Airflow OpenTelemetry POC

Journal on OpenTelemetry Proof of Concept for Airflow.

Period: Feb 22 ~ Mar 4, 2022

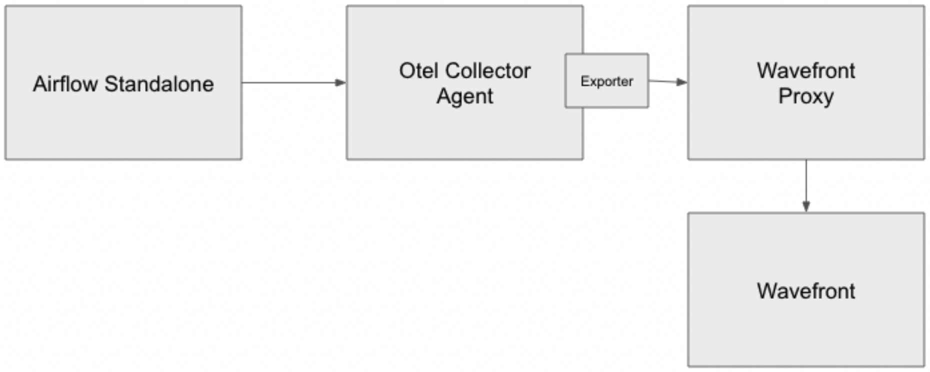
By Howard Yoo

Table of Contents

Overview

1. Setup Airflow on your local machine
    a. Put instrumentation code to produce:
        i. Metrics
        ii. Traces
        iii. (optionally) Logs
2. Setup the Otel Collecting Agent
3. Setup the Backend (Wavefront)

Then, monitor the airflow using Wavefront.



Running the Otel Collector

Using docker to run it would be the easiest:

💡 `docker run otel/opentelemetry-collector`

https://github.com/open-telemetry/opentelemetry-collector

- Traces exporter for wavefront (tanzu observability) is available here: https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/exporter/tanzuobservabilityexporter

The following yaml config file (otel-local-config.yaml) was created (referenced from the example config here: https://github.com/open-telemetry/opentelemetry-collector/blob/main/examples/local/otel-config.yaml)

```
extensions:
  memory_ballast:
    size_mib: 512
  zpages:
    endpoint: 0.0.0.0:55679

receivers:
  otlp:
    protocols:
      grpc:
      http:

processors:
  batch:
  memory_limiter:
    # 75% of maximum memory up to 4G
    limit_mib: 1536
    # 25% of limit up to 2G
    spike_limit_mib: 512
    check_interval: 5s

exporters:
  logging:
    logLevel: debug
  tanzuobservability:
    traces:
      endpoint: "http://localhost:30001"
    metrics:
      endpoint: "http://localhost:2878"

service:
  pipelines:
    traces:
      receivers: [otlp]
      processors: [memory_limiter, batch]
      exporters: [logging, tanzuobservability]
    metrics:
      receivers: [otlp]
      processors: [memory_limiter, batch]
      exporters: [logging, tanzuobservability]

  extensions: [memory_ballast, zpages]
```

The following run script was then devised to easily run the open telemetry docker image:

```
#!/bin/bash

docker run --rm -p 13133:13133 -p 14250:14250 -p 14268:14268 \
      -p 55678-55679:55678-55679 -p 4317:4317 -p 8888:8888 -p 9411:9411 \
      -v "${PWD}/otel-local-config.yaml":/otel-local-config.yaml \
      --name otelcol otel/opentelemetry-collector \
      --config otel-local-config.yaml; \
```

** The docker image did not run, due to the fact that the official image did NOT contain the tanzuobservability exporter (since it's classified as contrib package). Therefore, downloading the binary and running it from here : https://github.com/open-telemetry/opentelemetry-collector-contrib/releases was needed. The latest version id NOT have any binaries (but only source code), so had to build one on my macbook. Well, golang is kind of easy(?) to build, so just ran

Building the Otel Collector

1. Install the golang on your macOS (otel collector is written in golang)
2. `go install github.com/google/addlicense@latest`
   a. Note that go binaries (built) do get built under ~/go/bin directory. Best to make a path into that for it to run (or copy the command to the usual user path)
3. check out the source of otel collector using the command `git clone https://github.com/open-telemetry/opentelemetry-go-build-tools`
4. go into this directory, and run `make` command to build the commands.
5. copy the following:
   a. cp ./checkdoc/checkdoc /usr/local/bin
6. Get the impi package by `go get -u github.com/pavius/impi/cmd/impi`
   a. cp ~/go/bin/impi /usr/local/bin
7. Get the misspel package by `go get -u github.com/client9/misspell/cmd/misspell`
   a. cp ~/go/bin/misspell /usr/local/bin
8. Get the golangci-lint `go install github.com/golangci/golangci-lint/cmd/golangci-lint@v1.44.2`
   a. cp ~/go/bin/golangci-lint /usr/local/bin
9. run `make otelcontribcol` inside the opentelemetry source directory...

This would have opentelemetry collector to have ALL the exporters available in its binary, which ... will be kind of too much. So, I would say perhaps have only tanzu observability (wavefront) exporters instead of buildling everything - might be more sensible thing to do.

Anyway, the binary `otelcontribcol_darwin_arm64` will be located inside ./bin directory after the build is finished. Change the name (it's too long) to just `otelcol` and put it into the ~/otel/collector directory where the `otel-local-config.yaml` resides.

Installing Wavefront Proxy

Install using brew:

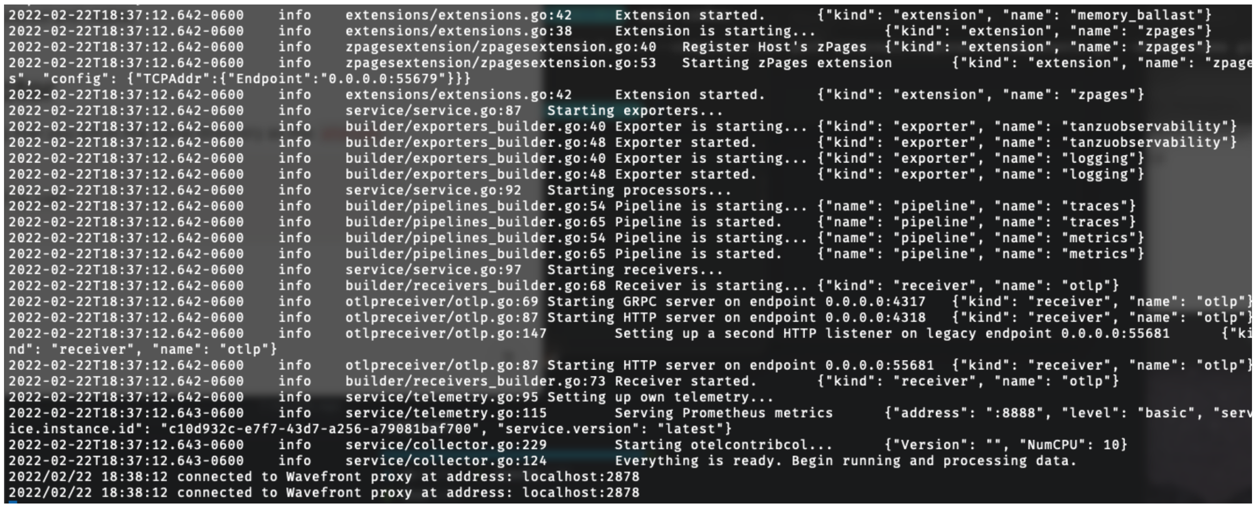
```
brew tap wavefrontHQ/wavefront
brew install wfproxy
```

- Installed and used Open JDK 13 instead of the provided jdk (inside wf proxy) - due to some security restrictions. Running fine.
- Followed the following instructions to set things up. https://docs.wavefront.com/opentelemetry.html
- Setup the proxy on wavefront.conf
    - setting the target Wavefront instance to try.wavefront.com
    - Assign correct tokens - proxy is registered and visible.

Running the OTEL collector

Run using the following command (assuming that you are in the same directory as the `otelcol` :

```
./otelcol --config ./otel-local-config.yaml
```



```
2022-02-22T18:37:12.642-0600    info    extensions/extensions.go:42     Extension started.      {"kind": "extension", "name": "memory_ballast"}
2022-02-22T18:37:12.642-0600    info    extensions/extensions.go:38     Extension is starting...        {"kind": "extension", "name": "zpages"}
2022-02-22T18:37:12.642-0600    info    zpagesextension/zpagesextension.go:40   Register Host's zPages  {"kind": "extension", "name": "zpages"}
2022-02-22T18:37:12.642-0600    info    zpagesextension/zpagesextension.go:53   Starting zPages extension       {"kind": "extension", "name": "zpage
s", "config": {"TCPAddr":{"Endpoint":"0.0.0.0:55679"}}}
2022-02-22T18:37:12.642-0600    info    extensions/extensions.go:42     Extension started.      {"kind": "extension", "name": "zpages"}
2022-02-22T18:37:12.642-0600    info    service/service.go:87   Starting exporters...
2022-02-22T18:37:12.642-0600    info    builder/exporters_builder.go:40 Exporter is starting... {"kind": "exporter", "name": "tanzuobservability"}
2022-02-22T18:37:12.642-0600    info    builder/exporters_builder.go:48 Exporter started.       {"kind": "exporter", "name": "tanzuobservability"}
2022-02-22T18:37:12.642-0600    info    builder/exporters_builder.go:40 Exporter is starting... {"kind": "exporter", "name": "logging"}
2022-02-22T18:37:12.642-0600    info    builder/exporters_builder.go:48 Exporter started.       {"kind": "exporter", "name": "logging"}
2022-02-22T18:37:12.642-0600    info    service/service.go:92   Starting processors...
2022-02-22T18:37:12.642-0600    info    builder/pipelines_builder.go:54 Pipeline is starting... {"name": "pipeline", "name": "traces"}
2022-02-22T18:37:12.642-0600    info    builder/pipelines_builder.go:65 Pipeline is started.    {"name": "pipeline", "name": "traces"}
2022-02-22T18:37:12.642-0600    info    builder/pipelines_builder.go:54 Pipeline is starting... {"name": "pipeline", "name": "metrics"}
2022-02-22T18:37:12.642-0600    info    builder/pipelines_builder.go:65 Pipeline is started.    {"name": "pipeline", "name": "metrics"}
2022-02-22T18:37:12.642-0600    info    service/service.go:97   Starting receivers...
2022-02-22T18:37:12.642-0600    info    builder/receivers_builder.go:68 Receiver is starting... {"kind": "receiver", "name": "otlp"}
2022-02-22T18:37:12.642-0600    info    otlpreceiver/otlp.go:69 Starting GRPC server on endpoint 0.0.0.0:4317   {"kind": "receiver", "name": "otlp"}
2022-02-22T18:37:12.642-0600    info    otlpreceiver/otlp.go:87 Starting HTTP server on endpoint 0.0.0.0:4318   {"kind": "receiver", "name": "otlp"}
2022-02-22T18:37:12.642-0600    info    otlpreceiver/otlp.go:147        Setting up a second HTTP listener on legacy endpoint 0.0.0.0:55681      {"ki
nd": "receiver", "name": "otlp"}
2022-02-22T18:37:12.642-0600    info    otlpreceiver/otlp.go:87 Starting HTTP server on endpoint 0.0.0.0:55681  {"kind": "receiver", "name": "otlp"}
2022-02-22T18:37:12.642-0600    info    builder/receivers_builder.go:73 Receiver started.       {"kind": "receiver", "name": "otlp"}
2022-02-22T18:37:12.642-0600    info    service/telemetry.go:95 Setting up own telemetry...
2022-02-22T18:37:12.643-0600    info    service/telemetry.go:115        Serving Prometheus metrics      {"address": ":8888", "level": "basic", "serv
ice.instance.id": "c10d932c-e7f7-43d7-a256-a79081baf700", "service.version": "latest"}
2022-02-22T18:37:12.643-0600    info    service/collector.go:229        Starting otelcontribcol...      {"Version": "", "NumCPU": 10}
2022-02-22T18:37:12.643-0600    info    service/collector.go:124        Everything is ready. Begin running and processing data.
2022/02/22 18:38:12 connected to Wavefront proxy at address: localhost:2878
2022/02/22 18:38:12 connected to Wavefront proxy at address: localhost:2878
```

