



POLITECHNIKA POZNAŃSKA

WYDZIAŁ INFORMATYKI I TELEKOMUNIKACJI
Instytut Informatyki

Praca dyplomowa magisterska

**MECHANIZM ZARZĄDZANIA KONTENERAMI W
PRZETWARZANIU BEZSERWEROWYM NA KRAWĘDZI**

Piotr Mazurek, 136767

Promotor
dr hab. inż. Anna Kobusińska, prof. PP

POZNAŃ 2022

Spis treści

1 Wstęp	1
1.1 Motywacja	1
1.2 Cel i zakres pracy	2
1.3 Struktura pracy	2
2 Podstawy teoretyczne	3
2.1 Konteneryzacja	3
2.1.1 Definicja	3
2.1.2 Różnice między kontenerami i maszynami wirtualnymi	3
2.1.3 Proces tworzenia i wdrażania aplikacji	4
2.1.4 Docker	6
2.1.5 Zalety konteneryzacji	9
2.2 Przetwarzanie bezserwerowe	10
2.2.1 Definicja	10
2.2.2 Modele usług chmurowych	10
2.2.3 Funkcje w przetwarzaniu bezserwerowym	12
2.2.4 Zalety przetwarzania bezserwerowego	13
2.2.5 Limity w przetwarzaniu bezserwerowym	15
2.2.6 Najpopularniejsi dostawcy rozwiązań przetwarzania bezserwerowego	17
AWS Lambda	17
Azure Functions	26
Google Cloud Functions	31
Porównanie cen usług poszczególnych dostawców chmurowych	34
2.3 Przetwarzanie na krawędzi	34
2.3.1 Definicja	34
2.3.2 Architektura	34
2.3.3 Przetwarzanie bezserwerowe na krawędzi	36
2.3.4 Przypadki użycia	36
2.3.5 Zagrożenia	37
3 Architektura zaproponowanego rozwiązania	39
4 Implementacja	41
4.1 Specyfikacja implementacji	41
4.1.1 Wymagania funkcjonalne	41
4.1.2 Wymagania pozafunkcjonalne	42
4.2 Opis implementacji technicznej	42
4.2.1 Wykorzystane technologie	42

4.2.2 Szczegóły implementacyjne	42
5 Przykładowe działanie	54
5.1 Rejestracja klienta	54
5.2 Logowanie klienta	55
5.3 Zmiana ustawień konta	55
5.4 Resetowanie hasła	56
5.5 Tworzenie aplikacji	58
5.6 Dodanie zależności	59
5.7 Tworzenie funkcji	59
5.8 Testowanie funkcji	60
5.9 Tworzenie punktu końcowego	62
5.10 Uruchomienie aplikacji	62
5.11 Żądanie wykonania funkcji uruchomionej aplikacji	63
5.12 Wykorzystanie rezultatów przetwarzania na krawędzi	64
6 Testy i porównanie z innymi dostępnymi rozwiązaniami	67
6.1 Testy poprawności działania aplikacji klienckiej	67
6.2 Testy czasu trwania przetwarzania aplikacji klienckiej	73
6.3 Porównanie z innymi dostępnymi rozwiązaniami	75
7 Podsumowanie	76
Literatura	77

Rozdział 1

Wstęp

1.1 Motywacja

W ostatnich latach, popularnym rozwiązaniem wykorzystywanym do tworzenia nowoczesnych aplikacji uruchamianych w środowisku chmurowym stały się kontenery. Mechanizm kontenerów pozwala programistom na tworzenie i wdrażanie aplikacji w szybki i bezpieczny sposób, a także charakteryzuje się wysoką przenośnością i minimalizowaniem wykorzystywanych zasobów [ibm22, BCC⁺17]. Zgodnie z raportem przeprowadzonym w 2020 roku przez *Forrester Consulting*, 86% spośród przodujących firm informatycznych w Stanach Zjednoczonych traktuje jako priorytet wykorzystanie kontenerów dla większej liczby aplikacji, a 81% z nich pragnie w większym stopniu wykorzystać usługi dostarczane przez platformy chmurowe [cap22].

Nowym i atrakcyjnym paradygmatem tworzenia aplikacji chmurowych wykorzystującym kontenery staje się przetwarzanie bezserwerowe (ang. serverless computing) [BCC⁺17]. Wzrost popularności przetwarzania bezserwerowego obrazuje dynamicznie rosnąca liczba wyszukań terminu "serverless" w wyszukiwarce Google, co pokazuje usługa *Google Trends*, umożliwiająca wgląd w informacje i statystyki na temat trendów tej wyszukiwarki internetowej oraz wzrost liczby artykułów, spotkań branżowych, czy wykładów na temat tego paradygmatu. Tworzenie aplikacji opartych na przetwarzaniu bezserwerowym niesie ze sobą wiele korzyści, takich jak obniżenie kosztów wdrażania aplikacji w środowisku chmurowym poprzez dokonywanie płatności jedynie za czas wykonywania funkcji aplikacji, zamiast opłat za alokowanie zasobów w chmurze. Ponadto, w podejściu tym dostawca chmurowy jest odpowiedzialny za zarządzanie infrastrukturą, monitoring, czy zapewnienie wysokiej skalowalności aplikacji, a programista odpowiada jedynie za tworzenie kodu aplikacyjnego [BCC⁺17].

Dostawcy chmurowi pracują nad wykorzystaniem przetwarzania bezserwerowego w przetwarzaniu na krawędzi (ang. edge computing), które ma na celu wykonywanie obliczeń i przetwarzanie danych jak najbliżej ich źródła. Zaletą wykorzystania przetwarzania bezserwerowego w przetwarzaniu na krawędzi jest brak konieczności obciążania klienta opłatą za czas pracy serwerów, w którym nie będą pojawiały się nowe dane do przetworzenia [ATC⁺21]. Przykładem serwisu oferującego przetwarzanie bezserwerowe na krawędzi jest serwis *Lambda@Edge*, będący rozszerzeniem serwisu AWS Lambda, oferowanego przez firmę AWS. Uruchamia on funkcje Lambda na serwerach zlokalizowanych najbliżej użytkownika, a więc na krawędzi, co redukuje opóźnienia w dostarczaniu odpowiedzi klientom [JTA21].

Urządzenia wykorzystywane do przetwarzania na krawędzi, takie jak na przykład telefony komórkowe, posiadają coraz większe zasoby obliczeniowe i mogłyby zostać wykorzystane do wykonania części obliczeń w bezserwerowym przetwarzaniu na krawędzi. Brakuje jednak na rynku

rozwiązań, oferującego klientowi możliwość wykonania części obliczeń po jego stronie i wykorzystania ich rezultatów przy dalszym przetwarzaniu w chmurze.

1.2 Cel i zakres pracy

Celem niniejszej pracy jest implementacja mechanizmu zarządzania kontenerami w przetwarzaniu bezserwerowym na krawędzi. Zaimplementowane rozwiązanie oferuje klientowi możliwość zarówno wykonania wszystkich żądanych obliczeń za pomocą zdalnego wywołania funkcji uruchamianych w kontenerach w chmurze, jak i możliwość wykonania całości lub części obliczeń przez klienta poprzez wysłanie mu kodów funkcji. Klient ma możliwość umieszczenia rezultatów swoich obliczeń w zapytaniu przesyłanym do chmury obliczeniowej, mającym na celu wykonanie pozostałoego przetwarzania. W takim przypadku chmura obliczeniowa nie wykona samodzielnie obliczeń wykonanych już przez klienta, a wykorzysta jego rezultaty w celu wykonania pozostałych koniecznych obliczeń. Ponadto, funkcje oznaczone jako idempotentne, po zakończeniu przetwarzania zwrócią w ciele odpowiedzi HTTP zawartość pamięci podręcznej zawierającej wyniki wykonania takich funkcji dla konkretnych argumentów, z którymi zostały wykonane. Klient podając zawartość pamięci podręcznej w zapytaniu HTTP przy kolejnym wykonaniu tej samej idempotentnej funkcji z tymi samymi argumentami otrzyma natychmiast wynik, ponieważ jest on zawarty w pamięci podręcznej. Ponadto, klient w celu implementacji funkcji może skorzystać z zewnętrznych bibliotek, definiując ich nazwy oraz wersje w pliku z zależnościami.

1.3 Struktura pracy

Niniejsza praca składa się z siedmiu rozdziałów. Rozdział drugi omawia podstawy teoretyczne konteneryzacji, przetwarzania bezserwerowego oraz przetwarzania na krawędzi. Rozdział 3 przedstawia architekturę zaproponowanego rozwiązania. Z kolei rozdział 4 opisuje szczegóły implementacyjne zaproponowanego rozwiązania. Rozdział ten przedstawia zarówno informacje związane ze specyfikacją tworzonego rozwiązania, jak również przedstawia opis szczegółów technicznych i wykorzystanych technologii. Rozdział 5 obrazuje przykładowe działanie zaproponowanego rozwiązania. Rozdział 6 poświęcony został przeprowadzonym testom, jak również porównaniu zaproponowanego rozwiązania z innymi dostępnymi rozwiązaniami. Rozdział 7 podsumowuje niniejszą pracę.

Rozdział 2

Podstawy teoretyczne

2.1 Konteneryzacja

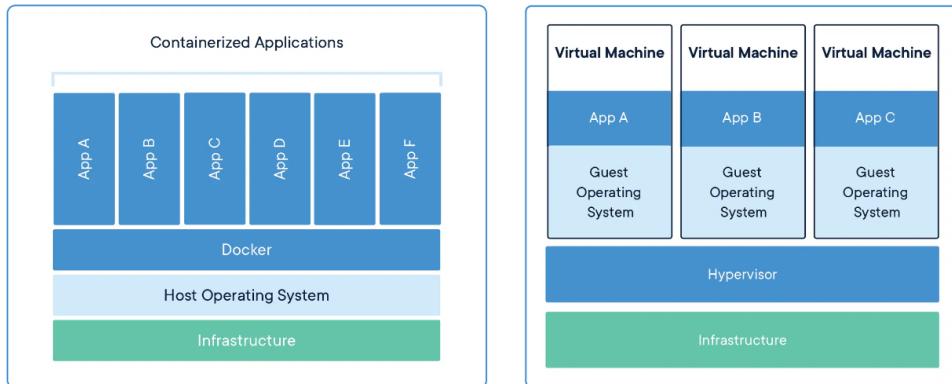
2.1.1 Definicja

Konteneryzacja jest techniką polegającą na tworzeniu wykonywalnej powłoki zawierającej kod wraz z potrzebnymi bibliotekami systemowymi oraz zależnościami, uruchomiony w ramach pojedynczego Unixowego procesu. Tak utworzona wykonywalna powłoka nosi nazwę kontenera [KM18, ibm22]. Kontenery charakteryzują się wysoką przenośnością i oszczędnością względem zasobów, dzięki czemu stały się jednostką obliczeniową nowoczesnych aplikacji uruchamianych w chmurze. Konteneryzacja pozwala programistom na tworzenie i wdrażanie aplikacji w szybki i bezpieczny sposób. Przed powstaniem mechanizmu konteneryzacji kod był pisany w specyficznym środowisku obliczeniowym. W przypadku przenoszenia tego kodu do innego środowiska często były generowane błędy, których przyczyną był brak spójności tych środowisk. Przykładem takiej tranzycji kodu z jednego środowiska do innego jest jego transfer z komputera stacjonarnego do maszyny wirtualnej lub z systemu Linux do Windows. Konteneryzacja eliminuje problem manualnego konfigurowania środowiska, instalacji bibliotek i oprogramowania w systemie operacyjnym, w celu poprawnego uruchomienia kodu aplikacyjnego. Kontener jest bytem niezależnym od systemu operacyjnego, dzięki czemu jest przenośny i możliwy do uruchomienia w taki sam sposób, niezależnie od systemu operacyjnego czy środowiska chmurowego, w którym jest uruchamiany [ibm22]. Standardem mechanizmu konteneryzacji jest *Docker* [ibm22] — narzędzie pozwalające na prostą enkapsulację procesu tworzenia przenośnej aplikacji i wdrażania jej na dużą skalę w dowolnym środowisku [KM18]. Kontenery są często określane jako "lekkie", ponieważ współdzielą jądro i zasoby z systemem operacyjnym na którym są uruchomione i nie wymagają tworzenia systemu operacyjnego dla każdej aplikacji, co jest konieczne w przypadku maszyn wirtualnych [ibm22]. Ponadto kontenery są efemeryczne — mogą zostać uruchamiane i zatrzymywane znacznie częściej niż maszyny wirtualne [KM18]. Są one tworzone na podstawie obrazów — pakietów oprogramowania potrzebnych aplikacji do poprawnego uruchomienia w kontenerze. Pakiety te zawierają kod aplikacji, środowisko uruchomieniowe (ang. runtime), narzędzia, biblioteki oraz konfiguracje systemowe [doc22]. Docker umożliwia programistom tworzenie własnych obrazów i umieszczanie ich w repozytoriach, z których można je pobrać w celu uruchomienia kontenerów [KM18].

2.1.2 Różnice między kontenerami i maszynami wirtualnymi

Zarówno kontenery jak i maszyny wirtualne oferują podobne korzyści wynikające z izolacji zasobów, jednak funkcjonują w inny sposób, ponieważ kontenery wirtualizują system opera-

cyjny, a maszyny wirtualne sprzęt komputerowy. Tym samym, kontenery są bardziej przenośne i wydajne[doc22]. Porównanie zasady działania kontenerów i maszyn wirtualnych przedstawiono na poniższym Rysunku 2.1.

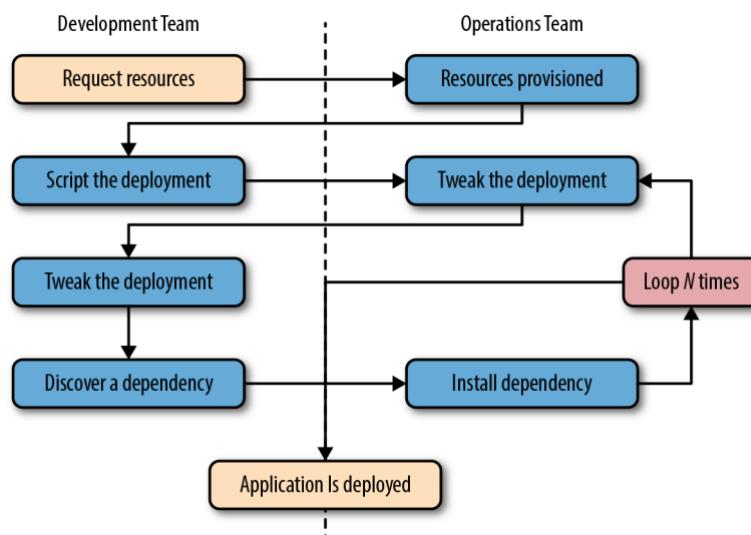


RYSUNEK 2.1: Porównanie zasady działania kontenerów i maszyn wirtualnych [doc22]

Maszyny wirtualne są abstrakcją sprzętu komputerowego, zamieniając jeden serwer w wiele. Hypervisor pozwala uruchomić wiele maszyn wirtualnych na pojedynczym serwerze. Każda maszyna wirtualna zawiera pełną kopię systemu operacyjnego wraz z aplikacjami oraz koniecznymi bibliotekami — może ona osiągać rozmiar dziesiątek Gigabajtów. Ponadto maszyny wirtualne często cechują się powolnym startem systemu (ang. slow boot). Z kolei kontenery są abstrakcją w warstwie aplikacyjnej, udostępniającą razem kod aplikacyjny i konieczne zależności. Wiele kontenerów może zostać uruchomionych na jednym serwerze współdzieląc jądro systemu i jednocześnie zapewniając, że każdy kontener jest od siebie odizolowany. Kontenery zajmują mniej przestrzeni dyskowej niż maszyny wirtualne — obrazy kontenerów zazwyczaj osiągają rozmiary dziesiątek Megabajtów, dzięki czemu mogą one uruchomić więcej aplikacji na pojedynczym serwerze [doc22].

2.1.3 Proces tworzenia i wdrażania aplikacji

Tradycyjny proces tworzenia i wdrażania aplikacji został przedstawiony na poniższym Rysunku 2.2.

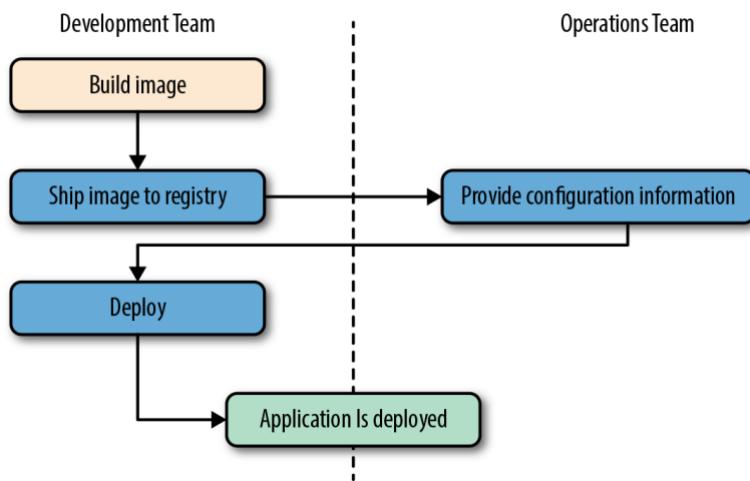


RYSUNEK 2.2: Tradycyjny proces tworzenia i wdrażania aplikacji [KM18]

Składa się on z następujących kroków:

1. Programiści (ang. Development Team) rozpoczynają proces tworzenia aplikacji od zgłoszenia zapotrzebowania na potrzebne dla aplikacji zasoby zespołowi DevOps (ang. Operations Team).
2. Zasoby zostają dostarczone i przekazane programistom.
3. Programiści tworzą skrypty i narzędzia służące wdrożeniu aplikacji.
4. Zespoły programistów oraz DevOps iteracyjnie nanoszą poprawki usprawniające proces wdrożenia aplikacji.
5. Programiści odkrywają kolejne zależności (np. zewnętrzne biblioteki), których potrzebowań będzie aplikacja lub potrzebę przydzielenia jej dodatkowych zasobów
6. Zespół DevOps instaluje potrzebne zależności oraz przydziela aplikacji dodatkowe zasoby
7. Kroki 5. oraz 6. powtarzane są n razy.
8. Aplikacja zostaje wdrożona [KM18].

Tradycyjne wdrażanie nowych złożonych aplikacji w środowisku produkcyjnym jest procesem czasochłonnym — może trwać kilka dni do tygodnia. Proces ten wymaga złożonej komunikacji różnych zespołów oraz uzależnienia ich prac od siebie nawzajem. Konteneryzacja w znacznym stopniu upraszcza ten proces, ponieważ programiści mogą bez konieczności komunikacji z zespołem DevOps tworzyć i kompilować obraz kontenera instalując w nim wszystkie zależności i konfiguracje, które są potrzebne do uruchomienia aplikacji [KM18]. Proces tworzenia i wdrażania aplikacji z wykorzystaniem konteneryzacji został przedstawiony na Rysunku 2.3.



RYSUNEK 2.3: Proces tworzenia i wdrażania aplikacji z wykorzystaniem konteneryzacji [KM18]

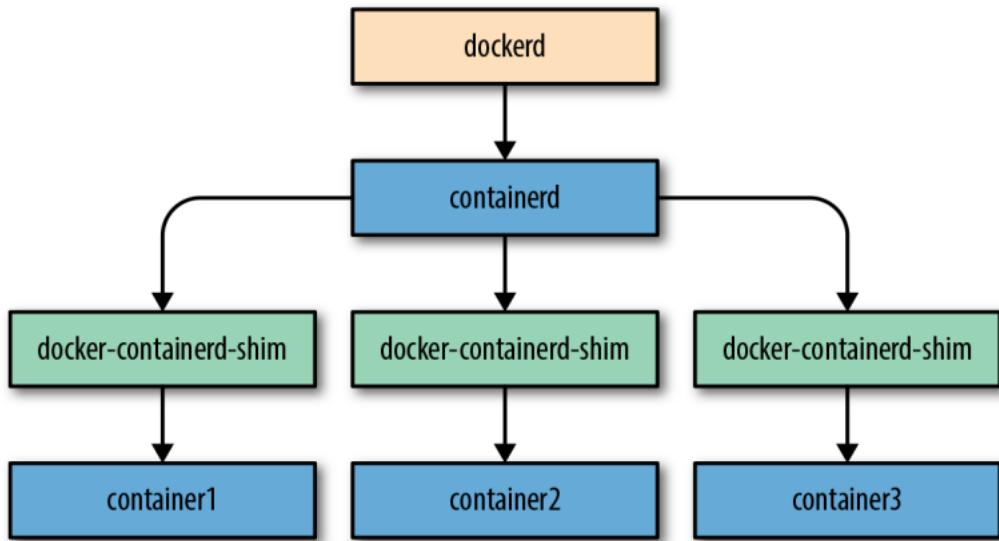
Korzystając z konteneryzacji, kroki wdrażania aplikacji wyglądają następująco:

1. Programiści przygotowują obraz kontenera i przekazują go do rejestru.
2. Zespół DevOps dostarcza szczegóły konfiguracyjne dotyczące kontenera i przygotowuje zasoby.
3. Programiści uruchamiają proces wdrożenia.
4. Aplikacja zostaje wdrożona [KM18].

2.1.4 Docker

Docker jest najbardziej znanym i najczęściej używanym narzędziem konteneryzacji [ibm22]. Wykorzystuje on jądro systemu Linux oraz jego własności takie jak *Cgroups* w celu przydzielenia limitów zasobów dostępnych dla procesów i ich procesów potomnych [KM18] oraz *namespaces* w celu segregacji procesów, tak aby były uruchamiane niezależnie od siebie. Docker dostarcza model wdrażania aplikacji oparty na obrazach kontenerów. Pozwala to na współdzielenie aplikacji ze wszystkimi jej zależnościami pomiędzy wieloma środowiskami [red22a].

Architektura



RYSUNEK 2.4: Architektura Docker [KM18]

Architektura Docker przedstawiona na Rysunku 2.4 jest oparta na modelu klient/serwer — klient za pomocą polecenia *docker* komunikuje się z serwerem przy pomocy Docker API. API to jest fasadą pozwalającą na komunikację z pięcioma komponentami, jakimi są *dockerd*, *containerd*, *runc*, *docker-containerd-shim* oraz *docker-proxy*. Komponent *dockerd* jest procesem wykorzystywanym do uruchamiania procesu serwera udostępniającego Docker API dostępnego dla klienta za pomocą polecenia *docker*, odpowiada za budowę obrazów kontenerów, zarządzanie siecią, voluminami, logowanie, raportowanie statystyk itd. Zlecenie uruchomienia kontenera jest przekazywane z *dockerd* do *containerd*, który odpowiada za jegoinicjalizację oraz zarządza jego cyklem życia. Przed decyzją o utworzeniu tego komponentu wszystko było obsługiwane w procesie *dockerd*. Rozwiązanie to miało pewne wady:

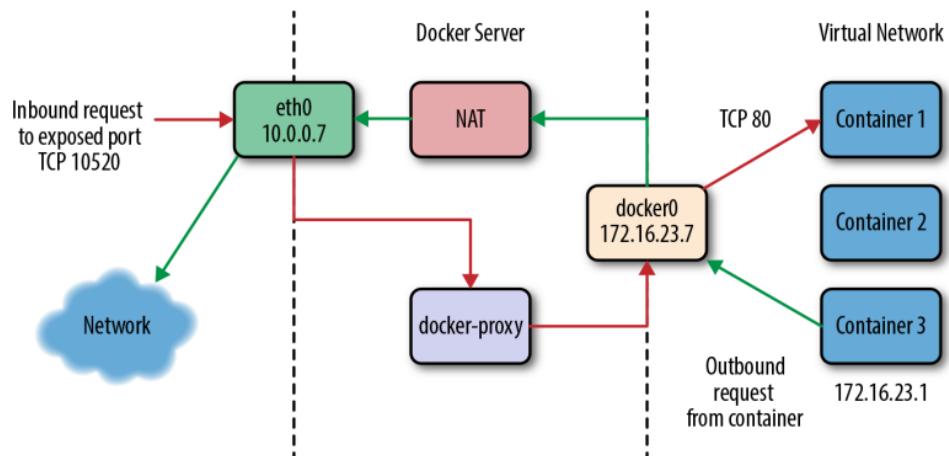
- proces *dockerd* miał do wykonania dużą liczbę zadań
- *dockerd* musiał nadzorować cykl życia kontenerów, zatem jego restart lub aktualizacja powodowała wyłączenie wszystkich uruchomionych kontenerów

Kolejną zaletą korzystania z *containerd* jest możliwość zintegrowania platform takich jak Kubernetes bezpośrednio z tym komponentem, bez konieczności korzystania z Docker API. Komponent *docker-containerd-shim* obsługuje deskryptory plików przekazane do kontenera takie jak *stdin*, *stdout* oraz zwraca ich kod zakończenia. Na każdy kontener przypada jeden komponent

docker-containerd-shim. Z kolei komponent runc jest to narzędzie wiersza poleceń (ang. Command Line Interface (CLI)) uruchamiane przez containerd, tworzące kontener, uruchamiające go, zbierające statystyki i raportujące zdarzenia cyklu życia kontenera. Komunikacja pomiędzy dockerd, a containerd odbywa się za pomocą gniazda sieciowego z wykorzystaniem API gRPC, gdzie dockerd jest klientem zlecającym zadania, a containerd serwerem [KM18].

Gdy klient za pomocą Docker API (jakim jest polecenie `docker`) uruchamia kontener, dockerd po otrzymaniu polecenia upewnia się, że obraz jest dostępny na dysku. Jeśli nie, zostanie on pobrany ze zdalnego repozytorium wskazanego w nazwie obrazu. Następnie dockerd konfiguruje kontener m. in. ustawiając sterowniki logowania, woluminów oraz systemu polików. Następnie za pomocą API gRPC zleca komponentowi containerd uruchomienie kontenera. Containerd pobiera obraz kontenera oraz wykorzystuje konfigurację otrzymaną od dockerd w celu wygenerowania pakietu *OCI* (*Open Container Initiative*). Narzędzie runc uruchamia ten pakiet, wraz z uruchomieniem docker-containerd-shim, co skutkuje wystartowaniem kontenera. Po wystartowaniu kontenera, runc kończy swoje zadanie, a jego procesy potomne są dziedziczone przez docker-containerd-shim, który staje się procesem-rodzicem procesu uruchomionego w kontenerze [KM18].

Ostatnim komponentem wchodząącym w skład architektury Docker jest *docker-proxy*. Jest on procesem uruchomionym na serwerze na porcie 10520 służącym do przekazywania żądania klienta do konkretnego kontenera. Przykład przekazania żądania do Kontenera 1 przedstawiono na poniższym Rysunku 2.5.



RYSUNEK 2.5: Sieć w Docker [KM18]

Na Rysunku 2.5 przedstawiono domyślną konfigurację sieci Docker, jaką jest tryb mostu (ang. bridge mode). Jest to tryb, w którym każdy kontener przyjmuje zachowanie komputera w prywatnej sieci lokalnej. Most sieciowy to urządzenie przekazujące ruch sieciowy między stronami połączenia. Serwer Docker działa jako wirtualny most sieciowy, z kolei kontenery są połączonymi do niego klientami. Każdy kontener posiada własny interfejs sieciowy podłączony do mostu sieciowego Docker, posiadającego własny adres IP przypisany do wirtualnego interfejsu. Docker tworzy prywatną podsieć korzystając z wolnego (zgodnie z *RFC 1918* opisującym alokacje adresów w sieciach prywatnych) bloku podsieci prywatnej. Docker sprawdza, który blok tej podsieci jest wolny, po czym przypisuje go do swojej sieci wirtualnej, tworząc w ten sposób połączenie mostem z siecią lokalną hosta za pomocą dostępnego na serwerze interfejsu *docker0*. Dzięki temu wszystkie kontenery znajdują się w jednej sieci i mogą się ze sobą komunikować. Pozwala to na połączenie portów serwera lub grupy portów z portami kontenera w taki sposób, by za pomocą tych portów serwera kontener był dostępny z zewnątrz. Jednak, aby dostać się do kontenera lub wysłać żądanie

na zewnątrz, zapytanie musi być przesłane przez interfejs docker0. Za zadanie przekazania zapytania klienta do docker0, tak aby następnie trafiło ono do odpowiedniego kontenera służy proces docker-proxy [KM18].

Na Rysunku 2.5 pokazano przykład obsługi żądania od klienta, w przypadku, gdy klient chce połączyć się z serwerem nginx uruchomionym w kontenerze 1 na porcie TCP 80. Jego żądanie najpierw trafi do interfejsu sieciowego serwera, na którym uruchomiony jest serwer Docker oraz kontenery (na Rysunku 2.5 jest to interfejs eth0 o adresie 10.0.0.7). Następnie żądanie zostanie przekazane na port TCP 10520 do procesu docker-proxy, który przekaże je na adres odpowiedniego kontenera i portu w prywatnej sieci Docker. Odpowiedź serwera nginx znajdującego się w kontenerze 1 zostanie przesłana tą samą drogą [KM18].

Z kolei obsługa żądania wychodzącego z kontenera chcącego nawiązać połączenie e zdalnym serwerem w internecie odbywa się bez pośrednictwa komponentu docker-proxy. Przykład takiego żądania wychodzącego z kontenera 3 został pokazany na Rysunku 2.5. Kontener posiada adres IP prywatnej sieci interfejsu docker0. W przypadku przedstawionym na Rysunku 2.5 kontener 3 posiada adres 172.16.23.1, a jego domyślną trasą jest 172.16.23.7, czyli adres w interfejsie docker0. Następnie żądanie z prywatnej sieci wirtualnej docker jest przekazywane przez warstwę *NAT (Network Address Translation)* i umieszczane w sieci zewnętrznej serwera z wykorzystaniem interfejsu eth0. Następnie serwer przesyła dalej żądanie do zdalnego serwera w internecie za pomocą swojego adresu publicznego 10.0.0.7, a odpowiedź wraca tą samą drogą [KM18].

Warstwy obrazu kontenera

Kontenery Docker są stworzone z ułożonych w stosie warstw systemu plików — każda z nich jest identyfikowana unikalnym skrótem (ang. hash). Robiąc zmiany w obrazie programista musi przebudowywać jedynie te warstwy, od których zaczyna się zmiana. Pozwala to na oszczędzenie czasu i przepustowości łącza sieciowego, ponieważ obrazy kontenerów są dostarczane do serwera w postaci warstw i programista nie ma konieczności dostarczenia tych warstw, które już zostały zapisane na serwerze i nie uległy zmianie [KM18].

Budowa obrazu kontenera

Docker umożliwia programistę budowanie obrazu kontenera poprzez stworzenie pliku o nazwie *Dockerfile*, zdefiniowaniu w nim warstw obrazu, a następnie zbudowaniu go za pomocą polecenia *docker build* udostępnionego przez Docker API, którego argumentem jest stworzony plik Dockerfile. Każde polecenie w pliku Dockerfile generuje nową warstwę w obrazie [KM18]. Na Listingu 1 przedstawiono przykładową implementację pliku Dockerfile uruchamiającą aplikację w środowisku uruchomieniowym (ang. runtime) Node.js.

