SERVERLESS

w 5 minut

PWN

Wstęp str. 3

Model cenowy usług serverless str. 5

Wpływ modelu płatności na architekturę aplikacji oraz wdrożenia str. 7

Wpływ modelu płatności na bezpieczeństwo str. 9

Wpływ modelu płatności na decyzje dotyczące rozwoju produktu str. 9

Ograniczenia techniczne na platformie AWS Lambda str. 11

Brak sesji użytkowników str. 11

Niedeterministyczne opóźnienia str. 12

Ograniczony czas wykonania str. 13

Brak pełnej kontroli nad przydzielaniem mocy obliczeniowej str. 14

Kiedy używać Lambdy? str. 15

Interesujące kwestie str. 16

Serverless w 5 minut

AWS Lambda jest usługą dostępną na platformie Amazon Web Services do uruchamiania kodu, który jest sterowany zdarzeniami, a *Serverless Application Model (SAM)* to rozwiązanie open source, które w znaczący sposób upraszcza konfigurację i wdrażanie oparte na tej usłudze. Wspólnie pozwalają na tworzenie w łatwy sposób automatycznie skalujących się API oraz serwisów zaprojektowanych pod wdrożenia klasy *serverless*. Ebook ***Serverless w 5 minut*** wyjaśnia finansowe i techniczne ograniczenia tworzenia aplikacji typu serverless. Przeczytasz w nim, kiedy oraz dlaczego AWS Lambda jest dobrym rozwiązaniem do budowy aplikacji oraz kiedy powinieneś używać czegoś innego.

Z publikacji dowiesz się także m.in.:

- jak wygląda model cenowy usług serverless i jaki ma on wpływ na architekturę aplikacji, model wdrożenia, bezpieczeństwo i decyzje dotyczące rozwoju produktu,
- jakie są ograniczenia techniczne na platformie AWS Lambda, takie jak np. brak sesji użytkowników czy ograniczony czas wykonania,
- kiedy warto używać *Lambdy*,
- jakie kwestie należy przeanalizować przed decyzją o skorzystaniu z AWS Lambda.

Przeczytaj ten praktyczny poradnik i dowiedz się, jak rozpocząć pracę z AWS Lambda oraz Serverless Application Model.

Wioleta Szczygielska-Dybciak

Wioleta Szczygielska-Dybciak
Wydawca Redakcji IT

Aplikacje typu serverless są na najniższym technicznym poziomie oprogramowaniem, które działa w środowisku dostawcy usług w pełni odpowiedzialnego za infrastrukturę oraz zadania operacyjne, takie jak zarządzanie siecią oraz przychodzącymi żądaniami, skalowanie na żądanie, monitorowanie oraz przywracanie po awariach.

Zamiast kontenerów, które opakowują nasze aplikacje z logiką biznesową systemem operacyjnym, serwerami (takimi jak web serwery albo brokerzy wiadomości) z logiką biznesową, aplikacje typu serverless dostarczają kod, który zostanie wykonany w odpowiedzi na zdarzenie oraz konfigurację źródeł takich zdarzeń, które wywołują napisany kod.

W chmurze *Amazon Web Services* środowisko uruchomieniowe dla takich architektur nosi nazwę AWS Lambda. Wspiera ono szeroką gamę źródeł zdarzeń, włączając w to przychodzące żądania HTTP, wiadomości w kolejkach, przychodzące wiadomości e-mail, zmiany zachodzące na poziomie rekordów w bazie danych, uwierzytelnienie użytkownika, wiadomości przychodzące w ramach połączenia typu *WebSocket*, synchronizację urządzenia klienta oraz wiele innych.

Ponieważ twórcy aplikacji nie przygotowują tej części kodu, która odpowiada za komunikację ze źródłem zdarzeń po stronie środowiska *AWS Lambda*, a jedynie obsługę samego zdarzenia, określamy takie aplikacje mianem *serverless* (bez serwerów). Jednak to pojęcie zostało straszliwie zniekształcone przez marketing i w związku z tym w Internecie możesz znaleźć masę słabych żartów, jak cały czas mamy do czynienia w takim środowisku gdzieś na dole z serwerami. Zdecydowanie lepszym porównaniem będzie *WiFi*. Kiedy przeglądasz internet z wykorzystaniem takiego połączenia, zwanego z angielska *wireless*, Twoje urządzenie korzysta z routera oddalonego o kilka metrów od Ciebie. I mimo że masz dostęp do internetu, nie musisz zarządzać oraz przejmować się połączeniem kablowym do wysyłania pakietów do internetu, dopiero najprawdopodobniej router będzie połączony kablem z siecią dostawcy. Identycznie jest w przypadku *serverless*. Oczywiście w tle mamy do czynienia z serwerami, siecią, maszynami wirtualnymi, ale nie musisz się nimi przejmować, aby uruchomić swój kod.

W porównaniu z uruchamianiem aplikacji w kontenerach lub bezpośrednim zarządzaniem maszynami wirtualnymi aplikacje typu serverless mają dwie olbrzymie zalety:

- *krótszy czas wprowadzenia produktu na rynek*, co skutkuje skuteczniejszą innowacją oraz dostarczaniem szybciej wartości klientom;
- *obniżony koszt zadań operacyjnych* dzięki lepszemu wykorzystaniu zasobów.

Badania przeprowadzone przez firmę IDC[1] w roku 2018 sugerują, że firmy, które skorzystały z podejścia *serverless* do budowy swoich aplikacji, średnio „obniżyły pięcioletni koszt operacyjny o 60% oraz były o 89% szybsze we wdrożeniach". To jest zgodne z moim doświadczeniem. Po migracji *MindMup* z klasycznego modelu, gdzie płaciliśmy za zarezerwowane zasoby, do *AWS Lambda* zredukowaliśmy koszty utrzymania infrastruktury o ponad dwie trzecie. Kolejny popularny przykład to *Yubl*, prezentowany przez Yana Cui na różnorakich międzynarodowych konferencjach. Obniżyli oni koszty operacyjne o 95%, przenosząc się na Lambdę. Oba przykłady są dogłębnie omówione w opracowaniu naukowym *Serverless Computing: Economic and Architectural Impact*[2], którego jestem współautorem wraz z Robertem Chatleyem.

