Obserwowalność, bezpieczeństwo i przyszłość

Observability · Zero Trust · mTLS · Platform Engineering · FinOps · Trendy

mgr inż. Jakub Woźniak

Politechnika Poznańska · Instytut Informatyki · Semestr letni 2025/26

Plan wykładu

Agenda

- Obserwowalność — monitoring vs observability, trzy filary
- Structured logging i correlation IDs
- OpenTelemetry — standard instrumentacji
- Distributed tracing i W3C Trace Context
- Metryki: RED, USE, Four Golden Signals
- Zero Trust Architecture — „never trust, always verify“
- OAuth 2.0 / OIDC, JWT, mTLS, service mesh
- Zarządzanie sekretami
- ☁ AI/ML inference, Platform Engineering, FinOps, Well-Architected
- Trendy i przyszłość systemów rozproszonych
- Case study: Design YouTube (A. Xu, rozdz. 14)
- Podsumowanie kursu

Obserwowalność

Monitoring vs Observability

Dwa różne pytania

Monitoring odpowiada na pytanie „**czy** coś jest zepsute?“ (known-unknowns).
Observability odpowiada na pytanie „**dlaczego** user 4821 zobaczył 500 ms o 16:35?” (unknown-unknowns).

Monitoring

- Predefiniowane dashboardy
- Znane scenariusze awarii
- Low-cardinality (region, status)
- Skaluje się słabo z liczbą serwisów

Observability

- Eksploracja ad-hoc
- Emergentne, nieprzewidziane interakcje
- High-cardinality (trace_id, user_id)
- Skaluje się z złożonością

W mikroserwisach 1 request = 10+ wywołań = kombinatoryczna eksplozja trybów awarii (nawiązanie do wykładu 6).

Trzy filary obserwowalności (+ czwarty)

Filar	Co to jest	Model kosztu
Logs	Dyskretne zdarzenia z timestampem (najlepiej JSON)	Per-event
Metrics	Agregaty numeryczne (counter, gauge, histogram)	Per-unikalna-seria
Traces	Ścieżka requestu przez serwisy (spans)	Per-trace
Profiles	Próbki CPU/pamięci skorelowane ze spanami (eBPF)	Per-sample

Czwarty filar — Continuous Profiling

OpenTelemetry Profiles: sygnał w fazie alpha (2026). Integracja eBPF, overhead 1–3% CPU. Korelacja profilu z `trace_id` / `span_id`.

Structured logging — JSON, nie tekst

Dlaczego JSON > plain text

- Parsowanie 10–100× szybsze (brak regexów)
- Każde pole niezależnie filtrowalne i agregowalne
- High-cardinality wartości (`trace_id`) w polach, nie w etykietach
- Lepsza kompresja (storage kolumnowy 3–5× wydajniejszy)

Minimalny zestaw pól w produkcji

`timestamp` (ISO 8601 UTC), `level`, `service`, `version`, `environment`, `event`, `trace_id`, `span_id`, `message`, `error.{type,code}`, `context.{...}`

Nie loguj danych wrażliwych

Nigdy: hasła, tokeny, PII (email zamiast `user_id`). Logi trafiają do wielu systemów.

Correlation ID — szycie requestu przez serwisy

Jeden identyfikator na całe żądanie

Generowany na **granicy systemu** (API Gateway, wejście do kolejki). Wszystkie logi, spany i zdarzenia w obsłudze jednego requestu dzielą ten sam `trace_id`.

Propagacja:

- **HTTP**: nagłówek `traceparent` (W3C) lub `X-Request-ID`
- **gRPC**: metadata w kontekście
- **Kolejki**: nagłówki wiadomości (Kafka, SNS, RabbitMQ)
- **SQL**: komentarz w zapytaniu `/* trace_id: xyz */`

SDK OpenTelemetry **automatycznie** wstrzykuje `trace_id` i `span_id` do logów, gdy aktywny jest span.

OpenTelemetry

OpenTelemetry — dominujący standard instrumentacji

Konsolidacja standardów

2017–2019: OpenTracing (CNCF) vs OpenCensus (Google) → 2019: fuzja w **OpenTelemetry** → 2023: traces + metrics stabilne → 2026: projekt **graduated** (poziom K8s, Prometheus).

