

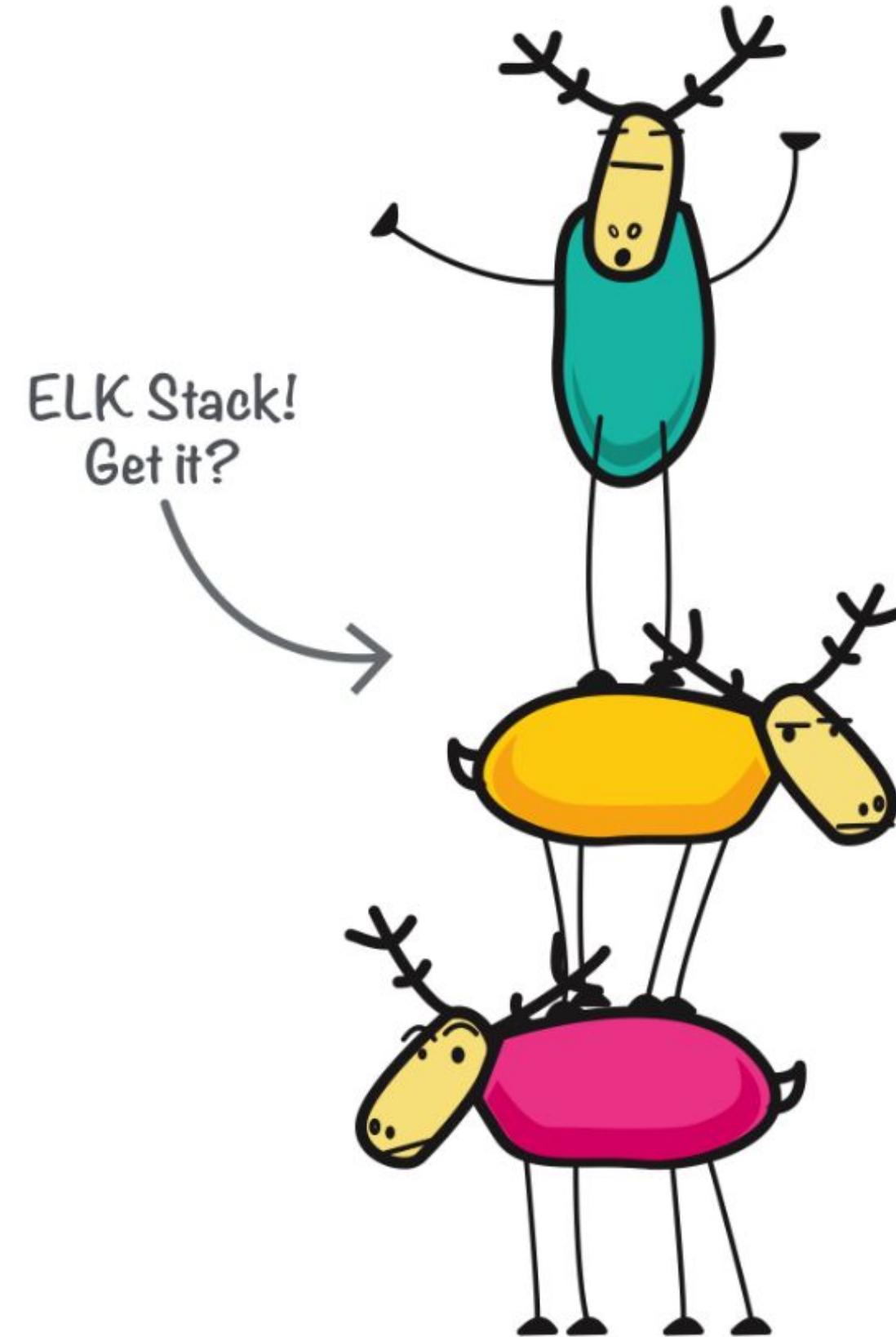
# Introduction to OpenTelemetry with Python

