

Cloud Computing Observability

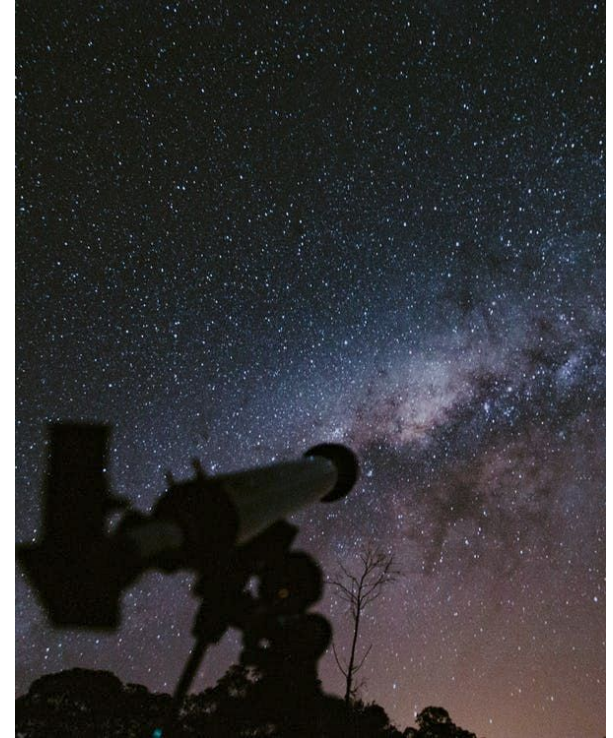
Was ist eigentlich Observability?

Wie gut weiß ich über den internen Zustand meines Systems Bescheid, wenn ich mir seine Ausgaben ansehe? (vgl. [Kal])

Was hat so ein modernes System als Ausgabe?

- Eine Webseite?
- Eine REST-API?
- E-Mails an den Admin?
- Ein spärlich gepflegtes Logfile?

Was ist, wenn das System aus vielen Microservices besteht?



Was ist eigentlich Observability?

“Observability” meint heute typischerweise drei Themen:

- Logs
- Metriken
- Traces

Alles zusammen ermöglicht einen Einblick auch in verteilte Microservices.



Das Ziel von Observability: Diagnosability

Was verstehe ich darunter?

*“**Strukturiertes** Vorgehen, um eine Anwendung im **Vorfeld** so mit Messfühlern auszustatten, dass ich im Fehlerfall den **Fehler schnell erkennen** kann und die zur **Behebung notwendigen Informationen** besitze.”*

Ein System ist gut diagnostizierbar, wenn man gesunde und ungesunde Zustände leicht erkennen und beheben kann

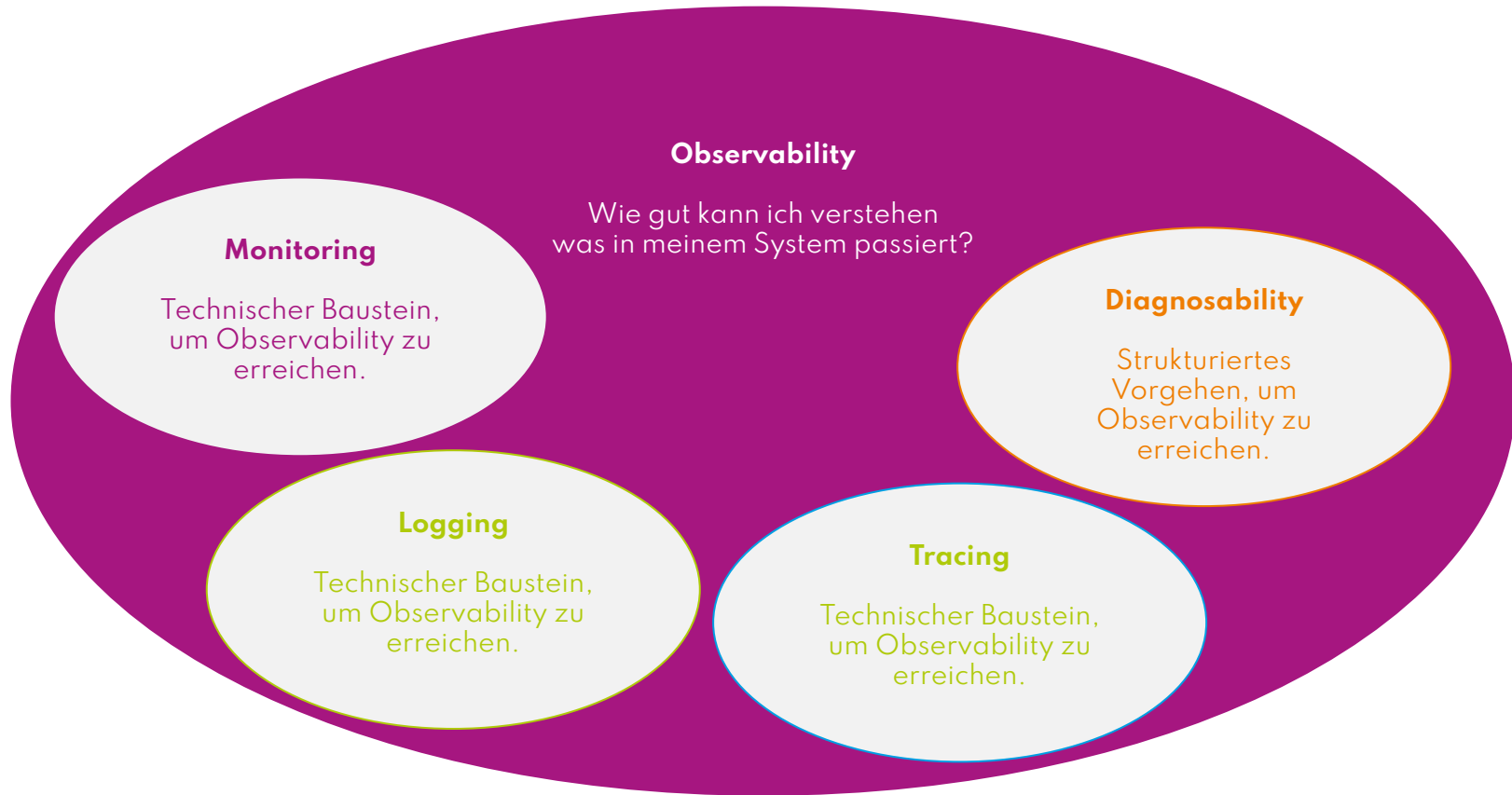
Ein diagnostizierbares System hat eine



- kurze Mean Time to Detect (MTDD)
- kurze Mean Time to Repair (MTTR)

und somit zu einer kurze Zeitspanne, in der Fehler unerkannt sind und überhaupt existieren.

Und was ist mit Observability und Monitoring?



Diagnosability: Strukturiertes Vorgehen ist notwendig, um das System nicht zu sehr zu beeinflussen

1. **Überblick schaffen:**

- a. Was sind zentrale Komponenten, z. B. Login, etc.
- b. Was sind unterstützende Komponenten, z. B. Batch-, Loader-Jobs, etc.

2. **Fehlergrenzen identifizieren:**

- a. Interne Fehlergrenzen: Schichten / Use Cases
- b. Externe Fehlergrenzen: Ein- und ausgehende Aufrufe

3. **Fehlerklassen definieren:**


- a. Schweregrade: Betrieb weiterhin möglich, Keine Auswirkungen vor Kunde, etc.
- b. Auswirkungen: Kunden stehen bestimmte Funktionen nicht zur Verfügung, etc.

