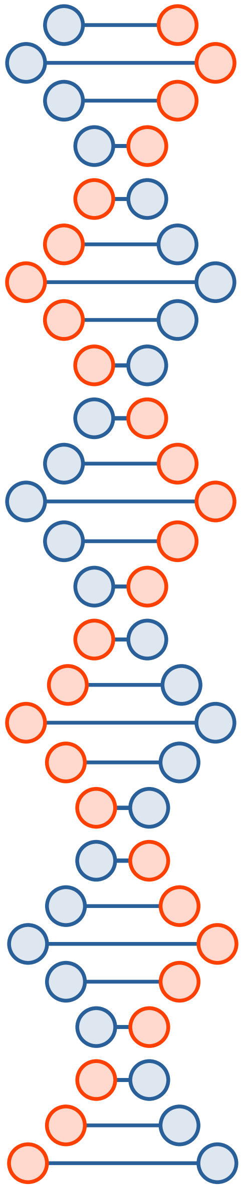


OpenTelemetry

iLGLabs

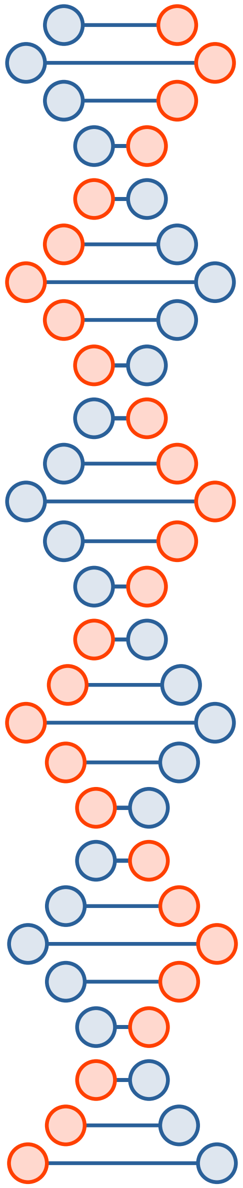
GnuGroup

Reinventing the way,

To Think,

Learn,

Work

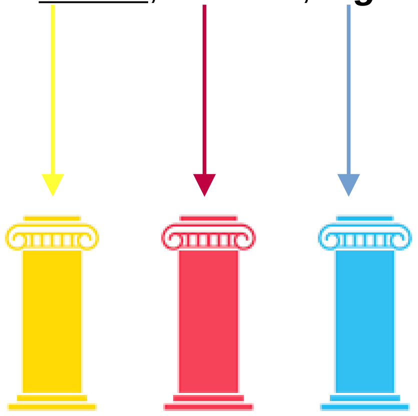


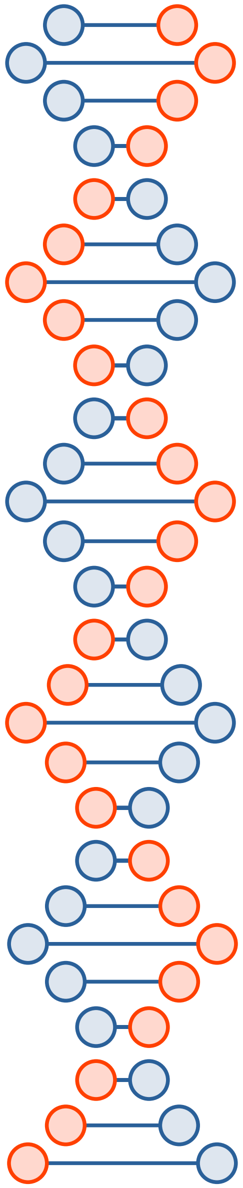
Observability

What is observability?

Observability is the ability to understand the internal state of a system by examining its outputs.

Context of software - Being able to understand the internal state of a system by examining its telemetry data, which includes <u>***traces***</u>, ***metrics***, ***logs***.





Making System Observable ?

System must be instrumented.

Code must emit traces, metrics, or logs.

Instrumented data must then be sent to an observability backend.

Metrics

A metric consists of a **single numeric value tracked over time.**

Traditional monitoring uses system-level metrics to track things like CPU, memory, and disk performance.

Why it’s important ?

This data is important for choosing among virtual machine instance types, with options for processor speed, RAM, and hard disk storage.

What it does not tell you ?

It doesn’t tell you about user experience, or how to improve the performance of your code.

Logs

Logs are text strings written to the terminal or to a file (often referred to as a “flat” log file).

Logs can be any arbitrary string, but programming languages and frameworks have libraries to generate logs from your running code.

Logs generate relevant data at different levels of specificity (e.g. INFO vs. DEBUG mode).

Pitfalls

Querying flat logs (as opposed to structured logs) is slow because of the computational complexity of indexing and parsing strings, which makes these tools impractical for debugging and investigating your production code in near-realtime.

Traces, Spans, Distributed traces

Trace is a *<u>visualization of the events in your system showing the calling relationship between parent and child events as well as timing data for each event.</u>*

Individual events that form a trace are called spans.

