The Pillars of Observability

Cheat sheets, Practice Exams and Flash cards 👉 www.exampro.co/kcna

What is Observability?
The ability to measure and understand how internal systems work in order to to answer questions regarding performance, tolerance, security and faults with a system / application.

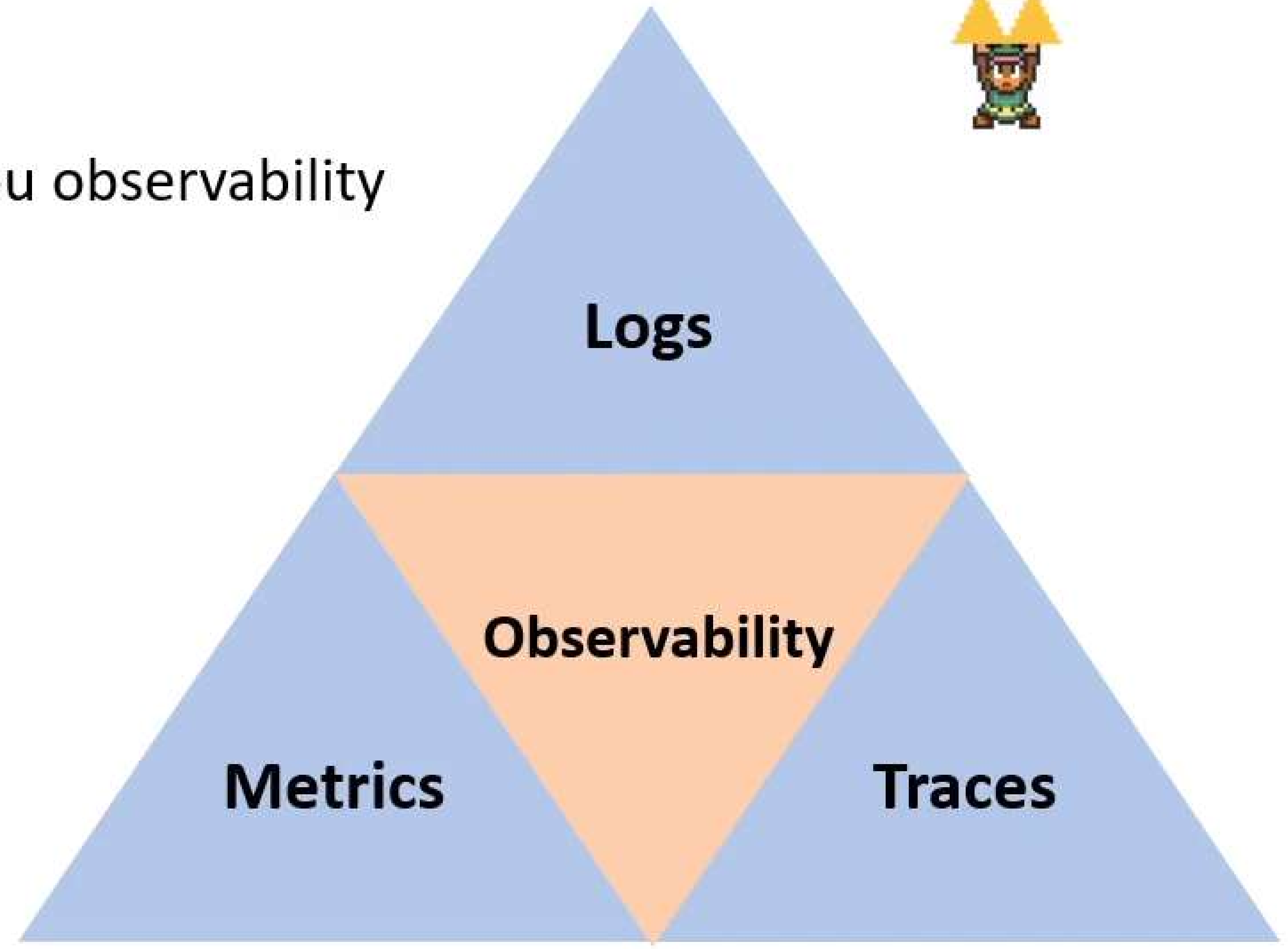
To obtain observability you need to use **Metrics**, **Logs** and **Traces**.
You have to use them together, using them in isolate does not gain you observability

Metrics
A number that is measured over period of time
eg. If we measured the CPU usage and aggerated it over an a period of time we could have an **Average CPU metric**

Logs
A text file where each line contains event data about what happened at a certain time.

