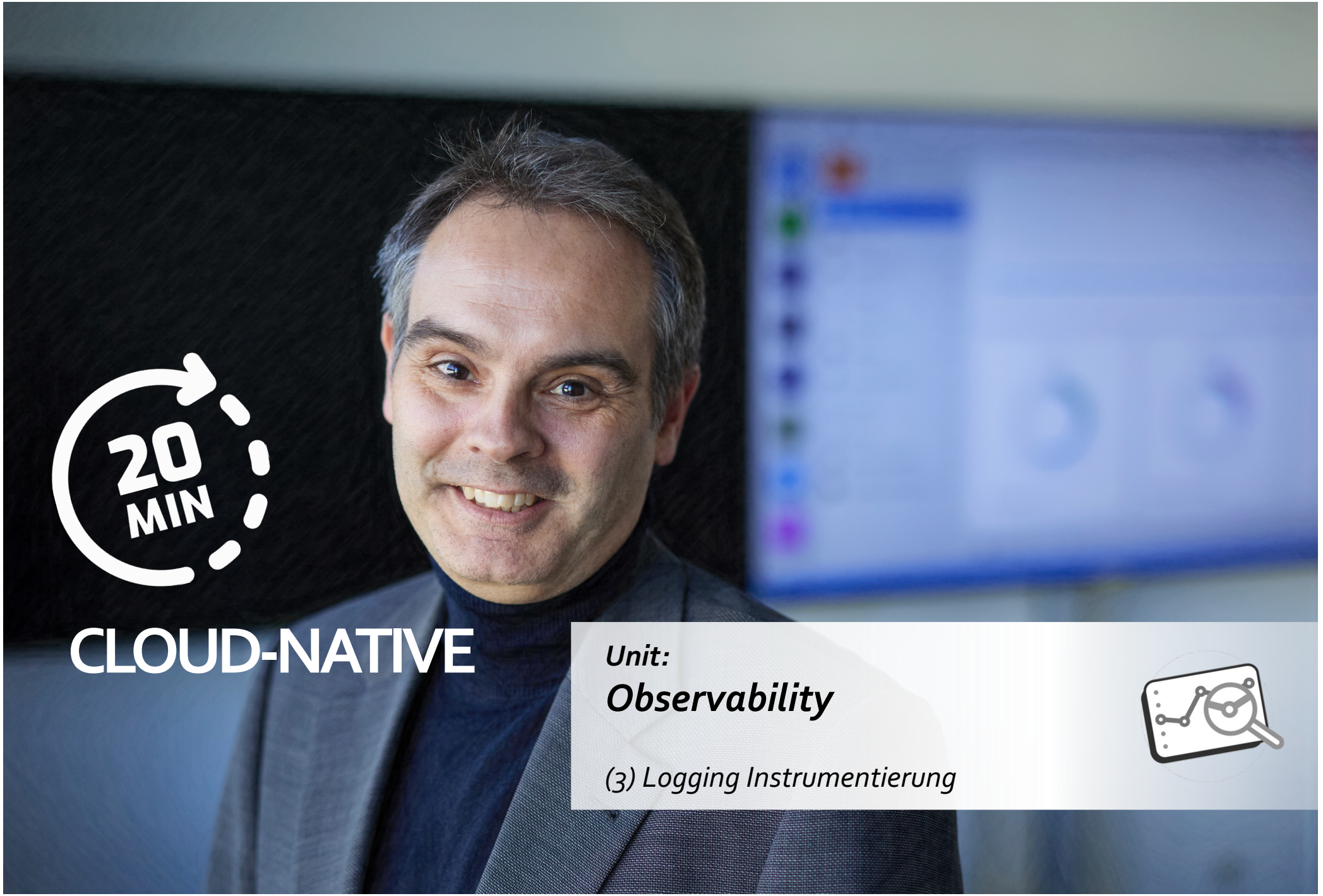



CLOUD-NATIVE

*Unit:*

## *Observability*

*(3) Logging Instrumentierung*



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

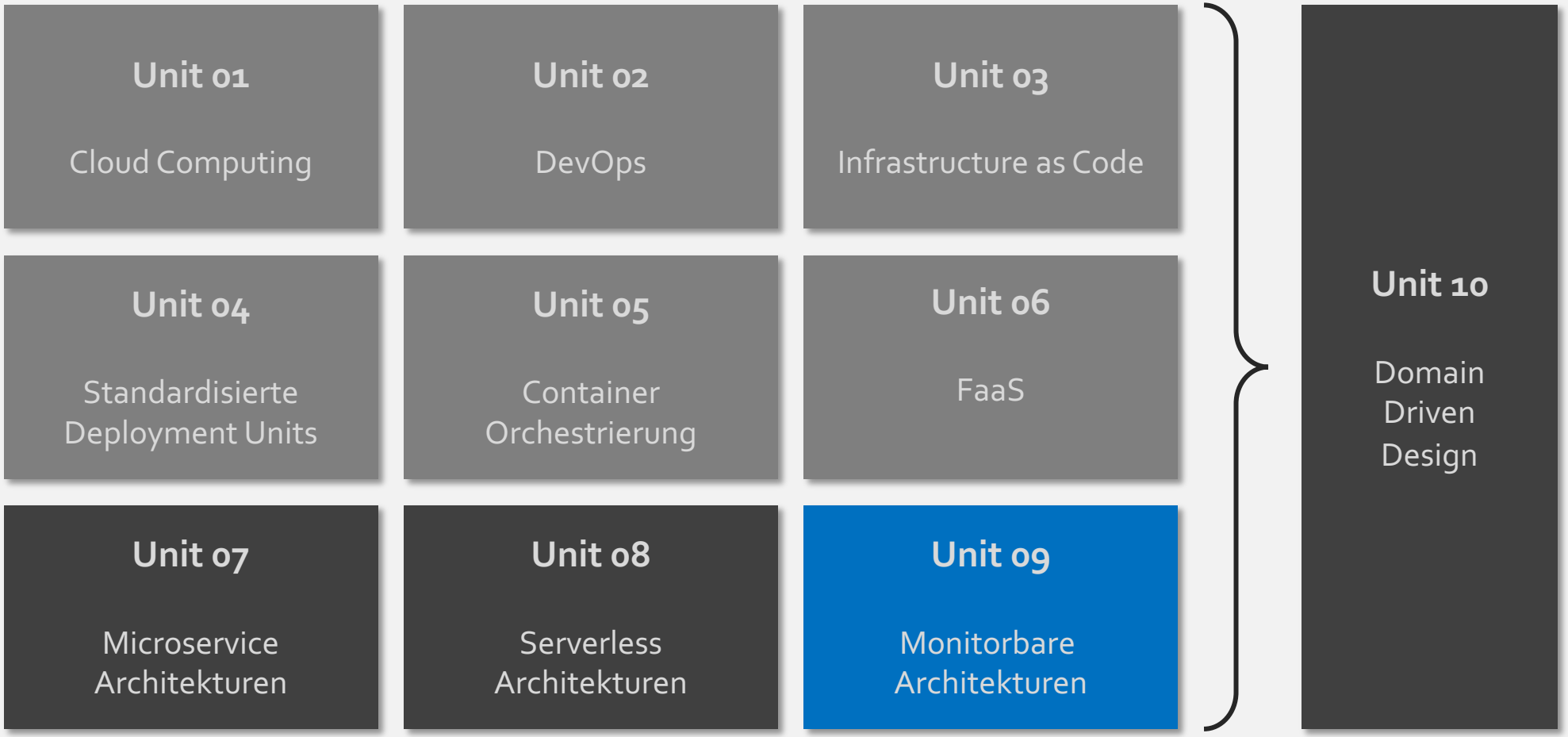
## 13.3 Automatisierte Instrumentierung

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

## 13.4 Zusammenfassung

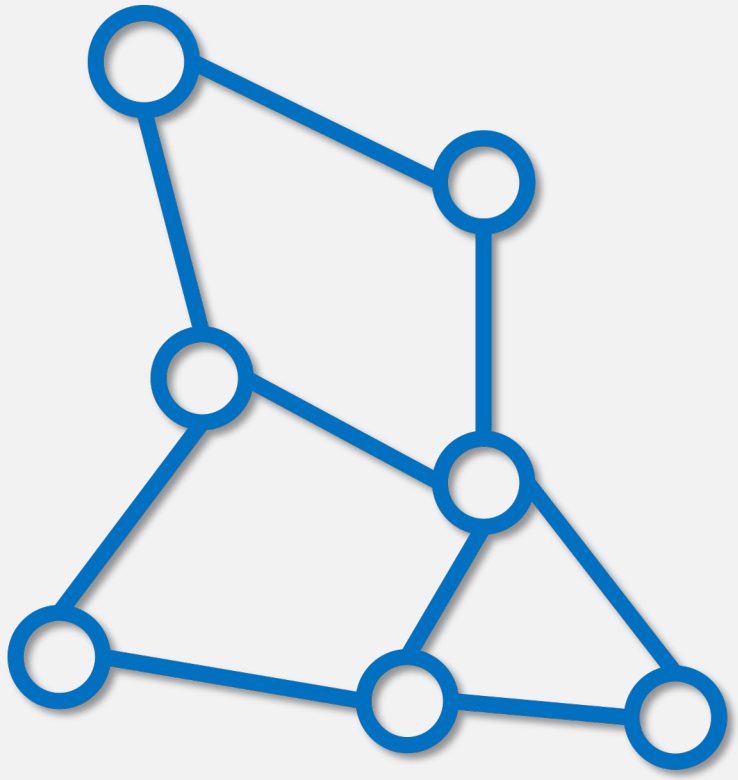
# INHALTSVERZEICHNIS

*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

*Logging*

- Mittels Logging (Protokollierung) lässt sich Einblick in **anwendungsspezifische Ereignisse** gewinnen, wenn diese von Prozessen ausgegeben werden
