpringBoot

  - **AWS**: AWS, Amazon
  - **Lambda**: Lambda, FaaS, Function as a Service, serverless function, serverless functions

- PB3:
  - **software development**: development process, development workflow, development lifecycle, software development, developing, software engineering, developer experience, patterns
  - **serverless**: serverless
  - **Lambda**: Lambda, FaaS, Function as a Service, serverless function, serverless functions
  - **AWS**: AWS, Amazon

Opracowane zapytania

Na bazie słów kluczowych i pozwiązanych z nimi terminów opracowano poniższe zapytania:

- PB1:

  („factor" OR „factors" OR „aspect" OR „aspects" OR „component" OR „components" OR „parameter" OR „parameters" OR „influence" OR „influences" OR „consideration" OR „considerations") AND („performance" OR „efficiency" OR „effectiveness" OR „speed" OR „throughput" OR „responsiveness" OR „latency" OR „execution time" OR „processing time" OR „productivity" OR „computational efficiency" OR „optimization") AND („Lambda" OR „FaaS" OR „Function as a Service" OR „serverless function" OR „serverless functions" OR „cloud function" OR „cloud functions") AND („AWS" OR „Amazon")

- PB2:

  („optimization" OR „optimisation" OR „optimizing" OR „improve" OR „improvement" OR „improving" OR „method" OR „methods" OR „strategy" OR „strategies" OR „approach" OR „approaches" OR „technique" OR „techniques" OR „procedure" OR „procedures" OR „practice" OR „practices" OR „mechanism" OR „mechanisms" OR „solution" OR „solutions" OR „pattern" OR „patterns") AND („performance" OR „efficiency" OR „effectiveness" OR „speed" OR „throughput" OR „responsiveness" OR „latency" OR „execution time" OR „processing time" OR „delay" OR „warm start" OR „cold start") AND („Java" OR „JVM" OR „Java Virtual Machine" OR „JDK" OR „Kotlin" OR „Scala" OR „GraalVM" OR „Spring" OR „SpringBoot") AND („Lambda" OR „FaaS" OR „Function as a Service" OR „serverless function" OR „serverless functions") AND („AWS" OR „Amazon")

- PB3:

  („serverless") AND („development process" OR „development workflow" OR „development lifecycle" OR „software development" OR „developing" OR „software engineering"

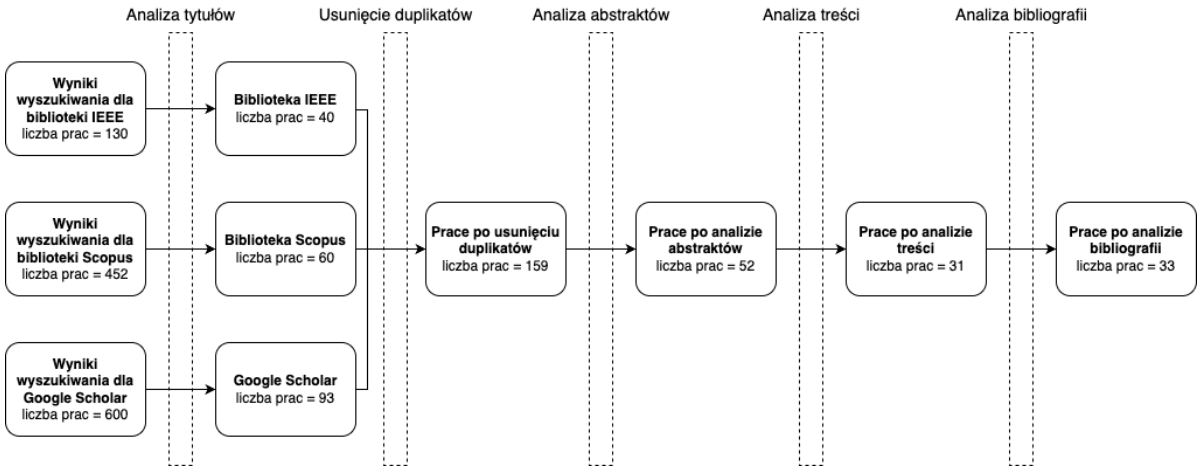
OR „developer experience” OR „patterns”) AND („Lambda” OR „FaaS” OR „Function as a Service” OR „serverless function” OR „serverless functions”) AND („AWS” OR „Amazon”)

W pierwszych wyszukaniach z użyciem powyższych zapytań zauważono, że znacząca część wyników nie odnosi się do serwisu AWS Lambda, na którym skupia się praca. Wyniki te odnosiły się przeważnie do innych usług lub platform bezserwerowych (np. OpenWhisk). Z tego powdou zdecydowano się na zawężenie wyników poprzez dodanie wymogu występowania terminów AWS i Lambda (oraz ich powiązanych terminów) w tytule lub abstrakcie. W wyszukaniach IEEE oraz Scopus warunek ten został dodany do istniejącej części zapytania. W wyszukiwarce Google Scholar, została dodana część wyszukująca jakikolwiek z powiązanych terminów z obu tych grup w tytule.

2.3.4. Selekcja literatury

W celu ograniczenia liczby analizowanych prac i zgromadzić badania niezbędne do dalszych analiz, określono poniższe kryteria włączenia oraz wyłączenia pozycji literaturowych [20]:

- praca została opublikowana w 2014 roku lub później - jest to rok kiedy po raz pierwszy zaprezentowano AWS Lambda, zatem założono, że prace jej temat nie zostały napisane wcześniej,
- praca zawiera się w dziedzinie informatyki (ang. computer science) lub inżynierii oprogramowania (ang. software engineering)
- praca została napisana w języku angielskim
- duplikaty pomiędzy poszczególnymi wyszukiwarkami oraz frazami nie są włączane do przeglądu literatury



Rysunek 2.1: Proces selekcji literatury [źródło: opracowanie własne]

Tabela 2.1: Liczba wybranych prac w zależności od etapu selekcji [źródło: opracowanie własne]

Fraza	Biblioteka cyfrowa	Liczba prac po zakończeniu danego etapu					
		Wyszukanie	Analiza tytułów	Usunięcie duplikatów	Analiza abstraktów	Analiza treści	Snowballing
1	IEEE	59	17	17	12	8	8
	Scopus	217 (200)	17	15	5	3	4
	Google Scholar	4050 (200)	50	40	9	4	4
2	IEEE	8	4	4	4	4	5
	Scopus	52	9	6	3	3	3
	Google Scholar	1550 (200)	26	23	6	2	2
3	IEEE	63	19	15	5	3	3
	Scopus	205 (200)	34	23	3	2	2
	Google Scholar	397 (200)	17	16	5	2	2
Suma		**1182**	**193**	**159**	**52**	**31**	**33**

Wcześniej opracowane zapytania zostały użyte do wyszukań w poszczególnych wyszukiwarkach. Dla prac wyszukanych w poszczególnych wyszukiwarkach, został zastosowany proces selekcji. Liczba prac w poszczególnych etapach procesu została przedstawiona w Tabeli 2.1. Dodatkowo, zobrazowano proces selekcji na Rysunku 2.1.

W przypadku, gdy dla danej frazy wyszukiwania odnaleziono więcej niż 200 publikacji, do dalszej analizy kwalifikowano pierwszych 200 pozycji, posortowanych według malejącej trafności. Następnie, każda z tych prac była poddawana procesowi selekcji, który polegał na poddaniu kryteriom włączenia i wyłączenia, analizie tytułów, abstraktów i treści. Jeśli praca została włączona do przeglądu literatury, dokonywano analizy jej bibliografii, wykorzystując metodę kuli śnieżnej (ang. snowballing sampling method) [37]. Artykuły włączone w ten sposób do przeglądu zostały przypisane do frazy, z której pochodziła pierwotna praca.

Dla każdej pracy naukowej, która została włączona do przeglądu literatury, przygotowano odpowiedni wpis w formacie BibTeX, umożliwiający jej późniejsze zacytowanie. Zapytania do bibliotek cyfrowych zostały wykonane w dniu 8 kwietnia 2025 roku.

2.3.5. Ocena jakości

2.4. Wyniki przeglądu

W ramach przeglądu wybrano 33 prace badawcze, które zostały przedstawione w Tabeli 2.2. Na bazie prac rozpatrzono postawione pytania badawcze. Odpowiedzi na nie zostały zawarte w kolejnych podrozdziałach.

Tabela 2.2: Prace wybrane w ramach przeglądu literatury [źródło: opracowanie własne]

Przypis	Autorzy	Rok publikacji
[58]	H. Puripunpinyo, M.H. Samadzadeh	2017
[33]	T. Elgamal et al.	2018
[53]	J. Manner et al.	2018
[41]	D. Jackson, G. Clynch	2018
[52]	W. Lloyd et al.	2018
[48]	K. Kritikos, P. Skrzypek	2018
[15]	D. Bardsley, L. Ryan, J. Howard	2018
[57]	M. Pawlik, K. Figiela, M. Malawski	2019
[64]	S. Shrestha	2019
[43]	A. Kaplunovich	2019
[51]	P. Leitner et al.	2019
[45]	D. Kelly, F. Glavin, E. Barrett	2020
[59]	D. Quaresma, D. Fireman, T. E. Pereira	2020
[25]	R. Cordingly et al.	2020
[63]	R. Ritzal	2020
[56]	J. Nupponen, D. Taibi	2020
[22]	R. Chatley, T. Allerton	2020
[36]	B. C. Ghosh et al.	2020
[73]	T. Yu et al.	2020
[19]	J. Carreira et al.	2021
[55]	M. Nazari et al.	2021
[61]	A. Raza et al.	2021
[30]	S. Eismann et al.	2021
[49]	D. Lambion et al.	2022
[27]	J. Dantas, H. Khazaei, M. Litoiu	2022
[66]	D. Taibi, B. Kehoe, D. Poccia	2022
[24]	R. Cordingly, S. Xu, W. Lloyd	2022
[23]	X. Chen et al.	2023
[34]	P. O. Ferreira Dos Santos et al.	2023
[54]	J. M. Menéndez, M. Bartlett	2023
[21]	A. P. Cavalheiro, C. Schepke	2023
[29]	A. Ebrahimi, M. Ghobaei-Arani, H. Saboohi	2024
[39]	D. Ivanov, A. Petrova	2024

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 [45] 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.

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 [59] 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 [33] 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 [57] 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.

Cordingly i inni [24] zaproponowali podejście CPU Time Accounting Memory Selection (CPU-TAMS), które pozwala na optymalizację wielkości pamięci. Bazuje ono na szczegółowych metrykach czasu procesora, takich jak czas w trybie użytkownika, jądra czy oczekiwanie na

I/O. Celem jest znalezienie wielkości pamięci, która zapewnia najlepszy stosunek wydajności do kosztu, a nie tylko najniższego kosztu lub najkrótszego czasu wykonania. Badania wykazały, że zasoby takie jak czas procesora, przepustowość I/O oraz sieci rosną wraz z wielkością pamięci. Jednak wzrost ten nie jest zawsze liniowy, często osiągając plateau przy wielkościach pamięci około 1.5-2GB dla I/O i sieci.

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 [23] 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 [49] 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 [45] 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 [53] 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 [29], 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.

Na popularność tematu zimnych startów wśród społeczności wskazują także Nazari i inni [55]. Podkreślają oni wpływ tego problemu na wydajność, gdzie często czas inicjalizacji może przewyższać czas potrzebny na wykonanie logiki biznesowej. W ramach przeglądu literatury zauważają, że poprawa zimnych startów to jeden z kierunków rozwoju platform bezserwerowych.

AWS Lambda wyróżnia się na tle innych platform FaaS w kontekście zimnych startów. Yu i inni autorzy [73] podkreślili różnicę w technologiach używanych przez dostawców funkcji bezserwerowych. Funkcje AWS Lambda uruchamiane są z użyciem mikro maszyn wirtualnych Firecracker. Pozwala to na osiągnięcie lepszych zimnych startów niż występujące w innych platformach (np. Google Cloud Platform). Autorzy zauważają jednak, że zimne starty dalej znacząco wydłużają czas odpowiedzi. Wynika to z potrzeby pobrania kodu (np. z S3) oraz inicjalizacji funkcji.

Język programowania

Znaczące różnice w wydajności pomiędzy językami w AWS Lambda podkreślili Jackson i Clynch [41]. 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 [25] 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 [64], 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

Drugim pytaniem badawczym rozpatrzonym w przeglądzie literatury jest „Jakie są istniejące metody optymalizacji wydajności funkcji AWS Lambda działających w ekosystemie Java?". W podrozdziale skupiono się na analizie aktualnego poziomu wiedzy na temat metod optymalizacji wydajności funkcji AWS Lambda w kontekście środowiska Java. Pozwoli to na identyfikację obszarów, które nie zostały jeszcze dokładnie zbadane i wymagają dalszych badań.

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 [58] 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 [56]. 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 [22]. 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 [27] 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 [45], gdy uruchomiony już kod dalej jest gotowy do działania. Taktyką z tym

związaną jest regularne uruchamianie funkcji zaproponowane przez [52] 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 [43] 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 [34]. 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

Jedną z metod optymalizacji nowoczesnych języków programowania jest JIT (ang. just-in-time compilation), czyli metoda kompilacji fragmentów kodu do kodu maszynowego, bezpośrednio przed ich wykonaniem. Metoda ta jest często wykorzystywana przez maszyny wirtualne Javy. Jednak według Carreira i innych autorów [19], technika ta jest niewystarczająco wspierana w funkcjach AWS Lambda. W ramach pracy zademonstrowano problem ciepłych startów, które uruchamiają niezoptymalizowany kod, co prowadzi do pogorszenia wydajności.

Autorzy zaproponowali platformę IGNITE, która wprowadza pojęcie gorących startów. Są to uruchomienia funkcji z użyciem dodatkowo zoptymalizowanego kodu (poprzez JIT). W ramach badania, stworzona została alternatywna platforma funkcji, oparta o kontenery Docker. Gorące starty były możliwe poprzez ponowne użycie, już uruchomionych wcześniej kontenerów, zawierających zoptymalizowany kod.

Zaproponowana metoda prowadziła do znaczącej redukcji czasu wykonania funkcji (nawet 55-krotnego dla Javy). Dodatkowo, widoczny był trend coraz lepszej poprawy wydajności, wraz z kolejnymi uruchomieniami funkcji.

GraalVM

Innym podejściem do optymalizacji funkcji Java w AWS Lambda jest wykorzystanie GraalVM. GraalVM to wysokowydajny zestaw JDK (ang. Java Development Kit), który może przyspieszyć działanie aplikacji opartych na technologii Java. Umożliwia kompilację AOT (ang. ahead-of-time) kodu Java do natywnego obrazu, który następnie uruchamiany jest niemal natychmiast oraz zużywa mało zasobów pamięci.

GraalVM to jedna z technik zaproponowanych przez Menendez i Bartlett [54]. Autorzy wykazali, że użycie tej metody pozwoliło na przyspieszenie zimnych startów o 83% oraz opóźnienia dla rozgrzanych funkcji o 55%. Zaznaczyli oni jednak, że użycie GraalVM może być bardziej skomplikowane w porównaniu do klasycznej maszyny wirtualnej Javy. Wynika to z konieczności użycia niestandardowych środowisk uruchomieniowych (ang. custom runtimes) w ramach AWS Lambda, które muszą zostać skonfigurowane przez programistę.

Optymalizacji z użyciem GraalVM oraz różnych frameworków Javy dokonał także Ritzal [63]. W ramach pracy autor przeanalizował użycie frameworków SpringBoot, Micronaut oraz Quarkus, działających zarówno w ramach klasycznego JVM oraz GraalVM. Dodatkowo, przetestowana została funkcja bazowa bez użycia żadnego z frameworków.

Wyniki eksperymentu pokazały, że w przypadku użycia zwykłego JVM funkcja bazowa charakteryzowała się najszybszymi zimnymi startami. Funkcje oparte o frameworki były znacząco wolniejsze. Jednak podczas użycia GraalVM, funkcja oparta o Micronaut posiadała najniższe opóźnienia podczas zimnych startów. Jednocześnie, charakteryzowała się ona niskim zużyciem pamięci.

SnapStart

SnapStart to funkcjonalność oferowana przez AWS w celu łagodzenia problemu zimnych startów dla funkcji napisanych w Javie. Jest to jedna z technik zaproponowanych przez Menendeza i Bartletta w celu optymalizacji funkcji AWS Lambda opartych o Javę [54]. Autorzy wykazali, że włączenie tej opcji w testowanym systemie pozwoliło na przyspieszenie

zimnych startów o 16% oraz zmniejszenie opóźnień dla rozgrzanych funkcji o 21%. Zaznaczyli oni jednak, że SnapStart posiada istotne ograniczenia, takie jak brak wsparcia dla architektury ARM64, niestandardowych środowisk uruchomieniowych (ang. custom runtimes), integracji z Amazon EFS czy możliwości połączenia funkcji z VPC.

2.4.3. Cechy rozwoju aplikacji w AWS Lambda

W podrozdziale rozpatrzono prace poruszające trzecie pytanie badawcze „Jakie są cechy rozwoju aplikacji w architekturze bezserwerowej AWS Lambda?". Skupiono się aspektach, które muszą zostać poruszone przez programistę podczas tworzenia systemów w architekturze bezserwerowej. Identyfikacja tych cech pozwoli później na dokładniejszą analizę wpływu konkretnych metod optymalizacji wydajności na te cechy.

Zarządzanie wielkością funkcji

Zbyte duże rozdrobnienie funkcji może powodować wiele pośrednich problemów, które zostały przedstawione przez Nupponen i Taibi [56]. Na bazie doświadczeń programistów, zebrali oni najczęstsze problemy związane z rozwojem oprogramowania w aplikacjach bezserwerowych. Bezpośrednio wskazują oni na problem zbyt wielu funkcji, który może utrudnić utrzymanie i zrozumienie systemu. Problemem jest także komunikacja między funkcjami, która może być szczególnie wymagana przy dużej ich liczbie. Autorzy podkreślają, że „asynchroniczne wywołania do i pomiędzy funkcjami bezserwerowymi zwiększają złożoność systemu" [56], a problemem bezpośrednich wywołań jest „Złożone debugowanie, luźna izolacja funkcji. Dodatkowe koszty, jeśli funkcje są wywoływane synchronicznie, ponieważ musimy płacić za dwie funkcje działające w tym samym czasie." [56] W przypadku zbyt dużych funkcji, problematyczne staje się także użycie zbyt dużej liczby bibliotek, co zwiększa ryzyko przekroczenia limitu wielkości artefaktu funkcji.

W kolejnej pracy Taibi kontynuuje analizy w tym zakresie wraz z Kehoe i Poccia [66]. Wykonali oni badanie ankietowe wśród doświadczonych praktyków pracujących z aplikacjami w architekturach bezserwerowych. Aż 38.46% badanych wskazało, że synchroniczne zapytania między funkcjami mają znaczny negatywny wpływ na stan aplikacji. Także około co 5. badany wskazał, że złą praktyką jest także dzielenie tego samego kodu między wieloma funkcjami. Same funkcje powinny być skupione wyłącznie na pojedynczym zadaniu biznesowym.

Jak wykazali Eismann i inni [30] systemy składające się powyżej 5 funkcji są bardzo rzadkie. W ich badaniach aż 82% analizowanych przypadków użycia zawierało pięć funkcji lub mniej, a 93% mniej niż dziesięć. Według autorów wynika to potencjalnie z dwóch głównych czynników:

> „Po pierwsze, bezserwerowe modele aplikacji zmniejszają ilość kodu, który programiści muszą napisać, ponieważ pozwalają im skupić się na logice biznesowej (...). Po drugie, wydaje się to wskazywać, że programiści wybierają obecnie raczej dużą ziarnistość dla rozmiaru funkcji bezserwerowych" [30]

Eismann i inni autorzy podkreślają, że optymalizacja wielkości funkcji jest interesującym tematem do dalszych badań.

W badaniach Leitnera i innych [51] potwierdzono, że aplikacje bezserwerowe zazwyczaj składają się z niewielkiej liczby funkcji. Aż 64% badanych wskazało, że ich aplikacje zawierają od 1 do 10 funkcji, co jest zbliżone do wyników Eismanna. W kontekście granularności funkcji, najczęściej stosowaną praktyką była drobna granularność, z funkcjami przypisanymi do pojedynczych metod REST (36% badanych). Jednak większa liczba funkcji może prowadzić do problemów z zarządzaniem, testowaniem i brakiem odpowiednich narzędzi do monitorowania, co stanowi wyzwanie w praktyce.

Integracja z zewnętrznymi usługami

Rozwój aplikacji w modelu serverless opiera się w dużej mierze na integracji z innymi usługami. Wynika to znacznie z bezstanowości AWS Lambda, które uruchamiane są na żądanie. Konieczność integracji podkreślają autorzy Ivanon i Petrova [39]. Pozwala to budować kompletne systemy w oparciu o funkcje AWS Lambda oraz zarządzane serwisy chmurowe. Kluczową rolę odgrywają tutaj różnego rodzaju wyzwalacze zdarzeń (na przykład zmiany danych w Amazon S3 czy DynamoDB) i wiadomości przesyłane przez Amazon SNS. AWS Lambda integruje się także z usługami takimi jak API Gateway, EventBridge czy Step Functions. Pozwala to budować pełne aplikacje bez własnej infrastruktury. Autorzy wskazują, że rozwój aplikacji serverless opiera się na łączeniu funkcji z zarządzanymi usługami.

Bezstanowość funkcji AWS Lambda wymusza integrację z zewnętrznymi serwisami, takimi jak bazy danych czy systemy plików. Ghosh i inni [36] zwracają uwagę, że taka komunikacja odbywa się po sieci i może znacząco zwiększać opóźnienia działania aplikacji. W ich badaniach dostęp do Amazon DynamoDB z funkcji Lambda był prawie czternastokrotnie wolniejszy niż dostęp do lokalnej bazy danych z tradycyjnej aplikacji. Problem ten narasta w bardziej złożonych systemach, gdzie sieciowe opóźnienia sumują się na ścieżce krytycznej. Autorzy podkreślają, że problem wynika z samej architektury serverless, gdzie funkcje są uruchamiane w izolacji. Mimo wad związanych z integracjami z serwisami zewnętrznymi, pozostają one konieczne w rozwoju systemów bezserwerowych.

Na powszechność integracji z serwisami chmurowymi wskazali również Eismann i inni [30]. Dokonali oni analizy przypadków użycia serverless z projektów otwartoźródłowych, białej i szarej literatury oraz konsultacji z ekspertammi. Wykazali oni, że aplikacje najczęściej korzystały z zewnętrznych rozwiązań przechowywania danych (np. Amazon S3) i baz danych (np. DynamoDB). Było to odpowiednio 61% i 47% analizowanych systemów. Użycie takich usług było spodziewane ze względu na bezstanowy charakter usług FaaS. Interesującym wynikiem badania jest, że jedynie 18% funkcji korzysta z rozwiązań API Gateway (bramka API).