By Jessica Garson





**Jessica Garson**  
Senior Developer  
Advocate at Elastic  
@JessicaGarson



**E** Elasticsearch

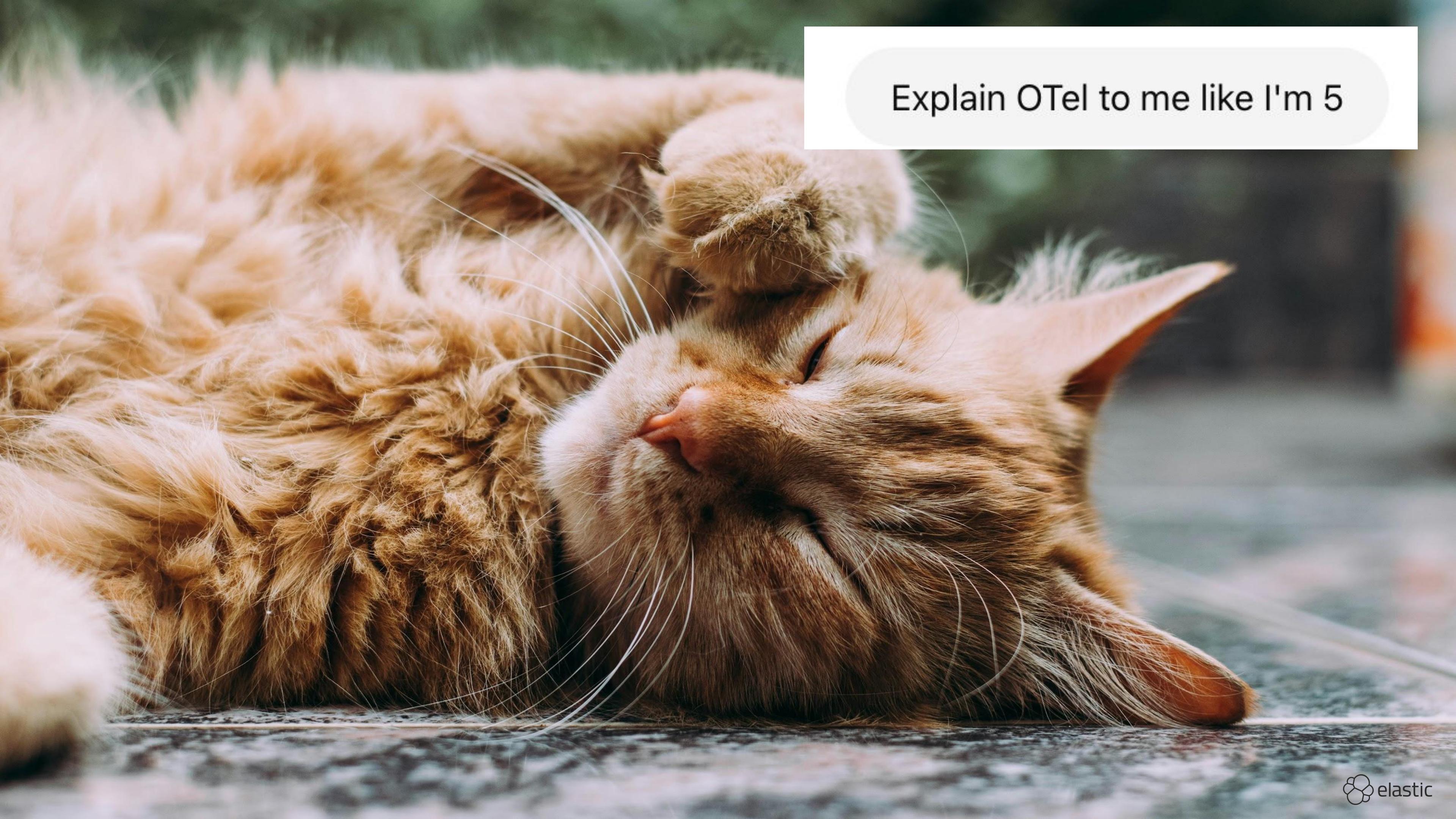
**L** Logstash

**K** Kibana

**github.com/JessicaGarson/Introduction-to-  
OpenTelemetry-with-Python**





A close-up, low-angle photograph of a ginger-colored cat sleeping soundly. The cat's eyes are closed, and its head is tilted back, resting on its front paws. Its long, white whiskers are clearly visible against its brown fur. The background is blurred, showing some greenery and a hint of another animal's fur.

Explain OTel to me like I'm 5



## ChatGPT

Sure! Imagine you have a big box of crayons (these are like different parts of a computer program) and you want to make a beautiful picture (which is like running the program to do something cool). But sometimes, while you're drawing, a crayon might break or not color the way it should. This can make your picture not turn out as nice as you wanted.