- Das Problem in Cloud-nativen Systemen ist, dass sich **Logs über zahlreiche Services verteilen** und Service für Service analysiert werden müssen
- Das kann Fehleranalysen komplex und zeitaufwändig machen
- Für Analysen ist es einfacher, wenn diese **Logs zentral** in einer durchsuchbaren Datenbank zentral hinterlegt wären
- Daher haben sich Tools zum **Log-Forwarding**, zur **Log-Aggregation** und für die zentralisierte **Log-Analyse** etabliert


*Log Routing*

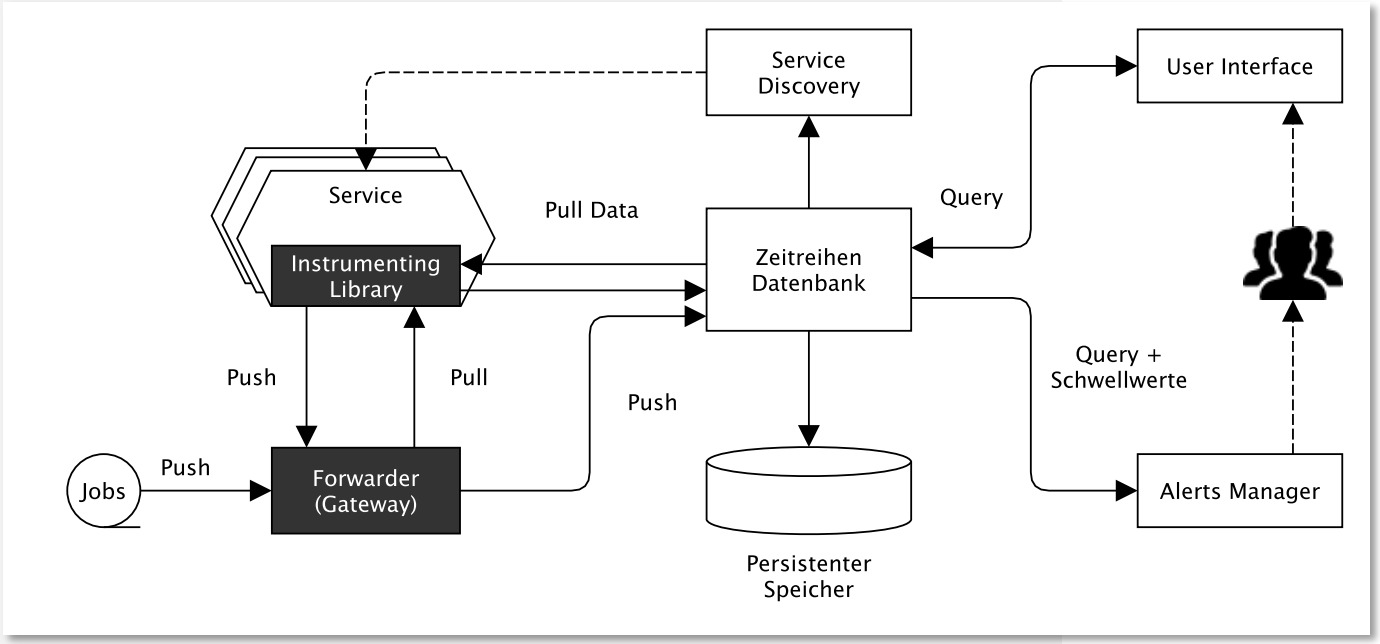
*Log Indexing + Querying*

*Log Analysis => Ops-Erkenntnisse*

# OBSERVABILITY ARCHITEKUR

*Logs als Stream von Ereignissen betrachten (12 Faktoren Prinzip Nr. 11!!!)*

- Logs machen das Verhalten einer laufenden Anwendung sichtbar
- Oft werden Logs in Log-Dateien geschrieben
- Implizit wird angenommen, dass diese Datei durch Systemadministratoren gelesen werden
- Nach dem **Prinzip Nr. 11** der 12 Faktoren sollen Logs als Stream von Ereignissen jedoch ungepuffert auf **stdout** geschrieben werden
- Container-Orchestratoren leiten diese Log-Streams dann in Dateien auf dem Host um