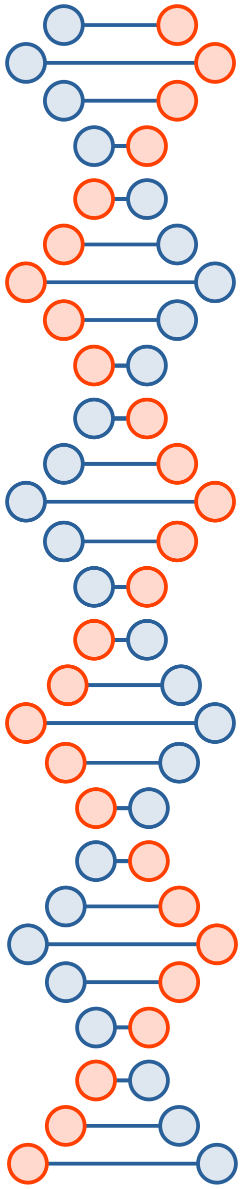
Each span stores the **start time, duration, and parent_id.**

Spans without a parent_id are rendered as root spans.

A *distributed trace connects calling relationships among events in distributed services.*

E.g

Service A calls Service B, which makes a database query and then hits a third-party API.



Structured Events

A structured event is a data format that allows you to **store key-value pairs** for an arbitrary number of fields or dimensions, often in JSON format.

At a minimum, *structured events usually have a timestamp and a name field.*

Instrumentation libraries can automatically add other relevant data like the request endpoint, the user-agent, or the database query.

Why OpenTelemetry ?

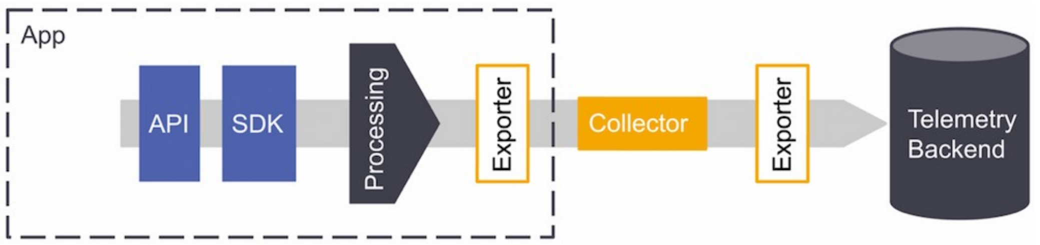
With the rise of cloud computing, micro services architectures, and increasingly complex business requirements, the need for software and infrastructure observability is greater than ever.

OpenTelemetry satisfies the need for observability while following two key principles:

- You own the data that you generate. There’s no vendor lock-in.
- You only have to learn a single set of APIs and conventions.

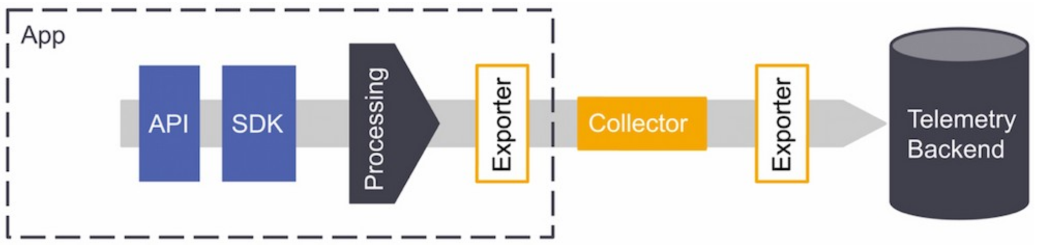
Both principles combined grant teams and organizations the flexibility they need in today’s modern computing world.

OpenTelemetry Architecture



OpenTelemetry provides a library framework that **receives**, **processes**, and **exports** telemetry, which requires a back end to receive and store the data.

OpenTelemetry Architecture



APIs and SDKs

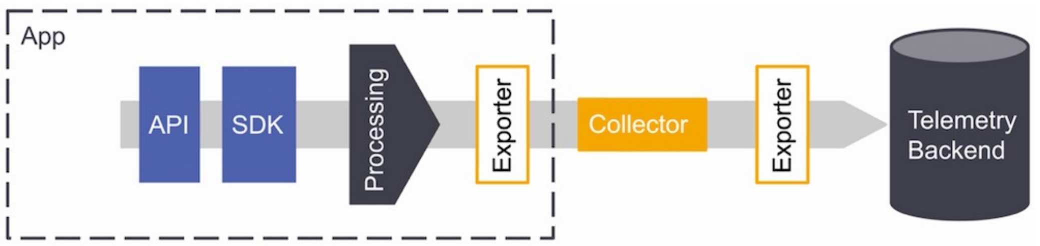
OpenTelemetry APIs define how applications speak to one another and are used to instrument an application or service.

