
# CLOUD-NATIVE

*Unit:*

## *Observability*

*(4) Distributed Tracing*



CLOUD NATIVE COMPUTING

creative commons

CLOUD NATIVE COMPUTING

*Urheberrechtshinweise*

Diese Folien werden zum Zwecke einer praktikablen und pragmatischen Nutzbarkeit im Rahmen der **CCo 1.0 Lizenz** bereitgestellt.


Sie dürfen die Inhalte also kopieren, verändern, verbreiten, mit eigenen Inhalten mixen, auch zu kommerziellen Zwecken, und ohne um weitere Erlaubnis bitten zu müssen.

Eine Nennung des Autors ist nicht erforderlich (aber natürlich gern gesehen, wenn problemlos möglich).


Diese Folien sind insb. für die Lehre an Hochschulen konzipiert und machen daher vom **§51 UrhG (Zitate)** Gebrauch.

Die CCo Lizenz überträgt sich nicht auf zitierte Quellen. Hier sind bei der Nutzung natürlich die Bedingungen der entsprechenden Quellen zu beachten.

Die Quellenangaben finden sich auf den entsprechenden Folien.

# KAPITEL 13

*Beobachtbare Architekturen*


## 13.1 Konsolidierung von Telemetriedaten

## 13.2 Instrumentierung von Systemen

- Logging
- Monitoring
- Tracing

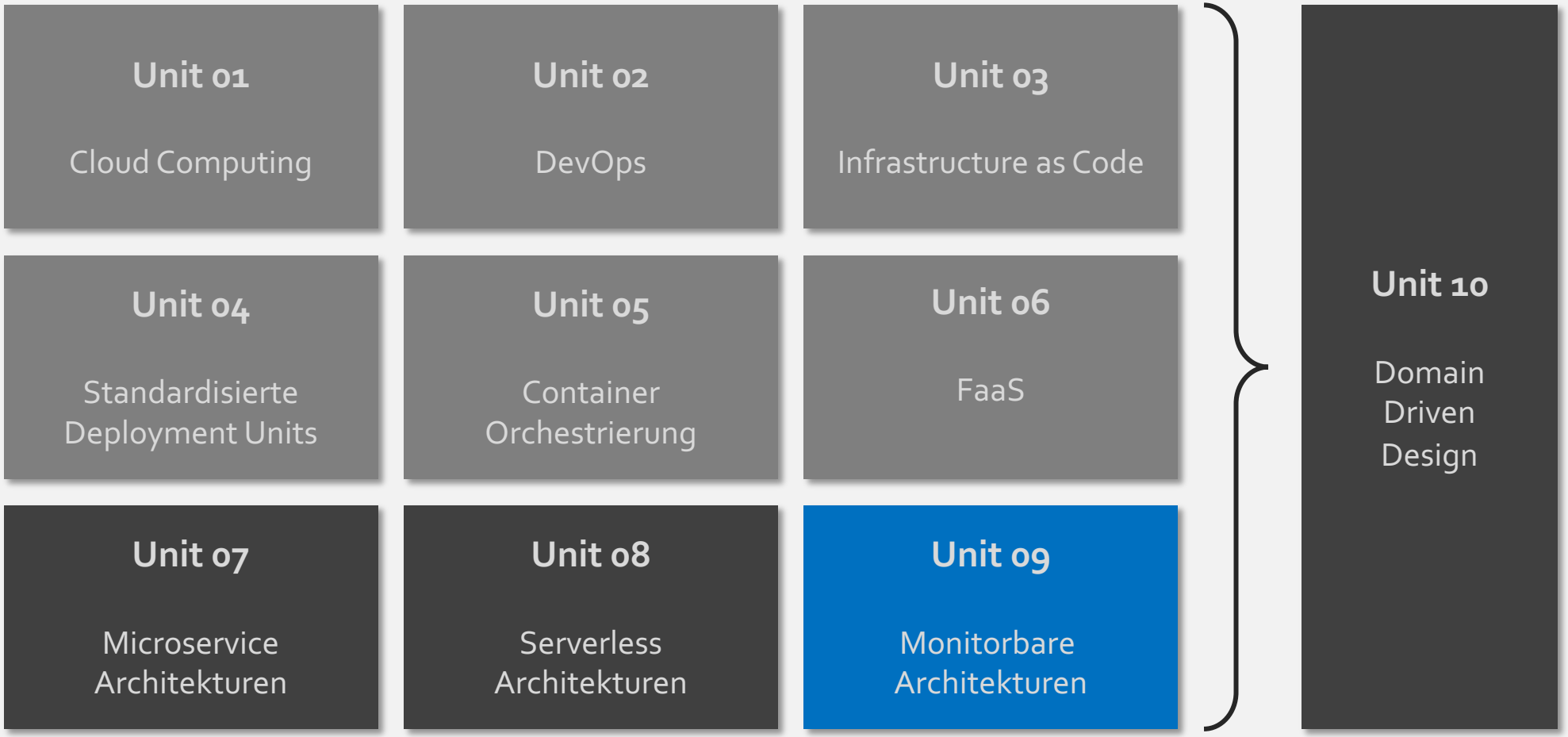
