Mastering Observability

Weniger Sammeln, mehr Verstehen

Severin Neumann
Juraci Paixão Kröhling

Über uns

Juraci Paixão Kröhling

Gründer, CEO @ OllyGarden

OTel Governance Commitee Member

Severin Neumann

OTel Governance Commitee Member

Was ist eigentlich Observability?

Was ist eigentlich Observability?

Zielsetzung

Ein einheitliches Verständnis für ein gemischtes Publikum.

Definition

„Die **Fähigkeit**, Fragen zu stellen und Antworten zu erhalten.“

Fundament

Telemetrie ist die Basis für alle Antworten

- Logs
- Metrics
- Traces

Wofür brauchen wir Observability?

Tag 0: Erkenntnis ohne Lösung

Unsere Services haben ein Problem, wir können es aber nicht lösen, da wir keine Antworten für unsere Fragen haben: Was hat sich geändert? Welche Symptome hängen zusammen? …

Tag 1: Der Bedarf an Telemetrie

Um Antworten zu finden, brauchen wir Telemetrie! Aber wir brauchen die **richtige** Telemetrie.

Der Weg zum Verständnis

Wie machen wir das? Genau darum geht’s in diesem Vortrag.

Was hat das mit OpenTelemetry zu tun?

Was hat das mit OpenTelemetry zu tun?

Das Backend-Problem
Logs, Metrics, Traces benötigen Formate, die vom Backend verstanden wird, um Fragen stellen zu können.

Proprietäre Fragmentierung
Formate sind proprietär oder auf einzelne Signale und Programmiersprachen beschränkt.

Mangelnde Skalierbarkeit
Das funktioniert für Monolithen, aber nicht für Applikationen aus vielen Services und Agenten.

Die Lösung: Ein offener Standard
Wir brauchen einen Standard, der offen und technologieübergreifend ist und Telemetrie verknüpft

OpenTelemetry: Der offene Standard für Telemetrie

- ☑️ **Offen** (Anbieter-Neutral)
- ☑️ **Technologie-übergreifend** (Technologie-Neutral)
- ☑️ **Vernüpfte Signale** (im Services, über Service-Grenzen hinweg)

OpenTelemetry: Der offene Standard für Telemetrie

Universal & Modern (MainFrame, Monolith, cloud-native, AI-native, ...)

Semantic Conventions (Standardisierte Benennung von Daten)

Framework & Tools (SDKs, Instrumentation Libraries ,Collector, Injector, Documentation, ...)

Neue Probleme!

Datenflut & Datenhorten
Angst-getriebenes Sammeln von Daten ohne klaren Fokus.

Fehlende Konventionen
Teams arbeiten ohne gemeinsame Standards und Absprachen.