OpenTelemetry is like having a magical friend who watches you while you draw. This friend takes notes on which crayons you use, how they perform, and if they break. Then, if something goes wrong, your magical friend can tell you exactly which crayon broke and why, so you can fix it and make sure your picture turns out great next time!

So, OpenTelemetry helps people who make computer programs make sure their programs run smoothly and fix any problems that happen.

# Agenda

The hope is you will walk away with a strong understanding of what OpenTelemetry (OTel) is and how to start integrating it into your Python applications



## Observability basics

- Observability fundamentals
- Metrics, logs and traces
- Jargon breakdown
- What is OTEL?
- Why use it

## Django Instrumentation

- How to implement OTEL into your Django application

## Flask Instrumentation

- How to implement OTEL into your Flask application

## Conclusion and Next steps

- The state of the project
- Key takeaways

# Observability fundamentals



# What is observability?

Observability helps you determine what's happening inside your system by analyzing the data it outputs

**What isn't happening as expected?**

**Going beyond detecting problems**

# Logs, Metrics, and Traces

Sometimes known as the three pillars of observability

## Logs

Logs are records of events in a system, documenting operations, errors, and activities to aid in troubleshooting, monitoring, and compliance.

## Metrics

Metrics are quantitative measurements that track the performance and health of a system

## Traces

Traces track the path and interactions of a request through a system.

# **Breaking down some common terms**

## **What is an agent?**

An observability agent is a software component or tool that collects, processes, and sends telemetry data from a system or application to an observability platform for analysis.

## **Instrumenting?**

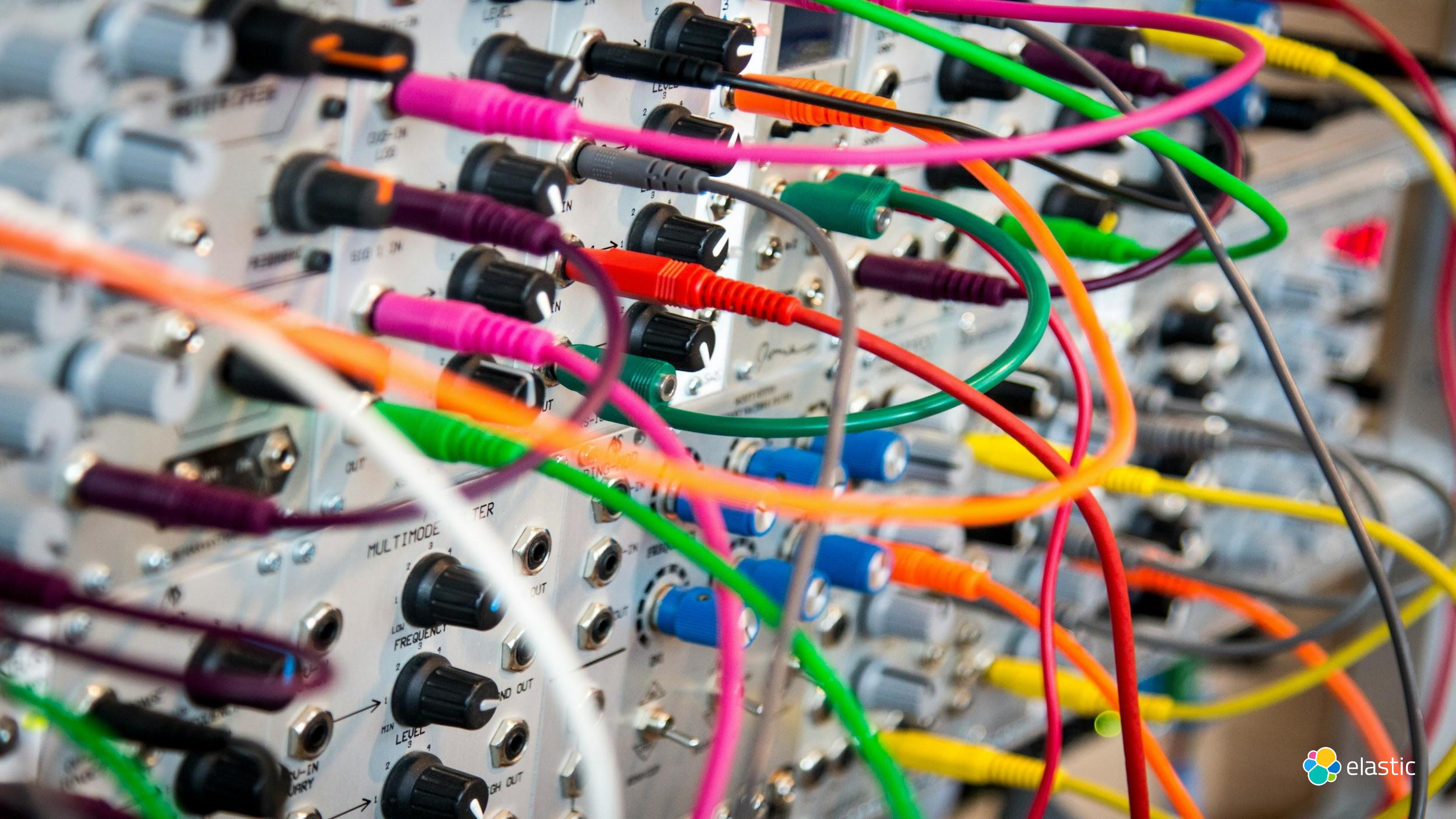
Instrumenting refers to the process of adding observability features to your application to collect telemetry data, such as traces, metrics, and logs.