Right now, there won't be too many data entering into the Wavefront Proxy, since we do not have any applications sending metrics / traces to it. However, as we will start instrumenting Airflow on OpenTelemetry, we'll be getting metrics and traces into Wavefront!

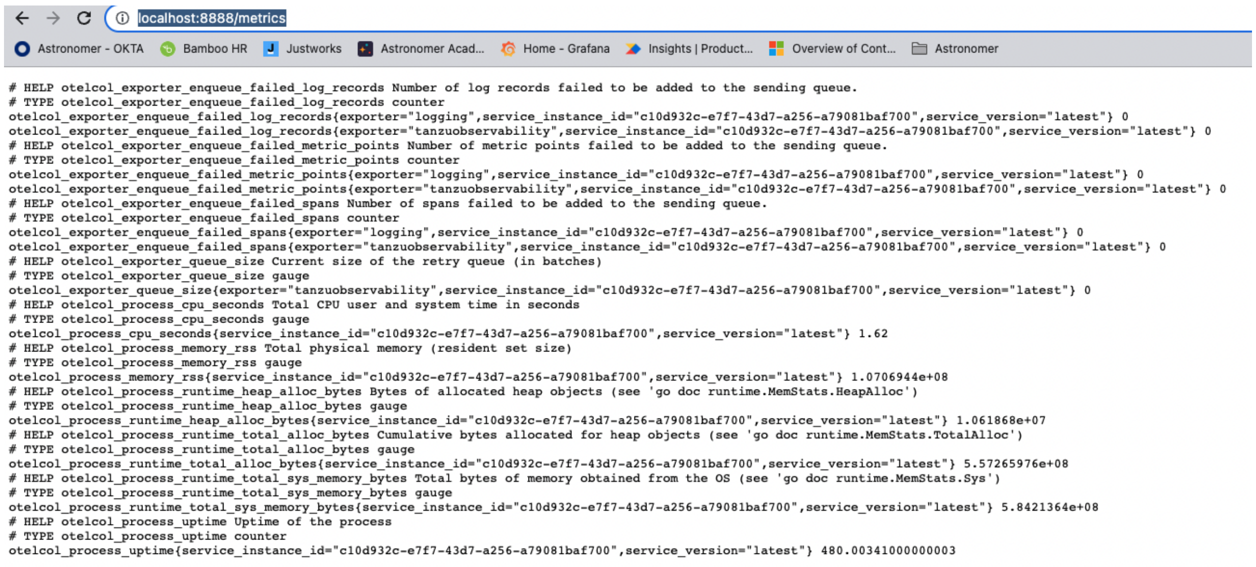
Ports

```
2022-02-22T18:37:12.642-0600  info  builder/receivers_builder.go:68 Receiver is starting... {"kind": "receiver", "name": "otlp"}
2022-02-22T18:37:12.642-0600  info  otlpreceiver/otlp.go:69 Starting GRPC server on endpoint 0.0.0.0:4317 {"kind": "receiver", "name": "otl
2022-02-22T18:37:12.642-0600  info  otlpreceiver/otlp.go:87 Starting HTTP server on endpoint 0.0.0.0:4318 {"kind": "receiver", "name": "otl
2022-02-22T18:37:12.642-0600  info  otlpreceiver/otlp.go:147  Setting up a second HTTP listener on legacy endpoint 0.0.0.0:55681  {"kind":
2022-02-22T18:37:12.642-0600  info  otlpreceiver/otlp.go:87 Starting HTTP server on endpoint 0.0.0.0:55681  {"kind": "receiver", "name": "o
2022-02-22T18:37:12.642-0600  info  builder/receivers_builder.go:73 Receiver started. {"kind": "receiver", "name": "otlp"}
2022-02-22T18:37:12.642-0600  info  service/telemetry.go:95 Setting up own telemetry...
2022-02-22T18:37:12.643-0600  info  service/telemetry.go:115  Serving Prometheus metrics  {"address": ":8888", "level": "basic", "service.i
```

By looking at the log, it looks like the following ports are listening:

- 4317 (OTLP on GRPC)
- 4318 (OTLP on HTTP)
- 55681 (OTLP on HTTP - for legacy support)

There is also a prom endpoint that produces metrics for prometheus:



```
# HELP otelcol_exporter_enqueue_failed_log_records Number of log records failed to be added to the sending queue.
# TYPE otelcol_exporter_enqueue_failed_log_records counter
otelcol_exporter_enqueue_failed_log_records{exporter="logging",service_instance_id="c10d932c-e7f7-43d7-a256-a79081baf700",service_version="latest"} 0
otelcol_exporter_enqueue_failed_log_records{exporter="tanzuobservability",service_instance_id="c10d932c-e7f7-43d7-a256-a79081baf700",service_version="latest"} 0
# HELP otelcol_exporter_enqueue_failed_metric_points Number of metric points failed to be added to the sending queue.
# TYPE otelcol_exporter_enqueue_failed_metric_points counter
otelcol_exporter_enqueue_failed_metric_points{exporter="logging",service_instance_id="c10d932c-e7f7-43d7-a256-a79081baf700",service_version="latest"} 0
otelcol_exporter_enqueue_failed_metric_points{exporter="tanzuobservability",service_instance_id="c10d932c-e7f7-43d7-a256-a79081baf700",service_version="latest"} 0
# HELP otelcol_exporter_enqueue_failed_spans Number of spans failed to be added to the sending queue.
# TYPE otelcol_exporter_enqueue_failed_spans counter
otelcol_exporter_enqueue_failed_spans{exporter="logging",service_instance_id="c10d932c-e7f7-43d7-a256-a79081baf700",service_version="latest"} 0
otelcol_exporter_enqueue_failed_spans{exporter="tanzuobservability",service_instance_id="c10d932c-e7f7-43d7-a256-a79081baf700",service_version="latest"} 0
# HELP otelcol_exporter_queue_size Current size of the retry queue (in batches)
# TYPE otelcol_exporter_queue_size gauge
otelcol_exporter_queue_size{exporter="tanzuobservability",service_instance_id="c10d932c-e7f7-43d7-a256-a79081baf700",service_version="latest"} 0
# HELP otelcol_process_cpu_seconds Total CPU user and system time in seconds
# TYPE otelcol_process_cpu_seconds gauge
otelcol_process_cpu_seconds{service_instance_id="c10d932c-e7f7-43d7-a256-a79081baf700",service_version="latest"} 1.62
# HELP otelcol_process_memory_rss Total physical memory (resident set size)
# TYPE otelcol_process_memory_rss gauge
otelcol_process_memory_rss{service_instance_id="c10d932c-e7f7-43d7-a256-a79081baf700",service_version="latest"} 1.0706944e+08
# HELP otelcol_process_runtime_heap_alloc_bytes Bytes of allocated heap objects (see 'go doc runtime.MemStats.HeapAlloc')
# TYPE otelcol_process_runtime_heap_alloc_bytes gauge
otelcol_process_runtime_heap_alloc_bytes{service_instance_id="c10d932c-e7f7-43d7-a256-a79081baf700",service_version="latest"} 1.061868e+07
# HELP otelcol_process_runtime_total_alloc_bytes Cumulative bytes allocated for heap objects (see 'go doc runtime.MemStats.TotalAlloc')
# TYPE otelcol_process_runtime_total_alloc_bytes gauge
otelcol_process_runtime_total_alloc_bytes{service_instance_id="c10d932c-e7f7-43d7-a256-a79081baf700",service_version="latest"} 5.57265976e+08
# HELP otelcol_process_runtime_total_sys_memory_bytes Total bytes of memory obtained from the OS (see 'go doc runtime.MemStats.Sys')
# TYPE otelcol_process_runtime_total_sys_memory_bytes gauge
otelcol_process_runtime_total_sys_memory_bytes{service_instance_id="c10d932c-e7f7-43d7-a256-a79081baf700",service_version="latest"} 5.8421364e+08
# HELP otelcol_process_uptime Uptime of the process
# TYPE otelcol_process_uptime counter
otelcol_process_uptime{service_instance_id="c10d932c-e7f7-43d7-a256-a79081baf700",service_version="latest"} 480.00341000000003
```

Python

https://opentelemetry.io/docs/instrumentation/python/

Since Airflow is pretty much written in python, I'll have to take a look into how to write a code in python to instrumentalize metrics in airflow first, and then the traces.

Python packages to install

The following packages need to be installed (as dependency to airflow) in order for airflow to be able to run opentelemetry:

- opentelemetry-api
- opentelemetry-sdk
- opentelemetry-exporter-otlp

Run a simple `pip install opentelemetry-*` listed above to install them.

