

Uniwersytet Mikołaja Kopernika  
Wydział Matematyki i Informatyki

Klaudia Augustyńska  
nr albumu: 265408  
informatyka

Praca magisterska

# Wykorzystanie Cloud Computing w aplikacjach mobilnych

Opiekun pracy dyplomowej  
dr Błażej Zyglarski

Toruń 2018



# Spis treści

<b>Wstęp</b>	<b>5</b>
Problem zastosowania chmury obliczeniowej w aplikacjach mobilnych	6
Cel pracy . . . . .	7
Opis rozdziałów . . . . .	8
<b>1. Wprowadzenie</b>	<b>9</b>
1.1. Charakterystyka Cloud Computingu . . . . .	9
1.2. Ewolucja Cloud Computingu . . . . .	13
1.3. Historia najnowsza. Współczesne wyzwania . . . . .	16
1.3.1. Rozwój chmur publicznych . . . . .	16
1.3.2. Konteneryzacja i DevOps . . . . .	17
1.3.3. Mikrousługi i serverless . . . . .	22
1.3.4. Bazy NoSQL, NewSQL i rozwiązania dla Big Data . . . . .	26
1.3.5. CDN i Fog Computing . . . . .	31
1.4. Wymagania stawiane współczesnym chmurom . . . . .	34
1.4.1. Zasadnicze cechy chmury . . . . .	35
1.4.2. Modele dostarczania usług . . . . .	35
1.4.3. Formy wdrożenia . . . . .	37
1.5. Dokąd zmierza rozwój Cloud Computingu . . . . .	38
<b>2. Porównanie różnych podejść</b>	<b>41</b>
2.1. Budowanie własnej chmury prywatnej . . . . .	41
2.2. Łączenie chmury prywatnej z chmurą publiczną . . . . .	43
2.3. Wykupienie serwera wirtualnego w chmurze . . . . .	45
2.4. Usługi typu PaaS oparte o maszyny wirtualne w chmurach publicznych . . . . .	48
2.5. Samodzielna instalacja orkiestratora kontenerów . . . . .	52
2.6. Usługa konteneryzacji w chmurach publicznych . . . . .	53
2.7. Usługi typu PaaS oparte o kontenery w chmurach publicznych	55
2.8. Serverless – usługi typu FaaS . . . . .	57

2.9. Serverless bez przywiązania do platformy . . . . .	58
2.9.1. Serverless Framework . . . . .	59
2.9.2. Kubeless . . . . .	60
<b>3. Analiza wyboru platformy dla projektu</b>	<b>61</b>
3.1. Opis projektu aplikacji mobilnej . . . . .	61
3.1.1. Geneza pomysłu . . . . .	61
3.1.2. Typowy projekt dla chmury . . . . .	63
3.2. Wymagania dotyczące wyboru podejścia oraz chmury . . . .	66
3.3. Analiza . . . . .	67
3.3.1. Wybór podejścia . . . . .	67
3.3.2. Wybór chmury . . . . .	71
3.3.3. Podsumowanie analiz . . . . .	73
<b>4. Opis wdrożenia aplikacji</b>	<b>77</b>
4.1. Prezentacja aplikacji . . . . .	77
4.1.1. Podstawowa obsługa aplikacji . . . . .	77
4.1.2. Dodawanie wydatku . . . . .	79
4.1.3. Łączenie w gospodarstwo domowe . . . . .	80
4.2. Implementacja API . . . . .	83
4.2.1. Teoretyczne podstawy Azure Functions . . . . .	83
4.2.2. Przygotowanie środowiska . . . . .	86
4.2.3. Baza danych Azure Table Storage . . . . .	88
4.2.4. Pierwsze funkcje . . . . .	90
4.2.5. Funkcje z dostępem na klucz . . . . .	92
4.2.6. Azure Queue Storage . . . . .	92
4.3. Aplikacja na Android . . . . .	94
<b>Podsumowanie</b>	<b>95</b>
<b>Spis rysunków</b>	<b>98</b>
<b>Spis tabel</b>	<b>100</b>
<b>Bibliografia</b>	<b>101</b>

# Wstęp

Cloud Computing to model stanowiący podstawę dla jednych z najprężniej rozwijających się technologii informatycznych obecnych czasów. Jego wykorzystanie odnosi się do praktycznie wszystkich dziedzin informatyki, poczynając od administrowania infrastrukturą komputerową, przez tworzenie systemów informatycznych, po wsparcie badań naukowych oraz pracy przeciętnych użytkowników komputerów.

Technologie oparte o Cloud Computing mają swoje podwaliny w starszych technologiach, sięgających lat 70. ubiegłego wieku. Chodzi zatem o produkt ewolucji technologicznej, nie zaś odrębną nową technologię. Dziś dynamiczny rozwój Cloud Computingu zawdzięczamy m.in. szybkiemu łączu internetowemu, upowszechnieniu komputerów PC oraz mocy obliczeniowej umożliwiającej wirtualizację na poziomie wykorzystania sztucznej inteligencji do inteligentnego zarządzania infrastrukturą komputerową (ang. *autonomic computing*). Złożenie tych czynników umożliwiło urzeczywistnienie idei po raz pierwszy wymienionej w 1961 r. przez Johna McCarthy’ego, który pisał, że zasoby komputerowe staną się użytecznością publiczną – podobnie jak prąd, który pobieramy z sieci elektrycznej, nie zaś z własnego generatora prądu.[1, 5] Stąd też analizując temat Cloud Computingu niejednokrotnie można spotkać się ze zwrotem „X jako usługa” (ang. *X as a service*), ponieważ praktycznie wszystko, co może być związane z wykorzystaniem komputerów, można dostarczać jako usługę. [1]

Za chmurami obliczeniowymi stoją olbrzymie centra danych, łączące w sieć tysiące komputerów. Wirtualizacja wykorzystywana w Cloud Computingu pozwala na automatyczną konfigurację nowych serwerów dołączanych do sieci oraz odpowiednie zachowanie sieci w przypadku gdy któraś jej część przestanie działać. Dzięki takim narzędziom można dowolnie rozbudować sieć, a więc uzyskać dowolną przestrzeń dyskową i moc obliczeniową. Dlatego model chmury obliczeniowej jest nieodzownie powiązany z technikami wirtualizacji oraz technikami pracy na systemach rozproszonych.

Duża i zarazem bardzo znacząca część technologii związanych z Cloud

Computingiem dotyczy programowania. Chodzi nie tylko o usługi usprawniające proces wytwórczy oprogramowania, ale również o technologie i wzorce wspierające skalowalność oraz obsługę dużej ilości danych. Chmura pozwala zautomatyzować wiele zadań, z drugiej strony jej ogromne możliwości stanowią duże wyzwanie, gdyż mogą oznaczać całkowitą zmianę podejścia do wytwarzania oprogramowania, wyboru technologii oraz wzorców architektonicznych. Wiele firm powoli podejmuje to wyzwanie przez np. stopniową rezygnację z serwerów firmowych na rzecz serwerów wirtualnych dostępnych w chmurze, wdrażanie strategii CI/CD czy tworzenie nowych projektów w architekturze mikrousług.

Biorąc pod uwagę to, w jak szybkim czasie pojawiło się wiele istotnych narzędzi szybko znajdujących zastosowanie w biznesie, istnieje duże zapotrzebowanie na opracowanie tego tematu całościowo.

## **Problem zastosowania chmury obliczeniowej w aplikacjach mobilnych**

Chmura obliczeniowa stanowi ważną część tzw. *networked society*, czyli obrazu, do którego współcześnie dąży technologia oraz sposób korzystania z niej przez społeczeństwo (rysunek 1). Jest to paradygmat, w którym korzystanie z elektronicznych asystentów czy Internetu Rzeczy (ang. IoT, *Internet of Things*) stanowi niezbędny element rzeczywistości.[5] W tym nowym obrazie świata chmura stanowi spoiwo, natomiast komunikacja z człowiekiem w dużej mierze odbywa się przy pomocy aplikacji mobilnych. Można spodziewać się rosnącego zapotrzebowania na aplikacje mobilne, których głównym zadaniem jest skuteczna komunikacja z chmurą.

Podczas wyboru technologii, na etapie analizy dostępnych rozwiązań okazuje się, że istnieje bardzo wiele podejść do tematu, z których duża część powstała w ciągu kilku ostatnich lat i od momentu publikacji zdobyła natychmiastową popularność. Dodatkowo wszystkie materiały, które pozwoliłyby to wszystko zrozumieć, występują w języku angielskim, najczęściej w kontekście wybranego specjalistycznego zastosowania.

Powyższy problem przyczynia się do braku zrozumienia technologii chmurowych jako całości. Powoduje to niską świadomość tego, jak w pełni wykorzystać możliwości chmury w kontekście tworzenia typowego projektu informatycznego, jakim jest aplikacja mobilna.



Rysunek 1: Rosnące zainteresowanie Internetem Rzeczy [26]

## Cel pracy

Celem pracy było rozstrzygnięcie, jak na obecny stan wiedzy podejść do problemu realizacji typowego projektu aplikacji mobilnej w oparciu o Cloud Computing. W rozumieniu niniejszej pracy, typowa aplikacja pozwala na wprowadzanie i otrzymywanie danych od serwera za pomocą udostępnionego API. W rezultacie miała zostać wybrana konkretna technologia, która pod względem teoretycznym najlepiej odpowiadałaby stawianym wymaganiom.

Wybrana technologia miała zostać użyta w praktyce do stworzenia aplikacji na system Android, komunikującej się z API udostępnianym przez chmurę. Aplikacja miała umożliwiać zapisywanie wydatków w sposób uwzględniający wydatki kilku osób w gospodarstwie domowym. System miał pozwolić założyć konto w serwisie, a następnie połączyć konta w grupy odpowiadające gospodarstwu domowemu. Kategorie wydatków miały pozwolić na uzgodnienie, kto ile płaci w danej kategorii (np. wydatki na higienę po 50%). Serwer miał być potrzebny do wspomagania rozliczeń pomiędzy poszczególnymi osobami przez dynamiczne naliczanie kto ma ile do oddania. Ponadto miał wspomagać kontrolę własnych wydatków przez uwzględnienie, że koszty życia to nie tylko pieniądze wydawane przez siebie samego, ale również wydatki poniesione przez inne osoby prowadzące to samo gospodarstwo domowe.

Ponieważ tradycyjne programy do zarządzania wydatkami nie posiadają wyżej wymienionych funkcjonalności, jak również zazwyczaj mogą działać bez serwera, to jest przykład aplikacji, która może nagle stać się popularna i powinna być w stanie wówczas obsłużyć większe obciążenie serwera.

## Opis rozdziałów

*Rozdział 1.* – *Wprowadzenie* zawiera najważniejsze informacje o tym, skąd wziął się Cloud Computing, jaki jest obecny stan wiedzy na jego temat oraz w jakim kierunku zmierza jego rozwój.

*Rozdział 2.* – *Porównanie różnych podejść* prezentuje dziewięć różnych podejść do wytwarzania w chmurze serwisów, z którymi może komunikować się aplikacja mobilna. Przy każdym podejściu czytelnik znajdzie listy konkretnych narzędzi oraz usług chmurowych, a także listy zalet i wad danego podejścia.

*Rozdział 3.* – *Analiza wyboru platformy dla aplikacji mobilnej* wyjaśnia, w jaki sposób chmura usprawnia tworzenie aplikacji mobilnych, oraz w odniesieniu do konkretnego projektu uzasadnia wybór rodzaju usługi chmurowej i dostawcy usługi.

*Rozdział 4.* – *Opis wdrożenia aplikacji* prezentuje aplikację oraz pokazuje, jak w praktyce wyglądało wykorzystanie wybranych narzędzi.



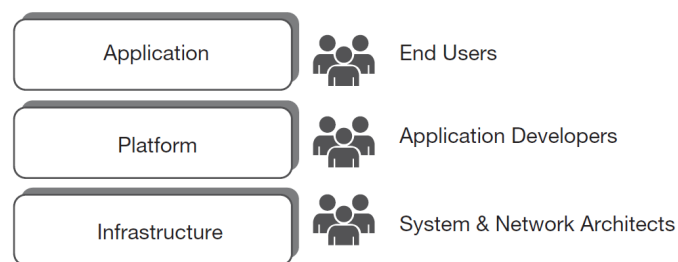
# Rozdział 1.

## Wprowadzenie

Niniejszy rozdział wyjaśnia, w jaki sposób ewoluowały technologie, by dać się później poznać jako Cloud Computing, a także czego należy oczekiwać od współczesnej chmury. Na końcu rozdziału opisano dokąd zmierza rozwój przedstawionych technologii.

### 1.1. Charakterystyka Cloud Computingu

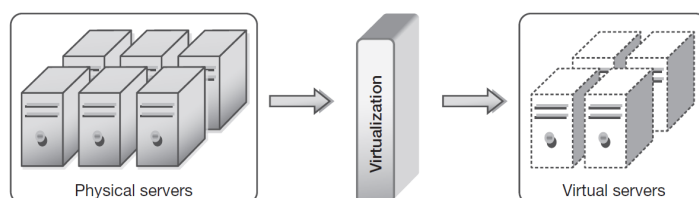
Cloud Computing to model, zgodnie z którym wszelkie zasoby informatyczne (oprogramowanie, przestrzeń dyskowa, dostęp do bazy danych itp.) dostarczane są w formie usługi. Istotną cechą jest wysoka **skalowalność** udostępnianych rozwiązań. Po stronie klienta ma to wyglądać tak, jak gdyby posiadał dostęp do nieskończonej mocy obliczeniowej i niekończącej się przestrzeni dyskowej, natomiast po stronie usługodawcy podłączenie nowych serwerów w celu podtrzymania tej iluzji nie powinno stanowić problemu. [2]



Rysunek 1.1: Różni użytkownicy chmury operujący na jej różnych warstwach [1]

Aby była możliwa tak wysoka elastyczność, fizyczne serwery są oddzielone warstwą abstrakcji, na której są widoczne jako **pula zasobów** takich jak przestrzeń dyskowa czy moc procesora (rysunek 1.2). Każdy program

czy serwer wirtualny działający w chmurze osadzany jest na wirtualnych zasobach; nie ma możliwości zdecydowania z którego konkretnego fizycznego zasobu chce się korzystać. Moc chmury obliczeniowej buduje się przez łączenie tanich, łatwo wymienialnych komponentów sprzętowych, w potężne zasoby wirtualne. [1]



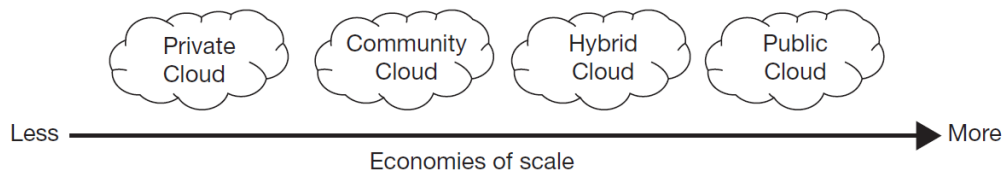
Rysunek 1.2: Serwery wirtualne korzystające z puli zasobów [1]

Poza rozwiązaniem problemu zarządzania ogromną ilością fizycznych serwerów, wirtualizacja zasobów pozwala na **lepsze wykorzystanie sprzętu**. W tradycyjnym modelu komputery muszą być przygotowane na wypadek gdyby zainstalowane na nich programy powodowały większe zużycie zasobów. Dzieje się tak nawet w przypadku komputerów PC – kupuje się specjalnie większe dyski i lepszy procesor, aby przydały się w przyszłości. W ten sposób wykorzystuje się niewielką część możliwości pojedynczego komputera, ponieważ przez większość czasu potrzeba mu znacznie mniej zasobów, niż fizycznie posiada. W przypadku wirtualnych zasobów, do fizycznej jednostki można dynamicznie przypisać zużycie powodowane przez wielu użytkowników, wiele wirtualnych systemów operacyjnych, w zgodzie z ustalonym algorytmem. Algorytm może definiować, że np. wszystkie komputery mają zostać obciążone po równo (round robin), czy że pojedynczy węzeł ma być wykorzystany w 100% [3]. Technikę tę nazywa się **równoważeniem obciążenia** (ang. *load balancing*).

Technika zrównoważonego obciążenia przynosi kilka ważnych korzyści, będących istotnymi cechami chmur obliczeniowych. Dzięki niej usługi działające w chmurze mogą być uruchomione na różnych węzłach, w tylu instancjach, ile wymagane jest do prawidłowego obsłużenia ruchu. W przypadku awarii którejś z instancji, użytkownicy usługi niczego nie odczuwają, gdyż zostają przekierowani na instancję co do działania której nie wykazano błędów. Wszystko to ułatwia tworzenie wysoce skalowalnego oprogramowania. Nie bez znaczenia jest także pozytywny wpływ na środowisko naturalne, ponieważ chmury obliczeniowe oznaczają optymalne zużycie istniejącego sprzętu komputerowego, więc nie trzeba produkować go więcej niż potrzeba ani niepotrzebnie zużywać energii elektrycznej.

Najbardziej spektakularne efekty wykorzystywania modelu chmury wiadać w przypadku dostawców posiadających największe centra danych na świecie, takich jak Amazon, Microsoft, Google czy IBM. Na kilkudziesięciu  $m^2$  gromadzą zasoby komputerowe, których równocześnie mogą używać miliony osób na całym świecie. Na jednym fizycznym komputerze zasoby mogą być wykorzystywane przez wiele osób niewiedzących o sobie nawzajem (po angielsku tę właściwość określa się jako *multi-tenancy*). Chmury o takiej architekturze mają potencjał ucieleśnić ideę, zgodnie z którą zasoby komputerowe mogą być dostarczane jako usługa użyteczności publicznej.

Wyżej wymieniony sposób myślenia o chmurze jest tym, co wyraźnie odróżnia chmury publiczne od chmur prywatnych. Nic nie stoi na przeszkodzie, aby samemu stworzyć prywatną sieć opartą o model chmury. Wówczas formalnie jest to chmura, lecz nie zapewnia niektórych ważnych korzyści, głównie przez konieczność samodzielnego administrowania serwerami, mniejszą ilość potencjalnych użytkowników oraz mniejszy potencjał puli zasobów. Przykładowo, jeśli firma nadal musi martwić się o zarządzanie fizycznymi jednostkami komputerowymi, to nie będzie miała możliwości wychwalać modelu chmury za brak konieczności zatrudniania specjalistów zajmujących się infrastrukturą komputerową.



Rysunek 1.3: Ekonomia skali rosnąca wraz z ilością użytkowników chmury [1]

## Korzyści i wady

Do głównych korzyści wynikających ze stosowania modelu chmury należą:

- **rozwiązanie problemu skalowania aplikacji**

Wraz z rozwojem Web 2.0, dostępnością szybkiego łącza internetowego, zwiększaniem ilości urządzeń podłączonych do Internetu oraz ilości transmitowanych danych, istnieje rosnące zapotrzebowanie na aplikacje będące w stanie obsłużyć duży ruch. Dzięki chmurom uruchomienie wielu instancji aplikacji na różnych węzłach, dodatkowo w odmiennych wersjach, sprowadza się do opisu – ile, w jakiej wersji, w jakich proporcjach.[4] Podobnie ułatwione jest korzystanie z baz NoSQL.

- **rozwój Big Data**

Chmury są w stanie zapewnić odpowiednie środowisko do przechowywania oraz przetwarzania dużych zbiorów danych. Udostępnienie tego środowiska jako usługi znacząco redukuje koszt przechowywania i analizy zbiorów danych, dzięki czemu próg wejścia w owo zagadnienie staje się dużo niższy. Za tym idzie wydajniejsze odkrywanie wiedzy z danych, co może mieć wpływ m.in. na rozwój medycyny.

- **ochrona środowiska**

W modelu chmury sprzęt komputerowy zostaje wydajniej wykorzystany, w związku z tym jest mniejsze zapotrzebowanie na nowy sprzęt. Również dzięki umieszczeniu kosztownych operacji po stronie chmury, urządzenia klienckie łączące się z chmurą nie muszą być regularnie wymieniane na silniejsze.

- **lepszą koncentracją na wybranym zadaniu**

Model chmury zwalnia użytkowników z dodatkowych czynności przy realizacji danego zadania. Przykładowo, jeśli komuś jest potrzebny serwer, to z chmurą publiczną nie musi martwić się zakupem sprzętu, zapewnieniem odpowiedniego pomieszczenia czy dostępem do Internetu. Może się zamiast tego skupić na właściwym skonfigurowaniu systemu na serwerze wirtualnym.

- **płatność tylko za rzeczywiste zużycie**

Chmura umożliwia rozliczanie na podstawie czasu używania procesora, ilości przesłanych megabajtów czy ilości danych przechowywanych w chmurze. W szczególności zmniejsza to początkowy koszt wdrażania czegokolwiek w chmurze.

- **gwarancja jakości usługi**

Każdy dostawca usług chmurowych udostępnia klientom SLA (ang. *Service-level agreements*), w którym zobowiązuje się do utrzymania określonego poziomu niezawodności usług (np. dostępność w 99,9% przypadków) oraz przewidywanego zachowania w przypadku niedotrzymania zapewnień (np. obniżki cen).

Ostatnią korzyścią, a jednocześnie kontrowersją, jest bezpieczeństwo danych w chmurach publicznych. Wysyłanie niejednokrotnie wrażliwych danych w nie do końca określone miejsce budzi obawy. Według autorów specjalistycznych wydawnictw[1, 2] obawy te są niesłuszne – pozostają zgodni

co do opinii, że główni dostawcy usług chmurowych są w stanie zapewnić najwyższy poziom bezpieczeństwa, a fizyczne przechowywanie danych na terenie firmy daje tylko ułudę bezpieczeństwa.

Do potencjalnych wad rozwiązań bazujących na chmurach obliczeniowych należą:

- **problem z przenoszeniem aplikacji z jednej chmury na drugą**

Znaczna część usług oferowanych przez największych dostawców usług chmurowych jest dedykowana dla tworzonej przez nich chmury. Oznacza to, że jeśli użytkownik chce mieć możliwość zmiany dostawcy, powinien uwzględnić to przy wyborze narzędzi pracy. Więcej na ten temat można przeczytać w rozdziale 2.

- **problemy prawne**

Istnieje ryzyko rozbieżności pomiędzy prawem, do którego dostosowywał się dostawca usług chmurowych, a prawem chroniącym dane w kraju, na terenie którego chce się korzystać z tych usług.

## 1.2. Ewolucja Cloud Computingu

Cloud Computing nie stanowi odrębnej, nowej technologii – jest to produkt ewolucji technologii rozwijanych od blisko 50 lat. Prześledzenie historii pozwala na zrozumienie, które koncepcje rzeczywiście wprowadzają nową jakość, a nie są jedynie od dawna istniejącą technologią, tyle że ubraną w ładnie brzmiące słowo.

Historia ma początek około lat 70., gdy używano wielkich superkomputerów zwanych *mainframe*. Można było z nich korzystać za pomocą terminali, określanych mianem „głupich” (ang. *dumb terminal*), ponieważ nie posiadały procesora i mogły być używane jedynie do operacji I/O. Jako że z serwera mógł korzystać jeden terminal naraz, serwer ustawiał je w kolejkę i musiały długo czekać na obsłużenie.

W latach 70. terminale zaczęły być wyposażone w mikroprocesory i być określane mianem „**inteligentnych terminali**” (ang. *intelligent terminal*). Mogły już partycypować przy uruchamianiu programów, co skróciło czas obsługi i dało początek modelowi klient-serwer.

Dalszy rozwój mikroprocesorów dał początek **komputerom PC**. Komputery te mogły działać samodzielnie i były znacznie tańsze od mainframe’a.

Pojawiły się również LAN (ang. *local area network*) i WAN (ang. *wide area network*), co umożliwiło łączenie komputerów PC w sieci bez konieczności łączenia z mainframe’em. Powstały sieci **P2P** (ang. *Peer-To-Peer*).

We wczesnych latach 80. wynaleziono **systemy rozproszone**. Ponieważ były w stanie przetwarzać dane równolegle, zachwiało to poglądem iż większą moc obliczeniową należy uzyskiwać przez wynajdowanie silniejszych procesorów. Systemy rozproszone wymagały wzmożonej ilości operacji komunikacji, lecz występowało to w parze z rozwojem LAN osiągającym przepustowość 100 Mbps oraz WAN osiągającym 64 kbps.

Następnie powstała koncepcja **klastrów komputerowych**. Klastery polegały na połączeniu komputerów tego samego typu (homogenicznych) w sieć LAN i wyłonieniu wśród nich zarządcy, który będzie zajmował się przydzielaniem zadań pozostałym węzłom. Gdyby któryś z węzłów miał awarię, to inne mogły przejąć jego zadanie. Dało to początek idei **puli zasobów**. W klastrze niezawodność osiągało się przez nadmiarowość zasobów.

Główną wadą klastrów była konieczność powierzenia zarządzania klastrem jednemu komputerowi. Stanowił on miejsce, od niezawodności którego zależała niezawodność całej sieci (ang. *single point of failure*). Rozwiązaniem okazały się **gridy**, w których każdy węzeł posiadał równy priorytet. Klient mógł podłączyć się do dowolnego komputera w gridzie, a ponadto komputery te mogły być różnego typu (heterogeniczne). Wkrótce gridy przeniknęły ze świata naukowego do świata biznesu i zaczynały być łączone przez WAN.

Dotychczas gdy uruchamiano się program na wybranym węźle gridu, to znajdował się na tym węźle dopóki proces nie został zakończony. Stanowiło to problem na drodze do skalowania w czasie rzeczywistym, gdzie na skutek zmienionych potrzeb byłoby dobrze mieć możliwość od nowa zaalokować zasoby bez zakłócania działającej usługi. Problem ten rozwiązała **wirtualizacja sprzętowa**, za pomocą której zadania były alokowane do maszyn wirtualnych. Technika ta umożliwiła **utility computing**, czyli model, w którym zasoby komputerowe mogą być dostarczane jako usługa, co stanowi serce chmur obliczeniowych. Prekursorem została firma Salesforce.com, która udostępnia oprogramowanie w formie usługi internetowej od 1999 r.

W 2001 r. firma IBM zbudowała pierwszy **autonomiczny system** (ang. *autonomic computing*) – system, który potrafi sobą zarządzać bez interwencji człowieka. Dzięki wykorzystaniu sztucznej inteligencji do działania, eliminuje się ryzyko związane z błędem ludzkim oraz złożoność związaną z koniecznością doglądania przez człowieka złożonego systemu. IBM wytyczył następujące cechy modelu:



Rysunek 1.4: System bez wirtualizacji [5]



Rysunek 1.5: Wirtualizacja sprzętowa [5]

- automatyczna konfiguracja – konfiguracja tworzy się automatycznie na podstawie zapotrzebowania,
- automatyczna naprawa błędów – system sam wykrywa błędy i reaguje na nie,
- automatyczna optymalizacja – system sam dba o optymalne użycie zasobów,
- automatyczna ochrona – system wykrywa próby ataków i zapobiega im.

Wirtualizacje w tym systemie działają zgodnie z „pętlą adaptacyjną”: obserwuj, decyduj, zareaguj.[5]

Duże znaczenie miało także wynalezienie metody **SOA** (ang. service oriented architecture), czyli techniki wytwarzania oprogramowania przez wydzielanie niezależnie działających komponentów, składających się razem na większy system informatyczny. Ważna była również koncepcja **Web 2.0**, nazwana tak po raz pierwszy w 2002 roku. Model Web 2.0 spowodował zmiany w sposobie myślenia o Internecie, zgodnie z którymi nie powinien jedynie służyć do dostarczania statycznej treści, lecz może być użyteczny również do udostępniania treści tworzonej przez użytkowników Internetu.

Wymienione wyżej technologie zestawione razem dały podwaliny chmurom obliczeniowym.

Przyjmuje się, że termin „*cloud computing*” po raz pierwszy został użyty przez CEO firmy Google, Erica Schmidta, w trakcie konferencji w 2006 r. Tego samego roku Amazon użył tej nazwy publikując pionierską w swoim gatunku usługę Elastic Compute Cloud (EC2), czyli możliwość wynajmu serwera wirtualnego w ich chmurze.

