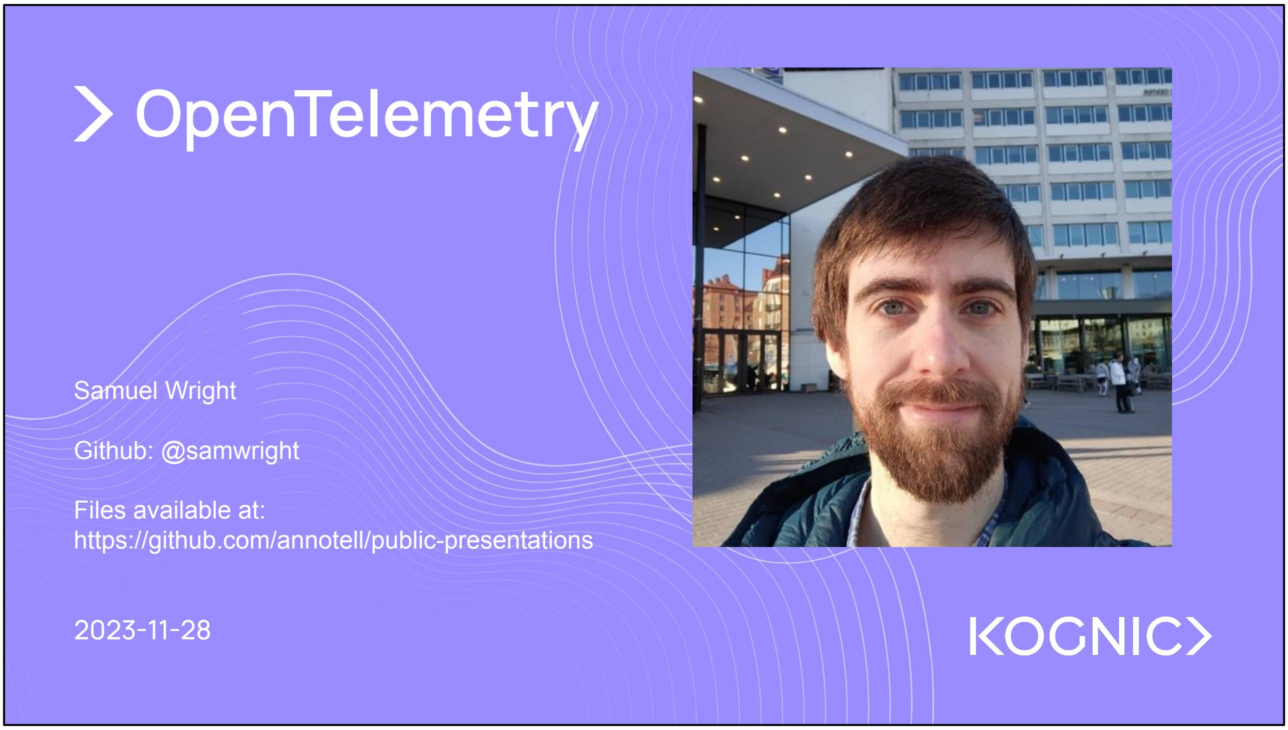
KOGNIC

Directing AI through Data

Hello! I'm from Kognic. We are a best-in-class sensor-fusion annotation solution for assisted driving, used to explore, shape, and explain datasets.



> OpenTelemetry

Samuel Wright

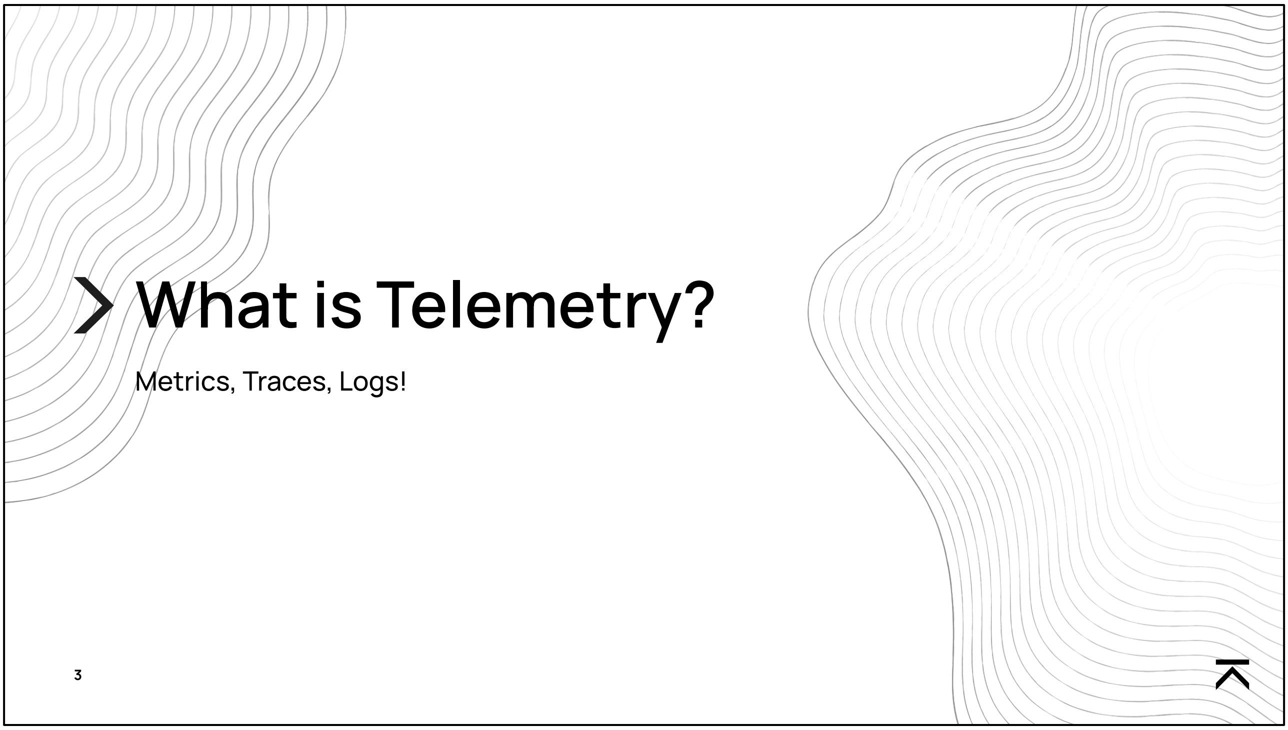
Github: @samwright

Files available at:
https://github.com/annotell/public-presentations

2023-11-28

KOGNIC>

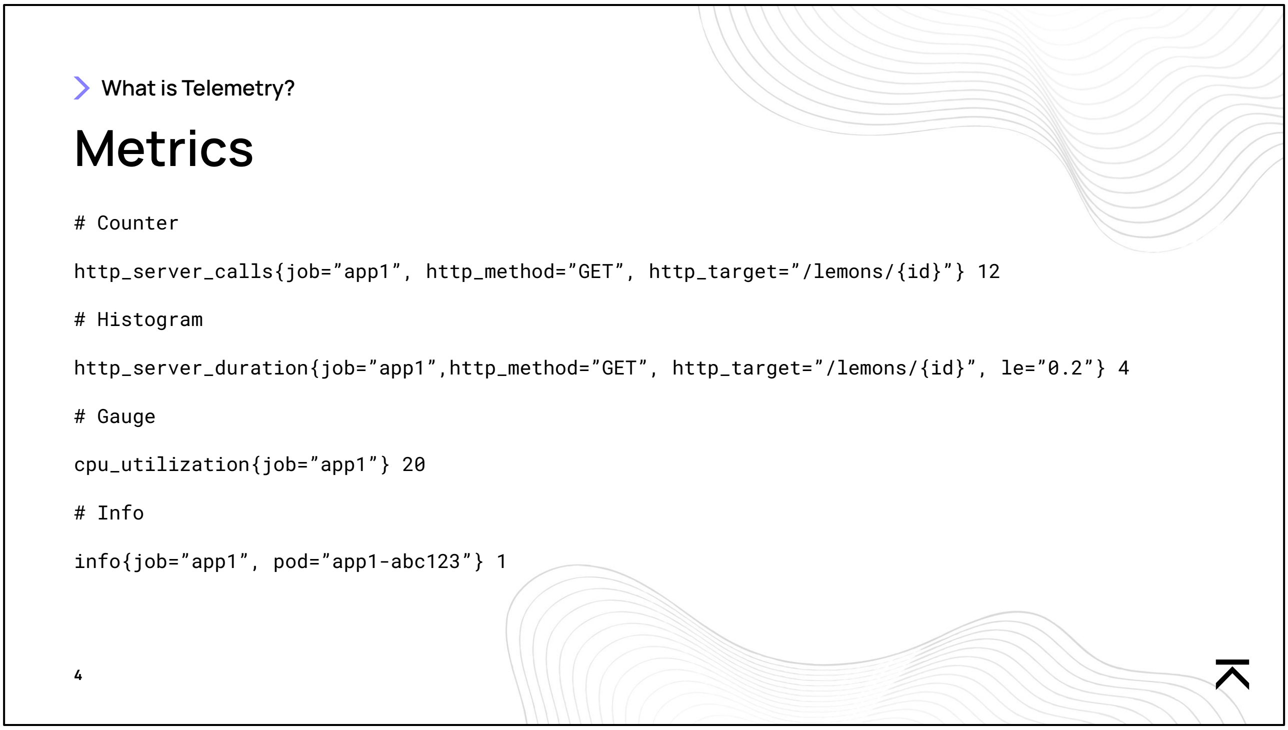
- Moved here in March this year
- Formerly a software engineer
- Been platform engineering on and off for 3 years
- You will find this presentation and all the otel config here



What is Telemetry?

Metrics, Traces, Logs!

- Before we get to OpenTelemetry, what is Telemetry?



What is Telemetry?

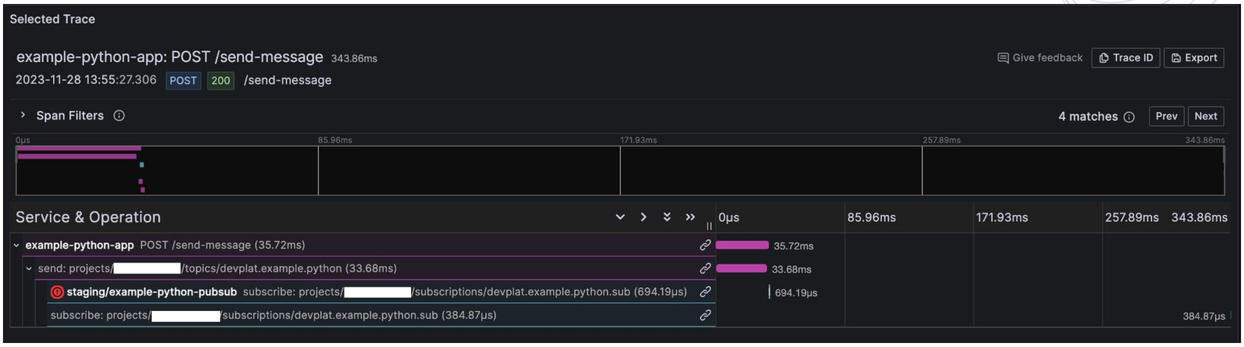
Metrics

```
# Counter

http_server_calls{job=”app1”, http_method=”GET”, http_target=”/lemons/{id}”} 12

# Histogram

http_server_duration{job=”app1”,http_method=”GET”, http_target=”/lemons/{id}”, le=”0.2”} 4

# Gauge

cpu_utilization{job=”app1”} 20

# Info

info{job=”app1”, pod=”app1-abc123”} 1
```

- We can have Gauge, Counter, Histogram metrics
- This is the prometheus format
- The labelset defines a time-series for the value on the right
- Adding metrics in apps is very easy because
  - the aggregation happens on the app - very specific and efficient
  - so long as you don’t have too many time-series, incrementing a counter costs no extra memory
- Labels provide enough context to answers questions like:
  - How many requests is the app handling?
  - How long does it take to handle a request?
  - How many of those requests resulted in an error?
- but they’re limited in cardinality.
- Some extra context available on ‘info’ metric

What is Telemetry?