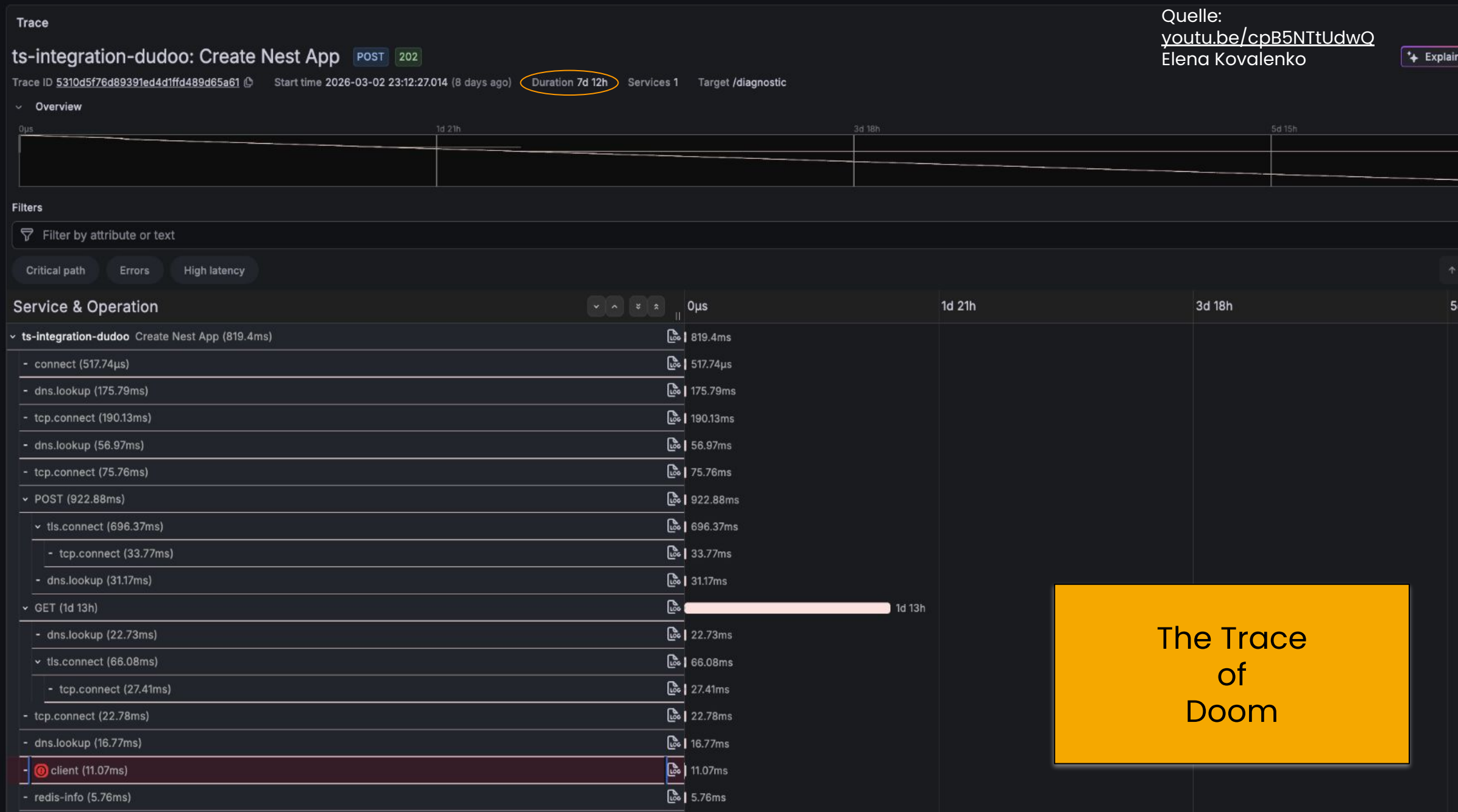
Kritische PII-Daten
Personenbezogene Informationen landen dort, wo sie nicht hingehören.

Mehr Daten als je zuvor.

Trotzdem keine Antworten.

Symptome In der Praxis

Quelle:
youtu.be/cpB5NTtUdwQ
Elena Kovalenko



The Trace
of
Doom

Nutzlose Logs & Traces

Dedup by Severity

Severity	Total	Unique	Savings	Reduction %
(empty)	22,198	4,636	17,562	79.1%
INFO	16,791	6,550	10,241	61.0%
ERROR	3,701	1,739	1,962	53.0%
Warning	3,234	78	3,156	97.6%
Normal	1,279	410	869	67.9%
DEBUG	476	4	472	99.2%
WARN	350	161	189	54.0%

4. Health Check Span Noise (Medium)

Top span names include health/readiness checks:

- `GET /health/readiness` (499 spans, pipeline-trigger)
- `GET /actuator/health/**` (244 spans, monitor)
- `GET /health` (210 spans, pipeline-controller)
- `GET /health/liveness` (166 spans, pipeline-trigger)
- `GET /actuator/prometheus` (102 spans, monitor)

Total: ~1,221 spans (2.7%) are health/metrics check noise.

