Serverless, FaaS i architektura na brzegu

Koncepcja serverless · FaaS · Cold starts · Edge · WebAssembly · Koszty

mgr inż. Jakub Woźniak

Politechnika Poznańska · Instytut Informatyki · Semestr letni 2025/26

Plan wykładu

Agenda

- Koncepcja serverless — ewolucja modelu odpowiedzialności
- FaaS — model wykonania, platformy, cykl życia funkcji
- Cold starts — anatomia, benchmarki, strategie mitygacji
- Wzorce event-driven serverless (fan-out, orkiestracja)
- Serverless poza FaaS — bazy danych, kontenery, orkiestratory
- Edge computing — obliczenia na brzegu sieci
- WebAssembly na serwerze — WASI, Spin, porównanie z kontenerami
- Ograniczenia i kiedy serverless to zły wybór
- ☁ Porównanie kosztów i punkt break-even
- Case study: Design Google Drive (A. Xu, rozdz. 15)

Koncepcja serverless

Ewolucja modelu odpowiedzialności

Serverless ≠ „bez serwera“ — oznacza „bez zarządzania serwerem”

Dostawca przejmuje pełną odpowiedzialność za infrastrukturę, skalowanie i dostępność.

Model	Zarządzasz	Dostawca zarządza
IaaS	Aplikacja, dane, runtime, OS	Wirtualizacja, sieć, dyski
CaaS	Aplikacja, dane, runtime	OS, orkiestracja kontenerów
PaaS	Aplikacja, dane	Runtime, OS, skalowanie
FaaS	Kod funkcji	Wszystko inne + skalowanie do zera
Serverless (pełny)	Logika biznesowa	Compute + storage + integracje

Kluczowe cechy serverless

1. **Brak zarządzania serwerami** — zero provisioningu, patchowania, capacity planning
2. **Płatność za użycie** — rozliczenie per wywołanie + czas wykonania (ms)
3. **Automatyczne skalowanie** — od zera do tysięcy instancji
4. **Sterowanie zdarzeniami** — funkcja reaguje na event (HTTP, kolejka, plik, timer)
5. **Bezstanowość wykonania** — każde wywołanie niezależne; stan w zewnętrznych usługach

Konsekwencja architektoniczna

Serverless **wymusza** dobrą architekturę: dekomponowanie na małe, bezstanowe jednostki ze zdefiniowanymi kontraktami (zdarzeniami).

Functions as a Service (FaaS)

Model wykonania — cykl życia funkcji (AWS Lambda)

Pięć faz wywołania

1. **Provisioning** (50–100 ms) — alokacja microVM, sieć
2. **Pobranie kodu** (50–500 ms) — ZIP z S3 lub obraz z ECR
3. **Inicjalizacja runtime** (50–1000 ms) — start Node.js / Python / JVM
4. **Inicjalizacja użytkownika** (0–10 000 ms) — klienty SDK, połączenia DB
5. **Wykonanie handlera** — właściwa logika biznesowa

Warm invocation

Fazy 1–4 pomijane — środowisko wykonania reużywane. Koszt: 5–50 ms.

Platformy FaaS — przegląd (2025/26)

Cecha	AWS Lambda	Azure Functions	Cloud Functions 2nd gen	Cloudflare Workers
Timeout	15 min	30 min (Flex)	60 min	30 s (CPU)
Pamięć	128 MB – 10 GB	256 MB – 16 GB	128 MB – 32 GB	128 MB
Skalowanie do zera	Tak	Tak (Flex)	Tak	Tak
Równoległość/inst.	1 req	Konfig.	Do 1000 req	Tak (isolates)
Cold start (typowy)	100–700 ms	500 ms – 2 s	1–5 s	< 5 ms
Billing	1 ms	100 ms (Flex)	100 ms	Per-request

Dane: oficjalna dokumentacja AWS/Azure/GCP/Cloudflare, 2025/26.

AWS Lambda — nowości 2025/26

SnapStart (Java, Python, .NET)

Migawka pamięci (CRIU) po inicjalizacji ⟶ przywrócenie zamiast ponownego startu.
Java: 583 ms ⟶ 104 ms (**5,6× szybciej**). Python: 2–3×. .NET: 3–5×.