## 13.3 Automatisierte Instrumentierung

- Service Meshs
- Traffic-Management
- Resilienz
- Sicherheit
- Management und Analyse von Verkehrstopologien

## 13.4 Zusammenfassung

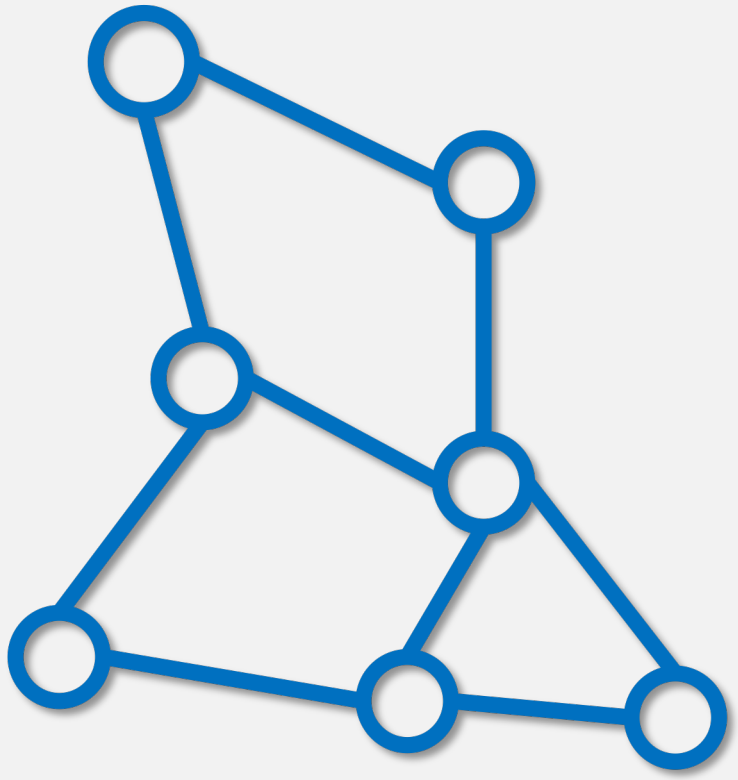
# INHALTSVERZEICHNIS

*Überblick über Units und Themen dieses Moduls*



# INHALTE

- Beobachtbarkeit von Systemen
- Metriken
- Logs
- Tracing



# ÜBERWACHUNG VON SERVICE-ARCHITEKTUREN

*Logging, Monitoring, Tracing*

Die Überwachung von Software erfolgt üblicherweise mittels der Erfassung von drei Arten von Telemetriedaten:

- Metriken im Rahmen eines **Monitorings** liefern **quantitative** Informationen zu Prozessen, die im System ausgeführt werden.
- Mittels **Logging** (Protokollierung) lässt sich **qualitativer** Einblick in anwendungsspezifische Ereignisse gewinnen, die von Prozessen verarbeitet werden.
- **Distributed Tracing** (verteilte Ablaufverfolgung) ermöglicht Einblick in den gesamten Lebenszyklus von Requests entlang von Systemkomponenten (Services), um so bspw. Fehler und Latenzen in der verteilten Verarbeitung zu erkennen.