Dlaczego dominuje:

- Vendor-neutral — niezależny od dostawcy
- Wspierany przez wszystkich hyperscalerów (AWS, GCP, Azure) i niezależnych (Datadog, Grafana, Honeycomb)
- OTLP jako de-facto format eksportu dla każdego backendu
- Auto-instrumentacja popularnych frameworków bez zmian w kodzie

opentelemetry.io/status · CNCF: 12 300+ kontrybutorów z 2 800+ firm

Architektura OTel: API → SDK → Collector → Backend

Cztery warstwy

1. **API** — niezależne od języka interfejsy (tworzenie spanów, metryk, logów)
2. **SDK** — implementacja per język; samplowanie, batching, eksportery
3. **Collector** — vendor-neutral; receivers → processors → exporters
4. **OTLP** — protokół: gRPC (`:4317`) lub HTTP/Protobuf (`:4318`)

Sygnał	Status (2026)
Traces	Stabilny
Metrics	Stabilny
Logs	Stabilny
Profiles	Alpha (eBPF, marzec 2026)

Distributed tracing

Span, trace, relacja parent-child

Anatomia

Span = pojedyncza operacja. Zawiera: `span_id` (64-bit), `trace_id` (128-bit), `parent_span_id`, `name`, `status`, `attributes`, `events`, czasy z dokładnością ns.

Trace = zbiór spanów połączonych tym samym `trace_id`. **Root span** — bez rodzica.

W3C Trace Context — nagłówek `traceparent`

```
00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01
```

`version` (2 hex) · `trace-id` (32 hex) · `parent-id` (16 hex) · `flags` (sampled)

W3C Trace Context Level 2 — w3.org/TR/trace-context

Sampling — head-based vs tail-based

Cecha	Head-based	Tail-based	Uwagi
Decyzja	Przy tworzeniu spanu	Po zakończeniu trace	—
Stan	Bezstanowe	Collector buforuje	Pamięć
Błędy	Może zgubić	Gwarancja 100%	Kluczowe
Typowa stopa	1–5% przy >10k RPS	100% błędy + 1% reszta	Hybryda

Po co samplować?

Koszt i wolumen. 100% sampling = ~$480/mies. (10M req/dzień). 1% = ~$4,80.

Tail sampling: 100% błędów + latency >2s, 1% reszty = oszczędność 80–90% przy zachowaniu debugowalności.

Narzędzia: Jaeger (Elasticsearch/Cassandra), Grafana Tempo (object storage, TraceQL), Zipkin, AWS X-Ray.

Metryki

RED, USE, Four Golden Signals

Metoda	Sygnały	Zakres	Autor
RED	Rate, Errors, Duration	Serwisy request-driven	T. Wilkie (Grafana)
USE	Utilization, Saturation, Errors	Zasoby (CPU, RAM, dysk)	B. Gregg
Golden Signals	Latency, Traffic, Errors, Saturation	Systemy user-facing	Google SRE

Wzorzec praktyczny

RED na warstwie serwisów + USE na warstwie infrastruktury = Golden Signals w obu.

Stos de-facto: **Prometheus** (zbieranie) + **Grafana** (wizualizacja).

Google SRE Book, rozdz. 6 · brendangregg.com/usemethod.html

Cardinality explosion — problem skalowalności Prometheusa

Jedna zła etykieta = miliony serii

`http_requests_total{method, status, user_id=<10M>}` = miliardy serii czasowych. Każda unikalna kombinacja nazwa + etykiety = jedna seria.

Skutki:

- Wzrost zużycia pamięci (każda seria indeksowana)
- Spowolnienie zapytań, eksplozja storage
- Prometheus **OOMKilled** przy >5–10M serii

Zakazane etykiety (nieograniczona kardynalność)

`user_id`, `request_id`, `trace_id`, `url` z parametrami, `error_message`.

Reguła: < 1000 unikalnych wartości per etykieta. High-cardinality ⟶ logi/trace, nie metryki.

Obserwowalność a SLO (nawiązanie do wykładu 1)

Obserwowalność zasila pomiar SLO

1. **Mierz SLI** — z metryk/trace liczysz realny p99, error rate
2. **Wykrywaj burn rate** — czy zużywasz error budget za szybko?
3. **Root cause** — gdy SLO złamane, trace + logi wskazują który serwis
4. **Feedback** — metryki incydentów → korekta SLO i alertów

eBPF — auto-instrumentacja bez kodu (2026)

OpenTelemetry eBPF Instrumentation (OBI): HTTP/gRPC tracing + RED metrics, overhead 1–3% CPU. 67% klastrów K8s używa narzędzia eBPF (CNCF Q1 2026).

Bezpieczeństwo

Zero Trust — „never trust, always verify“

NIST SP 800-207 (2020)

Brak domyślnego zaufania do zasobów na podstawie **lokalizacji sieciowej** lub właściciela. Każde żądanie — z biura czy z domu — wymaga uwierzytelnienia, autoryzacji i ciągłej walidacji.