- **Log Router** fassen diese Log-Dateien zusammen und leiten diese an Zeitreihen-Datenbanken zur Archivierung weiter
- Diese Archivierungsziele sind für die Anwendung weder sichtbar noch konfigurierbar
- Logs sind somit ein Stream von aggregierten, nach Zeit sortierten Ereignissen aller laufenden Prozesse einer Anwendung
- Logs in ihrer rohen Form sind üblicherweise ein **Textformat mit einem Ereignis pro Zeile**

# ERINNERUNG: 12 FAKTOREN METHODE

*XI. Logs*

```
print("Ich bin ein Log-Entry in Python")
```

*Sehr einfaches Logging (log to stdout), kein Einsatz spezifischer Logging-Libraries (reicht schon aus, um Logs konsolidieren zu können)*

```
import json
print(json.dumps({
        "event": "Log-Entry",
        "message": "Ich bin ein Log-Entry",
        "language": "Python"
}))
```

*Sehr einfaches strukturiertes Logging (log JSON to stdout), ebenfalls ohne Einsatz spezifischer Logging-Libraries (reicht schon aus, um Logs konsolidieren und zielgerichteter auswerten zu können in einer Log-Konsolidierung)*

CLOUD NATIVE COMPUTING

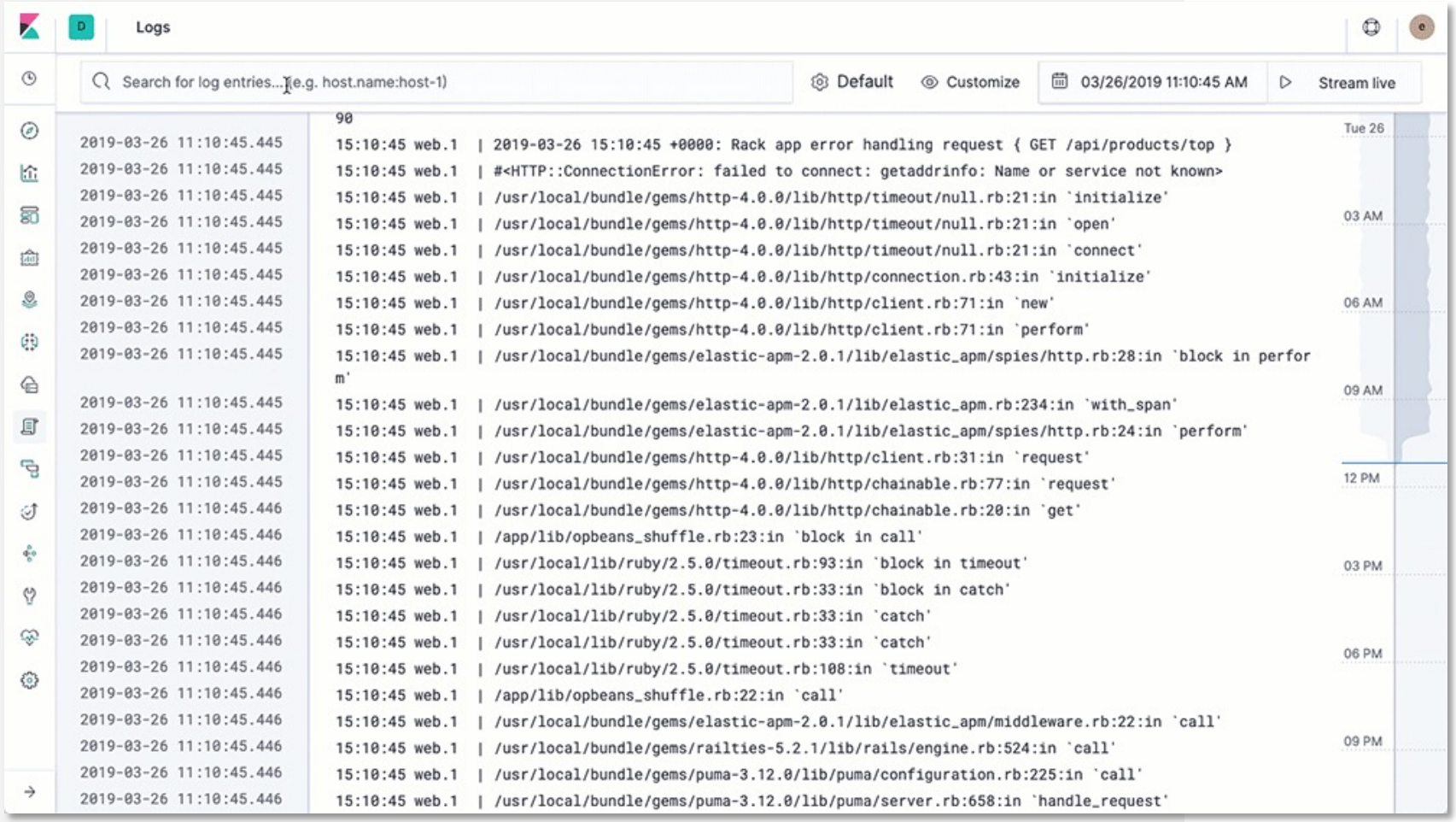
***Merke:***
*Komplizierter muss es mit einer funktionierenden Log-Konsolidierung nicht unbedingt sein.*

