



# Better Observability with OpenTelemetry

18 March 2024 for SREcon24 Americas  
Liz Fong-Jones (@lizthegrey), Field CTO, honeycomb.io

17 hours per developer-week,  
or \$300B per year globally, are  
lost to technical debt & bad  
code.

*The Developer Coefficient, Stripe, 2018*



# Developers have a lot on our plates.

We want devs to:

- Ship features faster
- Scale to demand
- Decrease downtime

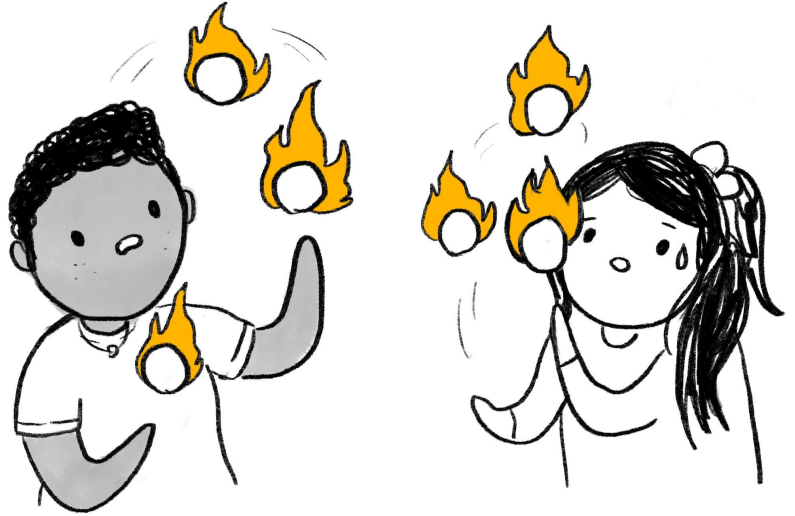


# Working in production is challenging.

We're afraid we'll break things.

We've forgotten what we've built by the time it ships.

We can't figure out what's wrong and fix it.





# But some organizations have solved it!

SRE & Progressive Delivery culture have the answers to some of these problems.

And a growing fraction of our industry is moving faster *and* safer!



# What do we still still struggle with?

- Microservices create complex interactions.
- Failures don't exactly repeat.
- Debugging multi-tenancy is painful.
- Monitoring no longer can help us.





# Liz Fong-Jones

Field CTO at Honeycomb.io

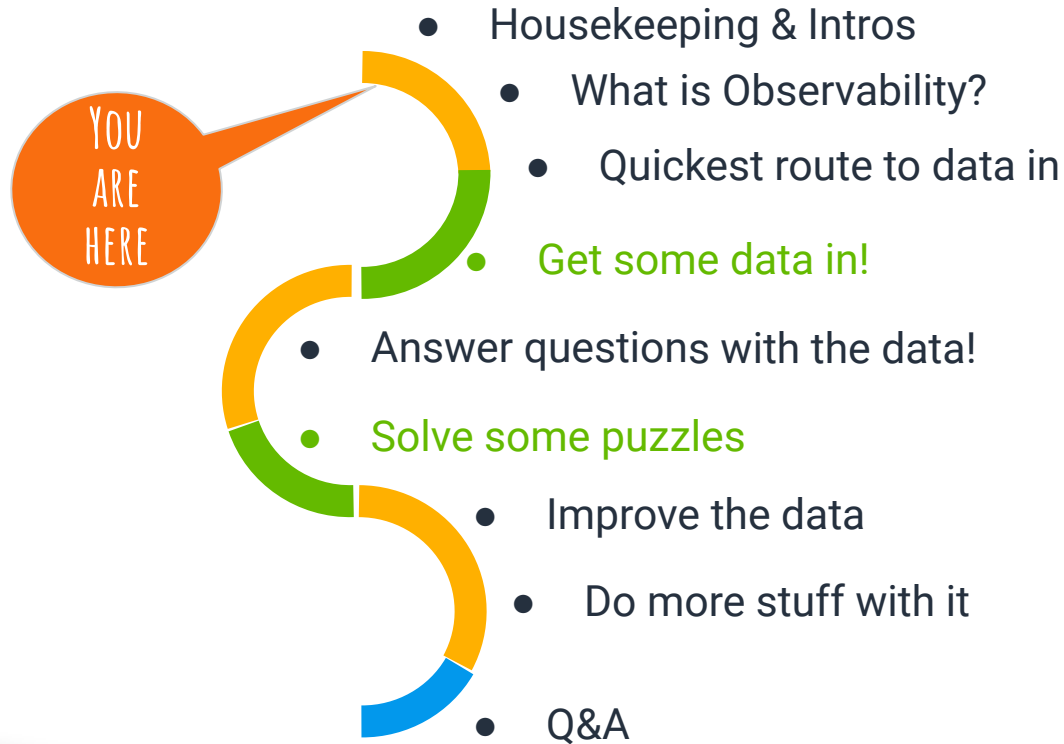
Fmr OTel Governance Committee (2020-2022)

Joined by some awesome TAs as well!



# Pre-break agenda

---



# Observability Basics



# How does observability help?

- We need to answer questions about our systems.

*What characteristics did the queries that timed out at 500ms share in common? Service versions? Browser plugins?*

- Instrumentation produces data.
- Querying data answers our questions.



## Monitoring and observability

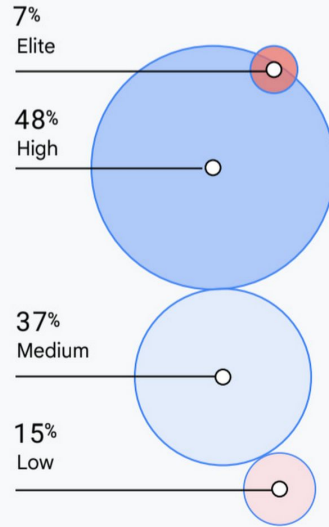
As with previous years, we found that monitoring and observability practices support continuous delivery. Elite performers who successfully meet their reliability targets are 4.1 times more likely to have solutions that incorporate observability into overall system health. Observability practices give your teams a better understanding of your systems, which decreases the time it takes to identify and troubleshoot issues. Our research also indicates that teams with good observability practices spend more time coding. One possible explanation for this finding is that implementing observability practices helps shift developer time away from searching for causes of issues toward troubleshooting and eventually back to coding.

## Open source technologies

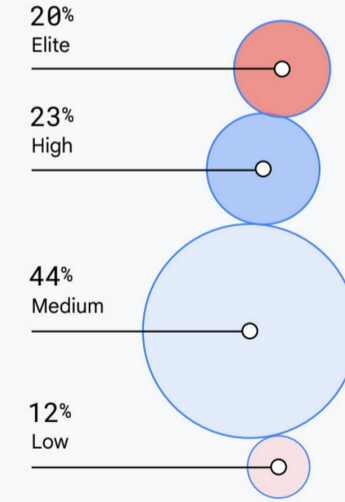
Many developers already leverage open source technologies, and their familiarity with these tools is a strength for the organization. A primary weakness of closed source technologies is that they limit your ability to transfer knowledge in and out of the organization. For instance, you cannot hire someone who is already familiar with your organization's tools, and developers cannot transfer the knowledge they have accumulated to other organizations. In contrast, most open source technologies have a community around them

that developers can use for support. Open source technologies are more widely accessible, relatively low-cost, and customizable. Elite performers who meet their reliability targets are 2.4 times more likely to leverage open source technologies. We recommend that you shift to using more open source software as you implement transformation.

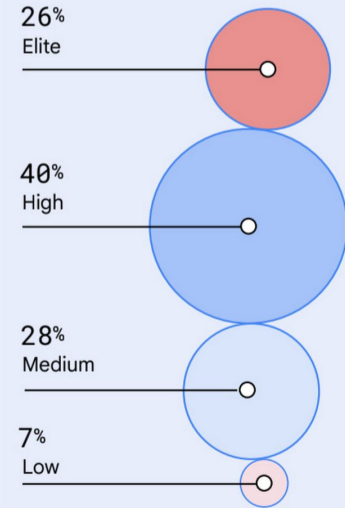
2018

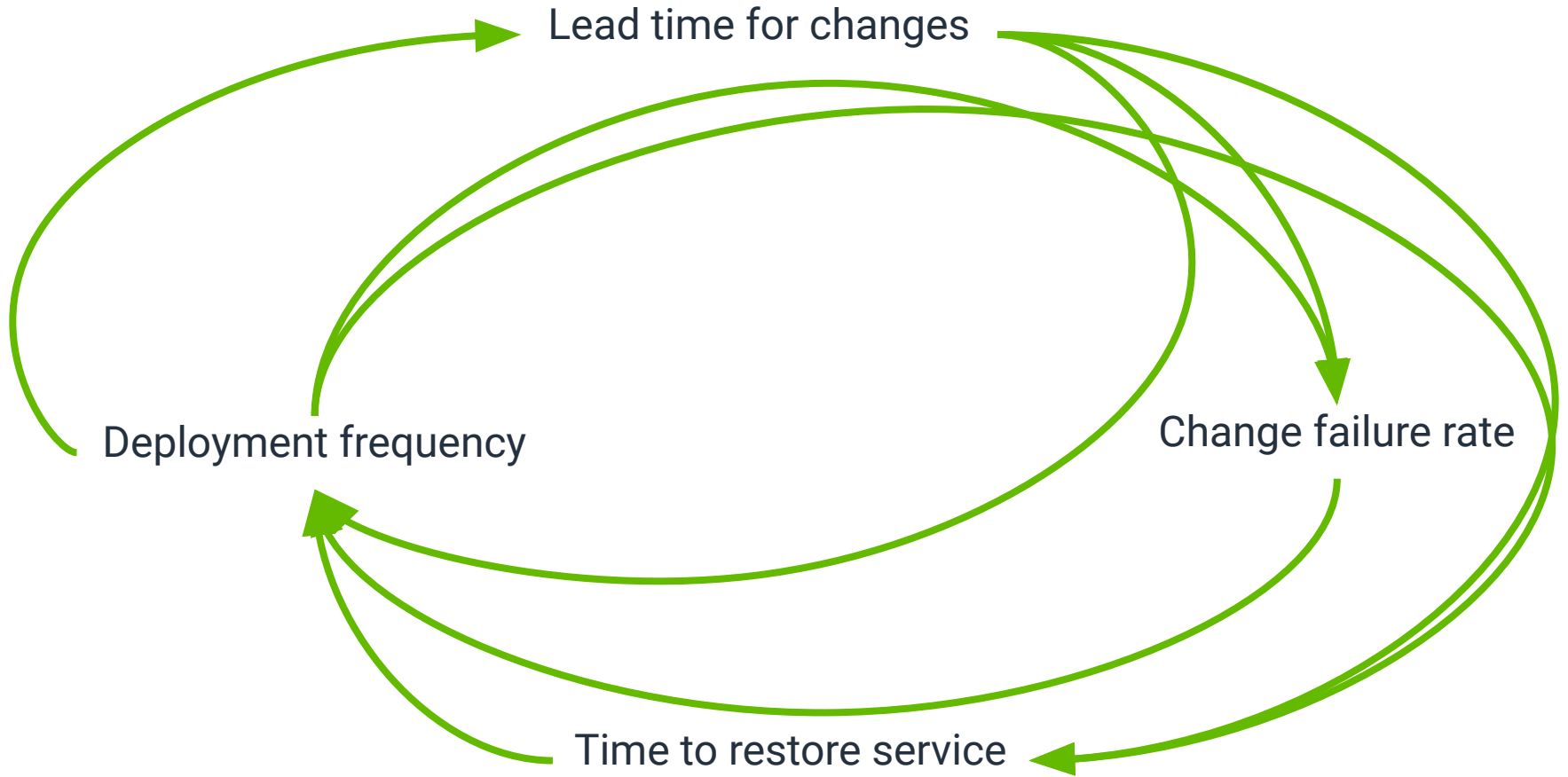


2019

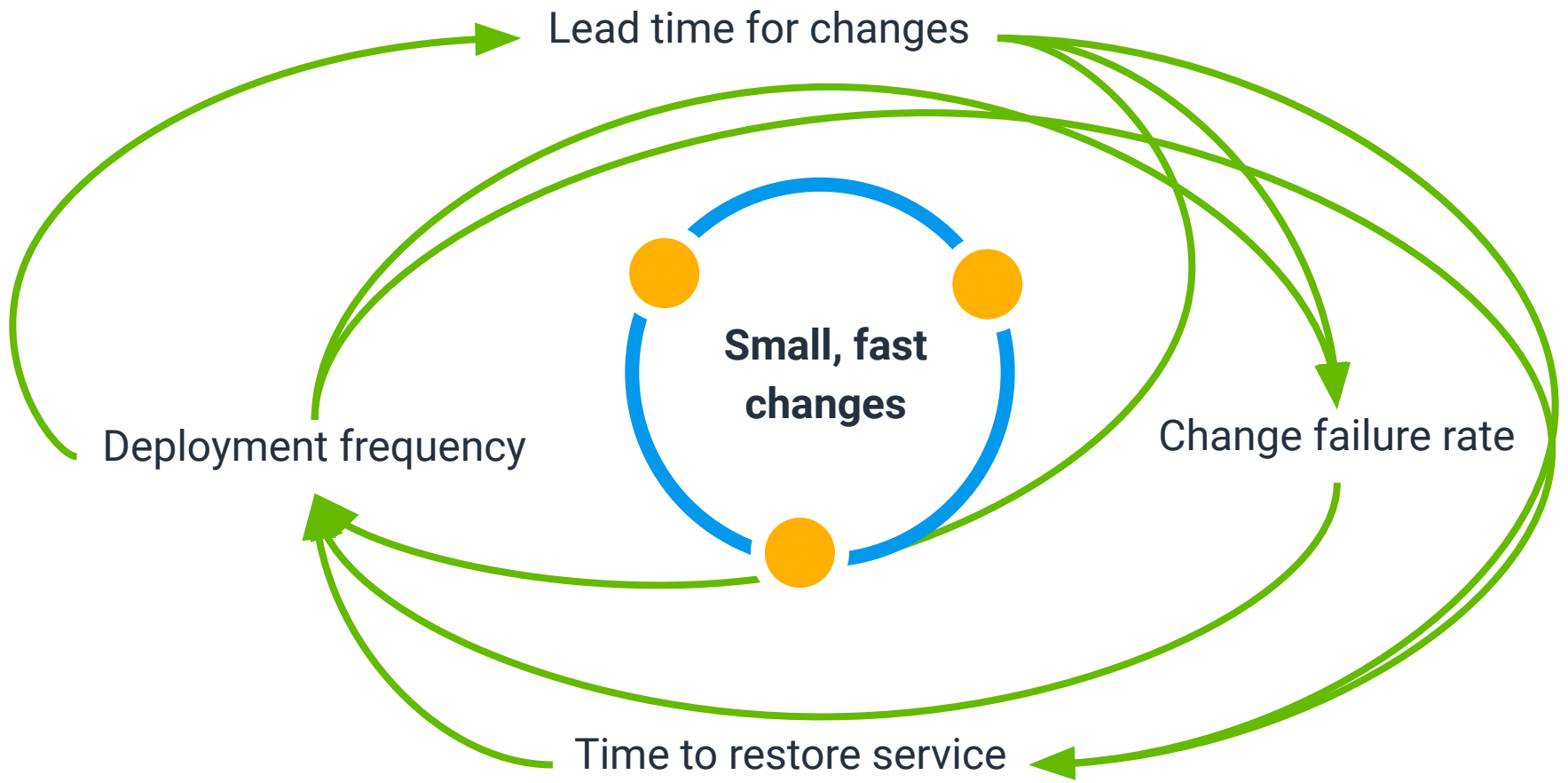


2021











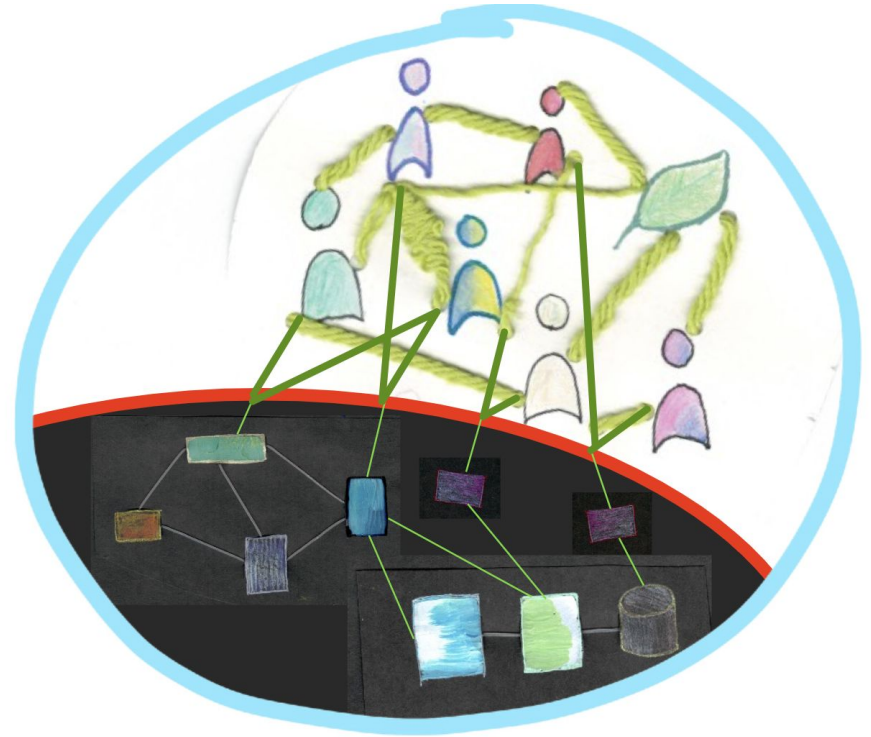
# Your software and tools are on your team.

The running software, plus the tools, plus the people on our team form a system,

a *symmathesy*: a learning system made of learning parts.

Our software and tools learn from us, because we change them.

Make the software teach you about itself and the world!



# so adapt your culture, not just the tools.

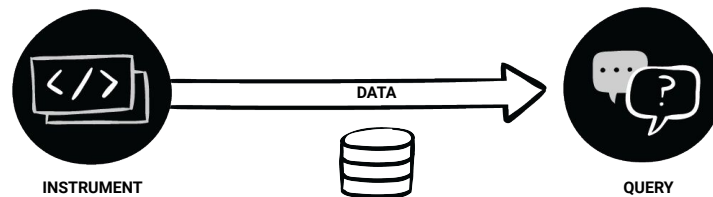
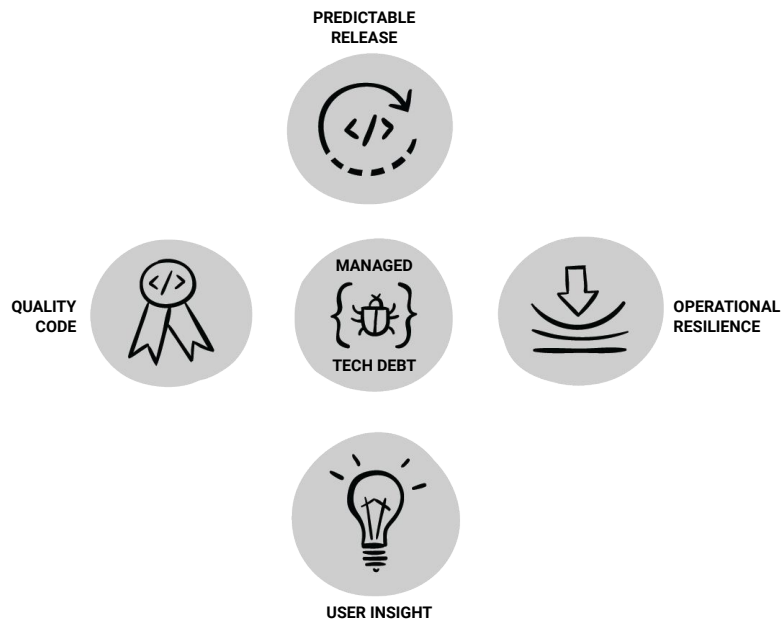
---

We need a culture of *curiosity*, not of fear.