Nutzlose Attribute

Attribute Size Analysis

Top Resource Attributes by Size

Attribute	Avg Chars	Max Chars	Occurrences
k8s.pod.annotation.cnpg.io/podSpec	2,048	2,048	30,985
process.command_args	710	1,129	56,107
k8s.pod.annotation.ad.VENDOR.com/postgres.checks	376	389	28,808
k8s.pod.annotation.ad.VENDOR.com/rabbitmq.checks	364	364	35,376
k8s.pod.annotation.ad.VENDOR.com/proxy.checks	307	307	11,058
k8s.pod.annotation.ad.VENDOR.com/elasticsearch.checks	209	304	19,492
k8s.pod.annotation.ad.VENDOR.com/rabbitmq.logs	207	207	35,376
k8s.pod.annotation.ad.VENDOR.com/keda-admission-webhooks.checks	147	147	896
k8s.pod.annotation.ad.VENDOR.com/keda-operator.checks	147	147	2,288
k8s.pod.annotation.ad.VENDOR.com/keda-operator-metrics-apiserver.checks	147	147	936

Attribute Chaos

```
service.name = --_name:_getbyorderidsandstatusandexpireat_:many_select_order_id_status_created_at_updated_at_st

service.name = 169.254.169.254

service.name = net/http

service.name = aws.s3

service.name = myapp-mongo

service.name = myapp-rest-validate-poc-<k8s-pod-hash>

service.name = java-aws-sdk

service.name = production_<region>
```

Quelle:
youtu.be/cpB5NTtUdwQ
Elena Kovalenko

Direkte Kosten. Indirekter Schaden.

Direkte Kosten

Mehr Telemetry benötigt mehr Infrastructure und Wartungskosten

- Storage,
- Egress,
- Ingestion,
- Verarbeitung,
- Wartung

Indirekter Schaden

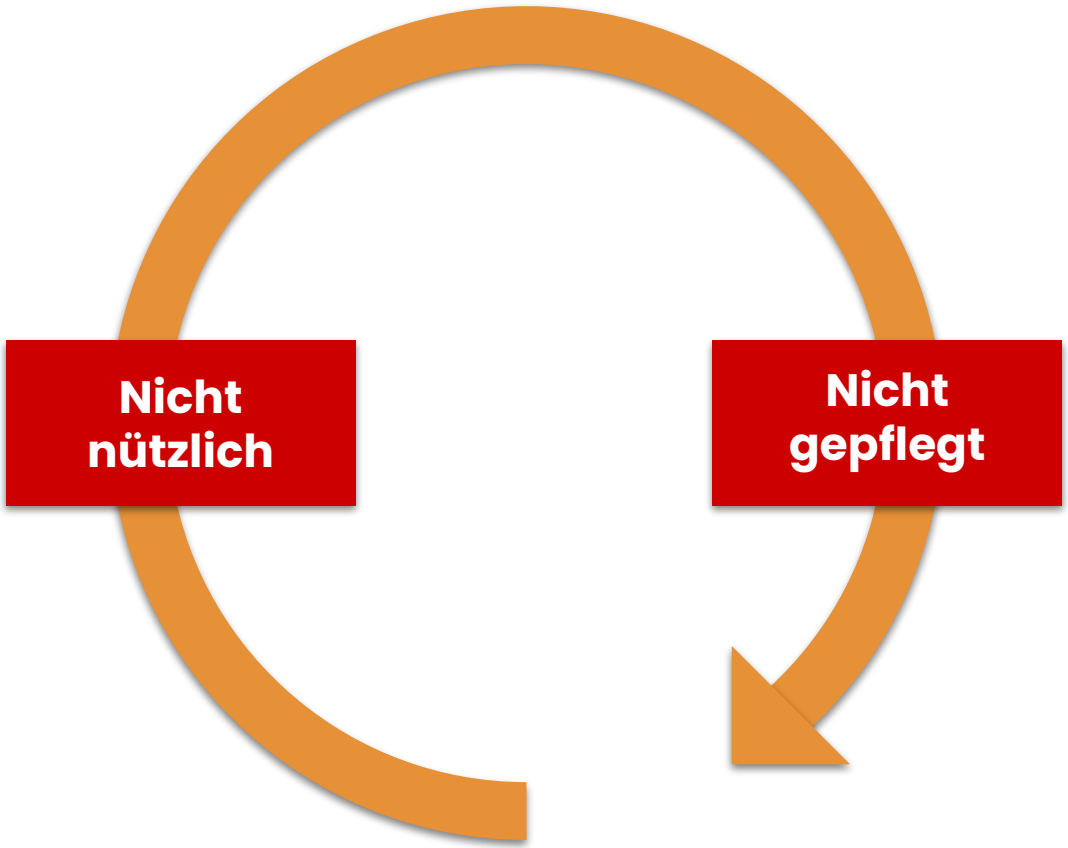
Während Incidence Response müssen wir erst das Rauschen filtern, bevor wir aktiv werden können

