TRUELAYER

Are we observable yet?

Conference > Rusty Days

Speaker > Luca Palmieri

31st of July 2020, 6pm (CEST)

Hello!

Luca Palmieri

Lead Engineer @ TrueLayer

Co-organiser of Rust London User Group.
Active OSS contributor/maintainer.
Author of Zero to Production (in progress)

Twitter: @algo_luca
Blog: https://lpalmieri.com

Agenda

1. What is Donate Direct?
2. Journey to production
3. Metrics
4. Logging
5. Distributed tracing

What is Donate Direct?

About TrueLayer

Our open banking platform powers instant, global access to financial data and payments.

Our investors

Tencent

TEMASEK

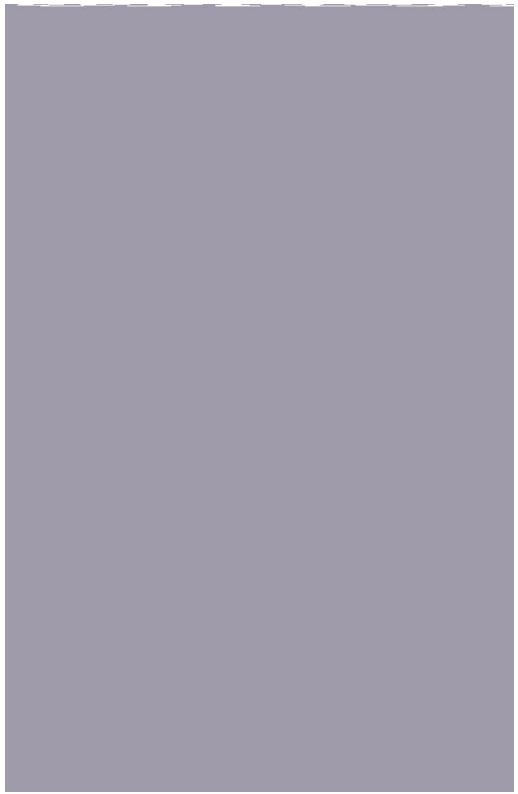
Northzone

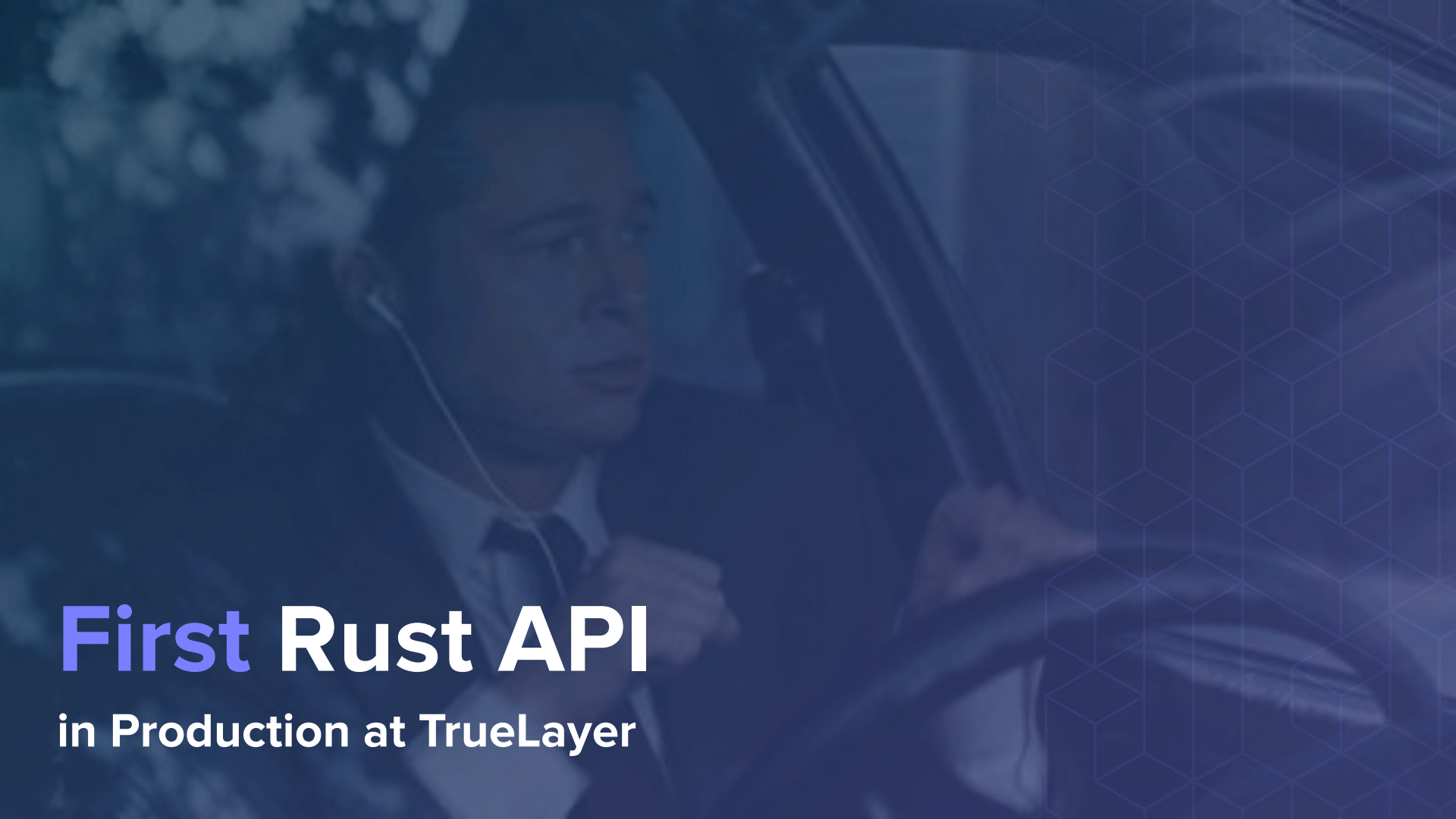
anthemis | group

connect

DonateDirect

https://donate.truelayer.com

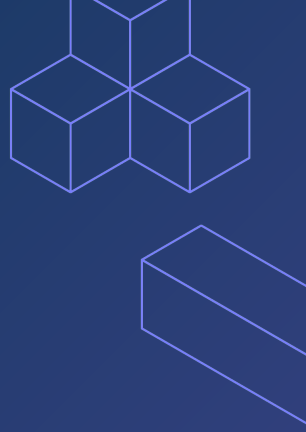




First Rust API

in Production at TrueLayer

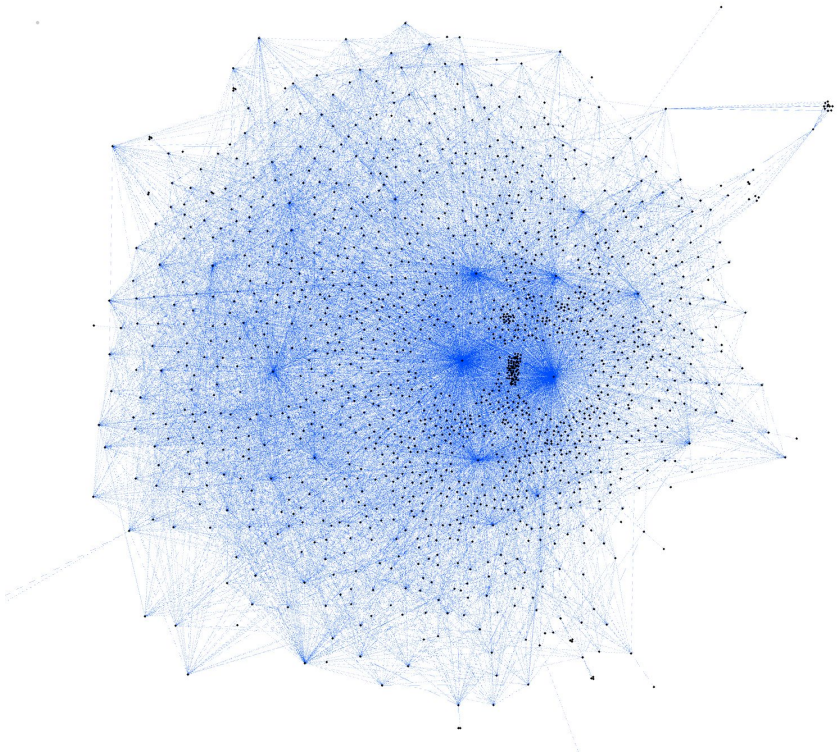
Journey to production




One does not simply walk into
Production

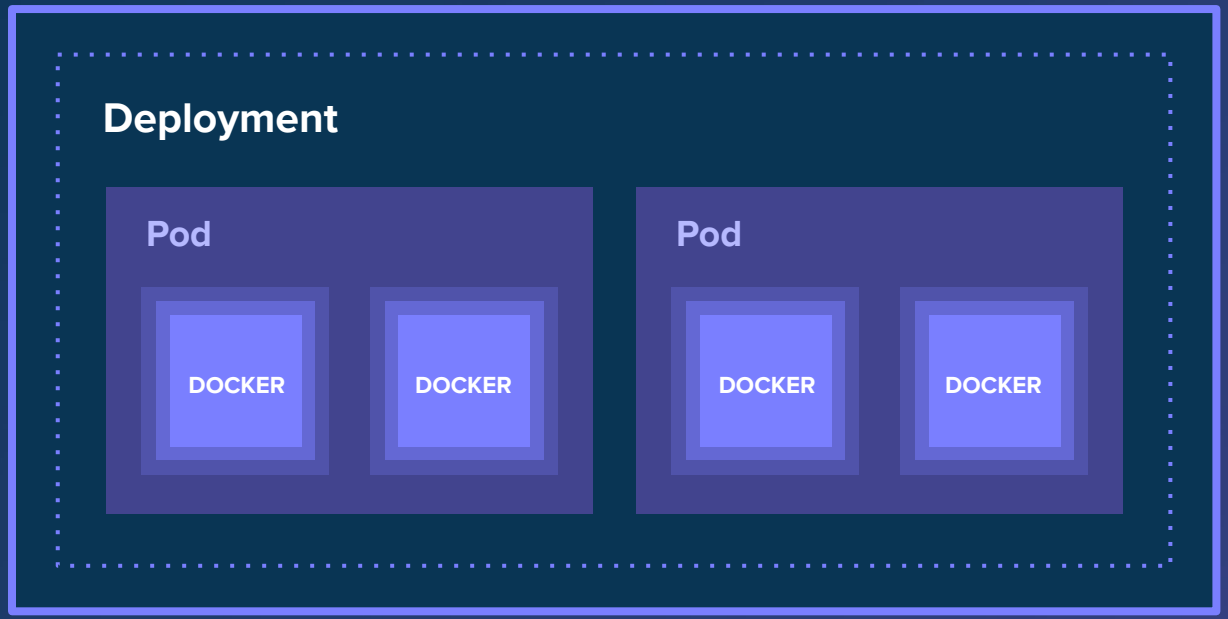
Production? Microservices!