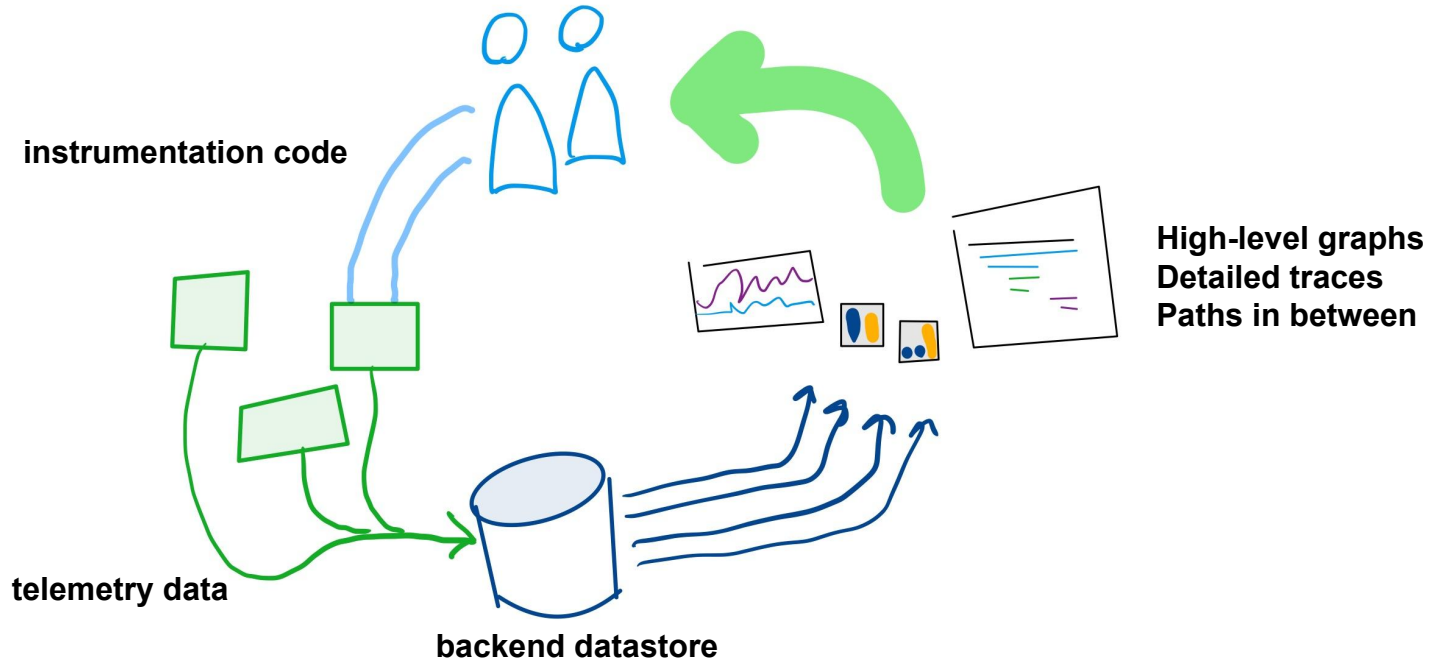
We need to accelerate feedback cycles.



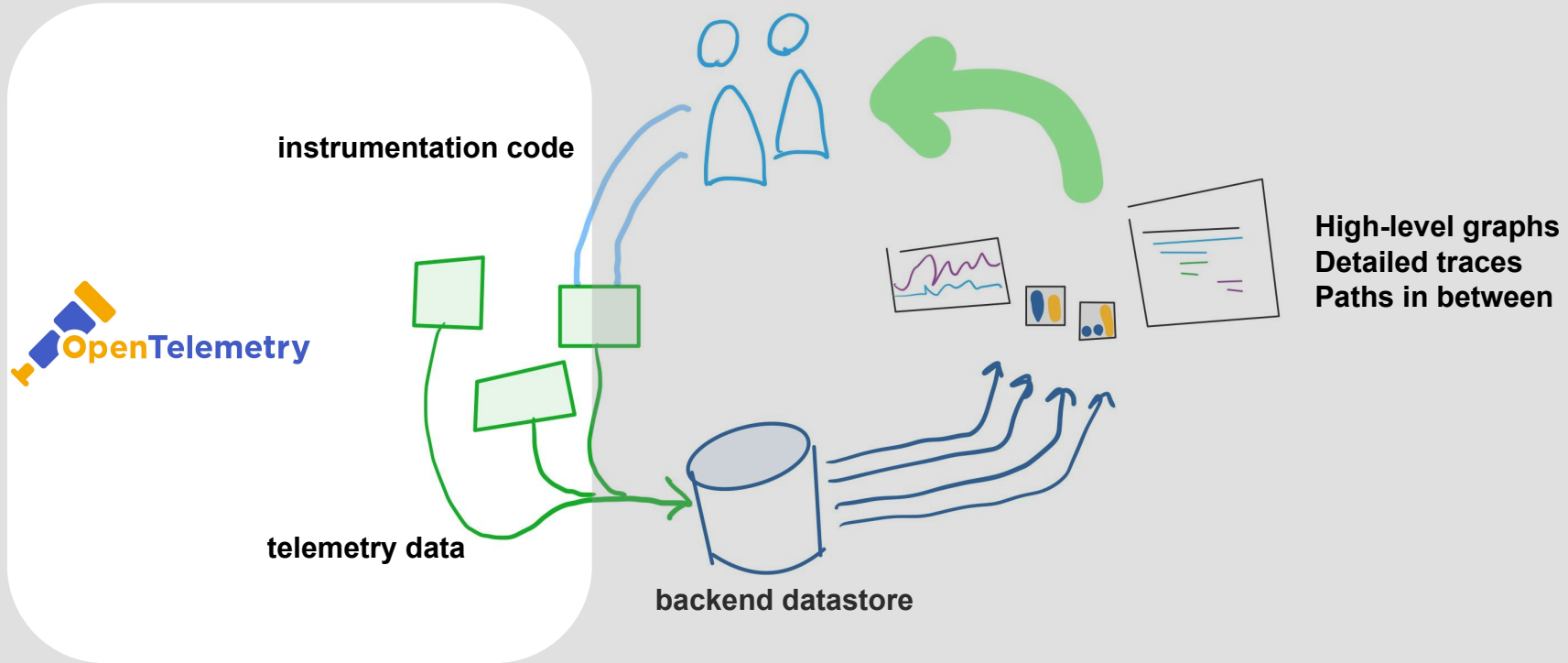
# Observability isn't just the data



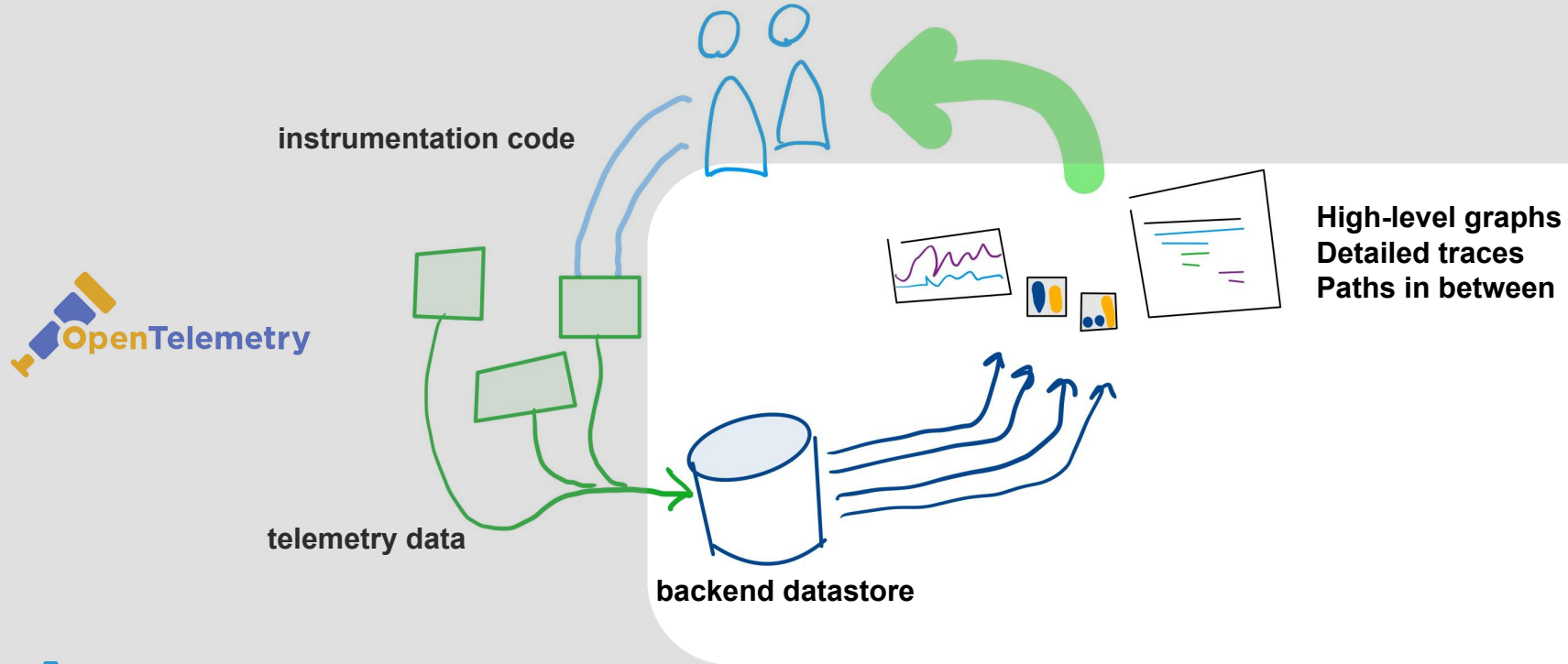
# How does observability work?



# How does observability work?



# How does observability work?





# Metrics, logs, and traces, oh my!

- Metrics
  - Aggregated summary statistics.
- Logs
  - Detailed debugging information emitted by processes.
- Distributed Tracing
  - Provides insights into the full lifecycles, aka **traces** of requests to a system, allowing you to pinpoint failures and performance issues. Unlimited cardinality!

Structured data can be transmuted into any of these!



# Emerging signals

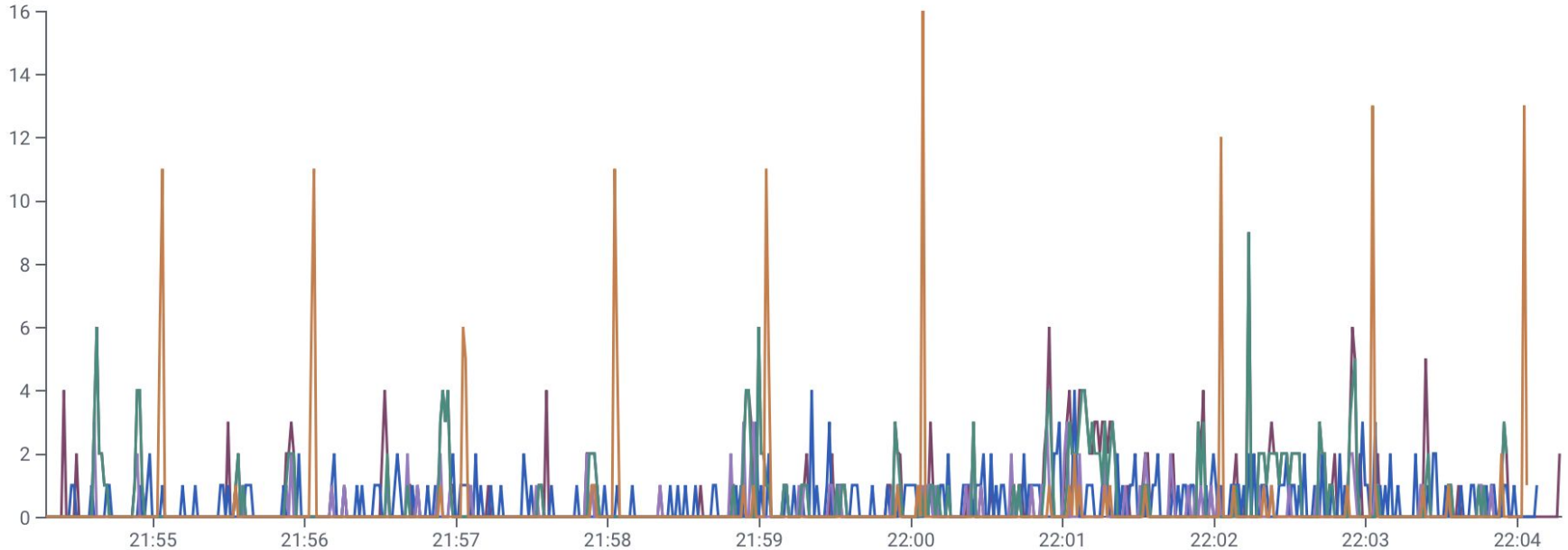
Continuous profiling

System change events

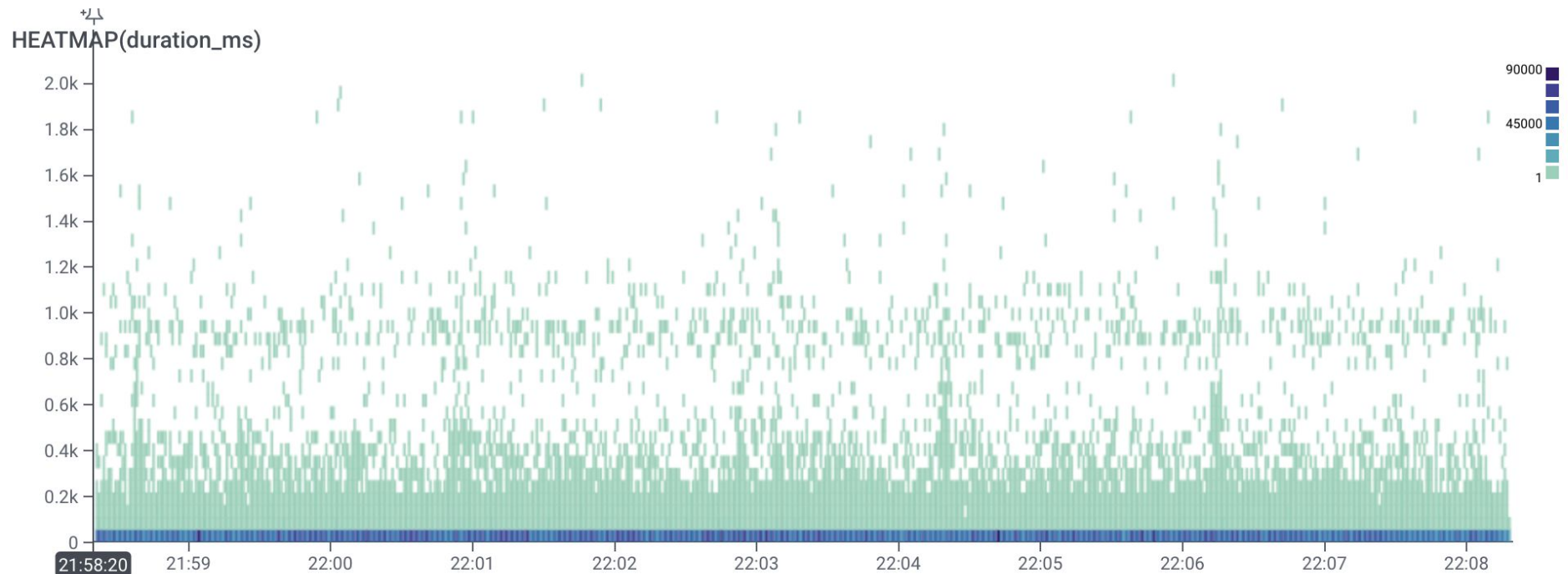


# Telemetry feeds graphs

COUNT



# Telemetry feeds heatmaps



# Telemetry feeds traces

Trace 041f1cbd0b6095e0bdc73889ac20120c at 2021-10-29 10:42:44

Rerun

Search spans

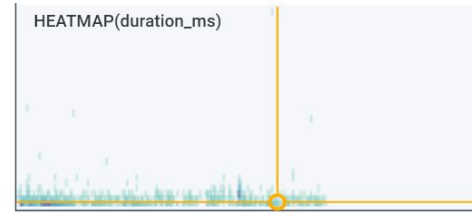


Fields

name	service_name	0s	0.005s	0.01s	0.015s	0.02136s
20 github.com/honey...ueryRunStatus-fm	poodle	21.36ms				
findByID	poodle	1.820ms				
launchdarkly.BoolVariations	poodle	0.2184ms				
launchdarkly.NumberVariation	poodle	23.9µs				
launchdarkly.JSONVariation	poodle	50.2µs				
launchdarkly.JSONVariation	poodle	36.0µs				
FindTeamBySlug	poodle	2.095ms				
FindWriteKeysByTeam	poodle	2.056ms				
CheckMembership	poodle	1.659ms				
GetRole	poodle	1.693ms				
ExternalAuthProvider	poodle	1.658ms				
launchdarkly.BoolVariations	poodle	0.1779ms				

poodle >  
github.com/honeycombio/launchdarkly.QueryRunStatus-fm

Distribution of span duration ?



Fields

trace|

- trace.span\_id  
df9fc50200e3ad1c
- trace.trace\_id  
041f1cbd0b6095e0bdc73889ac20120c

# About OpenTelemetry



# Vendor-neutral instrumentation

OpenCensus + OpenTracing  
= OpenTelemetry

OTel replaces need for  
vendor-specific SDKs

Supports tracing and  
context propagation, as well  
as metrics (and soon logs!)



# Timeline

- 2016 - OpenTracing founded
- 2017 - OpenCensus founded
- 2018 - OpenTelemetry formed
- 2019 - OpenTelemetry alpha
- 2020 - OpenTelemetry beta
- 2021 - OpenTelemetry GA





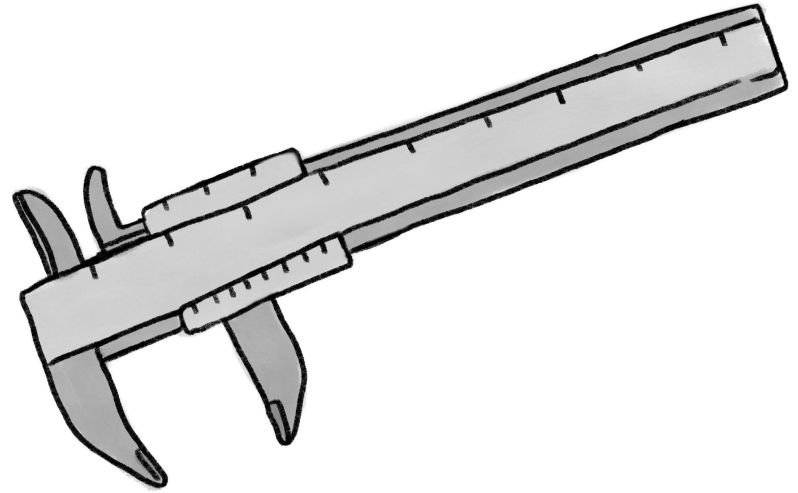
# Components of OpenTelemetry

Cross-language  
specification

OTel Collector

Per-language API & SDK

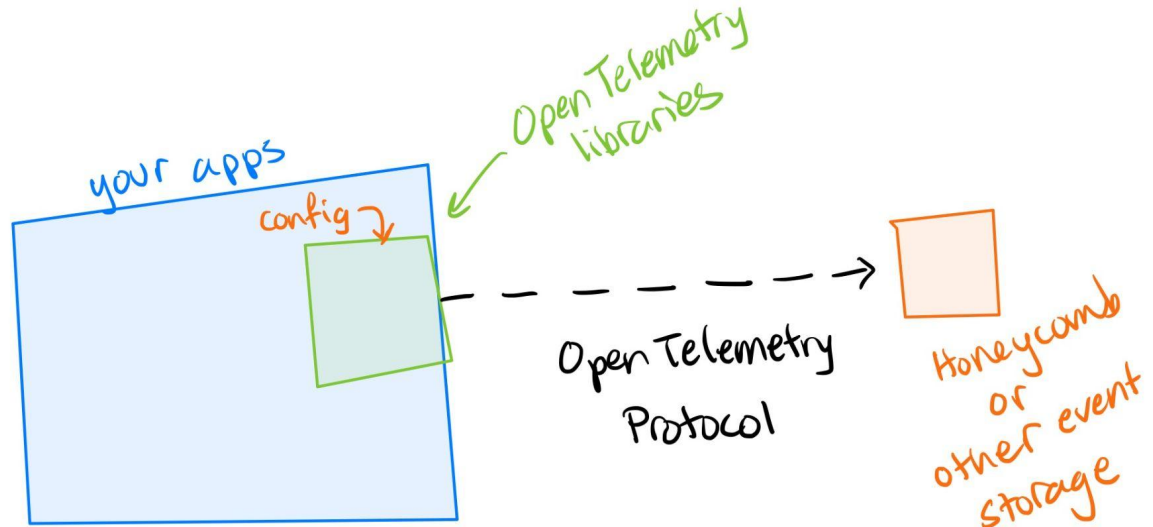
Auto-instrumentation  
libraries



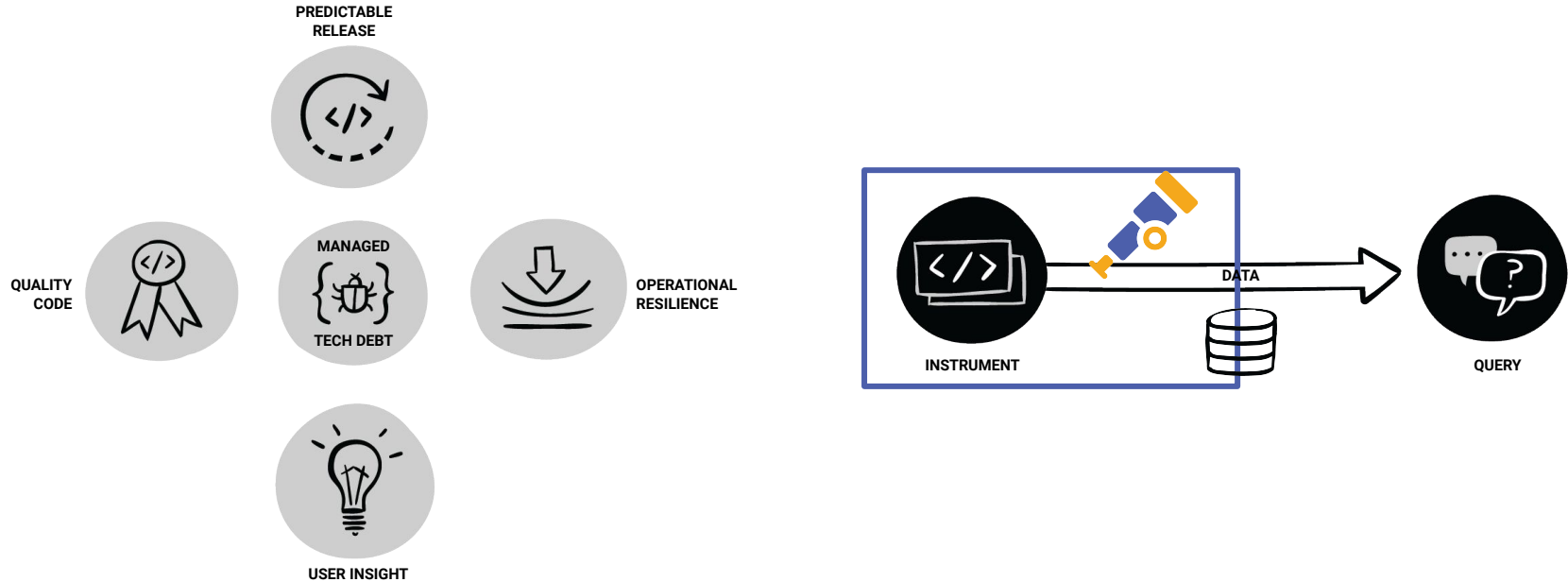
# This is OpenTelemetry

OpenTelemetry is a vendor-neutral standard for instrumentation.