*Das Prinzip des Loggens mittels print()-Statements auf der Standard-Ausgabe (stdout) ist so alt wie die Programmierung selbst und daher für viele Programmierer so verbreitet, dass diese kaum bewusst wahrnehmen, dass Code für eine bessere Observierbarkeit instrumentiert wird.*

# LOGGING

*Konsolidierung von Logs mittels Unified Logging*

Mittels eines Unified Logging Layer lassen sich alle Logs eines Cloud-nativen Systems zentralisiert durchsuchen. In entsprechenden Datenbanken, wie bspw. ElasticSearch, lassen sich Log-Daten nach Dienst, App, Host, Rechenzentrum weiteren Kriterien filtern, sodass man in den aggregierten Logs nach unerwarteten Ereignissen fahnden kann.




**Beispiel: ElasticSearch, LogStash und Kibana (ELK-Stack)**

**Quelle:** elastic.io

# LOGGING

*Übliche Log-Level*

Da das Loggen aller Ereignisse die für Logging verfügbaren Ressourcen innerhalb kurzer Zeit aufbrauchen kann und die Auffindbarkeit bestimmter Ereignisse erschweren würde, werden meist die folgenden üblichen Dringlichkeitsstufen genutzt, mittels derer das Logging von Events bspw. in Production-, Staging-, Test-Environments ein- oder ausgeschaltet werden kann.

Überlicherweise werden nur Ereignisse bis zu einem Log Level dann auch geloggt.

Die Log-Ebene Info bedeutet also, das Ereignisse der Kategorien Fatal, Error, Warning und Info geloggt würden. Aber nicht die Ebenen Debug und Trace.

| Log Level | Beschreibung |
|---|---|
| **Fatal** | Fehler, welcher zur Terminierung der Anwendung führt. |
| **Error** | Laufzeitfehler, welcher die Funktion der Anwendung behindert, oder unerwarteter Programmfehler. |
| **Warning** | Aufruf einer veralteten Schnittstelle, fehlerhafter Aufruf einer Schnittstelle, Benutzerfehler oder ungünstiger Programmzustand. |
| **Info** | Laufzeitinformationen wie der Start und Stopp der Anwendung, Benutzeranmeldungen und -abmeldungen, sowie durchgeführte Geschäftstransaktionen. |
| **Debug** | Informationen zum Programmablauf. Wird im Normalfall nur in der Entwicklung oder zur Nachvollziehung eines Fehlers verwendet. |
| **Trace** | Detaillierte Verfolgung des Programmablaufs, insbesondere zur Nachvollziehung eines Programmierfehlers. |

# LOGGING

*Best Practices für die Logging Instrumentierung am Beispiel von Python*

- Die Standardbibliothek von Python enthält bereits ein eingebautes und flexibles Logging-Modul
- Es können Logging-Konfigurationen für unterschiedliche Logging-Erfordernisse erstellt werden
- Viele andere Sprachen wie bspw. Java bieten ähnliche Logging Bibliotheken wie bspw. log4j
- Das Logging-Modul von Python kann mittels Handlern Logs für verschiedene Zielorte (stdout, Log-Dateien, Remote-Server, etc.) anlegen
- Zu protokollierende Ereignisse werden an die entsprechenden Handler weitergeleitet und dort entsprechend verarbeitet

```python
import logging
logging.basicConfig(level=logging.WARNING)

logging.debug('This debug message will not be logged')
logging.info('This info message will not be logged')
logging.warning('This warning message will be logged')
logging.error('This error message will be logged')
logging.critical('This critical message will be logged')
```

# LOGGING

*Best Practices für die Logging Instrumentierung am Beispiel von Python (Log Level)*

Das Python-Logging-Modul bietet grundsätzlich weniger (und stellenweise auch anders benannte) Stufen als andere Logging-Bibliotheken.

