POLSKO-JAPOŃSKA AKADEMIA
TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Tomasz Kyc
Nr albumu 27927

Porównanie klasycznego wytwarzania oprogramowania z podejściem bezserwerowym

Praca magisterska napisana pod kierunkiem:

dr inż. Mariusz Trzaska

Warszawa, lipiec 2024

Streszczenie

Praca dotyczy porównania wytwarzania oprogramowania w sposób klasyczny z wytwarzaniem oprogramowania za pomocą usług bezserwerowych. Popularność używania usług chmurowych, szybkość prototypowania oraz brak potrzeby dodatkowej administracji serwerami przekładają się na wzrost popularności podejścia bezserwerowego do tworzenia złożonych systemów lub pojedynczych aplikacji. Zagadnienia poruszane w pracy obejmują: techniki wytwarzania oraz dostarczania oprogramowania, usługi chmurowe typu bezserwerowego na przykładzie AWS, aplikacje internetowe, broker wiadomości na przykładzie RabbitMQ, nierelacyjnej bazy danych typu klucz-wartość na przykładzie silnika Redis oraz zagadnienia związane z podejściem infrastruktury jako kodu (*Infrastructure as a Code)*. Motywacją podjęcia tematu pracy jest popularyzacja podejścia bezserwerowego w społeczności programistów i osób technicznych, zwłaszcza w przypadku prototypowania nowych usług. Wykonano dwa prototypy aplikacji w ramach pracy – jeden z wykorzystaniem serwerowych usług oraz drugi – z wykorzystaniem bezserwerowych usług platformy AWS. Prototypy udostępniają funkcjonalność generowania oraz zarządzania krótkimi linkami, które są wykorzystywane w ramach akcji marketingowych czy jako odnośniki w ramach wewnętrznej dokumentacji przedsiębiorstw. Prototypy porównano na podstawie wybranych kryteriów takich jak wydajność, łatwość utrzymania i monitorowania, czas wdrożenia oraz wprowadzania zmian czy koszt.

Słowa kluczowe: serverless, API, AWS, klucz-wartość, noSQL, broker wiadomości, infrastruktura, aplikacja internetowa

Podziękowania

Autor pracy wyraża podziękowanie dr inż. Mariuszowi Trzaska za pomoc oraz za konsultacje dotyczące zagadnień związanych z podejściem klasycznym oraz bezserwerowym.

Spis treści

1. WSTĘP ... 6

1.1. Motywacja ... 6
1.2. Kontekst pracy ... 7
1.3. Cel pracy ... 7
1.4. Rezultat pracy ... 7
1.5. Organizacja pracy ... 7

2. WSTĘP DO TWORZENIA OPROGRAMOWANIA ... 9

2.1. Wprowadzenie do tworzenia oprogramowania ... 9
2.2. Wprowadzenie do konteneryzacji ... 9
2.3. Omówienie usług chmurowych i ich wykorzystania ... 10
2.4. Przedstawienie dodatkowych narzędzi oraz ich wpływu na proces tworzenia i dostarczania oprogramowania ... 11

3. TWORZENIE OPROGRAMOWANIA Z WYKORZYSTANIEM KLASYCZNEGO PODEJŚCIA ... 13

3.1. Wprowadzenie do tworzenia oprogramowania z wykorzystaniem fizycznej lub wirtualnej infrastruktury ... 13
3.2. Wymagania dotyczące kosztów oraz specjalistycznej wiedzy przed rozpoczęciem prac ... 14
3.3. Przegląd narzędzi wspomagających wytwarzanie oprogramowania w sposób klasyczny ... 14

4. TWORZENIE OPROGRAMOWANIA Z WYKORZYSTANIEM USŁUG BEZSERWEROWYCH ... 16

4.1. Wstęp do modelu bezserwerowego ... 16
4.2. Omówienie tworzenia oprogramowania z wykorzystaniem usług bezserwerowych ... 17
4.3. Wymagania dotyczące kosztów oraz wiedzy specjalistycznej przed rozpoczęciem prac ... 18
4.4. Przegląd narzędzi wspomagających wytwarzanie oprogramowania z wykorzystaniem usług bezserwerowych ... 18

5. WYKORZYSTANE TECHNOLOGIE ... 20

5.1. Wykorzystane technologie w wytwarzaniu oprogramowania w sposób klasyczny ... 20
5.2. Wykorzystane technologie w wytwarzaniu oprogramowania z użyciem usług bezserwerowych ... 20
5.3. Języki programowania, frameworki oraz biblioteki użyte w projekcie ... 21
5.4. Pozostałe narzędzia użyte w ramach projektu ... 22

6. PRAKTYCZNY PROJEKT PRZYKŁADOWEJ APLIKACJI INTERNETOWEJ DO SKRACANIA LINKÓW ... 24

6.1. Cel projektu ... 24
6.2. Wymagania funkcjonalne oraz niefunkcjonalne ... 24
6.3. Implementacja aplikacji internetowej z wykorzystaniem klasycznego podejścia ... 27
6.3.1 Architektura implementacji ... 28
6.3.2 Omówienie frameworka Spring ... 28
6.3.3 Omówienie implementacji na przykładzie funkcjonalności użycia skróconego linku ... 30
6.4. Implementacja aplikacji internetowej z wykorzystaniem usług bezserwerowych ... 34
6.4.1 Architektura implementacji ... 34
6.4.2 Omówienie Spring Cloud oraz Spring Cloud Function ... 35
6.4.3 Omówienie implementacji na przykładzie funkcjonalności użycia skróconego linku ... 35

7. ANALIZA PORÓWNAWCZA ... 37

7.1. Podobieństwa i różnice pomiędzy klasycznym podejściem oraz podejściem z wykorzystaniem usług bezserwerowych w tworzeniu oprogramowania ... 37
7.2. Porównanie wydajności pomiędzy klasycznym podejściem oraz podejściem z wykorzystaniem usług bezserwerowych ... 39

7.3. Porównanie utrzymania i monitorowania wdrożonych aplikacji pomiędzy klasycznym podejściem oraz podejściem z wykorzystaniem usług bezserwerowych 42
7.4. Porównanie kosztów pomiędzy klasycznym podejściem oraz podejściem z wykorzystaniem usług bezserwerowych 44
7.5. Porównanie czasu wdrożenia oraz wprowadzania zmian pomiędzy klasycznym podejściem oraz podejściem z wykorzystaniem usług bezserwerowych 46

PODSUMOWANIE 47

PRACE CYTOWANE 48

SPIS RYSUNKÓW 54

SPIS TABEL 55

SPIS LISTINGÓW 56

1. Wstęp

Wytwarzanie oprogramowania opiera się o wiele podejść, technologii oraz metryk. We współczesnym świecie coraz więcej organizacji zauważa w technologiach bezserwerowych szansę na przyspieszenie tworzenia nowych usług, gdzie obecnie czas dostarczenia produktu do użytkowników oraz koszt są najważniejszymi czynnikami. Z drugiej strony wiele przedsiębiorstw nadal używa podejścia klasycznego i korzysta z usług tworzonych, monitorowanych oraz zlokalizowanych w prywatnych centrach danych. Każde podejście ma swoje wady, jak i zalety.

W ramach pracy przedstawiono dwa podejścia w wytwarzaniu usług: klasyczne oraz bezserwerowe. Zbudowano generyczny prototyp aplikacji, który spełnia tą samą funkcjonalność biznesową oraz zapewnia ten sam kontrakt API[1]. W ramach implementacji użyto języka Java oraz chmury publicznej AWS [1]. Zaprezentowano także szereg technologii, dzięki którym budowanie obu typów aplikacji jest bardziej wydajne oraz zautomatyzowane.

Zbudowane aplikacje porównano pod kątem najważniejszych kryteriów takich jak wydajność, łatwość utrzymania, monitoring, koszty operacyjne oraz czas tworzenia oprogramowania oraz jego rozwoju. Praktyczne rezultaty porównania zaprezentowane w tabelach oraz wykresach pozwalają na zrozumienie, które rozwiązanie w przykładowej implementacji sprawdza się lepiej. Ponadto w podsumowaniu pracy zawarte zostały wskazania na przykłady użycia każdego z podejść na podstawie literatury oraz doświadczenia magistranta.

Praca została podzielona na jasno określone rozdziały, co ułatwia poruszanie się po jej treści. W rozdziale drugim omówiono proces tworzenia oprogramowania, usługi chmurowe, konteneryzację oraz dodatkowe narzędzia używane w ramach tworzenia systemów. Rozdział trzeci zawiera opis wytwarzania oprogramowania w podejściu klasycznym, wyzwania stojące przed rozpoczęciem prac w ramach tego podejścia oraz przydatne narzędzia. Usługi bezserwerowe oraz omówienie tego modelu tworzenia oprogramowania zostało opisane w rozdziale czwartym. Znajdziemy w nim również wymienione wymagania związane z użyciem, koszty, a także omówienie pomocnych narzędzi. Rozdział piąty zawiera opis wykorzystywanych technologii, języków programowania, frameworków oraz pozostałych narzędzi użytych w projekcie. W rozdziale szóstym przedstawiono praktyczny projekt aplikacji internetowej, którą zaimplementowano w podejściu klasycznym oraz z wykorzystaniem usług bezserwerowych. Rozdział siódmy zawiera analizę porównawczą dwóch podejść, która pokazuje wady i zalety każdego z nich dla zadanego przypadku aplikacji. Zawiera ona również informacje przydatne dla innych przykładów aplikacji oraz systemów. Ostatni, ósmy rozdział jest przeznaczony na podsumowanie rezultatów.

1.1. Motywacja

Motywacją do podjęcia tematu jest chęć zaprezentowania wad i zalet każdego z podejść omawianych w ramach pracy, a także pokazania przykładów ich użycia. Procesy wytwarzania oprogramowania ciągle się zmieniają, a wraz z nimi wykorzystywane narzędzia. Wzrost zainteresowania usługami bezserwerowymi skłania do refleksji nad możliwościami ich zastosowania w projektach informatycznych. Warte sprawdzenia jest potencjalnie szybsze i łatwiejsze budowanie aplikacji, o którym zapewniają dostawcy bezserwerowych usług.

Doświadczenie oraz zainteresowanie wytwarzaniem oprogramowania bezpośrednio wpłynęły na decyzję o wyborze tematu. Praca daje szansę na lepsze zrozumienie różnic oraz zastosowań każdego z opisywanych podejść.

[1] Kontrakt API – specyfikacja, na podstawie której usługa udostępniająca API, jak i klienci API opierają swoje implementacje. Usprawnia to proces integracji usług, rozwoju API oraz testowania. Więcej informacji [79]

1.2. Kontekst pracy

Wytwarzanie oprogramowania jest procesem, dzięki któremu ludzie mogą korzystać z tak wielu udogodnień jak choćby dostawa zakupów do domu, video-rozmowa z osobą z innego kraju czy możliwość posłuchania ulubionego podcastu. Organizacje tworzące aplikacje i systemy zwykle wykorzystują własną infrastrukturę zlokalizowaną w centrach danych do oferowania swoich usług. W opozycji do tego stoją dostawcy usług typu bezserwerowego oferujący gotowe usługi bez konieczności administracji nimi lub zatrudniania wykwalifikowanych specjalistów.

1.3. Cel pracy

Praca ta ma na celu porównanie dwóch podejść w zakresie tworzenie oprogramowania oraz zaprezentowanie i omówienie rezultatów, które mogą być pomocne w ramach prawdziwych przedsięwzięć informatycznych.

W ramach pracy omówiono szereg technologii wykorzystywanych przy obu podejściach, wymagania kosztowe oraz te z zakresu wiedzy specjalistycznej. Zwrócono również uwagę na potencjalne problemy podczas implementacji oraz na wcześniejszych etapach – przy tworzeniu zespołu projektowego i doborze kompetencji. Wymienione elementy pozwalają w sposób holistyczny podejść do wyboru konkretnego rozwiązania.

Przykład praktycznego projektu aplikacji internetowej, zaimplementowanej w każdym z podejść, pokazuje mocne i słabe strony każdej z metod. Ujawnia również, że wybór rozwiązania będzie zależny od typu aplikacji i sposobu jej działania.

1.4. Rezultat pracy

Wynikiem realizacji niniejszej pracy jest dokument, który zawiera porównanie wytwarzania oprogramowania w dwóch podejściach: klasycznym oraz z wykorzystaniem usług bezserwerowych. Dodatkowo zawiera on przykłady narzędzi, które automatyzują i usprawniają pracę w ramach każdej z metod, a także technologie, które są wspólne.

Praktyczny przykład projektu aplikacji internetowej pokazuje, w jaki sposób wygląda tworzenie oprogramowania oraz praca w ramach każdego z podejść. Podkreślone są sposoby testowania oprogramowania, koszty początkowe oraz operacyjne z uwzględnieniem różnic wynikających z każdej z metod. Sama aplikacja została stworzona w oparciu o podejście, które zapewnia elastyczność oraz generyczność, dzięki czemu zmieniane mogą być zewnętrzne zależności, a logika aplikacji pozostaje nienaruszona.

1.5. Organizacja pracy

Praca jest podzielona na kilka rozdziałów, które mają za zadanie przeprowadzić Czytelnika przez porównanie podejścia klasycznego oraz podejścia wykorzystującego usługi bezserwerowe.

Rozdział pierwszy zawiera wprowadzenie do pracy, motywacje podjęcia tematu, cel oraz rezultat dokumentu.

Rozdział drugi zawiera opis wytwarzania oprogramowania, wyzwań związanych z konteneryzacją, a także omówienie usług chmurowych oraz narzędzi używanych w ramach procesu.

Rozdział trzeci przedstawia podejście klasyczne, wymagania dotyczące kosztów oraz wiedzy specjalistycznej, a także wykorzystywane narzędzia.

W rozdziale czwartym omawiane jest podejście z wykorzystaniem usług bezserwerowych, wymagania dotyczące kosztów oraz specjalistów. Dodatkowo prezentowane są pomocne narzędzia ułatwiające tworzenie oprogramowania w tej metodzie.

Użyte technologie w ramach implementacji aplikacji internetowej są opisane w rozdziale piątym. Wymienione są języki oprogramowania, frameworki, a także inne narzędzia użyte w ramach projektu.

W ramach rozdziału szóstego opisane są wymagania funkcjonalne oraz niefunkcjonalne, a także konkretne biblioteki i rozwiązania użyte w projekcie aplikacji.

Końcowy rozdział siódmy zawiera analizę porównawczą obu podejść pod kątem wydajności, łatwości utrzymania i monitorowania aplikacji, kosztów, a także czasu wdrożenia oraz wprowadzania zmian. Znajdują się tam również wnioski oraz sugestie dotyczące możliwych dalszych kierunków badań.

2. Wstęp do tworzenia oprogramowania

Pierwszy podrozdział wprowadza Czytelnika w świat tworzenia oprogramowania, opisując najważniejsze elementy oraz kategorie dotyczące jego wytwarzania. W następnym podrozdziale przedstawione zostało zagadnienie związane z konteneryzacją oraz jej wpływem na proces wytwórczy. Podrozdział trzeci zawiera opis usług chmurowych, ich kategoryzacji oraz możliwych zastosowań. Na końcu tego rozdziału przedstawiono pozostałe narzędzia, które mają niezwykle istotny wpływ na czas, koszty oraz jakość wytwarzanego oprogramowania.

2.1. Wprowadzenie do tworzenia oprogramowania

We współczesnym świecie oprogramowanie odgrywa ważniejszą rolę niż kiedykolwiek. Ludzie mają z nim styczność na każdym kroku, często nie zdając sobie z tego sprawy. Poczynając od wejścia do sklepu, gdzie kamery sterowane są za pomocą oprogramowania, przez prowadzenie samochodów wyposażonych w komputery pokładowe, kończąc na grach mobilnych online, które są niezwykle skomplikowanym oprogramowaniem dostarczającym rozrywkę.

W ramach szerokiego zakresu typów oprogramowania możemy wyróżnić najpopularniejsze:

- **Oprogramowanie webowe (ang. *web software*)** – oprogramowanie dostarczane za pośrednictwem przeglądarki internetowej, niewymagające instalowania na urządzeniu klienta żadnych dodatkowych programów. Przykładem takiego oprogramowania może być aplikacja do komunikacji Messenger [2].
- **Oprogramowanie mobilne (ang. mobile *software*)** – oprogramowanie tworzone z myślą o smartfonach oraz tabletach.
- **Oprogramowanie wbudowane (ang. e*mbedded software*)** – to wszelkiego rodzaju oprogramowanie działające bezpośrednio na urządzeniach takich jak mikrokontrolery w terminalach płatniczych lub lodówkach.
- **Oprogramowanie dla komputerów biurkowych (ang. *desktop software*)** – oprogramowanie działające w postaci osobnych okien, spopularyzowane przez system operacyjny Windows [3]. Przykładem takiego programu jest Paint [4].
- **Oprogramowanie wykorzystujące wiersz linii poleceń (ang. *command line interface software*)** – oprogramowanie, które wykorzystuje wiersz linii poleceń do komunikacji z aplikacją. Przykładem takiego programu jest znane osobom biorącym udział w wytwarzaniu oprogramowania narzędzie Git [5].

Proces powstawania programów, od momentu definiowania wymagań, przez pisanie kodu aplikacji, jej udostępnianie, poprawianie oraz utrzymanie nosi nazwę procesu tworzenia oprogramowania.

2.2. Wprowadzenie do konteneryzacji

Wytwarzane oprogramowanie składa się z wielu elementów. Czasami oprogramowanie korzysta z bibliotek preinstalowanych w ramach systemu operacyjnego, na którym jest uruchamiane. Problemem może być sytuacja, gdy kilka aplikacji korzysta z tej samej biblioteki, ale do poprawnego działania wymaga różnych jej wersji.

Aby sprostać temu wyzwaniu oraz tworzyć odizolowane środowiska, wymyślono koncepcję kontenerów (ang. *containers*). Sam Newman w swojej książce „Budowanie mikrousług. Projektowanie

drobnoziarnistych systemów” [6] opisał je następująco: *„...możesz myśleć o kontenerze jako abstrakcji nad poddrzewem całego drzewa procesów systemowych, gdzie jądro wykonuje większość „ciężkiej pracy”.* Kontenery wymagają mniejszych zasobów sprzętowych, dzięki czemu na jednej maszynie można ich uruchomić znacznie więcej niż ich odpowiedników w postaci wirtualnych maszyn.

W pracy „Implementing a Secured Container Workload in the Cloud” [7] M. Raheem zwraca uwagę na problemy z bezpieczeństwem dotyczące możliwości uruchamiania niebezpiecznych komend w kontenerach. Wszelkie błędy są jednak na bieżąco rozwiązywane poprzez dodatkowe narzędzia firm trzecich takie jak skanery repozytoriów kontenerów oraz dostosowywanie technologii kontenerów do coraz wyższych wymagań związanych z bezpieczeństwem.

Warto wspomnieć o systemie plików w ramach kontenerów, który może być wykorzystany na wiele sposób. Przykładowy kontener bazujący na systemie Linux Ubuntu [8] będzie zawierał odpowiadające systemowi pliki oraz katalogi. Istnieje możliwość mapowania katalogu z hosta do konkretnego kontenera, dzięki czemu można współdzielić zmienne pliki takie jak konfiguracja aplikacji. Umożliwia to budowanie tzw. *wirtualnego systemu plików*, gdzie część katalogów może być katalogami istniejącymi wewnątrz kontenera, a inne mogą wskazywać na lokalizacje w ramach hosta kontenerów. Ponadto warto zwrócić uwagę na fakt, że pliki w ramach kontenera mają określony czas życia od momentu utworzenia kontenera do jego usunięcia. Może to być przydatne dla celów testów oprogramowania lub prób budowania skryptów ułatwiających pracę.