Traces



Selected Trace

example-python-app: POST /send-message 343.86ms

2023-11-28 13:55:27.306 POST 200 /send-message

Span Filters

4 matches

Service & Operation

example-python-app POST /send-message (35.72ms) — 35.72ms

send: projects/[redacted]/topics/devplat.example.python (33.68ms) — 33.68ms

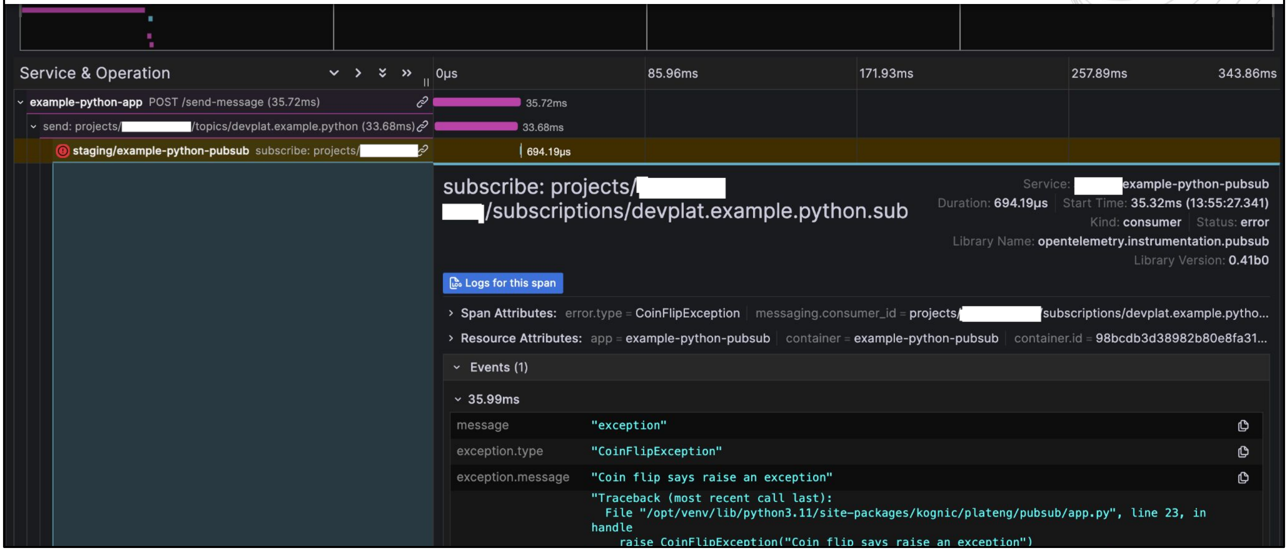
staging/example-python-pubsub subscribe: projects/[redacted]/subscriptions/devplat.example.python.sub (694.19μs) — 694.19μs

subscribe: projects/[redacted]/subscriptions/devplat.example.python.sub (384.87μs) — 384.87μs

- It shows a hierarchy of who called what when, providing lots of context
- A span contains:
    - A duration
    - Some attributes (any cardinality - our backend Tempo uses columnar storage, so there are no indices created for attributes, meaning they can be high-cardinality and verbose))
    - The “parent” span’s ID
    - The trace ID (shared by all spans in the trace)
    - A list of events
- In an app, when making a request:
    - Create a span in memory with random spanID + traceID
    - Pass span ID + trace ID in headers
- When receiving a request:
    - Get parent span ID + trace ID from headers
    - Create a span while handling request
- All apps periodically push spans
- Context propagation within app (request handling, through the app’ code, to a client request) can work with ThreadLocal storage, but requires instrumentation, e.g. when putting a job in a threadpool

What is Telemetry?

Traces



Service & Operation

0µs 85.96ms 171.93ms 257.89ms 343.86ms

- example-python-app POST /send-message (35.72ms) — 35.72ms
  - send: projects/[redacted]/topics/devplat.example.python (33.68ms) — 33.68ms
    - staging/example-python-pubsub subscribe: projects/[redacted] — 694.19µs

subscribe: projects/[redacted]/subscriptions/devplat.example.python.sub

Service: [redacted]example-python-pubsub
Duration: 694.19µs | Start Time: 35.32ms (13:55:27.341)
Kind: consumer | Status: error
Library Name: opentelemetry.instrumentation.pubsub
Library Version: 0.41b0

Logs for this span

› Span Attributes: error.type = CoinFlipException | messaging.consumer_id = projects/[redacted]subscriptions/devplat.example.pytho...

› Resource Attributes: app = example-python-pubsub | container = example-python-pubsub | container.id = 98bcdb3d38982b80e8fa31...

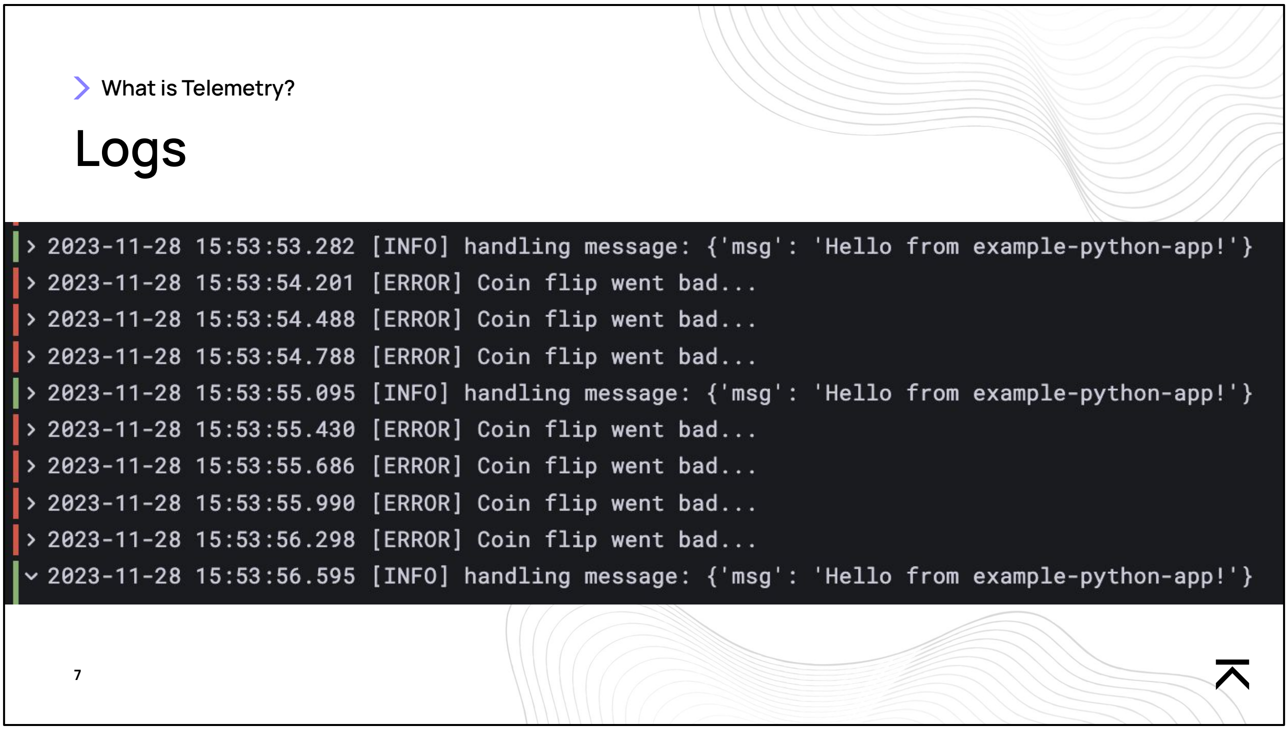
⌄ Events (1)

⌄ 35.99ms

message	"exception"
exception.type	"CoinFlipException"
exception.message	"Coin flip says raise an exception"

```
"Traceback (most recent call last):
  File "/opt/venv/lib/python3.11/site-packages/kognic/plateng/pubsub/app.py", line 23, in
handle
    raise CoinFlipException("Coin flip says raise an exception")
```

And here is a span in all its glory



What is Telemetry?

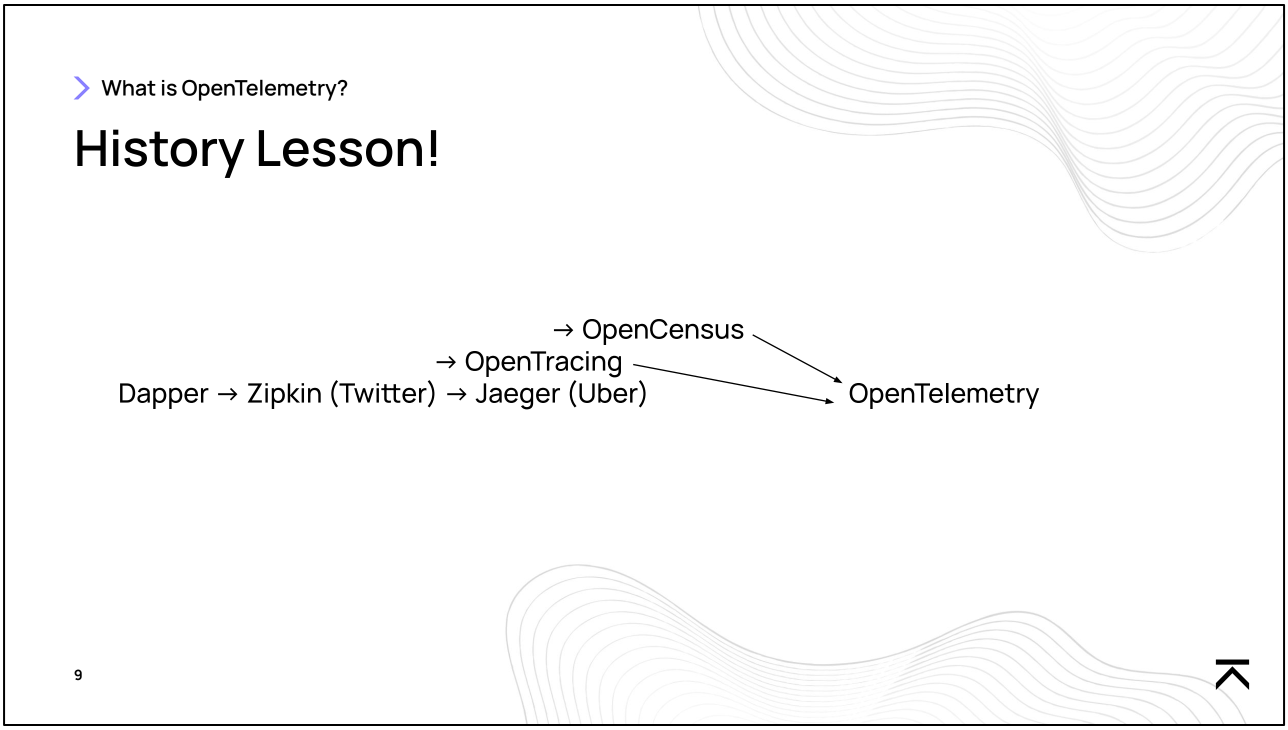
Logs

```
> 2023-11-28 15:53:53.282 [INFO] handling message: {'msg': 'Hello from example-python-app!'}
> 2023-11-28 15:53:54.201 [ERROR] Coin flip went bad...
> 2023-11-28 15:53:54.488 [ERROR] Coin flip went bad...
> 2023-11-28 15:53:54.788 [ERROR] Coin flip went bad...
> 2023-11-28 15:53:55.095 [INFO] handling message: {'msg': 'Hello from example-python-app!'}
> 2023-11-28 15:53:55.430 [ERROR] Coin flip went bad...
> 2023-11-28 15:53:55.686 [ERROR] Coin flip went bad...
> 2023-11-28 15:53:55.990 [ERROR] Coin flip went bad...
> 2023-11-28 15:53:56.298 [ERROR] Coin flip went bad...
⌄ 2023-11-28 15:53:56.595 [INFO] handling message: {'msg': 'Hello from example-python-app!'}
```

- Unstructured data send to stdout/stderr
- Kubernetes handles writing this to files on the node.
- *Can* be used for everything, but lack of structure adds difficulties
- Probably best used for app-level events (DB connection down, SIGTERM received, cleanup job finished)

What is OpenTelemetry?

Standing on the shoulders of giants



What is OpenTelemetry?

History Lesson!

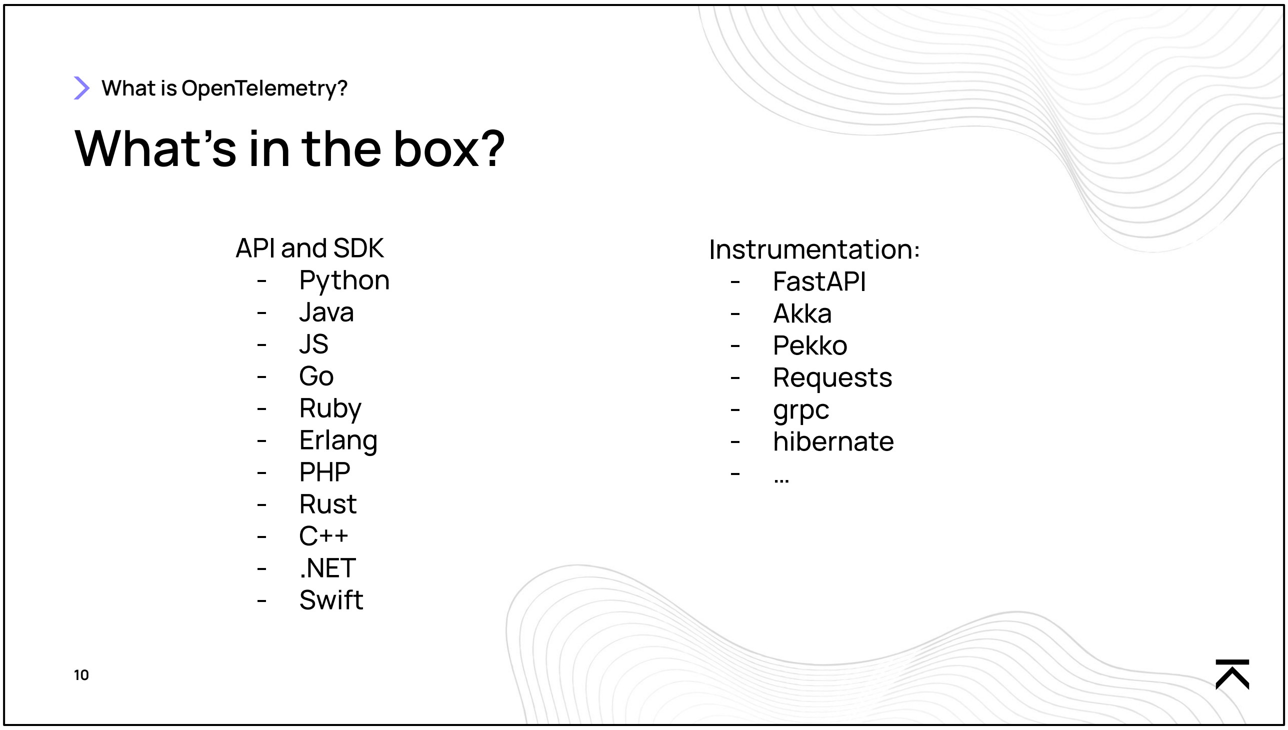
→ OpenCensus

→ OpenTracing

Dapper → Zipkin (Twitter) → Jaeger (Uber)