Dies macht die Dinge allerdings auch etwas einfacher, indem so einige potenzielle Unklarheiten beseitigt werden.

| Log Level | Beschreibung |
|---|---|
| **DEBUG** | Dieses Level sollte ausschließlich für Debugging-Zwecke in der Entwicklung verwendet werden. |
| **INFO** | Dieses Level sollte genutzt werden, wenn etwas Interessantes - aber Erwartetes - passiert (z. B. wenn ein Kunde einen Kauf in einem Online-Shop tätigt). |
| **WARNING** | Diese Ebene sollte verwendet werden, wenn etwas Unerwartetes oder Ungewöhnliches passiert. Es handelt sich nicht um einen Fehler, aber Sie sollten darauf achten. Z.B. wenn ein Nutzer sich mit falschen Credentials anzumelden versucht. |
| **ERROR** | Diese Stufe ist für Dinge, die schief laufen, aber normalerweise wiederherstellbar sind (z. B. interne Ausnahmen wie nicht vorhandene Dateien, die aber behandelt werden können, oder APIs, die Fehlerergebnisse zurückgeben). |
| **CRITICAL** | Diese Stufe sollten Sie nur in Situtationen verwenden, die Services unbrauchbar machen und zu einem Shutdown führen. |

# LOGGING

*Best Practices der Logging Instrumentierung am Beispiel von Python*

```python
import logging
import os

logging.basicConfig(
    level=os.getenv('LOGLEVEL', logging.WARNING),
    format='%(asctime)s | %(name)s | %(levelname)s | %(message)s',
    datefmt='%Y-%m-%dT%H:%M:%S%z'
)

log = logging.getLogger("service-name")

log.debug('This debug message will not be logged')
log.info('This info message will not be logged')
log.warning('This warning message will be logged')
log.error('This error message will be logged')
log.critical('This critical message will be logged')
```

(1) (d.h. nicht in eine Datei!)
(2)
(3)
(4)
(5)

1. Logge auf stdout (das Unified Logging System + die Plattform macht den Rest)
2. Definiere den Loglevel über eine Umgebungsvariable
3. Logge immer auch den Service-Namen in dem Ereignis ausgelöst wurde
4. Logge den Zeitpunkt, den Level und das Event
5. Logge Zeitpunkte im ISO8601 Format inkl. Zeitzone

```
2021-02-20T12:33:46+0100 | service-name | WARNING | This warning message will be logged
2021-02-20T12:33:46+0100 | service-name | ERROR | This error message will be logged
2021-02-20T12:33:46+0100 | service-name | CRITICAL | This critical message will be logged
```

# LOGGING

*Strukturiertes Logging am Beispiel von Python*

- Beim strukturierten Logging werden Lognachrichten in einem strukturierten Format protokolliert
- Das erleichtert es, Informationen aus den Lognachrichten zu extrahieren und zu analysieren
- Beim unstrukturiertem Logging sind oft aufwändige Parsing-Operationen erforderlich
- Es werden typischerweise standardisierte Format wie JSON oder XML genutzt
- Strukturiertes Logging ist besonders nützlich in komplexen Anwendungen wie es Cloud-native Systeme nun einmal sind
- Entwickler und Systemadministratoren können schneller und einfacher nach bestimmten Mustern oder Ereignissen in den Logdaten suchen und problematische Bereiche identifizieren

Wenn man Loggen nicht nur als „Debugging" Werkzeug, sondern als Mittel einer ganzheitlichen Observability ansieht, kann es Sinn machen den Log-Level auf INFO zu setzen, um auch ganz reguläre fachliche Ereignisse zu erfassen und nicht nur Ereignisse von Ausnahmesituationen.