Monzo's, not TrueLayer's



Source: *How does Monzo keep 1,600 microservices spinning?*

kubernetes



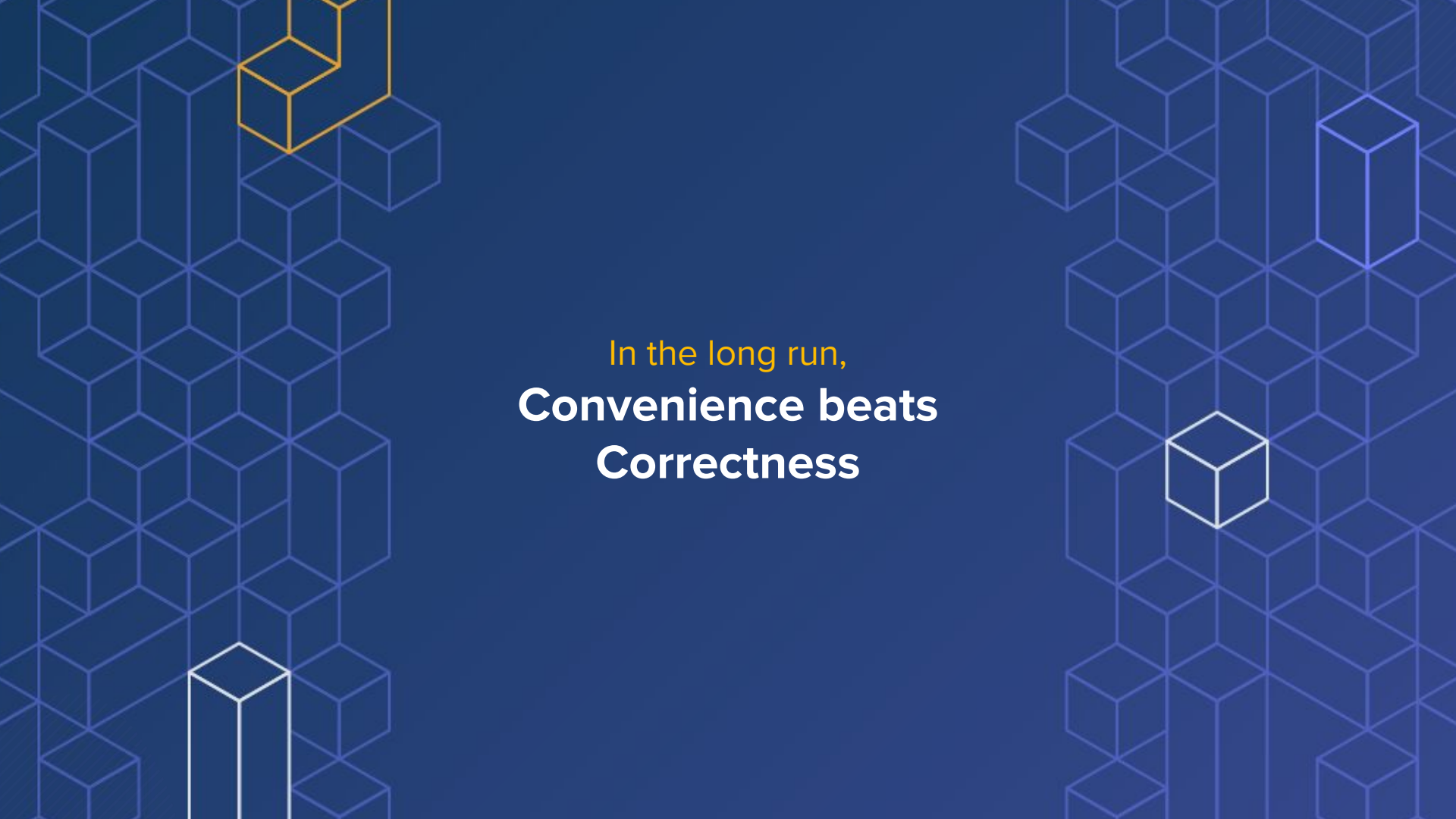
The Pre-Production Checklist

Fast

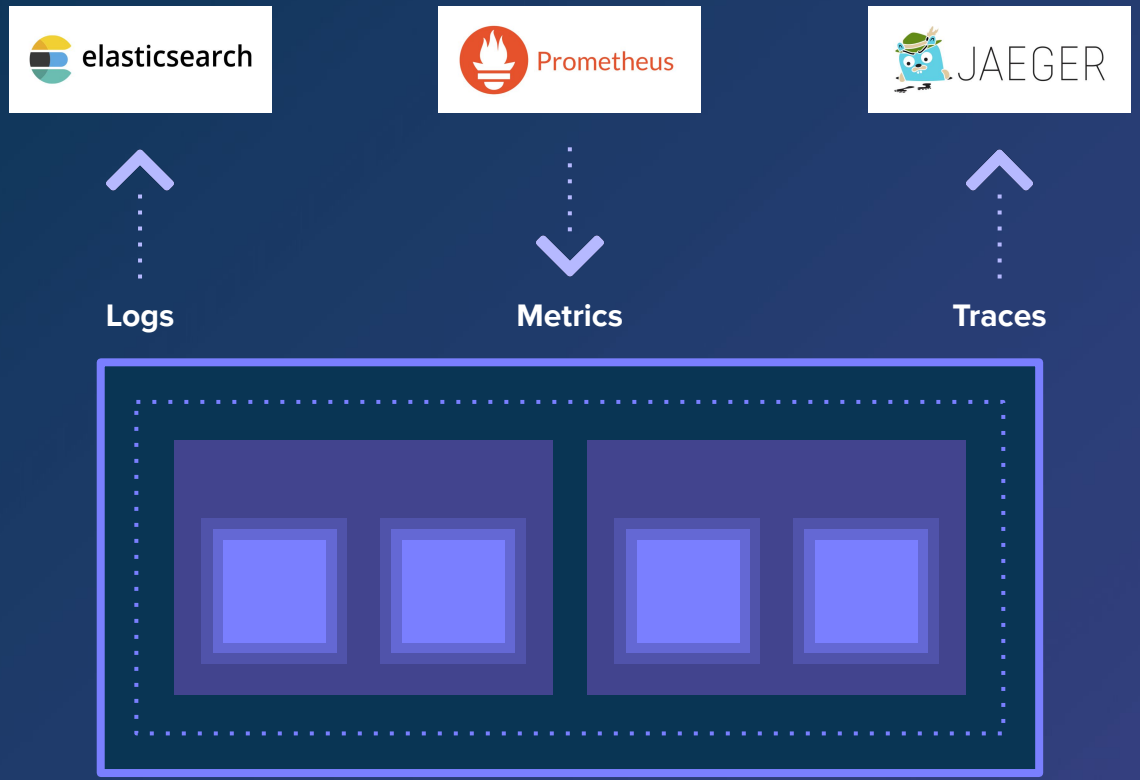
Reliable

Just ship it!

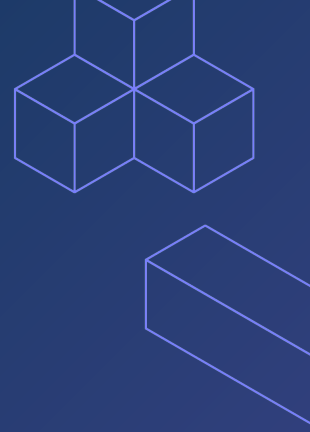
Metrics, Tracing, Logs, HPAs,
Alerts, Network Policies, Liveness/Readiness probes, ...