Initial Test

Created this simple test python client(?) to emit trace

```
from opentelemetry import trace
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import(
    OTLPSpanExporter,
)
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.sdk.resources import Resource

# exporter will be using port 4317 which is GRPC
# added http:// in the endpoint, since it looks like otel collector
# does not handle https by default. Since this is only a local test,
# this is perfectly fine.
span_exporter = OTLPSpanExporter(
    endpoint="http://localhost:4317"
)

resource = Resource(attributes={
    "service.name": "test-service",
    "application": "test-application"
})

tracer_provider = TracerProvider(resource=resource)
trace.set_tracer_provider(tracer_provider)
```

```
span_processor = BatchSpanProcessor(span_exporter)
tracer_provider.add_span_processor(span_processor)

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

with tracer.start_as_current_span("foo") as span:
  print("Hello world!")
  span.add_event("event message", {"event_attributes": 1})
  trace.get_current_span().set_attribute("some.message", "some message here")
  trace.get_current_span().set_attribute("TASK run id", "123123123132")

  with tracer.start_as_current_span("bar") as span2:
    print("say hi")
    span2.add_event("event two", {"log": "this is a log"})
    trace.get_current_span().set_attribute("TASK run id", "9087676")
```

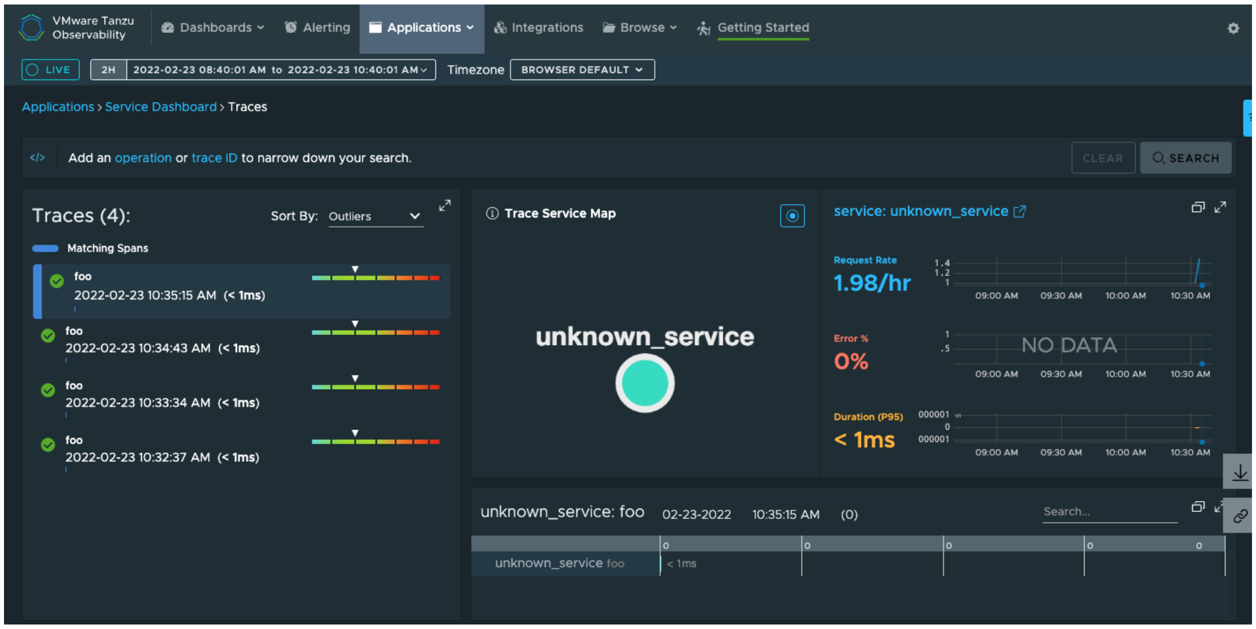
and ran with `python oteltest.py`

and got following response in otel collector's console output:

```
2022-02-23T11:36:30.879-0600  warn  zapgrpc/zapgrpc.go:191  [transport] transport: http2Server.HandleStreams failed to read frame: read tcp
2022-02-23T11:36:31.061-0600  INFO  loggingexporter/logging_exporter.go:40  TracesExporter  {"#spans": 2}
2022-02-23T11:36:31.061-0600  DEBUG loggingexporter/logging_exporter.go:49  ResourceSpans #0
Resource SchemaURL:
Resource labels:
     -> service.name: STRING(test-service)
     -> application: STRING(test-application)
InstrumentationLibrarySpans #0
InstrumentationLibrarySpans SchemaURL:
InstrumentationLibrary __main__
Span #0
    Trace ID       : f27200bfeff315e5d62bc1d562621ac0
    Parent ID      : 50e763f80bed85e0
    ID             : 230904234409e501
    Name           : bar
    Kind           : SPAN_KIND_INTERNAL
    Start time     : 2022-02-23 17:36:30.86448 +0000 UTC
    End time       : 2022-02-23 17:36:30.864496 +0000 UTC
    Status code    : STATUS_CODE_UNSET
    Status message :
Attributes:
     -> TASK run id: STRING(9087676)
Span #1
    Trace ID       : f27200bfeff315e5d62bc1d562621ac0
    Parent ID      :
    ID             : 50e763f80bed85e0
    Name           : foo
    Kind           : SPAN_KIND_INTERNAL
    Start time     : 2022-02-23 17:36:30.864418 +0000 UTC
    End time       : 2022-02-23 17:36:30.864505 +0000 UTC
    Status code    : STATUS_CODE_UNSET
    Status message :
Attributes:
     -> some.message: STRING(some message here)
     -> TASK run id: STRING(123123123132)
Events:
SpanEvent #0
     -> Name: event message
     -> Timestamp: 2022-02-23 17:36:30.86445 +0000 UTC
     -> DroppedAttributesCount: 0
     -> Attributes:
         -> event_attributes: INT(1)
SpanEvent #1
     -> Name: event two
     -> Timestamp: 2022-02-23 17:36:30.864489 +0000 UTC
     -> DroppedAttributesCount: 0
     -> Attributes:
         -> log: STRING(this is a log)
```

For now, I'm going to ignore the warnings - but I was glad to see the trace being received correctly.

I also checked the Wavefront side and found the following traces:


VMware Tanzu Observability
Dashboards
Alerting
Applications
Integrations
Browse
Getting Started
LIVE
2H
2022-02-23 08:40:01 AM to 2022-02-23 10:40:01 AM
Timezone
BROWSER DEFAULT
Applications > Service Dashboard > Traces
Add an operation or trace ID to narrow down your search.
CLEAR
SEARCH
Traces (4):
Sort By: Outliers
Matching Spans
foo 2022-02-23 10:35:15 AM (< 1ms)
foo 2022-02-23 10:34:43 AM (< 1ms)
foo 2022-02-23 10:33:34 AM (< 1ms)
foo 2022-02-23 10:32:37 AM (< 1ms)
Trace Service Map
unknown_service
service: unknown_service
Request Rate
1.98/hr
Error %
0%
NO DATA
Duration (P95)
< 1ms
09:00 AM
09:30 AM
10:00 AM
10:30 AM
unknown_service: foo 02-23-2022 10:35:15 AM (0)
Search...
unknown_service foo
< 1ms

So, there's obviously 'service' or 'application' that needs to be defined somewhere (by default, if wavefront does not detect a specific service id, it will simply define it as 'unknown_service' - in the future we should be putting the 'DAG name' as its service name.

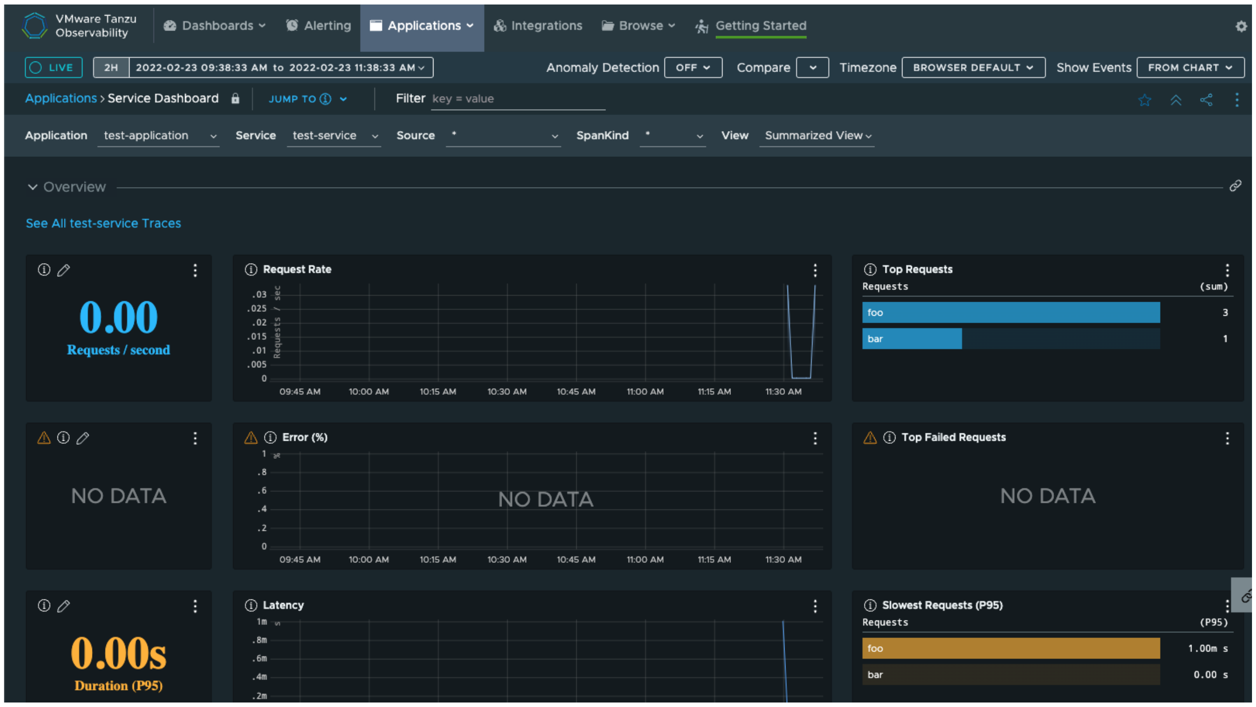
So, the initial test is successful. We'll be doing some more tests to test linkages, parent-child relationships, etc.

How to describe a DAG run

After doing some tests, have decided to try out these:

- Trace has resource def. of `application` and `service` . Looks like application should be of a particular instance of airflow (e.g. deployment) and hence should have the deployment name (or ID) of an airflow. This might be something that we can put into the configuration (airflow.conf) file.
- Service should be the 'DAG' (like DAG ID) - which should identify a particular service.
- DAG run should be root Span
- TASK run should be child Span of the root span (DAG run)
- Optionally, Executor should be the child Span of the TASK run.
- TASK runs should be linked to each other using span links - type of 'dependency.' The dependency should have the directional aspect of which is pointing to which.

Any other notes should be taken as we go along the test.



Branching the Airflow github repo

I forked apache airflow in https://github.com/apache/airflow into my howardyoo account, and then further created a fork called `opentelemetry-poc-1` and pulled it into my local env.