Lambda Managed Instances (re:Invent 2025)

Wiele requestów na jedno środowisko. Stała opłata + 15% management fee. Oszczędność 30–50% vs on-demand dla stałego ruchu.

Durable Functions (re:Invent 2025)

Timeout do 1 roku. Automatyczne checkpointy stanu. Brak opłaty za `context.wait()`. Workflow z człowiekiem w pętli.

Cold starts

Anatomia cold startu

Cold start = Provisioning + Pobranie kodu + Init runtime + Init użytkownika

Każda faza dodaje opóźnienie. Dominujący składnik zależy od języka i rozmiaru paczki.

Kiedy cold start się zdarza?

- Pierwsze wywołanie po wdrożeniu
- Skok współbieżności (nowe instancje)
- Timeout nieaktywności (zwykle 5–15 min)
- Aktualizacja konfiguracji

Cold starty dotyczą typowo 1–5% wywołań przy stabilnym ruchu.

Benchmarki cold startów — P50 / P99 (512 MB, ARM64)

Runtime	P50	P99	Uwagi
Rust (provided.al2023)	16 ms	22 ms	Najszybszy; brak GC
Go (provided.al2023)	38 ms	52 ms	Skompilowany; szybki start
Node.js 22	148 ms	210 ms	V8 startup + moduły
Python 3.13	171 ms	325 ms	Interpreter CPython
Java 21 (JVM)	698 ms	1100 ms	JVM + class loading
Java 21 + SnapStart	100–200 ms	200–400 ms	5,6× poprawa
.NET 8 (NativeAOT)	80–150 ms	150–250 ms	Kompilacja natywna

Źródło: github.com/maxday/lambda-perf (codzienne benchmarki).

Strategie mitygacji cold startów

1. Wybór runtime

SLA < 50 ms? ⟶ Rust / Go

SLA < 200 ms? ⟶ Node.js / Python

Istniejący kod Java? ⟶ SnapStart

2. Optymalizacja paczki

Tree shaking (esbuild, webpack)

Selektywne importy SDK

Minimalne obrazy kontenerowe

3. Provisioned Concurrency

Pre-inicjalizacja N środowisk.

Koszt: \$0,015/h per unit.

Opłacalność: SLA < 100 ms p99.

4. Init poza handlerem

Klienty SDK, pule połączeń — twórz **raz** na poziomie modułu, nie w handlerze.

Wzorce event-driven serverless

Źródła zdarzeń — trzy modele wywołania

Synchroniczne (request-response)

API Gateway ⟶ Lambda. ALB ⟶ Lambda. Wywołujący czeka na odpowiedź.

Asynchroniczne (fire-and-forget)

S3 ⟶ Lambda. SNS ⟶ Lambda. EventBridge ⟶ Lambda. Wywołujący nie czeka.

Event Source Mapping (polling)

SQS ⟶ Lambda. Kinesis ⟶ Lambda. DynamoDB Streams ⟶ Lambda. Kafka (MSK) ⟶ Lambda. Lambda sama odpytuje źródło; batch size konfigurowalny (1–10 000).

Wzorzec: Fan-out (jedno zdarzenie ⟶ wielu konsumentów)

Architektura

Źródło (np. upload do S3) ⟶ SNS/EventBridge ⟶ N kolejek SQS ⟶ N funkcji Lambda

Przykład — upload pliku:

1. **Walidacja** — sprawdzenie typu, rozmiaru
2. **Thumbnail** — generowanie miniaturki
3. **Skan antywirusowy** — ClamAV w kontenerze
4. **Indeksowanie metadanych** — zapis do DynamoDB
5. **Analityka** — zdarzenie do data lake

Korzyści

Dekopling: konsumenci nie wiedzą o sobie. Awaria jednego nie blokuje reszty. Każdy skaluje się niezależnie.

Orkiestracja: AWS Step Functions

Problem: złożone workflow z kolejnością, warunkami, kompensacjami

Choreografia (EventBridge) ⟶ trudna do debugowania przy 10+ krokach.

