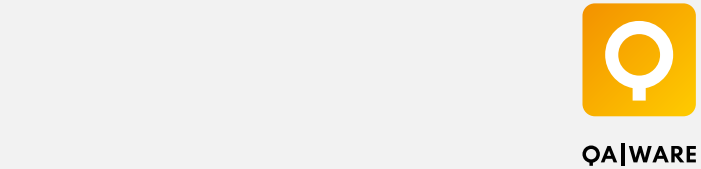
qaware.de

QA|WARE
SOFTWARE ENGINEERING

Observability

Franz Wimmer
franz.wimmer@qaware.de

What exactly is observability?



How well do I know the internal state of my system when I look at its outputs? (see [Kal])

What does such a modern system output?

- A website?
- A REST API?
- E-mails to the admin?
- A sparsely maintained log file?

What if the system consists of many microservices?

What exactly is observability?

"Observability" today typically refers to three topics:

- Logs
- Metrics
- Traces

All together, this enables insight into distributed microservices.

The goal of observability: Diagnosability. What do I understand by that?

*"A **structured approach** to equipping an application with **sensors** in advance so that I can **quickly identify the error** in the event of a **fault** and have the information necessary to repair it."*

A system can be easily diagnosed if you can easily recognize and correct healthy and unhealthy conditions.

A diagnosable system has a

- short Mean Time to Detect (MTTD)
- short Mean Time to Repair (MTTR)

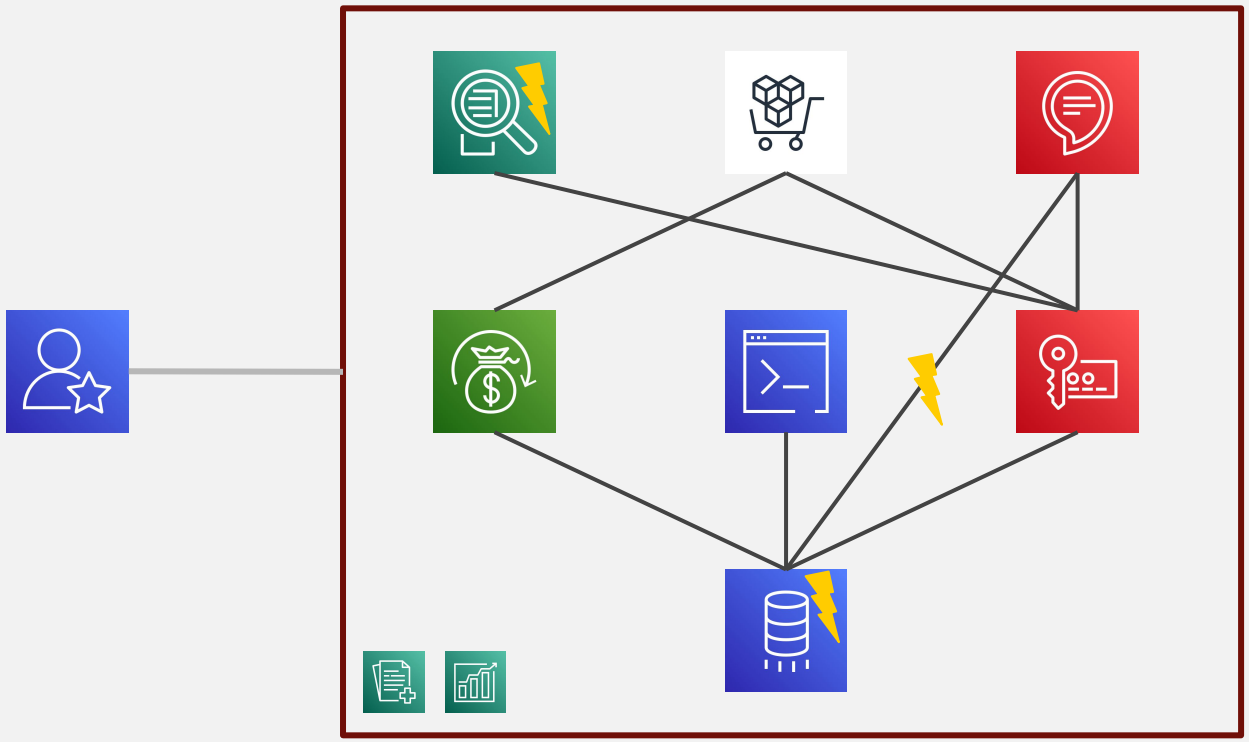
and thus to a short period of time in which errors remain undetected and even exist.

Diagnosability: A structured approach is necessary to avoid influencing the system too much.