[1] https://pages.awscloud.com/Gated_IDC_Generating_Value_Through_IT_Agility.html

[2] https://www.doc.ic.ac.uk/~rbc/papers/fse-serverless-17.pdf

W marcu 2018 roku firma Cloudability opublikowała wyniki badania *State of the Cloud*[3] . W badaniu pojawia się teza wraz z uzasadnieniem, że wykorzystanie architektury *serverless* wzrosło o ponad 667% w ciągu roku. To nie jest zaskakujący wynik, biorąc pod uwagę, że połączenie obniżonych kosztów operacyjnych i krótszego czasu wprowadzenia produktu na rynek niesie ze sobą bardzo silną biznesową motywację dla firm, aby skorzystać z usługi *AWS Lambda* i podobnych. Jeśli się głębiej nad tym zastanowić, to wspólny mianownik dla obu wymienionych zalet jest taki sam. To model cenowy usług *serverless*.

Model cenowy usług serverless

Praktycznie rzecz ujmując, *AWS Lambda* i podobne platformy to serwisy do zarządzania kontenerami na sterydach. Dostarczają one standardowych środowisk uruchomieniowych w celu szybkiego tworzenia aplikacji oraz algorytmów do automatycznego skalowania kontenerów w zależności od ilości pracy. Mimo że zbudowanie takiej platformy samodzielnie jest technicznym wyzwaniem, od strony twórców aplikacji to tylko drobny krok naprzód w kilkunastoletniej podróży, jaką odbywamy z wirtualizacją i parawirtualizacją. To właśnie dlatego część z architektów, w szczególności tych, którzy stworzyli niejeden system oparty na konteneryzacji, czasami skarży się, że *serverless* to marketingowa wydmuszka. AWS jednak zrobił rewolucyjną rzecz, którą notabene reszta dostawców zaczęła bardzo szybko kopiować, dotyczącą strony finansowej tego podejścia. Model cenowy rozwiązań klasy serverless jest dużo ważniejszy niż technologia.

Kiedy korzystasz z *AWS Lambda*, aby uruchamiać swój kod, płacisz tylko za to, jak długo to trwało, a nie za zarezerwowaną dostępność środowiska, na którym zostanie uruchomiony. Jeżeli aplikacja nic nie robi, nie płacisz nic. Jeśli nagle pojawi się milion użytkowników, Lambda stworzy tyle kontenerów, ile będzie potrzebowała, aby obsłużyć taki ruch, i usunie je zaraz po tym, jak nie będą potrzebne. Dodatkowo nie będziesz płacić za okresy przestoju, gdy na infrastrukturze nie ma uruchomionego kodu lub gdy Twój kod czeka na nadchodzące zadania.

Rezerwacja minimalnej przepustowości

W grudniu 2019 roku AWS umożliwił klientom rezerwację minimalnej przepustowości dla funkcji Lambda, co pozwala upewnić się, że zawsze będziemy mieli określoną liczbę procesów oczekujących na przychodzące żądania. W żargonie AWS nazywa się to zarezerwowaną współbieżnością (reserved concurrency). W przypadku takiej funkcjonalności płacimy stałą kwotę za zarezerwowaną przepustowość, niezależnie od tego, czy ją wykorzystujemy, czy nie. Większość aplikacji nie musi korzystać z takiej funkcjonalności, pod warunkiem że zostanie odpowiednio zaprojektowana.

[3] https://get.cloudability.com/ebook-state-of-cloud-2018.html

Cena w takim modelu zależy od dwóch czynników: maksymalnego zużycia pamięci w danym zadaniu oraz czasu poświęconego na wykonywanie kodu. Dla przykładu, zakładając, że skonfigurowaliśmy naszą funkcję z górną granicą zużycia pamięci na 512 MB w jednym z regionów w Stanach Zjednoczonych, *AWS* podliczy koszty następująco:

- $0.0000002 za pojedyncze żądanie;
- $0.000000833 za wykonanie 100 ms w zadeklarowanym przedziale zużycia pamięci.

Porównanie modelu płatności za zarezerwowane maszyny wirtualne i za zużycie nie jest proste, ponieważ zależy od wielu czynników. Cena takiej maszyny w najprostszym przypadku zależy od oczekiwanego obciążenia, a cena związana z wykorzystaniem maszyny zależy od zadania, jakie jest na niej wykonywane. Oto dwa przykłady z dwóch różnych biegunów.

Dla zadania wykonywanego nieregularnie, co pięć minut, przez 100 ms oraz potrzebującego 512 MB pamięci cena za funkcję Lambda to w dużym zaokrągleniu mniej niż 1 cent z dolara amerykańskiego na miesiąc. Wykorzystanie podobnej konfiguracji z usługi *AWS Elastic Compute Cloud* (EC2), zakładając, że mamy dwie identyczne maszyny, aby zapewnić wysoką dostępność w razie problemów, zgodnie z rekomendacjami AWS będzie to kosztowało 9 dolarów amerykańskich, trzy rzędy wielkości więcej. Zauważmy, że dla AWS Lambda nie musisz rezerwować zastępczej usługi w celu zapewnienia wysokiej dostępności, ponieważ to jest już wliczone w cenę.

Z drugiej strony pojedyncza funkcja Lambda, która otrzymuje ciągle żądania i która nigdy się nie zatrzymuje, kosztuje miesięcznie około 27 dolarów amerykańskich. Porównując same tylko koszty maszyn wirtualnych, już widać istotną różnicę. Jednak to tylko przy założeniu, że same maszyny wirtualne wystarczą, aby utrzymać taki stan bez przerwy z takim obciążeniem. Dużo bardziej prawdopodobne jest, że dla pracy ciągłej będziemy potrzebować cały klaster maszyn wraz z maszynami zapasowymi.

Dodatkowo praca z taką usługą wymaga operatorów, którzy planują możliwości klastra i którzy dbają o skalowanie całej floty, tak aby wytrzymała ona ciągłe obciążenie. W przypadku Lambdy to wszystko jest zawarte w cenie, wliczając dodatkowo obsługę błędów, logowanie, monitorowanie i wersjonowanie.