OpenTelemetry

- Google released a paper in 2010 about distributed tracing (Dapper)
- Then came Zipkin in 2012, OpenTracing in 2015 (CNCF project) + Jaeger in 2015 (later accepted by CNCF in 2019)
- Google released OpenCensus in 2017, and confusion reigned
- OpenCencus + OpenTelemetry merged in 2019

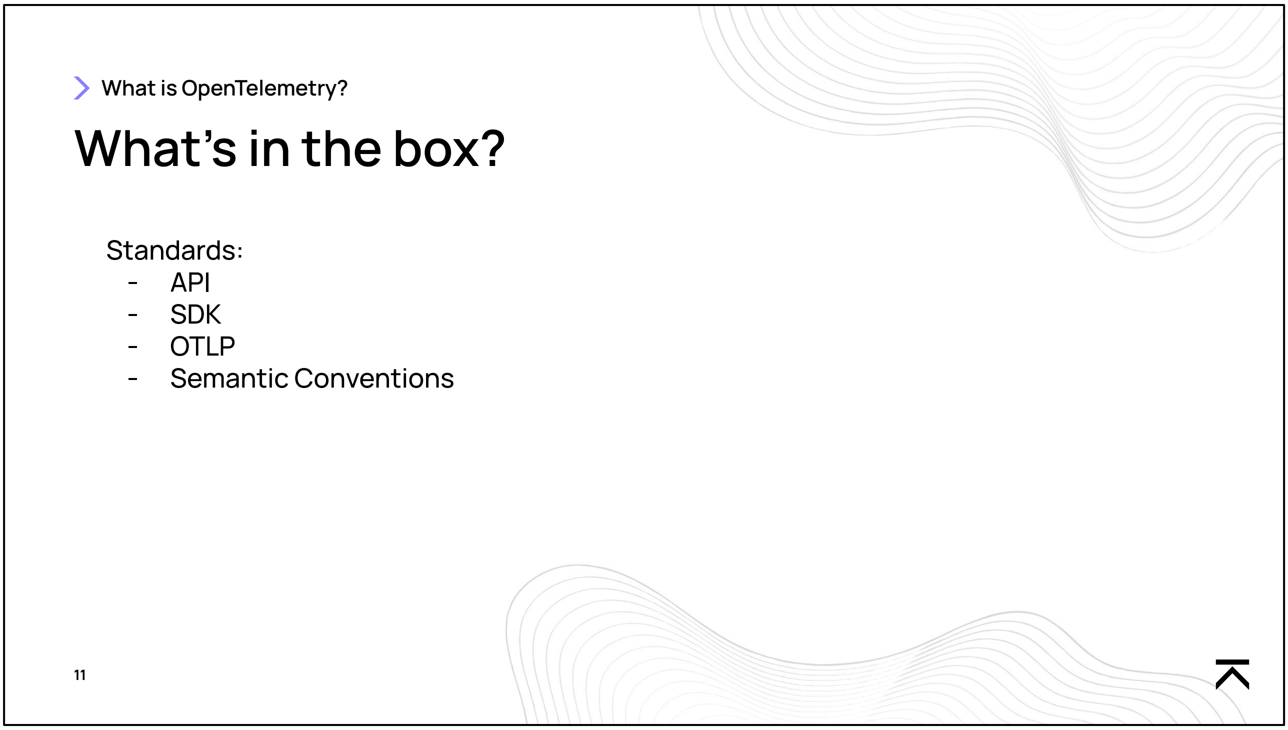


What’s in the box?

API and SDK
- Python
- Java
- JS
- Go
- Ruby
- Erlang
- PHP
- Rust
- C++
- .NET
- Swift

Instrumentation:
- FastAPI
- Akka
- Pekko
- Requests
- grpc
- hibernate
- …

- The API (for creating metrics etc) and SDK (e.g. exporter implementations) are there for lots of languages
- Each language has a large number of library instrumentations
- Exactly how instrumentation works is language-specific



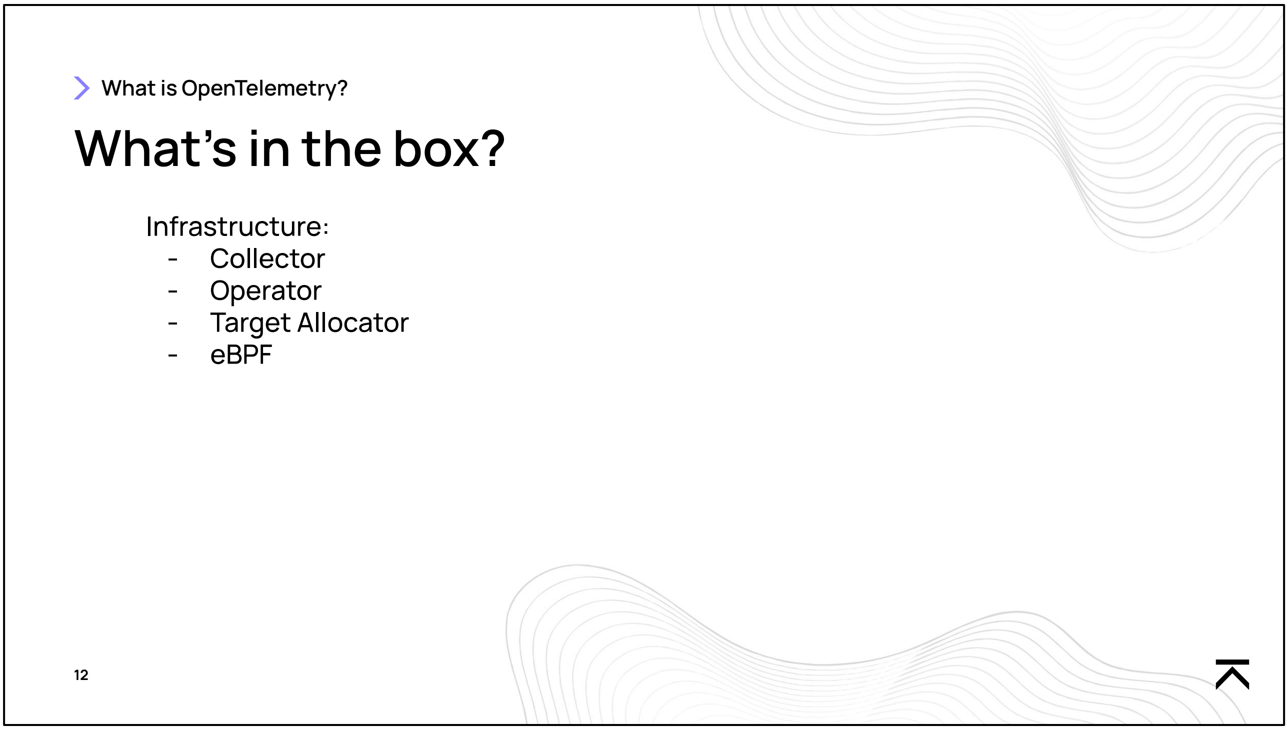
What is OpenTelemetry?

What’s in the box?

Standards:

- API
- SDK
- OTLP
- Semantic Conventions

- API: Creating a span or metric feels the same in every language
- SDK: Exporters should work the same across languages
- OTLP: wire format (protobuf + http) for sending all types of telemetry with common concepts
- Semantic Conventions: attribute names (labels in prometheus) should all look the same regardless of which instrumentated library generated them



What is OpenTelemetry?

What's in the box?

Infrastructure:
- Collector
- Operator
- Target Allocator
- eBPF

Infrastructure-wise, we have:

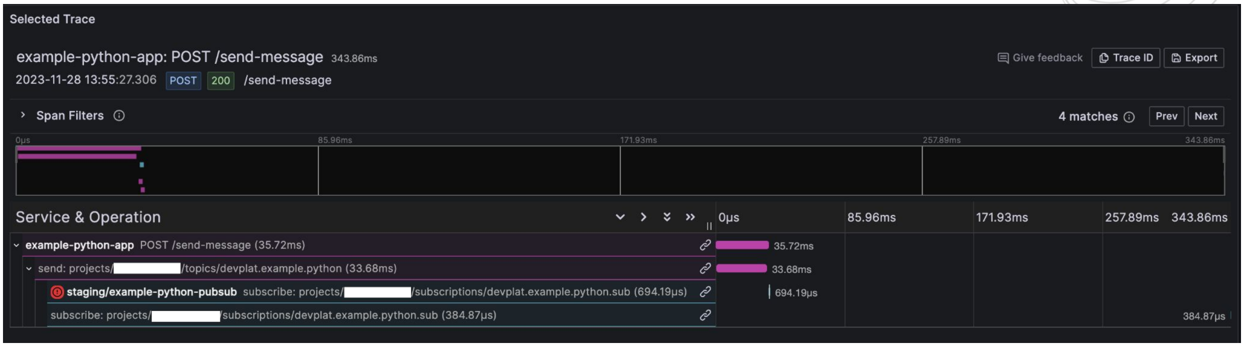
- Collector to act as a gateway or agent for app to send telemetry to
  - receive data in all formats, and export it in all formats
  - both push and pull directions
  - hiding the backend away from apps (where reconfiguring can be tricky)
  - it should keep telemetry if the backend goes down temporarily
  - and in the middle it should process the telemetry (rename things, add extra context to telemetry, and lots more)
- Operator
  - You can create collector CRs, but I just use the helm chart (which has extra features)
  - can instrument apps in-cluster (I'll come back to that later)
- Target Allocator
  - Can watch for prometheus CRs (ServiceMonitor, PodMonitor) and "allocate" those "targets" to available collectors
- eBPF
  - Never touched it, but of course there's eBPF.

How is Kognic using it?

Graphic Content Warning

How is Kognic using it?

Traces - Drilldown



Selected Trace

example-python-app: POST /send-message 343.86ms

2023-11-28 13:55:27.306 POST 200 /send-message

example-python-app POST /send-message (35.72ms) — 35.72ms

send: projects/[redacted]/topics/devplat.example.python (33.68ms) — 33.68ms

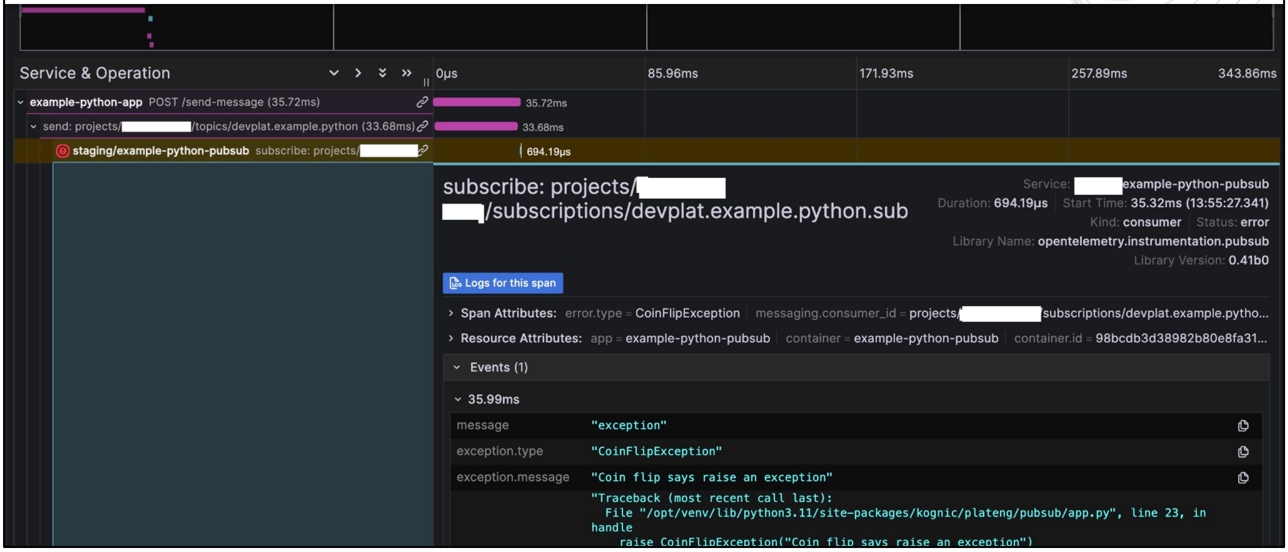
staging/example-python-pubsub subscribe: projects/[redacted]/subscriptions/devplat.example.python.sub (694.19μs) — 694.19μs

subscribe: projects/[redacted]/subscriptions/devplat.example.python.sub (384.87μs) — 384.87μs

- We can use grafana to search for traces as part of drilldown graphs
- But we can also generate metrics for RED observation from spans

How is Kognic using it?

Traces - Drilldown



Service & Operation | 0µs | 85.96ms | 171.93ms | 257.89ms | 343.86ms

example-python-app POST /send-message (35.72ms) — 35.72ms

send: projects/[redacted]/topics/devplat.example.python (33.68ms) — 33.68ms

staging/example-python-pubsub subscribe: projects/[redacted] — 694.19µs

subscribe: projects/[redacted]/subscriptions/devplat.example.python.sub

Service: [redacted]example-python-pubsub
Duration: 694.19µs Start Time: 35.32ms (13:55:27.341)
Kind: consumer Status: error
Library Name: opentelemetry.instrumentation.pubsub
Library Version: 0.41b0

Span Attributes: error.type = CoinFlipException messaging.consumer_id = projects/[redacted]subscriptions/devplat.example.pytho...

Resource Attributes: app = example-python-pubsub container = example-python-pubsub container.id = 98bcdb3d38982b80e8fa31...

