You have written your web application, now what?

Amit Saha
https://echorand.me
@echorand

Strategies to make your services failure ready

Monolith or a service among 1000s of microservices

About me

Site Reliability Engineer at Atlassian - I am located in Sydney, Australia

Past - Software engineer, DevOps Engineer (Red Hat/Freelancer.com/Sun Microsystems/Startups)

Author of “Doing Math with Python” (No Starch Press)

Blog: https://echorand.me

Agenda

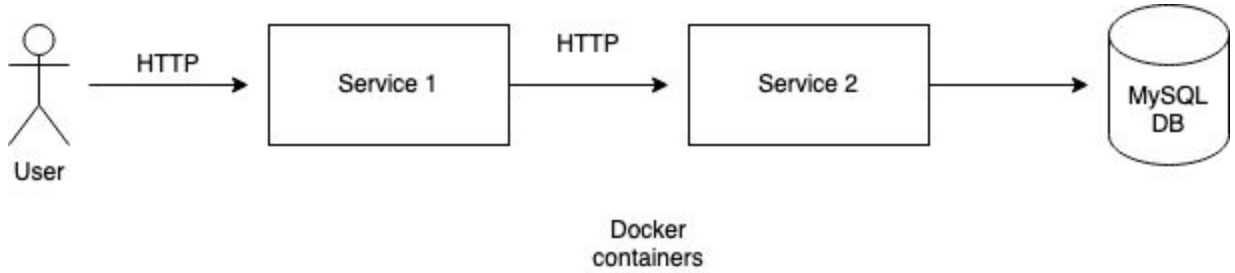
- Logging
- Metrics
- Distributed Tracing

Observability

- Automatic Scaling
- Chaos engineering

Resiliency

Our demo service architecture



Demos and resources

https://github.com/amitsaha/python-web-conf-2021

OBSERVABILITY
(Application Instrumentation)

Logging

Structured logging

- JSON formatted log lines
  - Each log line is a JSON formatted string
  - Your logs become easily searchable/indexable/graphable etc!
- Configure it for your web framework
  - Flask
  - Django
- Configure it for your WSGI runtime
  - gunicorn
  - uwsgi

I recommend: https://github.com/madzak/python-json-logger

Correlated request/error logging

1. Every incoming request is associated with a unique ID
2. Log the ID with any log message/error/exception associated with the request
3. Propagate the request ID to any upstream service calls

Super useful in micro-services / service oriented architecture

Set the request ID and attach it to the request context

```
REQUEST_ID_HEADERS = ["X-Trace-ID", "X-Request-ID"]


def get_or_set_request_id():
    # TODO: Ideally check if we are in a request context
    for h in REQUEST_ID_HEADERS:
        if h in request.headers:
            return request.headers[h]
    return str(uuid.uuid4())

@app.before_request
def log_request():
    request.request_id = get_or_set_request_id()
    # other stuff ...
```

Log the request ID automatically

```
class CustomJsonFormatter(jsonlogger.JsonFormatter):
    def add_fields(self, log_record, record, message_dict):
        super(CustomJsonFormatter, self).add_fields(log_record, record, message_dict)
        if not log_record.get('request_id'):
            log_record['request_id'] = request.request_id
```

Propagate the request ID

```
def do_stuff():
    headers = {'X-Request-ID': request.request_id}
    return requests.get('http://service2:8000', headers=headers)
```

Propagate the Request ID to requests to another service

- For RPC services, use a dedicated field in the request
- For gRPC, use metadata object to pass the ID

Logs from service1 and service2

```
service2_1    | {"message": "Request processed", "request_id": "a87f9a23-ddc0-4768-9f3d-80743a8897fc",
"request_path": "/", "response_status": 200, "timestamp": "2021-03-09T00:14:10.105562Z"}

service1_1    | {"message": "http://service2:8000 \"GET / HTTP/1.1\" 200 69", "timestamp":
"2021-03-09T00:14:10.107380Z", "request_id": "a87f9a23-ddc0-4768-9f3d-80743a8897fc"}
```

Send back the Request ID

```python
@app.after_request
def log_response(response):
    # other stuff..
    response.headers['X-Request-ID'] = request.request_id
    return response
```

When your user (external/internal) tells you of an issue, you can then ask them to provide the Request ID - which you can then use to start your investigation - how nice!