Model perymetrowy (castle-and-moat) jest niewystarczający

Chmura, praca zdalna, mikroserwisy, efemeryczne IP w K8s — sieci nie da się już „obwarować“. Granicą jest **tożsamość**, nie adres IP.

Tenety NIST:

- Wszystko to zasób
- Komunikacja zawsze szyfrowana
- Dostęp per-sesja

Mechanizm:

- PDP (Policy Decision Point)
- PEP (Policy Enforcement Point)
- Dynamiczna polityka kontekstowa

Google BeyondCorp — zero udanych ataków phishing po wymuszeniu kluczy sprzętowych (2017)

OAuth 2.0 vs OpenID Connect

Aspekt	OAuth 2.0	OpenID Connect
Cel	Autoryzacja (delegowany dostęp)	Uwierzytelnianie (tożsamość)
Pytanie	„Co aplikacja może?“	„Kto to jest?“
Token	Access + refresh token	**1.** ID token (zawsze JWT)
Standard	RFC 6749	Warstwa nad OAuth 2.0

Cztery role OAuth

Resource Owner (user) · Client (aplikacja) · Authorization Server (wydaje tokeny) · Resource Server (chroni zasób).

OAuth 2.1 — konsolidacja best practices

Cecha	OAuth 2.0	OAuth 2.1
PKCE	Opcjonalne (mobile)	**Obowiązkowe** dla wszystkich
Implicit grant	Dozwolony	**Usunięty**
Password grant (ROPC)	Dozwolony	**Usunięty**
Redirect URI	Wildcardy	Dokładne dopasowanie
Bearer w query string	Dozwolony	Tylko nagłówek/body

Authorization Code + PKCE — domyślny przepływ

`code_challenge = SHA256(code_verifier)`. Kod bezużyteczny bez `code_verifier` → ochrona przed przechwyceniem kodu.

Service-to-service: **Client Credentials** (JWT + mTLS zamiast gołego sekretu).

JWT — struktura i pułapki

`header.payload.signature` (base64url) — RFC 7519

Claims: `iss`, `sub`, `aud`, `exp`, `iat`, `jti`.
Podpis: HS256 (symetryczny, współdzielony sekret) lub RS256/ES256 (asymetryczny — klucz prywatny tylko u wydawcy).

Klasyczne podatności

1. **`alg: none`** — biblioteka akceptuje token bez podpisu (CVE-2015-9235)
2. **Algorithm confusion** — RS256 → HS256, klucz publiczny jako sekret HMAC
3. **Brak walidacji `exp`/`aud`** — token z innego serwisu akceptowany
4. **localStorage** — kradzież tokenu przez XSS

Best practice

Krótki `exp` (15 min) + rotacja refresh tokenów. Waliduj **wszystko** (`algorithms: ['RS256']`, `aud`, `iss`, `exp`). Token w httpOnly cookie, nie localStorage.

Mutual TLS — obie strony przedstawiają certyfikat

Zwykły TLS: serwer dowodzi tożsamości. mTLS: **klient też**. Spoofing IP/DNS nie wystarczy — potrzebny certyfikat workloadu.

SPIFFE/SPIRE — standaryzowana tożsamość

`spiffe://prod.example.com/ns/payments/sa/stripe-integration`

SVID (X.509 lub JWT) wydawany per workload. Klucz prywatny **nigdy** nie opuszcza poda. Automatyczna rotacja co ~5 min.

Problem rotacji certyfikatów

X.509 mają sztywny termin ważności. Ręczna rotacja w skali = źródło awarii. SPIRE atestuje workload (K8s ServiceAccount) i rotuje automatycznie.

Service mesh — mTLS „za darmo“

Cecha	Istio sidecar	Istio ambient	Linkerd
mTLS	Automatyczny	Automatyczny	Automatyczny
Proxy	Envoy per pod	ztunnel per node	micro-proxy (Rust)
Pamięć/pod	~50 MB	~0,5 MB ztunnel	~1 MB
Narzut mTLS	99–166%	8–33%	~33%

Istio Ambient Mesh (GA v1.24, XI 2024)

Zmiana modelu: bez sidecarów. **ztunnel** (L4 mTLS, per node) + opcjonalny **waypoint** (L7) per namespace. Oszczędność pamięci >90% vs sidecar. Inkrementalna adopcja (nawiązanie do service mesh z ZSR).

Linkerd 2.19 (2025): kryptografia post-kwantowa ML-KEM-768.

