AGH Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Wydział Fizyki i Informatyki Stosowanej

Praca magisterska

Paweł Augustyn

kierunek studiów: **informatyka stosowana**
specjalność: **grafika komputerowa i przetwarzanie obrazów**

Tworzenie aplikacji w technologii FaaS

Opiekun: **dr inż. Antoni Dydejczyk**

Kraków, październik 2020

Oświadczenie studenta

Uprzedzony(-a) o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (tj. Dz.U. z 2018 r. poz. 1191 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony(- a) o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.) „Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta.", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Jednocześnie Uczelnia informuje, że zgodnie z art. 15a ww. ustawy o prawie autorskim i prawach pokrewnych Uczelni przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli Uczelnia nie opublikowała pracy dyplomowej w terminie 6 miesięcy od dnia jej obrony, autor może ją opublikować, chyba że praca jest częścią utworu zbiorowego. Ponadto Uczelnia jako podmiot, o którym mowa w art. 7 ust. 1 pkt 1 ustawy z dnia 20 lipca 2018 r. – Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.), może korzystać bez wynagrodzenia i bez konieczności uzyskania zgody autora z utworu stworzonego przez studenta w wyniku wykonywania obowiązków związanych z odbywaniem studiów, udostępniać utwór ministrowi właściwemu do spraw szkolnictwa wyższego i nauki oraz korzystać z utworów znajdujących się w prowadzonych przez niego bazach danych, w celu sprawdzania z wykorzystaniem systemu antyplagiatowego. Minister właściwy do spraw szkolnictwa wyższego i nauki może korzystać z prac dyplomowych znajdujących się w prowadzonych przez niego bazach danych w zakresie niezbędnym do zapewnienia prawidłowego utrzymania i rozwoju tych baz oraz współpracujących z nimi systemów informatycznych.

……

(czytelny podpis studenta)

Kraków, 11 października 2020

Tematyka pracy magisterskiej i praktyki dyplomowej Pawła Augustyna, studenta drugiego roku studiów drugiego stopnia na kierunku informatyka stosowana, specjalności grafika komputerowa i przetwarzanie obrazów.

Temat pracy magisterskiej: **Tworzenie aplikacji w technologii FaaS**

Opiekun pracy: dr inż. Antoni Dydejczyk
Recenzenci pracy:
Miejsce praktyki dyplomowej: WFiIS AGH, Kraków

Program pracy magisterskiej i praktyki dyplomowej

1. Wybór tematyki oraz omówienie realizacji pracy magisterskiej z opiekunem.
2. Praktyka dyplomowa:
    - rozpoznanie pośród obecnych na rynku dostawców publicznych chmur obliczeniowych,
    - określenie zakresu pracy,
    - dobór technologii do stworzenia przykładowych aplikacji,
    - szczegółowe planowanie dalszych działań,
    - sporządzenie sprawozdania z praktyki.
3. Zapoznanie się z usługami działającymi w modelu FaaS dostępnymi w ramach ofert publicznych chmur obliczeniowych.
4. Stworzenie przykładowych aplikacji prezentujących potencjalne zastosowania opisywanego w ramach pracy modelu.
5. Przeprowadzenie testów wydajnościowych dla jednej ze stworzonych aplikacji oraz analiza uzyskanych wyników.
6. Opracowanie redakcyjne pracy.

Termin oddania w dziekanacie:

....
(podpis kierownika katedry) (podpis opiekuna)

Ocena promotora

Ocena recenzenta

Spis treści

1. Wstęp

Od wielu lat informatyka jest jedną z najprężniej rozwijających się branży w szeroko pojętej dziedzinie technologii. Z racji szybkiego tempa rozwoju, wiele rozwiązań będących swego rodzaju nowością i powiewem świeżości w ciągu kilka lat staje się przestarzałe, przez co są omijane szerokim łukiem przez potencjalnych użytkowników. Wydawać by się mogło, że w głównej mierze tyczy się to użytkowników końcowych i ich nieustannie rosnących oczekiwań względem tworzonych dla nich rozwiązań. Okazuje się, że potencjalne bolączki wynikające ze starzenia się takich rozwiązań dotyczą również ich twórców.

Każdy tworzony program jest oparty na zdefiniowanej w trakcie projektowania architekturze. Pierwszym typem wykorzystywanym do tworzenia oprogramowania był model aplikacji monolitycznych. Cała funkcjonalność biznesowa mieściła się w pojedynczej jednostce logicznej, jaką jest np. plik wykonywalny i w takiej też formie następowała dystrybucja stworzonego dzieła do docelowych użytkowników. Było to pierwsze podejście do tematu tworzenia oprogramowania, które zostało rozpowszechnione na tak szeroką skalę. Pomimo swoich wad, architektura ta po zastosowaniu pewnych poprawek wciąż znajduje swoje zastosowanie.

Kolejnym podejściem zastosowanym w projektowaniu architektury aplikacji jest szeroko pojęta architektura mikrousług (ang. microservices). Zakłada ono rozdzielenie poszczególnych funkcjonalności pomiędzy mniejsze części nazywane serwisami, które komunikują się ze sobą przy użyciu ściśle zdefiniowanych definicji API. Komunikacja ta najczęściej odbywa się z użyciem protokołów niezależnych od technologii, na przykład http. Wartym wspomnienia jest fakt, że wprowadzenie pojęcia mikrousług spowodowało odejście od tradycyjnej formy dystrybucji aplikacji w formie programów instalowanych na maszynach klientów na rzecz aplikacji webowych udostępnianych w formie portali internetowych. Poprzez znacznie mniejszy stopień skomplikowania pojedynczego serwisu architektura ta umożliwia znacznie szybsze wdrażanie nowych funkcjonalności co daje możliwość efektywniejszego spełniania i realizowania oczekiwań klientów. Nie jest to jednak rozwiązanie idealne, gdyż posiada swoje wady, których w architekturze monolitycznej próżno szukać. O ile dużym zyskiem jest znaczne uproszczenie organizacji kodu w ramach pojedynczej usługi, o tyle końcowy produkt jako całość staje się bardziej złożony, a poziom skomplikowania

testowania zależności pomiędzy poszczególnymi komponentami rośnie. Pomimo tego, architektura ta jest obecnie niepodzielnym liderem i zdecydowana większość nowo tworzonych rozwiązań właśnie na niej się opiera.

Jak można zauważyć na podstawie powyższych dwóch przykładów, widoczna jest tendencja do zmniejszania logicznych części i rozdzielania ich jako osobne jednostki logiczne, które składają się w finalny produkt. Nie inaczej jest w przypadku architektury serverless, której zastosowanie oznacza pójście w tym kierunku o krok dalej. Coraz mniejsze komponenty zyskują w niej swoją własną tożsamość poprzez wydzielenie ich do osobnych bytów, uruchamianych i zarządzanych niezależnie od pozostałych. W przypadku tej technologii tymi komponentami są funkcje, których odpowiedzialność powinna być prosta, a z założenia czasy ich wykonania są krótkie, co częściowo wynika ze stosunkowo niedużej złożoności.

Dzięki temu, że każdy komponent jest niezależnym bytem, twórca danej funkcjonalności może dostosować konfigurację danej funkcji do pełnionych przez nią zastosowań. Każda z funkcji uruchamiana jest w określonym zwirtualizowanym środowisku zarządzanym przez dostawcę chmury obliczeniowej, a po stronie użytkownika jest zdefiniowanie, jaka ilość zasobów takich jak czas procesora czy ilość pamięci operacyjnej ma być dostępna w ramach jej uruchomienia. Zwiększenie przypisanych zasobów wpływa na zwiększenie kosztów związanych z uruchomieniem danej funkcji, ale idąca za tym zaleta w postaci niespotykanej dotąd skalowalności takiego rozwiązania jest tutaj zdecydowanie dominująca. Szerszy opis tych oraz pozostałych cech modelu serverless stanie się dominującą częścią tej pracy.

Nieważne jak dokładnym byłby taki opis, zrozumienie tego modelu na podstawie samych definicji może okazać się trudne dla części czytelników. Celem niniejszej pracy jest dotarcie do jak najszerszego grona odbiorców, co zostanie poparte oparciem teoretycznego opisu architektury o odpowiednie przykłady. Mają one za zadanie pokazać czytelnikowi, jakie zagadnienia związane z projektowaniem aplikacji można rozwiązać przy użyciu architektury serverless w efektywny, prosty, a przede wszystkim opłacalny dla biznesu sposób. Dla wyrównania szans nie przedstawione zostaną również kontrprzykłady, które pozwolą pokazać, że rozwiązanie to nie stanowi panaceum w zagadnieniu projektowaniu aplikacji i istnieją kryteria, których istnienie może spowodować odrzucenie wykorzystania tego modelu.

Serverless sam w sobie nie istniałby, gdyby nie dostawcy publicznych chmur obliczeniowych chcący utrzymywać i dostarczać swoim klientom tak proste,

a jednocześnie zaawansowane narzędzie. W przypadku, gdy użytkownik nie dokonał jeszcze decyzji, z oferty którego dostawcy skorzystać, należy dostarczyć mu odpowiednie do tego kryterium. O ile możliwości konfiguracji poszczególnych rozwiązań zdają się być ważnym aspektem w podejmowaniu takiej decyzji, zdecydowanie ważniejszym czynnikiem przy podejmowaniu decyzji o skorzystaniu z danej oferty będą kwestie finansowe. Przy odpowiednim zrównoważeniu tych czynników oraz analizie przygotowanych scenariuszy użycia danych rozwiązań opracowane zostaną modele, które pozwolą pokazać, czy w danej sytuacji serverless jest jedynym słusznym wyborem.

2. Opis technologii Serverless

2.1. Zarys historyczny

Pierwsze wzmianki o modelu Serverless pojawiły się wraz z rokiem 2006. Wtedy to firma Zimki jako pierwsza udostępniła platformę umożliwiającą uruchamianie własnych fragmentów kodu w modelu łudząco przypominającym coś, co dziś nazywamy modelem funkcja jako serwis (ang. *function as a service*). [1] Jest to pierwszy znany serwis udostępniający funkcjonalność uruchamiania kodu na zewnętrznej platformie programistycznej, której model naliczania kosztów opierał się na takich metrykach jak wykorzystana ilość transferu danych, ilość zajętego miejsca na dysku czy też czas trwania wykonywanych operacji. Rozwój systemu przebiegał sprawnie i dosyć szybko zaczął cieszyć się stosunkowo dużym, jak na tamte czasy, zainteresowaniem ze strony klientów. Firmie Fotango (właścicielowi platformy) zależało jednak na przyciągnięciu jeszcze większej liczby klientów oraz szerszego spopularyzowania rozwiązania.

Zdecydowano, iż najlepszym rozwiązaniem powyższego problemu będzie uczynienie projektu Zimki otwartym oprogramowaniem. Miało to zachęcić do opracowania podobnych rozwiązań konkurentom. Z jednej strony mogło wydawać się to nieracjonalną decyzją biznesową, jednak w zamyśle zarządu firmy Fotango miało to wpłynąć pozytywnie nie tylko na rozwój rynku platform podobnych do Zimki, ale przede wszystkim zachęcić większą liczbę twórców oprogramowania do wykorzystywania tychże rozwiązań. Wielu potencjalnych klientów nie decydowało się bowiem na korzystanie z takiego rozwiązania z powodu swego rodzaju monopolu platformy. Z racji jej unikalności oraz braku konkurencji obawiano się tzw. blokady dostawcy (ang. *vendor lock-in*), czyli uzależnienia się od korzystania z konkretnego rozwiązania. Brak alternatywy mógł pociągnąć za sobą poważne konsekwencje, gdyż z perspektywy klientów mogło dojść do nieprzyjemnej sytuacji, kiedy z racji kontrolowania rynku firma mogłaby windować cenę takiego produktu, nieproporcjonalnie do oferowanych przez nią możliwości. Wydawało się, że jest to idealne rozwiązanie tego problemu, gdyż wzmianki o tej zmianie zostały ciepło przyjęte przez użytkowników.

Pomysł o otwarciu oprogramowania nie spodobał się jednak firmie matce przedsiębiorstwa Fotango, firmie Canon. Według zarządu firmy, podejście to nie zgadzało się z wartościami reprezentowanymi przez spółkę. Decyzja o przejściu na model otwartego oprogramowania miała zostać ogłoszona podczas konferencji Open Source Software Conference (OSCON) w 2007 roku. Fotango było jednym z głównym sponsorów całego wydarzenia, a tytuł ten dzieliło razem z takimi gigantami jak Google, Microsoft czy Intel. Tuż przed rozpoczęciem wydarzenia Canon zaprotestowało wykonaniu tego ruchu, co niemalże storpedowało przebieg całej konferencji. Otwarcie platformy miało być bowiem jedną z największych nowości zaprezentowanych w ramach tego wydarzenia. Pomimo protestu, szef projektu Zimki potwierdził informację o otwarciu oprogramowania, a tuż po jej ogłoszeniu prezes firmy Fotango podał się do dymisji. Następnie, na przestrzeni kilku miesięcy, wszelkie projekty prowadzone przez Zimki zostały zamknięte, co poskutkowało zamknięciem prac nad platformą. Z perspektywy czasu można stwierdzić, że decyzja ta mogła być przełomowa, a to, jak obecnie mógłby wyglądać rynek dostawców podobnych rozwiązań, gdyby Canon nie zdecydował się na zamknięcie tego projektu, pozostaje tylko w domysłach.

To, czego nie zauważyła firma Canon w produkcie swojej spółki „córki", dostrzegło Google uruchamiając w 2008 roku rozwiązanie Google App Engine. Nie jest to co prawda rozwiązanie działające dokładnie w modelu funkcja jako serwis, ale od początku swojego istnienia spełniało jedno z wprowadzonych przez model ten założeń, czyli dodania abstrakcji pomiędzy programem stworzonym przez dewelopera oraz infrastrukturą, na której program ten miałby zostać uruchomiony. Platforma ta umożliwia uruchamianie aplikacji webowych i dostarczanie ich klientom końcowym bez zaprzątania sobie głowy tym, na jakich maszynach kod ten będzie uruchamiany, organizowaniem okien czasowych na przeprowadzenie aktualizacji systemu operacyjnego czy też dostosowania ilości maszyn obsługujących dany serwis w zależności od przychodzącego z zewnątrz ruchu. W przeciwieństwie do rozwiązania Zimki produkt ten zyskał znacznie większą popularność, a co najważniejsze – istnieje do dziś. Google App Engine to serwis, który znacząco wpłynął na popularyzację publicznych chmur obliczeniowych, a dodatkowo dalej dzierży on tytuł jednego z najczęściej wykorzystywanych serwisów udostępnianych w ramach platformy Google Cloud. Wpływa na to jego prostota oraz fakt, że programista tworzący dane

rozwiązanie nie musi na pierwszym miejscu rozważać tego, w jakim środowisku uruchomieniowym jego produkt będzie funkcjonował.

Serwisem, który wpłynął na niesamowitą popularyzację rozwiązań funkcja jako serwis jest niewątpliwie Lambda udostępniana w ramach chmury Amazon Web Services. W momencie wprowadzenia serwisu w 2014, AWS już wtedy cieszył się tytułem lidera na rynku rozwiązań chmur obliczeniowych, a obecność takiego rozwiązania jak Lambda zdecydowanie pomogła jeszcze bardziej umocnić ten fakt.

Lambda dostarczana przez Amazon Web Services to pierwsze rozwiązanie działające w modelu funkcja jako serwis udostępnione przez tak dużego gracza na rynku publicznych chmur obliczeniowych, co niewątpliwie pozytywnie wpłynęło na jego popularyzację. Zgodnie z raportem przygotowanym przez firmę Gartner [2] w 2018 roku rozwiązania oparte o model serverless stanowiły 5% globalnego rynku IT, a do końca roku 2020 co piąta tworzona aplikacja ma być oparta na tej architekturze. Rosnący trend widoczny jest również w statystykach bazujących na liczbie użytkowników rozwiązania dostarczanego przez AWS Lambda – początkiem 2018 roku zaledwie co piąty użytkownik chmury publicznej firmy Amazon korzystał z serwisu AWS Lambda. Na przestrzeni dwóch lat wartość ta wzrosła ponad dwukrotnie, gdyż co drugi klient decyduje się na wykorzystanie tego serwisu w swoich rozwiązaniach. [3]

Struktury tworzone w modelu FaaS są stosunkowo proste w porównaniu do ogólnych rozmiarów aplikacji, co nie przeszkadza największym firmom w wyznaczaniu trendów w wykorzystywaniu go. Wraz z rozmiarem tworzonego środowiska rośnie chęć do korzystania z tego typu rozwiązań, co początkowo może wydawać się totalnie złym pomysłem, ale w istocie jest wprost przeciwnie. Kryje się za tym decyzja czysto biznesowa. Poziom skomplikowania architektury oraz jej rozmiar wpływają na to, jak bardzo firma zwraca uwagę na koszty wprowadzanego rozwiązania. W większości przypadków FaaS umożliwia bowiem znaczące zmniejszenie kwoty widocznej na rachunkach wystawianych za korzystanie z usług dostawcy chmury. Pomimo tego, że rozwiązanie to nie jest idealne, czynnik kosztów zdecydowanie przeważa wszelkie zauważalne w nim wady.

2.2 Opis teoretyczny

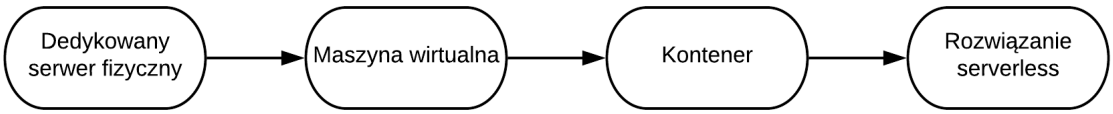
Model funkcja jako serwis (ang. *Function as a Service*) to jedno z nowszych podejść do tworzenia aplikacji rozproszonych. Zakłada on wydzielenie funkcjonalności do osobnych logicznych jednostek nazywanych funkcjami uruchamianych w niezależnym środowisku obliczeniowym. Użycie go umożliwia przeniesienie odpowiedzialności z zarządzania infrastrukturą, na której kod jest uruchamiany, z twórcy danego rozwiązania na dostawcę rozwiązania umożliwiającego uruchamianie wydzielonych jednostek kodu.

Dostarczana w ramach jednej jednostki logicznej funkcjonalność powinna stanowić bezstanową część aplikacji umożliwiającą zrealizowanie konkretnych zadań. Nie istnieje konkretne kryterium pozwalające stwierdzić, jak duża część operacji powinna być dostępna do zrealizowania przy wywoływaniu pojedynczej funkcji. Może to być zarówno jedno, ściśle określone zadanie bądź nawet cały serwis, którego akcje są dostępne poprzez wywołanie tej funkcji przy użyciu ustalonych z góry parametrów.