# ÜBERWACHUNG VON SERVICE-ARCHITEKTUREN

*Tracing Request-/Response-basierter Kommunikation*

- Dabei kann man grundsätzlich zwei Methoden zur Verfolgung unterscheiden
- Beide Methoden werden genutzt, um Fehler zu finden und die Leistung der Anwendung zu verbessern
- Bei **Black Box Tracing Systemen**, werden einfach erfassbare Informationen über die Kommunikation wie Zeitstempel, Speicherzugriffe, Netzwerkaktivität und Eingaben und Ausgaben aufgezeichnet ohne das die Anwendung instrumentiert werden muss
- Bei **Annotations-basiertes Tracing** Systemen werden sogenannte Annotationen im Code der Anwendung instrumentiert, um die Ausführung zu verfolgen.
- Die Annotationen können dann verwendet werden, um anwendungsspezifische Daten über die Ausführung zu sammeln, wie z.B. Funktionsaufrufe, Variablenwerte und Fehlermeldungen

Beide Arten von Tracing Systemen haben Vor- und Nachteile.

- **Black Box Tracing Systeme** erfordern keine Modifikation des Codes und können daher einfacher zu implementieren sein
- **Black Box Tracing** ermöglicht jedoch keine Anwendungs-spezifischen Informationen zu erfassen
- **Annotations-basiertes Tracing** ermöglicht detaillierte Informationen über die Ausführung
- **Annotations-basiertes Tracing** erfordert jedoch die manuelle Instrumentierung von Code und kann die Leistung der Anwendung beeinträchtigen

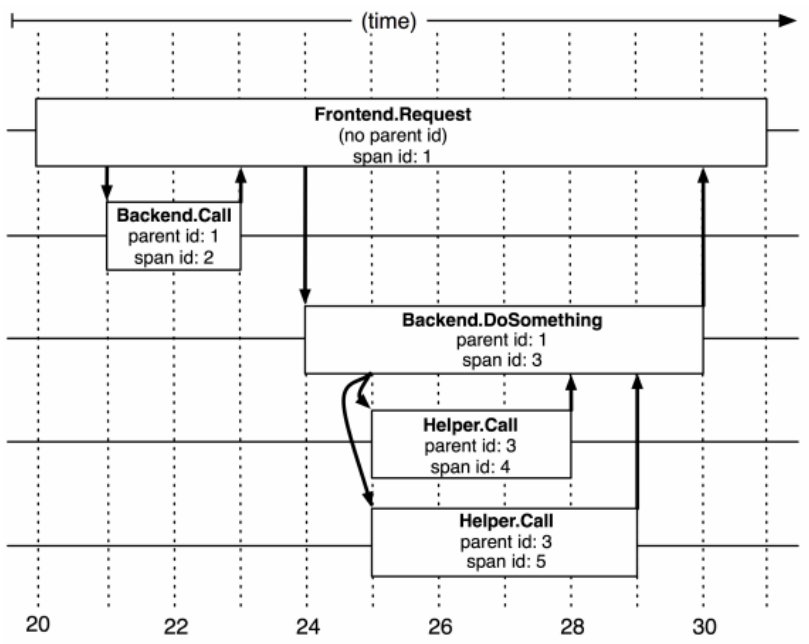
# ÜBERWACHUNG VON SERVICE-ARCHITEKTUREN

*Tracing Request-/Response-basierter Kommunikation*

**Distributed Tracing Systeme** folgen üblicherweise dem Annotations-basiertem Ansatz und ermöglichen die Rückverfolgung von Service Interaktionen im laufenden Betrieb und helfen beim Erfassen von Zeitreihendaten (Traces), die u.a. zur Behebung von Latenzproblemen in Servicearchitekturen hilfreich sein können. Zu den Funktionen gehören üblicherweise das Erfassen, die Auswertung von Traces und die Suche nach Traces in größeren Traces-Datensätzen.

Mit der Kenntnis einer Trace-IDs kann man einzelne Vorgänge nachvollziehen. Traces lassen sich aber auch meist mittels Abfragen anhand von Attributen wie bspw. Service-Name, Operationsname, Tags und Labels oder Dauern bestimmen.