Traces
A history of request that is travels through multiple Apps/services so we can pinpoint performance or failure.

Looks like they should have called it the Triforce of Observability



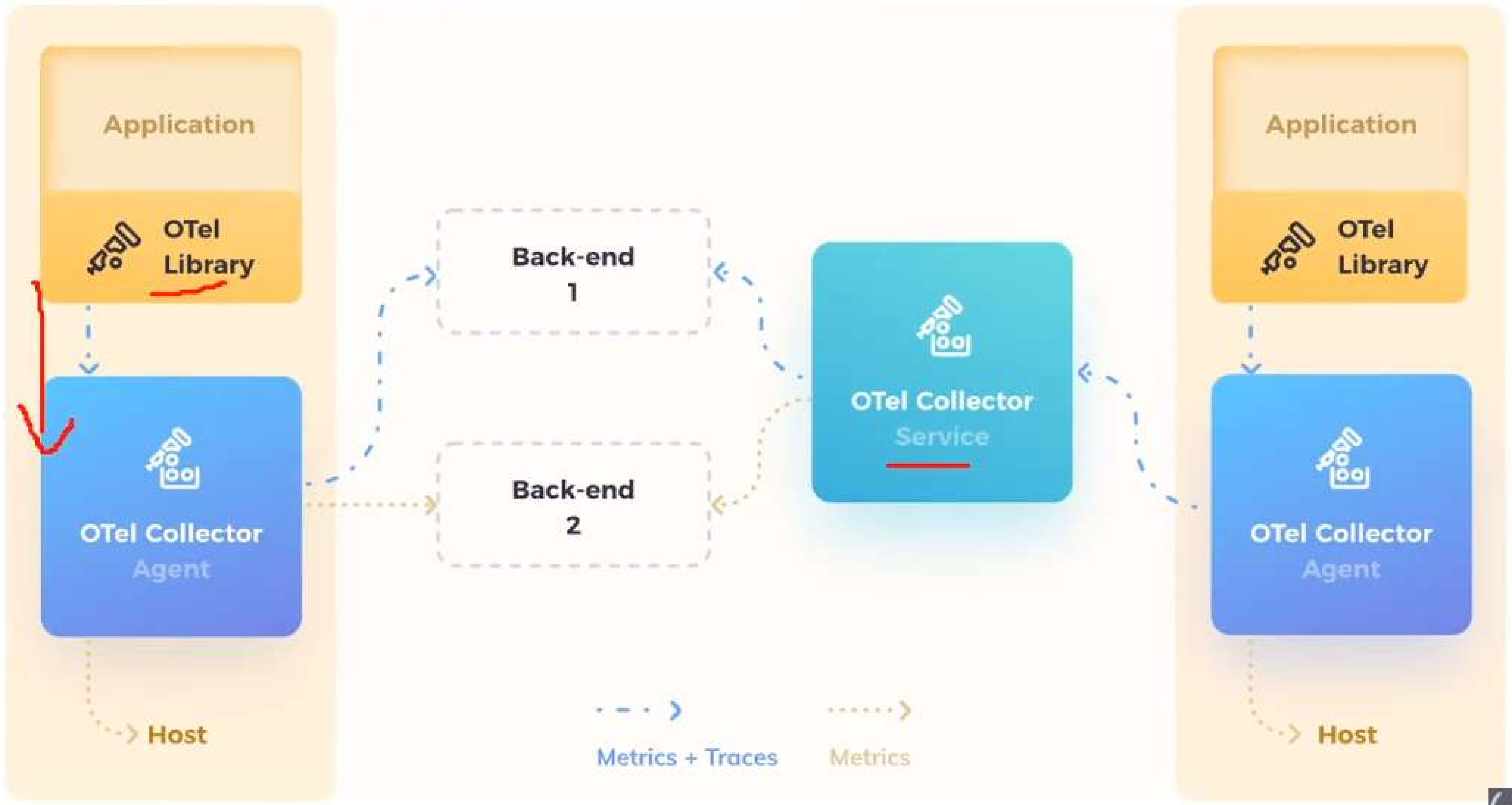
Open Telemetry

OpenTelemetry

Open Telemetry (OTEL) is a collection of open-source tools, APIs and SDKs to instrument, generate, collect, and export telemetry data

Open Telemetry is **standardizes** the way telemetry data (metrics, logs and traces) are generated and collected.