W każdym systemie rozproszonym krytycznym punktem procesu tworzenia architektury jest zdefiniowanie elastycznego kontraktu. Powinien on być w miarę stały w czasie, lecz nie implikuje to jego niezmienności. Chodzi przede wszystkim o maksymalne ograniczenie zmian niekompatybilnych wstecznie, które mogłyby spowodować niepożądane działanie po stronie wywołujących dany komponent klientów. Wynika to z faktu, że wykorzystywanie modelu FaaS powoduje jeszcze większe rozproszenie całego systemu na większą liczbę mniejszych komponentów. Umowa zawarta pomiędzy poszczególnymi serwisami na etapie ich projektowania umożliwia nie tylko sprawną integrację poszczególnych programów, ale również ułatwia procesy testowania. Aplikacja kliencka nie musi bowiem wykonywać zapytań do hosta w celu przetestowania swojego zachowania, gdyż znajomość kontraktu umożliwia podmianę takich zawołań na zapytania do mniejszej aplikacji imitującej poprawnie zwracane odpowiedzi.

Architektura serverless, którego częścią jest model FaaS, to kolejny krok w celu efektywniejszego wykorzystywania dostępnych zasobów fizycznych serwera hostującego dane rozwiązanie. Chodzi tu przede wszystkim o wprowadzenie większego poziomu abstrakcji pomiędzy kodem klienckim, a obsługującym jego

wykonanie maszyną. Zanim środowisko IT dotarło do tego miejsca, musiało upłynąć wiele lat.



Rys. 1: Od serwerów fizycznych do serverless (opracowano na podstawie [4])

Pierwszą, najoczywistszą metodą uruchamiania aplikacji było wykorzystanie fizycznych serwerów. Wiązało się to z koniecznością instalacji systemu operacyjnego oraz sterowników obsługujących poszczególne komponenty takie jak pamięć RAM. Na barkach administratora takiego systemu spoczywała również odpowiedzialność upewnienia się, że dostępne zasoby będą wystarczające do uruchomienia danego programu. Nie można zapomnieć również o wszelkich aktualizacjach bezpieczeństwa, których obecność na danym urządzeniu pozwala maksymalnie zabezpieczyć system przed niepożądanymi akcjami. Duży poziom skomplikowania takiej architektury powodował konieczność ciągłych testów aplikacji względem otaczającej ją infrastruktury w celu upewnienia się, że jakakolwiek zmiana w serwerze nie wprowadzi regresji.

W pewnym momencie zauważono, że rozwiązaniu temu daleko do efektywności. Większość serwerów świadczących usługi uruchamiania danych programów była wysoce przeskalowana. Zbyt duży udział zasobów fizycznych był niewykorzystywany przez utrzymywany program, co wraz z efektem skali w postaci wielu aplikacji działających niezależnie powodowało masę niepotrzebnie ponoszonych przez organizację kosztów. W tym celu wprowadzone zostały maszyny wirtualne. Serwer fizyczny wykorzystywany był do uruchomienia kilku maszyn wirtualnych, z których każda mogła symulować kompletnie inne środowisko uruchomieniowe, a z perspektywy uruchamianej aplikacji było do całkowicie niewidoczne. Krok w tę stronę spowodował wprowadzenie zdecydowanie bardziej przewidywalnego procesu wdrożenia oprogramowania, gdyż zmiany w systemie maszyny hosta były w zdecydowanej większości przeźroczyste dla uruchamianych nań maszyn wirtualnych. Dawało to również zdecydowanie większą elastyczność administratorowi takiego systemu w przenoszeniu go na inny serwer – maszynę wirtualną, zapisywaną w odpowiednim formacie pliku, wystarczyło przenieść na

innego hosta, co znacząco ułatwiało proces ewentualnego przywracania systemu po jego awarii. Pomimo tego, że główną wadę uruchamiania aplikacji bezpośrednio na serwerach fizycznych, którą był niski procent wykorzystywanych zasobów udało się ominąć, rozwiązaniu temu wciąż daleko było do ideału. Oprócz uruchomienia produktu, zasoby serwera musiały być również wykorzystywane na utrzymanie maszyn wirtualnych, co nie jest prostym obliczeniowo zadaniem.

Konieczność utrzymywania maszyn wirtualnych rozwiązano poprzez zmianę sposobu wirtualizacji. Zamiast wirtualizacji na poziomie systemu operacyjnego, która wykorzystywana jest do uruchamiania maszyn wirtualnych, wprowadzono mechanizm konteneryzacji. Polega ona na wirtualizacji na poziomie aplikacji. Rozwiązanie to pozwala na dystrybucję aplikacji wraz ze wszystkimi potrzebnymi do jej poprawnego działania zależnościami w formie kontenera, który jest tak naprawdę ograniczonym do absolutnego minimum systemem operacyjnym. Mechanizm ten znacząco zmniejsza narzut powstały w wyniku zarządzania maszynami wirtualnymi, gdyż z perspektywy maszyny hosta każdy z kontenerów traktowany jest jako standardowy proces. Z tego też powodu wszelkie mechanizmy związane z bezpieczeństwem są sterowane przez system operacyjny hosta, co jest niewątpliwie plusem i ogranicza prace konieczne do zabezpieczenia takiego rozwiązania. Fakt dostarczania aplikacji razem z potrzebnymi do jej uruchomienia zależnościami implikuje również wzrost poziomu zaufania do testów takiej aplikacji. Uruchomienie takiego kontenera przez dewelopera nie różni się zasadniczo niczym od uruchomienia go w środowisku produkcyjnym, co znacząco ułatwia proces testowania, pozwalając wykorzystać czas potrzebny do tej pory na dostosowanie testów do docelowej infrastruktury na zwiększenie ich efektywności.

Wszystkie przedstawione do tej pory mechanizmy łączy jedna wspólna cecha. Każdy zawiera mniej lub bardziej złożony opis tego, gdzie uruchomiona zostanie dana aplikacja. Mechanizm serverless pozwala wprowadzić abstrakcję, która umożliwia zaniechanie konieczności odpowiedzi na to pytanie. Wykorzystanie go powoduje zrzucenie odpowiedzialności na dostarczenie środowiska uruchomieniowego z twórcy danej aplikacji i przeniesienie jej na dostawcę rozwiązania serverless. Jedyne, co pozostaje po stronie dewelopera to spełnienie z góry zdefiniowanego kontraktu, co umożliwia dostarczenie przychodzących żądań do stworzonej funkcjonalności, przetworzenie ich oraz skonstruowanie właściwej odpowiedzi, która zostanie dostarczona do konsumenta.

Podczas projektowania każdej aplikacji jednym z ważniejszym aspektów, jakie trzeba wziąć pod uwagę jest przewidywany ruch oraz to, jak będzie on rósł. O ile z perspektywy skalowalności model ten zdaje się być idealnym, gdyż obsłużenie kilku tysięcy zapytań nawet w ciągu sekundy nie powinno stanowić żadnego problemu, o tyle cały czas trzeba mieć z tyłu głowy sposób naliczania opłat za korzystanie z takich rozwiązań. Użytkownik nie płaci z wyprzedzeniem żadnego abonamentu, w zapisach którego znalazłaby się informacja o przewidywanym ruchu. To, co generuje koszty to ilość przypisanych zasobów do funkcji oraz czas jej wykonywania w poszczególnych wywołaniach. Model FaaS świetnie nadaje się do zastosowań, w których przewidzenie ruchu użytkowników jest ciężkie do określenia i potrafi zmieniać się bardzo dynamicznie. Jednak kiedy aplikacja ma obsługiwać w miarę stały, wzmożony ruch użytkowników, rozwiązanie to przestaje być opłacalne i w takich przypadkach warto pomyśleć o zastosowaniu innego modelu architektury.

Model FaaS cechuje się też tym, że każde z zapytań realizowanych w tym samym momencie obsługiwane jest w osobnej instancji funkcji. Jest to jeden z najważniejszych fundamentów bezstanowości w tym modelu. Znaczy to tyle, że każde wykonanie ma przypisane dedykowane mu zasoby i pozostają one dostępne do końca realizowania danego procesu. Dostawca chmury obliczeniowej gwarantuje niezależność zasobów udostępnianych każdej jednostce obliczeniowej. Jednak nie wszystkie zasoby są zwalniane automatycznie po zakończeniu obsługi żądania, a celem zachowania części z nich jest optymalizacja i przyśpieszenie kolejnych wywołań.

W przypadku serwisu AWS Lambda jednostka logiczna, która realizowała dane żądanie pozostaje jeszcze przez pewien czas dostępna celem uruchomienia na niej napływających zapytań. Jednym z głównych zysków jest czas zaoszczędzony na uniknięciu oczekiwania na uruchomienie odpowiedniej jednostki arytmetyczno-logicznej, ale w tym miejscu benefity się nie kończą. Ponieważ środowisko uruchomieniowe jest cały czas dostępne, zachowany zostaje również kontekst wywołania funkcji, co umożliwia zachowanie np. połączenia z bazą danych czy pobranych plików. Wykorzystując te cechy, inżynier ma możliwość znaczącego skrócenia średniego czasu wykonania funkcji, co przekłada się bezpośrednio nie tylko na zwiększenie komfortu użytkowników, ale też i zmniejszony koszt takiego rozwiązania.

3. Od monolitu do FaaS

Fakt tak dużej historycznej popularności aplikacji monolitycznych wynika przede wszystkim z architektury całego systemu. Gdy w celu uruchomienia aplikacji wykorzystywano maszyny dedykowane lub też maszyny wirtualne, rozdzielanie aplikacji na mniejsze komponenty mijało się z celem. Wszelkie plusy spowodowane zmniejszeniem stopnia skomplikowania pojedynczego komponentu nie wyrównywały zwiększenia skali trudności zarządzania taką infrastrukturą. W dzisiejszych czasach decyzja dzielenia aplikacji na mniejsze twory jest dużo prostsza do podjęcia ze względu na powstałe rozwiązania mające na celu możliwie wysokie uproszczenie sterowania zasobami.

Pierwszym etapem jest przyjrzenie się architekturze zarządzanego systemu i zidentyfikowanie, które jego części są warte wyodrębnienia. Kryterium podziału jest rozdzielenie aplikacji na odrębne komponenty logiczne, z których każdy powinien pełnić odrębną funkcjonalność biznesową. Finalnie może okazać się, że powstałe serwisy znacząco różnią się pomiędzy sobą swoimi rozmiarami co wcale nie jest błędem. Prosty serwis służący do uwierzytelniania użytkowników zawiera dużo mniej logiki niż choćby system zarządzający zamówieniami użytkowników w sklepie internetowym. Można jednak śmiało stwierdzić, że jest to sytuacja, która może się zdarzyć. Warto jednak mieć na uwadze, że zbyt duże różnice pomiędzy wielkością poszczególnych komponentów mogą wskazywać na problem w podzieleniu odpowiedzialności pomiędzy serwisami. Tutaj z kolei jako przykład można użyć wspomniany wcześniej serwis służący uwierzytelnianiu użytkowników. Serwis ten nie powinien przechowywać np. danych klientów w sklepie internetowym, a jedynie dane potrzebne do poprawnej weryfikacji danych potrzebnych. Nawet to nie jest jednak wymogiem, gdyż wygodniejszym może okazać się trzymanie danych użytkowników w dedykowanej temu usłudze. Finalna decyzja pozostaje w rękach architektów danego systemu i to oni powinni stwierdzić, jaki podział będzie najlepszy w przypadku danego rozwiązania.

Po wydzieleniu komponentów kolejnym etapem jest zdefiniowanie interfejsów dostępnych dla klientów poszczególnych serwisów. Jest to w opinii autora tego tekstu najbardziej krytyczny moment definiowania architektury systemu rozproszonego, gdyż niepoprawny ich format wpłynie na duże spowolnienie późniejszego rozwoju systemu.

Interfejsy poszczególnych systemów powinny reprezentować biznesowe encje używane w całym systemie, których abstrakcja powinna uniemożliwić stwierdzenie jak wygląda wewnętrzna architektura wybranych komponentów. To, czy w ramach architektury danego serwisu dane przechowywane są w ramach relacyjnej bazy danych czy też bazy dokumentowej powinno być całkowicie przeźroczyste dla klienta danego serwisu, gdyż stanowi to szczegół implementacyjny, którego zmiana w dowolnym momencie życia systemu powinna być dla użytkownika zupełnie niewidoczna.

Zdefiniowany interfejs powinien być również stały w czasie. Przez jego niezmienność rozumie się kompatybilność wsteczną. Dodawanie nowych parametrów wejściowych dla poszczególnych punktów końcowych powinno zawierać zdefiniowanie takich domyślnych wartości, dla których wynik danego zapytania pozostaje niezmienny. Klienci danego serwisu mają wtedy możliwość na rozszerzenie swojej funkcjonalności poprzez dodanie wsparcia dla nowo zadeklarowanych pól. Łączy się to z innym zagadnieniem związanym z definiowaniem interfejsów, to znaczy idempotentnością. Dane API cechuje się taką własnością w sytuacji, gdy kolejne wywołania danego zapytania gwarantują użytkownikowi niezmienność odpowiedzi. Przykładem może być tutaj operacja aktualizacji danych użytkownika. Chcąc zaktualizować np. swój adres domowy wykorzystywany do realizacji i wysyłki zamówień w sklepie internetowym, każdorazowa prośba nadpisania tej danej powinna zakończyć się taką samą odpowiedzią dla tego samego adresu. Jeśli do serwisu dotarło N żądań, wszystkie N żądań powinno zostać albo zaakceptowane, albo odrzucone z powodu np. niepomyślnie przeprowadzonej walidacji.

Ważnym aspektem definicji interfejsu danego systemu jest również wyszczególnienie, które jego części powinny być dostępne tylko do użytku wewnętrznego, a które mogą być dostępne bezpośrednio dla użytkownika końcowego. Ma to duże znaczenie z punktu widzenia bezpieczeństwa całego systemu, gdyż część operacji mogłyby wpłynąć na jego integralność. Serwis udostępniający dane użytkowników powinien być w małej części dostępny na zewnątrz, np. na potrzeby pobrania listy użytkowników wraz z bardzo ograniczonym zakresem dostępnych danych, a możliwość odczytu hasła danego użytkownika powinna być zarezerwowana tylko i wyłącznie dla serwisu świadczącego usługę uwierzytelnienia celem walidacji danych wprowadzonych przez użytkownika w trakcie procedury logowania. Przykład ten bardzo dobrze pokazuje istotę odpowiedniego

zaprojektowania interfejsów pod kątem ich dostępności z zewnątrz, gdyż błąd w tej kwestii może zostać skrzętnie wykorzystany przez osoby chcące zaatakować dany system.

Kiedy poszczególne komponenty mają być dostępne z zewnątrz, warto pomyśleć o ujednoliceniu metod dostępowych do nich. Zmniejsza to narzut administracyjny wynikający z zarządzania serwisami i do pewnego stopnia pozwala ujednolicić budowę konsumujących ich klientów. Wszyscy mogą w takiej sytuacji skorzystać np. z udostępnianych w ramach danego zestawu komponentów bibliotek służących do poprawnego uwierzytelnienia się w serwisie. Wspomnianym mechanizmem jest użycie tzw. bramy API (ang. *API Gateway*), której wykorzystanie znacząco upraszcza dostęp do poszczególnych komponentów.

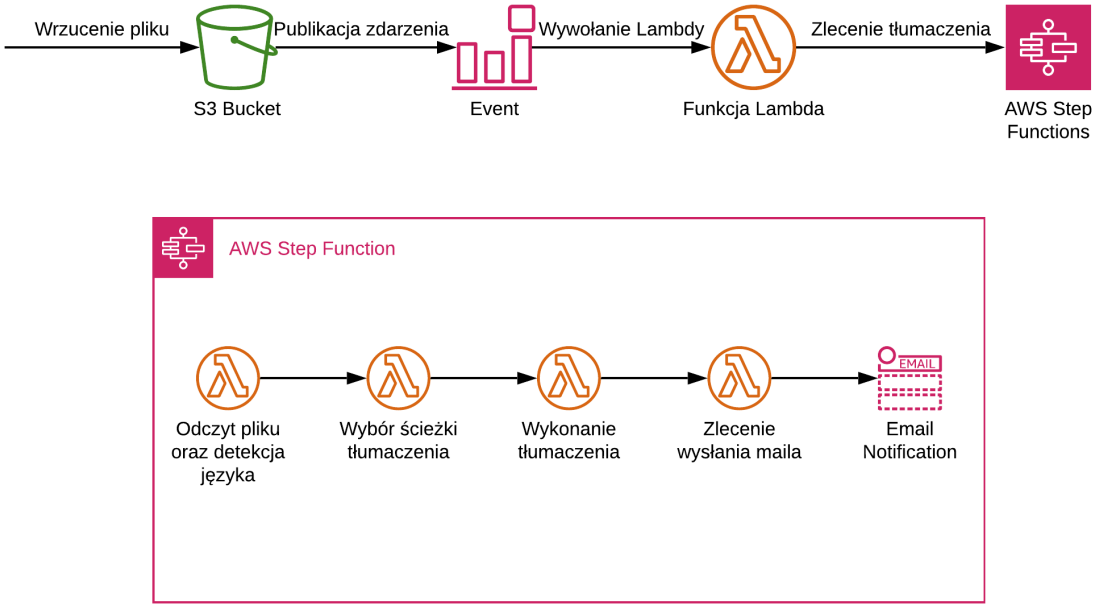
Głównym zadaniem bramy API jest agregacja poszczególnych usług w taki sposób, aby z zewnątrz były one widoczne jako pojedynczy serwis. Polega to na kierowaniu żądań przychodzących na dany punkt końcowy do odpowiedniego konsumenta na podstawie danych zawartych w żądaniu. Z reguły decyzja podejmowana jest na podstawie prefiksu znajdującego się w identyfikatorze zasobu, pobieranego z docelowego serwera, nazywanego URN, będącego drugą składową standardu URI, który jest powszechnie wykorzystywanym formatem reprezentacji zasobów w sieci. Dla przykładu reguły zapisane w bramie API mogą stanowić o tym, że zapytania wysłane na adres `https://example.com/auth` trafią do usługi obsługującej zapytania uwierzytelnienia użytkowników, podczas gdy zapytanie wskazujące na `https://example.com/users` zostanie skierowane do serwisu zapewniającego obsługę szeroko pojętych operacji na użytkownikach.

Wszystkie opisane do tej pory zasady skupiały się głównie na migracji z architektury monolitycznej do architektury mikro serwisów. Mają one również zastosowanie przy przejściu do architektury wykorzystującej funkcję jako serwis i jak najbardziej warto się nimi kierować. Istotą jest jednak zrozumienie różnic pomiędzy oboma wspomnianymi docelowymi architekturami, gdyż umożliwi to wybranie właściwego rozwiązania.