Demo

Key things to log

- HTTP applications
  - Request path/method, content type
  - Response status
  - Exceptions logged with metadata (request ID)

- Generic
  - Version identifier - git commit hash is super useful to have
  - Deployment ID to tie it to the build and deploy pipeline
  - Sampling may be useful, but can be slippery slope - don’t sample error logs

Generic strategies

- Decouple log processing and output selection from your application
  - Always log to stdout

- Logging pipeline
  - Use log forwarding solutions - fluentbit/fluentd/logstash
  - Make any decisions about output or processing in the pipeline - **not** in the application
  - Log forwarding solutions have various features for adding fields/dropping fields/associating metadata and more
  - Fluentbit also has “stream processing” which lets your run SQL queries on your logs

Metrics

What and why?

- What:
  - A number representing your system behavior
  - Service Level Indicator (SLI)
- Why:
  - Alerting
  - Auto-scaling
  - Setting Service Level Objectives (SLO) and Service Level Agreement (SLA)
    - Understanding system behavior and setting expectations
    - Don’t aim for 99.999% unless you need it

Monitoring HTTP/RPC Server applications

- Response status
- Request latency
- Uncaught exceptions
- Health checks

Monitoring Worker applications

- Task processing latency
  - Individual task latency tracking may be tricky
  - Distributed tracing (next topic) may help!
- Task processing errors
- Queue size
- Liveness probes
  - Healthcheck for workers

Things to keep in mind

- Not all monitoring solutions are suitable to WSGI's multiprocess model
  - Prometheus for example
  - Statsd is useful here
- Not all libraries support ASGI/aynscio applications
  - So you will need to do your research before you choose one

Distributed Tracing

Basic idea

A single user operation can involve call to one or more services, a database call and bunch of other stuff

- Let’s refer to each of these things as contexts

Latency metrics give us visibility into each individual context, not the whole picture

Distributed tracing aims to give us visibility into this whole picture

Tracking latency across services

- Propagate an *operational context* across services
  - A trace ID (similar to a logging request ID)

- Bridges the gap between responsibility and control
  - (Excellent write up on the topic: https://lightstep.com/deep-systems/)

- Also useful for asynchronous task latency tracking

Implementing
Metrics and Distributed tracing

Instrumenting your code

You plug in code to:

- Export monitoring data
- Export trace

How?

- Libraries
  - Vendor specific (NewRelic, HoneyComb, DataDog, ..)
  - Vendor neutral (OpenTelemetry)

Storage/Querying

Metrics

- Specialized storage needed for time series data
- Specialized query and calculation engine