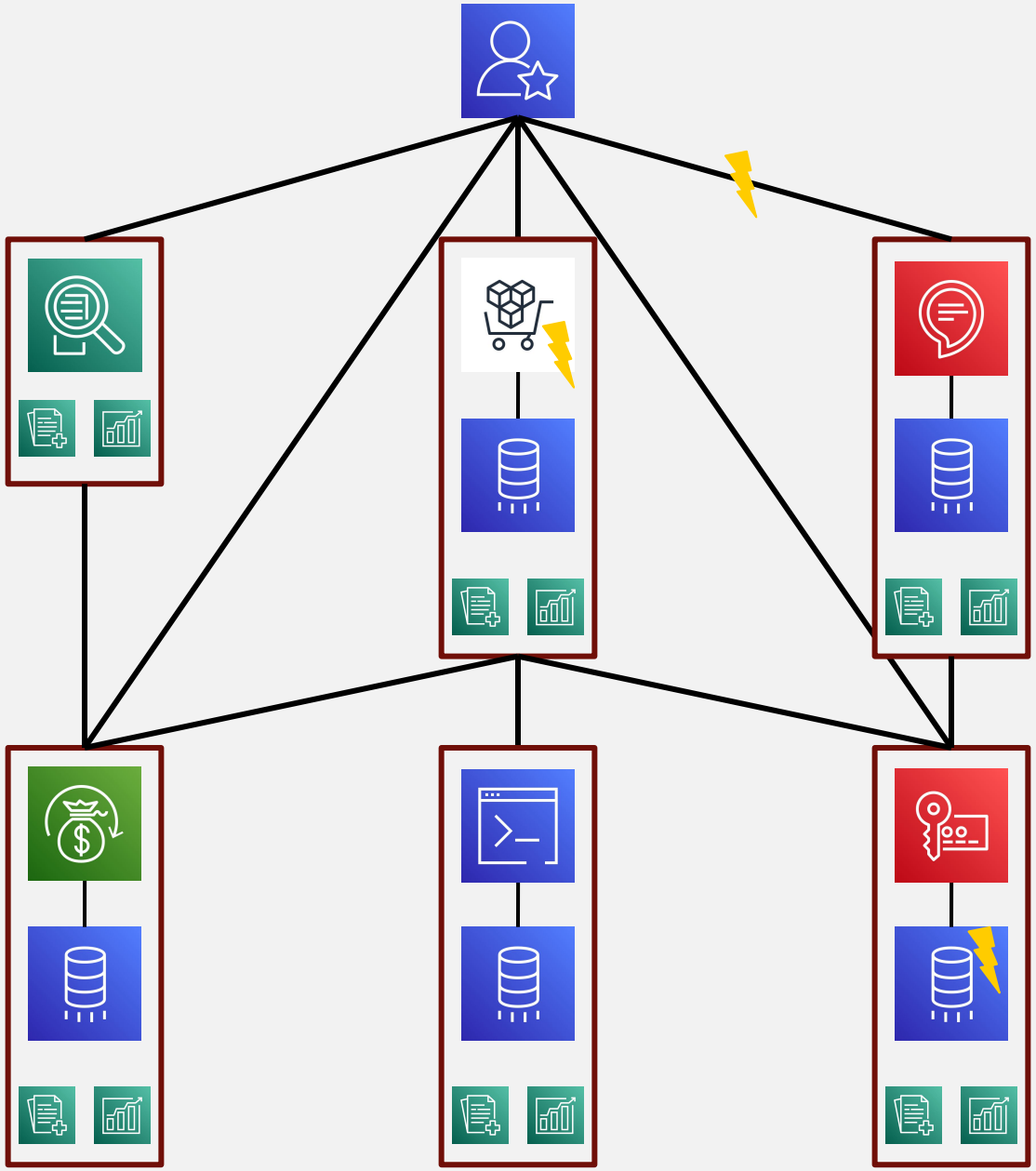
1. **Get an overview:**
   a. What are the central components, e.g. login, etc.?
   b. What are the supporting components, e.g. batch and loader jobs, etc.?
2. **Identify error limits:**
   a. Internal error limits: layers/use cases
   b. External error limits: incoming and outgoing calls
3. **Define error classes:**
   a. Severity levels: operation still possible, no impact on customer, etc.
   b. Impact: certain functions are not available to customers, etc.
4. **Determine the runtime data that is necessary for detection:**
   a. Uniformity: data has the same meaning; uniform data formats are defined, etc.
   b. Clearly defined: it is clear what the metric means, e.g. CPU load
5. **Define actions:**
   a. Alerting: who is notified and when
   b. Create playbooks for errors: How do I get the data, etc.

Why (Cloud) Observability?

Back then
The monolith



Today
Microservices



QA|WARE

The (Open Source) Stack

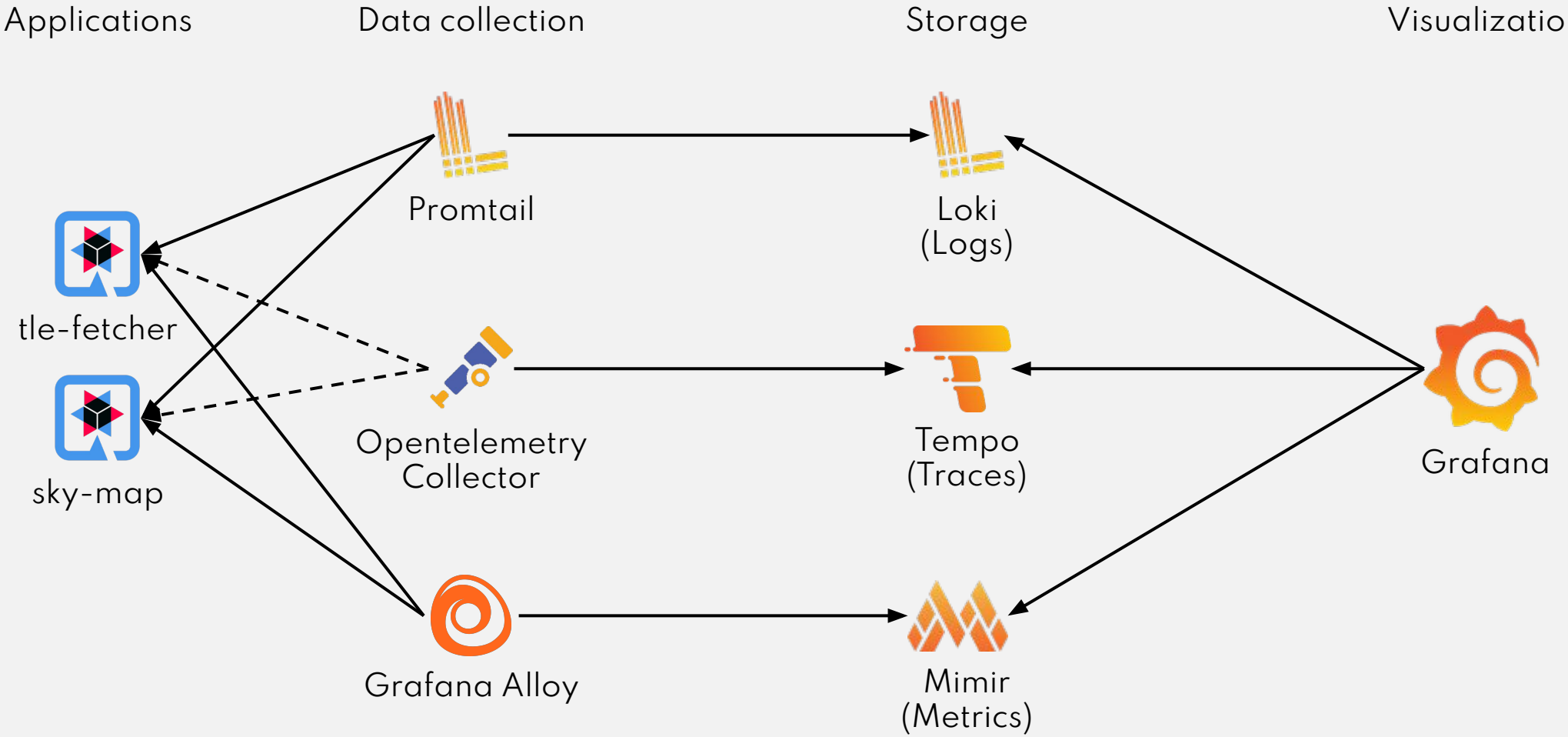
Today we use the LGTM stack.:

- Loki (Logs)
- Grafana (Visualization)
- Tempo (Traces)
- Mimir (Metrics)

https://grafana.com/oss/

The (Open Source) Stack

QA|WARE



Logs

Logs

- **Logs** are an important part of any IT system.
- In the cloud, we need to look at logs from many different services at the same time.
- Every system has logs, but they are always in a different **format.**

- Example: **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]`

- Example: **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. Please be structured. Please be well-considered. /1

- Define a log format and ensure that all services use the same format.
- Define a diagnostic context = information that helps in the event of an error, e.g.
  - TraceId
  - UserId
  - SessionId
- Use structured logging (e.g. JSON)
  - Simplifies handling in the analysis
- Use asynchronous logging (provided that the recipients support it) to prevent blockages
  - If all of this is sent over TCP etc. and then filtered and sorted later, the order doesn't matter 😎

Logging. Please be structured. Please be well-considered. /2

- Don't log too much, but log every
  - exception,
  - every error,
  - every meaningful piece of information …
  - … but not multiple times.
- Use log levels. To do this, define which category should have which level, e.g.:
  - WARN: Errors that affect a single request but not the stability of the service.
  - ERROR: Errors that affect the stability of the service.

Loki

Loki is the log aggregator in the Grafana ecosystem.

Passive log storage – logs have to be obtained differently.

Promtail is installed locally if possible and pushes the logs to **Loki.**

Since: 2018

License: AGPL 3.0

Architecture

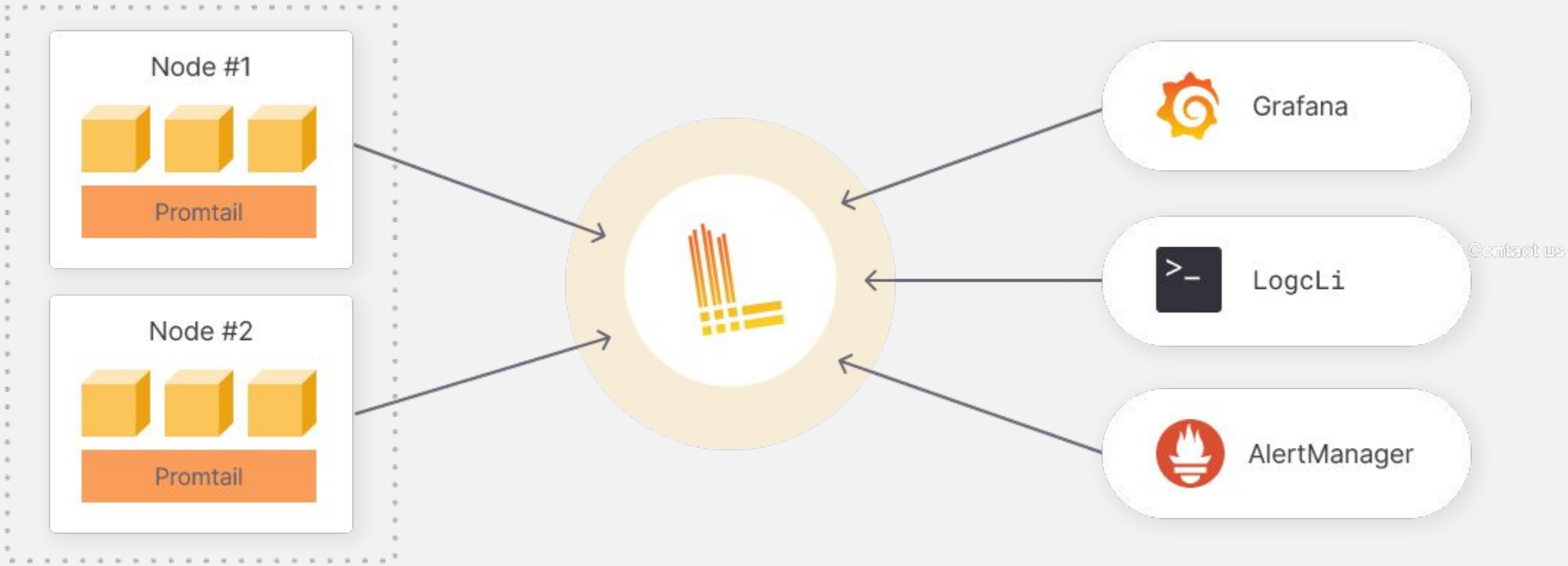


Image: https://grafana.com/oss/loki/

Promtail configuration


```
clients:
  - url: "https://loki.qaware.de/loki/api/v1/push"
    basic_auth:
      username: loki
      password: ****************

scrape_configs:
  - job_name: journal
    journal:
      json: true
      max_age: 12h
      labels:
        host: saturn
        job: systemd-journal
    relabel_configs:
      - source_labels: ['__journal__systemd_unit']
        target_label: 'unit'
```

Caution: don't use labels too dynamically!
https://grafana.com/blog/2020/04/21/how-labels-in-loki-can-make-log-queries-faster-and-easier/

LogQL

Loki has its own query language.

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

```
{host="$host", job="systemd-journal"}
  | json
  | line_format "{{ .unit }}: {{ .MESSAGE }}"
  |= "$search"
```

Metrics

Metrics: Basics

Metrics are measurements that reflect the current state of the system.

Examples:

- **CPU** load
- Size of the **JVM Heap**
- Number of **calls to an interface**

The metrics are provided by the system to be monitored.

Metrics can also have metadata.

Our metrics storage: Grafana Mimir

- Mimir (2022) stores **metrics**
- It contains a lot of code from the older tool Prometheus (2012)
- Technical foundation:
  - Time series database for numerical time series
  - Collectors for metrics and text-based format for metrics
  - Query language (PromQL) for time series
  - Alerting based on time series

Metrics: Basics

Grafana Alloy pulls metrics from all configured targets every **15 seconds**.