Step Functions — maszyna stanów jako JSON/YAML

- Parallel, Choice, Wait, Map (przetwarzanie batch)
- Wbudowane retry + catch + timeout
- Widoczność: wizualna mapa wykonania
- Koszt: $0,025 per 1000 przejść stanów (Standard); $0,001 (Express)

Kiedy orkiestracja, kiedy choreografia?

Orkiestracja: workflow z gwarancją kolejności, kompensacjami (saga)

Choreografia: luźno powiązane domeny, prostsze przepływy (nawiązanie do wykładu 4)

Serverless poza FaaS

Bazy danych serverless

Baza	Model	Skalowanie do zera	Opóźnienie	Koszt (10M req/mies.)
DynamoDB on-demand	NoSQL KV	Tak	< 10 ms p99	~$7,50
Aurora Serverless v2	SQL (PG/MySQL)	Min 0,5 ACU	< 10 ms	~$155
Neon	Postgres (HTTP API)	Tak	50–200 ms	~$83
PlanetScale	MySQL (Vitess)	Tak	50–200 ms	~$75

Trade-off

DynamoDB: najtańszy, ale brak SQL i JOINów. Aurora: pełny SQL, ale zawsze min. 0,5 ACU (~$43/mies.). Neon/PlanetScale: HTTP API = wyższe opóźnienie.

Platforma	Dostawca	Skalowanie do 0	Timeout	Koszt (10M req, 200 ms)
Fargate	AWS	Nie	Bez limitu	~$36/mies. (zawsze)
Cloud Run	GCP	Tak	60 min	~$17/mies.
Container Apps	Azure	Tak	30 min	~$103/mies.

Cloud Run — najlepszy stosunek elastyczności do ceny

Skalowanie do zera + do 1000 współbieżnych requestów na instancję + 60 min timeout + prosty deployment (`gcloud run deploy`).

Edge computing

Obliczenia na brzegu sieci — po co?

Problem: opóźnienie fizyczne

Prędkość światła ≈ 200 km/ms w światłowodzie. Warszawa ⟶ Virginia ≈ 40 ms RTT (same opóźnienie fizyczne, bez przetwarzania).

Rozwiązanie: przenieś logikę **bliżej użytkownika** — na serwery brzegowe (edge).

Zastosowania:

- **Personalizacja** — modyfikacja odpowiedzi bez RTT do originu
- **Testy A/B** — routing na podstawie hasha/cookie
- **Geolokalizacja** — przekierowanie do regionalnego originu
- **Uwierzytelnianie na brzegu** — walidacja JWT przed forwarding do originu
- **Optymalizacja obrazów** — resize/format on-the-fly
- **Ochrona przed botami** — challenge na brzegu, origin nie widzi ataku

Platformy edge — porównanie

Platforma	Runtime	Cold start	Lokalizacje	Wyróżnik
Cloudflare Workers	V8 isolates	< 5 ms	300+	Durable Objects, R2, zero egress
Lambda@Edge	Node.js, Python	50–200 ms	500+ (CloudFront)	Pełna integracja AWS
CloudFront Functions	JavaScript (ES 5.1)	< 1 ms	450+	Najszybsze, najtańsze, ograniczone
Vercel Edge Functions	V8 isolates	~50 ms	40+	Integracja z Next.js
Fastly Compute	WebAssembly	~50 µs	Mniej, ale wydajne	Najszybszy (Wasm AOT)

Cloudflare Workers — izolaty V8

Architektura: V8 isolates (nie kontenery)

Tysiące izolatów w jednym procesie. Każdy zajmuje 2–10 MB (vs 50–500 MB kontener). Start w ~5 ms. Brak cold startu w tradycyjnym sensie.

Ekosystem storage na brzegu:

- **Workers KV** — globalny key-value (eventually consistent)
- **Durable Objects** — koordynacja stanowa (strong consistency, SQLite)
- **R2** — object storage (S3-kompatybilny, **zero opłat za egress**)
- **D1** — SQLite na brzegu
- **Queues** — kolejki asynchroniczne

Ograniczenia

30 s CPU time. 128 MB pamięci. Brak natywnych binarek — tylko JS/Wasm. API specyficzne dla Cloudflare (vendor lock-in).