Traces - Specialized storage

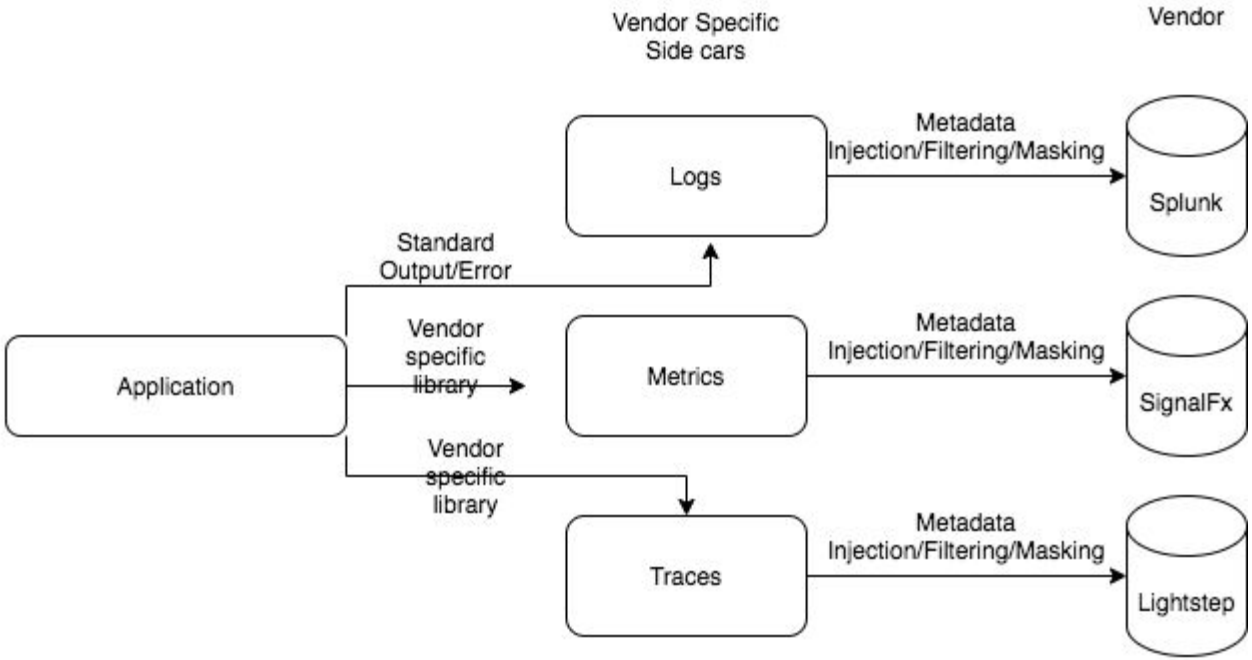
- General purpose storage are useful here: SQL database, ElasticSearch and others
- Specialized query and visualization capabilities needed

OpenTelemetry

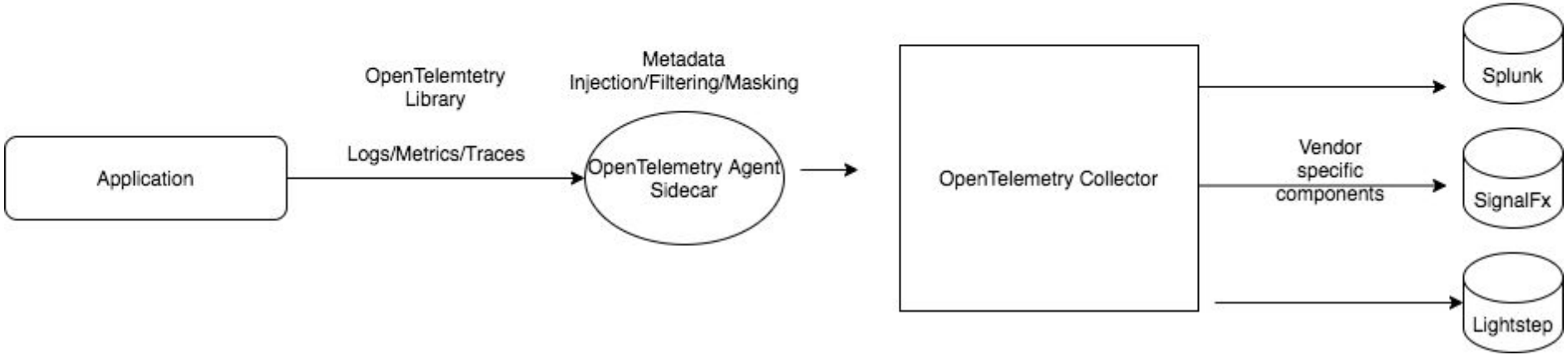
- Vendor neutral specification and SDKs for logging, monitoring and tracing
  - OpenCensus + OpenTracing
  - Logging not yet available for Python

- Automatic instrumentation libraries for tracing
  - SQL clients
  - HTTP clients