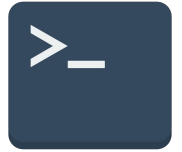
It has an instrumentation API, various SDKs, proxies, and wire formats pre-standardized.



# OpenTelemetry creates the needed data



# Regardless of which language or tech you use



# APIs

- Data types for all tracing, metrics, and logging concepts
- Methods to append data from your code

```
import "go.opentelemetry.io/otel"  
import "go.opentelemetry.io/otel/attribute"  
  
var tracer = otel.Tracer("library_name")  
  
func do_something(ctx, param):  
    ctx, span := tracer.Start(ctx, "function_name")  
    defer span.End()  
    span.SetAttributes(attribute.Int("parameter", param))
```



# Context Propagation

- Distributed context is an abstract data type that represents collection of entries.
- Each key is associated with exactly one value.
- It is serialized for propagation across process boundaries
- Passing around the context enables related spans to be associated with a single trace.
- W3C TraceContext is the de-facto standard.
  - B3 is more broadly compatible with existing systems.



# Automatic Instrumentation

OpenTelemetry has wrappers around common frameworks to propagate context and make it accessible.

```
import "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp"  
  
otelhttp.NewHandler(http.HandlerFunc(h), "h"))  
  
func h(w ResponseWriter, req *Request) {  
    ctx := req.Context()  
    span := trace.SpanFromContext(ctx)  
}
```



# SDKs, Exporters, and Collector Services, Oh My!

- OpenTelemetry's **SDK** implements trace & span creation.
- An **exporter** can be instantiated to send the data collected by OpenTelemetry to the backend of your choice.
  - E.g. Jaeger, Lightstep, Honeycomb, Stackdriver, etc.
  - OpenTelemetry Protocol (OTLP) is the lingua franca.
- OpenTelemetry **collector** proxies data between instrumented code and backend service(s). The exporters can be reconfigured without changing instrumented code.





# Vendor-neutral exporters

- **OTLP exporter**
  - Exports in OTel's native protobuf or JSON format
- **Jaeger exporter (now supported directly with OTLP)**
  - Jaeger was created at Uber and is now an open-source CNCF project
  - Stores and visualizes traces.
- **Prometheus exporter**
  - Prometheus is a TSDB inspired by Google's Borgmon
  - Stores time-series metrics. Note: OTel metrics are not finalized



# The OpenTelemetry Collector

- Collectors are useful for more than just proxying traces/metrics.
  - Receive multiple trace formats and marshal them into a new one.
  - Enhance trace/metric data with resources.
  - Perform sampling, filtering, or custom processors to modify attributes.
  - Much more - it's customizable!
- Run them as agents or in standalone mode.
  - A Kubernetes operator exists to aid in deployment.
  - Builds are available for x86/ARM Linux, Darwin, and Windows.



# How to get started



# Start with auto-instrumentation

Measure a few tightly connected, req-resp services.

Get useful trace data out.

Gather traces & system metrics with OTel Collector



# Pick a telemetry backend with confidence

Start with Jaeger & Prometheus or a vendor free tier

Try out providers as your needs grow

Simple configuration to tee or reroute data



# Instrument manually to get the most value

Add custom attributes relevant to business

Create spans covering smaller units of work

Implement context propagation across Kafka, SQL, etc



# Use observability data to level up

Service Level Objectives at top level, debug all the way down

Find single points of failure, dependency cycles, etc

Take automatic remediation actions



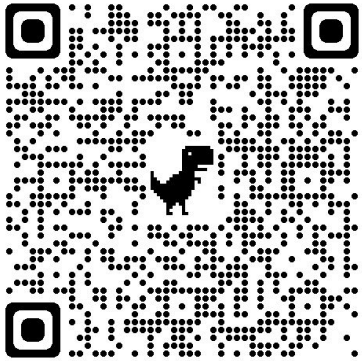
Into the Fray!





# Slides for today

- You will need to refer to material from the slides during the interactive work period.
- Please find a copy of the slides here:



# OTel API - packages, methods, & when to call

- Tracer
  - A Tracer is responsible for tracking the currently active span.
- Meter
  - A Meter is responsible for accumulating a collection of statistics.

You can have more than one. Why?

Ensures uniqueness of name prefixes.



## Code examples: Providers (Go)

- A global provider can have a TraceProvider registered.
- Use the TraceProvider to create a named tracer.

```
// Register your provider in your init code
tp, err := sdktrace.NewProvider(...)
global.SetTraceProvider(tp)
// Create the named tracer
tracer = global.TraceProvider().Tracer("workshop/main")
```



# Code examples: Providers (Python)

```
trace.set_preferred_tracer_provider_implementation(lambda T: TracerProvider())
```

```
// initialize tracer for the process  
tracer = trace.get_tracer(__name__)
```



# OTel API (Go) - Tracer methods, & when to call

- `tracer.Start(ctx, name, options)`
  - This method returns a child of the current span, and makes it current.
- `tracer.WithSpan(name, func() {...})`
  - Starts a new span, sets it to be active in the context, executes the wrapped body and closes the span before returning the execution result.
- `trace.SpanFromContext(ctx)`
  - Used to access & add information to the current span



# OTel API (Python) - Tracer methods, & when to call

- `tracer.start_span(name, parent=<span>, ...)`
  - This method returns a child of the specified span.
- `with tracer.start_as_current_span(name)`
  - Starts a new span, sets it to be active. Optionally, can get a reference to the span.
- `tracer.get_current_span()`
  - Used to access & add information to the current span



# OTel API (Go) - Span methods, & when to call

- `span.AddEvent(ctx, msg)`
  - Adds structured annotations (e.g. "logs") about what is currently happening.
- `span.SetAttributes(core.Key(key).String(value)...)...`
  - Adds a structured, typed attribute to the current span. This may include a user id, a build id, a user-agent, etc.
- `span.End()`
  - Often used with `defer`, fires when the unit of work is complete and the span can be sent



# OTel API (Python) - Span methods, & when to call

- `span.add_event(name, attributes)`
  - Adds structured annotations (e.g. "logs") about what is currently happening.
- `span.set_attribute(key, value)`
  - Adds an attribute to the current span. This may include a user id, a build id, a user-agent, etc.
- `span.end()`
  - Manually closes a span.





## Code examples (Go): Start/End

```
var tr = otel.Tracer("me")

func (m Model) persist(ctx context.Context) {
    ctx, span := tr.Start(ctx, "persist")
    defer span.End()

    // Persist the model to the database...
    [...]
}
```



## Code examples (Python): Start/End

```
tracer = trace.get_tracer(__name__)
```

```
def persist(data):  
    tracer.start_as_current_span("persistData")  
    // do work...  
    return result
```



## Code examples (Go): WithSpan

Takes a context, span name, and the function to be called.

```
ret, err := tracer.WithSpan(ctx, "operation",
    func(ctx context.Context) (int, error) {
        // Your function here
        [...]
        return 0, nil
    }
)
```



## Code examples (Go): CurrentSpan & Span

- Get the current span
  - `sp := trace.SpanFromContext(ctx)`
- Update the span status
  - `sp.SetStatus(codes.OK)`
- Add events
  - `sp.AddEvent(ctx, "foo")`
- Add attributes
  - `sp.SetAttributes(  
key.New("ex.com/foo").String("yes"))`



# Code examples (Python): Current Span & Span

- Get the current span
  - `span = tracer.get_current_span()`
- Update the span status
  - `span.set_status(Status(StatusCanonicalCode.UNKNOWN, error))`
- Add events
  - `span.add_event("foo", {"customer": "bar"})`
- Add attributes
  - `span.set_attribute("error", True)`



# Context Propagation

- Distributed context is an abstract data type that represents collection of entries.
- Each key is associated with exactly one value.
- It is serialized for propagation across process boundaries
- Passing around the context enables related spans to be associated with a single trace.
- W3C TraceContext is the de-facto standard.



## Automatic Instrumentation

OpenTelemetry has wrappers around common frameworks to propagate context and make it accessible.

```
import "go.opentelemetry.io/otel/plugin/othttp"  
  
othttp.NewHandler(http.HandlerFunc(h), "h"))  
  
func h(w ResponseWriter, req *Request) {  
    ctx := req.Context()  
    span := trace.SpanFromContext(ctx)
```



# Automatic Instrumentation (Python)

```
from opentelemetry.ext import http_requests
from opentelemetry.ext.flask import instrument_app

// instrument Requests library
http_requests.enable(trace.tracer_source())

// create flask app, then instrument
app = Flask(__name__)
instrument_app(app)
```





# SDKs, Exporters, and Collector Services, Oh My!

- OpenTelemetry's **SDK** implements trace & span creation.
- An **exporter** can be instantiated to send the data collected by OpenTelemetry to the backend of your choice.
  - E.g. Jaeger, Lightstep, Honeycomb, Stackdriver, etc.
- OpenTelemetry **collector** proxies data between instrumented code and backend service(s). The exporters can be reconfigured without changing instrumented code.



# Vendor-neutral exporters

- Jaeger exporter
  - Jaeger was created at Uber and is now an open-source CNCF project
  - Stores and visualizes traces.
  - Deprecated (Jaeger uses OTLP now)
- Prometheus exporter
  - Prometheus is a TSDB inspired by Google's Borgmon
- stdout/stderr streaming export
  - Inspect what is actually being sent over the wire.
  - No external setup required!



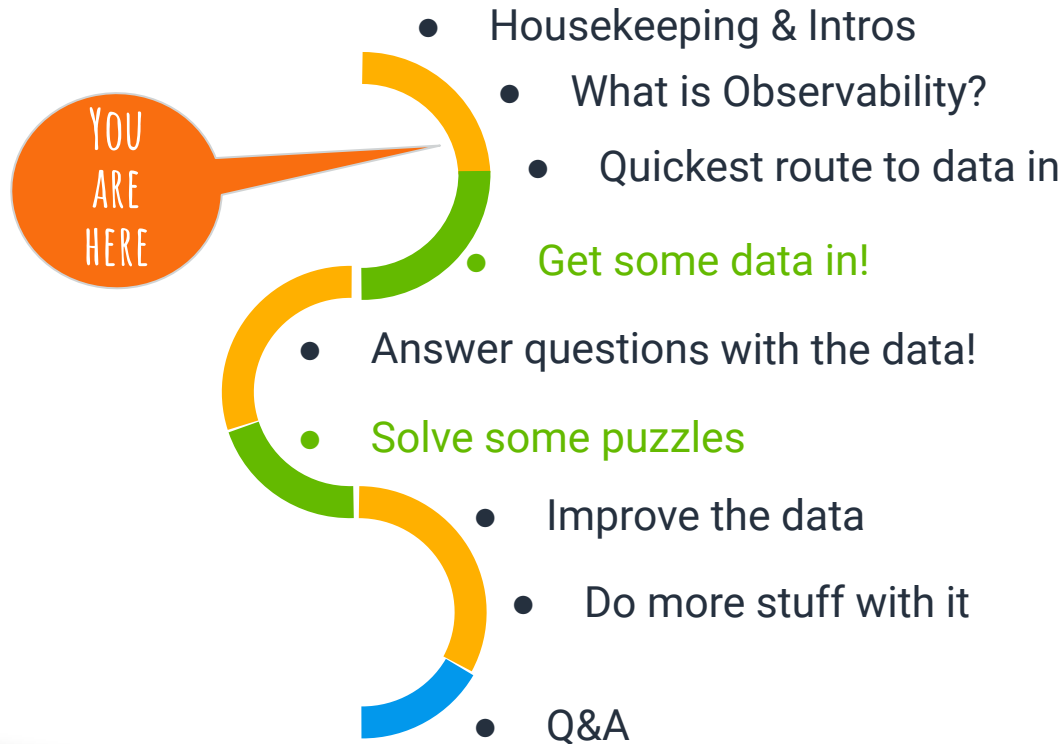
# **Observability requires telemetry.**

---

Let's get some data in!

# Pre-break agenda

---

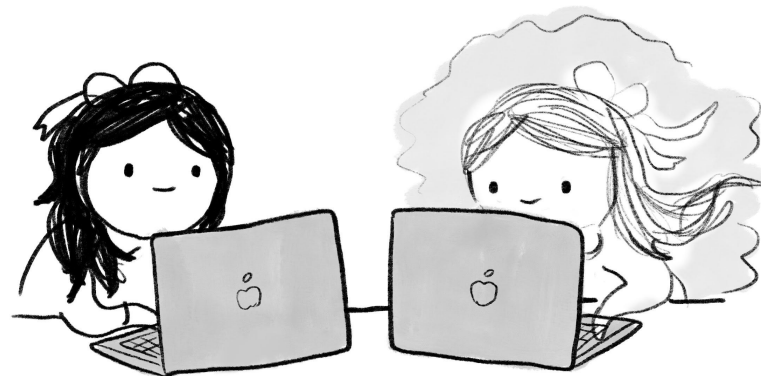


# Section Overview

---

In this section, we will:

- Run a demo app instrumented with OpenTelemetry
- Send telemetry from the demo app to Jaeger
- Get a hosted service Account
- Send telemetry from the demo app to hosted service



# Get the Demo App

---



<https://github.com/orgs/honeycombio/repositories?q=intro>

- Java
- .NET
- Node.js
- Python
- Go

- Java
- .NET
- Node.js
- Python
- Go



# Choose the GitHub Repo for your language

Go to <https://github.com/honeycombio>

Find a repository...

“intro”

Choose the repository (repo) for your language.

The screenshot shows the Honeycomb GitHub repository page. At the top, there's a navigation bar with 'Overview', 'Repositories', 'Packages', 'People', 'Teams', and 'Projects'. Below that is a search bar containing 'intro' and filters for 'Type', 'Language', and 'Sort'. A message indicates '5 results for all repositories matching intro sorted by last updated'. The results list five repositories, each with a title, a 'Public' badge, a description, and metadata like language, Apache-2.0 license, forks, stars, and update time.

Repository Name	Language	License	Forks	Stars	Issues	PRs	Updated
intro-to-o11y-java	Java	Apache-2.0	0	0	0	0	Updated 44 minutes ago
intro-to-o11y-dotnet	C#	Apache-2.0	4	0	0	1	Updated 1 hour ago
intro-to-o11y-go	Go	Apache-2.0	0	0	0	1	Updated 17 hours ago
intro-to-o11y-nodejs	JavaScript	Apache-2.0	0	1	0	1	Updated 17 hours ago
intro-to-o11y-python	Python	Apache-2.0	0	0	0	1	Updated 17 hours ago

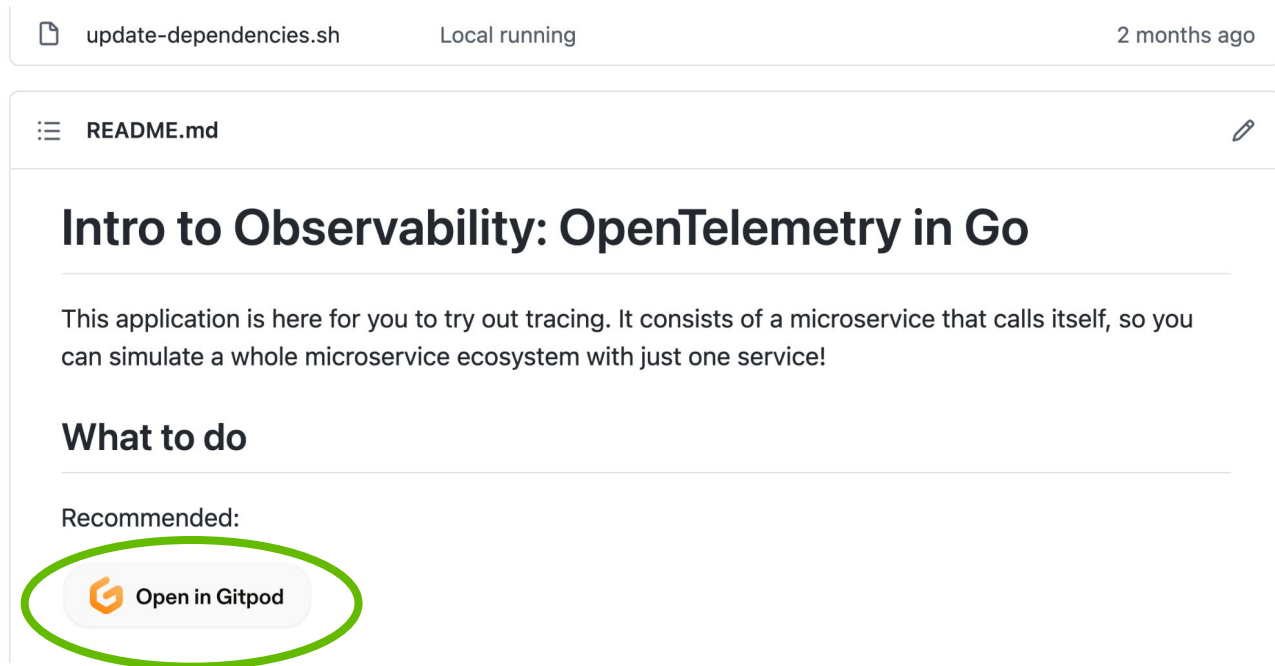


# Choose the GitHub Repo for your language

Go to <https://github.com/orgs/honeycombio/repositories?q=intro>

Choose the repository  
(repo) for your language.

**Scroll down and  
select the Gitpod button  
in the repo's README.**



update-dependencies.sh Local running 2 months ago


☰ README.md ✎

## Intro to Observability: OpenTelemetry in Go

This application is here for you to try out tracing. It consists of a microservice that calls itself, so you can simulate a whole microservice ecosystem with just one service!

### What to do

Recommended:

 Open in Gitpod



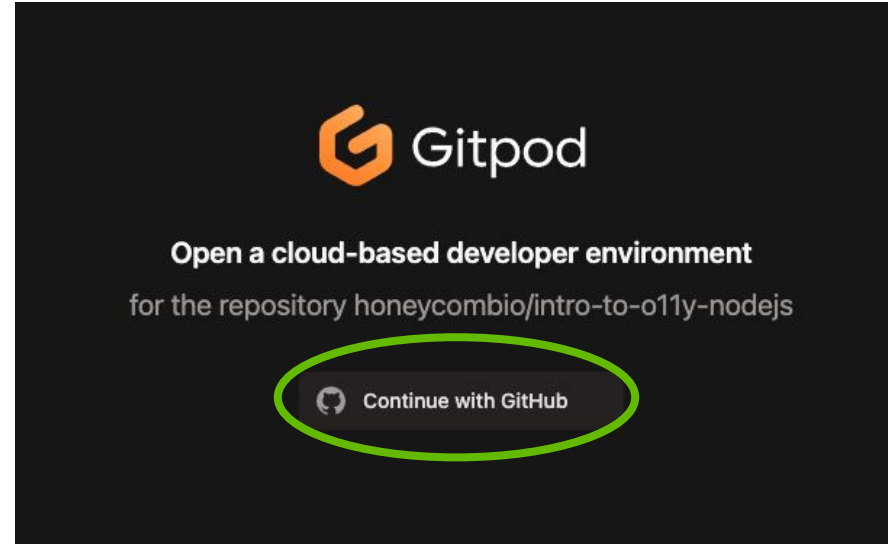


# Log in

---

A screen will appear and prompt you to sign-in with GitHub.

Select **Continue with GitHub**



# Gitpod Appears

Layout Reflects VSCode

Important areas include:

- Activity Bar
- File Explorer
- Tab Navigation
- Terminal

Select README.md in the File Explorer to see instructions.

The screenshot displays the Gitpod IDE interface with several key components highlighted in green:

- Activity Bar:** Located on the left side, it contains icons for switching between views like Explorer, Search, Source Control, Run and Debug, Extensions, Remote Explorer, and Gitpod.
- File Explorer:** The Explorer view on the left shows the file structure of the 'INTRO-TO-011Y-NODEJS' workspace, including files like .devcontainer, .vscode, node\_modules, src, static, .env.example, .gitignore, .gitpod.yml, deploy.sh, package-lock.json, package.json, README.md, and shrinkwrap.yaml.
- Tab Navigation:** At the top, a tab labeled 'Get Started' is visible.
- Terminal:** The bottom panel shows the output of running 'npm run start', displaying warnings about EBadENGINE, package installation progress, and a notice about a new version of npm (8.1.0 to 8.3.2).