Dodatkowo warto zaznaczyć, że coraz więcej firm stosuje jako standard dostarczanie swoich produktów w ramach kontenerów, które są gotowymi komponentami i mogą być zainstalowane na infrastrukturze klienta. Poprawia to przenośność oprogramowania oraz łatwość jego dostarczania jako jeden artefakt ze wszystkimi potrzebnymi bibliotekami oraz środowiskiem uruchomieniowym.

2.3. Omówienie usług chmurowych i ich wykorzystania

Usługi chmurowe stanowią bardzo ważny element w tworzeniu oraz dostarczaniu oprogramowania. Bez nich tworzenie produktów, które mają być dostępne dla szerokiego grona odbiorców, skalowalne oraz bezpieczne, byłoby niezwykle trudne.

Podając za [9], chmurę możemy zaklasyfikować jako:

- **Prywatną (ang. *private cloud*)** – wariant chmury obliczeniowej, gdzie infrastruktura organizacji nie jest udostępniana publicznie. Przykładem takich chmur mogą być chmury prywatne banków, które podlegają specyficznym regulacjom prawnym i dane przetwarzane przez nie mogą być przetwarzane w obrębie innych typów chmur.
- **Publiczną (ang. *public cloud*)** – w przeciwieństwie do chmury prywatnej, ten typ chmury jest publicznie dostępny dla użytkowników indywidualnych oraz dla organizacji, które nie mają swoich centrów danych i potrzebują mocy obliczeniowej. Dodatkowo dostawcy chmury publicznej oferują unikalne usługi jak własne typy baz danych lub gotowe usługi do przetwarzania obrazów. Trzema największymi przedstawicielami chmury publicznej są Amazon Web Services (AWS) [1], Google Cloud Platform (GCP) [10] oraz Microsoft Azure [11].
- **Hybrydowa (ang. *hybrid cloud*)** – połączenie chmur prywatnych oraz publicznych, gdzie część infrastruktury oraz aplikacji działa w prywatnym centrum danych firmy, a pozostała część w ramach chmury publicznej. Przykładami wykorzystania takiego podejścia są organizacje, które chcą korzystać z dobrodziejstw chmury publicznej, natomiast są zobowiązane do przechowywania podzbioru danych na swojej infrastrukturze z powodu regulacji prawnych.

Ważnym aspektem usług chmurowych jest możliwość wykorzystywania zasobów w ramach modelu PAYG (ang. *Pay As You Go*) polegającego na płatności tylko za te zasoby, które zostały zużyte. Odpowiednio dobrana architektura oprogramowania sprawia, że nie potrzeba posiadać ciągle

uruchomionego serwera wywołującego funkcję w danym interwale czasowym. Zamiast tego można użyć bezstanowych funkcji, które są uruchamiane w określonym harmonogramie i po ich zakończeniu automatycznie zatrzymywane. W takim przypadku opłata jest naliczana jedynie za czas działania funkcji, a nie za stale działający serwer.

Interesującą możliwością dla startupów są programy finansowe poszczególnych dostawców chmury publicznej, takie jak AWS Startups [12], Google for Startups Cloud Platform [13] oraz Microsoft for Startups [14]. Firmy oferują środki do wykorzystania w ramach platform, a także wsparcie techniczne. Dzięki temu startupy mogą szybko zweryfikować swój pomysł, wykorzystując gotowe komponenty bez ponoszenia dodatkowych kosztów i inwestowania we własną infrastrukturę. Zwykle startupy, jak i większe firmy, nie zmieniają wybranego na początku dostawcy chmury. Głównymi powodami są poniesione koszty lub system zorientowany na konkretne rozwiązania oferowane jedynie przez wybranego dostawcę.

2.4. Przedstawienie dodatkowych narzędzi oraz ich wpływu na proces tworzenia i dostarczania oprogramowania

Współczesny proces wytwarzania oprogramowania to nie tylko pisanie kodu aplikacji. To także używanie dodatkowych narzędzi, które ułatwiają jej wdrażanie, monitorowanie oraz poprawiają jakość i niezawodność. Z powodu coraz większego nacisku na szybkie dostarczanie zmian oraz poprawę błędów, narzędzia te stały się standardem w procesie wytwórczym.

Jednym z podstawowych narzędzi jest orkiestrator kontenerów, którego zadaniem jest wdrażanie, monitorowanie oraz skalowanie aplikacji uruchamianych w ramach kontenerów. Jak napisali w swojej książce Brendan Burns, Joe Beda oraz Kelsey Hightower „Kubernetes. Up & running. 2nd edition" [15]" "*Wiele osób przeniosło się na kontenery i orkiestratory takie jak Kubernetes z powodu dużych usprawnień związanych z niezawodnością wdrożeń, jakie zapewniają.*". Dodatkowo narzędzia te mają możliwość automatycznego uzdrawiania tj. za pomocą zdefiniowanych reguł mogą sprawdzać stan działania aplikacji. W momencie, gdy aplikacja przestaje poprawnie działać lub jej proces kończy się błędem, orkiestrator sam restartuje dany kontener z aplikacją. Przykładem takiego narzędzia jest Kubernetes [16], które dawniej było używane wewnątrz firmy Google. W pewnym momencie gigant technologiczny postanowił udostępnić je jako oprogramowanie typu open-source[2].

Kolejny typ to narzędzia do ciągłej integracji (ang. *continuous integration*), których zadaniem jest zapewnienie, aby wprowadzane nowe zmiany w kodzie były poprawnie zintegrowane z istniejącą zawartością. Narzędzia te wykrywają zmiany w ramach repozytorium kodu, a następnie przeprowadzają szereg weryfikacji od sprawdzenia kompilacji kodu, uruchomienie testów aplikacji po dowolną akcję zdefiniowaną przez użytkownika. Przykładem takiego narzędzia jest GitHub Actions [17].

Narzędzia do ciągłego dostarczania (ang. *continuous delivery*), takie jak ArgoCD [18], umożliwiają modelowanie procesu od momentu dodania kodu aż do wdrożenia na produkcję. Pomaga to w ocenie, czy nasza aplikacja i jej jakość jest gotowa do wdrożenia na system produkcyjny poprzez wykonanie wielu typów testów jak testy jednostkowe, integracyjne czy wydajnościowe.

W przypadku tworzenia infrastruktury aplikacji bardzo popularna stała się metoda infrastruktura jako kod (ang. *Infrastructure as a Code*). Umożliwia ona zamodelowanie baz danych, serwerów, zapór ogniowych (ang. *firewalls*) czy równoważników obciążenia (ang. *load balancers*) w postaci kodu. Pomaga to w audytowaniu zmian w infrastrukturze, szybkim powrocie do poprzednich zmian oraz posiadaniu jednego miejsca jako źródła prawdy. Przykładem takiego narzędzia jest Terraform [19],

[2] Inaczej otwarte oprogramowanie to typ, dla którego kod źródłowy jest udostępniany bezpłatnie oraz może być modyfikowany. Więcej informacji w [97]

który coraz częściej jest wykorzystywany w projektach informatycznych zorientowanych na chmurę publiczną.

Pomocne w pracy nad jakością kodu oraz wczesnym wychwytywaniem błędów są statyczne analizatory kodu. Służą one do lokalizowania potencjalnych błędów w kodzie, luk w zakresie bezpieczeństwa oraz zmniejszenia długu technologicznego. Ich rozbudowane funkcje pozwalają na zintegrowanie ze środowiskiem deweloperskim tak, że w momencie pisania kodu programista jest informowany o potencjalnym błędzie. Przykładem takiego narzędzia jest SonarQube [20].

Kod źródłowy wytwarzanego oprogramowania powinien być przechowywany w ramach repozytorium kodu – miejsca, które zapewnia jego wersjonowanie, przeglądanie oraz dyskusję nad nim. Platformy takie jak GitHub [21] czy GitLab [22] stały się pierwszym wyborem dla zespołów. Pomagają one w komunikacji, dyskusji nad dodawaniem nowych zmian w kodzie czy udostępnianiem fragmentów kodu (ang. *snippets*). Dodatkowo ich zaawansowane funkcje bezpieczeństwa pozwalają na definiowanie reguł dotyczących dostępu do kodu czy możliwości jego modyfikacji.

Aplikacje oraz systemy nie są kompletne bez monitoringu oraz alertowania. Jak podaje autor książki „*SRE with Java Microservices.*" Jonathan Schneider [23]: „*...skomplikowanie systemów rozproszonych złożonych z wielu mikroserwisów oznacza, że niezwykle ważne jest to, aby móc obserwować stan systemu.*". Monitoring systemu umożliwia obserwowanie jego wskaźników oraz tworzenie alertów. Dzięki niemu administrator jest informowany o awarii wcześniej, niż napiszą do niego użytkownicy. Ponadto dane zbierane przez systemy monitorujące takie jak Grafana [24] czy Prometheus [25] pomagają w analizach po awarii systemu.

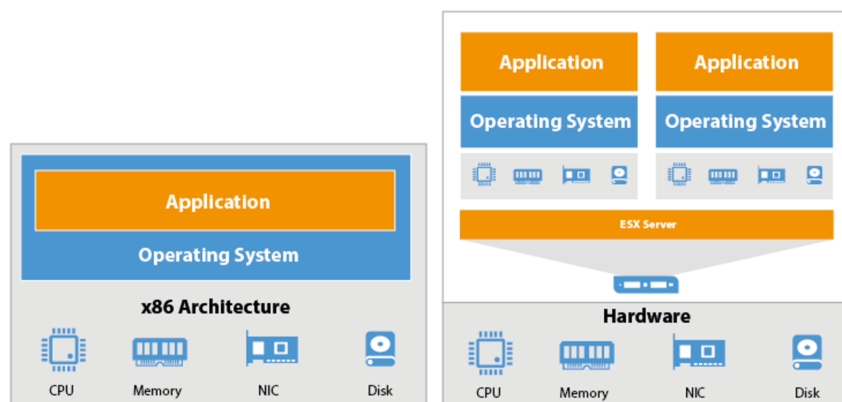
3. Tworzenie oprogramowania z wykorzystaniem klasycznego podejścia

Rozdział ten zawiera opis wytwarzania oprogramowania z wykorzystaniem klasycznych narzędzi. Na początku przedstawione są podstawowe elementy dotyczące infrastruktury. Dalej Czytelnik jest zapoznawany z kosztami oraz wymaganą wiedzą. Na końcu opisano narzędzia przydatne w trakcie tworzenia oprogramowania w ramach tego nurtu.

3.1. Wprowadzenie do tworzenia oprogramowania z wykorzystaniem fizycznej lub wirtualnej infrastruktury

W ramach procesu wytwarzania oprogramowania niezwykle ważnym elementem jest infrastruktura. Liczba rdzeni procesora, pamięci RAM czy bardzo ważne w przypadku baz danych rodzaj i prędkość odczytu/zapisu dysków to tylko niektóre aspekty infrastruktury. Istnieje wiele innych elementów takich jak typy systemów operacyjnych, z których będziemy korzystać, związane z nimi licencje czy infrastruktura sieciowa. Kolejnym aspektem jest bezpieczeństwo fizyczne oraz wirtualne serwerowni. Chcąc udostępnić usługę dla klientów należy odpowiednio skonfigurować warstwę sieciową.

W celu uniknięcia wysokich kosztów i zakupu wielu fizycznych serwerów, często spotykanym podejściem jest zakup jednego serwera i uruchamianie na nim wielu wirtualnych maszyn. Maszyny te są odseparowane na poziomie systemowym, natomiast nadal korzystają z przydzielonych zasobów sprzętowych fizycznego serwera. Dzięki takiemu podejściu dużo łatwiej jest zarządzać infrastrukturą oraz ją monitorować. Przykład porównania komponentów pomiędzy infrastrukturą fizyczną, a wirtualną pokazany jest na Rysunku 1.



Rysunek 1 Porównanie infrastruktury fizycznej i wirtualnej. Źródło: [26]

Kolejnym niezwykle ważnym aspektem są aktualizacje systemów operacyjnych i instalacja wymaganych wersji bibliotek niezbędnych do poprawnego funkcjonowania aplikacji. Przez brakujące komponenty aplikacja może się nie uruchomić. Czasami o brakujących bibliotekach można się dowiedzieć dopiero w momencie, gdy użytkownik spróbuje uruchomić zależną od nich funkcjonalność.

Testowanie jest również niezwykle istotne w procesie wytwarzania oprogramowania. W ramach korzystania z podejścia klasycznego możemy uruchomić daną usługę, taką jak baza danych lub broker wiadomości, na jednym ze środowisk testowych. Koszt utworzenia takiej dodatkowej instancji

jest niewielki. Zwykle zużywa mniejsze zasoby w porównaniu do konfiguracji produkcyjnej oraz umożliwia używanie danej instancji serwera bazy danych przez wiele zespołów.

Po przejściu przez powyższe aspekty i uzyskaniu działającej aplikacji, wymagane jest wydzielanie okien serwisowych. W ramach nich wgrywane są krytyczne aktualizacje czy przeprowadzane prace modernizacyjne (wymiana dysków lub rozszerzenie pamięci RAM w serwerze).

3.2. Wymagania dotyczące kosztów oraz specjalistycznej wiedzy przed rozpoczęciem prac

Opisane w poprzednim podrozdziale podejście klasyczne wymaga nakładów finansowych jeszcze przed rozpoczęciem tworzenia oprogramowania. Przykłady kosztów, które prezentuje literatura [27] to:

- **Zakup fizycznej infrastruktury** – składa się na to zakup serwerów, szaf serwerowych, urządzeń sieciowych oraz zapewnienie odpowiedniego łącza internetowego.
- **Wynajęcie lub zakup miejsca na potrzeby serwerowni** – dostęp do tego miejsca powinien być kontrolowany, a pomieszczenie wyposażone w szereg urządzeń dbających m.in. o odpowiednią temperaturę.
- **Zakup wymaganych licencji** – niezbędnych do działania niektórych aplikacji oraz umożliwiających uzyskanie wsparcia producentów w razie awarii.
- **Zatrudnienie specjalistów od infrastruktury** – potrzebnych nie tylko do opiekowania się i monitoringu infrastruktury fizycznej, ale także specjalistycznych rozwiązań, takich jak brokery wiadomości czy nierelacyjne bazy danych. Skompletowanie zespołu specjalistów posiadających niezbędne kompetencje może być czasochłonne i drogie.
- **Zakup i konfiguracja monitoringu** – niezbędnego dla działania zarówno infrastruktury jak i aplikacji, który informuje pracowników o awariach oraz umożliwia ich szybkie usuwanie.
- **Przeprowadzanie testów bezpieczeństwa** – infrastruktura fizyczna jest bardziej podatna na ataki, nie tylko z powodu fizycznego dostępu do serwerów, ale także braków w konfiguracji czy niezaktualizowania jednostek na czas. Stąd zachodzi potrzeba przeprowadzania okresowo testów bezpieczeństwa, aby eliminować potencjalne wektory ataku.

3.3. Przegląd narzędzi wspomagających wytwarzanie oprogramowania w sposób klasyczny

Organizacje decydujące się na wytwarzanie oprogramowania w sposób klasyczny potrzebują zestawu narzędzi, który ułatwia pracę oraz pozwala zarządzać stworzoną infrastrukturą. Poniżej stworzono listę narzędzi na podstawie [28, 29, 30], a także doświadczeń zawodowych magistranta:

- **Terraform** [19] – narzędzie do zarządzania stanem infrastruktury, implementujące podejście „*infrastruktura jako kod*". Głównymi zaletami narzędzia jest możliwość modelowania infrastruktury za pomocą autorskiego języka domeny, a także ilość modułów umożliwiających tworzenie różnych typów baz danych czy połączeń do chmur publicznych.
- **Ansible** [31] – narzędzie służące automatyzacji zadań na wielu serwerach. Umożliwia ono deklarowanie pożądanego stanu infrastruktury za pomocą plików konfiguracyjnych, a następnie, poprzez wewnętrzne wywołania poleceń, zapewnia pożądany stan na wskazanych serwerach. W środowisku inżynierów systemowych jest uważane za jedno z ważniejszych

narzędzi z powodu oszczędności czasu. Dzięki używaniu go razem z np. Terraform [19] braki jednego narzędzia są uzupełniane przez drugie.

- **OpenShift** [32] – platforma umożliwiająca stworzenie klastra Kubernetes [16], wykorzystując własną infrastrukturę. Pomaga ona w monitoringu, zarządzaniu siecią oraz dbaniu o bezpieczeństwo aplikacji uruchamianych w klastrze. Minimalizuje ilość pracy potrzebnej do prawidłowego działania aplikacji. Narzędzie to jest często wybierane, gdy firma myśli o budowie chmurny prywatnej (ang. *private cloud*).
- **Keycloak** [30] – platforma open-source do zarządzania tożsamością oraz dostępem użytkowników. Bardzo często używana w projektach informatycznych do przechowywania danych użytkowników oraz zarządzająca ich loginami, hasłami, procesami autoryzacji. Sprawdza się w przypadkach, gdzie organizacja nie ma systemu do zarządzania tożsamością użytkowników. Ponadto umożliwia połączenie do innych platform tożsamościowych takich jak Facebook [33] czy Google [34].
- **Prometheus** [25] – baza danych open-source, która za pomocą dodatkowych procesów tzw. agentów zbiera i przechowuje metryki aplikacji. Umożliwia ona tworzenie złożonych zapytań za pomocą domenowego języka PromQL [35]. Zwykle jest wykorzystywana w tandemie z Grafaną [24].
- **Grafana** [24] – narzędzie służące do wizualizacji metryk. Umożliwia ona pokazywanie danych w czasie rzeczywistym, dzięki czemu administratorzy mogą obserwować działania nawet bardzo skomplikowanych systemów. Ponadto umożliwia definiowanie alertów, które mogą być wysyłane m.in. za pomocą maili lub SMS.
- **GitLab** [22] – repozytorium kodu typu open-source, które pomaga w wersjonowaniu kodu oraz pracy nad nim w ramach dyskusji czy definiowania potoków CI/CD. Dzięki przejrzystemu graficznemu interfejsowi oraz zaawansowanemu modelowi uprawnień sprawdza się zarówno w prostych projektach jak i skomplikowanych, złożonych z wielu organizacji przedsięwzięciach informatycznych.

4. Tworzenie oprogramowania z wykorzystaniem usług bezserwerowych

Rozdział ten zawiera omówienie modelu bezserwerowego oraz przedstawienie sposobu jego użycia w ramach wytwarzania oprogramowania. Znaleźć można listę kosztów oraz zakres wiedzy specjalistycznej wymaganych w ramach prac w tym modelu. Dalej podany jest zbiór narzędzi, które ułatwiają prowadzenie prac w tym podejściu. Część z nich jest opisana na przykładzie chmury publicznej AWS [1], jednak ich odpowiedniki mogą być łatwo znalezione u innych dostawców.

4.1. Wstęp do modelu bezserwerowego

Zanim zostaną przedstawione sposoby tworzenia oprogramowania w ramach tego podejścia, należy zrozumieć co właściwie ono oznacza.