LISTING 1: Przykładowy plik Dockerfile

```

1  FROM node:16
2  WORKDIR /usr/src/app
3  COPY . .
4  RUN npm install
5  EXPOSE 8080
6  CMD [ "node", "app.js" ]

```

Plik ten zawiera sześć poleceń, a więc zbudowany na jego podstawie obraz kontenera będzie posiadał 6 warstw. Kod ten importuje środowisko uruchomieniowe Node.js, w którym będzie uruchomiona aplikacja (linia 1), następnie kopiuje pliki z lokalnego systemu plików do kontenera

(linie 2-3), instaluje konieczne dla aplikacji biblioteki (linia 4), a na koniec uruchamia aplikację udostępniając interakcję z nią za pomocą portu 8080 (linie 5-6).

Wersjonowanie obrazu kontenera

Docker posiada wbudowany mechanizm umożliwiający znakowanie (ang. tagging) obrazów podczas ich budowy. Dzięki temu programista może utworzyć i przechować wiele wersji aplikacji na serwerze — wystarczy nadać im różne oznaczenia podczas udostępniania. Oznaczenie aplikacji jako *latest* oznaczy, że zbudowana wersja obrazu kontenera jest najnowszą [KM18].

Skalowanie

Dzięki odseparowaniu aplikacji znajdującej się w kontenerze od środowiska, na którym kontener został uruchomiony, aplikacja nie jest zależna od żadnego serwera, czy też środowiska. Pozwala to na pionowe skalowanie aplikacji na własnych serwerach lub korzystając z różnych platform chmurowych takich jak AWS, czy Google Cloud Platform. Największy dostawcy chmurowi opracowali rozwiązania posiadające natywne wsparcie kontenerów Docker. Do najpopularniejszych implementacji korzystających z kontenerów Docker w chmurze publicznej należą:

- Amazon Elastic Container Service
- Google Kubernetes Engine
- Azure Container Service
- Red Hat OpenShift

Jednym z najpopularniejszych rozwiązań uruchamiania kontenerów Docker na dużą skalę jest *Kubernetes* [KM18]. Jest on narzędziem typu open source pozwalającym na automatyzowanie procesów wdrażania, zarządzania oraz skalowania skonteneryzowanych aplikacji [red22b].

2.1.5 Zalety konteneryzacji

- Przenośność — kontener tworzy przenośny pakiet oprogramowania, który jest niezależny od systemu operacyjnego, na którym jest uruchomiony, dzięki czemu jest on przenośny i jednorodny niezależnie od systemu operacyjnego i platformy dostawcy chmurowego, na której został uruchomiony [ibm22].
- Szybkość i koszt — kontenery, dzięki współdzieleniu jądra operacyjnego i zasobów z systemem operacyjnym, są bardziej wydajne i redukują liczbę koniecznych zasobów do ich uruchomienia. Są szybkie w uruchomieniu, ponieważ w przeciwieństwie do maszyn wirtualnych nie posiadają systemu operacyjnego, który musi zostać najpierw uruchomiony [ibm22].
- Izolacja w przypadku błędu — każda aplikacja uruchomiona w kontenerze jest odizolowana i niezależna od pozostałych. Awaria jednego kontenera nie wpływa na żywotność pozostałych kontenerów. Programiści mają możliwość naprawy błędów związanych z działaniem jednego kontenera bez konieczności zatrzymywania pracy pozostałych.
- Ułatwienie w zarządzaniu — przy pomocy platformy orkiestracji kontenerów np. narzędzia *Kubernetes*, programista ma możliwość skalowania, wersjonowania oraz monitoringu aplikacji [ibm22].

- Bezpieczeństwo — izolacja aplikacji w kontenerach zapobiega atakom na aplikację z innych kontenerów lub z systemu, na którym kontener jest uruchomiony. Programista może zdefiniować uprawnienia automatycznie blokujące niechcianą zewnętrzną ingerencję w kontenerze [ibm22].
- Niezmienność infrastruktury — dzięki kontenerom programiści mogą tworzyć serwery, które praktycznie nie wymagają zarządzania konfiguracją czy bibliotekami — zarządzanie aplikacjami polega na uruchamianiu kolejnych kontenerów na serwerze. W przypadku konieczności przeprowadzenia aktualizacji na serwerze np. serwera Docker lub jądra Linux, programista może uruchomić nowy serwer z zainstalowanymi aktualizacjami, dokonać migracji do niego kontenerów uruchomionych na starym serwerze wymagającym aktualizacji, a następnie zaktualizować bądź wyłączyć stary serwer.

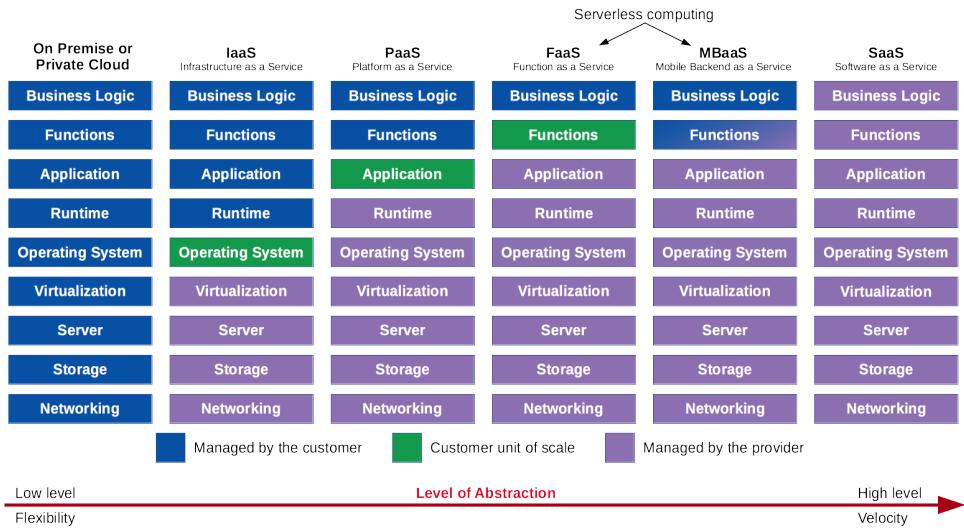
2.2 Przetwarzanie bezserwerowe

2.2.1 Definicja

Przetwarzanie bezserwerowe (ang. serverless) jest modelem przetwarzania, w którym kod programisty jest uruchamiany jako serwis, bez konieczności tworzenia oraz utrzymywania infrastruktury, na której uruchamiane są aplikacje tworzone przez programistę. Nie znaczy to, że model ten nie wymaga serwerów [Sti18]. Aplikacje programisty są wdrażane i uruchamiane na infrastrukturze chmurowej na zasadzie płatienia za korzystanie z nich, bez konieczności wynajmowania, czy kupowania serwerów. Zamiast programisty, to dostawca chmurowy jest odpowiedzialny za uruchamianie aplikacji, skalowanie ich, równoważenie obciążenia poszczególnych ich instancji, zapewnienie, aby miały one wystarczające zasoby oraz za ich monitoring. Pierwotnie termin serverless był definiowany jako *Backend as a Service (BaaS)*, ponieważ reprezentował aplikacje, które były częściowo lub całkowicie zależne od zewnętrznych serwisów [SS19]. Takie podejście pozwalało na opieranie się na logice biznesowej, która została już wcześniej zaimplementowana. Przykładem jest uwierzytelnienie, gdzie dotąd każda firma musiała zaimplementować swój kod odpowiedzialny za uwierzytelnienie, mimo że kody różnych firm były bardzo podobne z punktu widzenia logiki biznesowej. Doprowadziło to do powstania zewnętrznych serwisów takich jak *Auth0*, czy *Amazon Cognito*. Pozwoliły one zarówno aplikacjom webowym jak i mobilnym na posiadanie gotowego rozwiązania odpowiedzialnego za uwierzytelnianie, bez konieczności pisania przez programistów logiki z nim związanej [RC17]. Obecnie termin serverless jest utożsamiany z *Function as a Service (FaaS)*, ponieważ dostawcy przetwarzania bezserwerowego traktują aplikacje jako zbiór funkcji, wywołując je tylko, gdy nastąpi żądanie ich wykonania [SS19].

2.2.2 Modele usług chmurowych

Samo określenie "As a service" oznacza usługę chmury obliczeniowej dostarczaną przez zewnętrznego dostawcę w taki sposób, aby ułatwić programistę skupienie się na tych warstwach infrastruktury, które są dla niego najważniejsze. Każdy kolejny z modeli usług chmurowych przedstawionych na Rysunku 2.6 zmienia zakres infrastruktury za jaki musi być odpowiedzialny programista [red20].



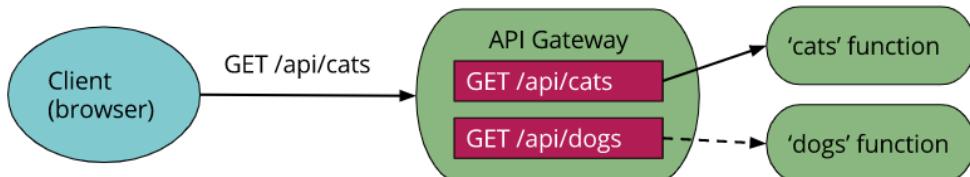
RYSUNEK 2.6: Modele usług chmurowych [Clo22b]

Rysunek 2.6 obrazuje odpowiedzialność użytkownika na każdym z poziomów przetwarzania chmurowego. Model *Traditional IT* nazywany też *On-premise* reprezentuje największy poziom odpowiedzialności za infrastrukturę [red20], w którym programista odpowiada zarówno za zarządzanie serwerami, miejscem składowania danych, konfiguracją sieci, jak i oprogramowanie. Model *Infrastructure as a Service (IaaS)* jest pierwszym krokiem odejścia od infrastruktur on-premise, w którym dostawca chmurowy przejmuje odpowiedzialność za wirtualizację, serwery, konfigurację sieciową i miejsce składowania danych. Programista nie musi uiszczać opłat za wykorzystanie wspomnianych wyżej elementów, ponieważ dostarczy je dostawca chmurowy. Na tym poziomie przetwarzania chmurowego programista ma dostęp do infrastruktury za pomocą *application programming interface (API)* lub aplikacji internetowej [red20]. Dodatkowo, dostawca chmurowy oferuje programistom zaporę ogniomową (ang. firewall), pulę adresów IP do wykorzystania na każdym z serwerów oraz system równoważenia obciążenia (ang. load balancer). Przykładem realizacji takiego modelu jest instancja *Amazon Elastic Compute Cloud (EC2)* [Sti18]. *Platform as a service (PaaS)* jest kolejnym krokiem ku zwiększeniu integracji z środowiskiem chmurowym polegającym na dostarczeniu programistom systemu operacyjnego, bazy danych oraz serwera webowego. Do odpowiedzialności programisty należy tylko budowanie i uruchamianie aplikacji. Przykładem realizacji tego modelu jest AWS Elastic Beanstalk, Azure Web Apps, czy Heroku [Sti18]. *Mobile Backend as a Service (MBaaS)* jest pojęciem używanym zamiennie z BaaS powstały ze względu na częste wykorzystywanie serwisów typu BaaS w aplikacjach mobilnych [RC17]. W podejściu *Software as a Service (SaaS)* to dostawca chmurowy dostarcza gotowe aplikacje [Sti18]. Odpowiada on za ich aktualizację, naprawę błędów oraz udostępnia korzystanie z nich za pomocą API lub aplikacji webowej. Przykładem implementacji SaaS jest Dropbox [red20]. Z kolei model FaaS polega na całkowitym przekazaniu dostawcy chmurowemu kontroli nad kontenerami, w których uruchamiane są funkcje w celu obsługi żądań. Taka architektura zapobiega konieczności posiadania nieustannie uruchomionych systemów i pozwala na wykonywanie obliczeń opartych na zdarzeniach. Oparty na nich kod jest tworzony przez programistę i uruchamiany w bezstanowych, efemerycznych kontenerach [Sti18], czyli takich, w których dostawca chmurowy usuwa je bądź tworzy nowe dynamicznie w zależności od potrzeb wynikających z poziomu obciążenia aplikacji [Rob18].

2.2.3 Funkcje w przetwarzaniu bezserwerowym

Dopóki nie zajdzie żądanie wywołania aplikacji będącej zbiorem funkcji, żadna jej instancja nie jest uruchomiona, dzięki czemu dostawca chmurowy nie marnuje zasobów serwerowych, czy też energii. Jest on odpowiedzialny za wszystkie detale operacyjne dotyczące aplikacji programisty, takie jak ustalenie lokacji serwera, na którym ma zostać uruchomiona aplikacja, sposobu jej replikacji, czasu uruchomienia nowych kontenerów. Ponadto zarządza on ich liczbą, w celu kontroli zajętości miejsca na nieużywanych serwerach. Kontener, w którym wywoływana jest funkcja aplikacji jest oparty na zdarzeniach i aktywowany tylko, gdy nastąpi zdarzenie [SS19]. Zatrzymanie instancji kontenera w celu zwolnienia zasobów obliczeniowych następuje w przypadku bezczynności tej instancji przez określony okres czasu. Zatem instancje funkcji przetwarzania bezserwerowego nie powinny pozostawać bezczynne [Jak20]. Kontenery pozwalają również funkcjom na emisję zdarzeń. Wyzwalacz (ang. trigger) zdarzenia zazwyczaj jest żądaniem HTTP, załadowaniem pliku do systemu plików, aktualizacją w bazie danych lub zdarzeniem zachodzącym w środowisku Internetu Rzeczy (ang. Internet of Things (IoT)) [SS19]. W zależności od wyzwalacza funkcja może skonsumentować ciało wejściowe zapytania HTTP lub samemu przeczytać jakieś informacje. Funkcja oparta na HTTP może przetworzyć obraz przekazany jako binarny strumień danych, zaaplikować filtr, a następnie zwrócić rezultat lub zapisać go w bazie danych [Jak20]. W podejściu serverless płacimy tylko za czas trwania funkcji [SS19]. Funkcje w przetwarzaniu bezserwerowym można również rozumieć jako bezserwerowe mikroserwisy [Sti18]. Każda funkcja ma jeden określony cel i wykonuje swoje przetwarzanie niezależnie od innych funkcji. Tworzone w ten sposób są one elastyczne i otwarte na możliwe zmiany [Rob18]. Przetwarzanie bezserwerowe jest bezstanowe oraz oparte na zdarzeniach, zatem funkcje również powinny być projektowane w ten sposób [Sti18]. Jakikolwiek stan funkcji, który ma być trwały (ang. persistent) musi być wyodrębniony poza jej instancję. W przypadku potrzeby tworzenia takich funkcji zorientowanych na stan (ang. state – oriented), do jego przechowania korzysta się zazwyczaj z bazy danych. Może być nią baza *Redis* służąca jako pamięć podrzeczna współdzielona pomiędzy aplikacjami, lub system plików np. Amazon S3. Pozwala ona przechowywać stan pomiędzy wieloma zapytaniami, aby dostarczyć argumenty wejściowe konieczne do przetworzenia funkcji [Rob18]. Funkcje powinny zawierać małą ilość kodu, być szybkie i łatwo skalowalne. Przykładowo każda funkcja może reprezentować jedną metodę API, wykonując jedno przetwarzanie [Sti18] — funkcje takie zazwyczaj mają ograniczony maksymalny czas trwania [Jak20].

Wyzwalaczem dla funkcji mających być odpowiedzią na zapytania HTTP klienta jest *API gateway*. Jest to serwer HTTP, w którym ścieżki (ang. routes) oraz punkty końcowe (ang. endpoints) są definiowane za pomocą konfiguracji i każda ścieżka ma powiązaną z nią funkcję obsługującą żądanie HTTP [Rob18].



RYSUNEK 2.7: Przykład działania API gateway [Rob18]

Na Rysunku 2.7 pokazano przykład działania API Gateway — w zależności od tego pod jaki adres wejdzie klient za pomocą przeglądarki, zostanie wykonana odpowiednia funkcja powiązana z tym adresem. Następnie po wykonaniu się, funkcja zwróci rezultat do API gateway, który

przetransformuje go do odpowiedzi HTTP i zwróci ją klientowi [Rob18].

2.2.4 Zalety przetwarzania bezserwerowego

- Krótki czas trwania i koszty tworzenia i wdrażania aplikacji — Jest to możliwe, ponieważ programista jest odpowiedzialny jedynie za tworzenie aplikacji, bez konieczności przygotowania infrastruktury na potrzebę jej uruchomienia. Ponadto, dostawcy chmurowi tacy jak AWS, Google, czy Azure dostarczają gotowe szablony funkcji, które mogą zostać użyte do natychmiastowego stworzenia wykonywalnej funkcji. Z kolei wersjonowanie funkcji ułatwia programistom wdrażanie ich na różnych środowiskach [Sti18]. W przetwarzaniu bezserwerowym wymagana jest mniejsza liczba linii kodu oraz programiści szybciej wdrażają i testują aplikacje. Dzięki korzystaniu zewnętrznych serwisów BaaS programiści nie muszą poświęcać czasu na pisanie złożonej, powtarzalnej logiki biznesowej, np. dzięki AWS Cognito dostają gotowy serwis do uwierzytelnienia, a dzięki serwisowi *Mailgun* mogą pominąć wymagającą pracę nad implementacją procesu wysyłania i otrzymywania emaili. Tworzenie aplikacji jest prostsze dla programistów, ponieważ przekazują jedynie kody źródłowe aplikacji dostawcy chmurowemu — plik zip w przypadku języków javascript, czy python lub plik jar w przypadku języków bazujących na Wirtualnej Maszynie Javy. W związku z brakiem konieczności monitorowania stanu serwerów przez programistów, mogą oni ograniczyć monitoring do zbierania metryk o aplikacji. Dzięki temu aplikacje te mogą dotyczyć czasu wykonania funkcji, metryk dotyczących klienta, zamiast obszarów takich jak przestrzeń dyskowa, czy zużycie procesora [RC17].
- Redukcja kosztów zasobów — firma nie jest zmuszona planować ile serwerów oraz zasobów takich jak RAM, czy CPU powinna przeznaczyć na poprawne działanie aplikacji charakteryzujących się wysoką dostępnością. W modelu serverless nie zajdzie ryzyko zjawiska *over-provisioningu*, czyli przeszacowania liczby koniecznych zasobów do poprawnego działania aplikacji przy dużym obciążeniu. Over-provisioning oznacza konieczność uiszczenia pełnej kwoty za infrastrukturę przygotowaną na maksimum obciążenia aplikacji nawet w okresach czasu, gdy jest ono niskie, a duża liczba możliwych do wykorzystania zasobów pozostaje bezczynna. W przetwarzaniu bezserwerowym firma nie musi planować, alokować i organizować zasobów. Zamiast tego pozwala ona dostawcy chmurowemu oszacować ile zasobów dany system potrzebuje w danym momencie czasu [RC17]. Firma płaci jedynie za czas wykonania funkcji, zatem nie ma ona obowiązku płacenia za uruchomione serwery, ani za stan, w którym aplikacja jest uruchomiona.
- Mniejsze ryzyko — im większa liczba różnorodnych komponentów z których składa się system, tym bardziej jest zarządzać takim systemem oraz więcej problemów może wystąpić, skutkiem czego są błędy w działaniu aplikacji. W przetwarzaniu bezserwerowym odpowiedzialność i ryzyko za utrzymanie systemu spoczywa na dostawcy chmurowym i programiści zarządzają ryzykiem w inny sposób - opierając się na ekspertyzie dostawcy, aby rozwiązać ewentualne problemy zamiast konieczności samodzielnego ich rozwiązywania. Wraz z przejęciem na model przetwarzania bezserwerowego zmniejsza się liczba różnych technologii za jakie bezpośrednio odpowiadają programiści. Im mniej technologii tym większe prawdopodobieństwo, że programista będzie obyty z jedną z nich, której to dotyczy awaria. Na przykład, jeśli firma skorzysta z bezserwerowego serwisu bazy danych, jak *Amazon DynamoDB* nie ma ryzyka, że programista będzie musiał zająć się awariami bazy danych, które zdarzają się relatywnie rzadko, w związku z czym programista często nie ma eksperckiej

wiedzy, żeby szybko zdiagnozować i naprawić problem. Mimo, że awarie DynamoDB czasem się zdarzają, to Amazon ma dedykowane zespoły zajmujące się konkretnymi serwisami, posiadające ekspercką wiedzę, zatem korzystając z takich bezserwerowych serwisów typu BaaS firma redukuje możliwy przestój czasu pracy komponentu swojej aplikacji [RC17].

- Silna skalowalność — w rozwiązaniach przetwarzania bezserwerowego problem skalowalności jest zarządzany przez dostawcę chmurowego. Programista nie jest odpowiedzialny za zapewnienie skalowalnego rozwiązania równoważącego ruch między wieloma instancjami aplikacji [Sti18], ponieważ skalowanie pionowe jest automatyczne i elastyczne — instancje kontenerów są uruchamiane i usuwane przez dostawcę chmurowego, w zależności od poziomu obciążenia aplikacji [Rob18].
- Silna dostępność — oprócz skalowalności, zwiększa się również dostępność, ponieważ kod programisty może zostać uruchomiony na serwerach w wielu regionach świata. Na Rysunku 2.8 przedstawiono rozmieszczenie centrów chmurowych dla 4 największych dostawców chmurowych.



RYSUNEK 2.8: Rozmieszczenie centrów chmurowych na świecie [Sti18]

2.2.5 Limity w przetwarzaniu bezserwerowym

- Zarządzanie stanem — aby bezstanowe funkcje przetwarzania bezserwerowego mogły manipułować stanem muszą komunikować się z serwisami pozwalającymi na przechowywanie stanu. Ta komunikacja może prowadzić do opóźnień oraz skutkuje wysoką złożonością komunikacyjną [RC17].
- Lokalne testowanie — w systemach serwerowych programiści często posiadają lokalne środowisko pozwalające uruchomić lokalnie aplikację w celu przeprowadzenia testów integracyjnych. Aplikacje bezserwerowe mogą się opierać na testach jednostkowych, natomiast przeprowadzenie testów integracyjnych jest dużo bardziej skomplikowane. Ze względu na zarządzanie infrastrukturą przez dostawcę chmurowego ciężko jest lokalnie połączyć komponenty aplikacyjne, tak aby odzworować obsługę błędów, logowanie, wydajność, czy sposób skalowania. Ponadto

aplikacje bezserwerowe składają się z wielu funkcji i serwisów BaaS. Uruchomienie tak wielu komponentów lokalnie oraz zarządzanie nimi w celu przetestowania konkretnego przypadku może być wyzwaniem. W związku z tym zaleca się, aby testy integracyjne przeprowadzać w chmurze [RC17].

- Zdalne testowanie — dostawcy chmurowi umożliwiają zdalne testowanie pojedynczej funkcji, lecz nie oferują rozwiązań zdalnego testowania całej aplikacji. Z tego powodu ciężko jest dokładnie przetestować złożoną aplikację bez założenia osobnego konta u dostawcy chmurowego, które daje pewność braku wpływu testów na środowisko i zasoby produkcyjne — pozwala uniknąć sytuacji, w której limity na zasoby zostaną przekroczone w wyniku testowania [RC17].
- Rozproszony monitoring — dostawcy chmurowi nie dostarczają rozwiązań rozproszonego monitoringu, umożliwiającego weryfikację efektów żądania przetwarzanego przez wiele funkcji [RC17].
- Opóźnienie — w tradycyjnych serwerowych systemach, gdy pojawiają się opóźnienia w komunikacji między komponentami aplikacji, można umieścić je bliżej siebie np. na tym samym serwerze lub nawet przenieść przetwarzanie do jednego procesu. Z kolei w przetwarzaniu bezserwerowym komunikacja między funkcjami odbywa się za pomocą protokołu HTTP, który może być wolniejszy od innych metod komunikacji. Ponadto komunikacja z zewnętrznymi serwisami BaaS może prowadzić do opóźnień w komunikacji [RC17].
- Uzależnienie od dostawcy chmurowego (ang. vendor lock-in) — w aplikacjach serwerowych wszystkie komponenty z jakich składa się aplikacja mogą być pod kontrolą programistów. Przejście w kierunku przetwarzania bezserwerowego wiąże się z utratą kontroli nad infrastrukturą, na której uruchamiane są aplikacje napisane przez programistów. Jedynym aspektem jakim programiści mogą manipulować jeśli chodzi o infrastrukturę są jej parametry konfiguracyjne udostępnione przez dostawcę chmurowego. Ponadto, programiści nie mają wpływu na wydajność platformy bezserwerowej oferowanej przez dostawcę. Przez to identycznie skonfigurowane funkcje mogą różnić się drastycznie wydajnością. Czynnikami wpływającymi na to są priorytety planisty chmurowego (ang. platform scheduler) oraz sposób alokowania zasobów przez dostawcę po otrzymaniu żądania. Jeśli doszłoby do jakiegokolwiek błędu w aplikacji związanego z samą platformą dostawcy, to programiści są uzależnieni od dostawcy chmurowego i nie mają możliwości naprawy tego błędu. Często nie wiadomo, czy i kiedy doszło do błędu platformy, gdyż dostawcy ukrywają przyczyny problemów niepoprawnego działania platformy. Programiści mają jedynie wpływ na naprawę błędu po stronie kodu aplikacyjnego. Firmy korzystające z usług chmurowych są uzależnione od dostawcy również pod względem bezpieczeństwa swoich aplikacji. Niestety nie wszystkie własności bezpieczeństwa serwisów mogą spełnić wymagania firm. Przykładowo, dostęp do serwisu AWS API Gateway zawsze będzie możliwy z jakiegokolwiek adresu IP, więc jeśli wymaganiem aplikacji jest, aby dostęp do niej był możliwy tylko z wybranych adresów, to nie można skorzystać z API Gateway [RC17].
- Limity zasobów — funkcje w przetwarzaniu bezserwerowym są uruchamiane w środowisku z ograniczoną liczbą zasobów takich jak CPU, pamięć, dysk, maksymalny czas wykonania. Na przykład w przypadku AWS Lambda limit maksymalnego czasu wykonania wynosi 5 minut, a maksymalny rozmiar pamięci jaką funkcja może wykorzystać wynosi 1.5GB.

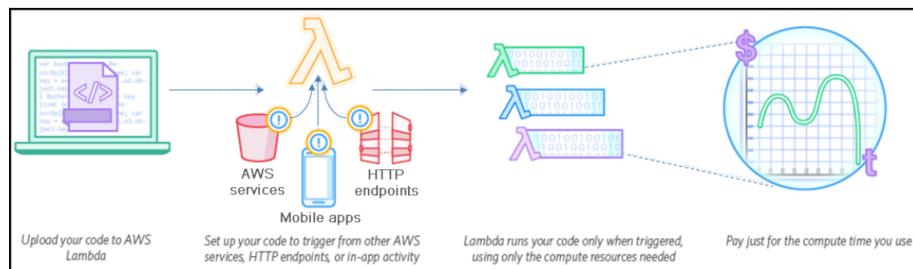
- Zimny start — stworzenie instancji funkcji w kontenerze wymaga określonego czasu. Określa się go zimnym startem (ang. cold start) [Rob18]. Może to prowadzić do opóźnienia odpowiedzi na żądanie klienta [Sti18]. Czas wymagany do startu funkcji może się różnić w zależności od wielu czynników i wahając się w granicach kilku milisekund do kilku sekund. Przykładem takiego czynnika może być język, z którego korzysta programista, liczba bibliotek, które wykorzystuje, liczba linii kodu funkcji, sposób jej implementacji czy konfiguracji po stronie dostawcy chmurowego [Rob18]. Istnieją sposoby pozwalające ograniczyć problem zimnego startu. Jeśli programista wie, że jego funkcja będzie uruchamiana tylko co jakiś czas, może on stworzyć planistę (ang. scheduler), który będzie wywoływał okresowo funkcję tak, aby dostawca chmurowy nie usunął instancji kontenera, w której się ona znajduje, z powodu jej nieaktywności.

2.2.6 Najpopularniejsi dostawcy rozwiązań przetwarzania bezserwerowego

Wśród najbardziej popularnych na dzień dzisiejszy dostawców wyróżnia się Amazon Web Services (AWS), Microsoft Azure oraz Google Cloud Platform (GCP) [Sti18].

AWS Lambda

Najpopularniejszym obecnie serwisem służącym do przetwarzania bezserwerowego jest AWS Lambda [RC17]. AWS był pierwszym z wiodących dostawców chmurowych, który zaoferował przetwarzanie bezserwerowe, co miało miejsce w listopadzie 2014 roku. Na początku w podejściu tym była możliwość pisania funkcji tylko we frameworku Node.js, jednak w kolejnych latach pojawiła się również możliwość pisania ich w językach C#, Java oraz Pythonie [Sti18], a obecnie można użyć jakiegokolwiek języka, który kompliuje się do Unixowego procesu [Rob18]. Za pomocą AWS Lambda programiści mogą w krótkim czasie tworzyć luźno powiązane, skalowalne i wydajne architektury systemu [SK17]. Zasada działania AWS Lambda została przedstawiona na poniższym Rysunku 2.9.



RYSUNEK 2.9: AWS Lambda [WG17]

Programista umieszcza kod swoich funkcji w serwisie AWS Lambda, konfiguruje wyzwalacze, które mają uruchomić funkcje — mogą nimi być np. inne serwisy AWS lub punkty końcowe HTTP, do których docierają zapytania od klientów. AWS uruchamia funkcje programisty tylko, gdy nastąpi zdarzenie ich wywołania, a same funkcje zużywają tylko tyle zasobów systemowych ile potrzebują. Programista płaci tylko za czas potrzebny na wykonanie obliczeń, a więc wykonanie kodu funkcji.

AWS Lambda powstało, ponieważ serwis EC2 nie potrafił obsługiwać, czy też odpowiedzieć na przychodzące zdarzenie np. zdarzenie umieszczenia pliku png w AWS S3. Dawniej, aby instancja EC2 mogła zareagować na takie zdarzenie programista musiał napisać algorytm powodujący okresowe sprawdzanie przez instancję zawartości AWS S3 jako, że EC2 nie miała wbudowanych

mechanizmów, aby ją o takim zdarzeniu poinformować. Tymczasem, funkcje AWS Lambda mogą być poinformowane o takim zdarzeniu, które je wywoła i mają możliwość bycia uruchamianym na kontenerach umieszczonych właśnie na wspomnianych instancjach EC2 [WG17].

Struktura funkcji

Na Listingu 2 przedstawiono przykładową implementację funkcji w AWS Lambda napisaną w języku JavaScript.