# Troubleshooting: Terminal not appearing

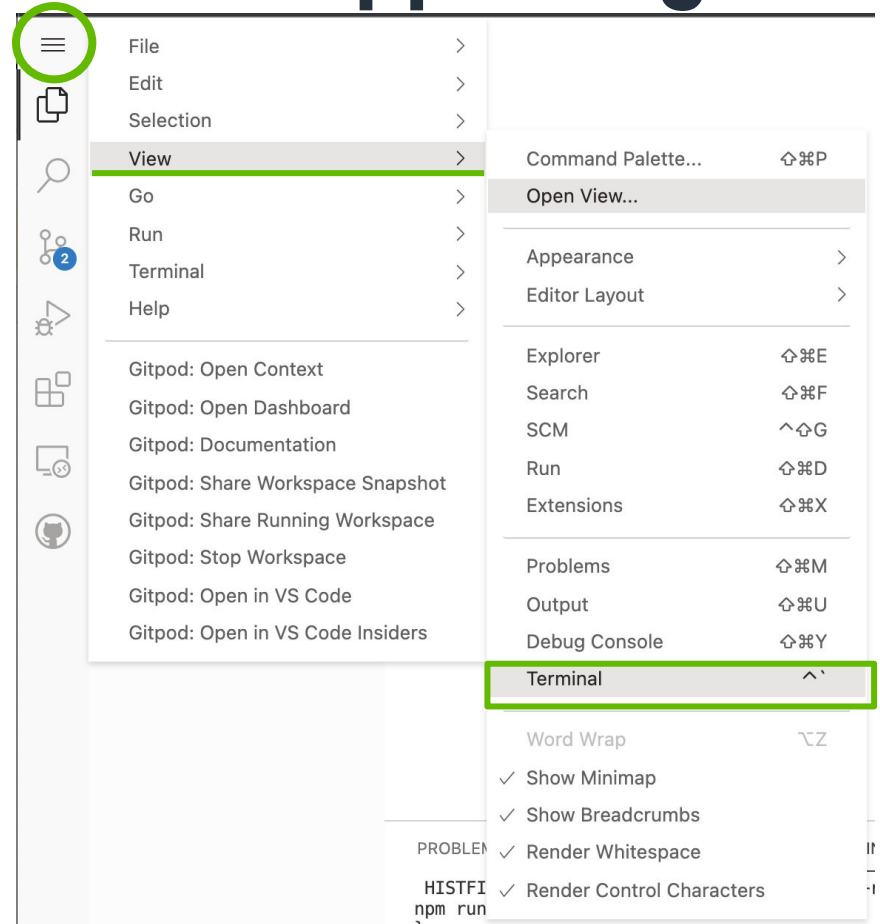
Go to the upper left corner to the

**Hamburger Menu > View > Terminal**

The terminal window will appear in the bottom half of the screen

Shortcut: **Ctrl+J** (Cmd+J on Mac)

Type `./run` or `./run.sh`



# How to Open an App Preview

In the Terminal area, a "Service is available on port xxxx" pop-up may appear. Select **Open Preview** to open the app display in a new tab.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
HISTFILE=/workspace/.gitpod/cmd-0 history -r; {  
npm run start  
}
```

```
gitpod /workspace/intro-to-o1ly-nodejs $ HISTFILE=/workspace/.gitpod/cmd-0 history -r; {  
> npm run start  
> }
```

```
> fibonacci-microservice@0.0.1 start  
> node ./src/index.js
```

```
Exporting to Honeycomb with APIKEY <undefined> and dataset otel-nodejs  
Listening on port 3000. Try: http://localhost:3000/  
□
```

+ v ^ x

▢ bash  
▢ npm run st...

 A service is available on port 3000  

Source: Gitpod We...

Make Public

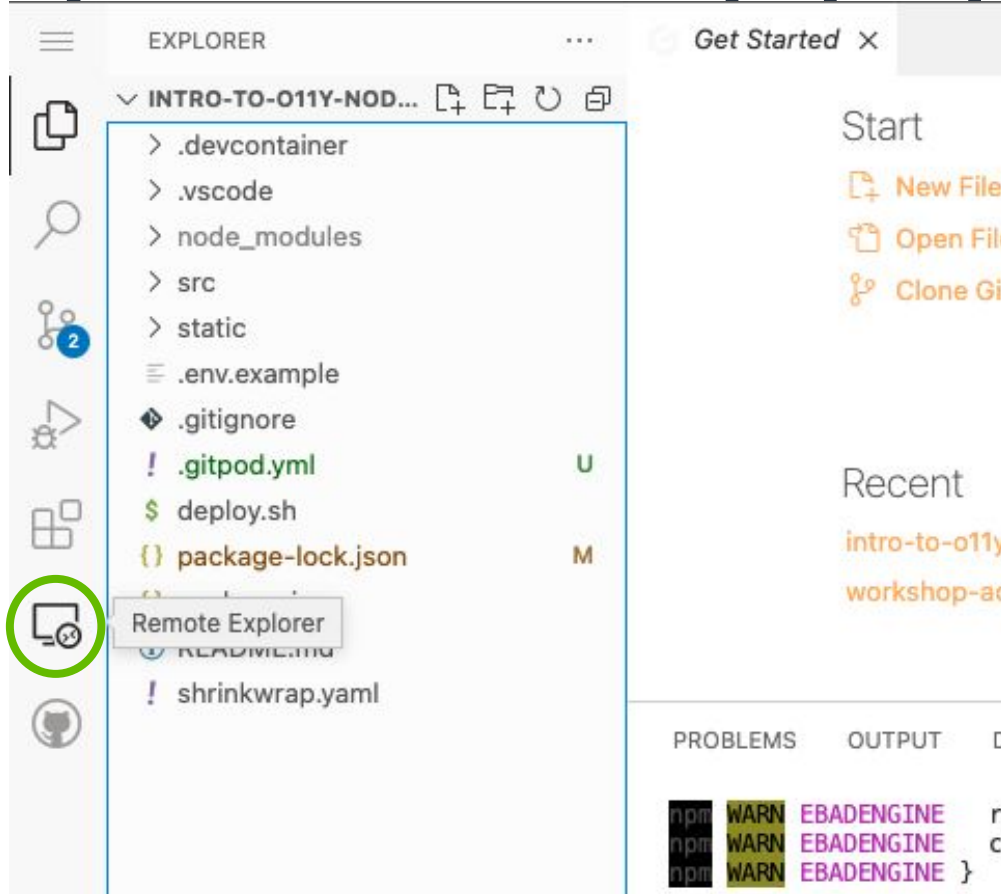
Open Preview

Open Browser



# Troubleshooting: If you don't see the pop-up

Go to **Remote Explorer**.

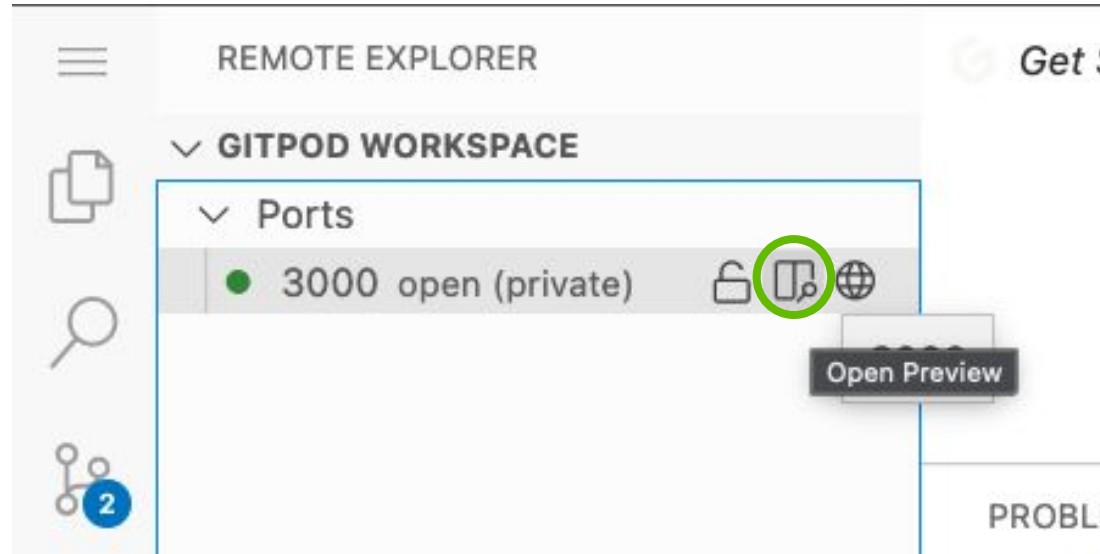


# Troubleshooting: If you don't see the pop-up

Go to Remote Explorer.

Expand *Ports* to see the listed port number. For example, *3000*

Choose the middle ***Open Preview*** icon for the app to appear in a new tab.




# Try out the app

---

## A sequence of numbers:

**Push Go!**

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 

**Push Stop!**



Let's review what's been done already\*





(some is in wrappers/distros/launchers)



# Add OTEL imports and set up SDK

```
import "go.opentelemetry.io/otel/trace"  
import "go.opentelemetry.io/otel"  
import sdktrace "go.opentelemetry.io/otel/sdk/trace"  
  
tp := sdktrace.NewTracerProvider(  
    sdktrace.WithSampler(sdktrace.AlwaysSample()))  
otel.SetTracerProvider(tp)
```



# Add trace spans to the logic

- `mux.Handle("/", http.HandlerFunc(rootHandler))`
  - Wrap `rootHandler` with HTTP plugin
  - Add `dbHandler` internal span.
- `mux.Handle("/fib", http.HandlerFunc(fibHandler))`
  - Returns `/fib?i=n-1 + /fib?i=n-2`
    - Wrap the handler
    - Add attributes for the parameters
    - Create spans for each parallel client call
    - Propagate the context to downstream calls.



## othttp instrumentation of root handler

```
import "go.opentelemetry.io/otel/plugin/othttp"  
  
func main() {  
    mux.Handle("/", othttp.NewHandler(  
        http.HandlerFunc(rootHandler), "root"))  
  
func rootHandler([...]) {  
    ctx := req.Context()  
    trace.SpanFromContext(ctx).AddEvent(ctx, "Ran root  
handler.")
```



# Internal spans & context propagation

```
func rootHandler([...]) {  
    ctx := req.Context()  
    dbHandler(ctx, "blue")  
}
```

```
func dbHandler(ctx context.Context, color string) int {  
    tr := global.TraceProvider().Tracer("dbHandler")  
    ctx, span := tr.Start(ctx, "database")  
    defer span.End()  
}
```



## TODO: Configure output to stdout

```
import "go.opentelemetry.io/otel/exporters/stdout/stdouttrace"  
  
func main() {  
    std, err := stdouttrace.New(stdouttrace.WithPrettyPrint())  
    if err != nil {  
        log.Fatal(err)  
    }  
    sdktrace.NewProvider(...,  
        sdktrace.WithSyncer(std))  
}
```



# Python



# Add OTel imports and set up SDK

```
# server.py
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import (
    SimpleExportSpanProcessor,
    ConsoleSpanExporter
)

from opentelemetry.instrumentation.requests import RequestsInstrumentor
from opentelemetry.instrumentation.flask import FlaskInstrumentor
```





## Add OTel imports and set up SDK / stdout

```
trace.set_tracer_provider(TracerProvider())
trace.get_tracer_provider().add_span_processor(
    SimpleExportSpanProcessor(
        ConsoleSpanExporter()
    )
)

tracer = trace.get_tracer(__name__)
```



# Instrument Flask and Requests

```
app = Flask(__name__)
```

```
instrument_app(app)
```

```
RequestsInstrumentor().instrument(tracer_provider=trace.get_
tracer_provider())
```



# What you should see...

## See Terminal/Debug Console in Gitpod

```
{
  "SpanContext": {
    "TraceID": "9850b11fa09d4b5fa4dd48dd37f3683b",
    "SpanID": "1113d149cffffa942",
    "TraceFlags": 1
  },
  "ParentSpanID": "e1e1624830d2378e",
  "SpanKind": "internal",
  "Name": "dbHandler/database",
  "StartTime": "2019-11-03T10:52:56.903919262Z",
  "EndTime": "2019-11-03T10:52:56.903923338Z",
  "Attributes": [],
  "MessageEvents": null,
  "Links": null,
  "Status": 0,
  "HasRemoteParent": false,
  ""
}
```

 Golang

Python 

```
Span(name="root", context=SpanContext(trace_id=0xe2b0888b60ef4828851aa290136d9978, span_id=0x960a301445cd7495, trace_state={}), kind=SpanKind.SERVER, parent=SpanContext(trace_id=0xe2b0888b60ef4828851aa290136d9978, span_id=0xa51ce1f847f9b967, trace_state={}), start_time=2020-03-03T00:17:03.789244Z, end_time=2020-03-03T00:17:03.795240Z)
```



# Understanding the output

JSON formatted info, output in order End() was called.

```
"SpanContext": {  
  "TraceID": "9850b11fa09d4b5fa4dd48dd37f3683b",  
  "SpanID": "1113d149cffffa942",  
  "TraceFlags": 1  
},  
"ParentSpanID": "e1e1624830d2378e",  
"SpanKind": "internal",  
"Name": "dbHandler/database",  
"StartTime": "2019-11-03T10:52:56.903919262Z",  
"EndTime": "2019-11-03T10:52:56.903923338Z",  
"Attributes": [],  
"MessageEvents": null,  
"Links": null,  
"Status": 0,  
"HasRemoteParent": false,  
"DroppedAttributeCount": 0,  
"DroppedMessageEventCount": 0,  
"DroppedLinkCount": 0,  
"ChildSpanCount": 0
```



# Attributes & MessageEvents

```
"Attributes": [  
  {  
    "Key": "http.host",  
    "Value": {  
      "Type": "STRING",  
      "Value": "opentelemetry-instructor.glitch.me"  
    }  
  },  
  {  
    "Key": "http.status_code",  
    "Value": {  
      "Type": "INT64",  
      "Value": 200  
    }  
  }  
],  
"MessageEvents": [  
  {  
    "Message": "annotation within span",  
    "Attributes": null,  
    "Time": "2019-11-03T10:52:56.903914029Z"  
  }  
],
```



# Have you tested the fibonacci output yet?

- Hit the Go and Stop button!

**A sequence of numbers:**

Go

0, 1, 1, 2, 3, 5, ●

Stop



## Visualize using Jaeger

We've set up Jaeger to receive traces. First set in `.env`:

```
JAEGER_ENDPOINT=52.205.133.101:4317
```

```
jaegerEndpoint, _ := os.LookupEnv("JAEGER_ENDPOINT")
```

```
jaeger := otelgrpc.NewClient(otelgrpc.WithInsecure(),  
    otelgrpc.WithEndpoint(jaegerEndpoint))
```

```
jExporter, _ := otel.New(ctx, jaeger)
```

```
tp, err := sdktrace.NewProvider(  
    sdktrace.WithConfig(...),  
    sdktrace.WithSyncer(std), sdktrace.WithBatcher(jExporter)
```



## Having trouble?

`stdr.SetVerbosity(8)` should work in golang for enabling SDK debug.

`logging.basicConfig(level=10)` should work for python.

JS: uncomment `opentelemetry.diag.setLogger(...)`





## Go look for your trace!

The Jaeger visualization URL is at (notice the port number):

<http://52.205.133.101:16686/search>

Put in your `SERVICE_NAME` value into the service name, and search for your recent traces!

Ask your neighbor for their `SERVICE_NAME` and compare!



# Plugging in your own exporter

- Initialize a custom exporter with an API key
  - examples: Stackdriver, Lightstep, Honeycomb, etc.
  - A current list of vendors working with OpenTelemetry can be found at <https://opentelemetry.io/registry/>



# But can we use Jaeger to debug these?

---

How many times is `/fib?index=1` *inside* a call to `/fib?index=5`?

How much longer (*P99*) does it take to evaluate `/fib` of 5 compared to of 4?

Why is this so slow as *index* increases?



# **One possible vendor choice**

---

Learn to use Honeycomb's product UI!

# Some alternative choices

---

## Aspecto

<https://app.aspecto.io/user/login>

## Signoz

<https://signoz.io/teams/>

*(during workshop we explored these first via live screensharing, but without slides corresponding, before moving onto Honeycomb)*



# Get a Honeycomb account

<https://honeycomb.io/signup>



## Create account

No credit card required.

First name

Last name

Email address (work)

By signing up, I agree to the Honeycomb [Terms of Service](#).

or

Already have an account? [Sign in](#)



# Create a team

---

*Note to Existing Users:*

Go to **ui.honeycomb.io/teams**

At the bottom, find “Create Team”

## Create new team

Teams organize members' access to data and shared work history in Honeycomb. We recommend using your organization name, if you'll be inviting collaborators to your account later.

We've provided a default team name for you based on your email address. You can change it here if you'd prefer a different name.

Team name

Create Team



# Get a Honeycomb API Key

The first time you create a team as a new user, it takes you directly to your API Key.



We are waiting for you to send us data. Please follow the instructions below to send events from your application

Last checked 9:49:10 PM  
[Check now](#)

## Send data to test

Honeycomb's auto-instrumentation allows you to send basic data quickly using the OpenTelemetry industry standard, before adding custom context.

[View these instructions in Honeycomb Docs.](#)

API key

[Manage API keys](#)

.....

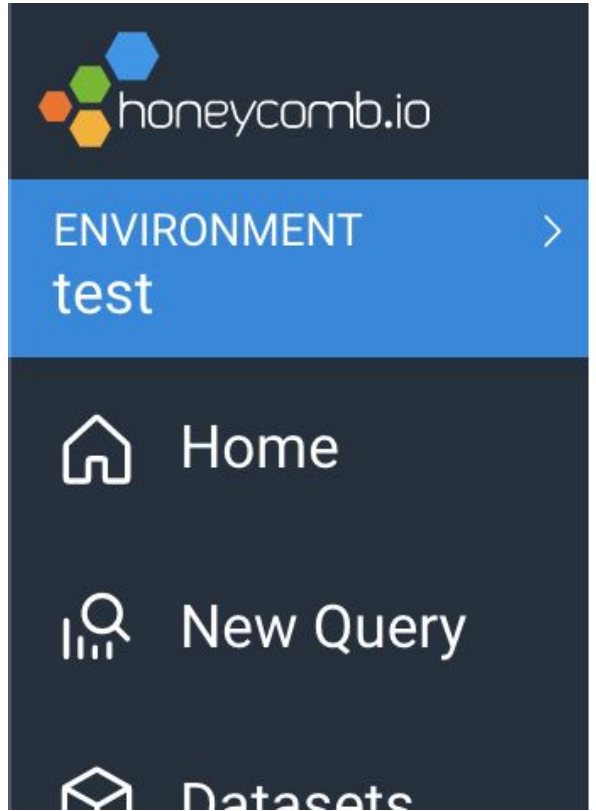




# Get a Honeycomb API Key

If you already have an account:

1. Click the environment selector at the top left
2. Choose **Manage environments** at the bottom.
3. View API Keys.

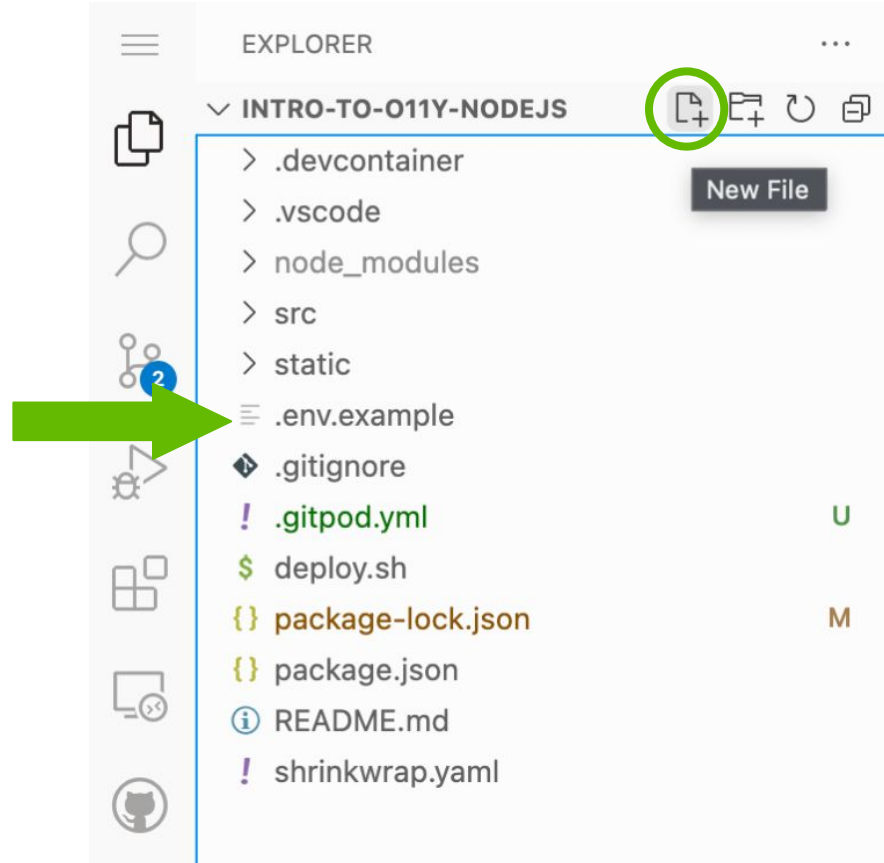


# Add credentials to send to Honeycomb

Provide your credential variables in the **.env** file.

Using the File Explorer:

- Create a new file named **.env**
- Copy the contents of **.env.example** into the **.env** file
- Replace the credential variables in the **.env** file as needed



# Add credentials to send to Honeycomb

---

Provide your credentials!

- HONEYCOMB\_API\_KEY identifies your team and environment.
  - Enter your API key from the Honeycomb UI
  
- SERVICE\_NAME groups your data (name it after your username)

⚙️ .env