As generally available for developers to use across popular programming languages (e.g., Ruby, Java, Python).

As they are part of the OpenTelemetry standard, they will work with any OpenTelemetry-compatible back-end system, *eliminating the need to re-instrument in the future*.

The SDK is also language specific, providing the bridge between APIs and the exporter. It can sample traces and aggregate metrics.

OpenTelemetry Architecture

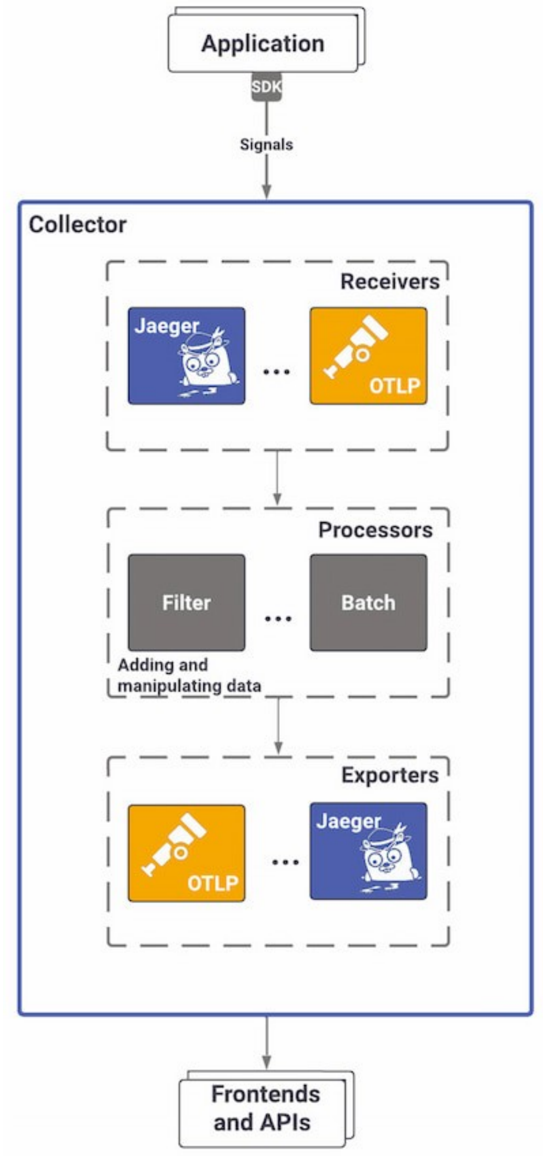


Collector

OpenTelemetry Collector is like a bakery: Regardless of how the raw ingredients are processed, you can still shape your bread in whatever way you fancy.

This means you don't need to alter your code to send data into whatever back end you use for storage and visualization.

Inside the OTel Collector pipeline



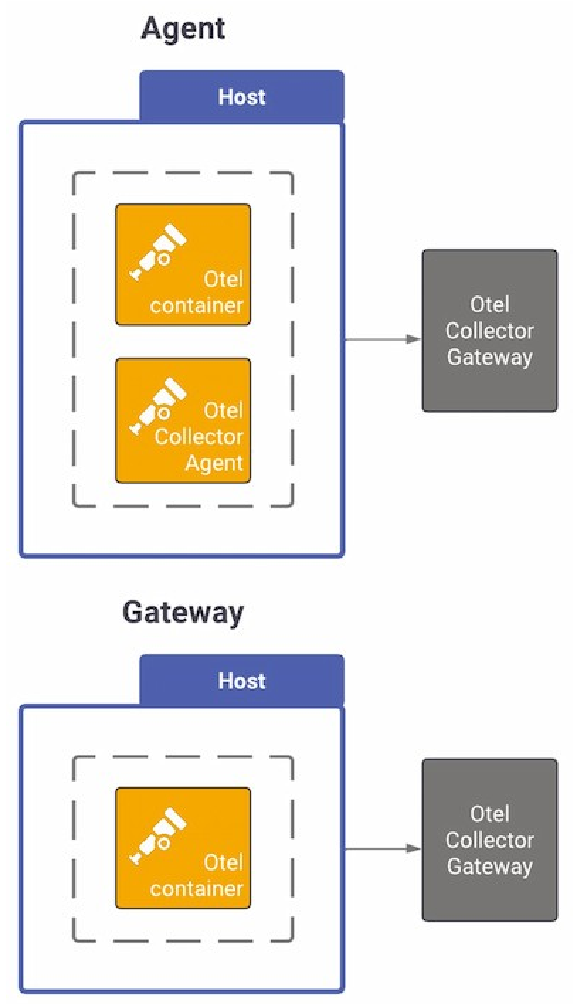
Collectors Job

Collector's job is to process, filter, aggregate, and batch telemetry, giving developers greater flexibility for receiving, shaping, and sending data to multiple back ends.