4. **Laufzeitdaten bestimmen, die zur Erkennung notwendigen sind:**

- a. Einheitlichkeit: Daten haben dieselbe Bedeutung; Einheitliche Datenformate sind definiert, etc.
- b. Klar definiert: Es ist ersichtlich, was die Metrik bedeutet, z. B. CPU-Load

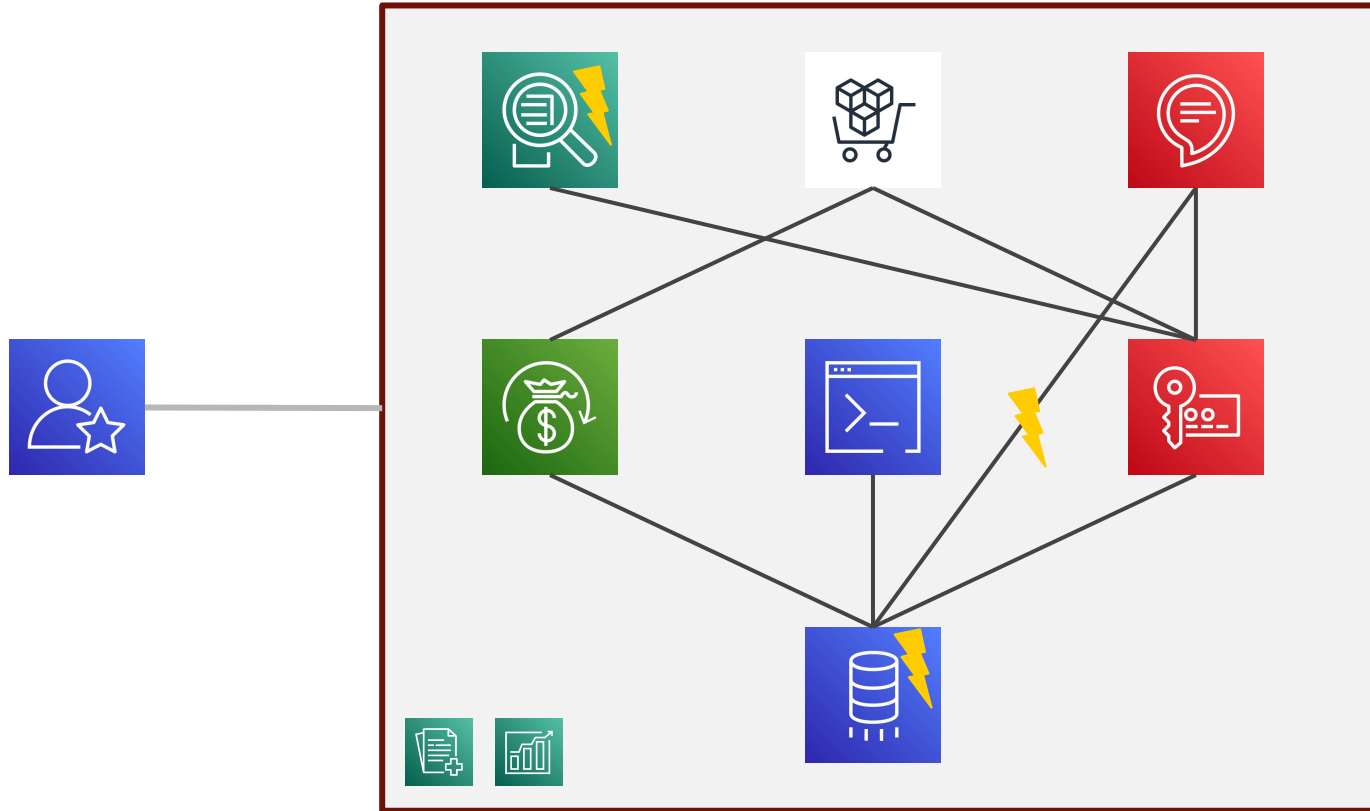
5. **Handlungen definieren:**

- a. Alerting: Wer wird wann benachrichtigt
- b. Playbooks für Fehler erstellen: Wie komme ich an die Daten etc.

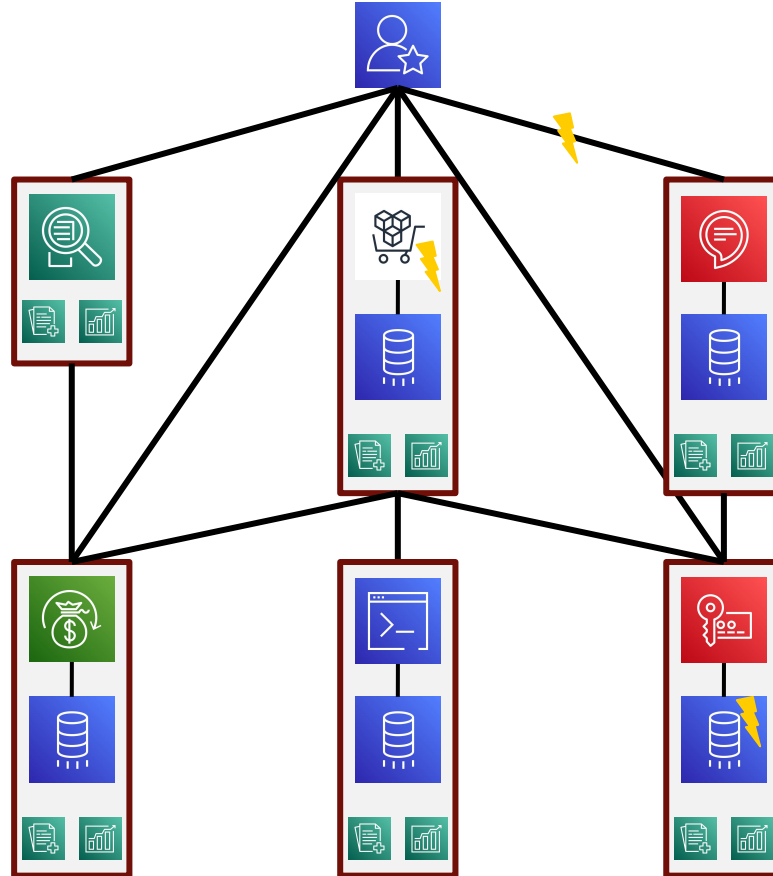


Warum (Cloud-)Observability?