In the long run,
Convenience beats
Correctness

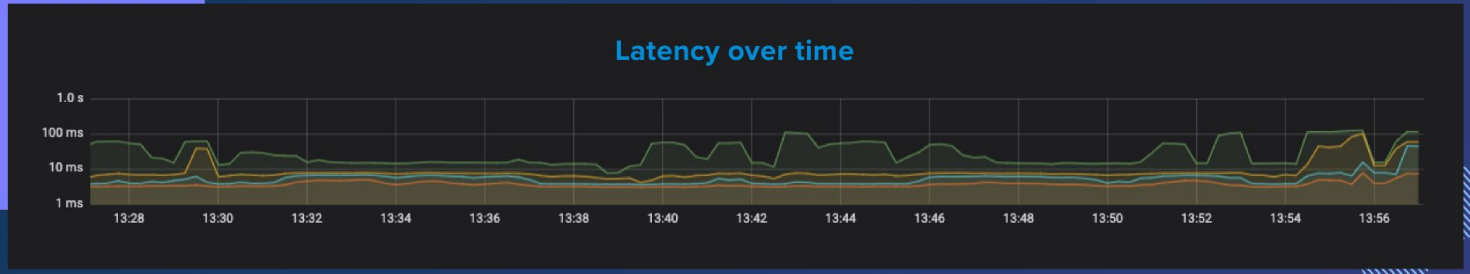
kubernetes

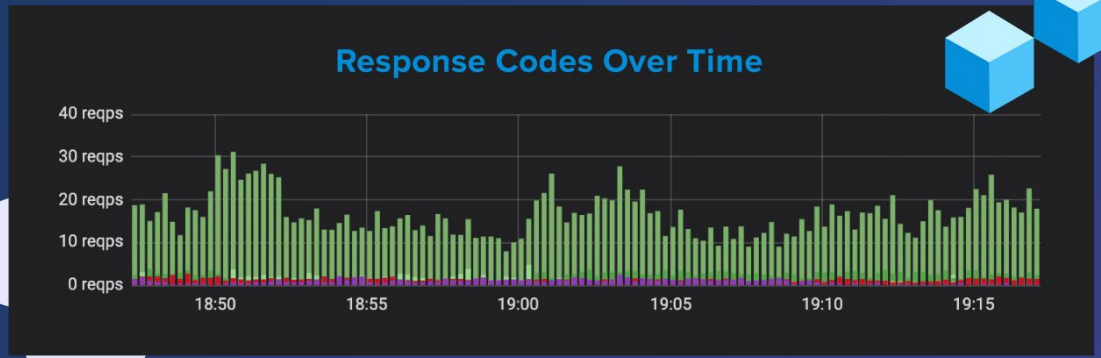


Metrics



Metrics

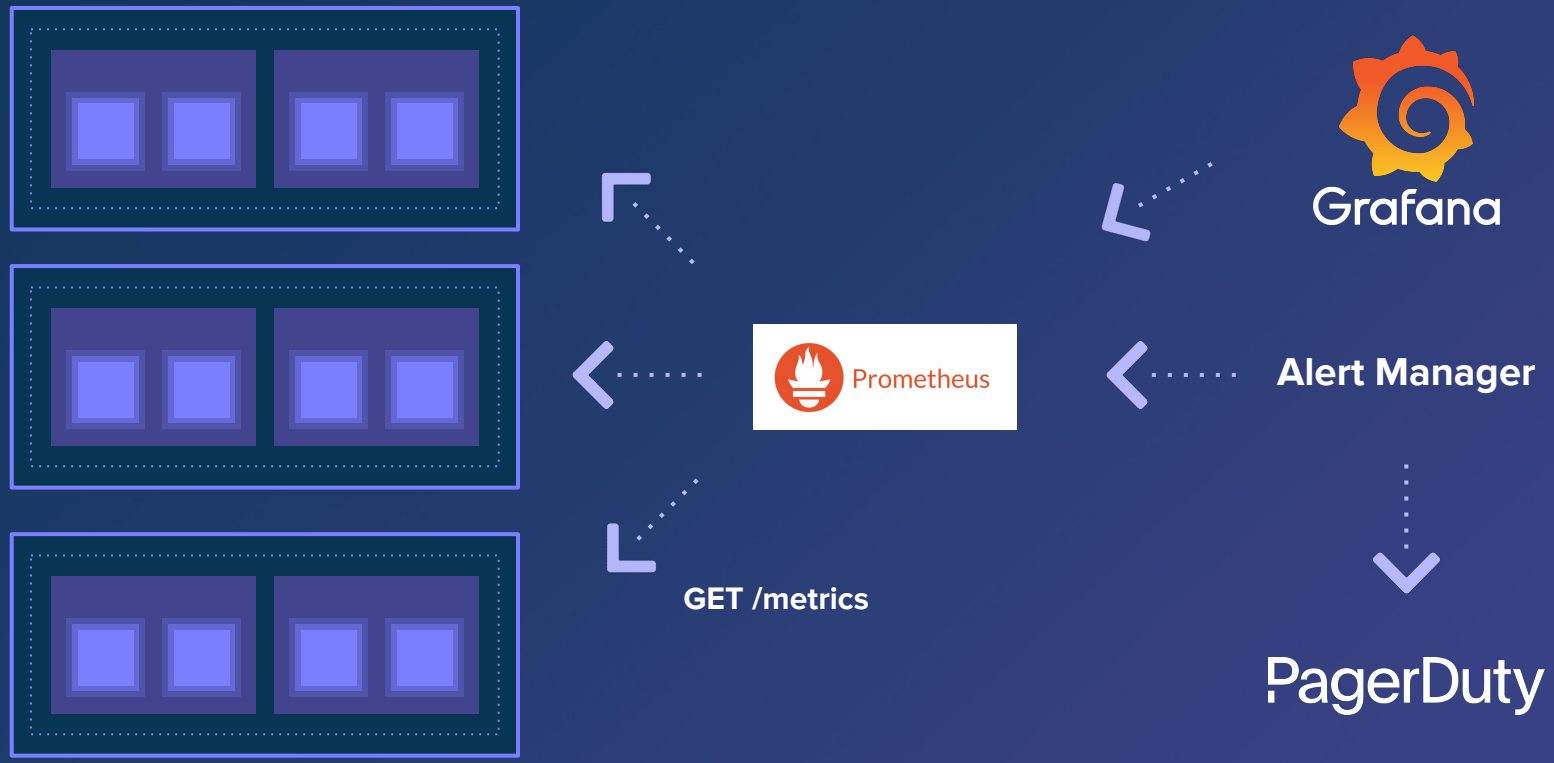

Latency over time
1.0 s
100 ms
10 ms
1 ms
13:28
13:30
13:32
13:34
13:36
13:38
13:40
13:42
13:44
13:46
13:48
13:50
13:52
13:54
13:56


Response Codes Over Time
40 reqps
30 reqps
20 reqps
10 reqps
0 reqps
18:50
18:55
19:00
19:05
19:10
19:15


Metrics give us an
aggregate picture
of the system state.

Metrics

```
# HELP http_requests_duration_seconds HTTP request duration in seconds for all requests
# TYPE http_requests_duration_seconds histogram
http_requests_duration_seconds_bucket{endpoint="/",method="GET",status="404",le="0.005"} 1601
http_requests_duration_seconds_bucket{endpoint="/",method="GET",status="404",le="0.01"} 1601
http_requests_duration_seconds_bucket{endpoint="/",method="GET",status="404",le="0.025"} 1601
http_requests_duration_seconds_bucket{endpoint="/",method="GET",status="404",le="0.05"} 1601
http_requests_duration_seconds_bucket{endpoint="/",method="GET",status="404",le="0.1"} 1601
http_requests_duration_seconds_bucket{endpoint="/",method="GET",status="404",le="0.25"} 1601
http_requests_duration_seconds_bucket{endpoint="/",method="GET",status="404",le="0.5"} 1601
http_requests_duration_seconds_bucket{endpoint="/",method="GET",status="404",le="1"} 1601
http_requests_duration_seconds_bucket{endpoint="/",method="GET",status="404",le="2.5"} 1601
http_requests_duration_seconds_bucket{endpoint="/",method="GET",status="404",le="5"} 1601
http_requests_duration_seconds_bucket{endpoint="/",method="GET",status="404",le="10"} 1601
http_requests_duration_seconds_bucket{endpoint="/",method="GET",status="404",le="+Inf"} 1601
http_requests_duration_seconds_sum{endpoint="/",method="GET",status="404"} 0.16400449699999978
http_requests_duration_seconds_count{endpoint="/",method="GET",status="404"} 1601
http_requests_duration_seconds_bucket{endpoint="/",method="HEAD",status="404",le="0.005"} 2
http_requests_duration_seconds_bucket{endpoint="/",method="HEAD",status="404",le="0.01"} 2
http_requests_duration_seconds_bucket{endpoint="/",method="HEAD",status="404",le="0.025"} 2
http_requests_duration_seconds_bucket{endpoint="/",method="HEAD",status="404",le="0.05"} 2
http_requests_duration_seconds_bucket{endpoint="/",method="HEAD",status="404",le="0.1"} 2
http_requests_duration_seconds_bucket{endpoint="/",method="HEAD",status="404",le="0.25"} 2v
```

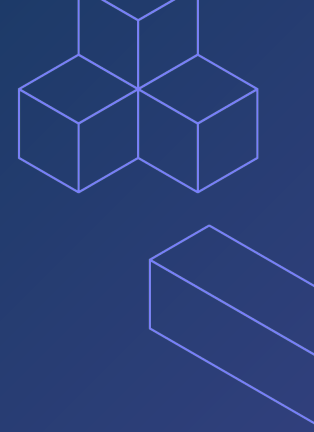

Grafana
Prometheus
Alert Manager
GET /metrics
PagerDuty

Metrics

`actix_web_prom` provides a pluggable middleware with standard Prometheus metrics out of the box.

```
let prom = PrometheusMetrics::new("api",
Some("/metrics"), None);

HttpServer::new(move || {
    App::new()
        .wrap(prom.clone())
        .service(web::resource("/health").to(health))
})
.bind("127.0.0.1:8080")?
.run()
.await?;
```