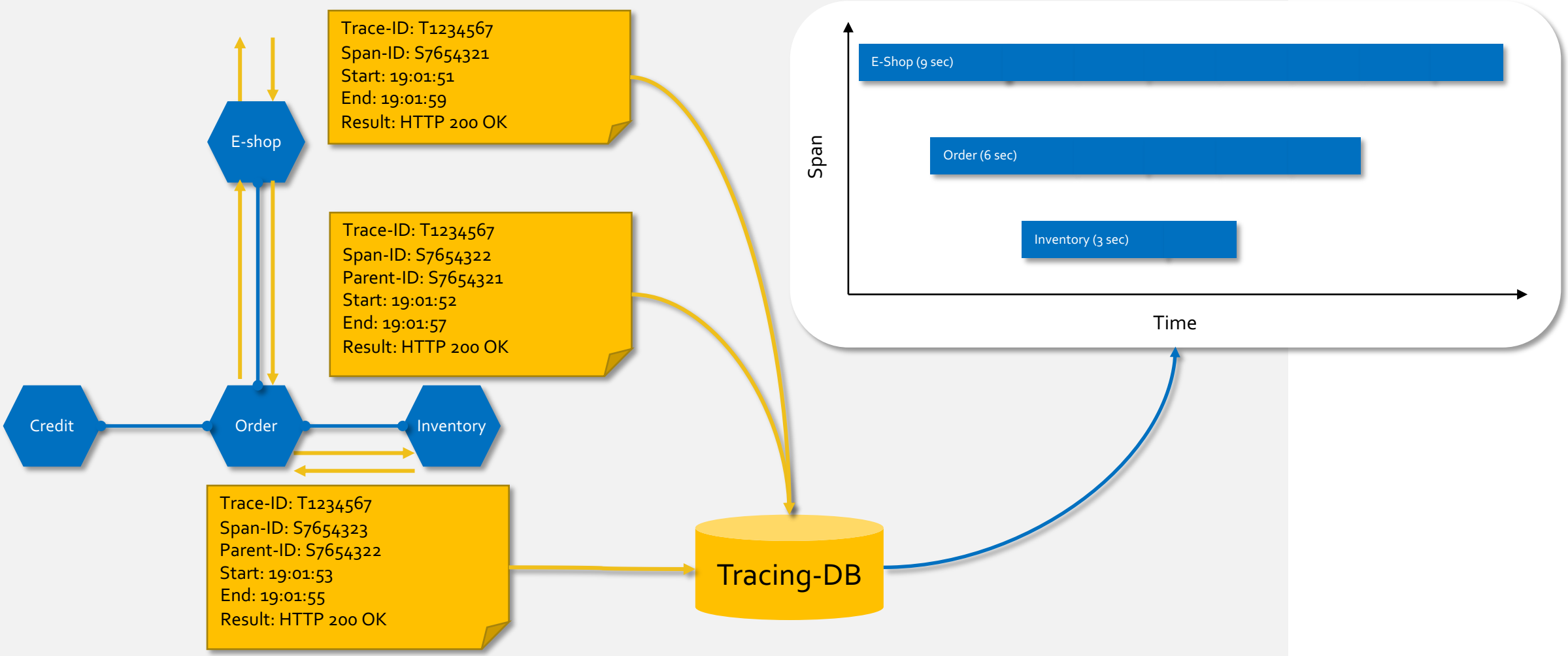
Oberflächen von Tracing-Systemen zeigen meist Abhängigkeitsdiagramme an, um das Gesamtverhalten einschließlich Fehlerpfaden oder Aufrufabfolgen von Services einfacher in Analysen erfassen zu können.




**Quelle:** Benjamin H. Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, Chandan Shanbhag; **Dapper, a Large-Scale Distributed Systems Tracing Infrastructure**, Google Technical Report, 2010
https://research.google/pubs/pub36356.pdf

# DAS PRINZIP

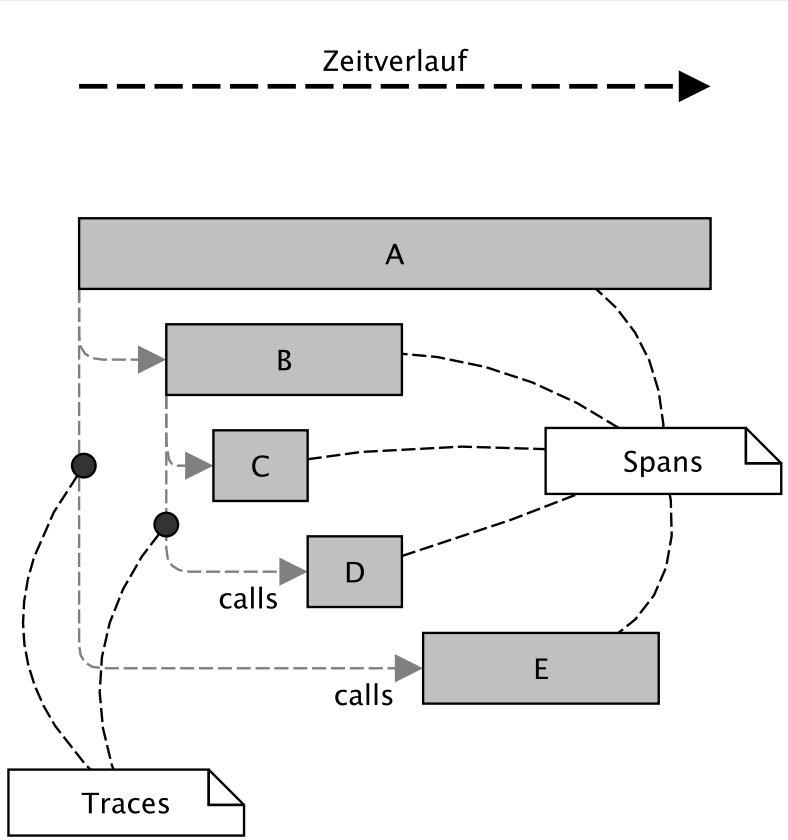
*Tracing Request-/Response-basierter Kommunikation*