Ciekawą definicję podejścia bezserwerowego (ang. *serverless*) można przytoczyć z pracy Erica Jonasa, Johann Schleier-Smith i innych „*Cloud Programming Simplified: A Berkeley View on Serverless Computing*” [36]:„*Podczas gdy funkcje chmurowe - pakowane jako oferty FaaS (Function as a Service) - stanowią rdzeń obliczeń bezserwerowych, platformy chmurowe zapewniają również wyspecjalizowane frameworki bezserwerowe, które spełniają określone wymagania aplikacji jako oferty BaaS (Backend as a Service). Mówiąc prościej, przetwarzanie bezserwerowe = FaaS + BaaS.*”. Cytowane rozwiązania typu BaaS odnoszą się zwykle do ofertowania zarządzanych baz danych jak AWS DynamoDB [37] czy rozwiązań dotyczących uwierzytelniania jak AWS Cognito [38]. Dostawcy usług chmurowych, widząc trend na podejście bezserwerowe, tworzą nowe i lepiej zarządzane serwisy.

Na podstawie literatury [39] możemy wyróżnić główne cechy aplikacji i podejścia bezserwerowego:

- **Zmniejszona odpowiedzialność za działania operacyjne** – korzystając z rozwiązań bezserwerowych jesteśmy odpowiedzialni jedynie za dostarczenie kodu, który będzie wykonywał określone działanie. Wszelkie operacje związane z bezpieczeństwem, aktualizacją serwerów oraz ich utrzymywaniem są po stronie dostawcy rozwiązań.
- **Skalowalność** – dostawca rozwiązania zapewnia skalowalność rozwiązania, czasami automatycznie na podstawie zapotrzebowania aplikacji np. wskaźnika ilości żądań na sekundę (ang. *requests per second, RPS*) lub innych metryk. Sprawia to, że nie musimy się martwić co się stanie wraz ze wzrostem popularności usługi.
- **Model kosztowy** – w ramach usług bezserwerowych często używanym modelem jest model P*ay As You Go (ang. PAYG*), który definiuje płatność jedynie za zużyte zasoby w ramach aplikacji. Nie trzeba już na samym początku kupować fizycznych serwerów, licencji oraz ponosić dodatkowych kosztów z powodu wynajmu powierzchni na serwerownię. Zamiast tego można wykupić jedynie tyle mocy obliczeniowej, ile jest potrzeba, a następnie w miarę rozwoju usługi odpowiednio ją skalować.
- **Efektywność programisty** – dzięki możliwości samodzielnego tworzenia przez programistę popularnych usług jak bazy danych czy brokery wiadomości, osoby tworzące kod mogą szybciej skupić się na dostarczaniu głównych funkcjonalności produktu, który budują.
- **Łatwość przeprowadzania eksperymentów** – dzięki łatwości tworzenia infrastruktury na żądanie oraz mniejszemu czasowi poświęcanemu na powtarzalne czynności, dużo łatwiej i szybciej można przeprowadzać eksperymenty pomagające w udoskonaleniu produktu.
- **Bezpieczeństwo i stabilność** – doświadczenia dostawców chmurowych w zakresie niezawodności i bezpieczeństwa są aplikowane na produkty, które oferują. W ramach używania usług typu

bezserwerowego wymuszane są sprawdzone mechanizmy bezpieczeństwa oraz niezawodności pozwalające budować lepsze i bardziej odporne na ataki i awarie aplikacje.

- **Mniej kodu** – dzięki wykorzystaniu gotowych usług, programiści mają mniej kodu do utrzymania i nie muszą wymyślać rozwiązań, które zostały już dopracowane przez twórców usług bezserwerowych. Dodatkowo zmniejsza to poziom złożoności aplikacji. Osobom dołączającym do projektu łatwiej jest zrozumieć ogólnie używane wzorce lub usługi dostawców chmurowych, zamiast własnych, często błędnie zaimplementowanych, rozwiązań.

4.2. Omówienie tworzenia oprogramowania z wykorzystaniem usług bezserwerowych

Wytwarzanie oprogramowania z wykorzystaniem usług bezserwerowych różni się od podejścia standardowego.

W omawianym podejściu na samym początku prac nie trzeba się martwić o zakup fizycznych serwerów, ich umiejscowienie oraz konfigurację urządzeń sieciowych. Zamiast tego – musimy wybrać jednego z dostawców chmury publicznej i założyć konto na jego platformie. Zwykle jest to dostawca, z którego usług już korzystamy. Dość częstym wyborem jest Microsoft Azure [11], jeśli firma korzysta z rozwiązań takich jak Microsoft Office365 [40] lub systemów operacyjnych z rodziny Windows [3]. W przeciwnym razie – zwykle wybiera się dostawcę na podstawie doświadczeń posiadanych w zespole specjalistów oraz porównania kosztów usług. Niezwykle ciekawą opcją są programy dla startupów, oferowane przez dostawców chmury. Zapewniają one pulę pieniędzy do wykorzystania w ramach platformy. Dla startupów jest to dodatkowa pomoc, która sprawia, że zwykle zostają z daną platformą na stałe.

Po dokonaniu wyboru dostawcy, zespoły stają przed decyzją, w jaki sposób chcą uruchamiać swoją aplikację. Usługi bezserwerowe oferują bardzo duży wybór zależnie od specyfiki aplikacji. W przypadku większości aplikacji dobrze sprawdzają się usługi pozwalające wdrożyć aplikację w postaci kontenera i minimalnie ją skonfigurować. Następnie platforma będzie odpowiadała za skalowanie i monitoring. Innym rozwiązaniem wartym uwagi są klastry Kubernetes [16] w pełni zarządzane oraz optymalizowane przez dostawcę chmurowego. Dobrze sprawdzają się przy aplikacjach, które wymagają większej kontroli oraz możliwości parametryzacji. Ostatnim opisanym typem są aplikacje, które mają być uruchamiane na żądanie. Wówczas dobrym wyborem są bezserwerowe serwisy typu FaaS (ang. *Function as a Service*), gdzie płacimy tylko za czas działania programu.

Bazy danych są nieodłącznym elementem każdej aplikacji, która potrzebuje przechowywać informacje. Dostawcy chmurowi, oprócz wsparcia dla baz open-source takich jak PostgreSQL [41] czy Redis [42], oferują własne typy baz danych. Przykładem jednej z nich jest AWS DynamoDB [37], która jest bazą typu klucz-wartość oraz umożliwia bardzo dobre skalowanie, bez względu na ruch oraz ilość przechowywanych danych.

Kolejnym ważnym aspektem jest testowanie. Dla usług bezserwerowych wiąże się ono z wyzwaniem, że kopii tych usług nie da się uruchomić lokalnie na komputerze programisty albo serwerze o dużo większych zasobach. Wówczas rozwiązania są dwa:

1. Korzystanie z atrap usług, które naśladują zachowanie prawdziwych usług oraz implementują ich interfejsy API. Warto zastrzec, że atrapy usług powinny być dostarczane przez dostawcę usług bezserwerowych, dzięki czemu będą one na bieżąco aktualizowane wraz ze zmianą API prawdziwej usługi. W przeciwnym przypadku powstają projekty open-source utrzymywane przez społeczność, lecz zwykle nie nadążają one za zmianami w API i wsparcie dla nich jest po pewnym czasie porzucane. Przykładem takiej atrapy jest narzędzie localstack [43] dla AWS [1], które imituje usługi bezpośrednio na komputerze programisty.

2. Korzystanie z prawdziwych instancji usług z minimalnymi zasobami na środowisku testowym. Dzięki takiemu podejściu dowiadujemy się automatycznie o zmianach w API usługi i mamy większą pewność, że testy w pełni oddają zachowanie aplikacji. Wadą takiego podejścia może być zwiększony rachunek za usługi chmurowe lub też wzajemne wpływanie przez programistów na wyniki testów.

Niezwykle częstą praktyką przy korzystaniu z usług bezserwerowych jest używanie podejścia *infrastruktura jako kod*, które ułatwia wersjonowanie oraz umożliwia łatwe przenoszenie konfiguracji pomiędzy środowiskami. Ponadto pomaga ono w dodawaniu oraz usuwaniu komponentów, których z czasem robi się coraz więcej. Narzędzia reprezentujące tę dziedzinę, takie jak Terraform [19], pomagają w modelowaniu infrastruktury, a gotowe moduły przygotowane przez dostawców chmury publicznej ułatwiają ich użycie.

4.3. Wymagania dotyczące kosztów oraz wiedzy specjalistycznej przed rozpoczęciem prac

Użycie rozwiązań bezserwerowych oraz usług dostawców chmurowych generuje koszty zupełnie inaczej niż w przypadku klasycznego podejścia. Tu małe są koszty początkowe, natomiast koszt comiesięczny pozostaje zwykle na stałym poziomie. Na koszt całkowity składają się:

- **Comiesięczny koszt związany ze zużyciem usług** - ważne jest, aby zrozumieć, że wraz ze wzrostem ilości użytkowników wzrasta również zapotrzebowanie na zasoby, co zwiększa koszty.
- **Koszt związany z zatrudnieniem specjalistów** – bardzo często pomijany aspekt. Wskazuje na potrzebę zatrudniania np. programistów, którzy potrafią wytwarzać oprogramowanie z wykorzystaniem usług bezserwerowych. Co więcej, w niektórych organizacjach powołuje się specjalnie zespoły, które zajmują się monitoringiem, rozwojem i utrzymaniem usług w ramach chmury publicznej, z której korzysta firma.

Pozostałe koszty, takie jak koszt licencji, zakupu serwerów, wydzielania miejsca na serwerownię, zakupu i konfiguracji monitoringu oraz przeprowadzania audytów bezpieczeństwa infrastruktury nie mają zastosowania dla tego podejścia. W większości przypadków usługi bezserwerowe nie wymagają licencji, przechodzą regularnie testy bezpieczeństwa i spełniają najwyższe certyfikacje i normy w tym zakresie. Przechodząc do monitoringu – każda z usług bezserwerowych ma już predefiniowane pulpity z metrykami oraz zdefiniowane wysyłanie podstawowych alertów na maila.

4.4. Przegląd narzędzi wspomagających wytwarzanie oprogramowania z wykorzystaniem usług bezserwerowych

W ramach wytwarzania oprogramowania z usługami bezserwerowymi niezwykle ważne jest zaplecze narzędziowe, które jest niezbędne do pracy w ramach tego środowiska. Na postawie doświadczeń magistranta oraz [44] przygotowano listę narzędzi wspomagających:

- **Terraform** [19] – narzędzie do zarządzania stanem infrastruktury, implementujące podejście „*infrastruktura jako kod*", które może być również używane w ramach podejścia klasycznego. Można znaleźć wiele przykładów użycia, gotowych modułów, a dostawcy usług bezserwerowych oferują wsparcie dla tego narzędzia.
- **Pulumi** [45] – narzędzie służące do zarządzania stanem infrastruktury, konfigurowane za pomocą jednego z wybranych języków programowania, takich jak Python czy Java. Ciekawym aspektem jest to, że nie trzeba uczyć się nowego specyficznego dla danego narzędzia języka

(jak w przypadku Terraform [19]), a zamiast tego można skorzystać z jednego ze znanych języków programowania.

- **localstack** [43] – narzędzie w pełni imitujące usługi chmurowe AWS [1], które można uruchomić na laptopie programisty. Pozwala ono na szybsze testowanie oraz obniżenie kosztów poprzez symulowanie wywołań do prawdziwych usług.
- **AWS Amplify** [46] – framework przeznaczony do budowy pełnych aplikacji w ramach chmury publicznej AWS [1]. Umożliwia łatwe dodawanie komponentów do swojej aplikacji poprzez abstrakcje. Przykładowo zamiast zastanawiania się nad nazwą serwisu w AWS [1], który odpowiada za przechowywanie, wskazujemy, że będziemy potrzebowali przestrzeni dyskowej (ang. *storage*) a framework powołuje dla naszej aplikacji odpowiednie obiekty. Narzędzie to pozwala na jeszcze szybsze tworzenie aplikacji i jeszcze bardziej abstrahuje pojęcia związane z infrastrukturą.
- **ArgoCD** [18] – narzędzia służące do ciągłego wdrażania (ang. *continuous deployment*) umożliwiające instalowanie aplikacji na klastrach Kubernetes [16]. Dzięki zaawansowanym funkcjonalnościom możliwe jest tworzenie jednoczesnych wdrożeń na wiele chmur publicznych.
- **Ngrok** [47] – narzedzie przydatne w przypadku testowania zwrotnych wywołań webowych (ang. *webhooks*). Część usług bezserwerowych umożliwia użycie *webhook*'ów jako powiadomień w przypadku wystąpienia danego zdarzenia.

Pozostałe narzędzia takie jak Prometheus [25], Grafana [24] czy Ansible [31], również są używane w połączeniu z usługami bezserwerowymi np. do ich konfiguracji czy monitoringu.

5. Wykorzystane technologie

W ramach rozdziału omówione są technologie, frameworki, języki oraz narzędzia wykorzystane w ramach implementacji aplikacji internetowej. Czytelnik jest przeprowadzony przez technologie użyte w wytwarzaniu oprogramowania w sposób klasyczny i z wykorzystaniem usług bezserwerowych. Na końcu opisano dodatkowe narzędzia użyte w ramach implementacji.

5.1. Wykorzystane technologie w wytwarzaniu oprogramowania w sposób klasyczny

W zakresie implementacji z wykorzystaniem podejścia klasycznego użyte zostały następujące technologie:

- Redis [48] – baza danych typu klucz-wartość. Dzięki trzymaniu obiektów w pamięci RAM oraz możliwości utrwalania ich na dysku zapewnia wysoką wydajność oraz trwałość danych. Dodatkowo oferuje struktury danych takie jak uporządkowane zestawy (ang. *sorted set*) oraz wsparcie dla operacji na ciągach znaków czy liczbach.
- RabbitMQ [49] – popularny broker wiadomości oferujący tworzenie kolejek wiadomości (ang. *message queue*). Używany w ramach komunikacji asynchronicznej w celu zmniejszenia powiązania pomiędzy komponentami systemu. Zapewnia wysoką wydajność oraz skalowalność.
- Keycloak [30] – oprogramowanie umożliwiające uwierzytelnianie i autoryzację. Zapewnia wiele integracji z popularnymi protokołami takimi jak LDAP[3] czy OAuth 2.0[4]. Dodatkowo oferuje połączenie kont z serwisami takimi jak Facebook [33] czy Google [34]. Rozwiązanie to jest bardzo popularne ze względu na łatwość konfiguracji oraz możliwość dostosowywania wyglądu stron logowania.

5.2. Wykorzystane technologie w wytwarzaniu oprogramowania z użyciem usług bezserwerowych

Implementacja wykorzystująca usługi bezserwerowe zawiera listę następujących technologii:

- AWS DynamoDB [37] – baza danych typu klucz wartość stworzona przez AWS [1]. Pozwala na przechowywanie dużych ilości danych oraz automatyczne skalowanie. Dzięki możliwości georeplikacji zapewnia minimalne opóźnienia, a redundancja danych zabezpiecza przed utratą informacji. Dodatkowo zawiera wiele funkcjonalności jak np. tworzenie strumieni zdarzeń ze zmianami w bazie. Bardzo często wykorzystywana w rozwiązaniach e-commerce w celu monitorowania aktywności klientów lub jako dodatkowa warstwa cache.
- AWS SQS [50] – zarządzalna usługa kolejkowania wiadomości. Umożliwia ona komunikację asynchroniczną pomiędzy komponentami systemu. Oferuje dodatkowy monitoring oraz integrację z pozostałymi usługami AWS [1] takimi jak AWS Lambda [51].

[3] LDAP (ang. *Lightweight Directory Access Protocol*) to protokół przeznaczony do udostępniania usług katalogowych w sieci jak np. baza użytkowników. Najbardziej powszechną implementacją jest usługa Active Directory firmy Microsoft.

[4] OAuth 2.0 to protokół pozwalający na budowę mechanizmów autoryzacyjnych. W ramach protokołu istnieje wiele świeżek (ang. *flows*), które mogą być implementowane i wykorzystywane do zabezpieczenia dostępu dla aplikacji mobilnych, webowych oraz okienkowych. Więcej informacji na oficjalnej stronie protokołu [81]

- AWS Lambda [51] – usługa umożliwiająca uruchamianie aplikacji oraz prostych funkcji bez potrzeby zarządzania infrastrukturą. Implementuje podejście FaaS (ang. *Function as a Service*), które umożliwia dostarczenie kodu i wykonanie go, bez konieczności martwienia się o zasoby. Usługa umożliwia również automatyczne skalowanie w zależności od ilości żądań użytkowników lub własnych parametrów.
- AWS Cognito [38] – usługa zapewniająca uwierzytelnianie, autoryzację i zarządzanie użytkownikami w aplikacjach. Zaletą jej jest wysokie bezpieczeństwo przechowywania danych użytkowników oraz łatwość w integracji z aplikacją. Umożliwia bezpieczne logowanie przez konta stron trzecich jak Facebook [33] czy Google [34].
- Spring Cloud for Amazon Web Services [52] – biblioteka oferująca łatwą integrację z usługami AWS [1] w ramach ekosystemu Spring [53]. Stworzona przez społeczność na bazie doświadczeń w interakcji z usługami chmurowymi. Pomaga w szybkim uruchomieniu projektu wykorzystującego AWS SQS [50] jako kolejkę wiadomości.
- Spring Cloud Function [54]– biblioteka umożliwiająca tworzenie kodu odpowiedzialnego za logikę biznesową w postaci funkcji. Stworzonych w ramach niej aplikacji można używać w środowiskach lokalnych, jak i chmurowych. Zaletą jest tu mnogość mechanizmów znanych ze Spring [53] takich jak wstrzykiwanie zależności czy udostępnianie metryk danej funkcji.
- AWS DynamoDB local [55] – technologia umożliwiająca uruchomienie lokalnie bazy danych na potrzeby rozwoju oraz testowania aplikacji. Pozwala na tworzenie projektu bez konieczności dostępu do internetu. Przydatna w szczególności, gdy programista ponosi wysokie koszty transferu danych sieciowych.

5.3. Języki programowania, frameworki oraz biblioteki użyte w projekcie

Implementacja projektu wymaga użycia języków programowania, frameworków oraz bibliotek. Co istotne – poniżej wymienione elementy są wspólne dla wersji wykorzystującej klasyczne podejście jak i tej z usługami bezserwerowymi. Pokazuje to jak wiele technologii może być wspólnie wykorzystywanych w ramach każdego z podejść. W projekcie zastosowano następujące elementy:

- Java [56] – jeden z najbardziej popularnych obiektowych języków programowania. Używany do tworzenia aplikacji internetowych, systemów finansowych czy gier komputerowych. Popularny ze względu na jego prostotę, bezpieczeństwo, wydajność oraz skalowalność.
- Python [57] – popularny wysokopoziomowy język programowania. Jest stosowany głównie w uczeniu maszynowym, inżynierii danych (ang. *data engineering*) czy tworzeniu aplikacji internetowych. Często też jest używany do tworzenia skryptów automatyzujących pracę czy testowania oprogramowania.
- Bash [58] – interpreter poleceń i język skryptowy używany w systemach operacyjnych takich jak Linux [59] czy macOS [60]. Używany przy tworzeniu skryptów automatyzujących pracę oraz bardziej zaawansowanej interakcji z systemem. Dzięki elastyczności oraz dostępności jest powszechnie stosowany przez programistów oraz administratorów systemów.
- Spring [53] – popularny framework dla języka Java [56], umożliwiający szybkie tworzenie skalowanych oraz elastycznych aplikacji. Dostarcza on gotowe komponenty do tworzenia aplikacji webowych (Spring Boot Web [60]), integracji z bazami danych (Spring Data [61]) czy zabezpieczania aplikacji (Spring Security [62]). Wykorzystywany w projektach open-source oraz projektach komercyjnych z uwagi na niezawodność, stabilność oraz dużą ilość dokumentacji.
- Lombok [63] – biblioteka używana do automatycznego generowania kodu. Wykorzystuje ona mechanizm dodawania do kodu adnotacji, dzięki którym biblioteka tworzy

powtarzalne metody wykorzystywane w Javie [56]. Dzięki niej programiści mogą więcej czasu poświęcić na tworzenie funkcjonalności biznesowych.