```
1 # Honeycomb API key.  
2 export HONEYCOMB_API_KEY=your-api-key-goes-here  
3  
4 # Service name (opentelemetry standard field). Can b  
5 SERVICE_NAME=fib-microsvc  
6
```



# Re-run Your App

---

1) If your app is still running, stop it with `Ctrl+C`

2) Save your `.env` file.

3) In the terminal, run the app with:

```
./run
```

Then, click “Go” and “Stop” in the app.



# Use the Gitpod App again

---


Make the sequence of numbers appear.  
Select **Go!**

... and then select **Stop**, after 5-7 numbers.

Repeat as needed to generate data.

## A sequence of numbers:

Go

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 

Stop

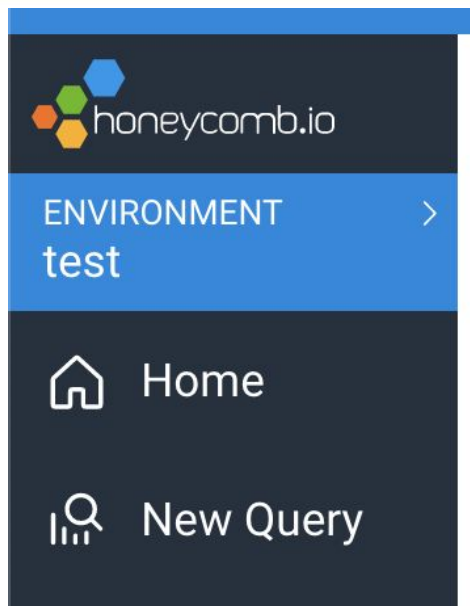


# Data should start flowing!

The blue box turns green

✓ Data received! Events from your application are now available to explore in Honeycomb. [Explore your data](#)

Click "Home"

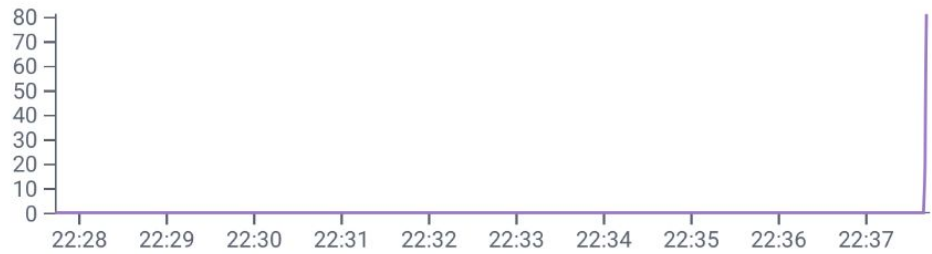


# Get Data In

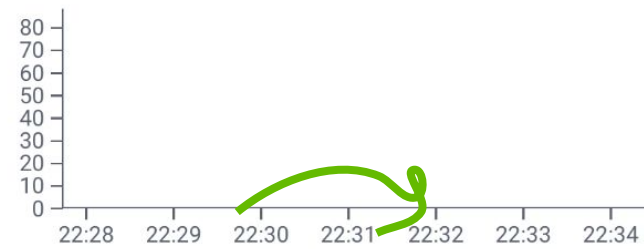
## Success!



Total spans ?



Latency ?



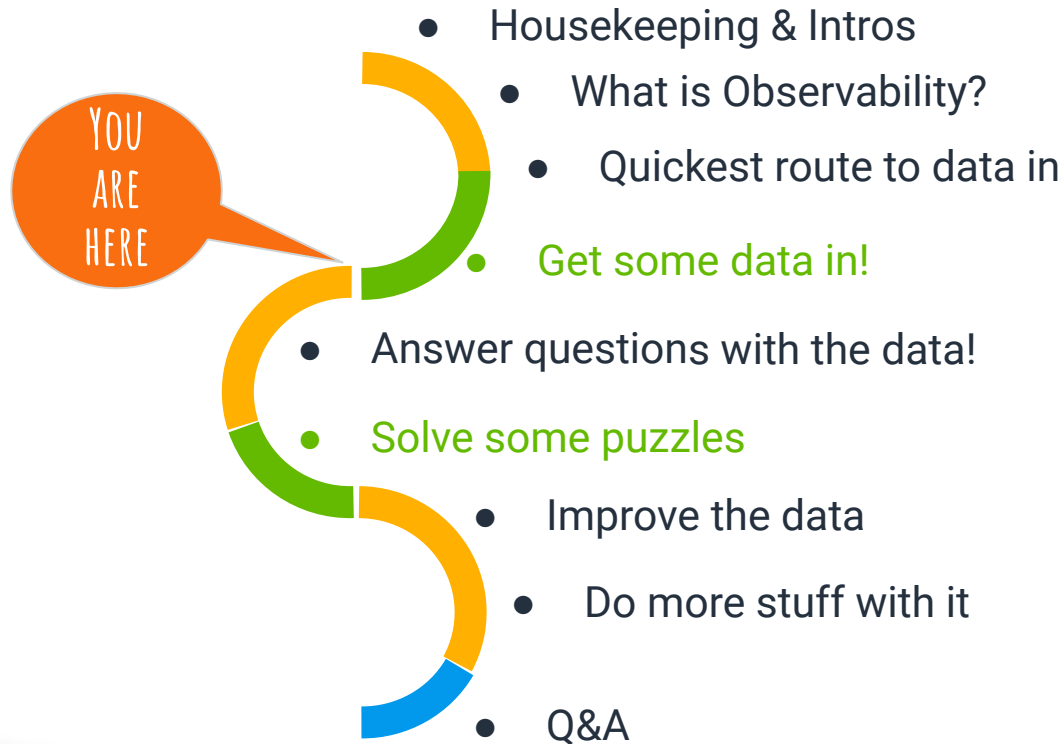
A spike of requests  
This is your goal

Error rate ?

Based on http status code

# Pre-break agenda

---





# Summary

---

In this section, we:

- Ran a demo app that is instrumented with OpenTelemetry
- Sent data to Jaeger
- Got a hosted service account
- Sent telemetry from the demo app to a hosted service
- Confirmed data ingestion from the hosted service's home screen



# Using data to answer questions

---

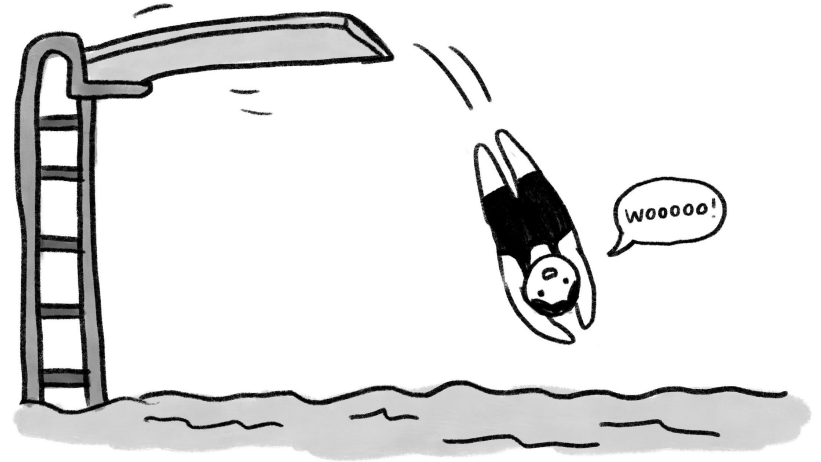
Learn to use a tracing product UI!

# Section Overview

---

In this section, we will:

- Learn about tracing UIs and tools
- Find out what makes our sequence of numbers app slow down



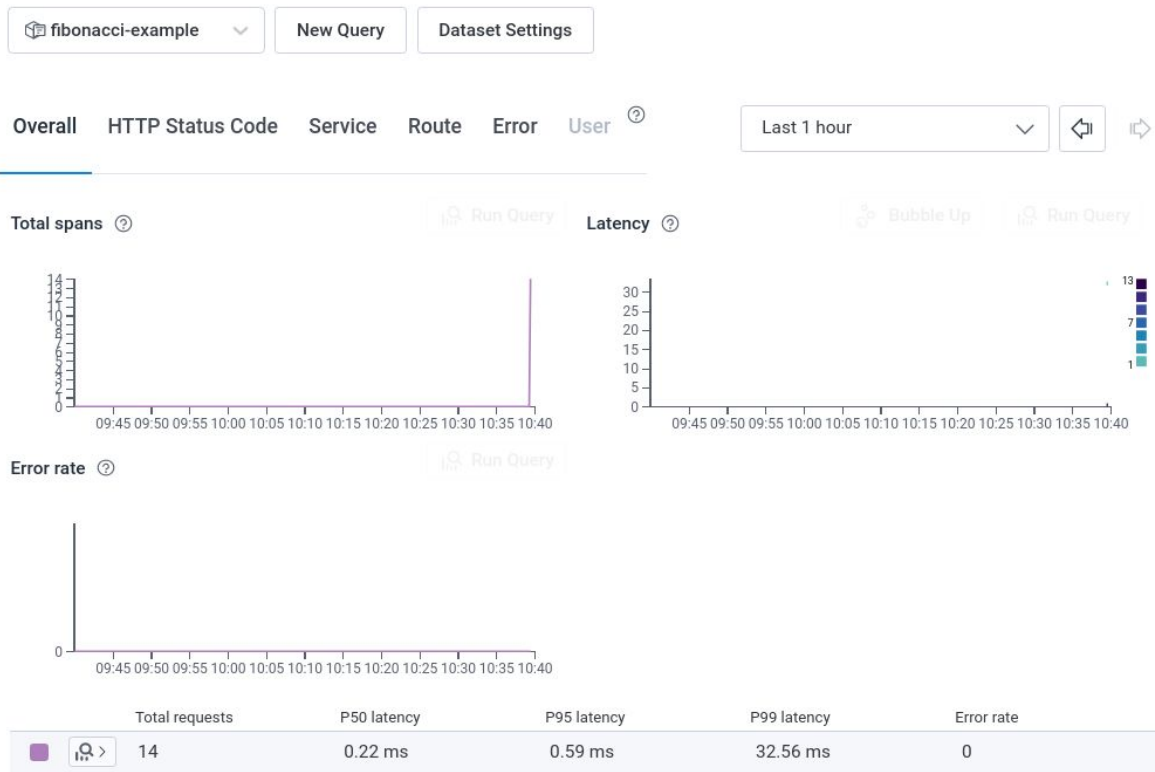
# Home

The basics:

Rate, error, duration graphs

You can always come to Home via the Honeycomb logo at top left.

## Home

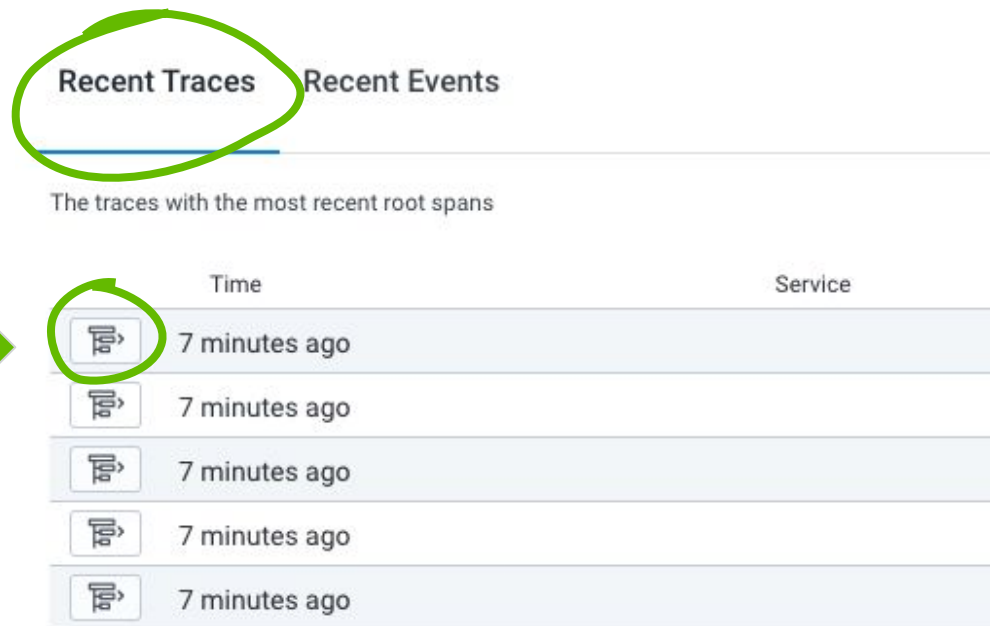


# Let's find & examine a trace!

After using the app...






Look for the trace in Recent Traces tab at the bottom of the Home screen.

Use the **View Trace** button on the row you want to examine further.



Recent Traces    Recent Events

The traces with the most recent root spans

	Time	Service
	7 minutes ago	
	7 minutes ago	
	7 minutes ago	
	7 minutes ago	
	7 minutes ago	



# Parts of a trace

Trace 041f1cbd0b6095e0bdc73889ac20120c at 2021-10-29 10:42:44

Rerun

Search spans

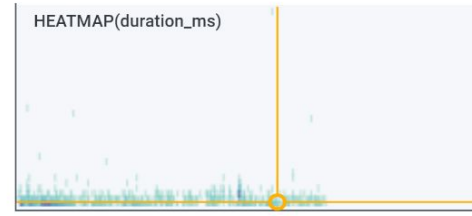


Fields

name	service_name	0s	0.005s	0.01s	0.015s	0.02136s
20 github.com/honey...ueryRunStatus-fm	poodle	21.36ms				
findByID	poodle	1.820ms				
launchdarkly.BoolVariations	poodle	0.2184ms				
launchdarkly.NumberVariation	poodle	23.9µs				
launchdarkly.JSONVariation	poodle	50.2µs				
launchdarkly.JSONVariation	poodle	36.0µs				
FindTeamBySlug	poodle	2.095ms				
FindWriteKeysByTeam	poodle	2.056ms				
CheckMembership	poodle	1.659ms				
GetRole	poodle	1.693ms				
ExternalAuthProvider	poodle	1.658ms				
launchdarkly.BoolVariations	poodle	0.1779ms				

poodle >  
github.com/honeycombio/launchdarkly.QueryRunStatus-fm

Distribution of span duration ?



Fields

trace|

str trace.span\_id  
df9fc50200e3ad1c

str trace.trace\_id  
041f1cbd0b6095e0bdc73889ac20120c

# Parts of a trace

Trace 041f1cbd0b6095e0bdc73889ac20120c at 2021-10-29 10:42:44

Rerun

Search spans

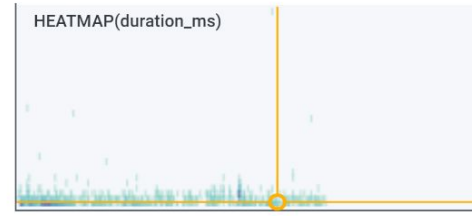


Fields

name	service_name	0s	0.005s	0.01s	0.015s	0.02136s
20 github.com/honey...ueryRunStatus-fm	poodle	span 21.36ms				
• findByID	poodle	span 1.820ms				
• launchdarkly.BoolVariations	poodle	span 0.2184ms				
• launchdarkly.NumberVariation	poodle	span 23.9µs				
• launchdarkly.JSONVariation	poodle	span 50.2µs				
• launchdarkly.JSONVariation	poodle	... 36.0µs				
• FindTeamBySlug	poodle	2.095ms				
• FindWriteKeysByTeam	poodle	2.056ms				
• CheckMembership	poodle	1.659ms				
• GetRole	poodle	1.693ms				
• ExternalAuthProvider	poodle	1.658ms				
• launchdarkly.BoolVariations	poodle	0.1779ms				

poodle > github.com/honeycombio/o...andler).QueryRunStat fm

Distribution of span duration ?



Fields

trace|

str trace.span\_id  
df9fc50200e3ad1c

str trace.trace\_id  
041f1cbd0b6095e0bdc73889ac20120c

# Parts of a trace

Trace 041f1cbd0b6095e0bdc73889ac20120c at 2021-10-29 10:42:44

Rerun

poodle >  
github.com/honeycombio/o...andler).QueryRunStat  
fm

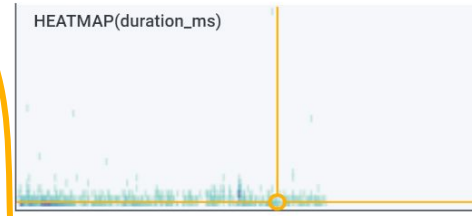
Search spans



Fields

name	service_name	0s	0.005s	0.01s	0.015s	0.02136s	
20 github.com/honey...ueryRunStatus-fm	poodle	root span					21.36ms
findByID	poodle	1.820ms					
launchdarkly.BoolVariations	poodle	0.2184ms					
launchdarkly.NumberVariation	poodle	23.9µs					
launchdarkly.JSONVariation	poodle	50.2µs					
launchdarkly.JSONVariation	poodle	36.0µs					
FindTeamBySlug	poodle	2.095ms					
FindWriteKeysByTeam	poodle	2.056ms					
CheckMembership	poodle	1.659ms					
GetRole	poodle	1.693ms					
ExternalAuthProvider	poodle	1.658ms					
launchdarkly.BoolVariations	poodle	0.1779ms					

Distribution of span duration ?



Fields

```
trace|
str trace.span_id
df9fc50200e3ad1c
str trace.trace_id
041f1cbd0b6095e0bdc73889ac20120c
```



# Parts of a trace

Trace 041f1cbd0b6095e0bdc73889ac20120c at 2021-10-29 10:42:44

Rerun

poodle >  
FindTeamBySlug

Search spans

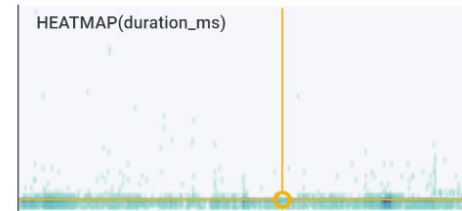


Fields

name  service\_name  0s 0.005s 0.01s 0.015s 0.02136s

name	service_name	0s	0.005s	0.01s	0.015s	0.02136s
20 github.com/honey...ueryRunStatus-fm	poodle	21.36ms				
findByID	poodle	1.820ms				
launchdarkly.BoolVariations	poodle	0.2184ms				
launchdarkly.NumberVariation	poodle	23.9µs				
launchdarkly.JSONVariation	poodle	50.2µs				
launchdarkly.JSONVariation	poodle	36.0µs				
FindTeamBySlug	poodle	2.095ms				
FindWriteKeysByTeam	poodle	2.056ms				
CheckMembership	poodle	1.659ms				
GetRole	poodle	1.693ms				
ExternalAuthProvider	poodle	1.658ms				
launchdarkly.BoolVariations	poodle	0.1770ms				

Distribution of span duration ?



Fields

trace.

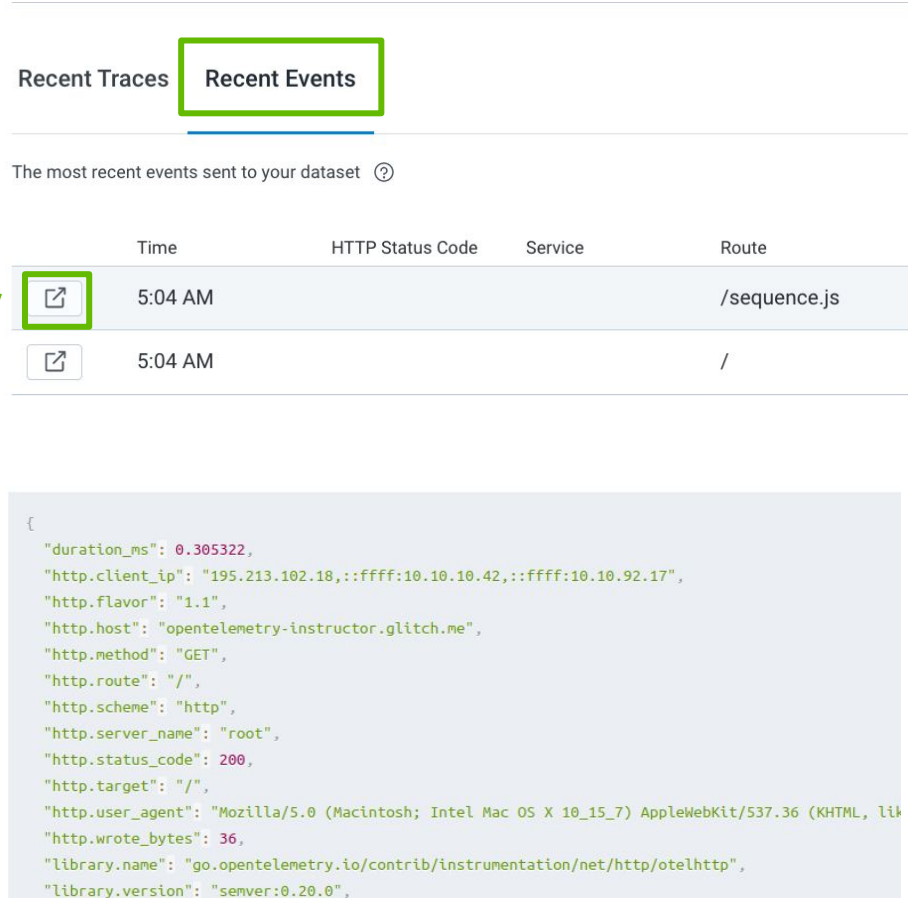
trace.parent\_id  
df9fc50200e3ad1c

trace.span\_id  
4da124d8596548ff

trace.trace\_id  
041f1cbd0b6095e0bdc73889ac20120c



# Raw data view

Also, the **Recent Events** tab at bottom of **Home** view shows raw JSON for events.



Recent Traces **Recent Events**

The most recent events sent to your dataset [?](#)

	Time	HTTP Status Code	Service	Route
	5:04 AM			/sequence.js
	5:04 AM			/