Zarządzanie sekretami

Czego nie robić

Hardcode w kodzie · commit do gita (nieodwracalny wyciek) · K8s Secrets bez szyfrowania (base64 ≠ szyfrowanie; plaintext w etcd).

HashiCorp Vault — dynamiczne sekrety

Zamiast przechowywać statyczne hasło — **generuj** unikalne, krótkożyjące poświadczenia na żądanie (TTL 1h). Automatyczne odwołanie po wygaśnięciu. Per-serwis = audyt + mały blast radius.

Cloud:

- AWS Secrets Manager + KMS
- GCP Secret Manager
- Azure Key Vault

Kubernetes:

- Szyfrowanie at-rest (KMS provider)
- External Secrets Operator
- SOPS / sealed-secrets (GitOps)

Defense in depth i confused deputy

Warstwy obrony

1. **API Gateway** — authN na brzegu, rate limiting, walidacja schematu
2. **Service mesh** — mTLS, AuthorizationPolicy L4/L7
3. **Logika aplikacji** — fine-grained uprawnienia (czy user **posiada** zasób?)
4. **Warstwa danych** — szyfrowanie at-rest, row-level security

Problem confused deputy

User A (niskie uprawnienia) nakłania serwis X (wysokie) do wywołania serwisu Y w jego imieniu.
Ochrona: token związany z zasobem (`aud`, `scope`), token exchange (zawężenie scope), least privilege.

☁ Chmura, AI i nowe dyscypliny

AI/ML inference jako problem systemu rozproszonego

Wąskie gardło 2025/26 to nie trening, lecz serving

Miliardy tokenów dziennie, latencja w ms. Koszt napędzany przez **throughput tokenów**, **KV-cache**, **głębokość kolejki** — nie przez CPU/RAM.

Narzędzie	Rola	Innowacja
vLLM	Silnik inferencji LLM	PagedAttention, continuous batching
KServe	Orkiestracja na K8s (CRD)	KV-cache aware routing, scale-to-zero
NVIDIA Triton	Heterogeniczna flota	TensorRT-LLM backend
AI Gateway	Warstwa przed modelem	Semantic caching, token rate limiting

Wzorce rozproszone: tensor/data/expert parallelism, disaggregated prefill-decode, RAG jako pipeline (embedding → vector store → retriever → LLM).

Platform Engineering — platforma jako produkt

Od „zespołu DevOps“ do Internal Developer Platform (IDP)

Cognitive load — nie możliwości infrastruktury — jest realnym ograniczeniem. Platforma self-service z **golden paths** (paved roads) redukuje obciążenie zespołów stream-aligned.

Backstage (Spotify, CNCF)

- Software Catalog (kto czego właściciel)
- Software Templates (scaffolding < 5 min)
- TechDocs (docs-as-code)
- 200+ pluginów

Team Topologies

- Stream-aligned (e2e domena)
- Platform (X-as-a-Service)
- Enabling (czasowe wsparcie)
- Complicated-subsystem

M. Skelton, M. Pais — Team Topologies *(2019)* · *backstage.io*

FinOps — koszt jako obywatel pierwszej klasy

Framework FinOps: Inform → Optimize → Operate

Widoczność kosztu per workload → rightsizing i eliminacja marnotrawstwa → dyscyplina przez automatyzację i governance.

Skala marnotrawstwa (2026)

~29% wydatków IaaS/PaaS to marnotrawstwo. Idle compute (35%) + overprovisioning (25%). Średnia utylizacja GPU: 23%. K8s: requesty napędzają rachunek, nie realne użycie.

Narzędzia:

- OpenCost (CNCF)
- Kubecost
- CAST AI (auto-optimize)

W praktyce:

- FinOps ma **widoczność**, rzadko **władzę**
- Działa: auto-shutdown z grace period
- Unit economics: koszt/request

Well-Architected Framework — sześć filarów

AWS Well-Architected — lekcje całego kursu w jednym frameworku

1. **Operational Excellence** — bezpieczne i przewidywalne wdrożenia
2. **Security** — ochrona danych i systemów
3. **Reliability** — poprawne działanie mimo awarii (wykład 6)
4. **Performance Efficiency** — efektywne użycie zasobów
5. **Cost Optimization** — wartość przy rozsądnym koszcie (FinOps)
6. **Sustainability** — minimalizacja śladu środowiskowego (dodany 2021)

Azure Well-Architected i Google Cloud Architecture Framework — analogiczne. Procesory ARM (Graviton) = 60% mniej energii vs x86.