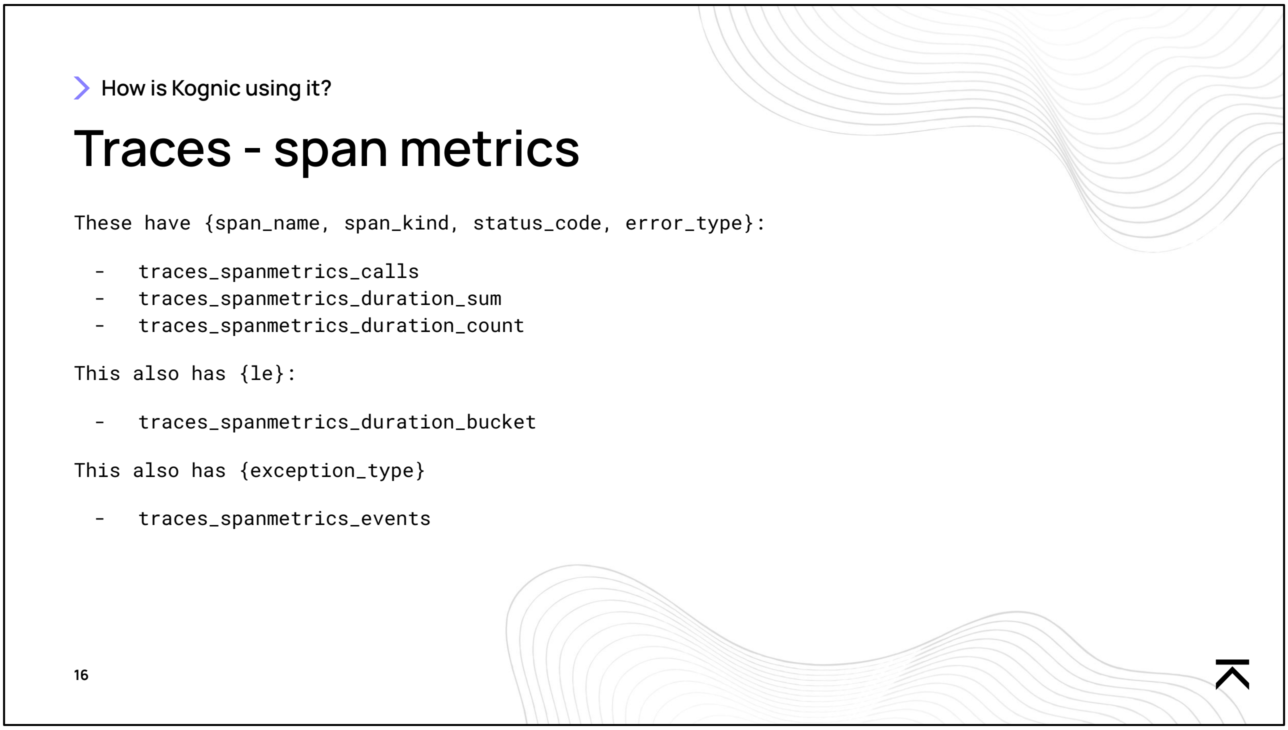
Events (1)

35.99ms

message	"exception"
exception.type	"CoinFlipException"
exception.message	"Coin flip says raise an exception"

```
"Traceback (most recent call last):
  File "/opt/venv/lib/python3.11/site-packages/kognic/plateng/pubsub/app.py", line 23, in
handle
    raise CoinFlipException("Coin flip says raise an exception")
```

- We can choose extra span attributes as metric labels



How is Kognic using it?

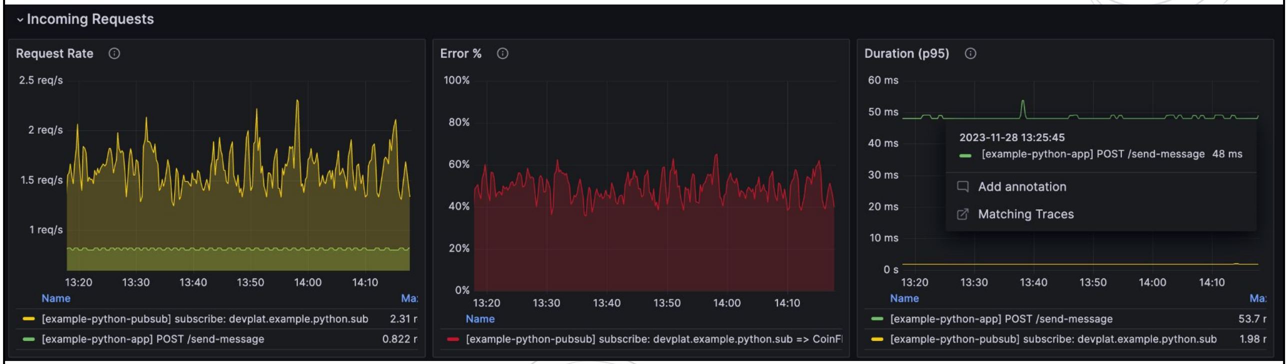
Traces - span metrics

```
These have {span_name, span_kind, status_code, error_type}:

  -   traces_spanmetrics_calls
  -   traces_spanmetrics_duration_sum
  -   traces_spanmetrics_duration_count

This also has {le}:

  -   traces_spanmetrics_duration_bucket

This also has {exception_type}

  -   traces_spanmetrics_events
```

- span_name: low-cardinality name of the request type: GET /books/{bookID}
- span_kind: server/client/producer/consumer/internal
- status_code: span status, not HTTP status. It’s ‘unset’ by default, or ‘error’ in some cases. Only user code can set it to ‘ok’
- error_type: set when status=error. low-cardinality name of the error

How is Kognic using it?

Traces - span metrics - RED graphs



Name	Ma:
[example-python-pubsub] subscribe: devplat.example.python.sub	2.31 r
[example-python-app] POST /send-message	0.822 r

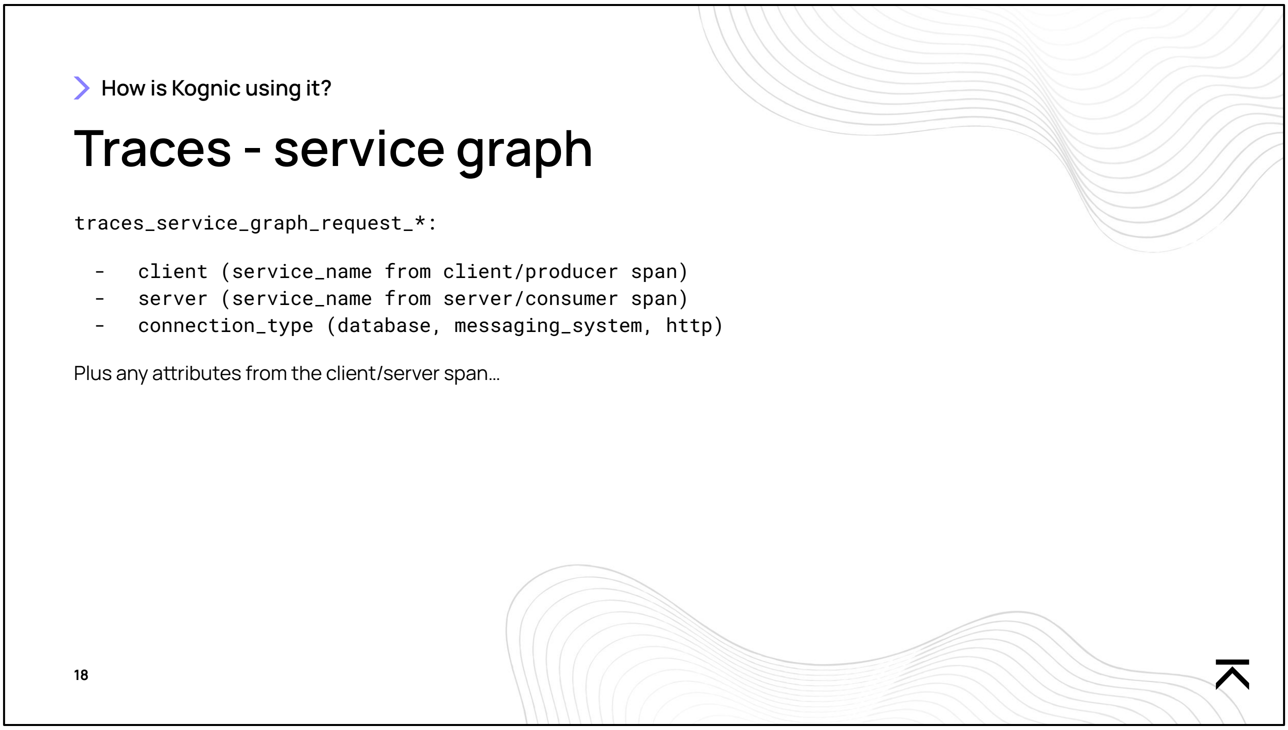
Name
[example-python-pubsub] subscribe: devplat.example.python.sub => CoinFl

2023-11-28 13:25:45
[example-python-app] POST /send-message 48 ms

Name	Ma:
[example-python-app] POST /send-message	53.7 r
[example-python-pubsub] subscribe: devplat.example.python.sub	1.98 r

The RED Method, from span metrics!
No matter what kind of span, we can use

- `span_name` to differentiate between requests
- `status_code` to tell if the request failed to be handled/sent
- `error_type` to group errors together
- `span_kind` to have different graphs for incoming, outgoing, and internal requests



How is Kognic using it?

Traces - service graph

```
traces_service_graph_request_*:

  -   client (service_name from client/producer span)
  -   server (service_name from server/consumer span)
  -   connection_type (database, messaging_system, http)
```

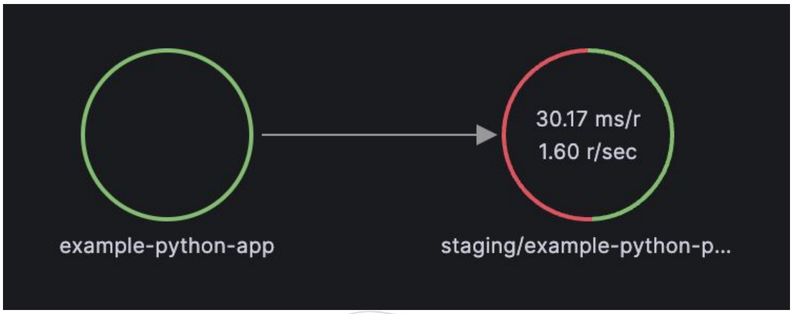
Plus any attributes from the client/server span…

From a trace, we can see when a client talks to a server, or a producer’s request is handled by a consumer.
We can generate metrics to represent these edges in a graph of services
We can also include any attributes from either side (e.g. client calls tend not to be templated, e.g. /books/123 - you could use these to grab the templated path from the server side, i.e. /books/{bookID}

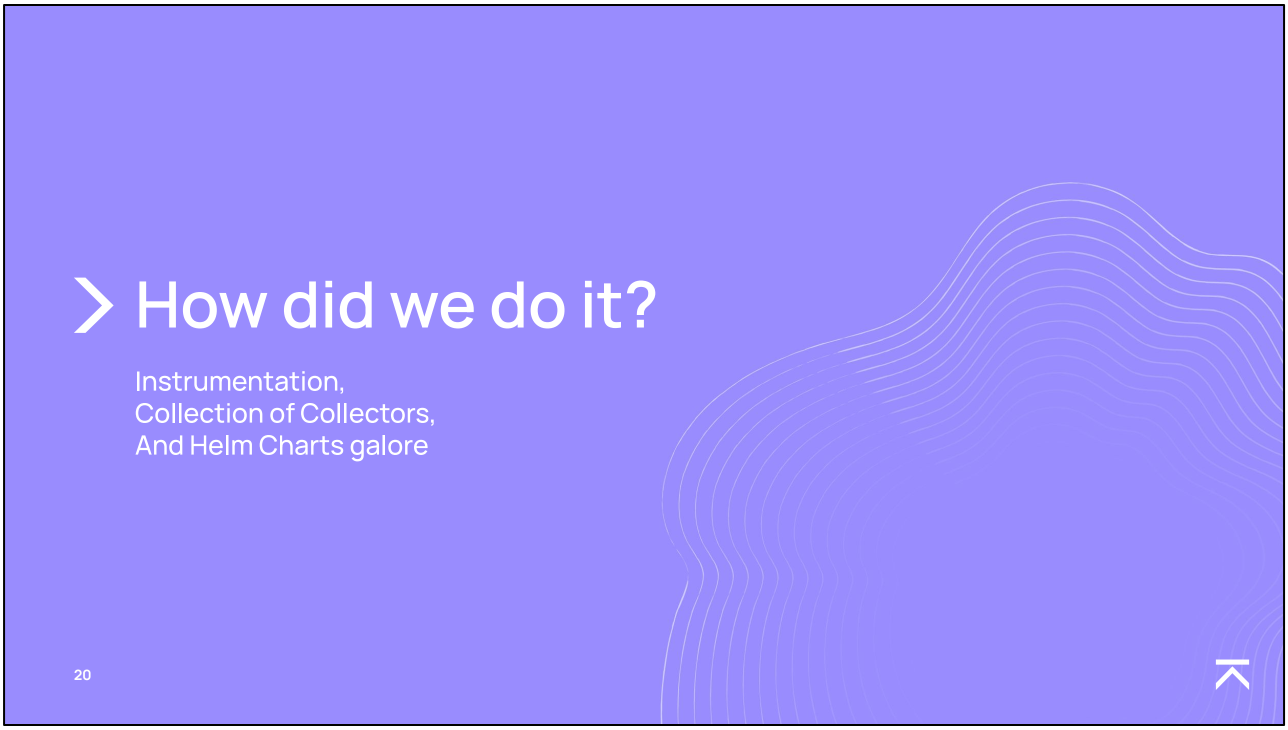
Are you ready to see the most amazing service graph?

How is Kognic using it?