```
{
  "duration_ms": 0.305322,
  "http.client_ip": "195.213.102.18,::ffff:10.10.10.42,::ffff:10.10.92.17",
  "http.flavor": "1.1",
  "http.host": "opentelemetry-instructor.glitch.me",
  "http.method": "GET",
  "http.route": "/",
  "http.scheme": "http",
  "http.server_name": "root",
  "http.status_code": 200,
  "http.target": "/",
  "http.user_agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/90.0.4431.97 Safari/537.36",
  "http.wrote_bytes": 36,
  "library.name": "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp",
  "library.version": "semver:0.20.0",
```



# Raw data view

Sometimes you need to look at spans as wide events/structured logs!

Honeycomb supports showing a table of spans and fields on those spans.

Go to the **Raw Data** tab.






Results   BubbleUp   Traces   **Raw Data**    Graph Settings

Sep 23 2021, 2:18 PM – Sep 23 2021, 2:52 PM (Granularity: 5 sec)

**DOWNLOAD**  
[CSV](#) | [JSON](#)  
 max rows returned: 1,000

**FIELDS**  
 All (30)

- Timestamp
- duration\_ms
- http.client\_ip
- http.flavor
- http.host
- http.method
- http.route
- http.status\_code
- http.status\_text
- http.target
- http.url
- http.user\_agent
- library.name
- library.version
- name
- net.host.ip
- net.host.name
- net.host.port
- net.peer.ip
- net.peer.port
- net.transport

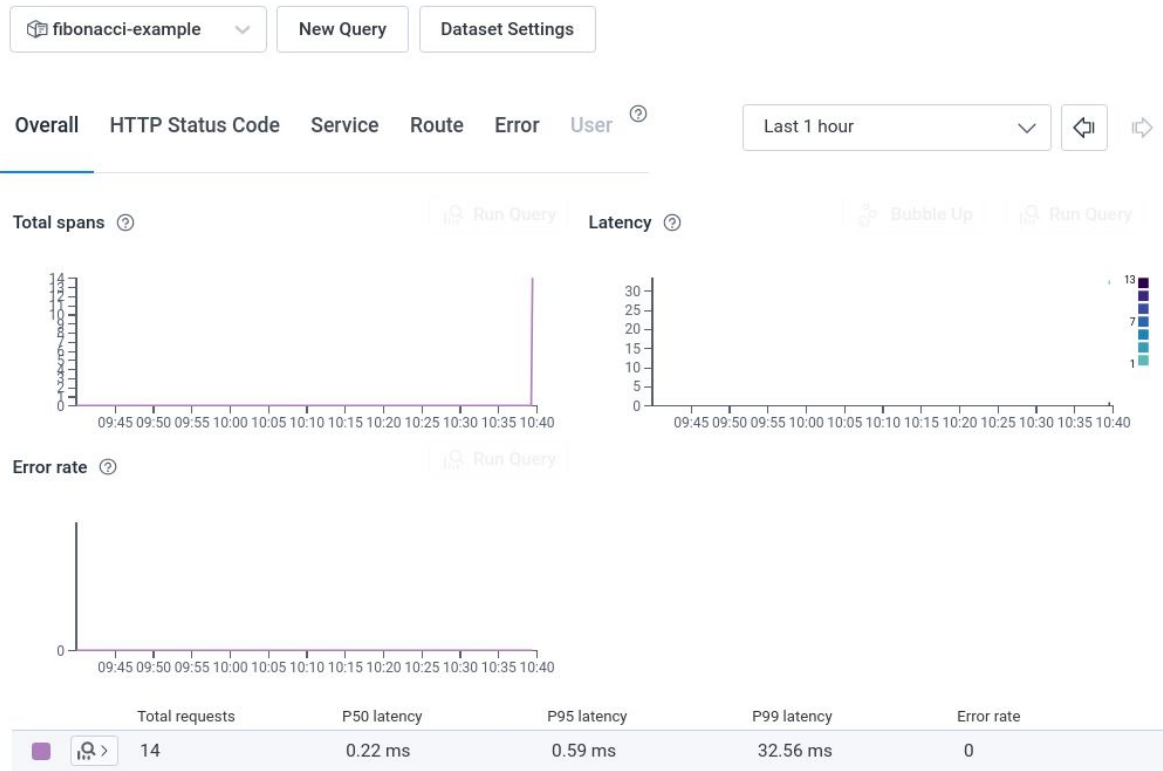
	Timestamp <span>→ UTC</span>	duration_ms	http.client_ip	http.flavor	http.host
	2021-09-23 14:38:01.636	1,391.0551	24.125.233.208	1.1	intro-to-o11y-nodejs.glitch.
	2021-09-23 14:38:00.579	1,007.47981	24.125.233.208	1.1	intro-to-o11y-nodejs.glitch.
	2021-09-23 14:38:00.039	479.43142	24.125.233.208	1.1	intro-to-o11y-nodejs.glitch.
	2021-09-23 14:37:59.619	367.54432	24.125.233.208	1.1	intro-to-o11y-nodejs.glitch.
	2021-09-23 14:37:59.307	268.29466	24.125.233.208	1.1	intro-to-o11y-nodejs.glitch.

# Hello data

Click the house to get back to Home

Click the first graph to zoom into it.

## Home



# Interactive Querying

Your service

A guess at "top-level spans"



sequence-of-numbers

Add to Board

Share

Last 10 minutes

## Total spans

<u>VISUALIZE</u> COUNT	<u>WHERE</u> OR(NOT(EXISTS(\$"trace.parent_id")), EQUALS(\$"span.kind", "server")) = true	<u>GROUP BY</u> None; don't segment	...
<u>+ ORDER BY</u>	<u>+ LIMIT</u>	<u>+ HAVING</u>	

Run Query  
Run a few seconds ago

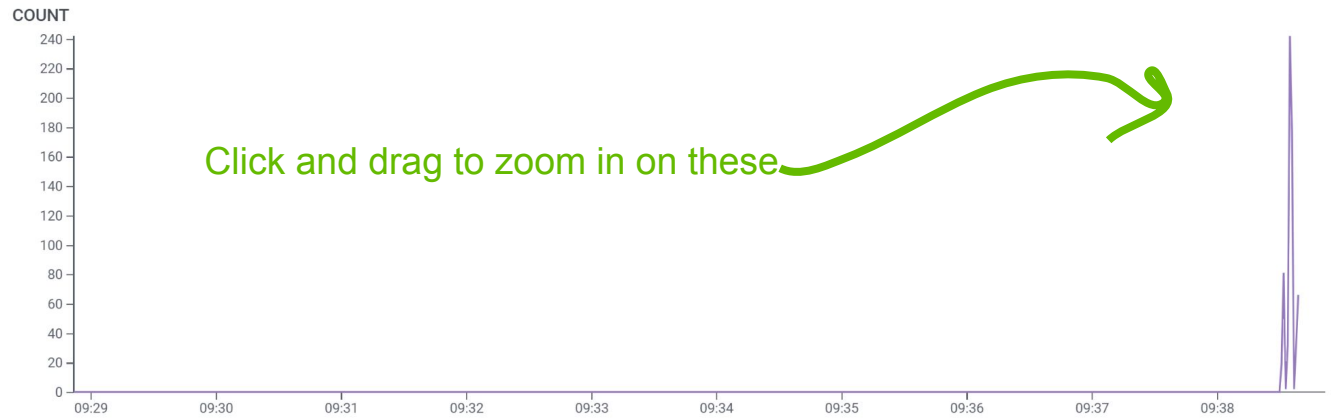
Results BubbleUp Traces Raw Data

Compare to

10 minutes prior

Graph Settings

May 18 2022, 9:28:52 AM - May 18 2022, 9:38:52 AM (Granularity: 1 sec)



# Total spans

**VISUALIZE** **WHERE** AND **GROUP BY** ...

**VISUALIZE**: x COUNT

**WHERE**: x OR(NOT(EXISTS(\$"trace.parent\_id")), EQUALS(...))

**GROUP BY**: attribute(s)

+ ORDER BY + LIMIT + HAVING

Run Query

Clear | Cancel

Results BubbleUp Traces Raw Data

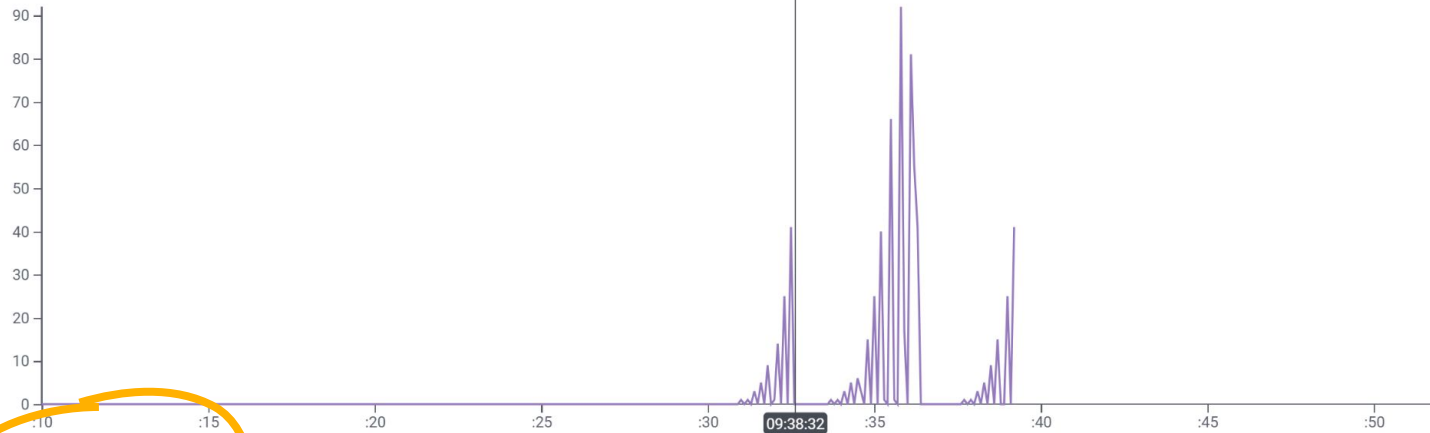
Compare to

42 seconds prior

Graph Settings

May 18 2022, 9:38:10 AM – May 18 2022, 9:38:52 AM (Granularity: 100 ms)

COUNT



Click into the "where" clause to change it

Click the X to remove the filter



COUNT

653

# Find only the root spans (aka is\_root)

WHERE AND ▾

trace.parent\_id does-not-exist

trace.parent\_id does-not-exist

+ LIMIT

The image shows a query builder interface. The 'WHERE' clause is selected, and the text 'trace.parent\_id does-not-exist' is entered. A red squiggly underline is under 'trace.parent\_id', indicating a validation error. A dropdown menu is open below the input field, showing the same text 'trace.parent\_id does-not-exist'. Below the input field, there is a '+ LIMIT' option.



VISUALIZE

COUNT

WHERE

trace.parent\_id does-not-exist

GROUP BY

None; don't segment

...

Run Query

Run a few seconds ago

+ ORDER BY

+ LIMIT

+ HAVING

Results

BubbleUp

Traces

Raw Data

Compare to

42 seconds prior



Graph Settings

May 18 2022, 9:38:10 AM – May 18 2022, 9:38:52 AM (Granularity: 100 ms)

COUNT

1.0  
0.9  
0.8  
0.7  
0.6  
0.5  
0.4  
0.3  
0.2  
0.1  
0

:10 :15 :20 :25 :30 :35 :40 :45 :50

Push to run the new query

COUNT

27



© 2022



# Add more visualizations!

✓ Define **aggregate function(s)** to visualize events

**VISUALIZE**

× COUNT  
avg(duration\_ms)

AVG(duration\_ms)  
RATE\_AVG(duration\_ms)

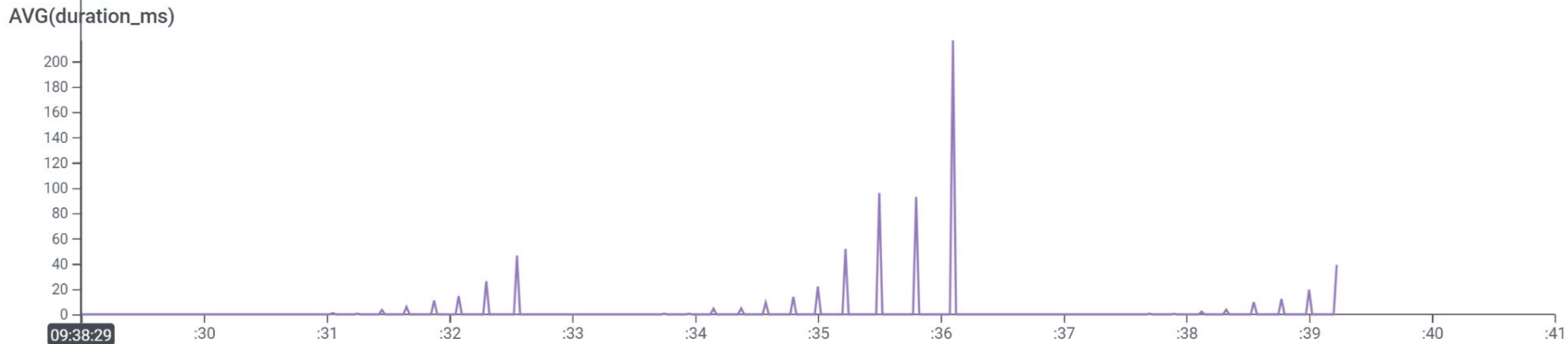
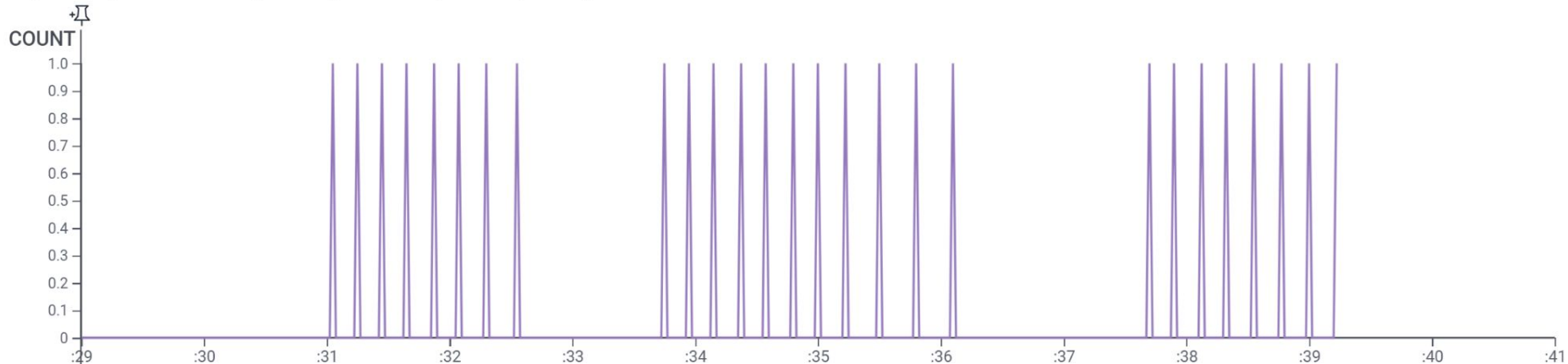
**WHERE**

× trace.parent\_id

**+ LIMIT**



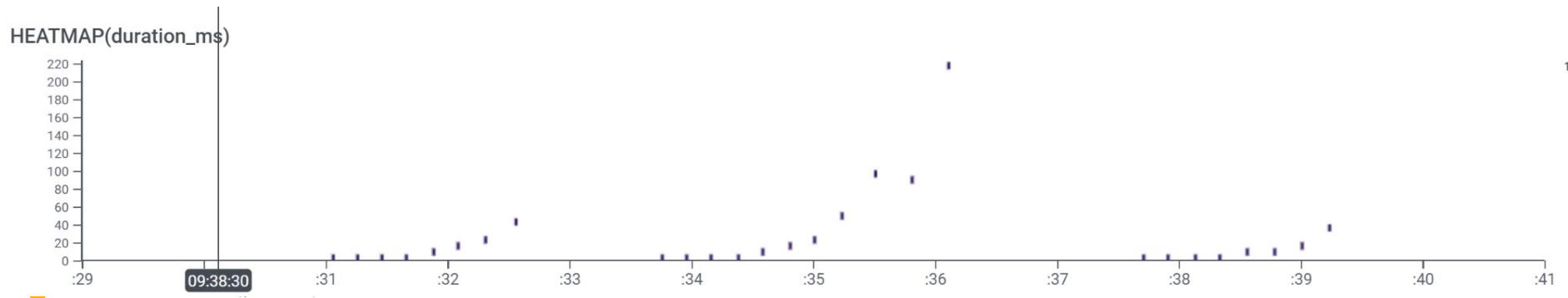
May 18 2022, 9:38:29 AM – May 18 2022, 9:38:41 AM (Granularity: 25 ms)



# Now add a heatmap

Under VISUALIZE, add  
HEATMAP(duration\_ms)  
then push Run Query.  
It doesn't look like much...