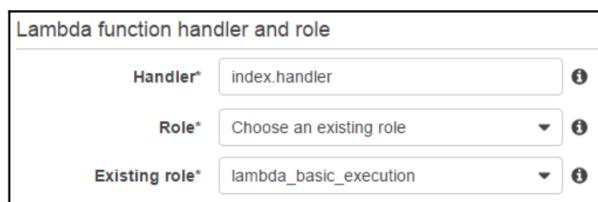
LISTING 2: Przykładowa funkcja w AWS Lambda

```

1 exports.handler = (event, context, callback) => {
2     console.log('value1 =', event.key1);
3     callback(null, event.key1);
4 };

```

Kod funkcji składa się z kilku elementów. Pierwszym z nich jest wywołanie funkcji (ang. invocation). Funkcja, w tym przypadku o nazwie "handler" jest eksportowana, a następnie wywoływana podczas wołania (ang. calling) pliku, w którym została zapisana. Plik ten zwykle przyjmuje nazwę "index", np. dla języka JavaScript "index.js". Zmienna "event" przechowuje dane o zdarzeniu, które wywołało funkcję. Z kolei "context" jest używany do pobrania kontekstowych informacji o funkcji programisty np. jej nazwy, ilości pamięci jaką ona konsumuje, ile czasu zajęło jej wykonanie. Za jego pomocą można też przerwać funkcję w określonym miejscu kodu bez dalszego przetwarzania za pomocą poleceń context.succeed(), context.fail() lub context.done(). Zmienna "callback" jest wykorzystywana do odesłania odpowiedzi do wywołującego i ma ona strukturę callback(error,result), gdzie w przypadku braku błędu w miejscu error podstawiona jest wartość null np. callback(null, event.key1). Użycie zmiennej "callback" jest opcjonalne, gdyż funkcja nie musi zwrócić rezultatu [WG17]. Nazwa przedstawionej funkcji, czyli "handler" może mieć dowolną nazwę np. "handlerFirmyX" jednak jest to domyślna nazwa, gdyż funkcja w pliku programisty, która ma być wywołana jest określana w formularzu konfiguracyjnym AWS właśnie w polu o nazwie handler co przedstawiono na poniższym Rysunku 2.10.



RYSUNEK 2.10: AWS Lambda formularz konfiguracyjny [WG17]

W polu "handler" programista podaje nazwę pliku zawierającego funkcję, która ma zostać wywołana oraz samą nazwę funkcji w formacie nazwaPliku.nazwaFunkcji. W tym przypadku handlerem jest "index.handler" [WG17].

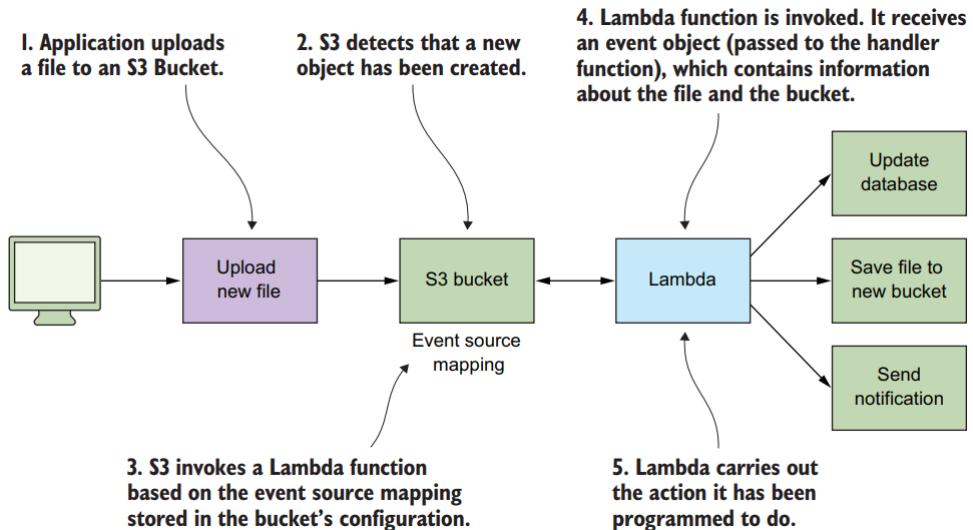
Funkcje pisane w AWS Lambda powinny być szybkie — im szybciej się one zakończą, tym mniej zapłaci firma, gdyż cennik AWS Lambda bazuje na liczbie zapytań, czasie trwania przetwarzania oraz ilości zaalokowanej na potrzeby realizacji funkcji pamięci [SK17]. Funkcje mają określony czas trwania, kryje się on pod parametrem *timeout* i wynosi 5 minut [Rob18]. Przykładem krótkich funkcji możliwych do napisania w AWS Lambda może być komendy dla botów w komunikatorach takich jak *Slack*. Bot może w nim odpowiadać na komendy, przetwarzać małe zadania, wysyłać raporty oraz powiadomienia [SK17].

Przetwarzanie zdarzeń

Niektóre firmy za pomocą AWS Lambda przetwarzają miliardy zdarzeń dziennie [RC17]. Zdarzenia te są przetwarzane w sposób maksymalnie zrównoległy. Funkcja reaguje na następujące ich typy [SK17]:

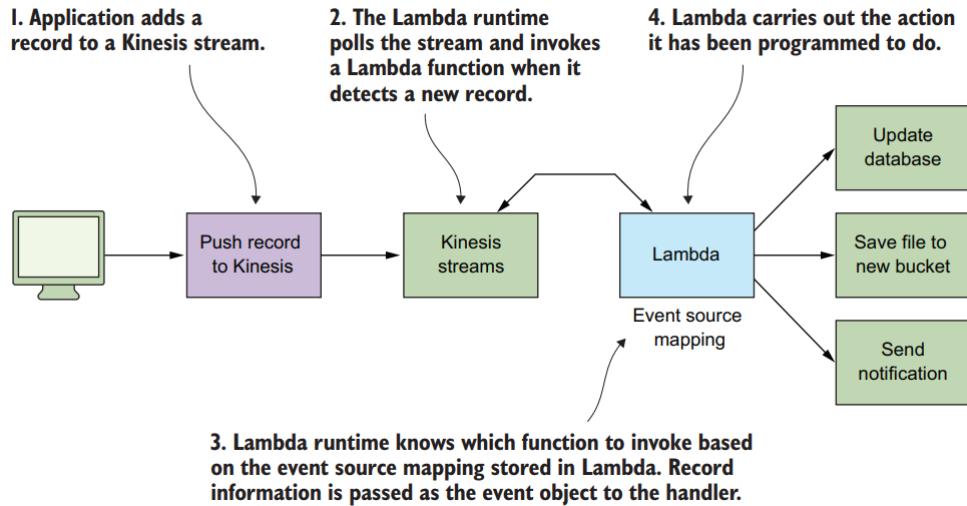
- zdarzenia zachodzące w AWS — mogą nimi być utworzenie, aktualizacja lub usunięcie obiektu w AWS S3, aktualizacja rekordu w tabeli DynamoDB, opublikowanie wiadomości w systemie kolejkowym *AWS SNS* lub zaplanowane zdarzenia (ang. scheduled events) tzw. *cron jobs* [WG17].
- zapytania HTTP przychodzące do API Gateway
- wywołania API przy wykorzystaniu *AWS SDK*
- ręczne wywołanie funkcji przy użyciu konsoli AWS (ang. AWS console)

Wywołanie funkcji spowodowane zajściem zdarzenia bazuje na 2 modelach: push oraz pull. W modelu push serwis np. *Amazon S3* publikuje zdarzenie do AWS Lambda i bezpośrednio wywołuje funkcję programisty. Na poniższym Rysunku 2.11 pokazano jak wygląda przykładowa implementacja modelu push.



Po umieszczeniu pliku w Amazon S3, będącym składowią danych (ang. data storage), serwis ten po wykryciu stworzenia w nim nowego obiektu emittuje zdarzenie, które bezpośrednio wywołuje funkcję AWS Lambda.

Z kolei w modelu pull AWS Lambda odpytuje źródłowy strumień zdarzeń (ang. streaming event source) jakim jest np. strumień serwisu DynamoDB (ang. DynamoDB stream) lub strumień serwisu Kinesis (ang. Kinesis stream) i decyduje o swoim wykonaniu przy spełnieniu określonych warunków. Na poniższym Rysunku 2.12 pokazano jak wygląda przykładowa implementacja modelu pull.

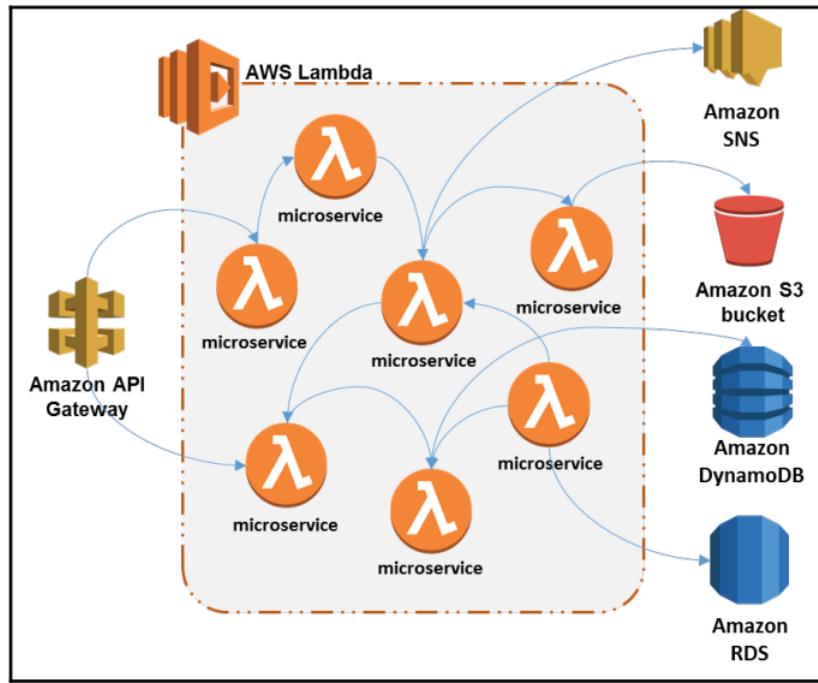


Na Rysunku 2.12 funkcja AWS Lambda zostanie wykonana jedynie w przypadku, gdy po odpytaniu serwisu Kinesis odkryje, że w źródle jego strumienia zdarzeń pojawił się nowy rekord z danymi.

Wzorce architektoniczne

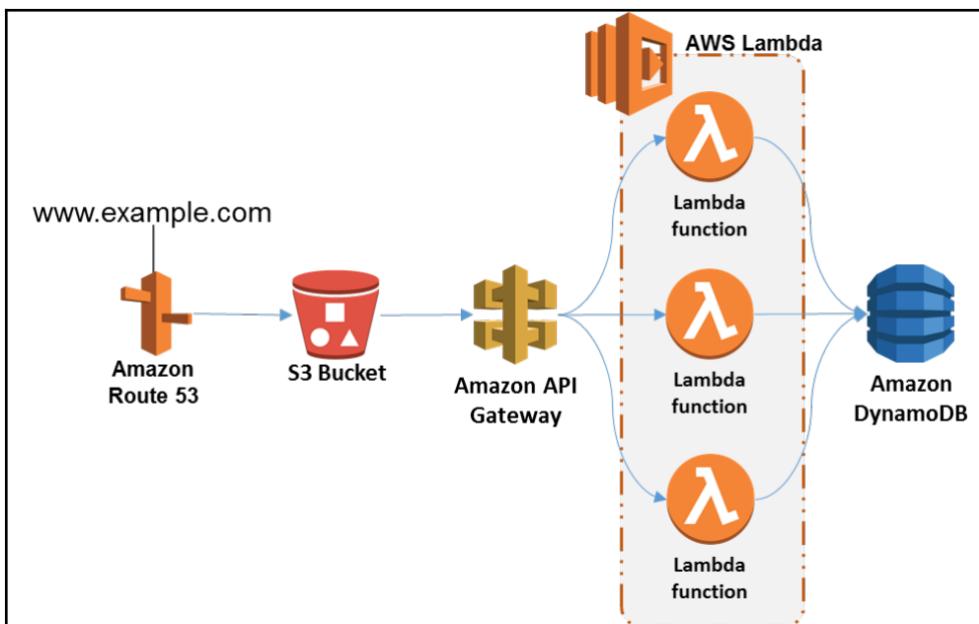
W związku z popularyzacją AWS Lambda powstały dla niej wzorce architektoniczne (ang. Lambda architecture patterns) będące dla programistów szkieletami dla projektowania aplikacji opartych na zdarzeniach. Zaliczają się do nich [WG17]:

- mikroserwisy serverless — mikroserwisy są projektowane i rozwijane tak, aby były niezależne i samowystarczalne, więc znajdują szerokie zastosowanie w AWS Lambda. Funkcja w przetwarzaniu bezserwerowym też jest niezależna i zwraca odpowiedź w skończonym czasie. Problemem tego rozwiązania może być zbyt duża liczba funkcji, a więc mikroserwisów, ponieważ zarządzanie nimi może stanowić wyzwanie dla programisty, zwłaszcza podczas wdrażania aplikacji i wprowadzania w niej zmian [WG17]. Przykład takiej architektury, w którym każda funkcja jest niezależnym mikroserwisem mogącym komunikować się niezależnie z różnymi serwisami AWS została przedstawiona na poniższym Rysunku 2.13.



RYSUNEK 2.13: mikroserwisy serverless [WG17]

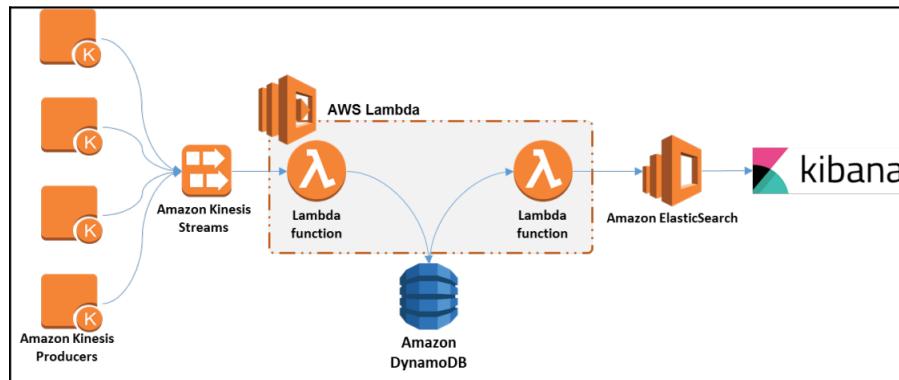
- wielowarstwowe aplikacje serverless (ang. serverless multi-tier applications) — jest jednym z najczęściej używanych wzorców architektonicznych, w których Lambda jest odpowiednią platformą do tworzenia logiki biznesowej tego typu aplikacji. Wzorzec ten polega na umieszczaniu kodów frontendowych strony w Amazon S3, logiki biznesowej aplikacji w funkcjach AWS Lambda, do których ruch kieruje API Gateway, natomiast do przechowywania i aktualizacji danych korzysta się z amazonowych serwisów takich jak DynamoDB, RDS lub ElastiCache [WG17]. Przykład takiej architektury został przedstawiony na poniższym Rysunku 2.14.



RYSUNEK 2.14: wielowarstwowe aplikacje serverless [WG17]

- przetwarzanie danych w czasie rzeczywistym (ang. real-time stream processing) — AWS Ki-

nesis jest efektywnym i skalowalnym rozwiązaniem umożliwiającym projektowanie i rozwój aplikacji, które mogą analizować duże ilości strumieniowych danych (ang. streaming data) [WG17]. Danymi tymi mogą być logi, zdarzenia systemowe, transakcje, czy dane z mediów społecznościowych [SK17, WG17]. Funkcje AWS Lambda mogą być skonfigurowane do bycia uruchomionym, gdy określona liczba rekordów danych, nazywana rozmiarem pakietu danych (ang. batch size), jest dostępna do przetworzenia. Połączenie Kinesis i AWS Lambda jest dobrym wyborem dla aplikacji, w których zachodzi konieczność analizy, gromadzenia lub zapisywania dużych rozmiarów danych w czasie rzeczywistym. Jeśli podczas przetwarzania pakietu danych dojdzie do błędu, to Lambda będzie próbowała ponowić dla niego przetwarzanie przez maksymalnie 24 godziny — ponieważ tyle wynosi czas przetrzymywania danych w AWS Kinesis aż do ich wygaśnięcia [SK17]. Przykład wzorca architektury realizującego przetwarzanie w czasie rzeczywistym został przedstawiony na poniższym Rysunku 2.15.



RYSUNEK 2.15: przetwarzanie danych w czasie rzeczywistym [WG17]

Na powyższym rysunku dane z różnych źródeł są przekazywane do AWS Kinesis, następnie cyklicznie funkcja AWS Lambda procesuje pakiet danych i wyniki procesowania umieszcza w bazie DynamoDB. Następnie, druga funkcja pobiera dane z DynamoDB i przekazuje je do serwisu *Amazon ElasticSearch* aby analityk mógł je przeanalizować i wyświetlić w narzędziu *Kibana*.

Powtarzające się zadania

Funkcje mogą być uruchamiane cyklicznie przez planistę (ang. scheduler), co czyni je efektywnymi w przypadku powtarzających się zadań takich jak importu i eksportu danych, tworzenie kopii zapasowych, ustawianie przypominaczy (ang. reminders) i powiadomień. Programiści często wykorzystują takie funkcje w celu wykonywania komendy ping, której argumentem jest ich strona internetowa, w celu sprawdzenia jej żywotności i wysłania maila lub wiadomości tekstowej w przypadku braku uzyskania odpowiedzi. Ponadto, programiści często wykorzystują funkcję w celu conocnych pobrań plików z serwerów, czy wysyłki dziennych wyciągów z konta użytkownikom.

API Gateway

AWS posiada własną implementację API Gateway o nazwie API Gateway, będący konfigurowalnym serwisem typu BaaS. Poza przekazywaniem odpowiednich zapytań HTTP odpowiednim funkcjom, API Gateway służy też do uwierzytelnienia, walidacji argumentów wejściowych, czy transformacji wyniku zwróconego przez funkcję [Rob18]. API Gateway dostarcza również pamięć

podręczną, aby zredukować opóźnienia i zmniejszyć obciążenie systemu zwracając szybciej odpowiedź klientowi. Serwis ten posiada również tzw. dławienie (ang. throttling) zmniejszający liczbę wywołań funkcji w czasie do konfigurowalnej liczby używając algorytmu *token bucket algorithm*. Można w ten sposób uchronić backend przed bycie nadmiernie obłożonym zapytaniami wysyłanymi przez klientów. Mechanizm *Lambda Proxy Integration* umożliwia serwisowi API Gateway odwzorowanie ciała zapytania HTTP (ang. request body) do zdarzenia i wywołanie nim funkcji AWS Lambda [SK17].

Skalowanie

W przypadku, gdy platforma otrzymuje zdarzenie będące wyzwalaczem danej funkcji, uruchamia ona kontener, aby uruchomić kod programisty. W sytuacji, gdy to zdarzenie wciąż jest obsługiwane przez danąinstancję funkcji, a zostanie otrzymane nowe zdarzenia, platforma uruchamia kolejną instancję funkcji, aby przetworzyć drugie zdarzenie. Taki automatyczny mechanizm pionowego skalowania będzie kontynuowany aż AWS Lambda uruchomi wystarczającą liczbę funkcji, aby móc każde zdarzenie uruchomić w osobnej instancji. Aby zabezpieczyć się przed atakiem typu DDoS, który mógłby uruchomić tysiące kontenerów, Amazon stworzył zmienną parametryzującą maksymalną możliwą liczbę jednocześnie zainicjalizowanych kontenerów Lambda na jednym koncie AWS. Domyślnie wynosi ona 1000, ale jej wartość może zostać zmieniona przez programistę [RC17].

Kontenery

Ważnym aspektem w zrozumieniu zaawansowanych metod wykorzystania AWS Lambda jest możliwość wielokrotnego użycia kontenera. Gdy funkcja jest uruchamiana po raz pierwszy, nowy kontener jest tworzony i kod funkcji jest do niego pobierany — funkcja jest "zimna" przy jej pierwszym uruchomieniu. W przypadku ponownego wywołania funkcji AWS Lambda może użyć tego samego kontenera i pominąć proces jej inicjalizacji — w takim przypadku funkcja jest "ciepła", czyli uruchomiona w zainicjalizowanym wcześniej kontenerze, dzięki czemu kod wykonuje się szybciej [SK17].

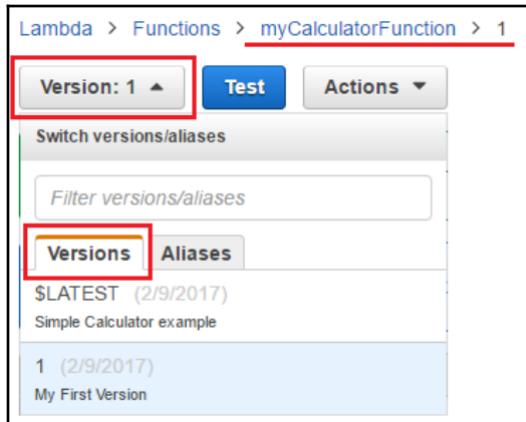
Zimny start

Istnieje kilka sposobów na rozwiązanie problemu zimnego startu, są to między innymi:

- W AWS za planistę może posłużyć serwis *AWS CloudWatch* pozwalający na zaplanowanie zdarzeń, które mają się wydarzyć co pewien okres czasu i być zapalnikiem wywołania funkcji programisty, tak aby funkcja została wywołana już w istniejącym kontenerze — aby była ciepła, co pozwoli uniknąć zimnego startu [SK17].
- zwiększyć rozmiar pamięci alokowanej do funkcji, ponieważ im więcej pamięci przydzieli jej programista, tym szybciej zostanie zainicjalizowana.
- wybrać odpowiedni język programowania — przykładowo w języku Java zimny start trwa najdłużej [SK17].

Wersjonowanie

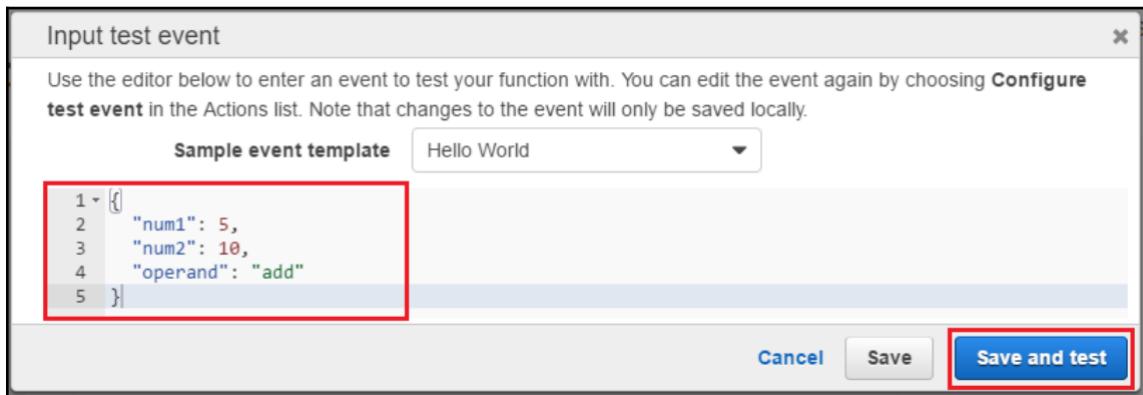
AWS Lambda posiada wbudowane narzędzia do wersjonowania i przypisywania aliasów funkcjom, co może być przydatne przy wdrażaniu aplikacji na wielu środowiskach, takich jak rozwojowe, testowe oraz produkcyjne [Sti18]. Za pomocą aliasów można odwoływać się do funkcji np. podczas konfiguracji jaką funkcję ma dane zdarzenie wywołać. Dzięki temu nie trzeba się odwoływać do funkcji poprzez jej *ARN* (*Amazon Resource Name*), czyli unikalny w ramach platformy AWS identyfikator zasobu [WG17]. Przykład nadania funkcji wersji i aliasu został przedstawiony na poniższym Rysunku 2.16.



RYSUNEK 2.16: Wersjonowanie w AWS Lambda [WG17]

Testowanie

Testować funkcję w AWS Lambda można bezpośrednio w konsoli AWS. Na tę potrzebę definiuje się ciało zdarzenia (ang. event body), które będzie argumentem wejściowym funkcji dostępnym w polu "event", a następnie naciska przycisk "Save and test", co powoduje wywołanie funkcji ze wskazanymi w ciele zdarzenia danymi, co przedstawiono na poniższym Rysunku 2.17.



RYSUNEK 2.17: Testowanie funkcji w AWS Console [WG17]

Jeśli funkcja zostanie poprawnie wykonana, to AWS Console zwróci jej rezultat w polu "Execution result".

W AWS Lambda istnieje też możliwość tworzenia testów jednostkowych korzystając z wielu bibliotek dla różnych języków. Na poniższym Listingu 3 przedstawiono przykład implementacji testu jednostkowego napisanego przy pomocy biblioteki *Mocha* w języku JavaScript.

LISTING 3: Przykładowy test jednostkowy w AWS Lambda

```
1 describe('myLambda',function(){
2     it('Check value returned from myLambda',function(){
3         expect(retval).toEqual("someValue");
4     })
5 })
```

Test ten sprawdza, czy wartość zwrócona przez funkcję o nazwie "myLambda" jest równa co do wartości łańcuchowi tekstowemu "someValue".

Monitoring

AWS Lambda do monitoringu wykorzystuje serwis *AWS CloudWatch*. Aby pozwolić funkcji na wysyłanie logów do serwisu CloudWatch należy zdefiniować odpowiednie polityki (ang. policies). Przykładowo, pozwolić funkcji na tworzenie grup logów (ang. log groups) oraz strumieni w CloudWatch oraz umieścić zdarzenia logowania w konkretnym strumieniu. Do konfiguracji uprawnień służy serwis AWS o nazwie *IAM policy*.

Do analizy ruchu w aplikacjach rozproszonych służy serwis *AWS X-Ray*. Bada on wywołania funkcji, ich wydajność, komunikację między aplikacjami oraz pozwala programistom na ich debugowanie [WG17].

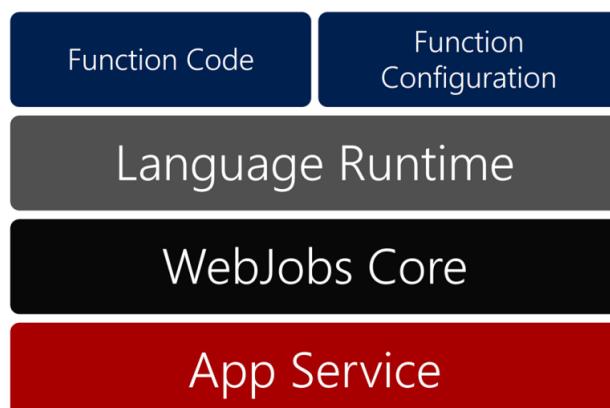
Wdrażanie kodu funkcji

Istnieje kilka sposobów wdrożenia kodu funkcji do AWS Lambda:

- programista może wpisywać kod bezpośrednio w przeglądarce za pomocą wbudowanego edytora kodu [WG17]
- programista może umieścić w AWS Lambda plik w formacie zip zawierający kody funkcji [WG17].
- AWS Lambda dostarcza bazowe obrazy dockerowe typu open-source, które programista może użyć do stworzenia obrazu kontenera. Do ich tworzenia służy *AWS Serverless Application Model (AWS SAM) command line interface (CLI)* lub natywne narzędzie kontenerowe jak *Docker CLI*. Po stworzeniu obrazu i umieszczeniu w nim kodów funkcyjnych, należy go umieścić w należącym do firmy Amazon repozytorium obrazów Dockerowych o nazwie *Amazon Elastic Container Registry (Amazon ECR)*. Aby wdrożyć obraz jako funkcję AWS Lambda należy w jej konfiguracji podać adres url do obrazu dockerowego umieszczonego w Amazon ECR [aws22].

Azure Functions

Serwisem przetwarzania bezserwerowego oferowanym przez Microsoft jest Azure Functions. Jest on oparty na serwisie *Azure App Service* [CP22], służącym do budowania i udostępniania aplikacji internetowych oraz do pełnienia roli backendu dla aplikacji mobilnych bez konieczności zarządzania infrastrukturą. Azure Functions wspiera zarówno środowiska Linux jak i Windows. Oferuje on autoskalowanie, wysoką dostępność, zautomatyzowane drożenie kodów z GitHuba, Azure DevOps lub dowolnego innego repozytorium Git. W Azure Functions, serwis App Service jest używany do skalowania instancji oraz zarządzania środowiskiem uruchomieniowym (ang. runtime) wykonującym kod funkcji Azure Functions. Dla funkcji opartych na systemie Windows, serwer jest uruchomiony jako PaaS i skala się w ramach środowiska uruchomieniowego .NET. Architektura serwisu Azure Functions przedstawiona została na Rysunku 2.18 [app22b].



RYSUNEK 2.18: Architektura Azure Functions [CP22]

Fundamentem architektury jest App Service. Wyżej znajduje się *WebJobs Core* dostarczający kontekst wykonawczy dla funkcji. Środowiska wykonawcze dla języka (ang. Language Runtime) uruchamia skrypty, wykonuje kod bibliotek i uruchamia framework dla podanego docelowego języka (ang. target language) wykorzystanego do implementacji funkcji np. Node.js jest wykorzystywany do uruchomienia funkcji JavaScript, a dla funkcji napisanych w C# jest to framework .NET. W tym środowisku programista może wykonywać swój kod (ang. function code) oraz modyfikować parametry konfiguracyjne funkcji (ang. function configuration) [CP22].

Funkcje w Microsoft można tworzyć i uruchamiać według trzech różnych planów:

- app service plan — jest to plan definiujący, które zasoby sprzętowe mają zostać użyte do uruchomienia funkcji. Można przypisywać plany do regionu, ustalić rozmiar i liczbę wirtualnych maszyn, które mają zostać użyte, wybrać parametry takie jak CPU i RAM jakie maszyny mają posiadać. Nie jest to typowy sposób przetwarzania bezserwego i powinno się z niego skorzystać, gdy wymagania systemowe narzucają tworzenie funkcji, których przetwarzanie będzie trwało przez długi okres czasu [KG].
- consumption plan — powinno się z niego skorzystać dla rozwiązania całkowicie zgodnych z ideą przetwarzania bezserwerowego. Plan ten pozwala skalować liczbę maszyn wirtualnych opierając się na liczbie otrzymywanych zdarzeń [azu22]. Programista płaci w nim jedynie za czas wykonywania funkcji, której limit czasu wynosi 5 minut [Saw19].
- premium plan — jest rozszerzeniem planu typu consumption oferującym tzw. ciepłe instancje (ang. pre-warmed instances), czyli takie, które po starcie zostają wypełnione danymi pamięci

podręcznych (ang. cache) utrzymywanych przez inne instancje i zawierających informacje na temat poprzednich wywołań funkcji w celu skrócenia czasu przetwarzania. W planie "consumption plan" firma płaci za czas wykonania funkcji i wykorzystaną pamięć, z kolei w planie premium za sekundy zużytego CPU oraz pamięci. Przy trwającym dłuższy okres przetwarzaniu premium plan może być bardziej wydajny kosztowo [KG].

Funkcje w Microsoft Azure są wsparte przez rozszerzenie o nazwie *app service plan*. Jest to plan definiujący, które zasoby sprzętowe mają zostać użyte do uruchomienia funkcji. Można przypisywać plany do regionu, ustalić rozmiar i liczbę wirtualnych maszyn, które mają zostać użyte, wybrać parametry takie jak CPU i RAM jakie te maszyny mają posiadać.

Dla rozwiązania zgodnego z definicją przetwarzania bezserwerowego powinno się skorzystać z tzw. *consumption plan* [CP22]. Pozwala on skalować liczbę maszyn wirtualnych opierając się na liczbie przychodzących zdarzeń [azu22]. Programista płaci w nim jedynie za czas wykonywania się funkcji, której limit czasu wynosi 5 minut [Saw19].