Wire protocol
a wire protocol refers to a way of getting data from point to point. Eg. SOAP, AMQP



Open Telemetry – Instrumentation

Cheat sheets, Practice Exams and Flash cards 👉 www.exampro.co/kcna

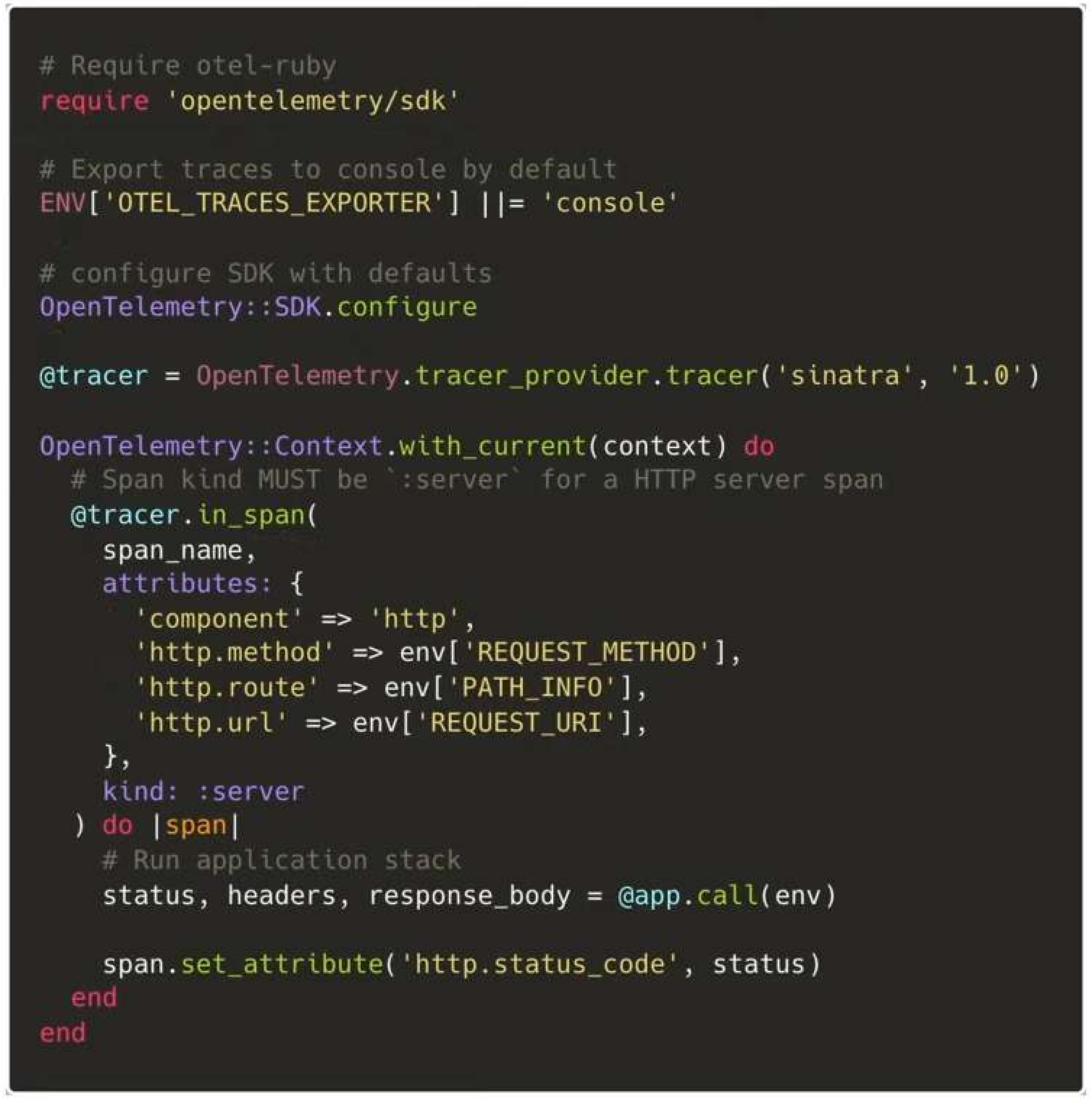
Instrumentation is the act of embedding a monitoring library into your existing application in order to capture monitoring data such as: metrics, traces or logging