Rozpoczynając projektowanie aplikacji opierającej się na modelu funkcja jako serwis, głównym celem dewelopera jest określenie, jakiego rodzaju zdarzenia powinny być obsługiwane. Ze względu na charakter tworzonej usługi oraz przede wszystkim model naliczania kosztów z korzystania z takiej architektury,

główną przesłanką przy wydzielaniu komponentów jest wyróżnienie takich zjawisk. Pojedynczy komponent powinien wykonywać realizację zapytań należących do wyspecyfikowanej grupy żądań. Zastosowanie takiego podejścia umożliwia szybkie opracowanie kodu, uniezależnienie go od pozostałych komponentów, a te aspekty przekładają się na minimalizację narzutu czasowego koniecznego do zarządzania taką aplikacją.

Jednym z opracowanych w ramach tej pracy przykładów zastosowania architektury funkcja jako serwis jest aplikacja służąca do tłumaczenia tekstów. Aplikacja ta została zbudowana w całości przy użyciu wybranych narzędzi dostarczanych przez Amazon Web Services, które z perspektywy programisty nazywane być mogą bezserwerowymi.



Rys. 2: Architektura aplikacji służącej do tłumaczenia tekstów

Funkcjonalność stworzonej usługi pozwala na wykonywanie tłumaczeń plików tekstowych napisanych w językach polskim bądź angielskim przy użyciu serwisów dostarczanych w ramach chmury AWS. Użytkownik może dostarczyć informację o tym, w jakim języku treści znajdują się w pliku bądź też zdać się na funkcjonalność aplikacji pozwalającą na detekcję języka źródłowego.

Aplikacja zbudowana jest z następujących komponentów: prosta przechowalnia plików Amazon S3 (ang. *Simple Storage Service*), funkcje AWS

Lambda, orkiestrator funkcji AWS Step Functions, usługi Amazon Comprehend i Amazon Translate oraz prosta usługa mailowa AWS Simple Email Service.

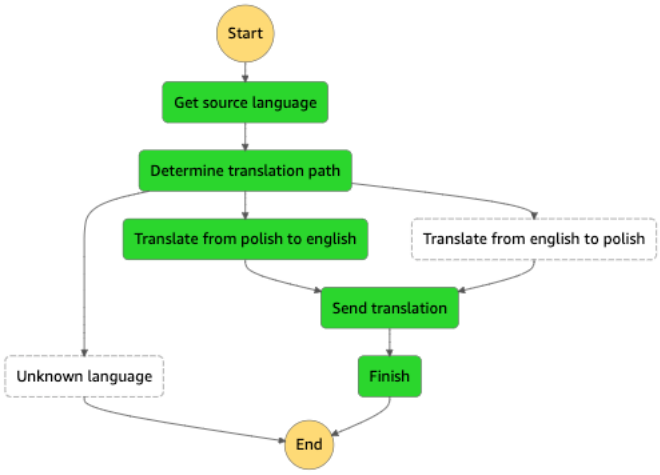
Przechowalnia plików jest inicjatorem wszystkich akcji wykonywanych przez aplikację. Celem zlecenia tłumaczenia pliku należy umieścić go odpowiednim kubełku (ang. *bucket*), co zostanie zarejestrowane jako odpowiednie zdarzenie. W ekosystemie chmury AWS, zdarzenie może zostać obsłużone na wiele różnych sposobów [4], poprzez umieszczenie w usłudze prostej kolejki *AWS Simple Queue Service*, umieszczenie zdarzenia w prostej usłudze powiadomień *AWS Simple Notification Service* lub obsłużone bezpośrednio przez wybraną funkcję *AWS Lambda*. W omawianej aplikacji, umieszczenie pliku w przechowalni Amazon S3 powoduje uruchomienie funkcji *AWS Lambda*, w ramach którego w przesyłanym do funkcji żądaniu umieszczone są metadane umożliwiające zidentyfikowanie pliku, w ramach którego modyfikacji wykonywane jest aktualne uruchomienie funkcji. Zadaniem funkcji jest odczytanie tych metadanych celem przekazania ich do nowego wykonania orkiestratora AWS Step Functions. Kod odpowiadający za realizację tej operacji prezentuje się następująco:

```
import boto3
import json
import os

sf = boto3.client('stepfunctions')

def start(event, context):
    entry = event['Records'][0]
    bucket = entry['s3']['bucket']['name']
    key = entry['s3']['object']['key']

    response = sf.start_execution(
        stateMachineArn=os.environ['SF_ARN'],
        input=json.dumps({
            "bucket": bucket,
            "key": key
        })
    )
```

Zastosowanie orkiestratora miało na celu dostarczenie mechanizmu stanowości w obrębie wykonywania poszczególnych funkcji Lambda, co odrobinę wychodzi poza założenia konceptu *funkcja jako serwis*, ale znacząco upraszcza architekturę aplikacji. Stan jest utrzymywany poza poszczególnymi komponentami, zatem pojedyncza funkcja wciąż pozostaje bezstanową. Jednak w kontekście

pojedynczego wykonania maszyny stanów, jaką niewątpliwie jest usługa dostarczana w ramach AWS Step Functions, stan jest jak najbardziej zachowywany. Jest to swego rodzaju nagięcie zasad obejmujących koncept bezstanowości aplikacji bezserwerowych, który znacząco ułatwia tworzenie takich aplikacji, jednocześnie wprowadzając pewne niewidoczne od samego początku wady. Wykorzystanie takiej technologii powoduje jeszcze mocniejsze powiązanie swojej aplikacji z udostępnianymi przez dostawcę chmury obliczeniowej usługami, co znacząco komplikuje ewentualną migrację do chmury oferowanej przez konkurenta. Problem ten określa się terminem blokady dostawcy (ang. *vendor lock-in*). O ile w przypadku zastosowania serwisu, którego zachowanie kontroluje się poprzez wywoływanie pewnych interfejsów programistycznych aplikacji API da się zamaskować poprzez zastosowanie odpowiedniej abstrakcji w kodzie (np. ukrycie implementacji wykorzystującej serwis Amazon Translate, służący do wykonywania tłumaczeń tekstów mogłoby zostać podmienione bliźniaczą implementacją odwołującą się do serwisu Google Translate), o tyle wykorzystanie maszyny stanów AWS Step Functions implikuje konieczność wyspecyfikowania łańcuchów wywołań poszczególnych funkcji w specjalnym języku zwanym językiem stanów Amazon (ang. *Amazon States Language*).

W ramach aplikacji, zdefiniowana została maszyna stanów o następującej budowie:



Rys. 3: Budowa maszyny stanów wykorzystywanej w aplikacji służącej do tłumaczenia tekstów

Każdy z bloczków widoczny na rys. 3 zdefiniowany jest przy użyciu encji nazywanej stanem. W ramach niej, wykonane może zostać wiele różnych akcji: wykonanie zadania przy użyciu skonfigurowanej funkcji AWS Lambda, dokonanie

wyboru następnego kroku, przerwanie wykonywania maszyny stanów, przekazanie dodatkowych parametrów do maszyny stanów, wprowadzenie opóźnienia przy przejściu do następnego kroku czy też rozpoczęcie wykonywania kilku gałęzi maszyny stanów manualnie. [5] Sama maszyna stanów, oprócz zdefiniowania logiki służącej do określenia kolejności wykonywanych kroków umożliwia przekazywanie parametrów do poszczególnych zawołań funkcji oraz zapisywania zwracanych przez nie wartości celem wykorzystania ich w obsłudze kolejnych stanów.

W stworzonej aplikacji, pierwsza z funkcji uruchamianych w trakcie wykonywania maszyny stanów służy do określenia języka tekstu. W przypadku, gdy informacja ta została wskazana przez użytkownika, funkcja odczytuje tą informację i przekazuje ją wgłąb celem wykorzystania przez następne komponenty. W przeciwnym razie, tekst poddawany jest analizie przy użyciu serwisu Amazon Comprehend, który dostarcza możliwość przetwarzania języka naturalnego (ang. *Natural Language Processing*). Jedną z funkcji serwisu jest detekcja języków obecnych w dostarczanych tekstach, a wynikiem jej działania jest tablica mówiąca o tym jaki procent tekstu stanowią poszczególne języki. Stworzona funkcja pobiera tą tabelę oraz wyszukuje w niej maksimum odpowiadającego językowi dominującemu, którego dwu lub trzyznakowy kod (zgodny z normą ISO 639-1) jest zwracany jako wynik działania funkcji. Kod opracowany w języku Python, realizujący opisaną funkcjonalność, prezentuje się następująco:

```
import boto3

s3 = boto3.client('s3')
comprehend = boto3.client('comprehend')
VALID_LANGS = ("pl", "en")

def start(event, context):
    bucket = event['bucket']
    key = event['key']
    text = read_file(bucket, key)
    lang = get_source_language(bucket, key, text)

    event['sourceLanguage'] = lang
    event['sourceText'] = text
    return event

def get_source_language(bucket, key, text):
    name = ".".join(key.split(".")[:-1])
    if contains_lang_in_name(name):
        return name[-2:]
    response = comprehend.detect_dominant_language(Text=text)
    if len(response['Languages']) == 0:
        return None
    most_important = sorted(response['Languages'], key=lambda k: k['Score'],
reverse=True)
    return most_important[0]['LanguageCode']

def read_file(bucket, key):
    response = s3.get_object(
        Bucket=bucket,
        Key=key
    )
    return response["Body"].read().decode('utf-8')

def contains_lang_in_name(name):
    if name[-3:] in [f"_{lang}" for lang in VALID_LANGS]:
        return True
    return False
```

Następnie wykonywany jest krok wyboru ścieżki tłumaczenia. Gdy w wyniku działania poprzedniej funkcji zwrócona została informacja o tym, że językiem dominującym jest język polski, wybierana jest ścieżka tłumaczenia z języka polskiego na angielski. Gdy tekst był napisany oryginalnie w języku angielskim, zostanie on przetłumaczony na polski. Gdyby jednak okazało się, że tekst został napisany w jakimkolwiek innym języku, zostanie wybrana trzecia możliwość – tekst nie zostanie przetłumaczony, a cały proces tłumaczenia zakończy się błędem. Funkcjonalność ta

została opracowana poprzez zastosowanie zadania typu *Choice*, dostępnego w ramach serwisu AWS Step Functions.

Kolejną w liście uruchamianych komponentów jest funkcja odpowiadająca za realizację procesu tłumaczenia. Na podstawie danych wejściowych dokonywane jest tłumaczenie z języka polskiego na angielski lub odwrotnie. Tłumaczenie wykonywane jest przy użyciu usługi Amazon Translate.

```
import boto3
import os

translate = boto3.client('translate')
s3 = boto3.client('s3')

SOURCE = os.environ['SOURCE']
TARGET = os.environ['TARGET']

def start(event, context):
    response = translate.translate_text(
        Text=event['sourceText'],
        SourceLanguageCode=SOURCE,
        TargetLanguageCode=TARGET
    )
    event['translatedText'] = response['TranslatedText']
    event['translatedLanguage'] = TARGET

    return event
```

Po zrealizowaniu operacji tłumaczenia uruchamiana jest następna funkcja odpowiedzialna za przesłanie oryginalnego oraz przetłumaczonego tekstu na zdefiniowany w trakcie procedury wdrożenia adres email. Do wykonania tej procedury użyty został serwis Amazon Simple Email Service udostępniający skalowalną i przystępną w użyciu usługę wysyłania wiadomości, a fragment kodu realizujący tę funkcjonalność wygląda następująco:

```
import boto3
import os

ses = boto3.client('ses')
ADDRESS = os.environ['ADDRESS']

def start(event, context):
    resp = ses.send_email(
        Source=ADDRESS,
        Destination={
            "ToAddresses": [ADDRESS]
        },
        Message={
            "Subject": {
                "Data": f"Translation: {event['key']} from
{event['sourceLanguage']} to {event['translatedLanguage']}"
            },
            "Body": {
                "Text": {
                    "Data":
f"Original:\n{event['sourceText']}\n\nTranslated:\n{event['translatedText']}"
                }
            }
        }
    )

    event['messageId'] = resp['MessageId']
    return event
```

Po pomyślnym wysłaniu wiadomości, maszyna stanów przechodzi do następnego kroku, którego zadaniem jest oznaczenie aktualnego wykonania maszyny za udane. W ramach AWS Step Functions ważne jest sprecyzowane określenie tego, z jakim wynikiem zakończyć ma się aktualne wykonanie. Programista posiada również możliwość jawnego wymuszenia zakończenia maszyny stanów niepowodzeniem, np. w przypadku zwrócenia konkretnej wartości z określnej funkcji. W przypadku omawianej aplikacji, dzieje się tak w przypadku wykrycia, że zarówno język polski jak i angielski nie są językami dominującymi w tłumaczonym tekście. W takiej sytuacji dochodzi do jawnego oznaczenia bieżącego wykonania maszyny stanów jako nieudane. Fakt ten mógłby zostać wykorzystany do przesłania powiadomienia użytkownikowi o nieudanym przetwarzaniu pliku poprzez maila.

Wdrażanie aplikacji zostało zaimplementowane przy użyciu Serverless Framework [6]. Jest to narzędzie znacząco upraszczające procedurę udostępniania aplikacji bezserwerowej w ramach usług jednego z wielu dostawców chmur

obliczeniowych. Uproszczenie to zostało osiągnięte dzięki wprowadzeniu warstwy abstrakcji do definiowania zasobów, jakie deweloper chciałby stworzyć w ramach swojej aplikacji. Podejście to zmniejsza znacząco narzut czasowy wymagany do pomyślnego wdrożenia aplikacji co potrafi mocno przyśpieszyć prace nad całym projektem. Architektura aplikacji przedstawiona na Rys. 2 zapisana przy użyciu definicji w Serverless Framework w formacie YAML prezentuje się następująco:

```
service: aws-python-translator

provider:
  name: aws
  runtime: python3.8
  region: us-east-1
  iamRoleStatements:
    - Effect: Allow
      Action:
        - comprehend:DetectDominantLanguage
      Resource: '*'
    - Effect: Allow
      Action:
        - translate:Translate*
      Resource: '*'
    - Effect: Allow
      Action:
        - ses:SendEmail
        - ses:SendRawEmail
        - ses:ListIdentities
        - ses:VerifyEmailIdentity
        - ses:DeleteIdentity
      Resource: '*'
    - Effect: Allow
      Action:
        - states:StartExecution
      Resource:
        Fn::Sub:
"arn:aws:states:#{AWS::Region}:#{AWS::AccountId}:stateMachine:TranslatorStepFunctio
nsStateMachine*"
    - Effect: Allow
      Action:
        - s3:*
      Resource: '*'

functions:
  email-registration:
    handler: custom_resource.handler
  trigger-translation:
    handler: trigger.start
    environment:
      SF_ARN:
        Ref: TranslatorStepFunctionsStateMachine
    events:
      - s3:
          bucket: aws-python-translations
          event: s3:ObjectCreated:*
  get-source-lang:
    handler: getSourceLang.start
  entopl:
    handler: translate.start
    environment:
      SOURCE: "en"
      TARGET: "pl"
```

```
  pltoen:
    handler: translate.start
    environment:
      SOURCE: "pl"
      TARGET: "en"
  send:
    handler: send.start
    environment:
      ADDRESS: ${opt:email, 'default'}

stepFunctions:
  stateMachines:
    translator:
      definition:
        Comment: Translate file uploaded to S3 to different language
        StartAt: Get source language
        States:
          Get source language:
            Type: Task
            Resource:
              Fn::GetAtt: [get-source-lang, Arn]
            Next: Determine translation path
          Determine translation path:
            Type: Choice
            Choices:
            - Variable: "$.sourceLanguage"
              StringEquals: pl
              Next: Translate from polish to english
            - Variable: "$.sourceLanguage"
              StringEquals: en
              Next: Translate from english to polish
            Default: Unknown language
          Translate from polish to english:
            Type: Task
            Resource:
              Fn::GetAtt: [pltoen, Arn]
            Next: Send translation
          Translate from english to polish:
            Type: Task
            Resource:
              Fn::GetAtt: [entopl, Arn]
            Next: Send translation
          Unknown language:
            Type: Fail
            Cause: Couldn't determine source language
          Send translation:
            Type: Task
            Resource:
              Fn::GetAtt: [send, Arn]
            Next: Finish
          Finish:
            Type: Succeed

resources:
  Resources:
    RegisterEmail:
      Type: Custom::EmailIdentity
      Properties:
        ServiceToken:
          Fn::GetAtt: [EmailDashregistrationLambdaFunction, Arn]
        Email: ${opt:email, 'default'}

plugins:
  - serverless-step-functions
  - serverless-s3-local
  - serverless-offline
```

```
  - serverless-python-requirements
  - serverless-pseudo-parameters

custom:
  pythonRequirements:
    dockerizePip: "non-linux"
```

Sekcja *provider* opisuje podstawową konfigurację dla tworzonych zasobów. Można w niej zaobserwować informację o tym, że procedura wdrożenia powinna przebiec w chmurze AWS w regionie Północnej Virginii, oznaczonym skrótowo symbolem *us-east-1*, a wszystkie funkcje napisane są w języku Python.

Podsekcja *iamRoleStatements* definiuje uprawnienia aplikacji w odniesieniu do usług dostępnych w ramach chmury Amazon Web Services. Odbywa się to przy użyciu zdefiniowania encji nazywanej rolą w obrębie serwisu AWS Identity and Access Management służącemu do zarządzania dostępem do poszczególnych usług dostawcy chmury obliczeniowej. Aplikacja ma przypisane uprawnienia do wykrywania dominującego języka w tekście (*comprehend:DetectDominantLanguage*), tłumaczenia (*translate:Translate**), zarządzania zasobami w serwisie Simple Email Service oraz uruchamiania nowych maszyn stanów w serwisie AWS Step Functions (*states:StartExecution*).

Z kolei w sekcji *functions* zdefiniowane zostały wszystkie funkcje AWS Lambda wchodzące w skład serwisu. W ramach tej sekcji możliwe jest określenie, jaka jednostka kodu (funkcja z języka Python) ma obsłużyć przychodzące żądanie. Możliwe jest również skonfigurowanie poszczególnych funkcji przy użyciu zmiennych środowiskowych (podsekcja *environment*) oraz uruchomienie wyzwalaczy w postaci zdarzeń, na które dana funkcja Lambda powinna reagować. W przypadku funkcji *trigger-translation* takim zdarzeniem jest umieszczenie pliku (*s3:ObjectCreated:**) w kubełku S3 o nazwie *aws-python-translations*.