Odpowiedni język programowania

Usługa AWS Lambda oferuje wsparcie dla różnych języków programowania. Ich odpowiedni wybór może być kluczowy ze względu na np. wydajność. Kluczową rolę tej decyzji w cyklu rozwoju oprogramowania wskazali Raza i inni autorzy [61]. Skupili się oni na modelu FaaS z perspektywy programisty, rozwijającego systemu w oparciu o te usługi. Decyzja o wyborze języka została uznana przez nich jako jednorazowa. Jej wybór może być podyktowany na przykład kosztem funkcji czy wymaganą wydajnością. Zaznaczają oni jednak, że zmiana

„wiązałaby się ze znacznymi kosztami rozwoju i wdrożenia, dlatego deweloper może podjąć taką decyzję tylko raz w cyklu życia aplikacji" [61]. Zmiana innych parametrów (jak np. wielkości pamięci) ma mniejsze konsekwencje i według programista dokonuje ich bez większego wysiłku. Mogą one jednak wymagać utworzenia odpowiedniego sposobu ich kontroli i aktualizacji, opartego o analizę ich wpływu na wydajność.

Bardsley, Ryan i Howard [15] podkreślają, że wybór języka programowania wpływa bezpośrednio na późniejszy rozwój aplikacji. Odpowiednie decyzje mogą ograniczyć opóźnienia i poprawić wydajność systemu. Autorzy wskazują, że różne komponenty aplikacji mogą używać różnych języków, w zależności od charakterystyki wywołań funkcji i wymagań wydajnościowcyh. W funkcjach reagujących na działania użytkownika (na przykład wykonanie akcji na stronie, jak kliknięcie przycisku) lepiej sprawdzają się języki interpretowane. Mogą to być na przykład Python czy Node.js, które szybciej inicjalizują kontener. Funkcje przetwarzające duże ilości danych lub działające w tle mogą natomiast korzystać z języków kompilowanych, takich jak Java czy C#. W takim podejściu każdy element systemu optymalizowany jest indywidualnie. Pozwala to zmniejszyć czas odpowiedzi tam, gdzie jest to kluczowe oraz zwiększyć wydajność tam, gdzie operacje są bardziej kosztowne.

Testowanie

Narzędziami, które mogą znacząco poprawić doświadczenie programistów pracujących z technologiami bezserwerowymi są różnego rodzaju frameworki. Ich analizy dokonali Skrzypek i Kritikos [48], którzy w swoim przeglądzie wskazali na kluczowe cechy tych narzędzi wspierające zespoły programistyczne. Zwrócili oni uwagę na wsparcie dla testowania, poprzez lokalne lub zdalne wywołanie wykonanie funkcji. Tylko niektóre narzędzia w ograniczony sposób wspierały testy jednostkowe (np. framework „Fn" dla Javy i JavaScript oraz framework „Serverless" dla Javascript). W narzędziach brakuje jednak wsparcia dla testów integracyjnych.

Chatley i Allerton [22] w swojej pracy nad frameworkiem Nimbus również mocno podkreślają, że testowanie jest jednym z głównych wyzwań w trakcie tworzenia aplikacji bezserwerowych. Autorzy na bazie badań ankietowych i własnych doświadczeń projektowych stwierdzają, że aktualne metody testowania są często trudne, powolne i kosztowne. Framework Nimbus wprowadza możliwość uruchomienia kompletnej aplikacji w lokalnym środowisku, które symuluje docelową infrastrukturę chmurową. Pozwala to na wykonanie testów integracyjnych dla funkcji w języku Java.

Testowanie było także cechą podejścia bezserwerowego, wskazanego przez Cavalheiro i Schepke [21]. Dokonali oni implementacji aplikacji z użyciem narzędzi AWS Lambda, Chalice (framework oparty o AWS Lambda) oraz biblioteki Flask (działającej lokalnie, jako podejście tradycyjne). Zwrócili oni uwagę na o wiele łatwiejsze testowanie z użyciem Flask. Umożliwiał on rozwój testów jednostkowych API, co było utrudnione w przypadku modelu serverless. W przypadku rozwoju AWS Lambda i Chalice trudności sprawiał także proces debuggowania.

Znaczenie testów jednostkowych podkreśla także Leitner i inni autorzy [51]. Poprzez wykonane badania ankietowe aż 87% badanych programistów wykorzystuje ten rodzaj testów w trakcie rozwoju oprogramowania serverless. Także 55% uważa, że aktualnie dostępne narzędzia są niewystarczające w obszarach jak testowanie czy wdrożenie.

2.4.4. Podsumowanie wyników przeglądu

W ramach podrozdziału zaprezentowano wnioski do wyników przeglądu literatury. Zostały one przedstawione w kontekście każdego pytania badawczego.

Wnioski odnośnie PB1: „Jakie są główne czynniki wpływające na wydajność funkcji AWS Lambda?"

Przegląd literatury pozwolił na identyfikację czynników i ich wpływu na wydajność funkcji AWS Lambda. Zauważono, że wiele z nich może być w łatwy sposób zaprogramowana przez twórcę funkcji. Takimi czynnikami są pamięć funkcji czy architektura procesora. Dzięki naturze usług FaaS konfiguracja infrastruktury jest bardzo prosta, a parametry te mogą być konfigurowane w trakcie całego cyklu rozwoju oprogramowania. Istotny wpływ na czas działania funkcji ma także język programowania w którym została ona stworzona. Bardzo szerokim obszarem badań są również zimne starty, które wpływają na czas wykonania funkcji. Warto jednak zaznaczyć, że na zimne starty wpływ wywiera wiele innych czynników.

Pierwszym ważnym aspektem jest wielkość pamięci funkcji. Został on poruszony w pięciu badanych pracach [45] [59] [33] [57] [24]. Popularność badań pod tym względem może wynikać z kilku czynników. Wpływ wielkości pamięci na wydajność oprogramowania jest dość oczywisty niezależnie od modelu w jakim ono działa (serwerowe lub bezserwerowe). Jednak w przypadku architektur serverless, wielkość pamięci jest bezpośrednio powiązana z kosztem działania. Naturalnie zachęca to do badań nad optymalizacją tego parametru. Dodatkowo, pamięć to czynnik, który jest bardzo prosto konfigurowalny przez programistę.

W badaniach wykazano, że pamięć oprócz wpływu na czas działania, ma także wpływ na czynniki, które nie są dostępne do konfiguracji (jak wydajność I/O czy sieciowa). Z analizy tego obszaru wynika, że w przypadku analizy wydajności AWS Lambda, bardzo istotnym elementem jest badanie działania funkcji dla różnych wielkości pamięci.

Mimo wystąpienia wyłącznie w dwóch pracach [23] [49] architektura procesora także jest interesującym aspektem do uwzględniania w badaniach. Niemożliwe jest jednak stwierdzenie, że któraś z dostępnych architektur (ARM64 i X86_64) pozwala na osiągnięcie lepszej wydajności funkcji. Optymalizacja efektywności AWS Lambda pod względem tego parametru wymaga konkretnej analizy danego przypadku i testów obu opcji.

Popularną strefą badań są także języki programowania dostępne dla AWS Lambda [41] [25] [64]. Autorzy często porównują szybkość języków kompilowanych (jak Java) i interpretowanych (jak Python). W tym zakresie wyniki często różnią się w zależności od użycia dla ciepłych i zimnych startów. Mimo badań nad wieloma językami programowania, interesujący jest jednak brak reprezentacji innych niż Java języków działających w JVM. Mogą to być na przykład Kotlin, Scala czy Groovy.

Języki wywodzące się z Javy (jak Kotlin) mogą być interesującym obszarem dalszych badań w kontekście wydajności. Każdy z tych języków posiada specyficzne aspekty, które mogą wpłynąć na wydajność. Dodatkowo, ich kompatybilność z Javą może usprawnić ich użycie przez istniejące już zespoły zaznajomione z tym językiem.

Bardzo szerokim obszarem badań są także zimne starty. Polegają one na inicjalizacji infrastruktury potrzebnej do uruchomienia kodu wywołanej funkcji, co powoduje dodatkowe opóźnienia. Wpływają one bezpośrednio na wydajność funkcji, a wpływ na zimne starty mają

także pozostałe czynniki przedstawione we wnioskach. Wynika z tego, że eksperymenty z użyciem metod poprawy wydajności powinny uwzględniać zarówno ciepłe, jak i zimne starty.

Wnioski odnośnie PB2: „Jakie są istniejące metody optymalizacji wydajności funkcji AWS Lambda działających w ekosystemie Java?"

Dokonanie przeglądu literatury pozwoliło na poznanie aktualnego stanu wiedzy w kontekście metod optymalizacji wydajności. Ważnym elementem poprawy wydajności jest zmniejszanie wielkości artefaktu, które może być dokonywane w całym procesie rozwoju oprogramowania. Składa się na niego wiele czynników (jak np. użyte biblioteki), co daje szeroką możliwość optymalizacji tego aspektu. W literaturze zawarte występują także metody, które nie są specyficzne dla Javy jak odpowiedni rodzaj wdrożeń (ZIP lub Docker) czy pingowanie. W ramach odpowiedzi na pierwsze pytanie badawcze wykazano znaczący wpływ języka programowania na wydajność, zatem ciekawym kierunkiem w literaturze jest propozycja użycia języków alternatywnych do Javy. Nie istnieje jednak wiele badań na temat bardzo skutecznych metod opierających się o usprawnienia kompilacji (jak techniki JIT czy GraalVM) oraz SnapStart.

Redukcja wielkości artefaktu może być dokononana poprzez różne techniki jak redukcja nieużywanego kodu, czy zmniejszenie liczby bibliotek. Jest ona wykonywana w celu poprawy wydajność oraz uniknięcia sytuacji przekroczenia limitu wielkości funkcji. Jednak ważnym aspektem tej metody jest stronienie od nadmiernego podziału funkcji. Zbyt duża ich liczba może prowadzić do wydłużenia czasu działania, ze względu na kosztowne zapytania między funkcjami. Dlatego każda optymalizacja powinna być dokładnie analizowana w zależności od optymalizowanego komponentu.

Znaleziono wyłącznie pojedynczą pracę porównującą wdrożenia oparte o ZIP i obrazy Docker [27]. Na jej bazie możemy jednak stwierdzić o skuteczności użycia ZIP jako metody optymalizacji sprawności AWS Lambda dla technologii Java. Technika ta może być jednak użyta w kombinacji z nowymi metodami.

Bardzo interesującym kierunkiem badań są techniki oparte o użycie innych języków. Na ten temat zawarto w wynikach dwie prace. Pierwsze z nich, proponuje użycie JavaScript jako całkowite zastąpienie Javy [34]. Metoda ta jednak nie jest rozwiązaniem dla programistów Javy i trudno nazwać ją metodą optymalizacji wydajności w ekosystemie Java. Wskazuje ona jednak wartościowy trend dla użycia odpowiednich języków w konkretnych elementach systemu.

Druga praca proponuje migracje aplikacji monolitycznych z Javy do języka JavaScript [43]. Stworzone narzędzie pozwoliło na migrację starych systemów do architektury bezserwerowej. Technika ta nie pozwala jednak na rozwój aplikacji opartych o Javę, a docelowo zastępuje ją JavaScriptem. Jednak ponownie, praca ukazuje ciekawy obszar do dalszych badań.

Bardzo skuteczne okazują się metody optymalizujące kompilację i uruchomienie kodu Java. Techniki jak JIT [19] czy GraalVM [54] [63] pozwalają na znaczące zmniejszenie opóźnień. Jednak liczba prac jest niewielka w porównaniu z skutecznością tego rozwiązania. Dodatkowo, prace skupiały się wyłącznie na wpływie na czas działania funkcji pomijając inne czynniki (np. czas budowania artefaktu). Z tego względu to pole może wymagać dokładniejszej analizy.

Niespodziewanym było znalezienie wyłącznie pojedynczego badania w kontekście SnapStart [54]. Metoda ta jest dostępna od roku 2022, a jej wdrożenie przez programistę jest stosunkowo proste. Sama technika nie niesie za sobą także dodatkowych kosztów, a pozwala na zmniejszenie

wpływu zimnych startów na efektywność AWS Lambda. Jest ona także dostępna dla innych języków działających w JVM, co nie zostało zbadane. Zatem obszar ten może wykazywać potencjał do dalszych badań.

Wnioski odnośnie PB3: „Jakie są cechy rozwoju aplikacji w architekturze bezserwerowej AWS Lambda?"

W trakcie rozwoju aplikacji w architekturze bezserwerowej od programisty wymagane są konkretne decyzje i działania. W celu optymalizacji architektury istotne jest tworzenie funkcji odpowiedniego rozmiaru (co także znacząco wpływa na wydajność). Systemy często oparte są o dodatkowe serwisy (np. bazodanowe), co dodatkowo utrudnia kolejny aspekt tej architektury, czyli testowanie. W przypadku testowania funkcji bezserwerowych dominują testy jednostkowe, a testy integracyjne są utrudnione. Ważnym elementem tworzenia nowego komponentu jest wybór języka, a jego twórca musi dostosować tę decyzję do konkretnych wymagań.

Tworzenie systemów w architekturze opartej o AWS Lambda cechuje się odpowiednią kompozycją tych funkcji. Ich wielkość oprócz znaczącego wpływu na wydajność ma także wpływ na rozwój systemu [56] [66]. Wynika z tego, że w badania nad metodami optymalizacji wydajności muszą zostać przeprowadzone w kontekście odpowiednich funkcji. Powinny być one nie za duże, a skupione na konkretnym zadaniu biznesowym.

Kluczową rolę w modelu bezserwerowym odgrywają zewnętrzne serwisy. Funkcje FaaS w swojej naturze są bezserwerowe, dlatego bardzo istotne jest użycie usług jak bazy danych [30]. Element ten jest istotny ze względu na potrzebę posiadania odpowiednich narzędzi do takich integracji. Mogą to być różne narzędzia SDK (ang. Software Development Kit), na przykład od integracji z usługami AWS. Z tego względu metody optymalizacji nie mogą ograniczać tak krytycznych zależności.

Fundamentalnym wyborem dla tworzonej funkcji AWS Lambda jest język programowania. W ramach poprzednich pytań badawczych wykazano ich znaczący wpływ na wydajność. Ma on także znaczny udział w rozwoju oprogramowania opartego o model serverless. Autorzy prac zaznaczają, że musi być on odpowiedni dla konkretnych wymagań biznesowych [61] [15]. Dlatego interesującym obszarem badań mogą być metody, które ułatwią użycie różnych języków w ramach AWS Lambda.

Trudności w procesie progamistycznym aplikacji opartych o AWS Lambda przynosi testowanie oprogramowania. Aktualnie testy funkcji opierają się głównie o testy jednostkowe, które są wspierane przez różne frameworki [48]. Brakującym elementem są testy integracyjne, które w tym modelu są mocno skomplikowane. Dlatego metody poprawy wydajności nie powinny utrudniać testowania, które jest już teraz jest wyzwaniem.

3. Wybrane metody optymalizacji

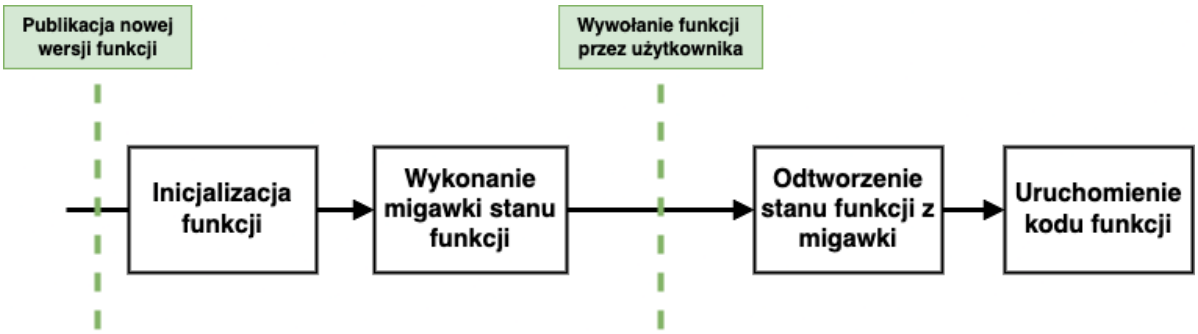
W ramach rozdziału dokonano charakteryzacji wybranych metod optymalizacji, które zostały zawarte w pracy. Wybór metod podyktowany był wynikami przeglądu literatury, na podstawie którego do badań włączono metody SnapStart i GraalVM. Według analizowanych badań techniki te były skuteczne, jednak ich reprezentacja w dotychczasowych publikacjach była niewielka. Kluczowym elementem rozdziału są nowe metody optymalizacji, które zostały zaproponowane w ramach pracy. Strategie te opierają o język Kotlin, który wybrano ze względu na brak prac poruszających inne języki ekosystemu Java niż sama Java (np. Kotlin, Scala, Groovy). Kotlin cechuje się różnymi potencjalnymi zastosowaniami, na bazie których wybrano trzy podejścia, które zostały zaproponowane jako nowe metody optymalizacji. W opisie każdej metody przedstawiono sposób jej działania, a także jej zalety i wady, co pomoże w odpowiedzi na trzeci pytanie badawcze sformułowane w pracy.

3.1. SnapStart

Jednym z istotnych czynników wpływających na wydajność funkcji AWS Lambda implementowanych w ekosystemie Java jest zjawisko tzw. zimnego startu. Wynika on z cyklu życia funkcji i etapu inicjalizacji (co zostało opisane w Rozdziale 1.3). W etap ten wchodzą procesy takie jak inicjalizacja maszyny wirtualnej Java czy uruchomienie statycznego kodu inicjującego [13]. W przypadku Javy zajmuje to więcej czasu niż dla innych języków (jak Python), co wydłuża zimne starty [41]. Znacząco oddziałuje to na wydajność funkcji, a może być szczególnie dotkliwe dla serwisów o niewielkiej aktywności. W odpowiedzi na potrzebę minimalizacji tych negatywnych skutków Amazon Web Services wprowadziło mechanizm znany jako AWS Lambda SnapStart. W ramach podrozdziału podjęto analizę tego rozwiązania w kontekście jego działania, zalet oraz ograniczeń.

Mechanizm SnapStart istotnie modyfikuje tradycyjny cykl życia funkcji AWS Lambda. Zasadnicza różnica polega na przeniesieniu kosztownego etapu inicjalizacji z momentu pierwszego wywołania funkcji na etap jej publikacji [9]. Oznacza to, że inicjalizacja funkcji nie jest wykonywana w momencie zapytania użytkownika (co wywołuje zimny start), lecz w momencie wgrania nowej wersji funkcji (oraz kodu) przez programistę. Inicjalizacja ta zawiera najdłuższe operacje dla rozwiązań Java jak utworzenie maszyny wirtualnej, załadowanie klas, czy wykonanie kodu inicjalizującego. Następnie, tworzona jest zaszyfrowana „migawka” (ang. snapshot) stanu pamięci i dysku w pełni gotowego środowiska wykonawczego. Gdy funkcja jest następnie wywoływana po raz pierwszy, nie zachodzi już standardowy zimny start. Zamiast tego środowisko jest odtwarzane z utworzonej migawki, co zostało przedstawione na Rysunku 3.1. Według dostawcy AWS metoda ta w optymalnych scenariuszach zmniejsza opóźnienie z kilku sekund do mniej niż sekundy [9].

Działanie tej metody jest technicznie możliwe dzięki użyciu środowiska wirtualizacji przez AWS Lambda. Jak opisują Agache i inni autorzy [4] usługa Lambda do izolacji poszczególnych



Rysunek 3.1: Proces uruchomienia AWS Lambda z użyciem metody SnapStart [źródło: opracowanie własne]

funkcji wykorzystuje dedykowane maszyny wirtualne typu microVM. Są one zarządzane przez lekki monitor maszyn wirtualnych (ang. VMM) o nazwie Firecracker. Posiada on cechy, które były kluczowe w minimalizacji problemu zimnego startu. Po pierwsze, celowo rezygnuje on z emulacji zbędnych urządzeń (jak emulacja systemu BIOS czy rozbudowanych kontrolerów PCI) [4]. Zmniejsza to złożoność i rozmiar stanu każdej maszyny wirtualnej. Dzięki temu wykonanie i odtworzenie migawki jest łatwiejsze. Po drugie, Firecracker jest w pełni kontrolowany przez interfejs REST API [4]. Umożliwia to precyzyjne zarządzanie całym cyklem życia każdej maszyny wirtualnej, włączając w to jej konfigurację, uruchomienie oraz zatrzymanie. Pozwala to na określenie fazy inicjalizacji funkcji oraz wykonanie migawki w odpowiednim momencie. Istotna jest również zapewniona przez Firecracker izolacja [4], co gwarantuje bezpieczeństwo tworzenia i odtwarzania migawek.

Samo użycie metody SnapStart jest bardzo proste i nie wymaga od programisty dużego nakładu pracy. Włączenie rozwiązania wymaga jedynie ustawienia odpowiedniej opcji podczas konfiguracji funkcji [9]. Nie oznacza to jednak, że SnapStart jest odpowiedni dla wszystkich funkcji. AWS podkreśla dwa typy aplikacji, które znacząco zyskają poprzez użycie SnapStart [9]. Są nimi wrażliwe na opóźnienia interfejsy API i potoki przetwarzania danych. Dodatkowo, metoda ta niesie za sobą pewne ograniczenia, które muszą zostać uwzględnione przed jej wdrożeniem.

Pierwszym aspektem jest kwestia unikalności stanu w funkcjach wykorzystujących SnapStart. Jak analizują Brooker i inni autorzy [18], klonowanie migawek wprowadza fundamentalne wyzwanie związane z przywróceniem unikalności maszyn wirtualnych, co jest niezbędne do poprawnego generowania unikalnych identyfikatorów czy sekretów kryptograficznych. Migawka zainicjowanego środowiska wykonywana jest jednorazowo, a następnie używana podczas wielu wywołań funkcji. Może to stanowić duże zagrożenie dla programisty AWS Lambda, gdy potrzebuje on generować unikalne wartości jak identyfikatory (np. UUID) czy jednorazowe tokeny. Narusza to znacznie poprawność logiki aplikacji oraz jej bezpieczeństwo. Jedną z metod naprawy tego problemu jest generowanie wartości losowych wyłącznie w metodzie wywołującej funkcje (zamiast w bloku statycznym kodu) [9]. Dodatkowo, ewentualne problemy z unikalnością funkcji SnapStart mogą zostać wykryte poprzez oprogramowanie SpotBugs [67]. Narzędzie wykonuje statyczną analizę kodu, walidując go poprzez reguły zapewnione przez AWS. Pozwala to programiście wykryć, a następnie naprawić fragmenty kodu powodujące problem z unikalnością.