Trendy i przyszłość

Emerging trends — co dojrzewa w 2025/26

Trend	Istota
eBPF / Cilium	Sieć + observability + security w jądrze; ~2× throughput vs iptables
WebAssembly server-side	WASI 0.2 stabilny; edge serverless, pluginy; cold start w µs
Durable execution	Temporal, AWS Step Functions, pg_durable — workflow przeżywa awarie
Modular monolith	Odwrót od „mikro-wszystkiego"; right-sizing zamiast 100 serwisów
Distributed SQL	CockroachDB, YugabyteDB, TiDB — skala zapisu + silna spójność
Data Mesh	Zdecentralizowana własność danych; data jako produkt

Prawo Conwaya — organizacja kształtuje architekturę

Conway (1968)

„Organizacja projektująca system wytworzy projekt, którego struktura kopiuje strukturę komunikacji tej organizacji.“

Inverse Conway Maneuver

Świadomie projektuj **strukturę zespołów**, by uzyskać pożądaną architekturę.
Chcesz mikroserwisy? ⟶ zespoły stream-aligned per domena.
Chcesz modularny monolit? ⟶ zespoły we wspólnym repo.

Wahadło: centralizacja ↔ decentralizacja

Lata 2010: decentralizacja (mikroserwisy). Lata 2020: re-centralizacja governance (platform teams, policy-as-code). 2026: hybryda — centralne standardy, zdecentralizowana własność.

Ponadczasowe zasady — przetrwają cykle trendów

Trwałe zasady projektowania

1. **„Nie ma rozwiązań, są tylko trade-offy“** (T. Sowell) — artykułuj kompromis jawnie
2. **Choose Boring Technology** (D. McKinley) — innowuj w biznesie, nie w infrastrukturze
3. **Design for failure** — każdy komponent zawiedzie (wykład 6)
4. **Dane przeżywają kod** — przepiszesz kod 5–10×, format danych zostaje
5. **Fallacies of Distributed Computing** nadal prawdziwe (wykład 6)

Systemy, które wygrywają: architektura techniczna odzwierciedla granice organizacyjne, infrastruktura jest na tyle prosta, by ją obsłużyć, koszt jest widoczny i ma właściciela.

Case study: Design YouTube

Wymagania i skala (A. Xu, rozdz. 14)

Wymagania funkcjonalne

Upload wideo. Streaming/oglądanie. Zmiana rozdzielczości. Niska latencja. Wsparcie mobile i web.

Estymacja (back-of-the-envelope, nawiązanie do wykładu 1):

- 5 mld użytkowników, 800 mln DAU
- 5 wyświetleń/dzień/user $\longrightarrow$ ~46 000 wyświetleń/s
- Upload:wyświetlenia = 1:200
- Storage: 500h wideo/min uploadowane

Łączy wszystkie wykłady

To synteza: estymacja, architektura, komunikacja, dane, odporność, serverless, obserwowalność, bezpieczeństwo.

Pipeline uploadu i transkodowania

Upload → object storage (S3) → kolejka → DAG transkodowania (równolegle: rozdzielczości, formaty, thumbnails) → CDN.

Wyzwanie	Rozwiązanie (wykład)
Streaming z niską latencją	CDN + adaptive bitrate (DASH/HLS) — w.9 edge
Transkodowanie w skali	DAG zadań, kolejka, fan-out — w.4, w.9
Metadane vs pliki	Strong vs eventual consistency — w.5
Odporność pipeline	DLQ, retry, idempotencja — w.6
Koszt storage	Tiering (hot/cold), deduplikacja — w.9, FinOps

Wasz system mikroserwisowy ma 40 serwisów. Production incident: latencja p99 wzrosła z 200 ms do 4 s, ale żaden dashboard nie pokazuje błędów.

Jak użyjecie obserwowalności (traces, metrics, logs), by znaleźć przyczynę?

Od czego zaczniecie — i dlaczego akurat od tego?

Źródła

Źródła i lektury

- M. Kleppmann — *Designing Data-Intensive Applications* (O'Reilly, 2017)
- S. Newman — *Building Microservices*, 2nd ed. (O'Reilly, 2021)
- A. Xu — *System Design Interview*, rozdz. 14 (YouTube)
- Google SRE Team — *SRE Book* (Golden Signals, error budgets)
- M. Skelton, M. Pais — *Team Topologies* (2019)
- B. Gregg — *Systems Performance*, 2nd ed. (USE Method)
- OpenTelemetry — opentelemetry.io/docs
- W3C Trace Context — w3.org/TR/trace-context
- NIST SP 800-207 — Zero Trust Architecture
- OAuth 2.1 — oauth.net/2.1 · SPIFFE — spiffe.io
- FinOps Foundation — finops.org · AWS Well-Architected — aws.amazon.com/architecture