Sekcja *stepFunctions* konfiguruje dostępne w ramach aplikacji maszyny stanów. Jej wykorzystanie było możliwe dzięki umieszczeniu w sekcji *plugins* odniesienia do dodatku *serverless-step-functions*, którego obecność umożliwia skonfigurowanie takiego zasobu. Dodatek ten niestety wymusza konieczność skorzystania z języka stanów Amazon co implikuje potrzebę zdefiniowania podobnego tworu w sytuacji, gdy programista zdecyduje się na umieszczenie swojej aplikacji w obrębie chmury innej niż AWS. Mógłby do tego wykorzystać na przykład usługę Azure Logic Apps dostępną w ramach chmury Microsoft Azure.

Znajdująca się blisko końca komórka *resources* pozwala na zdefiniowanie zasobów nie mających bezpośredniego wsparcia Serverless Framework. W przypadku chmury AWS wymaga ona utworzenia zasobu zgodnego ze składnią przyjmowaną przez serwis AWS CloudFormation, umożliwiającego definiowanie niemalże wszystkich zasobów dostępnych w tej chmurze za pomocą kodu. W tej konkretnej sytuacji tworzony jest obiekt stylizowany, na co wskazuje prefiks *Custom* mieszczący się w nazwie typu *Custom::EmailIdentity*. Taki typ zasobu obsługiwany jest przez skonstruowaną przez programistę funkcję AWS Lambda, a zasób ten wcale nie musi mieć swojego odzwierciedlenia w zasobach istniejących w obrębie chmury obliczeniowej. W ramach jego implementacji można odwołać się do dowolnych interfejsów programistycznych dostępnych z internetu, celem np. pobrania kursów walut. W stworzonej aplikacji zasób ten wykorzystywany jest do rozpoczęcia procesu weryfikacji adresu email, który po jego zakończeniu zostanie wykorzystany do przyjmowania przetłumaczonych tekstów. Proces ten nie jest możliwy do zrealizowania przy użyciu standardowych typów dostępnych w ramach AWS CloudFormation, stąd konieczne było wykorzystanie do tego typu stylizowanego.

Wspomniana wcześniej sekcja *plugins* odnosi się do wtyczek wykorzystywanych w obrębie projektu. Publicznie dostępne rozszerzenia Serverless Framework obejmują takie procesy jak umożliwienie wykonywania testów aplikacji napisanej w architekturze funkcja jako serwis bez konieczności posiadania dostępu do internetu (np. poprzez uruchomienie lokalnego serwera WWW implementującego interfejs programistyczny usługi Amazon Simple Storage Service) czy rozszerzenia składni ułatwiające programiście poprawne zdefiniowanie tworzonych w ramach chmury zasobów. Możliwe jest również stworzenie własnych wtyczek, które mogą obsługiwać logikę biznesową specyficzną dla rozwijanej aplikacji.

Sekcja *custom* kończąca opis architektury pozwala na konfigurację wykorzystywanych w projekcie wtyczek. W aplikacji będącej przedmiotem rozważań tego rozdziału zawiera ona ustawienia rozszerzenia służącego do poprawnego zbudowania paczki ze stworzonymi komponentami, które następnie zostaną umieszczone w ramach chmury i będą wykorzystywane do realizacji zapytań wykonywanych do funkcji AWS Lambda. Język Python, pomimo swojej wieloplatformowości, nie jest w 100% przenaszalny pomiędzy różnymi systemami operacyjnymi, gdyż część modułów korzysta z zewnętrznych bibliotek, napisanych w językach kompilowanych, których zbudowanie wymaga określenia docelowej

architektury systemu. Ponieważ wszyscy dostawcy rozwiązań funkcja jako serwis informują o tym, że dostarczane przez programistów aplikacje uruchamiane są na systemach z rodziny linux, w trakcie przygotowywania paczki ze zbudowanymi komponentami wiadomo, w jakim ekosystemie będą one wykonywane. Wykorzystanie wtyczki deleguje przygotowanie paczki do kontenera uruchomionego na platformie Docker, dzięki czemu wszystkie zależności, które są wykorzystywane przez dewelopera, zostaną zbudowane dla właściwego systemu operacyjnego, co zapewni kompatybilność i poprawne działanie aplikacji.

Serverless Framework został napisany w języku node.js, którego obecność na urządzeniu wymagana jest do jego uruchomienia. Co prawda implikuje to konieczność tworzenia rozszerzeń właśnie w tym języku, ale z racji jego prostoty oraz dostępności w obrębie wielu platform dostępne jest multum wtyczek i bardzo mało prawdopodobnym jest, by konieczne rozszerzenie nie zostało do tej pory zbudowane. Gdyby okazało się, że takowe nie jest dostępne, stworzenie własnego rozszerzenia powinno okazać się dość łatwe ze względu na przejrzystą dokumentację dostępną w ramach opisywanego rozwiązania.

Celem wykonania wdrożenia aplikacji należy wykonać polecenie *deploy* dostępne w obrębie aplikacji Serverless Framework. W przypadku opisywanej aplikacji, do wykonania komendy należy przekazać dodatkowy parametr *email* określający adres mailowy wykorzystywany do odebrania przetłumaczonych plików tekstowych:

```
serverless deploy --email <ADRES_EMAIL>
```

Z kolei aplikacja wdrożona aplikacja może zostać usunięta przy użyciu komendy *remove*:

```
serverless remove
```

Omawiany Framework dostarcza również wiele narzędzi służących do monitoringu i diagnostyki potencjalnych problemów, które mogą wystąpić w trakcie użytkowania aplikacji. Upraszcza on również testowanie aplikacji, które mogłyby być wdrażane na chmury różnych dostawców, udostępniając ujednolicony interfejs programistyczny umożliwiający tworzenie testów aplikacji niezależnie od wybranego twórcy chmury.

Ze względu na ograniczenia obecne w darmowej wersji wykorzystywanego Serverless Framework, nie jest możliwe użycie tego produktu w celu stworzenia odpowiedniego monitoringu. Jest to funkcjonalność, która dostępna jest tylko w płatnej

wersji omawianego produktu. Z tego powodu zdecydowano się na ręczne wdrożenie odpowiedniego rozwiązania służącego do ciągłej weryfikacji stanu aplikacji oraz powiadamiania administratora systemu o występujących w trakcie jej działania problemach. W tym celu wykorzystane zostały następujące narzędzia: serwis agregujący logi oraz metryki Amazon CloudWatch posiadający integrację z większością serwisów dostępnych w ramach chmury Amazon Web Services, usługa Amazon Simple Notification Service udostępniająca narzędzie do generowania notyfikacji oraz Amazon Simple Email Service, rozwiązanie umożliwiające dystrybucję wiadomości email.

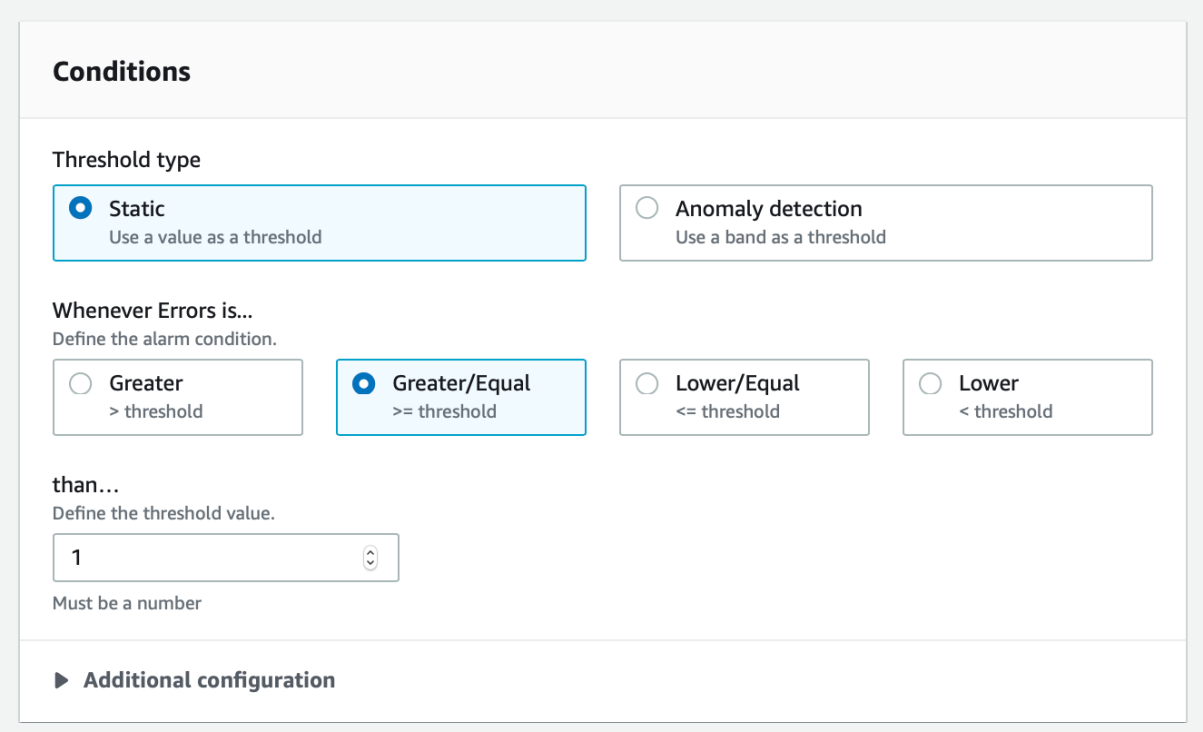
Ponieważ stworzony serwis nie udostępnia żadnych punktów końcowych w ramach swojego interfejsu programistycznego, monitoring oparty jest o weryfikację stanu wykonywania składających się na serwis funkcji AWS Lambda. Niepoprawne działanie któregokolwiek komponentu pozwala na poprawne wykrycie niepoprawnego działania nie tylko serwisu AWS Lambda, ale również wszystkich serwisów wywoływanych w ramach implementacji poszczególnych funkcji. Bazując na tych założeniach, jako akcję niepożądaną zdefiniowano niepoprawne działanie którejkolwiek z funkcji AWS Lambda, co w ramach dostępnych metryk oznaczone jest jako zakończenie funkcji błędem. Informacje o błędach występujących w ramach uruchomienia poszczególnych funkcji oraz czasy ich trwania zostały umieszczone na tablicy (ang. *dashboard*) umożliwiającej wyświetlanie w jednym widoku wielu widżetów, które dzięki odpowiedniej konfiguracji agregują interesujące z perspektywy administratora systemu dane. Ponieważ serwis Amazon CloudWatch jest, tak jak większość serwisów udostępnianych w ramach publicznej chmury AWS, serwisem podzielonym logicznie na regiony, wszelkie omawiane poniżej zasoby powinny być tworzone w tym samym regionie, w którym została wdrożona aplikacja, czyli w regionie Północnej Virginii oznaczonym symbolem *us-east-1*.

Aby stworzyć agregator dla pozostałych metryk opisujących zachowanie omawianego serwisu, po zalogowaniu się w konsoli Amazon Web Services należy przejść do strony głównej serwisu Amazon CloudWatch. Następnie, w menu widocznym po lewej stronie trzeba wybrać opcję „Dashboards”, a po załadowaniu strony kliknąć przycisk „Create Dashboard”. W wyniku tej akcji pojawi się tzw. pop-up, gdzie możliwe będzie stworzenie nowej, pustej tablicy o pożądanej przez administratora nazwie.

Serwis CloudWatch umożliwia również zdefiniowanie tzw. alertów, które generowane są w wyniku ustalonych zdarzeń. Administrator systemu zdecydował się na stworzenie alertów dla wszystkich funkcji AWS Lambda, które są wywoływane w sytuacji zakończenia wykonania którejkolwiek z funkcji błędem. Aby stworzyć alert, w menu dostępnym po lewej stronie serwisu CloudWatch należy wybrać opcję „Alarms", w wyniku czego zostanie wyświetlona tabela agregująca wszystkie alerty zdefiniowane w kontekście konta, na które zalogowany jest użytkownik oraz wybranego regionu. Po prawej stronie nagłówka tabeli znajduje się przycisk „Create alarm", którego naciśnięcie zainicjuje proces tworzenia alertu składający się z czterech kroków. W pierwszym z nich użytkownik konsoli wybiera metrykę, która ma zostać użyta do stworzenia alarmu. Po wybraniu opcji „Source" możliwe jest zdefiniowanie metryki jako słownika, zapisanego w formacie JSON. Poniżej znajduje się przykładowa metryka, sumująca wszystkie błędy wywołania funkcji o nazwie *aws-python-translator-dev-trigger-translation* umieszczonej w regionie Północnej Virginii, odpowiedzialnej za obsługę zdarzenia umieszczenia pliku w kubełku Amazon Simple Storage Service celem zainicjowania operacji tłumaczenia.

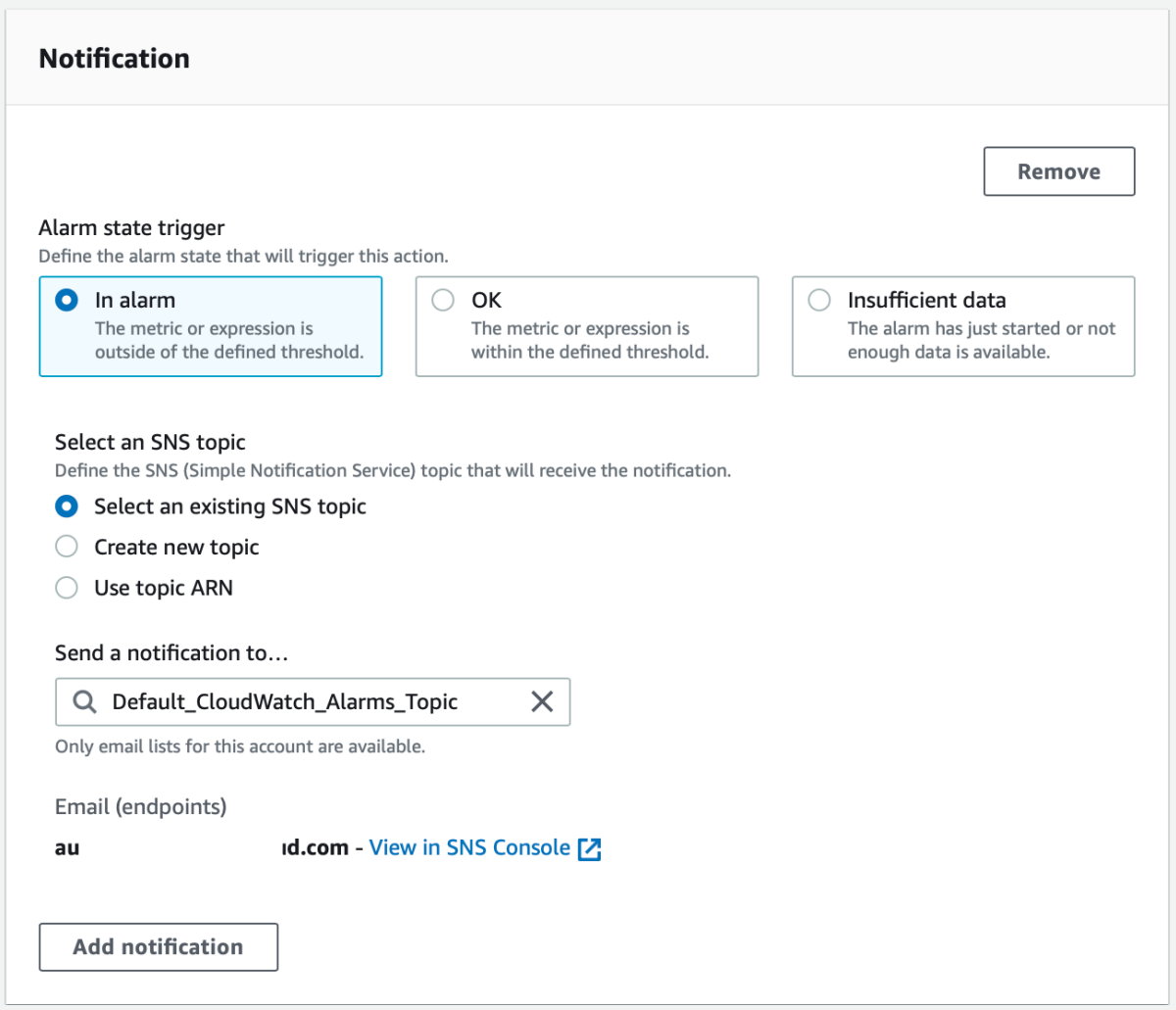
```
{
    "metrics": [
        [ "AWS/Lambda", "Errors", "FunctionName", "aws-python-translator-dev-
trigger-translation", "Resource", "aws-python-translator-dev-trigger-translation" ]
    ],
    "view": "timeSeries",
    "stacked": false,
    "period": 300,
    "region": "us-east-1",
    "stat": "Sum"
}
```

Po zaakceptowaniu wyboru metryki przyciskiem „Select metric" administratorowi ukaże się sekcja „Specify metric and conditions", dająca możliwość zdefiniowania zachowania wywołującego alert. Ponieważ zamierzonym zdarzeniem uruchomiającym alert jest wystąpienie jakiegokolwiek błędu, w sekcji „Conditions" należy ustawić typ progu (ang. *threshold type*) jako statyczny (ang. *static*), warunkiem wyzwalającym alarm powinna być opcja większe lub równe (ang. *Greater/Equal*), a w dostępnym poniżej polu tekstowym należy wprowadzić wartość 1. Po uzupełnieniu tych wartości można przejść do kolejnego kroku poprzez kliknięcie znajdującego się na dole strony przycisku „Next".



Rys. 4: Widok definiowania warunku wyzwalającego alarm

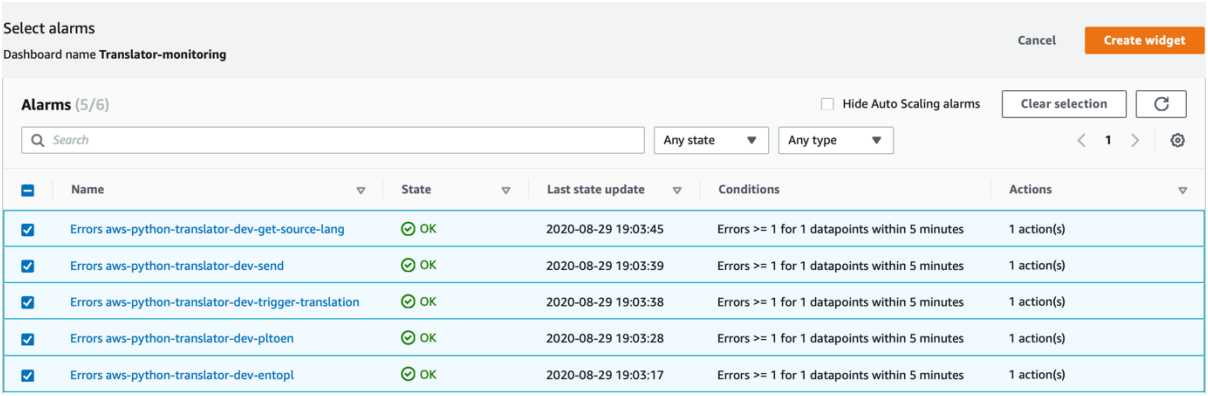
W kolejnym kroku administrator definiuje powiadomienie, które ma zostać wygenerowane w wyniku wystąpienia alarmu. Wyzwalaczem alertu powinno być znalezienie się metryki w stanie alarmu (ang. *In alarm*), co jest z perspektywy działania całego systemu zdarzeniem niepożądanym. Możliwy jest również wybór wątku SNS (ang. *SNS topic*), który powinien odebrać notyfikację celem rozdystrybuowania jej zasubskrybowanym adresom emailowym. Zdefiniowanie tych ustawień widoczne jest na Rys. 5. Na dole wspomnianego rysunku widoczny jest również przycisk „Add notification", którego przyciśnięcie umożliwi zdefiniowanie identycznego powiadomienia dla sytuacji, gdy alarm zostanie wygaszony i przejdzie do stanu OK.