WebAssembly na serwerze

Wasm poza przeglądarką — po co?

WebAssembly (Wasm) = przenośny, bezpieczny, szybki format binarny

Izolacja (sandbox) + deterministyczne wykonanie + brak cold startu.

Cytat (Solomon Hykes, współtwórca Dockera, 2019)

„Gdyby WASI istniało w 2008, nie musielibyśmy tworzyć Dockera.“

Czym Wasm różni się od kontenerów:

- **Sandbox na poziomie modułu** — nie wymaga jądra OS
- **Start w mikrosekundach** — vs setki ms dla kontenerów
- **Rozmiar obrazu** — 1–5 MB vs 50–500 MB
- **Gęstość** — tysiące instancji na jednej maszynie

WASI — WebAssembly System Interface

WASI = standardowy interfejs Wasm do systemu operacyjnego

Bytecode Alliance (Mozilla, Fastly, Intel, Microsoft).
WASI 0.2.1 (stabilny, 2024). WASI 0.3 (async, w trakcie). WASI 1.0 (cel: późny 2026).

Kluczowe interfejsy (WASI 0.2):

- `wasi:http` — klient/serwer HTTP
- `wasi:io` — strumienie I/O
- `wasi:filesystem` — dostęp do plików
- `wasi:cli` — interfejs linii poleceń

Component Model — kompozycja między językami

WIT (Wasm Interface Types): deklaratywna definicja interfejsu.
Canonical ABI: standaryzowana konwencja wywołań.
Kompozycja: łącz moduły z Rusta, Go, Pythona w jedną aplikację.

Spin (Fermyon) — framework mikroserwisów Wasm

Spin = event-driven microservices z WebAssembly

Projekt CNCF Sandbox. Języki: Rust, Go, JS/TS, Python.

Kluczowe cechy:

- **Triggery**: HTTP, Redis, Cron, MQTT
- **Wbudowane usługi**: KV store, SQLite, outbound HTTP
- **Serverless AI**: inferencja LLM na brzegu
- **Deploy**: lokalnie, Kubernetes (SpinKube), Fermyon Cloud

Przykład konfiguracji (spin.toml)

```
[[trigger.http]]
route = "/hello"
component = "hello"
```

developer.fermyon.com/spin

Wasm vs kontenery — porównanie wydajności

Metryka	Wasm (AOT)	Wasm (JIT)	Docker (Alpine)	Docker (pełny)
Cold start	1–10 ms	20–50 ms	200–500 ms	1–2 s
Rozmiar obrazu	1–5 MB	1–5 MB	5–50 MB	100–500 MB
Pamięć / instancja	1–10 MB	1–10 MB	50–200 MB	100–500 MB
Gęstość (1000 inst.)	1–10 GB	1–10 GB	50–200 GB	100–500 GB

Wasm wygrywa

- Serverless/FaaS (start < 1 ms)
- Edge (gęstość + start)
- Systemy pluginów (sandbox)
- Multi-tenant SaaS

Kontenery wygrywają

- Długo-żyjące serwisy
- Złożone zależności (pełny OS)
- GPU (ekosystem dojrzały)
- Istniejąca infrastruktura K8s

Ograniczenia serverless

Kiedy serverless to zły wybór?

1. Vendor lock-in

EventBridge, Step Functions, DynamoDB — API specyficzne dla AWS. Migracja = przepisanie.

2. Debugowanie i obserwowalność

Rozproszone, efemeryczne środowiska. Brak SSH. Logi w CloudWatch (opóźnienie 1–5 s).
Distributed tracing konieczny.

3. Limity

Lambda: 15 min timeout, 10 GB RAM, 6 MB payload (sync), 256 MB /tmp.
Cold starty przy nieprzewidywalnym ruchu.

4. Koszty przy stałym ruchu

Lambda ≈ 3–5× droższa niż Fargate/EC2 przy ciągłym obciążeniu (break-even: 3–8M req/mies.).