Kolejnym istotnym wyzwaniem podczas rozwoju aplikacji z technologią SnapStart jest zarządzanie połączeniami sieciowymi [9]. Połączenia nawiązane z zewnętrznymi usługami są standardową praktyką podczas tworzenia aplikacji AWS Lambda [30] [39]. Problematyczne stają się jednak te połączenia sieciowe, które nawiązano podczas inicjalizacji funkcji. Ponieważ inicjalizacja odbywa się przed faktycznym przetworzeniem żądania użytkownika, upływający czas może sprawić, że w momencie odtworzenia funkcji połączenia te nie będą już aktywne. Praktyką zalecaną przez AWS jest ponowne nawiązywanie lub dokładna walidacja istniejących połączeń [9]. Powinno to być wykonane bezpośrednio w metodzie wywołującej funkcje lub z wykorzystaniem metody „afterRestore". Metoda ta jest wywoływana bezpośrednio po odtworzeniu migawki stanu funkcji.

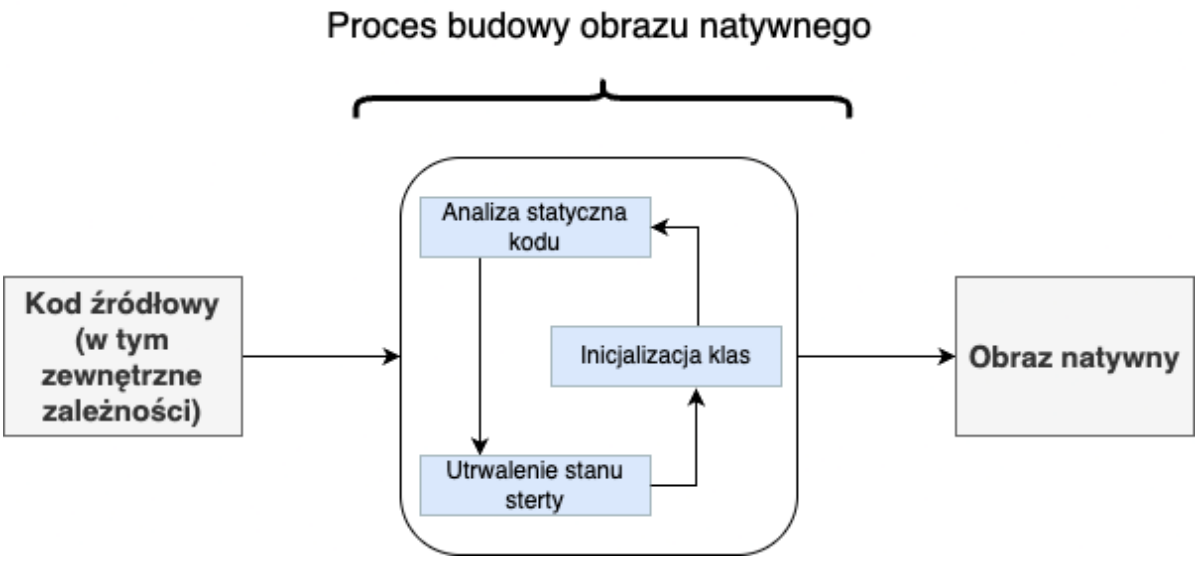
Strategicznym czynnikiem usług bezserwerowych są koszty, zatem powinny być one uwzględnione także przed użyciem SnapStart. Zgodnie z dokumentacją Amazon Web Services [9], użycie SnapStart dla środowisk uruchomieniowych Java nie wiążą się z dodatkowymi kosztami. Koszt wykonania funkcji z włączonym SnapStart nadal bazuje na standardowych rozliczeniach. Składa się na niego liczba przetworzonych żądań oraz łączny czas trwania wykonań.

Podsumowując, mechanizm SnapStart stanowi prostą w aktywacji metodę redukcji czasu zimnych startów. Sam mechanizm opiera się na wcześniejszym wykonaniu fazy inicjacji funkcji, a następnie wykonania migawki stanu. W momencie wywołania funkcji stan ten może zostać odtworzony. Znaczącą korzyścią metody jest brak dodatkowych kosztów. Wiąże się ona jednak z istotnymi utrudnieniami (jak zarządzanie połączeniami sieciowymi i problem z unikalnością stanu). Powinny być one uwzględnione przez programistę przed użyciem narzędzia.

3.2. GraalVM

Ważnym obszarem badań nad optymalizacją Javy i jej użycia w AWS Lambda, są technologie pozwalające na zmianę sposobu kompilacji i uruchamiana aplikacji. Jedną z technologii, które zyskusje na popularności w tym zakresie, jest GraalVM. Jest to możliwe m.in. dzięki użyciu kompilatora JIT (ang. Just-In-Time) w połączeniu z kompilacją AOT (ang. Ahead-Of-Time) [74]. GraalVM oferuje zaawansowaną architekturę pozwalającą na kompilację i uruchomienie aplikacji w postaci obrazów natywnych. Stanowi to alternatywę dla klasycznej maszyny wirtualnej Javy, a dodatkowo skupia się na jej wydajności. Poniższy podrozdział poświęcono analizie działania omawianego rozwiązania, jego zalet i słabych stron.

Kluczowym mechanizmem GraalVM jest kompilacja AOT (ang. Ahead-Of-Time) do postaci tzw. obrazów natywnych (ang. native images) [74]. Ma to bezpośredni wpływ na wydajność działania aplikacji. W modelu tradycyjnym, kod bajtowy Java jest interpretowany i kompilowany dynamicznie przez maszynę wirtualną w trakcie działania aplikacji. Podejście AOT przenosi znaczną część z tych operacji na etap budowania artefaktu. Istotnym elementem tego procesu jest agresywna, statyczna analiza kodu [75], w celu identyfikacji osiągalnych w trakcie działania części. Pozwala to na odrzucenie nieużywanych fragmentów kodu (np. z używanych bibliotek), co pozwala na zmniejszenie wielkości obrazu natywnego. Aspekt ten może być kluczowy w kontekście AWS Lambda, ze względu na wpływ wielkości artefaktu na wydajność [58]. Po analizie kodu, dokonywana jest inicjalizacja klas, a stan aplikacji, w tym częściowo zainicjalizowana sterta, jest utrwalany. W celu lepszej optymalizacji, operacje te są powtarzane, co zostało przedstawione na Rysunku 3.2.



Rysunek 3.2: Uproszczony proces budowy obrazu natywnego GraalVM [źródło: opracowanie własne]

Jako wynik kompilacji powstaje samodzielny, zoptymalizowany plik binarny. Nie wymaga on do uruchomienia pełnej maszyny wirtualnej Java, a jedynie minimalnego środowiska wykonawczego dostarczanego przez SubstrateVM, będącego częścią GraalVM [74]. Różnica ta ma fundamentalne znaczenie w kontekście wydajności AWS Lambda. Eliminowana jest konieczność wykonywania czasochłonnych operacji typowych dla startu tradycyjnej maszyny wirtualnej Java, takich jak ładowanie klas czy jej inicjalizacja. Wszystkie te zadania zostały już wykonane wcześniej, w procesie budowy obrazu natywnego. Dzięki temu tworzona przez programistę funkcja AWS Lambda nie będzie operować w zarządzanym środowisku Java. Zamiast tego, usługi muszą opierać się o niestandardowe środowiska wykonawcze, oferujące wyłącznie system operacyjny (Amazon Linux 2023 lub Amazon Linux 2) [8]. Ich użycie pozwala także na realizację drugiej zalety GraalVM, czyli redukcji zapotrzebowania na pamięć operacyjną [75].

Pomimo pozytywnego wpływu na wydajność, zastosowanie kompilacji AOT w GraalVM wiąże się także z ograniczeniami. Jednym z nich jest obsługa dynamicznych cech Javy, takich jak refleksja (ang. reflection), dynamiczne proxy, serializacja czy natywny interfejs Java (JNI) Wynika to z faktu użycia agresywnej statycznej analizy kodu. Napotyka ona trudności w przewidzeniu wszystkich dynamicznie ładowanych klas, pól i metod, które nie są jawnie osiągalne w kodzie źródłowym. Problem ten wymaga użycia dodatkowych mechanizmów GraalVM [38]. Polegają one na przygotowaniu dodatkowych metadanych dla klas, co wymaga jednak dodatkowej obsługi.

Kolejnym aspektem, który może negatywnie wpłynąć na rozwój oprogramowania przy użyciu GraalVM, jest czasochłonność procesu kompilacji. Generowanie w pełni zoptymalizowanego obrazu natywnego jest operacją bardziej złożoną niż standardowa kompilacja kodu Javy do postaci bajtowej. W praktyce oznacza to, że proces budowania artefaktu dla funkcji AWS Lambda może trwać odczuwalnie dłużej. Może mieć to znaczący wpływ na rozwój oprogramowania, szczególnie w przypadku częstych iteracji i tworzenia nowych wersji funkcji.

Dłuższy czas kompilacji może także wpłynąć na ogólną efektywność procesów ciągłej integracji i ciągłego dostarczania (ang. CI/CD).

Jednym z sposobów poprawy doświadczeń programistów przy pracy z GraalVM, jest użycie odpowiednich frameworków. Jednym z nich jest Quarkus [75], który został zaprojektowany z myślą o środowiskach chmurowych. Kluczową cechą Quarkusa jest przeniesienie jak największej liczby operacji inicjalizacyjnych i konfiguracyjnych na etap budowania aplikacji. Obejmuje to między innymi wstrzykiwanie zależności, przetwarzanie adnotacji oraz konfigurację rozszerzeń. Dzięki temu, w czasie budowania obrazu natywnego, Quarkus jest w stanie przeprowadzić szczegółową analizę aplikacji. Poprzez użycie odpowiednich annotacji pozwala on na oznaczenie klas niezbędnych dla mechanizmów refleksji czy proxy [60]. Dzięki temu jest on w stanie automatycznie wygenerować niezbędne metadane dla klas. Dane te następnie pozwalają na użycie wspomnianych mechanizmów w GraalVM. Innymi, konkurencyjnymi do Quarkusa frameworkami, które oferują wsparcie dla obrazów natywnych są Helidon i Micronaut [75]. Ich popularność wskazuje na wysokie zainteresowanie takimi technologiami w społeczności programistów Java, dlatego jest to interesujący kierunek rozwoju dla funkcji AWS Lambda.

3.3. Kotlin

W ramach systematycznego przeglądu literatury (przedstawionego w Rozdziale 2) zauważono, że aktualne badania skupiają się wyłącznie na języku Java. Pomijają one jednak inne języki z ekosystemu Java, także oparte o maszynę wirtualną Java (ang. JVM). Tymczasem na popularności zyskują alternatywne języki ekosystemu Javy. Wśród nich na szczególną uwagę zasługuje Kotlin, rozwijany przez firmę JetBrains. Jest on oficjalnie wspierany przez Google jako język programowania dla platformy Android, co wskazuje na jego solidne zastosowanie w tych systemach. Coraz większe uznanie zyskuje także jako działający po stronie serwera. Dlatego język ten jest interesującym obszarem badań w kontekście AWS Lambda. W tym rozdziale przedstawiony zostanie język programowania Kotlin, w tym jego zastosowanie w kontekście funkcji AWS Lambda.

Jednym z kluczowych czynników, które zwiększają popularność Kotlina, jest jego łatwa nauka przez programistów Javy. Dodatkowo, istnieje możliwość łatwej intergracji kodu napisanego w Kotlinie z istniejącym oprogramowaniem Java [1]. Te cechy czynią go interesującym kandydatem do analizy w kontekście optymalizacji wydajności rozwiązań dla usługi AWS Lambda. Dla zespołów programistycznych może stanowić on wartościowe rozszerzenie dotychczasowych możliwości. Kotlin oferuje bowiem alternatywę lub uzupełnienie dla tradycyjnie stosowanej Javy.

Kwestia wydajności Kotlina w porównaniu do Javy jest przedmiotem dyskusji. Jednak badania dotyczą najczęściej ich zastosowań w kontekście aplikacji mobilnych. Gajek i inni autorzy [35] przeanalizowali wydajność obu języków, poprzez użycie gry mobilnej uruchomionej na systemie Android. Wykazali oni, że w testowanym scenariuszu Java osiągnęła nieznacznie lepszą wydajność pod względem zużycia zasobów CPU i RAM. Było to jednak zastosowanie mobilne, a same różnice nie były znaczne. Należy jednak podkreślić, że warunki mobilne mogą być inne niż w systemach działających w usłudze AWS Lambda. Sam język Kotlin posiada mechanizmy, które mogą pozytywnie wpłynąć na wydajność.

Funkcje inline (ang. inline functions) w Kotlinie mogą przyczynić się do redukcji narzutu

wydajności podczas wywołań funkcji. Mechanizm ten polega na wstawieniu kodu ciała funkcji bezpośrednio w miejsce jej wywołania [1]. Jest to wykonywane w momencie kompilacji, a programista może określić, które funkcje powinny być w ten sposób optymalizowane. Eliminuje to koszt ich wywołania, co jest szczególnie przydatne w przypadku małych, często używanych funkcji. Dodatkowo, język pozwala na przekazywanie funkcji jako parametrów, na przykład w kolekcjach i metodach jak filtrowanie. W tych sytuacjach użycie funkcji inline może znacząco zmniejszyć liczbę operacji. Pozytywny wpływ mechanizmu inline został przedstawiony przez Bergstrom i innych autorów [16], gdzie jego użycie zmniejszyło czas wykonywania programów nawet do 8%.

Innym istotnym elementem Kotlina wspierającym wydajność są korutyny (ang. coroutines). Mogą być one użyteczne zwłaszcza w kontekście operacji wejścia-wyjścia (I/O). Systemy oparte o usługę AWS Lambda często integrowane są z zewnętrznymi serwisami (co zostało zauważone w ramach przeglądu literatury w Rozdziale 2). Wymaga to komunikacji opartej o operacje sieciowe. Tradycyjne podejście oparte na wątkach może konsumować dużą ilość zasobów serwera i prowadzić do blokowania wykonania. Korutyny pozwalają na pisanie asynchronicznego, nieblokującego kodu w sposób bardziej sekwencyjny i czytelny [1]. Na lepszą wydajność korutyn w porównaniu z tradycyjnymi wątkami wskazali Beronić i inni autorzy [17].

Implementacja mechanizmów poprawiających wydajność w języku programowania, pozwala następnie na ich użycie w bibliotekach, które są wykorzystywane przez programistów. Język Kotlin oferuje ciekawy ekosystem bibliotek, przeznaczonych na przykład do tworzenia aplikacji działających po stronie serwera. Są to biblioteki jak http4k czy ktor. Ktor to framework zaprojektowany do budowy asynchronicznych aplikacji serwerowych i klienckich, rozwijany przez firmę JetBrains. Jest on oparty w pełni o język Kotlin, a jego kluczową cechą jest natywne wsparcie dla korutyn. Z kolei http4k kładzie nacisk na prostotę i minimalizm. Architektura http4k opiera się na koncepcji funkcji jako podstawowych bloków aplikacji [2], co naturalnie współgra z modelem serverless i AWS Lambda. Samo narzędzie rezyguje z mechanizmów refleksji [2], co może mieć pozytywny wpływ na wydajność.

Rosnące znaczenie Kotlin dostrzega także Amazon Web Service, które oferuje bibliotekę AWS SDK dla Kotlina [10]. Jej celem jest zapewnienie programistom możliwości interakcji z usługami AWS w sposób naturalny dla tego języka. SDK ten został zaprojektowany od podstaw z myślą o Kotlinie, co przejawia się między innymi w wykorzystaniu korutyn do obsługi operacji asynchronicznych.

Duży wpływ na wydajność funkcji AWS Lambda ma wybrany język programowania [41] [25]. Wynika to na przykład z różnych przypadków biznesowych i operacji, które muszą wykonywać. Mimo to, często muszą one dzielić wspólny kod [58], co wskazuje na potrzebę wykorzystania mechanizmów, które to umożliwą. Z tego powodu bardzo interesującą dla AWS Lambda i jej wydajności, może okazać się inicjatywa Kotlin Multiplatform. KMP (Kotlin Multiplatform) to projekt, który powstał w szczególności dla aplikacji mobilnych. Pozwala on na kompilację lub translację tego samego kodu Kotlin do użycia na różnych platformach. Mogą to być na przykład Android, iOS, aplikacje desktopowe (JVM) lub webowe (JavaScript, Web Assembly) [1]. Oferuje to możliwość dzielenia kodu (np. logiki biznesowej) pomiędzy różnymi platformami, jednak przy możliwości zachowania natywnych komponentów widoku.

Mimo głównego przypadku użycia jakim są aplikacje klienckie, Kotlin Multiplatform może być obiecującym rozwiązaniem dla AWS Lambda. Po pierwsze oferuje on możliwość

translacji kodu Kotlin do JavaScript oraz kompilację do natywnych plików binarnych (opcje te zostaną przedstawione jako osobne metody w kolejnych rozdziałach). Pozwala to na ominięcie różnych niedogodności wynikających z użycia maszyny wirtualnej Javy. Jednak zachowane są przy tym zalety języka oraz wspiera to użycie już istniejących umiejętności programistów języków rodziny JVM. Po drugie, kod w KMP może być dzielony pomiędzy platformami. Umożliwia to bardzo elastyczny wybór środowiska uruchomieniowego AWS Lambda w ramach pojedynczego systemu. Jednocześnie, część kodu może być współdzielona między wszystkie funkcje niezależnie od wybranej platformy. Może to na przykład oznaczać, że klasy implementujące pewne struktury oraz zasady wynikające z reguł biznesowych, będą mogły być używane przez funkcje działające zarówno poprzez JavaScript, JVM, jak i natywne pliki binarne.

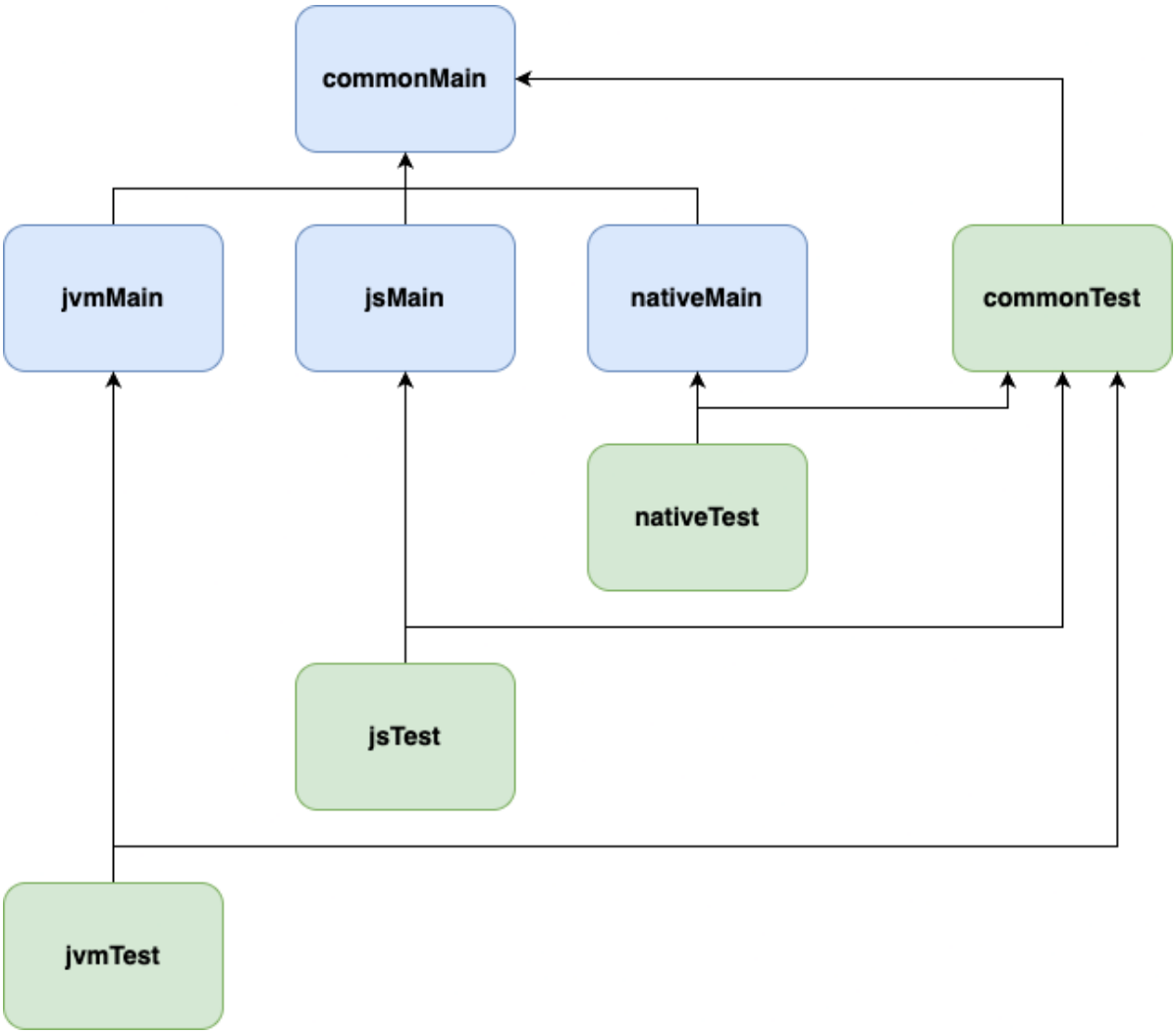
Jednym z czynników, które mogą być modyfikowane już podczas działania usług AWS Lambda jest pamięć. Jej rozmiar może być dostosowywany w zależności od wydajności monitorowanej funkcji. Użycie Kotlin Multiplatform pozwala na rozszerzenie tej metody. W zależności od obserwowanych parametrów (jak czas odpowiedzi lub opóźnienia zimnych startów) możliwe jest ponowne wykorzystanie tego samego kodu i budowa funkcji działającej na innej platformie. Przykładowo, po wdrożeniu funkcji działającej z użyciem JVM, może pojawić się potrzeba redukcji czasu zimnych startów. W takim wypadku Kotlin Multiplatform umożliwia translację kodu do JavaScript, który pozwoli na redukcję czasu inicjalizacji AWS Lambda.

Współdzielenie kodu pomiędzy platformami jest możliwe dzięki strukturze, którą oferuje Kotlin Multiplatform. Została ona opisana przez firmę JetBrains w ramach dokumentacji KMP [42]. Jej głównym elementem jest katalog „commonMain", który jest współdzielony pomiędzy wszystkimi platformami. Kompilator używa kodu współdzielonego jako dane wejściowe, aby w rezultacie utworzyć zestaw plików binarnych specyficznych dla danej platformy. Mogą to być na przykład pliki .class dla maszyny wirtualnej Javy, czy natywne pliki wykonywalne (np. dla platformy Linux).

Następnie programista może utworzyć kolejne katalogi, które będą zawierać kod specyficzny dla docelowych platform. Przykładowa struktura została przedstawiona na Rysunku 3.3, gdzie katalog commonMain jest współdzielony między JVM (jvmMain), JavaScript (jsMain) oraz platformy natywne (nativeMain). Docelowe platformy (ang. targets) są deklarowane w konfiguracji Gradle [42], a kod współdzielony jest przygotowywany wyłącznie dla nich. Katalogi dla docelowych platform są wymagane, gdyż Kotlin nie zezwala na użycie specyficznych elementów danej platformy w katalogu współdzielonym. Przykładem takiego elementu może być klasa „java.io.File", która jest dostępna wyłącznie dla maszyny wirtualnej Javy. Jej użycie w katalogu commonMain spowoduje błąd kompilacji.