Open Telemetry supports a variety of languages:

- C++
- .NET
- Erlang / Elixir
- Go
- Java
- Javascript
- Php
- Python
- Ruby
- Rust
- Swift

For certain frameworks there plug-and-play libraries to quickly instrument your apps:

- Spring
- ASP.NET Core
- Express
- Quarkus



```
# Require otel-ruby
require 'opentelemetry/sdk'

# Export traces to console by default
ENV['OTEL_TRACES_EXPORTER'] ||= 'console'

# configure SDK with defaults
OpenTelemetry::SDK.configure

@tracer = OpenTelemetry.tracer_provider.tracer('sinatra', '1.0')

OpenTelemetry::Context.with_current(context) do
  # Span kind MUST be `:server` for a HTTP server span
  @tracer.in_span(
    span_name,
    attributes: {
      'component' => 'http',
      'http.method' => env['REQUEST_METHOD'],
      'http.route' => env['PATH_INFO'],
      'http.url' => env['REQUEST_URI'],
    },
    kind: :server
  ) do |span|
    # Run application stack
    status, headers, response_body = @app.call(env)

    span.set_attribute('http.status_code', status)
  end
end
```

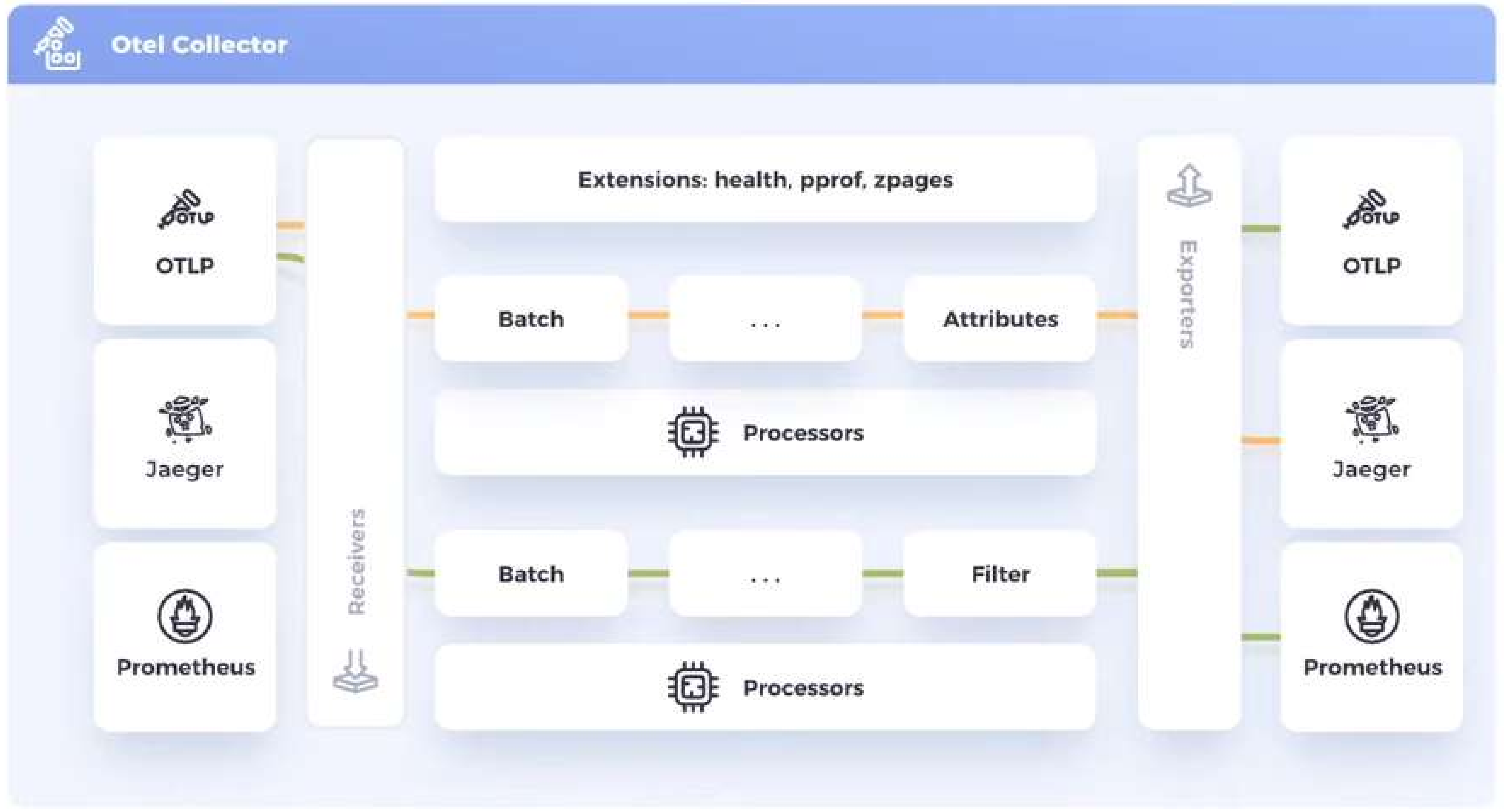
Open Telemetry – Collector

The Open Telemetry collector is an agent installed on the target machine, or as a dedicate server and is Vendor-agnostic way to receive, process and export telemetry data