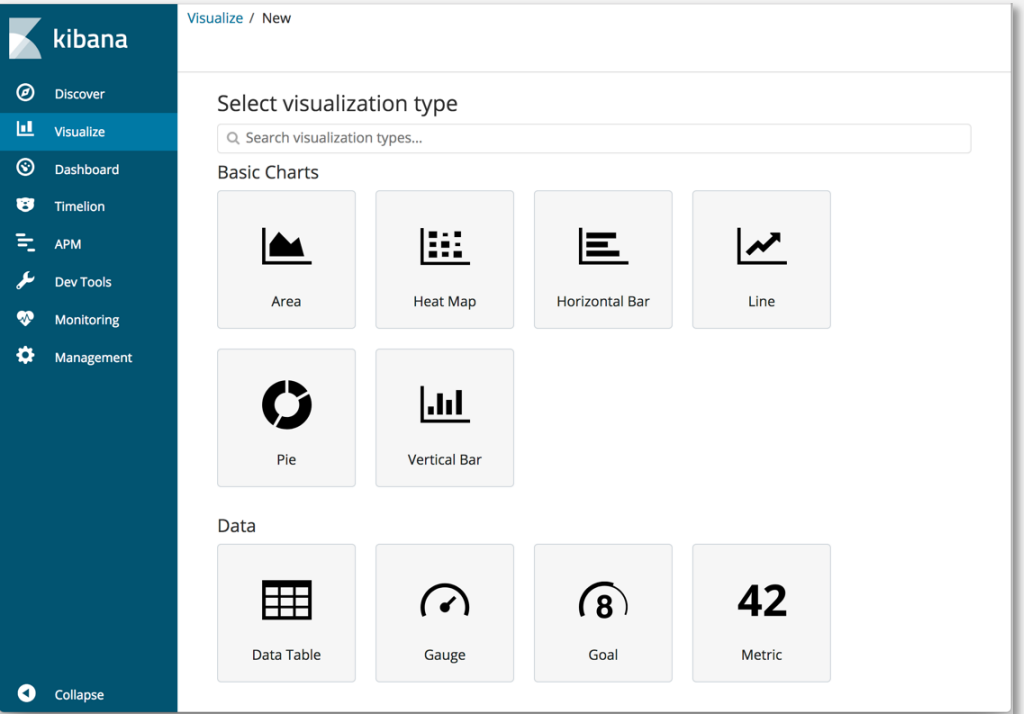
```python
import json, logging, os
from fastapi import FastAPI
from fastapi import HTTPException

logging.basicConfig(level=os.getenv('LOGLEVEL', logging.INFO))

app = FastAPI()

# Beispiel-Benutzerdaten
users = {
    "user1": {"password": "pass1"},
    "user2": {"password": "pass2"}
}

@app.post("/login")
async def login(username: str, password: str):
    if username not in users or users[username]["password"] != password:

        logging.warning(json.dumps({
          "usr": username, "operation": "login", "result": "failed"
        }))

        raise HTTPException(status_code=401, detail="Invalid login")

    logging.info(json.dumps({
      "usr": username, "operation": "login", "result": "success"
    }))

    return {"message": "Login successful!"}
```