Kotlin Multiplatform zawiera także integracje z testami oprogramowania. Jest to szczególnie ważne dla tworzenia oprogramowania z wykorzystaniem AWS Lambda, gdzie testowanie może być skomplikowane (co było jednym z wniosków przeglądu literatury w Rozdziale 2). Testy dla kodu współdzielonego powinny być zapisane w katalogu „commonTest", gdzie programista może użyć biblioteki kotlin.test [42]. Następnie testy są uruchamiane dla każdej docelowej platformy. Programista może także tworzyć przypadki testowe dla konkretnych platform, z użyciem technologii przez nie oferowanych. Następuje tutaj analogiczne współdzielenie kodu jak dla katalogów „main", co zostało także zawarte w Rysunku 3.3.

Specyficzne cechy Kotlina jak funkcjonalności języka, biblioteki czy projekt Kotlin Mul-



Rysunek 3.3: Przykładowa struktura pojektu Kotlin Multiplatform [źródło: opracowanie własne]

tiplatform mogą zapewnić znaczne wzrosty wydajności funkcji AWS Lambda. Mimo, że Kotlin jest językiem wywodzącym się z Javy oferuje już możliwości, które mogą pozwolić na osiągnięcie niższych czasów odpowiedzi. Dodatkowo, sposoby te nie zostały jeszcze przebadane. Dlatego Kotlin to obszar, który zasługuje na zawarcie go w badanich na temat wydajności AWS Lambda.

3.4. Kotlin/JS

Podczas wyboru języka programowania działającego w AWS Lambda częstym kryterium może być czy język jest kompilowany, czy interpretowany. Kotlin to domyślnie język kompilowany, jednak poprzez projekt Kotlin Multiplatform zapewnia on możliwość translacji do języka JavaScript. Jednocześnie zachowuje on możliwość używania Kotlina, który jest łatwy do nauki przez programistów Java. W ramach rozdziału opisano sposób działania Kotlin/JS

oraz wybrane cechy i ograniczenia, które wpływają na użycie w AWS Lambda.

Projekt Kotlin Multiplatform opiera się na możliwości wskazania docelowych platform dla kompilatora Kotlin, aby kod napisany w tym języku mógł być używany w środowiskach innych niż maszyna wirtualna Java. Podobnie jest w przypadku Kotlin/JS, który jest jedną z dostępnych docelowych platform KMP [1]. Cały proces opiera się na translacji między językami, która nie jest jednak bezpośrednia. Dodatkowym krokiem jest użycie reprezentacji pośredniej Kotlina (ang. Kotlin intermediate representation, IR) [1]. Po pierwsze, kod źródłowy Kotlina jest transformowany do reprezentacji pośredniej. Następnie reprezentacja ta jest stopniowo kompilowana do kodu JavaScript, który może zostać uruchomiony na docelowej platformie. Proces transformacji został przedstawiony na Rysunku 3.4. Każdy wspierany przez kompilator IR element języka Kotlin jest reprezentowany w formie reprezentacji pośredniej. Mogą to być m.in. funkcje (wraz ich argumentami, typami oraz modyfikatorami dostępu), instrukcje „return", czy instrukcje warunkowe wraz z wszystkimi rozgałęzieniami. W kolejnym etapie, dla poszczególnych elementów reprezentacji pośredniej, dobierane są ich odpowiedniki właściwe dla docelowej platformy, co w przypadku Kotlin/JS skutkuje wygenerowaniem kodu JavaScript.

Rysunek 3.4: Proces transformacji kodu Kotlin do kodu JavaScript [źródło: opracowanie własne na bazie repozytorium kompilatora IR [3]]

Istotnym elementem Kotlin/JS jest wsparcie dla dwóch środowisk wykonawczych: Node.js oraz przeglądarkowego. W przypadku przeglądarek Kotlin oferuje wsparcie dla DOM API (ang. Document Object Model) [1], co pozwala na łatwiejszy rozwój aplikacji klienckich działających w przeglądarce. Ważniejsze dla AWS Lambda jest jednak wsparcie dla Node.js, który jest

środowiskiem wykonawczym wspieranym przez usługę. Wybór środowiska jest łatwo określany poprzez konfigurację narzędzia budującego (np. Gradle) [1].

Język JavaScript jest znacznie wspierany przez liczną społeczność programistów, co skutkuje bardzo szeroką ofertą różnych bibliotek i frameworków. Dlatego kluczowym elementem narzędzi jak Kotlin/JS jest wsparcie dla tych aspektów języka. Po pierwsze, Kotlin/JS pozwala na bezpośrednie użycie kodu JavaScript z Kotlina, poprzez użycie funkcji „js()" [1]. Z jej użyciem programista może przekazać dowolny kod JavaScript, który zostanie wykonany w miejscu użycia funkcji. Co więcej, Kotlin Multiplatform oferuje możliwość użycia bibliotek z menadżera pakietów NPM. Wymaga to od programisty deklaracji tej zależności w konfiguracji Gradle, która jest podobna do deklaracji zwykłych zależności. Po tym, może on użyć specjalnej annotacji „JsModule" wraz z słowem kluczowym „external", które pełnią rolę adaptera [1]. Wynika to z dynamicznego typowania JavaScript oraz stytycznego Kotlina. Dla zapewnienia lepszych doświadczeń programisty, Kotlin/JS umożliwia także wygenerowanie typów TypeScript [1]. Mogą być one użyte w przypadu np. udostępnienia biblioteki napisanej z użyciem Kotlin/JS. Samo zapewnienie integracji z zewnętrznymi bibliotekami jest kluczowe w przypadku AWS Lambda, które wymaga integracji z np. AWS SDK.

Kolejnym ważnym aspektem AWS Lambda jest zmniejszanie wielkości artefaktu [58] [56]. W przypadku Kotlin/JS oferowane jest kilka mechanizmów, które mogą pozytywnie wpłynąć na jego rozmiar. Między innymi zapewnia on narzędzie DCE (ang. dead code elimination), które jest wbudowane w kompilator IR. Polega ono na eliminacji kodu, który nie jest wykorzystywany w aplikacji. Mogą to być na przykład funkcje z biblioteki standardowej Kotlina, które nie zostały użyte w kodzie funkcji. Dodatkowo, aby kod Kotlin został przekonwertowany do kodu JavaScript, musi zostać użyta annotacja „@JsExport" [1]. Dzięki temu, oznaczone nią funkcje czy klasy są traktowane jako elementy źródłowe dla mechanizmu DCE, co rozpocznie analizę używanego kodu właśnie od nich. Dodatkowo, kompilator dokonuje minifikacji (ang. minification) nazw, jak zmienne. Wszystko to pozwala na osiągnięcie mniejszego rozmiaru artefaktu, co jest istotnym aspektem w optymalizacji wydajności AWS Lambda.

Kotlin/JS posiada jednak także ograniczenia, które mogą negatywnie wpłynąć na rozwój oprogramowania. Po pierwsze, mimo integracji z bibliotekami zewnętrznymi, może wymagać ona znacznych nakładów pracy. Wynika to z użycia słowa kluczowego „external" i potrzeby utrzymywania kodu, który zawiera typy bibliotek zewnętrznych. Może to znacząco spowolnić wszelkie zmiany wersji bibliotek, które mogą zmienić swoje interfejsy. Dodatkowo, Kotlin/JS posiada znaczne ograniczenia w mechanizmie refleksji. Nie implementuje on całego API reflekcji Kotlina, a jednymie referencję klasy (::class), typy KType i KClass oraz powiązane z nimi funkcje „typeof()" (zwracającą typ) i „createInstance()" (tworzącą nową instancję klasy). Czynniki te powinny być uwzględnione w przypadku wyboru Kotlin/JS jako technologii systemu działającego w AWS Lambda.

3.5. Kotlin/Native

Bardzo skuteczną metodą optymalizacji wydajności może być rezygnacja z zarządzanych środowisk uruchomieniowych jak JVM. Jedną z metod jest kompilacja do kodu natywnego, który uruchamiany jest bezpośrednio na systemie operacyjnym poza maszyną wirtualną. Kotlin/Native oferuje zaawansową kompilację kodu Kotlin, który następnie może zostać

uruchomiony w AWS Lambda z użyciem Amazon Linux. W ramach rozdziału opisano sposób działania Kotlin/Native, jego możliwości i cechy, które mogą negatywnie wpłynąć na rozwój oprogramowania bezserwerowego.

Kluczowym elementami Kotlin/Native są kompilator oparty o LLVM oraz natywne implementacje bibliotek standardowych Kotlina [1]. Jak opisują Lattner i Adve [50], LLVM to framework kompilatora zaprojektowany do wspierania ciągłej i transparentnej analizy oraz transformacji programów. Definiuje on wspólną, niskopoziomową reprezentację kodu w formie SSA (ang. Static Single Assignment) z systemem typów niezależnym od języka, co umożliwia implementację cech języków wysokiego poziomu. Głównym celem LLVM jest umożliwienie analizy i transformacji programu na różnych etapach jego życia, w tym w czasie kompilacji, linkowania, uruchomienia oraz w czasie bezczynności między uruchomieniami. Jego użycie pozwala następnie na zbudowanie natywnych plików binarnych, które mogą zostać uruchomione bezpośrednio na docelowej platformie, dla której zostały skompilowane. Wymaga to jednak dokładnego określenia systemu operacyjnego i architektury już w momencie kompilacji.

Kompilacja Kotlina do kodu natywnego otwiera przez KMP możliwość tworzenia samodzielnych programów wykonywalnych, które nie wymagają zewnętrznego środowiska uruchomieniowego. Znajduje to zastosowanie w scenariuszach takich jak rozwój aplikacji mobilnych na platformę iOS, współdzielenie logiki biznesowej między różnymi platformami (np. Android i iOS) czy budowa narzędzi konsolowych. Dlatego ważnym aspektem Kotlin/Native jest współdziałanie z istniejącym kodem natywnym. Pozwala to na bezpośrednie wywoływanie funkcji z bibliotek napisanych w języku C, a na platformach firmy Apple również Objective-C [1]. Znacząco rozszerza to zakres dostępnych narzędzi i bibliotek, które mogą zostać użyte przez programistów.

Kotlin/Native oferuje wiele różnych platform docelowych, rozszerzając tym samym zakres zastosowań języka Kotlin. Wśród wspieranych systemów docelowych znajdują się platformy Apple (takie jak macOS, iOS, watchOS, tvOS), Android, a także systemy z rodziny Windows oraz Linux [1]. Szczególnie istotnie dla usługi AWS Lambda są jednak platformy linuxowe. Są to linuxX64 oraz linuxArm64, które pozwalają na uruchomienie kodu z użyciem odpowiednio architektur x86 oraz ARM. Pozwala to następnie na ich bezpośrednie użycie w AWS Lambda, działającej bezpośrednio w systemie Amazon Linux. Dzięki temu możliwa jest poprawa wydajności, szczególnie w aspekcie zimnych startów, które są znacznym wyzwaniem dla funkcji opartych o JVM.

Istotnym elementen funkcji bezserwerowych jest zarządzanie pamięcią, która wpływa bezpośrednio na koszty. W przypadku Kotlin/Native, ewolucja modelu zarządzania pamięcią znacząco wpłynęła na jego użyteczność i możliwości optymalizacyjne. Początkowa technologia ta opierała się na restrykcyjnym modelu z izolacją obiektów między wątkami [1]. Powodowało to skomplikowane zarządzanie stanem w operacjach współbieżnych. Aktualnie, Kotlin/Native implementuje nowy menedżer pamięci. Wprowadza on automatyczne zarządzanie pamięcią poprzez współbieżny, nieblokujący moduł zbierania śmieci (ang. garbage collector) [1]. Znacząco upraszcza to programowanie współbieżne, które teraz nie wymaga ręcznego zarządzania obiektami. Mechanizm ten wprowadza intuicyjne współdzielenie stanu, które jest analogiczne do środowiska JVM, jednak bez konieczności kosztownej obsługi maszyny wirtualnej Java.

Mimo potencjalnych zysków w ramach wydajności, użycie Kotlin/Native może nieść utrudnienia w kontekście integracji z bibliotekami zewnętrznymi. Język C nie jest oficjalnie wspierany jako język programowania AWS Lambda [8], co wynika zapewne z jego niewielkiej

popularności na tej platformie. Współdziałanie z kodem natywnym w Kotlin/Native skupia się jednak na platformach klienckich, co nie musi być do końca użyteczne w zakresie AWS Lambda. Amazon Web Services oferuje swoje SDK w języku C++, który nie jest jednak łatwo integrowalny z Kotlin/Native (w odróżnieniu od C i Objective-C). Jest to ważny czynnik, który powinien być uwzględniony przez programistów projektujących aplikacje bezserwerowe.

4. Opis badania

4.1. Cel i metodologia badań

Przeprowadzenie odpowiednich badań jest kluczowym elementem pracy, który ma na celu znaczące wsparcie odpowiedzi na postawione pytania badawcze. Po pierwsze, skupiono się na dwóch pierwszych pytaniach badawczych:

- PB1: Które metody optymalizacji pozwalają na najlepszą poprawę ogólnej wydajności funkcji AWS Lambda w ekosystemie Java?
- PB2: W jakim stopniu wybrane metody optymalizacji redukują czas zimnego startu funkcji Java w AWS Lambda?

Aby udzielić odpowiedzi na powyższe pytania, zdecydowano się na wykonanie testów wydajnościowych funkcji. Opierały się one na przygotowanych wcześniej funkcjach, które implementowały konkretne metody optymalizacji (opisane w Rozdziale 3). Specyfikacja poszczególnych funkcji została przedstawiona w kolejnych rozdziałach. Bazując na wnioskach wyciągniętych w ramach przeglądu literatury (opisanego w Rozdziale 2) zdecydowano się na rozszerzenie badania o wybrane parametry, które mogą wpłynąć na ogólną wydajność usługi. Były to wielkość pamięci funkcji oraz pomiar zarówno ciepłych, jak i zimnych startów.

W celu porównania skuteczności wybranych metod opracowano konkretne metryki, które zostały użyte w badaniu:

1. Czas działania funkcji podczas ciepłego startu (w milisekundach).
2. Czas działania funkcji podczas zimnego startu (w milisekundach).
3. Współczynnik wydajności funkcji.
4. Koszt działania funkcji (w dolarach amerykańskich).

Pierwsza metryka, czyli czas działania funkcji podczas ciepłego startu jest najprostszym pomiarem. Określa ona bezpośrednio wydajność funkcji w przypadku większości wywołań. Dlatego metryka ta pełni kluczową rolę w określeniu efektywności poszczególnych metod. Aby jednak zawrzeć w badaniu także drugi rodzaj startów (oraz odpowiedzieć na drugie pytanie badawcze), określono kolejną metrykę, czyli czas działania funkcji podczas zimnego startu. W celu dokonania jego pomiaru, niezbędne jest odpowiednie przygotowanie funkcji, które wymaga większych nakładów pracy w przygotowaniu środowiska badawczego, co zostało opisane w kolejnych rozdziałach. Pozwala to jednak na rzetelniejsze określenie skuteczności konkretnych metod. Same zimne starty są uznawane za problematyczne w przypadku użycia Javy w AWS Lambda [45] [53]. Dlatego metryka ta jest wymagana, by ująć w badaniu bardziej rzeczywiste przypadki użycia usługi.

Osobna analiza ciepłych i zimnych startów jest jednak niewystarczająca, aby rzetelnie określić efektywność metod w kontekście ich praktycznego użycia. W momencie działania, funkcje AWS Lambda doświadczają obu typów startów, z dominacją ciepłych startów w przypadku aktywnych funkcji. Z tego powodu rozszerzono badania o metryki, na które wpływ ma czas działania funkcji podczas obu wariantów uruchomień. Są to współczynnik wydajności funkcji oraz koszt jej działania.

Na potrzeby pracy został zaproponowany współczynnik wydajności funkcji (WWF). Wpływ na niego mają zarówno ciepłe, jak i zimne uruchomienia funkcji. Aby opracowany współczynnik lepiej odwzorowywał realne użycie usługi AWS Lambda zdecydowano się na wykorzystanie wag, które modyfikują oddziaływanie ciepłych i zimnych startów. Możliwe jest porównanie skuteczności metod optymalizacji wydajności poprzez porównanie wartości WWF, gdzie im wyższa wartość, tym efektywniejsza jest dana metoda. Współczynnik wydajności funkcji określony jest wzorem:

$$\mathrm{WWF} = \frac{1000}{t_c \cdot w_c + t_w \cdot w_w}$$

gdzie:

- t_c - średni czas zimnego startu funkcji [ms],
- t_w - średni czas ciepłego startu funkcji [ms],
- w_c - waga zimnych startów,
- w_w - waga ciepłych startów.

Kolejną metryką łączącą ciepłe i zimne starty jest koszt działania funkcji. Jest to jeden z głównych elementów wdrożenia technologii bezserwerowych, które rozliczane są w zależności od użycia usług. W przypadku AWS Lambda jest on określany ze względu na czas działania funkcji, wielkość pamięci oraz liczbę wywołań [14]. Ostatni czynnik nie został zawarty w metryce, gdyż nie jest on zależny od wydajności funkcji. Dwa pierwsze liczone są jako GB-sekundy, które są iloczynem wielkości pamięci (w GB) i czasu działania funkcji (w sekundach). Metryka ta jest istotna, gdyż w badaniu zostały zawarte różne wielkości pamięci funkcji. Z tego względu wartościowy jest także pomiar wpływu poszczególnych metod na wydajność kosztową, na którą składają się zarówno czas działania, jak i wielkość pamięci. Na potrzeby pracy opracowano wzór opisujący koszt działania funkcji przez 30 dni przy określonej liczbie zapytań na sekundę:

$$\mathrm{K} = P \cdot M \cdot r \cdot (p_c \cdot d_c + p_w \cdot d_w) \cdot 2592000$$

gdzie:

- K - całkowity koszt działania funkcji [USD],
- P - cena za jedną GB-sekundę [USD],
- M - pamięć funkcji [GB],
- r - liczba wywołań funkcji na sekundę,

- p_w - udział ciepłych startów (np. 0,95),
- p_c - udział zimnych startów (np. 0,05), gdzie $p_c = 1 - p_w$,
- d_c - średni rozliczany czas dla zimnych startów (w sekundach),
- d_w - średni rozliczany czas dla ciepłych startów (w sekundach),
- 2592000 - liczba sekund odpowiadająca okresowi 30 dni.

Kolejnym istotnym obszarem badawczym pracy jest analiza kompromisów w procesie rozwoju oprogramowania, które mogą wynikać z implementacji poszczególnych metod optymalizacji. W tym celu zaproponowano badania, które odpowiedzą na ostatnie pytanie badawcze:

- PB3: Jakie kompromisy w procesie rozwoju oprogramowania wiążą się implementacją poszczególnych metod optymalizacji wydajności funkcji Java w AWS Lambda?

Odpowiedź na pytanie wymaga zdefiniowania kryteriów, które pozwolą ocenić wpływ analizowanych technik na proces tworzenia systemów opartych o usługi bezserwerowe. Bazując na wnioskach z systematycznego przeglądu literatury (przedstawionego w Rozdziale 2) wybrano trzy kluczowe kryteria:

1. Czas budowy artefaktu
2. Wielkość artefaktu
3. Dostępność AWS SDK

Pierwszym analizowanym aspektem jest czas budowy artefaktu. Służy on ocenie wpływu metody optymalizacji na czas potrzebny do przygotowania plików wdrożeniowych funkcji. W ramach badania zmierzony zostanie całkowity czas budowy artefaktu dla każdej z analizowanych implementacji. Wybór tej miary wynika z faktu, że czas budowy bezpośrednio przekłada się na efektywność i szybkość iteracji w cyklu rozwoju oprogramowania. Jest to istotnie ważne z perspektywy programisty, zwłaszcza w architekturach bezserwerowych. W takich systemach tempo wdrożeń jest szybkie, kod jest z nimi bezpośrednio powiązany, a także często testowany w sposób integracyjny już w środowisku chmurowym.

Kolejnym istotnym czynnikiem branym pod uwagę jest wielkość artefaktu. Dotyczy on fizycznego rozmiaru pakietu wdrożeniowego funkcji. Dla każdej przygotowanej funkcji AWS Lambda zmierzona zostanie wielkość finalnego artefaktu. Ten parametr oceny wynika z istoty wielkości funkcji AWS Lambda, które wpływają na proces rozwoju oprogramowania oraz niosą ryzyko przekroczenia limitów wielkości pakietu wdrożeniowego [56].

Ostatnim z przyjętych mierników jest dostępność AWS SDK. Ocena tego elementu będzie polegała na weryfikacji, czy dana metoda optymalizacji pozwala na bezproblemową integrację ze standardowym zestawem narzędzi AWS SDK (ang. Software Development Kit). Jak wskazano w Rozdziale 2, funkcje FaaS często wykorzystują zewnętrzne serwisy AWS. Dostęp do tych usług odbywa się często poprzez SDK, dlatego metody optymalizacji nie powinny go ograniczać.

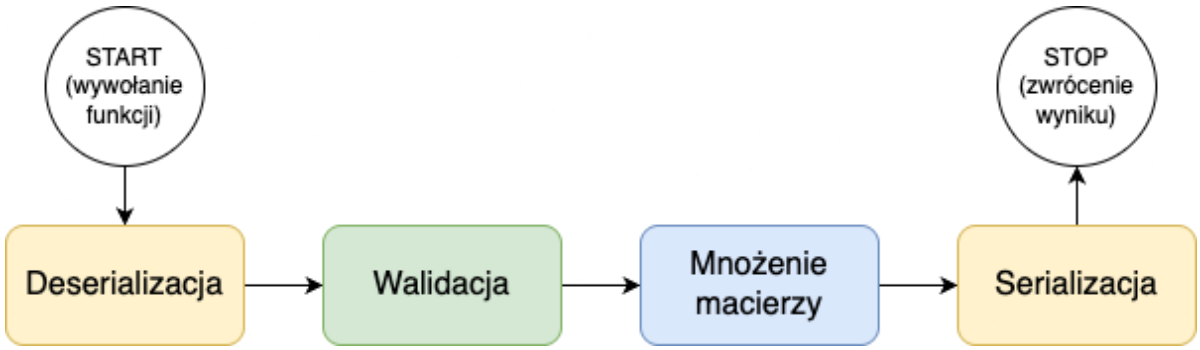
4.2. Implementacja funkcji

Dla przeprowadzenia badania niezbędne było określenie zadań, które będą realizowane przez funkcję AWS Lambda. W celu zapewnienia lepszej oceny skuteczności konkretnych metod optymalizacji oraz upodobnienia środowiska badawczego do realnych użyć usługi, przyjęto kilka założeń. Następnie, stworzono implementację usługi, zgodną z przyjętymi wcześniej zasadami.