Funkcje zorientowane na stan

W celu tworzenia funkcji zorientowanych na stan korzysta się z *Durable Functions*. Jest to rozszerzenie umożliwiające zarządzanie stanem, tworzenie punktów kontrolnych, odtwarzanie stanu aplikacji do konkretnego punktu kontrolnego za programistę, dzięki czemu może on się skupić na implementacji logiki biznesowej funkcji. Rozszerzenie to pozwala na tworzenie *entity functions* umożliwiających pisanie fragmentów kodu odpowiedzialnych za odczyt i aktualizację małych fragmentów stanu nazywanych *durable entities*. Entity functions wywoływanie są przez specjalny wywzalacz o nazwie *entity trigger*. Durable entities pozwalają na współdzielenie stanu w środowisku rozproszonym przez wiele zeskalowanych funkcji. Same entities zachowują się jak małe serwisy komunikujące się za pomocą wiadomości. Każdy entity ma unikalny identyfikator oraz wewnętrzny stan. Entities podobnie jak obiekty, czy serwisy wykonują pewne operacje. Wykonanie operacji może skutkować aktualizacją stanu danego entity, czy wywołaniem zewnętrznego serwisu. Entities komunikują się pomiędzy sobą, a także z klientem za pomocą niezawodnych kolejek (ang. reliable queues). Aby zapobiec konfliktom, wszystkie operacje wykonywane na pojedynczym entity wykonują się sekwencyjnie, jedna po drugiej. Entity functions programiści mogą tworzyć w językach .NET, JavaScript oraz Python. Na poniższym Listingu 4 przedstawiono przykładową implementację entity function napisaną w języku JavaScript [azu22].

LISTING 4: Przykładowa entity function

```

1  const df = require("durable-functions");
2  module.exports = df.entity(function(context) {
3      const currentValue = context.df.getState(() => 0);
4      switch (context.df.operationName) {
5          case "add":
6              const amount = context.df.getInput();
7              context.df.setState(currentValue + amount);
8              break;
9          case "reset":
10             context.df.setState(0);
11             break;
12          case "get":
13              context.df.return(currentValue);

```

```

14         break;
15     }
16 });

```

Funkcja, której definicja zaczyna się na Listingu 4 od linii 2 odpowiada za aktualizację stanu licznika. Przykładowo dla operacji "add" pobiera ona jego aktualną wartość z kontekstu (ang. context) za pomocą metody context.df.getState oraz dodaje do niej wartość odczytaną z argumentu wejściowego za pomocą metody context.df.getInput(). Następnie następuje aktualizacja stanu licznika poprzez dodanie wartości aktualniej oraz odczytanej za pomocą metody context.df.setState(currentValue + amount).

Rozszerzenie Durable functions pozwala także na pisanie tzw. *orchestrator functions*, czyli przepływów pracy (ang. workflows) składających się z wielu funkcji dzielących stan. Wynik orchestrator functions zapisywany jest do lokalnej zmiennej, niezależnie od faktu wywoływania innych funkcji — zarówno synchronicznych, jak i asynchronicznych. Funkcje te są niezawodne — postęp przetwarzania jest zapisywany w punktach kontrolnych przy użyciu w kodzie słów kluczowych "await" lub "yields". Stan lokalny nigdy nie jest tracony przy ponowieniu przetwarzania lub restarcie serwera. Funkcje te mogą trwać długo: sekundy, dni, miesiące, czy też nigdy się nie skończyć co nie jest typowe dla funkcji przetwarzania bezserwerowego [azu22]. Na poniższym Listingu 5 przedstawiono przykładową implementację orchestrator function napisaną w języku JavaScript.

LISTING 5: Przykładowa orchestrator function

```

1 const df = require("durable-functions");
2
3 module.exports = df.orchestrator(function*(context) {
4     const output = [];
5     output.push(yield context.df.callActivity("E1_SayHello", "Tokyo"));
6     output.push(yield context.df.callActivity("E1_SayHello", "Seattle"));
7     output.push(yield context.df.callActivity("E1_SayHello", "London"));
8
9     // returns ["Hello Tokyo!", "Hello Seattle!", "Hello London!"]
10    return output;
11 });

```

Każda komenda yield zapisuje stan przetwarzania domyślnie w serwisie *Azure Table*. Zapisany stan jest określony mianem historii orkiestracji (ang. orchestration history) [azu22].

Wyzwalacze

Dana funkcja może być wywołana tylko przez jeden wyzwalacz. Poza wywoływaniem funkcji, wyzwalacze pełnią też rolę powiązań (ang. bindings). Powiązania dostarczają deklaratywny sposób łączenia zasobów z kodem. Można zdefiniować wiele powiązań dla jednego wyzwalacza. Mogą być one przekazane w ramach argumentów wyjściowych. Ułatwiają one tworzenie i zarządzanie funkcjami ponieważ zrzucają z programisty odpowiedzialność zatworzenie połączenia do bazy danych lub systemu plików. Całą potrzebną informację o powiązaniach programista definiuje w specjalnym pliku functions.json. Przykładami powiązań są *CosmosDB* ułatwiający łączenie z bazą danych w celu ładowania i zapisywania plików, *Table Storage* pozwalający korzystać w funkcji z bazy danych typu klucz/wartość (ang. key/value storage) oraz *Queue Storage* pozwalający na łatwe pozyskiwanie elementów z koleiki lub umieszczanie ich na nowej. Przykład zawartości

pliku functions.json definiujący wyzwalacz oraz powiązanie został przedstawiony na poniższym Listingu 6.

LISTING 6: Przykładowy function.json

```

1   {
2     "bindings": [
3       {
4         "name": "$return",
5         "type": "queue",
6         "direction": "out",
7         "queueName": "images",
8         "connection": "AzureWebJobsStorage"
9       }
10    ],
11    "disabled": false
12  }

```

W tym przykładzie wyzwalacz umieszcza wiadomość na kolejce o nazwie "images".

Event Grid

Event Grid jest serwerem typu BaaS służącym do upraszczania bezserwerowych aplikacji, opartych na zdarzeniach poprzez zarządzanie kierowaniem ruchu (ang. routing) wszystkich zdarzeń z dowolnego źródła (jakim jest dowolny serwis i dowolna platforma) do dowolnego miejsca docelowego. Zdarzenia do Event Gridu publikowane są za pomocą wiadomości. Za jego pomocą można wysłać własną wiadomość gridową (ang. custom event grid message), która zostanie skonsumowana przez innego dostawcę chmurowego lub aplikację uruchomioną w modelu on-premise. Usługa zapewnia wysoką dostępność, wydajność, pracę z systemami wysoce skalowalnymi oraz dostarczanie wiadomości w czasie rzeczywistym [eve22, CP22].

Monitoring

Azure Functions jest zintegrowane z serwisem *Azure Application Insights* w celu monitorowania funkcji. Serwis ten zbiera logi, dane o wydajności i informacje o błędach. Automatycznie wykrywa on anomalie w wydajności i posiada rozbudowane narzędzia analityczne w celu prostego zdiagnozowania problemów i lepszego zrozumienia jakości napisanego kodu. Azure Application Insights posiada 7 poziomów logowania [azu22]:

- Trace — logi te posiadają najbardziej szczegółowe wiadomości zawierające dane wrażliwe związane z aplikacją. Domyślnie ten poziom jest wyłączony i nie powinien być wykorzystywany w środowisku produkcyjnym
- Debug — logi wykorzystywane do interaktywnego badania rezultatów działania kodu podczas rozwoju kodów aplikacji. Logi te powinny zawierać informacje potrzebne do debugowania.
- Information — logi śledzące ogólny przepływ działania aplikacji
- Warning — logi uwydatniające anomalie lub niespodziewane zdarzenia powstałe podczas działania aplikacji, ale nie powodujące konieczności zatrzymania aplikacji

- Error — logi wyjaśniające powód dlaczego obecne przetwarzanie funkcji zostało zatrzymane.
- Critical — logi opisujące przyczyny utraty żywotności całej aplikacji, błędy krytyczne w systemie
- None — wyłącza monitoring funkcji

Google Cloud Functions

Implementacja funkcji w rozwiązaniu przetwarzania bezserwerowego oferowana przez firmę Google o nazwie Google Cloud Functions (GCF) bazuje na frameworku ExpressJS [HMW22]. Funkcje mogą być pisane w językach Node.js, Python, Go, Java, .NET, Ruby oraz PHP [clo22a]. Istnieje kilka ich typów — mogą być oparte na protokole HTTP (ang. HTTP functions) albo na zdarzeniach (ang. event-driven functions). Funkcje oparte na zdarzeniach dzielą się na *background functions* oraz *CloudEvent functions* w zależności od silnika uruchomieniowego (ang. runtime) w jakim zostały napisane. Wyzwalaczami takich funkcji mogą być tematy Pub/Sub (ang. Pub/Sub topics) w systemach kolejkowych lub zmiany w systemie plików np systemie Google Cloud Storage (GCS) [clo22a].

Funkcje HTTP

Wywoływane są przy użyciu zapytań HTTP. Metody HTTP jakich można użyć przy tworzeniu tych funkcji to GET, PUT, POST, DELETE oraz OPTIONS. W GCF wszystkie funkcje HTTP są automatyczne zabezpieczone certyfikatem TLS, aby zapewnić bezpieczne połączenie [clo22a]. Implementacja przykładowej funkcji HTTP napisanej w języku Node.js została przedstawiona na poniższym Listingu 7.

LISTING 7: Przykładowa funkcja HTTP

```

1 const functions = require('@google-cloud/functions-framework');
2   functions.http('helloHttp', (req, res) => {
3     res.send(`Hello World`);
4   });

```

Przedstawiona funkcja na dowolne przychodzące zapytanie odpowiada wiadomością 'Hello world'.

Background functions

Są to funkcje wspierane dla języków Node.js, Go, Python oraz Java oparte na zdarzeniach takich jak wiadomość w systemach kolejkowych, zmiana w systemie plików GCS, zdarzenie pochodzące z zewnętrznego serwisu BaaS oferowanego przez Google np Firebase [clo22a]. Implementacja przykładowej background function napisanej w języku Node.js została przedstawiona na poniższym Listingu 8.

LISTING 8: Przykładowa Background function

```

1 exports.helloPubSub = (message, context) => {
2   const msg = message.data
3   ? Buffer.from(message.data, 'base64').toString()
4   : 'World';
5   console.log(msg);
6 };

```

Jest to funkcja, której wyzwalaczem jest wiadomość w systemie kolejkowym, która po otrzymaniu wiadomości loguje ją metodą console.log() (linia 5).

CloudEvent functions

Są to funkcje koncepcyjnie bardzo podobne do background functions. Mogą być pisane w językach PHP, C# oraz Ruby. Różnicą jest to, że wyzwalacze tych funkcji muszą być ustandaryzowanymi

zdarzeniami chmurowymi, czyli zdarzeniami typu *CloudEvents*. Ponadto, wywołanie CloudEvent function może wywołać inną CloudEvent function używając zapytań HTTP. Korzystając z tego typu funkcji reagujących na ustandaryzowane zdarzenia, firmy mogą niskim kosztem dokonać migracji swojej aplikacji do innego dostawcy chmurowego w przypadku rezygnacji z korzystania z usług Google [clo22a]. Implementacja przykładowej CloudEvent function napisanej w języku Ruby została przedstawiona na poniższym Listingu 9.

LISTING 9: Przykładowa CloudEvent function

```

1  require "functions_framework"
2  require "base64"
3
4  FunctionsFramework.cloud_event "hello_pubsub" do |event|
5    name = Base64.decode64 event.data["message"]["data"] rescue "World"
6
7    logger.info "Hello, #{name}!"
8 end

```

Funkcja na Listingu 9, której wyzwalaczem jest zdarzenie kolejkowe pobiera imię klienta z treści wiadomości i loguje je metodą logger.info (linia 7)

GCF, w przeciwieństwie do AWS Lambda, nie pozwala na uruchamianie funkcji bazujących na obrazach dockerowych tworzonych przez programistę. Google umożliwia jedynie uruchamianie całych aplikacji, bazujących na obrazach dockerowych, w bezstanowych kontenerach zarządzanych i skalowanych przez dostawcę chmurowego w ramach usługi *Cloud Run (GCR)* [HMW22]. Serwisem będącym odpowiednikiem tej usługi oferowanym przez AWS jest AWS App Runner [app22a].

Zewnętrzne biblioteki

GCF oferuje korzystanie z zewnętrznych bibliotek przy implementacji funkcji dla języków: Node.js, Python, Go, Java, .NET, Ruby oraz PHP. Przykładowo dla języka Node.js programista definiuje zewnętrzne biblioteki w pliku package.json o zawartości przedstawionej na poniższym Listingu 10 [clo22a].

LISTING 10: przykładowy package.json z definicją zewnętrznych bibliotek

```

1  {
2    "dependencies": {
3      "escape-html": "^1.0.3"
4    }
5  }

```

Po zdefiniowaniu zewnętrznej biblioteki, którą w tym przypadku jest escape-html (linia 3), programista może z niej skorzystać w celu implementacji funkcji, co pokazano na Listingu 11.

LISTING 11: przykładowa funkcja wykorzystująca zewnętrzną bibliotekę

```

1  const functions = require('@google-cloud/functions-framework');
2  const escapeHtml = require('escape-html');
3
4  functions.http('helloHttp', (req, res) => {
5    res.send(`Hello ${escapeHtml(req.query.name || req.body.name || 'World')}`);
6  });

```

Wdrażanie

Programista ma możliwość użyć repozytoria *Cloud Store* w celu wdrażania kodów funkcji, pobierając je z repozytoriów *GitHub* lub *Bitbucket*. Redukuje to czas potrzebny na manualne pakowanie kodów do pliku zip i umieszczania ich w platformie GCP za pomocą odpowiedniej komendy w terminalu [Sti18].

Lokalne testowanie

Google umożliwia lokalne testowanie funkcji za pomocą tzw. *Function Frameworks* lub *Cloud Native buildpacks*. Function Frameworks są to biblioteki typu open source używane w GCF w celu transformacji zapytania HTTP do wywołania funkcji specyficznego dla danego języka. Można je użyć do konwersji funkcji do serwisu HTTP, który można wywołać lokalnie. Z kolei Cloud Native buildpacks są używane do umieszczenia w kontenerze serwisów HTTP uzyskanych poprzez Functions Framework, dzięki czemu programista może przetestować daną funkcję korzystając z platformy Docker [clo22a].

Dodatkowo, Google udostępnia emulator funkcji, będący repozytorium Git pozwalającym na wdrażanie, testowanie i uruchamianie funkcji na lokalnej maszynie przed wdrażaniem jej bezpośrednio do GCP [Sti18].

Zimny start

Jeśli programista wie, że jego funkcja będzie uruchamiana tylko co jakiś czas, może on stworzyć planistę (ang. scheduler), który będzie wywoływał okresowo funkcję tak, aby dostawca chmurowy nie usunął instancji kontenera, w której się ona znajduje, z powodu jej nieaktywności. W przypadku GCP planistą jest wyzwalacz czasowy (ang. time trigger) o nazwie *App Engine Cron* korzystający z systemu kolejkowego wysyłając wiadomość o określonym temacie (ang. topic), na który nasłuchują zainteresowane nim funkcje [Sti18].

Logowanie

GCF oferuje automatyczne logowanie za pomocą narzędzia o nazwie *Stackdriver Logging*. Stackdriver przechowuje logi do 30 dni logując wszystko co programista potrzebuje wypisać w logach w celu monitoringu, a także np. dane o wydajności funkcji. Dodatkowo Stackdriver pozwala na debugowanie kodu w środowisku produkcyjnym [Sti18].

Porównanie cen usług poszczególnych dostawców chmurowych

Na poniższym Rysunku 2.19 przedstawiono ceny jakie oferują poszczególni dostawcy chmurowi za wykonywanie funkcji [Sti18].

AWS Lambda	Azure Functions	Google Cloud Functions
First million requests a month free	First million requests a month free	First 2 million requests a month free
\$0.20 per million requests afterwards	\$0.20 per million requests afterwards	\$0.40 per million requests afterwards
\$0.00001667 for every GB-second used	\$0.000016 for every GB-second used	\$0.000025 for every GB-second used

RYSUNEK 2.19: Ceny wykonywania funkcji oferowane przez dostawców chmurowych [Sti18]

W przypadku AWS Lambda oraz Azure Functions pierwszy milion żądań wykonania funkcji w miesiącu jest darmowy, z kolei w przypadku Google Cloud Functions darmowe są pierwsze 2 miliony, jednak w przeciwieństwie do pozostałych dwóch dostawców chmurowych dla, których każde kolejne żądanie kosztuje \$0.20, Google Cloud Functions po przekroczeniu liczby darmowych żądań kosztuje 2 razy, więcej - cena każdego kolejnego żądania wynosi \$0.40. Ponadto w przypadku Google programista płaci najwięcej za każdy Gigabajt pamięci użyty przez sekundę trwania funkcji. Cena ta wynosi \$0.000025, podczas gdy dla AWS Lambda koszt wynosi \$0.00001667, a dla Azure Functions \$0.000016 [Sti18].

2.3 Przetwarzanie na krawędzi

2.3.1 Definicja

Przetwarzanie na krawędzi (ang. edge computing) jest paradygmatem przetwarzania rozproszonego, w którym dane klienta są przetwarzane na krawędzi sieci — możliwie najbliżej źródła danych jakimi są urządzenia mobilne, czy sensory [SRCL⁺22, ADSGKT20]. W modelu tym część operacji takich jak filtrowanie i agregacja zarejestrowanych danych jest wykonywana po stronie urządzeń końcowych redukując rozmiar danych wysyłanych do chmury w celu dalszego przetwarzania [DIPW20]. Potrzeba przetwarzania danych na urządzeniach, które je rejestrują lub produkują wynika między innymi z powodu opóźnień komunikacyjnych występujących ze względu na konieczność przesłania danych, których tempo powstawania przekracza szybkość przesłania danych przez sieć, posiadającą ograniczoną przepustowość. Na przestrzeni lat systemy wyewoluowały z używania jednego komputera, w celu przetwarzania zadań, do korzystania ze scentralizowanych, stanowiących całość serwisów i aplikacji w chmurowych centrach danych (ang. cloud data centers). Rozwój technologii, którego przykładem są coraz wydajniejsze telefony komórkowe, domowe sieci internetowe, a także pojawiająca się potrzeba użytkowników do kontrolowania swoich aplikacji, doprowadziła do potrzeby stworzenia rozwiązania przenoszącego ciężar obliczeń na krawędź sieci — która może być reprezentowana przez urządzenie, sieć domową, czy centrum danych znajdujące się najbliżej użytkownika końcowego [ADSGKT20].

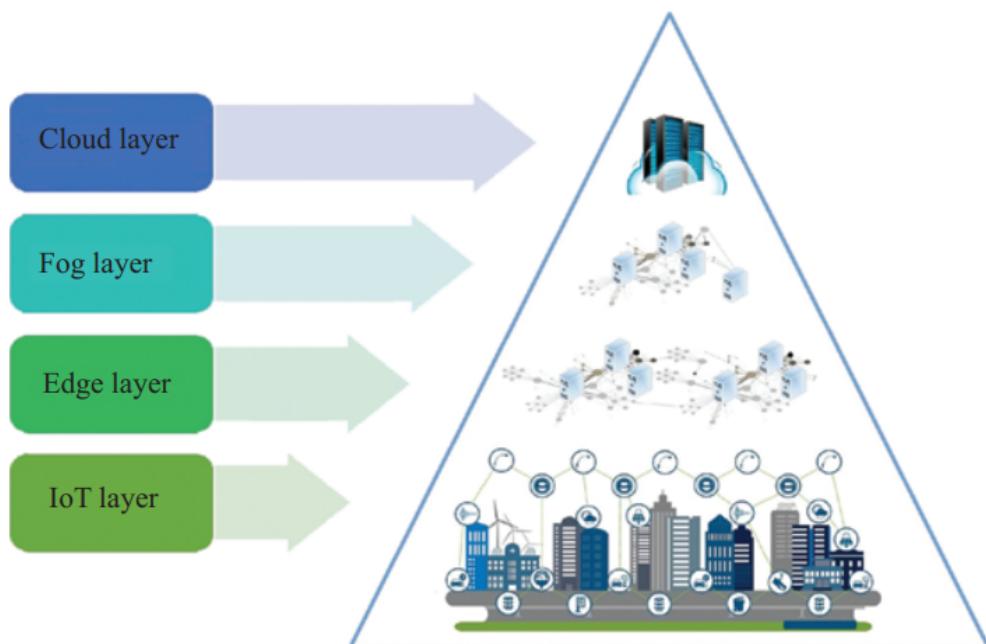
2.3.2 Architektura

Filozofia przetwarzania na krawędzi polega na wykonywaniu części obliczeń na krawędzi jak najbliżej źródła danych tak, aby zmniejszyć ograniczenia architektur zorientowanych na chmurowe centra danych (ang. Cloud-centric architectures) [ADSGKT20, SCAC⁺19].

Zaletami takiego rozwiązania są:

- Oszczędność zasobów — przesyłanie mniejszej ilości danych przez sieć, co pozwala na osiągnięcie mniejszego obciążenia sieci oraz zapełnienie mniejszego rozmiaru przestrzeni dyskowej w centrach danych.
- Redukcja opóźnień komunikacyjnych — przesyłanie mniejszej ilości danych powoduje szybsze dotarcie zapytania do centrum chmurowego. Ponadto, wykonanie części obliczeń po stronie urządzenia końcowego powoduje wykonanie mniejszej liczby obliczeń po stronie dostawcy chmurowego, co pozwala na szybsze otrzymanie odpowiedzi na żądanie klienta.
- Zwiększoną skalowalność — zdecentralizowane miejsca przechowywania danych i rezultatów obliczeń w postaci urządzeń końcowych ułatwiają skalowalność poprzez zwiększanie liczby tych urządzeń.
- Zwiększenie izolacji i prywatności — zapewniane dzięki przechowywaniu i przetwarzaniu wrażliwych danych na urządzeniach końcowych oraz braku konieczności ich przesłania przez sieć [SCAC⁺19].
- zwiększenie niezawodności w przypadku awarii sieci — pomimo braku połączenia sieciowego, część obliczeń może być wykonanych lokalnie na urządzeniu końcowym [KSDN22].

Zasobami przetwarzania na krawędzi mogą być sieć lub zasoby obliczeniowe uruchamiane po stronie użytkowników końcowych po jednej stronie oraz centrów chmurowych lub węzłów mgły (ang. fog nodes) po drugiej stronie. Architektura przetwarzania na krawędzi zazwyczaj składa się z czterech warstw: warstwy IoT (*Internet of Things*), warstwy na krawędzi (ang. edge layer), warstwy mglistej (ang. fog layer) oraz warstwy chmurowej. Składowe te przedstawia Rysunek 2.20 [ADSGKT20].



RYSUNEK 2.20: Architektura przetwarzania na krawędzi [ADSGKT20]

W architekturze zaprezentowanej na Rysunku 2.20 warstwa IoT zawiera miliony urządzeń oraz sensorów produkujących dane, wymieniające między sobą informacje za pomocą interfejsów sieciowych, monitorujące i kontrolujące swoją infrastrukturę. W warstwie tej znajdują się użytkownicy

końcowi przetwarzania na krawędzi. Urządzenia IoT często wymagają krótkich czasów odpowiedzi zamiast wysokiej przepustowości obliczeniowej i dużych rozmiarów baz danych. Przetwarzanie na krawędzi oferuje tym urządzeniom wystarczającą przepustowość obliczeniową i szybki czas odpowiedzi zaspokajający ich wymagania [ADSGKT20].

Urządzenia IoT mogą zostać wykorzystane jako węzły na krawędzi (ang. edge nodes) oferujące serwisy.

Warstwa na krawędzi składa się z serwerów nazywanych węzłami na krawędzi (ang. edge nodes) i pomostów (ang. gateways) mających większe zasoby obliczeniowej od urządzeń końcowych. Pomosty służą do zapewniania funkcjonalności sieciowych takich jak tunelowanie, przerywanie połączenia sieciowego, tworzenie zapory ogniodzielnej [KSDN22]. Węzły na krawędzi mogą zostać wykorzystane do wykonania obliczeń, do których urządzenia końcowe posiadają niewystarczające zasoby na ich wykonanie. Węzły te mogą również oferować serwisy, tak aby dostarczać usługi jak najbliżej urządzeń końcowych. Warstwa mgły oraz chmurowa jest wykorzystywana do zapewnienia zasobów obliczeniowych, których może brakować urządzeniom oraz na krawędzi [ADSGKT20].

2.3.3 Przetwarzanie bezserwerowe na krawędzi

Aplikacje IoT działają w oparciu o zdarzenia takie jak wykrycie temperatury powyżej pewnego poziomu stopni Celsjusza, czy wykrycie konkretnego poziomu wilgotności gleby, a następnie wykonaniu krótkich zadań związanych z wystąpieniem takiego zdarzenia. Zatem cechy takich aplikacji pasują do funkcjonalności jakie oferuje model przetwarzania bezserwerowego, który inicjalizuje funkcje zorientowane na zdarzenia tylko wtedy, gdy takowe wystąpią (co zapewnia efektywne wykorzystanie zasobów obliczeniowych) oraz pozwala uzyskać wysoką skalowalność [CCP21, JTA21]. Ponadto w modelu przetwarzania bezserwerowego klient nie będzie zmuszony płacić za czas pracy serwerów, w którym nie będą pojawiały się nowe dane do przetworzenia [ATC⁺21]. Zagrożeniem wiążącym się z bezserwerowym przetwarzaniem na krawędzi jest konieczność zapewnienia niezawodności — jeśli węzeł przetwarzający dane z urządzenia końcowego ulegnie awarii, zapytanie tego urządzenia może zostać odrzucone lub przetworzone wiele razy, co nie sprzyja wykorzystaniu takiego rozwiązania przykładowo przy przetwarzaniu danych medycznych, czy danych związanych z autonomiczną kontrolą jazdy pojazdu [XTQ⁺21]. Przykładem wykorzystującym rozwiązanie bezserwerowe na krawędzi jest serwis *Lambda@Edge*, będący rozszerzeniem serwisu AWS Lambda, oferowany przez firmę AWS. Uruchamia on funkcje Lambda na serwerach zlokalizowanych najbliżej użytkownika, a więc na krawędzi, co redukuje opóźnienia w dostarczaniu odpowiedzi klientom [JTA21].

2.3.4 Przypadki użycia

- Autonomiczne pojazdy — w celu poprawnego działania potrzebują gromadzić i analizować dane o otoczeniu takie jak warunki atmosferyczne, aby podjąć decyzje o tym jak się zachować. Decyzje te powinny wynikać z poprzednich doświadczeń i po ich podjęciu przez urządzenie końcowe jakim jest samochód powinny być wysyłane do najbliższego węzła na krawędzi utrzymywaneego przez producenta samochodu w celu ulepszania procesu podejmowania decyzji [KSDN22].
- Inteligentne miasta (ang. smart cities) — dane o mieście mogą być pobierane z różnych punktów końcowych sieci takich jak budynek, samochody, czy urządzenia za pomocą sensorów w celu rozwoju obszarów miejskich w oparciu o technologie informatyczne. Następnie dane są zbierane, przetwarzane i analizowane w celu efektywnego korzystania z zasobów miasta, tak

aby poprawić płynność ruchu drogowego w mieście, wydajniej zużywać energię za pomocą inteligentnego oświetlenia miejskiego, czy znajdować wolne miejsca parkingowe. Przykładem wykorzystania koncepcji inteligentnych miast jest Wiedeń zastępujący tradycyjne autobusy elektrycznymi, a ich ruch jest możliwy do śledzenia przez pasażerów za pomocą aplikacji monitorującej ruch drogowy. Kolejnym przykładem inteligentnego miasta jest Lubin, który po wdrożeniu systemu inteligentnego oświetlenia ulicznego redukującego strumień światła w przypadku braku ruchu drogowego zaoszczędził do 75% na opłatach za energię elektryczną [KSDN22, mub22].

- Automatyzacja przemysłu — wykorzystanie przetwarzanie na krawędzi w przemyśle zapewnia większą wydajność energetyczną i przemysłową monitorując parametry produkcyjne, analizując zebrane za pomocą urządzeń końcowych dane oraz podejmując na ich podstawie decyzje przyczyniające się do uzyskania wyższej produkcji niższym kosztem. Przetwarzanie na krawędzi znajduje zastosowanie w wykrywaniu oszustw, wydobycia ropy naftowej, analizie logistycznej, czy startach samolotów poprzez monitorowanie i analizę danych o pogodzie [KSDN22].
- Gry komputerowe — obecnie gry wymagają szybkiego połączenia sieciowego, a w przypadku powstawania platform takich jak *Google Stadia* jest to konieczne, ponieważ, nie renderują one gry na lokalnym urządzeniu końcowym gracza takim jak komputer, czy konsola, tylko na serwerze zdalnym i strumieniujących je do gracza. Przy użyciu przetwarzania na krawędzi, gra jest strumieniowana graczowi z najbliższego dostępnego serwera dostawcy platformy co zapewnia niskie opóźnienia i tym samym poprawia doświadczenia użytkownika (ang. user experience) [KSDN22, goo22].
- Służba zdrowia — ze względu na ograniczenia związane z przetrzymywaniem danych medycznych w chmurze wynikające z ogólnego rozporządzenia o ochronie danych (ang. General Data Protection Regulation (GDPR)), możliwe jest skorzystanie z przetwarzania na krawędzi i przechowywanie wrażliwych danych medycznych po stronie urządzeń końcowych [SRS17].