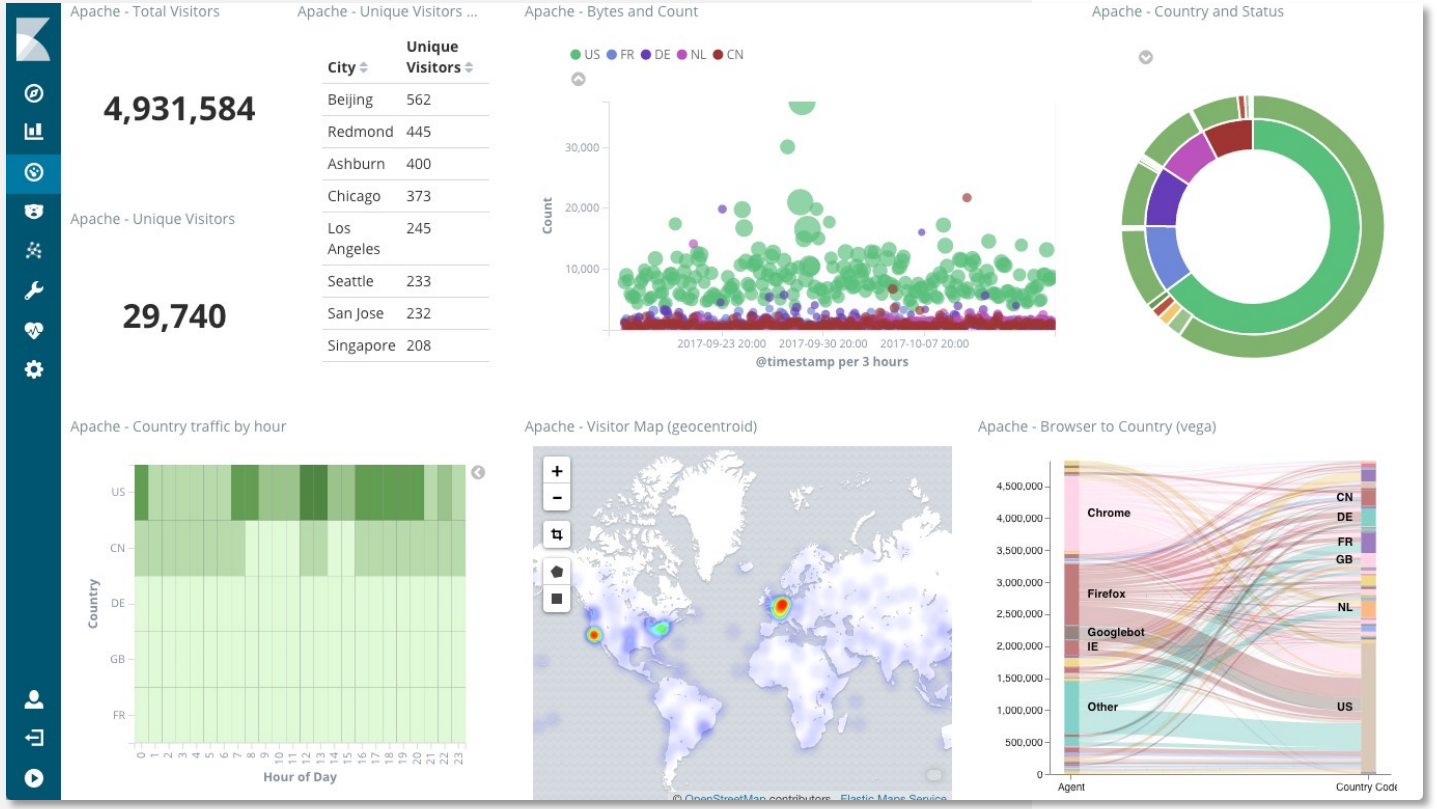
# LOGGING

*Visualisieren von Logs*

Mittels Queries lassen sich auch Häufigkeiten von Treffern in Form von Diagrammen, Karten und andere Arten von Visualisierungen unter Verwendung der zentralisierten Log-Daten erstellen.

Um Karten zu erstellen, sollten Logs Informationen wie geografische Koordinaten enthalten. Für quantitative Darstellungen sollten Logs numerische Daten beinhalten.



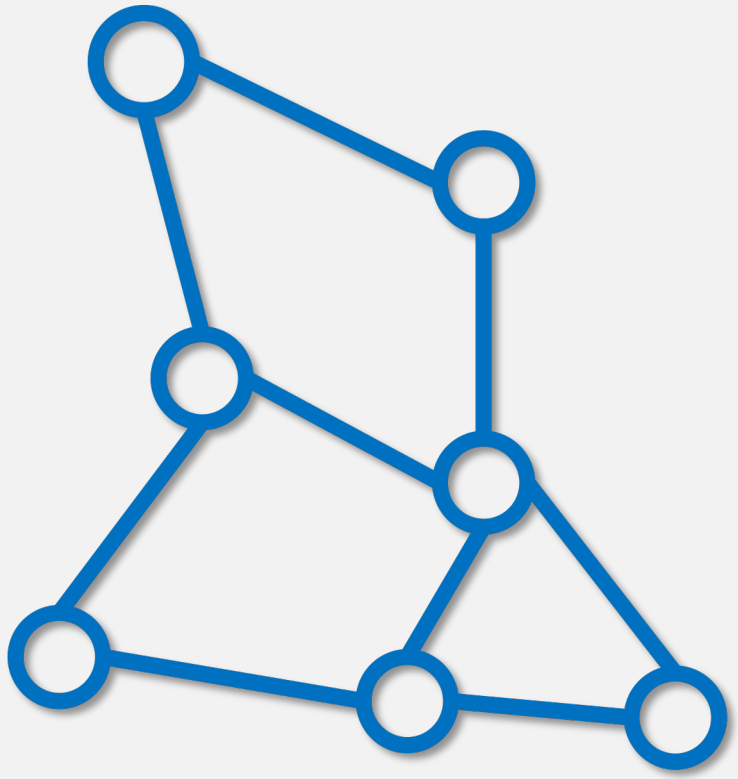

**Quelle:** elastic.io

Derartige Query-basierte Visualisierungen werden häufig in Dashboards zusammengefasst, um ein ganzheitliches Bild von Log- und Monitoring-Daten aufzubereiten, die es im Betrieb (Ops) ermöglichen Echtzeit-Darstellungen von Systemzuständen und -verhalten darzustellen (z.B. eine Häufung von fehlerhaften Login-Versuchen).

# AUSBLICK



- Beobachtbarkeit von Systemen
- Metriken
- Logs
- Tracing



# KONTAKT

*Disclaimer*

**Nane Kratzke**

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de