### 1.3. Historia najnowsza. Współczesne wyzwania

Odkąd Amazon uruchomił usługę EC2, nastąpił gwałtowny rozwój rozwiązań chmurowych. Praktycznie w tym samym czasie rozwinęły się technologie postrzegane w czasie pisania niniejszej pracy jako rewolucyjne. Na potrzeby opracowania wyróżniono następujące grupy zmian:

- rozwój chmur publicznych (sekcja 1.3.1),
- rozwój rozwiązań opartych o kontenery (sekcja 1.3.2),
- rozwój technik programistycznych wyznaczających najlepsze praktyki tworzenia aplikacji dla chmury (sekcja 1.3.3),
- rozwój technologii bazodanowych dla systemów rozproszonych oraz metod analizy danych (sekcja 1.3.4),
- rozwój technologii przyspieszających dostęp do usług chmurowych (sekcja 1.3.5).

#### 1.3.1. Rozwój chmur publicznych

W 2003 roku firma Citrix stworzyła Xen – system operacyjny przeznaczony tylko do wirtualizacji innych systemów operacyjnych. Niedługo potem Xen został udostępniony jako oprogramowanie otwartoźródłowe. W marcu 2006 r. Amazon na podstawie Xena opracował swój produkt EC2. W ciągu 18 miesięcy zaczęło z niego korzystać ponad pół miliona osób.[2]

W kolejnych latach powstały kolejne chmury publiczne oraz nastąpił rozwój rozwiązań typu PaaS (ang. *Platform as a Service*). W czerwcu 2007 r. powstał Heroku, będący pionierem w tej kategorii. Niecały rok później Google udostępnił usługę Google App Engine. W lutym 2010 r. Microsoft udostępnił Windows Azure (w kwietniu 2014 przemianowaną na Microsoft Azure [6]).

Do tego momentu powstały już 3 największe chmury publiczne: Amazon AWS, Google Cloud Platform, Microsoft Azure. Początkowo miały zróżnicowaną ofertę. W literaturze[2] z 2011 r. porównywanie tych platform sprowadzało się do zakresu oferowanych przez nie usług. Z biegiem czasu różnice zatarły się – obecnie usługa publikowana przez jednego dostawcę po kilku miesiącach jest dostępna także u innych. Tabela 1.1 prezentuje porównanie tempa rozbudowy zakresu usług.



Typ usługi	Dostawca		
	AWS	GCP	Azure
serwery wirtualne	2006 – Amazon EC2 [1]	2013 – Google Compute Engine [41]	2010 – Azure Virtual Machines [38]
platforma aplikacji	2011 – AWS Elastic Beanstalk [39]	2008 – Google App Engine [41]	2010 – Azure Cloud Services [38]
bazy SQL	2009 – Amazon Relational Database Service [40]	2011 – Google Cloud SQL [34]	2010 – Azure SQL Database [38]
bazy klucz-wartość	2007 – Amazon SimpleDb (w 2012 Amazon DynamoDB) [42]	2008 – Google Bigtable [33] (jako część App Engine)	2010 – Azure Table Storage [38] (2017 – Azure Cosmos DB)
serverless	2014 – AWS Lambda [7]	sierpień 2018 – Google Cloud Functions [43]	2016 – Azure Functions [7]
Kubernetes	2018 – Amazon Elastic Container Service for Kubernetes [36]	2015 – Google Kubernetes Engine [35]	2017 – Azure Container Service [37]

Tablica 1.1: Porównanie dat udostępniania usług chmurowych wśród głównych dostawców

### 1.3.2. Konteneryzacja i DevOps

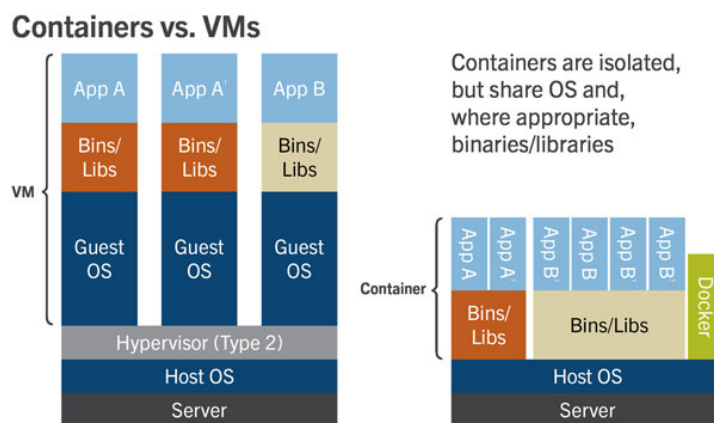
W tym samym czasie rozwijały się technologie przeznaczone do wystawiania własnych usług chmurowych. W 2010 r. powstała otwartoźródłowa platforma **OpenStack** służąca do świadczenia własnych usług typu IaaS. Rok później powstał **Cloud Foundry** – również otwartoźródłowy projekt, pozwalający na prowadzenie własnej platformy typu PaaS do tworzenia aplikacji w chmurze.

Cloud Foundry stanowił odpowiedź na główny problem chmur publicznych – tworzenie aplikacji na wybraną chmurę wiązało ją z mechanizmami

specyficznymi dla tej chmury. Brakowało pomostu, który oddzieliłby warstwę tworzenia aplikacji w chmurze od wybranego usługodawcy, bez konieczności tworzenia serwerów wirtualnych, na których trzeba instalować określone środowisko. [3]

Cloud Foundry jest niezależny od infrastruktury, może działać zarówno na OpenStacku jak i na serwerze wirtualnym w chmurze publicznej. Zapewnia możliwość wyboru dowolnych technologii do wytwarzania oprogramowania. Do działania używa kontenerów zgodnych ze standardem OCI (ang. *Open Container Initiative*). [3]

**Konteneryzacja** to metoda wirtualizacji na poziomie systemu operacyjnego, gdzie na działającym systemie operacyjnym można uruchomić wiele odizolowanych od siebie instancji przestrzeni użytkownika. Każda taka instancja to kontener. Kontener zapewnia odpowiednie środowisko do działania dla umieszczonej w nim aplikacji, np. środowisko uruchomieniowe .NET Core w odpowiedniej wersji. [3, 5]



Rysunek 1.6: Porównanie serwerów wirtualnych i kontenerów [5]

Cloud Foundry zapewnia orkiestrację kontenerów oraz obsługę procesu CI/CD (ciągłej integracji i ciągłego dostarczania, ang. *continuous integration / continuous delivery*). [3] Stanowi narzędzie pozwalające z powodzeniem wdrażać strategię DevOps.

**DevOps** to słowo po raz pierwszy użyte 2009 r. na konferencji w Belgii. Stanowi połączenie ze sobą słów „*development*” (rozwój) i „*operations*” (eksploatacja). Wskazuje to na zacieśnianie współpracy pomiędzy zespołami zajmującymi się rozwojem oprogramowania, a zespołami, które je wdrażają. Współpraca ta ma doprowadzać do automatyzacji procesów towarzyszących wytwarzaniu oprogramowania, co w efekcie minimalizuje czas potrzebny od wprowadzenia poprawki do wdrożenia na środowisko produkcyjne, zachowując przy tym niezawodność. Jako strategię DevOps określa się wszelkie

technologie i narzędzia inżynierii oprogramowania, które dają taki efekt. [8]

Temat DevOps jest mocno powiązany z chmurami, ponieważ jego koncepcje znacząco redukują złożoność towarzyszącą tworzeniu wysoce skalowalnego oprogramowania. Jednocześnie wychodzą one naprzeciw wymaganiom społeczeństwa coraz bardziej przyzwyczajonego do częstych aktualizacji oprogramowania, szybkiej reakcji na błędy oraz braku przestojów towarzyszących wdrażaniu nowych wersji. W przypadku chmur obliczeniowych, w których jedna usługa internetowa może mieć np. kilkadziesiąt instancji na różnych węzłach, taka automatyzacja ma kluczowe znaczenie. W związku z tym pojęcie DevOps obecnie wchodzi w kanon pojęć związanych z chmurą.

Istotny element DevOps stanowią wcześniej wspomniane kontenery. Idea kontenerów kielkowała już kilka lat wcześniej, np. Solaris Containers z 2004 r. lub OpenVZ (2005 r.) i LXC (Linux Containers – 2008 r.), które w istocie do działania wykorzystywały UNIX-owy program chroot z 1982 r.[44] Jednak prawdziwe ożywienie spowodowało dopiero pojawienie się Dockera – sprawił iż tworzenie kontenerów oraz przenoszenie ich pomiędzy systemami stało się łatwe. Na nim bazuje kilka projektów intensywnie rozwijanych w czasie pisania niniejszej pracy.

**Docker** to system kontenerów, który powstał jako element platformy dotCloud, świadczącej usługi typu PaaS. Spopularyzował się w dość osobliwy sposób. Założyciel myślał, że tylko wygłosi mały referat na PyCon w 2013 r., lecz okazało się, że na konferencji było dużo osób, a jego referat stanowił główny element programu. Do końca roku Docker miał 100 mln pobrań, 3 miesiące później – 300 mln, a w czerwcu 2017 – 13 miliardów.[45] Z kolei platforma dotCloud nie okazała się sukcesem – w 2014 r. Docker Inc. sprzedał ją niemieckiej firmie cloudControl GmbH, która w grudniu 2015 r. zbankrutowała, a w marcu 2016 r. zamknęła usługę.[46]

Logo Dockera dobrze opisuje jego istotę. Przedstawia wieloryba transportującego kontenery. Miało to nawiązywać do tego, iż załadowanie statku trwa dłużej niż transport statkiem. Ma to swoją alegorię do programowania. W tradycyjnym podejściu, gdyby zechcieć uruchomić np. aplikację napisaną w ASP.NET, najpierw trzeba byłoby poświęcić kilka godzin na pobieranie i instalację narzędzi: Visual Studio z narzędziami sieci Web, serwer IIS oraz SQL Server LocalDB. Natomiast w podejściu Dockera, posiadając zbudowany obraz można by tę aplikację uruchomić od razu. Razem z krokami budowania obrazu byłoby to:

1. napisanie pliku Dockerfile z definicją zależności: system Windows, MSBuild, .NET Framework, serwer IIS, menadżer pakietów NuGET,

2. użycie CLI Dockera oraz pliku Dockerfile do zbudowania tzw. obrazu Dockera,
3. użycie obrazu Dockera do zbudowania i uruchomienia aplikacji w kontenerze.

Obrazy Dockera działają wszędzie, gdzie jest zainstalowany Docker – niezależnie od systemu operacyjnego czy środowisk jakie ma zainstalowane. Ponadto zależności takie jak np. system operacyjny Ubuntu występują w wersjach „odchudzonych” – przykładowo obraz Minimal Ubuntu 18.04 zajmuje jedynie 29 MB.[47]. Najlżejszym obrazem Linuxa jest dystrybucja Alpine wielkości niecałych 4 MB, stąd używana jest jako baza dla obrazów zapewnianych przez Docker.[11] Podejście Dockera stwarza następujące możliwości:

- Na jednej maszynie może jednocześnie działać wiele aplikacji wymagających różnych zależności, lub nawet różniących się jedynie wersjami, w izolacji od siebie oraz maszyny, na której są uruchomione.
- W procesie wytwarzania oprogramowania gwarantuje użycie identycznych wersji zależności na środowisku deweloperskim, testowym oraz produkcyjnym. Eliminuje to problem, gdy na zgłoszony błąd słyszy się odpowiedź „u mnie działa”.
- Obrazy Dockera są lekkie: współdzielą wiele zasobów systemu operacyjnego, dzielą między sobą inne obrazy Dockera.[48] Na jednej maszynie może działać wiele kontenerów Dockera, podczas gdy maszyn wirtualnych może być zaledwie kilka.[5]

Powyższe znacząco ułatwia tworzenie aplikacji „dla chmury”, orkiestrację chmury oraz stosowanie strategii DevOps.

W 2014 r. Google udostępnił system **Kubernetes**, służący do orkiestracji kontenerów w chmurze. Wywodzi się z projektu Borg, będącego wewnętrznym narzędziem Google do obsługi 2 miliardów[10] kontenerów tygodniowo, dopracowywanym w tej firmie od ponad dekady.[49] Kubernetes jako główny system kontenerów wykorzystuje Dockera.[4] Kubernetes może zostać skonfigurowany zarówno na fizycznych serwerach, jak i na serwerach wirtualnych (można też je mieszać).[9]

Na pierwszy rzut oka Kubernetes przypomina Cloud Foundry, jest to jednak zupełnie inne narzędzie. Tabela 1.2 prezentuje porównanie tych dwóch platform.

Cloud Foundry	Kubernetes
powstał w 2011 r.	powstał w 2014. r.
wysoki poziom automatyzacji – rozpoznaje zależności aplikacji, sam buduje kontener i konfiguruje skalowanie aplikacji	więcej kontroli nad kontenerami, lecz także więcej pracy dla deweloperów. Dla większej automatyzacji należy sięgnąć po rozwiązania PaaS bazujące na Kubernetes, takie jak RedHat OpenShift [10]
minimalne wymagania – 40 GB HDD, 8 GB RAM, 4 rdzenie CPU	2 GB RAM, 2 rdzenie CPU. Na tyle lekki, że można go używać na Raspberry PI [50]
domyślnie używa własnego systemu kontenerów, ale pozwala na korzystanie z obrazów Dockera	stosuje obrazy Dockera jako główny system kontenerów, ale pozwala zastąpić go innym

Tablica 1.2: Główne różnice pomiędzy Cloud Foundry i Kubernetes

Do tego momentu kontenery jawią się jako antidotum na przywiązanie do wybranego dostawcy usług chmurowych. Aplikacje w kontenerze Dockera mogą używać szerokiego wachlarza technologii, nie tylko otwartoźródłowych jak np. MongoDB, Node.js, .NET Core. Jako zależność można podać np. OpenJDK czy obraz systemu Windows Server udostępniany przez Microsoft, następnie można do niego doinstalować inne zależności komendą RUN w Dockerfile. [11] Dostępność platformy Kubernetes wyznaczyła standard w zarządzaniu kontenerami na szeroką skalę.

Problemem, który w wielu przypadkach mógł podnosić barierę wejścia do stosowania takich rozwiązań, była konieczność samodzielnej instalacji Kubernetes lub podobnego narzędzia na serwerze wirtualnym w chmurze. Chmury publiczne wyszły naprzeciw tym oczekiwaniom i zaczęły udostępniać usługi typu **CaaS** (ang. *Containers as a Service*), nazywane też KaaS (ang. *Kubernetes as a Service*). Jako przykłady można wymienić:

- **Google Kubernetes Engine** (wcześniej: Google Container Engine) – usługa udostępniona w sierpniu 2015 r. [35] z gotowym do użycia Kubernetes. Można ją przetestować za darmo w ramach kredytu 300 USD do testowania Cloud Platform.[51]
- **AKS – Azure Kubernetes Service** (wcześniej: Azure Container Service) – początkowo od 2015 r. wspierała orkiestratory Mesosphere

DC/OS oraz Docker Swarm, jednak na fali popularności Kubernetes firma Microsoft uznała go za standard i w 2017 r. opracowała wsparcie dla Kubernetes. W związku z tym nastąpiła także zmiana nazwy usługi. [37] Usługa jest darmowa[52], opłaty są naliczane za użyte maszyny wirtualne, które przez rok również w ograniczonym zakresie są bezpłatne.[53]

- **Amazon ECS – Amazon Elastic Container Service** – usługa dostępna od kwietnia 2015 r., jest to własna implementacja orkiestracji kontenerów Dockera stworzona przez Amazona. Pozwala na 2 typy rozliczeń: płatność za zużycie zasobów przez kontener lub płatność za przechowanie i uruchamianie aplikacji na EC2.[56] Nie występuje na liście usług dostępnych do wypróbowania za darmo.[57]
- **Amazon EKS – Amazon Elastic Container Service for Kubernetes** – usługa została uruchomiona w czerwcu 2018 roku. Publikując usługę Amazon chwalił się, że na podstawie danych Cloud Native Computing Foundation (założonej wraz z powstaniem Kubernetes), 57% firm korzystających z Kubernetes robiło to na serwerach wirtualnych AWS. Płatność wynosi 0.20 USD za godzinę dla każdego klastra w usłudze, a dodatkowo za zużycie zasobów AWS.[58] Nie występuje na liście usług dostępnych do wypróbowania za darmo.[57]

Dostępność tych usług ucieleśnia wizję tworzenia aplikacji dla chmury tak, że:

- Do tworzenia aplikacji można wybrać dowolną technologię i wciąż koncentrować się na tworzeniu tej aplikacji, a nie zapewnianiu jej odpowiedniego środowiska na serwerze wirtualnym.
- Istnieje standardowy sposób tworzenia obrazu kontenera (Docker).
- Istnieje standardowy sposób orkiestracji kontenerów (Kubernetes). Wiedzę na temat korzystania z niego można zastosować do projektów działających na chmurach publicznych, prywatnych i hybrydowych.

#### 1.3.3. Mikrouslugi i serverless

Gwałtowny rozwój chmur obliczeniowych poskutkował również wytworzeniem nowych wzorców projektowych, wzorców architektonicznych oraz dobrych praktyk używanych podczas tworzenia aplikacji przeznaczonej dla chmury. W chmurze może działać praktycznie każdy program, wszakże odpowiednio

przygotowana maszyna wirtualna wszystko przyjmie. Chcąc jednak maksymalnie wykorzystać atuty chmury, a jednocześnie zapewnić niezawodność i jak najniższy koszt eksploatacji, należy zapoznać się ze specyficznym sposobem tworzenia aplikacji dla chmury. Aplikację zgodną z tymi wytycznymi określa się jako *cloud native*.

Jak zostało wspomniane wcześniej, platforma Heroku była pionierem jeśli chodzi o udostępnianie programistom platformy do tworzenia aplikacji w chmurze (2007 r.). Współzałożyciel Heroku, Adam Wiggins wraz z zespołem, w 2012 r. na podstawie doświadczeń zebranych w Heroku stworzyli metodykę „*twelve-factor*” (ang. *The Twelve-Factor App*). Jest to zbiór podstawowych 12. reguł dotyczących programowania oraz pracy z aplikacjami przeznaczonymi dla chmury. Reguły te są często przywoływane w literaturze zajmującej się tym tematem.[3, 12] Można się z nimi zapoznać odwiedzając stronę <https://12factor.net> (w czasie pisania pracy strona była dostępna w 13 językach, w tym w języku polskim).

Osobne „dobre praktyki” powstały dla systemów rozproszonych używających kontenerów (w tym Kubernetes). Dzielą się na:

- Wzorce dla kontenerów występujących na jednym węźle (ang. *Single-node patterns*). Należą do nich wzorce: *Sidecar*, *Ambassador*, *Adapter*.
- Wzorce dla kontenerów występujących na różnych węzłach (ang. *Multi-node patterns*). Koncentrują się na właściwych metodach koordynacji działań pomiędzy różnymi maszynami. Są to wzorce np. komunikacji z użyciem zdarzeń czy kolejek. [9, 13]

Praktyki wyznaczone przez powyższe wzorce w dużej mierze opierają się na **mikrouślugach**. Mikrouślugi wywodzą się z **SOA**. Termin SOA (ang. *Service oriented architecture*) został pierwszy raz użyty przez analityka firmy Gartner podczas wykładu. Utworzył on ten termin, ponieważ zwrot „klient/serwer” tracił na swoim pierwotnym znaczeniu, gdyż zamiast nazywać tak aplikację dla serwera lub klienta, ludzie zaczęli tak nazywać fizyczne maszyny. Następnie inni analitycy w 1996 r. publikowali na ten temat raporty. Bardziej znaczące użycie terminu SOA należało do Microsoftu. W 2000 r. opisywał zbiór standardów do komunikacji komputerów przez Internet, gdzie przedstawił architekturę SOA jako ważną, choć nie niezbędną, dla Web Services. Wkrótce termin podchwyciły inne firmy, w tym takie jak IBM, Oracle, HP, publikując nowe koncepcje czy narzędzia powiązane z SOA. [14]

SOA to po polsku architektura zorientowana na usługi. Polega na podzieleniu systemu informatycznego na odrębne części, które mogą znajdować się na różnych maszynach i komunikować się ze sobą przez sieć. Główne koncepcje SOA to:

- usługi – rozumiane jako samodzielne komponenty realizujące daną funkcjonalność biznesową,
- wysoka interoperacyjność – łatwe łączenie ze sobą systemów różnych typów,
- luźne wiązania – jak najmniejsza liczba zależności. [14]

SOA to bardzo ogólne pojęcie, nie definiuje jakiej wielkości powinny być poszczególne usługi. Wraz z rozwojem technologii chmurowych, nastąpiła potrzeba dostosowania paradygmatu SOA do wyzwań związanych ze skalowalnością, szybkością wytwarzania oprogramowania oraz potrzebą adaptacji nowych technologii dla istniejących rozbudowanych systemów. Odpowiedzią na ten problem okazały się mikrousługi.

**Mikrousługi** (ang. *microservices*) stanowią szczególny przypadek SOA. Ich główna cecha: mają być małe. Zamiast tworzyć jedną aplikację, która robi wszystko, system ma składać się z wielu małych, niezależnych od siebie komponentów, odpowiedzialnych za tylko jedną rzecz. [12]

Z jednej strony powinny grupować powiązane ze sobą funkcjonalności, z drugiej strony nie wolno doprowadzać do sytuacji, gdy serwis może stać się zbyt duży i trudny w utrzymaniu. Pomocne w wyobrażeniu jakiej wielkości powinna być przeciętna mikrousługa są następujące wskazówki:

- Zasada pojedynczej odpowiedzialności, wywodząca się z zasad SOLID, a zaadoptowana do usług. Polega na umieszczaniu razem elementów, które będą zmieniać się z tego samego powodu. [16]
- Kod mikrousługi powinien być na tyle mały, żeby dało się go przepisać w dwa tygodnie. [16]
- Mikrousługa powinna być na tyle mała, żeby do pracy z nią wystarczył jeden zespół programistów. [15]
- W przypadku gdy mikrousługa jest często używana i potrzebuje do działania wielu instancji, nie powinna jednocześnie powielać funkcjonalności rzadziej używanych i tym sposobem marnować zasobów. [15]



W odniesieniu do chmur, mikrousługi wprowadzają następujące udogodnienia: [12, 15, 16]

- **Efektywne skalowanie.** Gdy określona funkcjonalność systemu jest często używana, uruchamia się więcej instancji mikrousługi odpowiedzialnej za tę funkcjonalność. Nie trzeba zużywać więcej zasobów niż jest to potrzebne.
- **Lepsze zapanowanie nad kodem.** Do mniejszego programu łatwiej wprowadzać poprawki, w razie potrzeby przepisanie go do nowszych technologii nie powinno stanowić problemu. Każda mikrousługa może być napisana przy użyciu innych technologii. Większe zmiany w kodzie oddziałują tylko na daną mikrousługę, nie na cały system.
- **Szybsze wdrażanie.** Procesy ciągłej integracji i ciągłego dostarczania (CI/CD) szybciej działają, gdy mają do kopiowania, budowania i testowania mniejszą ilość kodu.

Popularność kontenerów, mikrousług oraz modelu *utility computing* (rozliczanie za rzeczywiste użycie zasobów komputerowych) doprowadziła do wytworzenia nowego paradygmatu tworzenia aplikacji w chmurze, znanego jako **serverless** (z ang. „bez serwera”). Serverless jest obecnie obiecującą nowością jeśli chodzi o przetwarzanie w chmurze. Nazwa ta zaczęła zdobywać popularność na początku 2016 r. [17]

Aplikacje tego typu zamiast być przez cały czas uruchomione na serwerze w oczekiwaniu na zapytania, mają być uruchamiane dopiero gdy przyjdzie określone zdarzenie (ang. *event-driven computation*). Zdarzeniem może być np. nowe zapytanie HTTP, nowy wpis w kolejce, wywołanie po upływie interwału czasowego. Wówczas zostaje powołana do życia aplikacja przeznaczona tylko do obsługi danego zdarzenia i działa tylko na krótki czas obsługi zdarzenia (jest to maksymalnie około kilka minut). Aplikacje te to w istocie mikrousługi, tyle że posiadające tylko jedną funkcję – stąd nazywa się je po prostu funkcjami. Dla lepszego odróżnienia od mikrousług określa się je jako „nanousługi” (ang. *nanoservices*). [7, 17, 13]

Platforma w chmurze stosująca model serverless określana jest jako **FaaS** (ang. *Function as a Service*). Pierwszym znaczącym dostawcą tej usługi był Amazon – już w 2014 r. uruchomił usługę AWS Lambda typu FaaS. W 2016 r. Microsoft udostępnił usługę Azure Functions. W Google Cloud Platform tego typu usługa została oficjalnie udostępniona w sierpniu 2018 (miesiąc przed wydaniem niniejszej pracy), choć wersja poglądowa była dostępna wcześniej.

Posługując się modelem serverless, usługodawca może stosować rozliczenie według ilości wywołań poszczególnych funkcji. Dzięki temu nie trzeba płać za programy beczynnie czekające na przychodzące zapytania.

Dla programisty użycie nanoserwisów oznacza mniej projektowania, jeśli chodzi o optymalną architekturę mikrousług. Mikrousługi grupowały ze sobą niektóre elementy, natomiast w nanousługach każda funkcja stanowi osobny byt. Według Maddie Stigler, autorki książki „*Beginning Serverless Computing*”, użycie architektury serverless oszczędza połowę czasu potrzebnego na stworzenie skalowalnych aplikacji od podstaw.[7]

Rozwijane są także inicjatywy niezwiązane z usługami FaaS konkretnych dostawców. W 2015 r. powstał projekt Serverless Framework,[59] służący do odseparowania logiki konkretnej usługi FaaS od logiki biznesowej funkcji. Ponadto zapewnia narzędzia wspierające proces tworzenia funkcji.

W 2017 r. powstał projekt Kubeless.[60] Jest to otwartoźródłowy framework dla Kubernetes, który ma działać w sposób podobny do sposobu wyznaczonego przez AWS Lambda, Azure Functions oraz Google Cloud Functions.[13]

Serverless Framework obsługuje wszystkie wyżej wymienione platformy, z Kubeless włącznie.[59]

#### 1.3.4. Bazy NoSQL, NewSQL i rozwiązania dla Big Data

W literaturze rozróżnia się trzy rewolucje w rozwoju technologii bazodanych. [18] W niniejszej sekcji zostaną przytoczone wszystkie trzy, ponieważ pokazuje to, jak kiedyś wyglądały bazy nierelacyjne (teraz postrzegane jako nowość), jak bardzo długo nie było potrzeby rezygnowania z relacyjnych baz danych, oraz jak na tym tle prezentuje się tempo rozwoju współczesnych baz danych.

Za pierwsze „bazy danych” uznaje się wszelkie techniki zapisu i odczytu danych z nośników, na które pozwalała technika – kart perforowanych, taśm magnetycznych, magnetycznych dysków twardych. Przed pierwszą rewolucją, każda aplikacja posiadała swoją własną implementację dostępu do danych. Nie istniały żadne mechanizmy chroniące przed uszkodzeniem danych, przed równoczesnym dostępem wielu użytkowników, nie wspominając o optymalizacjach.

Pierwszą rewolucją było stworzenie **systemu bazodanowego (DBMS** – ang. *Database Management System*). Wczesne systemy nie opierały się

na żadnych podstawach matematycznych. Były to tzw. bazy nawigacyjne, gdzie relacje uzyskiwało się definiując połączenia pomiędzy obiektami. Struktura projektu definiowała jakie będzie można tworzyć rodzaje zapytań do bazy. Przykłady tych wczesnych baz stanowią: IMS od firmy IBM (typ hierarchiczny) oraz IDMS będący implementacją standardu CODASTYL (typ sieciowy).