```
VISUALIZE  
-----  
COUNT  
AVG(duration_ms)  
HEATMAP(duration_ms)
```



# Generating a little more data...

In your app, select **Go!** ... and then make it **Stop** after 5-7 numbers several times.

## A sequence of numbers:

Go

0, 1, 1, 2, 3, 5, 8, 13, ●

Stop



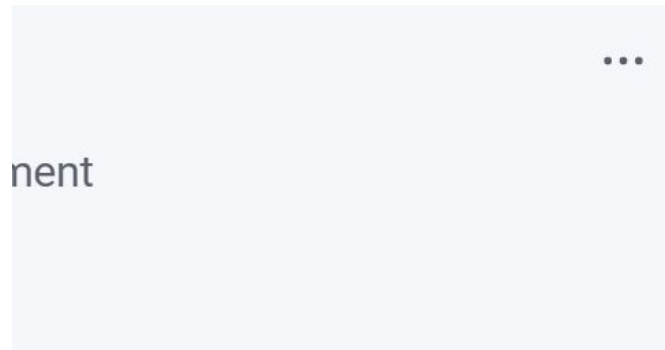
# Run the query again

Change the timeframe back to  
“Last 10 minutes”

Last 10 minutes



Then zoom in again on the part of  
the timeline with data.

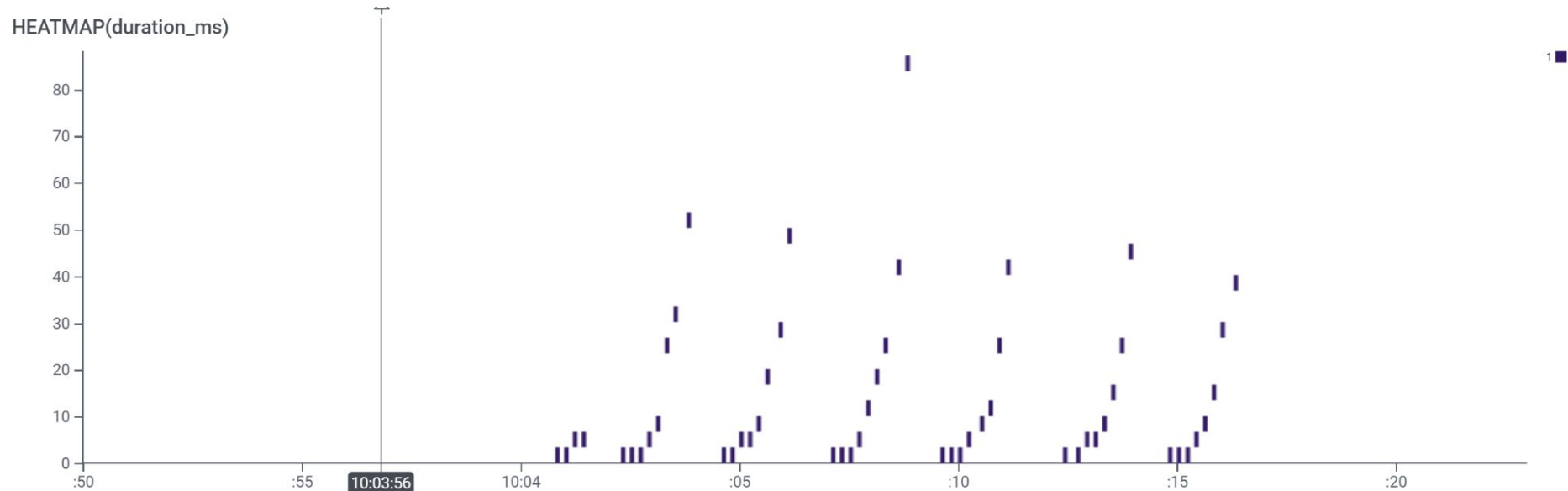


**Run Query**

Run a few  
seconds ago



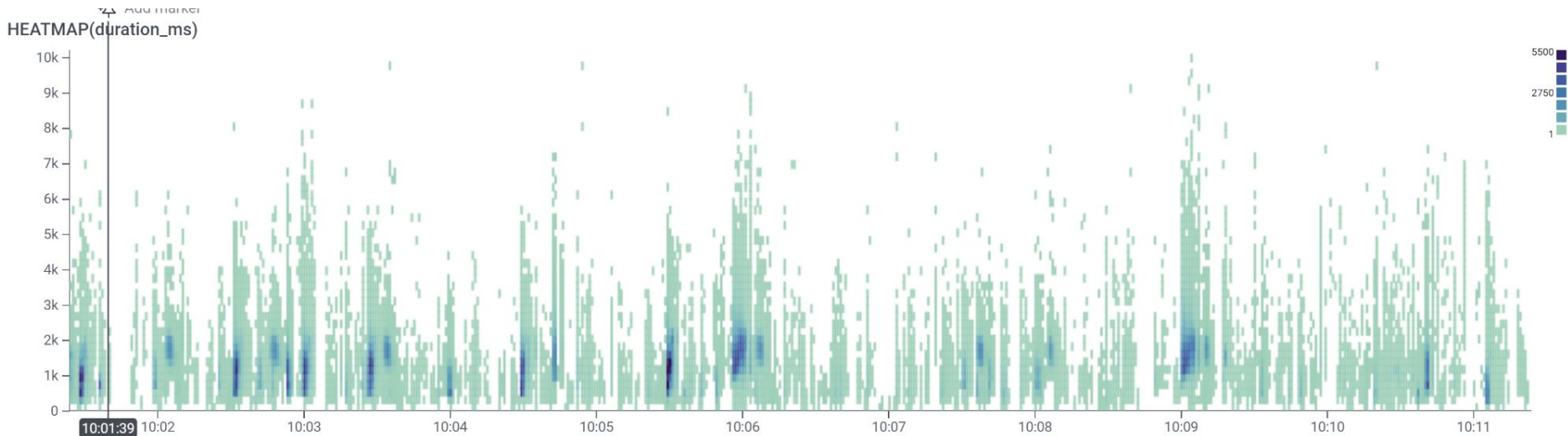
# The higher the dot, the slower the request.



# Heatmaps

Each area on the heatmap corresponds to a time of day, and a slowness of requests. (higher is slower)

The histogram's color shows the density in that area: how many requests took this long, at this time. (darker is more of them)



# GROUP BY and HEATMAP

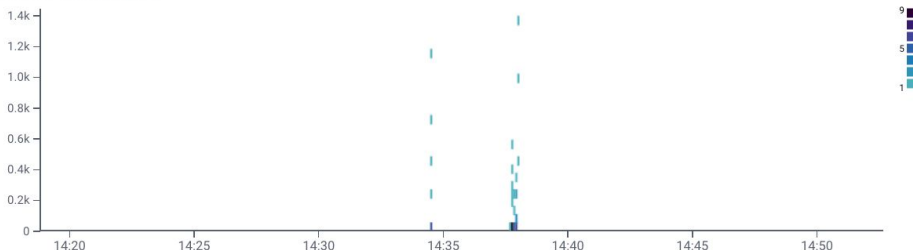
When in the **Results** tab...

You can hover over a group of results to highlight just those entries in the heatmap.

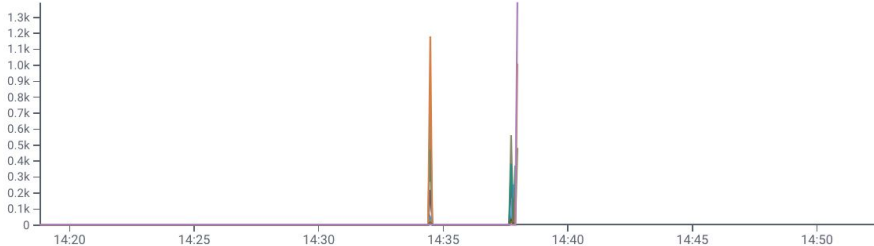
Results BubbleUp Traces Raw Data  Compare to Previous time range ⌵  Graph Settings



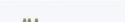

Sep 23 2021, 2:18 PM – Sep 23 2021, 2:52 PM (Granularity: 5 sec)

HEATMAP(duration\_ms)



P99(duration\_ms)

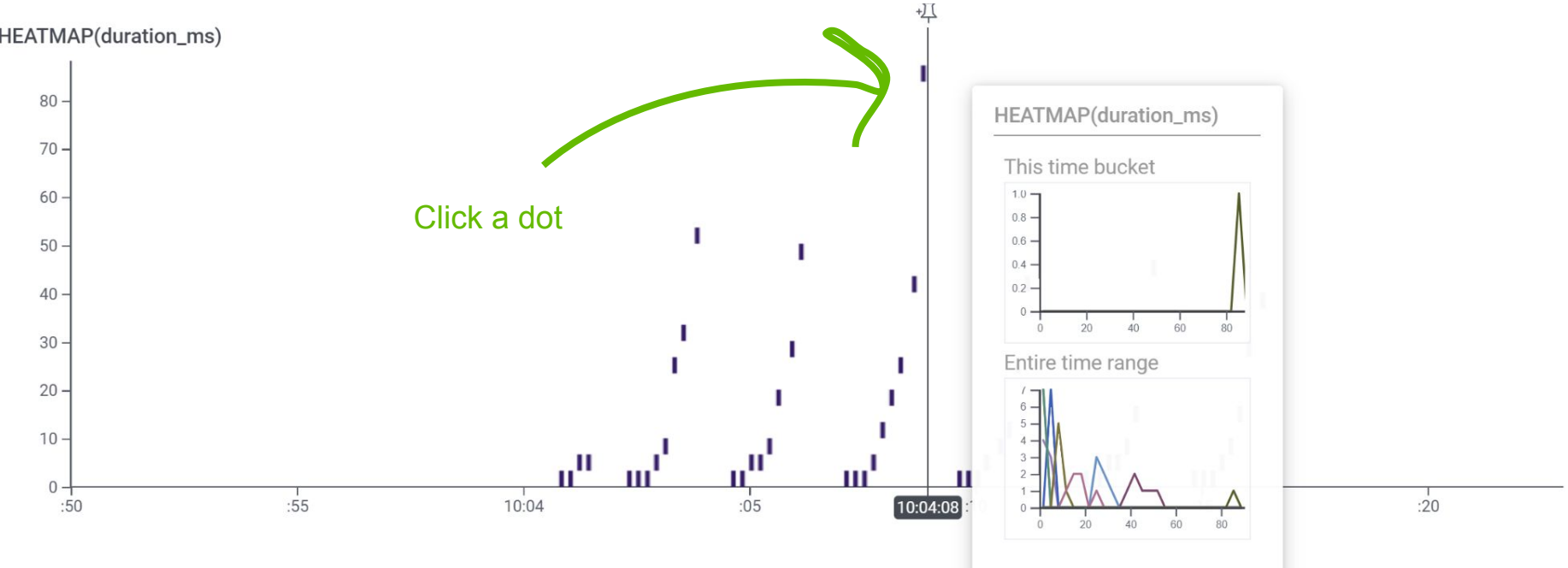


http.url	HEATMAP(duration_ms)	P99(duration_ms)
<a href="http://intro-to-o11y-nodejs.glitch.me/fib?index=9">http://intro-to-o11y-nodejs.glitch.me/fib?index=9</a>		1,391.0551
<a href="http://intro-to-o11y-nodejs.glitch.me/fib?index=8">http://intro-to-o11y-nodejs.glitch.me/fib?index=8</a>		1,176.96333
<a href="http://intro-to-o11y-nodejs.glitch.me/fib?index=7">http://intro-to-o11y-nodejs.glitch.me/fib?index=7</a>		728.68845
<a href="http://intro-to-o11y-nodejs.glitch.me/fib?index=6">http://intro-to-o11y-nodejs.glitch.me/fib?index=6</a>		470.62451





# Click on the heatmap to see a trace



# See an example!

← Trace d6ba7420b6bbd9b8e094dff4784e07d7 at 2022-05-18 10:04:08

Rerun

Search spans



0 spans with errors

Fields

name	service.name	0s	0.02s	0.04s	0.06s	0.0848s	
5 GET /fib	sequence-of-numbers	84.80ms					
• middleware - query	sequence-of-numbers	38.4µs					
• middleware - expressInit	sequence-of-numbers	23.6µs					
• request handler - /fib	sequence-of-numbers	2.82µs					
1 HTTP GET	sequence-of-numbers	59.75ms					
5 GET /fib	sequence-of-numbers	59.06ms					
• middleware - query	sequence-of-numbers	23.3µs					
• middleware - expressInit	sequence-of-numbers	17.4µs					
• request handler - /fib	sequence-of-numbers	2.05µs					
1 HTTP GET	sequence-of-numbers	43.70ms					
5 GET /fib	sequence-of-numbers	42.48ms					
• middleware - query	sequence-of-numbers	22.0µs					

## sequence-of-numbers > GET /fib

Distribution of span duration ?



### Fields

Filter fields and values in span

Timestamp  
2022-05-18T15:04:08.873779968Z

duration\_ms  
84.803072

http.flavor  
1.1

http.host

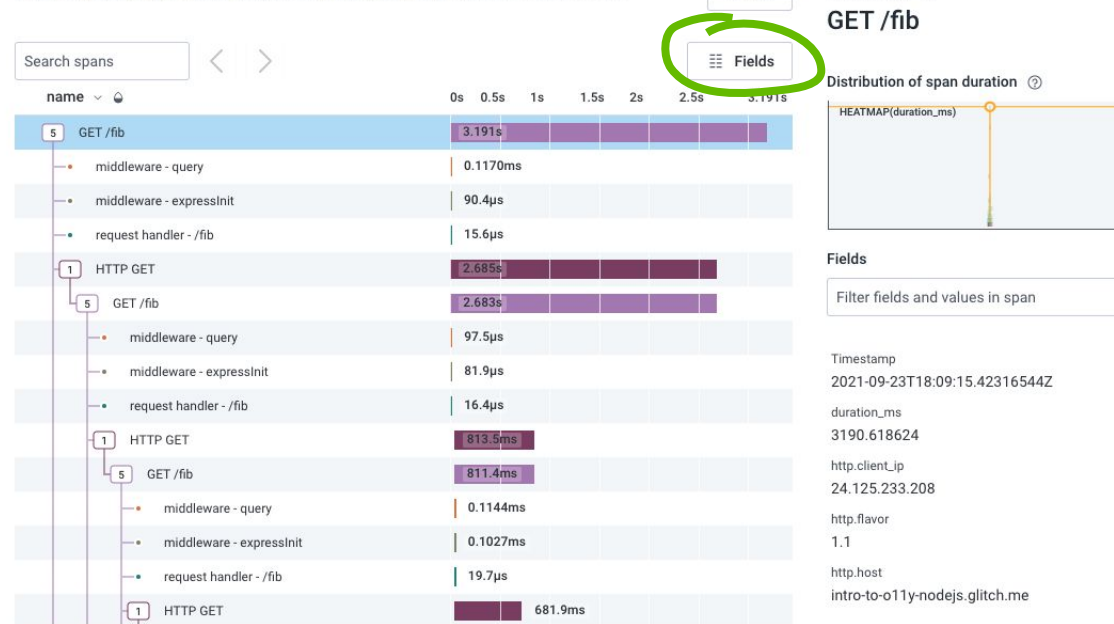
# Examine a trace

To customize:

Select the **Fields** button in the upper right corner and enter **http.target**.

**NOTE:** For Python and Go implementations, use **http.target**  
For others, use **http.url**

Trace a3d5681083cbde2abc55213852fc6eda at 2021-09-23 14:09:15



# Examine a trace

To customize:

Select the **Fields** button in the upper right corner and enter **http.target** (for Python and Go)

The display will update to show a **http.target** column.

Trace a3d5681083cbde2abc55213852fc6eda at 2021-09-23 14:09:15

Search spans < >

Fields

name < > http.target

0s 2s 3.191s

name	duration	http.target
GET /fib	3.191s	/fib
middleware - query	0.1170ms	
middleware - expressInit	90.4µs	
request handler - /fib	15.6µs	
HTTP GET	2.685s	/fib?index=7
GET /fib	2.683s	/fib
middleware - query	97.5µs	
middleware - expressInit	81.9µs	
request handler - /fib	16.4µs	
HTTP GET	813.5ms	/fib?index=6
GET /fib	811.4ms	/fib
middleware - query	0.1144ms	
middleware - expressInit	0.1027ms	
request handler - /fib	19.7µs	
HTTP GET	681.9ms	/fib?index=5

unknown > GET /fib

Distribution of span duration

HEATMAP(duration\_ms)

Fields

Filter fields and values in span

Timestamp  
2021-09-23T18:09:15.42316544Z

duration\_ms  
3190.618624

http.client\_ip  
24.125.233.208

http.flavor  
1.1

http.host  
intro-to-o11y-nodejs.glitch.me



# Search a trace

Search for spans that contain a phrase anywhere in their fields

The screenshot shows a search bar with the text "index=" and a dropdown menu set to "name". To the right of the search bar, it says "Selected: 1 of 133 spans found." Below the search bar is a table of spans. The first span is highlighted in blue and has a tree view expanded below it. The tree view shows a root span labeled "5" with a sub-span labeled "1" (HTTP GET) and another sub-span labeled "5" (GET /fib). The "5" span has three children: "middleware - query", "middleware - expressNit", and "request handler - /fib". The "1" span has one child: "HTTP GET". The "5" span has one child: "GET /fib". The "5" span has three children: "middleware - query", "middleware - expressNit", and "request handler - /fib". The "1" span has one child: "HTTP GET". The "5" span has one child: "GET /fib". The "5" span has three children: "middleware - query", "middleware - expressNit", and "request handler - /fib".

	name	http.url	
...	5 GET /fib	http://localhost:3000/fib?index=8	8
	middleware - query		3
	middleware - expressNit		2
	request handler - /fib		2
1	HTTP GET	http://127.0.0.1:3000/fib?index=7	5
5	GET /fib	http://127.0.0.1:3000/fib?index=7	5
	middleware - query		2
	middleware - expressNit		1
	request handler - /fib		2
1	HTTP GET	http://127.0.0.1:3000/fib?index=6	4
5	GET /fib	http://127.0.0.1:3000/fib?index=6	4
	middleware - query		2



# Search a span

On the right, search all the fields in the selected span.

## Fields

http.url

http://localhost:3000/fib?index=8

http.route

/fib

http.target

/fib

name

GET /fib



# Query by a value

After searching for a field (say, trace.trace\_id)

Click the three dots next to the value to get a handy menu

“Show only where field is value” gets you to a query

## Fields

trace

trace.span\_id

143911d7ff0976dd

trace.trace\_id

d6ba7.

Group by field

Show only where field is value

Show only where field is not value

Copy field name

Copy value



<b>VISUALIZE</b> COUNT HEATMAP(duration_ms)	<b>WHERE</b> service_name = sequence-of-numbers trace.trace_id = d6ba7420b6bbd9b8e094dff4784e07d7	<b>GROUP BY</b> None; don't segment	...
<b>ORDER BY</b> COUNT desc	<b>LIMIT</b> None	<b>HAVING</b> None; include all results	

**Run Q**  
Run a  
second:

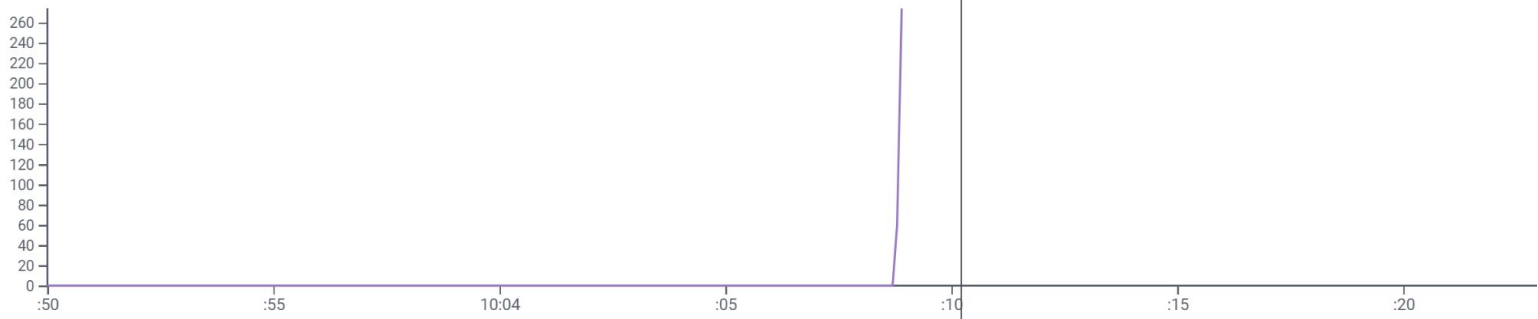
Results BubbleUp Traces Raw Data

Compare to 33 seconds prior

Graph Set

May 18 2022, 10:03:50 AM – May 18 2022, 10:04:23 AM (Granularity: 100 ms)

COUNT



HEATMAP(duration\_ms)





# Try customizing with GROUP BY

Add “name” to the GROUP BY then Run Query

**VISUALIZE** **WHERE** AND **GROUP BY** **Run Query**

× COUNT × HEATMAP(duration\_ms)

× service.name = sequence-of-numbers  
× trace.trace\_id = d6ba7420b6bbd9b8e094dff47...

× name

Clear | Cancel

Then scroll down to the table of results

name	COUNT	HEATMAP(duration_ms)
middleware - expressInit	67	
middleware - query	67	
GET /fib	67	
request handler - /fib	67	
HTTP GET	66	

# Add visualizations to see more results

In VISUALIZE, add:

P99(duration\_ms)

Then add that to the ORDER BY too.

Now see the slowest spans in the result table!

**VISUALIZE**

× COUNT × P99(duration\_ms)

---

**ORDER BY**

× P99(duration\_ms) desc

name ▾

GET /fib	67	59.06406
HTTP GET	66	43.69894
middleware - query	67	0.05222
middleware - expressInit	67	0.04173
request handler - /fib	67	0.00333

# Putting it all together: custom queries

VISUALIZE

WHERE

GROUP BY

ORDER BY

LIMIT

HAVING

<u>VISUALIZE</u> COUNT, SUM(..., HEATMAP(...	<u>WHERE</u> attribute = value, attribute exists...	AND ▾	<u>GROUP BY</u> attribute(s)
<u>+ ORDER BY</u>	<u>+ LIMIT</u>		<u>+ HAVING</u>

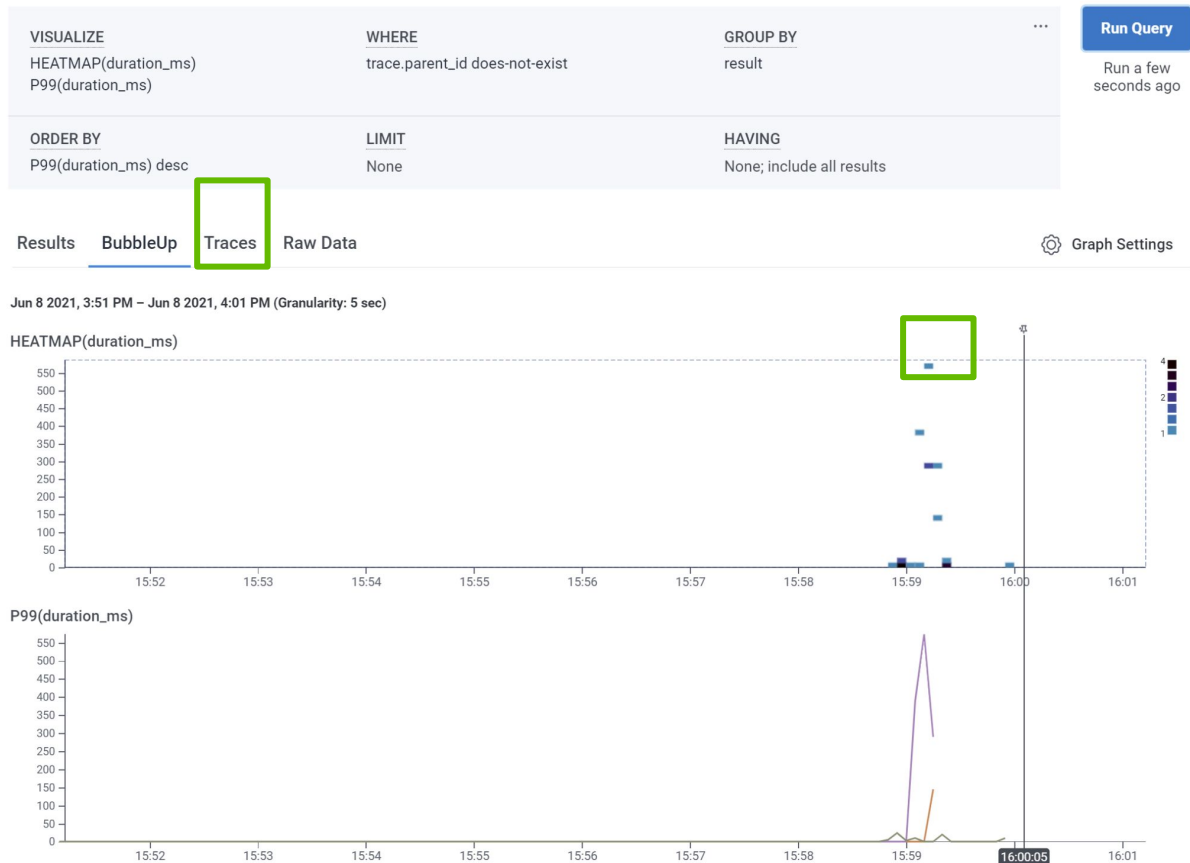


# Find an especially slow trace

Go to the **Traces** tab now.

Click on the slowest trace in the heatmap.

The next screen will be your detailed Trace display.



# Summary

---

In this section, we:

- Examined a trace for a specific request
- Identified outliers using Heatmaps and exemplars
- Customized our queries and used multiple visualizations
- Inspected our raw data and found the slowest traces



# Puzzles for the class

---

How many times is `/fib?index=1` *inside* a call to `/fib?index=5`?

How much longer (P99) does it take to evaluate `/fib` of 5 compared to of 4?

Why is this so slow as *index* increases?



# Puzzles for the class

---

**How many times is `/fib?index=1` inside a call to `/fib?index=5`?**

Hint: Find an example trace.

Use “WHERE http.url CONTAINS index=5”

Then, choose “Traces” tab.

**How much longer (*P99*) does it take to evaluate `/fib` of 5 compared to of 4?**

Hint: Use both

VISUALIZE P99(duration\_ms)

GROUP BY http.url

**Why is this so slow as index increases?**

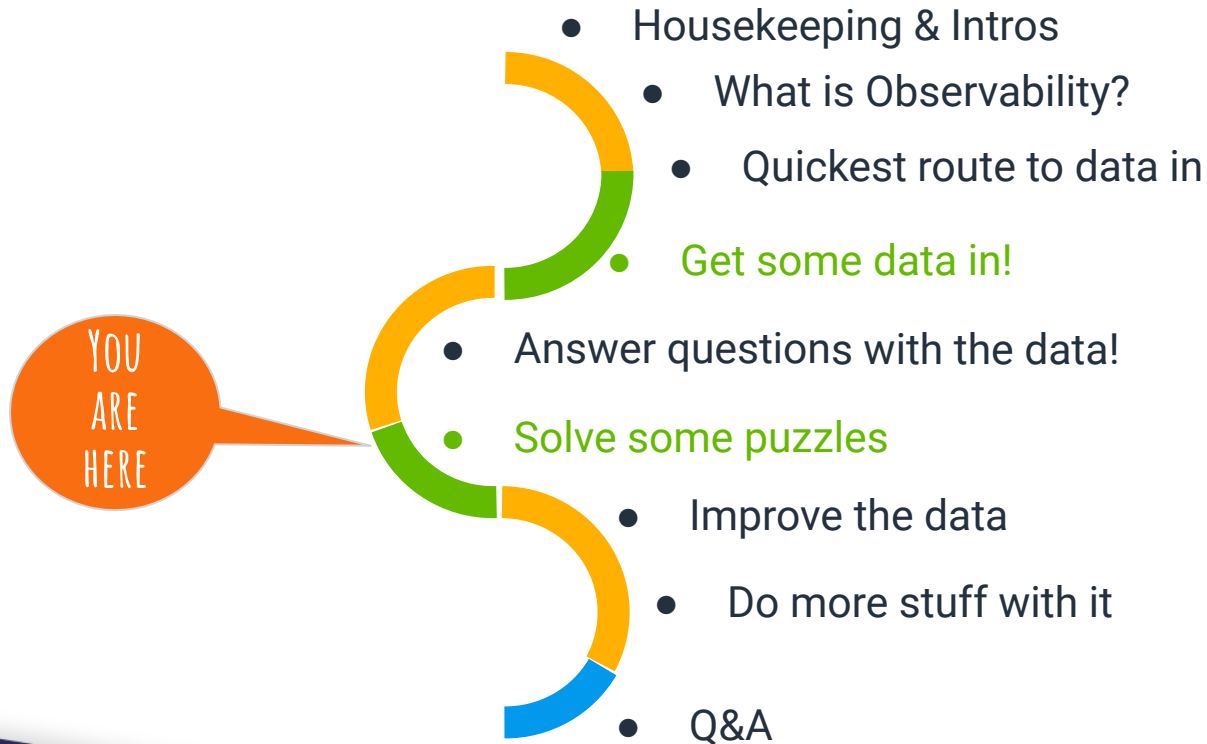
Don't forget to select the **Run Query** button!

There is **more than one method** to arrive at the answer.



# Pre-break agenda

---





# Puzzles for the class

---

How many times is `/fib?index=1` inside a call to `/fib?index=5`?

**Hint:** Find an example trace.

Use “WHERE `http.url` CONTAINS `index=5`”

Then, choose “Traces” tab.

In Query Builder, start with:  
VISUALIZE COUNT

**A Solution:**

WHERE `trace.parent_id` does-not-exist

GROUP BY `http.url`

[For Python and Go apps, use `http.target`]

Focus on `/fib?index=5`.

Below the graph, **select** [...] button in `http.url` column in the `/fib?index=5` row.

**Choose** "Show only where `http.url = http://intro-to-o11y-nodejs.glitch.me/fib?index=5`

**Select any point on graph** to see the Trace Detail View and **count** how many `/fib?index=1` you see.



# Puzzles for the class

---

How much longer (**P99**) does it take to evaluate /fib of 5 compared to of 4?

**Hint** - Use Query Builder & add to your query:  
VISUALIZE P99(duration\_ms)  
GROUP BY http.url

[For Python and Go apps, use *http.target* instead.]

**A Solution:**

**Return to your query builder display** by selecting the arrow next to the trace name in the upper left corner.

**Add to your query:**

VISUALIZE P99(duration\_ms)  
GROUP BY http.url

Two graphs now appear.

In the table of results, sort and compare duration\_ms for /fib of 5 vs. /fib of 4.



# Puzzles for the class

---

**Why is this so slow as index increases?**

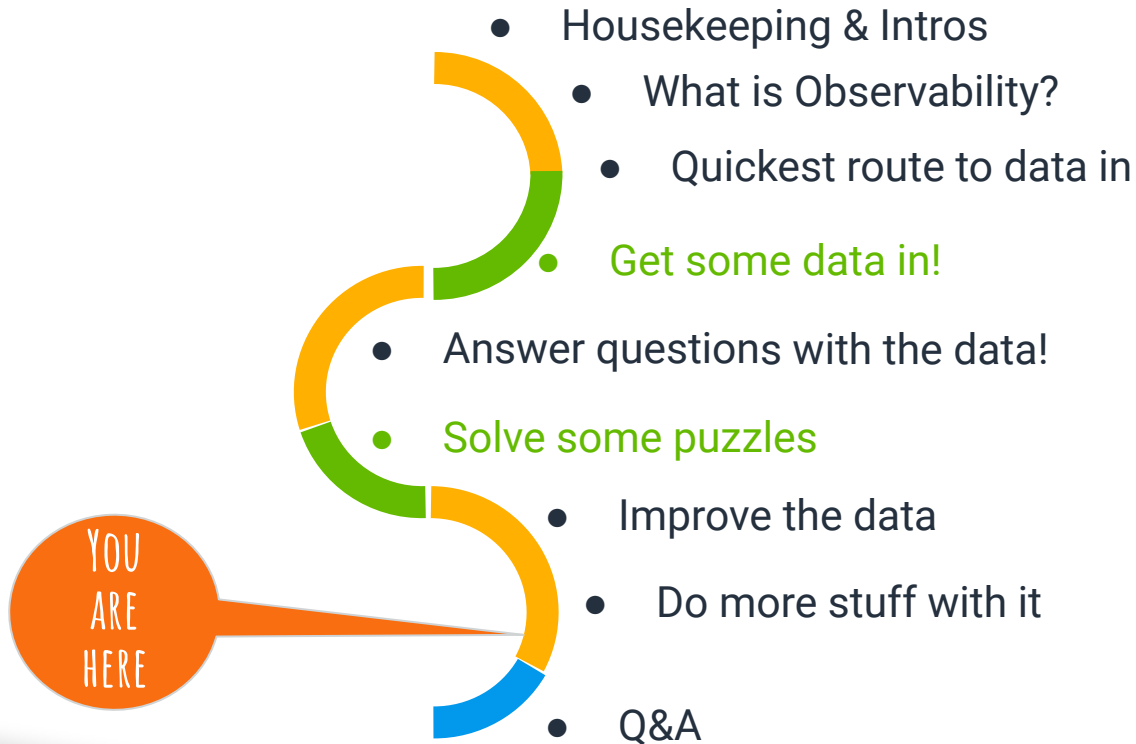
**Examine a Trace in Trace Detail view.**

Notice the double recursion and  
no caching present...



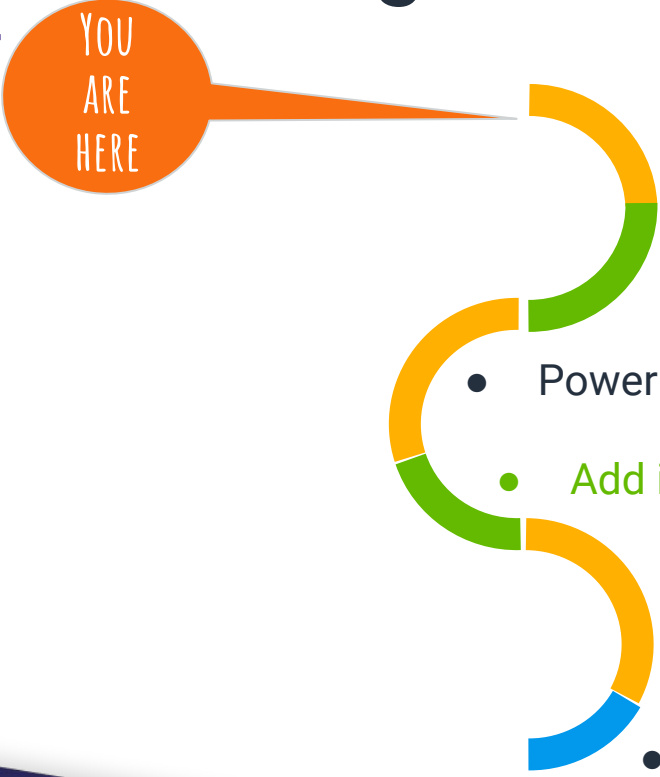
# Pre-break agenda

---



# Post-break agenda

YOU  
ARE  
HERE

- 
- Customize your demo application
  - Power querying of observability data
  - Add instrumentation to your own code
  - Let's share out our results
  - Q&A

# Add more fields and instrumentation

---

Edit or add at least one attribute name and one attribute value. (e.g. "\$firstName was here")

Look for, uncomment, and change the key and value parameters to `SetAttributes()`



## Now, let's instrument /fib.