Traces - span metrics - RED graphs



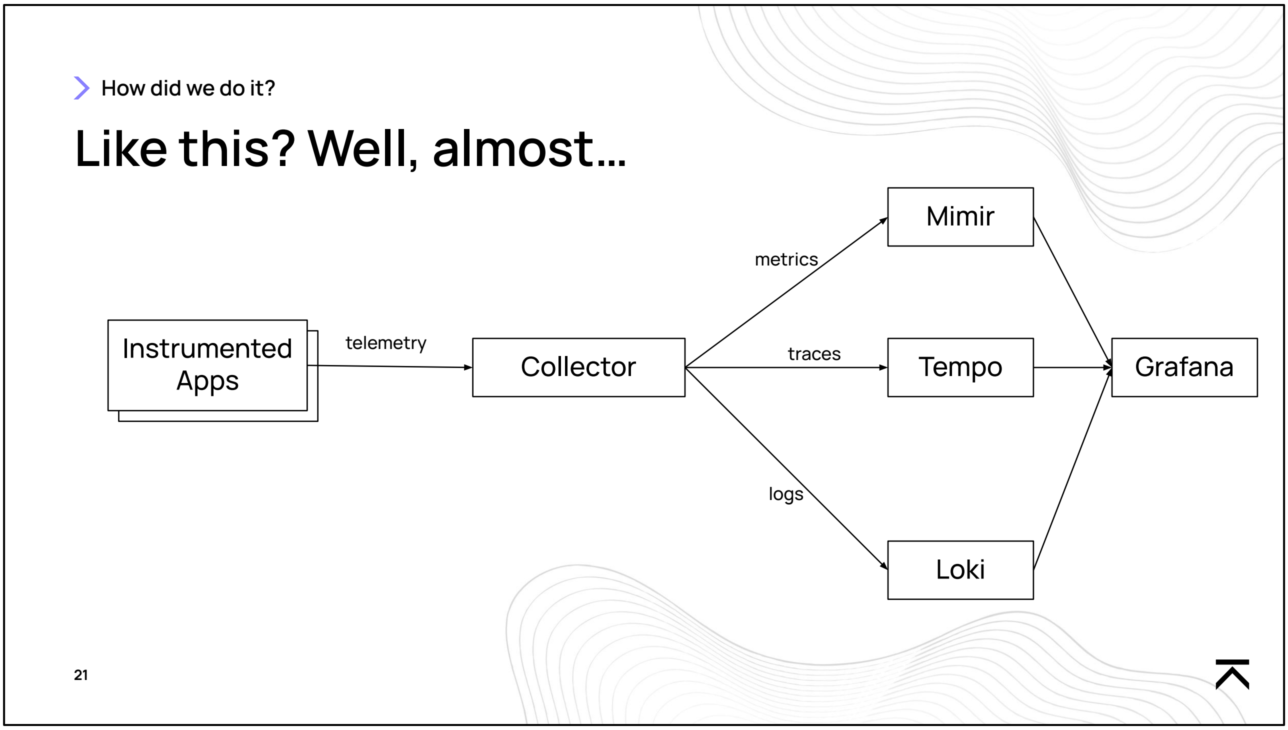
And here it is, in all it’s glory



How did we do it?

Instrumentation,
Collection of Collectors,
And Helm Charts galore

Extra points if you spotted that that is a haiku



How did we do it?

Like this? Well, almost…

Instrumented Apps

telemetry

Collector

metrics

traces

logs

Mimir

Tempo

Loki

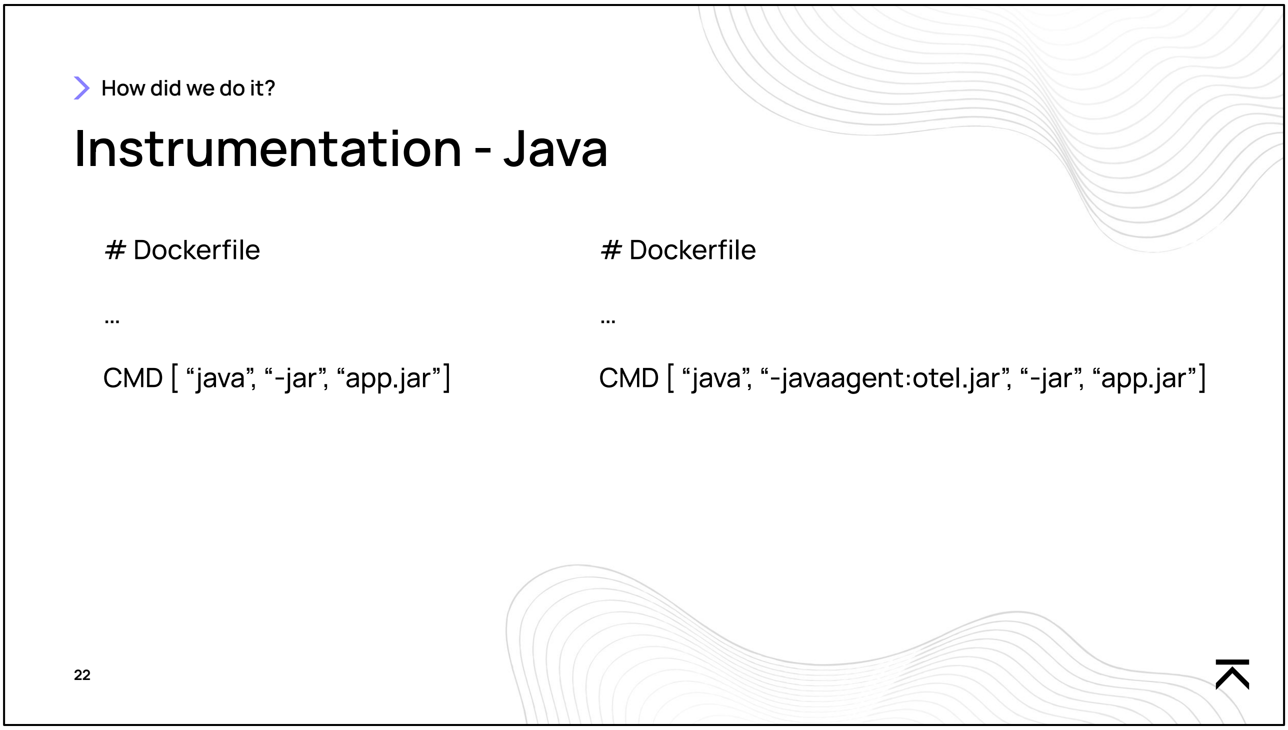
Grafana

Overall we want something like this. We need:

- to instrument the apps, and
- a collector to do processing (generate metrics from traces)

- Apps exporting metrics+traces over OTLP (which is a push format) don't need a collector, but having one means you can:
  - add k8s attributes
    - It watches k8s objects and records labels, annotations, various properties
  - apply arbitrary transformations once-and-for-all
    - To make k8s mixin graphs work, we rename `k8s.xxx.name` into `xxx`.
    - Drop spans with `kube-probe` agent (e.g. readiness probe)
  - generate spanmetrics:
  - generate servicegraph metrics:
  - apply tail-sampling to traces
    - Head-based sampling (apps choosing to only record 10% of spans) is tricky to manage
    - Instead, app send all spans to the collector, they're combined into complete traces, which can be filtered all together:
      - Keep traces with errors
      - Keep extra-long traces

- Keep 10% of the other traces
- then we can use 100% of the traces for metrics generation
- This collector should be able to handle the backend being unavailable



How did we do it?

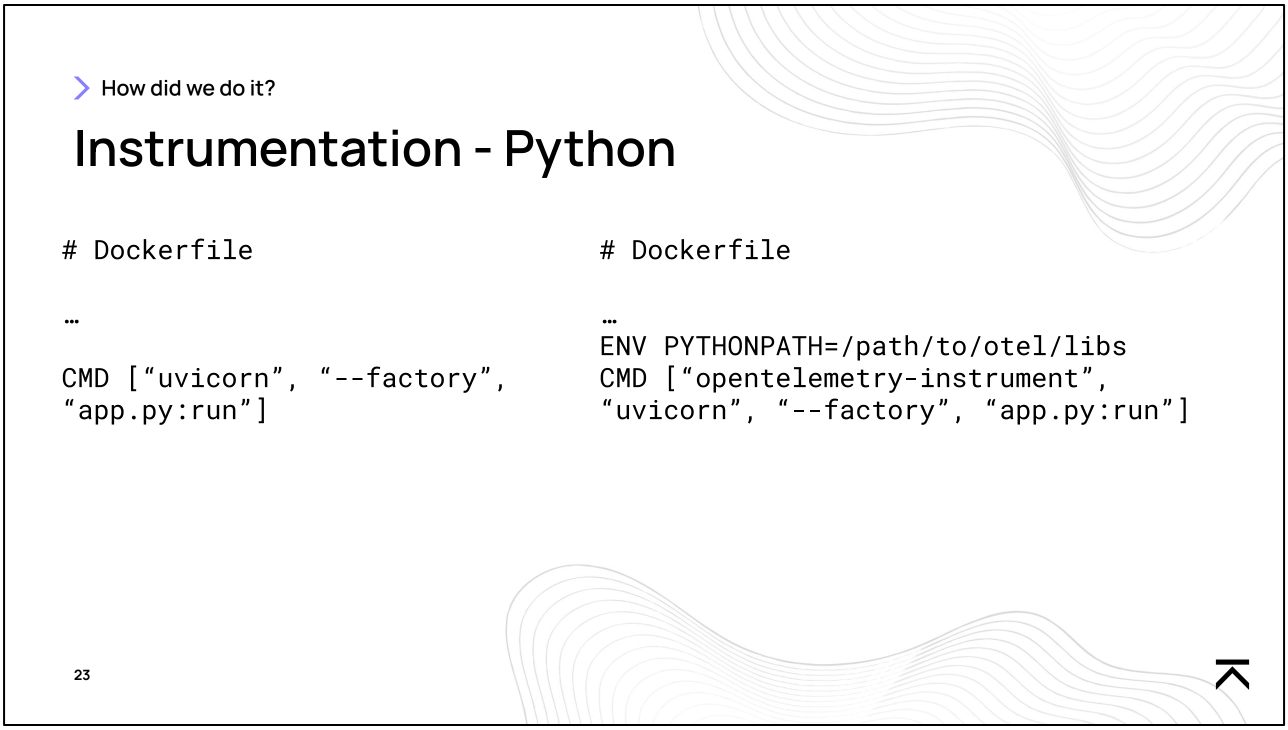
Instrumentation - Java

```
# Dockerfile

...

CMD [ “java”, “-jar”, “app.jar”]
```

```
# Dockerfile

...

CMD [ “java”, “-javaagent:otel.jar”, “-jar”, “app.jar”]
```

To instrument java apps, we need to add a big jar containing all the otel SDK and instrumentation as a javaagent, which

- checks what is available on the classpath,
- selectively copies the right instrumentation over to the main classloader
- performs the bytecode-manipulation



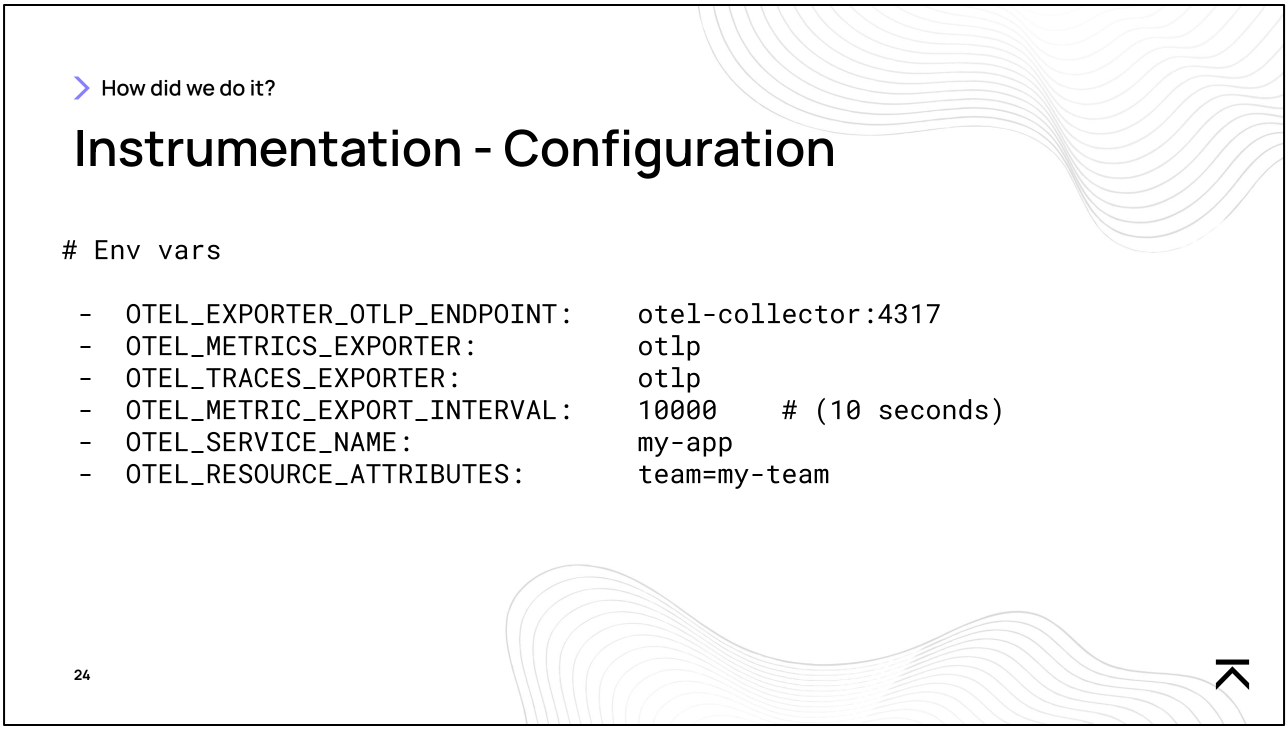
How did we do it?