- Immutables [64] – biblioteka używana do automatycznego generowania kodu dla obiektów niemutowalnych[5]. Biblioteka ta – podobnie jak Lombok [63] – korzysta z mechanizmu adnotacji, dzięki którym generuje gotowe klasy. Główną jej zaletą jest tworzenie obiektów niemutowalnych, co zmniejsza ryzyko wystąpienia błędów czy problemów związanych z wielowątkowością.
- Yauaa [65] – biblioteka używana do ekstrakcji informacji o przeglądarce użytkownika. Wykorzystuje ona nagłówek żądania HTTP *User-Agent* [66] do określenia informacji o przeglądarce, systemie operacyjnym czy urządzeniu, z którego korzysta użytkownik.
- MapStruct [67] – biblioteka służąca do automatycznego mapowania pomiędzy różnymi typami obiektów. Automatycznie generuje kod zaoszczędzając czas programiście. Dodatkowo umożliwia definiowanie własnych mapowań pomiędzy typami, dziedziczenie strategii mapowań oraz pobieranie dla mapowań danych z różnych źródeł.
- Testcontainers [68] – wielojęzykowy framework umożliwiający tworzenie kontenerów z poziomu kodu. Głównym zastosowaniem jest uruchamianie np. bazy danych lub kolejki wiadomości w trakcie testów aplikacji, aby sprawdzić integrację pomiędzy aplikacją, a usługami zewnętrznymi.
- JUnit [69] – framework służący tworzeniu testów w języku Java [56]. Oferuje wiele mechanizmów m.in. uruchamianie metod odpowiedzialnych za konfigurację, przygotowanie środowiska testowego czy tworzenie testów parametryzowanych[6].
- Rest-Assured [70] – biblioteka umożliwiająca testowanie REST API w sposób deklaratywny. Oferuje również wiele asercji na zwracanych odpowiedziach oraz wspiera różne formaty danych takie jak JSON, XML czy HTML.
- AssertJ [71] – biblioteka asercji wykorzystywana w ramach testów jednostkowych. Udostępnia szeroki zestaw asercji dla obiektów oraz struktur danych jak mapy oraz listy. Dodatkowo umożliwia tworzenie własnych komunikatów o błędach podczas testów, co ułatwia znalezienie przyczyny niepowodzenia testu.
- Gatling [72] – framework umożliwiający tworzenie testów wydajnościowych w ramach języka programowania Java [56]. Pozwala na przeprowadzenie testów symulujących prawdziwych klientów systemu. Narzędzie to jest używane do znajdywania problemów z wydajnością aplikacji przed lub po wdrożeniu produkcyjnym.

5.4. Pozostałe narzędzia użyte w ramach projektu

W ramach projektu użyto również szeregu innych narzędzi pomocnych w ramach wytwarzania oraz wdrażania aplikacji:

- Terraform [19] – narzędzie pomagające w zarządzaniu infrastrukturą w sposób deklaratywny. Dzięki niemu można wersjonować i testować konfigurację, a także generować plany zmian dotyczących infrastruktury.
- Docker Compose [73] – jak opisał to narzędzie Jeff Nickoloff w książce „Docker in Action" [74]: „*...Compose jest narzędziem do definiowania, uruchamiania i zarządzania serwisami, gdzie każdy*

[5] Typ obiektów, który po jego stworzeniu nie może być już zmieniony. Główne zastosowanie takich obiektów jest w systemach wykorzystujących wielowątkowość lub w aplikacjach, gdzie stan obiektów podczas ich życia nie zmienia się zbyt często.

[6] Rodzaj testów, dla których podajemy parametry wejściowe i oczekiwany rezultat testowanej funkcjonalności. Ułatwiają zmniejszenie powtarzalności kodu oraz dodawanie nowych przypadków testowych.

serwis to jedna lub więcej replik kontenera Docker.". Definiuje się go w ramach plików konfiguracyjnych w formacie YAML. Ułatwia pracę oraz testowanie lokalne danej aplikacji wraz z zewnętrznymi zależnościami, takimi jak baza danych czy kolejka wiadomości.

- GitHub Actions [17] – narzędzie umożliwiające tworzenie potoków CI/CD. Można w nich zdefiniować automatyczne testowanie kodu lub wdrażanie artefaktu na wybrane środowisko.
- Helm [75] – narzędzie do tworzenia pakietów, które pomagają w łatwiejszym wdrożeniu aplikacji wdrażanych na klaster Kubernetes [16]. Konfiguracja tworzona jest na podstawie plików YAML, w ramach których można korzystać z funkcji dostarczonych przez narzędzie. Dodatkowo umożliwia testowanie stworzonych pakietów oraz tworzenie generycznych szablonów aplikacji.
- AWS ELB [76] – usługa oferująca równoważenie obciążenia sieciowego dla aplikacji znajdujących się w chmurze publicznej AWS [1]. Serwis może równomiernie rozdzielać obciążenie sieciowe dla kontenerów czy maszyn wirtualnych. Oferuje również 3 podtypy równoważników obciążenia, z których każdy znajduje zastosowanie w ramach specyficznych rozwiązań.
- AWS CloudWatch [77] – usługa umożliwiająca monitorowanie i zbieranie metryk, logów oraz zdarzeń z różnych usług w ramach chmury publicznej AWS [1]. Umożliwia tworzenie automatyzacji na bazie zdarzeń oraz wywoływanie akcji po ich wystąpieniu.
- AWS EKS [78] – zarządzany przez AWS [1] klaster Kubernetes [16]. Umożliwia uruchamianie, skalowanie i zarządzanie aplikacjami opartymi na Kubernetes. Dodatkową funkcjonalnością jest łatwa integracja z pozostałymi usługami AWS takimi jak AWS ELB [76] czy AWS CloudWatch [77].

6. Praktyczny projekt przykładowej aplikacji internetowej do skracania linków

Niniejszy rozdział poświęcony jest aplikacji internetowej oraz opisowi jej dwóch implementacji, które powstały jako część pracy magisterskiej i stanowią duży wkład w porównanie podejścia klasycznego oraz bezserwerowego. W pierwszym podrozdziale przedstawiono cel projektu. Podrozdział drugi zawiera wymagania funkcjonalne oraz niefunkcjonalne. Ostatnie dwie części rozdziału zawierają kluczowe fragmenty implementacji w ramach każdego z podejść.

6.1. Cel projektu

W ramach projektu stworzone zostały dwie implementacje aplikacji internetowej umożliwiającej skracanie linków. Aplikacja ta udostępnia jedynie interfejs oparty o architekturę REST API. Decyzja o pominięciu tworzenia warstwy graficznego interfejsu użytkownika została podjęta przed rozpoczęciem prac. Wynika to z braku różnic w zakresie komunikacji pomiędzy częścią interfejsu graficznego (ang. *frontend*), a częścią serwerową aplikacji (ang. *backend*) w obu podejściach. W celu dostarczenia materiału porównawczego aplikacja internetowa implementuje ten sam kontrakt API [79], dzięki czemu zapewnia taki sam zestaw funkcjonalności. Każda z implementacji zawiera również części wspólne m.in. logikę domenową czy kontrakt API, które są niezmienne bez względu na użytą bazę danych lub brokera wiadomości. Ma to na celu usunięcie powtarzania kodu oraz wyodrębnienie modułów, które zawierają implementacje każdego z podejść.

Stworzone projekty służą do porównania w kilku aspektach implementacji aplikacji z wykorzystaniem klasycznego podejścia oraz analogicznej aplikacją utworzonej z wykorzystaniem usług bezserwerowych w ramach chmury publicznej AWS. Oba projekty zostały napisane w języku Java [56] wykorzystując framework Spring [53]. Jako nazwę aplikacji wybrano 'shorter' (ang. *skracacz*), co wiąże się bezpośrednio z jej funkcjonalnością.

6.2. Wymagania funkcjonalne oraz niefunkcjonalne

Niezwykle istotnym elementem wytwarzania oprogramowania jest precyzyjne określenie wymagań. Pozwala to zaprojektować system, którego oczekują osoby zamawiające oraz docelowi użytkownicy.

Tabela 1 zawiera listę wymagań dla oprogramowania tworzonego w ramach niniejszej pracy magisterskiej, które są wspólne dla obu implementacji.

Tabela 1 Spis wymagań dla implementacji projektu. Źródło: Opracowanie własne

Numer	Treść wymagania	Rodzaj wymagania	Priorytet	Czy zrealizowano?
1	Użytkownik będzie mógł wygenerować krótki link.	Wymaganie funkcjonalne	Wysoki	Tak

2	W systemie powinni być rozróżniani następujący użytkownicy: • Administrator • Użytkownik standardowy	Wymaganie funkcjonalne	Wysoki	Tak
3	Użytkownik o uprawnieniach Użytkownik standardowy będzie miał możliwość sprawdzania aktywności swoich linków oraz modyfikacji parametrów linku.	Wymaganie funkcjonalne	Wysoki	Tak
4	Użytkownik o uprawnieniach Administrator będzie miał możliwość odczytu, edycji i usuwania linków oraz zarządzania użytkownikami.	Wymaganie funkcjonalne	Wysoki	Tak
5	Hasła powinny być przechowywane w bazie danych w zaszyfrowanej postaci	Wymaganie funkcjonalne	Wysoki	Tak
6	Dostęp do danych formularzy oraz zarządzania użytkownikami powinien być dostępny dla użytkowników posiadających uprawnienia, określane po poprawnym zalogowaniu nazwą użytkownika i hasłem.	Wymaganie niefunkcjonalne	Wysoki	Tak
7	System powinien oferować możliwość wylogowania się użytkownikowi po zalogowaniu.	Wymaganie funkcjonalne	Wysoki	Tak
8	Wszystkie dane powinny być przechowywane w bazie danych typu klucz-wartość.	Wymaganie funkcjonalne	Wysoki	Tak
9	System powinien być zaimplementowany w oparciu o najpopularniejsze frameworki i biblioteki związane z Javą.	Wymaganie niefunkcjonalne	Wysoki	Tak
10	Dane dotyczące użytkowników oraz aplikacja uwierzytelniająca powinny być osobną aplikacją od aplikacji obsługującej zarządzanie formularzami i użytkownikami.	Wymaganie funkcjonalne	Wysoki	Tak

11	System automatycznie wyloguje zalogowanego użytkownika po czasie bezczynności.	Wymaganie funkcjonalne	Wysoki	Tak
12	Powinny istnieć kopie zapasowe danych systemu, umożliwiające jego odtworzenie.	Wymaganie niefunkcjonalne	Wysoki	Tak
13	Administrator może stworzyć konto dla użytkownika w aplikacji.	Wymaganie funkcjonalne	Wysoki	Tak
14	Użytkownik powinien móc pobrać dane dotyczące aktywności korzystania z linków.	Wymaganie funkcjonalne	Wysoki	Tak
15	Aplikacja powinna wspierać integrację z innymi systemami za pomocą REST API.	Wymaganie funkcjonalne	Wysoki	Tak
16	Aplikacja umożliwia zmianę danych użytkownika (email, hasło, nazwa użytkownika).	Wymaganie funkcjonalne	Wysoki	Tak
17	Czas wygenerowania krótkiego linku nie powinien być dłuższy niż 3 sekundy.	Wymaganie niefunkcjonalne	Wysoki	Tak
18	Analiza dotycząca linku powinna zawierać informacje o urządzeniu, przeglądarce, producencie oraz modelu urządzenia.	Wymaganie funkcjonalne	Wysoki	Tak
19	Dla każdego kliknięcia zbierane są następujące dane (o ile są dostępne w nagłówku żądania HTTP): • Data kliknięcia w link • Typ urządzenia • Producent urządzenia • OS urządzenia • Wersja OS • Nazwa przeglądarki	Wymaganie funkcjonalne	Wysoki	Tak

	• Wersja przeglądarki • Dokładna wersja przeglądarki			
20	Dane dotyczące aktywności po kliknięciu będą dostępne dla użytkownika po czasie nie dłuższym niż 3 minuty.	Wymaganie niefunkcjonalne	Wysoki	Tak
21	Aplikacja będzie pozwalać na tworzenie własnych (nielosowych) krótkich linków.	Wymaganie funkcjonalne	Wysoki	Tak
22	Po kliknięciu lub wysłaniu zapytania HTTP za pomocą krótkiego linku użytkownik zostanie przekierowany do strony, dla której stworzono krótki link.	Wymaganie funkcjonalne	Wysoki	Tak
23	Użytkownik może skorzystać z krótkiego linku bez potrzeby logowania.	Wymaganie funkcjonalne	Wysoki	Tak
24	Użytkownik może skorzystać z linku wyłącznie, jeśli znajduje się w sieci, która pozwala na dostęp do aplikacji.	Wymaganie funkcjonalne	Wysoki	Tak
25	Użytkownik może tworzyć krótkie linki, które odnoszą się do innych krótkich linków.	Wymaganie funkcjonalne	Wysoki	Tak

Wszystkie wyżej wymienione wymagania zostały zrealizowane. Jednoznacznie definiują one cechy aplikacji oraz funkcje, które są dostarczane w ramach projektu.

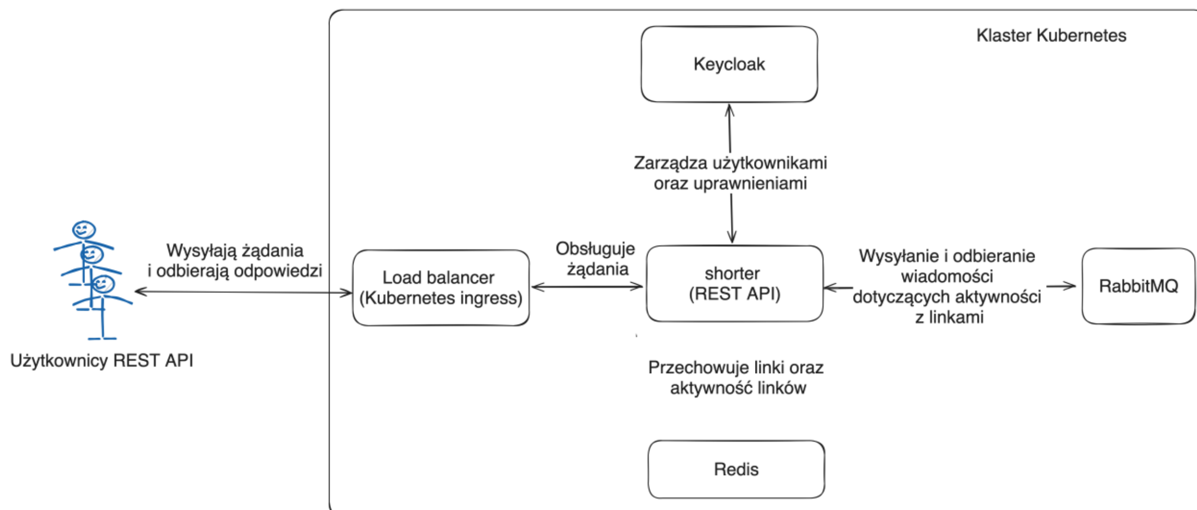
Dalsze podrozdziały przedstawiają kod z opisem mechanizmu działania, zrzuty ekranu oraz rysunki, które według autora są najważniejszymi fragmentami każdej z implementacji. Pozostałe materiały oraz kod źródłowy w ocenie magistranta nie są kluczowe dla zrozumienia działania każdego z podejść.

6.3. Implementacja aplikacji internetowej z wykorzystaniem klasycznego podejścia

W ramach podrozdziału omawiana jest architektura rozwiązania oraz zakres wykorzystania frameworka Spring [53]. Dodatkowo pokazana jest implementacja projektu, która została wykonana w ramach porównania.

6.3.1 Architektura implementacji

W celu lepszego zrozumienia implementacji projektu należy zapoznać się z wysokopoziomową architekturą rozwiązania pokazaną na Rysunku 2. W ramach tego podejścia zostały wykorzystane technologie, które mogą być zainstalowane w lokalnych serwerowniach lub chmurze publicznej.



Rysunek 2 Wysokopoziomowy diagram implementacji w podejściu klasycznym. Źródło: opracowanie własne

Rdzeniem rozwiązania jest klaster Kubernetes [16], na którym zainstalowana jest aplikacja. Aby odpowiednio rozprowadzić ruch klientów aplikacji zastosowano równoważnik obciążenia (ang. *load balancer*), który odpowiada za równomierne rozdzielenie żądań użytkowników dla wielu instancji aplikacji. Wykorzystano tutaj rozwiązanie Kubernetes Ingress [80], które oprócz funkcji równoważnika ruchu, umożliwia udostępnienie aplikacji użytkownikom. Posiada ono możliwość dodania szyfrowania połączenia pomiędzy użytkownikiem, a serwerem, co znacząco wpływa na bezpieczeństwo rozwiązania. Aplikacja przechowuje dane dotyczące linków oraz ich aktywności w nierelacyjnej bazie danych Redis [48]. Jej główna cecha – przechowywanie danych klucz-wartość sprawia, że dostęp do konkretnego linku jest szybki. Za autoryzację oraz zarządzanie użytkownikami jest odpowiedzialne rozwiązanie Keyclock [30]. Implementuje ono protokół OAuth2 [81], z którego korzysta aplikacja. Dzięki temu nie mamy w ramach aplikacji skomplikowanej logiki odpowiedzialnej za uwierzytelnianie czy przechowywanie haseł użytkowników. Ostatnim komponentem jest broker wiadomości RabbitMQ [49], który odpowiada za dystrybucję wiadomości dotyczących aktywności z użyciem linków.