Różnica między opłatami za zarezerwowane usługi a opłatami za wykorzystanie usługi jest również istotna dla testów oraz środowisk testowych. Kiedy firma płaci za zarezerwowane usługi, kopie środowisk produkcyjnych podrażają koszty operacyjne, nawet jeśli nie są używane. To dlatego najczęściej tego typu środowiska są odchudzane oraz przycinane do granic przyzwoitości i, koniec końców, odbiegają w znaczącym stopniu od produkcji. Dzięki rozliczaniu za zużycie środowiska nic nie kosztują, jeśli nie są używane, co oznacza, że znika motywacja do przycinania ich na siłę. Dla wielu organizacji środowiska testowe w architekturze klasy serverless są praktycznie darmowe.

W przypadku aplikacji stworzonych w tej architekturze to dostawca kontroluje infrastrukturę, nie twórcy aplikacji. To oznacza, że zespoły deweloperskie mogą się skupić na rzeczach, które są istotą działania biznesu, są ważne z punktu widzenia przewagi biznesowej i stanowią wyraźny wyróżnik na rynku, zamiast tracić czas na zadania operacyjne czy też te związane z infrastrukturą. Po migracji *MindMup* ze środowiska maszyn wirtualnych na *AWS Lambda* zdaliśmy sobie sprawę, że możemy usunąć znaczną część kodu źródłowego, który był związany z zadaniami dotyczącymi infrastruktury, i od końca 2016 nie napisaliśmy ani jednej linijki dotyczącej takiego zadania. Nie musieliśmy spędzać czasu na budowaniu i integracji narzędzi do monitorowania oraz skalowania systemu ani nie musieliśmy się martwić problemami operacyjnymi. Lambda pomogła nam znacznie szybciej wyjść od koncepcji do działającego oprogramowania dostarczonego użytkownikom.

Wpływ modelu płatności na architekturę aplikacji oraz wdrożenia

W aplikacjach typu *serverless* programiści tworzą kolejne funkcje, tak aby wykonać określony proces biznesowy, używając platformy do koordynacji, zarządzania stanem i komunikacji z klientami. W przypadku *AWS* model kosztów jest zorganizowany dookoła wykorzystanych, a nie zarezerwowanych zasobów. *Amazon Simple Storage Service (S3),* skalowalny system plików, rozlicza użytkowników za przesłane bajty do i z serwisu. *Amazon Simple Notification Service (SNS),* który odpowiada za mechanizm rozsyłania wiadomości i subskrypcji, rozlicza nas za każdą wysłaną wiadomość. Cała platforma jest tak zaprojektowana, że całkowity koszt Twojej aplikacji zależy od tego, jak często ją wykorzystujesz. W tym przypadku *AWS Lambda* to uniwersalny klej, który spaja te wszystkie usługi razem.

W swojej prezentacji *Why the Fuss about Serverless*[4] Simon Wardley słusznie zauważył, że rozwiązania klasy *serverless* to właściwe rozwiązania pasujące do definicji platform as a service, która została zawłaszczona przez zupełnie inne rozwiązania z powodu błędnego podejścia marketingowego. Bez wątpienia historia

[4] https://www.youtube.com/watch?v=b7Nc_FJiosk

zatoczy koło i za kilka lat okaże się, że to, co będzie nosiło nazwę serverless, nie będzie w ogóle podobne do obecnych rozwiązań. Jednak obecnie takie rozwiązania posiadają następujące trzy cechy:

- Dostawcy infrastruktury są odpowiedzialni za przychodzące żądania.
- Koszt jest wyliczany na podstawie zużycia, z dokładnością do pojedynczego żądania.
- Zadania operacyjne (wdrożenie, skalowanie, bezpieczeństwo, dostępność i monitoring) są wliczone w cenę.

Wspomniane trzy czynniki tworzą interesujące środowisko. Dla przykładu, biorąc pod uwagę rozliczanie zużycia za każde żądanie bez dodatkowych kosztów związanych z utrzymaniem środowiska, koszt wysłania miliona zapytań do pojedynczej wersji naszej aplikacji będzie taki sam jak do wysłania ich do dwóch lub więcej różnych wersji. Liczba żądań jest związana z kosztem, a nie ilością środowisk.

Gdyby zastanowić się nad historią, poczynając już od czasów architektury typu *mainframe*, wszystkie wdrożenia miały tę samą wadę, związaną z kosztem i wyzwaniem stworzenia kolejnego środowiska. Właśnie dlatego środowiska preprodukcyjne lub testowe wyglądały inaczej (były mniejsze) niż produkcyjne, programiści walczyli wraz z testerami o władzę i zasoby tego pojedynczego środowiska integracyjnego. Konteneryzacja oraz maszyny wirtualne radykalnie zmniejszyły czas związany z przygotowaniem nowych środowisk, ale nie obniżyły drastycznie kosztów utrzymania dla wielu środowisk. Dwie kopie środowiska produkcyjnego kosztują dokładnie dwukrotność kosztu pojedynczego. W tym przypadku istnieje bardzo silna motywacja do ponownego wykorzystania zasobów i upakowywania funkcjonalności razem, żeby zaoszczędzić.

Ostatnimi czasy pracowałem z klientem, u którego całe tworzenie raportów funkcjonowało na tym samym klastrze co środowisko produkcyjne. W rezultacie generowanie pojedynczego raportu spowalniało całą resztę aplikacji, które pracowały na tym samym sprzęcie. Rozwiązanie polegające na przygotowaniu środowiska dla każdego typu raportu jest nieekonomiczne, ponieważ przez większość czasu takie środowisko byłoby nieużywane. Z drugiej strony scalenie wszystkich generatorów w jeden klaster wymaga bardzo dokładnego planowania i zarezerwowania odpowiednich zasobów, a to z kolei często kończy się przepłacaniem (przeszacowanie, płacimy za niewykorzystane zasoby) lub problemami w przypadku nieco wyższego zapotrzebowania (niedoszacowanie). W przypadku *AWS Lambda* planowanie jest po stronie dostawcy, a my jako klienci nie mamy żadnej motywacji ani zysku z upakowywania różnych aplikacji na przygotowanych zasobach. Zamiast scalać komponenty, żeby zaoszczędzić pieniądze na wdrożeniach, co będzie powodowało dalsze problemy z konkurencją związane z zasobami oraz z wydajnością, Lambda zapewnia nam możliwość wdrażania każdego typu generatora wspomnianych raportów osobno, w izolowanej infrastrukturze. Skoro cena jest proporcjonalna do wykorzystania, a nie ilości zarezerwowanych zasobów, to koszt będzie identyczny niezależnie od sposobu, w jaki podzielimy nasze generatory. Jeżeli generatory mają różne