Multi-cloud i przenośność

Abstrakcje serverless — próba rozwiązania lock-in

- **Knative** — rozszerzenie Kubernetes o serverless (scale-to-zero, eventing). CNCF incubating.
- **Serverless Framework** — IaC (`serverless.yml`), multi-cloud deployment
- **SST** — TypeScript-first, AWS CDK, Live Lambda Development
- **OpenFaaS** — open-source FaaS na Kubernetes/Docker, dowolny kontener

Portability vs Optimization

Abstrakcja = mianownik wspólny = utrata cech specyficznych (SnapStart, Durable Objects, Step Functions).

Realnie: większość zespołów wybiera **jednego dostawcę** i optymalizuje.

Porównanie kosztów

Koszt per-request: Lambda vs Fargate vs Cloud Run vs EC2

Scenariusz: 512 MB pamięci, 200 ms średni czas wykonania.

Platforma	100K req/mies.	10M req/mies.	100M req/mies.
Lambda	~$1,60	~$362	~$3 620
Cloud Run	~$0,40	~$101	~$1 010
Fargate (2 taski)	~$72 (stała)	~$72	~$710
EC2 (RI 1-rok)	~$34 (stała)	~$34	~$70

Wniosek

Lambda wygrywa przy **niskim, nieregularnym** ruchu (płacisz za zero = $0).
Przy stałym obciążeniu **EC2 z Reserved Instance** jest 50× tańsze.

Punkt break-even — kiedy warto zmienić model?

Lambda vs Fargate

Break-even: **3–8M requestów/miesiąc** (zależy od czasu wykonania i pamięci).
Poniżej: Lambda tańsza (pay-per-use). Powyżej: Fargate tańszy (stała pojemność).

Cloud Run vs Lambda

Cloud Run tańszy w większości scenariuszy dzięki **współbieżności** (do 1000 req/instancję vs 1 req/instancję w Lambda).

Ukryte koszty serverless

- Czas inżynierów: debugowanie, observability, cold start tuning
- Egress: Lambda w VPC + NAT Gateway = $0,045/GB
- Ekosystem: CloudWatch, X-Ray, Step Functions — dodatkowe pozycje na fakturze

Badanie AWS/Deloitte (2019): 3h/mies. overhead inżynierski = $450 — może przechylić break-even na korzyść serverless.

Case study: Design Google Drive

Wymagania i estymacja skali

Wymagania funkcjonalne

Upload/download plików. Synchronizacja między urządzeniami. Udostępnianie. Historia wersji.

Estymacja (back-of-the-envelope):

- 500M użytkowników, 10M DAU
- Średnio 2 pliki/dzień/użytkownik = 20M uploadów/dzień ≈ 230 req/s
- Średni rozmiar pliku: 500 KB
- Storage: 10 TB/dzień przyrost

Kluczowe decyzje architektoniczne

- Metadane: **strong consistency** (relacyjna baza, PostgreSQL/DynamoDB)
- Pliki: **eventual consistency** (object storage, S3/GCS)
- Sync: **event-driven**, notification push

Architektura serverless — upload pipeline

Event-driven pipeline: S3 → SNS → SQS → Lambda → DynamoDB

Upload → S3 Event Notification → SNS (fan-out) → 3 kolejki SQS:

1. **Walidacja** — typ pliku, rozmiar, skan antywirusowy
2. **Przetwarzanie** — generowanie miniaturek (PIL/Sharp), ekstrakcja metadanych
3. **Indeksowanie** — zapis do DynamoDB (hash pliku, rozmiar, owner, wersja)

Dead Letter Queue (DLQ) dla każdej kolejki — awarie nie blokują pipeline.

Serverless tu wygrywa

0 req/s w nocy = $0 kosztów. 10 000 req/s przy flash uploadsach = automatyczne skalowanie. Brak capacity planning.

Block-level sync — synchronizacja przyrostowa

Chunking — podział pliku na bloki (4 MB)

Każdy blok hashowany (SHA-256). Upload: tylko bloki, których hash nie istnieje w storage. Modyfikacja 1 MB w pliku 1 GB = upload **jednego** bloku zamiast 1 GB (**250× oszczędność**).

Deduplikacja

Jeśli hash bloku istnieje (inny użytkownik, inna wersja) — nie zapisujemy ponownie. Dropbox: **40–50% oszczędności storage** dzięki deduplikacji.