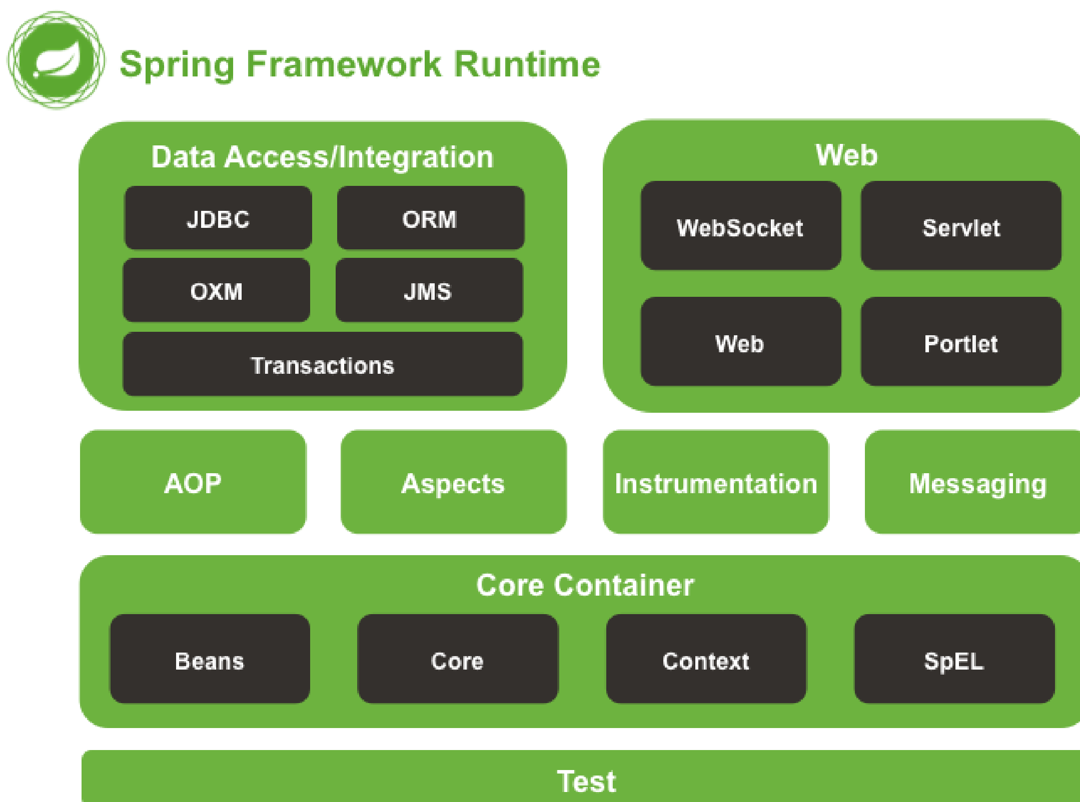
6.3.2 Omówienie frameworka Spring

Wykorzystywany w ramach implementacji framework Spring [53] to rozwiązanie dla aplikacji pisanych w językach takich jak Java [56], Kotlin [82] czy Groovy [83]. Ułatwia tworzenie i utrzymywanie aplikacji, które są wykorzystywane na dużą skalę. Programista chcąc użyć tego frameworka może skorzystać z wybranych modułów udostępnionych w ramach ekosystemu, dzięki czemu tworzona aplikacja nie zawiera zbędnych zależności. Poniżej znajduje się lista modułów z rodziny Spring użytych w ramach projektu:

- **IoC Container** – kontener aplikacji, który jest podstawowym modułem frameworka Spring. Dostarcza on mechanizmy umożliwiające wstrzykiwanie zależności (ang. *dependency injection*) [84] ułatwiające łączenie poszczególnych klas w ramach aplikacji.

- **Web** – moduł zawierający mechanizmy do budowania aplikacji internetowych. Udostępnia on mechanizmy tworzenia REST API poprzez adnotacje w kodzie. Umożliwia również dodanie walidacji otrzymywanych żądań oraz modyfikacji wysyłanych kodów odpowiedzi czy też dodawania nagłówków HTTP.
- **Data Access/Integration** – zawiera rozwiązania odpowiedzialne za integrację z różnymi bazami danych. Dodaje abstrakcję na tabele lub zbiory danych w postaci repozytorium (ang. *repositorty*), dzięki czemu implementacja w kodzie nie różni się znacząco mimo użycia odmiennych baz danych.
- **Messaging** – moduł umożliwiający integrację z brokerami wiadomości. Dostarcza abstrakcję, dzięki której programista z łatwością może podłączyć się do usługi oferującej odbieranie i wysyłanie wiadomości do kolejki lub innych rozwiązań opartych o komunikaty.
- **Test** – zbiór narzędzi ułatwiających tworzenie testów jednostkowych oraz integracyjnych. Współpracuje bardzo dobrze z innymi bibliotekami z ekosystemu Javy, takimi jak JUnit [69] czy testcontainers [68].

Wyżej opisane moduły oraz rozwiązania w ramach Spring przedstawia Rysunek 3.



Rysunek 3 Zbiór komponentów frameworka Spring. Źródło: [85]

Użyta w projekcie wersja Spring to 6.0.5 jest jedną z najnowszych wersji datowaną na 15 lutego 2023 roku [86].

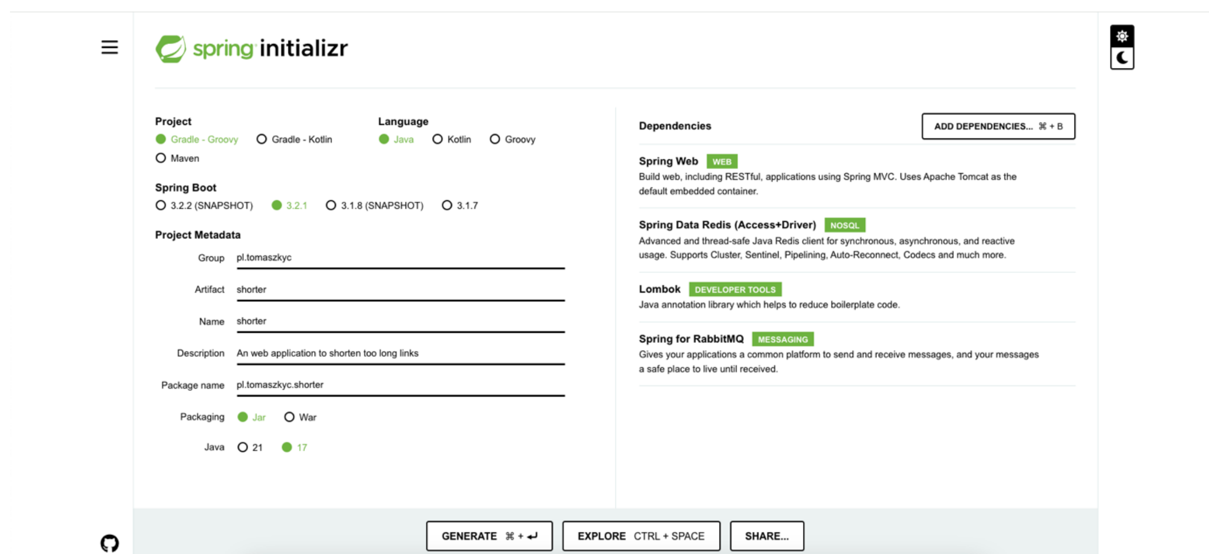
W ramach projektu użyto rozszerzenia Spring – Spring Boot. Pozwala to na szybsze prototypowanie i tworzenie aplikacji przez programistę dzięki dostarczeniu wielu elementów

skonfigurowanych w sposób zgodny z najlepszymi praktykami. Ten moduł automatycznie dostraja konfigurację aplikacji, a w efekcie pozwala programiście skupić się na dostarczaniu rozwiązania. W ramach projektu została użyta wersja Spring Boot 3.0.3 datowana na 23 lutego 2023 roku [87].

Pomocnym narzędziem w tworzeniu aplikacji z rodziny Spring jest Spring Initializr [88]. Umożliwia ono utworzenie szkieletu aplikacji wraz ze zdefiniowaniem:

- Języka programowania (do wyboru Java, Groovy lub Kotlin)
- Narzędzia do budowania projektu (Gradle/Maven)
- Wersji Spring Boot
- Wersji Javy
- Dodatkowych modułów, które mają zostać dodane do szkieletu aplikacji
- Metadanych projektu (nazwa i opis projektu)

Rysunek 4 prezentuje przykładowe skonfigurowanie szkieletu projektu tuż przed jego wygenerowaniem.



Rysunek 4 Tworzenie szkieletu aplikacji shorter przy wykorzystaniu narzędzia Spring Initializr. Źródło: [88]

6.3.3 Omówienie implementacji na przykładzie funkcjonalności użycia skróconego linku

Jedną z głównych funkcjonalności aplikacji jest możliwość przekierowania użytkownika do docelowej strony za pomocą utworzonego wcześniej krótkiego linku. Podana czynność dotyka większości komponentów aplikacji, stąd jest dobrym przykładem na przekrojowe pokazanie działania aplikacji.

Użycie skróconego linku przez użytkownika przebiega w następujących krokach:

- Użytkownik wysyła żądanie HTTP GET[7] do ścieżki ‘`/api/v1/link/xOOY44/redirect`’, gdzie ‘`xOOY44`’ jest przykładowym krótkim linkiem.

[7] W ramach protokołu HTTP zdefiniowane zostały czasowniki specyfikujące żądania. GET jest najbardziej popularnym czasownikiem używanym w ramach przeglądarek internetowych. Więcej informacji w [101]

- Żądanie jest przekazywane do klasy ‘LinkEndpoint’ i metody ‘redirectByShortLink’ pokazanej w Listingu 1. W ramach wywołania użytkownika tworzony jest obiekt zawierający krótki link, wartość nagłówka HTTP o nazwie ‘User-Agent’[8] oraz datę wystąpienia żądania.
- Tak stworzony obiekt jest przekazywany do instancji klasy ‘LinkApiService’ i metody ‘redirectByShortLink’ pokazanej w Listingu 2. W jej ramach pobierane są dane dotyczące krótkiego linku (w tym wartość długiego linku).
- Pobranie danych linku następuje poprzez wywołanie instancji klasy ‘LinkService’ i metody ‘findByShortLink’. Następnie pobierany jest link poprzez wywołanie metody ‘findById’ w ramach interfejsu ‘LinkRepository’. Pokazane jest to w Listingu 3.

Listing 1 Fragment klasy LinkEndpoint odpowiedzialny za przekierowywanie użytkowników. Źródło: Opracowanie własne

```
@RestController
@RequestMapping("/api/v1/link")
public class LinkEndpoint {

    private final LinkApiService linkApiService;
    private final Clock clock;

    public LinkEndpoint(LinkApiService linkApiService, Clock clock) {
        this.linkApiService = linkApiService;
        this.clock = clock;
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public LinkDto create(Principal principal,
                                        @RequestBody LinkDto linkDto) throws
Exception {
        return linkApiService.save(linkDto, principal);
    }

    @GetMapping("/{shortLink}/redirect")
    @ResponseStatus(HttpStatus.MOVED_PERMANENTLY)
    public ResponseEntity<?> redirectByShortLink(@PathVariable(name =
"shortLink") ShortLink shortLink,
                                                 @RequestHeader(value =
HttpHeaders.USER_AGENT, required = false) String userAgentHeader) {
        LinkDto linkDto = linkApiService.redirectByShortLink(new
LinkActivityMetadata(shortLink, userAgentHeader, ZonedDateTime.now(clock)));
        return createLinkRedirectResponse(linkDto);
    }

    private static ResponseEntity<Object> createLinkRedirectResponse(LinkDto
linkDto) {
        HttpHeaders headers = new HttpHeaders();
        headers.setLocation(URI.create(linkDto.longLink()));
        return new ResponseEntity<>(headers, HttpStatus.MOVED_PERMANENTLY);
    }
```

[8] Często występujący nagłówek HTTP w ramach żądań użytkowników. Zawiera on informacje na temat przeglądarki, systemu operacyjnego, typu urządzenia oraz innych istotnych szczegółów. Więcej informacji można znaleźć w [66]

Listing 2 Fragment klasy LinkApiService odpowiedzialny za przekierowywanie użytkowników.
Źródło: Opracowanie własne

```
LinkDto redirectByShortLink(LinkActivityMetadata linkActivityMetadata) {
    LinkDto linkDto =
LinkDto.fromLink(linkService.findByShortLink(linkActivityMetadata.shortLink())
            .orElseThrow(() -> new NotFoundException("Not found short link" +
linkActivityMetadata.shortLink().toString())));
    linkActivityService.parseUserAgentHeader(linkActivityMetadata);
    return linkDto;
}
```

Listing 3 Fragment klasy LinkService odpowiedzialny za pobranie linku po krótkim id.
Źródło: Opracowanie własne

```
public class LinkService {

    private final LinkRepository linkRepository;
    private final LinkActivityRepository linkActivityRepository;

    public LinkService(LinkRepository linkRepository,
                       LinkActivityRepository linkActivityRepository) {
        this.linkRepository = linkRepository;
        this.linkActivityRepository = linkActivityRepository;
    }


    public Optional<Link> findByShortLink(ShortLink shortLink) {
        return linkRepository.findById(LinkId.of(shortLink));
    }
```

- Do tego momentu kod w ramach dwóch podejść (klasycznego oraz bezserwerowego) jest taki sam. Różnice zaczynają się w implementacji interfejsu ‘`LinkRepository`’.
- W podejściu klasycznym wykorzystywany jest Redis [48]. Framework Spring [53] umożliwia łatwą integrację z tą bazą danych. W przypadku braku istnienia danej kolekcji w Redis – zostanie ona automatycznie utworzona na podstawie zdefiniowanej struktury w obiekcie klasy ‘`RedisLinkEntity`’. Zobrazowane jest to w Listingu 4.

Listing 4 Fragment klasy RedisLinkRepository odpowiedzialnej za wyszukanie linku.
Źródło: Opracowanie własne

```
@Repository
class RedisLinkRepository implements LinkRepository {

    private final RedisJpaLinkRepository redisJpaLinkRepository;

    RedisLinkRepository(RedisJpaLinkRepository redisJpaLinkRepository) {
        this.redisJpaLinkRepository = redisJpaLinkRepository;
    }

    @Override
    public Optional<Link> findById(LinkId linkId) {
        return redisJpaLinkRepository.findById(linkId.value())
                .map(RedisLinkEntity::to);
    }
```

- W przypadku nieznalezienia linku w klasie ‘`LinkApiService`’ rzucany jest wyjątek typu ‘`NotFoundException`’. Wyjątek jest przechwytywany przez Spring, a użytkownik otrzymuje odpowiedź o statusie HTTP 404[9].
- W przypadku znalezienia linku obiekt z jego metadanymi jest przekazywany do instancji klasy ‘`LinkActivityService`’ i jej metody ‘`parseUserAgentHeader`’.
- Następnie wywoływana jest metoda ‘`publish`’ na instancji interfejsu ‘`LinkActivityMetadataPublisher`’, implementowanego przez klasę ‘`RabbitMqLinkActivityMetadataPublisher`’. Klasa ta odpowiada za wysłanie wiadomości do instancji RabbitMQ [49]. Zachowanie to jest pokazane w Listingu 5.

Listing 5 Klasa RabbitMqLinkActivityMetadataPublisher odpowiadająca za publikowanie wiadomości na kolejkę RabbitMQ. Źródło: Opracowanie własne

```
@Component
class RabbitMqLinkActivityMetadataPublisher implements
LinkActivityMetadataPublisher {

    private static final Logger LOGGER =
LoggerFactory.getLogger(RabbitMqLinkActivityMetadataPublisher.class);

    private final RabbitTemplate rabbitTemplate;

    RabbitMqLinkActivityMetadataPublisher(RabbitTemplate rabbitTemplate) {
        this.rabbitTemplate = rabbitTemplate;
    }

    @Override
    public void publish(LinkActivityMetadata linkActivityMetadata) {
        LOGGER.debug("Sending to analyze link activity metadata: {}",
linkActivityMetadata);
        rabbitTemplate.convertAndSend(TOPIC_EXCHANGE_NAME, "foo.bar.baz",
                linkActivityMetadata);
    }
}
```

- Po wysłaniu wiadomości do kolejki zwracana jest odpowiedź do użytkownika z kodem odpowiedzi HTTP 301[10] oraz adresem pobranym z bazy danych. Odpowiedzialny za to fragment znajduje się w klasie ‘`LinkEndpoint`’ oraz metodzie ‘`createLinkRedirectResponse`’ pokazanej w Listingu 6.

Listing 6 Metoda odpowiedzialna za stworzenie odpowiedzi dla użytkownika. Źródło: Opracowanie własne

```
private static ResponseEntity<Object> createLinkRedirectResponse(LinkDto linkDto)
{
    HttpHeaders headers = new HttpHeaders();
    headers.setLocation(URI.create(linkDto.longLink()));
    return new ResponseEntity<>(headers, HttpStatus.MOVED_PERMANENTLY);
}
```

[9] Status HTTP 404 jest powszechnie znany użytkownikom internetu. Informuje on, że podana strona internetowa lub zasób nie istnieją. Istnieją również inne statusy odpowiedzi HTTP – więcej informacji w [100]

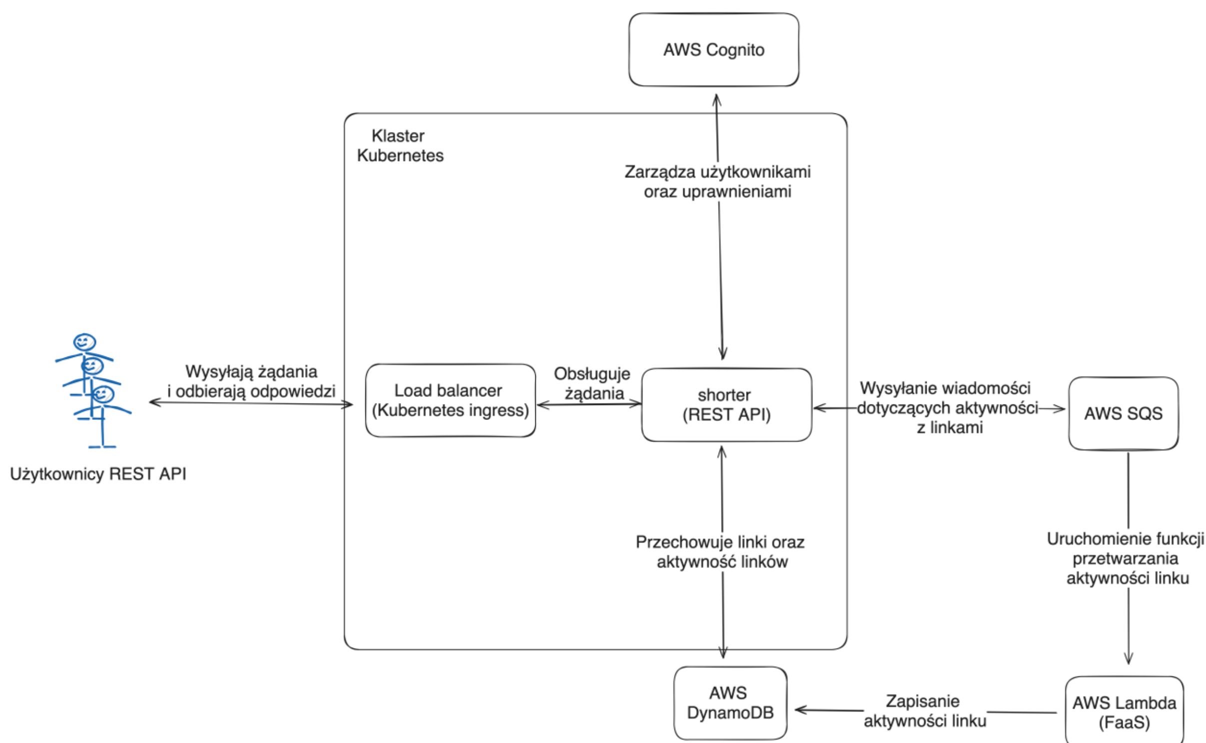
[10] Status HTTP 301 informuje przeglądarkę, że konkretny zasób został na stałe przeniesiony pod nowy adres internetowy. Więcej informacji w [100]

6.4. Implementacja aplikacji internetowej z wykorzystaniem usług bezserwerowych

Podrozdział przedstawia architekturę rozwiązania oraz wykorzystanie frameworka Spring Cloud [89]. Dodatkowo pokazana jest implementacja projektu, która została użyta w ramach porównania.