Po pierwsze, funkcja powinna wykonywać pewne obliczenia oraz nie być programem zwracającym jedynie tekstową wiadomość (np. „Hello World!"). Założenie to pozwoli na lepszą weryfikację metod, poprzez realne użycie zaimplementowanego kodu. Eismann i inni autorzy [30] dokonali przeglądu przypadków użycia technologii bezserwerowych. Wynika z niego, że najczęstsze przypadki użycia to implementacje interfejsu API, procesowanie wydarzeń, zadania wsadowe (ang. batch tasks) oraz wsparcie operacji i monitoring istniejących systemów. Wszystkie te przypadki wymagają wykonania choćby podstawowych operacji, dlatego stworzenie funkcji wykonującej określone obliczenia pozwoli na analizę metod, w środowisku bardziej zbliżonym do realnych zastosowań. W przypadku prostych funkcji, które jedynie zwracają pewien tekst niezależny od danych wejściowych, wywołanie skupiłoby się jedynie na procesie inicjalizacji danego narzędzia.

Drugim założeniem jest użycie zewnętrznych zależności, które powinny zostać zawarte w zbudowanym artefakcie. Wynika ono z dwóch powodów. Po pierwsze, biznesowe użycia AWS Lambda często wymagają zależności zewnętrznych (np. pakietów SDK dla innych usług [30]). Po drugie, wielkość pakietu wdrożeniowego wpływa na wydajność funkcji [58] [56]. Zatem aby zwiększyć jego rozmiar i upodobnić ostateczne artefakty do zastosowań praktycznych, zdecydowano się dodać biblioteki zewnętrzne. Są one jednak zorientowane na konkretnym zadaniu (co zostanie przedstawione w dalszej części rozdziału), a ich liczba nie powinna być zbyt duża.

Po trzecie, funkcja nie powinna używać usług zewnętrznych. Funkcje bezserwerowe bardzo często integrują się z innymi usługami [30], jednak odrzucono ich użycie w badanych funkcjach. Wynika to z ryzyka wpływu czynników, które są niezależne od badanych metod, na czas procesowania funkcji. Mogą to być różnego typu opóźnienia sieciowe (np. w przypadku zapytań do baz danych działających poza infrastrukturą AWS Lambda), które mogą różnić się między wywołaniami.



Rysunek 4.1: Wizualizacja procesu realizowanego przez badane funkcje AWS Lambda [źródło: opracowanie własne]

Zgodnie z powyższym opracowano zadanie, które będzie realizowane przez funkcje. Aby utrzymać prostotę implementacji, zdecydowano się na wykonanie przez nią obliczeń, którymi jest mnożenie macierzy. Są to już jednak wystarczające operacje, które wydłużą czas działania funkcji. Macierze, które zostaną poddane mnożeniu były przekazywane do funkcji jako dane wejściowe. Umożliwiło to wykonanie dwóch kolejnych operacji: walidacji oraz deserializacji. W tym celu użyto zewnętrznych zależności, które pozwoliły na zwiększenie rozmiaru artefaktu. Biblioteki dla konkretnych języków oraz zastosowań zostały przedstawione w Tabeli 4.1. Mimo interoperacyjności Javy i Kotlina, zdecydowano się na użycie bibliotek, które zostały zaimplementowane w konretnym języku używanym w danej funkcji. Cały proces realizowany przez funkcję AWS Lambda został przedstawiony na Rysunku 4.1. Jego pierwszym elementem była deserializacja danych wejściowych, które zostały przekazane jako napis (typ String) w formacie JSON. Już odczytane macierze zostały poddane walidacji sprawdzającej ich wielkość, a następnie pomnożone. Wynik działania został poddany serializacji z powrotem do postaci napisu i zwrócony jako wynik wywołania funkcji. Cały proces został analogicznie zaimplementowany w każdej z badanych funkcji.

Tabela 4.1: Zewnętrzne zależności wykorzystane w badanych funkcjach

Język programowania	**Przeznaczenie**	**Nazwa biblioteki**
Java	Serializacja i deserializacja	io.quarkus:quarkus-jackson
	Walidacja	io.quarkus:quarkus-hibernate-validator
Kotlin	Serializacja i deserializacja	org.jetbrains.kotlinx:kotlinx-serialization-json
	Walidacja	io.konform:konform

W ramach badania przygotowano 7 funkcji, które realizowały wybrane metody optymalizacji wydajności. Każda funkcja wykorzystywała specyficzne narzędzia, które zostały opisane poniżej, a ich wersje zostały podane w Tabeli 4.2. Funkcje wykorzystujące język Java używały jej w wersji 21, funkcje oparte o Kotlin używały go w wersji 2.1.0. W ramach pracy zrealizowano poniższe warianty:

- **Java JVM** - standardowa implementacja w języku Java, działająca w środowisku JVM, która służy jako baza do analizy skuteczności wybranych metod. Została zrealizowana w oparciu framework Quarkus, który wspiera wdrożenia do AWS Lambda [60] i znacząco przyspieszona rozwój oprogramowania. Framework został wybrany także z powodu integracji z natywnymi obrazami GraalVM i wsparciem dla usługi SnapStart [60].
- **Java JVM + SnapStart** - w celu wykorzystania metody SnapStart użyto tej samej implementacji jak w przypadku funkcji Java JVM. Funkcja ta działa także z użyciem maszyny wirtualnej Java ale z aktywowaną funkcją SnapStart.
- **Java GraalVM** - implementacja w języku Java, wykorzystująca ten sam kod jak w przypadku dwóch poprzednich funkcji, jednak skompilowany do obrazu natywnego GraalVM. W tym celu wykorzystano możliwości frameworka Quarkus oraz niestandardowe środowisko uruchomieniowe (ang. custom runtime) w AWS Lambda.

- **Kotlin JVM** - analogiczna implementacja w języku Kotlin, zrealizowana z użyciem narzędzia http4k zapewniającego integrację z serwisem AWS Lambda [2]. Narzędzie zostało użyte ze względu na jego oparcie o język Kotlin i natywne wsparcie dla tego języka programowania. Następnie funkcja została wdrożona i uruchamiana w środowisku JVM.

- **Kotlin JVM + SnapStart** - funkcja oparta o implementację poprzedniej funkcji, działająca z użyciem maszyny wirtualnej Java i z uruchomioną usługą SnapStart.

- **Kotlin/JS** - implementacja w języku Kotlin, która została następnie transpilowana do języka JavaScript. Nie zostały użyte żadne dodatkowe biblioteki (oprócz zależności określonych w Tabeli 4.1), a komponent oparty jest bezpośrednio o projekt Kotlin Multiplatform. We wdrożeniu funkcji użyto środowiska Node.js, oferowanego przez AWS Lambda [13].

- **Kotlin/Native** - implementacja w języku Kotlin, która została następnie skompilowana do binarnego pliku natywnego i wdrożona z użyciem niestandardowego środowiska uruchomieniowego (ang. custom runtime). Funkcja wykorzystuje bibliotekę kotlin-native-aws-lambda-runtime [40], która zapewnia możliwość budowy artefaktów natywnych i wsparcie obsługi żądań. Biblioteka oparta jest w całości o język Kotlin, framework Ktor oraz możliwości Kotlin Multiplatform.

Tabela 4.2: Biblioteki oraz frameworki użyte podczas implementacji badanych funkcji

Rodzaj funkcji	**Użyta biblioteka/framework**	**Wersja**
Java JVM	io.quarkus:quarkus-amazon-lambda	3.17.5
Java JVM + SnapStart		
Java GraalVM		
Kotlin JVM	http4k-serverless-lambda	6.9.0.0
Kotlin JVM + SnapStart		
Kotlin/JS	Brak	Brak
Kotlin/Native	io.github.trueangle:lambda-runtime	0.0.2

Wartym zaznaczenia w kontekście implementacji funkcji są zdolności Kotlin Multiplatform (opisane w Rozdziale 3.3), które zostały użyte podczas tworzenia funkcji. Główna część funkcji, tj. serializacja, walidacja, mnożenie oraz deserializacja, zostały umieszczone w module commonMain oraz współdzielone między wszystkie docelowe platformy. Następnie w modułach dla danych platform (jvmMain, jsMain oraz nativeMain) została zaimplementowane obsługa zapytania oraz wywołania funkcji z części commonMain. Pozwoliło to na badanie oparte o dokładnie ten sam kod, który został użyty w każdej z badanych funkcji. Możliwości współdzielenia kodu miały także wpływ na wybór bibliotek do walidacji oraz serializacji, które musiały wspierać kompilację dla każdej z badanych platform.

4.3. Wdrożenie funkcji

Kolejnym elementem niezbędnym do przeprowadzenia badania było wdrożenie testowanych funkcji już na platformę chmurową. Etap ten wymagał utworzenia także niezbędnej infrastruktury ciągłej integracji oraz procesu wdrażania, co umożliwiło automatyzację procesu budowy i aktualizacji funkcji AWS Lambda. Proces ten został zrealizowany z wykorzystaniem narzędzi infrakstruktury jako kod (ang. Infrastructure as a Code), co zapewniło powtarzalność i spójność środowiska testowego. Przygotowanie potoku ciągłego integracji jest także niezbędne w przypadku używania środowisk deweloperskich z procesorami o architekturze ARM64. Funkcje zostały wdrożone w architekturze x86-64, zatem do budowy artefaktów potrzebne było użycie maszyn operujących także w tym typie procesora.

Cały proces budowy artefaktów oraz ich wdrażania został zautomatyzowany z użyciem GitHub Actions. Do wytworzenia artefaktu użyto narzędzia Gradle, który pozwał na uruchomienie odpowiedniego zadania dla każdej z zaimplementowanych funkcji. Odpowiadało ono za kompilację kodu źródłowego, czego wynikiem były pliki gotowe do uruchomienia programu. Dla każdej z przygotowanych funkcji utworzono odpowiedni krok w procesie GitHub Actions, który uruchamiał proces budowy dla danej platformy i docelowej architektury x86-64 (np. Java JVM czy Kotlin/JS). Stworzone w ten sposób pliki były następnie przekazywane do drugiego kroku, który odpowiadał za wgranie ich do usługi S3 (ang. Simple Storage Service). Przechowywane tak pakiety były już gotowe do użycia w wdrażanych funkcjach AWS Lambda. Dodatkowo, opisany proces ciągłej integracji pozwolił na pomiar czasu budowy funkcji, co jest istotne w odpowiedzi na postawione pytania badawcze i zostało zawarte w wynikach badania. Wielkość artefaktów została zmierzona jako wielkośc plików przechowywanych w usłudze S3.

Kolejnym etapem było wdrożenie przechowywanych w serwisie S3 artefaktów jako funkcje AWS Lambda. W tym celu użyto narzędzia Terraform, które pozwala na wdrożenie infrastruktury chmurowej w oparciu o kod. W ramach konfiguracji Terraform dla każdej instancji AWS Lambda określono nazwę funkcji, lokalizację artefaktu w usłudze S3, sygnaturę wywoływanej metody (tj. pełną ścieżkę do metody, która będzie wywoływana po otrzymaniu zdarzenia), środowisko wykonawcze oraz czy funkcja aktywuje usługę SnapStart. Środowiska wykonawcze, w których uruchomiano poszczególne funkcje, zostały przedstawione w Tabeli 4.3. Każda funkcja została wdrożona dla następujących wielkości pamięci: 128 MB, 256 MB, 512 MB, 1024 MB oraz 2048 MB.

Istotnym faktem dla wdrożenia funkcji z użyciem Kotlin/Native jest potrzeba utworzenia dodatkowej warstwy AWS Lambda (ang. lambda layer). Warstwy te stanowią archiwum w formacie .zip, zawierające dodatkowy kod lub dane, takie jak zależności biblioteczne, niestandardowe środowiska wykonawcze czy pliki konfiguracyjne. Dla funkcji Kotlin/Native zbudowano warstwę, która zawierała zależność libcrypt. Biblioteka ta nie jest domyślnie dostępna w dystrybucji Amazon Linux 2023, jednak jest wymagana do działania przez technologię użytą w implementacji Kotlin/Native [40]. Potrzeba ta wynikała z użycia niestandardowego środowiska uruchomieniowego, co może wpłynąć na jakość procesu rozwoju oprogramowania i powinny być wzięte pod uwagę podczas użycia takich środowisk.

Tabela 4.3: Środowiska wykonawcze użyte w badanych funkcjach AWS Lambda

Rodzaj funkcji	**Identyfikator środowiska wykonawczego**
Java JVM	java21
Java JVM + SnapStart	java21
Java GraalVM	provided.al2023
Kotlin JVM	java21
Kotlin JVM + SnapStart	java21
Kotlin/JS	nodejs22.x
Kotlin/Native	provided.al2023

4.4. Środowisko badawcze

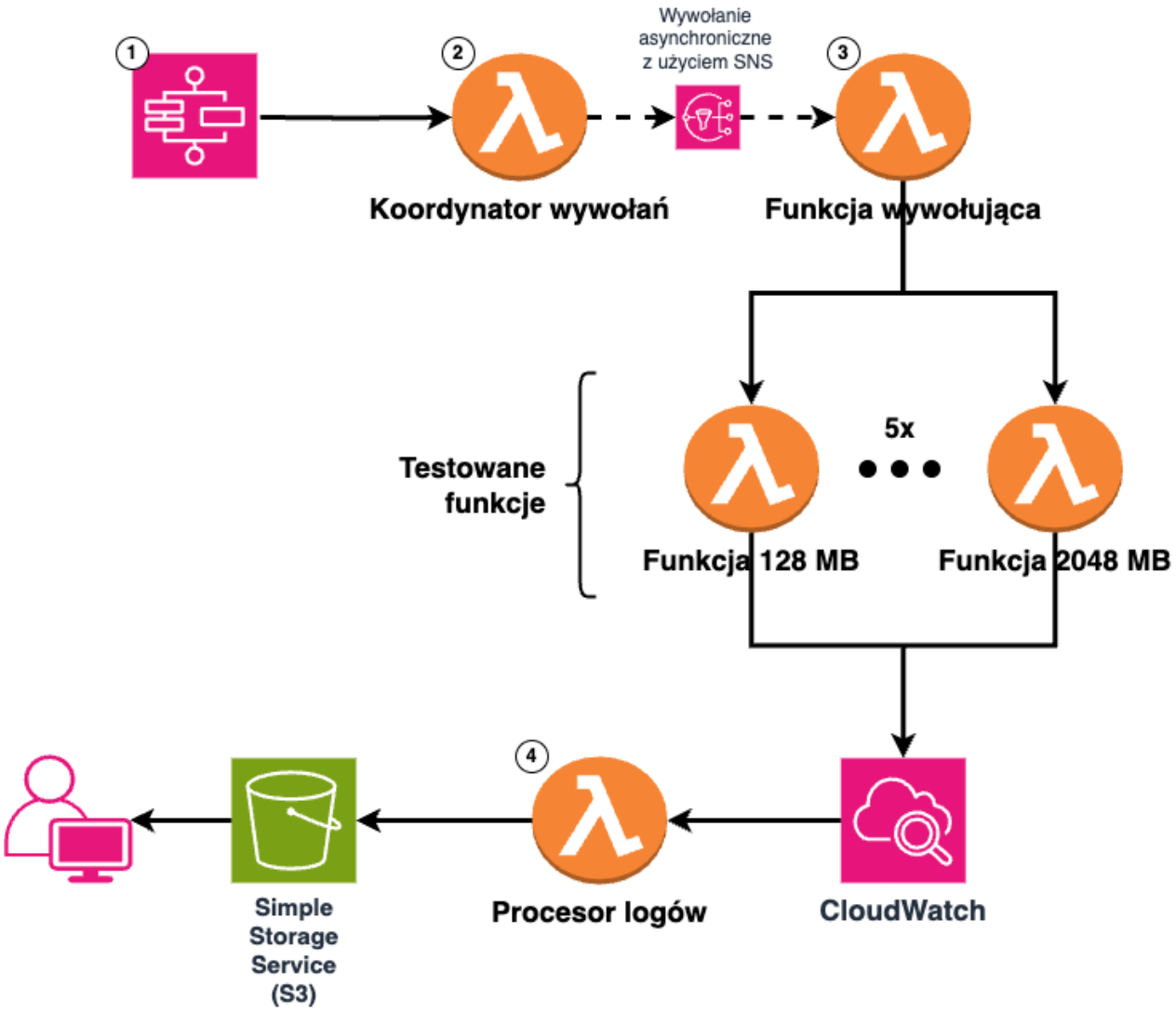
W celu pomiaru wartości metryk (określonych w Rozdziale 4.1) przygotowano odpowiednią infrastrukturę chmurową wdrożoną w środowisku AWS. Jej zadaniem było odpowiednie przygotowanie badanych funkcji, ich uruchomienie oraz zebranie wyników pomiarowych. Architektura środowiska badawczego została przedstawiona na Rysunku 4.2 oraz dokładnie opisana w poniższym rozdziale. Przygotowanie takiego środowiska pozwoliło na automatyczne przeprowadzenie eksperymentu oraz ułatwia jego ewentualne powtórzenie. Do wdrożenia środowiska wykorzystano bezserwerowe usługi oferowane przez AWS (m.in. AWS Lambda oraz serwisy integrujące się z nią).

Głównym zadaniem przygotowanego środowiska badawczego jest wywołanie przygotowanych wcześniej funkcji, aby przeprowadzić serię pomiarów ich wydajności. W tym celu środowisko zostało zaprojektowane tak, aby umożliwić kontrolowane wykonanie funkcji po określonej liczbie tzw. zimnych startów (ang. cold starts) oraz ciepłych startów (ang. warm starts). Po każdym wywołaniu, środowisko odpowiada za zebranie kluczowych metryk wydajnościowych. Zebrane wyniki są następnie przetwarzane i agregowane, co pozwala na generowanie finalnego pliku w formacie JSON, który stanowi podstawę do dalszej analizy badanych metod.

Kluczowym aspektem badania jest rozróżnienie między zimnym a ciepłym startem funkcji. Dlatego niezbędne było odpowiednie przygotowanie badanej funkcji przed jej wywołaniem, co pozwoliło osiągnąć oczekiwany rodzaj startu. Inicjalizacja ta różniła się od rodzaju startu oraz od faktu, czy aktywowana jest usługa SnapStart. W tym celu przygotowano specjalną funkcję AWS Lambda („funkcję wywołującą” oznaczoną numerem 3 na Rysunku 4.2) otrzymującą jako dane wejściowe kod ARN (ang. Amazon Resource Name) badanej funkcji AWS Lambda, która ma zostać przygotowana i wywołana.

W celu wiarygodnego pomiaru czasu zimnego startu standardowej funkcji Lambda (bez aktywowanej funkcji SnapStart), funkcja wywołująca przed każdym pomiarem celowo modyfikuje konfigurację badanej funkcji poprzez zmianę wartości jej zmiennej środowiskowej. Ta operacja wymusza na platformie AWS Lambda odrzucenie ewentualnie istniejącego, gotowego do użycia środowiska wykonawczego i utworzenie nowego, co dokładnie symuluje warunki zimnego startu [13]. Ciepłe starty dla funkcji standardowych mierzone są poprzez kolejne, następujące po sobie wywołania tej samej, już „rozgrzanej” instancji funkcji, bez wprowadzania zmian w jej konfiguracji.

Inaczej realizowane jest przygotowanie do testów dla funkcji z aktywowaną opcją SnapStart.



Rysunek 4.2: Architektura środowiska badawczego [źródło: opracowanie własne]

Wynika to z przygotowywania migawek stanu przez AWS Lambda dla każdej z utworzonych wersji funkcji. Aby zatem precyzyjnie zmierzyć czas zimnego startu funkcji wykorzystującej SnapStart, zastosowano metodę polegającą na opublikowaniu kolejnych wersji badanej funkcji. Każda z tak przygotowanych wersji jest następnie poddawana jednokrotnemu wywołaniu. Takie podejście gwarantuje, że każde zarejestrowane uruchomienie stanowi pomiar czasu potrzebnego na odtworzenie funkcji ze stanu zapisanego w migawce, rzetelnie testując mechanizm SnapStart. Po zakończeniu dedykowanej serii pomiarów, wszystkie wersje funkcji, które zostały utworzone wyłącznie na potrzeby tego etapu badania są usuwane. Metody te pozwalają na dokładne wywołania funkcji, które uruchamiane są zgodnie z planem badań.

Kolejnym elementem badania jest odpowiednie uruchomienie funkcji wywołującej. Proces ten został zaprojektowany w taki sposób, aby uniknąć przekraczania limitów API AWS Lambda, co było problemem podczas tworzenia pierwszych iteracji środowiska badawczego. Procedura opierała się m.in. na oczekiwaniu pomiędzy kolejnymi operacjami. Po pierwsze, w funkcji wywołującej pomiędzy kolejnymi aktualizacjami oraz wywołaniami funkcji, program

jest zatrzymywany. Sama funkcja wywołująca nie była uruchamiała pomiaru wszystkich funkcji jednocześnie. Do zarządzania pomiarami stworzono funkcję koordynatora wywołań (oznaczoną numerem 2 na Rysunku 4.2), której rolą jest wybór funkcji przeznaczonych do analizy na podstawie określonych kryteriów, takich jak użycie mechanizmu SnapStart czy język implementacji. Pozwala to na zmniejszenie badanych jednocześnie funkcji, a co za tym idzie zapytań do API.

Dla każdej z wybranych funkcji, koordynator przygotowuje i wysyła wiadomość do tematu w usłudze Amazon Simple Notification Service (SNS). Wiadomość ta zawiera identyfikator ARN badanej funkcji oraz pożądany typ startu (zimny lub ciepły), inicjując w ten sposób funkcję wywołującą. Istotnym elementem działania koordynatora jest również wprowadzenie kontrolowanego opóźnienia pomiędzy publikacją kolejnych wiadomości, co ma na celu rozłożenie obciążenia w czasie i zapobieganie wspomnianym problemom z limitami usług AWS.

Pomiędzy kolejnymi wywołaniami funkcji koorydnatora także zastosowano określone przerwy. W tym celu użyto usługi AWS Step Functions (oznaczonej numerem 1 na Rysunku 4.2), która pozwala na tworzenie i wizualizację procesów wchodzących w interakcję z różnymi serwisami AWS. Jej zadaniem było iteracyjne uruchamianie koordynatora dla kolejnych kombinacji parametrów (języka implementacji, aktywacji SnapStart oraz typu startów). Po każdym uruchomieniu następuje 30 minutowa przerwa, podczas której usługa czeka aby najnowsze wywołanie koordynatora zostało zakończone. Wydłuża to znacząco czas potrzebny na wykonanie badanie, jednak pozwala na uniknięcie zbytniego obciążenia API, a co za tym idzie blokady kolejnych wywołań.