It removes the need to run, operate, and maintain multiple agents/collectors.

This works with improved scalability and supports open-source observability data formats (e.g. Jaeger, Prometheus, Fluent Bit, etc.) sending to one or more open-source or commercial back-ends.

The local Collector agent is the default location to which instrumentation libraries export their telemetry data.



Prometheus

Prometheus is **an open-source** **systems monitoring and alerting toolkit** originally built at SoundCloud.
Prometheus collects and stores its metrics as time series data.
Prometheus is a **timeseries database**

Prometheus's main features are:

- a multi-dimensional data model with time series data identified by metric name and key/value pairs
- PromQL, a flexible query language to leverage this dimensionality
- no reliance on distributed storage; single server nodes are autonomous
- time series collection happens via a pull model over HTTP
- pushing time series is supported via an intermediary gateway
- targets are discovered via service discovery or static configuration
- multiple modes of graphing and dashboarding support

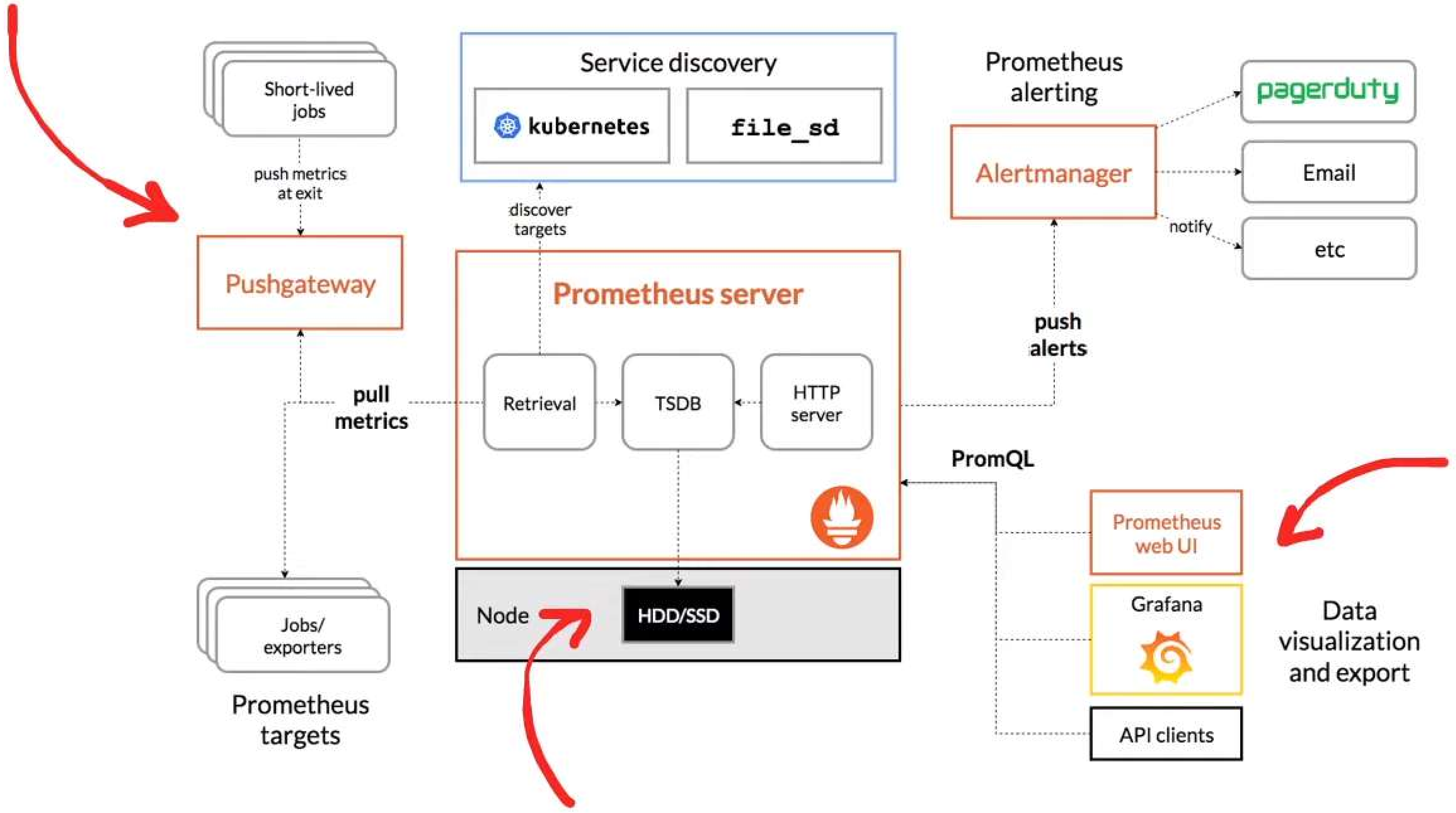
Prometheus values reliability.
You can always view what statistics are available about your system, even under failure conditions.