2.3.5 Zagrożenia

- Integracja systemu — zarządzanie systemem składającym się z wielu typów urządzeń końcowych z wieloma typami serwisów znajdujących się w warstwie na krawędzi lub w chmurze może stanowić wyzwanie. Przetwarzanie na krawędzi integruje różne platformy, topologie sieciowe, serwery i środowiska. Zarządzanie niejednorodnym systemem powoduje trudność w rozwoju systemu, zarządzaniu danymi i zasobami dla wielu aplikacji uruchomionych na różnych platformach w różnych lokalizacjach na świecie [ADSGKT20].
- Rozwój systemu — w rozwoju aplikacji opartych na usługach chmurowych programista zazwyczaj pisze ją w jednym języku w oparciu o jednego dostawcę chmurowego. Jednak w przypadku przetwarzania na krawędzi programista ma do czynienia z niejednorodnymi platformami, urządzeniami i środowiskami. Posiadają one często różne środowiska wykonawcze, wymagają korzystania z różnych języków programowania w celu rozwijania aplikacji w swoich środowiskach co może sprawić trudność programiście [ADSGKT20].
- Odkrywanie węzłów na krawędzi — z uwagi na obecność wielu urządzeń końcowych w wielu lokalizacjach na świecie przypisanie im manualnej konfiguracji ze wszystkimi możliwymi lokalizacjami węzłów na krawędzi, z którymi mogą się połączyć nie jest dobrym pomysłem, mając na uwadze możliwość awarii węzła lub możliwość dołączenia nowego węzła do sieci

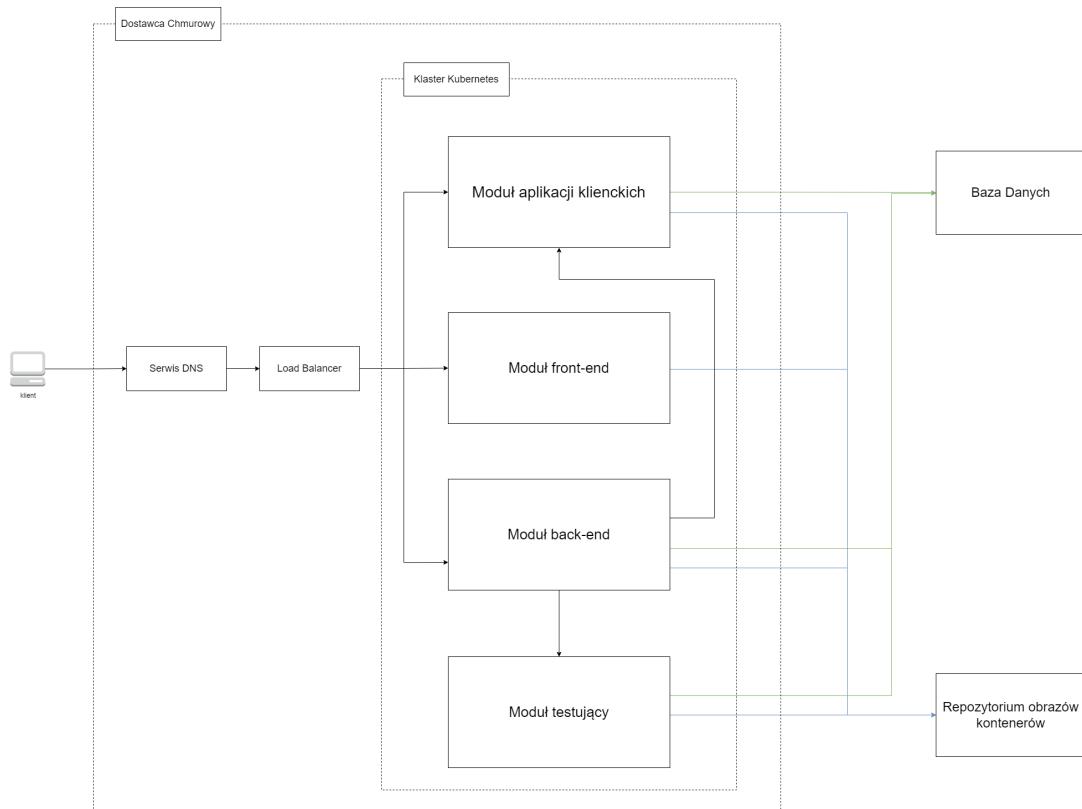
w wyniku konieczności obsługi większego ruchu w sieci na krawędzi. Należy zatem zapewnić mechanizm dynamicznego odnajdywania najbliższych węzłów w sieci, musi on działać poprawnie dla heterogenicznych typów urządzeń końcowych [ADSGKT20].

- Bezpieczeństwo — znalezienie niezawodnego rozwiązania bezpieczeństwa w systemie z niejednorodnymi platformami, urządzeniami i środowiskami jest złożonym zadaniem [ADSGKT20].

Rozdział 3

Architektura zaproponowanego rozwiązania

Na Rysunku 3.1 przedstawiono architekturę zaproponowanego rozwiązania.



RYSUNEK 3.1: Architektura zaproponowanego rozwiązania

W ramach architektury wyróżniamy część aplikacyjną klienta oraz część implementującą oferowaną funkcjonalność. Składa się ona z następujących modułów: front-end, back-end, testującego oraz modułu aplikacji klienckich. Moduły te uruchomione są w klastrze Kubernetes, uruchomionym na serwerach oferowanych przez dostawcę chmurowego. Klient może komunikować się z modułami w klastrze za pomocą serwisu DNS pozwalającego zarejestrować nazwę domenową i przy jej użyciu łączyć się z modułami klastra za pomocą protokołu HTTP. Ruch kierowany do klastra jest równoważony systemem równoważenia obciążenia (ang. Load Balancer). Każdy z modułów zawiera aplikację, budowaną na podstawie obrazów kontenerów znajdujących się w repozytorium obrazów

kontenerów, skąd obraz jest pobierany i instancjonowany w postaci aplikacji podczas uruchamiania klastra Kubernetes. Moduł front-end odpowiada za wyświetlenie klientowi w przeglądarce strony internetowej umożliwiającej tworzenie oraz testowanie aplikacji przetwarzania bezserwerowego na krawędzi. Za pomocą graficznego interfejsu (ang. GUI) użytkownika oferowanego przez ten moduł, klient ma możliwość stworzenia własnego konta, a następnie stworzenia aplikacji oraz kodów funkcji, z których będzie się ona składać. Stworzenie aplikacji, a następnie funkcji i punktów końcowych (ang. endpoints) umożliwiających wywołanie funkcji odbywa się poprzez wysyłanie zapytań do modułu back-end, a następnie zapis do bazy danych. Zaimplementowana funkcja może być przetestowana za pomocą modułu testującego. W tym celu klient za pomocą GUI wypełnia argumenty wejściowe funkcji z jakimi chce ją wywołać, a następnie naciska przycisk wywołujący żądanie HTTP przetestowania funkcji z podanymi argumentami wejściowymi. Żądanie to trafia do modułu backendowego, który uruchamia aplikację testującą w module testującym, a następnie wysyła do tego modułu żądanie przetestowania funkcji. Jeśli w kodzie funkcji pojawia się wywołanie innej istniejącej już funkcji, w celu jej wywołania moduł testujący pobiera jej kody z bazy danych. Po wykonaniu funkcji moduł testujący zwraca wynik modułowi backendowemu, a ten klientowi. Po stworzeniu i przetestowaniu aplikacji, klient przy pomocy GUI może ją uruchomić, tak aby była dostępna dla wszystkich w internecie, wysyłając żądanie uruchomienia do modułu backendowego. Moduł ten tworzy aplikację o nazwie nadanej przez klienta w module aplikacji klienckich. Jest ona dostępna dla klienta pod adresem poddomeny nazwy domenowej pozwalającej łączyć się klientowi z klastrem Kubernetes. Nazwą poddomeny jest nazwa aplikacji zdefiniowana przez klienta. Aplikacja ta przy inicjalizacji pobiera z bazy danych kody swoich funkcji oraz punkty końcowe, za pomocą których klient może te funkcje wywołać. Przy wywołaniu punktu końcowego tej poddomeny, klient wysyła do modułu aplikacji klienckich żądanie HTTP wywołania funkcji kryjącej się pod tym punktem końcowym, a w odpowiedzi dostaje jej wynik dla podanych argumentów wejściowych. W przypadku wywołania funkcji idempotentnych, klient dostaje rezultat pamięci podręcznej składającej się z wyników funkcji tego typu. Po umieszczeniu w ciele kolejnego zapytania zawartości pamięci podręcznej, aplikacja w module aplikacji klienckich dla funkcji idempotentnych skorzysta z jej wyników zamiast ponownego dokonania obliczeń, jeśli w tej pamięci znajduje się wywołanie funkcji o tej samej nazwie z tymi samymi argumentami wejściowymi. Klient może też wysłać do tego modułu żądanie pobrania kodów funkcji o podanych w ciele zapytania nazwach w celu ich lokalnego wywołania, a zatem w celu dokonania obliczeń na krawędzi. Uzyskane rezultaty klient może umieścić w ciele żądania HTTP wywołania punktu końcowego swojej aplikacji klienckiej. Dzięki temu aplikacja w module aplikacji klienckich przy wywołaniu funkcji wykonanej już wcześniej przez klienta, nie wywoła jej po swojej stronie, gdy klient wywołał ją dla takich samych argumentów wejściowych, a przypisze jej wynikowi wynik obliczeń klienta.

Rozdział 4

Implementacja

4.1 Specyfikacja implementacji

Poniżej przedstawiono wymagania funkcjonalne i pozafunkcjonalne stawiane zaproponowanemu rozwiązaniu.

4.1.1 Wymagania funkcjonalne

1. Klient ma możliwość utworzenia konta umożliwiającego mu korzystanie z usług zaproponowanego rozwiązania.
2. Klient ma możliwość zalogowania się na swoje konto.
3. Klient ma możliwość resetu hasła, za pomocą otrzymanego emaila z linkiem resetującym hasło.
4. Klient ma możliwość stworzenia, modyfikacji oraz usuwania aplikacji, funkcji, punktu końcowego.
5. Klient ma możliwość zdefiniowania w pliku z zależnościami zewnętrznych bibliotek, z których będzie korzystał implementując funkcje aplikacji.
6. Klient implementując funkcję, ma możliwość wywołania w jej kodach inną funkcję z arbitralnymi argumentami wejściowymi oraz skorzystać z wyniku jej przetwarzania.
7. Klient ma możliwość przetestowania stworzonej funkcji wywołując ją z arbitralnymi argumentami wejściowymi.
8. Klient ma możliwość uruchomienia i zatrzymania aplikacji.
9. Klient ma możliwość wykonywania zapytań do zdefiniowanych dla uruchomionej aplikacji punktów końcowych w celu wywołania funkcji przypisanej do tego punktu końcowego.
10. Klient musi otrzymać rezultat wywołanej za pomocą punktu końcowego funkcji oraz zawartość pamięci podręcznej dla funkcji idempotentnych.
11. Klient ma możliwość przekazania zawartości pamięci podręcznej w ciele zapytania HTTP do punktu końcowego w celu przyspieszenia obliczeń.
12. Klient ma możliwość wykonania skorzystania z przetwarzania na krawędzi pobierając kody wybranych funkcji i wykonanie ich lokalnie po swojej stronie, a następnie przekazaniu ich rezultatów w ciele zapytania HTTP do punktu końcowego w celu przyspieszenia obliczeń.

4.1.2 Wymagania pozafunkcjonalne

1. Bezpieczeństwo — Klient musi być uwierzytelniony w celu korzystania z systemu umożliwiającego tworzenie, modyfikacje, uruchamianie, zatrzymywanie, usuwanie oraz testowanie aplikacji.
2. Niezawodność — Aplikacja po uruchomieniu przez klienta musi działać w sposób niezawodny. System, umożliwiający klientowi tworzenie aplikacji musi działać w sposób niezawodny.
3. Przenośność — Zaproponowana implementacja aplikacji i zasobów znajdujących się w klaszterze Kubernetes może zostać przeniesiona do innego dostawcy chmurowego.

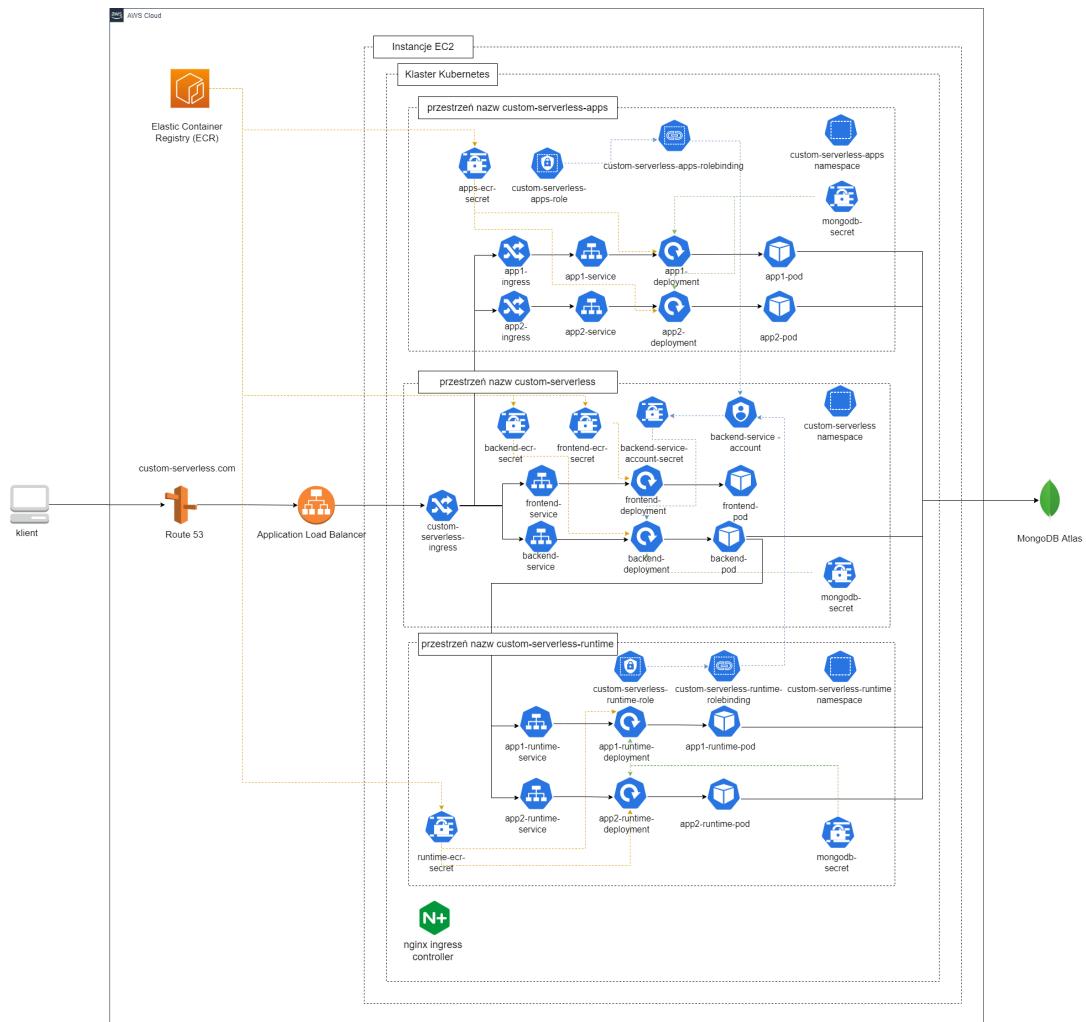
4.2 Opis implementacji technicznej

4.2.1 Wykorzystane technologie

- Docker — omówione w sekcji 2.1.4 narzędzie konteneryzacji.
- Kubernetes — platforma typu open source umożliwiająca zarządzanie i orkiestrację kontenerów w sposób imperatywny za pomocą komend w terminalu oraz w sposób deklaratywny za pomocą plików w formacie yaml, który umożliwia, między innymi, skalowanie aplikacji umieszczonych w kontenerach oraz równoważenie ich obciążenia. Ponadto, zapewnia on restart kontenera w przypadku awarii procesu, który został w nim uruchomiony oraz wersjonowanie umożliwiające programiście wycofanie do poprzedniej wersji obrazu kontenera np. w przypadku wystąpienia błędu w środowisku produkcyjnym[wha22c].
- Terraform — narzędzie typu infrastruktura jako kod (ang. Infrastructure as Code (IaC)), umożliwiające tworzenie infrastruktury u dowolnego dostawcy chmurowego za pomocą implementacji przejrzystych plików konfiguracyjnych pisanych w języku HCL, które można wersjonować, wielokrotnie używać oraz udostępniać innym programistom[wha22d].
- Angular — platforma rozwojowa oparta na języku TypeScript i HTML umożliwiająca tworzenie wysoce skalowalnych stron internetowych oraz oferująca narzędzia pozwalające na rozwój, tworzenie, testowanie oraz aktualizacje kodu. Ponadto, biblioteka ta pozwala na komunikację typu klient-serwer, czy tworzenie i zarządzanie formularzami[wha22a].
- Express.js — elastyczny framework oparty na środowisku uruchomieniowym Node.js, umożliwiający tworzenie aplikacji internetowych[wha22b].

4.2.2 Szczegóły implementacyjne

Szczegółowe komponenty zaproponowanej architektury, przedstawionej na Rysunku 3.1 zostały zaprezentowane na Rysunku 4.1.



RYSUNEK 4.1: Implementacja zaproponowanego rozwiązania

Dostawcą chmurowym wybranym w zaproponowanej implementacji jest AWS. Serwisy udostępniane przez tego dostawcę zostały wybrane do implementacji wybranych komponentów zaprezentowanej na Rysunku 3.1 architektury. Serwis DNS jest zaimplementowany przez serwis *Route 53*, z kolei komponent Load Balancer implementowany jest z wykorzystaniem serwisu *Application Load Balancer*. Jako repozytorium obrazów kontenerów wybrano *Elastic Container Registry (ECR)*. Nazwą domenową udostępnianą klientowi przez serwis Route 53 jest *custom-serverless.com*. Bazą danych wykorzystywaną w zaproponowanym rozwiążaniu jest baza *MongoDB*. Baza ta udostępniona jest za pomocą serwisu *MongoDB Atlas*, oferującego zarządzanie bazą danych w chmurze oraz przejmującą odpowiedzialność za jej uruchamianie, skalowanie i naprawę w przypadku wystąpienia awarii. Klaster Kubernetes, w którym znajdują się przedstawione na Rysunku 3.1 moduły, został zainstalowany na instancjach maszyn wirtualnych *EC2* oferowanych przez platformę AWS. Za dostęp klienta do klastra odpowiada komponent *Nginx Ingress Controller* nasłuchujący na każdym węźle klastra typu *Worker*, na porcie 30000, do którego ruch jest kierowany z serwisu Application Load Balancer. Moduły front-end oraz back-end znajdują się w przestrzeni nazw *custom-serverless*, model aplikacji klienckich w przestrzeni nazw *custom-serverless-apps*, z kolei moduł testujący w przestrzeni nazw *custom-serverless-runtime*.

Na Rysunku 3.1 przerywanymi liniami koloru pomarańczowego oznaczono przepływ komponentów architektury, zależnych od repozytorium ECR. Następnie, przerywanymi liniami koloru zielonego oznaczono przepływ komponentów architektury zależnych od bazy danych MongoDB. Z

kolej przerywanymi liniami koloru niebieskiego oznaczono przepływ zależności komponentów wykorzystywanych przez moduł back-end potrzebnych do wykonywania operacji w obrębie innych modułów takich jak uruchamianie aplikacji klienckiej w przestrzeni nazw *custom-serverless-apps*.

Tworzenie infrastruktury

Infrastruktura, na której oparte jest zaproponowane rozwiązanie została stworzona za pomocą narzędzia Terraform. Narzędzie to pozwala na tworzenie, modyfikowanie i usuwanie infrastruktury za pomocą jednej komendy dla każdej z wymienionych operacji. Przy pomocy narzędzia Terraform najpierw stworzona jest wirtualna sieć prywatna w chmurze, oferowana przez serwis *Amazon Virtual Private Cloud (VPC)*. Pozwala ona na tworzenie zasobów w AWS w prywatnej logicznej sieci wirtualnej tworzonej w ramach tego serwisu. Aby umożliwić dostęp z zewnątrz do tej sieci tworzony jest zasób *AWS Internet Gateway*. W celu określenia zasad jakie adresy IP mają mieć dostęp do sieci prywatnej tworzony jest komponent *AWS Route Table*, a w celu powiązania prywatnej sieci z punktem dostępowym Internet Gateway tworzony jest komponent *AWS Route Table Association*. Następnie tworzone są instancje EC2, na których zostanie zainstalowany klaster Kubernetes. Maszyny te w przypadku zaproponowanej implementacji posiadają system operacyjny Ubuntu w wersji 20.04. W celu określenia uprawnień ruchu wchodzącego i wychodzącego ze stworzonych instancji tworzone są zasoby typu *AWS Security Group*. Otwierają one instancjom EC2 port 22, aby móc połączyć się do nich za pomocą protokołu ssh oraz porty niezbędne do poprawnego działania klastra Kubernetes. Dla węzła typu *Control Plane* w klastrze tymi portami są:

- 6443 — umożliwiający połączenie się z komponentem *API server* umożliwiającym komunikację z Kubernetesem za pomocą REST API
- 2379-2380 — na potrzeby komponentu *etcd*, będącego bazą danych klastra
- 10250 — na potrzeby komponentu *kubelet*, odpowiedzialnego za uruchomienie kontenerów w Podach
- 10251 — na potrzeby komponentu *kube-scheduler* będącego planistą wdrażania nowych zasobów do klastra
- 10252 — na potrzeby komponentu *kube-node-manager* decydującego, na który węzeł typu *Worker* ma trafić nowy Pod
- 6783 — na potrzeby, *Weave Net* będącego implementacją *Container Network Interface (CNI)*, czyli mechanizmu pozwalającego na komunikację Podów w klastrze, czy nadawaniu im unikalnego adresu IP w obrębie całego klastra

Z kolei na potrzeby węzłów typu *Worker* otwarto porty:

- 10250 — na potrzeby komponentu *kubelet*
- 3000-32767 — są to porty z których mogą korzystać zasoby Kubernetes typu *Service*
- 6783 — na potrzeby *Weave Net*

Tworzone instancje są przypisywane do opisanego wyżej Security Group oraz do stworzonej wcześniej sieci VPC. Następnie tworzony jest obiekt typu *AWS Route 53 record* kierujący ruch sieciowy z nazw domenowych określonych wyrażeniem regularnym (ang. regular expression) “*.custom-serverless.com” (a więc z nazwy domenowej i jego poddomen) do komponentu Load Balancer. W kolejnym kroku tworzony jest obiekt typu *AWS Application Load Balancer (ALB)*

oraz obiekt typu *AWS ALB Listener*, tak skonfigurowany, aby obiekt Load Balancer nasłuchiwał na zapytania HTTP od klienta na porcie 80. W ostatnim etapie tworzony jest komponent typu *AWS ALB Target Group* przekierowujący ruch sieciowy z ALB do węzłów typu *Worker* na port 30000.

Instalacja klastra Kubernetes wraz z jego zasobami

W celu zainstalowania klastra Kubernetes oraz jego zasobów, zaimplementowany został skrypt bash. Instaluje on na każdym z węzłów EC2 komponent *containerd* będący środowiskiem uruchomieniowym dla kontenerów. Następnie na każdym z węzłów instalowane jest narzędzie *kubectl* służące do interakcji z komponentem *API Server* za pomocą komend w terminalu oraz komponent *kubelet*. W kolejnym kroku na węźle typu *Control Plane* instalowane jest narzędzie *kubeadm*, służące do stworzenia węzła tego właśnie typu, instalując za pomocą komendy *kubeadm init* takie komponenty jak *kube-scheduler*, *kube-node-manager*, czy generując odpowiednie certyfikaty konieczne do interakcji z komponentami w klastrze. Po zakończeniu komendy *kubeadm init*, węzły typu *Worker* dołączane są do klastra za pomocą komendy, której treść jest generowana za pomocą narzędzia *kubeadm* polecienniem *kubeadm token create --print-join-command*. Na koniec instalowane są wszystkie zasoby klastra, przykładowo zasób typu Deployment, Secret, a także *Nginx Ingress Controller*, zdefiniowane w postaci manifestów, będących plikami w formacie yaml zawierających definicję zasobów Kubernetes. Instalacja odbywa się za pomocą wywołania polecenia *kubectl apply* w terminalu, której argumentem są zdefiniowane manifesty.

Komunikacja z modułami w klastrze

Pełna ścieżka komunikacji między klientem, a modułami w klastrze wygląda następująco:

1. Zapytanie klienta do nazwy domenowej o wyrażeniu regularnym spełniającym wzorzec “*.custom-serverless.com” jest obsługiwane przez serwis Route 53.
2. Następnie zapytanie jest przekierowywane pod adres DNS obiektu *Application Load Balancer* na port 80.
3. W kolejnym kroku *Load Balancer* przekierowuje zapytanie do jednego z węzłów typu *Worker* na port 30000, na którym nasłuchiwa zasób *Nginx Ingress Controller*.
4. Zasób ten następnie przekierowuje ruch do zasobów typu *Ingress*, które spełniają wzorzec adresu DNS, który wpisał klient.

W zależności od wzorca zapytanie może trafić do:

- zasobu typu *Ingress* o nazwie *custom-serverless-ingress*, znajdującego się na Rysunku 4.1 w przestrzeni nazw *custom-serverless*, który przekaże zapytanie dalej do zasobu back-end i front-end typu Pod.
- zasobu typu *Ingress* odpowiedzialnego za ruch do konkretnej aplikacji klienckiej, której przekaże następnie to zapytanie.

Bezpośredni ruch do modułu testującego jest możliwy tylko z modułu back-end.

Moduł front-end

Moduł front-end znajduje się na Rysunku 4.1 w klastrze Kubernetes w przestrzeni nazw *custom-serverless*, a w jego skład wchodzą następujące zasoby Kubernetes:

- *frontend-ecri-secret* — zasób typu Secret
- *frontend-service* — zasób typu Service
- *frontend-deployment* — zasób typu Deployment
- *frontend* — zasób typu Pod
- *custom-serverless-ingress* — zasób typu Ingress

Zasób *frontend-ecri-secret* służy komponentowi *frontend-deployment* do dostępu do repozytorium ECR w celu stworzenia Poda na podstawie obrazu o nazwie *custom-serverless-frontend* zapisanego w ECR. Zapytanie klienta po kody strony internetowej po wejściu w przeglądarce na adres *custom-serverless.com* po dotarciu do komponentu *Nginx Ingress Controller* trafia dalej do *custom-serverless-ingress*. Następnie trafia do frontend-service, a w kolejnym kroku do zasobu typu Pod o nazwie *frontend*. Pod ten zawiera w swoim kontenerze serwer webowy nginx zawierający kody aplikacji front-end napisanej w technologii Angular w języku TypeScript. Po dotarciu zapytania do serwera nginx, zwraca on w odpowiedzi kody strony internetowej oferowane przez aplikację modułu front-end, które następnie wyświetli klientowi przeglądarka. Aplikacja ta pełni funkcję GUI, które służy do tworzenia, modyfikacji, usuwania i testowania aplikacji — klient korzystając z GUI oraz kodów strony internetowej wykonuje w przeglądarce żądania do modułu back-end wykonującego wymienione operacje.

Moduł back-end

Moduł back-end znajduje się na Rysunku 4.1 w klastrze Kubernetes w przestrzeni nazw *custom-serverless*, a w jego skład wchodzą następujące zasoby Kubernetes:

- *backend-ecri-secret* — zasób typu Secret
- *mongodb-secret* — zasób typu Secret
- *backend-service* — zasób typu Service
- *backend-deployment* — zasób typu Deployment
- *backend* — zasób typu Pod
- *custom-serverless-ingress* — zasób typu Ingress
- *backend-service-account* — zasób typu Service Account
- *backend-service-account-secret* — zasób typu Secret

Zasób *backend-ecri-secret* służy komponentowi *backend-deployment* do dostępu do repozytorium ECR w celu stworzenia Poda na podstawie obrazu o nazwie *custom-serverless-backend* zapisanego w ECR. Obraz ten zawiera aplikację modułu back-end napisaną w języku JavaScript we frameworku Express.js w środowisku uruchomieniowym Node.js. Żądanie klienta o przesłanie

kodów strony internetowej po wejściu w przeglądarce na adres `www.custom-serverless.com/api` po dotarciu do komponentu *Nginx Ingress Controller* trafia dalej do *custom-serverless-ingress*. Następnie zapytanie jest przekazywane do do *backend-service*, a w kolejnym kroku do zasobu typu Pod o nazwie *backend*, który zawiera w swoim kontenerze aplikację modułu back-end. Zasób *mongodb-secret* umożliwia aplikacji modułu back-end prowadzenie interakcji z bazą danych MongoDB. Zasób *backend-service-account* umożliwia korzystanie z ról *custom-serverless-apps-role* oraz *custom-serverless-runtime-role*, pozwalających tworzyć i usuwać aplikacje klienckie i testowe. Z kolei zasób *backend-service-account-secret*, pozwala aplikacji modułu back-end, na wykorzystanie *backend-service-account* w celu tworzenia i usuwania wymienionych aplikacji.

Aplikacja modułu back-end służy do wykonywania operacji tworzenia, modyfikacji, usuwania, uruchamiania aplikacji oraz tworzenia i usuwania środowisk uruchomieniowych do testowania aplikacji w module testującym i przekazywaniu mu zapytań klienta wykonanych w celu testowania funkcji. Aplikacja modułu back-end umożliwia również tworzenie konta użytkownika, jego uwierzytelnienie pozwalające zalogować się do systemu oraz możliwość resetu hasła za pomocą wysłania maila zawierającego token pozwalający je zresetować. Wyniki operacji tworzenia, modyfikacji lub usuwania: aplikacji, funkcji lub punktu końcowego aplikacja ta zapisuje w bazie danych MongoDB w serwisie MongoDB Atlas.

Dodanie nowego użytkownika polega na dodaniu w bazie MongoDB do tabeli *User* rekordu z nazwą użytkownika oraz skrótem hash jego hasła tworzonego za pomocą biblioteki *bcrypt*. Przy logowaniu użytkownika do systemu, z podanego przez niego hasła, za pomocą biblioteki *bcrypt*, tworzony jest skrót hash i porównywany z skrótem hash zapisanym w bazie. Jeśli skróty hash są identyczne oznacza to poprawne uwierzytelnienie użytkownika i zwrotanie mu w rezultacie w ciasteczka (ang cookie) tokenu JWT, za pomocą którego może korzystać z API serwera jako zalogowany użytkownik.

Z kolei stworzenie aplikacji polega na dodaniu obiektu w tabeli *Application* w bazie z parametrami:

- *name* — przyjmujący wartość nazwy aplikacji podanej przez użytkownika w zapytaniu
- *user* — przyjmujący wartość identyfikatora użytkownika wykonującego zapytanie w celu stworzenia aplikacji
- *endpoints* — listę punktów końcowych danej aplikacji będących w momencie tworzenia aplikacji pustą listą
- *functions* — listę funkcji danej aplikacji będących w momencie tworzenia aplikacji pustą listą
- *up* — przyjmujący wartość *true* lub *false* w zależności od tego, czy aplikacja jest uruchomiona, jeśli jest to up przyjmuje wartość *true*, natomiast w momencie tworzenia aplikacji przyjmuje on wartość *false*
- *packageJson* — string będący zawartością pliku package.json zdefiniowanym, aby umożliwić użytkownikowi korzystanie z zewnętrznych bibliotek

Funkcja jest tworzona poprzez dodanie do aplikacji, będącej obiektem w tabeli *Application* w bazie, w parametrze *functions* obiektu do listy, którą on zawiera, obiekt ten posiada następujące parametry:

- *name* — przyjmujący wartość nazwy funkcji przekazanej w zapytaniu przez klienta
- *content* — przyjmujący wartość kodu funkcji przekazanego w zapytaniu przez klienta

- *idempotent* — przyjmujący wartość *true* lub *false*, *true* oznacza, że funkcja jest idempotentna, czyli wywołana wiele razy zwróci ten sam rezultat dla takich samych argumentów wejściowych bez efektów ubocznych (ang. side effect). Jeśli funkcja jest idempotentna to jej wynik trafia do pamięci podręcznej zwracanej klientowi razem z rezultatem funkcji.

Przykładowy kod funkcji obrazujący jej strukturę został przedstawiony na Listingu 12.