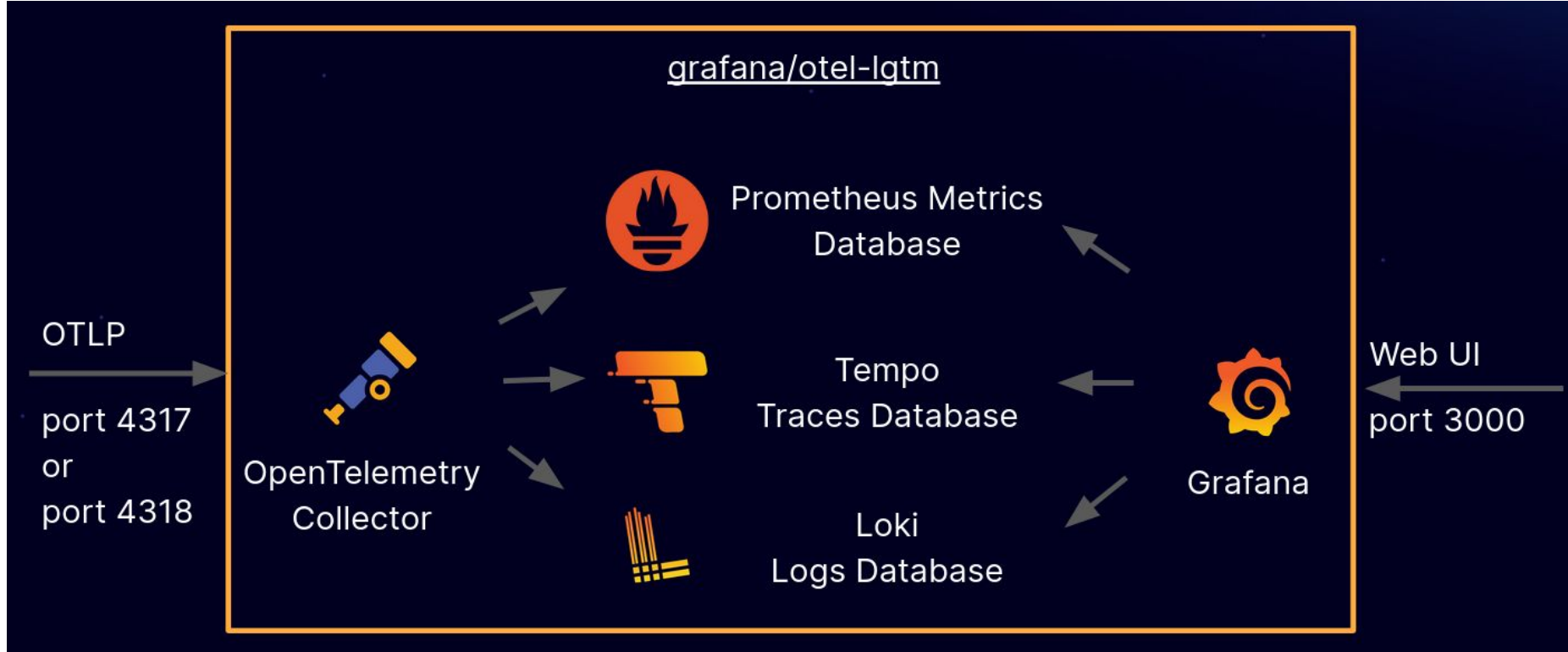
Damals: Der Monolith



Heute: Die Microservices



Der (Open-Source-)Stack



OpenTelemetry (OTel) ist ein Open-Source-Framework für Observability, mit dem Entwicklungsteams Telemetriedaten in einem einzigen, einheitlichen Format generieren, verarbeiten und übertragen können. Dieses Format wurde von der Cloud Native Computing Foundation ([CNCF](#)) entwickelt, um standardisierte Protokolle und Tools zum Erfassen und Weiterleiten von Metriken, Logs und Traces an Überwachungsplattformen bereitzustellen.

OpenTelemetry stellt anbieterneutrale SDKs, APIs und Tools bereit, um Ihre Daten zur Analyse an beliebige Observability-Backends übermitteln zu können.

[OpenTelemetry](#) entwickelt sich schnell zum vorherrschenden Observability-Telemetriestandard in Cloud-nativen Anwendungen. Die Einführung von OpenTelemetry ist wichtig für alle Unternehmen, die sich auf zukünftige Datenanforderungen vorbereiten möchten, ohne sich an einzelne Anbieter oder die Einschränkungen vorhandener Technologien zu binden.

Der Grafana Stack ist komplett OTeL kompatibel!

Logs



Grafana loki

Logs

- Logs sind ein wichtiger Bestandteil jedes IT-Systems.
- In der Cloud müssen wir Logs von vielen verschiedenen Services gleichzeitig betrachten.
- Jedes System hat Logs. Aber immer in einem anderen Format.

- Beispiel: **Quarkus**-Webservice:

```
2023-01-10 20:56:42,122 DEBUG [io.qua.mic.run.bin.ver.VertxHttpServerMetrics]
(vert.x-eventloop-thread-11) requestRouted null HttpRequestMetric
[initialPath=/q/metrics, currentRoutePath=null, templatePath=null,
request=io.vertx.core.http.impl.Http1xServerRequest@512b1fd6]
```

- Beispiel: **Nginx**-Webserver:

```
2a02:c207:3005:5132::1 - - [10/Jan/2023:00:00:13 +0100] 0.000
repo.saturn.codefoundry.de "POST /api/v4/jobs/request HTTP/1.1" 957 204 0 "-"
"gitlab-runner 15.6.1 (15-6-stable; go1.18.8; linux/amd64)"
```

Logging. Bitte schön strukturiert. Bitte gut überlegt

- Definiert ein Log-Format und stellt sicher, dass alle Services das gleiche Format nutzen.
- Definiert einen Diagnose-Kontext = Informationen die im Fehlerfall helfen, z. B.
 - TraceId
 - UserId
 - SessionId
- Nutzt Structured Logging, z. B. JSON, GELF etc.
 - Vereinfacht das Handling in der Analyse
- Nutzt asynchrones Logging (sofern das die Empfänger unterstützen), um Blockaden zu verhindern
 - OTLP oder Scraping

Logging. Bitte schön strukturiert. Bitte gut überlegt

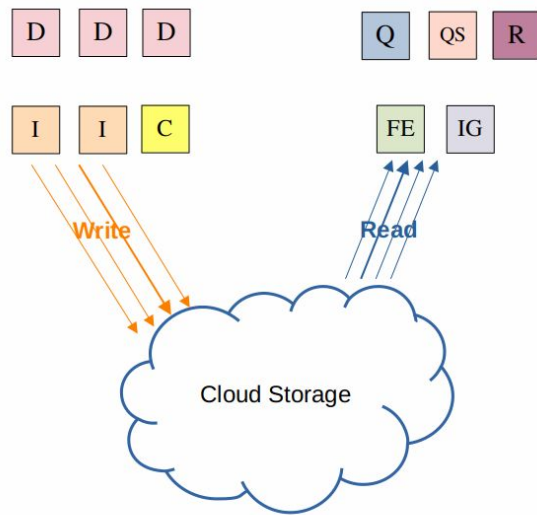
- Loggt nicht zu viel, aber
 - jede Exception
 - jeden Fehler
 - jede sinnvolle Information
 - ... aber nicht mehrfach.
- Nutzt Log-Level. Definiert dazu welche Kategorie welches Level haben soll, z.B.:
 - WARN: Fehler, die einen einzelnen Request betreffen, aber nicht die Stabilität des Services
 - ERROR: Fehler, die die Stabilität des Services betreffen

Loki

Loki ist der Log-Aggregator im Grafana-Ökosystem.

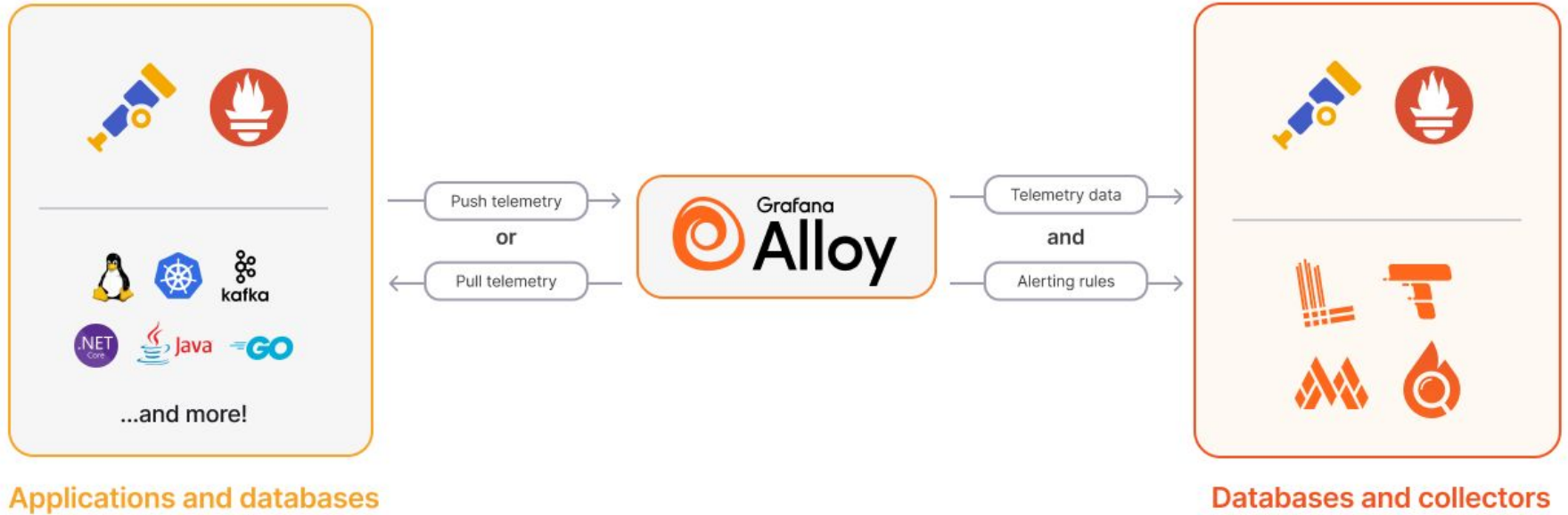
Passiver Log-Speicher - Logs müssen anders beschafft werden.

