

Metriken: Die Sprache deines Systems verstehen

IN REVIEW

Indem du die Perspektive wechselst, kannst du aus denselben Rohdaten – den Metriken – völlig neue Informationen für verschiedene Rollen im Unternehmen ziehen. Es geht darum, die richtigen Fragen zu stellen.

Hier sind die Informationen, die du aus deinen Daten ziehen kannst, aufgeschlüsselt nach den Perspektiven, die davon profitieren.

- **Perspektive 1: Der Business- & Produkt-Analyst**
 - 1. Feature-Nutzung und -Adoption: Was ist der Hit und was der Flop?
 - 2. User Journey & Conversion Funnels: Wo verlieren wir unsere Kunden?
 - 3. A/B-Testing und Feature-Flags: Welcher Weg ist der bessere?
 - 4. Kundensegmentierung durch Verhalten
- **Perspektive 2: Der Entwickler & SRE (Site Reliability Engineer)**
 - 1. Performance-Flaschenhals-Analyse: Wo brennt die Zeit?
 - 2. Resource Leak Detection: Das schleichende Desaster verhindern
 - 3. Impact-Analyse von Deployments: Freund oder Feind?
 - 4. Korrelation von Fehlern und Infrastruktur
- **Perspektive 3: Der Manager & Platform Owner**
 - 1. Kosten-Allokation (FinOps): Wer bezahlt die Rechnung?
 - 2. Service Level Objective (SLO) & Error Budget Monitoring: Wie viel Risiko können wir uns leisten?
 - 3. Kapazitätsplanung: In die Zukunft blicken
- **Fazit: Der Wandel in der Denkweise ist die größte Erkenntnis**

Perspektive 1: Der Business- & Produkt-Analyst

Dein Ziel: Du willst verstehen, wie Kunden das Produkt *wirklich* nutzen, welche Features sie lieben, welche sie ignorieren und wo sie auf Hürden stoßen, die den Geschäftserfolg bremsen.

1. Feature-Nutzung und -Adoption: Was ist der Hit und was der Flop?

- **Was du ableiten kannst:** Welche Teile deiner Anwendung sind die Arbeitspferde und welche sind teure Staubfänger? Verstehen, ob ein neues Feature überhaupt angenommen wird oder ob eine Kernfunktion an Relevanz verliert.
- **Wie du es machst:** Analysiere die `_count` oder `_total` Metriken von HTTP-Anfragen, gruppiert nach API-Endpunkt (`http.route` oder `uri`). Zähle, wie oft bestimmte Aktionen ausgelöst werden.
- **Beispiel aus deinen Daten:**
 - Vergleiche `de_signaliduna_evb_api_AbrufResource_abrufNeuzulassungen_..._total` mit `de_signaliduna_evb_api_AbrufResource_abrufWohnortwechsel_..._total`.
 - **Erweiterte Erkenntnis:** "90% unserer eVB-Abrufe sind für Neuzulassungen, aber nur 2% für einen Wohnortwechsel. Die Hypothese war, dass der Wohnortwechsel ein wichtiger Self-Service-Prozess ist. Die Daten zeigen das Gegenteil. **Statt den Prozess zu bewerben**, sollten wir vielleicht Interviews mit den 2% führen, um zu verstehen, warum sie es nutzen, und mit anderen, um herauszufinden, warum sie es *nicht* tun. Vielleicht ist der Prozess zu kompliziert, oder die Nutzer rufen in diesem Fall lieber an. Das ist eine strategische Entscheidung: Vereinfachen, entfernen oder gezielt bewerben?"

2. User Journey & Conversion Funnels: Wo verlieren wir unsere Kunden?

- **Was du ableiten kannst:** An welchem exakten Punkt brechen Benutzer einen mehrstufigen Prozess ab (z.B. ein Antragsformular, ein Onboarding, der Warenkorb-Checkout)? Dies sind deine teuersten Reibungspunkte.
- **Wie du es machst:** Instrumentiere jeden einzelnen Schritt eines kritischen Prozesses mit einem eigenen Zähler (Counter).
- **Beispiel (manuelle Instrumentierung):**
 - `app.antrag.schritt.abgeschlossen.total{schritt="1_persoেনliche_daten"}: 10.000 Mal`
 - `app.antrag.schritt.abgeschlossen.total{schritt="2_tarifauswahl"}: 8.000 Mal`
 - `app.antrag.schritt.abgeschlossen.total{schritt="3_zahlungsinformationen"}: 4.000 Mal`
 - **Erweiterte Erkenntnis:** "Zwischen Tarifauswahl und Zahlungsinformationen verlieren wir 50% der verbleibenden Nutzer! Das ist ein massives Problem. Liegt es am UI/UX? Fehlen Zahlungsmethoden, die die Nutzer erwarten (z.B. PayPal)? Sind die angezeigten Kosten überraschend? Wir können jetzt gezielt Hypothesen aufstellen und A/B-Tests für genau diesen Schritt entwerfen, anstatt blind im Prozess herumzudoktern."