# Logs alone are simply not enough sometimes



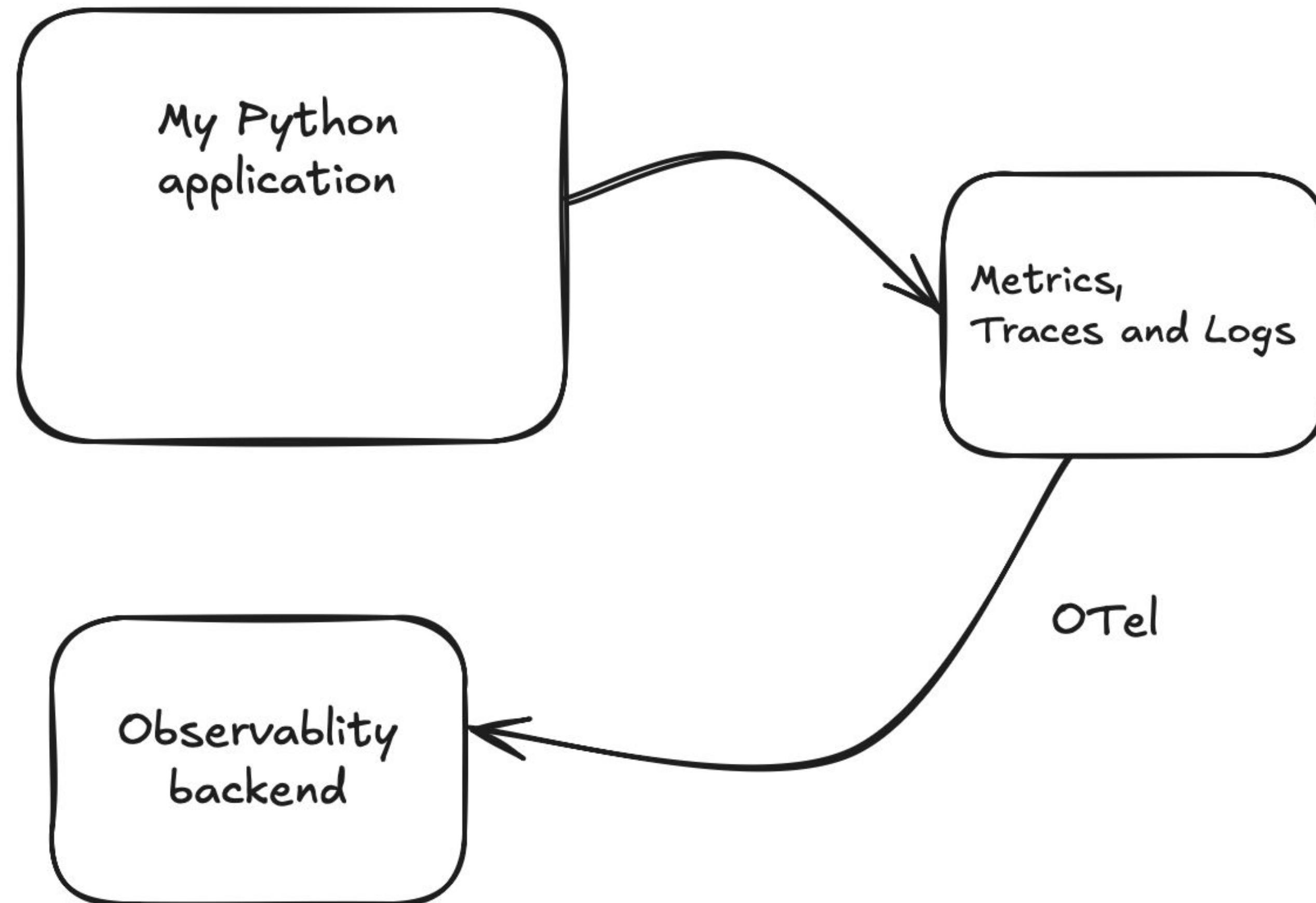
# What is OpenTelemetry (OTel)?