Drugą rewolucję zaczął Edgar Codd, będący matematykiem-programistą oraz wieloletnim pracownikiem firmy IBM. Był on pierwszą osobą, która zwróciła uwagę na wady istniejących rozwiązań wynikające z braku podstaw matematycznych, i w 1970 r. opublikował artykuł opisujący relacyjne bazy danych. Początkowo pozostał on bez odzewu. Dopiero w 1974 r. IBM zaczął prace nad prototypem zwanym Systemem R. Prototyp był gotowy w 1976 r. Poza modelem opracowanym przez Codd'a, na potrzeby systemu stworzono język SQL do tworzenia zapytań do bazy danych oraz opracowano zasady obsługi równoległych zapytań. Jim Gray zdefiniował transakcje ACID.

Pierwszą komercyjną **relacyjną bazą danych (RDBMS - *Relational Database Management System*)** była Oracle Database. Założyciel Oracle, Larry Ellison, był zaznajomiony zarówno z pracą Codd'a, jak i Systemem R. Wierząc w relacyjne bazy danych, w 1979 r. wydał pierwszą na świecie komercyjną relacyjną bazę danych obsługującą język SQL. [19]

W tym samym czasie (od roku 1977) pionierski stał się również projekt Ingres, stworzony przez Michaela Stonebraker'a. Początkowo jako język zapytań stosował QUEL, popularność bazy Oracle wymusiła jednak konieczność obsługi SQL. W połowie lat 80. na bazie Ingres powstał projekt Postgres (post-ingres), znany dzisiaj jako PostgreSQL.

Po sukcesie Oracle IBM opublikował komercyjną bazę SQL/DS w 1981 r. W późniejszych latach powstawały inne relacyjne bazy chętnie stosowane do dziś: Microsoft SQL Server, PostgreSQL, MySQL. Wszystkie te bazy co do zasady są do siebie podobne – opierają się o model Codd'a, transakcje ACID oraz język SQL. [19]

Potencjalna rewolucja miała szanse dokonać się po latach 80., gdy zaczął popularyzować się paradygmat programowania obiektowego. Istniejące języki zaczęły występować w wersji obiektowej, np. Object Pascal, powstały języki od początku obiektowe, np. Java. Niektórym programistom nie podobała się rozbieżność pomiędzy reprezentacją danych w obiekcie, a w relacyjnej bazie danych. W grudniu 1989 r. powstał manifest obiektowych baz danych (OODBMS, ang. *Object Oriented Database Management System*), w którym bazy relacyjne jawiły się jako relikty przeszłości, a obiektowe jako

ich następcy.[61] Jednak na przestrzeni lat 90. okazało się, że bazy obiektowe nie zdobyły rynku, nawet pod postacią nowych opcji w popularnych bazach jak np. Oracle. Według Guy’a Harrison’a, autora książki „*Next Generation Databases: NoSQL, NewSQL, and Big Data*”, założenia baz obiektowych koncentrowały się wokół korzyści dla programisty, ignorując korzyści dla biznesu (np. brak wsparcia dla języka SQL będącego powszechnie w użyciu). Dodatkowo dostępność narzędzi ORM (ang. *Object-relational mapping*) znacząco ułatwiła korzystanie z RDBMS w językach obiektowych, więc bazy relacyjne przestały irytować.

#### Trzecia rewolucja

Trzecia rewolucja rozpoczęła się za sprawą coraz większego znaczenia systemów rozproszonych oraz konieczności analizy dużej ilości danych. Zaczęło się od Google. W 1996 r. powstała wyszukiwarka Google, korzystająca z algorytmu PageRank. W 2005 r. Google był największą stroną internetową na świecie, stroną zbierającą i przetwarzającą dane o wszystkich innych stronach. Potrzebowano nowych narzędzi do rozwiązania problemu, z którym jako pierwsi mieli styczność. W kolejnych latach Google publikował szczegóły swoich rozwiązań, co miało duży wpływ na rozwój dalszych technologii:

- w 2003 r. – system plików **GFS** (Google File System) dla systemów rozproszonych, gdzie pula zasobów dyskowych jest widoczna jako jedno,
- w 2004 r. – model **MapReduce** pozwalający na przetwarzanie danych w systemach rozproszonych,
- w 2006 r. – baza danych **BigTable**.

Ponadto w 2005 r. Google opatentował Modular Data Center, czyli sposób na szybką rozbudowę centrum danych. Zamiast z osobna dodawać serwery, mogli dodawać od razu po tysiąc naraz.

Rozwiązania Google zainspirowały inne projekty. W 2004 r. powstał projekt Apache Nutch, który miał być otwartoźródłową wyszukiwarką internetową. Gdy Google opublikował GFS i MapReduce, dla Nutch to był sygnał jak rozwiązywać problemy skalowalności. W 2006 r. Yahoo! zainwestował w projekt, pomimo że był otwartoźródłowy, ponieważ chciał użyć go w swoim produkcie. Adaptację rozwiązań Google wyodrębniono jako osobny projekt. Tak w 2007 r. powstał **Apache Hadoop**. [20]

Hadoop to nie tylko projekt, ale podejście do skalowalnego przetwarzania danych. [20] Obecnie Hadoop to praktycznie synonim Big Data. [21] Termin **Big Data** (z ang. „duże dane”) po raz pierwszy został użyty w 1999 r., jednak zaczął być powszechnie używany odkąd Google użył tego terminu w dokumencie opisującym MapReduce w 2004 roku. Początkowo (w 2001 r.) opisywano Big Data za pomocą „trzech V”: *volume*, *variety*, *velocity* (ilość, różnorodność, szybkość). W 2014 r. rozszerzono to do „pięciu V” o: *veracity*, *value* (wiarygodność, wartość).[22] Właściwości te wyznaczają obszar zainteresowań, gdzie zastosowanie mają takie narzędzia jak Hadoop:

- *volume* – ilość danych,
- *variety* – stopień różnorodności źródeł danych,
- *velocity* – szybkość z jaką są tworzone nowe dane oraz jak szybko należy je przetworzyć,
- *veracity* – na ile można ufać tym danym, czy źródło danych jest zanieczyszczone,
- *value* – czy dane mają jakiegokolwiek znaczenie, czy analiza tych danych tylko tworzy koszty czy daje wartość.

Powyższe oznacza, że wcale nie trzeba mieć bardzo dużego zbioru danych, aby potrzebować Big Data. [20]

Wracając do Hadoopa, rok po jego wdrożeniu Yahoo! ogłosił, że ich klaster Hadoopa ma 5 petabajtów dysku i ponad 10 000 rdzeni CPU, co generowało im cały indeks wyszukiwarki. Podobnie Facebook skorzystał z Hadoopa w 2007 r. i już w 2008 r. mieli 2500 rdzeni CPU, a w 2012 r. 100 petabajtów dysku, co wyparło użycie rozwiązań Oracle w tej firmie.

Do głównych zalet Hadoopa zalicza się:

- tanie przechowywanie danych – jest to możliwe na zwykłych dyskach,
- skalowanie operacji I/O – ponieważ zamiast dodawać tylko dyski, dodaje się nowe komputery,
- niezadowność – jeśli zepsuje się jeden komputer, to jego rolę przejmie inny,
- „*schema on read*” – struktura danych tworzy się podczas czytania źródła danych, w przeciwieństwie do tradycyjnego modelu relacyjnego, gdzie przy zapisie danych należy się zastosować do istniejącego schematu (ang. *schema on write*).

Klasyczny Hadoop posiadał ograniczenia jeśli chodzi o użycie innego modelu przetwarzania zadań niż MapReduce. W 2012 r. powstała wersja 2.0, w której dołączono platformę YARN (ang. *Yet Another Resource Negotiator* lub rekursywnie *YARN Application Resource Negotiator*). YARN pozwala na użycie MapReduce jako jednego z możliwych frameworków do zastosowania. Zamiast niego można użyć np. Sparka, pozbawionego niektórych niedogodności MapReduce (krótkotrwałość zadań, brak możliwości trzymania roboczego zbioru danych w pamięci). [20, 21]

Stworzenie BigTable oraz Hadoopa spowodowało powstanie wielu nowych systemów bazodanowych, które miały lepiej działać na systemach rozproszonych. Dzielą się na:

- **Bazy NoSQL** – bazy nierelacyjne. Zalicza się do nich wszelkie rozwiązania, które dzięki niespełnieniu niektórych reguł ACID mogą pozwolić na lepszą skalowalność i/lub szybkość przetwarzania.
- **Bazy NewSQL** – bazy posiadające możliwość skalowania, zachowujące właściwości ACID baz relacyjnych.

W 2005 r. Michael Stonebraker (założyciel Ingres i PostgreSQL) opracował bazę C-Store, postulującą **model kolumnowy**. Pomysł opiera się o tradycyjną relacyjną bazę danych, gdzie w tabeli zostają zamienione miejscami wiersze z kolumnami. Taka zamiana powoduje szybsze działanie zapytań agregujących, gdyż wszystkie potrzebne dane znajdują się obok siebie. Poza tym, ponieważ obok siebie znajdują się dane bardzo podobnej postaci, można zastosować lepszą kompresję, nawet przez traktowanie każdej kolejnej danej jako zmianę względem poprzedniej.

W 2007 r. Amazon opracował bazę Dynamo, która stała się wzorcem dla baz w modelu **klucz-wartość**. Jedną z nich jest Apache Cassandra (2008 r.), baza typu **wide-column**. W bazach tego typu w miejscu wartości jest dynamiczna liczba kolumn, co pozwala np. na przeszukiwanie po danej kolumnie lub aktualizację tylko danej kolumny, zamiast całego obiektu. Podobnym projektem jest Apache HBase (2008 r.).

Również w 2007 r. powstała pierwsza **baza dokumentowa**, która mogła być postrzegana jako alternatywa dla RDBMS. Od 2000 r. powstawały bazy XML, ale raczej mające na celu zarządzanie już istniejącymi dokumentami w tym formacie. XML rozpowszechniał się za sprawą popularności AJAX, później w tej technice zaczął być zastępowany formatem JSON. Takie dane były już formą dokumentu, a stworzenie systemu do przechowywania ich

było kwestią czasu. W 2005 r. powstała baza dokumentowa Apache CouchDB, na razie obsługująca XML, a w 2007 r. była to pierwsza znacząca baza dokumentów JSON. W 2009 r. powstała baza MongoDB, przechowująca dokumenty JSON (wewnętrznie zapisywane jako BSON – *Binary JSON*). Stanowi dziś najpopularniejszą bazę tego typu.

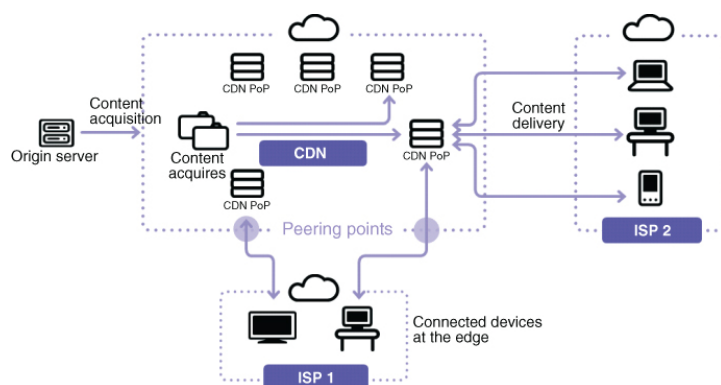
Kolejnym osiągnięciem z 2007 r. był artykuł Michaela Stonebrakera pt. „*The end of an architectural era: (it’s time for a complete rewrite)*” („Koniec współczesnych architektur, czas przepisać je na nowo” – tłumaczenie własne). Opisał tam koncepcję bazy H-store typu *in-memory*, która do transakcji nie potrzebuje wykonywać operacji I/O na dysku. Pomysł opiera się na lepszym wykorzystaniu pamięci RAM i dysków SSD. Komercyjną implementacją tego pomysłu była baza VoltDB, utworzona w 2008 roku. Jest to baza typu NewSQL.

Rok 2007 był przełomowy także dla modelu **baz grafowych** – powstała wtedy baza Neo4J. Bazy grafowe przypominają wczesne, nawigacyjne bazy danych, lecz operują na wyższym poziomie abstrakcji oraz pozwalają na przeszukiwanie grafu do zadanego poziomu. Są stworzone dla danych, gdzie najważniejszą właściwością są relacje z innymi danymi, np. relacje w social media. Do odpytywania bazy grafowej używa się języków Cypher oraz Gremlin. Neo4J stanowi bazę czysto grafową, jednak nie obsługuje systemów rozproszonych, gdyż łączenie wierzchołków grafu z różnych serwerów znacząco obniżyłoby jego wydajność. Do projektów przetwarzania grafów na systemach rozproszonych należą: Apache Giraph (2016 r.), GraphX (część Apache Spark, 2014 r.), Titan (2015 r.) – działający jako nakładka na HBase czy Cassandra.

### 1.3.5. CDN i Fog Computing

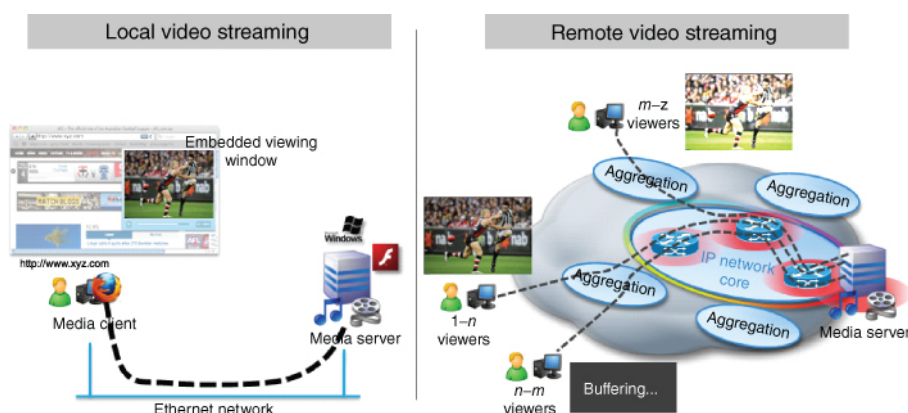
Jak było wspomniane wcześniej, chmury obliczeniowe gromadzą się co do zasady w gigantycznych centrach danych. Scentralizowany charakter chmur powoduje opóźnienia względem dostępności usług znajdujących się bliżej użytkownika. W celu zapewnienia wysokiej jakości usług (ang. QoS, *Quality of Service*) stosuje się sieci **CDN** (ang. *Content Delivery Network*). Węzły sieci CDN są szeroko rozpostarte po całym świecie, na krawędzi (ang. *edge*) dostawców usług internetowych (ang. ISP, *Internet service provider*), znacznie bliżej użytkowników końcowych niż odległość od centrów danych (rysunek 1.7). [23]

Początkowo większość treści znajdujących się w Internecie była statyczna, więc zadanie sieci CDN koncentrowało się na tworzeniu cache czę-



Rysunek 1.7: Schemat sieci CDN. [23]

sto pobieranej treści. Z czasem treść zaczęła się robić coraz bardziej dynamiczna, zaczęło się upowszechniać strumieniowanie wideo (rysunek 1.8). Spowodowało to powstanie drugiej generacji sieci CDN będącej w stanie obsłużyć te przypadki. W technologiach CDN od lat przewodzi założona w 1998 r. firma Akamai. Obecnie posiada ponad 240 000 serwerów CDN w ponad 130 krajach na całym świecie.[62] Do innych znaczących dostawców usług tego typu należą: Limelight, CloudFront (część Amazon AWS), Azure CDN. [1, 23]



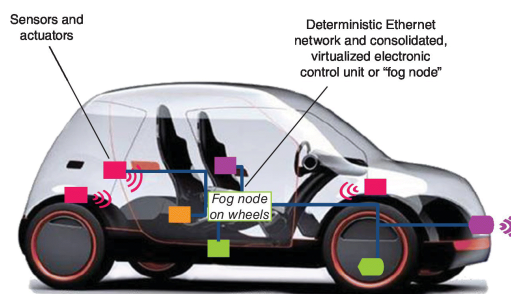
Rysunek 1.8: Optymalizacja strumieniowania wideo w sieciach CDN drugiej generacji. [23]

Sieci rozproszone CDN są oparte o model chmury (np. Akamai Intelligent Platform). Dzięki sieciom CDN strony internetowe takie jak Facebook czy Twitter (klienci Akamai), pomimo że gromadzą dane z całego świata, mogą działać w czasie rzeczywistym. [1]

Za sprawą rosnącego znaczenia **Internetu Rzeczy**, przed technologiami w modelu chmury obliczeniowej stoją kolejne wyzwania. Internet Rzeczy znajduje użycie już teraz, czego dobrym przykładem są inteligentne zegarki. Wizja *networked society* obejmuje jednak znacznie więcej, np. korzysta-



nie z samoprowadzących się samochodów (rysunek 1.9) czy wspomaganie opieki zdrowotnej przez choćby automatyczne powiadomienie rodziny o zawale serca u pacjenta. Tego rodzaju usługi wymagają braku jakichkolwiek opóźnień. Ma to zostać zapewnione przez dodatkową warstwę, która podobnie jak CDN znajdowałaby się bliżej użytkownika niż centrum danych, ale potrafiłaby także wstępnie przetworzyć dane czy też zwrócić wynik bez angażowania do tego chmury.



Rysunek 1.9: Zastosowanie Fog Computing w samoprowadzącym się samochodzie. [25]

W 2012 r. firma Cisco przedstawiła koncepcję **Fog Computing** („mgły obliczeniowej”). Mgła ma stanowić warstwę pośredniczącą pomiędzy urządzeniami IoT a chmurą. Warstwa ta ma być ulokowana na routerze danego gospodarstwa domowego. Do zadań mgły należy wstępna obróbka danych, komunikacja z chmurą (już z użyciem mniejszej ilości danych), a także obsługa urządzeń IoT nawet jeśli w danym momencie brakuje połączenia z Internetem. Przykładowo, takie zadania jak rozpoznawanie obrazów mogą odbywać się już na etapie routera, bez konieczności przesyłania danych przez Internet. Konkretną propozycją Cisco było oprogramowanie Cisco IOx. [24, 26]

Fog Computing nie stanowi jedynej koncepcji tego typu, jednakowoż brzmienie tej nazwy jest bardzo wyraziste i w literaturze wybija się na tle innych nazw. Być może z biegiem lat cała rodzina koncepcji przyspieszania technologii IoT będzie w ten sposób nazywana. Do innych koncepcji należą:

**Cyber Foraging** Pojęcie przedstawione w 2001 r. w artykule M. Satyanarayanan’a. Technika polegająca na odciążaniu urządzeń mobilnych przez przekazanie wymagających obliczeniowo zadań do serwerów-surogatów i zwrócenie temu urządzeniu wyniku z niskim opóźnieniem. W przypadku braku serwera znajdującego się wystarczająco blisko, zadanie zostaje wykonane na miarę ograniczonych możliwości urządzenia mobilnego. [24, 25]

**Cloudlet** Pojęcie przedstawione w 2009 r. w artykule współtworzonym przez Satyanarayanan’a. W odróżnieniu od Cyber Foragingu, cloudlety wykorzystują techniki wirtualizacji, co pozwala na ich skalowanie w zależności od wymagań użytkowników. Są to chmury znajdujące się bliżej użytkownika. Może to być np. chmura prywatna używana w przedsiębiorstwie. Nie ma wymogów, aby cloudlety komunikowały się dalej z chmurą publiczną. [24, 25]

**Multi-Access Edge Computing (MEC)** Koncepcja pierwotnie nazwana „Mobile Edge Computing” przez Europejski Instytut Norm Telekomunikacyjnych, gdyż była wiązana z telefonią komórkową oraz siecią radiową, lecz w 2017 r. została rozszerzona do wszystkich sieci. W tej koncepcji kładzie się nacisk na odciążanie chmur przez wykonywanie niektórych usług na krawędzi sieci (*edge*), tj. w pobliżu bramy sieciowej na różnych poziomach. Istotny element tej koncepcji stanowi upowszechnienie superszybkiej sieci 5G. [24, 25]

Obecnie można zaobserwować rozwój zarówno w kierunku Fog Computingu jak i Edge Computingu:

- W 2015 r. powstało OpenFog Consortium. W 2017 r. konsorcjum opublikowało dokument referencyjny definiujący architekturę dla systemu w modelu Fog Computing. [24, 26]
- W 2017 r. powstało oprogramowanie ParaDrop, otwartoźródłowe oprogramowanie dla routerów w modelu Fog Computing. [25]
- Główni dostawcy usług chmurowych udostępnili usługi typu *edge*: AWS Lambda@Edge (lipiec 2017) [63], Azure IoT Edge (czerwiec 2018 r.) [64], Google Cloud IoT Edge (wersja alpha, czerwiec 2018 r.) [65].

## 1.4. Wymagania stawiane współczesnym chmurom

Powszechnie korzysta się z atrybutów określonych przez amerykański Narodowy Instytut Standaryzacji i Technologii w 2011 r., zwanych **modelem NIST**. Wyróżnia on cechy zasadnicze, modele dostarczania usług i formy wdrożenia. [1, 6, 28]

### 1.4.1. Zasadnicze cechy chmury

Do cech zasadniczych należą:

**Samoobsługa** Aby skorzystać z usługi w chmurze czy zmodyfikować jej zakres, nie jest konieczna interwencja pracownika usługodawcy. Do samoobsługi służy panel administracyjny udostępniony przez usługodawcę.

**Szeroki dostęp przez sieć** Chmura powinna umożliwiać użytkownikom łączenie się przez sieć bez względu na to gdzie się znajdują i kiedy chcą skorzystać z usługi.

**Pula zasobów** Zasoby takie jak moc procesora, dyski, pamięć, mają być widoczne jako zasoby wirtualne oraz posiadać właściwość *multi-tenancy* (z jednego zasobu może korzystać wiele niezależnych osób, nie mając na to wpływu i nie wiedząc o tym).

**Wysoka elastyczność** Dostępność zasobów ma być dozowana tak, by dla użytkownika wyglądało to na nieskończoną ilość.

**Zliczanie zużycia** Dla każdego użytkownika stosowany jest mechanizm zliczania jego rzeczywistego zużycia danych usług. Zużycie jest transparentne dla obu stron, m.in. przez udostępnienie stosownych paneli służących do monitorowania i raportowania.

### 1.4.2. Modele dostarczania usług

Co do zasady istnieją 3 modele dostarczania usług:

- IaaS – Infrastruktura jako usługa (ang. *Infrastructure as a Service*),
- PaaS – Platforma jako usługa (ang. *Platform as a Service*),
- SaaS – Oprogramowanie jako usługa (ang. *Software as a Service*).

W literaturze i innych publikacjach często można spotkać się także z innymi podobnymi zwrotami w formie „... as a Service”, jednak zawsze jest to podkategoria dla tych powyższych. IaaS, PaaS i SaaS stanowią następujące po sobie warstwy. W przypadku wątpliwości związanej z klasyfikacją danej usługi, rozwiewa je schemat umieszczony na rysunku 1.10.

**IaaS** to wirtualne zasoby udostępnione jako usługa. Ponieważ wirtualizacja zasobów stanowi podstawową technikę stosowaną w modelu chmury,



Rysunek 1.10: Odpowiedzialność dostawcy usług chmurowych przy różnych ich rodzajach [26]

IaaS występuje zawsze gdy są udostępniane np. serwery wirtualne, dyski w chmurze czy sieci wirtualne. Przykładowe podgrupy to:

- NaaS (*Network* – sieć) – sieci wirtualne,
- CaaS (*Container* – kontener) – usługa konteneryzacji,
- KaaS (Kubernetes) – CaaS z wykorzystaniem Kubernetes.

Klasyfikacja usługi CaaS może uchodzić za niejednoznaczną, ponieważ IaaS kojarzy się głównie z serwerami wirtualnymi, a kontenery z tworzeniem aplikacji. Podobnie jak serwery wirtualne odpowiadają wirtualizacji z użyciem nadzorcy typu 1, tak kontenery odpowiadają wirtualizacji na poziomie systemu operacyjnego. Stąd usługi, w których zarządzanie kontenerami jest wykonywane dla użytkownika, można zaliczyć do grupy IaaS.

W **PaaS** jako platformę określa się każde oprogramowanie umożliwiające użytkownikowi (programiście) zarządzanie jego własnymi aplikacjami. Warstwa ta jest zawsze zbudowana na IaaS, gdyż owo oprogramowanie musi działać na jakimś systemie operacyjnym na maszynie wirtualnej. Warstwa PaaS obejmuje wszystko co jest potrzebne programiście, by jego aplikacja mogła działać w chmurze: wsparcie dla różnych języków programowania, systemy bazodanowe, serwer WWW. Przykładowe podgrupy to:

- FaaS (*Functions* – funkcje) – wdrażanie w modelu serverless,

- DBaaS (*Database* – baza danych) – usługa dzięki której nie trzeba samemu instalować bazy danych ani nią administrować,
- BaaS/MBaaS (*Mobile Backend*) – wspomaganie aplikacji użytkownika narzędziami służącymi np. do wysyłania notyfikacji na urządzenia mobilne.

Warstwa **SaaS** obejmuje gotowe do użycia aplikacje. Na tej warstwie cała aplikacja znajduje się pod kontrolą dostawcy usługi, użytkownik może ją jedynie skonfigurować. Jest to najbardziej dojrzały model dostarczania usługi chmurowej, był stosowany już w 1999 r. przez Salesforce. Do rozwiązań SaaS należą m.in. popularne aplikacje dostępne przez przeglądarkę internetową: Gmail, Dropbox, Microsoft Office 365. Przykładowe podgrupy to:

- DaaS (*Data* – dane) – usługa dostępu do danych zebranych np. przez Visa czy MasterCard, aby pomóc personalizować przekaz reklamowy,
- SECaaS (*Security* – bezpieczeństwo) – narzędzia wspierające bezpieczeństwo, np. programy antywirusowe.

### 1.4.3. Formy wdrożenia

Istnieją 4 formy wdrożenia, zależne od zakładanej ilości użytkowników chmury:

**Chmura prywatna** Jest to chmura stworzona na potrzeby danej organizacji, bez możliwości dostępu z zewnątrz. Fizyczne serwery tworzące chmurę mogą znajdować się wewnątrz sieci lokalnej organizacji (typ *on-premises*) bądź poza siecią lokalną, ale wciąż pod pełną kontrolą tej organizacji (typ *off-premises*), np. w wynajmowanej serwerowni. Ponieważ każdy kto ma dostęp do chmury jest ze sobą powiązany, nie występuje właściwość *multi-tenancy* prowadząca do optymalizacji kosztów.

**Chmura społecznościowa** Chmura dostępna dla wielu organizacji, które łączy wspólna misja czy cel, a zamknięta dla wszystkich spoza społeczności. Może być zarządzana przez jedną bądź wiele organizacji należących do społeczności. Podobnie jak chmura prywatna, może być *on-premises* bądź *off-premises*.

**Chmura publiczna** Chmura dostępna dla wszystkich, dostępna tylko przez sieć publiczną. Nie ma możliwości stworzenia chmury publicznej typu

*on-premises*. Posiada właściwość *multi-tenancy*. Zarządzanie odbywa się w pełni po stronie usługodawcy.

**Chmura hybrydowa** Połączenie chmury prywatnej bądź społecznościowej oraz chmury publicznej. Taki układ pozwala np. na obsługę standardowego ruchu za pomocą chmury prywatnej, a w przypadku wzmożonego obciążenia (godziny wieczorne, wybrany dzień w roku) przekierowanie części ruchu na chmurę publiczną.

## 1.5. Dokąd zmierza rozwój Cloud Computingu

Na podstawie historii najnowszej, a także mając na uwadze cele jakie stawia sobie technologia, autorka pracy dostrzega następujące kierunki rozwoju:

- **Dążenie do eliminacji problemu *vendor lock-in*** (uzależnienia technologicznego od wybranego dostawcy usług chmurowych)

Natychmiastowa popularność konteneryzacji, Kubernetes jako usługa oraz pomysły takie jak Kubeless i Serverless Framework pozwalają wnioskować, że dalsza ewolucja dąży do wypracowania uniwersalnych standardów wytwarzania aplikacji dla chmury.