zapotrzebowanie co do zużycia pamięci, będziemy mogli dodatkowo zaoszczędzić. W klastrze każdy z kontenerów musiałby zarezerwować tyle pamięci, ile potrzebowałby najbardziej pamięciożerny generator. W przypadku funkcji *Lambda* procesy są izolowane i możemy dobrać odpowiedni rozmiar pamięci do każdego generatora, a mniej pamięci oznacza niższy koszt.

Ponadto z Lambdą możemy stworzyć dowolną liczbę wersji naszej infrastruktury bez dodatkowych kosztów. Zamiast programistów i testerów, którzy kłócą się o pojedynczą kopię środowiska testowego, można stworzyć po jednej dla każdego zespołu lub nawet poszczególnej osoby. Wszystko z idealnym odwzorowaniem środowiska produkcyjnego, zwłaszcza gdy chodzi o zarezerwowane zasoby. Koszt będzie dokładnie taki sam, jak w przypadku rozwiązania z jednym współdzielonym środowiskiem, ponieważ znowu cena jest uzależniona od zużycia, a nie od zarezerwowanych zasobów czy ilości środowisk.

Wpływ modelu płatności na bezpieczeństwo

Wspomniany mechanizm ma również istotny wpływ na bezpieczeństwo aplikacji. Współczesne aplikacje serwerowe zazwyczaj posiadają wewnętrzną implementację, która odpowiada za uwierzytelnienie, kontrolę dostępu i rozdzielanie pracy. Taki proces potrzebuje dostępu do zewnętrznych usług, nierzadko również bazy danych. Zabezpieczenie całej infrastruktury jest możliwe, ale jest też problematyczne. Większość serwerów aplikacji webowych oraz bazodanowych uruchamianych jest zazwyczaj z uprawnieniami administratora. W razie włamania do systemu z wykorzystaniem takiego procesu w zasadzie nie ma żadnych innych zabezpieczeń, które mogłyby nas uchronić przed niecnymi zamiarami atakujących. W przypadku rozwiązań opartych na *serverless* jest znacznie łatwiej kontrolować dostęp oraz reguły bezpieczeństwa. Ponieważ nie ma motywacji do scalania zadań w większe serwisy, każda funkcja *Lambda* może wykonywać tylko małe zadanie, pracując w restrykcyjnych warunkach z ograniczonymi uprawnieniami. Bardzo łatwo zastosować zasadę minimalnych uprawnień *(principle of least priviledge)*, przyznając dostęp tylko do niezbędnych zasobów. Takie zachowanie pomaga ograniczyć zasięg ewentualnego włamania do minimum.

Wraz z cennikiem skupionym na zużyciu twórcy aplikacji dążą do optymalizacji w postaci dynamicznych środowisk i infrastruktury. Dostawca może łatwo usunąć zbędne lub zastąpić istniejące zasoby bez przerywania pracy aplikacji. To ma krytyczne znaczenie w kontekście aplikowania poprawek bezpieczeństwa. Pewnego dnia, w sierpniu 2018 roku, otrzymaliśmy wiadomość e-mail od jednego z administratorów firmy naszego klienta. Pytał nas o plany wdrożenia poprawek bezpieczeństwa dotyczących dziur w procesorach Intel[5] , które pojawiły się poprzedniego dnia. W trakcie wyszukiwania informacji na temat tych konkretnych błędów bezpieczeństwa natrafiłem na informację, że poprawki zostały już zaaplikowane przez Amazon na infrastrukturę, na której pracuje *AWS Lambda*. Nie musieliśmy z naszej strony wykonać żadnego działania w tej sprawie. To dostawca zadbał samodzielnie o to, aby stare środowiska uruchomieniowe zostały załatane, wykonując całą operację za nas, w dodatku bez zatrzymywania naszych aplikacji.

Wpływ modelu płatności na decyzje dotyczące rozwoju produktu

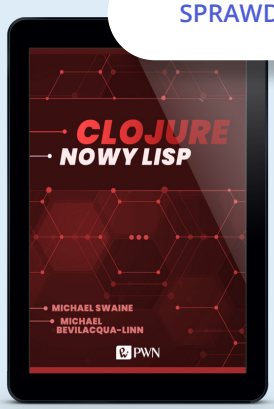
Cennik Lambdy wpływa również na sposób, w jaki organizacje wypuszczają nowe funkcjonalności. Czy kojarzysz sytuację, że na wymagania jednego klienta, wymagania, które notabene możesz przygotować w kilka dni, muszą czekać dodatkowe kilka tygodni, ponieważ tyle potrwa dostosowanie ich do wszystkich innych klientów i wdrożenie? Z *AWS Lambda* możesz wypuścić nową wersję dla tego konkretnego klienta tak szybko, jak tylko będzie ona gotowa, jednocześnie czekając na dostosowanie funkcjonalności do wszystkich. Dwa środowiska nie kosztują nas ani grosza więcej niż jedno współdzielone.

Podobnie, płatność za zużycie otwiera całkiem nowe możliwości związane z testami funkcjonalności. Jeśli chcesz przetestować, jak zmiana po stronie serwera wpłynie na konwersję, nic nie stoi na przeszkodzie, aby stworzyć drugie środowisko, na którym przetestujemy naszą zmianę, wysyłając tam 10% całkowitego ruchu. Liczba użytkowników pozostaje w dalszym ciągu taka sama, więc dwie wersje środowiska będą kosztowały dokładnie tyle co jedno środowisko. Uruchomienie testów A/B po stronie klienta jest z reguły tanie i szybkie. Organizacja tego samego po stronie serwera w klasycznych rozwiązaniach wiąże się

[5] Chodziło o błędy Meltdown oraz Spectre (od tłumacza).