- OTel is an observability framework that is open source and vendor neutral, designed to work with any backend system.
- It provides standardized APIs, libraries, and tools to collect telemetry data such as metrics, logs, and traces.



# Future proof





# Spans within traces?

## What?!?!

Spans are the building blocks of traces. They include the following information:

- Name
- Parent span ID (empty for root spans)

# Ohhhh!

## A span within a trace!

```
{  
    "name": "add_item_view_span",  
    "context": {  
        "trace_id": "0x81364006fe668cb26a54e20918be9620",  
        "span_id": "0x9a048a860bb7c9ac",  
        "trace_state": "[]"  
    },  
    "kind": "SpanKind.INTERNAL",  
    "parent_id": null,  
    "start_time": "2024-08-15T20:55:42.634538Z",  
    "end_time": "2024-08-15T20:55:42.637830Z",  
    "status": {  
        "status_code": "UNSET"  
    },  
    "attributes": {},  
    "events": [],  
    "links": [],  
    "resource": {  
        "attributes": {  
            "service.name": "to-do-list-app-manual",  
            "service.version": "1.0.0"  
        },  
        "schema_url": ""  
    }  
}
```

# Elastic Distribution of OpenTelemetry

The Elastic Distribution of OpenTelemetry (EDOT) for Python is a tailored implementation of the OpenTelemetry framework, specifically optimized to enhance the user experience within the Elastic Observability ecosystem.

# Elastic Distribution of OpenTelemetry Python

Stack

Serverless Observability

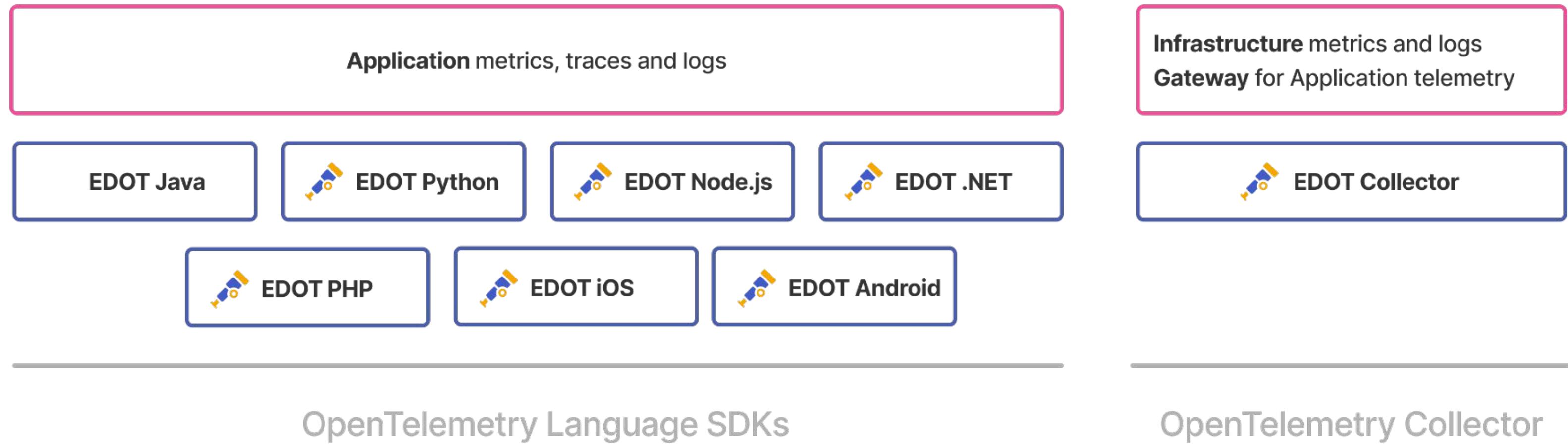
EDOT Python

---

The [Elastic Distribution of OpenTelemetry \(EDOT\) Python](#) is a customized version of [OpenTelemetry Python](#), configured for the best experience with Elastic Observability.