Before OpenTelemetry

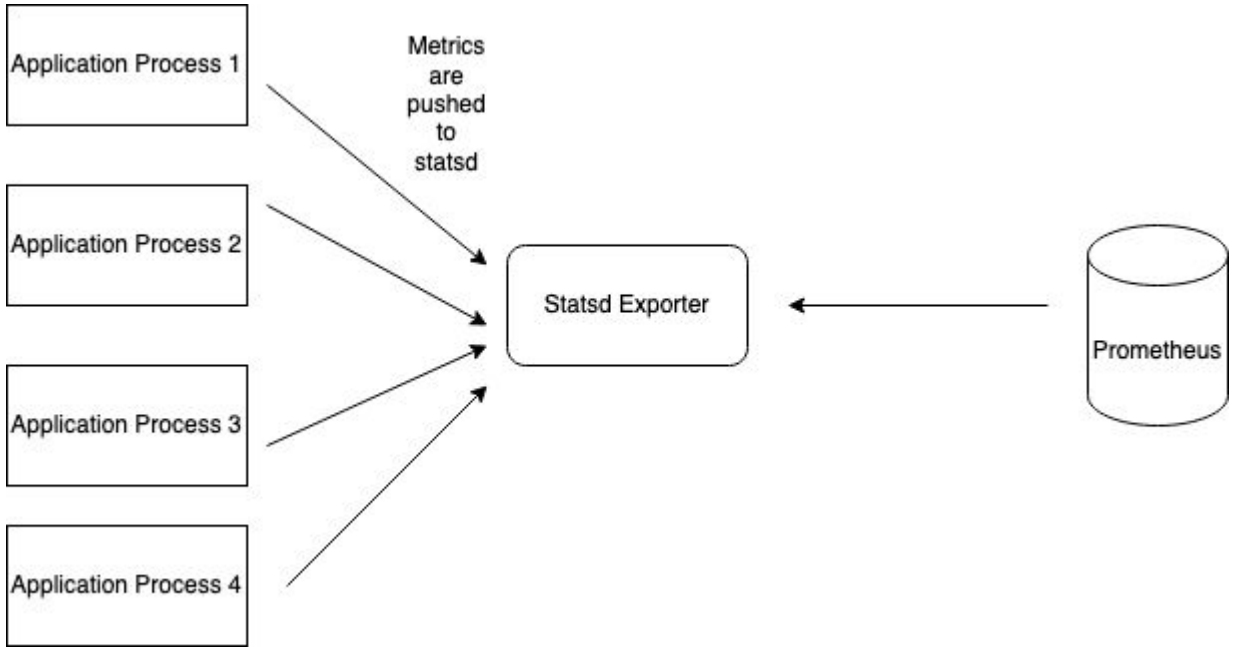


After OpenTelemetry



Instrumenting a WSGI application for metrics

- Statsd Exporter /Prometheus for Metrics
  - We don’t use OpenTelemetry for WSGI applications (see the Git repository for explanation)
  - For single process applications (aiohttp/asyncio), you *can* use OpenTelemetry in principal but scaling may be difficult



Flask app instrumentation for metrics

```
def start_timer():
    request.start_time = time.time()


def stop_timer(response):
    resp_time = time.time() - request.start_time
    statsd.histogram(REQUEST_LATENCY_METRIC_NAME,
                     resp_time,
                     tags=[
                         'service:service1',
                         'endpoint:%s' % request.path,
                     ]
                     )
    return response
```

Flask app instrumentation for metrics

```
def record_request_data(response):
    statsd.increment(REQUEST_COUNT_METRIC_NAME,
                     tags=[
                         'service:service1',
                         'method:%s' % request.method,
                         'endpoint:%s' % request.path,
                         'status:%s' % str(response.status_code)
                     ]
                     )
    return response
```

Flask app instrumentation for metrics

```
def setup_metrics(app):
    app.before_request(start_timer)
    # The order here matters since we want stop_timer
    # to be executed first
    app.after_request(record_request_data)
    app.after_request(stop_timer)
```

Exporting WSGI runtime metrics

- gunicorn/uwsgi pushes statsd metrics
  - Worker metrics, request statistics
  - https://docs.gunicorn.org/en/stable/instrumentation.html
  - Uwsgi: https://uwsgi-docs.readthedocs.io/en/latest/Metrics.html

```
"gunicorn", "--workers", "5", "--bind", "0.0.0.0:8000", "--statsd-host", "statsd:9125", "app:app"
```