The transport is usually done via HTTP:
GET /metrics

No aggregation is done here yet. The consumer is responsible for that.

```
# 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

Metrics have different types to visualize data:

- **Counter**

  A number that can either be incremented or reset.
- **Gauge**

  A metric that can change in either direction.
- **Histogram**

  Values in “buckets” - e.g. the duration of requests.
- **Summary**

  Similar to histogram, summarizes values.

PromQL

PromQL is a powerful query language for filtering/aggregating the desired metrics.

The entire time series:

```
http_requests_total
```

Filter by label:

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

Calculate rate:

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

Mimir: Architecture (Write)

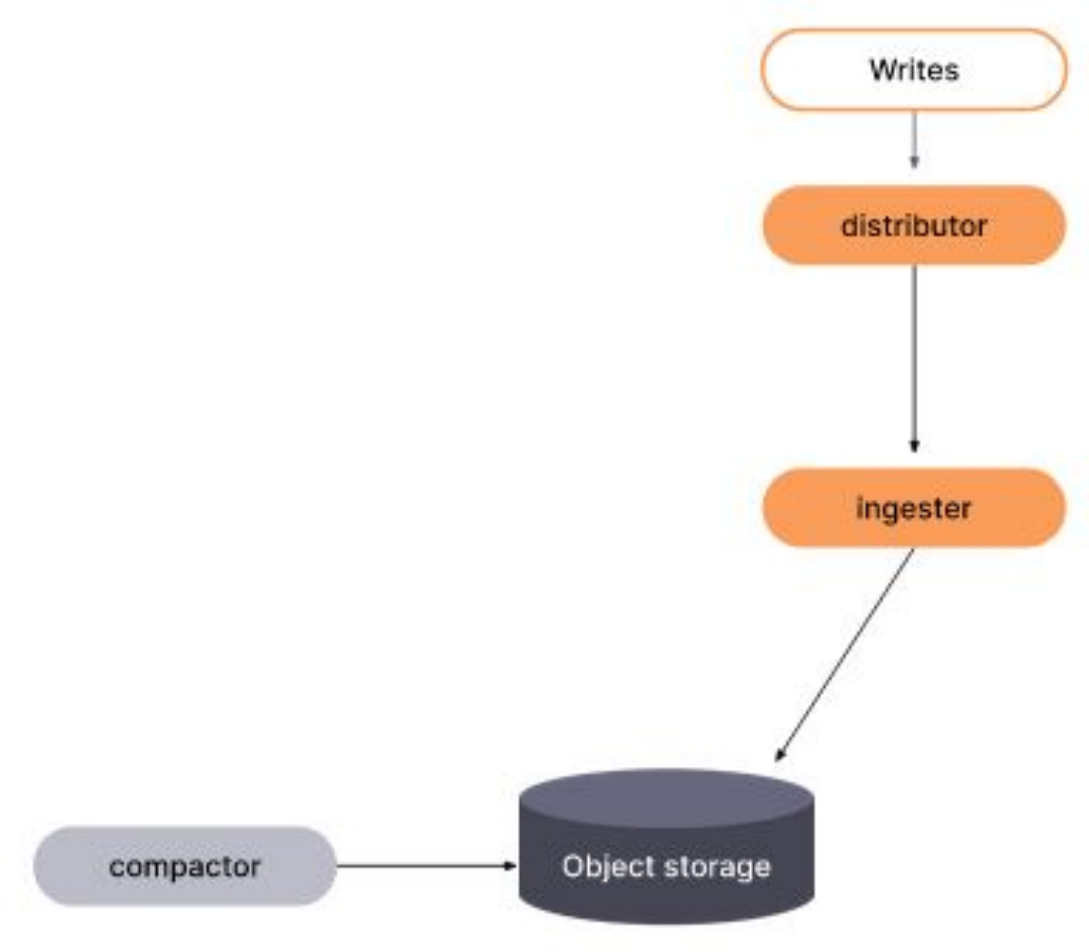


Bild: https://prometheus.io/docs/introduction/overview/

Mimir: Architecture (Read)

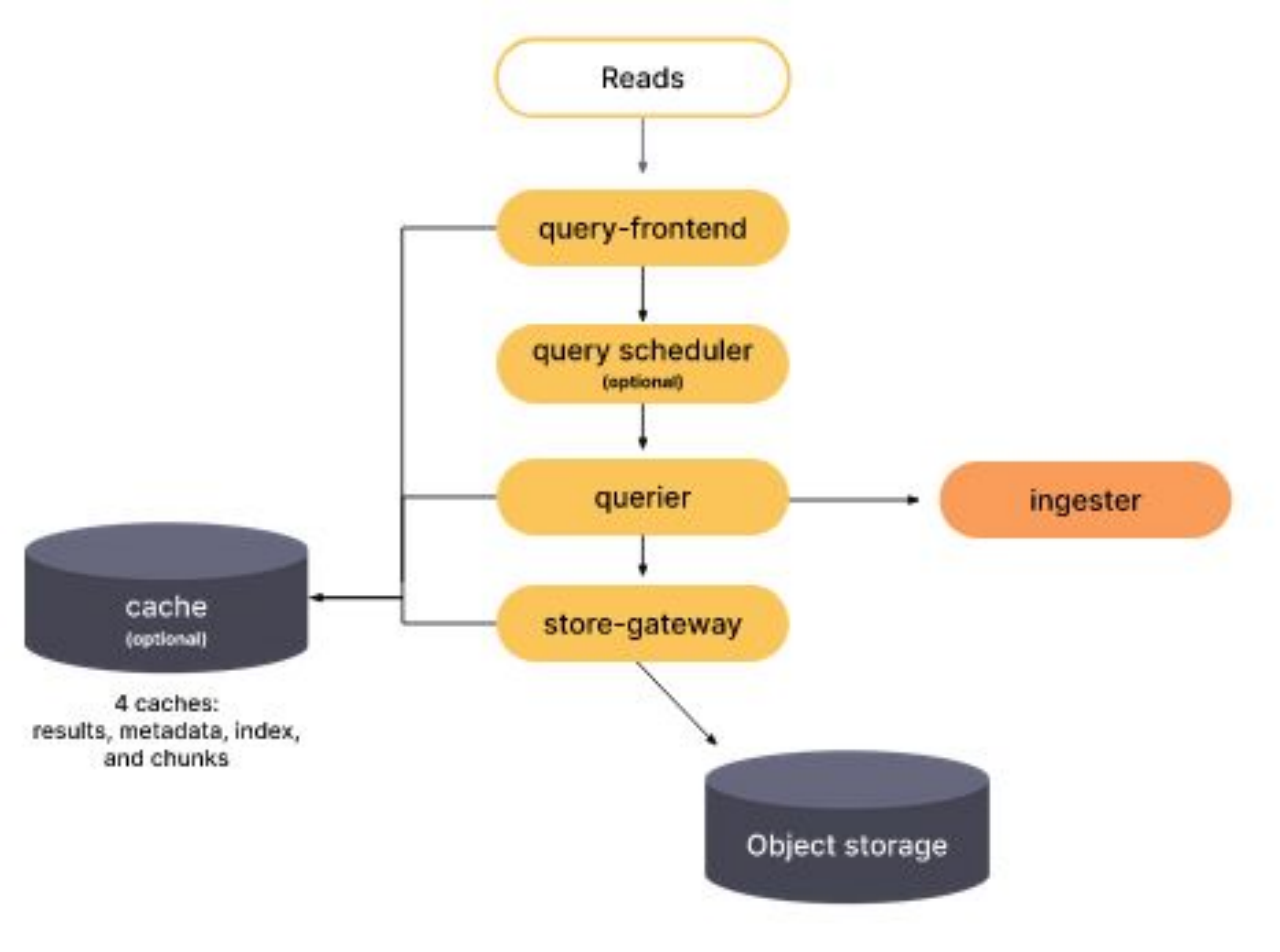


Bild: https://prometheus.io/docs/introduction/overview/

Configuration (application)
Example: Quarkus & Micrometer