→ MTTR geht rauf
→ Ausfallzeiten höher
→ Kosten höher

Vertrauensverlust

Telemetrie beantwortet nie die richtigen Fragen, Teams ignorieren sie

Ein Teufelskreis



Kein Fortschritt!

Telemetrie wird schlechter und schlechter

Incident Respone wird schwerer

Kosten werden höher

Teams sind im “Fire Fighting”-Modus gefangen

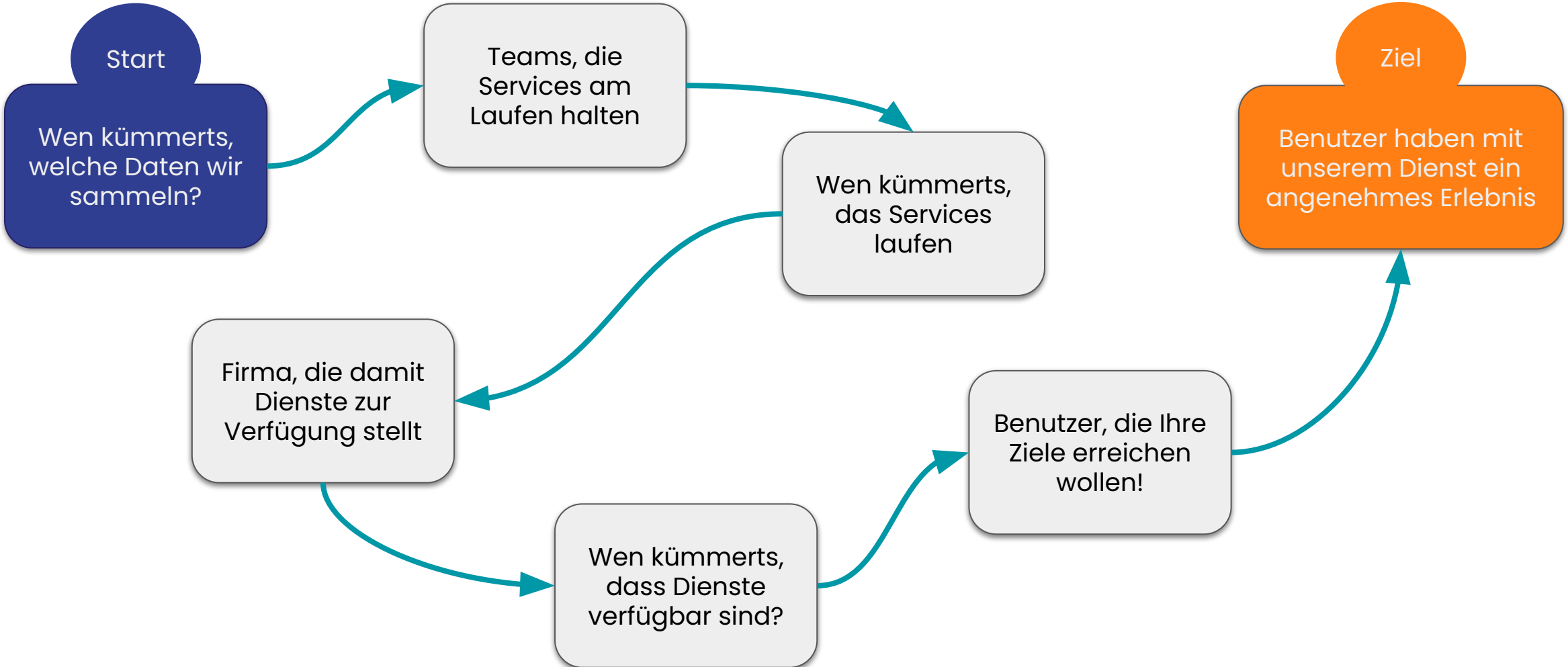
Observability ist ein Engineering Problem