- **Unifikacja systemów baz danych**

W ostatnich latach stworzono wiele innowacyjnych systemów baz danych. Ilość wariantów do wybrania stała się wręcz przytłaczająca. Dla jednego projektu może być potrzebne nawet kilka typów baz, dodatkowo nawet w ramach jednego typu są różne możliwości. Microsoft dostrzegł ten problem i stworzył usługę Azure CosmosDB, obsługującą wiele typów baz. Zdaniem autorki to nie jest ostatnie słowo wypowiedziane w tym kierunku. System pozbawiony problemu *vendor lock-in*, sam dobierający format przechowywania danych, byłby wielkim ułatwieniem dla programistów.

- **Usprawnienie modelu *serverless***

Usługa AWS Lambda powstała w 2014 r., a Google Cloud Functions wyszła z fazy pogładowej dopiero miesiąc temu. Według autorki, w przypadku usług FaaS konkurencja istnieje na tyle krótko, że dostarczyciele usług mogą się jeszcze wiele od siebie nawzajem nauczyć i zastosować nowe pomysły, które nieco zmieniają implementacje tego typu usług.

- **Wdrażanie Edge Computingu i Fog Computingu**

Pomimo że chmura stanowi podstawę dla Internetu Rzeczy, dobrze rozwinięta „warstwa pośrednia” jest niezbędna by tworzyć inteligentne miasta. Autorka spodziewa się większego zainteresowania koncepcją Edge, ponieważ nie wymaga aktualizacji routerów u użytkowników, więc cały proces mógłby przebiegać bez angażowania użytkowników. Router typu Fog mógłby znajdować się w gotowym nowoczesnym produkcie, np. w samoprowadzącym się samochodzie.

- **Dążenie do uproszczenia wdrażania kontenerów w chmurze**

Usługi typu CaaS były pierwszym krokiem do uproszczenia zarządzania kontenerami. Oferują przyjazne panele administracyjne, niewymagające korzystania z CLI. Pod koniec 2017 r. pojawił się nowy trend polegający na możliwości wdrażania kontenerów w chmurze bez konieczności zarządzania orkiestratorem oraz maszynami wirtualnymi. Amazon udostępnił AWS Fargate w listopadzie 2017 r.,[67] a Microsoft udostępnił Web App for Containers we wrześniu 2017 r.[71] oraz Azure Container Instances w kwietniu 2018 r.[66] Google nie posiada jeszcze swojego rozwiązania tego typu. Można zatem spodziewać się odpowiedzi ze strony Google oraz dopracowania tego typu rozwiązań w taki sposób, aby przeciętnemu programiście wiedza o konfiguracji Kubernetes nie była potrzebna w codziennej pracy.





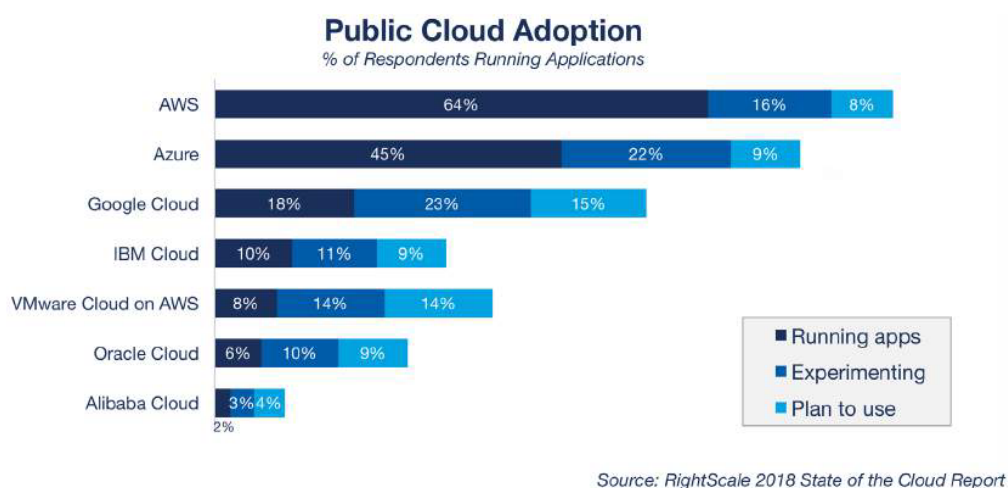
## Rozdział 2.

# Porównanie różnych podejść

Usługę wspierającą urządzenie mobilne w chmurze można stworzyć i udostępnić na wiele sposobów. Dla programisty, który nie korzysta z chmury na co dzień, wybór ten może okazać się zaskakująco szeroki.

W niniejszym rozdziale zostaną przedstawione i porównane konkretne sposoby tworzenia usługi w chmurze, w kolejności od najstarszych do najnowszych rozwiązań.

Ze względu na największą popularność (rysunek 2.1) oraz ilość dostępnych materiałów, w niniejszej pracy omawiając konkretne usługi chmurowe, koncentrowano się na chmurach AWS, Azure oraz Google Cloud.



Rysunek 2.1: Popularność dostawców usług chmurowych [29]

### 2.1. Budowanie własnej chmury prywatnej

Jest to rozwiązanie polegające na budowaniu wszystkiego samemu u podstaw. Potrzebny jest przynajmniej jeden serwer, połączenie z Internetem

oraz zespół osób dbających o infrastrukturę. Trzeba pamiętać nie tylko o standardowym przygotowaniu (odpowiednia temperatura pomieszczenia itp.), ale także o bieżących naprawach, konserwacji sprzętu, byciu na bieżąco z najnowszymi wymogami bezpieczeństwa.

Pod względem idei posiadania własnych serwerów, chmura różni się tym od tradycyjnego podejścia, że docelowo zależy nam na korzyściach wynikających z dobrego skalowania systemów. Im więcej serwerów jest potrzebne, tym większym wyzwaniem staje się utrzymywanie całej infrastruktury i bycie odpowiedzialnym za jej bezpieczeństwo.

### Podsumowanie

#### Główne narzędzia

- hardware, łącze internetowe, prąd
- system operacyjny dla serwera, np. CentOS, Debian, Windows Server
- nadzorca typu 1 (ang. *bare-metal hypervisor*), np. Xen, Oracle VM, Microsoft Hyper-V, VMware ESXi
- dla usług IaaS: OpenStack, OpenNebula
- dla usług CaaS: Kubernetes, Docker Swarm, Mesosphere DC/OS
- dla usług PaaS: Cloud Foundry, OpenShift

#### Perspektywy

- Pełna niezależność.
- Jedyna opcja dla organizacji, które muszą działać w obrębie sieci lokalnej, bez dostępu do Internetu (np. podczas ćwiczeń wojskowych).
- Brak zmartwienia związanego z brakiem wiedzy o tym, gdzie fizycznie znajdują się dane.

#### Ograniczenia

- Konieczność samodzielnego troszczenia się o infrastrukturę: serwis komputerów, temperatura w serwerowni, czekanie na kuriera z nowymi częściami, itp.

- Konieczność samodzielnego aktualizowania oprogramowania i dbałości o bezpieczeństwo danych.
- Brak możliwości skorzystania z ekonomii skali, obserwowanej w chmurach publicznych.
- Największy koszt wdrożenia chmury, szczególnie koszt początkowy zakupu infrastruktury i opłacenia specjalistów, którzy się nią zajmą.

## **2.2. Łączenie chmury prywatnej z chmurą publiczną**

Posiadając chmurę prywatną, w przypadku skalowania w górę nie trzeba koniecznie kupować nowych serwerów. Można skonfigurować chmurę hybrydową, łącząc chmurę prywatną z publiczną. Chmury publiczne zazwyczaj udostępniają interfejs do łączenia ich przez VPN z chmurą prywatną.

Istnieje także możliwość pokuszenia się o zakup gotowego zestawu dostosowanego do tworzenia chmury hybrydowej, ze wsparciem firm specjalizujących się w tych rozwiązaniach.

Zgodnie z raportem RightScale 2018[29], z rozwiązań hybrydowych w 2017 r. korzystało 58% ankietowanych, a rok później liczba ta spadła do 51%. Jednocześnie odnotowano nadanie większego priorytetu chmurom publicznym (zwiększenie z 29% do 38%), a mniejszego hybrydowym (spadek z 50% do 45%).

### **Podsumowanie**

#### **Główne usługi chmurowe**

Usługi przydatne przy tworzeniu chmury hybrydowej wymieniono w tabeli 2.1.

#### **Gotowe zestawy do tworzenia chmur hybrydowych**

Gotowe zestawy integrujące dedykowany zestaw fizycznych urządzeń z wybraną chmurą publiczną wymieniono w tabeli 2.2.

## 2.2. ŁĄCZENIE CHMURY PRYWATNEJ Z CHMURĄ PUBLICZNĄ

Typ usługi	Dostawca		
	AWS	GCP	Azure
wirtualna prywatna sieć	Virtual Private Cloud	Virtual Private Cloud	Virtual Network
dedykowane prywatne połączenie	Direct Connect	Cloud Interconnect	Azure Express Route
zarządzanie kontami użytkowników	Identity and Access Management	Cloud Identity	Azure Active Directory
wsparcie bezpieczeństwa chmury prywatnej	Inspector	b/d	Azure Security Center
integracja danych	Storage Gateway	b/d	StorSimple

Tablica 2.1: Główne interfejsy chmur publicznych dla chmur hybrydowych

Nazwa	Chmura publiczna	Zestawy
Azure Stack [68]	Microsoft Azure	Avanade Solution, Cisco Integrated System, Dell EMC Cloud, HPE ProLiant (HP), Huawei Hybrid Cloud Solution, Lenovo ThinkAgile SX, TERRA (Wortmann AG, Intel, and Microsoft)
Cisco and Google Cloud [69]	Google Cloud	Cisco Hybrid Cloud Platform for Google Cloud

Tablica 2.2: Gotowe rozwiązania dla chmur hybrydowych

### Perspektywy

- Największa elastyczność – korzystanie z zalet chmur prywatnych i publicznych.

- Możliwość wykorzystywania chmury publicznej tylko w przypadku gdy prywatna infrastruktura przestaje wystarczać.
- Możliwość tworzenia kopii zapasowej do chmury.
- Możliwość wykorzystywania wybranych specjalistycznych zastosowań chmury publicznej.

### Ograniczenia

- Konieczność utrzymywania rozwiązań zarówno chmury prywatnej jak i chmury publicznej.
- Spowolnienie rozwoju przez zachowywanie kompatybilności pomiędzy chmurami.

## 2.3. Wykupienie serwera wirtualnego w chmurze

Jest to podejście, w którym rezygnuje się z zarządzania fizyczną infrastrukturą komputerową. Odchodzą zatem ograniczenia spotykane przy budowaniu własnej chmury prywatnej (opisane w sekcji 2.1). Nadal jednak trzeba samodzielnie wybrać system operacyjny oraz doinstalować odpowiednie środowiska.

Serwer wirtualny w chmurze można łatwo pomylić ze zwykłym VPS (ang. *Virtual Private Server*) oferowanym przez firmy hostingowe. Hosting VPS istniał również w czasach „przed” Amazon EC2. Typowo polegał na podzieleniu zasobów jednej fizycznej maszyny na kilka mniejszych – wirtualnych, za pomocą oprogramowania typu Xen. Podobnie jak fizyczny serwer, VPS miał odgórnie ustaloną ilość RAM, wielkość dysku twardego i limity CPU. Można postrzegać VPS jako tańszą alternatywę dla fizycznego serwera.

Natomiast podejście „chmurowe” do serwerów wirtualnych różni się od tradycyjnego podejścia VPS w następujących względach:

- Dochodzi warstwa abstrakcji, dzięki której maszyna wirtualna nie jest ściśle przypisana do fizycznej maszyny. Można zmieniać ilość przydzielonych zasobów oraz zmieniać ilość wirtualnych maszyn hostujących aplikację.

- Rozliczanie odbywa się w modelu płatności za zużycie (*utility computing*). Płatność za tradycyjne serwery wirtualne odbywa się w formie abonamentu, np. w przypadku miesięcznego okresu rozliczeniowego, chcąc używać VPS przez kilka dni, nadal należałoby zapłacić za cały miesiąc.
- Zgodnie z modelem chmury NIST, panuje samoobsługa realizowana przy pomocy panelu administracyjnego. Nie ma konieczności składania zamówienia czy czekania na personel aż uruchomi skonfigurowaną maszynę.
- Wykonanie kopii zapasowej maszyny wirtualnej jest znacznie ułatwione przez chmurową warstwę abstrakcji.

W związku z dostępnością narzędzi przygotowanych do tworzenia chmur prywatnych, dostawcy serwerów VPS zaczęli stosować niektóre bądź wszystkie techniki wymienione powyżej. Wówczas główną różnicę stanowi zaplecze zasobów. W przypadku dużych chmur publicznych jest ono tak duże, że dla skalowania ilości maszyn wirtualnych w górę tworzy się iluzja nieskończoności zasobów, gdyż można zażyczyć sobie nawet dziesiątki tysięcy instancji serwerów wirtualnych. Dla mniejszych dostawców to wciąż może być zbyt wielkie wyzwanie.

Jednocześnie chmury publiczne nie stronią od świadczenia usług bardziej przypominających VPS. Spośród opcji rozliczania można wybrać takie uwzględniające rezerwację określonej puli zasobów na określony czas. Również istnieją możliwości wynajęcia serwera fizycznie odizolowanego od innych, np. w usłudze Amazon EC2 Dedicated Hosts.

Wynajmując prawdziwy serwer wirtualny w chmurze należy pamiętać, że to nie jest hosting, nawet jeśli go przypomina. Cechami chmury są skalowalność i płatność za zużycie. Podczas konfiguracji usługi IaaS istnieje możliwość indywidualnego doboru wielu parametrów, które należy dopasować w sposób optymalizujący koszty. W kosztach należy również uwzględnić dodatkowe usługi poza maszynami wirtualnymi, np. wynajęcie przestrzeni dyskowej na kopie zapasowe maszyn wirtualnych, usługi typu Load Balancer, itp.

Do tworzenia maszyn wirtualnych można używać napisanych przez siebie programów wykorzystujących API usługodawcy. Dla popularnych technologii (Java, Python itp.) są dostępne SDK do obsługi wybranej chmury oraz biblioteki obsługujące usługi IaaS wielu dostawców.

## Podsumowanie

### Serwery wirtualne w chmurach publicznych

- **Amazon Elastic Compute Cloud (EC2)** – Aby uruchomić nową instancję, należy wybrać obraz AMI (ang. *Amazon Machine Image*) zawierający system operacyjny wraz z oprogramowaniem, a także wybrać typ instancji zawierający wstępną konfigurację. Wraz z maszyną zostaje utworzony wirtualny dysk w usłudze Amazon Elastic Block Store. Można zmodyfikować domyślną konfigurację o dodanie m.in. load balancerów (Elastic Load Balancing), automatycznie skalującej się grupy (Amazon EC2 Auto Scaling), dodatkowych dysków w EBS. Wszystkie te usługi zostały zebrane wewnątrz panelu administracyjnego usługi EC2.
- **Azure Virtual Machines** – Wbrew mitom pozwala na utworzenie maszyn wirtualnych działających na systemach innych niż Windows, m.in. Ubuntu, Red Hat, SUSE, CoreOS, FreeBSD. Obrazy tych systemów znajdują się w Azure Marketplace. Do skalowania służą Virtual Machine Scale Sets. Ciekawą opcję stanowią Azure Extensions, czyli skrypty automatycznie uruchamiane po uruchomieniu maszyny wirtualnej. W przypadku maszyn działających na systemie Linux dostępna jest również opcja Cloud-init, gdzie można zdefiniować listę paczek do doinstalowania po pierwszym uruchomieniu systemu.
- **Google Cloud Platform** – Usługę Google wyróżnia prosty, schludny interfejs użytkownika. Do uruchomienia instancji maszyny wirtualnej z domyślnymi ustawieniami wystarcza jeden ekran z opcjami. W przypadku braku maszyn wirtualnych portal proponuje wykonać jeden projekt szacowany na 15 minut przy użyciu samouczka.

### Biblioteki przystosowane do obsługi wielu chmur IaaS

- **Apache LibCloud** – biblioteka dla języka Python. Obsługuje ponad 50 różnych dostawców.
- **Apache jClouds** – biblioteka dla języka Java.
- **Libretto** – biblioteka dla języka Go.
- **pkgcloud** – biblioteka dla node.js.
- **Fog** – biblioteka dla Ruby.

### Perspektywy

- Brak początkowej inwestycji w infrastrukturę.
- Brak zmartwień związanych z samodzielnym administrowaniem serwerami.
- Skalowalność infrastruktury.
- Możliwość instalowania dowolnych środowisk.
- Możliwość przeniesienia tradycyjnie napisanej aplikacji do chmury.

### Ograniczenia

- Większy koszt w porównaniu z usługami PaaS (np. koszty licencji oprogramowania).
- Konieczność samodzielnego administrowania oprogramowaniem.
- Konieczność samodzielnego dbania o bezpieczeństwo i aktualizacje.
- Skomplikowany model rozliczania – mnogość parametrów od których zależy całkowity koszt usług. Niewystarczająca uwaga natychmiast odbija się na karcie podanej do rozliczeń. Autorka pracy ma w tym punkcie zastrzeżenie do firmy Amazon, ponieważ z jednej strony dużymi literami ogłasza możliwość darmowego przetestowania usług, a z drugiej strony samo założenie konta kosztuje 1 USD, a założenie maszyny wirtualnej (która sama w sobie była bezpłatna) z domyślnymi opcjami kosztowało kolejnego dolara.
- Organizacje celujące w ograniczoną ilość odbiorców mogą potrzebować serwera wirtualnego, ale nie potrzebować skalowalności jaką daje chmura. Dla nich odległość do centrum danych mieszczącego się w innym kraju oraz korzystanie z oferty zagranicznych usługodawców mogą stanowić wady, a hosting typu VPS dostarczany z pobliskiej miejscowości lepszą alternatywę.

## 2.4. Usługi typu PaaS oparte o maszyny wirtualne w chmurach publicznych

Jest to pierwsza generacja usług typu PaaS, gdzie w odróżnieniu od tworzenia własnej maszyny wirtualnej i na niej uruchamianiu aplikacji, twórca



aplikacji ma skupić się jedynie na dostarczeniu aplikacji, a utworzenie odpowiednich maszyn wirtualnych znajduje się po stronie usługodawcy. Obecnie funkcjonują również inne sposoby usprawniające pracę programisty w kontekście usług chmurowych i te sposoby zostały omówione w kolejnych sekcjach.

Główną charakterystyką tych usług jest to, że są one specyficzne dla danej chmury. Każda posiada własne mechanizmy łączenia ze sobą różnych typów usług stworzonych dla developerów. Mogą różnić się m.in. wspieranymi technologiami. Przykładowo, firma Microsoft wspiera szerokie spektrum technologii, ale zawsze w pierwszej kolejności .NET Framework. Chmura Google zaczynała od wsparcia dla Javy i Pythona, a obecnie można tworzyć również w Node.js, PHP, Go, Ruby, .NET Core.

Wadą tego typu rozwiązań jest duże przywiązanie do wybranego dostawcy usług. Każdy udostępnia swój własny SDK dla wspieranych języków programowania i przeniesienie aplikacji napisanej w ten sposób do innego usługodawcy wymaga przepisywania znacznej części kodu lub może być niemożliwe.

Platformy aplikacji w chmurach publicznych są dobrze przygotowane do tworzenia aplikacji typu *cloud native*, zgodnych z metodyką *12-factor app*. Usługodawcy dbają również o łatwe stosowanie strategii DevOps na swoich platformach. Powyższe implikuje, iż tworząc tego typu aplikacje warto wcześniej dobrze zapoznać się z metodyką 12-factor, jak również z wzorcami projektowymi towarzyszącymi tworzeniu mikrousług.

Wsparcie dla tworzenia aplikacji w chmurze może polegać na:

- budowaniu back-endu dla aplikacji mobilnych – jest to zastosowanie, na którym koncentruje się niniejsza praca. Polega na udostępnianiu węzłów końcowych API (ang. *endpoints*), z którymi komunikuje się aplikacja mobilna. Odbywa się to przy pomocy REST lub SOAP;
- serwisie notyfikacji – umożliwia wysyłanie notyfikacji push z poziomu chmury;
- aplikacjach logicznych – umożliwiają definiowanie przepływów, np. segregacja wiadomości z Twittera a następnie automatyczne wysłanie e-maili;
- stronach internetowych – gotowe szablony stron internetowych opartych np. o Wordpress czy Joomla!;
- korelowaniu i analizie różnych źródeł danych.

### Podsumowanie

#### Usługi służące tworzeniu aplikacji

- **AWS Elastic Beanstalk** – aplikacje internetowe oraz mikrousługi. Obsługiwane języki: Java, PHP, Python, Node.js, Ruby, .NET, Go
- **Azure App Service**
  - **Web Apps** – aplikacje internetowe oraz mikrousługi. Obsługiwane języki: Java, PHP, Python, Node.js, Ruby, .NET
  - **Mobile Apps** – tworzenie backendu dla aplikacji mobilnych, możliwość synchronizacji offline, połączenie z Notification Hubs do wysyłania notyfikacji push
  - **WebJobs** – zadania wykonywane po odebraniu zdarzenia
- **Google App Engine** – platforma tworzenia mikrousług. Obsługiwane języki: Java, PHP, Python, Node.js, Ruby, .NET, Go
- **Google Cloud Endpoints** – narzędzia do generowania API i bibliotek na podstawie aplikacji w App Engine

#### Główne usługi wspierające tworzenie back-endu dla aplikacji mobilnych

##### AWS:

- **Simple Storage Services (S3)** – przechowywanie danych binarnych
- **Amazon DynamoDB** – baza obsługująca dwa modele danych: dokumentowe oraz klucz-wartość
- **Amazon Relational Database Service (RDS)** – relacyjna baza danych kompatybilna z różnymi silnikami baz danych: PostgreSQL, MySQL, MariaDB, Oracle Database, Microsoft SQL Server
- **Amazon Aurora** – baza kompatybilna z PostgreSQL oraz MySQL, zarządzana przez RDS

##### Azure:

- **Azure Storage**
  - **Blob Storage** – dane binarne

- **Table Storage** – baza NoSQL typu *wide-column*
- **Queue Storage** – kolejki komunikatów
- **Azure Cosmos DB** – baza NoSQL obsługująca wiele modeli danych: grafowe, dokumentowe, klucz-wartość oraz *wide-column*
- **Azure SQL Database** – baza relacyjna kompatybilna z MS SQL
- **Azure Notification Hubs** – wysyłanie notyfikacji push na urządzenia z różnymi systemami operacyjnymi
- **Azure Cognitive Services** – usługi ułatwiające uczenie maszynowe, np. rozpoznawanie obrazów

Google Cloud:

- **Google Cloud Datastore** – baza NoSQL typu dokumentowego
- **Google Cloud Firebase** – baza NoSQL typu dokumentowego, będąca następcą Datastore; aktualnie w wersji beta
- **Google Cloud SQL** – baza relacyjna kompatybilna z MySQL
- **Google Cloud Storage** – przechowywanie danych binarnych
- **Google Cloud Bigtable** – baza typu *wide-column*
- **Google Cloud Spanner** – baza relacyjna skalowalna horyzontalnie

Perspektywy

- Tańszy koszt w porównaniu z IaaS.
- Koncentracja na zadaniu tworzenia aplikacji.
- Dostęp do specjalistycznych usług, np. rozpoznawania obrazów.
- Wsparcie dla DevOps w zintegrowanym środowisku danej platformy.

Ograniczenia

- Problem *vendor lock-in* – trudność w przenoszeniu aplikacji na inną platformę.

- Konieczność zastosowania się do rozwiązań dyktowanych przez początkowo wybranego usługodawcę. Można napotkać na trudność we wprowadzaniu zmian do już istniejących aplikacji, ponieważ gdy usługodawca zmieni API, wprowadzając zmianę należy również dostosować aplikację do aktualnego API.
- Wiązanie się z konkretnym usługodawcą oznacza konieczność zatrudniania programistów wyspecjalizowanych w usługach tego dostawcy. Występuje tu potencjalna trudność w znalezieniu pracownika oraz płaceniu za jego wąską specjalizację.
- Mniejsza elastyczność niż w IaaS.
- Konieczność zapoznania się z dobrymi praktykami tworzenia aplikacji *cloud native*, w tym mikrousług.
- Utrudnione rozwijanie aplikacji offline, bez dostępu do platformy.

## 2.5. Samodzielna instalacja orkiestratora kontenerów

Jest to podejście, które do niedawna (do czasu publikacji usług typu CaaS) miało szczególne znaczenie, ponieważ pozwalało na korzystanie z Cloud Foundry czy Kubernetes pomimo braku specjalnego wsparcia po stronie chmur publicznych.

Rozwiązania te dzięki wykorzystaniu kontenerów eliminują problem *vendor lock-in*. Trzeba jednak samemu przejść przez proces instalacji odpowiedniego oprogramowania na serwerze wirtualnym lub fizycznej maszynie (ang. *bare-metal*).

Chmury publiczne wspierają konteneryzację jako usługę od stosunkowo niedawna. Wciąż 57% wdrożeń Kubernetes opiera się na Amazon EC2. Podejście oparte o samodzielną instalację niejednokrotnie będzie można spotkać w literaturze, na stronach internetowych oraz w już wdrożonych aplikacjach.

### Podsumowanie

Główne usługi chmurowe oraz zalety i wady dziedziczą się odpowiednio z wynajmu maszyny wirtualnej (sekcja 2.3) bądź z tworzenia chmury prywatnej (sekcja 2.1).

### Główne narzędzia

- Kubernetes
- Docker Swarm
- Mesosphere DC/OS
- Cloud Foundry

### Perspektywy

- Brak problemu *vendor lock-in*.
- Możliwość korzystania z obrazów Dockera, które uchodzą za standard obsługiwany na każdej platformie.
- Możliwość skalowania aplikacji w standardowy sposób.
- Możliwość korzystania z dowolnych technologii w odpowiednich wersjach.
- Możliwość zatrudniania programistów o różnych umiejętnościach, ponieważ nie muszą znać technologii podyktowanych przez wybranego dostawcę usług chmurowych. Co więcej, każda mikrousluga może być napisana w innej technologii, więc łatwiej znaleźć odpowiednich specjalistów.
- Możliwość rozwijania oprogramowania lokalnie na komputerze, mając pewność, że na środowisku produkcyjnym będą obowiązywały dokładnie te same narzędzia.

### Ograniczenia

- Konieczność samodzielnej instalacji potrzebnego środowiska.

## 2.6. Usługa konteneryzacji w chmurach publicznych

Na fali ogromnej popularności Dockera, a później Kubernetes, zaczęły powstawać usługi typu CaaS/KaaS, pozwalające na korzystanie z dobrodziejstw

## 2.6. USŁUGA KONTENERYZACJI W CHMURACH PUBLICZNYCH

orkiestratorów kontenerów bez konieczności ich instalowania. Usługi te zazwyczaj pozwalają zarządzać orkiestratorem z poziomu narzędzia CLI bądź przy pomocy graficznego interfejsu (rysunek 2.2).

Do zarządzania obrazami kontenerów usługodawcy proponują specjalne rejestry kontenerów.

The screenshot displays the Google Cloud Platform console for creating a new Kubernetes cluster. The top navigation bar shows 'Google Cloud Platform' and 'My First Project'. The left sidebar lists various services, with 'Klasy' (Clusters) selected. The main content area is titled 'Utwórz klaster Kubernetes'. A warning message at the top indicates that some fields cannot be changed after cluster creation. The form includes the following fields and options:

- Nazwa**: cluster-1
- Typ lokalizacji**: Strefa (selected), Region
- Strefa**: us-central1-a
- Wersja główna**: 1.9.7-gke.6 (domyślna)
- Pule węzłów**: default-pool, Liczba węzłów

Rysunek 2.2: Graficzny interfejs pozwalający skonfigurować Kubernetes w Google Kubernetes Engine

## Podsumowanie

### Usługi CaaS

- **Google Kubernetes Engine**
- **Azure Kubernetes Service (AKS)**
- **Amazon ECS – Amazon Elastic Container Service** – własna implementacja orkiestracji kontenerów Dockera stworzona przez Amazona

- **Amazon EKS** – Amazon Elastic Container Service for Kubernetes – wersja implementująca Kubernetes

### Usługi rejestru kontenerów

- Amazon Elastic Container Registry
- Azure Container Registry
- Google Container Registry

### Perspektywy

- Prosta możliwość wypróbowania i skorzystania z Kubernetes bez przechodzenia przez proces konfiguracji całej infrastruktury.
- Główne zalety dziedziczą się z zalet opisanych w sekcji poświęconej samodzielnej instalacji orkiestratora kontenerów (sekcja 2.5).