```
# Gradle:

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


# Maven:

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

Grafana Alloy

- Scrapes metrics…
- …and forwards them to Mimir.

- Easy to use (binary + config file)
- Good integration into Kubernetes, e.g. using the service discovery
- Configuration in flow mode

Configuration in flow mode (1)


```
// Discover Kubernetes pods to collect metrics from.
discovery.kubernetes "pods" {
  role = "pod"
}

// Scrape metrics from Kubernetes pods and send to a prometheus.remote_write
// component.
prometheus.scrape "default" {
  targets = discovery.kubernetes.pods.targets
  forward_to = [prometheus.remote_write.default.receiver]
}
```

Configuration in flow mode (2)

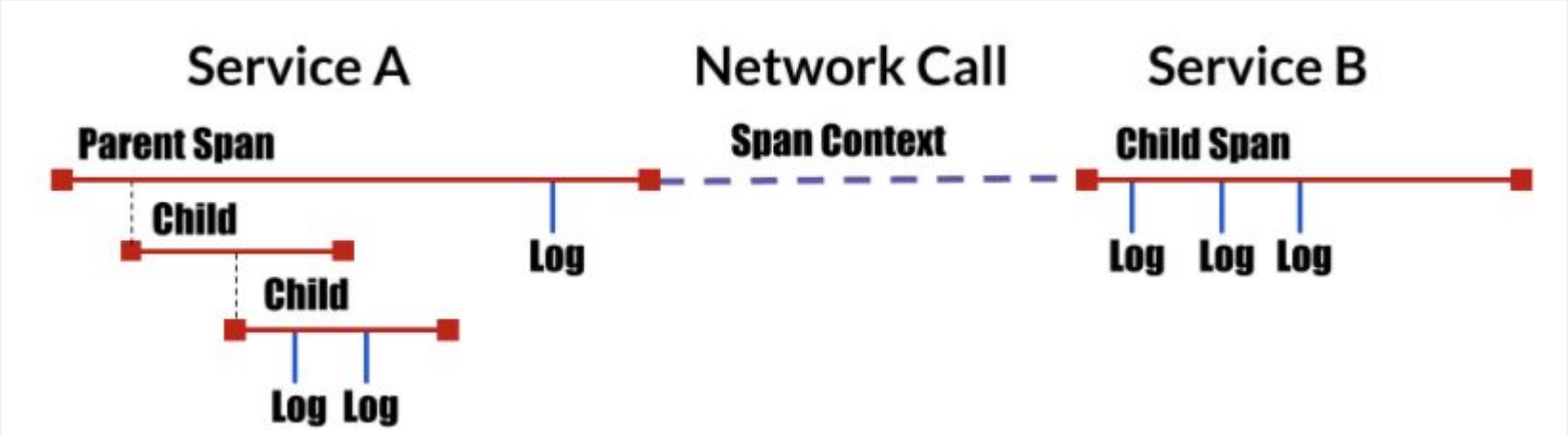

```
// Get an API key from disk.
local.file "apikey" {
  filename  = "/var/data/my-api-key.txt"
  is_secret = true
}

// Collect and send metrics to a Prometheus remote_write endpoint.
prometheus.remote_write "default" {
  endpoint {
     url = "http://localhost:9009/api/prom/push"

     basic_auth {
     username = "MY_USERNAME"
     password = local.file.apikey.content
     }
  }
}
```