Instrumentation - Python

```
# Dockerfile

…

CMD [“uvicorn”, “--factory”,
“app.py:run”]
```

```
# Dockerfile

…
ENV PYTHONPATH=/path/to/otel/libs
CMD [“opentelemetry-instrument”,
“uvicorn”, “--factory”, “app.py:run”]
```

Python is a bit simpler - the instrumentation libraries just need to be on the PYTHONPATH,
then you call your app with `opentelemetry-instrument` and it triggers the monkey-patching



How did we do it?

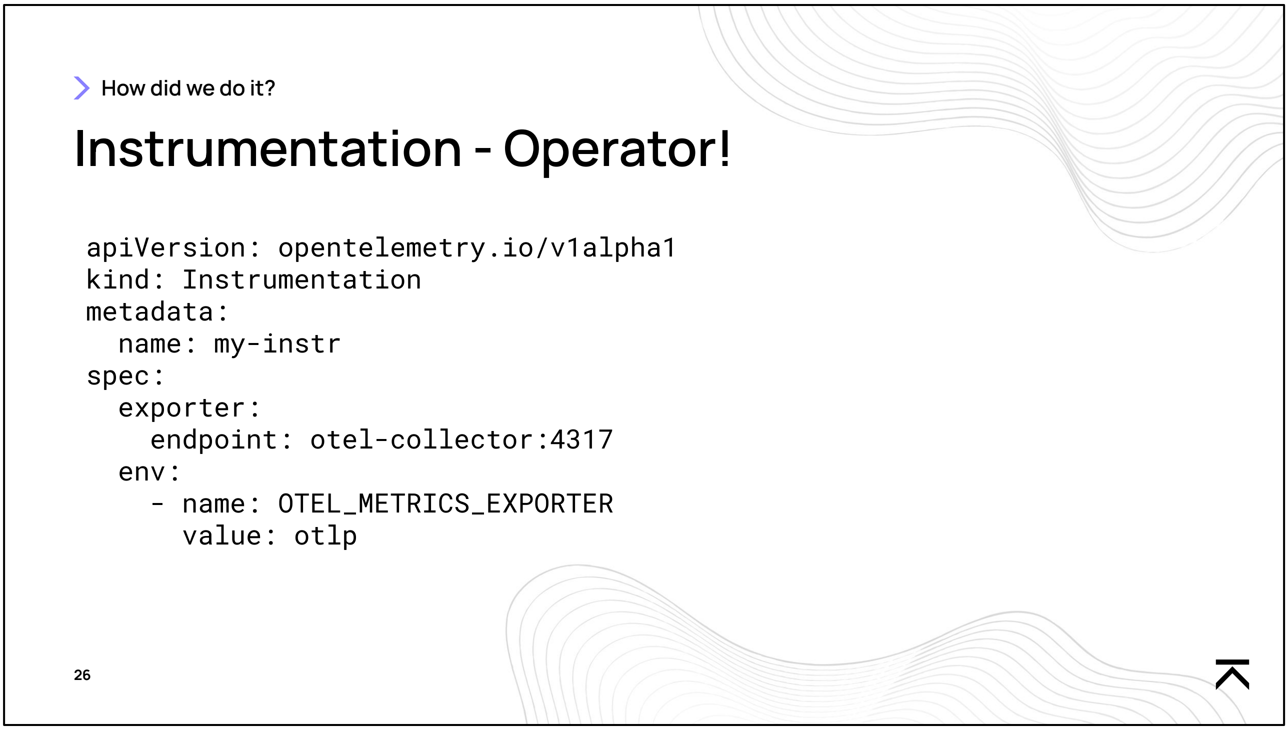
Instrumentation - Configuration

```
# Env vars

 -  OTEL_EXPORTER_OTLP_ENDPOINT:    otel-collector:4317
 -  OTEL_METRICS_EXPORTER:          otlp
 -  OTEL_TRACES_EXPORTER:           otlp
 -  OTEL_METRIC_EXPORT_INTERVAL:    10000    # (10 seconds)
 -  OTEL_SERVICE_NAME:              my-app
 -  OTEL_RESOURCE_ATTRIBUTES:       team=my-team
```

Then you also need to set up the SDK, probably using env vars

How did we do it?

There’s a quicker way...

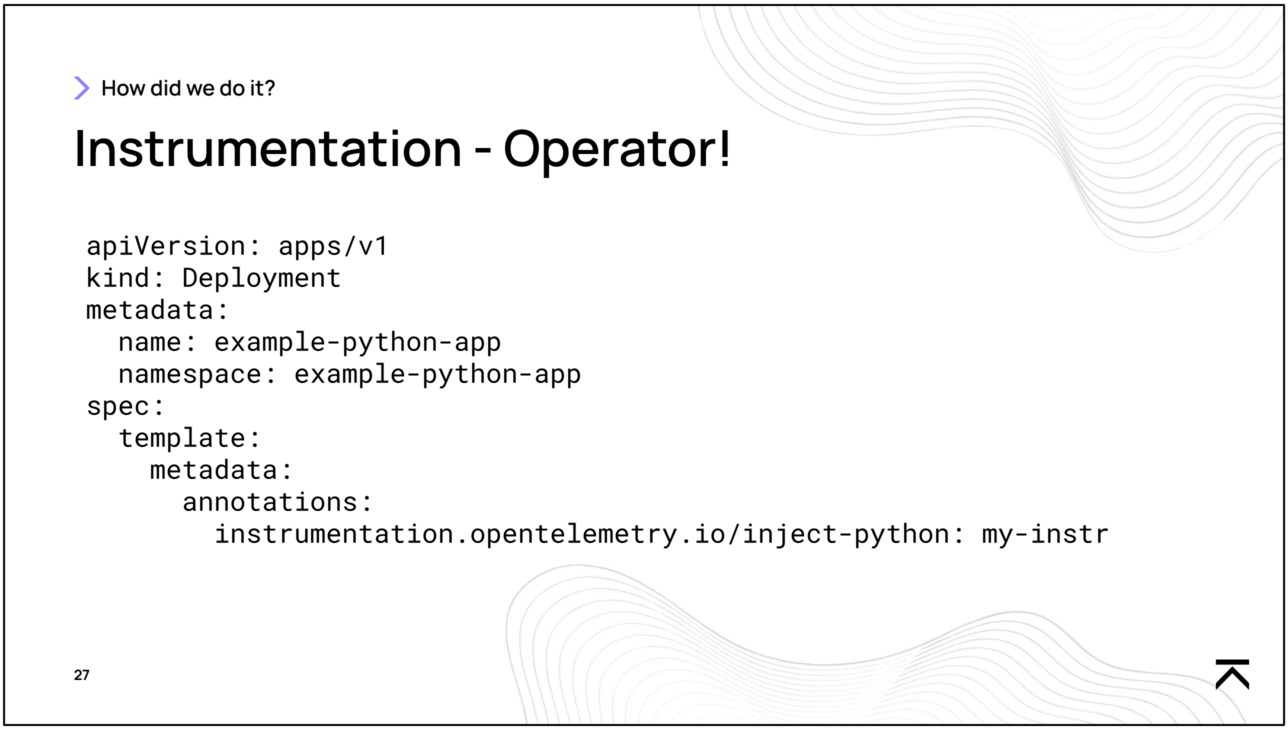


How did we do it?

Instrumentation - Operator!

```
apiVersion: opentelemetry.io/v1alpha1
kind: Instrumentation
metadata:
  name: my-instr
spec:
  exporter:
    endpoint: otel-collector:4317
  env:
    - name: OTEL_METRICS_EXPORTER
      value: otlp
```

You can create an Instrumentation CR in kubernetes, and …

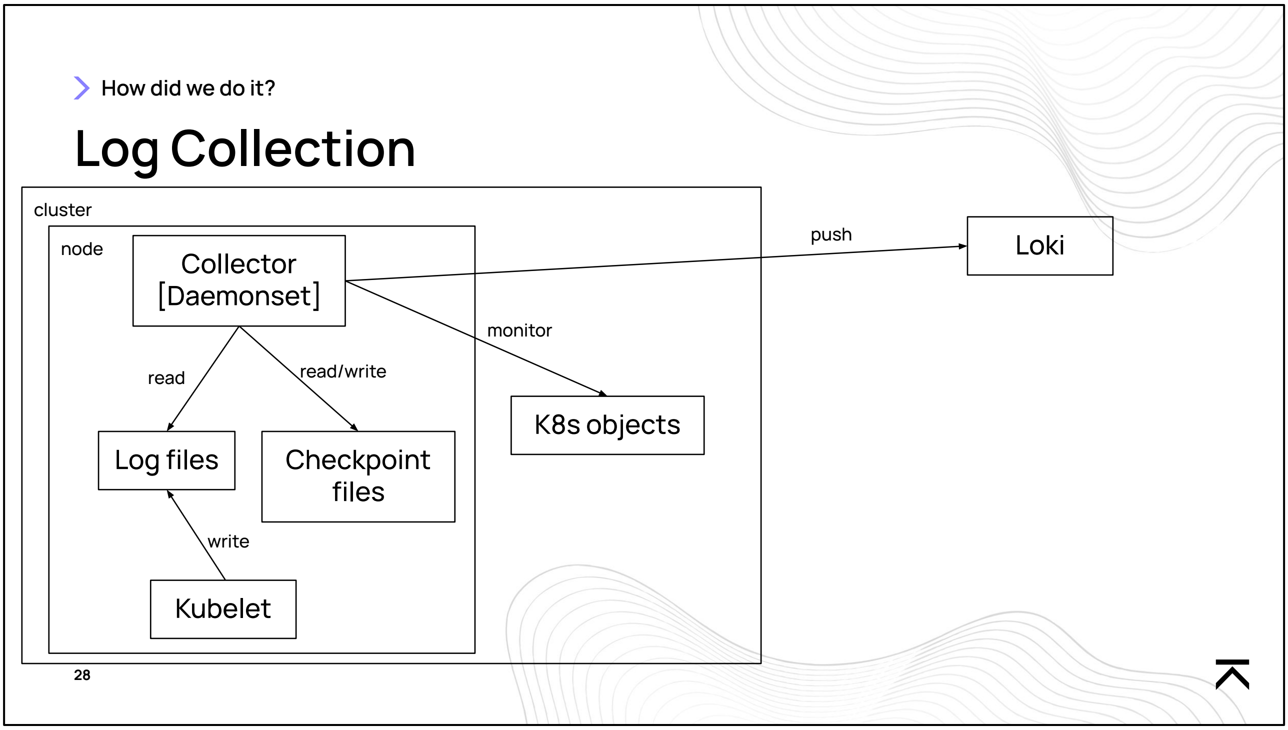


How did we do it?

Instrumentation - Operator!

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-python-app
  namespace: example-python-app
spec:
  template:
    metadata:
      annotations:
        instrumentation.opentelemetry.io/inject-python: my-instr
```

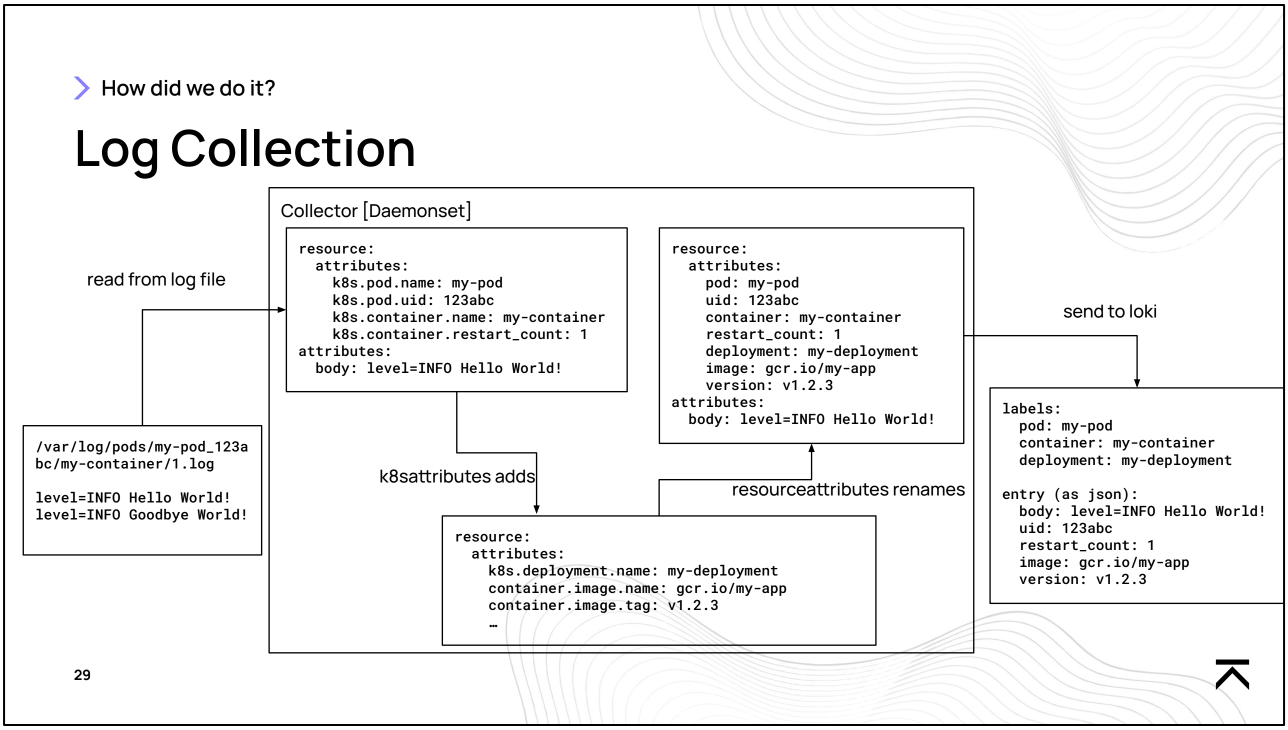
Annotate your pod, pointing to the instrumentation CR. It does all of those manual steps for you.
That’s what we’re using atm for a few non-critical apps



How did we do it?

Log Collection

- We’re using a daemonset to collect logs from k8s-provided pod log files:
    - Tried-and-tested
    - No real benefit in exporting them from the instrumented apps directly (not all languages are supported anyway)
    - When doing this with helm charts, make sure to enable checkpointing, otherwise restarting the pod means lost or duplicated logs
- note that its watching the k8s objects too - it’s keeping an up to date map of relevant names/labels/annotations/properties of pods, nodes, deployments, etc



How did we do it?

Log Collection

read from log file