6.4.1 Architektura implementacji

Podobnie jak w podejściu klasycznym – na Rysunku 5 przedstawiona została wysokopoziomowa architektura rozwiązania oraz użyte bezserwerowe usługi chmury publicznej AWS [1].



Rysunek 5 Wysokopoziomowy diagram implementacji w podejściu bezserwerowym.
Źródło: Opracowanie własne

Podobnie jak w implementacji dla podejścia klasycznego, głównym komponentem rozwiązania jest klaster Kubernetes [16]. W ramach niego został skonfigurowany Kubernetes Ingress [80] udostępniający funkcję równoważnika ruchu oraz zabezpieczający ruch kliencki poprzez jego szyfrowanie. Na tym kończą się podobieństwa do pierwszego podejścia. Usługa AWS Cognito [38] odpowiada za zarządzanie użytkownikami oraz uprawnieniami. Aplikacja korzysta z jej implementacji protokołu OAuth2 [81]. Dane dotyczące linków oraz ich aktywności są przechowywane w ramach nierelacyjnej bazy danych AWS DynamoDB [37], zapewniającej szybkość pobierania danych oraz trwałość. W momencie skorzystania z krótkiego linku dane o aktywności są wysyłane do kolejki w AWS SQS [50]. Ta usługa oraz AWS Lambda [51] są ze sobą zintegrowane tak, że pojawienie się nowej wiadomości uruchamia instancję AWS Lambda. Aplikacja, po analizie metadanych z aktywności linku, zapisuje je w AWS DynamoDB. Instancja funkcji działa do momentu, gdy nie ma nowych wiadomości do przetworzenia w kolejce, po czym samoczynnie się wyłącza. Dzięki takiemu rozwiązaniu z głównej aplikacji zdejmowana jest odpowiedzialność dotycząca analizy i ekstrakcji danych z aktywności linku. Użycie kolejki wyłącza dodatkowe powiązanie pomiędzy główną aplikacją, a funkcją w ramach AWS Lambda.

6.4.2 Omówienie Spring Cloud oraz Spring Cloud Function

Rodzina Spring Cloud [89] to zbiór frameworków oraz bibliotek, umożliwiających łatwe tworzenie mikroserwisów przy wykorzystaniu frameworka Spring. Oferuje ona różne funkcjonalności takie jak odkrywanie usług (ang. *service discovery*), monitorowanie żądań użytkowników (ang. *requests tracing*) czy zabezpieczanie usług w środowisku mikroserwisowym. Najbardziej popularnymi projektami są:

- **Spring Cloud Config** – umożliwia scentralizowane tworzenie konfiguracji dla wielu aplikacji. Wspiera również automatyczne odświeżanie konfiguracji w aplikacjach, które z niego korzystają.
- **Spring Cloud Vault** – dostarcza integrację z usługą HashiCorp Vault [90] w celu bezpiecznego dostarczania haseł dla usług, z których korzysta aplikacja.
- **Spring Cloud Gateway** – komponent dostarczający funkcjonalność bramy API (ang. *API Gateway*) [91], która może filtrować oraz przekierowywać żądania użytkowników.
- **Spring Cloud Circuit Breaker** – moduł umożliwiający proste dodanie w ramach aplikacji wzorca wyłącznika obwodu (ang. *circuit breaker*) wraz z wybraną implementacją.

Nowym elementem, nieobecnym w ramach podejścia klasycznego, jest użycie modułu Spring Cloud Function [54]. Projekt pochodzi ze wspominanego Spring Cloud i umożliwia tworzenie łatwych, bezstanowych funkcji. Można je wdrażać na wielu platformach takich jak AWS Lambda [51] czy Google Cloud Functions [92], stąd tworzone funkcje są niezależne od dostawcy usług chmurowych. Dodatkowo moduł umożliwia uruchamianie lokalnie funkcji oraz jej testowanie, co zwiększa niezawodność. Użyta w projekcie wersja Spring Cloud to 2022.0.3 datowana jest na 25 maja 2023 roku [93].

6.4.3 Omówienie implementacji na przykładzie funkcjonalności użycia skróconego linku

Możliwość przekierowania użytkownika do docelowej strony za pomocą krótkiego linku to podstawowa funkcjonalność aplikacji. Z racji uruchomienia większości usług w ramach implementacji jest ona dobrym przykładem do jej opisania. Ścieżka od momentu wysłania żądania przez użytkownika do momentu wywołania implementacji ‘`LinkRepository`’ została już opisana w punkcie **6.3.3**. W tym podrozdziale zostaną opisane jedynie te elementy, które są odmienne od implementacji z wykorzystaniem podejścia klasycznego.

Różnice dotyczące użycia przez użytkownika skróconego linku w implementacji z wykorzystaniem podejścia bezserwerowego:

- Implementacją interfejsu ‘`LinkRepository`’ jest klasa ‘`DynamoDbLinkRepository`’. Opakowuje repozytorium zdefiniowane w interfejs ‘`DynamoDbInternalLinkRepository`’, który umożliwia wykonywanie operacji na tabeli w ramach AWS DynamoDB [37]. Pobieranie linku po krótkim id pokazane jest w Listingu 7.
- Biblioteka do komunikacji z usługą AWS DynamoDB [37] wymaga wcześniejszego założenia tabeli. Za przechowywanie definicji tabeli oraz jej utworzenie odpowiada Terraform [19]. Należy pamiętać o uruchamianiu narzędzia przed wdrażaniem aplikacji na konkretne środowisko lub zautomatyzowaniu tej czynności. Fragment pliku opisujący konfigurację tabeli dla AWS DynamoDB znajduje się w Listingu 8.
- Implementacją interfejsu ‘`LinkActivityMetadataPublisher`’ jest klasa ‘`AwsSqsLinkActivityPublisher`’. Odpowiada ona za wysyłkę wiadomości do usługi bezserwerowej AWS SQS [50], co jest pokazane w Listingu 9.

Listing 7 Fragment klasy DynamoDbLinkRepository odpowiedzialnej za wyszukanie linku. Źródło: Opracowanie własne

```
@Slf4j
@RequiredArgsConstructor
class DynamoDbLinkRepository implements LinkRepository {

    private final DynamoDbInternalLinkRepository dynamoDbInternalLinkRepository;
    private final AwsDynamoDbLinkMapper mapper;

    @Override
    public Optional<Link> findById(LinkId linkId) {
        return dynamoDbInternalLinkRepository.findById(linkId.value())
                .map(mapper::toDomain);
    }
```

Listing 8 Fragment pliku konfiguracyjnego Terraform z definicją tabeli ‘links’ w usłudze AWS DynamoDB. Źródło: Opracowanie własne

```
resource "aws_dynamodb_table" "links_table" {
  name           = "links"
  billing_mode   = "PAY_PER_REQUEST"
  hash_key       = "shortLink"

  attribute {
    name = "shortLink"
    type = "S"
  }
}
```

Listing 9 Klasa AwsSqsLinkActivityPublisher odpowiadająca za publikowanie wiadomości do usługi AWS SQS. Źródło: Opracowanie własne

```
@Component
@Slf4j
@RequiredArgsConstructor
class AwsSqsLinkActivityPublisher implements LinkActivityMetadataPublisher {

    private final QueueMessagingTemplate messagingTemplate;
    private final AwsSqsLinkActivityMetadataMapper mapper;

    @Override
    public void publish(LinkActivityMetadata linkActivityMetadata) {
        var linkActivityMetadataDto = mapper.fromDomain(linkActivityMetadata);

messagingTemplate.convertAndSend(QueueConfiguration.LINK_ACTIVITIES_QUEUE_NAME,
linkActivityMetadataDto);
    }
}
```

Pokazane powyżej implementacje w połączeniu z pozostałymi współdzielonymi modułami aplikacji wypełniają funkcjonalność przekierowania użytkownika za pomocą krótkiego linku.

7. Analiza porównawcza

Rozdział ten zawiera analizę dotyczącą podobieństw i różnic pomiędzy tworzeniem oprogramowania z wykorzystaniem podejścia klasycznego, a tworzeniem oprogramowania z wykorzystaniem usług bezserwerowych. Zebrano w nim różne aspekty takie jak wydajność, sposób utrzymania i monitorowania wdrożonych aplikacji, koszty czy czas potrzebny na wdrożenie i wprowadzanie zmian. Na ich podstawie można określić, które z podejść sprawdzi się lepiej w konkretnym scenariuszu. Pomocne w ocenie jest również to, że w każdym z tych podejść zaimplementowano ten sam zestaw funkcjonalności biznesowych z takim samym kontraktem API [79].

Pierwszy podrozdział skupia się na ogólnych podobieństwach i różnicach pomiędzy dwoma podejściami. Następny zawiera opis podejścia, wykorzystywanych narzędzi oraz wyników przeprowadzonych testów wydajnościowych dla każdej z implementacji. W kolejnym podrozdziale znajduje się porównanie sposobu utrzymania i monitorowania wdrożonych aplikacji. Zebrane są tam również narzędzia oraz opisane spostrzeżenia magistranta dotyczące implementacji. Podrozdział 7.4 zawiera opis kosztów dla każdego z podejść na podstawie przeprowadzonych testów wydajnościowych. Dodatkowo znajduje się tam opis dodatkowych nakładów wymaganych przy tworzeniu zespołu. Ostatni podrozdział zawiera obserwacje dotyczące czasu wdrożenia oraz wprowadzania zmian pomiędzy dwoma podejściami.

Pojawiające się opinie magistranta oparte są na doświadczeniach zawodowych. Żadne z podejść nie było preferowane i starano się w sposób obiektywny przedstawić wady i zalety każdej z metod tak, aby było to jak najbardziej wartościowe dla Czytelnika.

7.1. Podobieństwa i różnice pomiędzy klasycznym podejściem oraz podejściem z wykorzystaniem usług bezserwerowych w tworzeniu oprogramowania

W ramach podrozdziału zawarte jest porównanie ogólnych podobieństw i różnic pomiędzy opisywanymi podejściami. Zebrane zostały one w formie tabelarycznej w celu łatwiejszej analizy.

Tabela 2 zawiera zestawienie cech wspólnych oraz różnic w ramach każdego z podejść. Tworzona była na podstawie doświadczeń zawodowych magistranta, implementacji aplikacji w ramach pracy oraz zebranych informacji na temat każdej z metod wytwarzania oprogramowania.

Tabela 2 Zestawienie najważniejszych podobieństw i różnic pomiędzy podejściem klasycznym, a podejściem z wykorzystaniem usług bezserwerowych. Źródło: opracowanie własne

Cecha	Podejście klasyczne	Podejście z wykorzystaniem usług bezserwerowych
Możliwość uruchomienia usługi lokalnie	Tak	Dla części usług tak, natomiast nie jest to zalecane z uwagi na zmiany w ramach API usług.
Początkowe koszty (np. licencje, serwery, sieć)	Tak	Nie

Stałe koszty	Tak – nawet jeśli aplikacja jest używana rzadko	Nie - w większości usług płacimy tylko wtedy, kiedy ich używamy
Możliwość szybkiego eksperymentowania	Zwykle nie – wymagana jest wiedza do utworzenia usługi	Tak – łatwo utworzyć instancję usługi poprzez GUI[11] na czas eksperymentu, po czym ją usunąć
Możliwość tworzenia rozwiązania bez stałego dostępu do internetu	Tak – można lokalnie utworzyć kontenery z potrzebnymi usługami i tworzyć aplikację	Nie – wymagany dostęp do internetu w celu komunikacji z usługami
Próg wejścia	Niski nawet w przypadku braku doświadczenia z użyciem konkretnych usług	Wysoki z powodu wymaganych dodatkowych elementów (np. polityk bezpieczeństwa, aby móc skomunikować się z usługą; nauka korzystania z interfejsu platformy)
Dokumentacja (czytelność, zastosowanie usługi, przykłady użycia w różnych językach programowania)	Zwykle dobra czytelność, przykłady użycia tylko w najpopularniejszych językach programowania takich jak Java lub Python	Bardzo dobra czytelność dokumentacji - zawiera wyjaśnienia i sytuacje, kiedy wykorzystać usługę, przykłady w wielu językach programowania
Możliwość skorzystania z usług w ramach najbardziej popularnych języków programowania (np. Java, Python)	Tak	Tak
Stopień niezawodności	Średni – zależy od serwerowni, centrum danych. Mają na niego wpływ awarie prądu czy dostawcy internetu.	Wysoki – zwykle usługi bezserwerowe w ramach chmury publicznej są dostępne przez większość czasu. Nie mają na niego wpływu awarie prądu czy dostawcy internetu.
Niezawodność sieci i opóźnienia[12]	Wysoka niezawodność sieci (małe ryzyko zgubienia pakietu) oraz niskie opóźnienia	Niska niezawodność sieci (wysokie ryzyko zgubienia pakietu) oraz wyższe opóźnienia

[11] GUI (ang. *Graphical User Interface*) – oznacza graficzny interfejs użytkownika, który ułatwia interakcję z komputerem. Więcej informacji w [99]

[12] Aspekt ten został opisany w ramach zbioru błędnych założeń podczas budowy systemów rozproszonych o nazwie „*Fallacies of Distributed Computing*". Więcej informacji w [98]

7.2. Porównanie wydajności pomiędzy klasycznym podejściem oraz podejściem z wykorzystaniem usług bezserwerowych

Podrozdział zawiera opis wykonanych testów wydajnościowych, narzędzi użytych do ich wykonania, a także analizę rezultatów. Głównym celem testów jest pokazanie działania aplikacji obsługującej zaplanowany ruch kliencki w zależności od implementacji.

W celu przeprowadzenia testów wydajnościowych należy użyć specjalistycznych narzędzi, które będą mierzyć ilość żądań na sekundę czy średnie czasy odpowiedzi usługi. Część z nich umożliwia zaawansowaną parametryzację zapytań, opóźnianie ich wysyłki oraz zmianę ilości wysyłanych żądań w czasie. Narzędziem użytym do testów wydajnościowych w ramach pracy jest Gatling [72], które umożliwia pisanie scenariuszy testowych w języku Java. Niewątpliwą zaletą jest składnia ułatwiająca tworzenie testów oraz ich zrozumienie dla nietechnicznych osób.

Stworzony scenariusz testowy został przygotowany w sposób pozwalający sprawdzić przykładowe zachowanie użytkownika REST API. Czas trwania testu został podzielony na dwie fazy:

1. Wysyłanie niewielkiej ilości żądań, aby zasymulować niewielki ruch kliencki. Czas trwania to 1 minuta.
2. Wysyłanie docelowej ilości żądań, które mają sprawdzić zachowanie serwisu pod obciążeniem. Długość tego etapu to 59 minut.

Opisywany scenariusz zawiera następujące kroki:

1. Wygenerowanie tokenu dostępowego dla aplikacji (jednorazowo przed rozpoczęciem wszystkich testów).
2. Utworzenie krótkiego linku na podstawie predefiniowanego żądania i sprawdzenie poprawności zwróconej odpowiedzi.
3. Użycie wygenerowanego krótkiego linku do przekierowania w celu wygenerowania danych dotyczących aktywności. Co ważne, wyłączone jest podążanie za przekierowaniem w ramach testu z powodu możliwego wpływu na rezultaty.
4. Pobranie danych dotyczących aktywności dla wygenerowanego krótkiego linku i sprawdzenie poprawności zwróconej odpowiedzi.

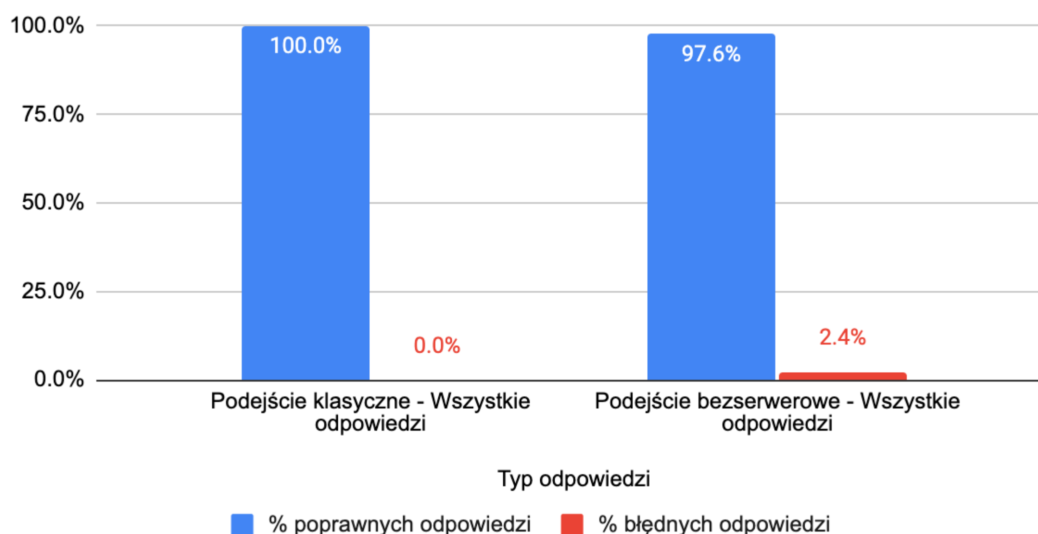
Przeprowadzone testy wydajnościowe pokazały różnice w zachowaniu obu implementacji. Pierwszą z różnic jest procent poprawnych i błędnych odpowiedzi. Dla podejścia klasycznego odpowiedzi oczekiwane stanowią 100%, natomiast w drugim przypadku 97,6%. Określono kilka powodów błędnych odpowiedzi (stanowiących 2,4% wszystkich żądań użytkowników) dla podejścia wykorzystującego usługi bezserwerowe. Głównym była wydłużona ścieżka dla każdego z żądań użytkowników, co powodowało zwiększone czasy oczekiwania i tym samym większe obciążenie procesora i pamięci. Dodatkowo część z żądań wysyłanych do usług zewnętrznych kończyła się niepowodzeniem i nie była ponawiana. Inny typ błędów to osiągnięcie limitu wysyłanych żądań do usługi AWS DynamoDB [37] w ramach konta. Limity te można zwiększyć, natomiast w pewnych przypadkach, z powodu współdzielenia zasobów z innymi użytkownikami, nie jest to możliwe. Przykład takiej wiadomości jest pokazany w Listingu 10. Porównanie wartości procentowych jest zaprezentowane na wykresie Rysunek 6.

Dodanie w ramach aplikacji wzorców takich jak ponawianie operacji (ang. retry) [94] oraz użycie wyłącznika obwodu (ang. circuit breaker) [95] pozwala minimalizować wpływ zewnętrznych usług na naszą aplikację. Wspomiane wzorce pomogą w przypadku, gdy dostawca usługi bezserwerowej zwraca komunikat o chwilowej niedostępności usługi. Wówczas można ponowić żądanie z lekkim opóźnieniem. W przypadku awarii usługi firm trzecich wyłącznik obwodu wykryje ją i będzie wykonywał rezerwową akcję (ang. fallback). Ma to na celu oszczędzenie zasobów dla pozostałych

żądań poprzez niewykonywanie skazanych na porażkę akcji. Dodatkowo daje czas na przywrócenie do prawidłowego działania zewnętrznej usługi.

