

Introduction to OpenTelemetry with Django

Jessica Garson

@JessicaGarson

@JessicaGarson@macaw.social

Senior Developer Advocate



I had attempted to find a short primer on starting with OpenTelemetry with Python but couldn't find much more than a [demo application](#).



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.



Observability fundamentals

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



What isn't operating as expected

Beyond detecting problems

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.

What is OpenTelemetry (OTel)?

OpenTelemetry (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.

OpenTelemetry enables comprehensive monitoring
across services and platforms.

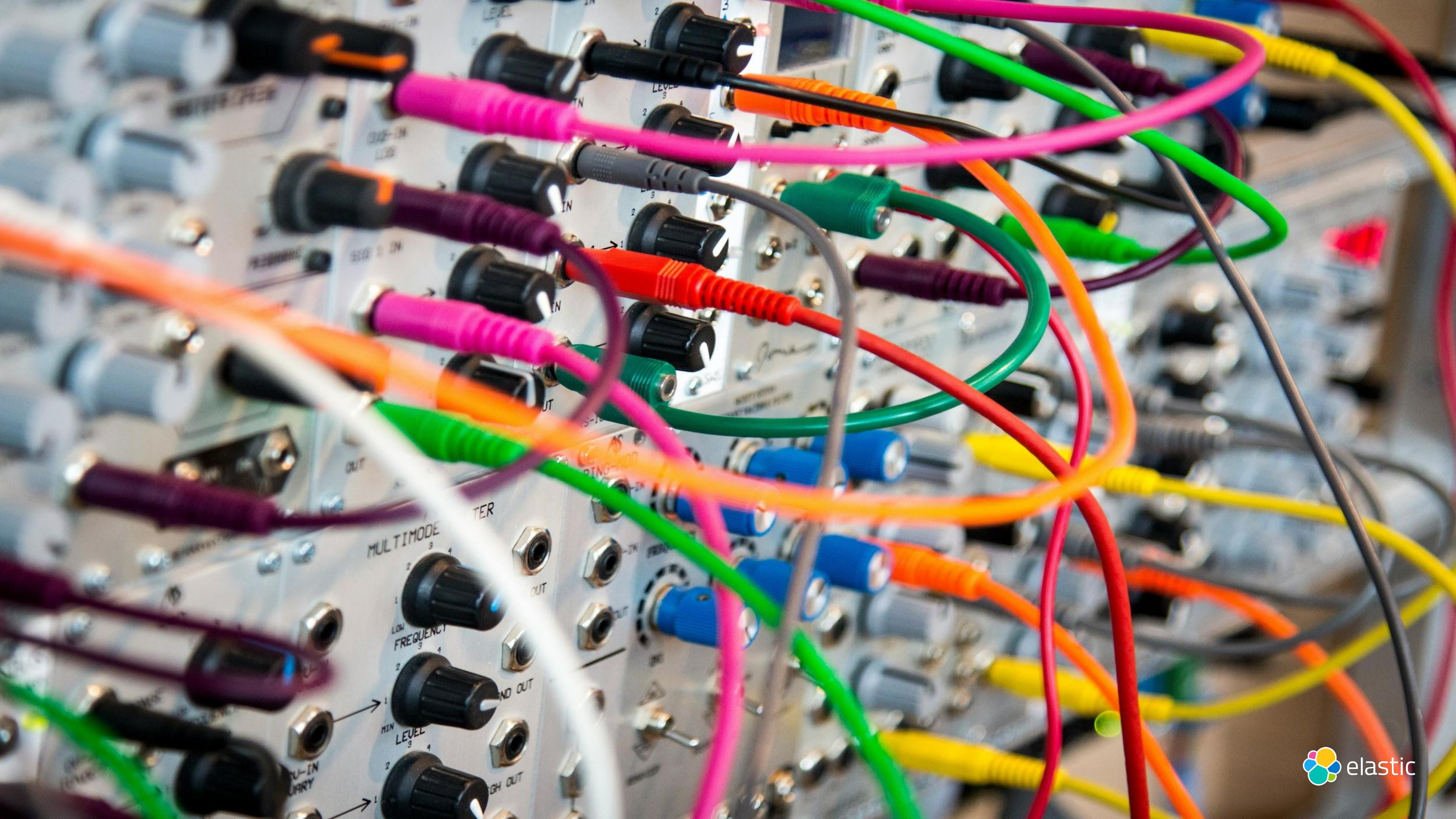
Why?

A photograph of a stack of logs in a forest. The logs are cut trees, showing their circular cross-sections. They are piled up in a somewhat haphazard manner, with some logs leaning against each other. The background is filled with dense, dark forest trees. The lighting suggests it might be late afternoon or early morning, with dappled sunlight filtering through the canopy.

Logs alone are simply not
enough sometimes

Um, what?