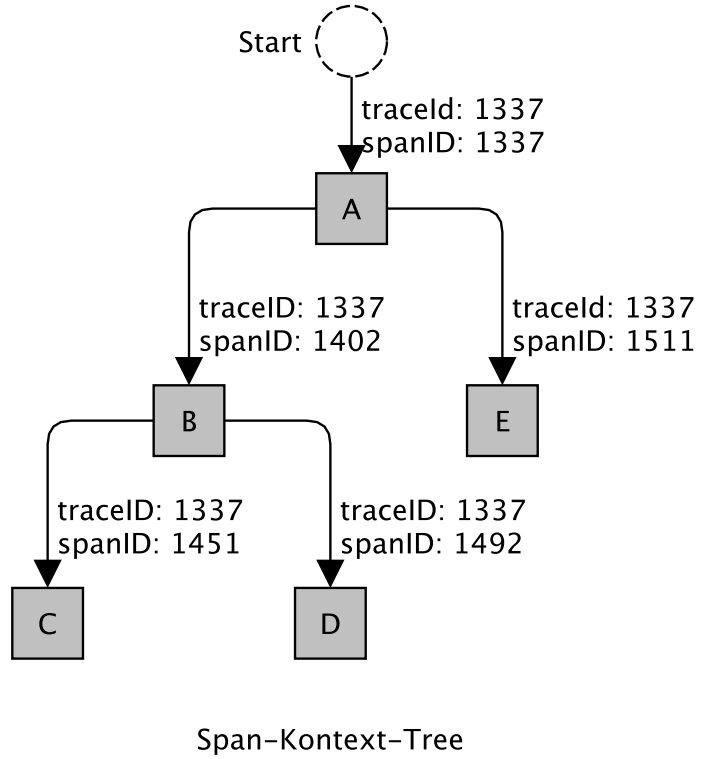


# TRACING

*Begrifflichkeiten: Traces + Spans*

- **Trace:** Die Beschreibung einer Transaktion, die sich durch ein verteiltes System bewegt.
- **Span:** Eine benannte, zeitgesteuerte Operation, die einen Teil des Workflows darstellt. Spans können mit Daten annotiert werden (Log), die zur Verfolgung und Auswertung von Transaktionen erforderlich sind.
- **Span-Kontext:** Trace-Informationen, die die verteilte Transaktion zwischen Services begleiten. Der Span-Kontext enthält die Trace-ID, die Span-ID, eine Parent-ID und alle anderen Daten, die das Tracing-System zur Verfolgung von Transaktionen in nachgeschalteten Services benötigt.



# TRACING

*Instrumentierung*

```python
from opentracing_instrumentation.request_context import get_current_span, span_in_context

def init_tracer(service):
    logging.getLogger('').handlers = []
    logging.basicConfig(format='%(message)s', level=logging.DEBUG)
    config = Config(
        config={
            'sampler': {
                'type': 'const',
                'param': 1,
            },
            'logging': True,
        },
        service_name=service,
    )
    return config.initialize_tracer()

def check_cinema(movie):
    with tracer.start_span('CheckCinema', child_of=get_current_span()) as span:
        with span_in_context(span):
            num = random.randint(1,30)
            time.sleep(num)
            cinema_details = "Cinema Details"
            flags = ['false', 'true', 'false']
            random_flag = random.choice(flags)
            span.set_tag('error', random_flag)
            span.log_kv({'event': 'CheckCinema' , 'value': cinema_details })
            return cinema_details
```

Tracing Initialisierung

Tracing Instrumentierung

Code muss „instrumentiert" werden, um Trace-Daten an Verteilte Tracing-Systeme zu melden. Dies bedeutet normalerweise die Konfiguration und Einbettung von Tracing-Code unter Nutzung von Instrumentenbibliotheken für spezifische Tracing-Systeme.