### Ograniczenia

- Konieczność zapoznania się z odpowiednim konfigurowaniem orkiestratora kontenerów. Jest to konfiguracja dotycząca infrastruktury, czyli dla twórcy aplikacji nie jest związana z programowaniem.
- Pod względem tworzenia aplikacji wypada znać i umieć stosować w praktyce zasady dotyczące tworzenia aplikacji dla chmury, w szczególności mikrousług.

## 2.7. Usługi typu PaaS oparte o kontenery w chmurach publicznych

Aby na chmurze publicznej móc korzystać z kontenerów Dockera, nie trzeba koniecznie znać się na Kubernetes. Chmury publiczne wypracowały rozwiązania polegające na wdrożeniu w chmurze zadanego obrazu, bez konieczności konfigurowania infrastruktury na jakiej są uruchamiane.

Rozwiązania tego typu służą głównie w celu uruchomienia w chmurze pojedynczego kontenera. Jako inny przypadek użycia Azure podaje chaotyczne zapotrzebowanie (ang. *traffic in spikes*, czyli ruch, który na wykresie wyglądałby jak kolce). Natomiast dla długo działających kontenerów ze stabilnym ruchem, Azure rekomenduje wykorzystanie CaaS. [70].

Podobnie jak w przypadku usług CaaS (sekcja 2.6), do zarządzania obrazami kontenerów można użyć usługi rejestru kontenerów.

Opisywane podejście w czasie tworzenia pracy uchodzi za nowatorskie – pierwsze rozwiązania tego typu pojawiły się pod koniec 2017 r., a firma Google póki do nie przedstawiła swojej propozycji.

### Podsumowanie

#### Usługi instancji kontenerów

- **AWS Fargate** – usługi CaaS od Amazona, ECS i EKS, posiadają dwa tryby wdrożenia kontenera: tryb Fargate oraz tryb EC2. Fargate to możliwość wdrożenia kontenera bez konieczności zarządzania orkiestratorem i maszynami wirtualnymi. [72]
- **AWS Elastic Beanstalk – Deploying Elastic Beanstalk Applications from Docker Containers** – opcja wdrażania aplikacji w Elastic Beanstalk za pomocą obrazu Dockera
- **Azure Container Instances** – możliwość wdrażania instancji kontenerów rozliczana na podstawie sekund [70]
- **Azure App Service – Web App for Containers** – rozwinięcie App Service o możliwość wdrożenia obrazu kontenera.

#### Usługi rejestru kontenerów

Tak samo jak w sekcji dotyczącej usługi CaaS (sekcja 2.6).

#### Perspektywy

- Niska bariera wejścia jeśli chodzi o wdrażanie obrazu kontenera do chmury publicznej.
- Możliwość odroczenia nauki orkiestratora kontenerów do czasu wzrostu popularności aplikacji.
- Główne zalety dziedziczą się z zalet opisanych w sekcji poświęconej samodzielnej instalacji orkiestratora kontenerów (sekcja 2.5).

#### Ograniczenia

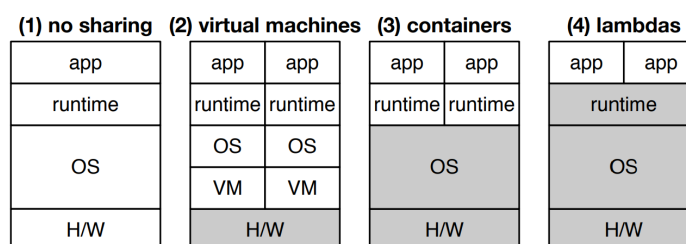
- Podejście nie rozwiązuje całkowicie problemu nauki orkiestratora kontenerów.



- Pod względem tworzenia aplikacji wypada znać i umieć stosować w praktyce zasady dotyczące tworzenia aplikacji dla chmury, w szczególności mikrousług.
- Usługi tego typu są bardzo nowe, co może powodować zwiększone ryzyko napotkania na nietypowe problemy lub dostosowywania się do zmian względem pierwotnej wersji.

## 2.8. Serverless – usługi typu FaaS

Odkąd w 2014 r. Amazon udostępnił usługę AWS Lambda, model serverless zaczął być postrzegany jako kolejna rewolucja po kontenerach. W tym podejściu jeszcze większa część pracy pozostaje po stronie usługodawcy, a programiście zostaje głównie część związana z tworzeniem aplikacji (rysunek 2.3).



Rysunek 2.3: Porównanie ilości warstw współdzielonych przez różne typy komponentów osadzonych w chmurze. [31]

Technologie serverless wciąż znajdują się w fazie rozwoju, jednak wydają się bardzo obiecująco. Zgodnie z badaniem przytoczonym w artykule[30], podejście serverless może powodować oszczędności względem architektur monolitycznych oraz mikrousługowych na poziomie do 70.08%. Powoduje również znaczną oszczędność czasu pracy nad projektem.[7]

## Podsumowanie

### Usługi FaaS

- **AWS Lambda** – pierwsza platforma tego typu. Obsługiwane języki: Java, Python, Node.js, C#/.NET Core, Go
- **Azure Functions** – platforma serverless bazująca na WebJobs. Obsługiwane języki (bez uwzględniania wsparcia eksperymentalnego i pogładowego):

- w wersji 1.x: C#, F#/.NET Framework, Node.js 6
- w wersji 2.x (oficjalnie dostępnej od 24.09.2018 r. [73]): C#, F#/.NET Core 2, Node.js 8 & 10
- **Google Cloud Functions** – platforma serverless oficjalnie dostępna od sierpnia 2018 r. [43] Obsługiwane języki: Java, Python

### Główne usługi wspierające tworzenie back-endu dla aplikacji mobilnych

Tak jak w sekcji dotyczącej usług PaaS w oparciu o maszyny wirtualne (sekcja 2.4).

### Perspektywy

- Mniejszy koszt utrzymania aplikacji przez brak konieczności zajmowania zasobów w oczekiwaniu na przychodzące zapytania.
- Brak konieczności umiejętnego projektowania mikrousług.
- Skalowanie pozbawione nadmiarowości.
- Szybki czas wdrożenia aplikacji do chmury (2x szybciej niż mikrousługi) [7]

### Ograniczenia

- Wady dziedziczą się z podejścia PaaS do tworzenia aplikacji w chmurze publicznej (sekcja 2.4, za wyjątkiem konieczności nauki projektowania mikrousług).
- Utrudnienia w uruchamianiu dłużej trwających zadań.
- Ograniczona możliwość testowania funkcji offline.

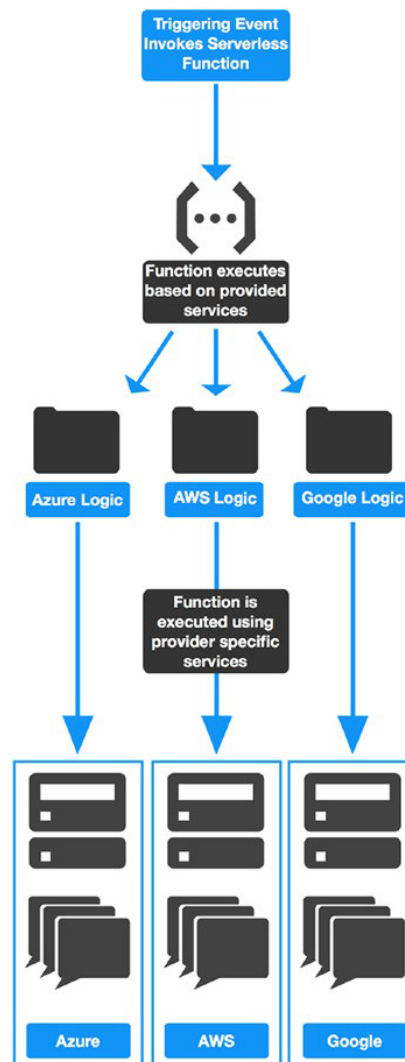
## 2.9. Serverless bez przywiązania do platformy

Podczas gdy usługi FaaS w chmurach publicznych jawią się jako najtańsze oraz posiadające najmniejszą barierę wejścia do chmury, wciąż dziedziczą po innych rozwiązaniach typu PaaS starszy problem przywiązania do usługodawcy. Jest to dość istotny problem, biorąc pod uwagę to jaką popularność zdobył Docker oraz Kubernetes stanowiąc wybawienie od niego.

W literaturze przedmiotu znaleziono dwa projekty, które mają rozwiązać ten problem.

### 2.9.1. Serverless Framework

Jest to narzędzie do rozwijania i wdrażania funkcji w sposób oddzielający logikę konkretnego dostawcy usługi FaaS od sposobu pisania funkcji. Dzięki temu jeden kod jest gotowy do działania na różnych platformach (w tym Kubeless, o którym mowa w następnej sekcji). [7]



Rysunek 2.4: Oddzielenie logiki aplikacji od logiki dostawcy usługi FaaS przy pomocy Serverless Framework [7]

Rozwiązanie problemu *vendor lock-in* skutkuje następującymi ograniczeniami:

- Narzędzia będące nakładkami na inne narzędzia zawsze dążą do obsługi wspólnego mianownika tych narzędzi. Wyklucza to (lub

znacząco utrudnia) użycie pewnych specyficznych usług typu PaaS danego dostawcy.

- Konieczność uczenia się dodatkowego narzędzia.

### 2.9.2. Kubeless

Jest to framework działający na Kubernetes, pozwalający uruchomić mechanizm serverless podobny do tego spotykanego w usługach FaaS. [13]

Kubeless obsługuje technologie: Python, Node.js, Ruby, PHP, Go, .NET, Ballerina. Funkcje tworzone z obrazów Dockera są w trakcie rozwoju.

Podstawowym ograniczeniem związanym z używaniem Kubeless jest konieczność zdobycia najpierw Kubernetes. Bez względu na to, czy będzie on zainstalowany na fizycznym serwerze czy maszynie wirtualnej (ręcznie albo przez KaaS), wciąż trzeba przede wszystkim zapewnić mu zasoby na których można go uruchomić. Eliminuje to zatem prawdopodobnie najważniejszą korzyść jaką wnosi serverless na chmurze publicznej, czyli brak stałych opłat za utrzymanie funkcji. Co prawda w Kubeless funkcja jest uruchomiona tylko przez chwilę, ale Kubernetes już działa cały czas.

## Rozdział 3.

# Analiza wyboru platformy dla projektu

### 3.1. Opis projektu aplikacji mobilnej

Pomysł na aplikację mobilną tworzoną w ramach pracy, to realny przykład aplikacji, dla której chmura może stanowić niezbędne wsparcie. Powody ku temu są następujące:

- do działania wymaga serwera,
- komunikuje się używając REST-owego API,
- działa w oparciu o treść tworzoną przez użytkowników,
- w teorii ma szansę zyskać popularność, czyli może być wymagana zdolność do skalowania w górę.

#### 3.1.1. Geneza pomysłu

Pomysł polega na stworzeniu nowatorskiej aplikacji służącej do zapisywania wydatków. Tego typu aplikacje dotychczas tworzone były z myślą o jednym użytkowniku i raczej nie wymagały do działania serwera. Z doświadczenia autorki pracy, takie podejście jest skuteczne do czasu, gdy mieszka się samemu, a wszystkie wydatki tworzy się z myślą o sobie.

W przypadku gdy mieszka ze sobą więcej osób, liczby przestają cokolwiek mówić, gdyż może być tak, że jedna osoba robi więcej zakupów czy inna płaci rachunki. Powstają problemy jak się nawzajem rozliczać. Szczególnie biorąc pod uwagę, że różne kategorie wydatków można rozliczać w różny sposób, np.:

- wydatki na higienę, gdzie z części rzeczy korzystają wszyscy, a część służy konkretnym osobom,
- wydatki na samochód, gdzie za naprawy mogliby płacić wszyscy użytkownicy po równo, a za paliwo zgodnie z użytkowaniem,
- wydatki na rachunki, gdzie można ustalić, że osoba zarabiająca więcej będzie płaciła proporcjonalnie więcej,
- wydatki na jedzenie, gdzie jedzenie dla jednej osoby może kosztować znacznie więcej niż dla drugiej.

Gdy gospodarstwo domowe liczy więcej osób, wzajemne rozliczenia stają się skomplikowane. Trudno wówczas prowadzić budżet domowy, kontrolować gdzie „wyciekają” pieniądze. Można starać się, aby wydatki były zapisywane na jednym komputerze, można pewne obliczenia przeprowadzić ręcznie. Nie zapewnia to jednak wygody oddzielenia własnych wydatków od wydatków innych, jak również nie umożliwia sprawdzania np. ile wynoszą wydatki na jedzenie w danym miesiącu.

Biorąc pod uwagę powyższe, idealnym rozwiązaniem problemu zapisywania wydatków przez kilka osób byłaby aplikacja, która umożliwiłaby połączenie kilku użytkowników w grupę i przy pomocy serwera przeprowadzałyby wszystkie potrzebne obliczenia. Gwarantowałyby to niespotykaną wcześniej precyzję w rozliczeniach, przy jednoczesnej prostocie obsługi.

Do obsługi wystarczyłaby jakakolwiek aplikacja kliencka zdolna do połączenia z Internetem. Serwer zadbałby o właściwą synchronizację pomiędzy instancjami aplikacji. Aby umożliwić precyzyjne rozliczenia, dodawanie wydatku powinno umożliwić wyszczególnienie, że należy do kategorii, w której panuje określony sposób rozliczeń.

Jest możliwe ułatwić tę czynność przez rozpoznawanie obrazu, gdzie podstawowe parametry (cena na paragonie, data zakupu, nazwa sklepu) byłyby wczytywane automatycznie. Z doświadczenia autorki w zapisywaniu wydatków wynika, że zazwyczaj prawie wszystkie pozycje na paragonie odnoszą się do jednej kategorii. Aby wygodnie wyszczególnić pojedynczą rzecz na paragonie, wystarczy odjąć ją od ceny całkowitej i przypisać do innej kategorii. Możliwość wyszczególnienia wydatków z danego paragonu byłaby sporym ułatwieniem.

Również z doświadczenia autorki spory problem stanowi rozliczanie wydatków na jedzenie. Trudno wyznaczyć model, w którym owo rozliczenie przeprowadzone jest jednocześnie sprawiedliwie oraz wygodnie. Możliwym

rozwiązaniem, będącym elementem pomysłu na aplikację, byłoby automatyczne tworzenie przeliczników dla kategorii „jedzenie”, bazując na wprowadzonych danych takich jak: waga, wiek, płeć, wzrost i poziom aktywności fizycznej u danej osoby. Wykorzystując siłę serwera w chmurze, wszystkie potrzebne obliczenia byłyby przeprowadzone automatycznie.

Kolejną rzeczą będącą utrapieniem tradycyjnych aplikacji są wycieczki zagraniczne i wydatki w obcej walucie. Trudno w takich sytuacjach podtrzymywać kontrolę nad tym, ile ma się pieniędzy, jeśli część występuje w innej walucie. Dlatego pomysł uwzględnia także obsługę różnych walut.

Tak skonstruowana aplikacja praktycznie nie wymagałaby większego wysiłku niż tradycyjne programy do zapisywania wydatków. Za to posiadając dodatkowe informacje byłyby w stanie generować wiele przydatnych raportów, np.:

- ile wydaje się jako gospodarstwo domowe w danej kategorii,
- ile dana osoba wydaje na ubrania,
- ile wspólnie wydaje się na życie,
- ile dana osoba wydaje na jedzenie,
- w których sklepach najczęściej się wydaje pieniądze,
- ile dana osoba lub wszyscy razem posiadają pieniędzy w jakich walutach.

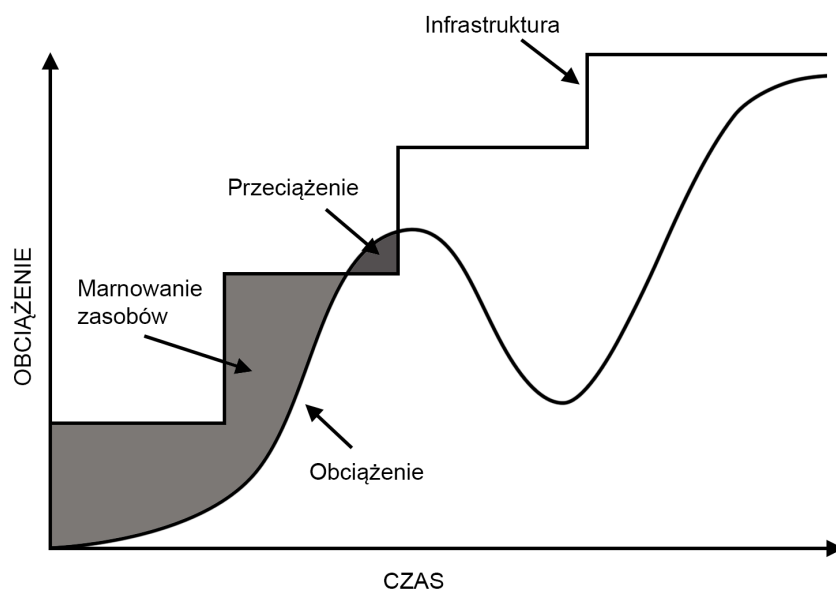
### **3.1.2. Typowy projekt dla chmury**

Opisany w sekcji 3.1.1 pomysł stanowi przykład typowego projektu informatycznego. Rozwiązuje problem życia codziennego przez umiejętne zebranie informacji, przetworzenie jej i zwrócenie wartościowych wyników.

Jednocześnie nie jest to pomysł czysto teoretyczny, czyli taki, z jakim często można mieć styczność przerabiając różne samouczki do programowania. Tego rodzaju projekty są często zbyt uproszczone, by pozwolić zetknąć się z rzeczywistymi problemami. Nie jest to szandarowy program do napisania, taki jak „lista zadań”, który został już przerobiony przez innych pod każdym możliwym kątem.

Realizacja tego pomysłu nie byłaby możliwa bez serwera. Serwer można teoretycznie utworzyć na tradycyjnej infrastrukturze niezgodnej z modelem chmury. Wówczas niestety aplikacji grozi tzw. „syndrom Naszej-Klasy”

(nazwany tak w książce „*Windows Azure. Wprowadzenie do programowania w chmurze*”), objawiający się tym, że autor aplikacji sam nie przewiduje jej nagłego wzrostu popularności. Sukces staje się wtedy zwiastunem klęski, ze względu na wdrożenie aplikacji w sposób niepozwalający na obsługę większego ruchu. Zirytowani użytkownicy odwracają swoje zainteresowanie od aplikacji, która ma problemy z dostępnością. [32] Zależność tę widać na rysunku 3.1.



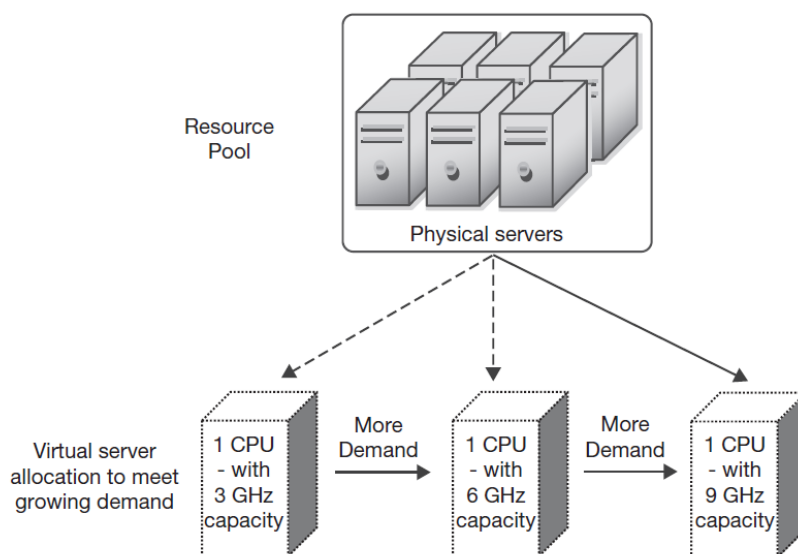
Rysunek 3.1: Wykres obrazujący problem tradycyjnego podejścia do skalowania aplikacji [32]

Biorąc pod uwagę powyższe, wykorzystanie chmury okaże się przydatne właśnie w momencie, kiedy aplikacja okaże się sukcesem. Jeśli aplikacja nie ma szansy na sukces, nie ma sensu jej w ogóle wdrażać. Jeśli ma szansę na sukces, to lepiej zapewnić jej skalowalne środowisko.

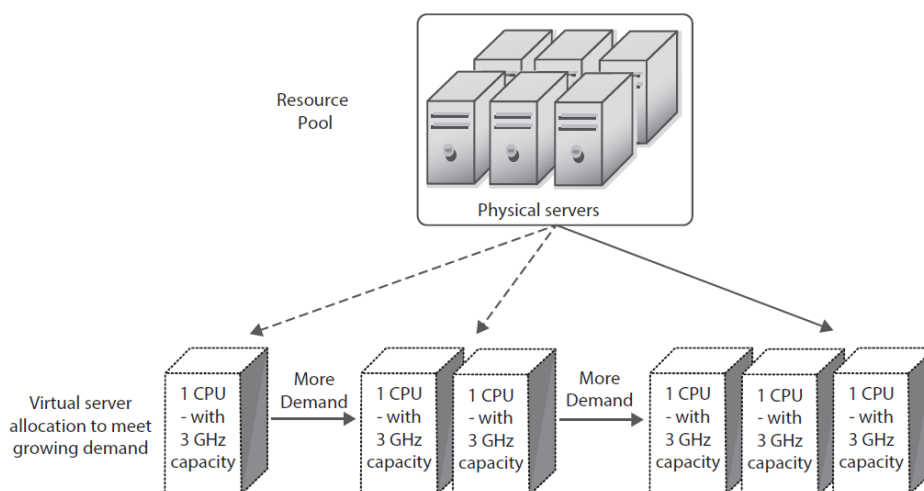
Ważnym elementem skalowalności aplikacji jest jej budowa. Nawet aplikacja wdrożona w chmurze, jeśli zostanie stworzona w sposób tradycyjny, jest narażona na problemy ze skalowalnością. Tradycyjnie większą ilość zasobów osiągało się przez skalowanie wertykalne, czyli wymianę serwera na taki o lepszych parametrach sprzętowych (rysunek 3.2). Niestety specjalistyczne zasoby komputerowe o wysokiej mocy są drogie, a i one mają swoje granice.

Z pomocą przychodzi skalowanie horyzontalne. Pozwala połączyć siły wielu komputerów, które pod względem zasobów mogą być przeciętne i tanie (rysunek 3.3). Nie istnieje wówczas problem ze skalowalnością ani z przerwą w dostarczaniu usług. Możliwość skalowania horyzontalnego wymaga użycia odpowiedniej architektury aplikacji.





Rysunek 3.2: Skalowanie wertykalne – wraz z rosnącym zapotrzebowaniem wymiana serwera na taki z lepszymi parametrami [1]



Rysunek 3.3: Skalowanie horyzontalne – wraz z rosnącym zapotrzebowaniem dokładanie serwerów o tej samej mocy [1]

Chmura idealnie nadaje się do skalowania horyzontalnego. Tworzona z setek tysięcy maszyn, daje możliwość obsłużenia nawet nadspodziewanie wysokiego ruchu. Z kolei usługi chmurowe wspomagają proces tworzenia aplikacji typu *cloud native*, tworzonych za pomocą mikrousług i korzystających z baz NoSQL.

Kolejnym ważnym aspektem, dla którego warto rozważyć użycie chmury, jest dostępność usług wykorzystujących uczenie maszynowe do np. rozpoznawania obrazu, a także usług związanych z przetwarzaniem dużej ilości

danych. W dłuższej perspektywie ułatwienia takie jak automatyczne czytanie danych ze zdjęcia paragonu lub możliwość wykonania analizy dużego zbioru danych mogą stanowić dużą wartość dodaną.

Wymienione powyżej przesłanki wskazują na to, że omawiany projekt posiada cechy typowego projektu informatycznego i z powodzeniem obrazuje sens wykorzystania chmury do tworzenia nawet eksperymentalnych projektów.

### 3.2. Wymagania dotyczące wyboru podejścia oraz chmury

Poniżej wymieniono podstawowe kryteria wyboru:

**Koszt (jak najniższy)** Choć doceniane są droższe narzędzia, za którymi mogłyby iść dodatkowe korzyści, preferowane są podejścia oraz usługodawcy, dzięki którym można optymalizować koszty. Istotna jest tu niska bariera uruchomienia aplikacji w chmurze, szczególnie biorąc pod uwagę, iż jest to projekt tworzony przez jedną osobę i jest wysoce prawdopodobne, że zanim powstanie stabilna wersja, może upłynąć dużo czasu.

**Poziom skomplikowania (jak najniższy)** Im wyższy poziom skomplikowania, tym więcej różnych dziedzin wiedzy należałoby połączyć i dysponować tym większym zasobem specjalistów. Jako projekt realizowany przez jedną osobę i przy niskich kosztach, powinien stosować taki zakres rozwiązań, by ta jedna osoba mogła się nimi swobodnie posługiwać.

**Czas dostępu (jak najkrótszy)** Preferowane są te chmury, które posiadają centrum danych w pobliżu lub na terenie Polski. Ma to znaczenie dla jakości usługi oraz ewentualnych dodatkowych kosztów ponoszonych na CDN.

**Technologie (najlepiej .NET)** Ponieważ autorka pracy posiada największe doświadczenie w technologii .NET, usługodawca wspierający tę technologię będzie miał przewagę nad innymi usługodawcami podczas wyboru. Używanie technologii, które już się zna, redukuje koszt i czas wdrożenia aplikacji.

**Przenoszenie na inną platformę (nieistotne)** Zdaniem autorki pracy, jest większe prawdopodobieństwo, że do tego projektu mogą się przydać usługi specyficzne dla danej chmury, niż prawdopodobieństwo by koszt uruchomienia projektu na innej chmurze był znacząco niższy i warty przenoszenia aplikacji.

### 3.3. Analiza

W niniejszej sekcji przedstawiono, jak istniejące podejścia zestawiają się z wymaganiami postawionymi projektowi. Na podstawie analiz dokonano wyboru podejścia oraz chmury publicznej, na której został zrealizowany projekt.

#### 3.3.1. Wybór podejścia

##### **Budowanie chmury prywatnej lub hybrydowej**

1. **Koszt** – serwer to wydatek rzędu kilku tysięcy złotych, do tego należy zadbać o odpowiednie pomieszczenie, odpowiednie łącze internetowe, o publiczny adres IP. Do tego należałoby opłacić wsparcie firmy zajmującej się administrowaniem serwerów, ponieważ nawet zakładając, że autorka sama będzie na bieżąco ze wszystkimi zagadnieniami dotyczącymi serwerów, to i tak musiałaby mieć zmiennika. Wszystkie te czynniki dyskwalifikują podejście budowania własnej chmury.
2. **Poziom skomplikowania** – jest bardzo wysoki. Należy być na bieżąco z administracją fizycznych serwerów, oprogramowaniem zapewniającym wirtualizację oraz oprogramowaniem do świadczenia usług typu IaaS bądź CaaS. Wszystkie te rzeczy nie mają zbyt wiele wspólnego z programowaniem.
3. **Czas dostępu** – dla użytkowników znajdujących się na terenie kraju czas dostępu byłby wyjątkowo niski, przy założeniu, że kwestie administrowania serwerem i utrzymania właściwego poziomu bezpieczeństwa przebiegałyby bez zarzutu.
4. **Technologie** – istniałaby możliwość zastosowania dowolnych technologii. Wymagałyby jednak zakupu licencji, co mogłoby okazać się dodatkowym kosztem.

5. **Przenoszenie na inną platformę** – istniałaby możliwość stworzenia chmury hybrydowej i dalszego operowania na maszynach wirtualnych.