Use EDOT Python to start the OpenTelemetry SDK with your Python application, and automatically capture tracing data, performance metrics, and logs. Traces, metrics, and logs can be sent to any OpenTelemetry Protocol (OTLP) Collector you choose.

A goal of this distribution is to avoid introducing proprietary concepts in addition to those defined by the wider OpenTelemetry community. For any additional features introduced, Elastic aims at contributing them back to the upstream OpenTelemetry project.



# Django Example

# The classic Django example

**To Do List**

- New task!** - Cool! A new task! Delete
- Another task!** - Yay! Delete

**Add New Item**

# Install the required packages



```
pip install django django-environ elastic-opentelemetry opentelemetry-instrumentation-django  
opentelemetry-bootstrap --action=install
```

# Automatic Instrumentation

The process by which an agent modifies the bytecode of your application's classes, often to insert monitoring code

# .env example



```
OTEL_EXPORTER_OTLP_HEADERS="Authorization=ApiKey%20yourapikey"  
OTEL_EXPORTER_OTLP_ENDPOINT="https://your/host/endpoint"
```

# settings.py



```
env = environ.Env()

environ.Env.read_env(os.path.join(BASE_DIR, '.env'))

OTEL_EXPORTER_OTLP_HEADERS = env('OTEL_EXPORTER_OTLP_HEADERS')
OTEL_EXPORTER_OTLP_ENDPOINT = env('OTEL_EXPORTER_OTLP_ENDPOINT')
```



```
def main():
    """Run administrative tasks."""
    os.environ.setdefault("DJANGO_SETTINGS_MODULE", "todolist_project.settings")
    DjangoInstrumentor().instrument()

    # Set up resource attributes for the service
    resource = resources.Resource(attributes={
        resources.SERVICE_NAME: "to-do-list-app",
        resources.SERVICE_VERSION: "1.0.0"
    })

    trace_provider = TracerProvider(resource=resource)
    trace.set_tracer_provider(trace_provider)

    otlp_exporter = OTLPSpanExporter()

    # Set up the BatchSpanProcessor to export traces
    span_processor = BatchSpanProcessor(otlp_exporter)
    trace.get_tracer_provider().add_span_processor(span_processor)

try:
    from django.core.management import execute_from_command_line
except ImportError as exc:
    raise ImportError(
        "Couldn't import Django. Are you sure it's installed and "
        "available on your PYTHONPATH environment variable? Did you "
        "forget to activate a virtual environment?"
    ) from exc
execute_from_command_line(sys.argv)
```

# manage.py



# Running the code



# Manual instrumentation

Manual instrumentation requires incorporating particular code segments into your application to collect and transmit telemetry data.



**So when would you want to use manual instrumentation?**

# Step 1: Delete this line from your manage.py file



```
# DjangoInstrumentor().instrument()
```

## Step 2: Update your views.py

```
tracer = trace.get_tracer(__name__)

meter = get_meter(__name__)

view_counter = meter.create_counter(
    "view_requests",
    description="Counts the number of requests to views",
)

view_duration_histogram = meter.create_histogram(
    "view_duration",
    description="Measures the duration of view execution",
)
```



```
def index(request):
    start_time = time()

    with tracer.start_as_current_span("index_view_span") as span:
        items = ToDoItem.objects.all()
        span.set_attribute("todo.item_count", items.count())
        response = render(request, 'todo/index.html', {'items': items})

    view_counter.add(1, {"view_name": "index"})
    view_duration_histogram.record(time() - start_time, {"view_name": "index"})

    return response
```





```
def add_item(request):
    start_time = time()

    with tracer.start_as_current_span("add_item_view_span") as span:
        if request.method == 'POST':
            form = ToDoForm(request.POST)
            if form.is_valid():
                form.save()
                span.add_event("New item added")
                response = redirect('index')
            else:
                response = render(request, 'todo/add_item.html', {'form': form})
        else:
            form = ToDoForm()
            response = render(request, 'todo/add_item.html', {'form': form})

    view_counter.add(1, {"view_name": "add_item"})
    view_duration_histogram.record(time() - start_time, {"view_name": "add_item"})

    return response
```





```
def delete_item(request, item_id):
    start_time = time()

    with tracer.start_as_current_span("delete_item_view_span") as span:
        item = get_object_or_404(TodoItem, id=item_id)
        span.set_attribute("todo.item_id", item_id)
        if request.method == 'POST':
            item.delete()
            span.add_event("Item deleted")
            response = redirect('index')
        else:
            response = render(request, 'todo/delete_item.html', {'item': item})

    view_counter.add(1, {"view_name": "delete_item"})
    view_duration_histogram.record(time() - start_time, {"view_name": "delete_item"})

return response
```