Da die Instrumentierung dann leicht Tracing-System-spezifisch werden kann, entwickeln sich auch Tracing-Standards wie bspw. Open-Telemetry API um zu vermeiden, das eine Instrumentierung zwar mit ZIPKIN aber nicht mit Jaeger funktioniert.

CLOUD NATIVE COMPUTING

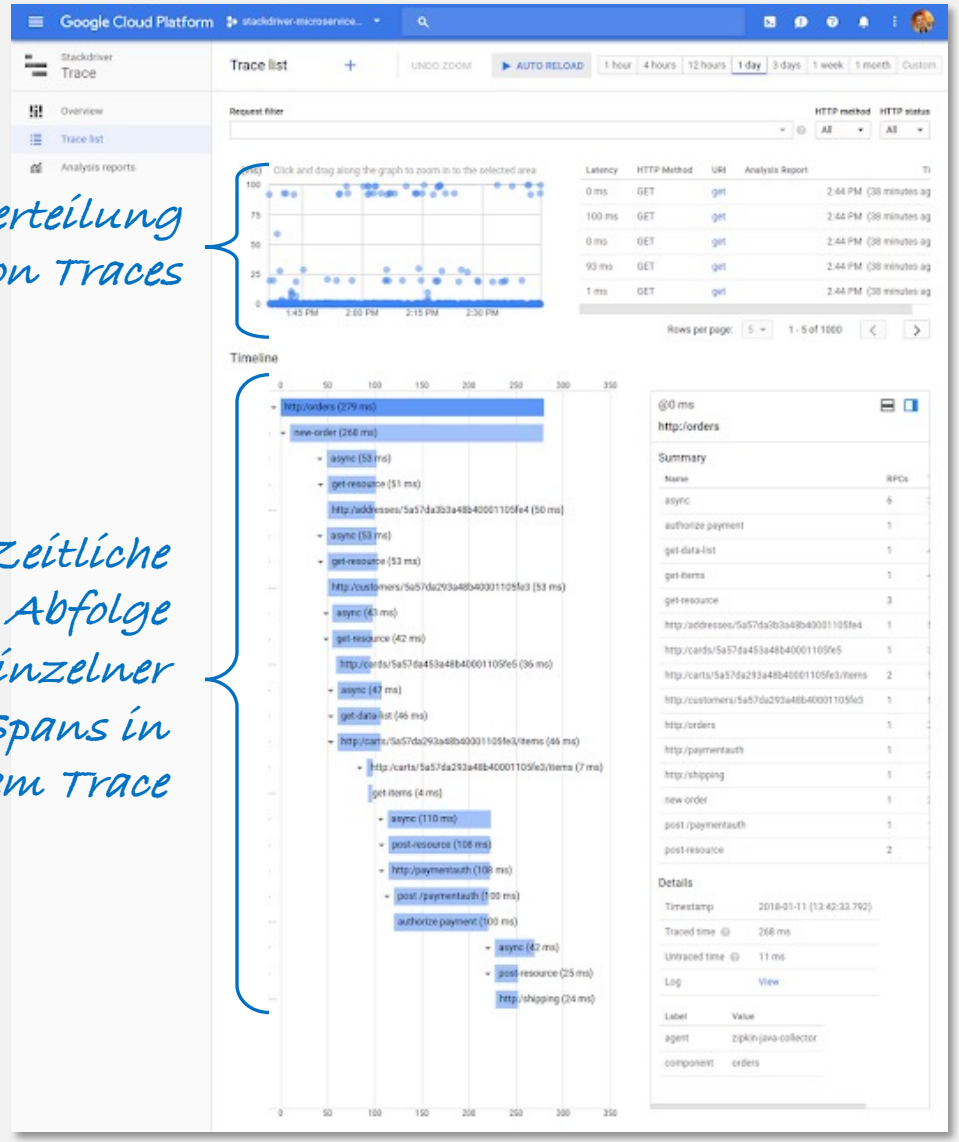
OpenTelemetry

# TRACING

*Beispiel eines Managed Services*

## Google Trace:

*„Cloud Trace ist ein System für verteiltes Tracing, mit dem die Latenzdaten von Anwendungen erfasst und in der Google Cloud Console angezeigt werden. So kann man den Ablauf von Anfragen in Anwendungen verfolgen und detaillierte Leistungsdaten echtzeitnah erhalten. Mit Cloud Trace werden automatisch alle Traces eine Anwendung analysiert, um detaillierte Latenzberichte zu generieren, um bspw. Leistungsverschlechterungen zu erkennen. Außerdem können Sie damit Traces von allen Ihren VMs, Containern oder App Engine-Projekten erfassen.“*