#### Wynajem maszyny wirtualnej

1. **Koszt** – przy wynajmie maszyny wirtualnej należy przez cały czas, gdy maszyna jest włączona, opłacać przynajmniej zasoby zużywane przez uruchomiony specjalnie na potrzeby wynajmującego system operacyjny. Do tego dochodzi utrzymanie uruchomionej tam aplikacji, a w przypadku skalowania zwielokrotnienie tych opłat zgodnie z ilością działających maszyn wirtualnych.
2. **Poziom skomplikowania** – uruchomienie tradycyjnie napisanej aplikacji nie byłoby skomplikowane, gdyż odbywałoby się dokładnie tak jak na każdym innym komputerze. Konfiguracja bardziej zaawansowanego środowiska, np. kilku maszyn wirtualnych obsługujących MongoDB, przysporzyłaby już więcej kłopotu.
3. **Czas dostępu** – dla Amazon EC2 oraz Google Compute Engine najbliższy region to Frankfurt w Niemczech, najbliższej znajduje się Azure Virtual Machines w Magdeburgu w Niemczech.
4. **Technologie** – jak w chmurze prywatnej.
5. **Przenoszenie na inną platformę** – istnieją metody przenoszenia obrazów maszyn wirtualnych pomiędzy chmurami.

#### Usługi typu PaaS w oparciu o maszyny wirtualne

1. **Koszt** – niższy niż w przypadku maszyn wirtualnych, ponieważ w PaaS maszyną wirtualną zarządza dostawca usługi, więc może zapewnić optymalizację. Nadal występuje płatność za sam fakt, że aplikacja jest aktywna w chmurze, chociaż nikt z niej nie korzysta.
2. **Poziom skomplikowania** – z jednej strony nie jest za wysoki, gdyż można tworzyć aplikacje z wykorzystaniem znanych i lubianych technologii, a chmury użyć w celu ich wdrożenia. Bardziej skomplikowane jest zastosowanie się do potrzebnych wzorców architektonicznych, dzięki którym aplikacje będą dobrze się skalować.
3. **Czas dostępu** – dla AWS Elastic Beanstalk oraz Google App Engine najbliższy region to Frankfurt w Niemczech, najbliższej znajduje się Azure Virtual Machines w Magdeburgu w Niemczech.

4. **Technologie** – wszystkie trzy chmury obsługują platformę .NET, jednak w przypadku Google jest to tylko .NET Core, wciąż uchodzący za nowość.
5. **Przenoszenie na inną platformę** – nie należałoby się na to nastawiać.

### Samodzielna instalacja orkiestratora kontenerów

1. **Koszt** – tak jak w maszynie wirtualnej, z tym że skalując kontenery można umieścić wiele kontenerów na jednej maszynie wirtualnej.
2. **Poziom skomplikowania** – należy dobrze zapoznać się z systemem, np. Kubernetes, nie tylko jego obsługą, lecz także instalacją i konfiguracją na wielu maszynach.
3. **Czas dostępu** – tak jak w maszynie wirtualnej.
4. **Technologie** – dowolne, raczej niekorzystające z PaaS usługodawcy.
5. **Przenoszenie na inną platformę** – łatwe i zgodne ze standardami.

### Usługa konteneryzacji w chmurach publicznych

1. **Koszt** – tak jak w samodzielnej instalacji orkiestratora kontenerów, a dodatkowo koszt usługi ułatwiającej zadanie wykorzystania maszyn wirtualnych do instalacji na nich orkiestratora. Jedynie w przypadku Azure taka usługa jest darmowa.
2. **Poziom skomplikowania** – należy dobrze zapoznać się z obsługą systemu orkiestracji kontenerów, np. Kubernetes.
3. **Czas dostępu, Technologie, Przenoszenie na inną platformę** – tak jak w samodzielnej instalacji orkiestratora kontenerów.

### Usługi typu PaaS oparte o kontenery w chmurach publicznych

1. **Koszt** – tak jak w usłudze konteneryzacji, przy czym skalowanie dla instancji kontenerów dobierane jest dynamicznie. Azure co do tego wprost pisze, że jeśli ruch jest zrównoważony, to taniej wyjdzie skorzystać z usługi konteneryzacji.

2. **Poziom skomplikowania** – teoretycznie jest niski, lecz w przypadku chęci utrzymania większej liczby kontenerów i tak będzie potrzebna migracja do samodzielnego zarządzania orkiestratorem.
3. **Czas dostępu, Technologie, Przenoszenie na inną platformę** – tak jak w samodzielnej instalacji orkiestratora kontenerów.

#### Serverless – usługi typu FaaS

1. **Koszt** – najniższy, ponieważ ta usługa jako jedyna nie wymaga płatności za aplikację, która beczynnie czeka aż ktoś wyśle zapytanie. Jednocześnie raczej nie ponosi się kosztu źle zaprojektowanej aplikacji, gdyż natura nanousług wyklucza tworzenie mikrousług o zbyt dużym zakresie odpowiedzialności. W razie gdyby funkcja mimo wszystko zbyt długo się wykonywała, są na to nałożone limity, które sprawią, że po pewnym krótkim czasie jej wykonywanie zostanie przerwane.
2. **Poziom skomplikowania** – wymaga nauczania się sposobu funkcjonowania usługi FaaS, ale dzięki temu nie trzeba martwić się o właściwą architekturę usług, co znacząco upraszcza proces tworzenia. Każda usługa stanowi osobny byt, więc jest wykluczone stworzenie zbyt skomplikowanej siatki zależności pomiędzy usługami.
3. **Czas dostępu** – najdalej znajduje się Google Functions w Belgii, następnie AWS Lambda we Frankfurcie w Niemczech, najbliżej znajduje się Azure Functions w Magdeburgu w Niemczech.
4. **Technologie** – język pisania usług jest praktycznie dowolny, jednakowoż z dużym prawdopodobieństwem będą bazować na innych usługach PaaS wybranego dostawcy.
5. **Przenoszenie na inną platformę** – biorąc pod uwagę specyficzne API dla usługi FaaS oraz zależności do innych usług PaaS, przenoszenie na inne platformy może być praktycznie niewykonalne.

#### Kubeless

1. **Koszt** – taki jak w samodzielnym instalowaniu orkiestratora kontenerów bądź z pomocą usługi CaaS.
2. **Poziom skomplikowania** – duży, ponieważ wymaga nauki orkiestratora (w tym być może jego instalacji) oraz doinstalowania Kubeless i nauki tego narzędzia.

3. **Czas dostępu** – tak jak w maszynie wirtualnej.
4. **Technologie** – dowolne.
5. **Przenoszenie na inną platformę** – w teorii łatwe, w praktyce nie wiadomo, czy projekt Kubeless w ogóle przetrwa próbę czasu i stanie się jakimkolwiek standardem.

### Serverless Framework

1. **Koszt** – taki jak w FaaS
2. **Poziom skomplikowania** – taki jak w FaaS, plus dodatkowo nauka Serverless Framework, który proponuje swoje rozwiązania na każdym etapie tworzenia funkcji.
3. **Czas dostępu** – tak jak w FaaS
4. **Technologie** – możliwość pisania we wielu językach, nawet jeśli chmura docelowa w praktyce go nie obsługuje. Z tym, że sprowadzając wszystkie platformy do jednego mianownika, ogranicza się możliwość skorzystania z innych usług PaaS chmury publicznej. Z kolei jeśli i tak doprowadza się do uzależnienia od wybranego dostawcy, to po co zakładać dodatkową warstwę abstrakcji.
5. **Przenoszenie na inną platformę** – przeniesienie kodu mikrousług byłoby łatwe, należałoby się jednak zapoznać także z zagadnieniami migracji danych.

### 3.3.2. Wybór chmury

Jako że praktycznie każde podejście jest możliwe do zrealizowania u każdego dostawcy, w niniejszej sekcji zostaną porównani pod względem pobocznych cech.

#### Amazon

Plusy:

- + Najczęściej wybierany dostawca usług chmurowych.
- + Innowacyjny – usługi EC2, SimpleDb, Lambda były pionierskimi rozwiązaniami.

### 3.3. ANALIZA

---

Minusy:

- Niejasny model rozliczeń, który nie budzi zaufania. Pobieranie po dolarze od każdego, kto chce przetestować usługę, nie wygląda dobrze. Skomplikowany model testowania usług, gdzie część usług jest do testowania za darmo, a za inne są pobierane opłaty.
- Konieczność podania karty kredytowej do rejestracji.
- Zdaniem autorki mało przyjazne panele administracyjne, przepełnione zbędną treścią.

#### **Azure**

Plusy:

- + Drugi najczęściej wybierany dostawca usług chmurowych.
- + Czasami innowacyjny, np. baza Cosmos Db.
- + Udostępnia bezpłatną subskrypcję dla studentów, która nie wymaga podawania karty kredytowej.
- + Zdaniem autorki przyjazny panel administracyjny.
- + Dobre wsparcie dla technologii .NET.
- + Centrum danych znajduje się najbliżej Polski.

Minusy:

- Niedopracowania objawiające się np. nie do końca przetłumaczonym interfejsem użytkownika.

#### **Google Cloud**

Plusy:

- + Trzeci najczęściej wybierany dostawca usług chmurowych.
- + Tworzony przez firmę, która dała początek wielu technologiom, na których opiera się współczesny Cloud Computing: GFS, MapReduce, BigTable, Kubernetes.
- + Pośród głównych graczy najdłużej prowadzi usługi PaaS.
- + Klarowne rozliczenia.



- + Zdaniem autorki bardzo przyjazny, wyjątkowo intuicyjny panel administracyjny. Zamiast zasypywać użytkownika stronami prezentującymi wiele przykładów, proponowany jest jeden samouczek w formie interaktywnej.

Minusy:

- Technologie .NET są obsługiwane w drugiej kolejności, np. w Google Functions póki co nie posiada dla nich wsparcia.
- Wiele stabilnych wersji usług zostało opublikowanych najpóźniej spośród pozostałych największych graczy. Wiele specyficznych usług dostępnych w AWS i Azure w ogóle nie ma odpowiednika w Google Cloud.
- Google Functions najbliżej znajduje się w Belgii.

### 3.3.3. Podsumowanie analiz

W świetle wymagań stawianych projektowi, najlepiej wypada podejście bazujące na **usługach w modelu serverless**. Wymaga ono zużycia najmniejszej możliwej liczby zasobów, więc jest tańsze od pozostałych metod. Na czas tworzenia i testowania aplikacji zużycie jest minimalne, ponieważ płatność odbywa się tylko za egzekucję funkcji, więc nie trzeba płacić za utrzymanie w chmurze aplikacji oczekującej na połączenia.

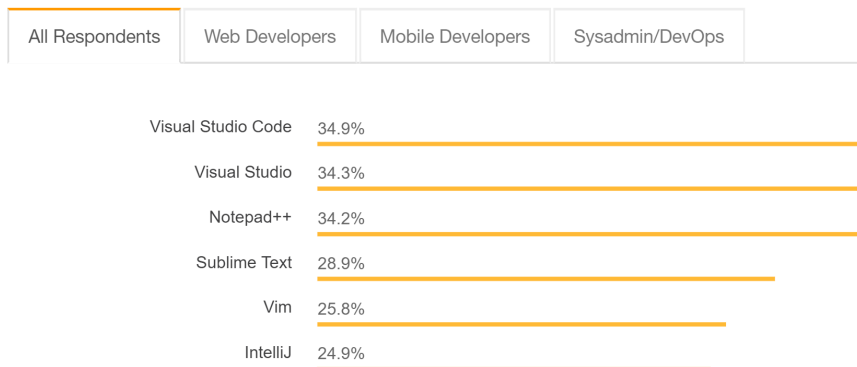
Podejście serverless zapewnia również najmniejszą barierę wejścia do chmury z wysoce skalowalną aplikacją. Nie trzeba w nim brać odpowiedzialności za właściwą architekturę mikrousług. Lektura książek[12, 15, 16] o mikrousługach uświadomiła autorkę pracy, że tworząc architekturę mikrousług należy równocześnie uwzględnić wiele czynników, a uchybienie któregoś może spowodować więcej szkody niż pożytku. Stworzenie dobrej architektury wymagałoby wcześniejszego doświadczenia, inaczej nie byłoby rzeczą dziwną, gdyby po wdrożeniu jednej architektury wyklarowała się potrzeba przepisanie jej na nowo.

Jeśli chodzi o chmurę, na której dobrze byłoby zrealizować to podejście, autorka zdecydowanie skłania się ku **Azure**. Na ten wybór w szczególności wpłynęła możliwość darmowej subskrypcji dla studentów, która daje największe poczucie bezpieczeństwa podczas pierwszych starć z chmurą. Wybór ten zdaje się również być bezpieczny pod względem dostępności technologii .NET oraz przyjaznych w użyciu narzędzi. Microsoft słynie z dobrej jakości

### 3.3. ANALIZA

narzędzi dla programistów, na potwierdzenie tego wystarczy spojrzeć na wyniki badania przeprowadzonego wśród programistów przez Stack Overflow w 2018 r. (rysunek 3.4) – dwa najczęściej używane edytory programistyczne to Visual Studio Code oraz Visual Studio.

#### Most Popular Development Environments



Rysunek 3.4: Fragment wykresu z badania Developer Survey 2018 pokazującego popularność różnych środowisk programistycznych. [74]

Gdyby jeszcze nie wynaleziono podejścia serverless, to za najlepszą z opcji uchodziłyby **kontenery**. Kontenery są znacznie lżejsze niż maszyny wirtualne, a do tego pozwalają tworzyć w technologiach, które nie są zarezerwowane dla wybranego dostawcy. Używając kontenerów wiele można zdziałać na lokalnej maszynie, gdyż aplikacja dysponuje wówczas wszędzie tym samym środowiskiem do uruchamiania.

W przypadku kontenerów należałoby poświęcić wiele uwagi tworzeniu właściwej architektury. Projektowanie mikrousług to **nie jest** rozbięcie zwykłej usługi na mniejsze kawałki, ponieważ w mikrousługach obowiązują inne zasady programowania i komunikowania ze sobą usług, niż w dużych usługach. Na przykład w zwykłym SOA dość typowe jest wiązanie ze sobą usług za pomocą usługi dostępu do bazy danych, jednej dla wszystkich usług. W mikrousługach byłby to podstawowy błąd architektoniczny.

Obecnie kontenery można wdrażać w chmurze na trzy sposoby. Zdaniem autorki, na sam początek wystarczyłaby usługa instancji kontenerów, ponieważ jest przystosowana do uruchamiania małej ilości kontenerów i nie wymaga konfigurowania maszyn wirtualnych czy orkiestratora. W drugiej kolejności należałoby wybrać pomiędzy samodzielną instalacją Kubernetes lub skorzystaniem z KaaS.

Usługa KaaS jest rekomendowaną opcją przez Kubernetes dla osób, które dopiero chcą się zaznajomić z tym narzędziem. Można wtedy od razu zetknąć się z obsługą oprogramowania, zamiast przechodzić przez proces

konfiguracji maszyn wirtualnych i samodzielnej instalacji. Jednak dla osoby, która już dobrze zna Kubernetes, czynności te mogą nie stanowić bariery, przy czym ich automatyzacja to dodatkowy koszt, np. w Amazon każda godzina działania klastra w usłudze KaaS to dodatkowe 0.20 USD. W przypadku Azure usługa KaaS jest darmowa, więc wybór pomiędzy KaaS a samodzielną instalacją nie stanowiłby szczególnej różnicy.



## Rozdział 4.

# Opis wdrożenia aplikacji

W tym rozdziale zostanie opisana konkretna implementacja rozwiązania z użyciem usługi Azure Functions.

Projekt składa się z:

- REST API (C#/.NET Framework 4.6.1, Visual Studio),
- aplikacji klienckiej na system Android (Java 8, Android API 24, Android Studio).

### 4.1. Prezentacja aplikacji

Na początek zostanie zaprezentowane, jaki jest sposób działania aplikacji przedstawionej w rozdziale 3.

#### 4.1.1. Podstawowa obsługa aplikacji

##### 1. Rejestracja użytkownika.

W celu korzystania z aplikacji należy założyć konto, podając login i hasło (rysunek 4.1).

##### 2. Logowanie

Posiadając konto należy się zalogować (rysunek 4.2). Logowanie zwraca token, który jest potrzebny, by móc korzystać z funkcji API dla zalogowanych. Token automatycznie zapisuje się do menedżera kont w Androidzie (rysunek 4.3). Raz uzyskany może zostać ponownie użyty do automatycznego logowania.

##### 3. Wstępna konfiguracja konta

Jest to strona, która pokazuje się tylko podczas pierwszego logowania. Po przejściu konfiguracji, logowanie prowadzi już prosto do strony głównej aplikacji.

Konfiguracja służy wprowadzeniu podstawowych danych o użytkowniku (rysunek 4.4), tj.: waga, wzrost, płeć, imię, data urodzenia, poziom aktywności fizycznej, ilość posiadanych pieniędzy.

Pieniądze wprowadza się za pomocą „portfeli” (rysunek 4.5) – na przykład można podać osobno kwotę na koncie w PLN, gotówkę w PLN i konto walutowe w EUR. Pozwoli to później lepiej kontrolować stan portfeli, przy czym nic nie stoi na przeszkodzie by przy tworzeniu statystyk sumować portfele z tej samej waluty.

#### 4. Strona główna

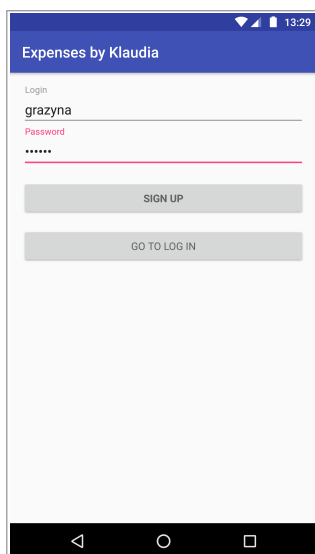
Zalogowany, skonfigurowany użytkownik łąduje na stronie głównej (rysunek 4.6). Jest to aktywność stworzona na bazie *navigation drawer*, czyli menu wysuwanego z boku ekranu (rysunek 4.7).

Z poziomu *navigation drawer* dostępne są opcje:

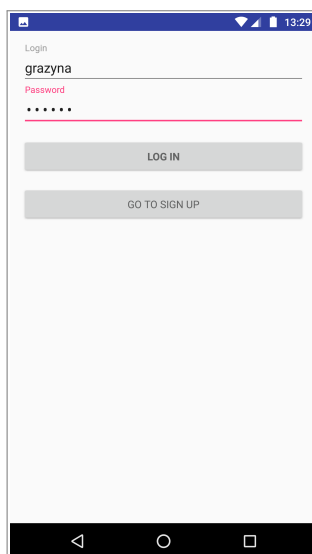
- *Podsumowanie* – podstawowa statystyka wyświetlana po zalogowaniu,
- *Profil* – zarządzanie profilem użytkownika (możliwość usunięcia konta),
- *Gospodarstwo domowe* – dla kont połączonych w grupę można tam podejrzeć statystykę dla poszczególnych kont,
- *Zaproszenia* – miejsce gdzie można zaprosić inne konto do grupy oraz potwierdzać takie zaproszenia,
- *Kategorie* – lista kategorii wraz z ustawionymi przelicznikami dla danej kategorii, jaki % płaci dana osoba z grupy.

Z poziomu menu typu „więcej opcji” jest dostępna opcja „wyloguj”.

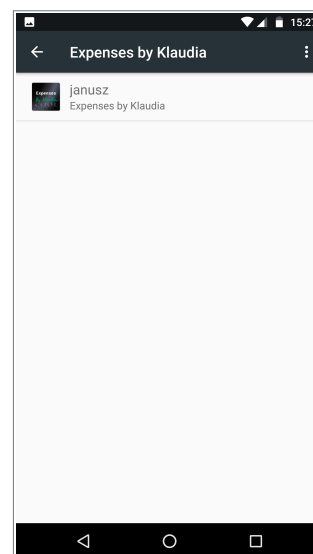
Na stronie „podsumowanie”, będącej domyślnie wyświetlanym fragmentem strony głównej, widnieje wielki przycisk pozwalający dodać wydatek.



Rysunek 4.1: Rejestracja



Rysunek 4.2: Logowanie



Rysunek 4.3: Przechowywanie danych zalogowanego użytkownika w Accounts Managerze

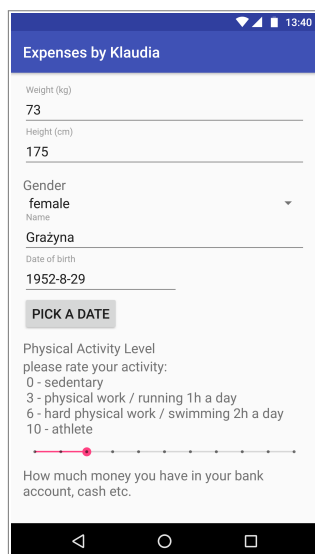
### 4.1.2. Dodawanie wydatku

Dzięki zastosowaniu kategorii z przelicznikami oraz listy portfeli walutowych, dodawanie wydatku prezentuje się bardzo prosto nawet mimo możliwości tworzenia docelowo skomplikowanych raportów o wielu osobach (rysunki 4.8, 4.9).

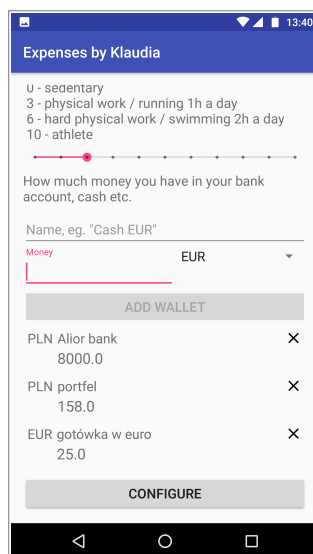
Aby dodać wydatek, należy:

1. **Wybrać datę**, jeśli ma być inna niż bieżący dzień. Można tego dokonać używając systemowej kontrolki kalendarza.
2. **Wybrać kategorię wydatku**. Jeśli konto należy do grupy, to od kategorii zależy jak później będą naliczane długi. *Przykład: gdy dana osoba dodaje wydatek na wspólne jedzenie, innym automatycznie naliczy się ile powinni oddać tej osobie, zgodnie z ich zapotrzebowaniem kalorycznym.*
3. **Podać sumę na paragonie**. Od tej sumy ewentualnie zostaną odjęte poszczególne pozycje na paragonie, które nie pasują do kategorii głównej.
4. **Zaznaczyć portfel**, jeśli domyślnie zaznaczony nie pasuje. Jest to źródło, z którego był opłacony wydatek. Tym sposobem wydatek może być w obcej walucie.

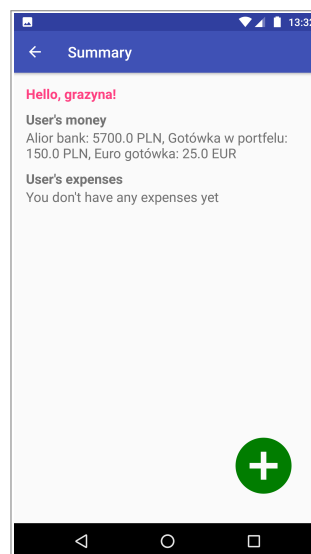
## 4.1. PREZENTACJA APLIKACJI



Rysunek 4.4: Dane do wstępnej konfiguracji



Rysunek 4.5: Portfele we wstępnej konfiguracji



Rysunek 4.6: Strona główna przed konfiguracją grupy

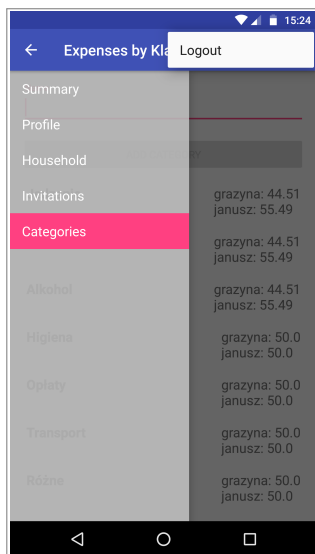
5. (opcjonalnie) **Podać szczegóły z paragonu.** Przydatna opcja odejmująca wiele pracy (rysunek 4.9). *Przykład: dodawany jest paragon z supermarketu, na nim samo jedzenie i do tego żel pod prysznic. Żeby zachować właściwe przypisanie do kategorii, nie trzeba ręcznie odejmować żelu pod prysznic i osobno go dodawać. Wystarczy podać go jako szczegół na paragonie pasujący do kategorii „higiena” i cała reszta dopasuje się automatycznie.*

### 4.1.3. Łączenie w gospodarstwo domowe

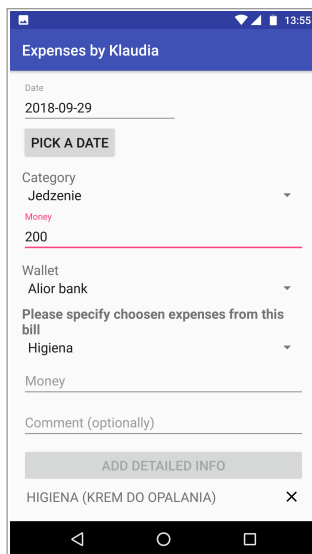
Aplikacja może działać w trybie pracy dla jednej osoby, ale jej główne przeznaczenie stanowi zapisywanie wydatków przez kilka osób. W tym celu należy skonfigurować gospodarstwo domowe. Scenariusz ten odbywa się w następujący sposób:

1. **Dodanie i skonfigurowanie kont** – każda osoba powinna najpierw założyć konto.
2. **Wysyłanie zaproszeń** – po zalogowaniu należy przejść do sekcji „zaproszenia” i tam wysłać zaproszenie danej osobie, podając jej login (rysunek 4.10).
3. **Akceptacja zaproszenia** – każda osoba posiadająca zaproszenie może je wyświetlić i zaakceptować w sekcji „zaproszenia” (rysunek 4.11).

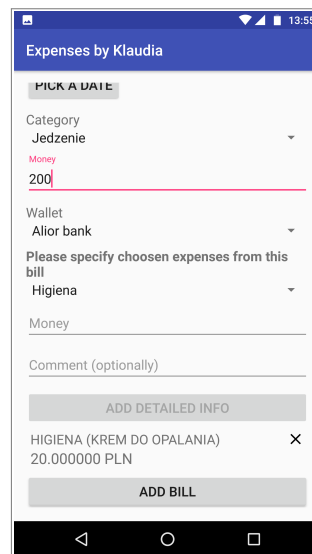




Rysunek 4.7: Nawigacja w aplikacji



Rysunek 4.8: Dodawanie wydatku



Rysunek 4.9: Dodawanie szczegółów paragonu

Po akceptacji zaproszenia konto zmienia status na należące do gospodarstwa domowego. W sekcji „podsumowanie” pojawiają się **dodatkowe wartości** (rysunek 4.12):

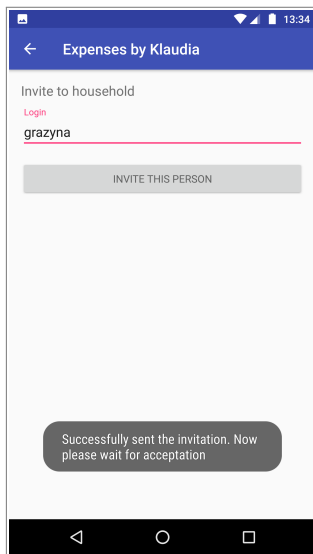
- wydatki całego gospodarstwa domowego, przedstawione jako suma wszystkich walut posiadanych przez wszystkich uczestników,
- pieniądze znajdujące się w gospodarstwie domowym, przedstawione jak wyżej,
- długi użytkownika i/lub zobowiązania innych osób względem użytkownika.

**Długi i zobowiązania** działają w taki sposób, że jeśli dana osoba dokonała wydatku w imieniu innej osoby (rysunek 4.13), to odpowiednio zaznaczona kategoria automatycznie naliczy tej osobie pieniądze do oddania (rysunek 4.14).