```
import "go.opentelemetry.io/otel/api/key"

func main() {
    mux.Handle("/fib", othttp.NewHandler(
        http.HandlerFunc(fibHandler), "fib"))
    mux.Handle("/fibinternal", othttp.NewHandler(
        http.HandlerFunc(fibHandler), "fibInternal"))

    func fibHandler([...]) {
        ctx := req.Context()
        // Record the input value.
        trace.SpanFromContext(ctx).SetAttribute(key.Int("input", i))
        [...]
        trace.SpanFromContext(ctx).SetAttribute(key.Int("result", ret))
    }
}
```



## We'll also want client info.

```
import "go.opentelemetry.io/otel/plugin/httptrace"
func fibHandler(...) {
    [...]
    clientCall := func(ictx context.Context) {
        trace.SpanFromContext(ictx).SetAttribute(key.Int("req", i))
        req, _ := http.NewRequestWithContext(ictx, "GET", url, nil)
        ictx, req = httptrace.W3C(ictx, req)
        httptrace.Inject(ictx, req)
        res, err := client.Do(req)
    }
    err := tr.WithSpan(ctx, "fibClient", clientCall)
```

Exercise for the reader: record error statuses and results.





# Python - Instrument Request Specifically

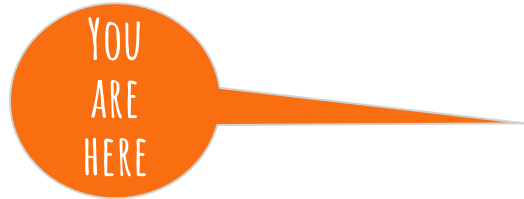
```
with tracer.start_as_current_span("getMinusOne") as span:  
    span.set_attribute("payloadValue", value - 1)  
    respOne = requests.get('http://127.0.0.1:5000/fibInternal',  
minusOnePayload)
```

```
with tracer.start_as_current_span("getMinusTwo") as span:  
    span.set_attribute("payloadValue", value - 2)  
    respTwo = requests.get('http://127.0.0.1:5000/fibInternal',  
minusTwoPayload)
```



# Post-break agenda

---



- Customize your demo application
- Power querying of observability data
- Add instrumentation to your own code
- Let's share out our results
- Q&A



# About the Collector...

---

# This could be an entire talk...

---

<http://3.236.120.251:55679/debug/servicez>

I have a demo collector running... I'll live configure it to write to logz.io, honeycomb and the jaeger instance.

You can repoint your apps to `grpc://3.236.120.251:4317/` and use my routing.

Now only YAML needs updating to change routing.



# Using OpenTelemetry with Logz.io

- You can export telemetry from OpenTelemetry Collector to the Logz.io backend using the Logz.io exporter for OTel Collector:  
<https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/exporter/logzioexporter>
- Here's a guide on how to configure OTel Collector to send to Logz.io and using the OTel community demo app reference:  
<https://logz.io/learn/how-to-run-opentelemetry-demo-with-logz-io/>

```
exporters:
  logzio/traces:
    account_token: "<<TRACING-SHIPING-TOKEN>>"
    region: "<<LOGZIO_ACCOUNT_REGION_CODE>>"

  prometheusremotewrite:
    endpoint: "https://<<LISTENER-HOST>>:8053"
    headers:
      Authorization: "Bearer <<METRICS-SHIPING-TOKEN>>"

  logzio/logs:
    account_token: "<<LOGS-SHIPING-TOKEN>>"
    region: "<<LOGZIO_ACCOUNT_REGION_CODE>>"

processors:
  batch:
    send_batch_size: 10000
    timeout: 1s

service:
  pipelines:
    traces:
      receivers: [ otlp ]
      processors: [ batch ]
      exporters: [ logzio/traces, logzio/logs, spanmetrics ]

    metrics:
      receivers: [ otlp, spanmetrics ]
      exporters: [ prometheusremotewrite ]

  logs:
    receivers: [ otlp ]
    processors: [ batch ]
    exporters: [ logzio/logs ]
```

# Using OpenTelemetry with Logz.io

- Logz.io provides an OpenTelemetry distro you can use:

<https://github.com/logzio/otel-collector-distro>

- The easiest way to install it, is with the wizard:

<https://docs.logz.io/docs/user-guide/telemetry-collector/>

- Offers good telemetry collection presets for various self-hosted and managed environments

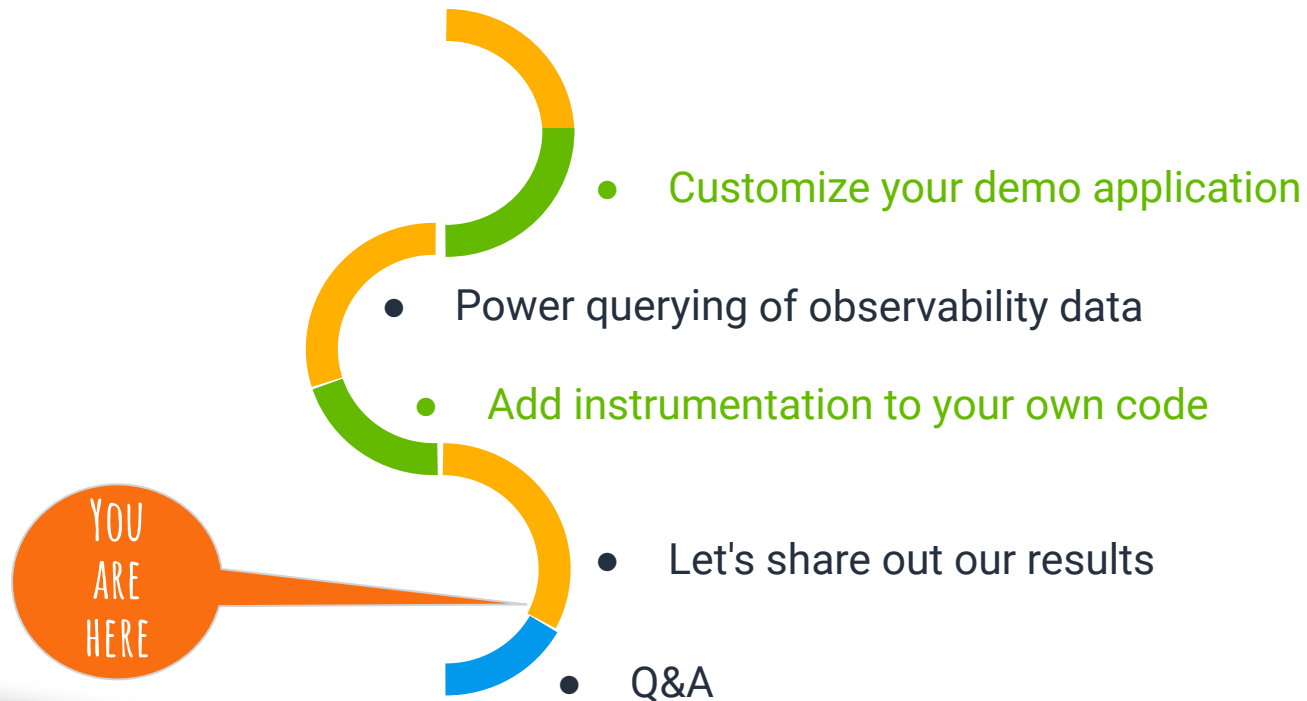
The screenshot displays the 'Telemetry collector' setup wizard in the Logz.io dashboard. The interface is divided into three main sections: a left sidebar, a central progress indicator, and a main content area.

- Left Sidebar:** Contains navigation icons for Search, Home, Logs, Metrics, Traces, SIEM, KBS 360, Data Hub, and Settings.
- Central Progress Indicator:** Shows three steps: 1. Setup your integration (active), 2. Configure your data sources, and 3. Finalize your telemetry collector.
- Main Content Area:** Titled 'Step 1/3', it features the heading 'Setup your integration in less than 5 minutes' and a 'Cancel' button. It is organized into three categories:
  - Localhost:** Includes 'Windows' (Windows Machine), 'Linux' (Linux Machine), and 'Mac' (Mac Machine).
  - Kubernetes:** Includes 'EKS' (AWS Kubernetes), 'AKS' (Azure Kubernetes), 'GKE' (GCP Kubernetes), and 'DigitalOcean' (Managed Kubernetes).
  - Amazon Web Services:** Includes 'AWS Logs via Cloudwatch', 'AWS Metrics via Cloudwatch', and 'EC2 Monitoring' (Linux Logs & Metrics).

The top right corner of the dashboard shows the user 'o11y.us' with 'Admin' and 'Main account' options, and a 'Help' icon.

# Post-break agenda

---



**Share with your colleagues**



# Show and tell

---

If you have a coworker here, you can share with them!

Otherwise, add a coworker to your team and show them!



# Bee a hero, make others heroes!

---

Instrumenting with OTel is easy\*

Auto-instrumentation is easier.

Sending small amounts data is free with most providers!

Querying reveals insights

We're here to help you!

Feel free to add OTel to your own apps!



# Instrument your own real applications

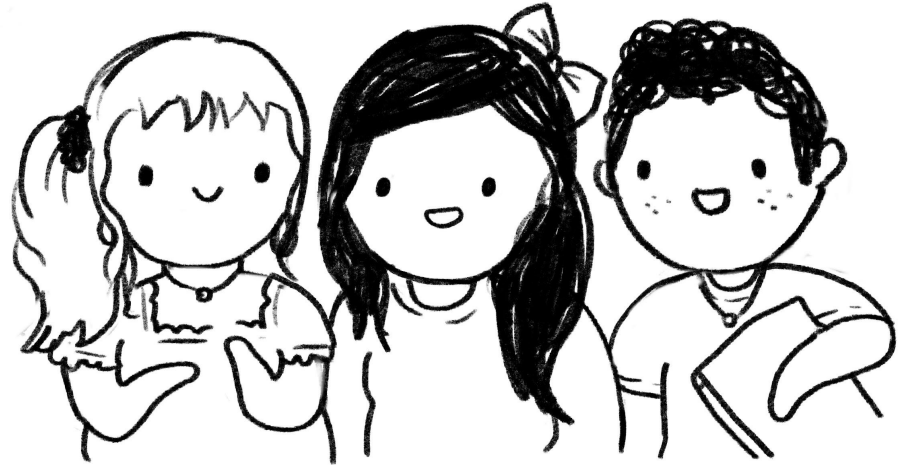
- OTel is GA right now.
  - Calling the API is safe to do.
  - SDKs for metrics change, but default to no-op.
  - OTel is in use by unicorns and publicly traded companies, and we do promise stability now!
- We've applied for CNCF graduation!
- Your instructors and TAs are here to help!



# Here's what you learned today

---

- Why observability matters
- How to add instrumentation and get telemetry data flowing to Jaeger
- How to use Jaeger & hosted backends to answer questions
- How to improve o11y & debugging workflows
- How to share these lessons with your team!





# Wrap-up

---

# A meetup for people who use OTel

---



## OpenTelemetry in Practice Meetup Group

 ~~San Francisco, CA~~ online

 263 members · Public group 

 Organized by **Rynn M.** and 2 others

Share:    

# Get the newsletter!

---



<https://opentelemetryinpractice.net>

# Ways to get involved

## CNCF Observability

### Technical Advisory Group (TAG)

- CNCF Slack  
**#tag-observability** channel

## OpenTelemetry

- CNCF Slack **#opentelemetry** channel
- OpenTelemetry.io
- [github.com/open-telemetry](https://github.com/open-telemetry)

## OpenSLO

- OpenSLO Slack
- OpenSLO.com

## Schedule 30 minutes

- [hny.co/meet/liz](https://hny.co/meet/liz)

## Find us on Twitter

- [twitter.com/lizthegrey](https://twitter.com/lizthegrey)
- [twitter.com/opentelemetry](https://twitter.com/opentelemetry)





# Q&A / free instrumentation time



# Thank you!

[hny.co/liz](https://hny.co/liz); [@lizthegrey](https://twitter.com/lizthegrey)