Ostatnim elementem badania była agregacja wyników pomiarów. W tym celu użyto usługi AWS CloudWatch, który pozwala na przesyłanie wszystkich logów wywoływanych funkcji. Po każdym wywołaniu, funkcja AWS Lambda automatycznie tworzy raport, który zawiera:

- czas wykonania,
- opłacony czas wykonania,
- czas inicjalizacji,
- czas odtworzenia stanu z migawki (w przypadku aktywnego SnapStart)
- opłacony czas odtworzenia stanu z migawki (w przypadku aktywnego SnapStart)

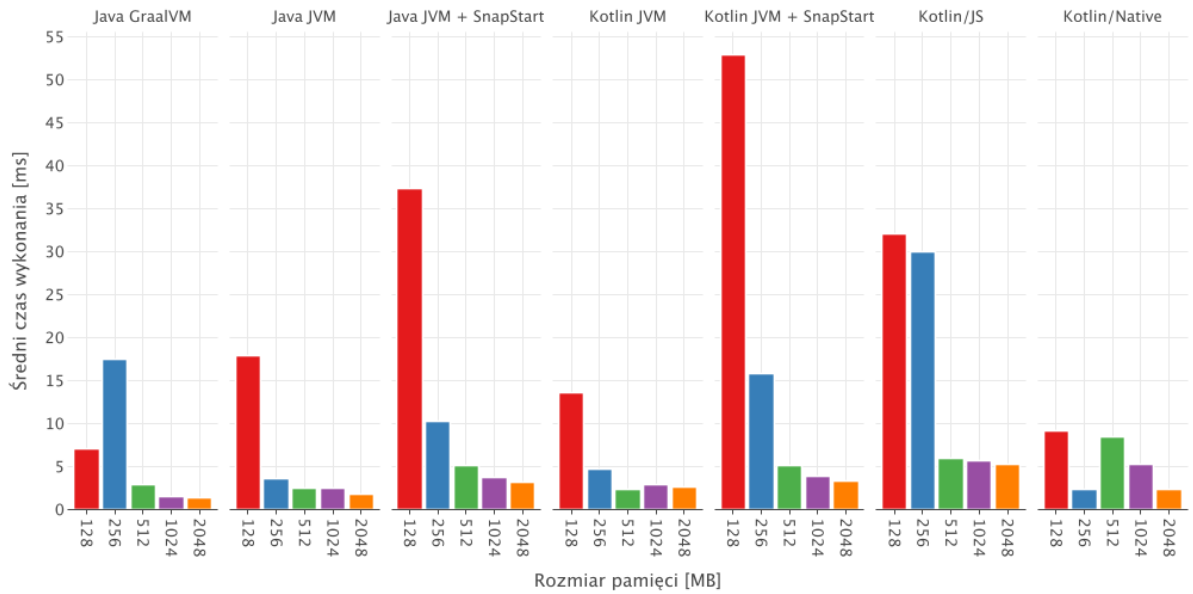
Wszystkie te dane są następnie przesyłane do usługi CloudWatch. Aby móc dokonać analizy wyników niezbędne było ich zagregowanie i zapisanie (np. w pliku JSON). Wykorzystano do tego przygotowaną funkcję procesora logów (oznaczoną numerem 4 na Rysunku 4.2), która pobierała logi z raportami z usługi CloudWatch, transformowała oraz zapisywała w pliku JSON. Następnie plik był wgrywany do usługi S3, co pozwalało na pobranie przez użytkownika. Tak przygotowana agregacja logów umożliwiła następnie niezbędną analizę wyników badania.

5. Wyniki

W ramach poniższego rozdziału przedstawiono wyniki badania, które będą kluczem w odpowiedzi na postawione w pracy pytania badawcze. Rozdział obejmuje wyniki testów wydajnościowych funkcji oraz analizę wpływu metod optymalizacji na kryteria związane z procesem rozwoju oprogramowania. W obrębie tej części pracy została zawarta obiektywna analiza wyników, co posłuży w następnym rozdziale do przedstawienia wniosków na ich bazie.

5.1. Ciepły start

Pierwszym kryterium badanym w testach wydajnościowych był czas działania funkcji podczas ciepłych startów. Oznacza to sytuację, gdy funkcja była wywołana w momencie, gdy aktywna była jedna z instancji funkcji. Powoduje to, że nie jest wymagana inicjalizacja usługi, a kod rozpoczyna działanie bezpośrednio po wywołaniu. Średni czas wykonania w zależności od metody i rozmiaru pamięci został przedstawiony na Rysunku 5.1. Dokładne wartości oraz różnica względem funkcji bazowej (Java JVM) zostały zaprezentowane w Tabeli 5.1.



Rysunek 5.1: Średni czas wykonywania funkcji (ciepły start) z użyciem wybranych metod w zależności od rozmiaru pamięci [źródło: opracowanie własne]

Dla każdego z badanych rodzajów funkcji widoczne jest znaczne zróżnicowanie czasu wykonania w zależności od rozmiaru pamięci. Sama skuteczność metod w poprawie wydajności różni się także ze względu na pamięć funkcji. Pierwsza metoda, czyli usługa SnapStart, nie przyniosła poprawy czasu wykonania w żadnej z badanych wielkości pamięci. Zastosowanie

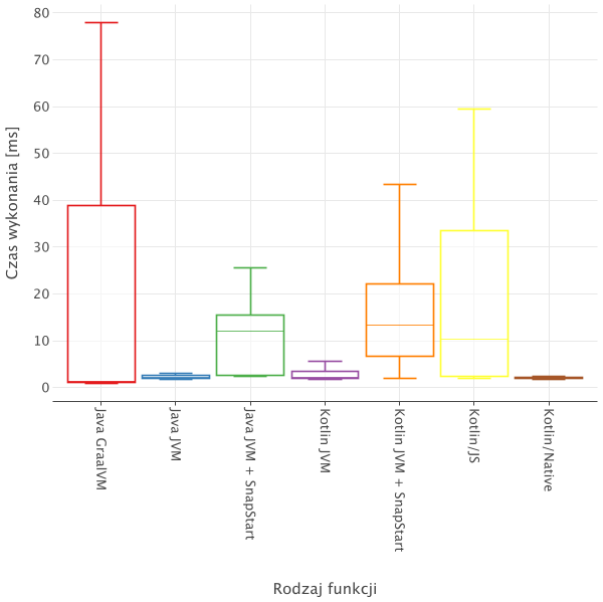
obrazów natywnych GraalVM pozwala na zmniejszenie czasu w niektórych rozmiarach pamięci: 128 MB, 1024 MB oraz 2048 MB. Podobnie zachowuje się funkcja napisana w Kotlinie, działające w ramach JVM: tutaj poprawa jest widoczna dla pamięci 128 MB i 512 MB.

Funkcja działające w oparciu o kod JavaScript, stworzony poprzez translację Kotlin/JS, niesie negatywny wpływ na czas wykonania. Metoda ta nie przyniosła poprawy w żadnym z badanych rozmiarów pamięci, przy czym w małych rozmiarach (128 MB, 256 MB) wpłynęła znacząco negatywnie na wydajność. Dodatkowo, nie widać znacznej poprawy efektywności wraz z wzrostem rozmiaru pamięci z 512 MB do 2048 MB. Ciekawa zależność widoczna jest w przypadku funkcji Kotlin/Native, która wskazuje wykazuje poprawę czasu działania dla małych wielkości pamięci (128 MB i 256 MB). Szczególna poprawa widoczna jest dla rozmiaru 256 MB, gdzie metoda ta osiągnęła niższe czasy wykonania niż funkcja Java JVM dla większych wielkości pamięci (512-2048 MB).

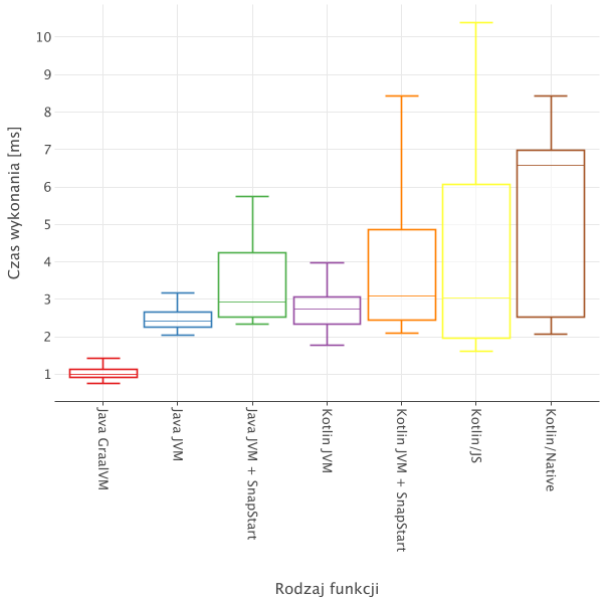
Tabela 5.1: Porównanie średnich czasów działania funkcji podczas ciepłego startu względem funkcji bazowej [źródło: opracowanie własne]

Rodzaj funkcji *Czas [ms] (% różnicy)*	**Rozmiar pamięci [MB]**				
	128	**256**	**512**	**1024**	**2048**
Java JVM	17.88	3.65	2.55	2.47	1.81
Java JVM + SnapStart	37.36 (109%)	10.3 (182%)	5.14 (102%)	3.74 (51%)	3.25 (80%)
Java GraalVM	7.08 (-60%)	17.47 (379%)	2.87 (13%)	1.47 (-40%)	1.39 (-23%)
Kotlin JVM	13.55 (-24%)	4.7 (29%)	2.41 (-5%)	2.88 (17%)	2.62 (45%)
Kotlin JVM + SnapStart	52.91 (196%)	15.87 (335%)	5.12 (101%)	3.88 (57%)	3.4 (88%)
Kotlin/JS	32.15 (80%)	29.97 (721%)	5.93 (133%)	5.65 (129%)	5.25 (190%)
Kotlin/Native	9.19 (-49%)	2.37 (-35%)	8.44 (231%)	5.3 (115%)	2.35 (30%)

Na Rysunkach 5.2 oraz 5.3 przedstawiono wykresy pudełkowe z czasami wykonania funkcji dla rozmiarów pamięci 256 MB i 1024 MB. Funkcje oparte wyłącznie o maszynę wirtualną wykazały najmniej zróżnicowane wyniki. Aktywacja funkcji SnapStart znacząco obniżyła stabilność czasów odpowiedzi, podobnie jak Kotlin/JS. Warte zwrócenia uwagi są jednak funkcje natywne. Obrazy GraalVM wykazują jednolite wyniki, oprócz pamięci 256 MB, gdzie czas wykonania znacząco różnił się w poszczególnych wywołaniach. Podobna zależność występuje dla funkcji Kotlin/Native, które osiągają niestabilne wyniki dla rozmiarów pamięci 512 MB oraz 1024 MB.



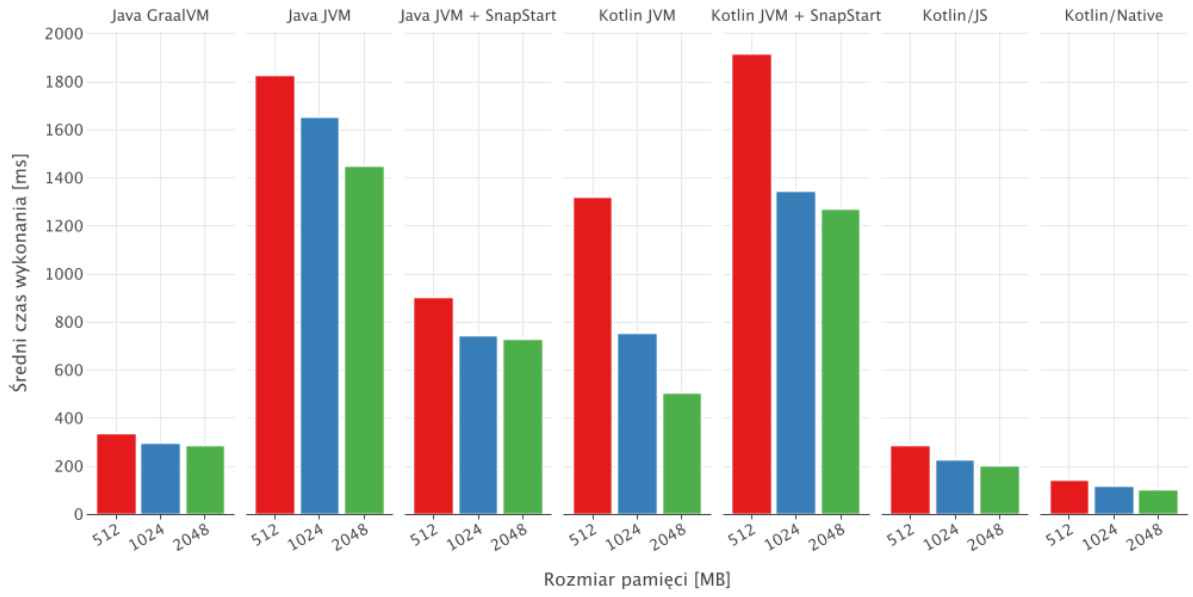
Rysunek 5.2: Czas wykonania funkcji (ciepły start, 256 MB) [źródło: opracowanie własne]



Rysunek 5.3: Czas wykonania funkcji (ciepły start, 1024 MB) [źródło: opracowanie własne]

5.2. Zimny start

Drugim badanym kryterium jest czas działania funkcji podczas zimnych startów, które wymagają odpowiedniej inicjaliacji funkcji. Z reguły są one dłuższe niż ciepłe starty, co może szczególnie wpłynąć na ogólną wydajność [45] [53]. Średni czas działania funkcji został przedstawiony na Rysunkach 5.5, 5.4 oraz w Tabeli 5.2.



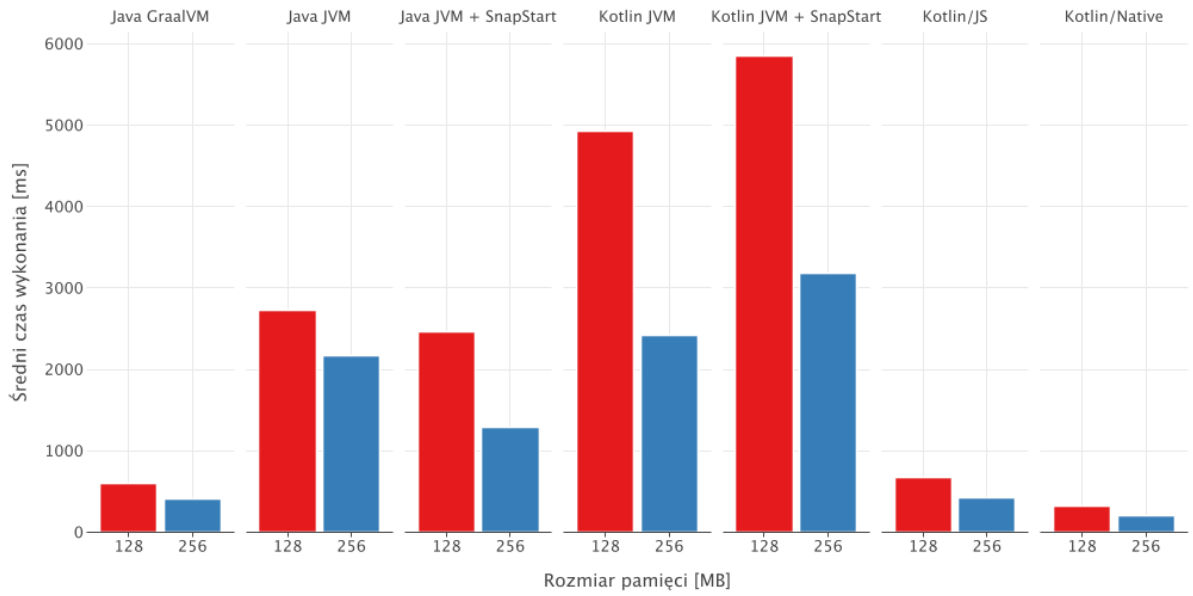
Rysunek 5.4: Średni czas wykonywania funkcji (zimny start) dla rozmiarów pamięci: 512 MB, 1024 MB, 2048 MB [źródło: opracowanie własne]

Wpływ aktywacji usługi SnapStart jest różny w zależności od języka oprogramowania. Dla Javy, usługa ta pozwoliła na poprawę czasu działania funkcji, w szczególności dla większych rozmiarów pamięci. W przypadku języka Kotlin, jej aktywacja wpłynęła negatywnie na wydajność, wydłużając czas procesowania. Samo użycie Kotlina (działającego w oparciu o JVM) pozwoliło na przyspieszenie obliczeń dla większych rozmiarów pamięci (512-2048 MB).

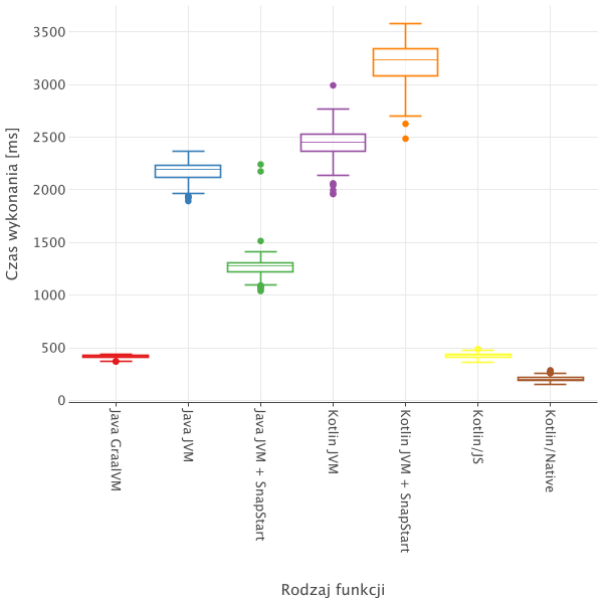
Znaczna poprawa wydajności została osiągnięte dla funkcji opartych o GraalVM oraz Kotlin/JS. Obie metody uzyskały podobne wyniki, przy czym funkcja natywna GraalVM osiągnęła poprawę w przypadku rozmiarów pamięci 128 MB i 256 MB. Dla wielkości 512-2048 MB, Kotlin/JS osiągnął niższe czasy działania niż funkcja GraalVM. Niezależnie od rozmiaru pamięci najkrótszy czas działania w przypadku zimnego startu osiągnęły funkcje oparte o technologię Kotlin/Native. Metoda ta pozwoliła także na osiągnięcie wyników o podobnej stabilności jak pozostałe funkcje, co zostało przedstawione na Rysunkach 5.2 i 5.3. Mocno zróżnicowane wyniki zostały osiągnięte przez funkcję Kotlin z aktywowaną funkcją SnapStart.

Tabela 5.2: Porównanie średnich czasów działania funkcji podczas zimnego startu względem funkcji bazowej [źródło: opracowanie własne]

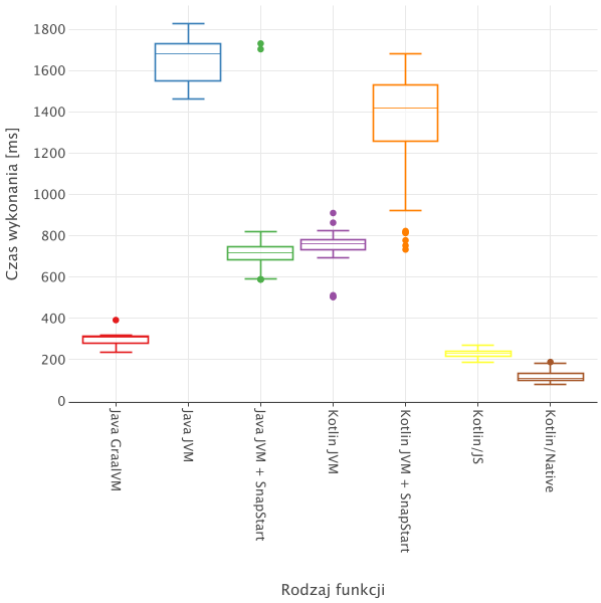
Rodzaj funkcji	**Rozmiar pamięci [MB]**				
Czas [ms] (% różnicy)	**128**	**256**	**512**	**1024**	**2048**
Java JVM	2726	2165	1827	1652	1450
Java JVM + SnapStart	2466 (-10%)	1287 (-41%)	901 (-51%)	745 (-55%)	731 (-50%)
Java GraalVM	596 (-78%)	417 (-81%)	335 (-82%)	299 (-82%)	287 (-80%)
Kotlin JVM	4927 (81%)	2426 (12%)	1320 (-28%)	754 (-54%)	504 (-65%)
Kotlin JVM + SnapStart	5853 (115%)	3185 (47%)	1915 (5%)	1347 (-18%)	1272 (-12%)
Kotlin/JS	679 (-75%)	424 (-80%)	287 (-84%)	228 (-86%)	201 (-86%)
Kotlin/Native	328 (-88%)	204 (-91%)	143 (-92%)	119 (-93%)	105 (-93%)



Rysunek 5.5: Średni czas wykonywania funkcji (zimny start) dla rozmiarów pamięci: 128 MB, 256 MB [źródło: opracowanie własne]



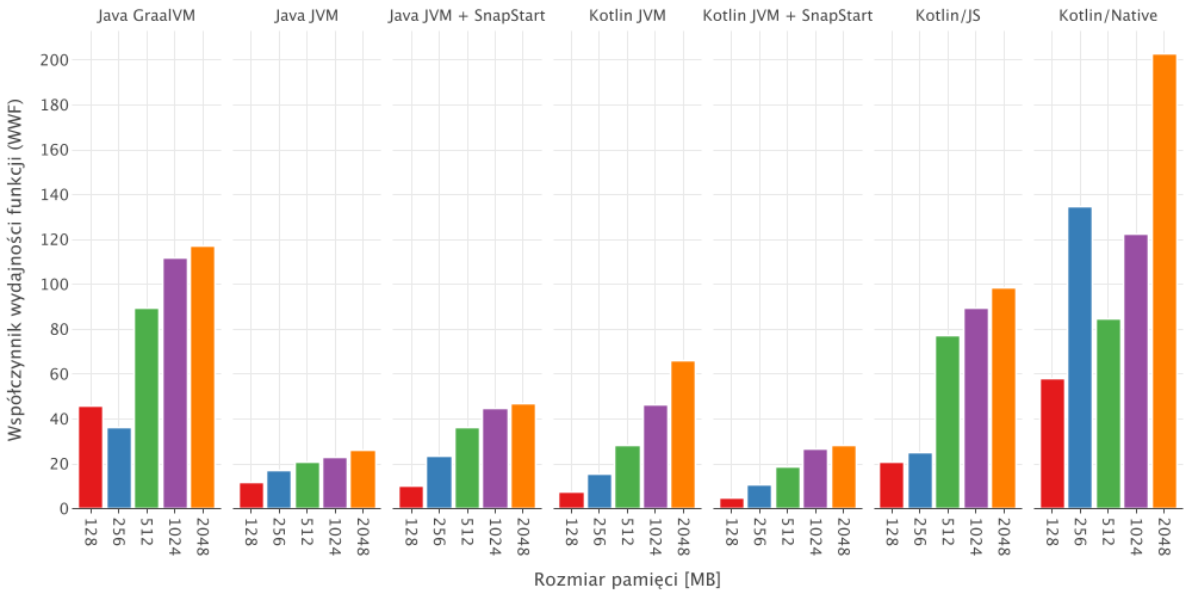
Rysunek 5.6: Czas wykonania funkcji (zimny start, 256 MB) [źródło: opracowanie własne]



Rysunek 5.7: Czas wykonania funkcji (zimny start, 1024 MB) [źródło: opracowanie własne]

5.3. Współczynnik wydajności funkcji

Kolejnym badanym kryterium jest współczynnik wydajności funkcji (WWF), opisany w Rozdziale 4.1. Pozwala on na jednoczesną analizę zarówno ciepłych, jak i zimnych startów, a wyższe wartości WWF oznaczają wyższą wydajność funkcji. Wartości WWF zostały obliczone na bazie średnich czasów wykonania podczas zimnych i ciepłych startów. W ramach obliczeń przyjęto wagi: 0,975 dla ciepłych startów oraz 0,025 dla zimnych startów. Wyniki dla poszczególnych metod i rozmiarów pamięci zostały przedstawione na Rysunku 5.8.