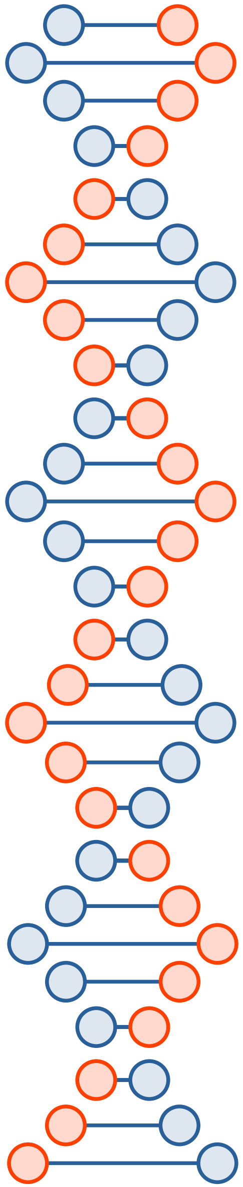
- It works with two primary deployment models:

➔ **Agent** that lives within the application or in the same host as the application, acting as a source of data for the host (by default, OpenTelemetry assumes a local collector is available)

➔ **Gateway** working as a data pipeline that receives, exports, and processes telemetry

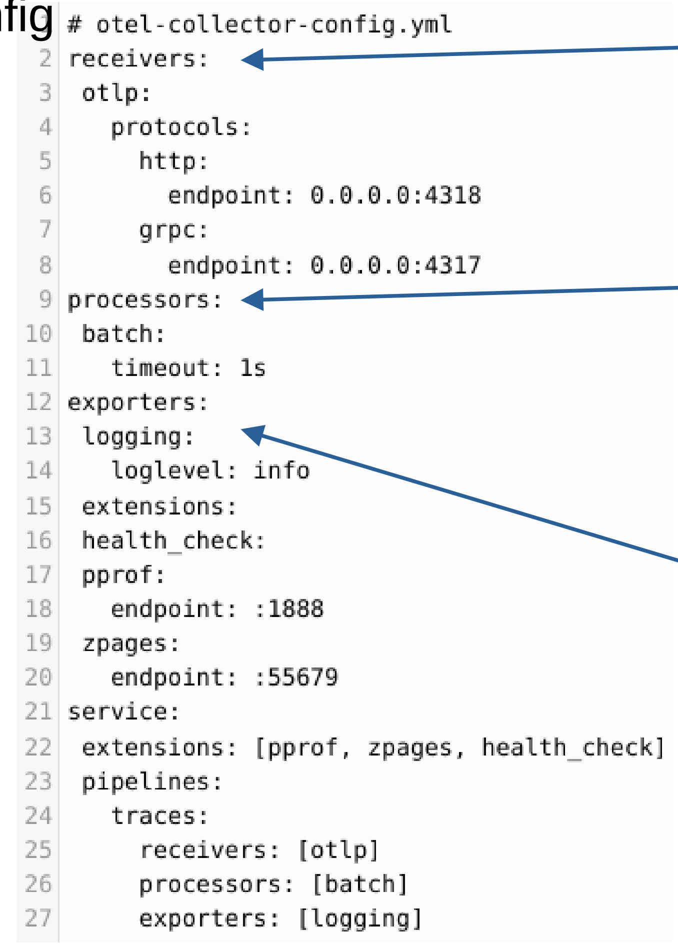


Collector Agent and Gateway setup



3 Components

Yaml config



```yaml
# otel-collector-config.yml
receivers:
 otlp:
   protocols:
     http:
       endpoint: 0.0.0.0:4318
     grpc:
       endpoint: 0.0.0.0:4317
processors:
 batch:
   timeout: 1s
exporters:
 logging:
   loglevel: info
 extensions:
 health_check:
 pprof:
   endpoint: :1888
 zpages:
   endpoint: :55679
service:
 extensions: [pprof, zpages, health_check]
 pipelines:
   traces:
     receivers: [otlp]
     processors: [batch]
     exporters: [logging]
```

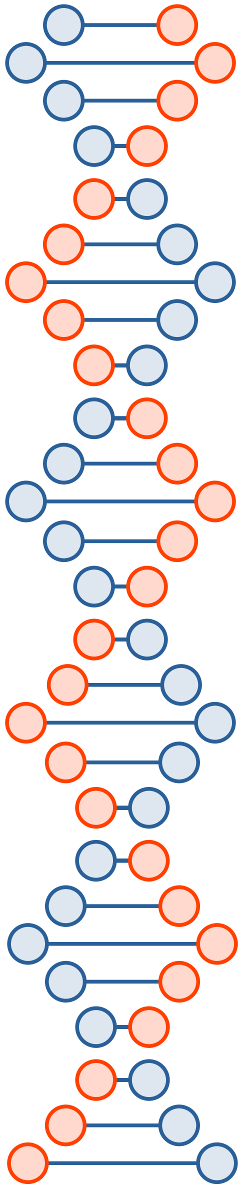
Receivers (e.g., Jaeger, Prometheus) are in charge of pushing or pulling the applications' signals by listening for calls on particular ports on the Collector. They work with both gRPC and HTTP protocols.