>	*	2024-05-10	08:00:06.117	55	54440319	...	http://api.nasa.gov/neo/rest/v1/neo/sentry/544...
>	*	2024-05-10	08:00:06.117	56	54440642	...	NaN
▼	*	2024-05-10	08:00:06.117	57	54440644	...	http://api.nasa.gov/neo/rest/v1/neo/sentry/544...
<div>Copy Similar entries Expand nested fields Hide log summary</div>							
▼	{						
	insertId: "663e0c460001c9a60e5e65c3"						
▶	labels: {2}						
	logName: "projects/ncav-293119/logs/run.googleapis.com%2Fstdout"						
	receiveTimestamp: "2024-05-10T12:00:06.338017672Z"						
▶	resource: {2}						
	textPayload: "57 54440644 ... http://api.nasa.gov/neo/rest/v1/neo/sentry/544..."						
	timestamp: "2024-05-10T12:00:06.117158Z"						
	}						
>	*	2024-05-10	08:00:06.117	[43 rows x 18 columns]			



Future proof



The classic Django example

To Do List

New task! - Cool! A new task!

Another task! - Yay!

Add New Item

Benefits

- Enhanced visibility into the execution flow and call timing
- Metrics on how users utilize services to pinpoint unnecessary ones
- The ability to detect patterns, such as peak usage times, which aids in production planning and scaling.

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.

Spans within traces? What?!?!



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

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

```
{  
  "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": ""  
  }  
}
```

OHHH!

A span
within a
trace!

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



```
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"
```

manage.py



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

    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()

    span_processor = BatchSpanProcessor(otlp_exporter)
    trace.get_tracer_provider().add_span_processor(span_processor)
```



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')
```

Demo

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

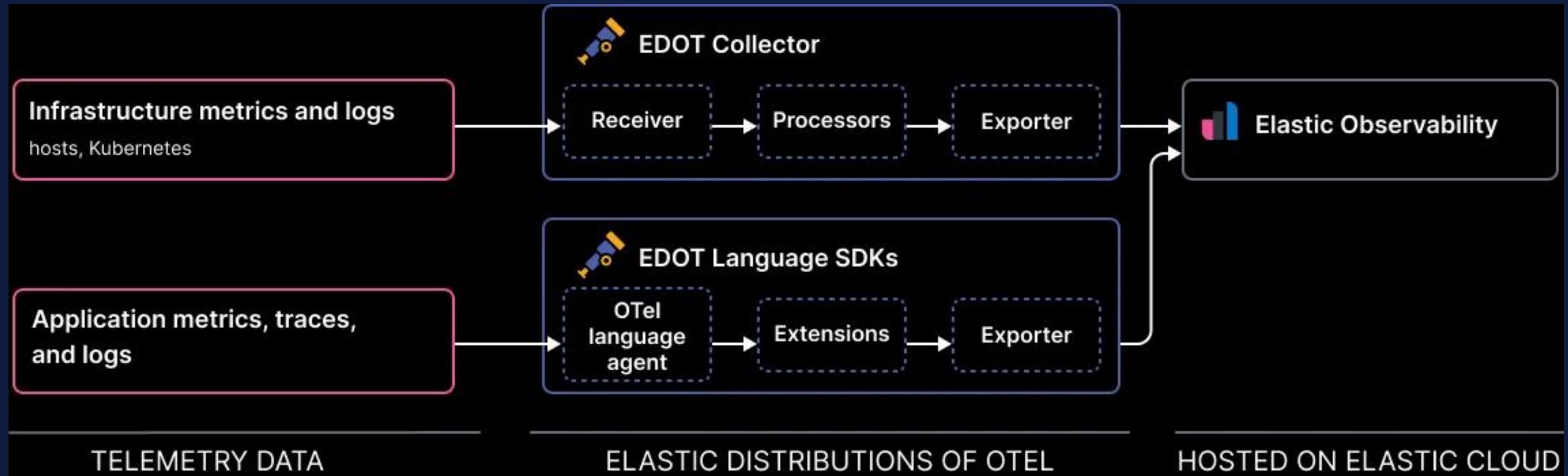
    def save(self, *args, **kwargs):
        with tracer.start_as_current_span("save_todo_item_span") as span:
            span.set_attribute("todo.title", self.title)
            if self.pk:
                span.add_event("Updating ToDoItem")
            else:
                span.add_event("Creating new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

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

OpenTelemetry Collector

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

How does Elastic fit into this?



The Elastic Stack natively supports the OpenTelemetry protocol (OTLP). This means trace data and metrics collected from your applications and infrastructure can be sent directly to the Elastic Stack.

State of adoption

WORK in PROGRESS

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

Closing thoughts

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

It's very easy to start instrumenting your code.

It is highly configurable and extensible

It scales well from large applications to
small applications

Next steps



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

Thank you!