intro to tracing-based SLOs

Shelby Spees
Site Reliability Engineer
Equinix

@shelbyspees at #SLOconf

hi I'm Shelby
I'm an SRE at Equinix
and this is an introduction to tracing-based SLOs

why tracing?

@shelbyspees at #SLOconf

tracing is great because it can capture so much data about the state of your service within the scope of each request.

use OpenTelemetry

auto-instrumentation for HTTP and gRPC

— request duration
— status codes
— client calls vs. server responses

@shelbyspees at #SLOconf

using HTTP or gRPC instrumentation like with OpenTelemetry means our traces automatically track things like request duration and status code
so out of the box we can already measure latency and error rate.

opentelemetry also automatically differentiates between client calls and server-side responses, which helps a lot when we want to look at different kinds of workloads

traces are fancy structured logs


```
{
    "Timestamp": "2022-04-15T18:24:32.987575281Z",
    "duration_ms": 42.076192,
    "http.method": "GET",
    "http.scheme": "https",
    "http.status_code": 200,
    "http.host": "api.awesome-service.net",
    "http.target": "/all-the-things",
    "service.name": "awesome-service",
    "trace.trace_id": "11d8692d1cb9d55436c1a656999e0608",
    "trace.span_id": "4f87aa2321768f18",
    "trace.parent_id": "3a7caf3aaba06a5f"
}
```


@shelbyspees at #SLOconf

while tracing libraries do a lot of work to keep track of trace state, the actual data they generate is basically just structured logs that have a couple special fields to connect them.

BUT what those connections give us is the ability to see the **relationship** between events. that makes a huge difference when we're debugging.

observability == exploration

→ broad view: what's slow?
→ zoom in: what's different about this slow traffic?
→ zoom out: is this kind of traffic always slow?
→ zoom in...

@shelbyspees at #SLOconf

we get the most benefit from tracing when using modern observability tools that accept our raw trace data and allow us to explore it interactively across lots of high-cardinality dimensions.

that ability to query on raw trace data means we don’t have to decide up front what math to do on what dimensions. instead, our observability tools do the math on the fly at query time.

tracing-based SLOs

@shelbyspees at #SLOconf

of course we're not redefining our SLOs on the fly, but if your tools support querying across lots of arbitrary dimensions then they can also support defining an SLI based on equally arbitrary dimensions.

this is a big deal. while latency and error rate are important for pretty much every service, the most important thing for your service, your key differentiator? it's going to be unique to that service.
that means you might not be able to measure it with auto-instrumented trace data.

instrument your code

```
func (m Action) BootDeviceSet(ctx context.Context, ...) (result string, err error) {
	tracer := otel.Tracer("pbnj")
	ctx, span := tracer.Start(ctx, "client.SetBootDevice", trace.WithAttributes(
		attribute.String("bmc.device", device),
		attribute.Bool("bmc.persistent", persistent),
		attribute.Bool("bmc.efiBoot", efiBoot),
	))
	defer span.End()
	// . ˚ ₒ . ˚ ✯ . ˚ ☆ . ˚  . ₒ .
	//    ✧ set boot device ✧
	// . ˚ ₒ . ˚ ☆ . ˚ ✯ . ˚  . ₒ .
}
```

@shelbyspees at #SLOconf

so, it’s important to instrument your code.
what’s nice about tracing is that the data from custom instrumentation gets interwoven with existing auto-instrumented traces.
rather than sending a bunch of disparate data points, we’re enriching our existing traces with more context.
defining SLOs from trace data means that when we add custom instrumentation for measuring the most important parts of our service, it's also there the rest of the time to help us with debugging
dual-purpose data!

defining a tracing-based SLI

— filter to what’s relevant

— define a condition for "good"

@shelbyspees at #SLOconf

to define our SLI we need to do two things
filter to what's relevant and define a condition for "good"
the same as any other kind of SLI
when we're talking about tracing-based SLIs, different tools have different ways of defining your filters and conditions
so in my slides I'm just using pseudocode