LISTING 12: Przykładowa implementacja funkcji

```

1  async (args) => {
2      let rezultatInnejFunkcji = await call("innaFunkcja", args);
3      // pozostałe obliczenia
4      let result = {
5          "x": rezultatInnejFunkcji
6      };
7      return result;
8 }
```

Wymaganiem wobec programisty jest stworzenie jedno–argumentowej funkcji, której argumentem jest obiekt zawierający argumenty wejściowe programisty. Obiekt ten może posiadać dowolną nazwę, domyślnie przyjmuje ona nazwę *args* (linia 1). Aby funkcja ta mogła wykonywać operacje asynchroniczne wymaganiem języka JavaScript jest użycie słowa kluczowego *async* przed nawiasem otwierającym definicję argumentów funkcji. Funkcja może wywoływać inną funkcję, korzystając z metody *call* (linia 2). Metoda *call* jest funkcją asynchroniczną przyjmującą 2 argumenty: nazwę funkcji, którą ma wywołać oraz argumenty wejściowe, z których wywoływana funkcja ma skorzystać. Uzyskany w wyniku wywołania metody *call* rezultat można przechować w zmiennej. Aby to uczynić należy użyć przed nazwą metody *call* słowa kluczowego *await* używanego w języku JavaScript w celu oczekiwania na wynik operacji asynchronicznej, a następnie zwrócenia jej wyniku. Na koniec programista może zwrócić wynik definiowanej funkcji w postaci obiektu JSON (linia 7).

Stworzenie punktu końcowego polega na dodaniu do aplikacji będącej obiektem w tabeli *Application* w bazie w parametrze *endpoints* obiektu do listy, którą on zawiera, obiekt ten posiada następujące parametry:

- *functionName* — przyjmujący nazwę funkcji przekazanej w zapytaniu przez klienta, która klient później wywoła za pomocą punktu końcowego
- *url* — przyjmujący nazwę punktu końcowego przekazanego w zapytaniu przez klienta, który zostanie przez niego wywołany za pomocą protokołu HTTP, aby uruchomić funkcje o nazwie podanej w parametrze *functionName*

Wykonanie zleconego przez klienta żądania uruchomienia aplikacji polega na zmianie w bazie danych w obiekcie aplikacji w tabeli *Application* parametru *up* na *true*, a następnie wykonaniu żądań przez moduł back-end do komponentu API server klastra Kubernetes za pomocą Rest API w celu stworzenia trzech komponentów:

- zasobu typu Deployment — tworzącego zasób typu Pod z aplikacją kliencką, której obraz kontenera pobierany jest z repozytorium ECR. Nazwa tworzonego zasobu typu Deployment oraz typu Pod ma w sobie nazwę aplikacji stworzonej przez klienta. Obraz kontenera posiada zmienną środowiskową o nazwie *PACKAGE_JSON*, do której przypisywana jest wartość

parametru *packageJson* aplikacji zapisanego w bazie danych w tabeli *Application*. W tym parametrze klient zdefiniował zewnętrzne biblioteki jakie mają być zainstalowane w kontenerze. Biblioteki te są instalowane w momencie inicjalizacji kontenera.

- zasobu typu Service — kierującego ruch do zasobu typu Pod z aplikacją kliencką.
- zasobu typu Ingress — kierującego żądanie klienta wykonane pod adres poddomeny strony *custom-serverless.com*, której nazwą poddomeny jest nazwa aplikacji, do zasobu typu Service, a w efekcie zasobu typu Pod z aplikacją kliencką.

Przykładowy przepływ kroków dający klientowi możliwość uzyskania rezultatu wykonania funkcji przetwarzania bezserwerowego za pomocą aplikacji klienckiej wygląda następująco:

1. Stworzenie aplikacji o nazwie "app1".
2. Stworzenie funkcji o nazwie "funkcja1".
3. Stworzenie punktu końcowego o parametrze url o treści "endpoint1" oraz o parametrze *functionName* przyjmującym wartość "funkcja1"
4. Uruchomienie aplikacji "app1".
5. Wykonanie zapytania na adres "app1.custom-serverless.com/endpoint1" wywołującego funkcję o nazwie "funkcja1" i zwracającego w odpowiedzi jej rezultat.

W przypadku awarii i restartu klastra Kubernetes, przy ponownym starcie modułu back-end wszystkie aplikacje posiadające parametr *up* o wartości *true* zostaną ponownie uruchomione.

Zatrzymanie aplikacji polega na usunięciu zasobów typu Deployment, Service oraz Ingress z których składa się aplikacja.

Usunięcie aplikacji polega na jej zatrzymaniu jeśli jest uruchomiona, a więc również usunięciu zasobów typu Deployment, Service oraz Ingress z których składa się aplikacja, a następnie usunięciu w bazie danych obiektu z aplikacją w tabeli *Application*.

Testowanie funkcji aplikacji polega najpierw na wykonaniu przez klienta zapytania do aplikacji modułu back-end pod adres *www.custom-serverless.com/api/runtime* z parametrem będącym nazwą aplikacji, do której testowana funkcja należy. Po otrzymaniu zapytania, aplikacja modułu back-end wysyła żądanie do komponentu API serwer klastra Kubernetes w celu sprawdzenia, czy aplikacja testująca należąca do modułu testującego o nazwie takiej jak aplikacja, do której należy testowana funkcja jest już uruchomiona. Jeśli nie, aplikacja modułu back-end tworzy zasób typu Service oraz zasób typu Deployment, z których składa się aplikacja testująca oraz zwraca klientowi odpowiedź, z wiadomością, że aplikacja testująca nie jest jeszcze gotowa. W takiej sytuacji klient otwiera połączenie websocket z aplikacją modułu back-end w oczekiwaniu na uruchomienie aplikacji testującej. Aplikacja modułu back-end po otwarciu połączenia websocket tworzy obiekt typu *informer* udostępniony przez Kubernetes pozwalający nasłuchiwać na stan wybranych zasobów swojego klastra. Aplikacja modułu back-end nasłuchuje na stan zasobu typu Pod tworzzonego przez zasób typu Deployment, aż w metadanych Poda stan jego cyklu życia zmieni wartość na "Running", co oznacza, że aplikacja testująca została uruchomiona w klastrze. Po wykryciu stanu "Running" aplikacja modułu back-end za pomocą połączenia websocket wysyła klientowi informację, że aplikacja testująca została uruchomiona. W takiej sytuacji klient zamknie połączenie websocket i wysyła zapytanie przetestowania aplikacji wraz z kodem funkcji, który pragnie przetestować pod adres *www.custom-serverless.com/api/test*. Aplikacja modułu back-end przekierowuje to żądanie do modułu testującego do aplikacji testującej, a otrzymany rezultat zwraca klientowi.

Przy kolejnym żądaniu klienta wykonanym pod adres `www.custom-serverless.com/api/runtime`, moduł back-end zwróci klientowi rezultat, informujący, że aplikacja testowa już jest uruchomiona. Dzięki temu klient od razu po otrzymaniu tego rezultatu wysyła żądanie pod adres `www.custom-serverless.com/api/test` w celu przetestowania funkcji.

Czas trwania aplikacji testującej jest ograniczony i określony w definicji zasobu typu Service w parametrze `metadata.labels.expire` i przyjmuje on wartość znacznika czasowego (ang. timestamp), wynoszącą znak czasu w chwili obecnej + 5 minut. Po uruchomieniu aplikacji testującej przy każdym kolejnym jej wywołaniu, wartość parametru `metadata.labels.expire` zmienia wartość na znacznik czasowego w chwili nowego wywołania aplikacji testującej + 5 minut. Jeśli natomiast aplikacja testująca nie była wywoływana od ponad 5 minut, wykryje to wywoływaną okresowo co 5 minut, w aplikacji modułu back-end, funkcja typu `cron`, pytająca klaster Kubernetes o uruchomione aplikacje testowe niewykorzystywane od ponad 5 minut, a następnie usunie zasób typu Service oraz zasób typu Deployment, z których składa się aplikacja testująca.

Moduł aplikacji klienckich

Moduł aplikacji klienckich znajduje się na Rysunku 4.1 w klastrze Kubernetes w przestrzeni nazw `custom-serverless-apps`, a w jego skład wchodzą następujące zasoby Kubernetes:

- `apps-ecr-secret` — zasób typu Secret
- `mongodb-secret` — zasób typu Secret
- `{appName}-service` — zasoby typu Service
- `{appName}-deployment` — zasoby typu Deployment
- `{appName}-pod` — zasoby typu Pod
- `{appName}-ingress` — zasoby typu Ingress
- `custom-serverless-apps-role` — zasób typu Role
- `custom-serverless-apps-rolebinding` — zasób typu RoleBinding

Moduł ten składa się z wielu aplikacji klienckich, a każda z nich składa się z zasobów `{appName}-ingress`, `{appName}-service`, `{appName}-deployment`, `{appName}-pod`, gdzie `appName` przyjmuje wartość nazwy aplikacji stworzonej przez klienta. Komponenty te tworzone są przez moduł back-end na żądanie przy pomocy GUI strony `custom-serverless.com` przez klienta. Aplikacja modułu back-end ma uprawnienia do tworzenia zasobów w przestrzeni nazw `custom-serverless-apps` dzięki zasobowi `custom-serverless-apps-role`, opisujące uprawnienia do tworzenia i usuwania wymienionych wyżej komponentów, a także dzięki zasobowi `custom-serverless-apps-rolebinding`, który umożliwia powiązanie stworzonej roli z zasobem `backend-service-account`, znajdującym się w module back-end.

Zapytanie wysyłane na adres aplikacji klienckiej (będącej poddomeną strony internetowej `custom-serverless.com`), czyli na adres `{appName}.custom-serverless.com`, gdzie `appName` jest nazwą aplikacji klienckiej, po dotarciu do komponentu `Nginx Ingress Controller` trafia dalej do `{appName}-ingress`. W dalszej kolejności zapytanie trafia do `{appName}-service`, a następnie do Poda o nazwie `{appName}-pod`. Pod ten zawiera w swoim kontenerze aplikację kliencką, napisaną w języku JavaScript we frameworku Express.js w środowisku uruchomieniowym Node.js. Jej obraz jest pobierany

z repozytorium ECR za pomocą sekretu *apps-ecr-secret*. Aplikacja ta komunikuje się z bazą danych, z wykorzystaniem zasobu *mongodb-secret*.

Klient może sprawdzić, czy aplikacja kliencka została już uruchomiona za pomocą wykonania zapytania do punktu końcowego `{appName}.custom-serverless.com/up`. Jeśli zwróci on rezultat w formacie JSON w postaci `{"status": "up"}` oznacza to, że aplikacja faktycznie jest uruchomiona.

Za pomocą zapytania wysłanego pod adres `{appName}.custom-serverless.com/edge` klient ma możliwość pobrać kody funkcji, których nazwy podał w ciele zapytania. W odpowiedzi, oprócz kodów funkcji oraz informacji, czy są one idempotentne, otrzymuje on też kod funkcji pozwalający wykonać zagnieźdzone ich wywołania za pomocą funkcji *call*, takie jak na Listingu 12 w linii 2. Po wykonaniu za pomocą kodu dostarczonego przez aplikację kliencką żądanego kodu funkcji, klient dostaje odpowiedź, której przykładowa postać została przedstawiona na Listingu 13.

LISTING 13: Przykładowy rezultat przetwarzania na krawędzi

```

1  {
2      "inner-function": {
3          "eyJhIjo0LCJiIjo1fQ==": {
4              "result": 9
5          }
6      }
7  }
```

Na Listingu 13 przedstawiono rezultat wykonania przetwarzania na krawędzi przez klienta funkcji o nazwie "inner-function" z argumentami wejściowymi, których obiekt JSON w postaci `{"args": {"a": 4, "b": 5}}` zmieniony do formatu *base64* ma postać "eyJhIjo0LCJiIjo1fQ==" (linia 3). Dla takich argumentów wejściowych funkcja "inner-function" zwróciła rezultat `{"result": 9}`, który może zostać wykorzystany przy wykonaniu w chmurze funkcji przetwarzania bezserwerowego. Aplikacja kliencka w chmurze wykorzysta ten rezultat zamiast samodzielnie wykonania funkcji "inner-function" w przypadku, gdy zostanie poproszona o wykonanie tej funkcji dla tych samych argumentów wejściowych, z jakimi wykonał ją klient.

Klient, aby wywołać funkcję przetwarzania bezserwerowego w chmurze, wykonuje żądanie do punktu końcowego adresu aplikacji. Przykładowo, w celu wykonania funkcji "outer-function", możliwej do wykonania przy pomocy punktu końcowego "outer" dla aplikacji o nazwie "app1" należy podać adres: "app1.custom-serverless.com/outer". Przykładowe ciało zapytania wysłane pod ten adres zostało przedstawione na Listingu 14.

LISTING 14: Przykładowe ciało zapytania funkcji bezserwerowej

```

1  {
2      "args": {
3          "a": 4,
4          "b": 5,
5          "c": 6
6      },
7      "edgeResults": {
8          "inner-function": {
9              "eyJhIjo0LCJiIjo1fQ==": {
10                  "result": 9
11              }
12          }
13      }
14  }
```

```

13     },
14     "cache": {
15       "other-function": {
16         "eyJhIjo0LCJiIjo1fQ==": {
17           "result": 10
18         }
19       }
20     }
21   }

```

Ciało zapytania, którego przykład został przedstawiony na Listingu 14 może zawierać trzy parametry:

- *args* — parametr zawierający argumenty wejściowe do wykonania funkcji ”outer-function”
- *edgeResults* — parametr zawierający rezultat przetwarzania na krawędzi wykonanego przez klienta, przykład zawartości tego parametru został przedstawiony na Listingu 13.
- *cache* — parametr o tej samej strukturze, co *edgeResults*, przedstawiony na Listingu 13, zawierający wyniki funkcji idempotentnych, zwróconych w wyniku ostatniego wywołania funkcji ”outer-function” przy pomocy żądania wysłanego do aplikacji klienckiej.

Aplikacja kliencka podczas inicjalizacji pobiera z bazy MongoDB wszystkie funkcje oraz punkty końcowe zdefiniowane przez programistę dla uruchamianej aplikacji. Po otrzymaniu zapytania, którego przykład został przedstawiony na Listingu 14, aplikacja uruchamia wywołaną funkcję według następującego schematu:

1. W pierwszym kroku sprawdza, czy wynik tej funkcji dla podanych w parametrze *args* argumentów wejściowych znajduje się już w obiekcie przechowywanym w parametrze *edgeResults* — jeśli tak, to zwraca klientowi w parametrze *result* odpowiedzi w formacie JSON, wynik znajdujący się w parametrze *edgeResults* dla zdekodowanych z formatu base64 do formatu JSON argumentów wejściowych
2. Jeśli *edgeResults* nie zawiera rezultatu funkcji, a wywoływana funkcja jest idempotentna, to aplikacja sprawdza, czy pamięć podrzczna w parametrze *cache*, posiada wynik funkcji dla podanych argumentów wejściowych — jeśli tak to zwraca wynik w parametrze *result* odpowiedzi wraz ze zwróceniem w parametrze *cache* pamięci podrzcznej dla klienta.
3. Jeśli funkcja nie została wykonana na krawędzi przez klienta, nie jest idempotentna lub jest idempotentna, ale jej wyniku dla podanych argumentów wejściowych nie ma w pamięci podrzcznej, to aplikacja wykonuje obliczenia, a więc kod żądanej do wykonania funkcji i zwraca klientowi w parametrze *result* wynik. Jeśli funkcja jest idempotentna, to dodatkowo dodaje ten sam wynik do zwracanej w parametrze *cache* pamięci podrzcznej.

Te same trzy przypadki są analizowane i przetwarzane przez aplikację kliencką w przypadku napotkania w kodzie funkcji metody *call* wywołującej inną funkcję — najpierw aplikacja kliencka sprawdza, czy jej wynik jest zdefiniowany w parametrze *edgeResults*, jeśli tak to wynik funkcji *call* jest od razu zwracany. Jeśli nie, sprawdzane jest czy funkcja jest idempotentna i czy jest w cache, a jeśli ten przypadek też nie jest spełniony to aplikacja wykonuje kody wywołanej za pomocą *call* funkcji, zwraca jej rezultat, a jeśli była ona funkcją idempotentną to dodaje jej rezultat do pamięci podrzcznej w parametrze *cache*.

Moduł testujący

Moduł testujący znajduje się na Rysunku 4.1 w klastrze Kubernetes w przestrzeni nazw *custom-serverless-runtime*, a w jego skład wchodzą następujące zasoby Kubernetes:

- *runtime-ecr-secret* — zasób typu Secret
- *mongodb-secret* — zasób typu Secret
- *{appName}-runtime-service* — zasoby typu Service
- *{appName}-runtime-deployment* — zasoby typu Deployment
- *{appName}-runtime-pod* — zasoby typu Pod
- *custom-serverless-runtime-role* — zasób typu Role
- *custom-serverless-runtime-rolebinding* — zasób typu RoleBinding

Moduł ten składa się z wielu aplikacji testujących, a każda z nich składa się z zasobów *{appName}-runtime-service*, *{appName}-runtime-deployment*, *{appName}-runtime-pod*, gdzie *appName* przyjmuje wartość nazwy aplikacji, której funkcja jest testowana przez klienta. Komponenty te tworzone są przez moduł back-end na żądanie klienta związane z przetestowaniem funkcji, przy pomocy GUI strony *custom-serverless.com*. Aplikacja modułu back-end ma uprawnienia do tworzenia zasobów w przestrzeni nazw *custom-serverless-apps* dzięki zasobowi *custom-serverless-runtime-role* opisującemu uprawnienia do tworzenia i usuwania wymienionych wyżej komponentów, a także dzięki zasobowi *custom-serverless-runtime-rolebinding* umożliwiającemu powiązanie stworzonej roli z zasobem *backend-service-account*, znajdującym się w module back-end.

Po stworzeniu aplikacji testującej przez moduł back-end, gdy klient wysyła żądanego przetestowania funkcji, jest ono przekazywane z aplikacji modułu back-end do zasobu *{appName}-runtime-service*, a następnie do Poda o nazwie *{appName}-runtime-pod*, który zawiera w swoim kontenerze aplikację testującą, napisaną w języku JavaScript we frameworku Express.js. Jej obraz jest pobierany z repozytorium ECR za pomocą sekretu *runtime-ecr-secret*. Aplikacja ta komunikuje się z bazą danych, dzięki zasobowi *mongodb-secret*.

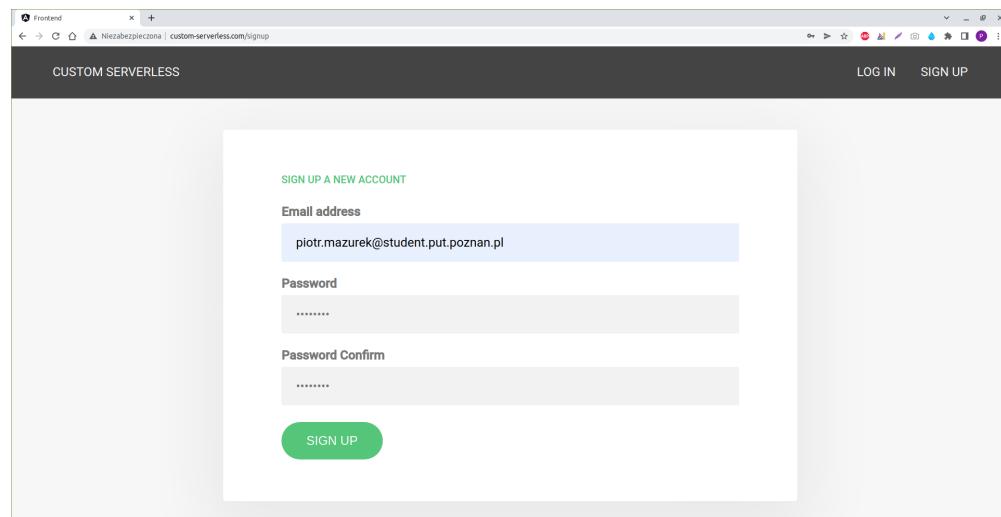
Po otrzymaniu żądania klienta, aplikacja pobiera z bazy MongoDB listę funkcji należących do aplikacji, wykonuje kod funkcji przesłany przez klienta w zapytaniu, a przy napotkaniu metody *call* wywołującej inną funkcję stosuje ten sam scenariusz 4.2.2, co w przypadku przetwarzania funkcji *call* za pomocą aplikacji klienckiej: najpierw sprawdzana jest zawartość parametru *edgeResults*, następnie *cache*, a w przypadku niespełnienia warunków z nimi związanych, aplikacja testująca wykonuje kody funkcji wywołane metodą *call* i zwraca jej wynik w celu dalszego przetwarzania. Po wykonaniu kodów funkcji, wynik jej wraz z ewentualną utworzoną pamięcią podręczną w parametrze *cache* jest zwracany aplikacji modułu back-end, która przekazuje wynik klientowi i wyświetla go w przeglądarce za pomocą kodów aplikacji modułu front-end.

Rozdział 5

Przykładowe działanie

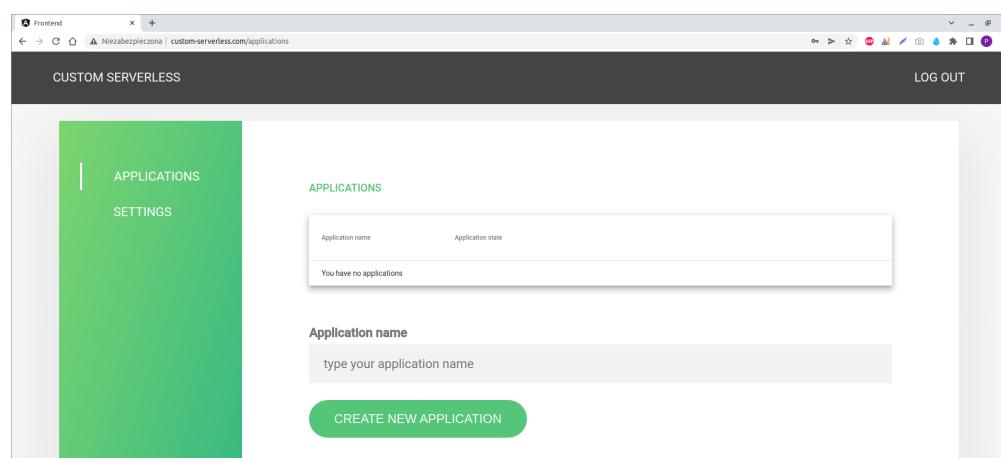
5.1 Rejestracja klienta

W celu korzystania z systemu, klient ma obowiązek rejestracji poprzez wypełnienie formularza rejestracyjnego.



RYSUNEK 5.1: Rejestracja klienta

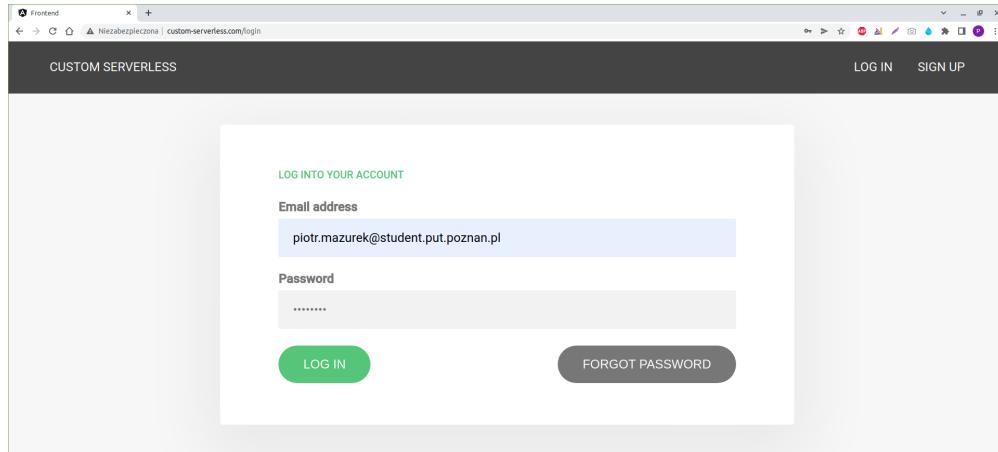
Formularz rejestracyjny, w którym klient wypełnia pola: adres email, hasło oraz potwierdzenie hasła, jest dostępny na stronie www.custom-serverless.com/signup.



RYSUNEK 5.2: Uwierzytelnienie po rejestracji

Po rejestracji klient staje się użytkownikiem uwierzytelnionym i zostaje przekierowany pod adres www.custom-serverless.com/applications, w którym magląd w stworzone aplikacje. Początkowo klient nie ma żadnej stworzonej aplikacji. Aby się wylogować, klient wybiera zakładkę "LOG OUT" znajdująca się w prawym górnym rogu ekranu. Po jej naciśnięciu zostaje wylogowany z systemu i przekierowany pod adres www.custom-serverless.com/login w celu ponownego uwierzytelnienia.

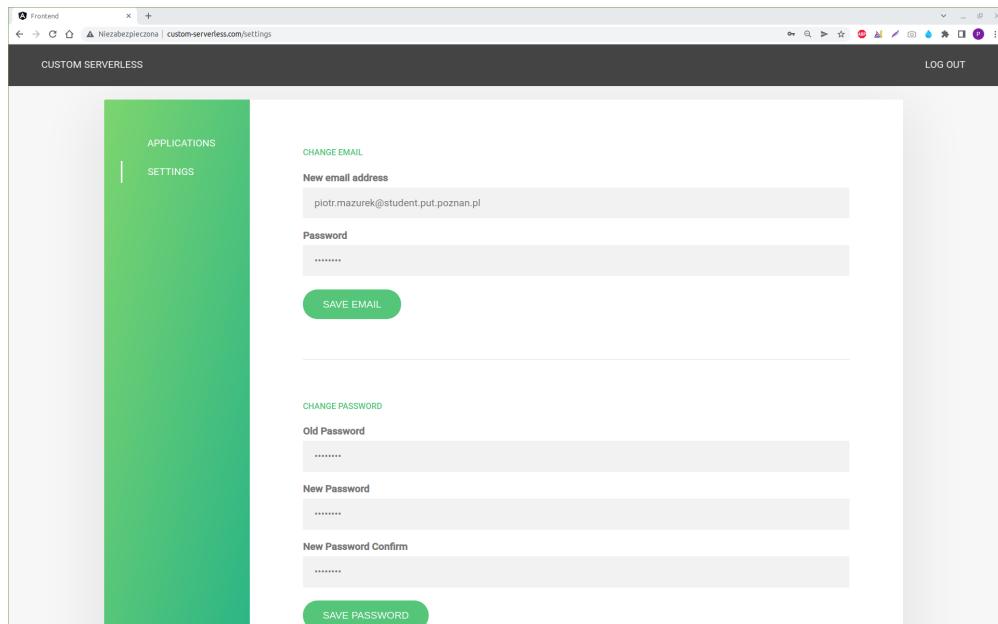
5.2 Logowanie klienta



RYSUNEK 5.3: Logowanie klienta

W celu zalogowania do systemu, klient zostaje przekierowany pod adres www.custom-serverless.com/login, a następnie wypełnia formularz uwierzytelniający, wpisując adres email oraz hasło.

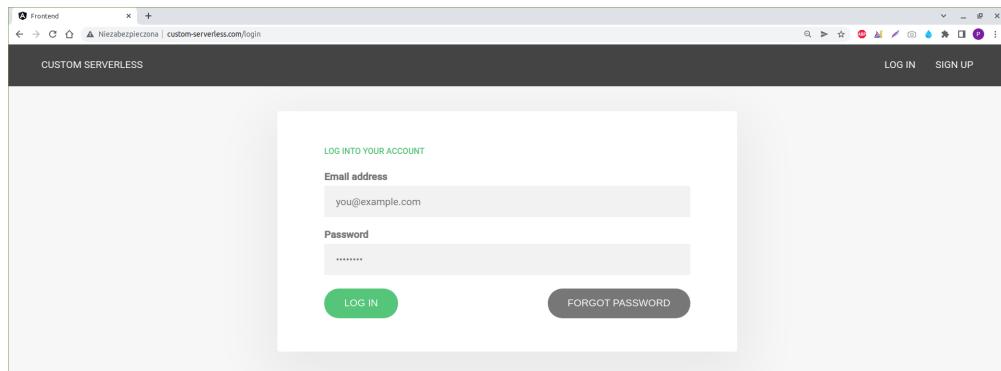
5.3 Zmiana ustawień konta



RYSUNEK 5.4: Zmiana ustawień konta

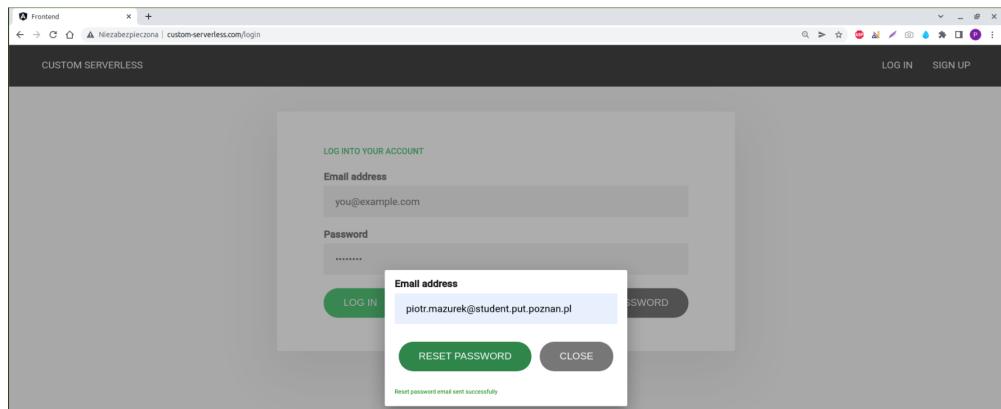
W celu zmiany ustawień konta, takich jak zmiana adresu email lub hasła, klient wybiera zakładkę "SETTINGS" znajdująca się po lewej stronie ekranu lub jest przekierowany pod adres www.custom-serverless.com/settings, a następnie wypełnia formularze zmiany danych dotyczących konta.

5.4 Resetowanie hasła



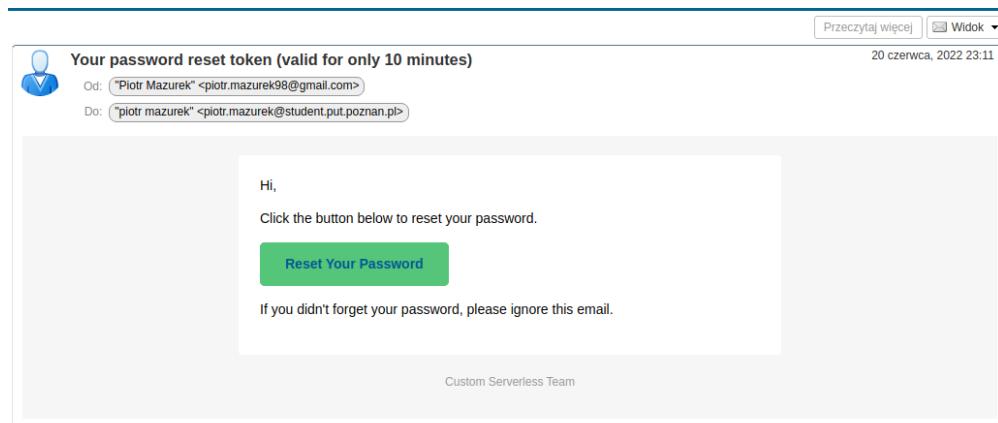
RYSUNEK 5.5: Początek procesu resetowania hasła

W przypadku zapomnienia hasła przez klienta ma on możliwość jego zresetowania. Aby to uczynić, w pierwszym kroku naciska on przycisk "FORGET PASSWORD".