Working on configs

The first thing that I wanted to work on was the configuration. Currently, there's a configuration for statsd_* inside `airflow.cfg` file, so I wanted to follow the same convention by introducing a bunch of configuration parameter of `otel_*` configs.

```
[metrics]

# StatsD (https://github.com/etsy/statsd) integration settings.
# Enables sending metrics to StatsD.
statsd_on = False
statsd_host = localhost
statsd_port = 8125
statsd_prefix = airflow

# OpenTelemetry (https://github.com/open-telemetry) integration settings.
# enables metrics and traces to Otel collecting agent
otel_on = True
otel_host = localhost
otel_port = 4317
otel_prefix = airflow
```

- Assumption is that http:// protocol will be used, and 4317 should be the otel collector's GRPC port.

Adding Traces

Tracer - should be singleton on a single Apache instance (usually resides inside a module)

Usually traces works 'tail' based - which means the Trace information will be finalized and submitted not when it STARTs, but when it ENDS. Therefore, the following cases can be considered for producing trace / spans

- When DAG run ends (closes off with the run)
- When TASK run ends

- When Executor run ends(?)

I have to dig into the details, but it looks like the one that produces when the tasks are ended - is driven by the 'scheduler.' So, it is possible that the scheduler should be the one that produces such traces.

- Added codes similar to how `Stats` works in current airflow, named it `Trace` .
- Has dummy tracer which will get instantiated when configuration is not available or disabled.
- OtelTraceLogger will be instantiated when otel_on is detected in `airflow.cfg` file.
- Accessing it will be via class method as such:

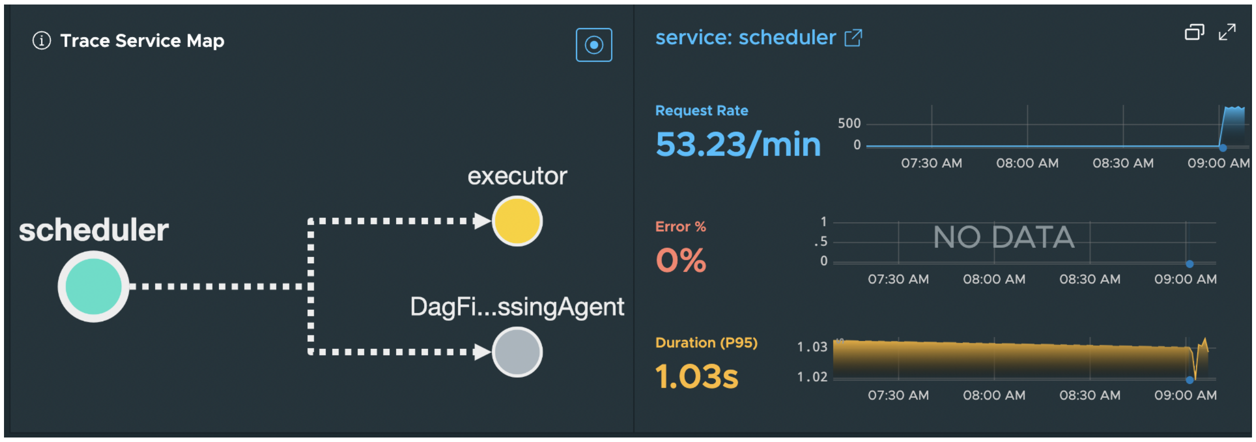
```
from airflow.stats import Trace
...
tracer = Trace.get_tracer("my_tracer")
with tracer.start_as_current_span("something") as span:
    ....
```

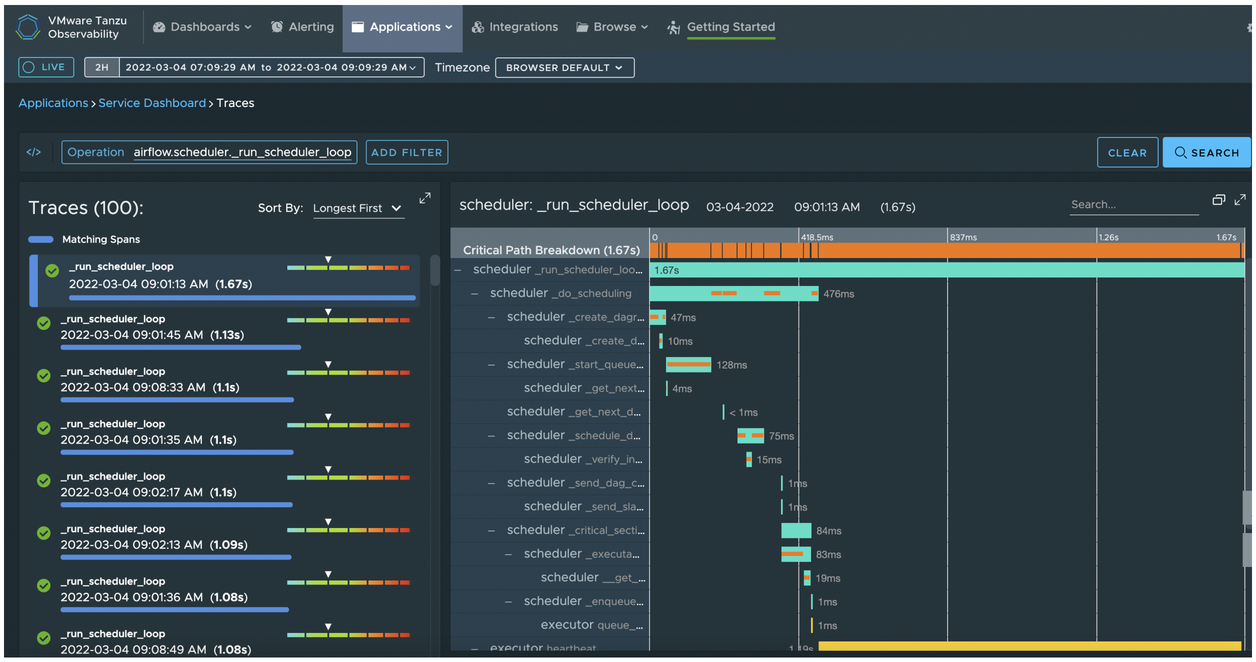
First goal is to create traces for DAG runs, followed by TASK runs.

- there has to be a consideration for creating span with specific Trace ID either from context or from database, as task runs can be executed remotely / asynchronously outside of the parent's domain.

Traces for Scheduler

Adding traces for scheduler involves using Trace to generate tracer which can then be either used inside a function or as a decorator. The following traces were implemented to monitor the performance of each 'schedule loop' that scheduler goes through every 1~2 seconds.

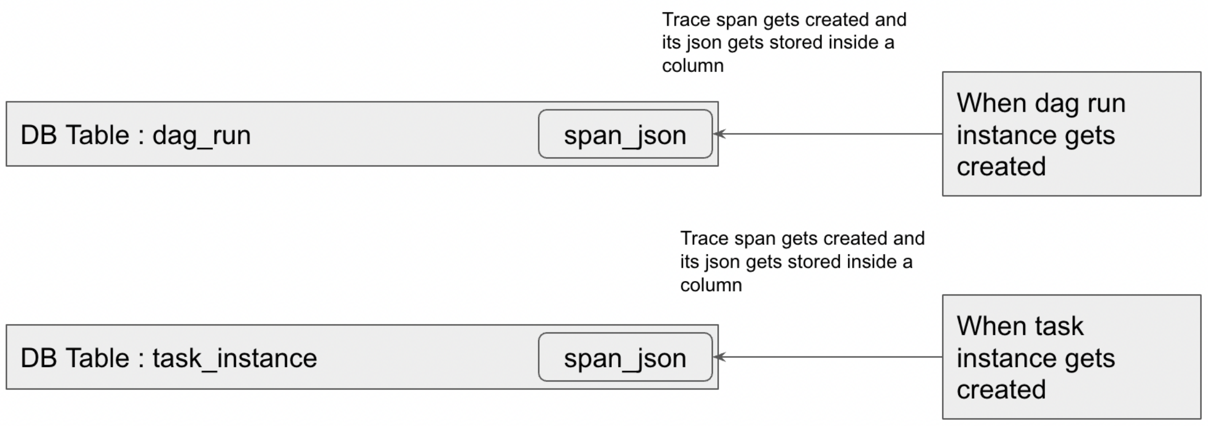



VMware Tanzu Observability
Dashboards
Alerting
Applications
Integrations
Browse
Getting Started
LIVE
2H
2022-03-04 07:09:29 AM to 2022-03-04 09:09:29 AM
Timezone
BROWSER DEFAULT
Applications > Service Dashboard > Traces
Operation airflow.scheduler._run_scheduler_loop
ADD FILTER
CLEAR
SEARCH
Traces (100):
Sort By: Longest First
Matching Spans
_run_scheduler_loop 2022-03-04 09:01:13 AM (1.67s)
_run_scheduler_loop 2022-03-04 09:01:45 AM (1.13s)
_run_scheduler_loop 2022-03-04 09:08:33 AM (1.1s)
_run_scheduler_loop 2022-03-04 09:01:35 AM (1.1s)
_run_scheduler_loop 2022-03-04 09:02:17 AM (1.1s)
_run_scheduler_loop 2022-03-04 09:02:13 AM (1.09s)
_run_scheduler_loop 2022-03-04 09:01:36 AM (1.08s)
_run_scheduler_loop 2022-03-04 09:08:49 AM (1.08s)
scheduler: _run_scheduler_loop 03-04-2022 09:01:13 AM (1.67s)
Search...
Critical Path Breakdown (1.67s)
0
418.5ms
837ms
1.26s
1.67s
scheduler _run_scheduler_loo... 1.67s
scheduler _do_scheduling 476ms
scheduler _create_dagr... 47ms
scheduler _create_d... 10ms
scheduler _start_queue... 128ms
scheduler _get_next... 4ms
scheduler _get_next_d... < 1ms
scheduler _schedule_d... 75ms
scheduler _verify_in... 15ms
scheduler _send_dag_c... 1ms
scheduler _send_sla... 1ms
scheduler _critical_secti... 84ms
scheduler _executa... 83ms
scheduler __get_... 19ms
scheduler _enqueue... 1ms
executor queue_... 1ms
executor heartbeat

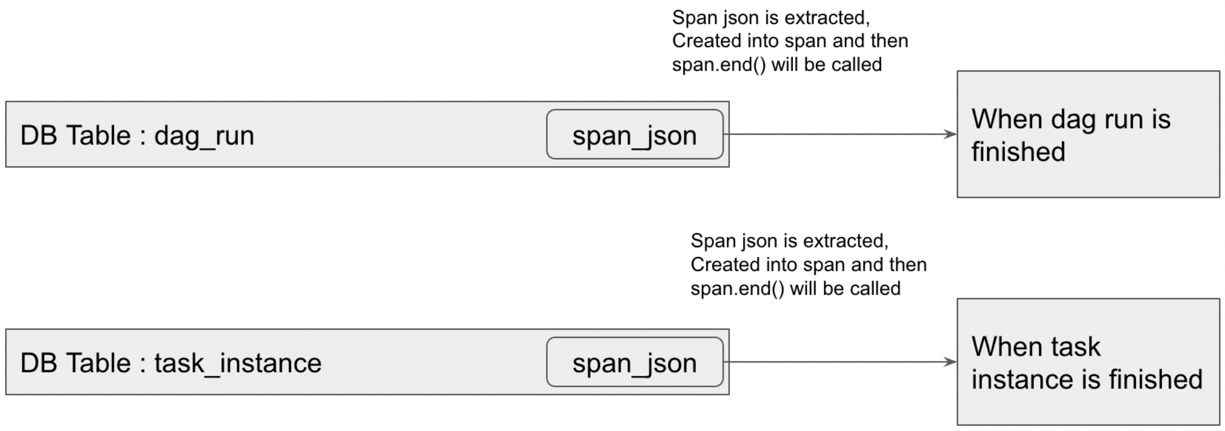
Traces for DAG Runs

DAG runs are a little bit difficult than tracing scheduler runs. That is because a DAG run does not necessarily run everything at the 'run time,' but can span across wider time periods. Also, a single DAG run consists of 'asynchronous' execution of executors fetching the task instance commands and running them separately - so propagating the proper span context can be challenging.