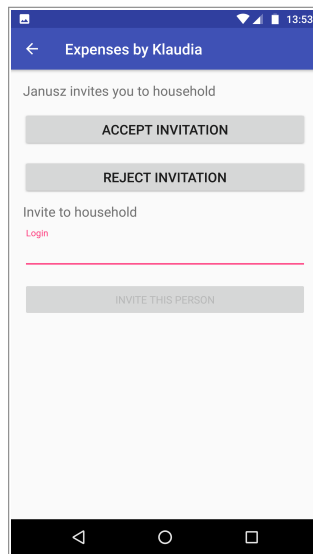
Dołączenie nowej osoby do gospodarstwa domowego powoduje **agregację kategorii** gospodarstwa oraz nowej osoby (rysunek 4.15). Domyślne kategorie posiadają swój sposób agregacji, tzn. jedzenie zawsze agreguje się zgodnie z zapotrzebowaniem kalorycznym, a pozostałe kategorie dzielą wydatki po równo na każdą z osób. Można także dodawać swoje kategorie.

**Stan finansów każdej z osób w gospodarstwie domowym** można podejrzeć w sekcji „gospodarstwo domowe”.

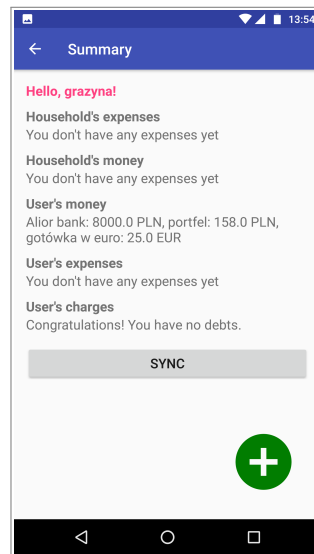
## 4.1. PREZENTACJA APLIKACJI



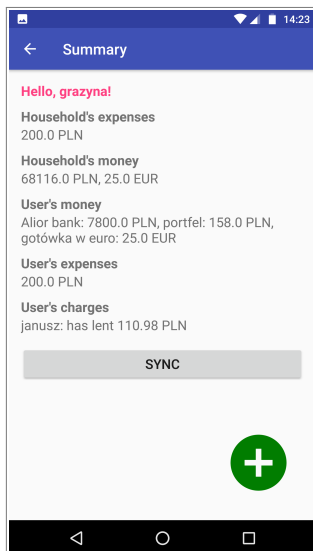
Rysunek 4.10: Wysłanie zaproszenia



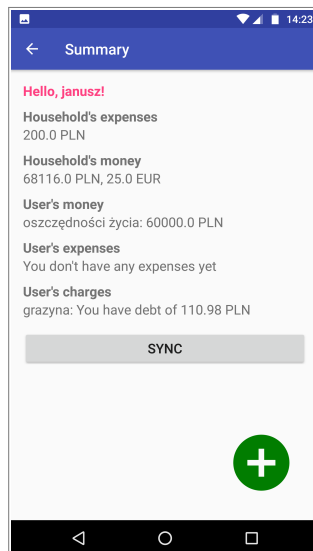
Rysunek 4.11: Akceptacja zaproszenia



Rysunek 4.12: Strona główna po akceptacji zaproszenia



Rysunek 4.13: Strona główna po dodaniu wydatku



Rysunek 4.14: Strona główna u drugiej osoby



Rysunek 4.15: Przeliczniki dla kategorii (automatycznie wyliczone)

## 4.2. Implementacja API

W niniejszej sekcji opisano, jak wyglądała praca z Azure Functions.

### 4.2.1. Teoretyczne podstawy Azure Functions

Azure Functions stanowi usługę wchodzącą w skład Azure App Services. Została wbudowana na bazie Azure WebJobs, która umożliwia uruchamianie zadań w tle.

Dostępne są dwie wersje środowiska uruchomieniowego Azure Functions: [73]

- 1.x – działa na .NET Framework;
- 2.x – działa na .NET Standard; do dnia 24.09 występowała w wersji poglądowej i zalecano korzystanie z 1.x.

Aplikacja utworzona w Azure Functions dostępna jest pod adresem postaci:

`https://NAZWA_APLIKACJI.azurewebsites.net/api/`

### Cechy funkcji

Każda funkcja to bezstanowy kod, który reaguje na dane zdarzenie. Jako kilka podstawowych przykładów **wyzwalaczy** (ang. *triggers*) można wymienić:

- Http trigger – obsługa zapytania HTTP.

Należy zdefiniować ścieżkę (np. `/api/person/get/name`), obsługiwane rodzaje zapytania (GET, POST itd.) oraz **poziom autoryzacji**.

Są obsługiwane 3 poziomy autoryzacji. Poziom **Anonymous** oznacza, że funkcja jest dostępna bez konieczności podawania klucza. W poziomach **Function** oraz **Admin** konieczne jest podanie klucza. W ramach aplikacji są zdefiniowane klucze hosta (ang. *host keys*) i tego typu kluczem można uzyskać dostęp do poziomu Admin oraz Function. Poza tym każda funkcja posiada swoją kolekcję kluczy, którymi można się do nich zautoryzować, jeśli mają dostęp Function.

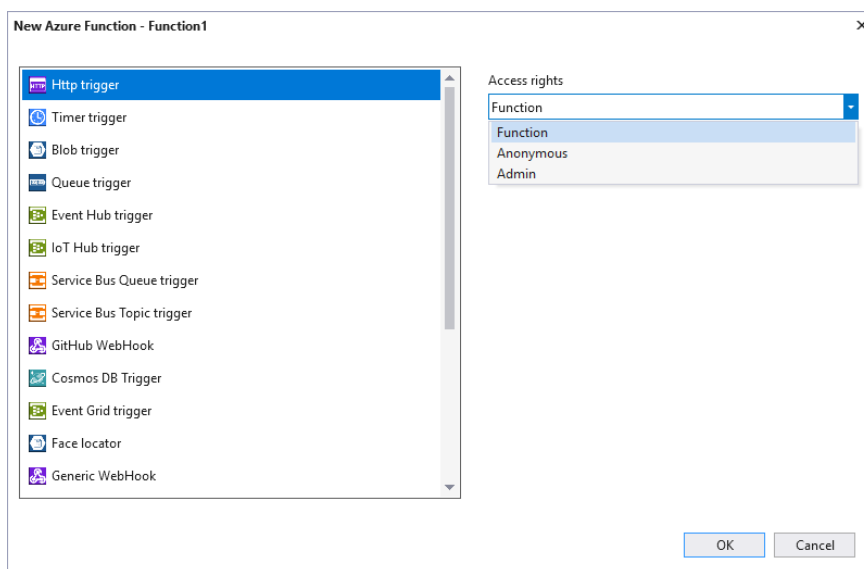
Klucz podaje się przez dodanie parametru `code` do zapytania HTTP, np.:

`/api/user/get/grazyna?code=miejsceNaToken`

- Timer trigger – funkcja zostanie wywołana po upływie określonego interwału czasowego.

Interwał określa się za pomocą wyrażenia CRON, np. `0 * * * * *` oznacza „0 sekund, każdej minuty, każdej godziny, każdego dnia, każdego miesiąca, każdego dnia tygodnia” (czyli co minutę). [78]

- Queue trigger – został dodany wpis do danej kolejki w Azure Queue Storage .
- Blob trigger – został dodany plik do kontenera w Azure Blob Storage.
- CosmosDb trigger – został dodany lub zmodyfikowany wpis w bazie Azure CosmosDb.



Rysunek 4.16: Tworzenie nowej funkcji w Visual Studio. Widoczne różne rodzaje wyzwalaczy oraz dostępne poziomy autoryzacji dla wyzwalacza Http.

Funkcja może posiadać także **wiązania** (ang. *bindings*) do danych, wejściowe lub wyjściowe. Może to służyć np. do odczytu danych z Azure Table Storage lub użycia zwracanego przez funkcję obiektu jako nowego wpisu do kolejki. [79]

Każda funkcja posiada wiązanie do dziennika **logów**. Jest to niezwykle istotny element funkcji, ponieważ (podobnie jak w mikrousługach) funkcje nie są typem aplikacji, gdzie żeby sprawdzić dlaczego coś nie działa można sobie na spokojnie prześledzić kod w trybie debuggowania. Bez szczegółowych logów trudno znaleźć przyczynę błędu.

## Budowa projektu

Do tworzenia aplikacji można podejść na dwa sposoby: [75]

1. każda funkcja to skrypt,
2. cała aplikacja to biblioteka.

**Skrypty** działają bardziej wieloplatformowo. Można je tworzyć w językach C#, F# oraz JavaScript (w wersji 2.x także poglądowo w Javie). Istnieje wtedy możliwość uruchamiania funkcji lokalnie przy pomocy paczki Azure Functions Core Tools dla menadżera pakietów npm. W wersji 2.x można w ten sposób postąpić na wielu platformach, wszędzie gdzie działa .NET Core, czyli m.in. na Linuksie i macOS. Paczki Core Tools można używać z poziomu linii komend lub korzystając z Visual Studio Code z doinstalowanym rozszerzeniem Azure Functions for Visual Studio Code.[76]

Aplikacje ze skryptów można edytować z poziomu portalu Azure.

Każda funkcja utworzona jako skrypt składa się z:

- **pliku ze skryptem**, gdzie w przypadku C# i F# obowiązują rozszerzenia .csx i .fsx oraz styl pisanie skryptu,
- **pliku *function.json***, znajdującego się w tym samym folderze. Plik ten opisuje m.in. wiązania (ang. *bindings*) z mechanizmami uruchamiającymi funkcję lub usługami takimi jak Azure Storage. Wskazuje się tam również rodzaj autoryzacji.

Wspólnie dla wszystkich funkcji definiuje się:

- **plik z zależnościami projektu**. Dla NuGet jest to *project.json*, dla npm *package.json* i dla Mavena *pom.xml*,
- **plik *host.json*** przechowujący opis konfiguracji na produkcji. Tam można np. zdefiniować przedrostek dla ścieżki API (inny niż domyślny */api*),
- (w wersji 2.x i pracy lokalnie) **plik *extensions.json*** rejestrujący wiązania występujące w paczkach NuGet. [77].

**Biblioteka** to podejście przeznaczone dla osób korzystających z Visual Studio i języka C#. Polega na tym, że funkcje w projekcie są prekompilowane do formatu .dll i w takiej formie publikowane do Azure.

Podejście to daje następujące korzyści: [80]

- + szybszy czas pełnego uruchomienia prekompilowanej funkcji względem skryptu,
- + korzystanie ze wszystkich ułatwień Visual Studio, łącznie z IntelliSense, pisanie testów, łączeniem z głównym repozytorium kodu itp.,
- + brak konieczności tworzenia pliku *function.json* – można korzystać z atrybutów WebJobs do definiowania wyzwalaczy i wiązań,
- + brak konieczności tworzenia plików *package.json* oraz *extensions.json* – zarządza nimi NuGet,
- + wszystkie potrzebne narzędzia instalują się wraz z narzędziami Azure Functions Tools.

Odbywa się to kosztem następujących niedogodności:

- nie można edytować kodu funkcji z poziomu portalu Azure,
- kod jest związany z Visual Studio.

### 4.2.2. Przygotowanie środowiska

W niniejszej sekcji zostanie opisany dobór narzędzi dla podejścia „biblioteka”, gdyż w taki sposób był tworzony projekt.

Do tworzenia aplikacji lokalnie wystarczy mieć zainstalowane **Visual Studio** z pakietem roboczym „Projektowanie dla platformy Azure” i doinstalowanym rozszerzeniem „Azure Functions and Web Jobs Tools”. Można w ten sposób tworzyć funkcje, które nie są zależne od usług Azure.

Jest możliwość symulowania lokalnie usług Azure Storage. W tym celu należy zainstalować:

- **Microsoft SQL Server**, gdyż za jego pomocą odbywa się symulacja,
- **Microsoft Azure Storage Emulator**, który należy mieć włączony, aby móc łączyć się z „lokalnym” Azure Storage.

Warto również zainstalować **Microsoft Azure Storage Explorer**. Jest to specjalna aplikacja służąca do zarządzania przechowywaniem danych w Azure. Pozwala zebrać w jednym miejscu zasoby z różnych kont Azure oraz różnych subskrypcji (np. symulowana „lokalna” i „Azure dla studentów”). Obsługuje Azure Storage oraz w wersji poglądowej Azure CosmosDb i Azure Data Lake Store.

W celu opublikowania aplikacji w chmurze należy posiadać **konto w Azure** i założyć tam **plan subskrypcji**, czyli ustalony sposób rozliczania się z dostawcą. Jednym z możliwych wyborów jest plan „Azure dla studentów”, gdzie nie trzeba podawać danych karty kredytowej, tylko uczelniany adres e-mail do weryfikacji. Dla omawianego projektu utworzono ten właśnie rodzaj subskrypcji.

Posiadając te dane można przystąpić do założenia **aplikacji funkcji** (ang. *Function app*). Należy wybrać opcję „Utwórz zasób” (ang. *Add resource*) i tam wyszukać Function App. W celu utworzenia aplikacji trzeba podać:

- nazwę aplikacji – pod tą nazwą aplikacja będzie dostępna w domenie `azurewebsites.net`, dlatego musi być unikalna;
- subskrypcję;
- grupę zasobów (ang. *resource group*) – jest to koncepcja łączenia w kolekcje zasobów utworzonych w różnych usługach Azure. Można wybrać istniejącą lub utworzyć nową;
- plan hostingu – wybór, czy płatność ma być tylko za zużycie, czy określić prognozowane zużycie zgodnie z ustalonym planem;
- lokalizację – Azure posiada centra danych na całym świecie, więc można wybrać lokalizację znajdującą się względnie blisko miejsca, gdzie ma być używana aplikacja. Zmniejsza to opóźnienia w komunikacji;
- magazyn (ang. *storage*) – do działania jest wymagany zasób Azure Storage. Można użyć istniejącego lub utworzyć nowy;
- czy włączyć Application Insights – to jest dodatkowo płatna usługa pozwalająca na lepsze monitorowanie aplikacji.

Dla omawianego projektu wybrano kolejno: nazwę *expenses-by-klaudia*, subskrypcję *Azure dla studentów*, nową grupę zasobów *expenses-by-klaudia*, plan „Zużycie”, lokalizację „Europa Zachodnia”, nowy magazyn *expenses-byklauda8d*, Application Insights *wyłączone*.

Wykonawszy powyższe kroki można z poziomu Visual Studio w opcjach projektu wybrać „**Publikuj**” i tam podać dane do konta Azure, wybrać subskrypcję oraz aplikację funkcji.

Istnieje także możliwość konfiguracji CI/CD, tak że każdy commit do repozytorium będzie skutkował wdrożeniem nowej wersji aplikacji funkcji.

### 4.2.3. Baza danych Azure Table Storage

Zanim zacznie się pisać funkcje operujące na bazie danych, wypada wybrać konkretną bazę oraz zapoznać się z zasadami korzystania z niej.

Dla Azure Functions jako bazę danych można wybrać: Azure Table Storage oraz Azure Cosmos Db. Ponieważ Cosmos Db jawiła się jako nowość, a większość dostępnych materiałów była przygotowana pod Table Storage, dla projektu wybrano Table Storage. Cosmos Db oferowała podobną funkcjonalność oraz możliwość tworzenia baz dokumentowych i grafowych.

Table Storage to baza NoSQL typu *wide-column*. Bazę dzieli się na tabele, jednak nie są to tabele znane z baz relacyjnych, lecz bardziej logiczna jednostka grupująca encje rozpostarte na różnych maszynach.

Podstawowe atrybuty encji to: [81]

- **Partition key** – encje posiadające ten sam *Partition key* zawsze znajdują się na tej samej maszynie. Pozwala to na transakcyjną edycję wielu rekordów naraz (ang. EGT – *Entity Group Transactions*).
- **Row key** – klucz rozróżniający pomiędzy sobą encje dzielące ten sam *Partition key*.
- **ETag** – wartość Timestamp, kiedy encja została utworzona.

Ważne informacje o bazie: [81]

- Każda encja może mieć rozmiar maksymalnie 1 MB oraz 252 dodatkowe kolumny (wraz z tymi obowiązkowymi jest 255).
- Dane zawsze sortują się zgodnie z kolejnością alfabetyczną w kluczu i to jest jedyna kolejność, w jakiej mogą zostać zwrócone. Można je filtrować tylko przez porównywanie napisów. Najlepsze zapytania to te, które wyszukują po samym kluczu. Mniej wydajne wyszukują także po wartości kolumny. Najgorsze zapytania to te, w których nie podano *Partition key*, ponieważ wymagają przeszukania wszystkich węzłów w poszukiwaniu pasujących wartości. [82]
- Chcąc zachować spójność pomiędzy encjami o różnej wartości *Partition key*, należy korzystać z kolejek Azure Queue Storage. Jednocześnie nie zastępuje to EGT, gdyż nie gwarantuje tego, że nikt nie będzie wykonywać operacji odczytu danych, które właśnie podlegają edycji.
- w Table Storage bardziej kosztowne jest zbyt częste odpytywanie bazy danych, niż duplikowanie danych.



Z powyższych względów, baza tego typu wymaga zupełnie innego podejścia do projektowania niż w modelu relacyjnym. Za to model relacyjny może być przydatny, aby odpowiedzieć sobie na pytania:

- **W jakiej kolejności mają być zwracane dane?**

Właściwą kolejność należy zapewnić przez odpowiedni klucz. Przykładowo, żeby móc zwrócić ostatnio dodane wartości, w kluczu powinna występować odwrócona data. Jeśli mogą być zwracane na różne sposoby, możliwe że będzie trzeba je zduplikować z użyciem innych kluczy.

- **W których miejscach występują listy elementów powiązanych z innymi?**

Relacje jeden do wielu i wiele do wielu odwzorowuje się w ten sposób, że listy są konwertowane do formatu JSON i zapisywane jako część encji, z którymi są powiązane. Wiąże się to z duplikowaniem danych.

- **Czy istnieje potrzeba agregowania danych?**

Należy ograniczać liczbę zapytań do bazy, a agregacja danych zdecydowanie tego nie ułatwia. Podobnie jak listy, zagregowaną wartość należy przechowywać w encji, która z tej wartości korzysta. Przeprowadzenie agregacji jest wydajniejsze, jeśli encje znajdują się na tej samej partycji.

- **Czy są dane, które rzadko się zmieniają lub dane, które często się zmieniają?**

W zależności od tego należy manewrować pomiędzy większym duplikowaniem danych, a większą ilością operacji odczytu.

- **Czy są dane, które mogą być tak duże, że przekroczą 1 MB lub 252 kolumny?**

Należy wówczas zastosować inne techniki, np. obrazki przechowywać w Blob Storage lub podzielić na kilka mniejszych encji z kluczem złożonym, tak by móc je łatwo ze sobą powiązać.

- **Czy są dane, co do których wystąpi potrzeba masowego usuwania?**

Takie dane łatwiej będzie usuwać, usuwając od razu całą partycję.

Szczegóły na temat projektowania modelu danych można przeczytać na stronie *Azure Storage Table Design Guide*.<sup>[81]</sup>

Zgodnie ze spostrzeżeniem autorki, dobrą metodą poszukiwania modelu danych jest mieć przed sobą projekt ekranów. To najlepiej odzwierciedla, które dane znajdują się blisko siebie, tak by móc się do nich dostać przy pomocy jednego zapytania.

### 4.2.4. Pierwsze funkcje

Pierwszymi funkcjami, jakie powstały w projekcie, były funkcje przeznaczone do rejestracji i zalogowania użytkownika o dostępie anonimowym:

- AddUser – dodawanie użytkownika,
- GetSalt – pobranie soli przypisanej do użytkownika o danym loginie,
- LogIn – logowanie z użyciem nazwy użytkownika i hasła z solą, podanego funkcji skrótu.

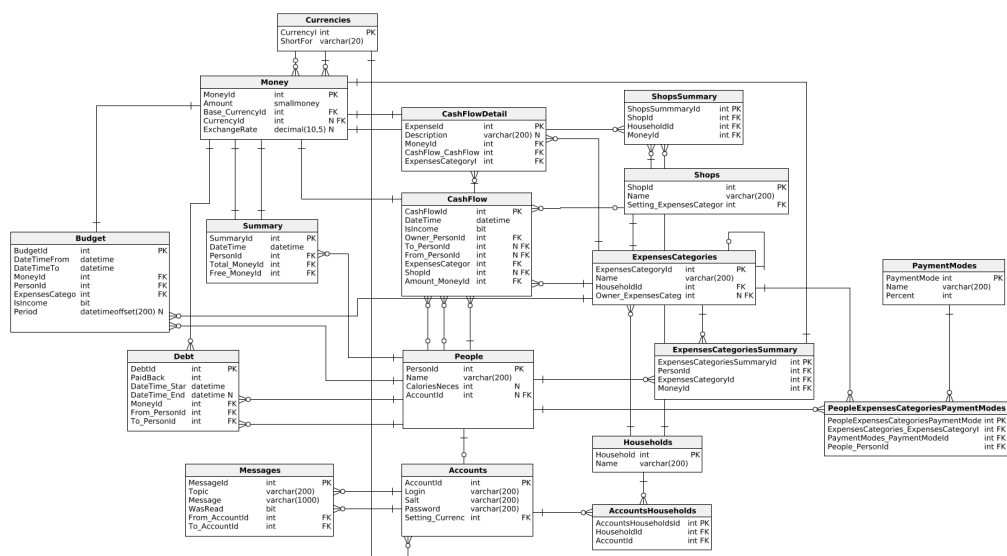
W przyszłości do logowania można wykorzystać usługę Azure Active Directory.

Następnie, korzystając z Key Management API [83], utworzono funkcje o dostępie Function, do których dostęp mają tylko zalogowani użytkownicy:

- DeleteUser – usuwanie użytkownika,
- ConfigureUser – przesłanie danych ze wstępnej konfiguracji,
- InviteToHousehold – wysyłanie zaproszenia,
- GetNewMessages – wyświetlanie wiadomości od zadanej daty,
- AcceptInvitationToHousehold – akceptowanie zaproszenia,
- AddCategory – dodawanie kategorii,
- GetCategories – lista kategorii dla danego użytkownika,
- GetWallets – pobieranie listy portfeli użytkownika.

Na etapie tych funkcji pojawiły się następujące wyzwania:

1. Jak zaprojektować bazę?
2. Jak testować funkcje?
3. Jak korzystać z wyzwalacza Http i wiązania z Table Storage?



Vertabelo

Rysunek 4.17: Model relacyjny pomagający zrozumieć dane w aplikacji.

W celu uzyskania odpowiedzi na pytanie 1. stworzono wstępny model relacyjny (rysunek 4.17) dla tego typu aplikacji oraz projekt ekranów.

Model relacyjny był pomocny w zrozumieniu zależności pomiędzy danymi, ponieważ denormalizacja danych w bazie NoSQL również jest oparta na tych zależnościach. Projekt ekranów natomiast uświadomił, że w przeciwieństwie do modelu relacyjnego, model NoSQL nie może być uzyskany w sposób teoretyczny, lecz w oparciu o dane, które razem występują w aplikacji.

W związku z powyższym autorka doszła do wniosku, że warto utworzyć **aplikację testową**, która jednocześnie będzie odzwierciedleniem ekranów oraz ułatwi testowanie API. Aplikacja testowa powstała w dobrze znanej autorce technologii WPF. Aplikacja ta przybliżyła również problem korzystania z API – z tego względu powstała **biblioteka repozytorium** przykrywająca wywołania poszczególnych punktów końcowych API.

Pisanie kodu pierwszych funkcji stanowiło nieoczekiwany problem. Zawiłości związane z różnymi technikami tworzenia na Azure powodowały, że ciężko było znaleźć odpowiednie przykłady kodu, które pomogłyby poprawnie zadeklarować wyzwalacze i wiązania. Mnogość języków programowania oraz rozbieżność między podejściem skryptowym a bibliotecznym nie pomagają wdrożyć się w tym temacie.

Kolejnym nieoczekiwanym problemem było pozyskanie kluczy dostępu dla funkcji. Poziom autoryzacji jest jedną z pierwszych rzeczy opisujących funkcje, a porady jak dodać klucze sprowadzają się do generowania ich z po-

ziomu portalu Azure. W dokumentacji poniekąd brakuje omówienia tematu, jak dokonać tego programistycznie, lub choćby czemu nie robić tego programistycznie. Dowiedziawszy się o Key Management API autorka użyła go do generowania kluczy podczas rejestracji użytkownika.

### 4.2.5. Funkcje z dostępem na klucz

Po wdrożeniu aplikacji na Azure niemiłym zaskoczeniem było to, że Key Management API nie mogło działać poprawnie bez specjalnego tokenu JWT (ang. *JSON Web Token*), będącego elementem Azure Active Directory. Token ten można było uzyskać programistycznie korzystając z Kudu API dostępnego pod domeną `*.scm.azurewebsites.net`. Dodatkowo, żeby zautoryzować się w Kudu, należy w kodzie programu zawrzeć login i hasło do Azure.

Poziom komplikacji w uzyskaniu klucza stał się tym bardziej niezrozumiały. Konieczność podawania danych do Azure utrudniła również proces CI/CD, bo plik z danymi był potrzebny dla działania aplikacji, a nie mógł być dodany do repozytorium kodu.

### 4.2.6. Azure Queue Storage

Funkcjonalność dodawania wydatku musiała zostać zaimplementowana z użyciem kolejki, ponieważ w razie gdyby wielu użytkowników zdecydowało się dodawać wydatki w tym samym czasie, mogłoby dojść do uszkodzenia danych. Kolejka daje pewność, że zadania będą przetwarzane po kolei.

Sposób realizacji tego zagadnienia pokazuje, jak bardzo różne jest podejście skalowalnych aplikacji z bazą NoSQL od tradycyjnych, monolitycznych aplikacji korzystających z relacyjnych baz danych.

Dla dodawania wydatku powstały następujące funkcje:

- `GetDataForCashFlow` – zwraca DTO z kategoriami i portfelami, na wypadek gdyby użytkownik nie miał ich jeszcze w cache aplikacji klienckiej,
- `AddCashFlow` – realizuje dodawanie wydatku,
- `DequeueAddCashFlow` – realizuje automatyczne naliczanie, kto ile zapłacił w danym gospodarstwie domowym za nowy wydatek,
- `GetCashSummary` – zwraca DTO z podsumowaniami.

**AddCashFlow** dodaje wydatek po kilka razy, aby w przyszłości umożliwić wybieranie wydatków różnych osób, ale powiązane innym elementem. Mogą to być wydatki całego gospodarstwa domowego, wydatki jednej osoby, wydatki w danej kategorii, wydatki w danym sklepie. Opiera się to na założeniu, że dane o wydatku raczej nie będą podlegać częstej edycji, za to mogą być często wyświetlane. Stąd też decyzja projektowa o duplikowaniu danych.

Aby listy wydatków mogły być pobierane w kolejności **od najnowszych do najstarszych**, należało uwzględnić właściwość Table Storage polegającą na umieszczaniu kolejnych encji w kolejności alfabetycznej. Chcąc wybierać encje na podstawie daty, należałoby w kluczu umieścić datę. Umieszczenie daty w zwykłym formacie umożliwiłoby wybieranie encji w kolejności od najstarszych. W związku z tym, aby móc wybierać najnowsze encje, data musi być zapisana odwrotnie.

Datę można odwrócić zakładając pewną datę minimalną oraz datę maksymalną. Wówczas bieżącą datę można umieścić w odległości od daty maksymalnej równej rzeczywistej odległości od daty minimalnej. Tak przekodowane daty zapewniają odwróconą chronologię.

Kolejny element klucza stanowi GUID (ang. *globally unique identifier*) upewniający, że żadna para wydatków dodana w podobnym czasie nie otrzyma tego samego klucza.

Reprezentację wydatku w bazie, w odniesieniu do gospodarstwa domowego oraz danej osoby, można obejrzeć w tabeli 4.1.