3. A/B-Testing und Feature-Flags: Welcher Weg ist der bessere?

- **Was du ableiten kannst:** Welche von zwei Versionen eines Features (A oder B) performt besser in Bezug auf Geschäftsziele (Conversion), technische Stabilität (Fehler) oder Kundenzufriedenheit (Latenz)?
- **Wie du es machst:** Füge bei der Instrumentierung ein Label (Tag) für die Feature-Variante hinzu (z.B. `version="A"` oder `feature_flag=new_checkout_2024`).
- **Beispiel:**
 - `app.bezahlprozess.erfolgreich.total{version="A"}` vs. `app.bezahlprozess.erfolgreich.total{version="B"}`.

- **Erweiterte Erkenntnis:** "Version B hat nicht nur eine 20% niedrigere *technische* Fehlerrate, sondern auch eine 5% höhere *Conversion-Rate* (`.erfolgreich.total`). Das bedeutet, sie ist stabiler **und** bringt mehr Umsatz. Die Entscheidung ist klar: Wir schalten Version A ab und rollen B für alle aus. Die Metriken haben uns eine Geschäftsentscheidung mit direktem finanziellem Mehrwert geliefert."

4. Kundensegmentierung durch Verhalten

- **Was du ableiten kannst:** Gibt es verschiedene Nutzergruppen ("Power-User" vs. "Gelegenheitsnutzer")? Wie verhalten sie sich unterschiedlich?
- **Wie du es machst:** Gruppieren die Nutzungsmetriken nicht nur nach Endpunkt, sondern auch nach Nutzer-ID oder Kundensegment (falls verfügbar). Analysiere die Häufigkeit und Vielfalt der genutzten API-Calls pro Nutzer.
- **Erkenntnis:** "10% unserer Nutzer verursachen 80% der API-Aufrufe für die Funktion 'Datenexport'. Diese Power-User nutzen unser Produkt völlig anders. Wir sollten sie direkt ansprechen, um ihre Bedürfnisse besser zu verstehen. Vielleicht brauchen sie eine dedizierte API oder ein höheres Rate-Limit, wofür sie sogar bereit wären zu zahlen."

Perspektive 2: Der Entwickler & SRE (Site Reliability Engineer)

Dein Ziel: Du willst Performance-Flaschenhälse aufspüren, die Ursache von Fehlern in Sekundenschnelle eingrenzen und die technischen Auswirkungen jeder Code-Änderung sofort sehen.

1. Performance-Flaschenhals-Analyse: Wo brennt die Zeit?

- **Was du ableiten kannst:** Welcher Teil einer Anfrage ist der Schuldige für eine hohe Latenz? Ist es die Datenbank, ein externer API-Aufruf, die Nachrichten-Queue oder die eigene Business-Logik?
- **Wie du es machst:** Nutze Histogramm-Metriken (`_bucket`) für die Dauer der verschiedenen internen Operationen eines Requests. Vergleiche die Perzentile (p90, p95, p99).
- **Beispiel:**
 - Ein Endpunkt hat eine p95-Latenz von 800ms (`http.server.request.duration...`).
 - Die p95-Latenz der Datenbankabfragen innerhalb dieses Requests ist 750ms (`db.client.operations.duration...`).
 - Die p95-Latenz eines externen Service-Aufrufs ist 20ms (`rpc.client.duration...`).
 - **Erweiterte Erkenntnis:** "Das Problem liegt zu über 90% in der Datenbank. Die Anwendungslogik und der externe Call sind unschuldig. Anstatt Code-Reviews durchzuführen, müssen wir sofort die exakte SQL-Abfrage identifizieren (z.B. über Tracing), ihren `EXPLAIN`-Plan analysieren und prüfen, ob Indizes fehlen oder die Abfrage umgeschrieben werden muss. Die Metrik gibt uns die genaue Richtung für die Fehlersuche vor."