So, when the DAG is created (and eventually sync'ed into DB later), a new span is also created, and its span's JSON representation would be stored inside an additional column called 'span_json' on `dag_run` and `task_instance` table.

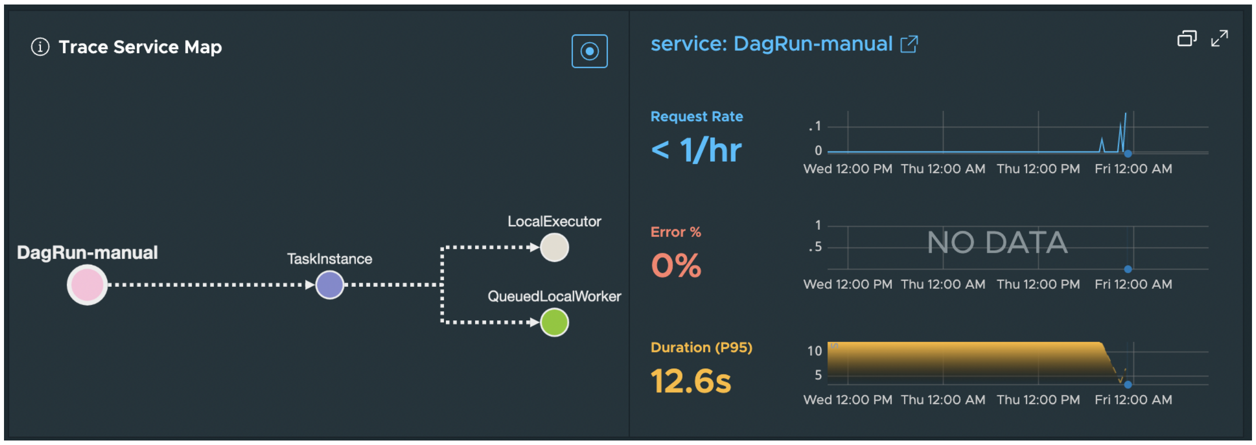

Trace span gets created and its json gets stored inside a column
DB Table : dag_run
span_json
When dag run instance gets created
Trace span gets created and its json gets stored inside a column
DB Table : task_instance
span_json
When task instance gets created

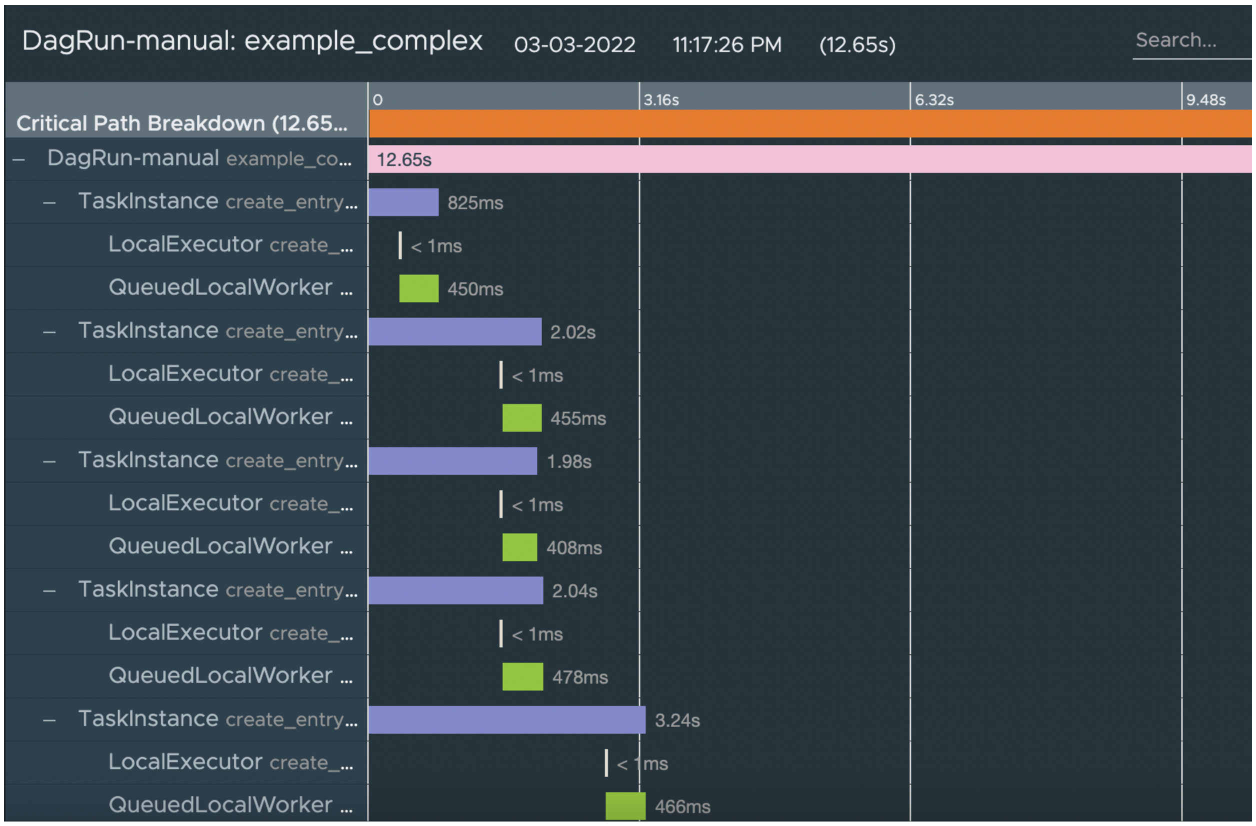
At the end of the DAG execution (probably going to be triggered by something, but it doesn't matter), the code will than extract the JSON data from `span_json` column, and then it will parse it back to a span object. Then, we can call span.end() to then emit the data to open telemetry, without having to worry about the context or the attributes - as those will be safely persisted and maintained inside the database.


Span json is extracted,
Created into span and then
span.end() will be called
DB Table : dag_run
span_json
When dag run is finished
Span json is extracted,
Created into span and then
span.end() will be called
DB Table : task_instance
span_json
When task instance is finished

Here is an example of how a DAG run looks in two categories, manually triggered DAG runs, and DAG runs triggered by the scheduler. Each has different characteristics, as scheduler would contain scheduler triggering the run, while manual DAG run does not.