Logging



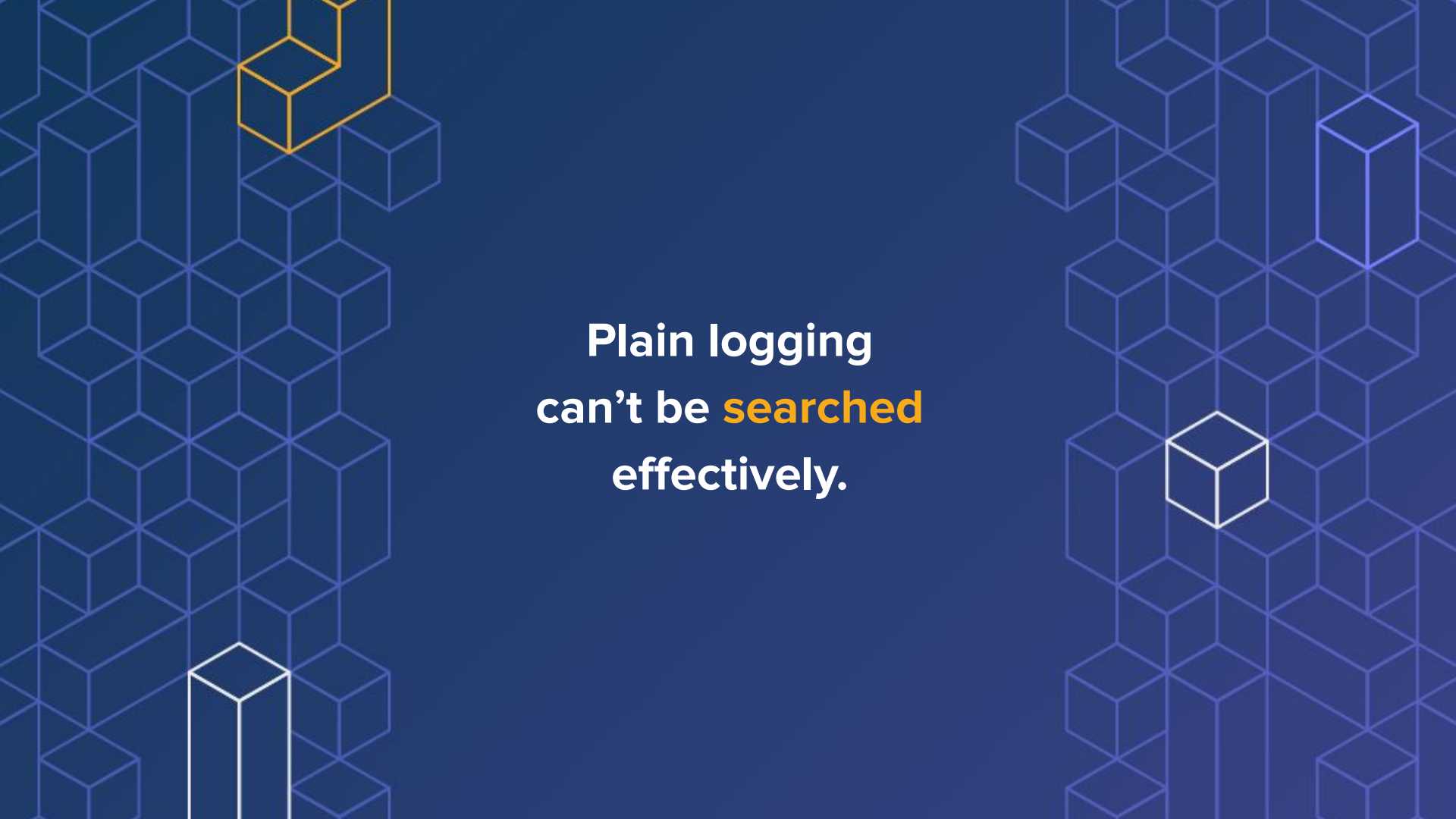

What happened
to request XYZ?

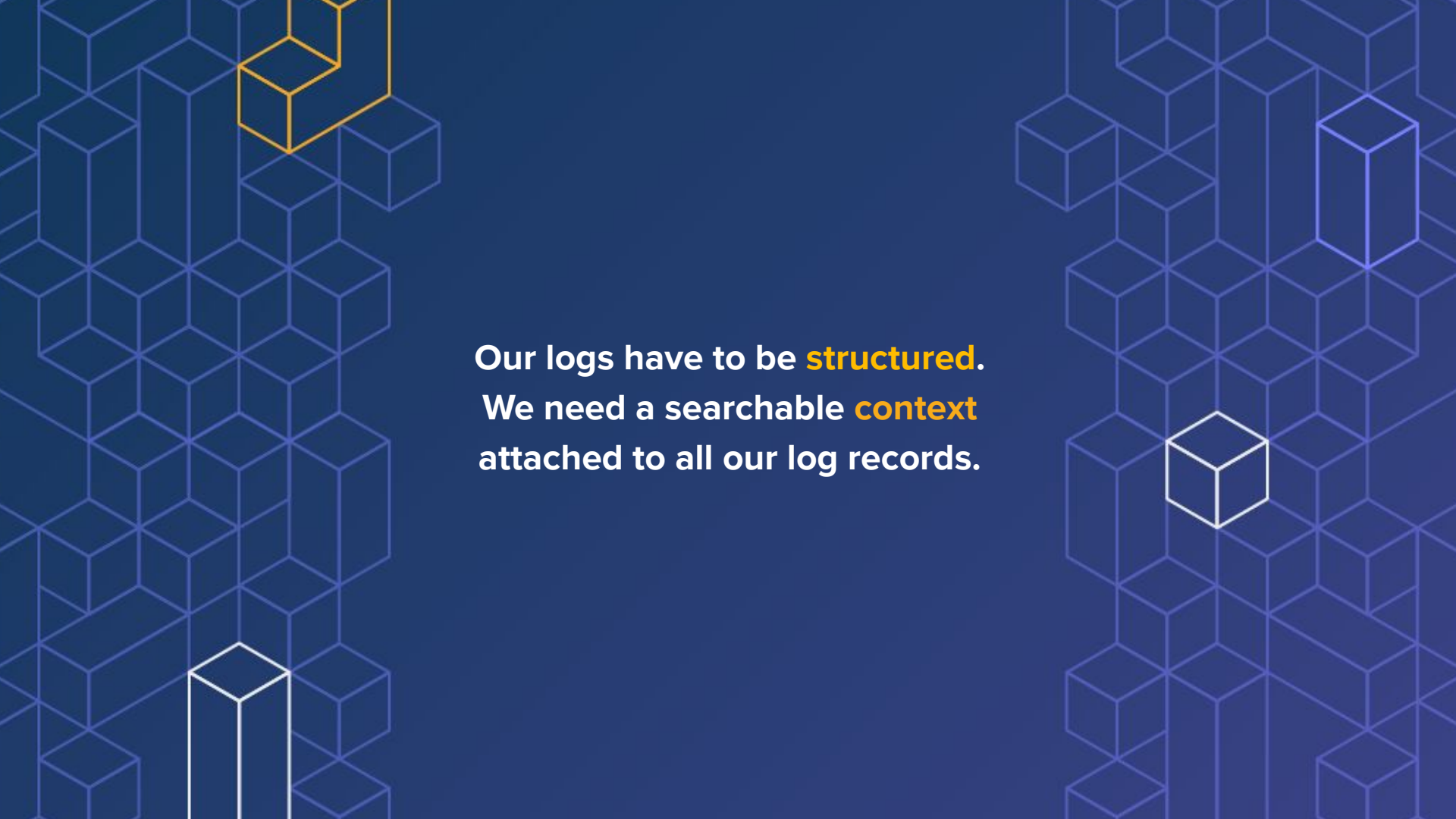
Logging


```rust
use log::{info, trace, warn};

pub fn shave_the_yak(yak: &mut Yak) {
    trace!("Commencing yak shaving");

    loop {
        match find_a_razor() {
            Ok(razor) => a{
                info!("Razor located: {}", razor);
                yak.shave(razor);
                break;
            }
            Err(err) => {
                warn!("Unable to locate a razor: {}, retrying", err);
            }
        }
    }
}
```

Logging


```
[2020-07-03T05:33:04Z WARN yak_shave] Unable to locate a razor: not found, retrying
[2020-07-03T05:33:04Z WARN yak_shave] Unable to locate a razor: not found, retrying
[2020-07-03T05:33:04Z WARN yak_shave] Unable to locate a razor: not found, retrying
[2020-07-03T05:33:04Z INFO yak_shave] Razor located: The Razor
```


Plain logging
can’t be searched
effectively.


Our logs have to be structured.
We need a searchable context
attached to all our log records.