Traces

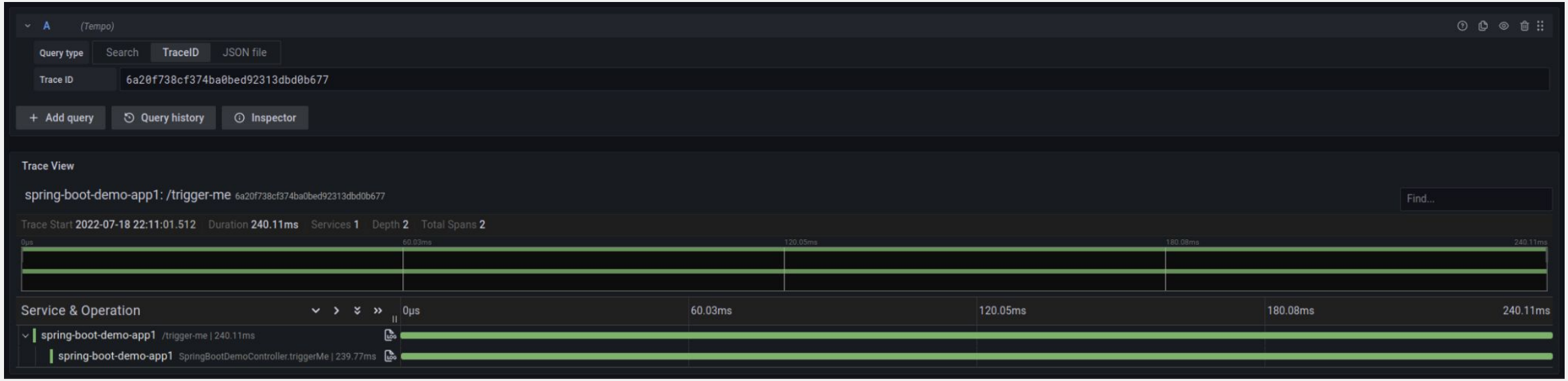
Distributed Tracing

- Distributed Tracing: Technology for tracing calls and processes in distributed software systems.
- Today's considerations go back to the paper: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure
- Idea: Each service forwards information from the caller to the next service and back again. This results in a directed graph.
  - Each service must be instrumented. Otherwise there will be a gap.
  - The smallest unit is called a span. A span has a duration and descriptive information. A trace is a set of 1..n spans.
- Many implementations exist for different programming languages and technologies.
  - https://opentelemetry.io/docs/languages/
  - https://opentelemetry.io/docs/zero-code/

Distributed Tracing

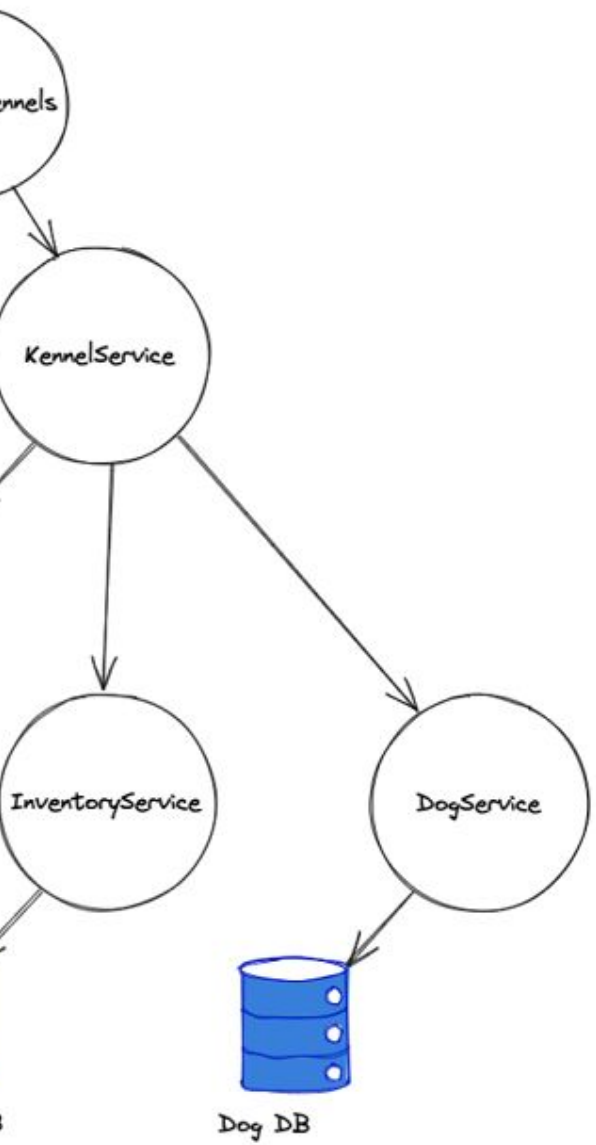


Traces: Foundation



Traces visualize which paths a request has taken through the microservices……

… and where an error may have occurred!