Processors sit between receivers and exporters; they enable us to shape the data by filtering, formatting, and enriching it before it goes through the exporter to a back end.

Common use cases include data sanitization to remove sensitive or private information, exporting metrics from spans, or deciding which signals are saved to the back end.

Exporters can *push or pull data into one or multiple configured back ends or destinations* (e.g., Kafka, OTLP).

They work by transforming the data into a different format if needed and sending it to the endpoint defined. An exporter creates a layer of separation between instrumentation and the back-end configuration so users can switch back ends without re-instrumenting the code.



We set the OTLP receiver to add HTTP and gRPC endpoints in the collector.

To process our data, we use a batch processor that will compress and segment it;

it's configurable for batching by time and size.

Using the batch processor is highly recommended, as it reduces the number of outgoing connections.

We also define two exporters: logging that will print into the console and Jaeger, where we'll send the traces.

The service section is where we set up how all the previous elements come together in the pipeline.

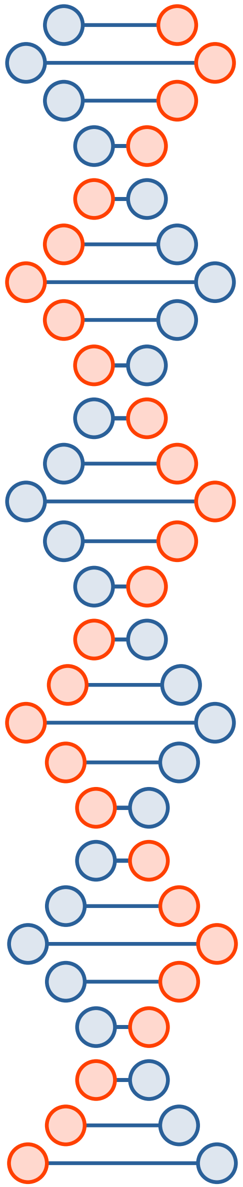
OpenTelemetry Protocol

OpenTelemetry Protocol (OTLP) is one of the reasons for OpenTelemetry's success.

It's an agnostic protocol specification that defines the encoding for data and the transport protocol for sending traces, metrics, and logs.

It can send data from the SDK to the Collector and from the Collector to the chosen back end.

Using the Collector elements, we can abstract from third-party frameworks by configuring the proper receiver.



OpenTelemetry Key Concepts

All observability journeys must begin with ***instrumenting an application*** to emit signals from services as they execute.

OpenTelemetry gives you several components that'll help you add proper instrumentation to services and have each operation execution result in one or multiple spans, metrics, or logs.

Instrumentation

There are mainly two ways to instrument applications using OpenTelemetry:

a) Manual

b) Auto-instrumentation.

These become available by adding the OpenTelemetry SDK to your project.

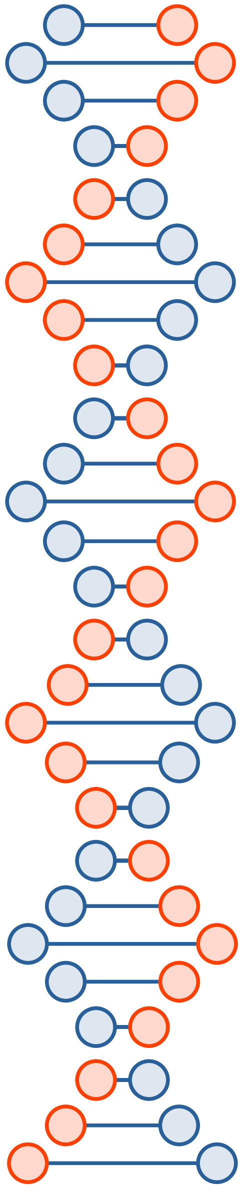
Auto-instrumentation makes it possible to collect application-level telemetry without manual changes to the code — it allows tracing a transaction's path as it navigates different components, including:

- Application frameworks
- Communication protocols
- Data stores

Manual

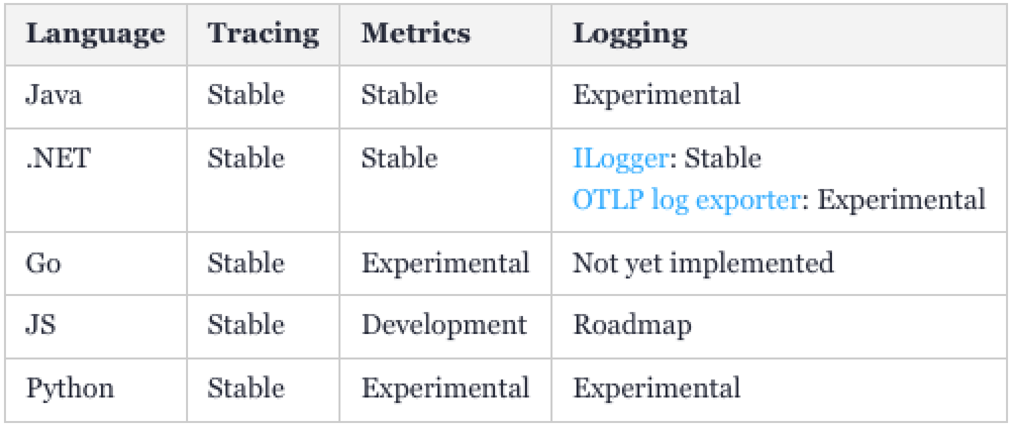
Manual instrumentation ***lets you decide how and where to add observability code*** to your project. Four instrumentation libraries are available:

- **Core** contains all language instrumentation libraries available.
- **Instrumentation** adds to the Core library by adding extra language-specific capabilities.
- **Contrib** includes additional helpful libraries and standalone utilities that don't fit the scope of the previous two.
- **Distribution** adds vendor-specific customization.



Languages and Support Status

OpenTelemetry is a collection of tools, APIs, and SDKs available in multiple languages.



Language	Tracing	Metrics	Logging
Java	Stable	Stable	Experimental
.NET	Stable	Stable	ILogger: Stable OTLP log exporter: Experimental
Go	Stable	Experimental	Not yet implemented
JS	Stable	Development	Roadmap
Python	Stable	Experimental	Experimental

Telemetry Sources

Telemetry data, or signals, are any output collected from the system, and when analyzed together, this output provides a view of the relationships and dependencies of the distributed system.

Currently, OpenTelemetry supports three categories of telemetry:

a) logs,

b) traces

c) metrics.

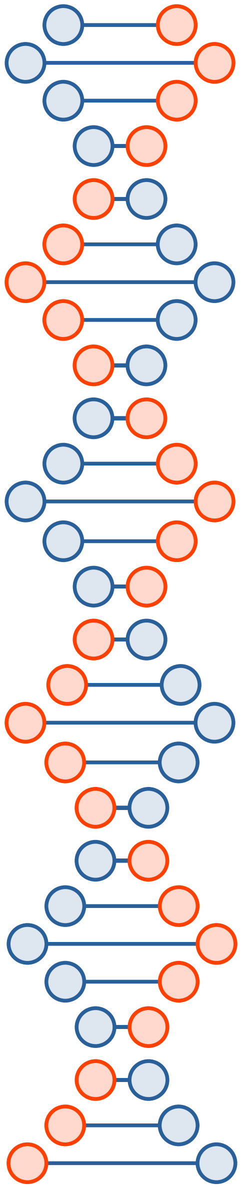
like ***profiling*** or user data. ? Not right now.

Metrics

A metric is a ***numerical representation of a value calculated*** or aggregated for a service captured at runtime

e.g.

- size of a message broker,
- number of errors per second,
- process memory utilization,
- error rate



Metrics Event

The moment one of these measurements is captured is known as a metric event — it comprises of

1) Measurement

2) Time of capture and

3) Associated metadata.

Application and request metrics are essential indicators of availability and performance.

OpenTelemetry Metrics API processes the raw measurements, summarizing them to give developers visibility into their services' operational metrics.