**Quelle:** https://cloud.google.com

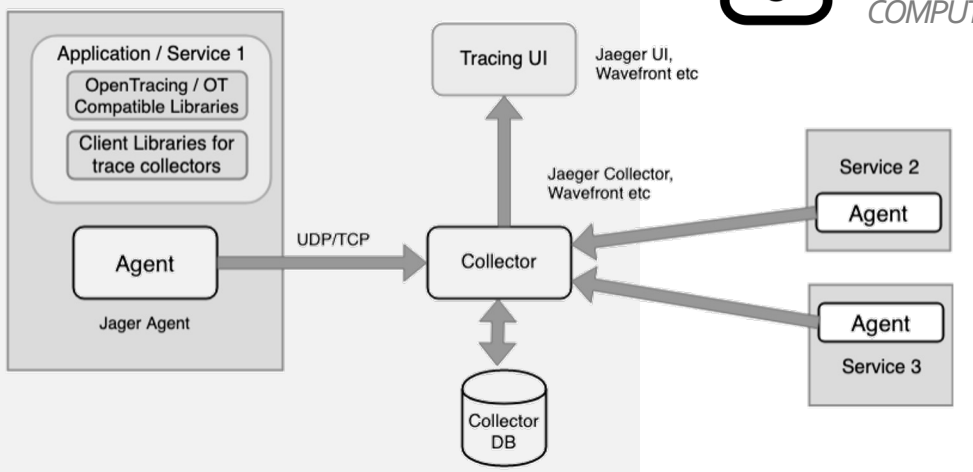
CLOUD NATIVE COMPUTING

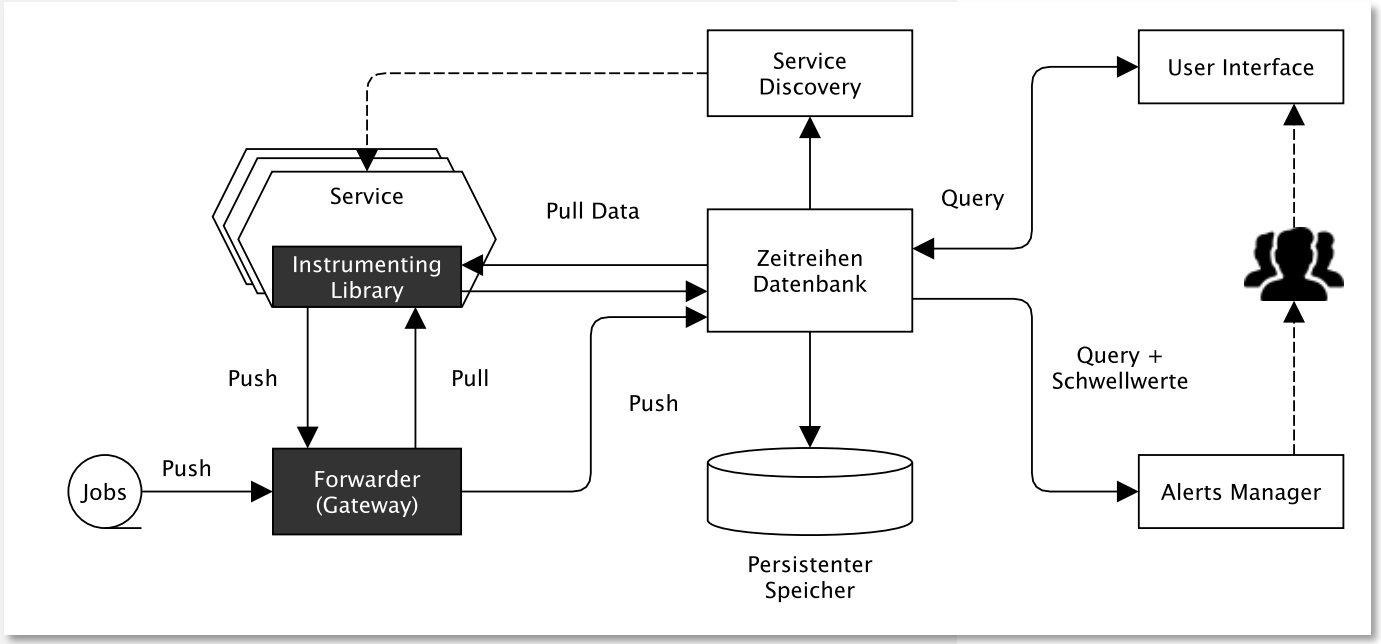
# TRACING

*Distributed Tracing Architektur*

- Viele Tracing-Systeme benötigen auf Agenten, hier einmal am Beispiel des Jaeger Tracing Systems dargestellt
- Alle Kommunikation mit dem Tracing Collector wird erst einmal durch einen **Agenten** gegeben der lokal auf dem Host vorhanden ist (oder als Sidecar Pod eingebunden wird)
- Dieser kann dann die Trace-Daten an einen **Collector** (Forwarder) weitergeben, ohne dass der Service weiß, dass er gerade getraced wird bzw. die Adresse des Tracing-Service kennen muss
- Dieses Prinzip haben wir bereits bei der Log-Konsoldierung kennen gelernt und überträgt das Prinzip des **Log-Forwardings** (Regel 11/12 der 12 Faktore) quasi analog auf das Distributed Tracing
- In unserer allgemeinen **Observability Architektur** folgen Distributed Tracing Ansätze also denselben Pfad wie auch die Log-Konsolidierung

**Tracing Architektur am Beispiel von Jaeger**




**Observability Architektur**

# BEST-PRACTICES

*Distributed Tracing Instrumentierung ist eine der aufwändigsten Arten der Instrumentierung*

## Allgemeine Empfehlungen für die Instrumentierung von Services

- Nutzung **standardisierter Formate** wie OpenTracing oder OpenTelemetry
- Definition **sinnvoller Spans**, die den Kontext und die Logik Ihrer Anwendung widerspiegeln
- Fokus bei der Instrumentierung auf **relevante Services**, um sicherzustellen, dass alle Traces erfasst werden und eine vollständige Sicht auf die gesamte Anwendung gewährleistet ist