## Step 3: Update your models.py

```
from django.db import models
from opentelemetry import trace

tracer = trace.get_tracer(__name__)

class ToDoItem(models.Model):
    title = models.CharField(max_length=100)
    description = models.TextField(blank=True)
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.title

    def save(self, *args, **kwargs):
        with tracer.start_as_current_span("save_todo_item_span") as span:
            span.set_attribute("todo.title", self.title)
            if self.pk:
                span.add_event("Updating ToDoItem")
            else:
                span.add_event("Creating new ToDoItem")

        super(ToDoItem, self).save(*args, **kwargs)
```

# Running the code



```
python manage.py runserver 8080
```

# Flask example



# No code instrumentation for automatic instrumentation



pip install elastic-opentelemetry



```
opentelemetry-bootstrap --action=install
```



```
export OTEL_RESOURCE_ATTRIBUTES="service.name=flask-todo-app"
export OTEL_EXPORTER_OTLP_HEADERS="Authorization=<your_authorization_header_value>"
export OTEL_EXPORTER_OTLP_ENDPOINT="<your_elastic_cloud_url>"
```



opentelemetry-instrument flask run

# To-Do List

Add new task

ADD TASK

New task!

Delete

# Manual instrumentation in Flask

<https://github.com/JessicaGarson/Introduction-to-OpenTelemetry-with-Python/blob/main/Flask/manual-instrumentation/app.py>



```
flask run -p 5050
```