Resumable upload

HTTP Range Requests. Sesja uploadu w DynamoDB. Przerwanie ⟶ wznowienie od ostatniego bloku.

Powiadomienia o zmianach — strategie

Strategia	Opóźnienie	Skalowalność	Złożoność	Zastosowanie
Long polling	1–30 s	Średnia	Niska	Fallback
WebSocket	< 100 ms	Niska	Wysoka	Real-time (czat, kolaboracja)
SSE	< 100 ms	Średnia	Średnia	Jednokierunkowy push
Hybrid	< 100 ms	Wysoka	Średnia	Produkcja (WS → SSE → polling)

Produkcyjny wzorzec: kaskada z fallbackiem

Klient próbuje WebSocket. Jeśli nie działa (proxy, firewall) → SSE. Jeśli nie działa → long polling. At-least-once delivery + exponential backoff.

Rozwiązywanie konfliktów synchronizacji

Problem: dwóch użytkowników edytuje ten sam plik offline

Oba urządzenia mają wersję 5. Urządzenie A tworzy wersję 6a, B tworzy 6b. Kto wygrywa?

Strategie:

1. **Last Writer Wins (LWW)** — proste, deterministyczne, ale utrata danych
2. **Operational Transformation (OT)** — transformacja operacji (Google Docs)
3. **Kopie konfliktowe** — brak utraty danych (Dropbox/Google Drive: „kopia konfliktu z…“)
4. **Vector Clocks** — wykrywanie przyczynowości, identyfikacja współbieżnych zmian

Google Drive: pierwsza przetworzona wersja wygrywa; reszta jako kopie konfliktu + historia wersji.

Wasz zespół ma serwis przetwarzający **10 req/s** na co dzień, ale raz w miesiącu — **10 000 req/s** przez 2h (flash sale).

Czy wybieracie **serverless** (Lambda), **kontenery** (Fargate/Cloud Run), czy **dedykowane VM** z autoscaling?

Jakie trade-offy bierzecie pod uwagę? (koszt, opóźnienie, złożoność operacyjna, czas wdrożenia)

Podsumowanie

Kluczowe wnioski

1. **Serverless** to model odpowiedzialności, nie technologia — od IaaS do pełnego serverless
2. **FaaS** wygrywa przy zmiennym, nieprzewidywalnym ruchu; przegrywa kosztowo przy stałym obciążeniu
3. **Cold starty**: Rust/Go (16–38 ms) vs Java (698 ms); SnapStart skraca 5,6×
4. **Event-driven**: fan-out (SNS/EventBridge) dla dekouplingu; Step Functions dla workflow z gwarancjami
5. **Edge computing**: V8 isolates (Cloudflare) lub Wasm (Fastly) — logika < 5 ms od użytkownika
6. **WebAssembly**: start w mikrosekundach, 85% mniejsze obrazy, ale ekosystem jeszcze młody
7. ☁ **Koszty**: Lambda break-even z Fargate przy 3–8M req/mies.; Cloud Run najlepszy stosunek ceny do elastyczności

Źródła i lektury

- A. Xu — *System Design Interview*, rozdz. 15 (Google Drive)
- AWS Documentation — Lambda, Step Functions, EventBridge (docs.aws.amazon.com)
- M. Roberts — „Serverless Architectures“ (martinfowler.com, 2018)
- Y. Cui — *Production-Ready Serverless* (Manning)
- J. Daly — „Serverless Microservice Patterns for AWS“ (jeremydaly.com)
- Cloudflare Blog — „How Workers Works“ (blog.cloudflare.com)
- Bytecode Alliance — WASI Specification (wasi.dev)
- Fermyon — Spin Documentation (developer.fermyon.com/spin)
- github.com/maxday/lambda-perf — codzienne benchmarki cold startów
- AWS Well-Architected Serverless Lens (docs.aws.amazon.com/wellarchitected)

Następny wykład

Wykład 10: Obserwowalność, bezpieczeństwo i przyszłość systemów rozproszonych

Distributed tracing · OpenTelemetry · Zero Trust · mTLS · Platform Engineering