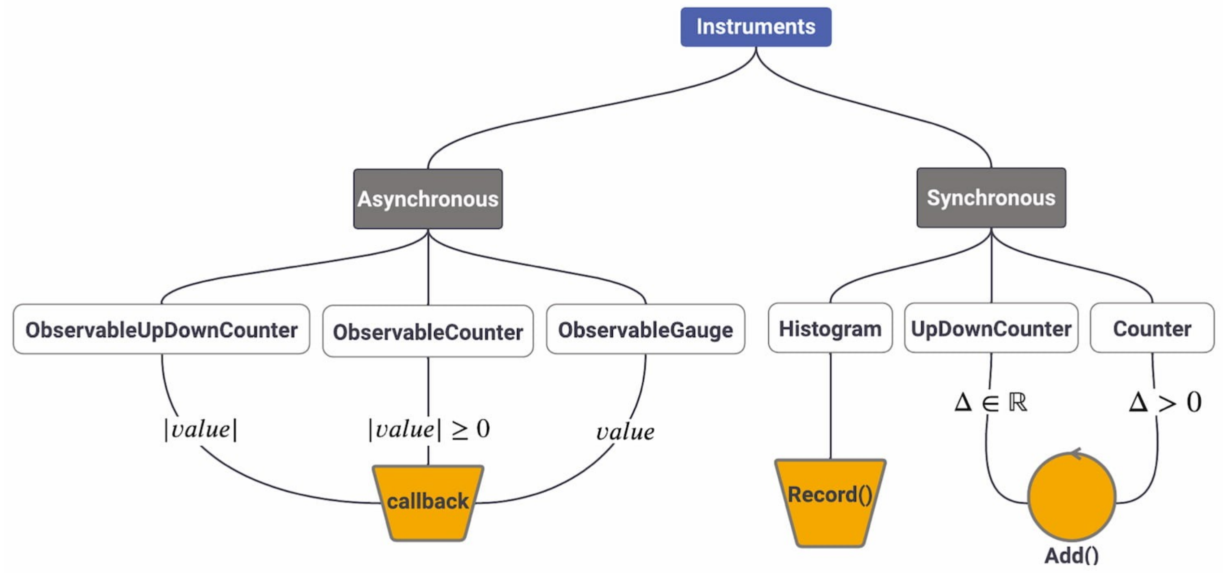
Metric API's

In the Metrics API, you have **six available instruments** that are associated with a specific meter at creation time.

These can be ***synchronous*** or ***asynchronous***;

1) Synchronous instruments are invoked inline with the application code execution,

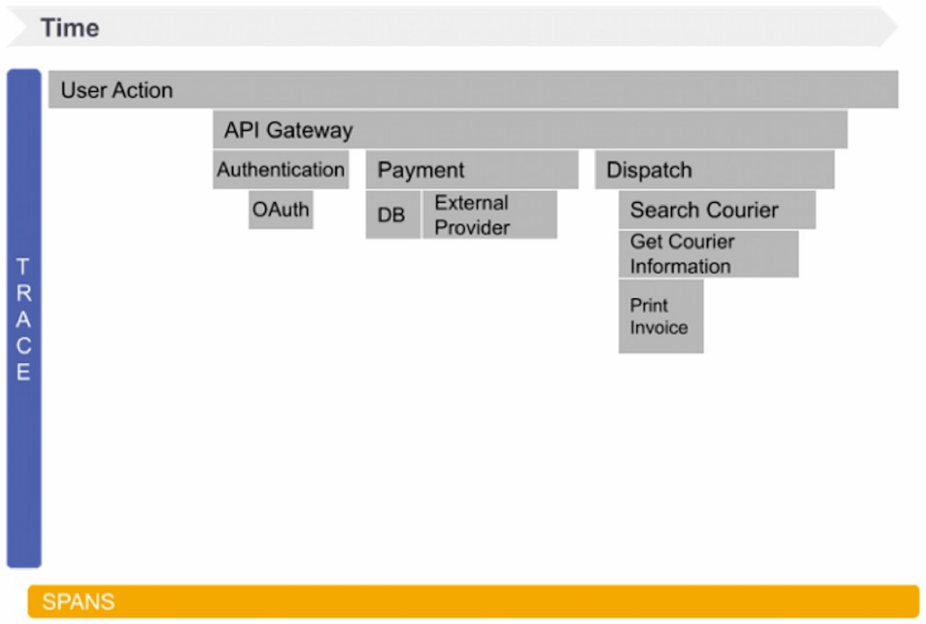
2) Asynchronous instruments allow the user to register a callback function responsible for reporting the measurements.



Traces

A trace represents the ***flow of a single transaction*** or request as it goes through the system.

They provide a ***holistic view of the chain of events triggered by requests*** and are defined by a tree of nested *spans — one for each unit of work they represent and a parent span*



Tracers

- **TracerProvider** is a factory for Tracers; it's initialized once and lives for the duration of the application's lifecycle. It's the first step in tracing with OpenTelemetry.
- **Tracer** creates spans containing supplementary information about what is happening for a given operation. It is created from Tracer Providers, and in some SDKs, a global Tracer already exists (Python, .NET).
- **Trace Exporter** sends traces to a back end; it can be standard output like the OpenTelemetry Collector or any open-source or vendor back end of your choice.
- **Trace Context**