Rys. 5: Definiowanie powiadomienia w ramach tworzonego alertu

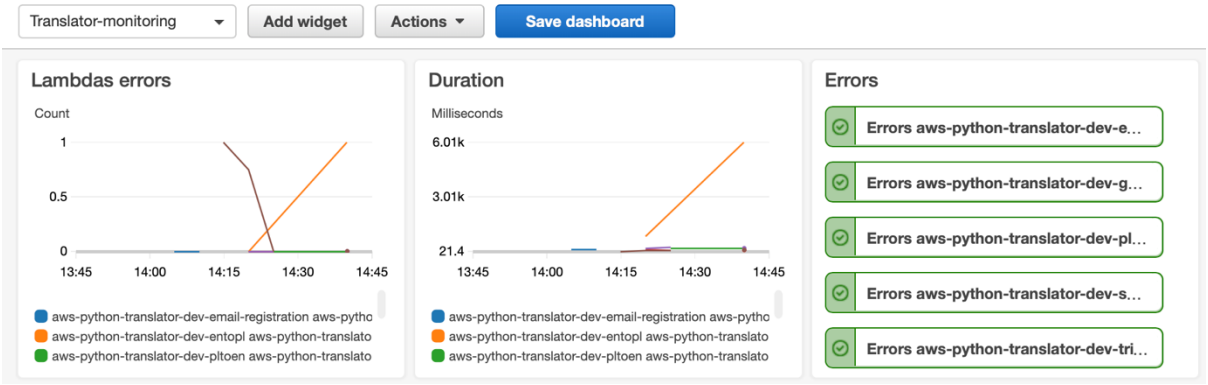
W kolejnym kroku administrator systemu ma możliwość nazwania stworzonego alertu oraz umieszczenia w razie potrzeby dłuższego opisu stworzonego zasobu. Następnie wyświetlony zostanie ekran podsumowujący tworzony alarm, na dole którego znajduje się przycisk „Create alarm”, którego kliknięcie zakończy proces tworzenia alarmu.

Na stworzonym wcześniej agregatorze metryk możliwe jest wyświetlenie stanu wybranych alertów. W tym celu, po wejściu w widok głównego agregatora należy wybrać przycisk „Add widget”, następnie zaznaczyć opcję „Alarm status”. Po wyświetleniu tabeli ze wszystkimi zdefiniowanymi alertami i wybraniu interesujących administratora opcji, stworzenie widżetu można zatwierdzić przyciskiem „Create widget”.



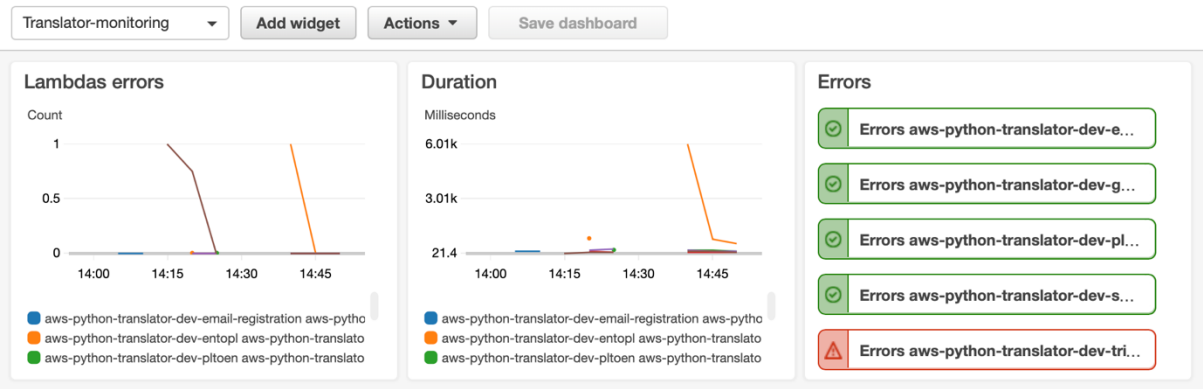
Rys. 6: Tworzenie widżetu grupującego alerty

W analogiczny sposób możliwe jest stworzenie widżetów wyświetlających wykresy prezentujące wybrane dane. W ramach omawianego serwisu, zdecydowano się na wyświetlenie wykresów prezentujących ilość występujących błędów oraz czasów wykonywania poszczególnych komponentów.



Rys. 7: Agregator wartościowych dla serwisu metryk

Celem weryfikacji poprawnego działania monitoringu zasymulowano niepoprawne działanie jednej z funkcji AWS Lambda. Symulacja ta przebiegła prawidłowo. Alert ustawiony dla funkcji odpowiadającej za zainicjowanie procesu tłumaczenia został wyzwolony, co potwierdza wyświetlenie odpowiedniej informacji na tablicy grupującej widżety (Rys. 8) oraz otrzymanie przez administratora systemu odpowiedniego powiadomienia na zdefiniowany adres mailowy (Rys. 9).



Rys. 8: Agregator metryk z widocznym wykryciem niepożądanej sytuacji

AWS Notifications

ALARM: "Errors aws-python-translator-dev-trigger-translation" in US East (N. Virginia)

To:

You are receiving this email because your Amazon CloudWatch Alarm "Errors aws-python-translator-dev-trigger-translation" in the US East (N. Virginia) region has entered the ALARM state, because "Threshold Crossed: 1 out of the last 1 datapoints [1.0 (29/08/20 14:51:00)] was greater than or equal to the threshold (1.0) (minimum 1 datapoint for OK -> ALARM transition)." at "Saturday 29 August, 2020 14:56:38 UTC".

View this alarm in the AWS Management Console:
https://us-east-1.console.aws.amazon.com/cloudwatch/home?region=us-east-1#s=Alarms&alarm=Errors%20aws-python-translator-dev-trigger-translation

Alarm Details:
- Name: Errors aws-python-translator-dev-trigger-translation
- Description:
- State Change: OK -> ALARM
- Reason for State Change: Threshold Crossed: 1 out of the last 1 datapoints [1.0 (29/08/20 14:51:00)] was greater than or equal to the threshold (1.0) (minimum 1 datapoint for OK -> ALARM transition).
- Timestamp: Saturday 29 August, 2020 14:56:38 UTC
- AWS Account: 510122856002
- Alarm Arn: arn:aws:cloudwatch:us-east-1:510122856002:alarm:Errors aws-python-translator-dev-trigger-translation

Threshold:
- The alarm is in the ALARM state when the metric is GreaterThanOrEqualToThreshold 1.0 for 300 seconds.

Monitored Metric:
- MetricNamespace: AWS/Lambda
- MetricName: Errors
- Dimensions: [FunctionName = aws-python-translator-dev-trigger-translation]
- Period: 300 seconds
- Statistic: Sum
- Unit: not specified
- TreatMissingData: missing

State Change Actions:
- OK:
- ALARM: [arn:aws:sns:us-east-1:510122856002:Default_CloudWatch_Alarms_Topic]
- INSUFFICIENT_DATA:

Rys. 9: Treść powiadomienia otrzymanego przez administratora systemu

Stworzona aplikacja czerpie pełnymi garściami z tego, co oferuje model funkcja jako serwis, a dzięki zastosowaniu publicznie dostępnego Serverless Framework aplikację udało się wdrożyć w szybki i łatwy sposób. Jednakże, implementacja aplikacji została celowo uzależniona w jawny sposób od interfejsów programistycznych udostępnianych przez biblioteki dostarczane przez Amazon Web Services. Wiąże to mocno implementację aplikacji z wybranym dostawcą chmury obliczeniowej. Miało to pokazać fakt, że skorzystanie z rozwiązań potencjalnie ułatwiających pracę programiście nie zawsze zadziała na jego korzyść. Umiejętność

poprawnego wykorzystania dostępnych na rynku narzędzi jest kluczem do stworzenia dobrego oprogramowania, które będzie łatwe w późniejszym utrzymaniu.

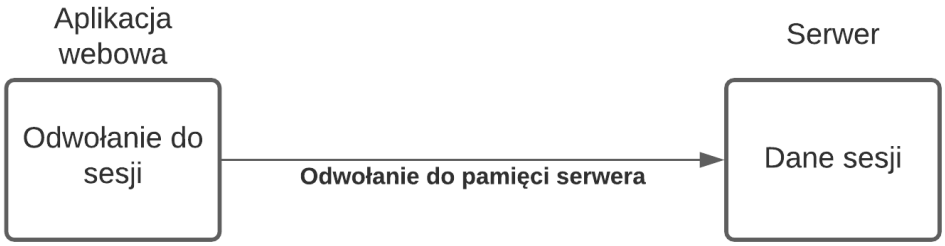
Implementacja całej aplikacji tworzy również pewną wyrwę w fundamentach architektury bezserwerowej. Wykorzystanie serwisu AWS Step Functions umożliwiło stworzenie aplikacji przechowującej swój stan, co niejako stoi naprzeciw założeniom modelu funkcja jako serwis. W tym przypadku podejście to znacząco upraszcza logikę klienta wymaganą do korzystania z dostarczonego rozwiązania, jednakże może wprowadzić pewne problemy nie tyle ideologiczne, co techniczne. W przypadku chęci skorzystania np. z zewnętrznej bazy danych, konieczne jest utworzenie stosownego połączenia do wykonywania na niej operacji. Ze względu na ulotny charakter nie tylko funkcji AWS Lambda, ale wszystkich rozwiązań bezserwerowych, powoduje to konieczność tworzenia takich połączeń wraz z każdym uruchomieniem odpowiedniego komponentu. Nie jest to operacja natychmiastowa i proces ten może wpłynąć negatywnie na wydajność takiego komponentu oraz wszystkich aplikacji klienckich korzystających z takich usług.

4. Stanowość aplikacji stworzonych w architekturze funkcja jako serwis

Zagadnienie stanowości bądź bezstanowości aplikacji to bardzo rozbudowany temat rozważań nie ograniczający się wyłącznie do architektury funkcja jako serwis, ale w zasadzie do wszystkich tworzonych aplikacji webowych, gdyż z perspektywy klienta końcowego implementacja powinna być całkowicie przeźroczysta. W przypadku architektury bezserwerowej nie jest to jednak zagadnienie równie trywialne jak w pozostałych przypadkach, gdyż głównym założeniem, które przyświecało jego tworzeniu była właśnie bezstanowość. Nie znaczy to, że stworzenie aplikacji stanowej opartej na modelu będącym przedmiotem tej pracy jest niemożliwe. Wartym przemyślenia jest jednak sposób, w jaki potencjalny użytkownik ma zostać zidentyfikowany przez aplikację bądź jak rejestrowane mają być dotychczasowe interakcje poszczególnych komponentów, gdyż zrealizowanie tych mechanizmów w sposób nieprawidłowy może znacząco utrudnić potencjalny rozwój stworzonej aplikacji w przyszłości.

Bardzo ważnym aspektem w komunikacji z zewnętrznym serwerem jest zapewnienie możliwie bezpiecznego sposobu zidentyfikowania klienta wysyłającego żądanie. Nie można doprowadzić do sytuacji, kiedy klient A jest w stanie podszyć się pod klienta B w celu uzyskania danych wrażliwych, do których nie powinien mieć dostępu. Czym mechanizmy uwierzytelniania użytkownika dostępne w ramach modelów bezstanowych różnią się od tych znanych z rozwiązań stanowych i jak wpływa to na ich bezpieczeństwo?

Proces uwierzytelniania wykorzystujący stanowość aplikacji polega na przechowywaniu danych użytkownika w ramach zasobów aplikacji w postaci sesji. Użytkownik, wykonując zapytania do serwera, każdorazowo przesyła identyfikator swojej sesji. Jeśli identyfikator zostanie odnaleziony w bazie danych po stronie serwera, tożsamość użytkownika zostanie pomyślnie zweryfikowana i wszelkie akcje zlecone przez klienta będą odbywać się w kontekście użytkownika. Oznacza to, że dane dotyczące sesji wszystkich użytkowników muszą być dostępne dla serwera.



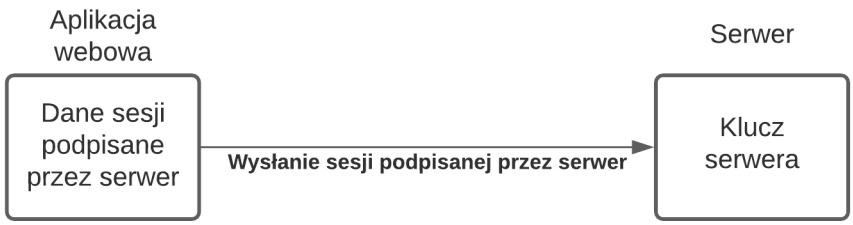
Rys. 10: Stanowy scenariusz uwierzytelniania

Plusem takiego rozwiązania jest możliwość zarządzania sesją użytkownika po jej utworzeniu. Dość łatwą operacją wydaje się unieważnienie sesji w dowolnym momencie zarządzone przez serwer. W tym scenariuszu to on jest właścicielem zasobu sesji, a użytkownik tylko z niego korzysta przy użyciu identyfikatora. Możliwe jest również zmodyfikowanie sesji użytkownika w przypadku, gdy wymaga ona rozszerzenia. Sama logika służąca do obsługi sesji jest również stosunkowo prosta w przypadku, gdy sesja ma być używana w obrębie pojedynczego serwera.

Gdy sesja ma być jednak rozdystrybuowana pomiędzy wiele odseparowanych od siebie serwerów, mechanizm do zarządzania nią staje się zdecydowanie bardziej skomplikowany. Najmocniej objawiającym się problemem jest synchronizacja danych sesji pomiędzy poszczególnymi serwerami. Problem mógłby zostać rozwiązany poprzez powiązanie sesji użytkownika z konkretnym serwerem, gdzie mogłoby zostać wysłane żądanie zweryfikowania sesji, ale jest to rozwiązanie wysoce nieskalowalne. W takim scenariuszu wzrost liczby serwerów do obsługi sesji nie zapewniłby wysokiej dostępności, gdyż awaria pojedynczego serwera odpowiedzialnego za zarządzanie sesją spowodowałaby utratę danych zalogowanych użytkowników, których sesje przypisane były do uszkodzonego serwera. Warto mieć też na uwadze, że ze względu na konieczność przechowywania sesji po stronie serwera, wzrost liczby aktywnych użytkowników powoduje zwiększenie ilości wykorzystywanych przez serwer zasobów, co przy dużej liczbie zapytań może negatywnie wpłynąć na wydajność całego systemu.

Podejściem rozwiązującym opisane wyżej problemy jest skorzystanie z bezstanowego (z perspektywy serwera) mechanizmu uwierzytelniania użytkowników. Zostało ono stworzone właśnie w celu zaadresowania problemów związanych z wysoką dostępnością i skalowalnością dostępnych na rynku mechanizmów autoryzacji. W celu przechowywania danych sesji generowany jest

specjalny żeton (ang. *token*) przechowywany przez klienta, który zawiera informacje umożliwiające zidentyfikowanie użytkownika. Aby uniemożliwić modyfikację żetonu po jego stworzeniu, wykorzystywane są mechanizmy kryptograficzne. Żeton jest podpisywany przy użyciu kluczy szyfrujących, najczęściej przy wykorzystaniu pary kluczy asymetrycznych.



Rys. 11: Bezstanowy scenariusz uwierzytelniania

Ponieważ to użytkownik jest odpowiedzialny za przesłanie swojej sesji serwerowi w ramach żądania, po stronie serwera znacząco zmniejsza się narzut wynikający z jej obsługi. Miejsce na przechowywanie danych o sesjach użytkowników staje się niepotrzebne, gdyż wszelkie informacje wymagane do uwierzytelnienia zawarte są w żetonie, każdorazowo wysyłanym wraz z żądaniem. Jedynym zadaniem serwera, które musi zostać wykonane w celu potwierdzenia tożsamości klienta, jest stwierdzenie poprawności podpisu żetonu. Dzięki takiemu podejściu rozwiązany jest problem skalowalności, zauważalny przy podejściu stanowym. Serwer otrzymujący żądanie nie musi przesyłać identyfikatora użytkownika do serwera uwierzytelniającego, a jedynym wymogiem jest dostęp do klucza kryptograficznego pozwalającego stwierdzić poprawność żetonu. Dzięki temu w prosty sposób pojedyncza sesja użytkownika może być użyta do uwierzytelnienia w wielu różnych usługach.

Fakt, że to po stronie użytkownika obecna jest informacja o sesji wpływa negatywnie na bezpieczeństwo takiego rozwiązania. Serwer obsługujący żądanie nie jest w stanie usunąć sesji z pamięci użytkownika, w związku z czym problem potencjalnego wycieku żetonu jest zdecydowanie trudniejszy do obsłużenia. Możliwym rozwiązaniem tego problemu jest unieważnienie żetonu, lecz wymagałoby to implementację mechanizmu służącego do sprawdzania ważności konkretnego żetonu poprzez wykonanie zapytania do serwera uwierzytelniającego oraz mechanizmu umożliwiającego oznaczenie żetonu jako nieważny. Nawet bez wprowadzania takiego rozwiązania, bezstanowe podejście do zagadnienia weryfikacji

tożsamości użytkownika zdaje się być dużo bardziej skomplikowane niż mechanizm wykorzystujący ciasteczka. W związku z tym w przypadku systemów nierozdystrybuowanych czy też aplikacji monolitycznych może okazać się, że implementacja takiego rozwiązania jest zbyteczna, gdyż programista nie zaobserwuje żadnych plusów dostarczanych właśnie przez taki mechanizm. Wartym zaznaczenia jest również fakt, że sesja użytkownika nie może być zmodyfikowana, dopóki nie wygaśnie i klient nie wyśle żądania o odnowienie sesji. O ile w przypadku stanowym nie stanowiło to żadnego problemu, o tyle w omawianym scenariuszu nie jest to tak oczywiste. Co prawda, serwer mógłby w odpowiedzi na żądanie klienta dostarczyć informacje z prośbą o odnowienie sesji klienckiej, jednakże nie istnieje mechanizm, dzięki któremu mógłby się upewnić, że sesja ta faktycznie została odnowiona. Pojawia się tu ponownie problem wstecznej kompatybilności. Ewentualne rozszerzenia komponentów muszą zostać wprowadzone w taki sposób, by obsłużyć możliwość braku pewnych danych w żetonie dostarczanym przez użytkownika.