Manual DAG Runs

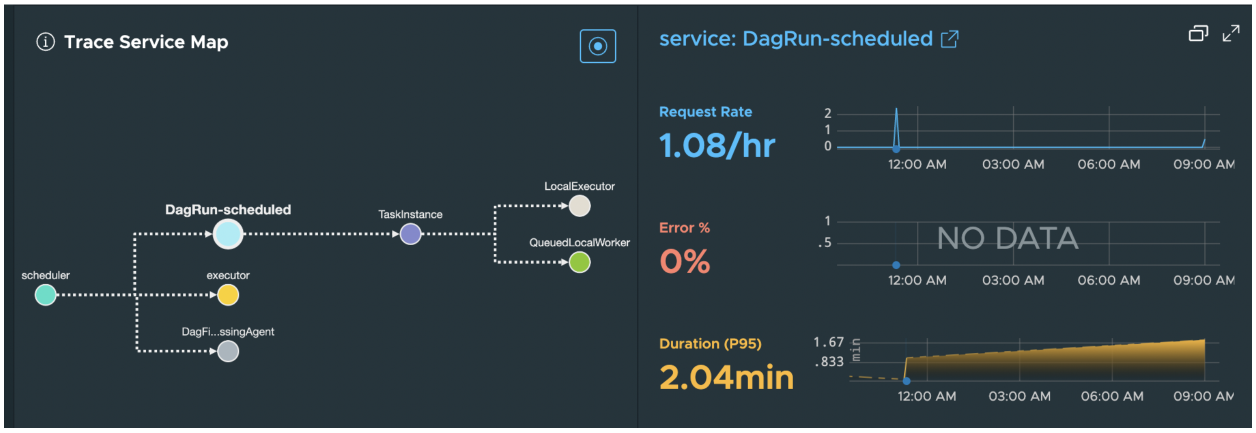


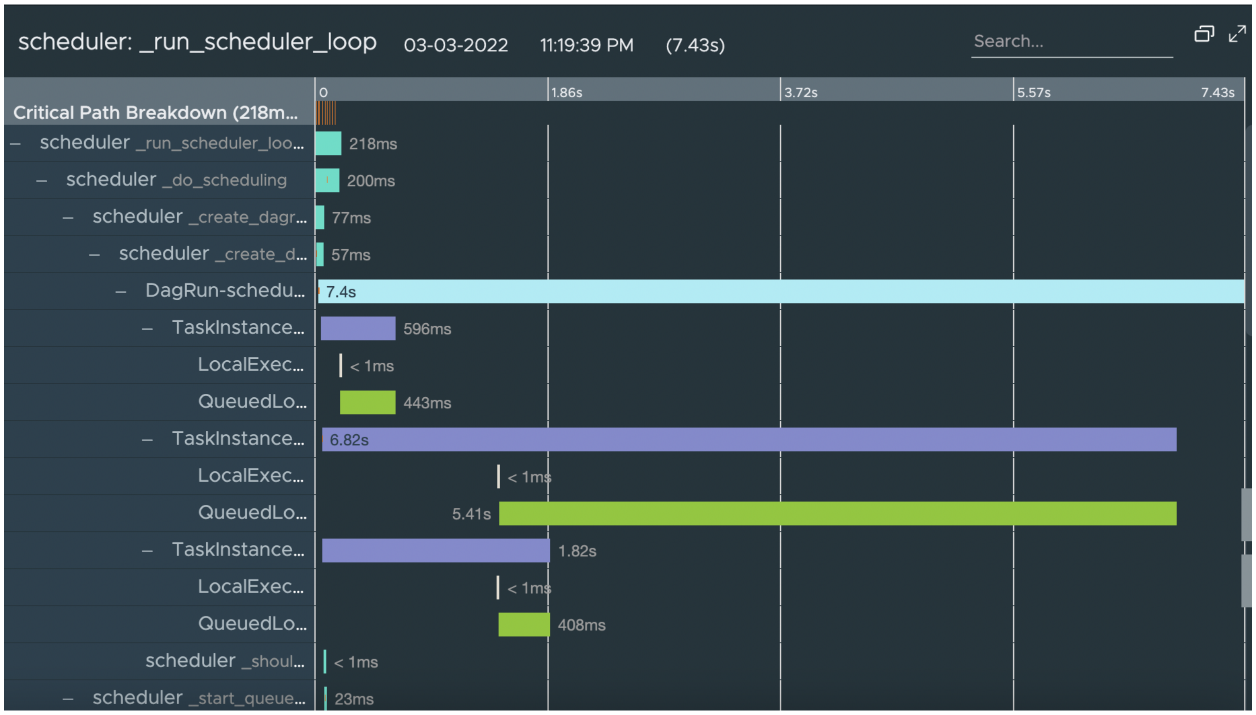

DagRun-manual: example_complex 03-03-2022 11:17:26 PM (12.65s)

Search...

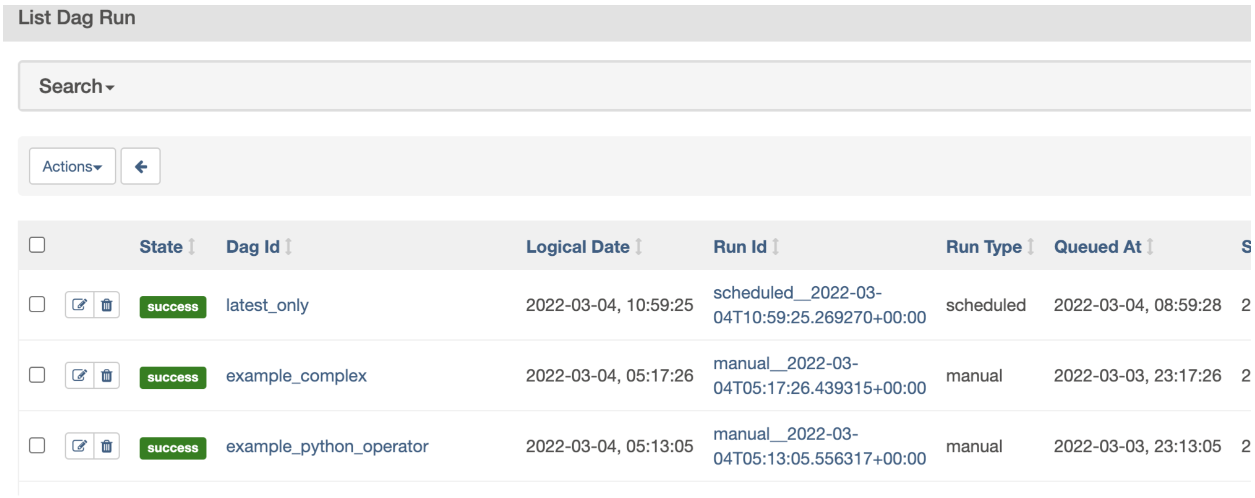
	0 · 3.16s · 6.32s · 9.48s
Critical Path Breakdown (12.65...	
– DagRun-manual example_co...	12.65s
– TaskInstance create_entry...	825ms
LocalExecutor create_...	< 1ms
QueuedLocalWorker ...	450ms
– TaskInstance create_entry...	2.02s
LocalExecutor create_...	< 1ms
QueuedLocalWorker ...	455ms
– TaskInstance create_entry...	1.98s
LocalExecutor create_...	< 1ms
QueuedLocalWorker ...	408ms
– TaskInstance create_entry...	2.04s
LocalExecutor create_...	< 1ms
QueuedLocalWorker ...	478ms
– TaskInstance create_entry...	3.24s
LocalExecutor create_...	< 1ms
QueuedLocalWorker ...	466ms

Scheduled DAG Runs



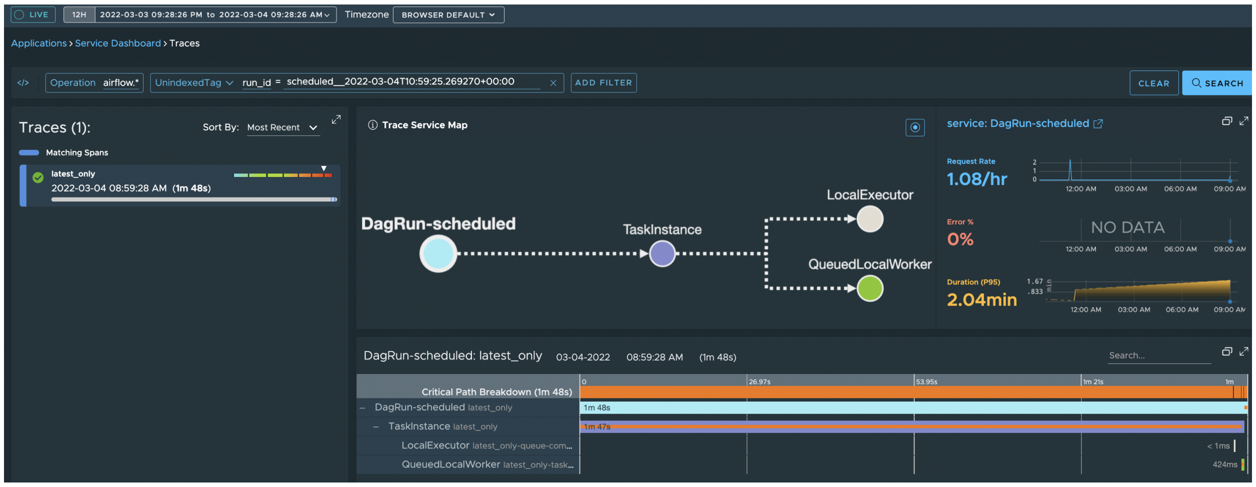


So, if you see something in airflow that looks something like this:



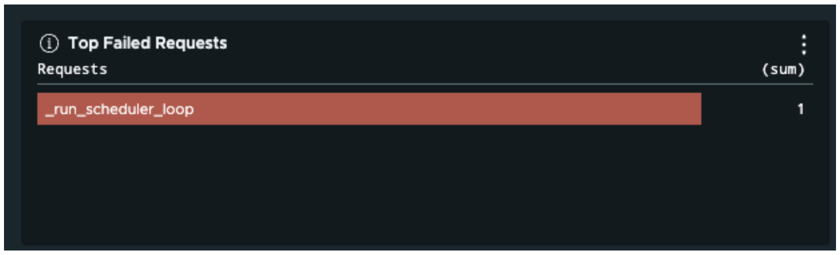
		State	Dag Id	Logical Date	Run Id	Run Type	Queued At	S
		success	latest_only	2022-03-04, 10:59:25	scheduled__2022-03-04T10:59:25.269270+00:00	scheduled	2022-03-04, 08:59:28	2
		success	example_complex	2022-03-04, 05:17:26	manual__2022-03-04T05:17:26.439315+00:00	manual	2022-03-03, 23:17:26	2
		success	example_python_operator	2022-03-04, 05:13:05	manual__2022-03-04T05:13:05.556317+00:00	manual	2022-03-03, 23:13:05	2

By using the ‘run_id’ scheduled__2022-03-04T10:59:25.269270+00:00, you can easily search in the trace list to see the details of the run as such:

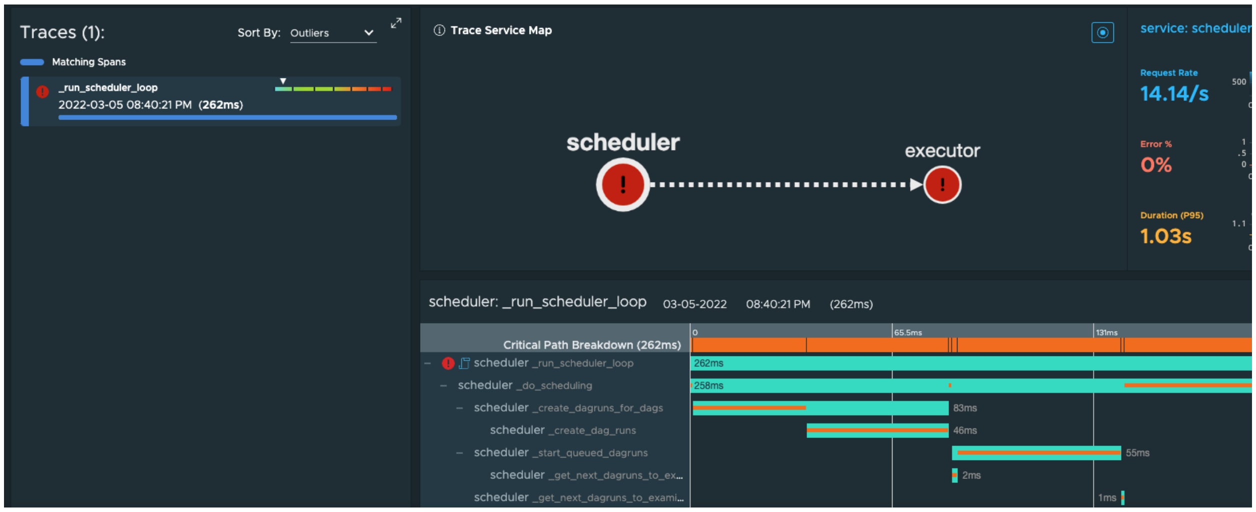

LIVE
12H 2022-03-03 09:28:26 PM to 2022-03-04 09:28:26 AM
Timezone BROWSER DEFAULT
Applications > Service Dashboard > Traces
Operation airflow.*
UnindexedTag run_id = scheduled__2022-03-04T10:59:25.269270+00:00
ADD FILTER
CLEAR
SEARCH
Traces (1):
Sort By: Most Recent
Matching Spans
latest_only
2022-03-04 08:59:28 AM (1m 48s)
Trace Service Map
DagRun-scheduled
TaskInstance
LocalExecutor
QueuedLocalWorker
service: DagRun-scheduled
Request Rate
1.08/hr
Error %
0%
NO DATA
Duration (P95)
2.04min
DagRun-scheduled: latest_only 03-04-2022 08:59:28 AM (1m 48s)
Critical Path Breakdown (1m 48s)
DagRun-scheduled latest_only 1m 48s
TaskInstance latest_only 1m 47s
LocalExecutor latest_only-queue-com... < 1ms
QueuedLocalWorker latest_only-task... 424ms