SLI: overall traffic

@shelbyspees at #SLOconf

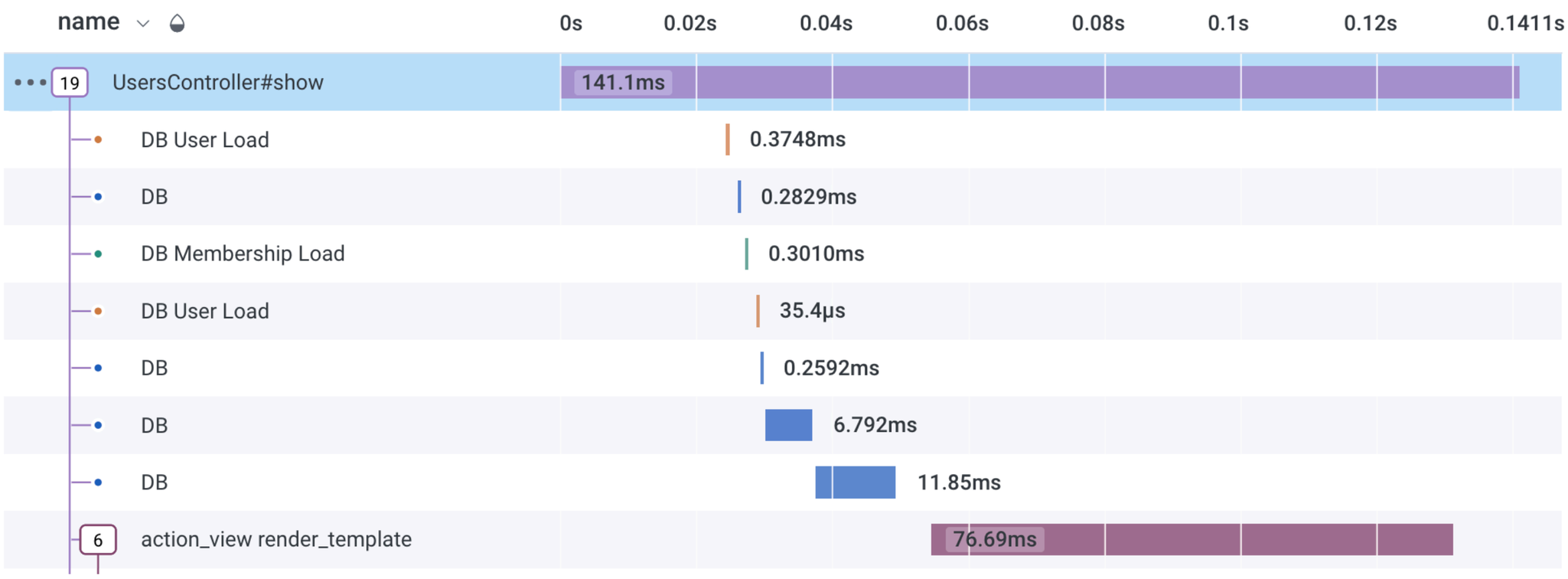
first let's define an SLI for latency and error rate on overall traffic to our API service

filter to what's relevant

```
IF(
    // root spans
    trace.parent_id == undefined
          AND
    // responding to client calls
    span.kind == "server"
)
```


we want to filter to spans that best represent the end-user experience.
for overall traffic to our API service we can look at server spans at the root of traces

root span of the trace



@shelbyspees at #SLOconf

here’s an example of the kind of span we’re looking at
the root span of a trace tells us the duration for synchronous requests
those are the ones where there's a human at the other end waiting for a page to load, or a software service waiting for an API response

filter to what's relevant

```
IF(
        trace.parent_id == undefined
                AND
        span.kind == "server"
                AND
        // filter out employee traffic
        user.staff != true
                AND
                // filter out internal bot traffic
                user.bot != true
    )
```

@shelbyspees at #SLOconf

we also want to make sure we're getting an accurate representation of the customer experience, so let's filter out requests from staff users and internal bot users.
these user fields come from custom instrumentation.