Rysunek 5.8: Współczynnik wydajności funkcji w zależności od rozmiaru pamięci [źródło: opracowanie własne]

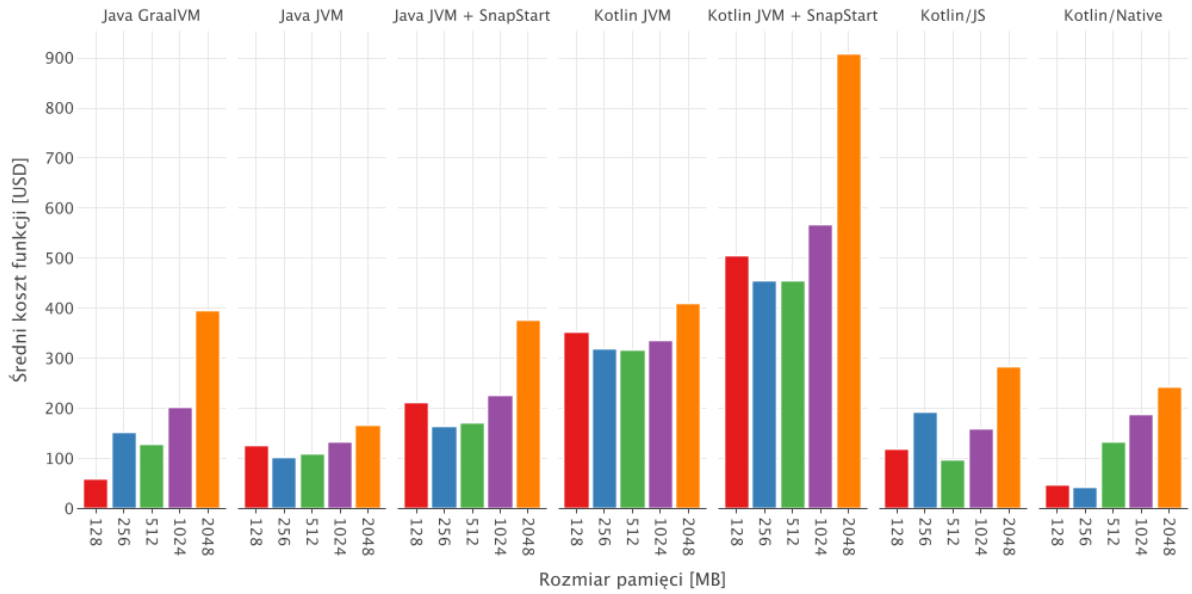
Dla funkcji Java opartej o JVM, wartość współczynnika rośnie wraz z wzrostem pamięci. Podobny trend występuje dla funkcji Kotlin JVM, gdzie jednak wzrost ten jest bardziej dynamiczny. Dla małych rozmiarów pamięci funkcje Kotlin charakteryzują się mniejszą wydajnością, jednak zmienia się to dla wyższych wartości pamięci (od 512 MB wzyż). Skuteczność użycia usługi SnapStart zależy od użytego języka programowania. W przypadku Javy, wydajność poprawiła się dla wszystkich rozmiarów pamięci oprócz 128 MB. Dla Kotlina wydajność pogorszyła się (w porównaniu z funkcjami Kotlin JVM) dla wszystkich rozmiarów pamięci, gdzie wraz z zwiększaniem wielkości pamięci różnica pogłębia się.

Znaczną poprawę wydajności dla wszystkich rozmiarów pamięci zauważono w funkcjach Java GraalVM, Kotlin/JS oraz Kotlin/Native. W przypadku dwóch pierwszych widoczny jest znaczny wzrost wartości WWF dla rozmiarów pamięci niemniejszych niż 512 MB. Dla tych rozmiarów pamięci obie te funkcje wykazują podobną wydajność, jednak z przewagą dla funkcji Java GraalVM. Przewaga ta jest znacznie większa dla mniejszych pojemności pamięci (128 MB i 256 MB). Najlepszą wartością współczynnika charakteryzuje się jednak użycie Kotlin/Native, gdzie współczynnik osiągnął najlepsze wartość dla wszystkich rozmiarów pamięci oprócz 512 MB. Jednak dla pamięci 512 MB, wartość WWF jest około 4% niższa dla Kotlin/Native niż Java GraalVM. Warte zwrócenia uwagi są jednak badania przeprowadzone

dla rozmiarów pamięci 256 MB i 2048 MB, gdzie Kotlin/Native osiągnął znacznie wyższe wartości WWF niż pozostałe funkcje. W porównaniu z Java GraalVM, czyli drugą najbardziej skuteczną metodą, wartości są wyższe o około 275% i 74%.

5.4. Koszt funkcji

Miarą, która wynika bezpośrednio z czasu działania funkcji oraz rozmiaru jej pamięci jest koszt jej działania. W ramach badania został on obliczony na bazie średnich czasów procesowania funkcji, co zostało opisane w Rodziale 4.1. W obliczeniach przyjęto 500 wywołań funkcji na sekundę. Wyniki dla poszczególnych metod i rozmiarów pamięci zostały przedstawione na Rysunku 5.9.



Rysunek 5.9: Średni koszt funkcji w zależności od rozmiaru pamięci [źródło: opracowanie własne]

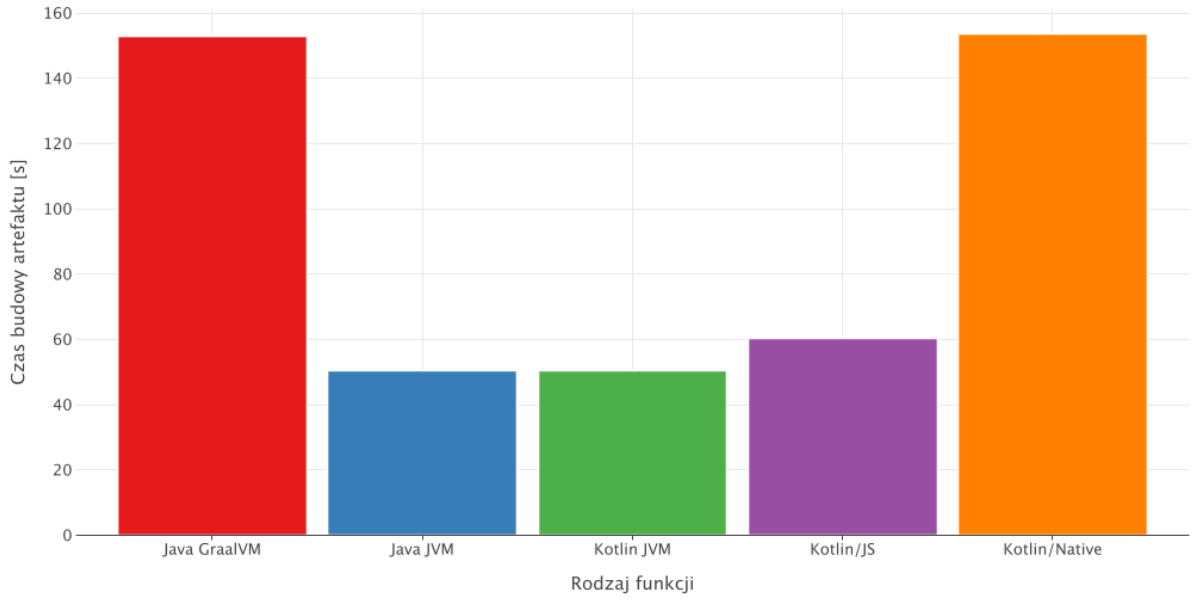
Analizując cztery funkcje oparte o JVM można zauważyć, że koszt jest najniższy dla pamięci 256 MB lub 512 MB. Interesującym faktem jest, że rozmiaru 128 MB koszt działania jest wyższy, co przeczy oczekiwaniom, że koszty maleją wraz z spadkiem wielkości pamięci, niezależnie od jej wartości. Dla funkcji bez aktywowanej usługi SnapStart koszt dla pamięci 128 MB jest nawet wyższy niż dla wielkości 512 MB. Podczas aktywnej usługi SnapStart, badane funkcje wykazują znaczny wzrost średnich kosztów wraz z wzrostem pamięci z 1024 MB do 2048 MB. Sama metoda SnapStart wpływa jednak negatywnie na koszt działania. Znaczne oddziaływanie rozmiaru pamięci na koszt funkcji jest istotnie widoczny dla Kotlin/JS. Koszt jest znacznie zróżnicowany, jednak dla pewnych przypadków (pamięć 128 MB i 512 MB) pozostaje on niższy niż w przypadku funkcji Java JVM.

Obie z badanych funkcji natywnych (Java GraalVM, Kotlin/Native) wykazują wzrost kosztów wraz z wzrostem rozmiaru pamięci. Dla każdego rozmiaru pamięci oprócz 128 MB, koszty funkcji Java GraalVM są wyższe niż dla analogicznej funkcji Java JVM. Widoczny

jest także znaczny wzrost kosztów dla pamięci 2048 MB, podobnie jak w przypadku usługi SnapStart. W przypadku użycia Kotlin/Native i pamięci 128 MB lub 256 MB możliwe jest osiągnięcie najniższych kosztów działanio (odpowiednio 48 i 43 USD). Dla większych rozmiarów najlepsze wyniki zostały osiągnięte przez Kotlin/JS (rozmiary 512 MB) i Java JVM (rozmiary 1024 MB i 2048 MB).

5.5. Wpływ na rozwój oprogramowania

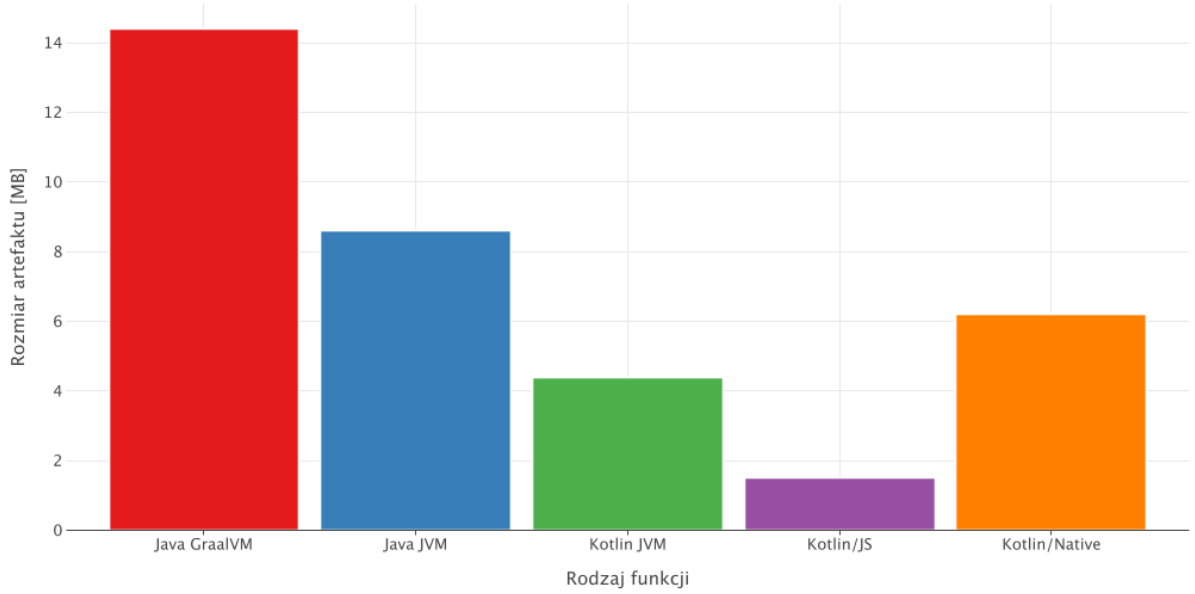
Obszarem, który został także ujęty w badaniu jest wpływ poszczególnych metod na proces rozwoju oprogramowania. Oceniono go z użyciem kryteriów, które zostały opisane w Rozdziale 4.1. W ramach rozdziału dokonano oceny metod, pod względem poszczególnych kryteriów.



Rysunek 5.10: Średni czas budowy artefaktu dla poszczególnych rodzajów funkcji [źródło: opracowanie własne]

Średni czas budowy artefaktów został przedstawiony na Rysunku 5.10. Funkcje z aktywną usługą SnapStart korzystają z artefaktów wytworzonych dla funkcji Java JVM i Kotlin JVM, dlatego nie zostały ujęte jako osobne rodzaje. Czas tworzenia plików wdrożeniowych dla funkcji Java JVM i Kotlin JVM jest podobny, co wskazuje na nieznaczny wpływ Kotlina na ten proces. W przypadku użycia Kotlin/JS translacja wymaga więcej czasu i widoczny jest jego wzrost o około 20%. Istotny jest jednak wpływ metod natywnych, gdzie wzrost wynosi już około 200%. Wyniki dla tych metod są podobne niezależnie od wybranego języka programowania (Javy lub Kotlina).

Wielkość artefaktu dla poszczególnych rodzajów funkcji został przedstawiony na Rysunku 5.11. Dla tego kryterium, istotny wpływ ma wybrany język programowania. Dla Javy wielkość plików jest większa, szczególnie dla obrazów natywnych GraalVM, gdzie osiątnięto rozmiar około 14 MB (około 67% niż analogiczny plik JAR dla funkcji Java JVM). Kotlin wykazuje się mniejszymi artefaktami, gdzie dla funkcji JVM rozmiar pliku JAR jest około 49% mniejszy



Rysunek 5.11: Wielkość artefaktu dla poszczególnych rodzajów funkcji [źródło: opracowanie własne]

niż dla analogicznego artefaktu Java. Translacja do języka JavaScript przyniosła bardzo pozytywne rezultaty gdzie osiągnięto najniższy rozmiar, około 1.5 MB. Użycie kompilacji do binarnych plików natywnych z Kotlin/Native zwiększyło rozmiar artefaktu do około 6.2 MB, czyli około 41% więcej niż plik JAR dla Kotlin JVM. Metoda ta jednak pozwoliła na budowę o wiele mniejszego artefaktu niż analogiczna kompilacja z użyciem GraalVM. Artefakt Kotlin/Native był w tym przypadku o 57% mniejszy.

Dostępność bibliotek AWS SDK dla poszczególnych rodzajów funkcji została przedstawiona w Tabeli 5.3. W przypadku funkcji Java JVM, Kotlin JVM oraz Java GraalVM dostępne są dedykowane biblioteki AWS SDK, co znacząco ułatwia integrację z usługami AWS. Sytuacja zmienia się w przypadku Kotlin/Native, dla którego AWS SDK nie jest jeszcze dostępne. Dla Kotlin/JS istnieje możliwość użycia AWS SDK dla JavaScript, jednak nie jest to rozwiązanie bezpośrednio zintegrowane z Kotlinem. Wiąże się z koniecznością dodatkowych nakładów pracy i oraz utrzymywania dodatkowego kodu, co może znacząco wpłynąć na jakość rozwoju oprogramowania.

Tabela 5.3: Dostępność AWS SDK dla rodzajów funkcji [źródło: opracowanie własne]

Rodzaj funkcji	**Dostępność AWS SDK**	**Komentarz**
Java JVM	Tak	Dedykowane AWS SDK dla Javy [5].
Kotlin JVM	Tak	Dedykowane AWS SDK dla Kotlina [7].
Java GraalVM	Tak	Wsparcie dla GraalVM poprzez AWS SDK dla Javy [5].
Kotlin/Native	Nie	AWS SDK dla Kotlina nie jest dostępne dla tej platformy. W momencie pisania pracy (maj 2025), wsparcie to jest w trakcie rozwoju [7].
Kotlin/JS	Tak	Poprzez użycie AWS SDK dla JavaScript [6]. Nie jest jednak bezpośrednio zintegrowane z Kotlinem i użycie wymaga dodatkowych nakładów pracy.

6. Wnioski i dyskusja

6.1. Odpowiedzi na pytania badawcze

W ramach podrozdziału udzielono odpowiedzi na pytania badawcze sformułowane w pracy. Odpowiedzi bazowały na wynikach opisanych w Rozdziale 5 oraz przypadku trzeciego pytania badawczego także na cechach metod opisanych w Rozdziale 3.

Odpowiedź na pierwsze pytanie badawcze

Pierwsze pytanie badawcze postawione w pracy brzmi: „Które metody optymalizacji pozwalają na najlepszą poprawę ogólnej wydajności funkcji AWS Lambda w ekosystemie Java?". W celu odpowiedzi w pracy zawarto cztery różne kryteria: czas działania funkcji podczas ciepłych startów, czas działania funkcji podczas zimnych startów, współczynnik wydajności funkcji oraz koszt działania funkcji. Badania pod ich kątem wykazały, że efektywność poszczególnych metod jest bardzo zależna od konkretnego przypadku użycia (np. często lub rzadko używane funkcje) oraz wielkości pamięci.

W przypadku funkcji o wysokiej aktywności (czyli z dominacją ciepłych startów), żadna z badanych metod nie wykazała poprawy wydajności dla każdej z badanych wielkości pamięci. Dla czasu ciepłych startów widoczny jest negatywny wpływ usługi SnapStart (zarówno w przypadku Javy, jak i Kotlina) oraz zastosowania języka interpretowanego (w przypadku użycia Kotlin/JS). Metody bazujące na kompilacji kodu do natywnych plików binarnych są najbardziej obiecujące. Są to platformy GraalVM oraz Kotlin/Native, których efektywność możemy rozpatrzyć pod kątem małych i dużych rozmiarów pamięci. Dla małych rozmiarów pamięci (128 MB oraz 256 MB) Kotlin/Native zapewnił poprawę względem klasycznej funkcji Java JVM, o odpowiednio 49% i 35%. GraalVM pozwolił na większą poprawę dla pamięci 128 MB (o 60%), jednak w przypadku pamięci 256 MB cechował się wydłużeniem czasu odpowiedzi. Dla pamięci 512 MB obie metody cechowały się dłuższym czasem procesowania niż klasyczna funkcja Java JVM, jednak GraalVM był znacznie szybszy (pogorszenie o 12%, w porównaniu z 231% dla Kotlin/Native). W przypadku większych pamięci, czyli 1024 MB i 2048 MB, GraalVM skrócił czas procesowania o odpowiednio 40% i 23%.

Zatem dla funkcji AWS Lambda o wysokiej aktywności i wysokim ponownym użyciu rozgrzanych już instancji najbardziej uniwersalnym wyborem będzie użycie GraalVM. Jednak w przypadku chęci użycia mniejszej pamięci, Kotlin/Native będzie sposobem bardziej stabilnym (ze względu na brak znacznego pogorszenia wydajności dla pamięci 256 MB). Dodatkowo, metoda ta wykazuje bardzo wysoką poprawę efektywności kosztowej. Dla rozmiarów pamięci 128 MB i 256 MB Kotlin/Native pozwolił na uzyskanie najniższych kosztów, zatem może to być istotny czynnik w wyborze metod optymalizacji przez zespoły programistyczne.

Analizując przypadek funkcji mniej aktywnych (czyli z większym udziałem zimnych startów), metody wykazywały inne właściwości niż dla ciepłych startów. Poprzez ocenę współ-

czynnika wydajności funkcji (WWF) można stwierdzić, że większość metod pozwoliły na pewną poprawę wydajności. Jedynie dla mniejszych pamięci (128 MB, 256 MB) metody jak SnapStart i Kotlin prowadziły do pogorszenia wydajności (oprócz użycia Javy z aktywnym SnapStart i pamięci 256 MB). Dla pamięci wielkości 128 MB, 256 MB ponownie najbardziej skuteczne okazały się metody oparte o kompilację do natywnych plików binarnych. Szczególną uwagę przykuwa jednak użycie Kotlin/Native, który zapewnił znacznie większy wzrost wydajności niż GraalVM. Wykazał się on najbardziej efektywną metodą dla funkcji o pamięci 128 MB, a także dla rozmiaru 256 MB, gdzie wzrost wydajności był kolosalny. Kotlin/Native osiągnał trzykrotnie większą wartość współczynnika niż GraalVM, który był drugą najbardziej skuteczną metodą. Czas uruchomienia funkcji dla pamięci 128 MB i 256 MB był także najniższy dla Kotlin/Native. Zostanie to omówione szczegółowe w ramach odpowiedzi na drugie pytanie badawcze. Dodatkowo, w ramach tych wartości pamięci możliwe było uzyskanie najniższych kosztów działania funkcji wśród wszystkich badanych konfiguracji. Zostało to otrzymane dzięki użyciu kompilacji Kotlina do natywnych plików binarnych. Wszystkie te aspekty wskazują na bardzo wysoką skuteczność tej metody w zakresie niewielkich rozmiarów pamięci (128 MB, 256 MB).

Dla aktywnych funkcji o rozmiarze pamięci od 512 MB do 2048 MB, trzy z badanych metod wykazały istotną skuteczność. Były to Kotlin/JS, Kotlin/Native oraz GraalVM. Dla pamięci 512 MB, metody wykazały zbliżoną skuteczność pod względem współczynnika wydajności funkcji (przy czym GraalVM był najbardziej wydajny). Wraz z wzrostem pamięci rosła wydajność wszystkich trzech metod, jednak istotny jest tutaj wzrost wydajności dla funkcji używających Kotlin/Native. Dla rozmiaru pamięci 1024 MB Kotlin/Native uzyskał najwyższą wartość współczynnika, o około 10% wyższą niż kolejny GraalVM. Jednak dla rozmiaru 2048 MB, różnica ta zdecydowanie zwiększyła się, gdzie Kotlin/Native osiągnął wynik o około 74% wyższy od GraalVM. Wartym uwagi jest także bardzo wysoka wydajność Kotlin/Native w konfiguracji pamięci 256 MB, gdzie było to połączenie o drugiej najwyższej wartości współczynnika. Dodatkowo, Kotlin/Native cechował się wysoką efektywnością kosztową, gdzie pozwolił na uzyskanie niższego kosztu działania niż GraalVM w prawie wszystkich z badanych rozmiarów pamięci (oprócz 512 MB).