Grafana Alloy/OTEL Collector ist dafür zuständig die Logs einzusammeln und nach Loki zu senden.



Legend	
C	Compactor
D	Distributor
FE	Query frontend
I	Ingestor
IG	Index Gateway
Q	Querier
QS	Query Scheduler
R	Ruler

Grafana Alloy



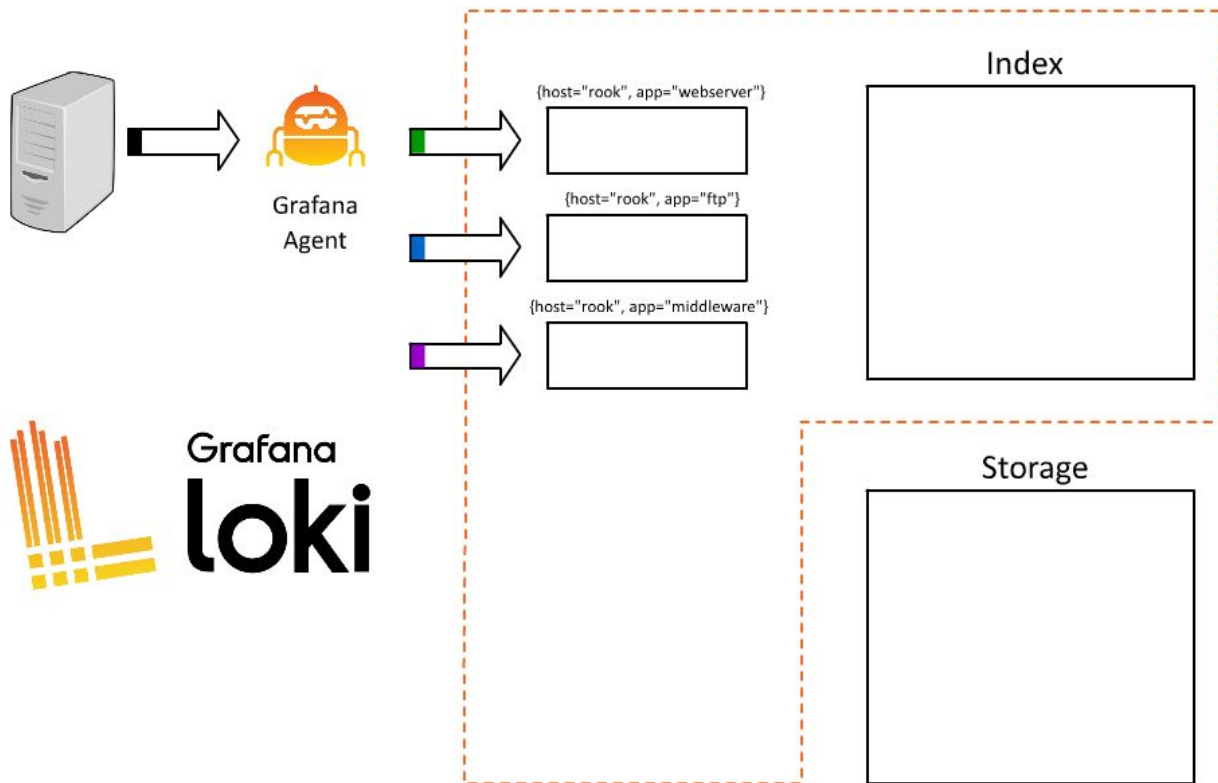
Alloy Config

```
local.file_match "local_files" {  
  path_targets = [{"__path__" = "/var/log/*.log"}]  
  sync_period = "5s"  
}  
  
loki.source.file "log_scrape" {  
  targets      = local.file_match.local_files.targets  
  forward_to   = [loki.write.grafana_loki.receiver]  
  tail_from_end = true  
}  
  
loki.write "grafana_loki" {  
  endpoint {  
    url = "http://localhost:3100/loki/api/v1/push"  
  }  
}
```

Zum selbst ausprobieren:

<https://killercoda.com/grafana-labs/course/alloy/send-logs-to-loki>

Loki Ingestion



Vorsicht: Labels nicht zu dynamisch verteilen!

<https://grafana.com/blog/2023/12/20/the-concise-guide-to-grafana-loki-everything-you-need-to-know-about-labels/>

LogQL

Loki hat eine eigene Query Language.

```
count_over_time({foo="bar"}[1m]) > 10
```

```
{host="$host", job="systemd-journal"}
```

```
| json
```

```
| line_format "{{ .unit }}: {{ .MESSAGE }}"
```

```
|= "$search"
```

Loki Abfragen

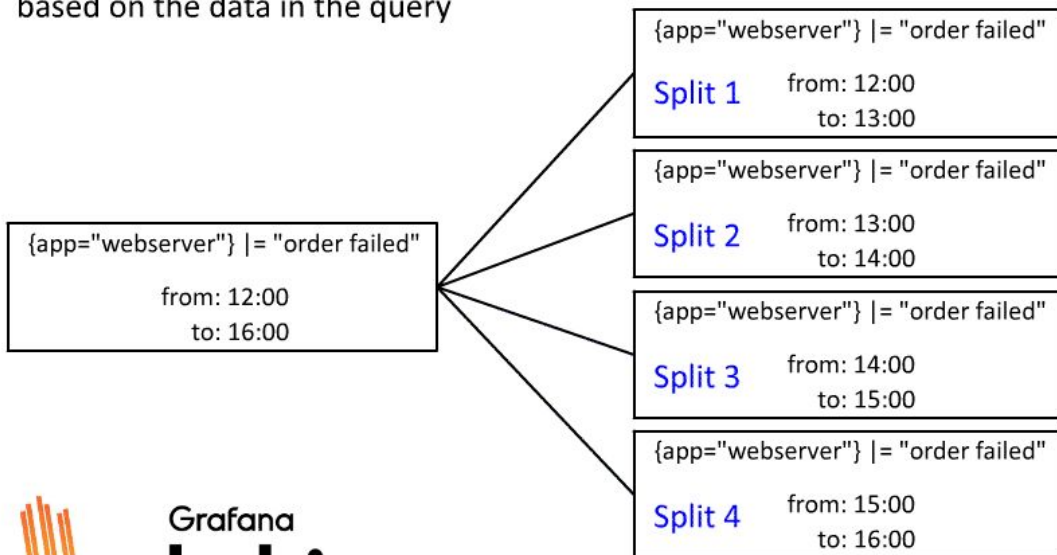
When Loki executes a query,
first the query is split into smaller time ranges
based on the value of:

`split_queries_by_interval`



Loki Abfragen

Each of these splits is then sharded
based on the data in the query



Loki Abfragen

First the query is split and sharded into subqueries

Query



Metriken



Metriken: Grundlagen

Metriken sind Messwerte, die den aktuellen Zustand des Systems abbilden.

Beispiele:

- CPU-Auslastung
- Größe des JVM-Heaps
- Anzahl der Aufrufe einer Schnittstelle

Die Metriken werden vom zu überwachenden System bereitgestellt.

Metriken können auch Metadaten haben.