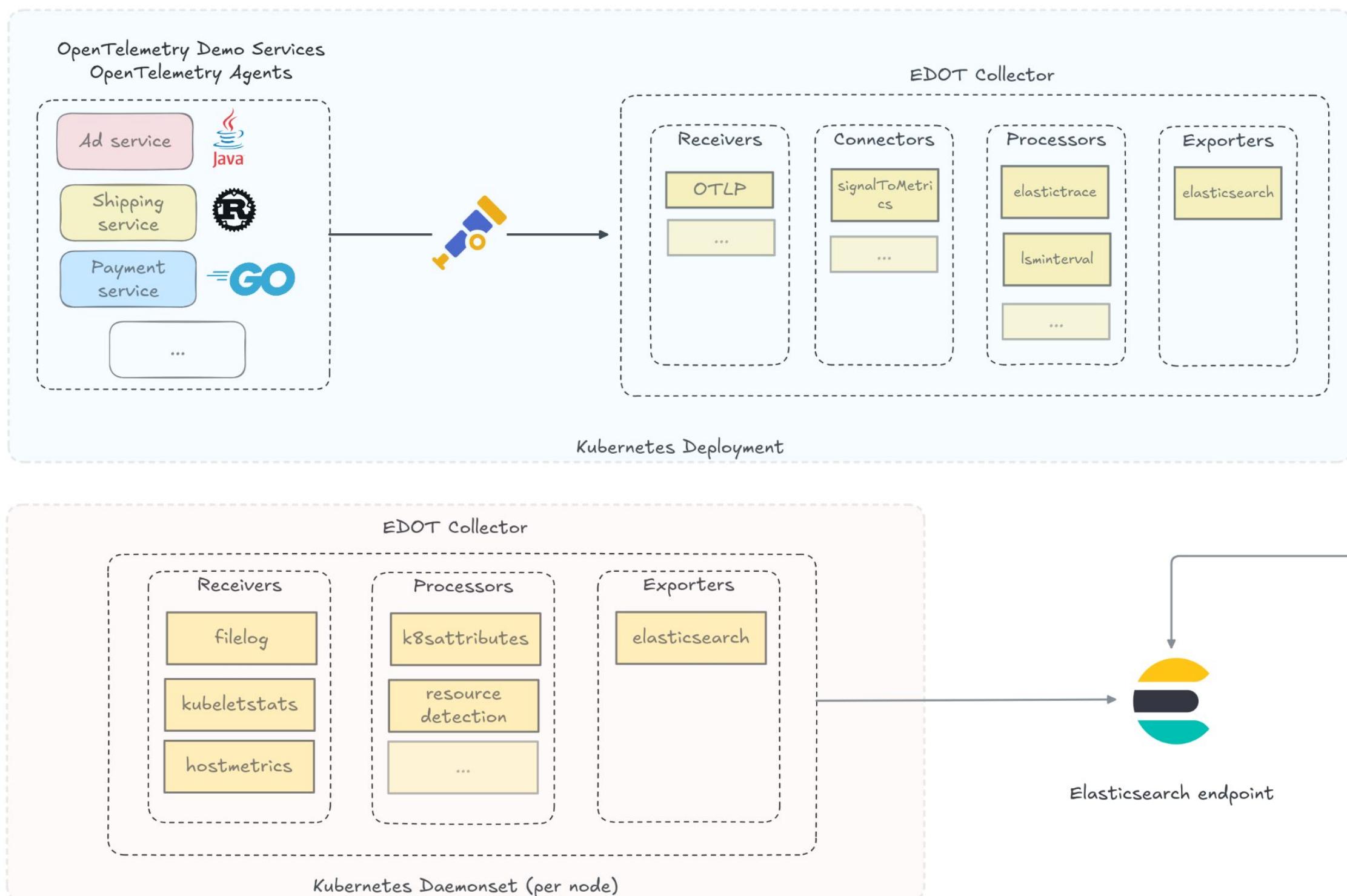
# OpenTelemetry Collector

A solution for receiving, processing, and exporting telemetry data. It eliminates the need to manage and maintain multiple agents or collectors.

# You can also play with our demo application

<https://github.com/elastic/opentelemetry-demo>

<https://otel.demo.elastic.co>



# Closing thoughts

**Elastic ❤️ OTEL**

# <https://opentelemetry.io/status/>

## Language APIs & SDKs

For the development status, or maturity level, of a [language API or SDK](#), see the following table:

Language	Traces	Metrics	Logs
C++	<a href="#">Stable</a> ↗	<a href="#">Stable</a> ↗	<a href="#">Stable</a> ↗
C#/.NET	<a href="#">Stable</a> ↗	<a href="#">Stable</a> ↗	<a href="#">Stable</a> ↗
Erlang/Elixir	<a href="#">Stable</a> ↗	<a href="#">Development</a> ↗	<a href="#">Development</a> ↗
Go	<a href="#">Stable</a> ↗	<a href="#">Stable</a> ↗	<a href="#">Beta</a> ↗
Java	<a href="#">Stable</a> ↗	<a href="#">Stable</a> ↗	<a href="#">Stable</a> ↗
JavaScript	<a href="#">Stable</a> ↗	<a href="#">Stable</a> ↗	<a href="#">Development</a> ↗
PHP	<a href="#">Stable</a> ↗	<a href="#">Stable</a> ↗	<a href="#">Stable</a> ↗
Python	<a href="#">Stable</a> ↗	<a href="#">Stable</a> ↗	<a href="#">Development</a> ↗
Ruby	<a href="#">Stable</a> ↗	<a href="#">Development</a> ↗	<a href="#">Development</a> ↗
Rust	<a href="#">Beta</a> ↗	<a href="#">Beta</a> ↗	<a href="#">Beta</a> ↗
Swift	<a href="#">Stable</a> ↗	<a href="#">Development</a> ↗	<a href="#">Development</a> ↗

# Why use OTEL?

OpenTelemetry (OTel) is an open-source, vendor-neutral observability framework designed to integrate with any backend system.

01

It's very easy to start instrumenting your code.

02

It is highly configurable and extensible

03

It scales well from large applications to small applications

04

Instead of managing separate libraries for metrics, traces, and logs, OTEL gives you a single framework



**Let me know if this talk inspires you to build anything. I'm @JessicaGarson on most platforms.**

**github.com/JessicaGarson/Introduction-to-  
OpenTelemetry-with-Python**



# Thank you!