PartitionKey	RowKey
household_janusz	householdCashflow_7982.04.04_23:59:59_a5ac98f0-1536-4a2e-be0f-be0faf0781a5
household_janusz	userCashflow_grazyna_7982.04.04_23:59:59_a5ac98f0-1536-4a2e-be0f-be0faf0781a5

Tablica 4.1: Wielocłonowe klucze jednego wydatku dodanego dwa razy, z odwróconą datą

Korzystanie z kolejek przysparza niedogodność w postaci **dużych opóźnień**. Zanim zacznie być przetwarzany nowy wpis z kolejki, mija do kilkunastu sekund. Z kolei w aplikacji klienckiej trzeba sprawić wrażenie, jakby wszystko przebiegało błyskawicznie. Trzeba wtedy pewne kalkulacje wykonać lokalnie, aby widok mógł się odświeżyć, a na rzeczywiste zmiany na serwerze należy poczekać.

### 4.3. Aplikacja na Android

Wybór systemu Android został podyktowany popularnością tego systemu. Pomimo że autorka jest entuzjastką technologii .NET, zdecydowała się na stworzenie natywnej aplikacji. Wyjście poza stos technologiczny .NET stanowi w praktyce o interoperacyjności Azure Functions.

Przy wyborze technologii autorce zależało na możliwie dużej dostępności materiałów, dlatego postanowiła stworzyć projekt w **Javie 8** oraz korzystać z kontrolki użytkownika dostępnych w **Android Studio**.

Do łączenia się z API aplikacja korzysta ze specjalnie utworzonej w tym celu **biblioteki**. Biblioteka zajmuje się budowaniem właściwych zapytań, w tym np. zastosowania funkcji skrótu dla hasła z solą. Funkcja skrótu z .NET Framework oraz typy danych użyte do tworzenia obiektów DTO w komunikacji z API okazały się kompatybilne z odpowiednikami w Javie. Zostało to również przetestowane przy pomocy **testów jednostkowych**.

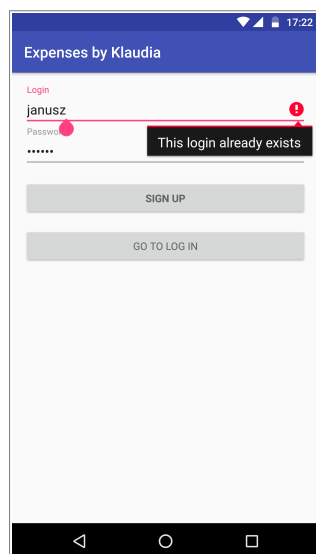
Wszystkie zapytania tworzone w aplikacji wysyłają się **asynchronicznie**. Na czas wykonywania zadań w aplikacji pokazuje się kręcące kółko ukazujące postęp. W razie błędów użytkownik jest informowany o problemach w komunikacji (przykład widać na rysunku 4.18). Do uproszczenia wielu zadań z tym związanych korzystano z wyrażień lambda z Javy 8.

Dużo uwagi przyłożono do zapewnienia o **poprawności danych**. Mniej zapytań kończących się odpowiedzią „Bad Request” to szybsze działanie aplikacji i mniejszy koszt korzystania z Azure Functions. Dlatego nawet w prostej aplikacji dobra walidacja danych jest bardzo istotna.

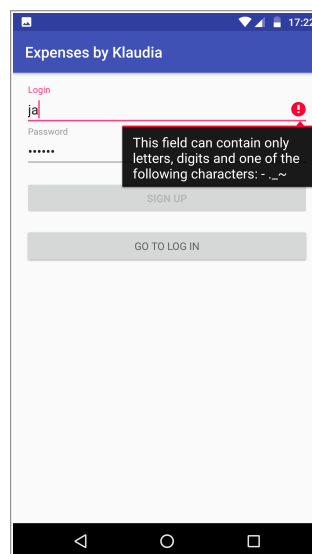
W aplikacji został zaimplementowany mechanizm walidacji szczegółowo informujący o błędach już na etapie wypełniania pola (przykład na rysunku 4.19). Każde pole posiada swoją listę reguł walidacyjnych i brak przestrzegania którejs z nich skutkuje poinformowaniem o tym fakcie użytkownika oraz zablokowaniem możliwości przesłania formularza.

Aplikacja nigdzie nie przechowuje hasła użytkownika. Jedynie zapisuje klucz uzyskany po zalogowaniu do systemowego **Accounts Managera**. Podczas uruchamiania aplikacji sprawdzane jest, czy istnieje konto w Accounts Managerze i czy posiada klucz. Jeśli tak, to użytkownik może się automatycznie zalogować za pomocą tego klucza. W wiadomości zwrotnej jest przekazywana informacja o tym, czy konto było skonfigurowane, i w zależności od tego użytkownik zostaje przekierowany do strony konfiguracji bądź do strony głównej.

Dla łatwej obsługi aplikacji wykorzystano mechanizm *navigation drawer*,



Rysunek 4.18: Czytelnie przedstawiona odpowiedź z serwera



Rysunek 4.19: Szczegółowa informacja walidacji formularza

polegający na wysuwanym menu po lewej stronie. Składa się z aktywności głównej oraz fragmentów dla każdej z pozycji występującej w menu.

Aby ograniczyć liczbę zapytań do serwera, aplikacja zapisuje ostatnio pobrane dane wraz z datą ich pobrania i zgodnie z ustalonym interwałem sprawdza, czy już czas znów je pobrać.





# Podsumowanie

Przed przystąpieniem do niniejszej pracy autorka posiadała wiedzę sprzed kilku lat – że istnieją platformy pozwalające wdrażać w chmurze aplikacje utworzone w określonych technologiach. Na etapie przeglądu literatury okazało się, że pogląd ten nie ma już prawie nic wspólnego z dzisiejszą rzeczywistością. Ze wszystkich stron wybijała się ekscytacja nowymi rewolucyjnymi technologiami i kolejnymi najlepszymi podejściami do pracy z chmurą.

Połączenie tego wszystkiego w całość umożliwiło spojrzenie z szerszej perspektywy na wszystkie te nowości oraz określić jakich obszarów dotyczą. Pozwoliło to wyłonić podejście, które prawdopodobnie najlepiej wypełnia swoje zadanie spośród dostępnych opcji. Cel pracy został osiągnięty.

Podczas opracowywania poruszanych zagadnień dokonano kilku spostrzeżeń.

Technologie chmurowe dążą do jak największych uproszczeń. Trwa wyścig o to, kto opublikuje narzędzie, które jeszcze bardziej ułatwi pracę. Paradoksalnie zamiast do uproszczenia sprawy, poniekąd **prowadzi to do poziomu skomplikowania znacznie utrudniającego bycie na bieżąco w temacie**. Wciąż jest za wcześnie, by wszystkie te rewolucje się ustabilizowały i znalazły swoje miejsce w bardziej obiektywnych wydawnictwach.

Materiały szkoleniowe dotyczące chmur są zazwyczaj **bardzo mało obiektywne**: książki wynoszą na piedestał wybrane narzędzie lub chmurę, blogi dotyczą tego w czym ich autorzy się akurat obracają, a chmury publiczne tworzą wrażenie, jakby poza nimi nic nie istniało. Znaczna część materiałów przypomina bardziej broszury marketingowe niż rzetelne źródła. Za przykład można podać prezentacje na Youtube, gdzie na jednym filmie występująca osoba zachwala Azure Kubernetes Service jako pozbawione wad rozwiązanie wszystkich problemów, a na trochę nowszym filmie ta sama osoba opowiada, że Kubernetes Service jednak stwarzał pewne utrudnienia, ale rozwiązanie stanowi Azure Container Instances. Przy takiej narracji każda nowość jawi się jako przewrót technologiczny, a gdy takie nowości pojawiają się praktycznie co miesiąc, ciężko już stwierdzić, co rze-

---

czywiście stanowi kamień milowy, a co nie.

Projekt programistyczny stworzony w ramach pracy pokazał, jak w praktyce wygląda tworzenie wysokoskalowalnej aplikacji dla chmury. Pomimo iż rzeczywiście wiele aspektów zostało ułatwionych, to wciąż wymaga przyswojenia **sporej dawki wiedzy**, zanim stworzy się aplikację przy pomocy wybranego podejścia. Pojawiają się również przeszkody spowodowane używaniem rozwiązań, które nie zostały dotychczas wystarczająco dobrze opisane w dokumentacji oraz nie wyłoniono jeszcze standardowych problemów danej technologii. Może pojawić się pytanie, czy dana aplikacja naprawdę musi być gotowa obsłużyć tak wielki ruch, by warto było godzić się na utrudnienia spowodowane korzystaniem z nie do końca dojrzałych technologii oraz wzorców architektonicznych dla mikrousług i baz NoSQL?

Ambiwalentne wrażenie budzi dążenie do **monopolizacji usług chmurowych**. Wielkość centrów danych największych dostawców istotnie tworzy iluzję nieskończonych zasobów komputerowych, lecz prowadzi to do sytuacji, gdy inne firmy mogą mieć zbyt małą siłę przebicia, by jakkolwiek konkurować z gigantami chmurowymi. Wtedy nawet posiadając innowacyjny produkt ciężko liczyć na sukces, jeśli po chwili ta sama funkcjonalność będzie dostępna u wszystkich największych graczy, a w razie nieuczciwych praktyk rynkowych pokażą jeszcze potencjał swoich działów prawnych.

Podsumowując, prawdopodobnie najważniejszą rzeczą potrzebną do podjęcia właściwych decyzji jest staranie się o obiektywizm. O ten z kolei trudno przy niskiej dostępności aktualnych materiałów o wydźwięku bardziej akademickim niż marketingowym. Warto zadać sobie trud odnalezienia rzetelnych źródeł. Przy tak szybkim tempie rozwoju technologii chmurowych należy śledzić ten temat, gdyż inaczej istnieje ryzyko skończenia jak programista kart perforowanych w dobie komputerów PC.

# Spis rysunków

1.	Rosnące zainteresowanie Internetem Rzeczy [26] . . . . .	7
1.1.	Różni użytkownicy chmury operujący na jej różnych war- stwach [1] . . . . .	9
1.2.	Serwery wirtualne korzystające z puli zasobów [1] . . . . .	10
1.3.	Ekonomia skali rosnąca wraz z ilością użytkowników chmury [1] . . . . .	11
1.4.	System bez wirtualizacji [5] . . . . .	15
1.5.	Wirtualizacja sprzętowa [5] . . . . .	15
1.6.	Porównanie serwerów wirtualnych i kontenerów [5] . . . . .	18
1.7.	Schemat sieci CDN. [23] . . . . .	32
1.8.	Optymalizacja strumieniowania wideo w sieciach CDN dru- giej generacji. [23] . . . . .	32
1.9.	Zastosowanie Fog Computing w samoprowadzącym się samo- chodzie. [25] . . . . .	33
1.10.	Odpowiedzialność dostawcy usług chmurowych przy różnych ich rodzajach [26] . . . . .	36
2.1.	Popularność dostawców usług chmurowych [29] . . . . .	41
2.2.	Graficzny interfejs pozwalający skonfigurować Kubernetes w Go- ogle Kubernetes Engine . . . . .	54
2.3.	Porównanie ilości warstw współdzielonych przez różne typy komponentów osadzonych w chmurze. [31] . . . . .	57
2.4.	Oddzielenie logiki aplikacji od logiki dostawcy usługi FaaS przy pomocy Serverless Framework [7] . . . . .	59
3.1.	Wykres obrazujący problem tradycyjnego podejścia do ska- lowania aplikacji [32] . . . . .	64
3.2.	Skalowanie wertykalne – wraz z rosnącym zapotrzebowaniem wymiana serwera na taki z lepszymi parametrami [1] . . . . .	65

3.3. Skalowanie horyzontalne – wraz z rosnącym zapotrzebowaniem dokładanie serwerów o tej samej mocy [1] . . . . .	65
3.4. Fragment wykresu z badania Developer Survey 2018 pokazującego popularność różnych środowisk programistycznych. [74] . . . . .	74
4.1. Rejestracja . . . . .	79
4.2. Logowanie . . . . .	79
4.3. Przechowywanie danych zalogowanego użytkownika w Accounts Managerze . . . . .	79
4.4. Dane do wstępnej konfiguracji . . . . .	80
4.5. Portfele we wstępnej konfiguracji . . . . .	80
4.6. Strona główna przed konfiguracją grupy . . . . .	80
4.7. Nawigacja w aplikacji . . . . .	81
4.8. Dodawanie wydatku . . . . .	81
4.9. Dodawanie szczegółów paragonu . . . . .	81
4.10. Wysłanie zaproszenia . . . . .	82
4.11. Akceptacja zaproszenia . . . . .	82
4.12. Strona główna po akceptacji zaproszenia . . . . .	82
4.13. Strona główna po dodaniu wydatku . . . . .	82
4.14. Strona główna u drugiej osoby . . . . .	82
4.15. Przeliczniki dla kategorii (automatycznie wyliczone) . . . . .	82
4.16. Tworzenie nowej funkcji w Visual Studio. Widoczne różne rodzaje wyzwalaczy oraz dostępne poziomy autoryzacji dla wyzwalacza Http. . . . .	84
4.17. Model relacyjny pomagający zrozumieć dane w aplikacji. . .	91
4.18. Czytelnie przedstawiona odpowiedź z serwera . . . . .	95
4.19. Szczegółowa informacja walidacji formularza . . . . .	95

# Spis tablic

1.1. Porównanie dat udostępniania usług chmurowych wśród głównych dostawców . . . . .	17
1.2. Główne różnice pomiędzy Cloud Foundry i Kubernetes . . . .	21
2.1. Główne interfejsy chmur publicznych dla chmur hybrydowych	44
2.2. Gotowe rozwiązania dla chmur hybrydowych . . . . .	44
4.1. Wieloczłonowe klucze jednego wydatku dodanego dwa razy, z odwróconą datą . . . . .	93



# Bibliografia

- [1] Sandeep Bhowmik. *Cloud Computing*. 1st. New York, NY, USA: Cambridge University Press, 2017.
- [2] Jothy Rosenberg i Arthur Mateos. *Chmura obliczeniowa. Rozwiązania dla biznesu*. Gliwice: Helion, 2011.
- [3] Rick Farmer, Rahul Jain i David Wu. *Cloud Foundry for Developers*. 1st. Birmingham, UK: Packt Publishing Ltd., wrz. 2017.
- [4] Kelsey Hightower, Brendan Burns i Joe Beda. *Kubernetes: Up and Running*. 1st. O'Reilly Media, Inc., 2017.
- [5] Nick Antonopoulos i Lee Gillam. *Cloud Computing. Principles, Systems and Applications*. 2nd. Cham, Switzerland: Springer International Publishing AG, 2017.
- [6] Florian Klaffenbach, Jan-Henrik Damaschke i Oliver Michalski. *Implementing Azure Solutions*. 1st. Birmingham, UK: Packt Publishing Ltd., maj 2017.
- [7] Maddie Stigler. *Beginning Serverless Computing. Developing with Amazon Web Services, Microsoft Azure, and Google Cloud*. 1st. Apress, 2018.
- [8] Joakim Verona. *Practical DevOps*. 1st. Birmingham, UK: Packt Publishing Ltd., 2016.
- [9] Gigi Sayfan. *Mastering Kubernetes*. 1st. Birmingham, UK: Packt Publishing Ltd., May 2017.
- [10] Jonathan Baier. *Getting Started with Kubernetes*. 2nd. Birmingham, UK: Packt Publishing Ltd., maj 2017.
- [11] Russ McKendrick i Scott Gallagher. *Mastering Docker*. 2nd. Birmingham, UK: Packt Publishing Ltd., lip. 2017.
- [12] Gaurav Kumar Aroraa, Lalit Kale i Kanwar Manish. *Building Microservices with .NET Core*. 1st. Birmingham, UK: Packt Publishing Ltd., czer. 2017.

- [13] Brendan Burns. *Designing Distributed Systems*. 1st. Sebastopol, CA, USA: O'Reilly Media, Inc., sty. 2018.
- [14] Nicolai M. Josuttis. *SOA in Practice*. O'Reilly Media, Inc., sierp. 2007.
- [15] Susan J. Fowler. *Production-Ready Microservices*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2017.
- [16] Sam Newman. *Budowanie mikrouslug. Wykorzystaj potencjał architektury uslugi*. Gliwice: Helion S.A., 2015.
- [17] Sanjay Chaudhary, Gaurav Somani i Rajkumar Buyya. *Research Advances in Cloud Computing*. 1st. Singapore, Singapore: Springer Nature Singapore Pte Ltd., 2017.
- [18] Guy Harrison. *Next Generation Databases: NoSQL, NewSQL, and Big Data*. O'Reilly Media, Inc., sty. 2016.
- [19] Jan L. Harrington. *Relational Database Design and Implementation*. 4th. Morgan Kaufmann, kw. 2016.
- [20] Douglas Eadline. *Hadoop 2 Quick-Start Guide: Learn the Essentials of Big Data Computing in the Apache Hadoop 2 Ecosystem*. Addison-Wesley Professional, paź. 2015.
- [21] Vijay Srinivas Agneeswaran. *Big Data Analytics Beyond Hadoop: Real-Time Applications with Storm, Spark, and More Hadoop Alternatives*. PH Professional Business, maj 2014.
- [22] Saurabh Gupta i Shilpi Saxena. *Practical Real-time Data Processing and Analytics*. Birmingham, UK: Packt Publishing Ltd., wrz. 2017.
- [23] Dom Robinson, Ramesh K. Sitaraman i Mukaddim Pathan. *Advanced Content Delivery, Streaming, and Cloud Services*. Wiley-IEEE Press, wrz. 2014.
- [24] Carla Mouradian i in. "A comprehensive survey on fog computing: State-of-the-art and research challenges". W: *IEEE Communications Surveys & Tutorials* 20 (2017), s. 416–464.
- [25] Flavio Bonomi, Bharath Balasubramanian i Mung Chiang. *Fog for 5G and IoT*. Wiley, kw. 2017.
- [26] Perry Lea. *Internet of Things for Architects*. Birmingham, UK: Packt Publishing Ltd., sty. 2018.
- [27] Arvind Ravulavaru. *Enterprise Internet of Things Handbook*. Birmingham, UK: Packt Publishing Ltd., kw. 2018.



- [28] Peter Mell i Timothy Grance. “A comprehensive survey on fog computing: State-of-the-art and research challenges”. W: *National Institute of Standards and Technology NIST Special Publication 800-145* (2011).
- [29] *RightScale 2018 State of the Cloud Report*. Spraw. tech. RightScale, Inc., sty. 2018.
- [30] Theo Lynn i in. “A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms”. W: *2017 IEEE 9th International Conference on Cloud Computing Technology and Science* (2017).
- [31] Scott Hendrickson i in. “Serverless Computation with OpenLambda”. W: *Elastic* 60 (2016), s. 80.
- [32] Zbigniew Fryźlewicz i Daniel Nikończuk. *Windows Azure. Wprowadzenie do programowania w chmurze*. Gliwice: Helion, 2012.
- [33] *BigTable – Wikipedia*. URL: <https://pl.wikipedia.org/wiki/BigTable> (term. wiz. 09.09.2018).
- [34] *Google Cloud SQL: your database in the cloud*. URL: <http://googlecode.blogspot.com/2011/10/google-cloud-sql-your-database-in-cloud.html> (term. wiz. 09.09.2018).
- [35] *Google Container Engine is Generally Available*. URL: <https://cloudplatform.googleblog.com/2015/08/Google-Container-Engine-is-Generally-Available.html> (term. wiz. 11.09.2018).
- [36] *Amazon EKS – Now Generally Available*. URL: <https://aws.amazon.com/blogs/aws/amazon-eks-now-generally-available/> (term. wiz. 09.09.2018).
- [37] *Introducing AKS*. URL: <https://azure.microsoft.com/en-us/blog/introducing-azure-container-service-aks-managed-kubernetes-and-azure-container-registry-geo-replication/> (term. wiz. 09.09.2018).
- [38] *Introducing Windows Azure*. URL: <https://azure.microsoft.com/en-us/blog/introducing-windows-azure/> (term. wiz. 09.09.2018).
- [39] *AWS Elastic Beanstalk – Wikipedia*. URL: [https://en.wikipedia.org/wiki/AWS\\_Elastic\\_Beanstalk](https://en.wikipedia.org/wiki/AWS_Elastic_Beanstalk) (term. wiz. 09.09.2018).

- [40] *Amazon Relational Database Service*. URL: [https://en.wikipedia.org/wiki/Amazon\\_Relational\\_Database\\_Service](https://en.wikipedia.org/wiki/Amazon_Relational_Database_Service) (term. wiz. 09.09.2018).
- [41] *Google Cloud Platform – Wikipedia*. URL: [https://en.wikipedia.org/wiki/Google\\_Cloud\\_Platform](https://en.wikipedia.org/wiki/Google_Cloud_Platform) (term. wiz. 09.09.2018).
- [42] *Amazon SimpleDB – Wikipedia*. URL: [https://en.wikipedia.org/wiki/Amazon\\_SimpleDB](https://en.wikipedia.org/wiki/Amazon_SimpleDB) (term. wiz. 09.09.2018).
- [43] *Cloud Functions serverless platform is generally available*. URL: <https://cloud.google.com/blog/products/gcp/cloud-functions-serverless-platform-is-generally-available> (term. wiz. 12.09.2018).
- [44] *Operating-system-level virtualization – Wikipedia*. URL: [https://en.wikipedia.org/wiki/Operating-system-level\\_virtualization](https://en.wikipedia.org/wiki/Operating-system-level_virtualization) (term. wiz. 10.09.2018).
- [45] *Docker’s Tools of Mass Innovation: Explosive Growth From Open-Source Containers to Commercial Platform for Modernizing and Managing Apps*. URL: <http://www.hostingadvice.com/blog/dockers-explosive-growth-from-open-source-containers-to-commercial-platform/> (term. wiz. 02.02.2018).
- [46] *CloudControl – Wikipedia*. URL: <https://en.wikipedia.org/wiki/CloudControl> (term. wiz. 10.09.2018).
- [47] *Minimal Ubuntu, on public clouds and Docker Hub*. URL: <https://blog.ubuntu.com/2018/07/09/minimal-ubuntu-released> (term. wiz. 10.09.2018).
- [48] *How is Docker different from a virtual machine?* URL: <https://stackoverflow.com/a/16048358/5194338> (term. wiz. 10.09.2018).
- [49] *Borg: The Predecessor to Kubernetes*. URL: <https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/> (term. wiz. 10.09.2018).
- [50] *Setup Kubernetes on a Raspberry Pi Cluster easily the official way*. URL: <https://blog.hypriot.com/post/setup-kubernetes-raspberry-pi-cluster/> (term. wiz. 01.02.2018).
- [51] *Wypróbuj bezpłatnie Cloud Platform*. URL: <https://console.cloud.google.com/freetrial/signup/> (term. wiz. 11.09.2018).

- 
- [52] *Azure Kubernetes Service (AKS) pricing*. URL: <https://azure.microsoft.com/en-us/pricing/details/kubernetes-service/> (term. wiz. 11.09.2018).
- [53] *Utwórz bezpłatne konto platformy Azure już dzisiaj*. URL: <https://azure.microsoft.com/pl-pl/free/> (term. wiz. 11.09.2018).
- [54] *Amazon EC2 Container Service is Now Generally Available*. URL: <https://aws.amazon.com/about-aws/whats-new/2015/04/amazon-ec2-container-service-is-now-generally-available/> (term. wiz. 11.09.2018).
- [55] *Amazon Elastic Container Service*. URL: <https://aws.amazon.com/ecs/> (term. wiz. 11.09.2018).
- [56] *Amazon Elastic Container Service Pricing*. URL: <https://aws.amazon.com/ecs/pricing/> (term. wiz. 11.09.2018).
- [57] *AWS Free Tier*. URL: <https://aws.amazon.com/free/> (term. wiz. 11.09.2018).
- [58] *Amazon EKS Pricing*. URL: <https://aws.amazon.com/eks/pricing/> (term. wiz. 11.09.2018).
- [59] *Serverless Framework – Wikipedia*. URL: [https://en.wikipedia.org/wiki/Serverless\\_Framework](https://en.wikipedia.org/wiki/Serverless_Framework) (term. wiz. 12.09.2018).
- [60] *Wydania do Kubeless*. URL: <https://github.com/kubeless/kubeless/releases?after=v0.2.1> (term. wiz. 12.09.2018).
- [61] *The Object-Oriented Database System Manifesto*. URL: <https://www.cs.cmu.edu/~clamen/OODBMS/Manifesto/> (term. wiz. 13.09.2018).
- [62] *About Akamai: Facts & Figures*. URL: <https://www.akamai.com/us/en/about/facts-figures.jsp> (term. wiz. 16.09.2018).
- [63] *Lambda@Edge now Generally Available*. URL: <https://aws.amazon.com/about-aws/whats-new/2017/07/lambda-at-edge-now-generally-available/> (term. wiz. 16.09.2018).
- [64] *Azure IoT Edge generally available for enterprise-grade, scaled deployments*. URL: <https://azure.microsoft.com/pl-pl/blog/azure-iot-edge-generally-available-for-enterprise-grade-scaled-deployments/> (term. wiz. 16.09.2018).
- [65] *Bringing intelligence to the edge with Cloud IoT*. URL: <https://cloud.google.com/blog/products/gcp/bringing-intelligence-edge-cloud-iot> (term. wiz. 16.09.2018).

- [66] *Azure Container Instances now generally available*. URL: <https://azure.microsoft.com/en-us/blog/azure-container-instances-now-generally-available/> (term. wiz. 25.09.2018).
- [67] *Introducing AWS Fargate*. URL: <https://aws.amazon.com/about-aws/whats-new/2017/11/introducing-aws-fargate-a-technology-to-run-containers-without-managing-infrastructure/> (term. wiz. 25.09.2018).
- [68] *Azure Stack partners*. URL: <https://azure.microsoft.com/en-us/overview/azure-stack/partners/> (term. wiz. 18.09.2018).
- [69] *Cisco and Google Cloud*. URL: <https://cloud.google.com/cisco/> (term. wiz. 18.09.2018).
- [70] *Azure Container Instances and container orchestrators*. URL: <https://docs.microsoft.com/en-us/azure/container-instances/container-instances-orchestrator-relationship> (term. wiz. 25.09.2018).
- [71] *General availability: App Service on Linux and Web App for Containers*. URL: <https://azure.microsoft.com/en-us/updates/general-availability-of-app-service-on-linux-and-web-app-for-containers/> (term. wiz. 25.09.2018).
- [72] *AWS Fargate*. URL: <https://aws.amazon.com/fargate/> (term. wiz. 25.09.2018).
- [73] *Azure Functions—Functions runtime 2.0 now generally available*. URL: <https://azure.microsoft.com/en-us/updates/azure-functions-functions-runtime-2-0-ga/> (term. wiz. 25.09.2018).
- [74] *Developer Survey Results 2018*. URL: <https://insights.stackoverflow.com/survey/2018/> (term. wiz. 27.09.2018).
- [75] *Supported languages in Azure Functions*. URL: <https://docs.microsoft.com/en-us/azure/azure-functions/supported-languages> (term. wiz. 30.09.2018).
- [76] *Code and test Azure Functions locally*. URL: <https://docs.microsoft.com/en-us/azure/azure-functions/functions-develop-local> (term. wiz. 30.09.2018).
- [77] *Azure Functions C# script (.csx) developer reference*. URL: <https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference-csharp> (term. wiz. 30.09.2018).

- [78] *Timer trigger for Azure Functions. Configuration.* URL: <https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-timer/#configuration> (term. wiz. 20.05.2018).
- [79] *Azure Functions triggers and bindings concepts. Supported bindings.* URL: <https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings/#supported-bindings> (term. wiz. 30.09.2018).
- [80] *Develop Azure Functions using Visual Studio.* URL: <https://docs.microsoft.com/en-us/azure/azure-functions/functions-develop-vs> (term. wiz. 30.09.2018).
- [81] *Azure Storage Table Design Guide: Designing Scalable and Performant Tables.* URL: <https://docs.microsoft.com/en-us/azure/cosmos-db/table-storage-design-guide> (term. wiz. 24.05.2018).
- [82] *Azure Table Storage – Design for querying.* URL: <https://docs.microsoft.com/en-us/azure/storage/tables/table-storage-design-for-query> (term. wiz. 30.09.2018).
- [83] *Key management API.* URL: <https://github.com/Azure/azure-functions-host/wiki/Key-management-API> (term. wiz. 04.08.2018).