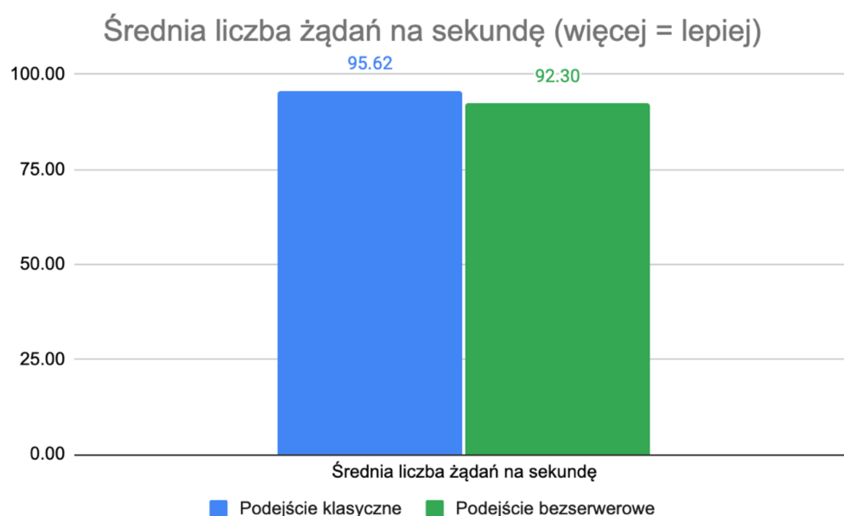
Listing 10 Błąd przekroczenia ilości żądań dla usługi AWS DynamoDB. Źródło: opracowanie własne

```
com.amazonaws.services.dynamodbv2.model.RequestLimitExceededException: Throughput
exceeds the current throughput limit for your account. Please contact AWS Support
at https://aws.amazon.com/support request a limit increase (Service:
AmazonDynamoDBv2; Status Code: 400; Error Code: RequestLimitExceeded; Request ID:
B5K796LDL1A4II29C499O07USNVV4KQNSO5AEMVJF66Q9ASUAAJG; Proxy: null)
    at
com.amazonaws.http.AmazonHttpClient$RequestExecutor.handleErrorResponse(AmazonHttpC
lient.java:1819) ~[aws-java-sdk-core-1.11.951.jar!/:na]
```



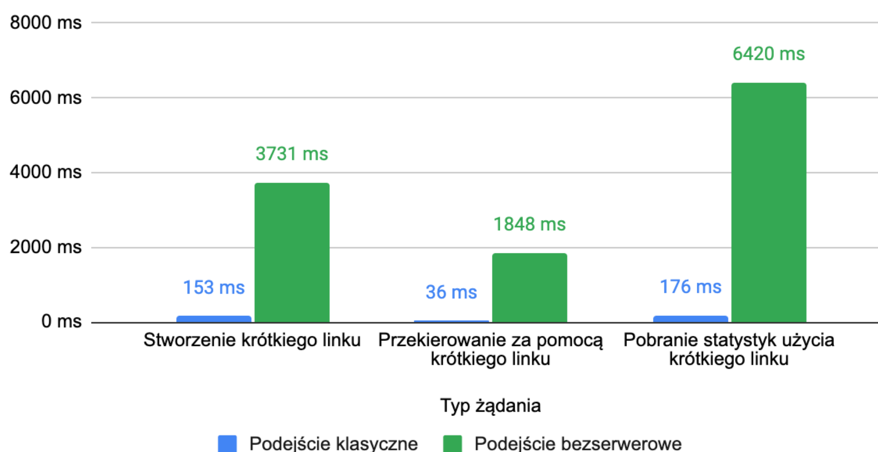
Rysunek 6 Wykres: Testy wydajnościowe - procentowy udział poprawnych oraz błędnych odpowiedzi. Źródło: Opracowanie własne

Kolejnym ważnym aspektem dla aplikacji internetowych jest wydajność mierzona w liczbie żądań na sekundę (ang. *RPS – Requests Per Second*). Im większa jest ta wartość, tym więcej akcji użytkownika można wykonać jednocześnie. Ma to również przełożenie na ilość instancji aplikacji oraz realne koszty jej działania. Istnieją implementacje aplikacji, gdzie jedna wydajna implementacja obsługuje wszystkie żądania użytkowników, a mniej wydajna wersja wymaga uruchomienia dwóch instancji. W przypadku stworzonej aplikacji - wartości dla obu implementacji są podobne z nieznaczną przewagą podejścia klasycznego. Ma to związek z wydłużoną ścieżką, jaką musi pokonać każde wywołanie, do zewnętrznych usług w ramach podejścia bezserwerowego. Porównanie średniej ilości żądań znajduje się na wykresie Rysunek 7.



Rysunek 7 Wykres: Testy wydajnościowe - porównanie średniej liczby żądań na sekundę. Źródło: Opracowanie własne

Średni czas odpowiedzi na żądanie to kolejny czynnik określający wydajność. Jego wartość ma szczególne znaczenie w aplikacjach, na których polegają inne serwisy lub użytkownik oczekuje natychmiastowej odpowiedzi. Zebrane podczas testów dane pokazują, że w takich przypadkach aplikacje tworzone w sposób klasyczny mają dużą przewagę w postaci braku dodatkowych opóźnień. Średnie czasy odpowiedzi dla aplikacji tworzonych w podejściu bezserwerowym są wysokie. Składa się na to kilka elementów. Po pierwsze, dla tworzenia krótkiego linku oraz pobierania dla niego statystyk następuje weryfikacja tokenu dostępu do API. Wymusza to dodatkową komunikację z usługą, która generuje i weryfikuje dostępowy token. Po drugie na wyższą średnią wpływa dłuższa obsługa każdego z żądań, co powoduje utworzenie kolejki w oczekiwaniu na przetworzenie. Pobyt w kolejce akcji wysłanej przez użytkownika REST API zwiększa czas przetworzenia. Czas ten maksymalnie może wynieść 60 sekund. Dla części wysłanych zapytań czas 60 sekund został osiągnięty co powodowało zamknięcie takiego połączenia po stronie aplikacji. Porównanie średniej czasów obsługi żądań znajduje się na wykresie Rysunek 8.



Rysunek 8 Wykres: Testy wydajnościowe - porównanie średniego czasu odpowiedzi dla różnych funkcjonalności. Źródło: Opracowanie własne

Zebrane dane pokazują pewne różnice pomiędzy dwoma podejściami i pozwalają na wyciągnięcie ogólnych wniosków. W przypadku, gdy użytkownicy aplikacji wymagają minimalnych opóźnień, lepiej sprawdzi się korzystanie z klasycznego podejścia. Przykładem takich rozwiązań są systemy tradingowe lub skomplikowane systemy finansowe. Dodatkowy narzut związany z komunikacją z usługami bezserwerowymi jest widoczny i znaczny, co obrazuje wykres na Rysunku 8. Jeśli nie ma takiego wymagania lub też narzut związany z opóźnieniami jest mniej ważny niż koszty utrzymania infrastruktury, wówczas można skorzystać z podejścia z wykorzystaniem usług bezserwerowych.

7.3. Porównanie utrzymania i monitorowania wdrożonych aplikacji pomiędzy klasycznym podejściem oraz podejściem z wykorzystaniem usług bezserwerowych

W podrozdziale zawarte są obserwacje i wnioski, płynące z porównania utrzymania oraz monitorowania wdrożonych aplikacji pomiędzy omawianymi podejściami.

Tabela 3 prezentuje podobieństwa i różnice w sposobie sprawdzania stanu aplikacji oraz jej utrzymywania w odpowiedniej kondycji. Zawiera także aspekty dotyczące metryk aplikacyjnych, a także powiązanych usług czy zmian na poziomie infrastruktury.

Tabela 3 Zestawienie najważniejszych podobieństw i różnic w utrzymaniu i monitorowaniu aplikacji wdrożonych w ramach podejścia klasycznego oraz podejścia z wykorzystaniem usług bezserwerowych. Źródło: Opracowanie własne

Omawiany element monitorowania i utrzymania aplikacji	Podejście klasyczne	Podejście z wykorzystaniem usług bezserwerowych
Zbieranie danych dotyczących użycia procesora, pamięci, dysku dla klastra Kubernetes [16]	Dostarczone w większości instalacji klastra	Dostarczone w większości instalacji klastra
Zbieranie metryk i monitorowanie usług zależnych (np. baza danych, broker wiadomości)	Potrzeba ręcznego skonfigurowania zbierania metryk (np. Prometheus [25]) oraz stworzenia ekranów z podsumowaniami (np. Grafana [24])	Dostarczone w ramach rozwiązania
Początkowa konfiguracja usługi (np. baza danych, broker wiadomości) jest poprawna dla większości przypadków	Zwykle tak	Zwykle tak
Możliwość zmiany wszystkich parametrów konfiguracyjnych usługi	Tak	Nie – udostępnienie do zmiany wybranych parametrów konkretnej usługi

Możliwość ustawienia notyfikacji (np. SMS, email) w przypadku awarii	Wymagana ręczna konfiguracja oraz integracja pomiędzy kilkoma usługami	Tak
Automatyczne wgrywanie poprawek bezpieczeństwa oraz nowych wersji	Nie. Konfiguracja możliwa poprzez dodatkowe oprogramowanie.	Tak
Wymuszanie przejścia na nowsze wersje po pewnym okresie	Nie	Tak
Potrzeba przeprowadzania prac administracyjnych (np. kopie zapasowe, czyszczenie katalogów z plikami tymczasowymi)	Tak	Nie
Potrzeba wymiany i rozbudowy elementów infrastruktury (np. dyski, dodawanie pamięci RAM dla serwerów)	Tak	Nie
Wsparcie pomocy technicznej w przypadku problemów z usługą	Zwykle nie. Uzyskanie wsparcia możliwe po zakupie odpowiedniej licencji.	Tak
Możliwość użycia podejścia *Infrastruktura jako Kod* w celu wersjonowania i śledzenia zmian w komponentach	Nie	Tak. Dla większości usług bezserwerowych oferowanych w ramach chmury publicznej
Możliwość monitorowania całkowitych kosztów konkretnej aplikacji i wymaganej infrastruktury	Nie	Tak, ale należy wdrożyć kilka zmian w ramach infrastruktury (m.in. oznaczanie zasobów przynależnych do aplikacji)

Powyższe zestawienie pokazuje, że w kontekście monitorowania oraz utrzymania, mniej pracochłonnym oraz oferującym wiele rzeczy „na start" okazuje się wykorzystanie usług bezserwerowych. Wpływa to na szybkość dostarczania oraz prototypowania nowych rozwiązań. Należy również pamiętać, że w przypadku specyficznych wymagań dotyczących aplikacji (np. konkretna wersja bazy danych) bezserwerowa usługa bazy danych może jej nie wspierać. Wówczas należy skorzystać z klasycznego podejścia i własnymi siłami utworzyć potrzebną instancję.

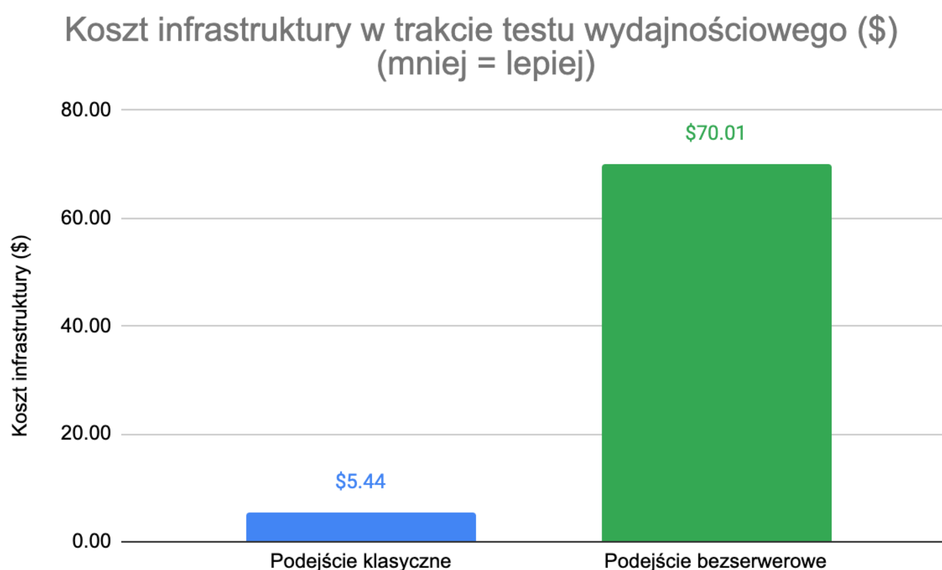
7.4. Porównanie kosztów pomiędzy klasycznym podejściem oraz podejściem z wykorzystaniem usług bezserwerowych

W podrozdziale zawarte jest porównanie kosztów pomiędzy klasycznym podejściem, a podejściem z wykorzystaniem usług bezserwerowych. Zostaną do tego użyte dane zebrane podczas testów wydajnościowych. Dodatkowo analizie poddano kilka typów kosztów w ramach tworzenia oprogramowania.

W ramach wykonywania testów wydajnościowych zebrano dane dotyczące zużycia zasobów dla każdego z podejść. Ruch wygenerowany w ramach testu wydajnościowego dla każdego z podejść prezentuje Tabela 4. Natomiast porównanie całkowitego kosztu przeprowadzonych prób znajduje się na Rysunku 9.

Tabela 4 Podsumowanie liczby żądań w ramach testów wydajnościowych. Źródło: Opracowanie własne

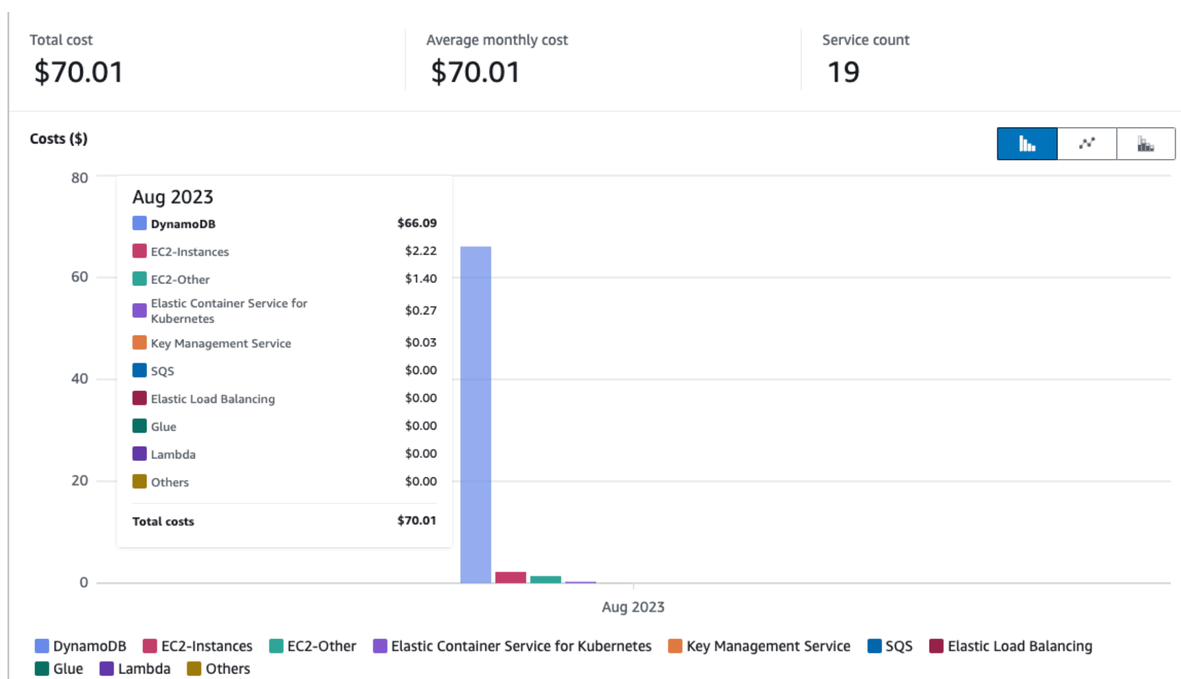
Typ żądania	Liczba żądań dla podejścia klasycznego	Liczba żądań dla podejścia bezserwerowego
Stworzenie krótkiego linku	115100	115100
Przekierowanie za pomocą krótkiego linku	115100	115100
Pobranie statystyk użycia krótkiego linku	115100	115100



Rysunek 9 Wykres: Koszt infrastruktury w trakcie testu wydajnościowego ($)**.** Źródło: Opracowanie własne

Początkowo można przypuszczać, że proporcje powinny być odwrotne. Nic bardziej mylnego! Wyjaśnienie wysokich kosztów dla podejścia bezserwerowego znajduje się w rozbiciu na poszczególne usługi. Jak można zobaczyć na wykresie zaprezentowanym na Rysunku 10 – ponad 94% wszystkich kosztów stanowi usługa AWS DynamoDB [37]. Usługa ta posiada dwie możliwości konfiguracji:

zadeklarowanie konkretnego użycia (niższe koszty) lub płacenie za to, co się zużyło (wyższe koszty). W przypadku implementacji została użyta druga opcja, nieposiadająca darmowych zasobów do wykorzystania. Mogło to wpłynąć na koszt korzystania z usługi.



Rysunek 10 Wykres prezentujący koszty w rozbiciu na usługi AWS podczas wykonywania testu wydajnościowego z wykorzystaniem usług bezserwerowych. Źródło: opracowanie własne

W ramach podsumowania warto również wspomnieć o dodatkowych kosztach dla każdego z podejść. W przypadku podejścia klasycznego już przed rozpoczęciem prac należy się liczyć z kosztami. Najważniejsze z nich to zakup sprzętu oraz licencji, a także zapewnienie odpowiedniego miejsca oraz specjalistów mogących odpowiednio skonfigurować i monitorować infrastrukturę. Dalsze koszty stałe w trakcie prac ograniczają się do rzadkiej modernizacji zakupionego sprzętu oraz kosztów wynagrodzeń dla personelu. W przypadku podejścia z wykorzystaniem usług bezserwerowych nie ponosi się na początku kosztów związanych z infrastrukturą. Część z usług posiada również darmowe limity możliwe do wykorzystania, które amortyzują początkowe korzystanie z usług [96]. Należy pamiętać o specjalistach. Zwykle potrzebni są programiści posiadający wiedzę z zakresu tworzenia aplikacji z wykorzystaniem podejścia bezserwerowego, a także mający doświadczenie z chmurą publiczną. Ich wynagrodzenie jest wyższe i są oni rzadziej spotykani na rynku pracy niż programiści z doświadczeniem w podejściu klasycznym.

Zebrane informacje pokazują, że nie ma tu „jednoznacznego zwycięzcy”. W przypadku firm, które nie mogą sobie pozwolić na wysokie koszty początkowe (np. startupy), naturalnym wyborem jest wykorzystanie chmury publicznej i oferowanych tam usług bezserwerowych. Oferują one również możliwość monitorowania całkowitych kosztów w rozbiciu na pojedyncze aplikacje. Z kolei część firm widzi w podejściu klasycznym większe bezpieczeństwo i niezawodność, pomimo początkowo wyższych kosztów.

7.5. Porównanie czasu wdrożenia oraz wprowadzania zmian pomiędzy klasycznym podejściem oraz podejściem z wykorzystaniem usług bezserwerowych

Podrozdział zawiera porównanie czasu wdrożenia oraz wprowadzania zmian w projekcie pomiędzy analizowanymi metodami. Na jego zawartość składają się doświadczenia zawodowe magistranta, a także wiedza nabyta podczas dwóch implementacji aplikacji będącej częścią pracy.