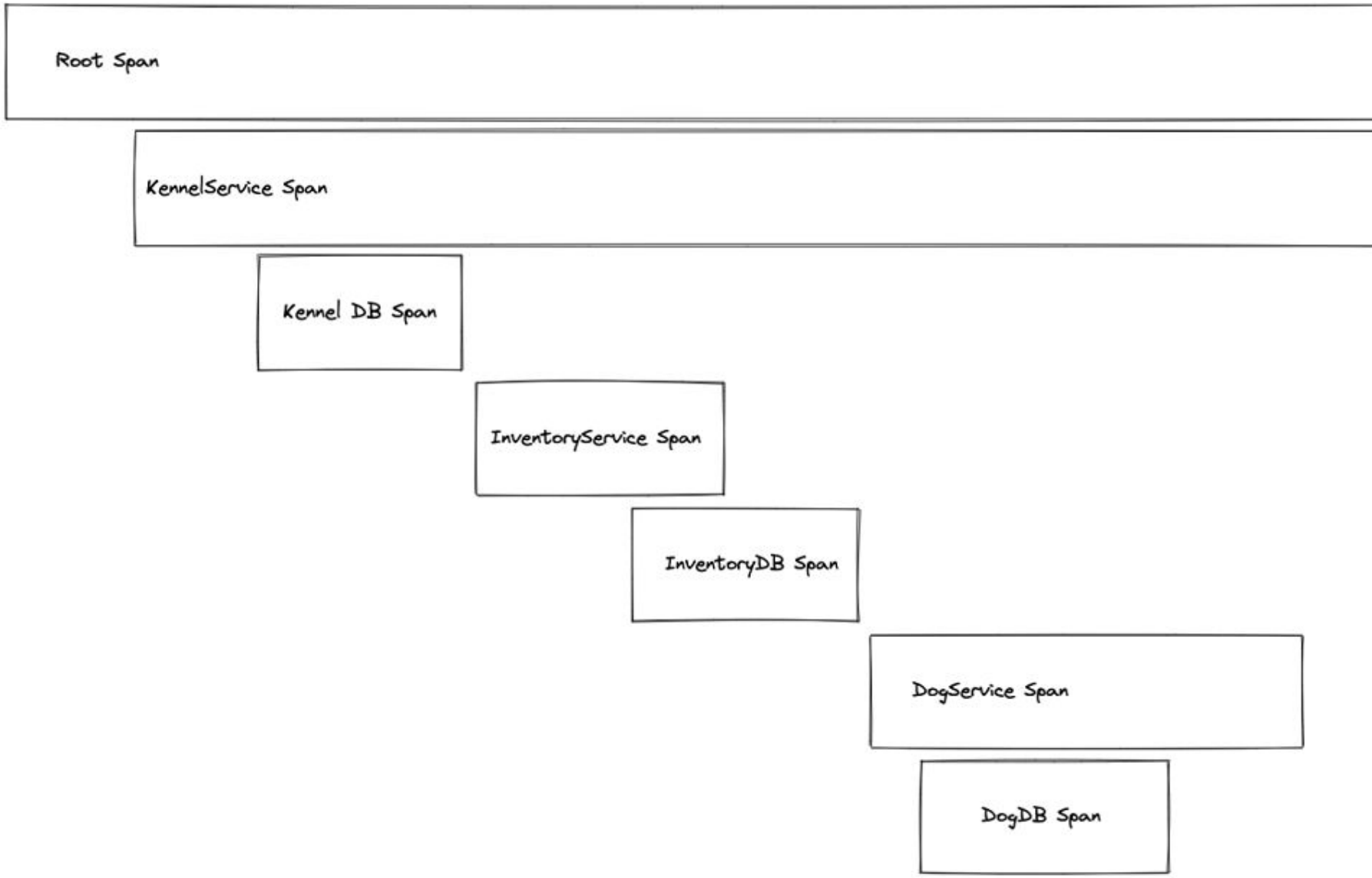


Bild: https://grafana.com/blog/2021/09/23/intro-to-distributed-tracing-with-tempo-opentelemetry-and-grafana-cloud/

Grafana Tempo

Traces in the Grafana ecosystem!

SDKs for various platforms

Compatible with various tracing agents:

- Jaeger
- Zipkin
- Opentelemetry

Since: 2020

License: AGPL 3.0

Architecture

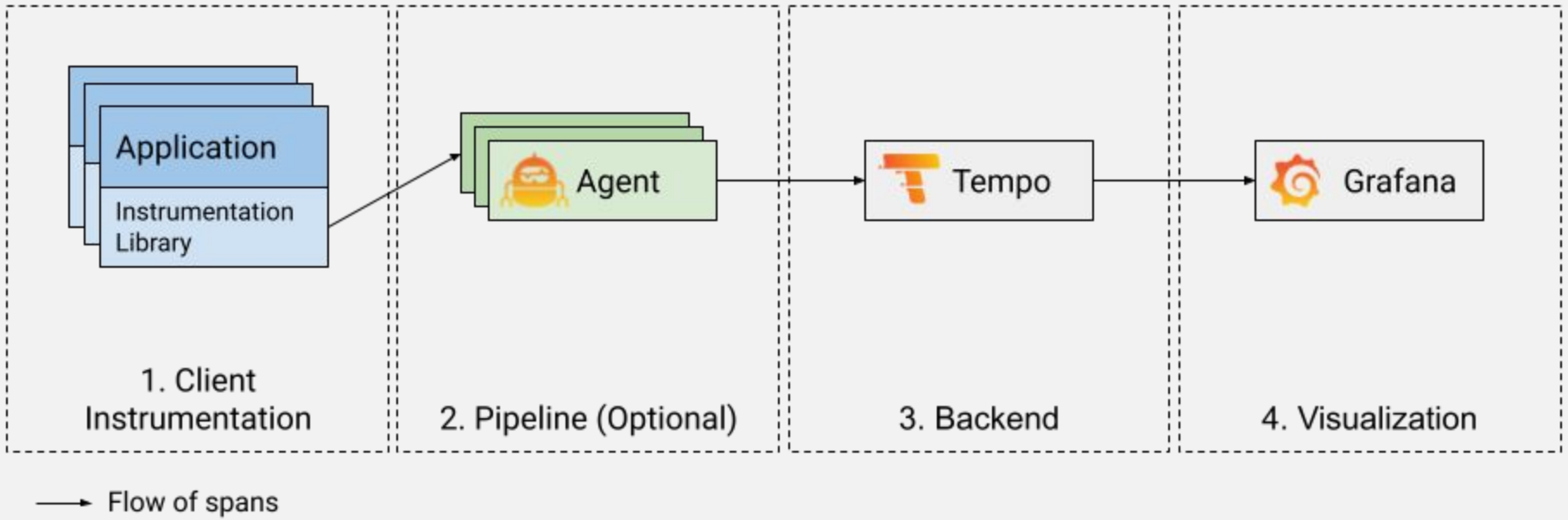


Bild: https://grafana.com/docs/tempo/latest/getting-started/

Combining metrics and traces? Exemplars!

QA|WARE

Supported in Micrometer 1.9, for example

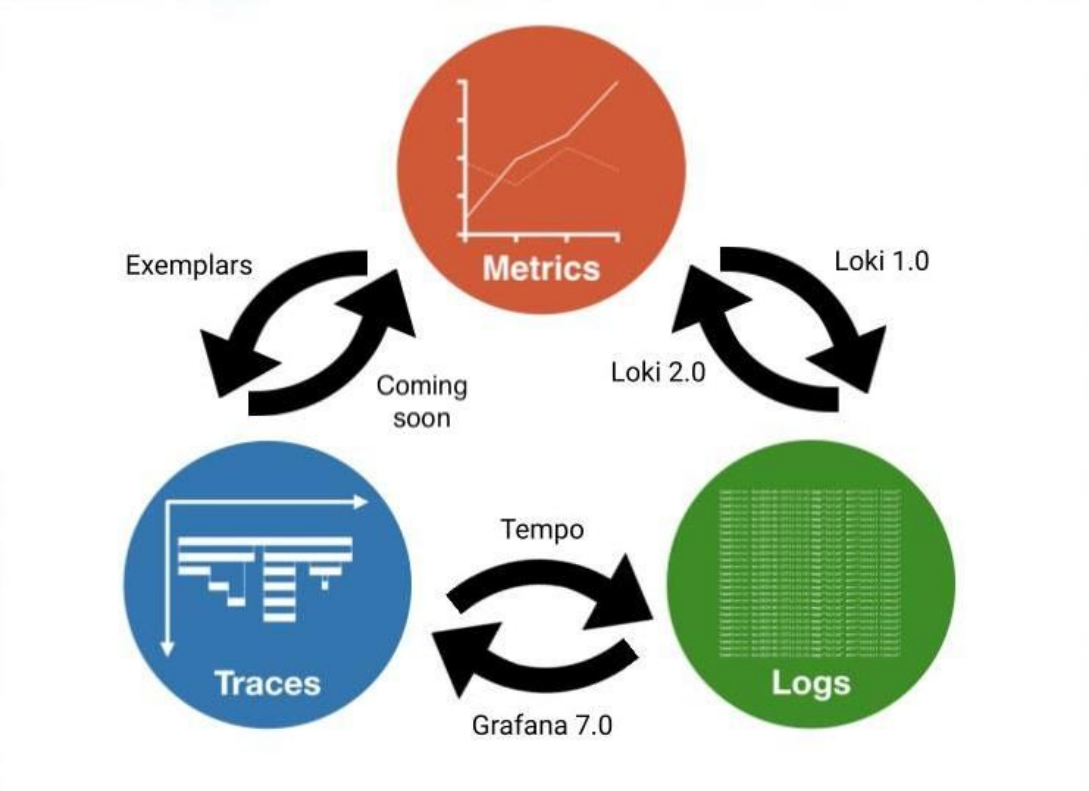


Image:
https://grafana.com/blog/2021/03/31/intro-to-exemplars-which-enable-grafana-tempos-distributed-tracing-at-massive-scale/