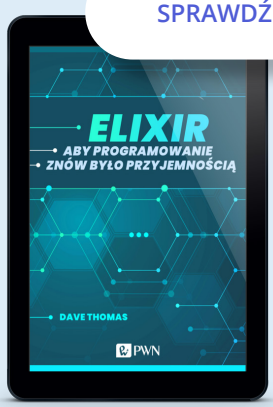
Poznaj funkcyjne języka programowania

SPRAWDŹ



Clojure. Nowy Lisp
Autor: Michael Bevilacqua-Linn, Michael Swaine

SPRAWDŹ

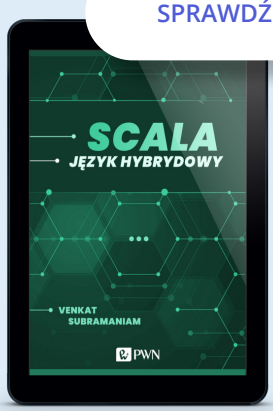


Elixir. Aby programowanie znów było przyjemnością
Autor: Dave Thomas

SPRAWDŹ

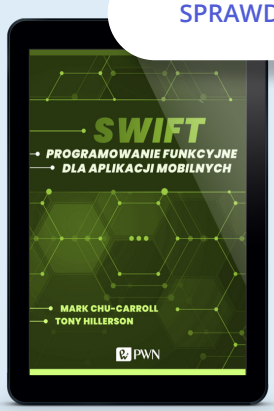
Haskell. Plac ćwiczeń dla uczonych
Autor: Paul Callaghan

SPRAWDŹ



Scala. Język hybrydowy
Autor: Venkat Subramaniam

SPRAWDŹ



Swift. Programowanie funkcyjne
dla aplikacji mobilnych
Autor: Mark Chu-Carroll, Tony HillersonPaterson

SPRAWDŹ

11 zasad projektowania architektury
oprogramowania
Autor: Michael Keeling

z podwojeniem kosztów. Pięć lat temu tylko firmy z olbrzymimi zasobami, takie jak Google lub Facebook, mogły uruchamiać ciągle takie eksperymenty na swojej infrastrukturze serwerowej. Dzięki *Lambdzie* jest to dostępne dla wszystkich, nawet zespołów jednoosobowych. Nie kosztuje to więcej niż utrzymanie pojedynczego środowiska produkcyjnego.

Jeszcze innym interesującym aspektem analizowanego modelu cenowego jest granulacja kosztów na poziomie pojedynczego żądania. Aby poprawnie wyliczyć całkowity koszt, *AWS* zlicza każde żądanie i ta informacja jest dostępna również dla nas, klientów. Dysponując tymi informacjami, jesteśmy w stanie w pełni przeanalizować przepływ kapitału przez aplikację z dokładnością do pojedynczego zasobu. Możemy wyliczyć bardzo dokładnie, ile kosztuje nas obsługa pojedynczego klienta, włącznie z decyzją, czy klient jest wart obsługi, czy nie. Poza tym możemy bardzo łatwo policzyć, ile kosztuje nas utrzymanie poszczególnych funkcjonalności w naszej aplikacji, i z taką wiedzą możemy racjonalnie zadecydować, czy warto w nią dalej inwestować.

Ograniczenia techniczne na platformie AWS Lambda

Lambda jest stosunkowo młodą usługą i ewoluuje bardzo szybko. Bardzo często przygotowuję wystąpienia na konferencje i bawi mnie przeglądanie slajdów pod kątem poprawek związanych z ograniczeniami i problemami z *serverless*. Kwestie, takie jak czas startu aplikacji albo brak zgodności z wymaganiami PCI (Payment Card Industry), były poważnymi ograniczeniami w 2017 roku, ale zostały poprawione. Bardzo częstym zarzutem, z jakim *Lambda* się spotyka, jest brak umów SLA (Service Level Agreements), ale w październiku 2018 roku *AWS* opublikował dokument regulujący tę kwestię[6] (obecnie jest ona na poziomie 99,95%).

Wtedy, kiedy będziesz czytać tę książkę, wymienione niżej ograniczenia mogą ulec zmianie, ale w tym momencie są cztery ważne problemy, o których warto pamiętać w kontekście wykorzystania Lambdy:

- Brak sesji użytkowników;
- niedeterministyczne opóźnienia;
- czas wykonania ograniczony do 15 minut;
- brak pełnej kontroli nad przydzielaniem mocy obliczeniowej.

Brak sesji użytkowników

Ponieważ to dostawca kontroluje proces skalowania, nie będziesz w stanie zadecydować o tym, jak wiele kontenerów przygotować do wykonywania poszczególnych funkcji. To *Lambda* zadecyduje, czy potrzebujesz dodatkowy zasób do obsłużenia żądania i kiedy wykorzystać ponownie poprzedni kontener lub usunąć ten aktywny. Może się tak zdarzyć, że dwa następujące po sobie żądania pochodzące od tego samego użytkownika zostaną obsłużone przez tę samą instancję funkcji lub dwie różne.

[6] https://aws.amazon.com/lambda/sla/

Nie ma możliwości, aby z wykorzystaniem samej Lambdy kontrolować przepływ żądań lub zapewnić w jakikolwiek inny sposób, żeby żądania od tego samego źródła trafiły w tej samej kolejności do tego samego celu. Można to uzyskać, korzystając z dodatkowych usług, takich jak *Amazon Kinesis*, ale to *Lambda* cały czas decyduje o tym, kiedy stworzyć nowy kontener, a kiedy ponownie użyć poprzedniego.