Podsumowując, niemożliwe jest wybranie jednej najbardziej efektywnej metody optymalizacji wydajności dla wszystkich przypadków użycia funkcji AWS Lambda w ekosystemie Java. Dla funkcji o bardzo wysokiej aktywności, GraalVM byłby najbardziej uniwersalną techniką, która jednak nie jest skuteczna dla każdej konfiguracji pamięci. Podczas użycia funkcji o mniejszej aktywności i wyższym znaczeniu zimnych startów, bardzo sprawną techniką jest Kotlin/Native, który cechuje się także dobrą efektywnością kosztową, w porównaniu z alternatywnym GraalVM. Wartym uwagi jest jednak przypadek funkcji o niewielkim rozmiarze pamięci (128 MB, 256 MB) i używających Kotlin/Native. W ramach tych funkcji metoda ta pozwala na poprawę wydajności w ramach każdego z badanych aspektów. Redukuje ona czas ciepłych startów, osiąga najniższy czas działania dla zimnych startów oraz najwyższy współczynnik wydajności funkcji w ramach tych rozmiarów pamięci. Wszystko to jest możliwe przy jednocześnie najniższych z otrzymanych w badaniu kosztów działania. Z tego względu Kotlin/Native powinien być rozważany w przypadku decyzji o wyborze metody optymalizacji wydajności.

Odpowiedź na drugie pytanie badawcze

Drugim pytaniem badawczym sformułowane w pracy jest: „W jakim stopniu wybrane metody optymalizacji redukują czas zimnego startu funkcji Java w AWS Lambda?". W celu udzielenia odpowiedzi w badaniu zawarto pomiar czasu działania funkcji podczas zimnych startów. Dokładne wyniki zostały przedstawione w Rozdziale 5.2. W tym przypadku skuteczność metod była generalnie wysoka. Jedynym wyjątkiem było zastosowanie funkcji SnapStart w połączeniu z Kotliniem, co wpłynęło negatywnie na efektywność podczas zimnych startów. Dla Javy użycie SnapStart zredukowało czas zimnych startów od 10% (dla pamięci 128 MB) do 55% (dla pamięci 1024 MB). Użycie Kotlina także było skuteczne, choć tylko dla większych rozmiarów pamięci (512 MB, 1024 MB, 2048 MB), gdzie poprawa wynosiła odpowiednio 28%, 54% oraz 65%.

Najbardziej wydajnymi metodami były te oparte o kompilacje do natywnych plików binarnych (GraalVM, Kotlin/Native) oraz używające języków interpretowanych (Kotlin/JS). Porównując GraalVM oraz Kotlin/JS, pierwsza metoda była średnio bardziej skuteczna dla małych pamięci (128 MB, 256 MB). Zapewniła ona poprawę czasu odpowiedzi o odpowiednio 78% i 81%, gdy dla Kotlin/JS wynik ten wynosił odpowiednio 75% i 80%. W przypadku pamięci 512 MB, 1024 MB i 2048 MB GraalVM zredukował czas odpowiedzi o odpowiednio 82%, 82% i 80%, a Kotlin/JS zredukował czas odpowiedzi o odpowiednio 84%, 84% i 86%. Jednak dla obu tych metod róznice w skuteczności są niewielkie.

Najefektywniejszą metodą okazał się jednak Kotlin/Native. Dla każdej z badanych wielkości pamięci pozwolił on na największą redukcję czasu działania. Wynosiła ona od 88% (dla pamięci 128 MB) do 93% (dla pamięci 1024 MB i 2048 MB) w porównaniu z bazową funkcją Java JVM. Dla rozmiarów pamięci 128 MB i 256 MB, funkcje z Kotlin/Native charakteryzowały się zimnymi startami odpowiednio o około 45% i 51% niższymi w porównaniu do GraalVM. Z kolei dla konfiguracji z pamięcią 512 MB, 1024 MB i 2048 MB, czasy zimnego startu były odpowiednio o 50%, 48% i 48% krótsze niż dla Kotlin/JS.

Odpowiedź na trzecie pytanie badawcze

Trzecim pytaniem badawczym, które zostało uwzględnione w pracy, jest: „Jakie kompromisy w procesie rozwoju oprogramowania wiążą się implementacją poszczególnych metod optymalizacji wydajności funkcji Java w AWS Lambda?". Wpływ na rozwój oprogramowania to istotny czynnik, który powinien być wzięty pod uwagę przez zespoły programistyczne podczas wyboru odpowiedniej metody optymalizacji wydajności. Znaczenie może zostać ocenione pod kątem różnych kryteriów jak: czas budowy artefaktu, jego wielkość, dostępność narzędzi SDK (ang. Software Development Kit) czy kosztem działania. W ramach oceny wszystkie metody są porównywane do bazowej implementacji Java działającej z użyciem JVM.

Podczas użycia usługi SnapStart programista dalej wykorzystuje ten sam artefakt i platformę, jak w funkcji bazowej. Zatem nie ma ona wpływu na czynniki jak czas budowy, wielkość artefaktu czy dostępność SDK. Istotny jest jednak wpływ na koszt funkcji, który był wyższy dla każdego z badanych rozmiarów pamięci oraz dla obu języków (Java i Kotlin). W trakcie tworzenia systemów korzystających z SnapStart programista powinien uwzględnić jednak kilka aspektów związanych z stanem funkcji (co zostało opisane w Rozdziale 3.1). Wpływa to na połączenia sieciowe funkcji (szczególnie ważne podczas integracji z innymi

serwisami) oraz generowania wartości unikalnych (jak identyfikatory). Wynika to z sposobu działania SnapStart, który tworzy migawki stanu funkcji, wykorzystywane później w wielu wywołaniach.

Użycie GraalVM niesie za sobą pewne kompromisy dla rozwoju oprogramowania, które są jednak umiarkowane. Istotny jest wpływ na tworzony artefakt funkcji, którego budowa znacznie wydłuża się (w porównaniu z tworzeniem klasycznego pliku JAR), a rozmiar zwiększa się. GraalVM poprzez zmianę w sposobie kompilacji zmienia także zachowanie tworzonego kodu. Wpływa to na np. mechanizm refleksji, którego użycie wymaga od programisty dodatkowej pracy związanej z odpowiednim oznaczeniem metod i klas. Możliwe jest jednak dalej użycie AWS SDK, by integrować funkcje z innymi usługami. Mimo poprawy wydajności kosztów, ważnym aspektem jest jednak wzrost kosztów dla większych pamięci funkcji (1024 MB, 2048 MB).

Poprzez użycie języka Kotlin programista zyskusje szeroke możliwości związane z projektem Kotlin Multiplatform. Język ten jest zbliżony do Javy i w przypadku użycia razem z JVM, nie niesie za sobą znacznych kompromisów w procesie rozwoju oprogramowania. Należy jedynie wspomnieć o negatywnym wpływie na koszt działania, który średnio był wyższy niż dla analogicznych funkcji w Javie. Większy wpływ na tworzenie aplikacji mają metody oparte o Kotlin/JS oraz Kotlin/Native. Pierwszym z nich jest czas tworzenia artefaktu, który jest wyższy dla obu metod (jednak znacznie wyższy dla Kotlin/Native). Warto jednak zaznaczyć, że rozmiar artefaktu był mniejszy dla wszystkich funkcji opartych o Kotlin, w porównaniu z klasyczną Javą. Pewnym kompromisem są także integracje z zewnętrznymi serwisami, którą mogą być utrudnione ze względu na dostępność AWS SDK. Jest ono niedostępne dla Kotlin/Native oraz dostępne pośrednio dla Kotlin/JS, co wymaga dodatkowej integracji. Ograniczeniami dla tych sposobów są także ograniczone możliwości użycia mechanizmu refleksji, podobnie jak w przypadku GraalVM.

6.2. Zagrożenia trafności wyników

Trafność wnioskowania

Istotnym zagrożeniem dla wniosków wyciągniętych na podstawie eksperymentu może być niska moc statystyczna wyników. Badanie opierało się na dużej, jednak określonej, liczbie wywołań dla każdej z funkcji i konfiguracji. Mimo przeprowadzenia wielu testów, zmienność środowiska chmurowego i potencjalne losowe zmiany w czasach odpowiedzi mogły wpłynąć na wyniki. W ramach badania starano się zminimalizować ten wpływ, np. poprzez brak integracji z zewnętrznymi serwisami. Aby jeszcze bardziej zmniejszyć to ryzyko możliwe byłoby wydłużenie czasu eksperymentu i regularne wywoływanie funkcji przez dłuższy okres (np. kilku dni). Proces ten wymaga jednak dłuższego badania, które przekracza zakres aktualnej pracy.

Trafność wewnętrzna

Potencjalnym czynnikiem, który mógłby negatywnie wpłynąć na trafność wewnętrzną eksperymentu jest dynamiczna natura środowiska chmurowego. Inne procesy działające w tym samym czasie w infrastrukturze AWS i niezwiązane z badaniem, mogły teoretycznie wpłynąć

na wydajność badanych funkcji (np. poprzez chwilowe obciążenie sieci lub współdzielonych zasobów).

Innym aspektem są potencjalne błędy lub różnice w implementacji funkcji. W przypadku języka Kotlin postarano się zminimalizować ten wpływ poprzez współdzielenie kodu między wszystkiem metody oparte o ten język. Jednak poszczególne metody stosują różne biblioteki i frameworki, których wybór jest ograniczony poprzez daną metodę (np. wsparcie dla kompilacji natywnej). Sama ich analiza i porównanie może być jednak wymagające, co stanowi potencjalny kierunek rozwoju pracy.

Trafność konstruktu

Trafność konstruktu mogła być zagrożona ze poprzez przyjęte w pracy miary wydajności oraz kompromisów w procesiu rozwoju oprogramowania. Wydajność została zmierzona za pomocą czasu wykonania (ciepłe i zimne starty), kosztu oraz autorskiego współczynnika WWF. Trafność ta może być jednak zagrożona ze względu na zróżnicowane proporcje ciepłych i zimnych startów w systemach opartych o funkcje AWS Lambda. W ramach badania przyjęto jedynie pojedyncze ich propocje, zatem istnieje ryzyko niezawarcia w badaniu szczególnych przypadków użycia badanej usługi.

Wpływ na proces rozwoju oprogramowania oceniono na podstawie czasu budowy artefaktu, jego wielkości, dostępności AWS SDK, kosztu działania funkcji oraz innych czynników opisanych w Rozdziale 3. Opis procesu rozwoju oprogramowania jedynie w tych aspektach jest jednak uproszczony. Niesie to zatem ryzyko nieprawidłowej oceny konkretnych metod. Aby zapewnić lepszą jakość badania pod tym względem, możliwa byłaby analiza konkretnych projektów korzystających z danych metod. Jest to jednak utrudnione ze względu na niewielką popularność części z nich.

Trafność zewnętrzna

W badaniu zaimplementowano bardzo ogólne zadanie obliczeniowe czyli mnożenie macierzy oraz powstrzymano się od integracji zewnętrznych usług. Miało to na celu ograniczenie wpływu zewnętrznych czynników na czas działania funkcji. Jak wykazano w przeglądzie literatury dodatkowe serwisy są jednak bardzo częstym elementem architektur bezserwerowych. Z tego powodu zaimplementowane operacje mogą być niewystarczające aby przełożyć wyniki badania na przypadki innych systemów. Systemy te mogą opierać się np. o wywołania dodatkowych API, które nie zostały uwzględnione w eksperymencie.

AWS Lambda zapewnia istotną abstrakcję na infrastrukturę chmurową, która jednak dalej odgrywa istotną rolę w wydajności funkcji. Eksperyment przeprowadzono w jednym regionie AWS i dla jednej architektury procesora (x86-64). Jak wykazał przegląd literatury, wydajność może różnić się w zależności od regionu geograficznego oraz architektury (np. ARM64). Zatem wyniki mogą nie być w pełni reprezentatywne dla wszystkich dostępnych konfiguracji.

Zakończenie

Bibliografia

[1] Dokumentacja Kotlin. https://kotlinlang.org/docs/home.html. [Dostęp: 15.05.2025].

[2] Dokumentacja narzędzia http4k Core. https://www.http4k.org/ecosystem/http4k/. [Dostęp: 16.05.2025].

[3] Repozytorium kodu źródłowego kompilatora Kotlin IR. https://github.com/JetBrains/kotlin/tree/master/compiler/ir/. [Dostęp: 16.05.2025].

[4] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, Feb. 2020. USENIX Association.

[5] Amazon Web Service. Repozytorium AWS SDK dla Java. https://github.com/aws/aws-sdk-java-v2, 2025. Dostęp: 20.05.2025.

[6] Amazon Web Service. Repozytorium AWS SDK dla JavaScript. https://github.com/aws/aws-sdk-js-v3, 2025. Dostęp: 20.05.2025.

[7] Amazon Web Service. Repozytorium AWS SDK dla Kotlin. https://github.com/awslabs/aws-sdk-kotlin, 2025. Dostęp: 20.05.2025.

[8] Amazon Web Services. Aws lambda developer guide. https://docs.aws.amazon.com/lambda/latest/dg/welcome.html. [Dostęp: 13.05.2025].

[9] Amazon Web Services. Aws lambda snapstart developer guide. https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html. [Dostęp: 13.05.2025].

[10] Amazon Web Services. Aws sdk for kotlin developer guide. https://docs.aws.amazon.com/sdk-for-kotlin/latest/developer-guide/home.html. [Dostęp: 13.05.2025].

[11] Amazon Web Services. Amazon ec2 documentation, 2025. Dostęp 21.03.2025 z https://docs.aws.amazon.com/ec2/.

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

[13] Amazon Web Services. Aws lambda documentation. https://docs.aws.amazon.com/lambda/latest/dg/, 2025. Dostęp: 21.03.2025.

[14] Amazon Web Services. AWS Lambda pricing. https://aws.amazon.com/lambda/pricing/, 2025. Dostęp: 20.05.2025.

[15] D. Bardsley, L. Ryan, and J. Howard. Serverless performance and optimization strategies. In *2018 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 19–26, 2018.

[16] L. Bergstrom, M. Fluet, J. H. Reppy, and N. Sandler. Practical inlining of functions with free variables. *CoRR*, abs/1306.1919, 2013.

[17] D. Beronić, L. Modrić, B. Mihaljević, and A. Radovan. Comparison of structured concurrency constructs in java and kotlin - virtual threads and coroutines. In *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1466–1471, 2022.

[18] M. Brooker, A. C. Catangiu, M. Danilov, A. Graf, C. MacCarthaigh, and A. Sandu. Restoring uniqueness in microvm snapshots, 2021.

[19] 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.

[20] B. Cartaxo, G. Pinto, and S. Soares. Rapid reviews in software engineering, 2020.

[21] A. P. Cavalheiro and C. Schepke. Exploring the serverless first strategy in cloud application development. pages 89 – 94, 2023. Cited by: 0.

[22] 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.

[23] 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.

[24] R. Cordingly, S. Xu, and W. Lloyd. Function memory optimization for heterogeneous serverless platforms with cpu time accounting. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*, pages 104–115, 2022.

[25] 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.

[26] Y. Cui. Should "serverless"just mean "function-as-a-service"?, October 2024. Dostęp: 28.04.2025.

[27] 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.

[28] Datadog. The state of serverless. Web page, August 2023. Dostęp: 31.05.2025.

[29] 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.

[30] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup. A review of serverless use cases and their characteristics, 2021.

[31] 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.

[32] N. Ekwe-Ekwe and L. Amos. The state of faas: An analysis of public functions-as-a-service providers. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, pages 430–438. IEEE, July 2024.

[33] 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.

[34] 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.

[35] P. Gajek and M. Plechawska-Wójcik. Performance comparison of the java and kotlin programming languages based on an auto-scroller mobile game. *Journal of Computer Sciences Institute*, 33:285–291, Dec. 2024.

[36] B. C. Ghosh, S. K. Addya, N. B. Somy, S. B. Nath, S. Chakraborty, and S. K. Ghosh. Caching techniques to improve latency in serverless architectures. In *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 666–669, 2020.

[37] L. A. Goodman. Snowball Sampling. *The Annals of Mathematical Statistics*, 32(1):148 – 170, 1961.

[38] GraalVM Team. *Reflekcja w obrazie natywnym*, 2024. Dostęp: 13.05.2025.

[39] D. Ivanov and A. Petrova. Serverless computing architectures and applications in aws. *MZ Journal of Artificial Intelligence*, 1(1):1–10, Jun. 2024.

[40] V. Ivanovichev. Kotlin Native Runtime for AWS Lambda. https://github.com/trueangle/kotlin-native-aws-lambda-runtime. [Dostęp: 16.05.2025].

[41] 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.

[42] JetBrains. Kotlin Multiplatform Development. https://www.jetbrains.com/help/kotlin-multiplatform-dev. [Dostęp: 15.05.2025].

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

[44] B. Kehoe. Serverless is a state of mind, 2018. Dostęp: 28.04.2025.

[45] 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.

[46] B. Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele Univ.*, 33, 08 2004.

[47] B. Kitchenham, L. Madeyski, and D. Budgen. How should software engineering secondary studies include grey material? *IEEE Transactions on Software Engineering*, 49(2):872–882, 2023.

[48] 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.

[49] 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.

[50] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004.

[51] P. Leitner, E. Wittern, J. Spillner, and W. Hummer. A mixed-method empirical study of function-as-a-service software development in industrial practice. *Journal of Systems and Software*, 149:340–359, 2019.

[52] 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.

[53] 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.

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

[55] M. Nazari, S. Goodarzy, E. Keller, E. Rozner, and S. Mishra. Optimizing and extending serverless platforms: A survey. In *2021 Eighth International Conference on Software Defined Systems (SDS)*, pages 1–8, 2021.

[56] 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.

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

[58] 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.

[59] 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.

[60] Quarkus. *Dokumentacja Quarkus*. Red Hat, 2025. Dostęp: 13.05.2025.

[61] A. Raza, I. Matta, N. Akhtar, V. Kalavri, and V. Isahagian. SoK: Function-As-A-Service: From An Application Developer's Perspective. *Journal of Systems Research*, 1(1), 2021.

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

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

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

[65] Stack Overflow. 2024 developer survey - most popular technologies: Programming, scripting, and markup languages. Web page, 2024. Dostęp: 31.05.2025.

[66] 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.

[67] The SpotBugs team. Spotbugs static analysis for java. https://spotbugs.github.io/. Dostęp: 13.05.2025.

[68] L. van Donkersgoed. Postnl's serverless journey: Business overview, oct 2023. Dostęp: 04.05.2025.

[69] 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.

[70] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, and A. Iosup. Serverless is more: From paas to present cloud computing. *IEEE Internet Computing*, 22(5):8–17, 2018.

[71] 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.

[72] Y. Xiao and M. Watson. Guidance on conducting a systematic literature review. *Journal of Planning Education and Research*, 39(1):93–112, 2019.

[73] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 30–44, New York, NY, USA, 2020. Association for Computing Machinery.

[74] M. Šipek, B. Mihaljević, and A. Radovan. Exploring aspects of polyglot high-performance virtual machine graalvm. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1671–1676, 2019.

[75] M. Šipek, D. Muharemagić, B. Mihaljević, and A. Radovan. Enhancing performance of cloud-based software applications with graalvm and quarkus. In *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1746–1751, 2020.

Spis rysunków

1.1 Przykładowa implementacja funkcji AWS Lambda w języku Java [źródło: opracowanie własne] . . . 11
1.2 Cykl życia funkcji AWS Lambda [źródło: opracowanie własne na bazie dokumentacji AWS Lambda [13]] . . . 12
1.3 Proces uruchomienia AWS Lambda w kontekście zimnych i ciepłych startów [źródło: opracowanie własne na bazie dokumentacji AWS Lambda [13]] . . . 13

2.1 Proces selekcji literatury [źródło: opracowanie własne] . . . 20

3.1 Proces uruchomienia AWS Lambda z użyciem metody SnapStart [źródło: opracowanie własne] . . . 38
3.2 Uproszczony proces budowy obrazu natywnego GraalVM [źródło: opracowanie własne] . . . 40
3.3 Przykładowa struktura pojektu Kotlin Multiplatform [źródło: opracowanie własne] 44
3.4 Proces transformacji kodu Kotlin do kodu JavaScript [źródło: opracowanie własne na bazie repozytorium kompilatora IR [3]] . . . 45

4.1 Wizualizacja procesu realizowanego przez badane funkcje AWS Lambda [źródło: opracowanie własne] . . . 52
4.2 Architektura środowiska badawczego [źródło: opracowanie własne] . . . 57

5.1 Średni czas wykonywania funkcji (ciepły start) z użyciem wybranych metod w zależności od rozmiaru pamięci [źródło: opracowanie własne] . . . 59
5.2 Czas wykonania funkcji (ciepły start, 256 MB) [źródło: opracowanie własne] . 61
5.3 Czas wykonania funkcji (ciepły start, 1024 MB) [źródło: opracowanie własne] . 61
5.4 Średni czas wykonywania funkcji (zimny start) dla rozmiarów pamięci: 512 MB, 1024 MB, 2048 MB [źródło: opracowanie własne] . . . 61
5.5 Średni czas wykonywania funkcji (zimny start) dla rozmiarów pamięci: 128 MB, 256 MB [źródło: opracowanie własne] . . . 63
5.6 Czas wykonania funkcji (zimny start, 256 MB) [źródło: opracowanie własne] . 63
5.7 Czas wykonania funkcji (zimny start, 1024 MB) [źródło: opracowanie własne] . 63
5.8 Współczynnik wydajności funkcji w zależności od rozmiaru pamięci [źródło: opracowanie własne] . . . 64
5.9 Średni koszt funkcji w zależności od rozmiaru pamięci [źródło: opracowanie własne] . . . 65
5.10 Średni czas budowy artefaktu dla poszczególnych rodzajów funkcji [źródło: opracowanie własne] . . . 66

5.11 Wielkość artefaktu dla poszczególnych rodzajów funkcji [źródło: opracowanie własne] 67

Spis tabel

2.1 Liczba wybranych prac w zależności od etapu selekcji [źródło: opracowanie własne] 21
2.2 Prace wybrane w ramach przeglądu literatury [źródło: opracowanie własne] . . 22

4.1 Zewnętrzne zależności wykorzystane w badanych funkcjach 53
4.2 Biblioteki oraz frameworki użyte podczas implementacji badanych funkcji . . . 54
4.3 Środowiska wykonawcze użyte w badanych funkcjach AWS Lambda 56

5.1 Porównanie średnich czasów działania funkcji podczas ciepłego startu względem funkcji bazowej [źródło: opracowanie własne] 60
5.2 Porównanie średnich czasów działania funkcji podczas zimnego startu względem funkcji bazowej [źródło: opracowanie własne] 62
5.3 Dostępność AWS SDK dla rodzajów funkcji [źródło: opracowanie własne] . . . 68