2. Resource Leak Detection: Das schleichende Desaster verhindern

- **Was du ableiten kannst:** Verbraucht unsere Anwendung schleichend immer mehr Ressourcen (Speicher, DB-Verbindungen, Threads), was unweigerlich zu einem Absturz oder Performance-Einbruch führen wird?
- **Wie du es machst:** Beobachte Metriken wie Heap-Nutzung, Thread-Anzahl oder aktive DB-Verbindungen über einen langen Zeitraum (Tage /Wochen). Achte auf den Trend.
- **Beispiel aus deinen Daten:**
 - Betrachte den Graphen von `jvm_memory_used_bytes{area="heap"}` über eine Woche.
 - **Erweiterte Erkenntnis:** "Normalerweise sollte der Heap-Verbrauch wie ein Sägezahn aussehen (steigt an, dann räumt die Garbage Collection auf). Unser Graph zeigt aber, dass die 'Täler' des Sägezahns nach jedem Deployment ein bisschen höher liegen. Das ist ein klassisches Speicherleck. Ein bestimmtes Objekt wird referenziert und kann nicht mehr freigegeben werden. Wir müssen den Heap-Dump von kurz vor und kurz nach einem Deployment vergleichen, um den Schuldigen zu finden."

3. Impact-Analyse von Deployments: Freund oder Feind?

- **Was du ableiten kannst:** Hat unser letztes Release die Welt besser oder schlechter gemacht? Hat es die Performance verbessert, neue Fehler eingeführt oder die CPU-Last erhöht?
- **Wie du es machst:** Markiere den Zeitpunkt eines Deployments in deinen Graphen (via Annotation). Vergleiche Fehlerraten (`http.server.requests.count{status="5xx"}`), Latenzen (`...duration`) und Ressourcennutzung (`process.cpu.load`) direkt vor und nach dem Release.
- **Erweiterte Erkenntnis:** "Seit dem Deployment von Version 2.4.1 ist die p99-Latenz um 30% gestiegen **und** die Rate der 4xx-Fehler hat sich verdoppelt. Das neue Feature hat nicht nur einen Performance-Impact, sondern scheint auch client-seitige Fehler zu provozieren. Vielleicht hat sich die API-Schnittstelle auf eine Weise geändert, die für die Clients nicht abwärtskompatibel ist. Das ist kritisch! Wir sollten ein Rollback in Erwägung ziehen oder sofort einen Hotfix bereitstellen."

4. Korrelation von Fehlern und Infrastruktur

- **Was du ableiten kannst:** Treten Fehler nur auf einem bestimmten Host, in einer bestimmten Cloud-Region oder in einem bestimmten Kubernetes-Pod auf?
- **Wie du es machst:** Gruppieren deine Fehlermetriken (`_count{status_code="500"}`) nach Infrastruktur-Labels wie `host`, `availability_zone` oder `pod_name`.
- **Erkenntnis:** "99% aller 500er-Fehler des Login-Service kommen von einem einzigen Pod! Die anderen 9 Pods laufen einwandfrei. Das Problem ist also kein Code-Fehler, sondern wahrscheinlich ein lokales Problem auf diesem Pod (z.B. eine defekte Festplatte, ein Netzwerkproblem, eine 'verklebte' Verbindung). Anstatt den Code zu debuggen, können wir den betroffenen Pod einfach neustarten und das Problem ist sofort gelöst."

Perspektive 3: Der Manager & Platform Owner

Dein Ziel: Du willst Kosten kontrollieren, Kapazitäten strategisch planen, Risiken minimieren und sicherstellen, dass die teure Plattform die versprochenen Service-Levels (SLOs) einhält.