To ograniczenie, wraz z nazewnictwem dotyczącym funkcyjnego podejścia, bardzo często powoduje zamieszanie związane z tym, jak *Lambda* działa. Funkcje nie są bezstanowe, przynajmniej nie w znaczeniu związanym z programowaniem funkcyjnym, gdy nic nie jest ponownie wykorzystywane między poszczególnymi wywołaniami. Każde z wywołań dalej uruchomione jest na kontenerze, który ma lokalną pamięć. I jeśli platforma zadecyduje o ponownym skorzystaniu z niego, dwa kolejne żądania mogą dzielić stan przez lokalną pamięć kontenera. Ten sam kontener ma również tymczasową przestrzeń dyskową, więc jest możliwe skorzystanie z niej i zapisanie pliku. Nie mamy jednak żadnej gwarancji, dotyczącej ponownego wykorzystania tego stanu między żądaniami, oraz nie mamy wpływu na sposób rozdzielania żądań.

Najlepszym wyjściem jest założenie, że stan pamięci będzie *zaw*sze nienaruszony między dwoma kolejnymi żądaniami obsłużonymi przez Lambdę. Należy tak projektować funkcje, aby nie współdzieliły niczego. Możemy oczywiście korzystać z pamięci podręcznej lub na początku przygotowywać pewne wyliczenia, jednakże sesja użytkownika oraz jego stan musi być składowany poza funkcjami *Lambda*.

Niedeterministyczne opóźnienia

Funkcje *Lambda* są zoptymalizowane, aby móc dostarczać jak najwyższą przepustowość, a nie najszybszy czas dostępu. Zatem ważniejsze jest obsłużenie wielu równoległych żądań, tak aby żadne z nich nie czekało, od tego, żeby obsłużyć każde z nich w jak najkrótszym czasie. Oznacza to, że w niektórych przypadkach żądania będą musiały poczekać, aż wystartuje nowa instancja funkcji, a w niektórych nie. Opóźnienia w przetwarzaniu będą trudne do przewidzenia.

W żargonie *serverlessowym* istnieje pojęcie zimnego startu (*cold start*), z którym mamy do czynienia wówczas, gdy nadchodzące żądanie musi poczekać na stworzenie nowej funkcji. Na początku istnienia serwisu zimny start Lambdy wynosił średnio kilka sekund. W tym czasie powstało wiele artykułów na temat, jak ominąć ten problem dzięki podtrzymywaniu ciepłych instancji na siłę. Ta sytuacja uległa jednak drastycznej poprawie i w wielu przypadkach można zignorować te porady. *AWS* nie publikuje żadnych informacji na temat zimnego startu, ale nasze testy sugerują, że jeśli chodzi o *JavaScript* lub *Python*, ta wartość oscyluje w granicach sekundy w pesymistycznym przypadku. Jeśli chodzi o języki, które posiadają maszynę wirtualną, takie jak *Java*, ten parametr będzie odpowiednio dłuższy i dodatkowo zależny od rozmiaru aplikacji. To jeszcze jeden powód, dlaczego preferujemy lekkie środowiska i języki programowania w przypadku platformy *Lambda*. Jeśli chodzi o bardziej drobiazgową analizę tego problemu (jednak w dalszym ciągu nieoficjalną), polecam sprawdzić artykuł autorstwa Mikhaila Shilkova[7].

Pod koniec 2019 roku *AWS* znacząco zredukował czas startu w odniesieniu do funkcji *Lambda*, które korzystają z połączenia do usługi VPC (Virtual Private Cloud). W przypadku aplikacji, które nie mogą pozwolić sobie na długotrwałą inicjalizację, dostawca umożliwił rezerwację zasobów, co praktycznie rozwiązało problem w sprawie *zimnego startu*.

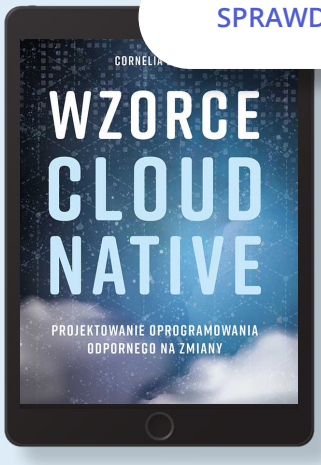
[7] https://mikhail.io/serverless/coldstarts/aws/

Ograniczony czas wykonania

Kolejnym znaczącym obostrzeniem jest maksymalny czas wykonania. Obecnie *Lambda* jest ograniczona domyślnie do 3 sekund, ale możemy ten czas wydłużyć, pamiętając jednak o tym, że górny limit wynosi 15 minut. Jeśli zadanie nie zostanie wykonane w zadanym czasie, platforma zatrzyma środowisko uruchomieniowe oraz odeśle błąd związany z przekroczeniem czasu wykonania.

Wspomniany górny limit jest na ten moment nieprzekraczalny. Dłuższe zadania muszą być podzielone na mniejsze i wykonywane w kolejnych wywołaniach lub za pomocą innego serwisu. *AWS* ma w swoim portfolio usługi, takie jak *Fargate*, które na to pozwalają w modelu *serverless*, wszystko jednak kosztem dłuższego *zimnego startu* oraz wyższego kosztu za zużycie.

SPRAWDŹ

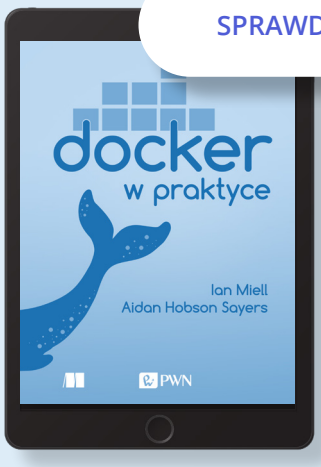


Wzorce Cloud Native
Projektowanie oprogramowania odpornego na zmiany
Autor: Cornelia Davis

SPRAWDŹ

Mikroserwisy
Wzorce z przykładami w języku Java
Autor: Chris Richardson

SPRAWDŹ



Docker w praktyce
Autorzy: Ian Miell, Aidan Hobson Sayers

Gdy znamy to ograniczenie, w wielu przypadkach, projektując aplikację, będziemy w stanie przygotować odpowiednie rozwiązanie tego problemu. Przykładem może być wykorzystanie funkcji do wystartowania zdalnego zadania z długim czasem oczekiwania oraz oczekiwanie na odpowiedź za pomocą kolejnej funkcji. W praktyce pierwsza funkcja wysyła żądanie do serwisu, który na zakończenie długotrwałego procesu wykona zupełnie inną funkcję *Lambda*. Można wykorzystać do koordynacji koleją usługę dostępną na platformie *AWS*, zwaną *Step Functions.* Koordynacja i oczekiwanie na odpowiedź może trwać nawet do roku, a po otrzymaniu odpowiedzi możemy wykonać kolejną funkcję. Dzięki takiej usłudze nie trzeba już nigdy więcej płacić za oczekiwanie.