Prometheus

- Prometheus (seit 2012) hat sich als Monitoring-Tool im Cloud Native Umfeld durchgesetzt.
- Open Source Implementierung von Borgmon (Borg ist das interne Kubernetes von Google)
- Zeichnet sich aus durch:
 - Einfache Nutzbarkeit (keine horizontale Skalierung, Long-Term Storage, etc.)
 - Gute Integration in Kubernetes, z. B. nutzen der Service Discovery
 - Zeitreihendatenbank für numerische Zeitreihen
 - Kollektoren für Metriken und text-basiertes Format für Metriken
 - Abfragesprache (PromQL) für Zeitreihen
 - Alerting auf Basis von Zeitreihen
- Lösungen mit zusätzlichen Funktionen von Anbietern existieren, die kompatibel sind
 - Unterstützen PromQL, z. B. InfluxDB, Elasticsearch in Zukunft 😊
 - Metrik-Format, z. B. Grafana Cloud

Prometheus

Prometheus zieht alle 15s Metriken von allen konfigurierten Zielen.

Der Transport erfolgt meist über HTTP: GET /metrics

Hier passiert noch keine Aggregation. Dafür ist der Konsument zuständig.

```
# HELP process_open_fds Number of open file descriptors.
# TYPE process_open_fds gauge
process_open_fds 8
# HELP process_start_time_seconds Start time of the process since unix epoch in seconds.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1.65566242485e+09
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 1.782685696e+09

# HELP http_server_requests_seconds
# TYPE http_server_requests_seconds summary
http_server_requests_seconds_count{method="GET",status="200",uri="/tle",} 1.0
http_server_requests_seconds_sum{method="GET",status="200",uri="/tle",} 8.183356641
# HELP http_server_requests_seconds_max
# TYPE http_server_requests_seconds_max gauge
http_server_requests_seconds_max{method="GET",status="200",uri="/tle",} 8.183356641
```

Metric Types

Metriken haben verschiedene Typen, um Daten abzubilden:

- **Counter**

Eine Zahl, die entweder inkrementiert oder zurückgesetzt werden kann.

- **Gauge**

Eine Metrik, die sich beliebig nach oben oder unten verändern kann.

- **Histogram**

Werte in "Buckets" - z.B. die Dauer von Requests

- **Summary**

Ähnlich zum Histogramm, fasst Werte zusammen

PromQL

PromQL ist eine mächtige Query Language, um die gewünschten Metriken zu filtern / aggregieren.

Die ganze Zeitreihe:

```
http_requests_total
```

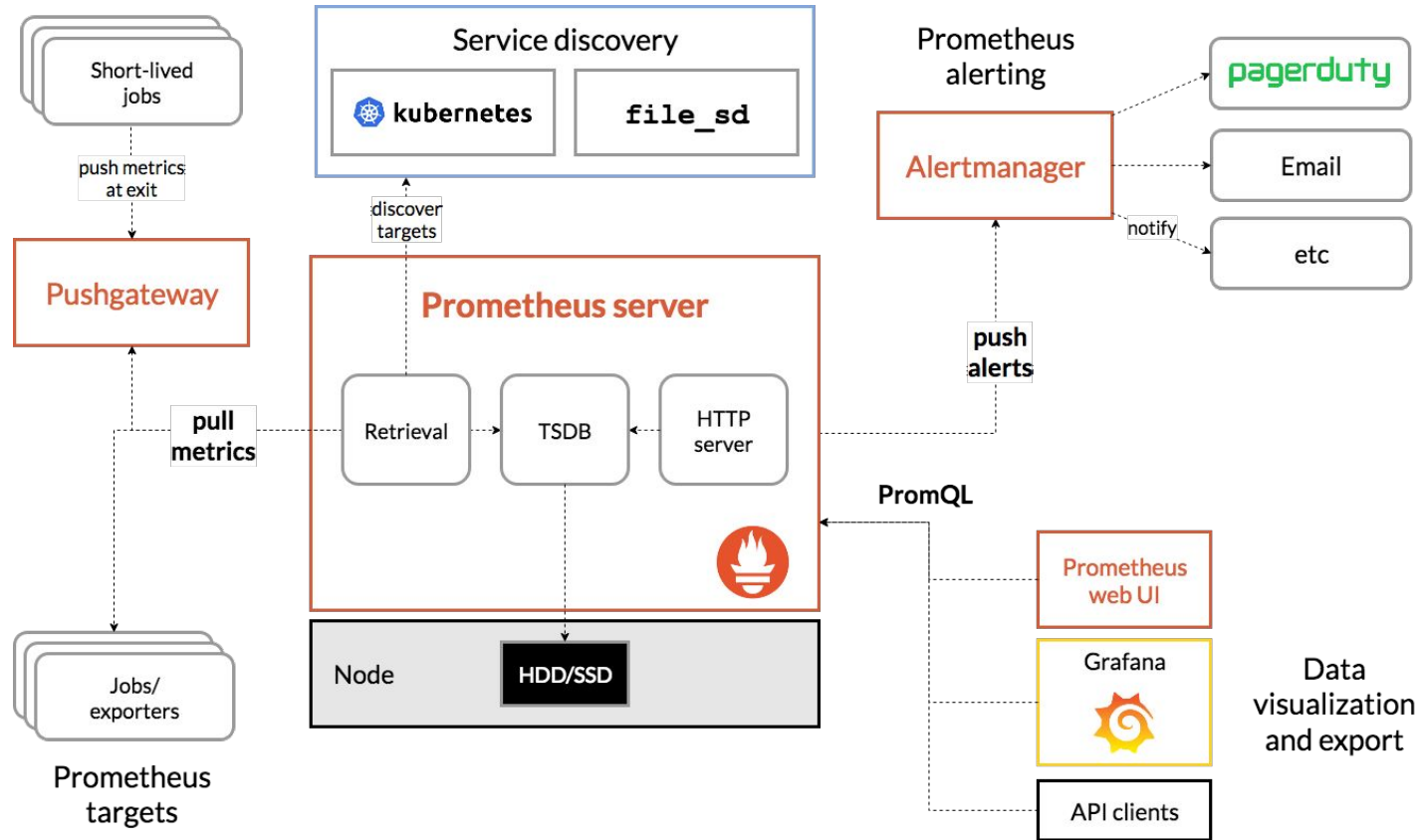
Filter nach Labels:

```
http_requests_total{job="apiserver", handler="/api/comments"}
```

Rate berechnen:

```
rate(http_requests_total[5m])[30m:1m]
```

Prometheus: Architektur



Konfiguration der Anwendung - Beispiel Quarkus & Micrometer

Gradle

```
implementation("io.quarkus:quarkus-micrometer-registry-prometheus")
```

Maven

```
<dependency>  
  <groupId>io.quarkus</groupId>  
  <artifactId>quarkus-micrometer-registry-prometheus</artifactId>  
</dependency>
```


Konfiguration der Anwendung - Quarkus & Microprofile-Metrics

```
@Gauge(unit = MetricUnits.NONE, name = "queueSize")
public int getQueueSize() {
    return queue.size;
}
```

Gradle

```
implementation("io.quarkus:quarkus-smallrye-metrics")
```

Maven

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-metrics</artifactId>
</dependency>
```

Konfiguration von Prometheus - Einsammeln der Metriken

```
global:
  scrape_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.

scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
  - job_name: "prometheus"
    static_configs:
      - targets: ["localhost:9090"]