Errors

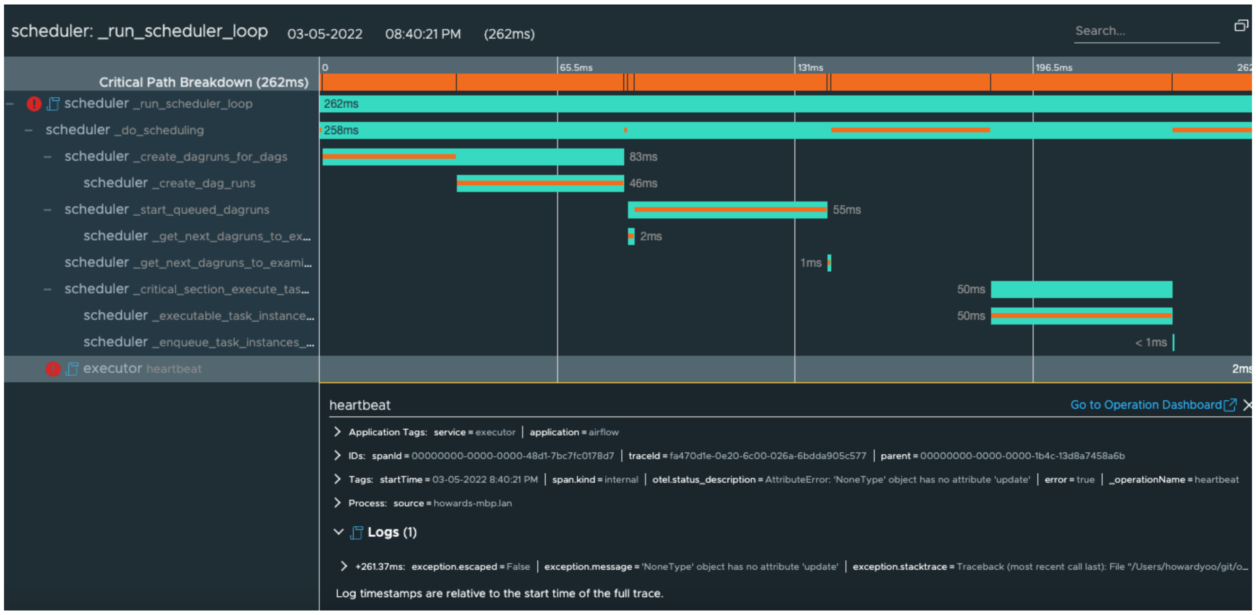
Traces also capture events like error cases in a form of span logs, a good example is when a schedule loop fails:



Which can then get to the trace the resulted in error:


Traces (1):
Sort By: Outliers
Matching Spans
_run_scheduler_loop
2022-03-05 08:40:21 PM (262ms)
Trace Service Map
scheduler
executor
service: scheduler
Request Rate
14.14/s
Error %
0%
Duration (P95)
1.03s
scheduler: _run_scheduler_loop 03-05-2022 08:40:21 PM (262ms)
Critical Path Breakdown (262ms)
scheduler _run_scheduler_loop 262ms
scheduler _do_scheduling 258ms
scheduler _create_dagruns_for_dags 83ms
scheduler _create_dag_runs 46ms
scheduler _start_queued_dagruns 55ms
scheduler _get_next_dagruns_to_ex... 2ms
scheduler _get_next_dagruns_to_exami... 1ms

By looking at the traces, we can diagnose that ‘executor’ heartbeat was the root case of the error:


scheduler: _run_scheduler_loop 03-05-2022 08:40:21 PM (262ms)
Search...
Critical Path Breakdown (262ms)
0
65.5ms
131ms
196.5ms
scheduler _run_scheduler_loop
262ms
scheduler _do_scheduling
258ms
scheduler _create_dagruns_for_dags
83ms
scheduler _create_dag_runs
46ms
scheduler _start_queued_dagruns
55ms
scheduler _get_next_dagruns_to_ex...
2ms
scheduler _get_next_dagruns_to_exami...
1ms
scheduler _critical_section_execute_tas...
50ms
scheduler _executable_task_instance...
50ms
scheduler _enqueue_task_instances_...
< 1ms
executor heartbeat
2ms
Go to Operation Dashboard

heartbeat

Application Tags: service = executor | application = airflow

IDs: spanId = 00000000-0000-0000-48d1-7bc7fc0178d7 | traceId = fa470d1e-0e20-6c00-026a-6bdda905c577 | parent = 00000000-0000-0000-1b4c-13d8a7458a6b

Tags: startTime = 03-05-2022 8:40:21 PM | span.kind = internal | otel.status_description = AttributeError: 'NoneType' object has no attribute 'update' | error = true | _operationName = heartbeat

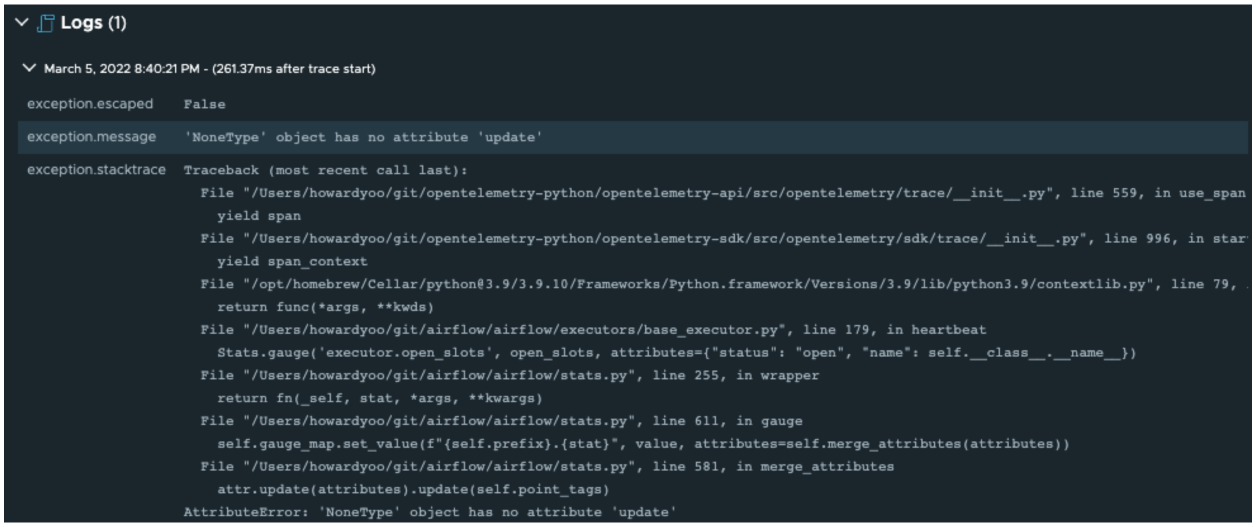
Process: source = howards-mbp.lan

Logs (1)

+261.37ms: exception.escaped = False | exception.message = 'NoneType' object has no attribute 'update' | exception.stacktrace = Traceback (most recent call last): File "/Users/howardyoo/git/o…

Log timestamps are relative to the start time of the full trace.

Opening up the error will reveal the error message that has Traceback of the code where the error has happened, pretty convenient to quickly troubleshoot any runs failing.



Logs (1)

March 5, 2022 8:40:21 PM - (261.37ms after trace start)

```
exception.escaped     False
exception.message     'NoneType' object has no attribute 'update'
exception.stacktrace  Traceback (most recent call last):
                        File "/Users/howardyoo/git/opentelemetry-python/opentelemetry-api/src/opentelemetry/trace/__init__.py", line 559, in use_span
                          yield span
                        File "/Users/howardyoo/git/opentelemetry-python/opentelemetry-sdk/src/opentelemetry/sdk/trace/__init__.py", line 996, in star
                          yield span_context
                        File "/opt/homebrew/Cellar/python@3.9/3.9.10/Frameworks/Python.framework/Versions/3.9/lib/python3.9/contextlib.py", line 79, .
                          return func(*args, **kwds)
                        File "/Users/howardyoo/git/airflow/airflow/executors/base_executor.py", line 179, in heartbeat
                          Stats.gauge('executor.open_slots', open_slots, attributes={"status": "open", "name": self.__class__.__name__})
                        File "/Users/howardyoo/git/airflow/airflow/stats.py", line 255, in wrapper
                          return fn(_self, stat, *args, **kwargs)
                        File "/Users/howardyoo/git/airflow/airflow/stats.py", line 611, in gauge
                          self.gauge_map.set_value(f"{self.prefix}.{stat}", value, attributes=self.merge_attributes(attributes))
                        File "/Users/howardyoo/git/airflow/airflow/stats.py", line 581, in merge_attributes
                          attr.update(attributes).update(self.point_tags)
                      AttributeError: 'NoneType' object has no attribute 'update'
```

Adding Metrics

https://open-telemetry.github.io/opentelemetry-python/getting-started.html#add-metrics describes examples of how you should add metrics. I did not want to introduce too much additional coding to replace what's currently been implemented using `stats.py` , and thus was thinking maybe I could use much of existing framework, but only replace what's implemented in stats.py to change the behavior. A good example of how dogstatsd was implemented would give me much of the precursor.