Brak pełnej kontroli nad przydzielaniem mocy obliczeniowej

Ostatni problem związany jest z alokacją mocy obliczeniowej. Dzisiaj w klasycznych rozwiązaniach opartych na kontenerach możemy doprecyzować, jakimi dysponują zasobami i mocą obliczeniową, wliczając w to różne rodzaje i rozmiary *CPU* oraz *GPU*, oraz typy zasobów zoptymalizowanych pod kątem konkretnych zastosowań. Jeśli chodzi o *Lambdę*, nie mamy takiej możliwości. Jedyna decyzja dotyczy ilości pamięci, od 128 MB do prawie 4 GB. Nie ma sposobu, aby np. dołączyć do naszej funkcji *GPU*.

Dodatkowo decyzja dotycząca pamięci wpływa pośrednio na moc obliczeniową. *Lambda* przeznacza określoną część wirtualnego procesora proporcjonalnie do wybranej pamięci. Od wartości 1792 MB otrzymujemy na wyłączność jeden wirtualny procesor[8]. W przypadku Node.js wszystkie operacje i tak wykorzystują tylko jeden rdzeń, więc jeżeli chodzi o to środowisko, nie ma sensu przydzielać więcej niż wyżej wspomnianą wartość. W odniesieniu do innych środowisk, np. *Java*, możemy w pełni wykorzystać wiele rdzeni i wtedy alokacja maksymalnej dopuszczalnej wartości, jeśli chodzi o pamięć, bezpośrednio przełoży się na szybsze czasy odpowiedzi i niższy koszt wykonania zadań, które potrzebują więcej czasu od procesora.

Warto pamiętać, że część ceny związana ze zużyciem jest uzależniona od dwóch rzeczy: od zaalokowanej pamięci oraz od czasu trwania wykonania naszego kodu. Biorąc pod uwagę fakt, że więcej pamięci przekłada się proporcjonalnie na procesor, może to doprowadzić do sytuacji, w której teoretycznie droższa *Lambda* będzie w efekcie tańsza, co jest niezgodne z intuicją. Co prawda, wybierając funkcję z większą ilością pamięci, podrażamy opcję cenową, ale jeśli to znacząco obniży czas wykonania, możemy dzięki temu uzyskać niższy koszt całkowity.

Ponieważ nie mamy jednak bezpośredniej kontroli nad alokacją mocy obliczeniowej, najlepszą metodą na obniżenie kosztów jest eksploracja możliwych kombinacji. Na szczęście, jak wcześniej wspominałem, platforma ułatwia przygotowanie wielu środowisk testowych, nie pociągając za sobą dodatkowych kosztów, oraz sama zmiana przydzielonej pamięci jest prosta i natychmiastowa. W przypadku zadań, które stają się za wolne lub zbyt drogie, gorąco polecam eksperymenty związane z przydzielaniem odpowiedniej ilości pamięci, tak aby odnaleźć idealną wartość, jeśli chodzi o cenę oraz wydajność. Dzięki otwartemu projektowi Alexa Casalboniego *AWS Lambda* Power Tuning[9] będziesz w stanie porównać wydajność (i koszty) w odniesieniu do różnych konfiguracji Twoich funkcji *Lambda*.

[8] https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html

[9] https://github.com/alexcasalboni/aws-lambda-power-tuning

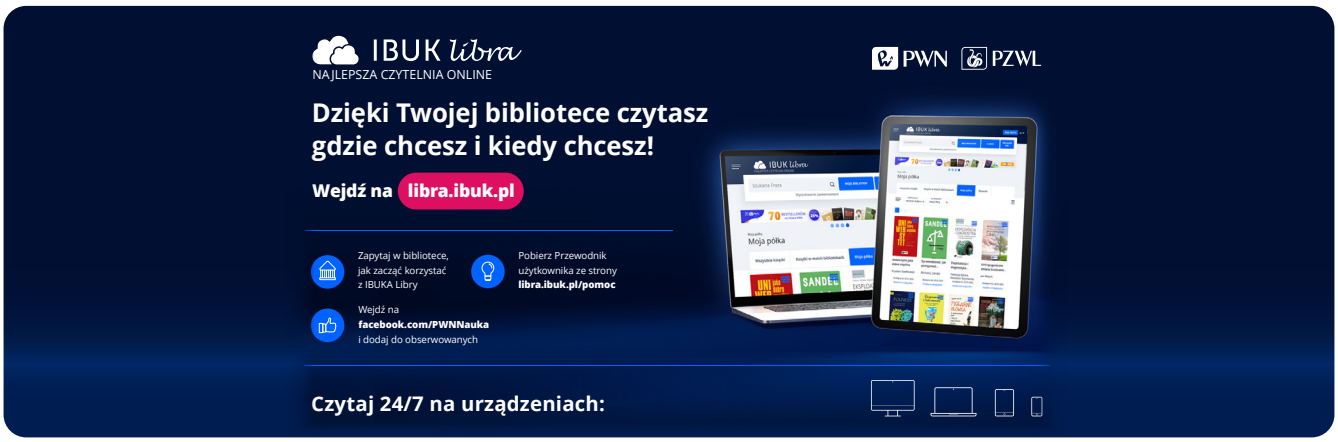
Kiedy używać Lambdy?

Lambda pasuje idealnie w sytuacjach, w których przepustowość jest kluczowa, a zadania łatwo zrównoleglić. Typowym przykładem są żądania w aplikacjach webowych, które wymagają wykonania odpowiedniej logiki biznesowej w zależności od przesłanych parametrów lub stanu zapisanego w zewnętrznym serwisie, np. bazie danych. Inny bardzo dobry przykład to automatyczna obsługa przychodzących zdarzeń, np. wiadomości e-mail lub zapytań od *botów*. Obsługa każdego z pojedynczych żądań, które zajmuje średnio najwyżej paręset milisekund, nie będzie zauważalne przez użytkownika końcowego, a *Lambda* pozwoli na to, żeby każdy został odpowiednio szybko obsłużony, niezależnie od obciążenia i zmiennych wzorców związanych z ruchem.