Logging


```
fn yak_shave(user_id: Uuid) {
    let start = Instant::now();
    debug!("[YAK-SHAVE START]", o!("user_id" => user_id));

    // Some business logic
    debug!("[YAK-SHAVE EVENT] So cool!", o!("user_id" => user_id));
    sub_unit_of_work(..);

    let elapsed = start.elapsed();
    debug!("[YAK-SHAVE END]",
           o!("user_id" => user_id,
              "elapsed_milliseconds" => elapsed.as_millis()));
}
```

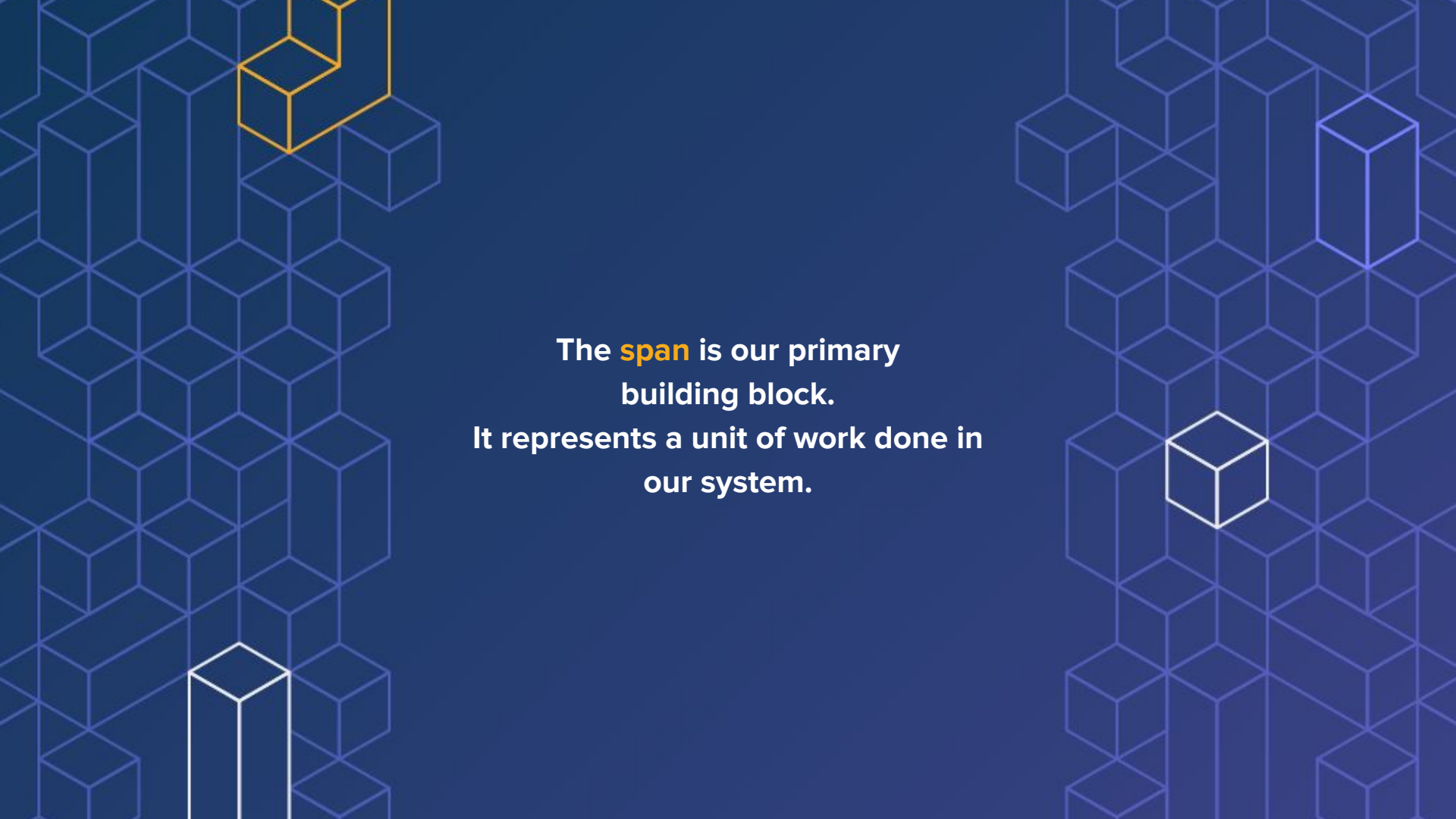
Logging


```
{"v":0,"name":"tracing_demo","msg":"[YAK-SHAVE - START]", "level":20, "hostname":"luca-XPS-15-9570", "pid":22305,
"time":"2020-07-03T05:50:40.459345507+00:00","user_id":"user-identifier"}

{"v":0,"name":"tracing_demo","msg":"[YAK-SHAVE - EVENT] So cool!","level":20,"hostname":"luca-XPS-15-9570","pid":22305,
"time":"2020-07-03T05:50:40.459424677+00:00","user_id":"user-identifier"}

{"v":0,"name":"tracing_demo","msg":"[SUB-UNIT-OF-WORK - START]","level":20,"hostname":"luca-XPS-15-9570","pid":22305,
"time":"2020-07-03T05:50:40.459493239+00:00","user_id":"user-identifier","task":1}

{"v":0,"name":"tracing_demo","msg":"[SUB-UNIT-OF-WORK - END]","level":20,"hostname":"luca-XPS-15-9570","pid":22305,
"time":"2020-07-03T05:50:40.459563036+00:00","elapsed_milliseconds":0,"user_id":"user-identifier","task":1}

{"v":0,"name":"tracing_demo","msg":"[YAK-SHAVE - END]","level":20,"hostname":"luca-XPS-15-9570","pid":22305,
"time":"2020-07-03T05:50:40.459626734+00:00","elapsed_milliseconds":0,"user_id":"user-identifier"}
```

Structured logging

Orphan log events are the wrong abstraction.

```
fn yak_shave(user_id: Uuid) {
    let start = Instant::now();
    debug!("[YAK-SHAVE START]", o!("user_id" => user_id));

    debug!("[YAK-SHAVE EVENT] So cool!", o!("user_id" =>
user_id));
    // Do we pass the logger down?
    // Do we pass user_id to keep in the context?
    sub_unit_of_work(..);

    let elapsed = start.elapsed();
    debug!("[YAK-SHAVE END]",
           o!("user_id" => user_id,
              "elapsed_milliseconds" => elapsed.as_millis()));
}
```


The span is our primary
building block.
It represents a unit of work done in
our system.

Structured Logging


```rust
use tracing::{debug_span, debug};

fn yak_shave(user_id: Uuid) {
    let span = debug_span!("yak-shave", user_id);
    let _enter = span.enter()

    // This event is attached to the yak-shave span
    // No need to capture again the user_id property
    debug!("So cool!");
    sub_unit_of_work(..);
}
```

Spans!

Procedural macros can be used to reduce the visual code noise.