If you need 100% accuracy, such as for per-request billing, Prometheus is not a good choice as the collected data will likely not be detailed and complete enough.

In such a case you would be best off using some other system to collect and analyze the data for billing, and Prometheus for the rest of your monitoring.

Prometheus

Prometheus scrapes metrics from **instrumented jobs**, either directly or via an intermediary push gateway for short-lived jobs.

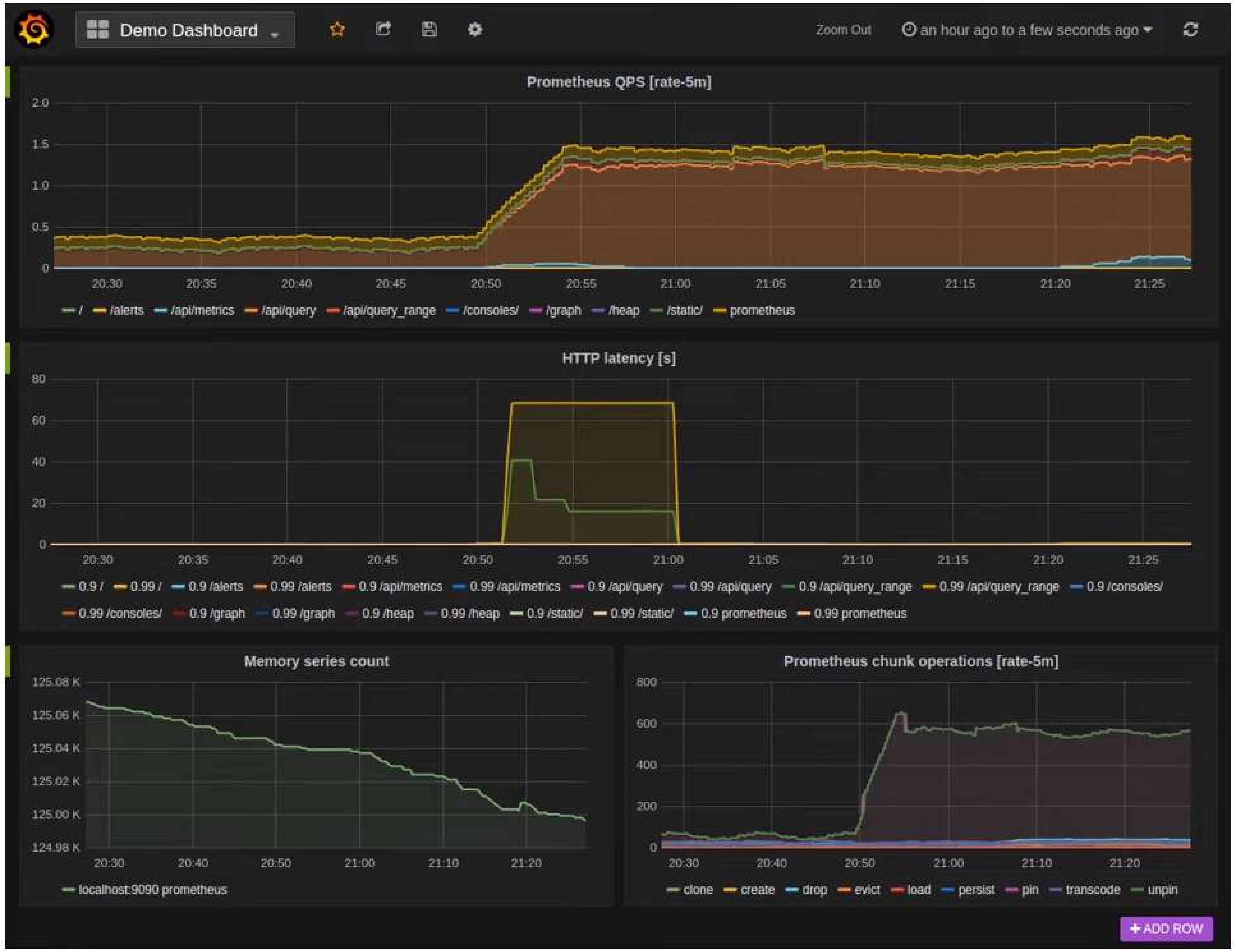


Grafana or other API consumers can be used to visualize

It stores all scraped samples locally and runs rules over this data to either aggregate and record new time series from existing data or generate alerts