Dla podejścia klasycznego najbardziej czasochłonnym elementem jest tworzenie wymaganej infrastruktury oraz zakupy licencji. Warto również zwrócić uwagę na konfigurację sieciową, nadanie dostępów do zasobów czy zaktualizowanie dokumentacji projektowej. Zwykle część rzeczy jest możliwa do automatyzacji np. przygotowanie serwera, instalacja bazy danych. Pozostałe działania muszą być zrealizowane manualnie przez specjalistów. W przypadku wprowadzania zmian czas ten jest zdecydowanie krótszy, ale nadal wymaga w niektórych sytuacjach długiej ścieżki decyzyjnej do analizy i zatwierdzenia zmiany. Nierzadko wymagane jest zbieranie danych audytowych dotyczących zmian w ramach infrastruktury. W przypadku klasycznego podejścia źródłem takiej wiedzy jest dokumentacja oraz system obsługi zgłoszeń. Oba te źródła mogą stanowić bazę wiedzy na temat zmian w infrastrukturze, o ile są one używane i aktualizowane.

W przypadku podejścia z wykorzystaniem usług bezserwerowych większość elementów infrastruktury jest możliwa do utworzenia i zarządzania z poziomu kodu. Utworzenie środowiska odbywa się w sposób deklaratywny tj. programista definiuje stan i parametry usługi w pliku konfiguracyjnym. Następnie narzędzie takie jak Terraform [19] dba o wykonanie pliku. Zwykle utworzenie nowej bazy danych nie zajmuje więcej niż 20 sekund, włącznie z politykami bezpieczeństwa oraz zdefiniowaniem ustawień kopii zapasowych. Na potrzeby przeprowadzenia testów wydajnościowych opisanych w podrozdziale 7.2 utworzenie całego środowiska testowego zajmowało średnio 15 minut. W przypadku wprowadzania zmian czas jest równie krótki – zwykle ogranicza się do zmiany w pliku konfiguracyjnym oraz dodaniu go do repozytorium. Po zatwierdzeniu zmiany i dodaniu jej do głównej gałęzi można skonfigurować automatyczny proces, który zaktualizuje infrastrukturę środowiska. Takie podejście pomaga w audytowaniu zmian wdrażanych w ramach infrastruktury aplikacji oraz jej wersjonowaniu. Umożliwia dokładne odtworzenie środowiska dla konkretnej wersji aplikacji w celu analizy błędu.

Na podstawie zebranych danych szybszym rozwiązaniem okazuje się podejście z wykorzystaniem usług bezserwerowych. Dodatkowo pozwala ono zaoszczędzić czas przy audytowaniu zmian w infrastrukturze czy tworzeniu nowego środowiska dla aplikacji.

Podsumowanie

Niniejsza praca magisterska, zgodnie z tytułem, została poświęcona analizie porównawczej wytwarzania oprogramowania w sposób klasyczny oraz przy wykorzystaniu usług bezserwerowych. Została opracowana na podstawie danych z literatury oraz źródeł dostępnych w Internecie. Dla każdego z podejść opisano wyzwania z nimi związane, możliwe do wykorzystania narzędzia, a także model kosztowy. Poruszono zagadnienia dotyczące współczesnego procesu wytwarzania oprogramowania, konteneryzacji oraz rodzajów chmur. Na potrzeby analizy porównawczej utworzono w języku Java dwie implementacje aplikacji internetowej, po jednej dla każdej z analizowanych metod. Wykorzystano framework Spring do budowy rozwiązań. Tak utworzone aplikacje dostarczyły materiału do opisania i porównania wydajności, kosztów, czasu potrzebnego na wdrożenie i wprowadzenie zmian oraz sposobu utrzymania i monitorowania wdrożonych rozwiązań.

W ramach pracy zostały przedstawione podobieństwa i różnice dla oprogramowania wytwarzanego w podejściu klasycznym i z wykorzystaniem usług bezserwerowych. Autor wierzy, że praca zawiera wartościowe informacje przydatne przy wyborze sposobu tworzenia nowego oprogramowania oraz jest cennym źródłem wiedzy na temat kosztów, wydajności oraz innych aspektów każdego z podejść.

Prace cytowane

[1] Strona główna Amazon Web Services (AWS): https://aws.amazon.com/, data dostępu: 22.10.2023.

[2] Strona opisująca komunikator Messenger: https://about.meta.com/technologies/messenger/, data dostępu: 6.1.2024.

[3] Strona główna systemu Windows: https://www.microsoft.com/en-us/windows, data dostępu: 6.1.2024.

[4] Strona główna programu Paint: https://www.microsoft.com/en-us/windows/paint, data dostępu: 6.1.2024.

[5] Strona główna narzędzia Git: https://git-scm.com/, data dostępu: 6.1.2024.

[6] S. Newman, „Budowanie mikrousług. Projektowanie drobnoziarnistych systemów", strona 243, 2022.

[7] M. Raheem, Praca magisterska "Implementing a Secured Container Workload in the Cloud": https://www.theseus.fi/bitstream/handle/10024/406583/muftau_raheem.pdf, data dostępu: 23.9.2023.

[8] Strona główna systemu operacyjnego Linux Ubuntu: https://ubuntu.com/, data dostępu: 6.1.2024.

[9] A. Mateos and J. Rosenberg, "Chmura obliczeniowa. Rozwiązania dla biznesu", strona 43, 2011.

[10] Strona główna Google Cloud Platform (GCP): https://cloud.google.com/, data dostępu: 6.1.2024.

[11] Strona główna Microsoft Azure: https://azure.microsoft.com/en-us, data dostępu: 6.1.2024.

[12] Strona główna programu 'AWS for Startups': https://aws.amazon.com/startups/credits, data dostępu: 6.1.2024.

[13] Strona programu 'Google for Startups Cloud Program': https://cloud.google.com/startup, data dostępu: 6.1.2024.

[14] Strona programu 'Microsoft for Startups': https://www.microsoft.com/en-us/startups, data dostępu: 6.1.2024.

[15] B. Burns, J. Beda and K. Hightower, "Kubernetes. Up & running. 2nd edition", strona 56, 2019.

[16] Strona Kubernetes: https://kubernetes.io/, data dostępu: 22.10.2023.

[17] Dokumentacja narzędzia GitHub Actions: https://docs.github.com/en/actions, data dostępu: 22.10.2023.

[18] Strona ArgoCD: https://argo-cd.readthedocs.io/en/stable/, data dostępu: 05.11.2023.

[19] Strona Terraform: https://www.terraform.io/, data dostępu: 22.10.2023.

[20] Strona SonarQube: https://www.sonarsource.com/products/sonarqube/, data dostępu: 22.10.2023.

[21] Strona GitHub: https://github.com/, data dostępu: 22.10.2023.

[22] Strona GitLab: https://about.gitlab.com/, data dostępu: 22.10.2023.

[23] J. Schneider, "SRE with Java Microservices", strona 66, 2020.

[24] Strona Grafana: https://grafana.com/, data dostępu: 22.10.2023.

[25] Strona Prometheus: https://prometheus.io/, data dostępu: 22.10.2023.

[26] Artykuł opisujący różnice w sposobie tworzenia kopii zapasowych pomiędzy fizycznymi i wirtualnymi maszynami: https://www.veeam.com/blog/why-virtual-machine-backups-different.html, data dostępu: 12.10.2023.

[27] A. Mateos and J. Rosenberg, "Chmura obliczeniowa. Rozwiązania dla biznesu", strony 31-38, 2011.

[28] S. Jourdan and P. Pomès, "Infrastructure as Code (IAC) Cookbook.", strona 40, 2017.

[29] K. Morris, "Infrastructure as Code. 2nd edition", strona 38, 2021.

[30] Strona Keycloak: https://www.keycloak.org/, data dostępu: 22.10.2023.

[31] Strona Ansible: https://www.ansible.com/, data dostępu: 22.10.2023.

[32] Strona projektu OpenShift: https://www.redhat.com/en/technologies/cloud-computing/openshift, data dostępu: 22.10.2023.

[33] Strona opisująca Facebook: https://about.meta.com/technologies/facebook-app/, data dostępu: 6.1.2024.

[34] Strona główna Google: https://www.google.com/, data dostępu: 6.1.2024.

[35] Strona poświęcona PromQL: https://prometheus.io/docs/prometheus/latest/querying/basics/, data dostępu: 6.1.2024.

[36] E. Jonas and J. Schleier-Smith, Cloud Programming Simplified: A Berkeley View on Serverless: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf, data dostępu: 15.10.2023.

[37] Strona usługi AWS DynamoDB: https://aws.amazon.com/dynamodb/, data dostępu: 05.11.2023.

[38] Strona usługi AWS Cognito: https://aws.amazon.com/cognito/, data dostępu: 05.11.2023.

[39] N. Dabit, „Full Stack Serverless”, strony 33-41, 2020.

[40] Strona Microsoft365: https://www.microsoft.com/pl-pl/microsoft-365, data dostępu: 05.11.2023.

[41] Strona PostgreSQL: https://www.postgresql.org/, data dostępu: 05.11.2023.

[42] Strona Redis: https://redis.io/, data dostępu: 05.11.2023.

[43] Strona projektu localstack: https://github.com/localstack/localstack, data dostępu: 05.11.2023.

[44] N. Dabit, "Full Stack Serverless", strony 46-66, 2020.

[45] Strona Pulumi: https://www.pulumi.com/product/, data dostępu: 05.11.2023.

[46] Strona frameworka AWS Amplify: https://aws.amazon.com/amplify/, data dostępu: 05.11.2023.

[47] Strona ngrok: https://ngrok.com/, data dostępu: 05.11.2023.

[48] Strona główna Redis: https://redis.io/, data dostępu: 6.1.2024.

[49] Strona brokera wiadomości RabbitMQ: https://www.rabbitmq.com/, data dostępu: 05.11.2023.

[50] Strona usługi AWS SQS: https://aws.amazon.com/sqs/, data dostępu: 05.11.2023.

[51] Strona usługi AWS Lambda: https://aws.amazon.com/lambda/, data dostępu: 05.11.2023.

[52] Strona projektu Spring Cloud AWS: https://spring.io/projects/spring-cloud-aws, data dostępu: 19.11.2023.

[53] Strona frameworka Spring: https://spring.io/, data dostępu: 19.11.2023.

[54] Strona Spring Cloud Functions: https://docs.spring.io/spring-cloud-function/docs/current/reference/html/spring-cloud-function.html, data dostępu: 31.12.2023.

[55] Strona projektu AWS DynamoDB local: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBLocal.html, data dostępu: 19.11.2023.

[56] Strona języka programowania Java: https://www.java.com/pl/download/help/whatis_java.html, data dostępu: 19.11.2023.

[57] Strona języka programowania Python: https://www.python.org/, data dostępu: 19.11.2023.

[58] Strona główna projektu Bash: https://www.gnu.org/software/bash/, data dostępu: 6.1.2024.

[59] Wikipedia - Artykuł 'Linux': https://pl.wikipedia.org/wiki/Linux, data dostępu: 19.11.2023.

[60] Dokumentacja modułu Spring Boot Web: https://docs.spring.io/spring-boot/docs/current/reference/html/web.html, data dostępu: 19.11.2023.

[61] Strona projektu Spring Data: https://spring.io/projects/spring-data, data dostępu: 19.11.2023.

[62] Strona projektu Spring Security: https://spring.io/projects/spring-security, data dostępu: 19.11.2023.

[63] Strona biblioteki Lombok: https://projectlombok.org/, data dostępu: 19.11.2023.

[64] Strona biblioteki Immutables: https://immutables.github.io/, data dostępu: 19.11.2023.

[65] Strona biblioteki YAUAA: https://yauaa.basjes.nl/, data dostępu: 19.11.2023.

[66] Wikipedia - Artykuł 'User Agent': https://pl.wikipedia.org/wiki/User_agent, data dostępu: 31.12.2023.

[67] Strona narzędzia Mapstruct: https://mapstruct.org/, data dostępu: 19.11.2023.

[68] Strona TestContainers: https://testcontainers.com/, data dostępu: 19.11.2023.

[69] Strona JUnit: https://junit.org/junit5/, data dostępu: 19.11.2023.

[70] Strona RESTAssured: https://rest-assured.io/, data dostępu: 19.11.2023.

[71] Strona biblioteki AssertJ: https://joel-costigliola.github.io/assertj/, data dostępu: 19.11.2023.

[72] Strona główna narzędzia Gatling: https://gatling.io/, data dostępu: 19.11.2023.

[73] Strona narzędzia Docker Compose: https://docs.docker.com/compose/, data dostępu: 19.11.2023.

[74] J. Nickoloff, "Docker In Action", Manning Publications Co, strona 232, 2016.

[75] Strona narzędzia Helm: https://helm.sh/, data dostępu: 19.11.2023.

[76] Strona usługi AWS ELB: https://aws.amazon.com/elasticloadbalancing/, data dostępu: 19.11.2023.

[77] Strona usługi AWS CloudWatch: https://aws.amazon.com/cloudwatch/, data dostępu: 19.11.2023.

[78] Strona usługi AWS EKS: https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html, data dostępu: 19.11.2023.

[79] Artykuł opisujący kontrakty API oraz podejście 'Consumer Driven Contracts': https://softwareskill.pl/consumer-driven-contract, data dostępu: 22.10.2023.

[80] Dokumentacja opisująca moduł Kubernetes Ingress: https://kubernetes.io/docs/concepts/services-networking/ingress/, data dostępu: 31.12.2023.

[81] Strona protokołu OAuth: https://oauth.net/, data dostępu: 05.11.2023.

[82] Strona języka programowania Kotlin: https://kotlinlang.org/, data dostępu: 31.12.2023.

[83] Strona języka programowania Groovy: https://groovy-lang.org/, data dostępu: 31.12.2023.

[84] Artykuł opisujący mechanizm 'Dependency Injection' w Spring: https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html, data dostępu: 31.12.2023.

[85] Oficjalna dokumentacja frameworka Spring: https://docs.spring.io/spring-framework/docs/4.3.x/spring-framework-reference/html/overview.html, data dostępu: 31.12.2023.

[86] Strona poświęcona wydaniu Spring w wersji 6.0.5: https://spring.io/blog/2023/02/15/spring-framework-6-0-5-available-now/, data dostępu: 31.12.2023.

[87] Strona poświęcona wydaniu Spring Boot w wersji 3.0.3: https://spring.io/blog/2023/02/23/spring-boot-3-0-3-available-now/, data dostępu: 31.12.2023.

[88] Strona Spring Initializr: https://start.spring.io/, data dostępu: 31.12.2023.

[89] Strona Spring Cloud: https://spring.io/projects/spring-cloud/, data dostępu: 31.12.2023.

[90] Strona główna HashiCorp Vault: https://www.vaultproject.io/, data dostępu: 31.12.2023.

[91] Opis wzorca API Gateway: https://microservices.io/patterns/apigateway.html, data dostępu: 6.1.2024.

[92] Strona usługi Google Cloud Functions: https://cloud.google.com/functions, data dostępu: 31.12.2023.

[93] Strona poświęcona wydaniu Spring Cloud w wersji 2022.0.3: https://spring.io/blog/2023/05/25/spring-cloud-2022-0-3-aka-kilburn-is-available/, data dostępu: 31.12.2023.

[94] Strona opisująca wzorzec architektoniczny 'retry': https://learn.microsoft.com/en-us/azure/architecture/patterns/retry, data dostępu: 6.1.2024.

[95] Strona opisująca wzorzec architektoniczny 'circuit breaker': https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker, data dostępu: 6.1.2024.

[96] Strona zawierająca wypis darmowych usług i limitów dla AWS: https://aws.amazon.com/free/, data dostępu: 2.1.2024.

[97] Wikipedia - Artykuł 'Otwarte oprogramowanie': https://pl.wikipedia.org/wiki/Otwarte_oprogramowanie, data dostępu: 6.1.2024.

[98] Wikipedia - Artykuł 'Fallacies of Distributed Computing': https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing, data dostępu: 1.1.2024.

[99] Wikipedia - Artykuł 'GUI': https://pl.wikipedia.org/wiki/Graficzny_interfejs_u%C5%BCytkownika, data dostępu: 1.1.2024.

[100] Artykuł opisujący kody odpowiedzi HTTP: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status, data dostępu: 31.12.2023.

[101] Wikipedia - Artykuł 'HTTP': https://pl.wikipedia.org/wiki/Hypertext_Transfer_Protocol, data dostępu: 31.12.2023.

[102] Strona opisująca system macOS oraz jego wersję Sonoma: https://www.apple.com/macos/sonoma/, data dostępu: 6.1.2024.

Spis rysunków

Rysunek 1 Porównanie infrastruktury fizycznej i wirtualnej 13
Rysunek 2 Wysokopoziomowy diagram implementacji w podejściu klasycznym 28
Rysunek 3 Zbiór komponentów frameworka Spring 29
Rysunek 4 Tworzenie szkieletu aplikacji shorter przy wykorzystaniu narzędzia Spring Initializr 30
Rysunek 5 Wysokopoziomowy diagram implementacji w podejściu bezserwerowym 34
Rysunek 6 Wykres: Testy wydajnościowe - procentowy udział poprawnych oraz błędnych odpowiedzi 40
Rysunek 7 Wykres: Testy wydajnościowe - porównanie średniej liczby żądań na sekundę 41
Rysunek 8 Wykres: Testy wydajnościowe - porównanie średniego czasu odpowiedzi dla różnych funkcjonalności 41
Rysunek 9 Wykres: Koszt infrastruktury w trakcie testu wydajnościowego ($) 44
Rysunek 10 Wykres prezentujący koszty w rozbiciu na usługi AWS podczas wykonywania testu wydajnościowego z wykorzystaniem usług bezserwerowych 45

Spis tabel

Tabela 1 Spis wymagań dla implementacji projektu 24
Tabela 2 Zestawienie najważniejszych podobieństw i różnic pomiędzy podejściem klasycznym, a podejściem z wykorzystaniem usług bezserwerowych...37
Tabela 3 Zestawienie najważniejszych podobieństw i różnic pomiędzy w utrzymaniu i monitorowaniu aplikacji wdrożonych w ramach podejścia klasycznego oraz podejścia z wykorzystaniem usług bezserwerowych..42
Tabela 4 Podsumowanie liczby żądań w ramach testów wydajnościowych 44

Spis listingów

Listing 1 Fragment klasy LinkEndpoint odpowiedzialny za przekierowywanie użytkowników..........31
Listing 2 Fragment klasy LinkApiService odpowiedzialny za przekierowywanie użytkowników.......32
Listing 3 Fragment klasy LinkService odpowiedzialny za pobranie linku po krótkim id. 32
Listing 4 Fragment klasy RedisLinkRepository odpowiedzialnej za wyszukanie linku. 32
Listing 5 Klasa RabbitMqLinkActivityMetadataPublisher odpowiadająca za publikowanie wiadomości na kolejkę RabbitMQ...33
Listing 6 Metoda odpowiedzialna za stworzenie odpowiedzi dla użytkownika. 33
Listing 7 Fragment klasy DynamoDbLinkRepository odpowiedzialnej za wyszukanie linku. 36
Listing 8 Fragment pliku konfiguracyjnego Terraform z definicją tabeli 'links' w usłudze AWS DynamoDB. 36
Listing 9 Klasa AwsSqsLinkActivityPublisher odpowiadająca za publikowanie wiadomości do usługi AWS SQS...36
Listing 10 Błąd przekroczenia ilości żądań dla usługi AWS DynamoDB. 40