Pomimo wad, bezstanowa implementacja weryfikacji tożsamości użytkownika świetnie sprawdza się w aplikacjach opartych na modelu funkcja jako serwis, których główną cechą jest łatwość skalowania. Podejście stanowe stanowiłoby wąskie gardło w obsłudze przychodzących zapytań, podczas gdy uwierzytelnianie oparte o żetony tworzone było z myślą, by maksymalnie ograniczyć wpływ tej funkcjonalności na wydajność całej aplikacji.

Świetnym przykładem aplikacji, gdzie bezstanowy mechanizm uwierzytelniania sprawdziłby się idealnie mógłby być prosty serwis do zarządzania plikami graficznymi, udostępniający interfejs programistyczny zgodny z architekturą REST. Jako że jedną z głównych cech tej architektury jest bezstanowość, zastosowanie jej w połączeniu z modelem funkcja jako serwis zdaje się być jedynym słusznym podejściem przy projektowaniu interfejsu. To samo tyczy się wyboru typu mechanizmu uwierzytelniania. Ze względu na bardzo duże rozdrobnienie aplikacji na wiele małych komponentów, zastosowanie stanowego mechanizmu stanowiłoby, wraz ze wzrostem liczby klientów, coraz większy problem.

Serwis został zbudowany w oparciu o serwis AWS Lambda, wykorzystujący serwis Amazon S3 do przechowywania plików. Rolę bramy wejściowej udostępniającej interfejs zgodny z REST pełni Amazon API Gateway, usługa umożliwiająca szybkie tworzenie i sprawne zarządzanie interfejsami programistycznymi aplikacji, zapewniająca również narzędzia do monitoringu

i zabezpieczania tworzonych zasobów. Do wdrożenia projektu użyty został Serverless Framework, a definicja tworzonych w ramach niego zasobów prezentuje się następująco:

```
service: aws-python-img-ops

provider:
  name: aws
  runtime: python3.8
  region: us-east-1
  apiGateway:
    binaryMediaTypes:
      - '*/*'
    metrics: true
  iamRoleStatements:
    - Effect: Allow
      Action:
        - s3:*
      Resource: '*'

functions:
  worker:
    handler: handler.start
    timeout: 10
    environment:
      BUCKET: !Ref Bucket
      REGION: us-east-1
    events:
      - http:
          path: /images
          method: get
      - http:
          path: /images/{image}
          method: get
      - http:
          path: /images/{image}
          method: put
      - http:
          path: /images/{image}
          method: delete

resources:
  Resources:
    Bucket:
      Type: AWS::S3::Bucket
      DeletionPolicy: Retain

plugins:
  - serverless-s3-local
  - serverless-offline
  - serverless-python-requirements

custom:
  pythonRequirements:
    dockerizePip: "non-linux"
```

Wartym wyróżnienia jest użycie podsekcji *apiGateway* w ramach definicji zasobu *provider* oraz zastosowanie typu *http* przy definiowaniu zdarzeń

wyzwalających stworzoną funkcję. W ramach parametryzowania podsekcji odpowiedzialnej za konfigurację bramy wejściowej załączona została informacja o wspieranych binarnych typach mediów, które mogą być przesyłane w ramach komunikacji z aplikacją. Jawna definicja wspieranych typów mediów była wymagana w celu umożliwienia poprawnego zwracania w odpowiedziach danych takich jak obrazy czy pliki. Odbywa się to w połączeniu z umieszczeniem właściwego identyfikatora zwracanego typu w nagłówku odpowiedzi *Content-Type*. Dodatkowo, w ramach omawianej podsekcji zawarto informację o włączeniu rozszerzonych metryk, dzięki czemu w ramach monitoringu serwisu możliwa jest kontrola każdego z punktów końcowych osobno. Z kolei wpisy obecne w sekcji *events* wewnątrz definicji funkcji odpowiadają zdefiniowanym punktom końcowym, do których użytkownik może się odwoływać przy użyciu odpowiednich metod http. W ramach omawianego serwisu zdefiniowane następujące punkty końcowe:

- `GET /images`

  Operacja umożliwiająca pobranie informacji o wszystkich plikach graficznych zarejestrowanych w serwisie. Zwracana odpowiedź w formacie JSON:

  ```
  {
      "message": [
          {
              "Key": 'string',
              "Size": 123
          },
      ]
  }
  ```

  message – lista słowników

  Key (ciąg znaków) – nazwa pliku

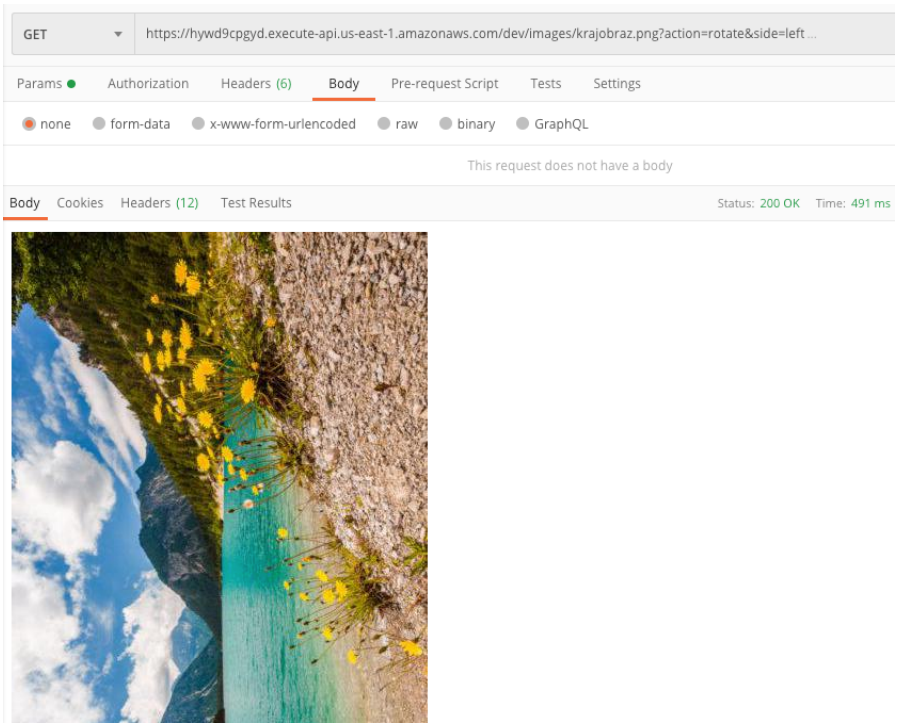
  Size (liczba całkowita) – rozmiar pliku (wyrażony w bajtach)

- `GET /images/{image}`

  Operacja służąca wyświetleniu pliku graficznego o nazwie `{image}`. Umożliwia ona również dokonywanie modyfikacji wyspecyfikowanej w ramach zapytania. W przypadku istniejącego klucza, użytkownik otrzyma o tym informację poprzez uzyskanie kodu odpowiedzi 200. Jeśli klucz nie istnieje, zwrócony zostanie komunikat „nie istnieje” wraz z kodem 404. Dostępne modyfikatory obrazka:

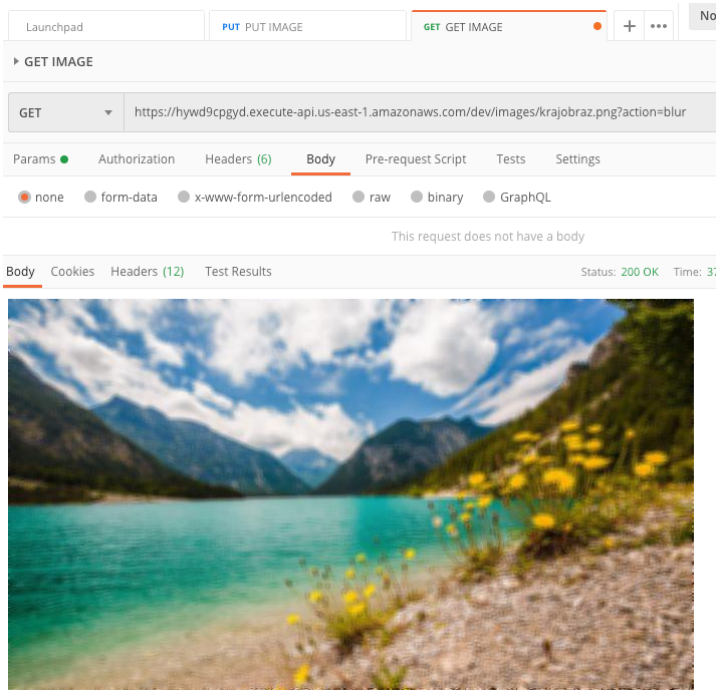
  - *rotate* – umożliwia wyświetlenie obróconego obrazka, specyfikując rodzaj obrotu przy użyciu parametru *side* o wartościach: *left* (obrazek obrócony o 90°

w lewo), *right* (90° w prawo) lub *top-down* (obrazek wyświetlony do góry nogami). Przykład: `GET /images/image.png?action=rotate&side=left`



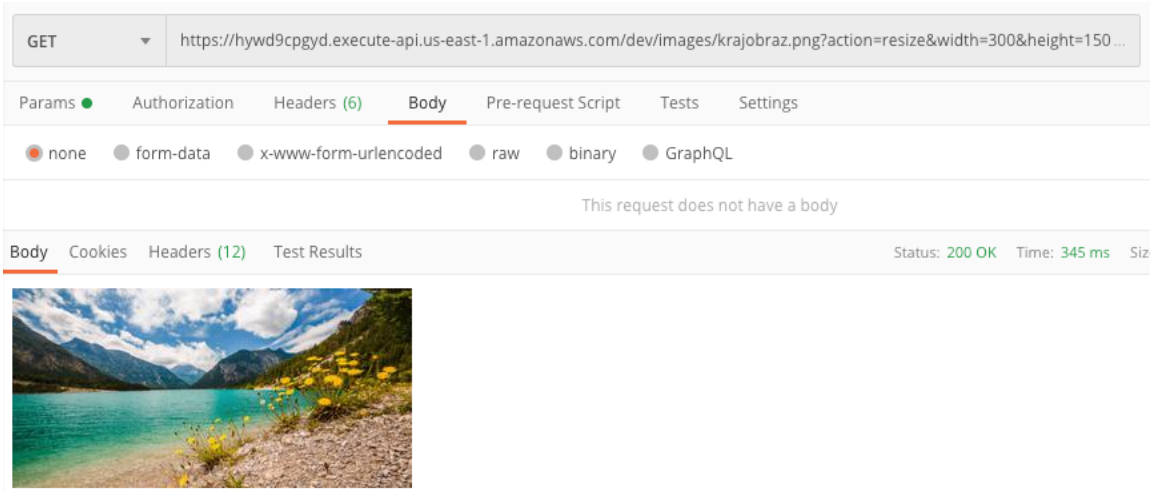
Rys. 12: Przykład wykonania operacji obrotu obrazka

- *blur* – służy do rozmazania pliku graficznego.
  Przykład: `GET /images/image.png?action=blur`



Rys. 13: Przykład wykonania operacji rozmazania obrazka

- *resize* – umożliwia modyfikację rozmiaru wyświetlanej grafiki przy użyciu dodatkowych parametrów odpowiadających za szerokość (*width)* i wysokość (*height)*. Przykład:
  `GET /images/image.png?action=resize&width=200&height=200`



Rys. 14: Przykład wykonania operacji obrotu grafiki

- `PUT /images/{image}`

  Operacja służąca do przesłania grafiki do serwisu celem późniejszego zapisania jej w ramach przechowalni plików Amazon Simple Storage Service. W ramach zapytania utworzony zostanie plik o nazwie `{image}`, o ile nazwa ta nie jest wykorzystywana przez inny zasób znajdujący się w ramach kubełka przechowującego zasoby. Poprawnemu umieszczeniu grafiki towarzyszyć będzie otrzymana z aplikacji wiadomość „stworzono” wraz z kodem odpowiedzi 201. W przypadku, gdy nazwa pliku jest już zajęta, zwróconym kodem odpowiedzi będzie 409, a w przypadku próby przesłania przez użytkownika pliku niebędącego grafiką otrzyma on o tym informację poprzez kod 400. Grafika winna być przesłana w formie strumienia bitowego, co przy użyciu aplikacji Postman można uzyskać poprzez wybór opcji *binary* jako typu przesyłanych danych i załączenie odpowiedniej grafiki w przesyłanym żądaniu.
- `DELETE /images/{image}`

  Operacja służąca do usunięcia zasobu o identyfikatorze {image}. W przypadku, gdy zasób istnieje, zostaje on skasowany, a akcji tej towarzyszy zwrócenie odpowiedzi wraz z kodem 200. Gdy podany przez klienta klucz jest niepoprawny (grafika o danej nazwie nie istnieje w kontekście aplikacji), otrzyma on odpowiedź

ze stosownym komunikatem zawartym w ciele odpowiedzi wraz z towarzyszącym mu kodem odpowiedzi 404.

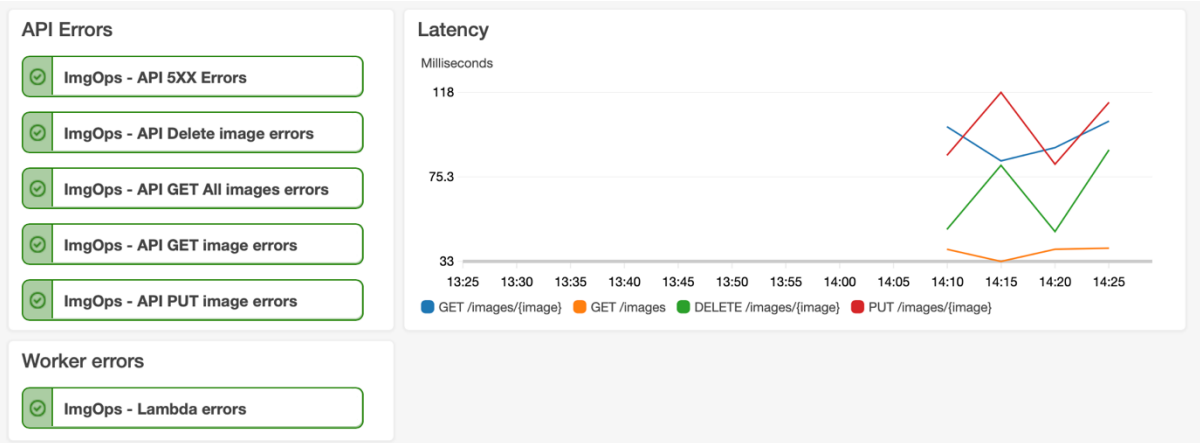
Kod realizujący przekierowywanie żądania do odpowiedniej funkcji odpowiedzialnej za obsługę zapytania wygląda następująco:

```
import boto3
import botocore
import json
from collections import namedtuple
from src.aws.handler import (
    listImages,
    getImage,
    putImage,
    removeImage
)

handlers = {
    '/images': {
        "GET": listImages
    },
    '/images/{image}': {
        "GET": getImage,
        "PUT": putImage,
        "DELETE": removeImage
    }
}

def start(event, context):
    try:
        handler = handlers[event['resource']][event['httpMethod']]
    except KeyError as err:
        print(err)
        return wrongRequest()

    return handler(event)

def wrongRequest():
    return {
        "statusCode": 400,
        "body": json.dumps({
            "error": "Wrong method or resource"
        })
    }

def get_log_event(e):
    to_log = {x: e.get(x) for x in ('resource', 'path', 'httpMethod',
'pathParameters', 'body')}
    to_log['requestId'] = e['requestContext']['requestId']
    return to_log
```

Cały kod aplikacji dostępny jest w ramach repozytorium GitHub, dostępnego pod adresem https://github.com/pawelaugustyn/praca-magisterska, w folderze *aws-python-img-ops*.

Celem monitorowania aplikacji udostępniającej w swoich ramach interfejs programistyczny zgodny z architekturą REST należy skupić się na przede wszystkim na dwóch komponentach: bramie wejściowej oraz części aplikacyjnej. Dzięki wykorzystaniu serwisów dostarczanych w ramach chmury Amazon Web Services, zadanie to można zrealizować przy użyciu usługi Amazon CloudWatch, która posiada integrację zarówno z serwisem AWS Lambda wykorzystywanym do obsługi żądań, jak i Amazon Api Gateway stanowiącym bazę dla zbudowanej bramy wejściowej. Metryki, które mogą interesować administratora systemu, odnoszą się do błędów zwracanych przez oba serwisy.

W ramach monitoringu opracowano agregator metryk przy użyciu serwisu Amazon CloudWatch, który zbiera informacje dotyczące błędów występujących w aplikacji oraz czasów odpowiedzi dla każdego z dostępnych punktów końcowych. Było to możliwe dzięki włączeniu rozszerzonego monitoringu dla bramy wejściowej.