YOU ARE AN ENGINEER. BEHAVE LIKE ONE.

Wer nicht weiss, warum er Daten sammelt, sammelt die falschen Daten.

Der Ausweg beginnt nicht mit besseren Tools, sondern mit einer anderen Frage.

Wen kümmerts?



Die andere Frage

Die Frage ist nicht: **“Was können wir alles sammeln?”**

Sondern: **“Was muss ich wissen, damit meine Nutzer ihr Ziel zuverlässig erreichen können?”**

Das 2-Uhr-Szenario

Welche Frage würde ich stellen, wenn meine Services down sind und Nutzer in Asien nicht bedient werden können?

Die Telemetrie-Prüfung

Kann ich diese Frage mit meiner derzeitigen Telemetrie überhaupt beantworten?

Das ist Engineering:
Top-down statt bottom-up, **Zweck-orientiert** statt angstgetrieben

Daten aus dem richtigen Grund sammeln

Logs & Events (inkl. Life Cycle Events): Sind belastbare Hinweise, kein Rauschen.

Nur speichern, wenn sie „Evidenz" liefern. Immer mit Traces oder Metrics verknüpfen.

Schlechtes Beispiel

Nichts-sagender Log Eintrag:

```
[Error] Transaction failed
```

Kein Kontext, kein Hinweis auf die Ursache.

Gutes Beispiel

Log Eintrag mit Mehrwert:

```
Trace id: …
Span id: …
Severity text: "ERROR"
body: "db connection pool exhausted"
```

Präzise Fehlerbeschreibung, die zur Lösung führt.

Fokus auf **Aktion**: Daten müssen Fakten belegen, nicht nur Platz belegen.

Daten aus dem richtigen Grund sammeln

Metrics: Abweichungen vom Sollzustand messen

Guter Anfang: Golden Signals
Latency, Traffic, Errors, Saturation

Top-Down Strategie
Kritische Transaktionen, Service-Boundaries, Services, Infrastructure

Alerting: Zweckorientiert & Actionable

- Nur für Metriken Alerts erzeugen, die direkt die **Zuverlässigkeit der Nutzerziele** betreffen.
- Ein Alert muss **actionable** sein. Verknüpfung mit anderer Telemetrie ist essentiell für die Ursachenanalyse.

Fokus auf **Relevanz**: Weniger Rauschen, mehr Klarheit durch kontextbezogene Metriken.

Daten aus dem richtigen Grund sammeln

Traces: Geschäftskontext & Transaktionsfluss abbilden

Strategisches Sampling

Fokus auf kritische Kundentransaktionen. Alles andere selektiv sampeln.

Auto-Instrumentierung

Ideal für Dependencies, Supporting Services oder Services außerhalb des aktuellen Fokus.

Code-based Instrumentation

- Ermöglicht die korrekte Abbildung des spezifischen Geschäftskontexts.
- Best Practice: Business-relevante Spans und Attribute definieren.

Fokus auf **Geschäftswert**: Traces müssen die Realität der Nutzerinteraktion widerspiegeln.