  - job_name: "Saturn node exporter"
    scheme: https
    static_configs:
      - targets: ["saturn.qaware.de"]
    basic_auth:
      username: "*****"
      password: "*****"
```

Prometheus großer Bruder: Mimir

- Distributed Tracing: Technik zum Nachvollziehen von Aufrufen und Abläufen in verteilten Software Systemen.
- Heutige Überlegungen gehen zurück auf das Paper: [Dapper, a Large-Scale Distributed Systems Tracing Infrastructure](#)
- Idee: Jeder Service schickt Informationen vom Aufrufer zum nächsten Service weiter und auch wieder zurück. Dabei entsteht ein gerichteter Graph.
 - Jeder Service muss instrumentiert sein. → Ansonsten entsteht eine Lücke.
 - Die kleinste Einheit heißt Span. Ein Span hat eine Dauer und besitzt beschreibende Informationen. Ein Trace ist eine Menge aus 1..n Spans
- Viele Implementierungen existieren für unterschiedliche Programmiersprachen und Technologien.
 - <https://opentracing.io/docs/supported-languages/>
 - <https://opentracing.io/docs/supported-tracers/>
- Versuche der Community das zu vereinheitlichen: OpenTelemetry

Traces

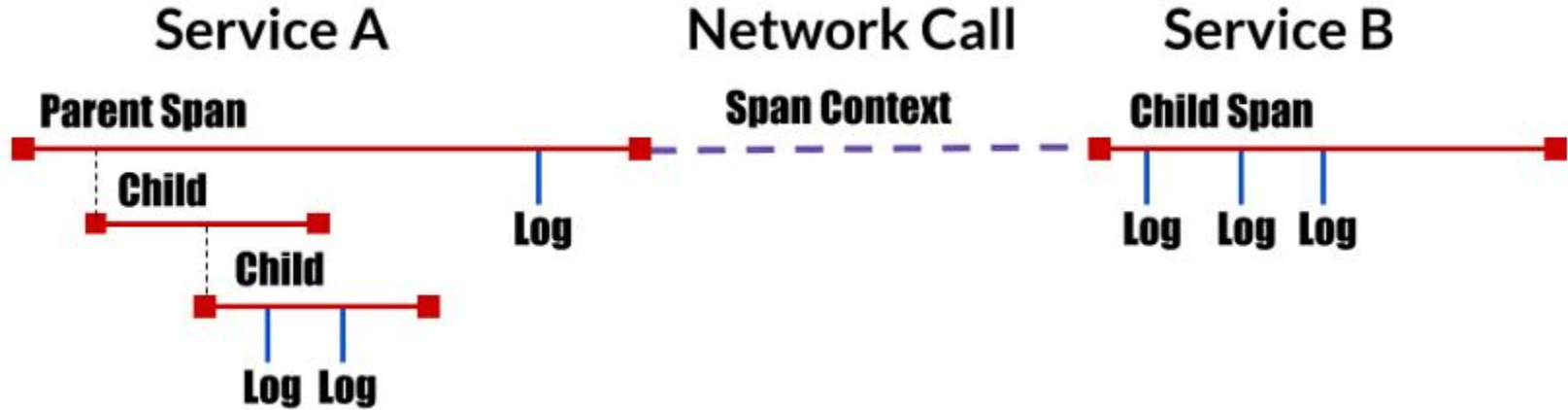


Grafana Tempo

Distributed Tracing: Im Wesentlichen basiert alles auf Google Dapper.

- Distributed Tracing: Technik zum Nachvollziehen von Aufrufen und Abläufen in verteilten Software Systemen.
- Heutige Überlegungen gehen zurück auf das Paper: [Dapper, a Large-Scale Distributed Systems Tracing Infrastructure](#)