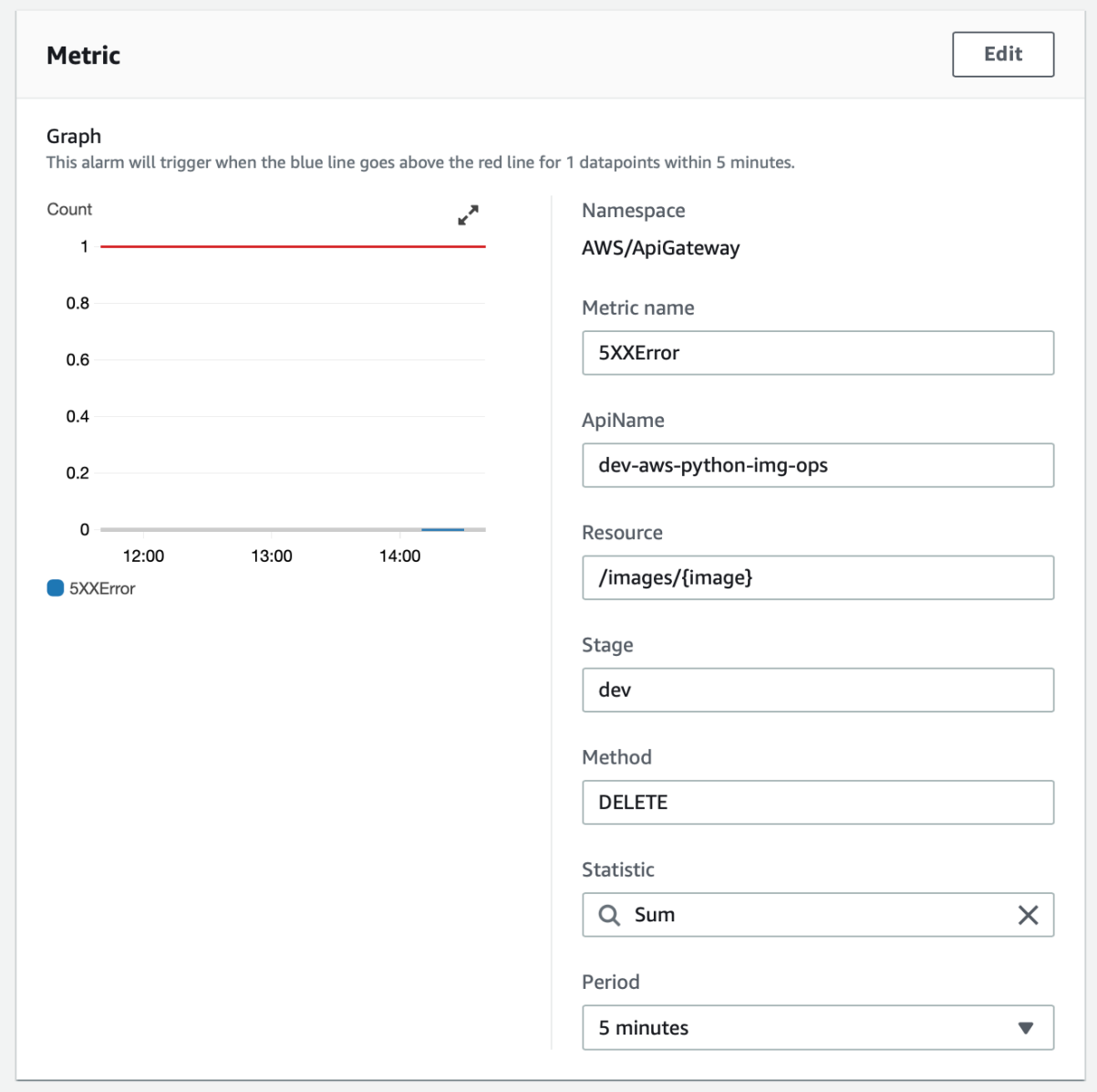
Rys. 15: Agregator wartościowych dla serwisu metryk

Obecny na Rys. 15 wykres przedstawiający czasy odpowiedzi został zbudowany przy użyciu funkcji tworzenia wykresów, szerzej opisanej w ramach rozdziału 3. Widoczne wartości to średnie czasy odpowiedzi dla każdego z punktów końcowych, zgrupowane w 5-minutowych przedziałach (klucz *period*). Z kolei, klucz *liveData* zapewnia wyświetlanie danych na bieżąco, gdyż ustawienie jego wartości na wartość prawdziwą powoduje pokazanie również dla niekompletnych okresów 5-

minutowych, co odnosi się w głównej mierze do danych populowanych na żywo w trakcie odczytu danych z wykresu. Pełna definicja stworzonego wykresu zapisana jest poniżej jako słownik w formacie JSON.

```
{
    "metrics": [
        [ "AWS/ApiGateway", "Latency", "ApiName", "dev-aws-python-img-ops",
"Resource", "/images/{image}", "Stage", "dev", "Method", "GET" ],
        [ "...", "/images", ".", ".", ".", "." ],
        [ "...", "/images/{image}", ".", ".", ".", "DELETE" ],
        [ "...", "PUT" ]
    ],
    "view": "timeSeries",
    "stacked": false,
    "region": "us-east-1",
    "period": 300,
    "stat": "Average",
    "liveData": true,
    "setPeriodToTimeRange": true
}
```

Widoczny po lewej stronie na Rys. 15 stan aplikacji to widżet grupujący stworzone na potrzeby serwisu alerty. Wszystkie z nich opierają się na zbieraniu danych z metryki o nazwie *5XXError*, oznaczającej zwrócenie w ramach odpowiedzi kodu o wartości większej niż 500, co zgodnie ze standardem http w wersji 1.1 [8] oznacza błąd w przetwarzaniu żądania po stronie serwera. Oprócz zdefiniowania alertów odpowiadających za każdy z punktów końcowych osobno, twórca aplikacji zdecydował się na wprowadzenie alarmu opartego o wystąpienia kodu odpowiedzi większego niż 500 w ramach całej bramy wejściowej. Ma to na celu uniknięcia sytuacji, gdy w ramach rozwoju aplikacji poprzez dodanie nowego punktu końcowego, w wyniku niedopatrzenia administratora, występujące w ramach jego wywołań błędy nie zostaną wykryte. Prawdą jest, że zidentyfikowanie który z punktów końcowych zachowuje się nieprawidłowo nie jest już tak oczywiste, jednak zdefiniowanie alertu agregującego błędy występujące w ramach całego serwisu jest zdecydowanie pomocne w kontekście utrzymania aplikacji.



Rys. 16: Przykładowa definicja metryki dla błędów punktu końcowego bramy wejściowej

W lewym dolnym rogu Rys. 15 widoczny jest również alert odpowiadający za wykrywanie błędów funkcji AWS Lambda odpowiadającej za obsługę przychodzących żądań. Zdefiniowanie alertów dla komponentów obsługujących zapytania daje ogólną wiedzę na temat poprawnego bądź nieprawidłowego działania aplikacji, lecz nie daje pełnego oglądu na działanie całego serwisu, którego częścią jest również API zgodne ze stylem REST. Wynika to z faktu, że z pozoru poprawne działanie komponentu odpowiedzialnego za realizację logiki biznesowej nie jest równoznaczne równoznacznemu zachowaniu serwisu. W sytuacji, gdy zwrócona przez funkcję AWS Lambda odpowiedź nie odpowiada oczekiwanemu przez bramę wejściową formatowi dochodzi do pozornie rozbieżnego stanów alarmów

odpowiedzialnych za monitorowanie zachowania bramy wejściowej oraz funkcji. Podczas gdy alert przypisany do interfejsu programistycznego zostanie wywołany, z perspektywy działania komponentu realizującego obsługę zapytania wszystko wydaje się w porządku. Jest to spowodowane faktem, że niepoprawnie sformułowane wyjście, odebrane przez serwis Amazon Api Gateway, nie może być wykorzystane do sformułowania odpowiedzi dla użytkownika. Z tego właśnie powodu niezwykle istotne jest monitorowanie obu części aplikacji – tej dostępnej dla użytkownika oraz komponentów niedostępnych z zewnątrz, do których finalnie trafiają zapytania. W połączeniu z odczytaniem informacji dostępnej w logach pozwala to na szybkie zidentyfikowanie miejsca, gdzie występuje problem.

Stworzona aplikacja opiera się na modelu bezserwerowym, przez co uwierzytelnianie oparte o żetony wydaje się być idealnie dopasowanym rozwiązaniem. Klucz wykorzystywany do podpisania żetonu mógłby zostać dostarczony do funkcji AWS Lambda poprzez parametr konfiguracyjny (przekazanie przez zmienną środowiskową) lub pobierany przy każdym uruchomieniu funkcji poprzez wysłanie zapytania do wybranego serwera uwierzytelniającego, którego adres mógłby być przekazany w ten sam sposób. Zastosowanie takiego podejścia eliminuje potrzebę korzystania z miejsca przechowującego dane o użytkownikach posiadających dostęp do zadanej funkcjonalności, gdyż w ramach żetonu zawarte mogą być dowolne dane pozwalające na autoryzację danych zapytań. Przykładem implementującym taką funkcjonalność jest rozwiązanie JSON Web Token (JWT).

Ponieważ żeton posiada formę słownika JSON kodowanego w formacie *base64*, operator odpowiedzialny za konfigurację serwera pozwalającego na tworzenie żetonów jest w stanie zdefiniować klucze, pod którymi przechowywane są dowolne dane. W celu określenia dostępu do pewnych zasobów stosuje się tzw. mechanizm zakresów (ang. *scopes*). Polega on na utworzeniu w ramach żetonu klucza o nazwie *scope*, w ramach którego przechowywana jest informacja o zasobach dostępnych dla użytkownika identyfikującego się zawartym w zapytaniu żetonem. Mechanizm ten umożliwia nie tylko określenie serwisów, do których klient ma dostęp, lecz również wyspecyfikowanie akcji, jakie dany użytkownik może wykonać. Zadaniem programisty odpowiadającego za stworzenie danego serwisu jest określenie możliwych zakresów uprawnień oraz odpowiednie ich egzekwowanie w momencie weryfikacji poprawności żetonu.

5. Porównanie rozwiązań zgodnych z modelem funkcja jako serwis dostępnych w ramach wybranych dostawców publicznych chmur obliczeniowych

Rosnąca popularność rozwiązań chmurowych powoduje, że każdy z dostawców publicznych chmur stara się wprowadzić jak najszerszy i najlepszy zakres dostarczanych usług. Zgodnie z najnowszym trendem, stawiającym modele bezserwerowe na topie ze względu na małe koszty oraz niski czas wdrażania aplikacji, chmury takie jak Amazon Web Services, Google Cloud Platform czy też Microsoft Azure zapewniają dostęp do szerokiej gamy rozwiązań opartych na tych właśnie modelach. Jest to odpowiedź na stale rosnące zapotrzebowanie na tego typu usługi oraz chęć zwiększenia swojej konkurencyjności, co ma zachęcić możliwie najszersze grono klientów do wyboru swojej firmy jako dostawcy rozwiązań chmury publicznej.

W ramach chmury Amazon Web Services serwisem udostępnianym w ramach modelu funkcja jako serwis jest AWS Lambda. Został on udostępniony dla wszystkich klientów tego dostawcy w listopadzie 2014 roku, będąc pierwszą tego rodzaju usługą dostępną w ofercie dużego gracza na rynku publicznych chmur obliczeniowych. Wdrożenie tej usługi spowodowało znaczący wzrost popularności takowych rozwiązań, co poskutkowało masowym wdrażaniem podobnych usług przez innych dostawców – Google Cloud Functions w przypadku chmury Google Cloud Platform oraz Azure Functions, serwisu dostępnego w ofercie produktu Microsoft Azure. Wspomniane rozwiązania opierają się na zbliżonych do siebie założeniach, jednak obecne w ich implementacjach różnice wpływają na unikalność każdego z serwisów.

Przedmiotem tego rozdziału jest omówienie różnic w procesie tworzenia prostej aplikacji bezserwerowej celem wdrożenia jej na chmury Google Cloud Platform, Microsoft Azure oraz Amazon Web Services, jak i porównanie rozwiązań oferowanych przez wspomnianych dostawców publicznych chmur obliczeniowych. W ramach wszystkich wymienionych chmur stworzono prostą aplikację opartą na modelu funkcja jako serwis, wykonującą prostą operację wygenerowania tablicy pseudolosowo wygenerowanych wartości numerycznych oraz jej posortowania. Po wdrożeniu aplikacji, parametry funkcji zostały dostosowane w taki sposób, aby czasy realizacji wszystkich przychodzących zapytań były do siebie zbliżone, niezależnie od tego, w usłudze którego dostawcy odbyło się wdrożenie. Ponieważ serwis Azure Functions

nie daje możliwości wykonania takiej konfiguracji, gdyż wszystkie funkcje uruchomione w jego ramach funkcje posiadają przypisane dwa gigabajty pamięci operacyjnej wraz z nieokreśloną ilością wirtualnych rdzeni procesora (brak informacji w dokumentacji), aplikacje wdrożone w chmurach Amazon Web Services oraz Google Cloud Platform wysterowano tak, aby maksymalnie zbliżyć średnie czasy odpowiedzi dla wszystkich trzech serwisów. Limit dla serwisu dostępnego w chmurze Microsoft Azure nie dotyczy wersji premium omawianej usługi, jednak dostęp do takiego planu wiązał się z dodatkowymi opłatami, co spowodowało konieczność wykorzystania podstawowej wersji tego serwisu. Sama zaś operacja wysterowania polega na przypisaniu pożądanej ilości dostępnej pamięci operacyjnej dla każdej uruchamianej instancji funkcji, co wiąże się z proporcjonalnym wzrostem mocy obliczeniowej. Finalna ilość wirtualnych rdzeni procesora pozostaje nieznana w przypadku serwisów w chmurach AWS i Azure, natomiast w przypadku serwisu Google Cloud Functions na każdy gigabajt pamięci operacyjnej przysługuje procesor o zegarze jednego gigaherca. W wyniku konfiguracji, pojedynczy czas wykonywania kodu (suma czasów inicjalizacji tablicy oraz jej sortowania) aplikacji wdrożonej w usługach Google Cloud Functions, Azure Functions oraz AWS Lambda wynosi w przybliżeniu 2.7 sekundy. Wartość ta nie uwzględnia czasu zmierzonego z perspektywy użytkownika wykonującego zapytanie do aplikacji.

Aplikacja służąca do przeprowadzenia testów składa się z dwóch elementów – prostej klasy odpowiadającej za wygenerowanie tablicy, posortowanie jej i zmierzenia czasu tych dwóch czynności oraz punktu wejścia-wyjścia, który różni się w zależności od dostawcy chmury obliczeniowej. Opisywana klasa prezentuje się następująco:

```
class Sorter:
    def __init__(self):
        initialize_time = time.time()
        self.__list = [randint(0, 1000) for _ in range(1000000)]
        self.__init_time = time.time()-initialize_time

    def sort(self):
        start_time = time.time()
        sorted(self.__list)
        end_time = time.time()

        return {
            "sortTime": end_time - start_time,
            "initTime": self.__init_time
        }
```

Każdy z dostawców definiuje dość podobny do siebie interfejs służący do obsługi żądań przychodzących do funkcji serverless. W przypadku chmury Google Cloud Platform wystarczającym jest zwrócenie pożądanej wartości na wyjściu, aby ta została przekazana użytkownikowi wykonującemu zapytanie, natomiast w przypadku pozostałych omawianych rozwiązań, należy je dodatkowo opakować w zdefiniowane w ramach dokumentacji struktury. Różnice te zostały przedstawione w załączonych poniżej fragmentach kodu aplikacji, specyficznych dla poszczególnych rozwiązań:

- AWS Lambda

```python
from sorter import Sorter
import json

def start(event, context):
    sorter = Sorter()
    times = sorter.sort()

    return {
        "statusCode": 200,
        "body": json.dumps(times)
    }
```

- Google Cloud Functions

```python
from sorter import Sorter
import json

def start(request):
    sorter = Sorter()
    times = sorter.sort()

    return json.dumps(times)
```

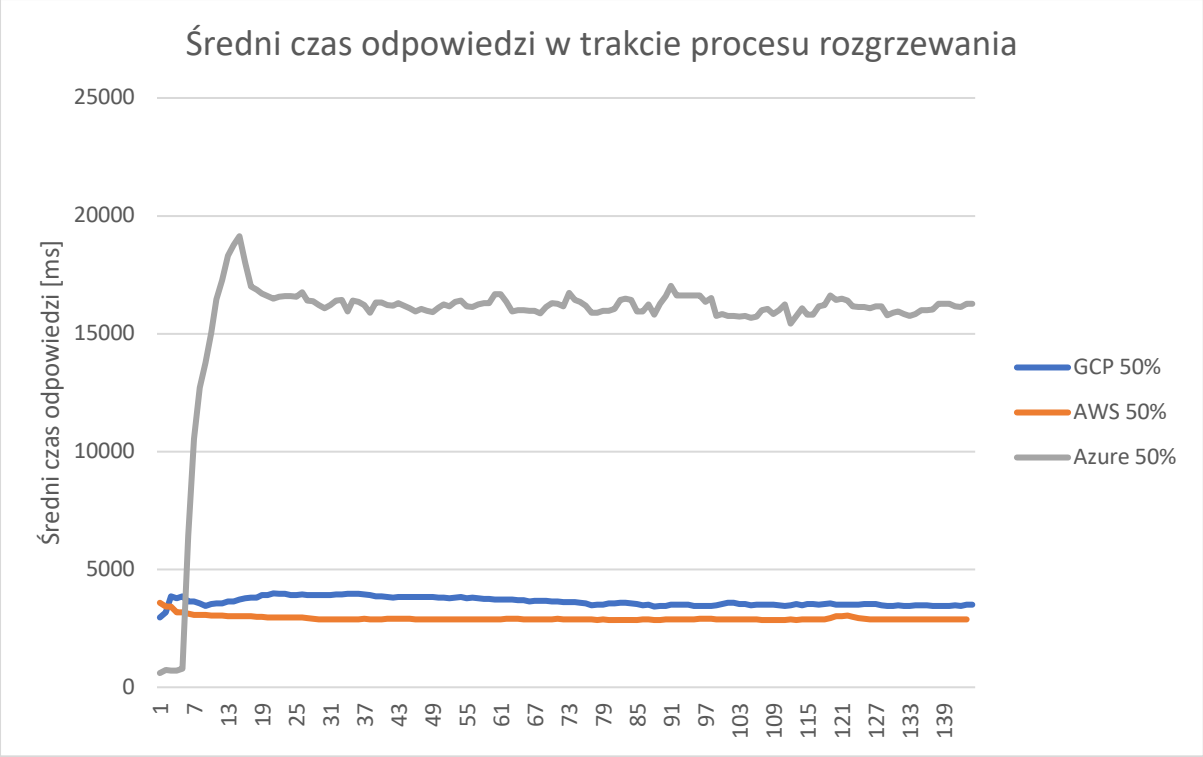
- Azure Functions

```python
from sorter import Sorter
import json
import azure.functions as func

def main(req: func.HttpRequest) -> func.HttpResponse:
    sorter = Sorter()
    times = sorter.sort()

    return func.HttpResponse(json.dumps(times))
```

Kod opisywanych aplikacji dostępny jest w ramach repozytorium GitHub, dostępnego pod adresem https://github.com/pawelaugustyn/praca-magisterska.