Visualization

Grafana

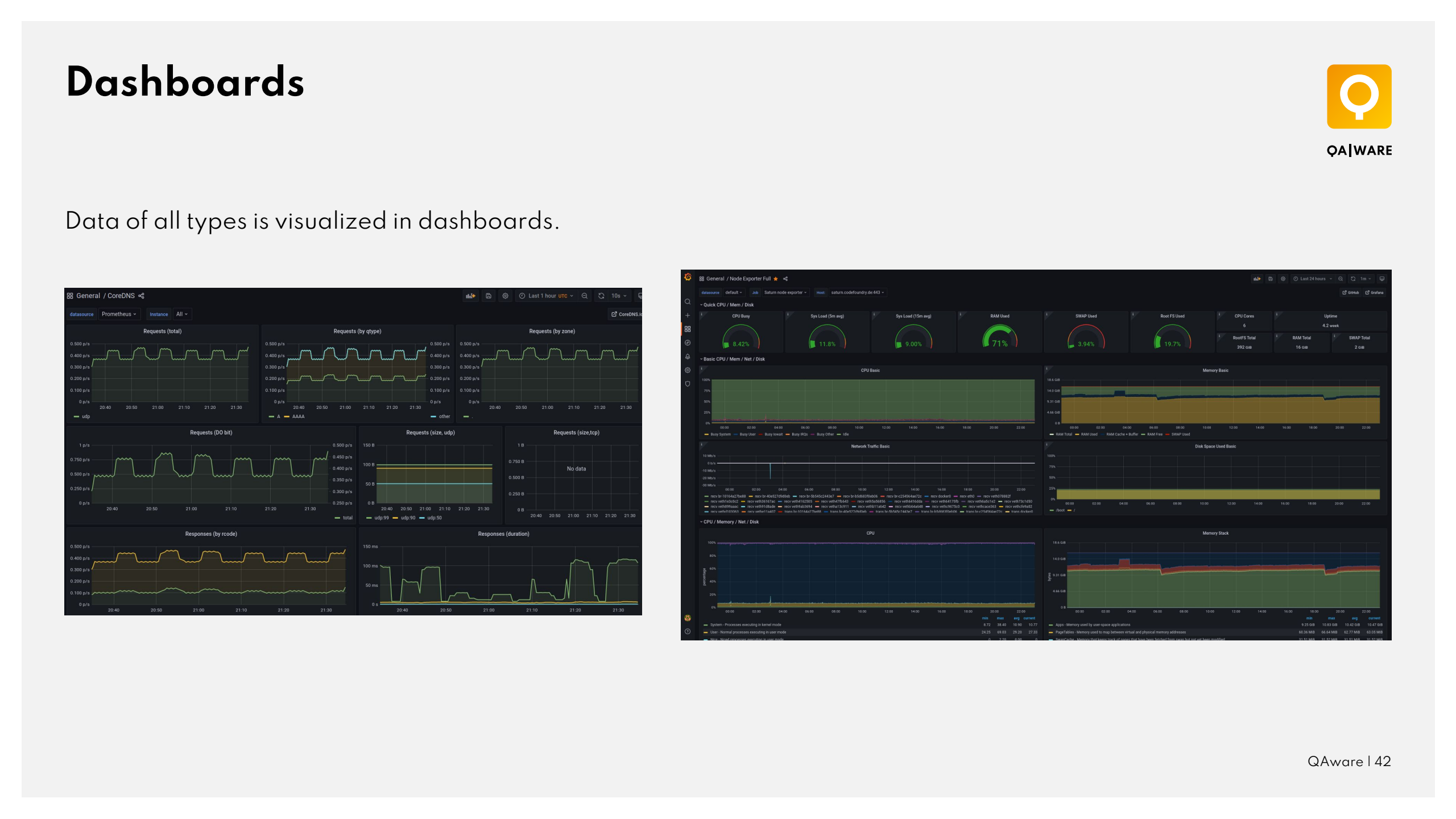
Visualization for many different data sources:

- Mimir
- Loki
- Tempo
- InfluxDB
- …

Since: 2014

License: AGPL 3.0

Dashboards



Data of all types is visualized in dashboards.

Grafana - Dashboards

Don't worry: you don't have to laboriously click together dashboards yourself.

https://grafana.com/grafana/dashboards/

Showcase

More from the Grafana universe

Load tests: k6

Built with Go – the tests are written in JavaScript.

Test results end up in an InfluxDB – visualization is done with Grafana.

https://k6.io/

Frontend observability: Faro

https://grafana.com/oss/faro/

Further reading

Blog: https://blog.qaware.de/posts/cloud-observability-grafana-spring-boot/
Showcase: https://github.com/zalintyre/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/
Mimir: https://grafana.com/docs/mimir/latest/
Loki: https://grafana.com/oss/loki/
Tempo: https://grafana.com/oss/tempo/

Sources

[Kal] R.E. Kalman,
On the general theory of control systems,
IFAC Proceedings Volumes,
Volume 1, Issue 1,
1960,
Pages 491-502,
ISSN 1474-6670,
https://doi.org/10.1016/S1474-6670(17)70094-8.
(https://www.sciencedirect.com/science/article/pii/S1474667017700948)

qaware.de

QA|WARE
SOFTWARE ENGINEERING

QAware GmbH
Aschauer Straße 32
81549 München
Tel. +49 89 232315-0
info@qaware.de


twitter.com/qaware


linkedin.com/company/qaware-gmbh


xing.com/companies/qawaregmbh


slideshare.net/qaware


github.com/qaware