```
/var/log/pods/my-pod_123a
bc/my-container/1.log

level=INFO Hello World!
level=INFO Goodbye World!
```

Collector [Daemonset]

```
resource:
  attributes:
    k8s.pod.name: my-pod
    k8s.pod.uid: 123abc
    k8s.container.name: my-container
    k8s.container.restart_count: 1
attributes:
  body: level=INFO Hello World!
```

k8sattributes adds

```
resource:
  attributes:
    k8s.deployment.name: my-deployment
    container.image.name: gcr.io/my-app
    container.image.tag: v1.2.3
    …
```

resourceattributes renames

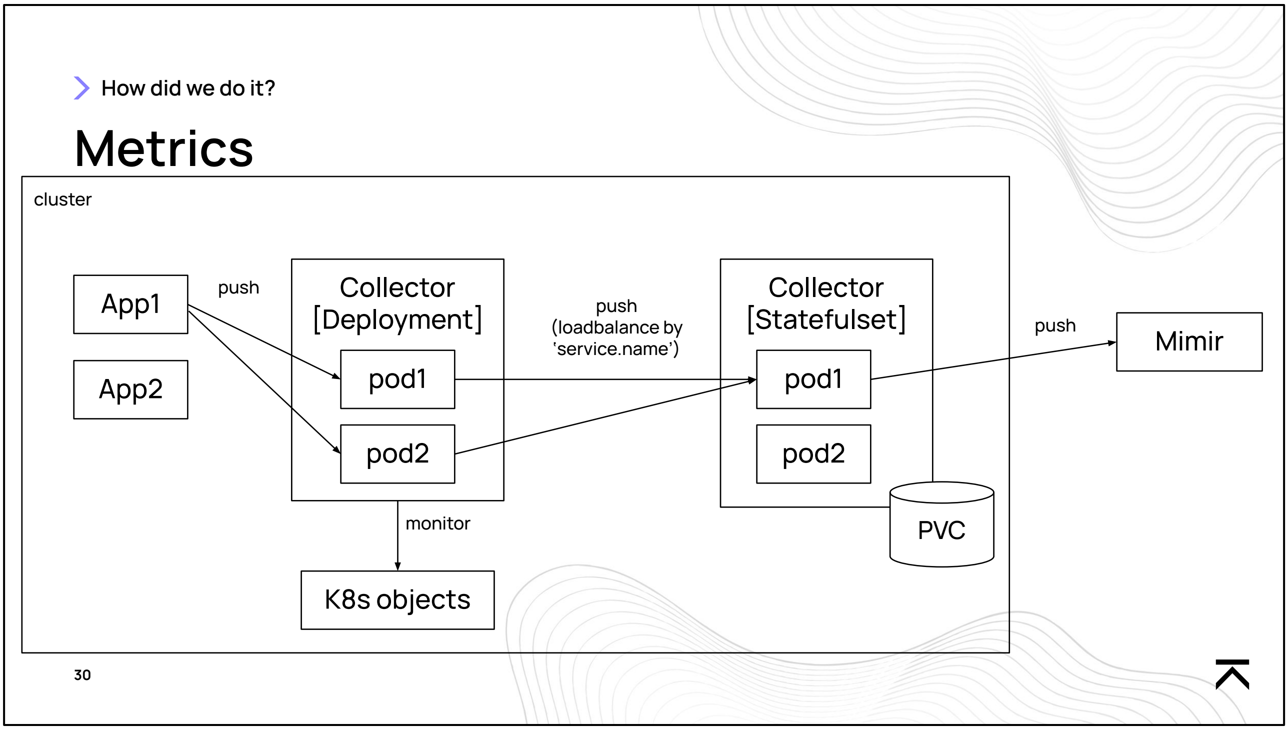
```
resource:
  attributes:
    pod: my-pod
    uid: 123abc
    container: my-container
    restart_count: 1
    deployment: my-deployment
    image: gcr.io/my-app
    version: v1.2.3
attributes:
  body: level=INFO Hello World!
```

send to loki