Długotrwałe przetwarzanie danych albo zadania obliczeniowe, które będą trwać krócej niż 15 minut lub które mogą być podzielone na niezależne segmenty o wspomnianym czasie trwania, to również dobry przykład wykorzystania tej platformy. W takich przypadkach zimny start nie będzie miał praktycznie żadnego wpływu na proces przetwarzania. Dobrym przykładem takich zadań są konwersje plików do różnych formatów, generowanie miniaturek lub obrazów podglądu oraz cykliczne raporty.

Zadania, które potrzebują wysokiej dostępności oraz stabilnej infrastruktury, ale nie mają wyśrubowanych wymagań co do czasu trwania, to również dobry przykład użycia. Jednym z takich przypadków użycia jest obsługa płatności oparta na zewnętrznych usługodawcach, np. *PayPal* lub *Stripe*. Obsługa takich żądań i przychodzących notyfikacji musi być stabilna oraz relatywnie szybko reagować na zmieniające się wzorce ruchu, za to nie ma wymagań, aby czas odpowiedzi był jak najniższy.

Lambda w tym momencie nie jest przystosowana do zadań, które wymagają gwarancji najniższych czasów odpowiedzi, np. handel wysokich częstotliwości (*high-frequency trading, HFT*) lub systemy czasu rzeczywistego, ze sztywnymi gwarancjami co do czasu wykonania. Jeśli zadanie wymaga, aby było obsłużone *zawsze* poniżej pewnego progu, np. 10 lub 20 ms, dużo lepszym rozwiązaniem będzie klasyczny model, w przypadku którego zarezerwujemy odpowiednie zasoby i podłączymy go bezpośrednio do źródła żądań.



Kolejna sytuacja, w przypadku której *Lambda* się nie sprawdzi, dotyczy zadań wymagających dłuższego czasu wykonania niż 15 minut. Dobrym przykładem będzie transkodowanie dużych plików wideo. Innym przykładem jest wykorzystanie stanowego połączenia sieciowego i nieustanna konsumpcja danych – z powodu ograniczenia nałożonego na czas wykonania będziemy musieli rozłączać i podłączać się ponownie.

Trzecią kategorią przypadków, w odniesieniu do których nie polecam używać *AWS Lambda*, jest przetwarzanie wymagające dużej mocy obliczeniowej i koordynacji, np. renderowanie wideo. Zadania tego typu lepiej wykonywać na przeznaczonej do tego i zarezerwowanej infrastrukturze, w której kontrolujemy alokację zasobów i którą możemy dostosowywać na żądanie (włączając w to *CPU* oraz *GPU*).

Jako ostatni negatywny przykład chciałbym zaprezentować przypadek, który nie wymaga żadnych operacji wykonywanych na żądanie, np. serwowanie statycznych plików stron internetowych. W teorii nic nie stoi na przeszkodzie, aby wykorzystać do tego Lambdę, jednak będzie to strata pieniędzy. Wyjdzie znacznie taniej i szybciej, jeżeli skorzystamy ze specjalizowanej usługi, np. CDN (*Content Delivery Network*).

Interesującym aspektem wykorzystania Lambdy jest jej integracja z pozostałymi usługami na platformie *AWS*. Dzięki temu, krok po kroku, jesteśmy w stanie transformować nasze aplikacje, aby lepiej korzystały z architektury typu *serverless*, korzystając z przeznaczonych specjalizowanych usług, które lepiej pasują do tego konkretnego przypadku niż *AWS Lambda*.

Interesujące kwestie

- Przeanalizuj swoje aplikacje, nad którymi obecnie pracujesz, i postaraj się zidentyfikować pojedyncze zadania wykonywane przez kod aplikacji.
- Zastanów się nad czasem wykonania i przepustowością każdego z takich zadań, włączając w to wykonania równoległe, czasem trwania i ograniczeniami związanymi z sesją użytkowników. Zidentyfikuj te, które warto uruchomić jako funkcje *Lambda*.
- Przygotuj minimalny zestaw uprawnień wobec wybranych zadań i porównaj je z prawami dostępu, jakie mają obecnie.
- Postaraj się wyestymować koszt uruchomienia tych zadań na platformie *AWS Lambda*.

Zainteresował Cię ten temat?
Kup książkę i czytaj dalej:

Działaj z Serverless. Wprowadzenie do AWS Lambda i architektury Serverless

Autor: Gojko Adžić

SPRAWDŹ

Zainteresowały Cię nasze książki?

Znajdziesz je na:

PWN KSIĘGARNIA INTERNETOWA

Księgarnia Internetowa PWN oferuje szeroki zakres publikacji: podręczniki akademickie, książki naukowe i popularnonaukowe, słowniki języka polskiego i słowniki języków obcych. Znajdziesz w niej zarówno publikacje papierowe, jak i książki w wersji elektronicznej e-booki i audiobooki.

Przejdź do Księgarni Internetowej PWN

IBUK

IBUK.pl jest platformą pozwalającą kupować i wypożyczać e-booki. Można je wypożyczać zarówno pojedynczo - już od 4,92 PLN na dobę oraz w abonamentach - ceny zaczynają się nawet od 19,90 PLN miesięcznie. W ofercie dostępne sa także audiobooki.

Przejdź do Ibuk.pl

Śledź nas na Facebooku:

PWN STEM

COPYRIGHT I DANE KONTAKTOWE

Niniejszy e-book jest dziełem twórcy i wydawcy. Prosimy, abyś przestrzegał praw, które im przysługują. Nie publikuj go w Internecie. Cytując jego fragmenty, nie zmieniaj ich treści i koniecznie zaznacz, czyje to dzieło. Kopiując jego część, rób to jedynie na użytek osobisty.

© Wydawnictwo Naukowe PWN SA 02-460 Warszawa, ul. Gottlieba Daimlera 2

pwn.pl