Metrics Demo

Initialize OpenTelemetry for Tracing

```
# Initialize the tracing machinery
resource = Resource({"service.name": "service1"})

OTEL_AGENT = os.getenv('OTEL_AGENT', "otel-agent")
otlp_exporter = OTLPSpanExporter(endpoint=OTEL_AGENT + ":4317", insecure=True)

trace.set_tracer_provider(TracerProvider(resource=resource))
tracer = trace.get_tracer(__name__)
span_processor = BatchExportSpanProcessor(otlp_exporter)
trace.get_tracer_provider().add_span_processor(span_processor)
```

Automatic instrumentation for tracing

```
# Service 1
FlaskInstrumentor().instrument_app(app)
RequestsInstrumentor().instrument()

# Service 2
FlaskInstrumentor().instrument_app(app)
MySQLInstrumentor().instrument()
```

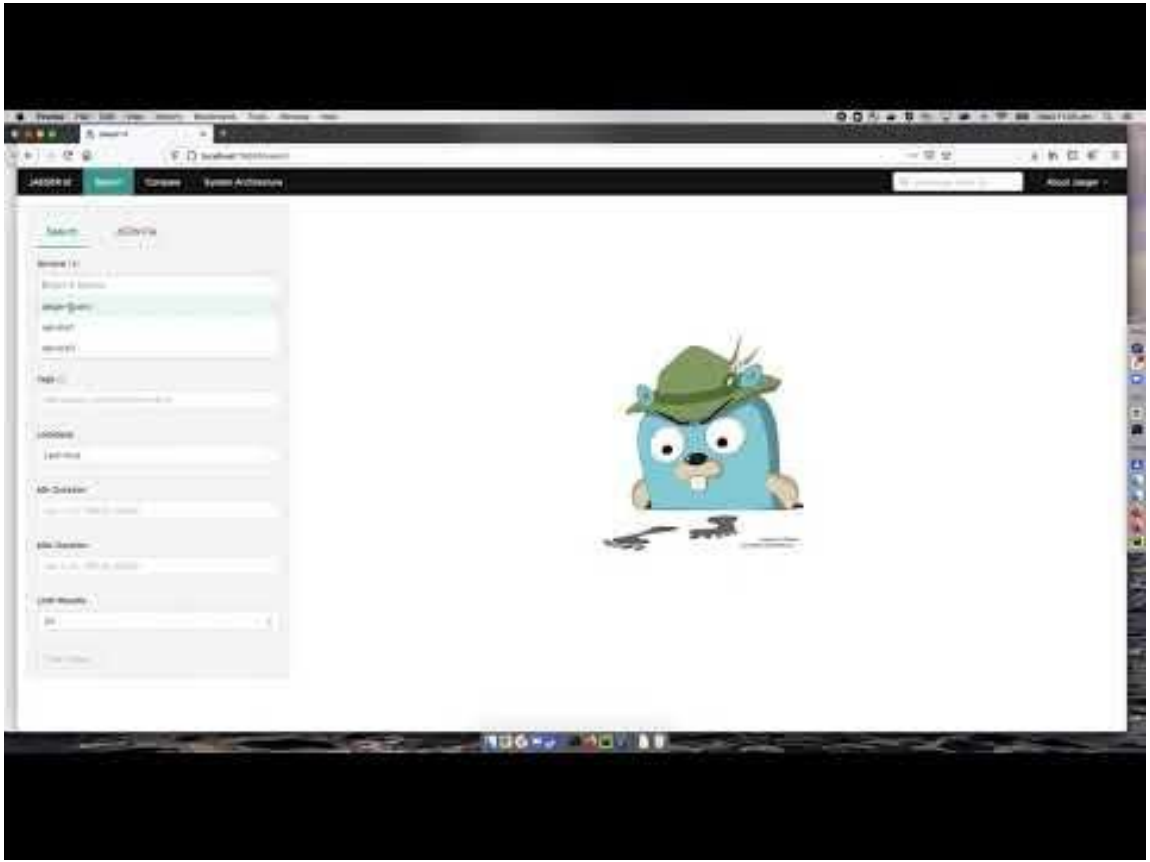
Create a span when making a request to service 2