Semantic Conventions: Die drei Regeln

Semantik verbindet alle Daten, um Sinn in ihnen zu erkennen.

Konsistenz

Alle verwenden dieselben Wörter auf dieselbe Weise

```
deployment = “prod”

env = “live”

deployment.environment.name = “production”
```

Einmaligkeit

Ein Name bezeichnet exakt eine **realle** Sache

- Service A: `service.name = “api”`
- Service B: `service.name = “api”`

- Service A: `service.name = “checkout-api”`
- Service B: `service.name = “product-api”`

Beständigkeit

Identitäten ändern sich nicht, wenn das System sich ändert

```
service.name = “checkout-v1”

service.name = “checkout-v2”

service.name = “checkout”
service.version = “v1”

service.name = “checkout”
service.version = “v2”
```

Wie sammele ich die richtige Telemetrie?

Was mache ich wenn ich nicht die richtige Telemetrie gesammelt haben?

Die Hoheit über die eigenen Daten

Die wichtigste Regel

Vertraue nicht darauf, dass wir (Experten) wissen, was DU brauchst.

Telemetry Governance

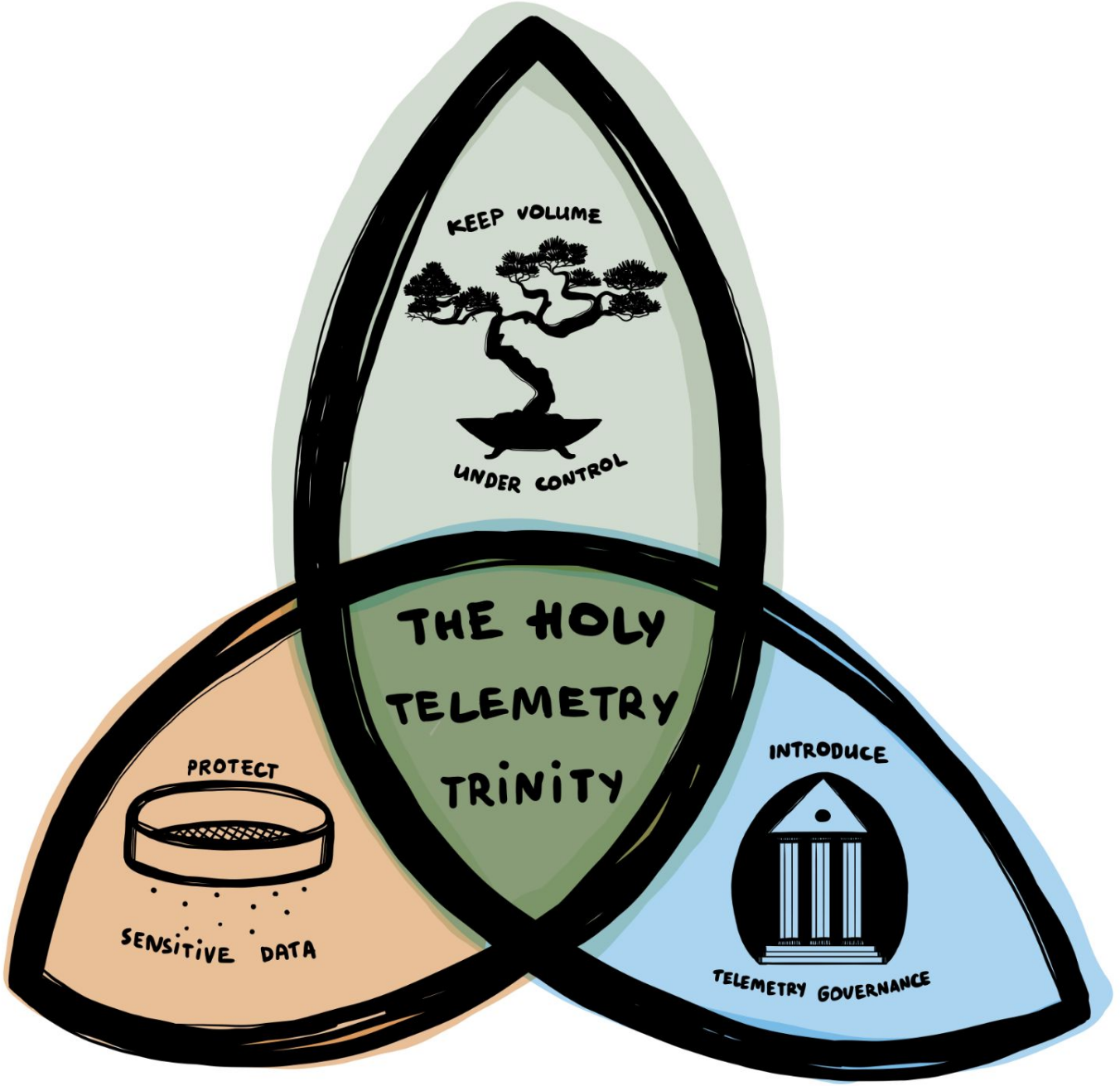
Du musst definieren, was du benötigst, um „die Frage“ zu beantworten. Tools wie **OpenTelemetry Weaver** helfen dabei.