- Berücksichtigung der **Performance-Auswirkungen** der Instrumentierung, insbesondere bei hochfrequentierten Anwendungen. Verwendung von **Sampling (repräsentative Stichprobe)**
- Nichts instrumentieren, was nicht auch ausgewertet wird!

## Empfehlungen für die Priorisierung zu instrumentierender Services

- **Architekturanalyse**: Beginn bei User-Interfacing Services (API-Gateway), Instrumentierung entlang deren Abhängigkeiten
- **Priorisierung** von Services, anhand:
  - **Wichtigkeit** aufgrund von **Nutzer- oder Geschäftsanforderungen**
  - **Kritikalität** aufgrund von Kriterien wie Nutzeranzahl, Geschäftskritikalität, Auswirkung auf Usability
  - Services, die häufige **Engpässe oder Performance-Probleme** verursachen, detaillierter instrumentieren
  - Services mit **vielen Abhängigkeiten** haben (oft gRPC, REST)

# ZUSAMMENFASSUNG

*Zu Observability*

CLOUD NATIVE COMPUTING

**Metrics**

Quantitative Zusammenhänge

**Logging**

Qualitative Ereignisse

**Tracing**

Verfolgen von Transaktionen in verteilten Systemen

**Strukturiertes Logging mit Log-Konsolidierung**

- Ermöglicht das Erfassen quantitativer Daten `{ “rpm“: 5 }`
- Ermöglicht das Erfassen qualitativer Ereignisse `{ “login“: “success“, “user“: “Max Mustermann“ }`
- Ermöglicht auch die Verfolgung von Transaktionen `{ “trace-id“: “456321“, “span-id“: “123456“ }`
- Ermöglicht auch beliebige Kombinationen dieser drei Aspekte

*Leider werden diese drei Aspekte immer noch häufig isoliert betrachtet. Die Konsequente Nutzung strukturierten Loggings könnte diese drei Säulen zu einem integrierten Ansatz zusammenführen.*

*Entsprechende Lösungen sind aber kaum zu finden.*

# KONTAKT

*Disclaimer*

**Nane Kratzke**

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de