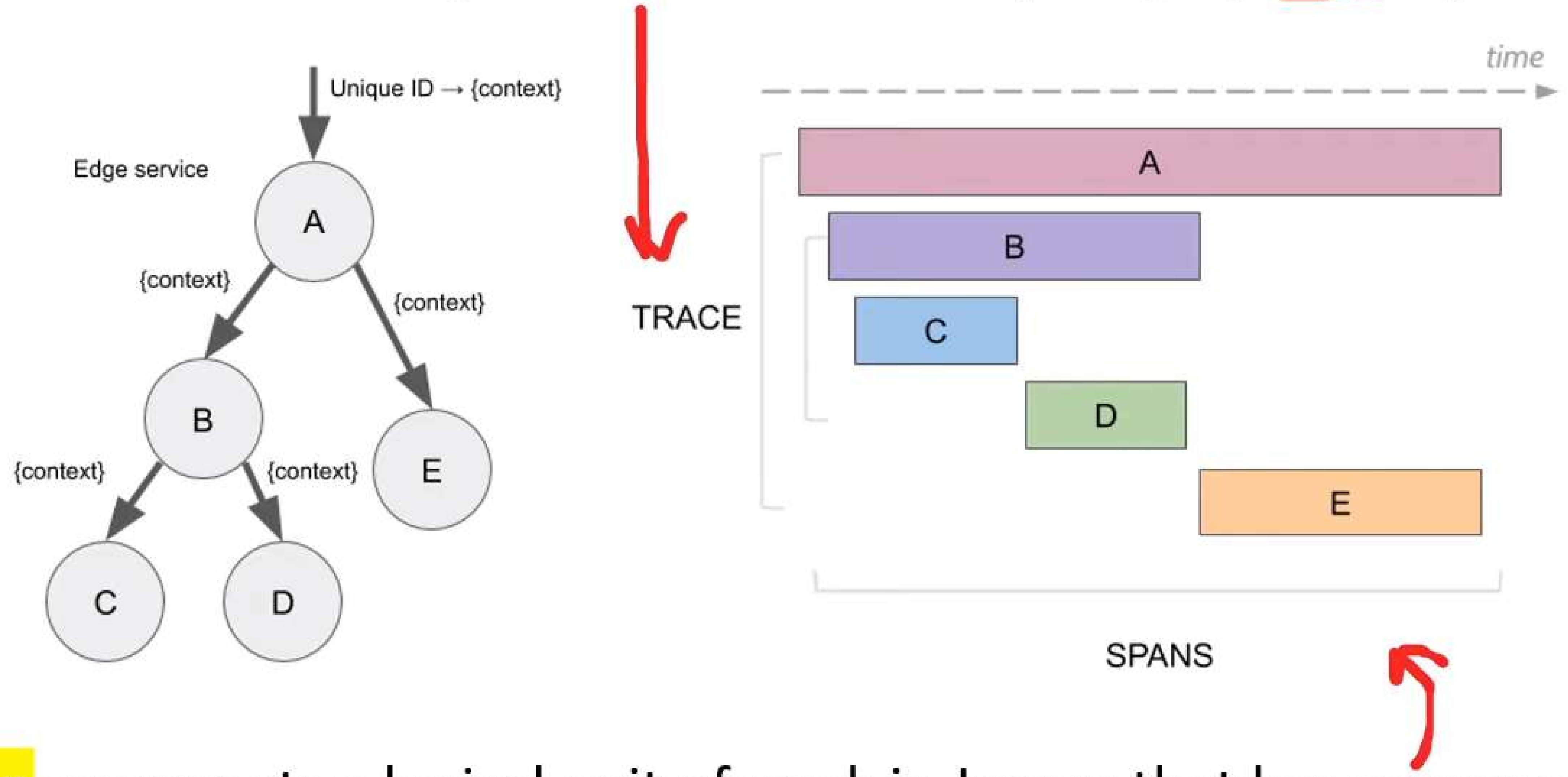
Grafana

Grafana is an open source **analytics and interactive visualization** web application
Grafana is commonly used along with a timeseries database like: InfluxDB, Prometheus or Graphite


Demo Dashboard
Zoom Out
an hour ago to a few seconds ago
Prometheus QPS [rate-5m]
/ /alerts /api/metrics /api/query /api/query_range /consoles/ /graph /heap /static/ prometheus
HTTP latency [s]
0.9 / 0.99 / 0.9 /alerts 0.99 /alerts 0.9 /api/metrics 0.99 /api/metrics 0.9 /api/query 0.99 /api/query 0.9 /api/query_range 0.99 /api/query_range 0.9 /consoles/
0.99 /consoles/ 0.9 /graph 0.99 /graph 0.9 /heap 0.99 /heap 0.9 /static/ 0.99 /static/ 0.9 prometheus 0.99 prometheus
Memory series count
localhost:9090 prometheus
Prometheus chunk operations [rate-5m]
clone create drop evict load persist pin transcode unpin
+ ADD ROW

Traces and Spans

A **trace** is a data/execution path through the system,
and can be thought of as a **directed acyclic graph (DAG)** of spans



A **span** represents a logical unit of work in Jaeger that has an operation name, the start time of the operation, and the duration.
Spans may be nested and ordered to model causal relationships.

Cost Management

Cheat sheets, Practice Exams and Flash cards 👉 www.exampro.co/kcna

- Label Resources
  - Visualize Costs Prometheus and Grafana
- Finding Idle And Unallocated Resources
  - Visualize Idle CPU, Memory and Storage Prometheus and Grafana