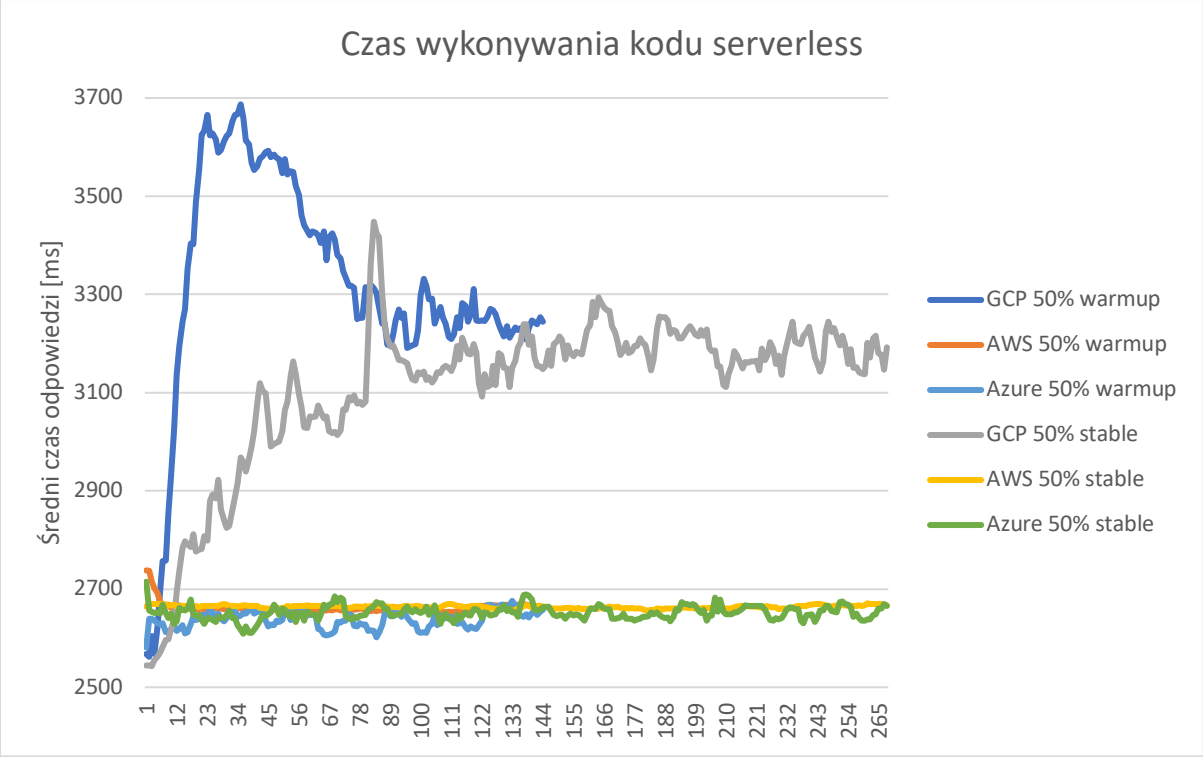
Celem przeprowadzenia testów wydajnościowych, wykorzystano zmodyfikowaną wersję programu Locust, napisanego w języku Python. Jest to popularne narzędzie służące do wykonywania testów obciążeniowych, których celem jest sprawdzenie, jaki wpływ na działanie testowanej aplikacji ma ilość użytkowników jednocześnie z niej korzystających. W trakcie pierwszej fazy przeprowadzenia testów okazało się, że możliwości wspomnianego narzędzia są niewystarczające dla potrzeb autora pracy, gdyż produkt ten służy wyłącznie do badania czasów odpowiedzi z perspektywy użytkownika końcowego. Ponieważ czas wykonywania kodu aplikacji był dostępny w ramach odpowiedzi zwracanej przez serwis, Locust został rozszerzony o możliwość odczytu odpowiedzi celem pobrania z niej interesujących użytkownika wartości. Dzięki temu możliwe było zmierzenie różnicy pomiędzy czasem oczekiwania przez użytkownika na odpowiedź, a czasem wykonywania funkcji serverless, co można w mniej lub bardziej bezpośredni sposób przełożyć na jakość oferowanych rozwiązań pracujących w modelu funkcja jako serwis.

Test podzielono na dwie części – tzw. proces nagrzewania oraz operowanie na rozgrzanych komponentach. Proces rozgrzewania polega na stopniowym wzroście liczby użytkowników (w tym przypadku było to dodawanie 3 użytkowników w każdej sekundzie) aż do osiągnięcia zadanego w warunkach początkowych limitu, który wynosił 240 użytkowników. Użytkownicy wykonywali zapytanie, a po otrzymaniu odpowiedzi wstrzymywali się z wysłaniem kolejnego żądania na czas z przedziału sekunda – dwie sekundy. Proces rozgrzania ma na celu przygotowanie infrastruktury na zwiększenie przychodzącego ruchu. W przypadku każdego z serwisów, instancja bezserwerowego komponentu nie jest wygaszana z chwilą zakończenia obsługi żądania, lecz pozostaje ona w stanie oczekiwania na kolejne żądania przez bliżej nieokreślony czas. Zmniejsza to ryzyko wystąpienia tzw. zimnego startu, czyli sytuacji, kiedy w czas obsługi żądania wchodzi również czas oczekiwania na przygotowanie infrastruktury celem uruchomienia aplikacji, co zwiększa czas wymagany na otrzymanie odpowiedzi przez użytkownika. Z kolei przypadki testowe badające zachowanie rozgrzanych komponentów trwały po 10 minut.



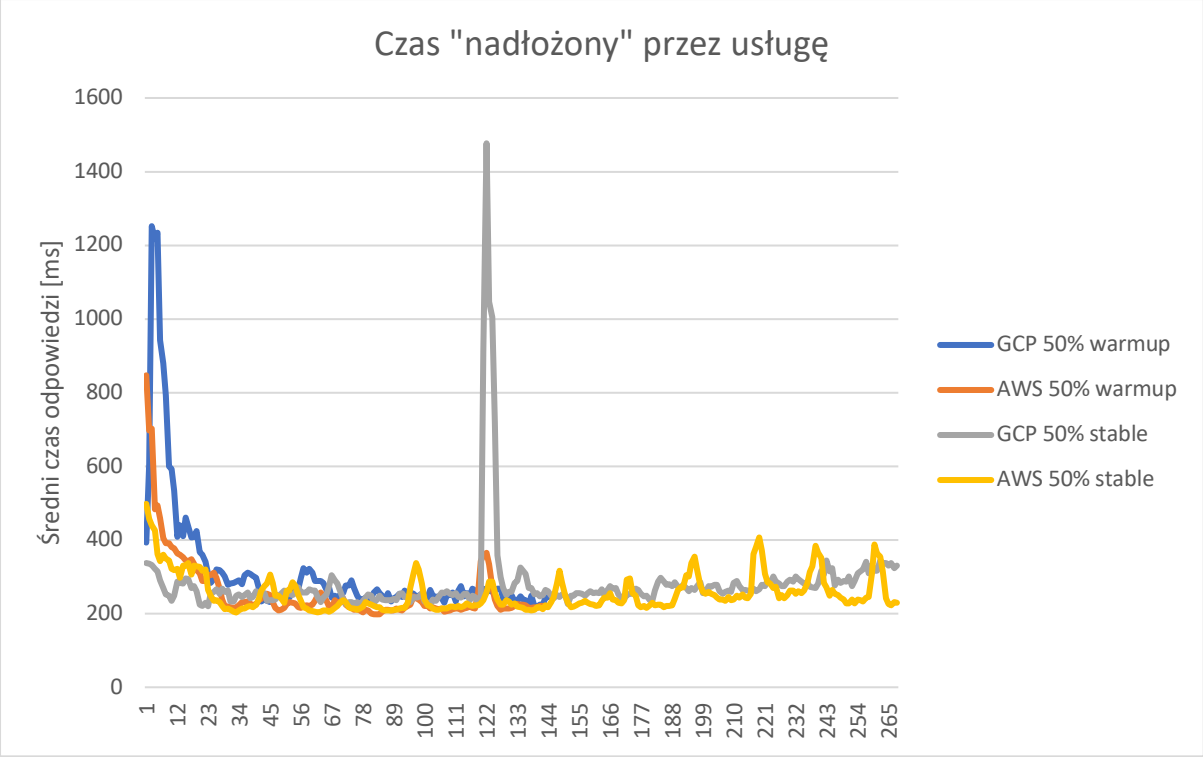
Rys. 17: Średni czas odpowiedzi w trakcie procesu rogrzewania

Jako pierwszy zmierzony został czas oczekiwania na odpowiedź widoczny z perspektywy użytkownika. Pomiar ten wykonano w trakcie omówionego w poprzednim akapicie procesu rozgrzewania funkcji. Na Rys. 17 przedstawiono średnie czasy odpowiedzi zaobserwowane w trakcie trwania przypadku testowego, którego długość wynosiła 5 minut. Już na pierwszy rzut oka widoczna jest bardzo duża różnica pomiędzy rozwiązaniami oferowanymi w ramach chmur GCP i AWS, a rozwiązaniem dostępnym w chmurze Microsoft Azure. Wynika to najprawdopodobniej z braku możliwości skorzystania z wersji premium serwisu Azure Functions, której to dokumentacja mówi o aktywowaniu funkcji rozgrzewania. Zatem, dla wersji standardowej proces ten jest najprawdopodobniej niedostępny, co wpływa na znacząco mniejszą wydajność.



Rys. 18: Pomiar czasów wykonywania kodu

Następnie przeprowadzono pomiar czasów wykonywania kodu, czyli realnego czasu działania aplikacji od momentu otrzymania zapytania do wykonania zleconych zadań i zwrócenia odpowiedzi. Na Rys. 18 umieszczono pomiary zaobserwowane w trakcie procesów rozgrzewania (oznaczonych na wykresie jako *warmup*) oraz właściwego działania aplikacji (podpisanych jako *stable*). Tutaj z kolei widać dość duże rozbieżności pomiędzy rozwiązaniem Google'a, a pozostałymi. Wartymi zaobserwowania są bardzo zbliżone czasy wykonywania kodu po stronie chmur AWS oraz Azure, co dość mocno kontrastuje z danymi z Rys. 17, gdzie czas realizacji zapytania widoczny przez użytkownika w przypadku usługi Azure'a był znacząco większy od pozostałych rozważanych produktów. Z tego względu na Rys. 19, przedstawiającym narzut czasowy dodany przez oprogramowanie dostawców rozwiązań funkcja jako serwis zdecydowano się nie umieszczać danych pomiarowych dla chmury Microsoft Azure, gdyż zbyt mocno zaciemniały one cały pomiar z powodu kilkudziesięciokrotnie wyższych wartości.



Rys. 19: Czas nadłożony przez usługę

Podczas mierzenia narzutu czasowego usług funkcji jako serwis zaobserwowano piki w przypadku rozwiązania dostępnego w chmurze Google Cloud Platform. Ponieważ te nieregularne zachowania widoczne były w zdecydowanej większości przypadków testowych, podczas dokonywania analizy zebranych wyników zdecydowano się zachować widoczne na wykresie maksimum dla serii „GCP 50% *stable*", odpowiadającej średnim czasom obsługi zapytania bez czasu wykonywania samego kodu w przypadku rozgrzanym. Powtarzalność takich wyników może świadczyć o pewnej niestabilności testowanego rozwiązania.

Zaprezentowane wyniki pokazują, że dla testowanej aplikacji najlepszą wydajność osiągnięto dla wdrożenia w ramach serwisu AWS Lambda. Może to wynikać z pewnego rodzaju dojrzałości tego rozwiązania, gdyż spośród omawianych trzech usług, to właśnie AWS Lambda została wdrożona jako pierwsza. Warto jednak zauważyć, że dla aplikacji o innej charakterystyce wykorzystania pamięci operacyjnej oraz czasu procesora wyniki te mogłyby rozłożyć się zupełnie inaczej. Deweloperzy powinni zatem mieć na uwadze, że dla każdej tworzonej aplikacji należy wykonać odpowiedni zestaw testów w ramach usług, pomiędzy którymi dokonany ma być wybór, aby upewnić się, że wybrano rozwiązanie najlepiej dostosowane do aktualnych

potrzeb. Dodatkowo, serwis Azure Functions w wersji premium mógłby okazać się dużo lepszym konkurentem dla pozostałych usług w porównaniu do podstawowej wersji oferowanych w jego ramach usług.

Ponieważ wszystkie omawiane serwisy opierają się na modelu funkcja jako serwis, proces tworzenia oraz wdrażania aplikacji nie różni się znacząco. Największą różnicą jest sposób definiowania bramy wejściowej do serwisu w ramach chmury AWS w porównaniu do chmur Azure oraz GCP. W przypadku tych drugich, definiując punkty końcowe bramy wejściowej deweloper określa ścieżkę, pod jaką dostępny jest zasób, a rozróżnienie metody http użytej do zrealizowania zapytania jest częścią implementacji aplikacji. Z kolei, w ramach serwisu Amazon Api Gateway funkcję obsługującą zapytanie przypisuje się do pary ścieżka – metoda http. To, czy jest to lepsze bądź gorsze podejście do rozwiązywania tej kwestii pozostaje subiektywną oceną dewelopera tworzącego aplikację. Kolejną różnicą są możliwości konfiguracji komponentów. W przypadku serwisu AWS Lambda, ilość przypisanej pamięci operacyjnej dla pojedynczej funkcji może być równa dowolnej liczbie z zakresu 128-3092 megabajty, będącej wielokrotnością liczby 64. Funkcji zdefiniowanej w ramach serwisu Google Cloud Functions można przypisać 128, 256, 512, 1024 lub 2048 megabajtów pamięci, co jest dużo mniej elastycznym rozwiązaniem. Z kolei dla Azure Functions w wersji standardowej nie przewidziano żadnej możliwości sterowania przypisanymi zasobami.

Przygotowanie aplikacji działającej w modelu funkcja jako serwis oraz wdrożenie jej nie jest zbyt skomplikowanym i czasochłonnym zajęciem, a wybór dostawcy chmury obliczeniowej nie wpływa znacząco na czas wymagany na przygotowanie takiego programu. Z tego powodu przy podejmowaniu decyzji, z oferty którego dostawcy powinno się skorzystać, warto zastanowić się nad kierunkiem rozwoju nie tylko aktualnie tworzonej aplikacji, ale całego rozwiązania, jakie ma powstać, gdyż umożliwi to dokonanie wyboru na podstawie innych usług oferowanych w ramach wybranej chmury publicznej. Warto również skupić się na aspekcie finansowym wybranego rozwiązania, aby optymalizując koszty móc jak najlepiej wykorzystać oferowane przez dostawcę usługi.

6. Zakończenie

Model funkcja jako serwis to stosunkowo świeże podejście do tworzenia aplikacji, które z roku na rok zdobywa coraz większą popularność. Pomimo dość młodego wieku jest to już na tyle dojrzałe rozwiązanie, że coraz więcej firm decyduje się skorzystać właśnie z takiego rodzaju architektury przy projektowaniu nowoczesnych systemów rozproszonych. Z tego też powodu, rozwiązania oparte na tym modelu są dostępne w ramach usług coraz większej ilości publicznych chmur obliczeniowych.

Oprócz modelu funkcja jako serwis warto pochylić się również nad zagadnieniem usług bezserwerowych, będącym nieco szerszym tematem. Ich głównym założeniem jest zapewnienie wysokiego poziomu skalowalności i niezawodności przy zachowaniu wysokiego poziomu bezpieczeństwa. Zastosowanie takich usług ułatwia stworzenie aplikacji, która w każdej chwili będzie gotowa na przyjęcie dużej liczby użytkowników, chcących otrzymać dostęp do oferowanych w jej ramach zasobów.

Chociaż model funkcja jako serwis został stworzony głównie z myślą o aplikacjach bezstanowych, nie implikuje to konieczności zastosowania innego modelu w przypadku tworzenia aplikacji stanowych, gdyż dzięki dodatkowym narzędziom udostępnianym przez dostawców chmur obliczeniowych jest to wykonalne. Opisana w ramach rozdziału 3. aplikacja korzystająca z serwisu AWS Step Functions może być przykładem takiego właśnie serwisu.

Rozdział 4. opisuje z kolei w jak prosty i szybki sposób możliwe jest stworzenie aplikacji, z którą komunikacja następuje przy użyciu protokołu HTTP/1.1 poprzez wysyłanie żądań do punktów końcowych zgodnych ze standardem REST. Pokazuje to jak model funkcja jako serwis może być zastosowany do stworzenia małego mikroserwisu, łącząc plusy obu tych rozwiązań.

Choć dostawcy publicznych chmur obliczeniowych udostępniający usługę umieszczania własnych aplikacji stworzonych w modelu będącym tematem tej pracy stworzyli bardzo podobne rozwiązania, można dostrzec pomiędzy nimi różnice zarówno w możliwościach konfiguracji wdrożenia, jak i wydajności działania samej aplikacji. Daje to możliwość wyboru rozwiązania odpowiednio do wymagań zdefiniowanych dla tworzonych aplikacji.

Celem pracy było przedstawienie modelu funkcja jako serwis poprzez opis teoretyczny poparty odpowiednimi przykładami aplikacji, które przedstawiają dobre oraz słabe strony omawianego modelu. Warto jednak zaznaczyć, że opisywane przykłady to tylko kropla w morzu potencjalnych zastosowań tego modelu. Z kolei porównanie do nieco bardziej wiekowych metod wdrażania aplikacji, takich jak używanie dedykowanych serwerów czy też kontenerów pokazuje, że rozwój przebiega w kierunku wydzielania komponentów celem ułatwienia pracy programiście. Proces tworzenia tej pracy był bardzo miłym okresem dla autora, który jeszcze bardziej utwierdził się w przekonaniu o słuszności wykorzystywania modelu funkcja jako serwis do tworzenia rozwiązań informatycznych.

Bibliografia

[1] M. Bejda, „How a Camera Company Started The Serverless Revolution (Then Killed It),” [Online]. Adres: https://www.mbejda.com/how-a-camera-company-started-the-serverless-revolution-then-killed-it/. [Data uzyskania dostępu: 10 czerwca 2020].

[2] Jennifer Garfinkel, „Gartner Identifies the Top 10 Trends Impacting Infrastructure and Operations for 2019,” 4 Grudzień 2018. [Online]. Adres: https://www.gartner.com/en/newsroom/press-releases/2018-12-04-gartner-identifies-the-top-10-trends-impacting-infras. [Data uzyskania dostępu: 13 czerwca 2020].

[3] Datadog, „The State of Serverless,” Datadog, Luty 2020. [Online]. Adres: https://www.datadoghq.com/state-of-serverless/. [Data uzyskania dostępu: 13 czerwca 2020].

[4] J. Harding, „The Origins of Serverless,” 6 Czerwiec 2018. [Online]. Adres: https://dashbird.io/blog/origin-of-serverless/. [Data uzyskania dostępu: 15 czerwca 2020].

[5] Amazon Web Services, "Amazon Simple Storage Service - Developer Guide," [Online]. Adres: https://docs.aws.amazon.com/AmazonS3/latest/dev/NotificationHowTo.html. [Data uzyskania dostępu: 17 sierpnia 2020].

[6] Amazon Web Services, „AWS Step Functions - Developer Guide,” [Online]. Adres: https://docs.aws.amazon.com/step-functions/latest/dg/concepts-states.html. [Data uzyskania dostępu: 17 sierpnia 2020].

[7] Serverless, Inc., „The Serverless Application Framework,” [Online]. Adres: https://www.serverless.com.

[8] Internet Engineering Task Force, „Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content,” Czerwiec 2014. [Online]. Adres: https://www.ietf.org/rfc/rfc7231.txt. [Data uzyskania dostępu: 30 sierpnia 2020].