⚠️ Due to the fact that opentelemetry's Metric implementation was NOT complete on version 1.9.1 (which was installable via pip), I had to clone the main branch of opentelemetry (master) and do a editable build(?) as such:

```
pip uninstall opentelemetry-sdk
pip uninstall opentelemetry-api
....
git clone https://github.com/open-telemetry/opentelemetry-python.git
cd ./opentelemetry-python
```

```
pip install -e ./opentelemetry-sdk
pip install -e ./opentelemetry-api
```

But running my test metric python script yielded in error:

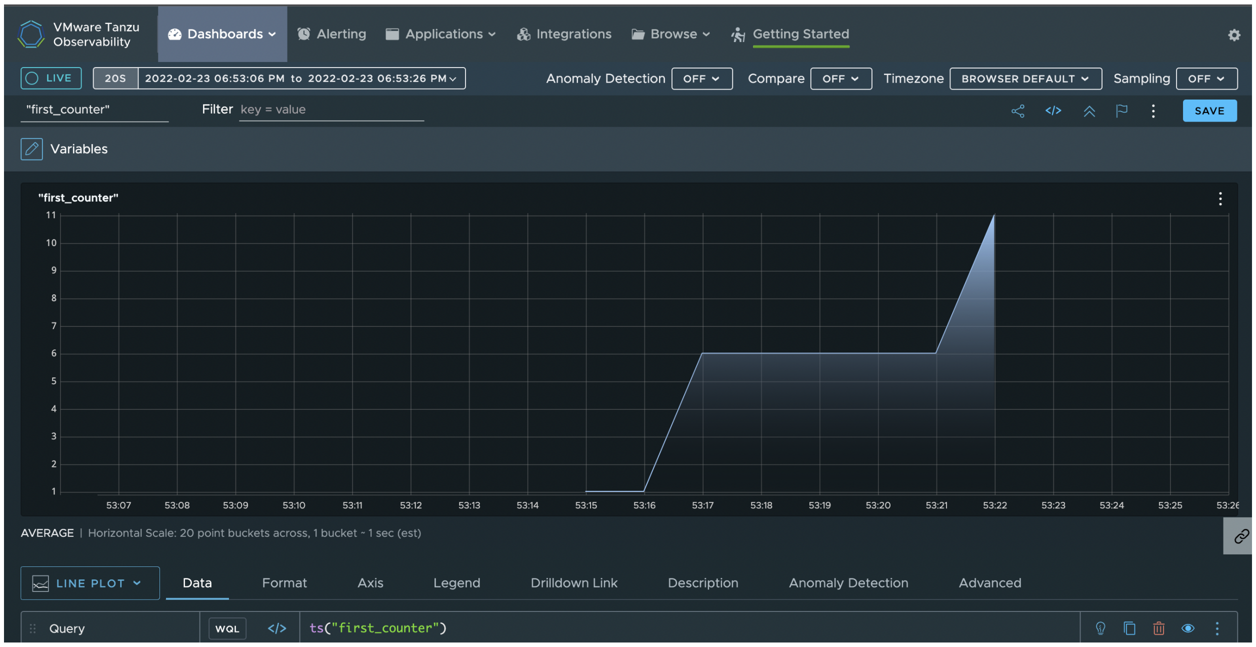
There was a minor bug at `_view_instrument_match.py` line 71, which was corrected as such:

```
if attributes not in self._attributes_aggregation.keys():
            with self._lock:
                # self._attributes_aggregation[attributes] = self._aggregation()
                self._attributes_aggregation[attributes] = self._aggregation
```

And the whole thing started to work perfectly! So, went ahead and created a PR for this bug fix in open telemetry git repo : https://github.com/open-telemetry/opentelemetry-python/pull/2479

this produced a counter metric which looked like this in Wavefront:

```
from re import I
import time
from opentelemetry._metrics import get_meter_provider, set_meter_provider
from opentelemetry.sdk._metrics.export import ConsoleMetricExporter
from opentelemetry.exporter.otlp.proto.grpc._metric_exporter import (
    OTLPMetricExporter,
)
from opentelemetry.sdk._metrics import MeterProvider
from opentelemetry.sdk._metrics.export import PeriodicExportingMetricReader

print("setting up OTLP metric exporter..")
exporter = OTLPMetricExporter(endpoint="http://localhost:4317", insecure=True)
# exporter = ConsoleMetricExporter()

print("setting up metric reader..")
reader = PeriodicExportingMetricReader(exporter=exporter, export_interval_millis=1000)
provider = MeterProvider(metric_readers=[reader])

print("setting meter provider..")
set_meter_provider(provider)

print("getting meter provider..")
# meter = get_meter_provider().get_meter("getting-started")
meter = provider.get_meter("howard")
print(meter)

print("creating counter..")
counter = meter.create_counter("first_counter")
print("adding 1")
counter.add(1)

time.sleep(3)
print("adding 5")
counter.add(5)

time.sleep(5)
print("adding 5")
counter.add(5)
```

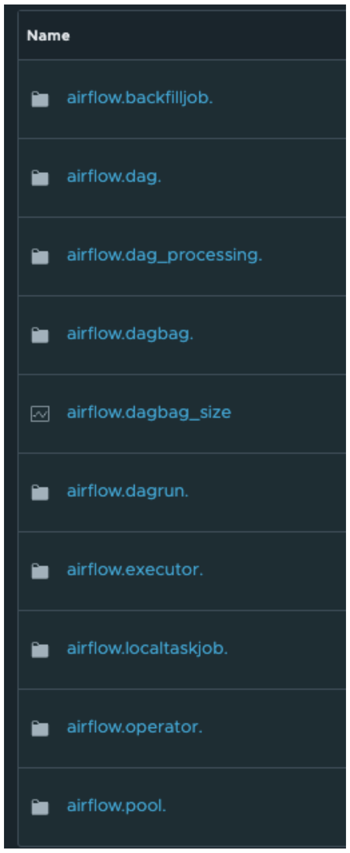


```
ts("first_counter")
```

So, essentially, we are now ready to also instrument OTEL on airflow to produce metrics.

So, where should I be adding these metrics? Well, at the same location as the existing code that produces metrics on stats.

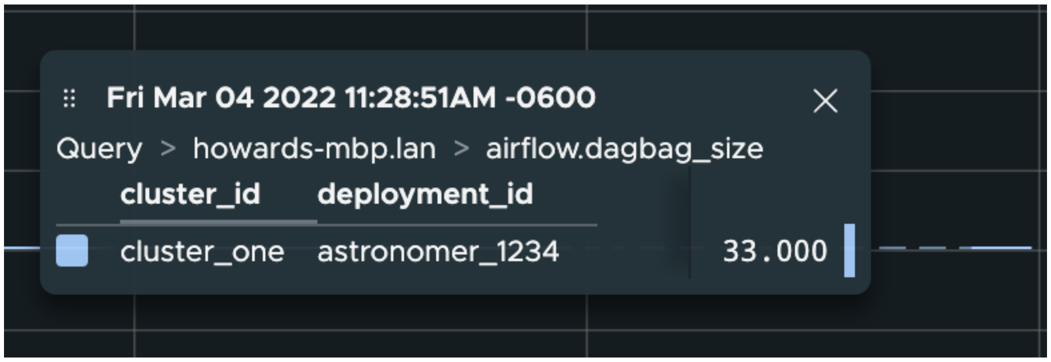
- Did some research yesterday - and found out that while counters work pretty well, Gauges require a completely different strategy to collect and report data. Opentelemetry pretty delegate the collection(or observing) of gauge data using call back method, which needs to report all the measurements (and perhaps maintain them) as necessary.
- Today, worked on to modify stats.py (which contains statsd implementation of airflow) to also process and send metrics to OTEL (using OTEL exporter) - along with additional configuration that user can enable otel instead of statsd. Currently, the same set of metrics are being delivered to Wavefront.



Name
airflow.backfilljob.
airflow.dag.
airflow.dag_processing.
airflow.dagbag.
airflow.dagbag_size
airflow.dagrun.
airflow.executor.
airflow.localtaskjob.
airflow.operator.
airflow.pool.

Next step would be to not only send the same metrics, but send in more advanced way (such as supporting point tags) so that metrics could have much more context. Also, possibly adding additional instrumentation to collect additional metrics could be explored.

- Also used a section called [metrics_tags] section to denote any static tags (e.g. deployment_id, cluster_id, etc..) to attach to the metrics.



Fri Mar 04 2022 11:28:51AM -0600

Query > howards-mbp.lan > airflow.dagbag_size

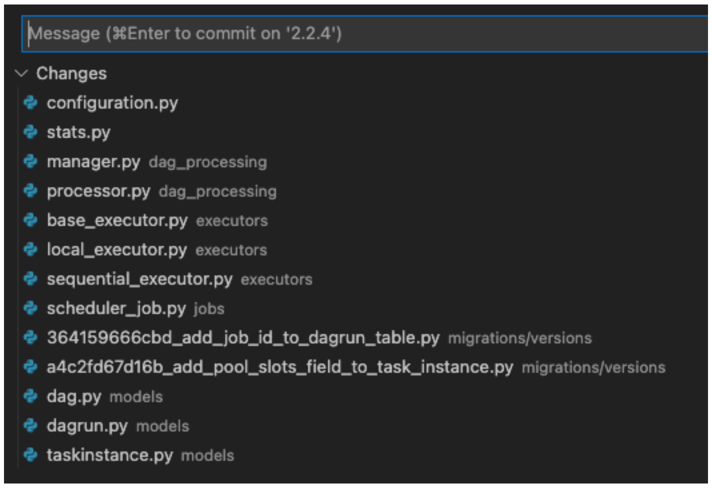
cluster_id	deployment_id	
cluster_one	astronomer_1234	33.000

Works to be done

- add default metrics tags to Traces as well - there isn't much automation that can be done, so looking for a nicer way to do it.

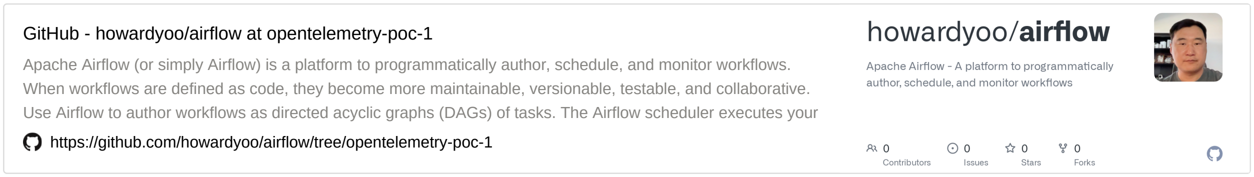
Source Codes

Here are the list of sources that has changed for this poc:



```
Message (⌘Enter to commit on '2.2.4')
Changes
configuration.py
stats.py
manager.py dag_processing
processor.py dag_processing
base_executor.py executors
local_executor.py executors
sequential_executor.py executors
scheduler_job.py jobs
364159666cbd_add_job_id_to_dagrun_table.py migrations/versions
a4c2fd67d16b_add_pool_slots_field_to_task_instance.py migrations/versions
dag.py models
dagrun.py models
taskinstance.py models
```

Please refer to here :



GitHub - howardyoo/airflow at opentelemetry-poc-1

Apache Airflow (or simply Airflow) is a platform to programmatically author, schedule, and monitor workflows. When workflows are defined as code, they become more maintainable, versionable, testable, and collaborative. Use Airflow to author workflows as directed acyclic graphs (DAGs) of tasks. The Airflow scheduler executes your

https://github.com/howardyoo/airflow/tree/opentelemetry-poc-1

howardyoo/**airflow**

Apache Airflow - A platform to programmatically author, schedule, and monitor workflows

0 Contributors 0 Issues 0 Stars 0 Forks

for details related to the source codes.

Note that the source code was created from the branch `2.2.4`.