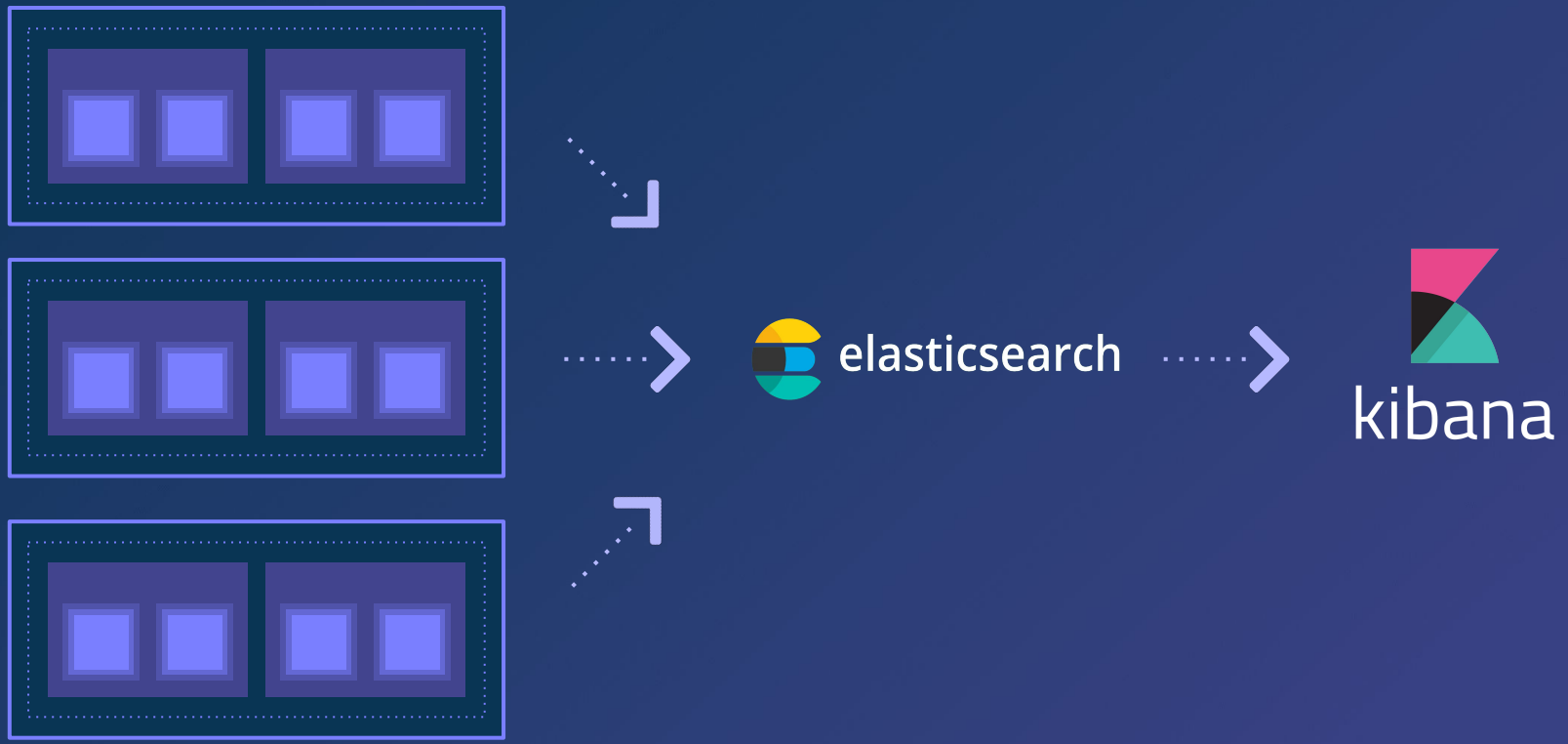
Convenient, hence we will use it more!

```
#[tracing::instrument]
fn yak_shave(user_id: Uuid) {
    debug!("So cool!");
    sub_unit_of_work(..);
}
```

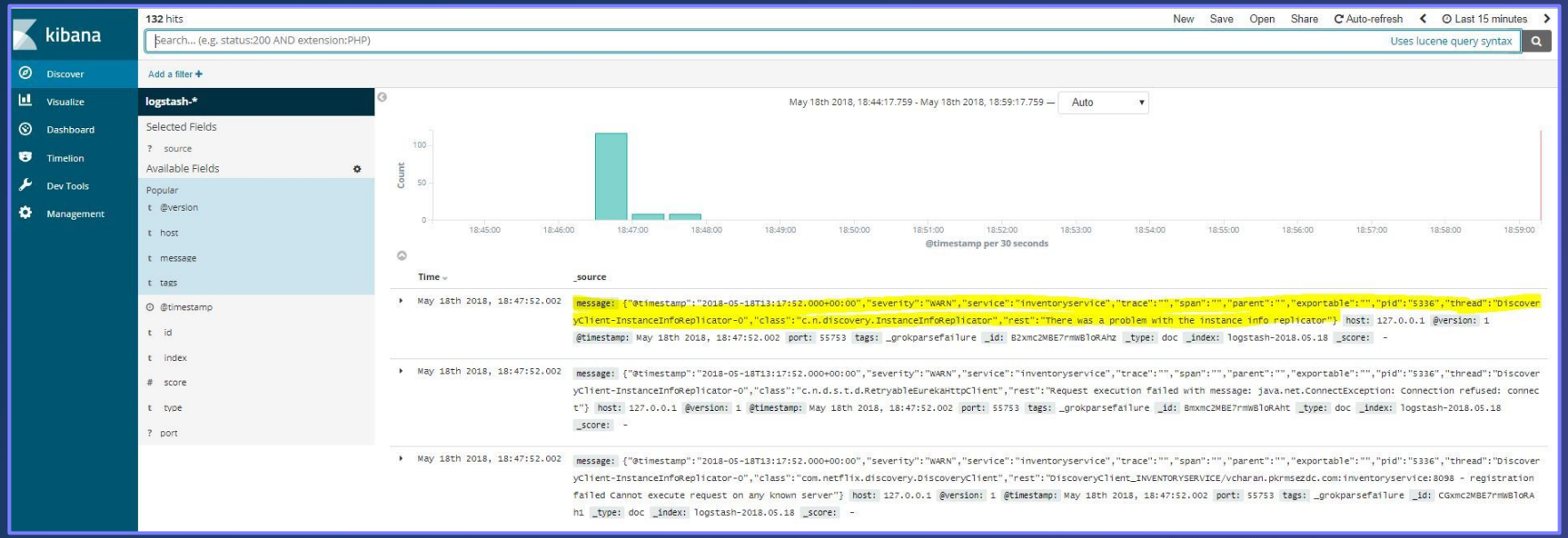
Instrument once, ship everywhere

Using different subscribers we can:

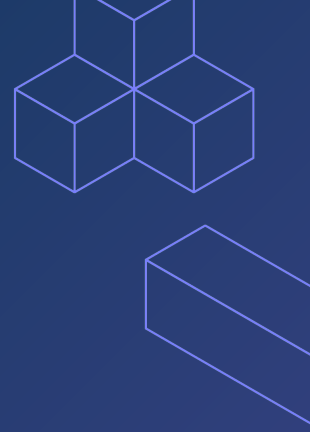
- Log to stdout or to another sink;
- Send spans and events to a distributed tracing system.