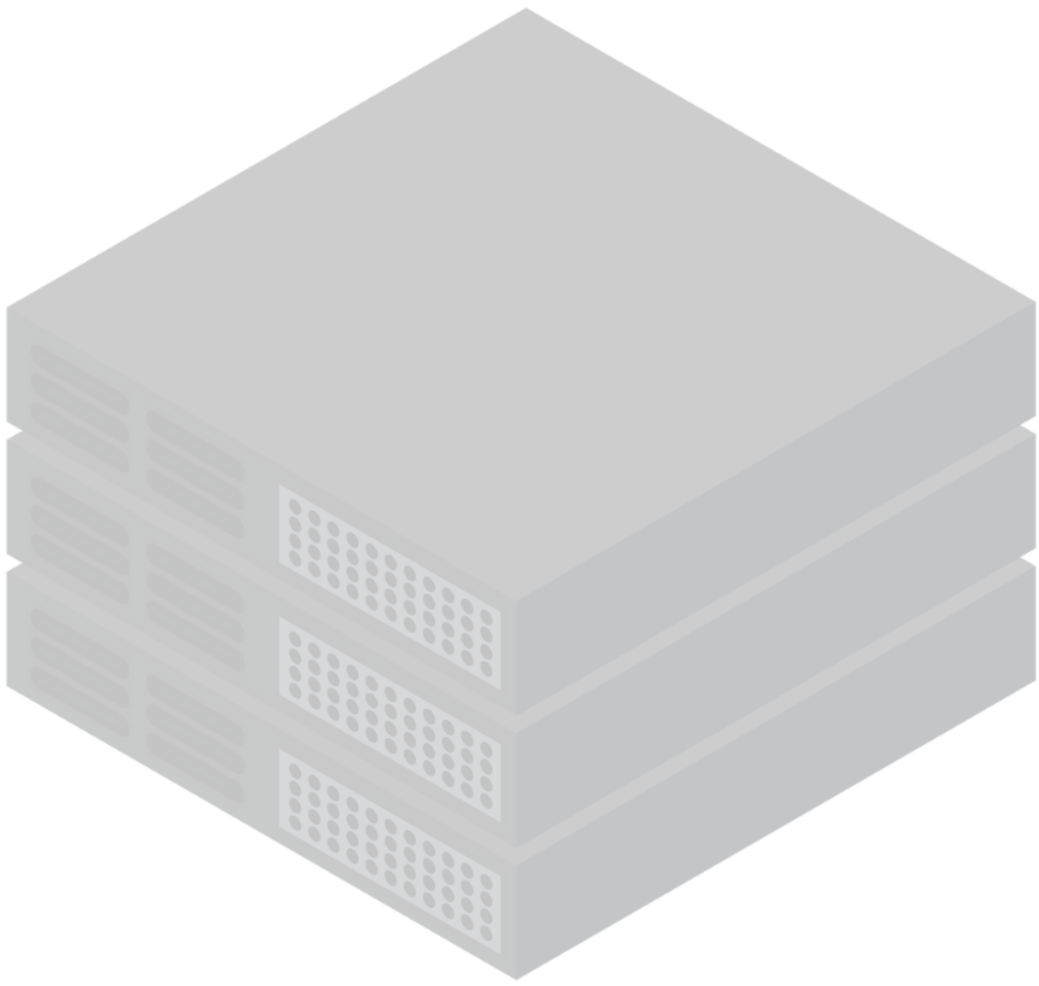
define a condition for “good”

```
IF(
        // should not return a server error
        http.status_code < 500
                AND
        // should return in less than 1s
        duration_ms < 1000
)
```

@shelbyspees at #SLOconf

now we define our condition for good
remember that OpenTelemetry’s auto-instrumentation automatically captures the request duration and whether it was successful.
so you can add OpenTelemetry today and define an SLI just like this one