- Workload Rightsizing
  - VerticalPodAutoscaler — adjusting CPU, Mem of Pods
  - HorizontalPodAutoscaler — add remove pods to meet the demand
- Cluster Downsizing Opportunities
  - ClusterAutosscaler Add or remove Nodes to meet the demand
- Using Free Trials Of Kubernetes Cost Tools eg. Kubecost
- Estimate future costs, use a Load-Testing — eg. SpeedScale, K6, JMeter, Gatling
  - What does a 15 node EKS cluster cost for the year?
- Utilization waste (running but not using or not live)
  - Serverless Architect that Scales to Zero when no traffic for a period of time.
- Technical debt
  - Evaluate architecture to reduce amount of pods
  - Evaluate cloud-native technologies

Kubernetes System Logs and Klogs

Cheat sheets, Practice Exams and Flash cards 👉 www.exampro.co/kcna

System Logs

- System component logs record events happening in cluster, which can be very useful for debugging.
- You can configure log verbosity to see more or less detail.
- Logs can be as coarse-grained as showing errors within a component, or as fine-grained as showing step-by-step traces of events (like HTTP access logs, pod state changes, controller actions, or scheduler decisions).

In KubeCTL you can view logs from pods with the **logs** commands →

```
kubectl logs nginx --all-containers=true
```

Klog

- Klog is the Kubernetes logging library.
- klog generates log messages for the Kubernetes system components.