- Idee: Jeder Service schickt Informationen vom Aufrufer zum nächsten Service weiter und auch wieder zurück. Dabei entsteht ein gerichteter Graph.
 - Jeder Service muss instrumentiert sein. → Ansonsten entsteht eine Lücke.
 - Die kleinste Einheit heißt Span. Ein Span hat eine Dauer und besitzt beschreibende Informationen. Ein Trace ist eine Menge aus 1..n Spans
- Viele Implementierungen existieren für unterschiedliche Programmiersprachen und Technologien.
 - <https://opentracing.io/docs/supported-languages/>
 - <https://opentracing.io/docs/supported-tracers/>
- Mittlerweile wird vor Allem **OpenTelemetry** genutzt

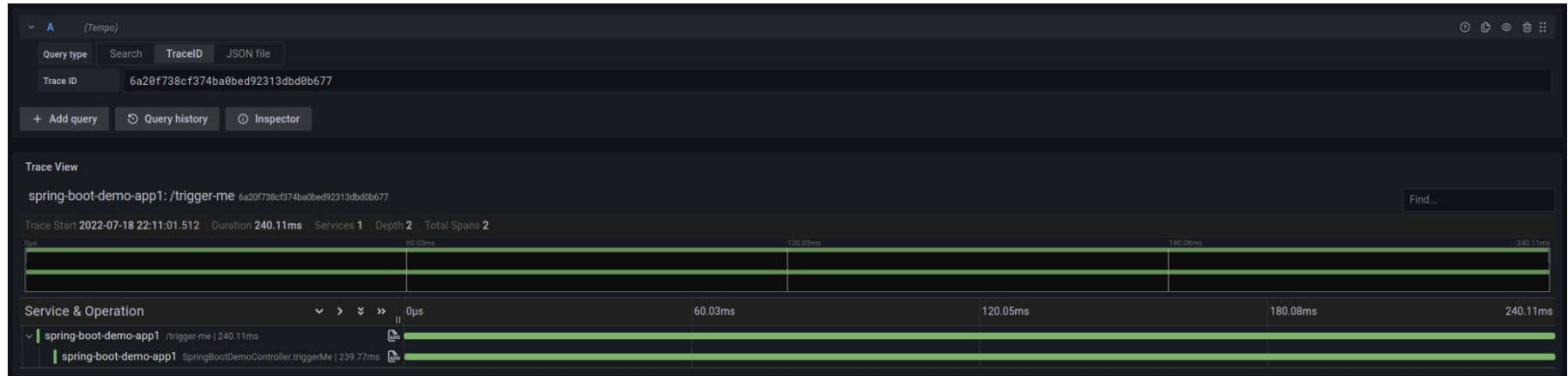
Distributed Tracing: Im Wesentlichen basiert alles auf Google Dapper.



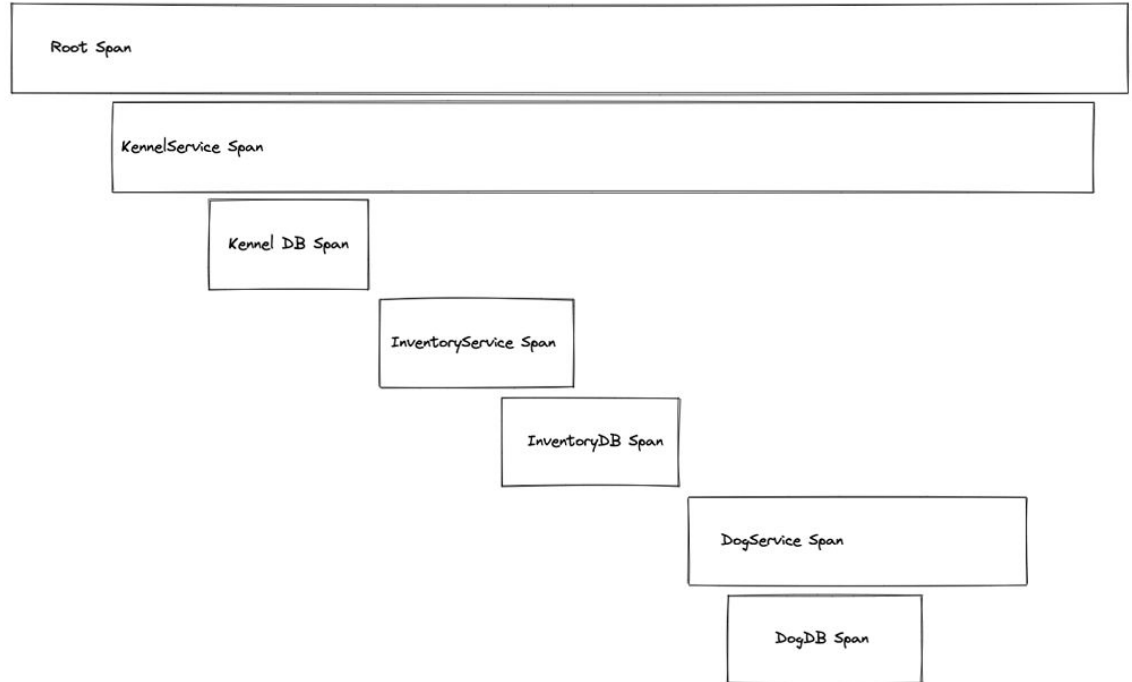
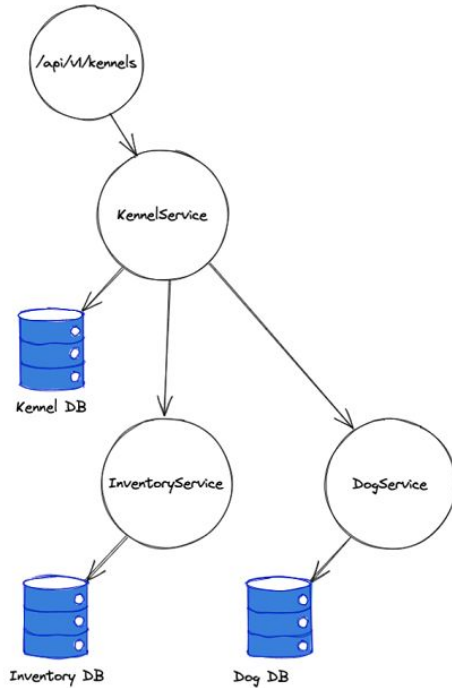
Traces: Grundlagen

Traces visualisieren, welche Wege ein Request durch die Microservices genommen hat...

... und wo vielleicht ein Fehler passiert ist!



Traces: Grundlagen



Grafana Tempo

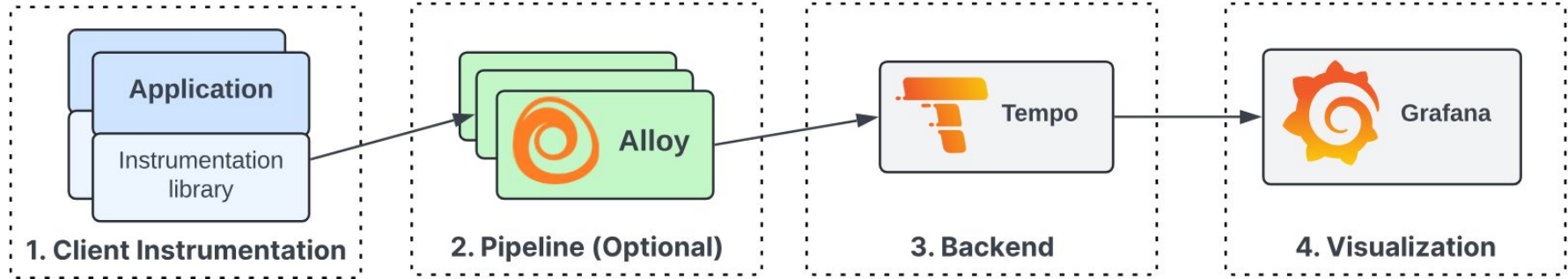
Traces im Grafana-Ökosystem!

SDKs für verschiedene Plattformen

Kompatibel zu verschiedenen Tracing Agents:

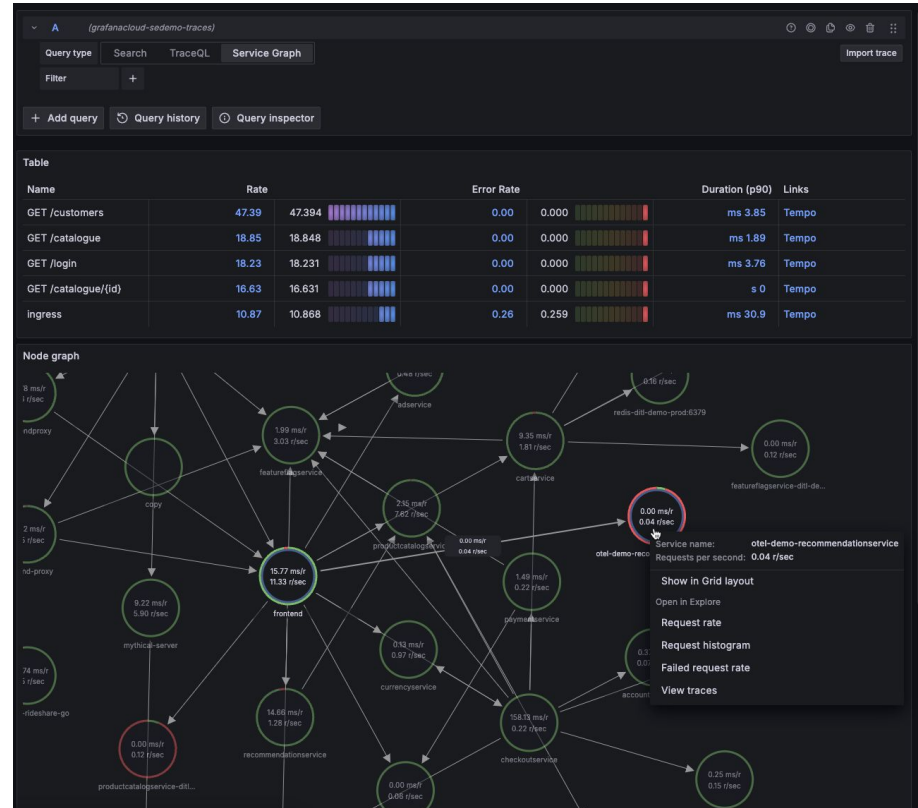
- Jaeger
- Zipkin
- Opentracing
- **Opentelemetry**

Architektur



Metriken, Traces und Logs verbinden

- Trace to Metrics
- Metrics from traces
- Traceld in Logs finden



Visualisierung



Grafana

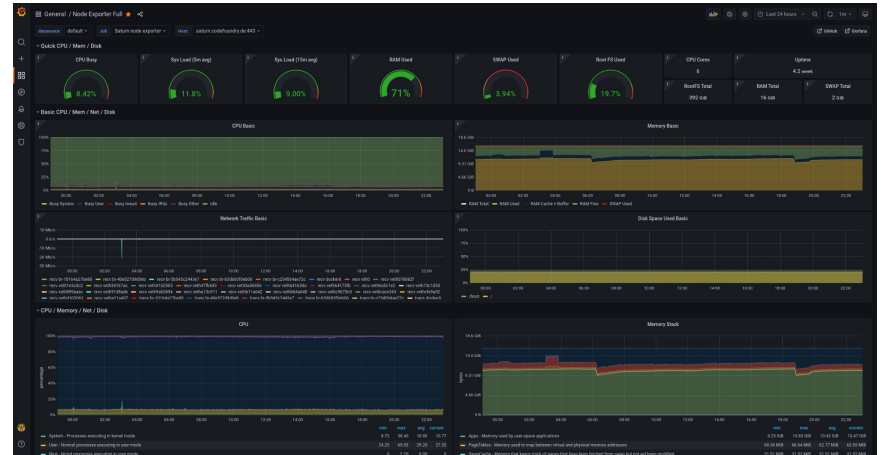
Visualisierung für viele verschiedene Datenquellen:

<https://grafana.com/docs/grafana/latest/datasources/#built-in-core-data-sources>

- Prometheus
- Loki
- Tempo
- InfluxDB
- ...

Grafana - Dashboards

Daten aller Art werden in Dashboards visualisiert



Grafana - Dashboards

Keine Angst: Man muss Dashboards nicht mühsam selbst zusammenklicken.

<https://grafana.com/grafana/dashboards/>



Mehr aus dem
Grafana-Universum

Lasttests: k6

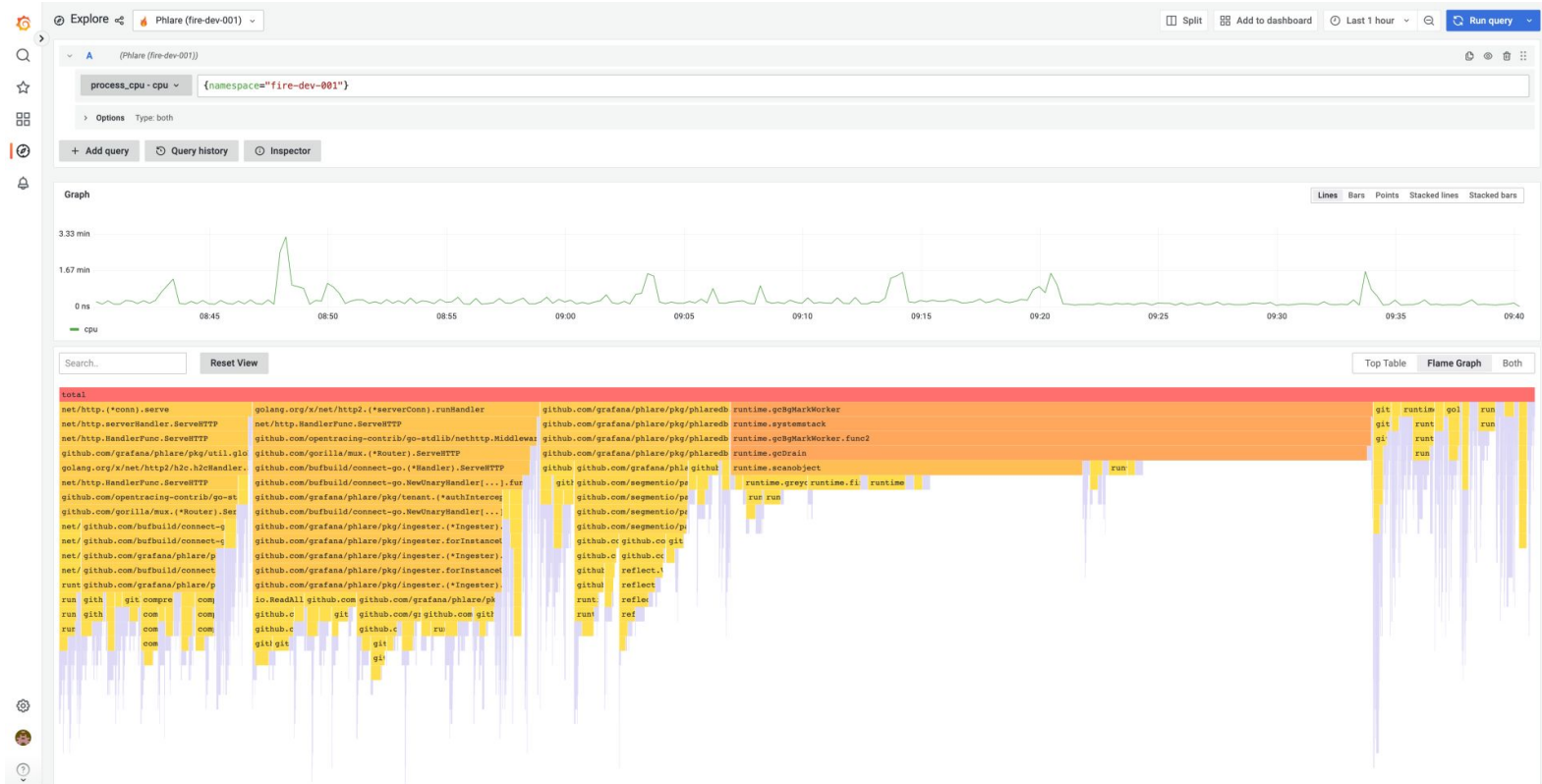
Lasttests? k6!

Gebaut mit Go - die Tests werden in JavaScript geschrieben.

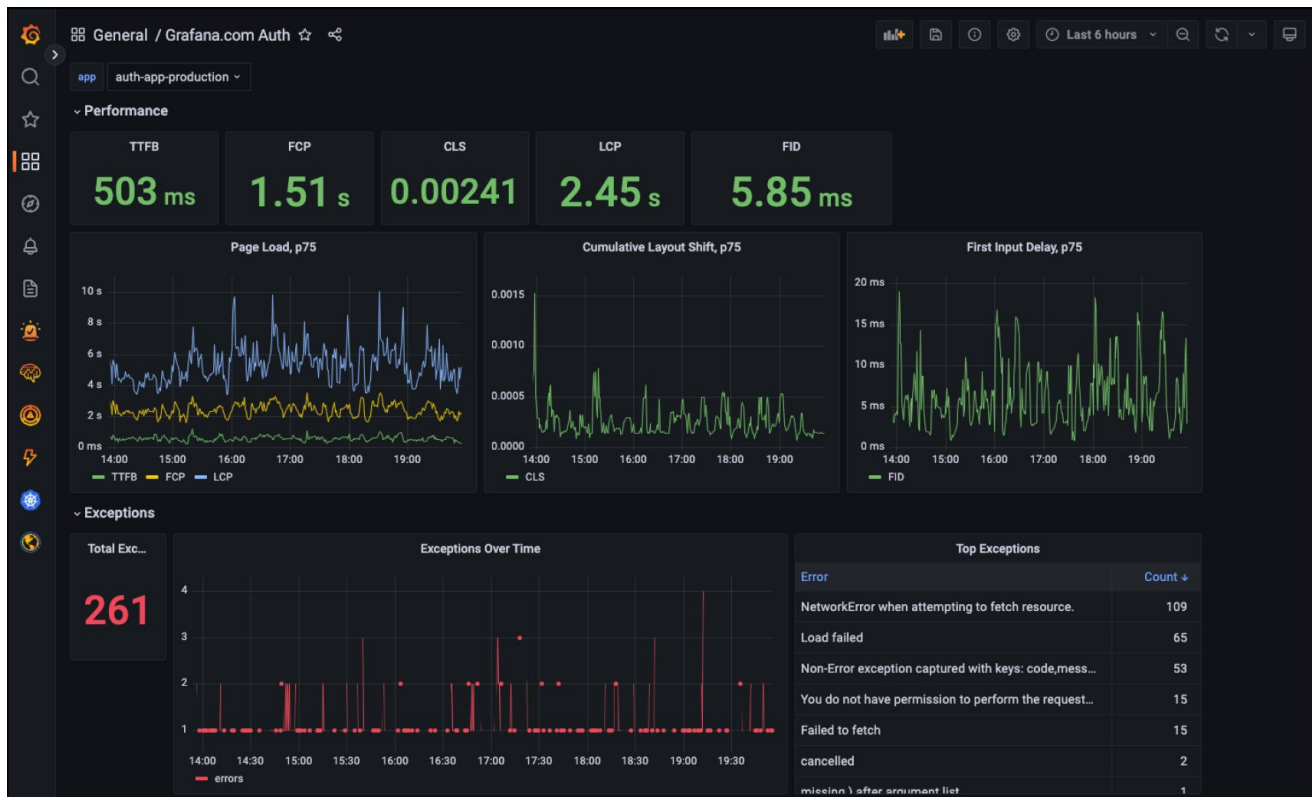
Testergebnisse landen in einer InfluxDB - visualisiert wird mit Grafana.



Profiling: Pyroscope



Frontend Observability: Faro



Andere Observability Stacks und Plattformen

- Properitär
 - Splunk: https://www.splunk.com/de_de/products/observability-cloud.html
 - Datadog: <https://www.datadoghq.com/>
 - NewRelic: <https://newrelic.com/de>
 - Cloud Provider Toolset (Cloudwatch, X-Ray, Azure Monitor, Google Cloud's observability suite)
- Properitär/OS
 - Elastic: <https://www.elastic.co/elastic-stack>

Further Reading

Blog: <https://blog.qaware.de/posts/cloud-observability-grafana-spring-boot/>

Grafana @ Heise Mastering Kubernetes:

<https://de.slideshare.net/QAware/cloud-observability-mit-loki-prometheus-tempo-und-grafana>

Grafana: <https://grafana.com/>

Prometheus: <https://prometheus.io/>

Loki: <https://grafana.com/oss/loki/>

Tempo: <https://grafana.com/oss/tempo/>