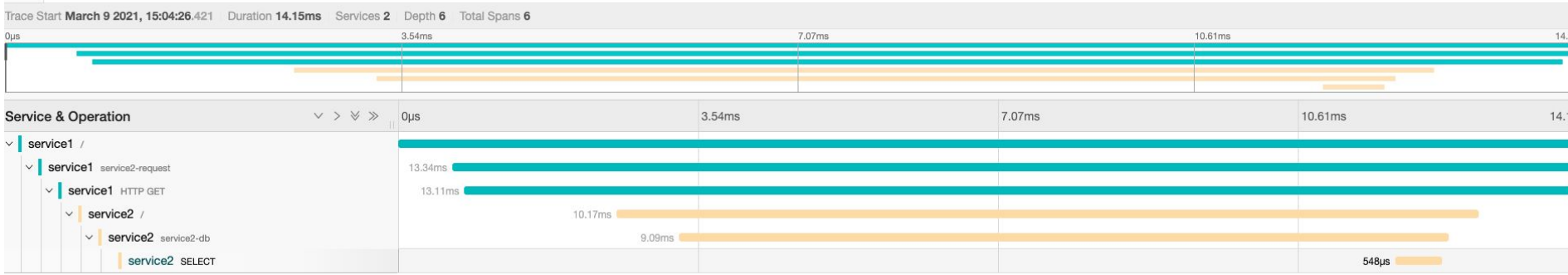
```
@app.route('/')
def index():
    # We create a span here
    with tracer.start_as_current_span("service2-request"):
        data = do_stuff()
    return data.text, 200
```

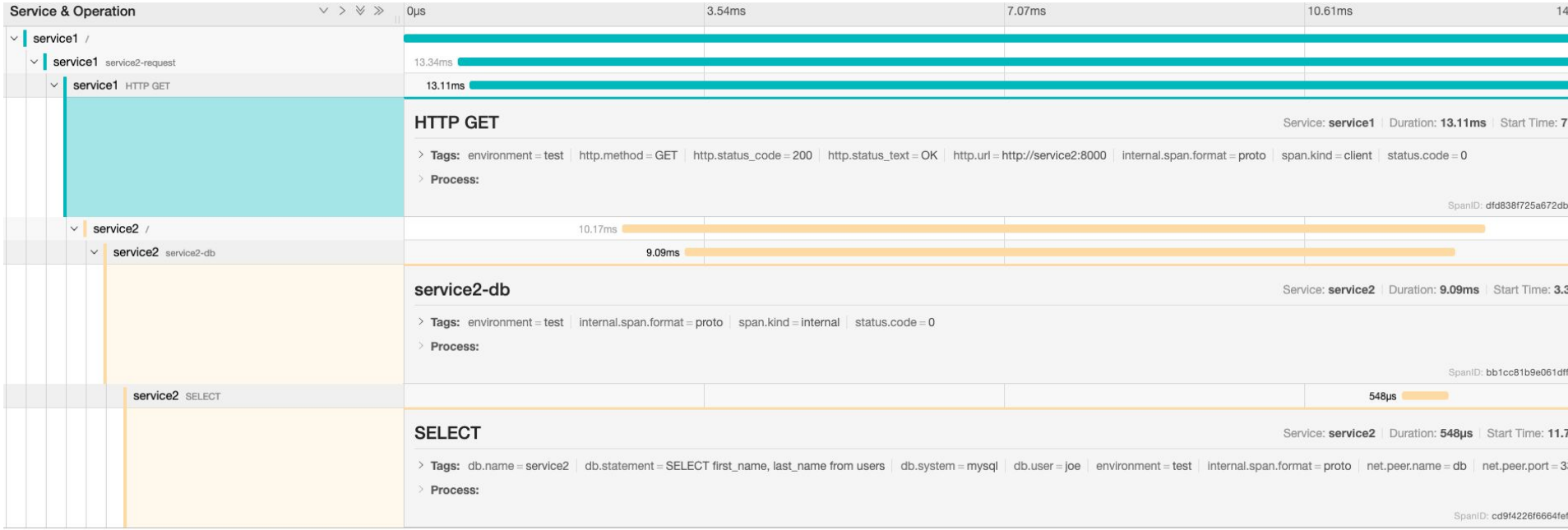
Create a span when making a DB query

```
@app.route('/')
def index():
    with tracer.start_as_current_span("service2-db"):
        # TODO - Move this to app initialization rather than per request
        cnx = mysql.connector.connect(user='joe', password='password',
                                      host='db',
                                      database='service2')
        # TODO Query DB
    return data.format(rows), 200
```