```
labels:
  pod: my-pod
  container: my-container
  deployment: my-deployment

entry (as json):
  body: level=INFO Hello World!
  uid: 123abc
  restart_count: 1
  image: gcr.io/my-app
  version: v1.2.3
```

And inside the collector we can set up a pipeline, where we:

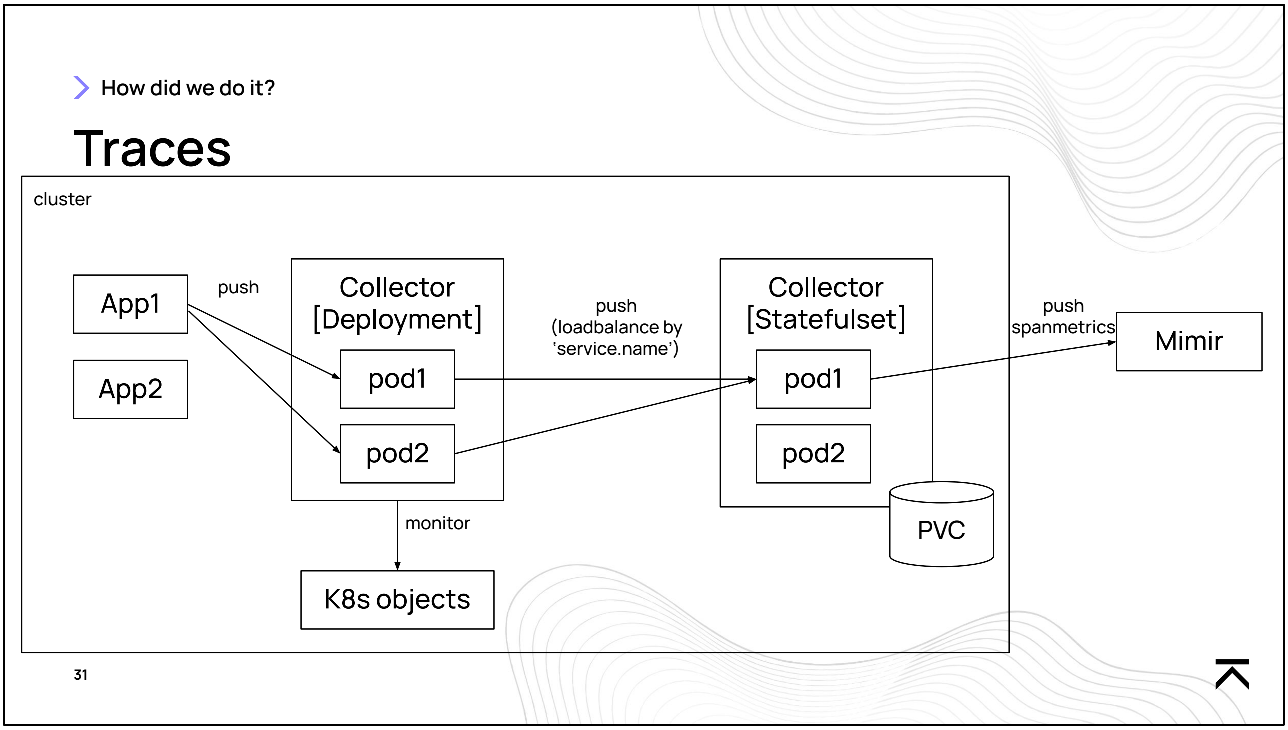
- Read the files from disk
- Correlate against the k8s information to add more attributes
- rename the attributes to match what our k8s mixins expect
- send to loki, choosing which attributes will be chosen as “labels” (like in prometheus)



How did we do it?

Metrics

- We need multiple instances for HA, but that means:
  - An app might send its metrics to multiple collectors, which then send similar metrics to the backend (Mimir for us), which can complain (out-of-order-samples, non-identical-metric-copies)
- So we loadbalance, using ‘service.name’ as a routing key. That means one service’s metrics always goes to the same backend collector
- The PVC is there for the Write-Ahead Log (WAL) so if the backend goes down we have some time to recover without data loss



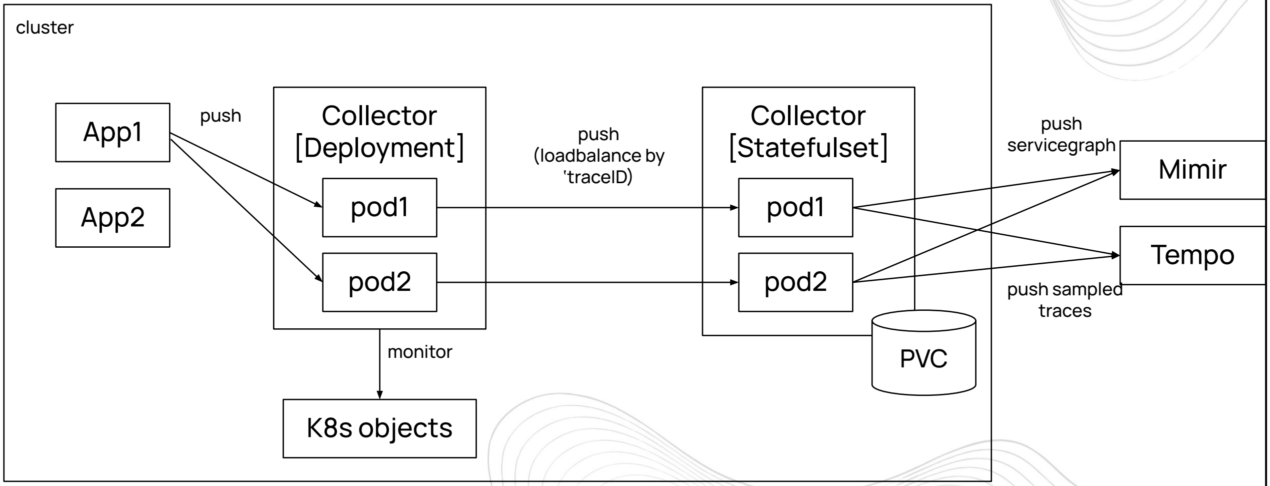
How did we do it?

Traces

Same here, spanmetrics for a single app should be generated on a single collector to avoid issues

How did we do it?

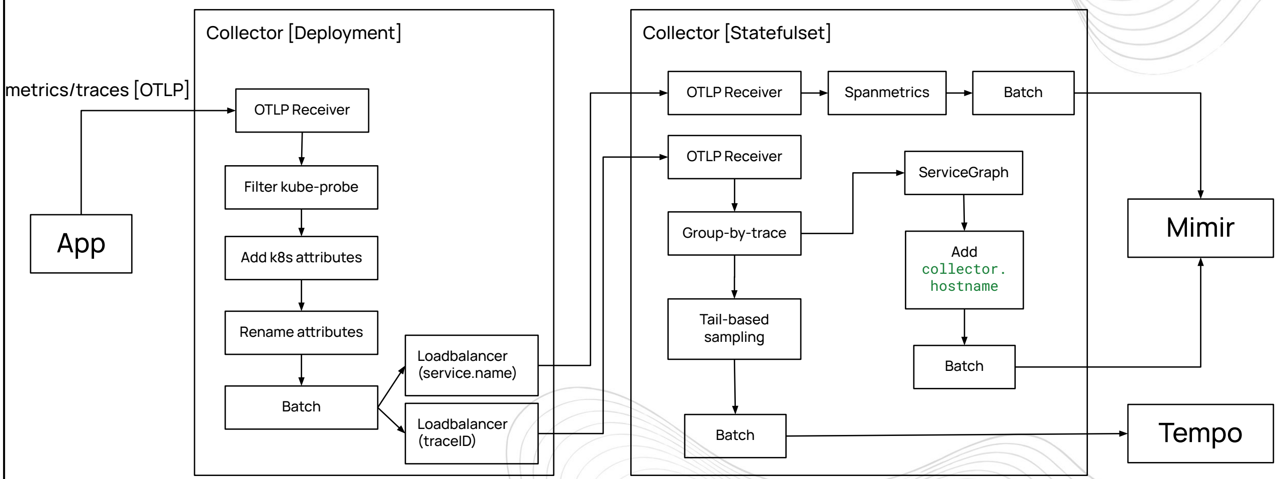
Traces



- Spans for the same trace can end up on different instances, so processors requiring complete traces (tail-based sampling, service graph) won’t work
- so we need to also loadbalance, routing by traceID

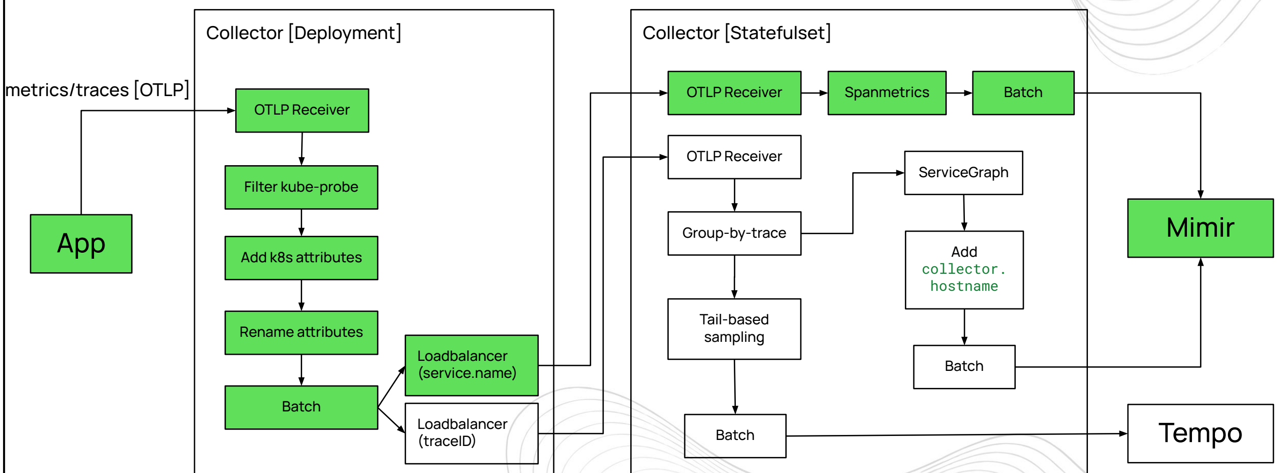
How did we do it?

Metrics + Traces



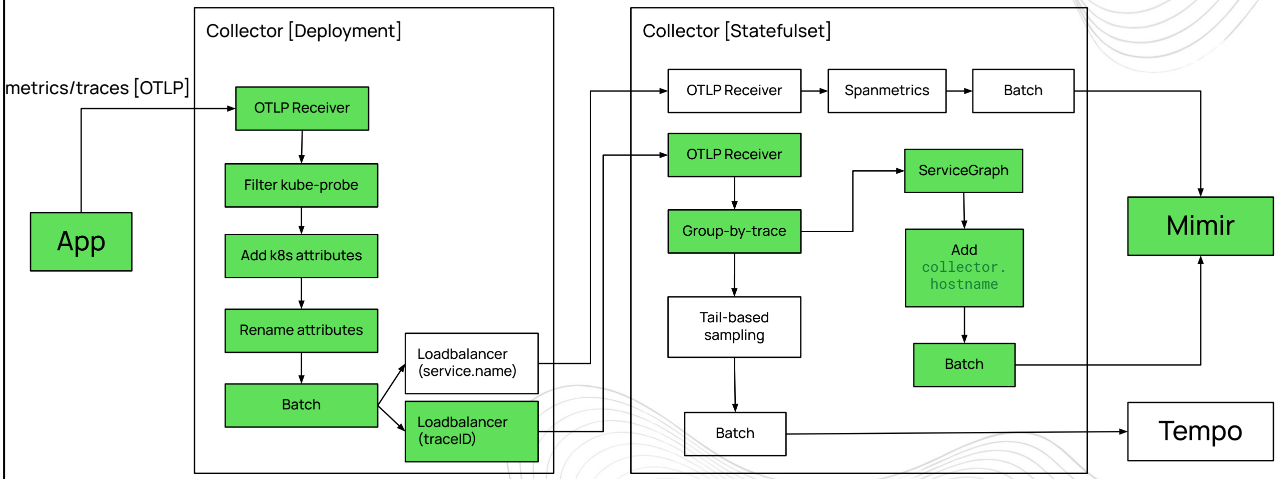
How did we do it?

Metrics + Traces



How did we do it?

Metrics + Traces



metrics/traces [OTLP]

App

Collector [Deployment]

OTLP Receiver

Filter kube-probe

Add k8s attributes

Rename attributes

Batch

Loadbalancer (service.name)

Loadbalancer (traceID)

Collector [Statefulset]

OTLP Receiver

Spanmetrics

Batch

OTLP Receiver

Group-by-trace

Tail-based sampling

Batch

ServiceGraph

Add `collector.hostname`

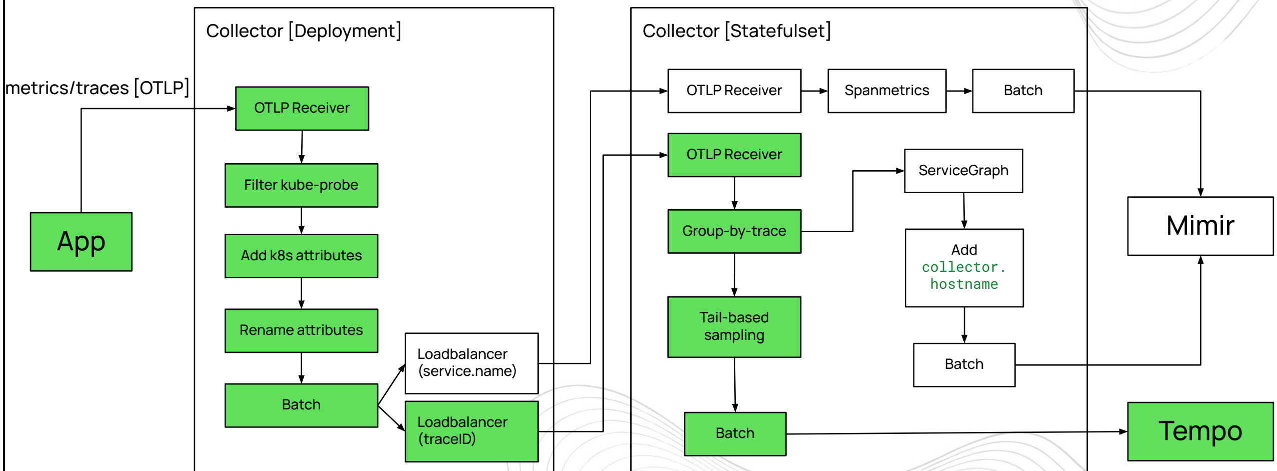
Batch

Mimir

Tempo

How did we do it?

Metrics + Traces



Our Experiences

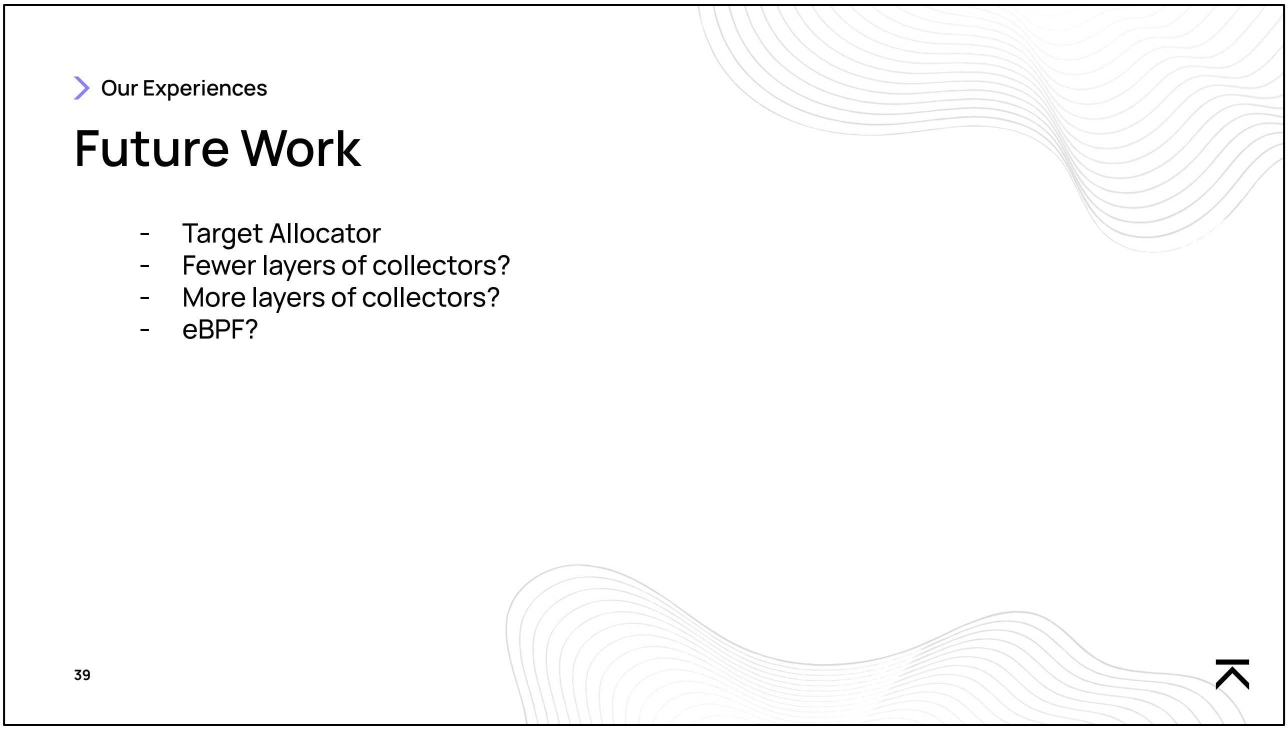
Our Experiences

So far…

- Load Balancing
- Instrumentation Operator
- Collector stability
- Tail-Based Sampling
- Community
- Span Metrics vs Server/Client Metrics

- Load balancing:
  - Wasn’t obvious how to set all the parts up from docs, like when to use traceID routing
- Instrumentation Operator:
  - K8s operator doesn't fit our way of working, so we are moving to instrumenting at Build time instead.
  - Easy to do
  - Safer for devs to own the integration and testing
  - Good idea to create a “distribution” where you can add little bits of glue code and default values
- Collector stability: Collectors have been rock solid, except for once when memory usage went up to 10GB. We needed to configure the memory settings correctly to avoid that.
- Community:
  - Getting attention upstream has been easy, and they have regular public calls, which is good. It's still worth setting up distributions for each language as a place to put little glue code (getting container name from cgroups file).
- Tail-based sampling:
  - Great that way can choose only a subset of the spans for storage.

    - Annoying thing is that we can’t use exemplars
    - which are links from a graph to a specific trace
- Span Metrics vs Server/Client Metrics:
    - Spanmetrics are great! I prefer them to http_client_, http_server_, ... because:
        - it covers http server/client, pubsub producer/consumer, and "internal" spans in the same format
        - in code you can keep adding attributes to the current span, which can then become labels on the spanmetrics. That's harder to do with metrics if they're generated in library code.

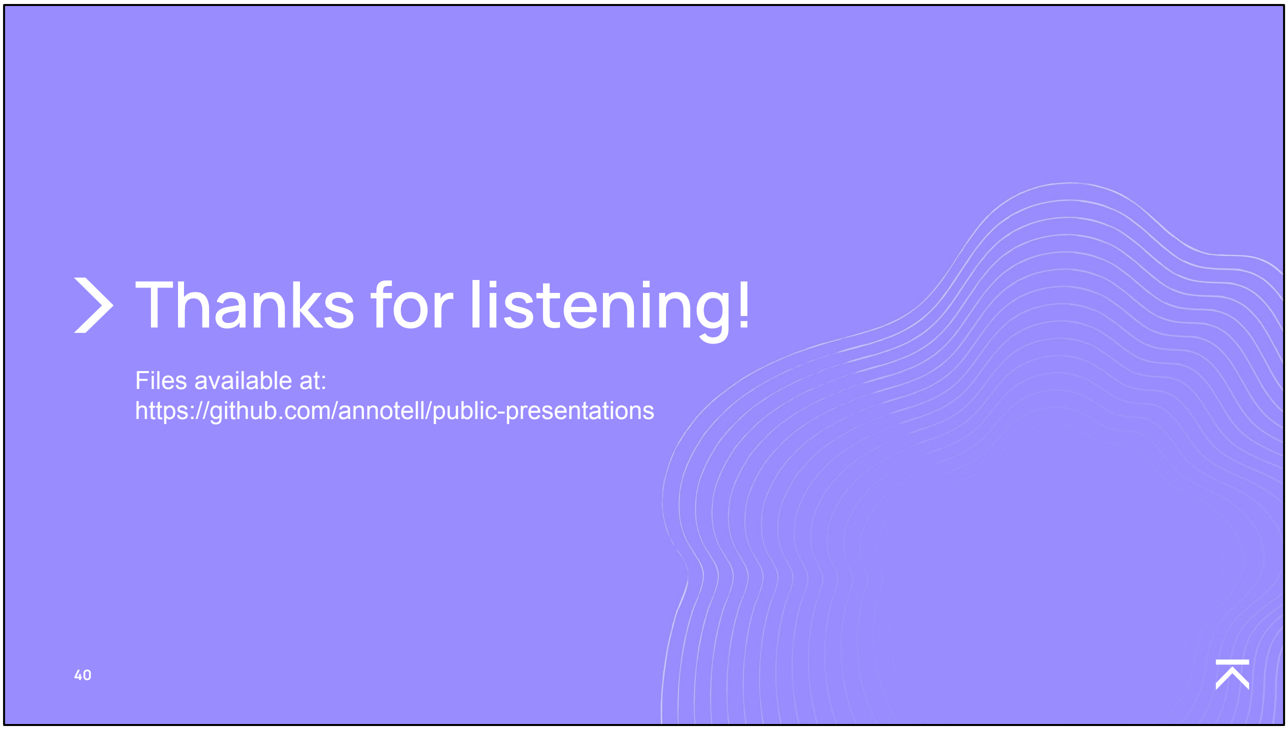


Our Experiences

Future Work

- Target Allocator
- Fewer layers of collectors?
- More layers of collectors?
- eBPF?

- Target Allocator
  - Currently we have multiple prometheus agents polling everything multiple times, pushing metrics multiple to the backend for it to then deduplicate it
  - Instead, we can deploy a Target Allocator that assigns polling targets to the available collectors running.
- Fewer layers of collectors?
  - We could have all the LB pipelines in the collector and have it talk to itself over different ports
- More layers of collectors?
  - Maybe it’s better to split out those pipelines for easier scaling/debugging when something goes wrong?
- eBPF?
  - no time soon, but cool idea



Thanks for listening!

Files available at:
https://github.com/annotell/public-presentations