1. Manuelle Instrumentierung

Ich sammle gezielt das, was ich wirklich brauche.

2. Tuning der Auto-Instrumentierung

Kontrolle über Infra-Monitoring und Non-Core-Services mittels Collector, KI-Hilfe und Werkzeugen.



Quelle:
youtu.be/cpB5NTtUdwQ
Elena Kovalenko

Collector-Level: Schnelle Gewinne

- Attribut-Reduktion
  - Real-Audit: 236 Resource-Attribute, gebraucht: 85.
- Log-Deduplizierung
  - Connection-Retries: 300 Einträge → 1 Eintrag pro Vorfall.
- PII-Redaktion
  - JVM-Trust-Store-Passwörter
  - Steuernummern
  - JWTs
  - Alles schon im Wild gefunden…

Governance auf SDK-Ebene & KI gestützt

- Declarative SDK Config
  - YAML statt Boilerplate, einheitlich über alle Services.
- SDK Wrapper
  - Eine Init-Zeile, alle Standards (Resource, Sampling, Exporters) implizit.
- Agentic Instrumentation Review
  - Agenten lesen Code + Telemetrie, prüfen gegen Conventions, schlagen Fixes vor.
- OTel Weaver

Service Reliability Automation

Service Reliability Automation

Kosteneffizienz

Bessere Telemetrie senkt nicht nur unsere Kosten.

Signalqualität

Sie reduziert die "Noise-To-Signal-Ratio" erheblich.

Governance & KI

Semantic Conventions erhöhen das Verständnis für Mensch & KI!

Das ist eine essentielle Grundlage für gute Service Reliability Automation.