SLI: provisioning



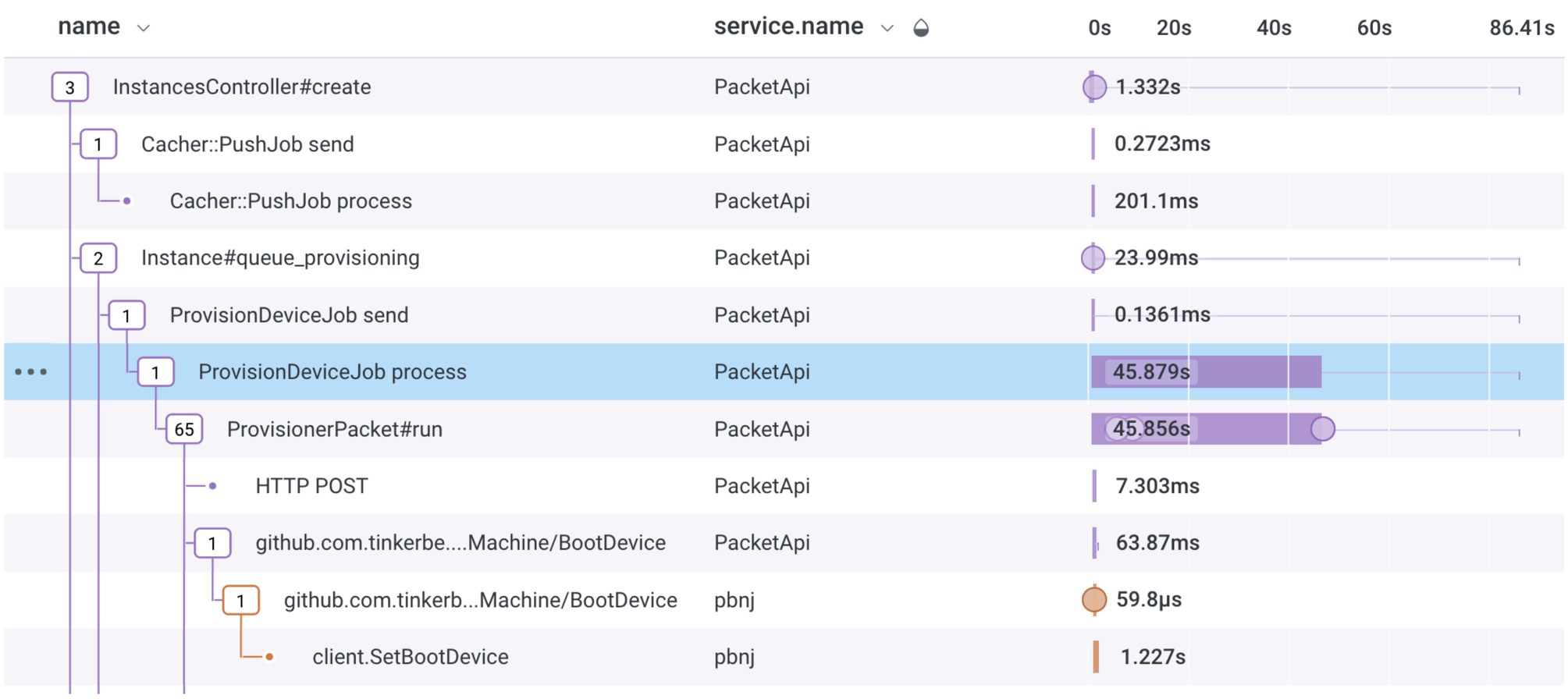
provisioning is our bread and butter at equinix metal
so we've been doing a lot of work to instrument and measure the bare metal provisioning process

filter to what's relevant

```
IF(
        // just the provision job
        name == "ProvisionDeviceJob process"
)
```