Distributed tracing demo




Trace Start March 9 2021, 15:04:26.421 Duration 14.15ms Services 2 Depth 6 Total Spans 6
0µs
3.54ms
7.07ms
10.61ms
Service & Operation
0µs
3.54ms
7.07ms
10.61ms
service1 /
service1 service2-request
13.34ms
service1 HTTP GET
13.11ms
service2 /
10.17ms
service2 service2-db
9.09ms
service2 SELECT
548µs


Service & Operation
0µs
3.54ms
7.07ms
10.61ms
14
service1 /
service1 service2-request
13.34ms
service1 HTTP GET
13.11ms
HTTP GET
Service: service1 | Duration: 13.11ms | Start Time: 7
Tags: environment = test | http.method = GET | http.status_code = 200 | http.status_text = OK | http.url = http://service2:8000 | internal.span.format = proto | span.kind = client | status.code = 0
Process:
SpanID: dfd838f725a672db
service2 /
10.17ms
service2 service2-db
9.09ms
service2-db
Service: service2 | Duration: 9.09ms | Start Time: 3.3
Tags: environment = test | internal.span.format = proto | span.kind = internal | status.code = 0
Process:
SpanID: bb1cc81b9e061dff
service2 SELECT
548µs
SELECT
Service: service2 | Duration: 548µs | Start Time: 11.7
Tags: db.name = service2 | db.statement = SELECT first_name, last_name from users | db.system = mysql | db.user = joe | environment = test | internal.span.format = proto | net.peer.name = db | net.peer.port = 3
Process:
SpanID: cd9f4226f6664fef

Using SaaS Vendors for Observability

If the demonstrated implementation is not suitable for your production infrastructure, consider using a SaaS vendor:

- Honeycomb.io (Metrics/Distributed Tracing/Logging)
- Datadog (Logging/Metrics/Distributed Tracing)
- Newrelic (Logging/Metrics/Distributed Tracing)
- Lightstep (Distributed Tracing)

For metrics, these solutions will usually run an agent along with your application and give you a fair bit of data out of the box

Using SaaS Vendors for Observability

For distributed tracing, your code will need instrumentation (to propagate the Trace Ids)

Go with the vendor which is compatible with OpenTelemetry - since you don’t get vendor lock-in

Takeaways

Don’t ship your code without instrumentation

- Logs
- Metrics
- Traces

You (will) need them!

RESILIENCY
(Application/Infrastructure)

Setting up automatic horizontal scaling

- Use metrics to auto-scale your web application instances or worker process instances
  - Cloud providers, Container orchestrators (Kubernetes) has support for this

- Which metrics to scale one?
  - Anything that corresponds to your load
    - You may need to downsize/upsize your current resource allocations to understand the variation of CPU/Memory with traffic
    - Look at patterns of the various metrics - application as well as system metrics
  - Web/RPC applications - CPU usage/Number of requests
  - Worker processes - Number of messages in queue

Simulate failure scenarios (Chaos Engineering)

- Inject latency into calls to dependencies
  - Tests your time out behavior, error reporting and alerting
- Inject a random instance termination
  - Tests your auto scaling behavior
- Get creative!
  - Terminate your worker processes - (For WSGI runtimes)
  - Fill up disk space and observe what happens

Other important things

Rate limiting

- Protect your services from intentional and unintentional malicious behavior

Circuit breaking

- Reduce the blast radius

Service meshes may help here as well

Summary

- Your service will fail badly at some point of time for some reason in some capacity

- Focus on observability of your service from the get go - structured logs, metrics and traces

- Run regular chaos engineering exercises as an exploration of your service's production readiness

- Implement resiliency patterns

Questions/Comments

Thank you for attending my presentation

If you have any questions or comments, please get in touch via @echorand or email me amitsaha.in@gmail.com

Demos/Resources: https://github.com/amitsaha/python-web-conf-2021