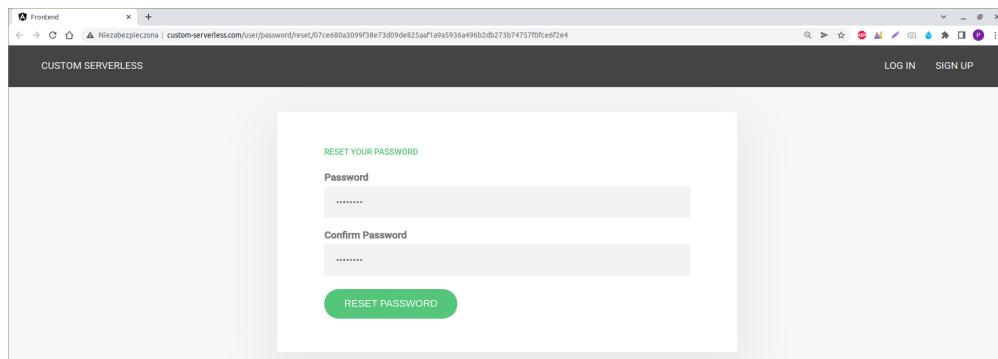
RYSUNEK 5.6: Prośba o wysłanie adresu email z linkiem resetującym hasło

Następnie podaje on swój adres email i naciska przycisk "RESET PASSWORD", a w wyniku dostaje informację, że na jego adres email został wysłany link resetujący hasło.



RYSUNEK 5.7: Treść wiadomości email

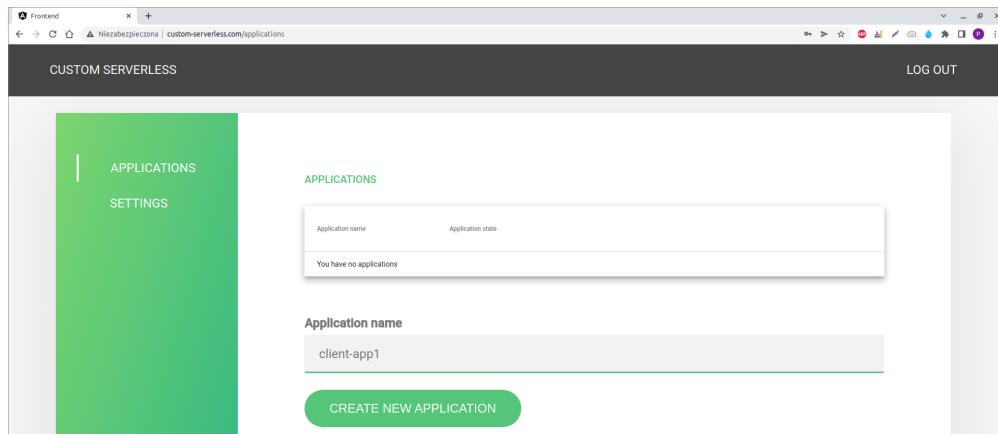
Po wejściu na swoją skrzynkę pocztową, klient widzi pokazaną na Rysunku 5.7 treść wiadomości email. Naciskając przycisk "Reset Your Password", klient zostanie przekierowany do strony zawierającej w swoim adresie URL token resetujący, pozwalający zresetować hasło. Temat wiadomości, informuje klienta, że token ten jest ważny przez 10 minut — a zatem w ciągu 10 minut od momentu naciśnięcia przycisku "FORGET PASSWORD" klient ma możliwość zresetowania hasła przy pomocy tego tokenu.



RYSUNEK 5.8: Formularz resetu hasła

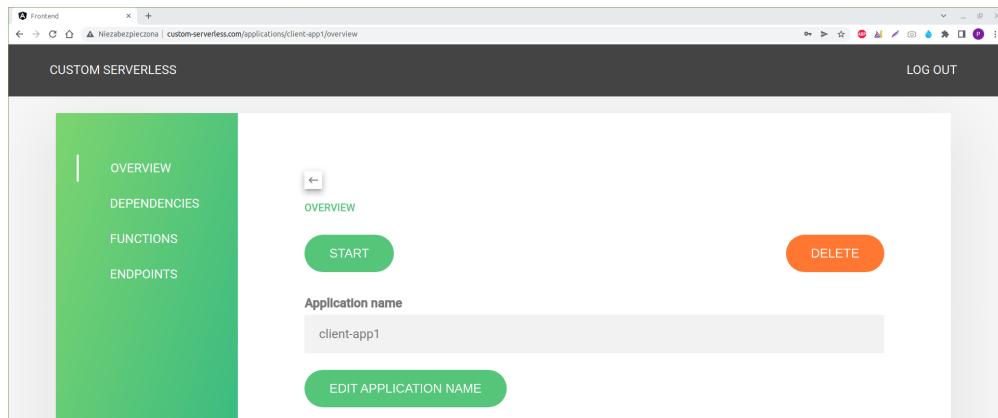
Po naciśnięciu przycisku "Reset Your Password" klient zostanie przeniesiony do formularza pozwalającego zresetować hasło. Po wypełnieniu pól hasła oraz potwierdzenia hasła oraz naciśnięciu przycisku "RESET PASSWORD" klient wysyła zapytanie zawierające w swoim ciele wartości tych pól oraz token resetujący, znajdujący się w adresie URL strony, na której udostępniony jest formularz. Na Rysunku 5.8 token ten ma wartość "07ce680a3099f38e73d09de825aaaf1a9a5936a496b2db273b74757f0fce6f2e4". W odpowiedzi od modułu back-end, klient dostaje informację o pomyślnym resecie hasła i automatycznie zostaje uwierzytelniony i przeniesiony pod adres www.custom-serverless.com/applications.

5.5 Tworzenie aplikacji



RYSUNEK 5.9: Formularz tworzenia aplikacji

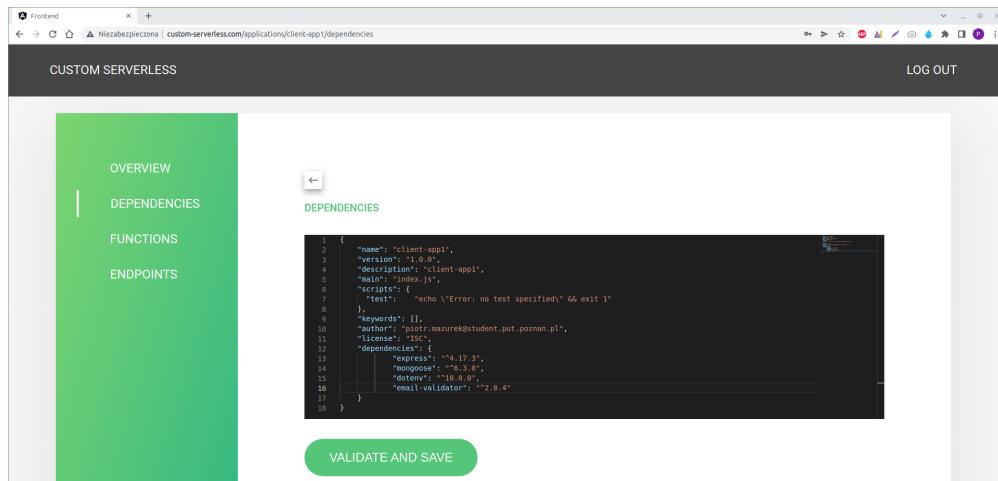
Klient ma możliwość stworzenia aplikacji za pomocą formularza dostępnego na stronie www.custom-serverless.com/applications, w którym podaje nazwę tworzonej aplikacji — w tym przypadku nowa aplikacja posiada nazwę "client-app1".



RYSUNEK 5.10: Sekcja widoku ogólnego aplikacji

Po utworzeniu aplikacji klient zostaje przeniesiony do sekcji jej widoku ogólnego (ang. *overview*), dostępnego pod adresem www.custom-serverless.com/applications/{appName}/overview, gdzie w przykładzie na Rysunku 5.10 *appName* przyjmuje wartość "client-app1". Korzystając z pozostałych zakładek po lewej stronie, klient może zmienić zależności aplikacji — zakładka *Dependencies*, dodać funkcje — zakładka *Functions* oraz punkty końcowe — zakładka *Endpoints*.

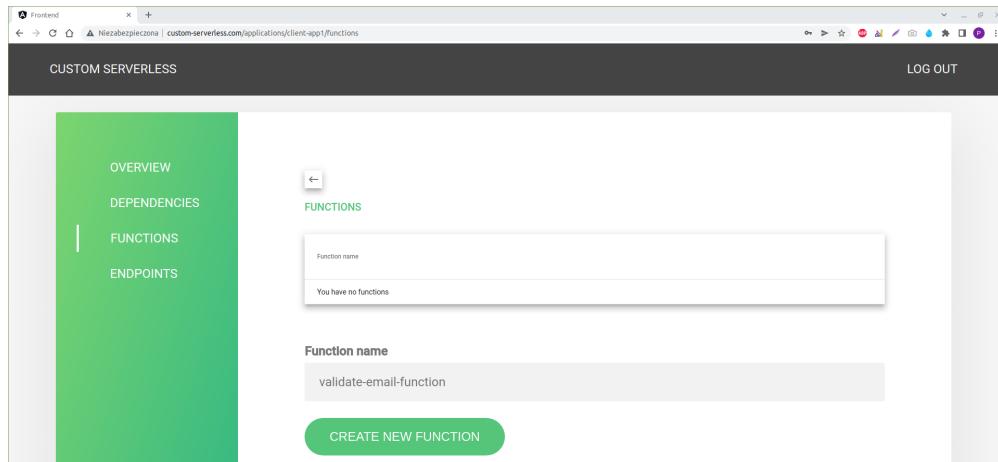
5.6 Dodanie zależności



RYSUNEK 5.11: Dodanie zależności

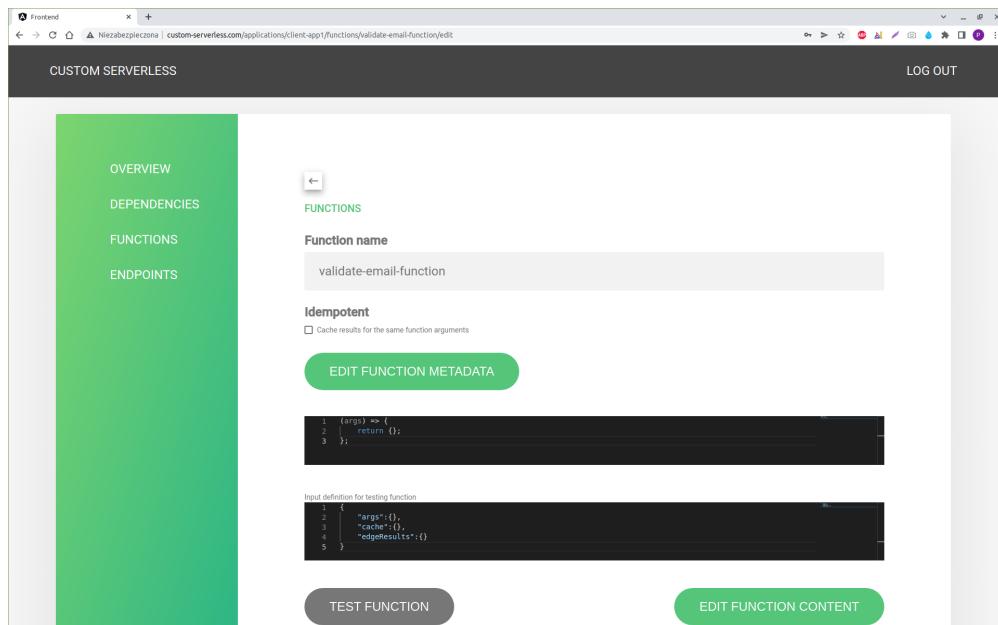
Przechodząc do zakładki *Dependencies*, klient ma możliwość dodania zależności zmieniając zawartość pliku package.json za pomocą interaktywnego edytora. Przykładowo, na Rysunku 5.11 pokazano dodanie przez klienta (w linii 16 edytora) zależności *email-validator* w wersji *2.0.4*.

5.7 Tworzenie funkcji



RYSUNEK 5.12: Tworzenie funkcji

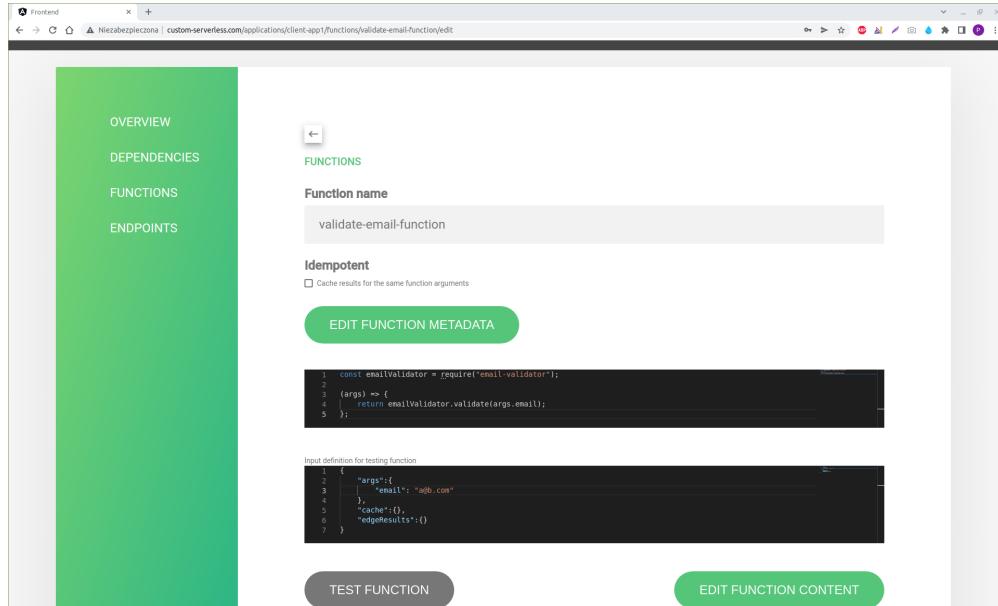
Przechodząc do zakładki *Functions* klient ma możliwość stworzenia funkcji, korzystając z zawartego na stronie formularza. W przykładzie przedstawionym na Rysunku 5.12 tworzy on funkcję o nazwie ”validate-email-function”.



RYSUNEK 5.13: Ekran edycji funkcji

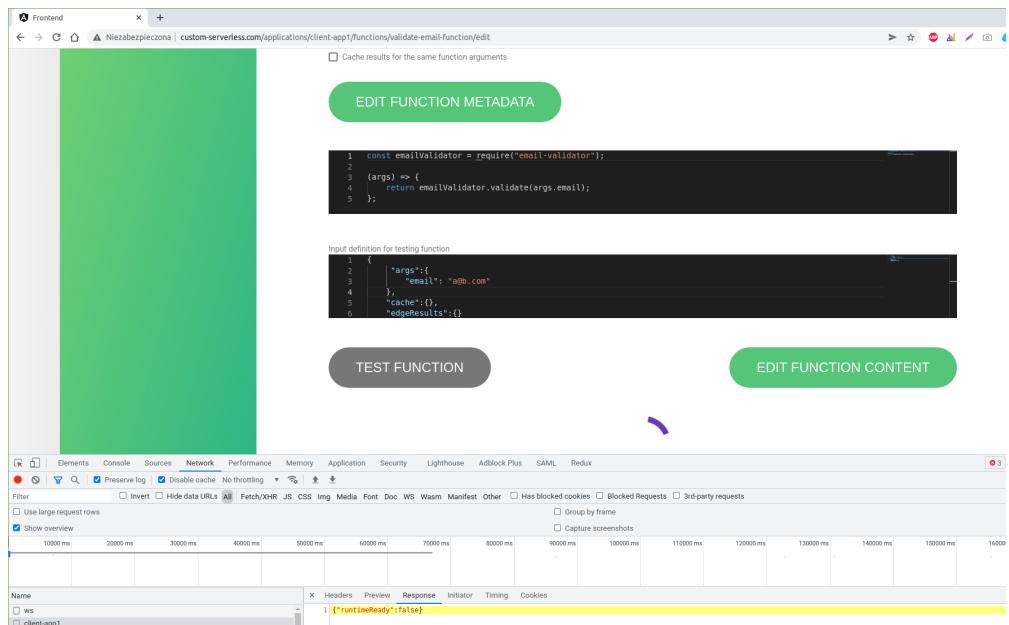
Po stworzeniu funkcji klient zostaje przeniesiony do strony edycji funkcji, której domyślny widok tuż po stworzeniu został przedstawiony na Rysunku 5.13.

5.8 Testowanie funkcji



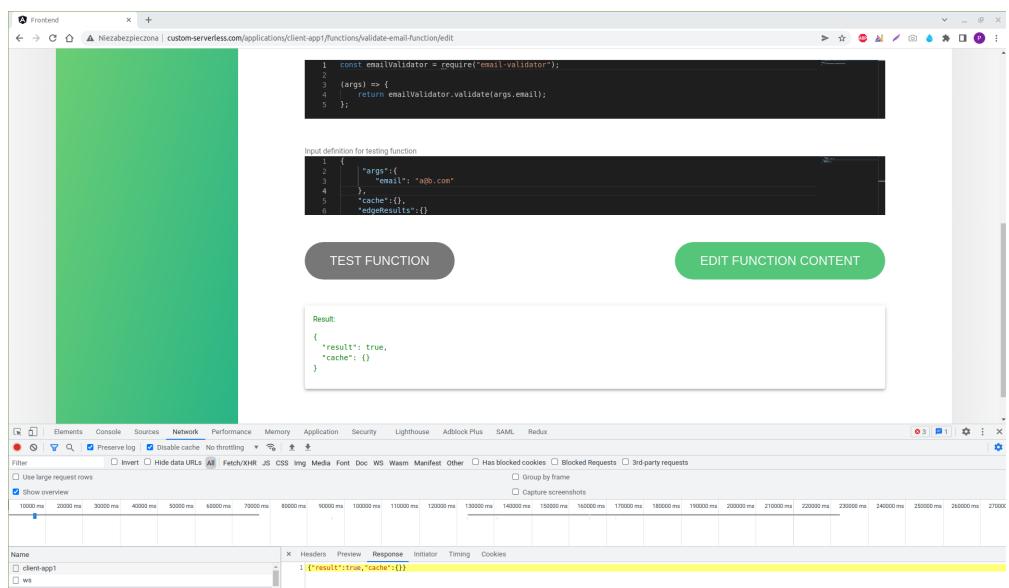
RYSUNEK 5.14: Przykład testowanej funkcji

W celu przetestowania funkcji, której przykład pokazany na Rysunku 5.14 korzysta ze zdefiniowanej wcześniej, na Rysunku 5.11, biblioteki *email-validator*, klient definiuje argumenty wejściowe do testowanej funkcji w dolnym edytorze w polu args, a następnie naciska przycisk "TEST FUNCTION".



RYSUNEK 5.15: Uruchomienie testowania funkcji

Na Rysunku 5.15 ukazano uruchomienie testowania funkcji. Klientowi wyświetlił się widok oczekiwania na jej wynik, a w zakładce *Sieć* narzędzia *DevTools* pokazano, że najpierw moduł back-end został odpytany, czy aplikacja testująca już jest uruchomiona w klastrze i po otrzymaniu odpowiedzi przeczącej — {“runtimeReady”: false}, klient nawiązał połączenie websocket w oczekiwaniu na zakończenie jej inicjalizacji.

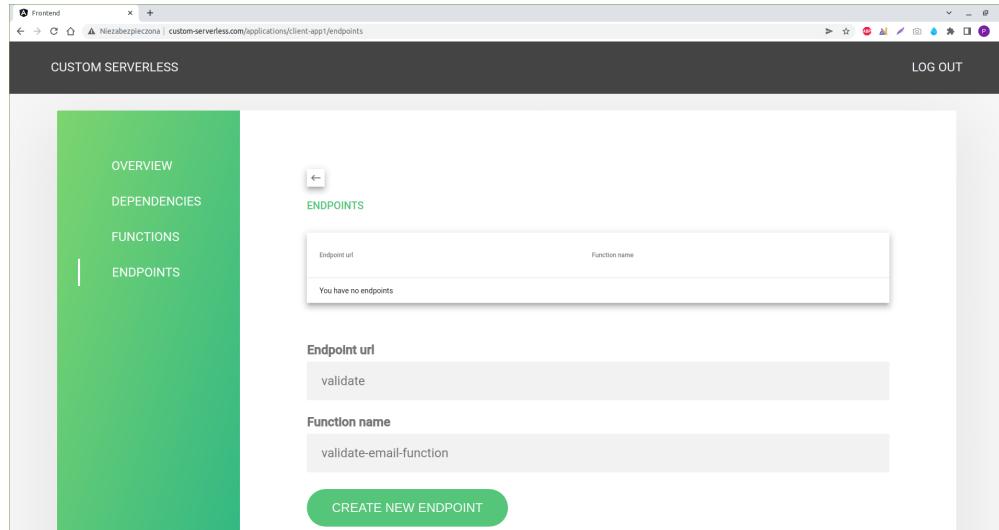


RYSUNEK 5.16: Wynik testowania funkcji

Na Rysunku 5.16 ukazano zapytanie klienta przesiane pod adres www.custom-serverless.com/api/test, związane z przetestowaniem kodu funkcji po uzyskaniu informacji za pomocą protokołu WebSocket, że aplikacja testująca jest już uruchomiona. Ukażano również rezultat przetwarzania funkcji informujący, że podany w argumencie wejściowym adres email jest zapisany w poprawnym formacie. Został on zaprezentowany zarówno w postaci graficznej na stronie internetowej, wyświetlony czcionką o kolorze zielonym, jak i w narzędziu *DevTools*.

w postaci odpowiedzi HTTP zwróconej przez moduł testujący, a następnie przekazanej dalej przez moduł back-end.

5.9 Tworzenie punktu końcowego

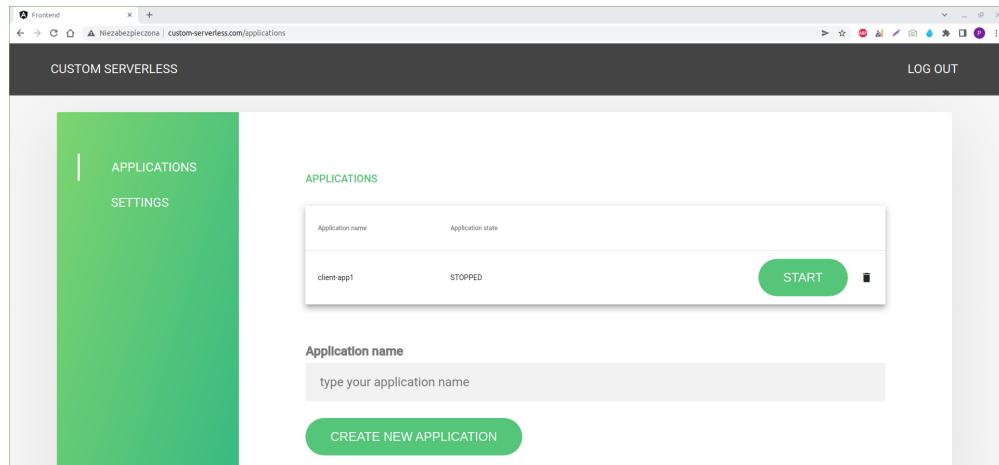


RYSUNEK 5.17: Tworzenie punktu końcowego

Przechodząc do zakładki *Endpoints* klient ma możliwość stworzenia punktu końcowego (ang. endpoint) korzystając z zawartego na stronie formularza. W przykładzie przedstawionym na Rysunku 5.17, tworzy on punkt końcowy o adresie "validate", po przekierowaniu na który klient będzie miał możliwość wywołania zdefiniowanej wcześniej funkcji o nazwie "validate-email-function".

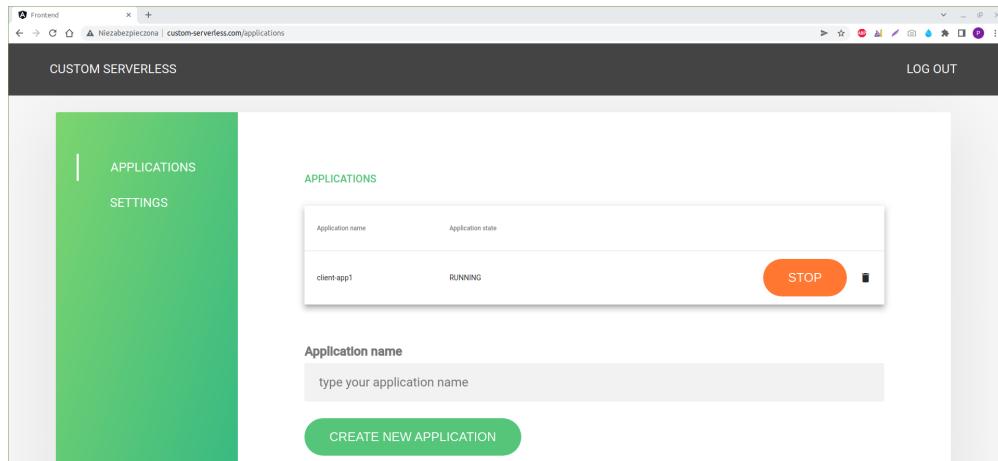
5.10 Uruchomienie aplikacji

Klient ma możliwość uruchomienia aplikacji za pomocą strony udostępnianej przez zakładkę *Overview* pokazaną na Rysunku 5.10, lub korzystając ze strony www.custom-serverless.com/applications, której widok z utworzoną wcześniej aplikacją o nazwie "client-app1" został zaprezentowany na poniższym Rysunku 5.18.



RYSUNEK 5.18: Widok listy aplikacji

Na zaprezentowanej na Rysunku 5.18 liście aplikacji, po naciśnięciu przycisku "START", w wierszu z aplikacją o nazwie "client-app1", aplikacja ta zostaje uruchomiona, co widoczne jest poprzez zmianę etykiety i koloru przycisku "START" na "STOP", co zaprezentowano na Rysunku 5.19.



RYSUNEK 5.19: Widok listy aplikacji z uruchomioną aplikacją "client-app1"

Klient może zweryfikować, czy aplikacja kliencka "client-app1" została już uruchomiona korzystając z adresu "client-app1.custom-serverless.com/up", co zaprezentowano na Rysunku 5.20.



RYSUNEK 5.20: Weryfikacja stanu aplikacji "client-app1"

5.11 Żądanie wykonania funkcji uruchomionej aplikacji

Klient może wywołać zdefiniowaną na Rysunku 5.14 funkcję za pomocą adresu "validate" zdefiniowanego na Rysunku 5.17, wykonując zapytanie HTTP, na przykład przy użyciu narzędzia *Postman*, co zaprezentowano na Rysunku 5.21.

RYSUNEK 5.21: Przykład wywołania punktu końcowego aplikacji "client-app1"

Po wysłaniu żądania HTTP na adres "client-app1.custom-serverless.com/validate", wywołując tę samą funkcję i korzystając z tych samych argumentów wejściowych, co w przypadku testowania funkcji zaprezentowanego na Rysunku 5.16 otrzymano te same rezultaty.

5.12 Wykorzystanie rezultatów przetwarzania na krawędzi

Poniższy przykład przedstawiony na Listingu 17 ilustruje wykonanie początkowych obliczeń, a więc funkcji o nazwie "inner-function", przez klienta lokalnie na urządzeniu końcowym, a więc na krawędzi, a następnie wykorzystanie rezultatów jej obliczeń do wykonania drugiej funkcji o nazwie "outer-function" w chmurze. Funkcja "inner function" i ma postać przedstawioną na Listingu 15, z kolei funkcja "outer-function" ją wywołuje i korzysta z rezultatów jej obliczeń, a jej kod został przedstawiony na Listingu 16. Obie funkcje są idempotentne.

LISTING 15: Kod funkcji "inner-function"

```
1  async (args) => {
2      return {"result": args.a + args.b};
3  };
```

LISTING 16: Kod funkcji "outer-function"

```
1  async (args) => {
2      let output = await call("inner-function", {"abreturn {out: output.result * args.c};
4  };
```

costam

LISTING 17: Przykładowy program kliencki realizujący część obliczeń na krawędzi

```

1  const axios = require("axios");
2
3  (async function edge() {
4      let edgeArgs = {
5          "a": 4,
6          "b": 5,
7      };
8
9      let serverlessArgs = {
10         "a": 4,
11         "b": 5,
12         "c": 6
13     };
14
15     let request = {
16         "functions": ["inner-function"]
17     };
18
19     let edgeResponse = await axios.post(
20         'http://test-app.custom-serverless.com/edge',
21         request
22     );
23     let runEdgeFunction = eval(edgeResponse.data.runEdgeFunction);
24     let edgeResults = await runEdgeFunction(
25         "inner-function",
26         edgeResponse.data.functions, edgeArgs
27     );
28     console.log(JSON.stringify(edgeResults));
29     let result = await axios.post(
30         'http://test-app.custom-serverless.com/outer',
31         {
32             args: serverlessArgs,
33             edgeResults: edgeResults
34         }
35     );
36     console.log(JSON.stringify(result.data));
37 })();

```

Powyższy Listing 17 obrazuje najpierw deklarację skorzystania przez klienta z biblioteki *axios*, w celu wykonania żądań HTTP (linia 1). Następnie klient w celu lokalnego wykonania obliczeń funkcji "inner-function" pobiera jej kody (linia 19). W kolejnym kroku (linia 23) kompliuje kody funkcji o nazwie "runEdgeFunction", którą otrzymał od aplikacji klienckiej w celu lokalnego wykonania funkcji, o których kod poprosił (poprosił o kod funkcji "inner-function"). W następnym kroku wykonuje kody funkcji "inner-function" za pomocą funkcji "runEdgeFunction" (linia 24). Rezultat tych obliczeń wyświetlany w linii 28 został zaprezentowany na Listingu 18.

LISTING 18: Rezultat wykonania funkcji "inner-function"

```

1  {
2      "inner-function": {
3          "eyJhIjo0LCJiIjo1fQ==": {
4              "result": 9
5          }
6      }
7  }

```

Następnie klient na Listingu 17, w linii 29, wykonuje w chmurze funkcję "outer-function", którą wywołuje pod adresem "http://test-app.custom-serverless.com/outer", a w ciele zapytania podaje argumenty wejściowe funkcji w parametrze *args* oraz podaje wynik przetwarzania na krawędzi w parametrze *edgeResults* (zawartość tego parametru została przedstawiona na Listingu 18). Wyświetlony w linii 36 rezultat przetwarzania został przedstawiony na poniższym Listingu 19.

LISTING 19: Rezultat wykonania funkcji "outer-function"

```

1  {
2      "result": {
3          "out": 54
4      },
5      "cache": {
6          "outer-function": {
7              "eyJhIjo0LCJiIjo1LCJjIjo2fQ==": {
8                  "out": 54
9              }
10         }
11     }
12 }

```

Rezultat wykonania funkcji przedstawiono na Listingu 19, w parametrze *result* (linia 2) i wynosi on {"out": 54}. Obie funkcje są idempotentne, natomiast wynik "inner-function" został przekazany w parametrze *edgeResults* przez klienta, zatem aplikacja kliencka nie wykonała tej funkcji i nie umieściła jej rezultatu w pamięci podręcznej. Natomiast aplikacja kliencka wykonała idempotentną funkcję "outer-function", zatem umieściła jej wynik w pamięci podręcznej, której zawartość zwróciła klientowi w parametrze *cache* (linia 5).