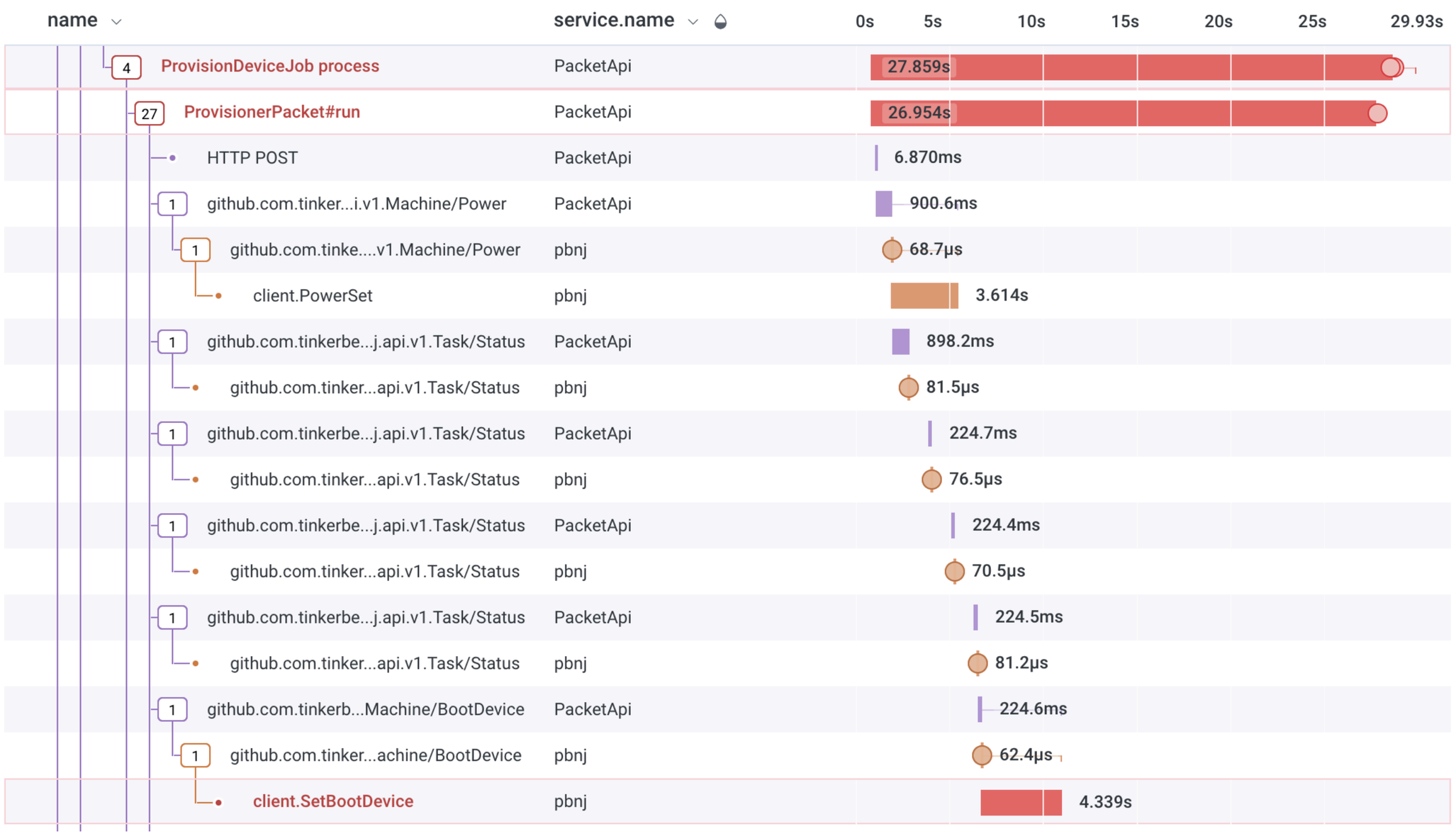

our provisions are orchestrated by a background job called ProvisionDeviceJob

ProvisionDeviceJob



@shelbyspees at #SLOconf

here's what that looks like in a trace.
this ProvisionDeviceJob is our best proxy for the end-to-end provisioning process



@shelbyspees at #SLOconf

tracing our provisioning process is a huge help for debugging if ProvisionDeviceJob fails, we can go look at the trace to see what step had the error

filter to what’s relevant

```
IF(
        // just the provision job
        name == "ProvisionDeviceJob process"
                AND
        // filter out internal test projects
        project.test != true
)
```

@shelbyspees at #SLOconf

for our SLI, we also have the ability to filter out test projects, so let’s do that.
if one of our devs is enthusiastically reproducing a provisioning error, I don't want that to wake up whoever's on-call.
the test provisions are still generating traces. the data is there for our dev to use when debugging. we're not changing anything at write-time
we're just filtering out those test provisions for the SLO query

define a condition for “good”

```
IF(
        // should not fail
        error != true
)
```

@shelbyspees at #SLOconf

and then our condition for "good" says it shouldn't error. we're not currently tracking how long provisions take because there are a few machine-side provision steps that aren't directly orchestrated by ProvisionDeviceJob. we have work in progress to improve our instrumentation and capture that in our trace data separately, so we're planning to update our SLI when that lands
nobody ever said bare-metal observability would be easy.

you can do this!

— add OpenTelemetry auto-instrumentation
— observe and learn
— define basic SLIs
— add custom instrumentation
— observe and learn
— define ✨ fancy ✨ SLIs

@shelbyspees at #SLOconf

but don't be intimidated. you can do this.
you can start observing latency and errors with OpenTelemetry now.
and then over time you can go in and instrument the most critical parts of your service
and decide whether you need to further refine your SLI to make it the best representation of your customer's experience.

thanks for watching!

@shelbyspees at #SLOconf

thanks for watching! hit me up in the conference slack or on twitter if you have questions or any cool observability stories.