1. Kosten-Allokation (FinOps): Wer bezahlt die Rechnung?

- **Was du ableiten kannst:** Welches Team, welcher Service oder welches Feature verursacht die meisten Kosten (z.B. durch teure DB-Abfragen, hohen Log-Output, CPU-intensive Berechnungen oder massiven Netzwerk-Traffic)?
- **Wie du es machst:** Bereichere deine Metriken mit standardisierten Labels wie `team`, `cost_center` oder `product`. Dies kann oft über Kubernetes-Labels oder Service-Konfigurationen automatisiert werden.
- **Beispiel:**
 - Gruppieren `sum(rate(kafka_producer_byte_rate[5m])) by (team)`.
 - **Erweiterte Erkenntnis:** "Team 'Data-Analytics' schreibt 80% des gesamten Datenvolumens in Kafka. Das verursacht nicht nur Kafka-Kosten, sondern auch Kosten für Storage und Verarbeitung in den nachgelagerten Systemen. **Anstatt die Infrastruktur pauschal für alle zu vergrößern**, können wir gezielt mit Team 'Data-Analytics' sprechen. Brauchen sie wirklich alle diese Daten in Echtzeit? Können sie die Daten vorfiltern oder aggregieren? So werden aus technischen Metriken strategische Hebel zur Kostenoptimierung."

2. Service Level Objective (SLO) & Error Budget Monitoring: Wie viel Risiko können wir uns leisten?

- **Was du ableiten kannst:** Wie zuverlässig ist unser Service für unsere Kunden? Und wie viel "Spielraum für Fehler" (Error Budget) haben wir diesen Monat noch, bevor wir unser Versprechen an den Kunden brechen und Innovationen stoppen müssen?
- **Wie du es machst:** Definiere ein klares SLO, z.B. "99.9% aller Login-Anfragen im Monat müssen erfolgreich und in unter 500ms beantwortet werden". Das **Error Budget** sind die 0.1%, die fehlschlagen oder zu langsam sein dürfen.
- **Beispiel:**
 - Erstelle ein Dashboard, das `1 - (rate(login_failures) + rate(logins_too_slow)) / rate(login_requests)` anzeigt.
 - **Erweiterte Erkenntnis:** "Wir sind am 20. des Monats und haben bereits 90% unseres Error Budgets verbraucht. Dies ist eine rote Flagge für das Management! Es bedeutet, dass wir unser Qualitätsversprechen an den Kunden zu brechen drohen. Die Konsequenz ist eine datengetriebene Regel: **Alle nicht-kritischen Deployments für den Login-Service werden sofort gestoppt**. Das Team muss sich auf Stabilität konzentrieren. Das Error Budget verwandelt die Debatte 'Stabilität vs. Features' in eine messbare, rationale Entscheidung."

3. Kapazitätsplanung: In die Zukunft blicken

- **Was du ableiten kannst:** Wann müssen wir unsere Infrastruktur (Festplatten, Datenbank-Verbindungen, CPU-Kerne, Lizenzen) erweitern, um zukünftiges Wachstum oder saisonale Spitzen (z.B. Weihnachtsgeschäft) abzufangen?
- **Wie du es machst:** Analysiere langfristige Trends (Monate/Quartale) von Schlüsselmetriken und lege eine Trendlinie darüber, um den zukünftigen Bedarf zu extrapolieren.
- **Beispiel aus deinen Daten:**
 - Trend von `pg_database_size_bytes` oder `node_filesystem_size_bytes`.
 - Trend des maximalen `hikaricp_connections_active`.
 - **Erweiterte Erkenntnis:** "Unsere Datenbank wächst linear um 20 GB pro Monat. Wir werden in 4 Monaten die aktuelle Festplattenkapazität von 90% erreichen, was unser Alarm-Schwellenwert ist. **Anstatt in Panik zu geraten, wenn der Alarm losgeht**, können wir jetzt, mit 4 Monaten Vorlauf, die Erweiterung budgetieren, planen und ohne Stress durchführen. Das spart Geld und reduziert das Risiko eines Ausfalls."

Fazit: Der Wandel in der Denkweise ist die größte Erkenntnis

Die wichtigste Information, die du aus deinen Daten ziehen kannst, ist der Wandel von einer reaktiven ("Oh nein, es ist kaputt!") zu einer **datengetriebenen, proaktiven Kultur**.

Behandle deine Observability-Daten wie ein erstklassiges Produkt. Sie erzählen eine fortlaufende Geschichte über dein System, deine Nutzer und dein Geschäft. Anstatt sie nur als Werkzeug zur Fehlerbehebung im Keller zu betrachten, bringe sie in die Vorstandsetage. Nutze sie, um bessere Produkte zu bauen, fundierte Geschäftsentscheidungen zu treffen und deine teuren Ressourcen intelligent zu planen.