Rozdział 6

Testy i porównanie z innymi dostępnymi rozwiązaniami

6.1 Testy poprawności działania aplikacji klienckiej

Przeprowadzone testy miały na celu weryfikację rozwiązania pod względem poprawności, poprzez wywołanie aplikacji klienckiej ze zmieniającymi się wartościami parametrów *args*, *cache*, *edgeResults*.

Przedmiotem testów było obliczenie wartości równania:

$$(x + y)^2 \quad (6.1)$$

W tym celu zaproponowano dwie funkcje — odpowiednio o nazwach ”add-function” i ”square-function”. Pierwsza z funkcji przyjmuje dwa argumenty wejściowe *x* oraz *y*, a następnie wykonuje operację dodawania tych dwóch zmiennych. Druga z funkcji o nazwie ”square-function”, przyjmuje dwa argumenty wejściowe *x* oraz *y*, a następnie wywołuje przy ich pomocy funkcję ”add-function” oraz zwraca jej wynik podniesiony do kwadratu. Ciała tych funkcji zostały przedstawione na Listingach 20 oraz 16. Obie funkcje są idempotentne. Funkcja ”square-function” zdefiniowana została w aplikacji o nazwie ”test-app” i jest możliwa do wywołania poprzez odwołanie do adresu ”square”.

LISTING 20: Kod funkcji ”add-function”

```
1  async (args) => {
2      return {"result": args.x + args.y};
3  };
```

LISTING 21: Kod funkcji ”square-function”

```
1  async (args) => {
2      let addResult = (await call("add-function", args)).result;
3      return {"result": addResult * addResult};
4  };
```

Do przeprowadzenia testów wykorzystano framework języka JavaScript o nazwie *Vitest*. Każdy z testów został zdefiniowany za pomocą metody *it*, przyjmującej jako argumenty nazwę testu oraz treść funkcji testującej. Metoda *expect* służy do sprawdzania warunków poprawności testu. Przykład definicji testu został przedstawiony na Listingu 22.

LISTING 22: Przykład definicji testu

```

1 import { it } from 'vitest';
2
3 it('nazwa testu', () => {
4     // ciało funkcji testującej
5 });

```

W ramach testów funkcji 21 rozważono następujące przypadki:

- W pierwszym przypadku sprawdzono, czy żądanie klienta wysłane z argumentami x=4, y=5 bez podania w zapytaniu parametrów *cache* oraz *edgeResults*, zwróci poprawny wynik. Ponadto, sprawdzono czy w związku z tym, że obie funkcje są idempotentne, zostanie zwrócona pamięć podręczna zawierająca rezultaty funkcji "square-function" oraz "add-function" dla argumentów wejściowych, z którymi zostały one wywołane.

LISTING 23: test 1

```

1 it(
2     'should compute function in cloud and return valid result and cache',
3     async () => {
4         let x = 4;
5         let y = 5;
6         let args = {
7             x: x,
8             y: y
9         }
10        let request = {
11            args: args
12        }
13        let expectedAddFunctionResult = x + y;
14        let expectedSquareFunctionResult =
15            Math.pow(expectedAddFunctionResult, 2);
16
17        let response =(await axios.post(
18            'http://test-app.custom-serverless.com/square',
19            request)
20        ).data;
21
22        let addFunctionArgsEntry =
23            Object.keys(response.cache['add-function']).find(cacheEntry =>
24                deepEqual(
25                    JSON.parse(Buffer.from(cacheEntry, 'base64').toString()),
26                    args
27                )
28            );
29        let squareFunctionArgsEntry =
30            Object.keys(response.cache['square-function']).find(cacheEntry =>
31                deepEqual(
32                    JSON.parse(Buffer.from(cacheEntry, 'base64').toString()),
33                    args
34                )
35            );
36
37        expect(response.data).toEqual({
38            addFunctionResult: expectedAddFunctionResult,
39            squareFunctionResult: expectedSquareFunctionResult
40        });
41
42        expect(addFunctionArgsEntry).toEqual(args);
43        expect(squareFunctionArgsEntry).toEqual(args);
44    }
45);

```

```

34         )
35     );
36
37     expect(response.result.result).toBe(expectedSquareFunctionResult);
38
39     expect(response.cache['add-function']).toBeDefined();
40     expect(addFunctionArgsEntry).toBeDefined();
41     expect(response.cache['add-function'][addFunctionArgsEntry].result)
42         .toBe(expectedAddFunctionResult);
43
44     expect(response.cache['square-function']).toBeDefined();
45     expect(squareFunctionArgsEntry).toBeDefined();
46     expect(
47         response.cache['square-function'][squareFunctionArgsEntry].result
48     ).toBe(expectedSquareFunctionResult);
49 }

```

W ramach testu przedstawionego na Listingu 23 zostały sprawdzone następujące warunki:

- Sprawdzenie, czy wartość równania 6.1 dla argumentów $x = 4$ oraz $y = 5$ jest spełniona, zatem sprawdzane jest, czy rezultat zwrócony przez aplikację przetwarzania bezserwrowego jest równy rezultatowi funkcji "add-function" podniesionej do kwadratu (linia 37).
- Sprawdzenie, czy wywołanie funkcji idempotentnych "add-function" i "square-function" skutkuje zapisaniem ich rezultatów w pamięci podręcznej (linie 39 oraz 44).
- Sprawdzenie, czy wpis w pamięci podręcznej wykonany w rezultacie wywołania funkcji "add-function" posiada rezultat dla argumentów wejściowych zdefiniowanych w linii 4 (linia 40).
- Sprawdzenie, czy wpis w pamięci podręcznej wykonany w rezultacie wywołania funkcji "square-function" posiada rezultat dla argumentów wejściowych zdefiniowanych w linii 4 (linia 45).
- Sprawdzenie, czy wpis w pamięci podręcznej wykonany w rezultacie wywołania funkcji "add-function" dla argumentów wejściowych zdefiniowanych w linii 4, posiada prawidłowy rezultat wynoszący $x + y$ (linia 41).
- Sprawdzenie, czy wpis w pamięci podręcznej wykonany w rezultacie wywołania funkcji "square-function" dla argumentów wejściowych zdefiniowanych w linii 4, posiada prawidłowy rezultat wynoszący wartość równania 6.1, a więc, czy jest równy wartości "add-function" podniesionej do kwadratu (linia 46).
- W kolejnym przypadku sprawdzono, czy żądanie klienta wyslane z argumentami $x=4$, $y=5$ oraz pamięcią podręczną zawierającą dla funkcji "add-function" niepoprawny rezultat wynoszący 10, spowoduje wykorzystanie tego rezultatu przez aplikację kliencką i w konsekwencji zwrócenie klientowi rezultatu wynoszącego $10^2 = 100$ zamiast poprawnego $(4 + 5)^2 = 81$.

LISTING 24: Test 2

```

1  it('should return result based on cache', async() => {
2      let x = 4;
3      let y = 5;

```

```

4      let args = {
5          x: x,
6          y: y
7      }
8      let base64Args = Buffer.from(JSON.stringify(args))
9          .toString('base64');
10     let cache = {};
11     cache['add-function'] = {};
12     cache['add-function'][base64Args] = { result: 10 };
13     let request = {
14         args: args,
15         cache: cache
16     }
17     let expectedAddFunctionResult = x + y;
18     let expectedSquareFunctionResult =
19         Math.pow(expectedAddFunctionResult, 2);
20
21     let response = (await axios.post(
22         'http://test-app.custom-serverless.com/square',
23         request)
24     ).data;
25     expect(response.result.result)
26         .not.toBe(expectedSquareFunctionResult);
27     expect(response.result.result).toBe(100);
28 });

```

W ramach testu przedstawionego na Listingu 24 zostały sprawdzone następujące warunki:

- Sprawdzenie, czy w wyniku podania przez klienta wartości pamięci podręcznej z niepoprawnym wynikiem wynoszącym 10 dla funkcji "add-function" dla argumentów $x = 4$ oraz $y = 5$ (linia 12), rezultat przetwarzania będzie poprawny (linia 25), czy niepoprawny (linia 27).
- W kolejnym przypadku sprawdzono, czy żądanie klienta wysiane z argumentami $x=4$, $y=5$ oraz pamięcią podręczną zawierającą dla funkcji "add-function" zły rezultat wynoszący 10, a także niepoprawny wynik przetwarzania tej funkcji na krawędzi, wynoszący 11, spowoduje wykorzystanie wyniku przetwarzania na krawędzi przez aplikację kliencką. Konsekwencją takiej sytuacji byłoby zwrócenie klientowi rezultatu wynoszącego $11^2 = 121$. Oznacza to, że aplikacja kliencka zignorowała zawartość pamięci podręcznej, w wyniku czego rezultat na niej nie bazuje, a więc nie wynosi $10^2 = 100$. Ponadto, w związku ze skorzystaniem z rezultatu przetwarzania na krawędzi, nie zostanie wykonane przetwarzanie funkcji "add-function" w aplikacji klienckiej, co spowoduje brak poprawnego rezultatu równania wynoszącego $(4 + 5)^2 = 81$.

LISTING 25: Test 3

```

1 it('should return result based on edgeResults', async() => {
2     let x = 4;
3     let y = 5;
4     let args = {
5         x: x,

```

```

6          y: y
7      }
8  let base64Args = Buffer.from(JSON.stringify(args))
9      .toString('base64');
10 let cache = {};
11 cache['add-function'] = {};
12 cache['add-function'][base64Args] = { result: 10 };
13 let edgeResults = {};
14 edgeResults['add-function'] = {};
15 edgeResults['add-function'][base64Args] = { result: 11 };
16 let request = {
17     args: args,
18     cache: cache,
19     edgeResults: edgeResults
20 }
21 let expectedAddFunctionResult = x + y;
22 let expectedSquareFunctionResult =
23     Math.pow(expectedAddFunctionResult, 2);
24
25 let response =(await axios.post(
26     'http://test-app.custom-serverless.com/square',
27     request)
28 ).data;
29 expect(response.result.result)
30     .not.toBe(expectedSquareFunctionResult);
31 expect(response.result.result).not.toBe(100);
32 expect(response.result.result).toBe(121);
33 });

```

W ramach testu przedstawionego na Listingu 25 zostały sprawdzone warunki polegające na sprawdzeniu, czy w wyniku podania przez klienta wartości przetwarzania na krawędzi z niepoprawnym wynikiem wynoszącym 11 dla funkcji "add-function" dla argumentów $x = 4$ oraz $y = 5$ (linia 15):

- rezultat przetwarzania nie jest równy $(x + y)^2$ (linia 29).
 - zawartość pamięci podręcznej wynoszącej 10 (linia 12) jest ignorowana, a zatem rezultat przetwarzania jest nie równy $10^2 = 100$ (linia 31).
 - rezultat przetwarzania jest równy $11^2 = 121$ (linia 32).
- W ostatnim przypadku sprawdzono poprawność pobrania, a następnie przetworzenia przez klienta na krawędzi funkcji "add-function" dla argumentów $x=4$, $y=5$. Następnie sprawdzono, czy przesłanie w żądaniu wysłanym pod adres "square" poprawnego wyniku przetwarzania na krawędzi funkcji "add-function", pozwoliło na uzyskanie poprawnego rezultatu równego $(x + y)^2$.

LISTING 26: Test 4

```

1 it('should compute edgeResults correctly', async() => {
2     let x = 4;
3     let y = 5;
4     let args = {
5         x: x,

```

```

6          y: y
7      }
8      let base64Args = Buffer.from(JSON.stringify(args))
9          .toString('base64');
10
11     let edgeFunctionsResponse = await axios.post(
12         'http://test-app.custom-serverless.com/edge',
13         {
14             "functions": ["add-function"]
15         }
16     );
17
18     let runEdgeFunction = eval(
19         edgeFunctionsResponse.data.runEdgeFunction
20     );
21     let edgeResults = await runEdgeFunction(
22         "add-function",
23         edgeFunctionsResponse.data.functions,
24         args
25     );
26
27     let request = {
28         args: args,
29         edgeResults: edgeResults
30     }
31     let expectedAddFunctionResult = x + y;
32     let expectedSquareFunctionResult =
33         Math.pow(expectedAddFunctionResult, 2);
34
35     let response =(await axios.post(
36         'http://test-app.custom-serverless.com/square',
37         request)
38     ).data;
39
40     expect(edgeResults["add-function"][base64Args].result)
41         .toBe(expectedAddFunctionResult);
42     expect(response.result.result).toBe(expectedSquareFunctionResult);
43 });

```

W ramach testu przedstawionego na Listingu 26 zostały sprawdzone następujące warunki:

- Sprawdzenie, czy w wyniku przetwarzania na krawędzi funkcji "add-function" dla argumentów $x = 4$ oraz $y = 5$ (linia 4) otrzymano poprawny rezultat wykonania tej funkcji wynoszący $x + y$ (linia 40).
- Sprawdzenie, czy w wyniku przetwarzania na krawędzi funkcji "add-function" otrzymano poprawny rezultat przetwarzania funkcji "square-function" w aplikacji klienckiej wynoszący $(x + y)^2$ (linia 42).

Powyższe testy o nazwach podanych w pierwszym argumencie metody `it` uruchomiono w terminalu za pomocą komendy `npm test`, która wykorzystuje framework `vitest` do uruchomienia testów. Wszystkie powyższe testy zakończyły się z wynikiem pozytywnym (nie zwróciły żadnego błędu), a zatem wszystkie warunki zdefiniowane w metodach `expect` we wszystkich zdefiniowanych testach zostały spełnione, co zostało przedstawione na Rysunku 6.1.

```

→ client-app git:(feature/integration-tests) ✘ npm test

> custom-serverless-client-app@1.0.0 test
> vitest --run --reporter verbose

RUN v0.15.2 /home/piotr/studia/praca_magisterska/custom-serverless/client-app

✓ integration-tests/integration.test.js (4) 404ms
  ✓ should compute function in cloud and return valid result and cache
  ✓ should return result based on cache
  ✓ should return result based on edgeResults
  ✓ should compute edgeResults correctly

Test Files 1 passed (1)
Tests 4 passed (4)
Time 1.50s (in thread 404ms, 370.18%)

→ client-app git:(feature/integration-tests) ✘

```

RYSUNEK 6.1: Wynik wykonania testów poprawności działania aplikacji

6.2 Testy czasu trwania przetwarzania aplikacji klienckiej

Przedstawione testy pokażą zysk czasowy wynikający z korzystania z wyników przetwarzania na krawędzi oraz z wyników pamięci podręcznej.

W ramach testów skierowanych pod adres o nazwie "heavy" aplikacji "test-app" wywołana jest zewnętrzna funkcja o nazwie "heavy-outer", przedstawiona na Listingu 28, która w swoim kodzie wykonuje idempotentną funkcję "heavy-inner" przedstawioną na Listingu 27.

LISTING 27: Kod funkcji `heavy-inner`

```

1 const bcrypt = require('bcryptjs');
2 async (args) => {
3     return await bcrypt.hash(args.text, 16);
4 }

```

LISTING 28: Kod funkcji `heavy-outer`

```

1 async (args) => {
2     let result = await call("heavy-inner", {text: args.text});
3     return result;
4 }

```

Funkcja "heavy-inner" oblicza skrót hash za pomocą biblioteki `bcrypt`, z wykorzystaniem metody `bcrypt.hashSync`. Metoda ta przyjmuje 2 argumenty: wejściowy tekst, na podstawie którego wyliczany jest skrót oraz sól (ang. salt), którego wartość liczbowa jest losową liczbą bitów dodawaną jako argument wejściowy do funkcji hash-ującej — im większa liczba bitów, tym bezpieczniejszy i bardziej losowy jest skrót hash, ale również tym dłuższe są obliczenia. Wartość 16 wybrano w sposób empiryczny, aby zasymulować funkcję wykonującą obliczenia zajmujące co najmniej kilka sekund. Funkcja "heavy-outer" odpowiada jedynie za wywołanie funkcji "heavy-inner" oraz zwrócenie jej wyniku.

Przyśpieszenie obliczeń w chmurze dzięki wykorzystaniu z wyników przetwarzania na krawędzi oraz wyników pamięci podręcznej przedstawiono na podstawie poniższych testów integracyjnych, mierzących czas przetwarzania wywołania punktu końcowego o nazwie "heavy":

- W pierwszym z testów przedstawionym na Listingu 29 zmierzono czas wykonania całości obliczeń w chmurze bez korzystania z wyników przetwarzania na krawędzi oraz wyników pamięci podręcznej.

LISTING 29: Test przedstawiający pełne obliczenia w chmurze

```

1  const start = Date.now();
2  let result = await axios.post(
3      'http://test-app.custom-serverless.com/heavy',
4      {args: {text: "Piotr"}}
5  );
6  const stop = Date.now();
7  console.log(`[full cloud computation] = ${((stop - start)} milliseconds`);
```

W powyższym teście zmierzono czas wywołania żądań skierowanych pod adres "http://test-app.custom-serverless.com/heavy" mierząc datę początku i końca obliczeń (linie 1 oraz 6), a następnie wyświetlając ich różnicę co daje wynik w milisekundach (linia 7). Czas przetwarzania testu zmierzony w linii 7 wyniósł 7122 milisekund.

- W kolejnym z testów, przedstawionym na Listingu 30, zmierzono czas wykonania żądania wysłanego pod adres "heavy", gdzie aplikacja kliencka nie przetwarza samodzielnie kodów funkcji "heavy-inner" ze względu na otrzymaną w zapytaniu pamięć podręczną.

LISTING 30: Test przedstawiający obliczenia w chmurze korzystające z pamięci podręcznej klienta

```

1  const start = Date.now();
2  let result = await axios.post(
3      'http://test-app.custom-serverless.com/heavy',
4      {
5          args: {"text": "Piotr"},
6          cache: {
7              "heavy-inner": {
8                  "eyJOZXh0IjoiUG1vdHIifQ==":
9                      "$2a$16$MsP/7ZSsZ00AcV6UjqOV2u3z90rYm5cew/0c28yGYeTQ6hPNbXqZa"
10             }
11         }
12     }
13 );
14 const stop = Date.now();
15 console.log(
16     ` [cloud computation using cache] = ${((stop - start)} milliseconds`  

17 );
```

W przedstawionym powyżej teście w linii 6 klient podaje w ciele zapytania zawartość pamięci podręcznej w parametrze *cache*.

Czas przetwarzania testu na Listingu 30 zmierzony w linii 15 wyniósł 80 milisekund.

- W następnych z testów przedstawionym na Listingu 31 zmierzono czas wykonania punktu końcowego "heavy", gdzie aplikacja kliencka nie przetwarza samodzielnie kodów funkcji "heavy-inner" ze względu na otrzymane w zapytaniu rezultaty przetwarzania na krawędzi.

LISTING 31: Test przedstawiający obliczenia w chmurze korzystające z wyników przetwarzania na krawędzi klienta

```

1  const start = Date.now();
2  let result = await axios.post(
3    'http://test-app.custom-serverless.com/heavy',
4    {
5      args: {"text": "Piotr"},
6      edgeResults: {
7        "heavy-inner": [
8          "eyJ0ZXh0IjoiUG1vdHIifQ==",
9          "$2a$16$MsP/7ZSsZ00AcV6Ujq0V2u3z90rYm5cew/0c28yGYeTQ6hPNbXqZa"
10        ]
11      }
12    }
13  );
14  const stop = Date.now();
15  console.log(
16    `[cloud computation using edgeResults] = ${((stop - start)} milliseconds`
17  );

```

W przedstawionym powyżej teście, w linii 6, klient podaje w ciele zapytania zawartość wyników przetwarzania na krawędzi w parametrze *edgeResults*.

Czas przetwarzania testu zmierzony w linii 15 na Listingu 31 wyniósł 86 milisekund.

Zatem wykorzystanie pamięci podręcznej oraz wyników przetwarzania na krawędzi przez klienta, dzięki braku konieczności wykonywania kodu funkcji "heavy-inner" przez aplikację kliencką znacznie skróciło czas obliczeń wykonywanych w chmurze.

6.3 Porównanie z innymi dostępnymi rozwiązaniami

Żaden z omówionych w sekcji 2.2.6 najpopularniejszych dostawców przetwarzania bezserwerowego nie oferuje klientowi możliwości wykonywania części lub całości obliczeń na krawędzi po stronie urządzenia końcowego klienta. Ponadto, żaden z dostawców nie zwraca klientowi rezultatów funkcji idempotentnych umieszczonych w pamięci podręcznej w celu przyspieszenia obliczeń. Powyższą funkcjonalność oferuje zaproponowane w niniejszej pracy rozwiązanie. Ponadto, tylko jeden z najpopularniejszych dostawców, którym jest platforma Cloud Functions, oferuje funkcjonalność zaproponowanego rozwiązania polegającą na możliwości definiowania przez klienta zewnętrznych bibliotek w pliku z zależnościami, które na podstawie zawartości tego pliku zostaną automatycznie zainstalowane w kontenerze z aplikacją przetwarzania bezserwerowego w momencie jej inicjalizacji. Przewagą oferty najpopularniejszych dostawców przetwarzania bezserwerowego nad zaproponowanym rozwiązaniem jest z kolei automatyczne inicjowanie kontenera z kodem funkcji w momencie jej wywołania zapytaniem HTTP przez klienta. Tymczasem w zaproponowanym rozwiąaniu wymagane jest, aby klient najpierw manualnie uruchomił aplikację, w celu wywołania jej funkcji. Z kolei zaproponowane rozwiązanie może być uznane za bardziej elastyczne, gdyż to klient ma kontrolę nad czasem, w którym jego aplikacja jest dostępna dla zewnętrznych użytkowników.

Rozdział 7

Podsumowanie

Celem pracy było stworzenie rozwiązania przetwarzania bezserwerowego pozwalającego na wykonanie części lub całości obliczeń po stronie klienta. W stworzonym rozwiążaniu kody zawierające funkcje przetwarzania bezserwerowego są uruchamiane w kontenerach w chmurze. Kody te mogą również zostać wykonane na urządzeniu wykorzystywanym do przetwarzania na krawędzi przez klienta.

Programista tworzący funkcje ma możliwość skorzystania z zewnętrznych bibliotek bez konieczności umieszczania pliku zip z kodami bibliotek w chmurze oraz bez konieczności samodzielnego tworzenia obrazu kontenera z zainstalowanymi bibliotekami, a następnie umieszczenia go w repozytorium obrazów w celu umożliwienia dostawcy chmurowemu uruchomienia funkcji, tak jak ma to miejsce w przypadku usługi AWS Lambda. Programista może skorzystać z zewnętrznej biblioteki w kodzie funkcji, poprzez dodanie jej nazwy i wersji do pliku package.json, zawierającego listę zależności. Następnie przy uruchamianiu funkcji w celach testowych lub uruchamiając aplikacje dla klientów, biblioteki te zostaną automatycznie zainstalowane na podstawie zawartości pliku package.json przy inicjalizacji kontenera. Zaproponowane rozwiązanie okazało się również skuteczne w kontekście przyśpieszenia czasu realizowanych obliczeń, poprzez zwracanie klientowi wartości rezultatów wywołań funkcji idempotentnych znajdujących się w pamięci podręcznej.

Zaproponowane rozwiązanie wyróżnia się możliwością wykonywania części lub całości obliczeń na krawędzi, czego nie oferują najpopularniejsi dostawcy chmurowi. Ponadto, spośród wspomnianych dostawców, tylko platforma Cloud Functions oferuje funkcjonalność zaproponowanego rozwiązania, polegającą na możliwości definiowania przez klienta zewnętrznych bibliotek w pliku z zależnościami.

Mögliwe kierunki dalszej pracy obejmują rozszerzenie zaproponowanego rozwiązania o automatyczną inicjalizację aplikacji klienckiej, po otrzymaniu przez system żądania HTTP związanego z wywołaniem funkcji. Kolejną możliwą do rozwinięcia funkcjonalnością jest możliwość automatycznego skalowania aplikacji klienckich w przypadku wysyłania do nich dużej liczby żądań HTTP, co zapewniłoby zrównoleglenie wykonywania funkcji i w konsekwencji szybsze dostarczenie rezultatu funkcji klientowi.

Literatura

- [ADSGKT20] Auday Al-Dulaimy, Yogesh Sharma, Michel Gokan Khan, and Javid Taheri. *Introduction to edge computing*. 09 2020.
- [app22a] apprunner. [on-line]<https://aws.amazon.com/apprunner/>, (dostęp 04.06.2022).
- [app22b] appservice. [on-line]<https://docs.microsoft.com/en-us/azure/app-service/>, (dostęp 05.06.2022).
- [ATC⁺21] Mohammad S Aslanpour, Adel N Toosi, Claudio Cicconetti, Bahman Javadi, Peter Sbarski, Davide Taibi, Marcos Assuncao, Sukhpal Singh Gill, Raj Gaire, and Schahram Dustdar. Serverless edge computing: vision and challenges. In *2021 Australasian Computer Science Week Multiconference*, pages 1–10, 2021.
- [aws22] awslambda. [on-line]<https://docs.aws.amazon.com/lambda/>, (dostęp 07.06.2022).
- [azu22] azurefunctions. [on-line]<https://docs.microsoft.com/en-us/azure/azure-functions>, (dostęp 04.06.2022).
- [BCC⁺17] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Isahagian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. *Serverless Computing: Current Trends and Open Problems*, pages 1–20. 12 2017.
- [cap22] capitalonecontainers.
[on-line]<https://www.capitalone.com/tech/cloud/container-adoption-statistics/>, (dostęp 15.06.2022).
- [CCP21] Claudio Cicconetti, Marco Conti, and Andrea Passarella. On realizing stateful faas in serverless edge networks: State propagation. In *2021 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 89–96. IEEE, 2021.
- [clo22a] cloudfunctions. [on-line]<https://cloud.google.com/functions/docs/>, (dostęp 04.06.2022).
- [Clo22b] Cloud Service Models.
<https://mediacomem.github.io/comem-webdev-docs/latest/subjects/cloud>, (dostęp 01.06.2022).
- [CP22] Jeremy Likness Cecil Phillip. *Serverless apps: Architecture, patterns, and Azure implementation*. (dostęp 05.06.2022).
- [DIPW20] Anirban Das, Shigeru Imai, Stacy Patterson, and Mike P Wittie. Performance optimization for edge-cloud serverless platforms via dynamic task placement. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 41–50. IEEE, 2020.
- [doc22] dockercontainer. [on-line]<https://www.docker.com/resources/what-container/>, (dostęp 10.06.2022).

- [eve22] eventgrid.
[on-line]<https://azure.microsoft.com/pl-pl/services/event-grid/#overview>, (dostęp 05.06.2022).
- [goo22] googlestadia. [on-line]<https://stadia.google.com/>, (dostęp 14.06.2022).
- [HMW22] Robin Hartauer, Johannes Manner, and Guido Wirtz. Cloud function lifecycle considerations for portability in function as a service. pages 133–140, 01 2022.
- [ibm22] ibmcontainerisation. [on-line]<https://www.ibm.com/cloud/learn/containerization>, (dostęp 08.06.2022).
- [Jak20] Michał Tomasz Jakóbczyk. *Practical Oracle Cloud Infrastructure*. Springer, 2020.
- [JTA21] Hamza Javed, Adel N Toosi, and Mohammad S Aslaniour. Serverless platforms on the edge: A performance analysis. *arXiv preprint arXiv:2111.06563*, 2021.
- [KG] Benjamin Kettner and Frank Geisler. Pro serverless data handling with microsoft azure.
- [KM18] Sean P Kane and Karl Matthias. *Docker: up & running: shipping reliable containers in production*. O'Reilly Media, 2018.
- [KSDN22] K Anitha Kumari, G Sudha Sadasivam, D Dharani, and M Niranjanamurthy. *Edge Computing: Fundamentals, Advances and Applications*. CRC Press, 2022.
- [mub22] mubismartcities. [on-line]<https://mubi.pl/poradniki/inteligentne-miasto/>, (dostęp 14.06.2022).
- [RC17] Michael Roberts and John Chapin. *What is Serverless?* O'Reilly Media, Incorporated, 2017.
- [red20] IaaS vs paas vs saas.
[on-line]<https://www.redhat.com/en/topics/cloud-computing/iaas-vs-paas-vs-saas>, 2020.
- [red22a] redhatdocker.
[on-line]<https://www.redhat.com/en/topics/containers/what-is-docker>, (dostęp 10.06.2022).
- [red22b] redhatkubernetes.
[on-line]<https://www.redhat.com/en/topics/containers/what-is-kubernetes>, (dostęp 11.06.2022).
- [Rob18] Mike Roberts. Serverless architectures.
[on-line]<https://martinfowler.com/articles/serverless.html>, 2018.
- [Saw19] Rahul Sawhney. Beginning azure functions. 2019.
- [SCAC⁺19] Inés Sittón-Candanedo, Ricardo S Alonso, Juan M Corchado, Sara Rodríguez-González, and Roberto Casado-Vara. A review of edge computing reference architectures and a new global edge proposal. *Future Generation Computer Systems*, 99:278–294, 2019.
- [SK17] Peter Sbarski and Sam Kroonenburg. *Serverless architectures on Aws: with examples using Aws Lambda*. Simon and Schuster, 2017.
- [SRCL⁺22] Nour Alhuda Sulieman, Lorenzo Ricciardi Celsi, Wei Li, Albert Zomaya, and Massimo Villari. Edge-oriented computing: A survey on research and use cases. *Energies*, 15(2):452, 2022.
- [SRS17] Vaishali Sharma, PV Rai, and KK Sharma. Edge computing: Needs, concerns, and challenges. *International Journal of Scientific and Engineering Research*, 8(4):154, 2017.

- [SS19] Slobodan Stojanović and Aleksandar Simović. *Serverless Applications with Node. js: Using AWS Lambda and Claudia. js*. Manning Publications, 2019.
- [Sti18] Maddie Stigler. *Beginning Serverless Computing*. Springer, 2018.
- [WG17] Yohan Wadia and Udita Gupta. *Mastering AWS Lambda*. Packt Publishing Ltd, 2017.
- [wha22a] whatisangular. [on-line]<https://angular.io/guide/what-is-angular>, (dostęp 17.06.2022).
- [wha22b] whatisexpress. [on-line]<https://expressjs.com/>, (dostęp 17.06.2022).
- [wha22c] whatiskubernetes. [on-line]<https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>, (dostęp 17.06.2022).
- [wha22d] whatisterraform. [on-line]<https://www.terraform.io/intro>, (dostęp 17.06.2022).
- [XTQ⁺21] Renchao Xie, Qinjin Tang, Shi Qiao, Han Zhu, F Richard Yu, and Tao Huang. When serverless computing meets edge computing: architecture, challenges, and open issues. *IEEE Wireless Communications*, 28(5):126–133, 2021.



© 2022 Piotr Mazurek

Instytut Informatyki, Wydział Informatyki i Telekomunikacji
Politechnika Poznańska

Skład przy użyciu systemu L^AT_EX na platformie Overleaf.