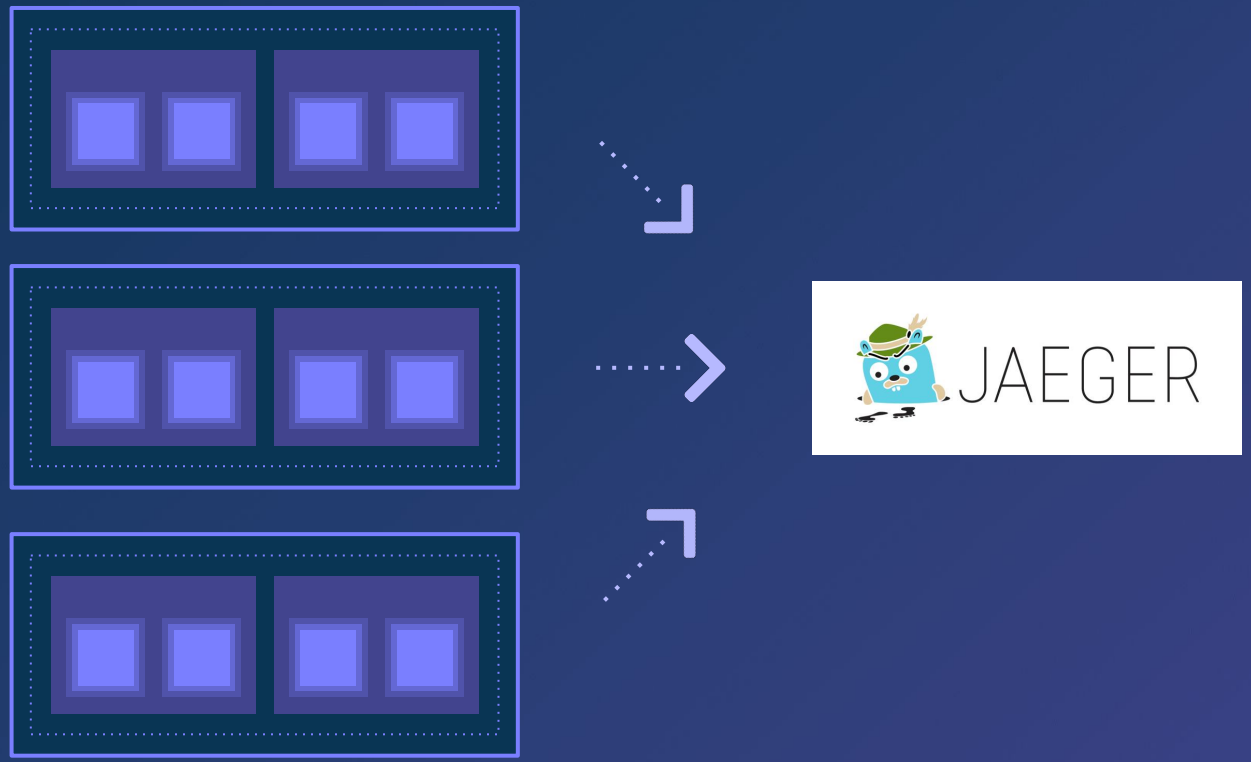

elasticsearch
kibana

Kibana

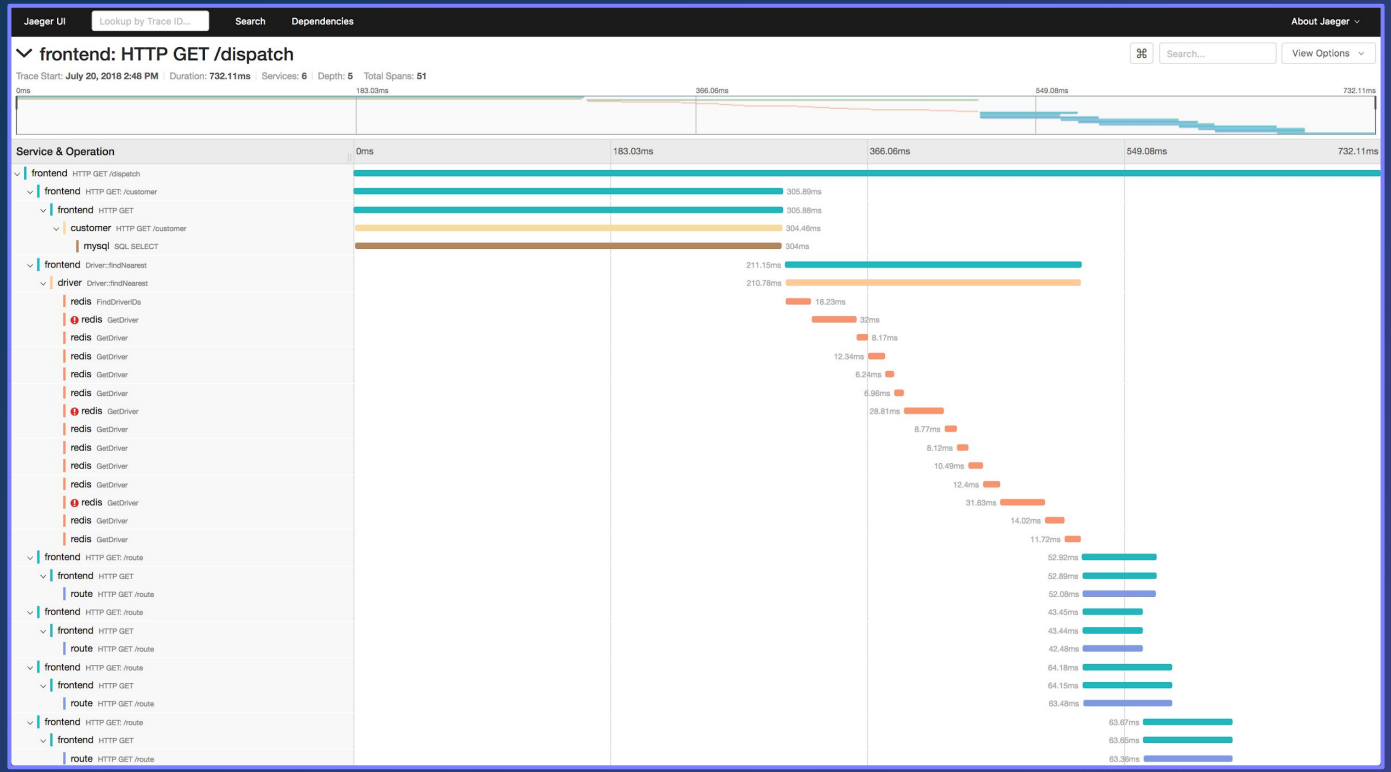

kibana
132 hits
New Save Open Share Auto-refresh Last 15 minutes
Search... (e.g. status:200 AND extension:PHP)
Uses lucene query syntax
Discover
Visualize
Dashboard
Timelion
Dev Tools
Management
Add a filter +
logstash-*
Selected Fields
? source
Available Fields
Popular
t @version
t host
t message
t tags
@timestamp
t id
t index
score
t type
? port
May 18th 2018, 18:44:17.759 - May 18th 2018, 18:59:17.759 — Auto
Count
@timestamp per 30 seconds
Time
_source
May 18th 2018, 18:47:52.002
May 18th 2018, 18:47:52.002
May 18th 2018, 18:47:52.002

Distributed tracing



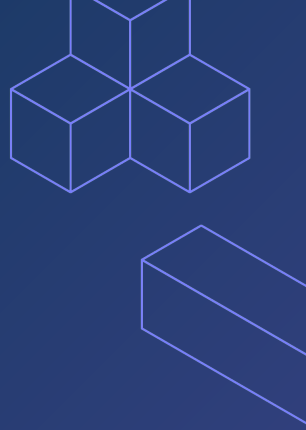

JAEGER

Jaeger



TrueLayer

Recap



Key takeaways

1. Lack of telemetry is a ticking bomb
2. Diagnostic instrumentation has to be easy
3. Metrics to alert and monitor system state
4. High cardinality is key to being able to detect and triage unknown unknowns
5. Span as unit of work abstraction
6. You must be able to trace a request across different services

Hiring!

1x Rust Back End Engineers

Milan office (Italy)
Core Banking project.

Opening: https://apply.workable.com/truelayer/j/37748BA121/

LinkedIn: luca-palmieri
Twitter: @algo_luca
Web: https://lpalmieri.com

Q&A

Luca Palmieri

Lead Engineer @ TrueLayer

Co-organiser of Rust London User Group.
Active OSS contributor/maintainer.
Author of Zero to Production (in progress)

Twitter: @algo_luca
Web: https://lpalmieri.com