Mehr dazu im Vortrag: **"AI für SRE richtig einsetzen"**

```
2024-03-15 10:24:01 INFO  [http] POST /api/v1/orders started
2024-03-15 10:24:01 INFO  [auth] Authentication successful
2024-03-15 10:24:01 INFO  [orders] Order validated
2024-03-15 10:24:01 INFO  [inventory] Checking stock for product 8842
2024-03-15 10:24:04 ERROR [inventory] Request to inventory-svc timed out
2024-03-15 10:24:04 INFO  [inventory] Retrying (attempt 2/3)
2024-03-15 10:24:07 ERROR [inventory] Request to inventory-svc timed out
2024-03-15 10:24:07 INFO  [inventory] Retrying (attempt 3/3)
2024-03-15 10:24:10 ERROR [inventory] Request to inventory-svc timed out
2024-03-15 10:24:10 ERROR [orders] Order failed
2024-03-15 10:24:10 INFO  [http] Response sent: 503
2024-03-15 10:24:11 INFO  [http] POST /api/v1/orders started
2024-03-15 10:24:11 INFO  [auth] Authentication successful
2024-03-15 10:24:11 INFO  [orders] Order validated
2024-03-15 10:24:11 INFO  [inventory] Checking stock for product 2SEQ
2024-03-15 10:24:14 ERROR [inventory] Request to inventory-svc timed out
2024-03-15 10:24:14 INFO  [inventory] Retrying (attempt 2/3)
2024-03-15 10:24:15 ERROR [health] inventory-svc health check failed
2024-03-15 10:24:17 ERROR [inventory] Request to inventory-svc timed out
2024-03-15 10:24:17 INFO  [inventory] Retrying (attempt 3/3)
2024-03-15 10:24:18 ERROR [pool] Connection pool usage at 91%
2024-03-15 10:24:20 ERROR [inventory] Request to inventory-svc timed out
2024-03-15 10:24:20 ERROR [orders] Order failed
2024-03-15 10:24:20 INFO  [http] Response sent: 503
2024-03-15 10:24:20 ERROR [inventory-svc] db pool exhausted (50/50)
2024-03-15 10:24:21 INFO  [http] POST /api/v1/orders started
2024-03-15 10:24:21 INFO  [auth] Authentication successful
2024-03-15 10:24:21 INFO  [orders] Order validated
2024-03-15 10:24:21 INFO  [inventory] Checking stock for product 1147
2024-03-15 10:24:24 ERROR [inventory] Request to inventory-svc timed out
2024-03-15 10:24:24 INFO  [inventory] Retrying (attempt 2/3)
```


```
{
  "ts": "10:24:01",
  "lvl": "INFO",
  "svc": "order-svc",
  "msg": "order successfully validated",
  "trace_id": "a3f5b8e1d4c2a001",
  "span_id": "0a1b2c3d",
  "order_id": "ord_28a7f",
  "items": 3
}
```

```
{
  "ts": "10:24:10",
  "lvl": "ERROR",
  "svc": "order-svc",
  "msg": "inventory check failed",
  "trace_id": "a3f5b8e1d4c2a001",
  "span_id": "1a2b3c4d",
  "retries": 3,
  "err": "timeout",
  "target": "inventory-svc",
  "occurrences": 9
}
```

```
{
  "ts": "10:24:20",
  "lvl": "ERROR",
  "svc": "inventory-svc",
  "msg": "db pool exhausted",
  "trace_id": "b7c2d9f0e3a14502",
  "span_id": "9c0d1e2f",
  "db_active": 50,
  "db_max": 50,
  "db_pending": 127
}
```

```
{
  "ts": "10:24:30",
  "lvl": "WARN",
  "svc": "order-svc",
  "msg": "circuit breaker open",
```

Weniger Daten.
Mehr Verstehen.
Mehr Kontrolle.

Key Takeaways

Vorher

- Terabytes ohne Zweck
- Incident = erst Rauschen Filtern
- Niemand vertraut der Teelemtry
- Reaktives feuerlöschen
- Fear-driven, bottom up

Nachher:

- Daten mit klarem Zweck
- Root-Cause schneller finden
- Telemetrie als zuverlässige Basis
- Ermöglicht Service Reliability Automation
- Engineering-minded, top-down

Key Takeaways

Weniger sammeln. Mehr verstehen.

1. Observability ist ein Engineering-Problem.

   Nicht die Menge an Daten entscheidet, sondern ob sie die Fragen beantworten, die eure Nutzer wirklich interessieren.

2. Top-down schlägt bottom-up.

   Beginnt mit der Frage — „Was muss ich wissen, damit meine Nutzer ihr Ziel erreichen?" — und sammelt dann gezielt. Nicht umgekehrt.

3. Gute Telemetrie ist die Voraussetzung für alles, was danach kommt.

   Für schnellere Incident-Response. Für weniger Tokens bei agentenbasiertem Debugging. Für echte proaktive Reliability.