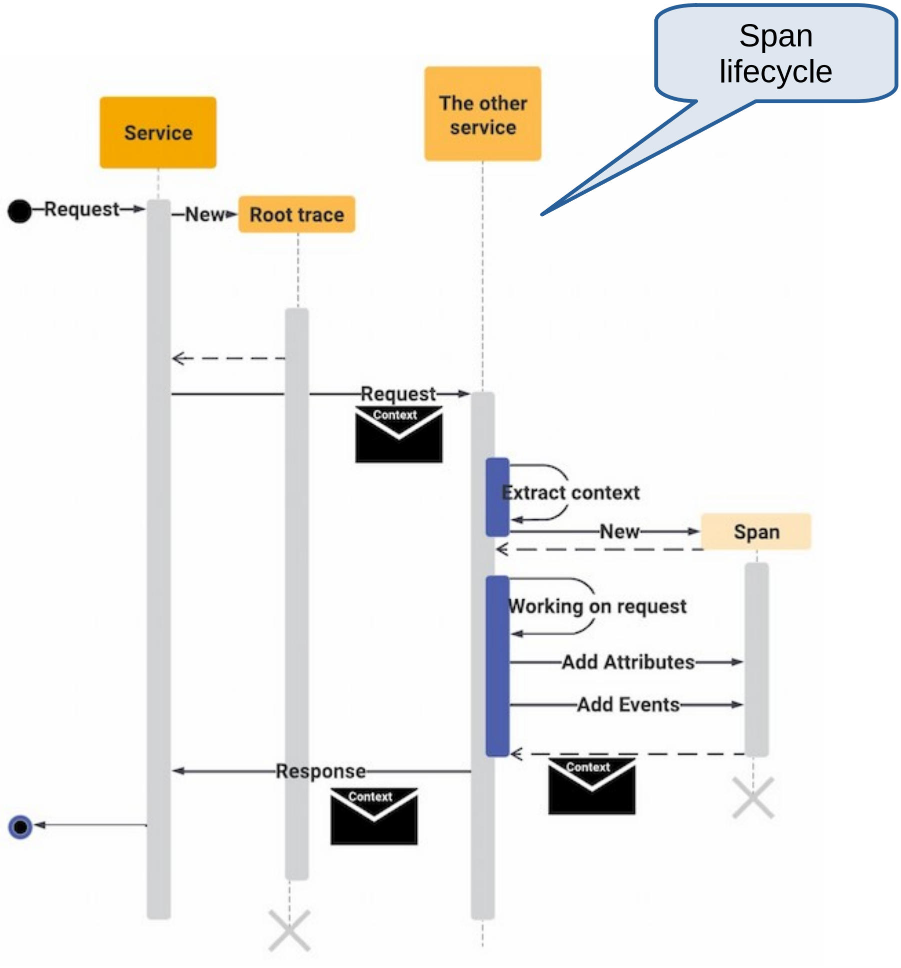
```json
{
 "data": [
   {
     "traceID": "81289be65e00618d84366dfe2f7fc1a2",
     "spans": [
       {
         "traceID": "81289be65e00618d84366dfe2f7fc1a2",
         "spanID": "e03e8cca690f81c1",
         "operationName": "read_json_from_file",
       }
       // ...
       {
         "traceID": "81289be65e00618d84366dfe2f7fc1a2",
         "spanID": "75473187e1bc7579",
         "operationName": "word-by-language"
         // ...
       },
       {
         "traceID": "81289be65e00618d84366dfe2f7fc1a2",
         "spanID": "45c0d587ebdddf60",
         "operationName": "/words"
         // ...
       }
     ],
     "processes": {
       "p1": {
         "serviceName": "untranslatable-python",
         "tags": []
       }
     },
     "warnings": null
   }
 ],
 "total": 0,
 "limit": 0,
 "offset": 0,
 "errors": null
}
```

An example of a trace with three spans

Spans

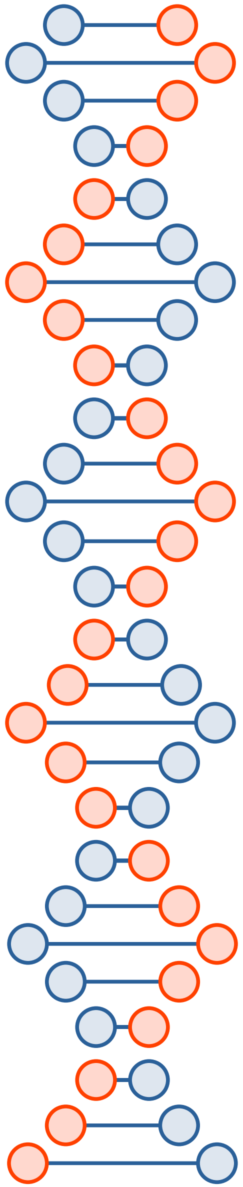
In OpenTelemetry, a span includes the following information:

- **Name**
- **Start and End Timestamps**
- Span Context is an object that can't be changed after creation, containing:
  - Its own ID
  - Trace ID – a unique 16-byte array that identifies the trace that the span is part of, and all spans contained in that trace share this ID.
  - Trace Flags – present in all traces and, through binary encoded data, provide more details on the trace.
  - Trace State – a key-value list carrying vendor-specific trace information, so multiple tracing systems can participate in a trace.
- Span Attributes are key-value pairs added to a span to help analyze the trace data. The extra information will help you better understand and search for specific traces.
- Span Events are typically used to mark a singular point in time during the span's duration, similar to adding an annotation on a span.
- Span Links associate one span with one or more, implying some relationship; they are optional but an excellent way of associating trace spans.
- Span Status



```json
{
  "traceID": "81289be65e00618d84366dfe2f7fc1a2",
  "spanID": "e03e8cca690f81c1",
  "operationName": "read_json_from_file",
  "references": [
    {
      "refType": "CHILD_OF",
      "traceID": "81289be65e00618d84366dfe2f7fc1a2",
      "spanID": "75473187e1bc7579"
    }
  ],
  "startTime": 1659199980429164,
  "duration": 143,
  "tags": [
    {
      "key": "otel.library.name",
      "type": "string",
      "value": "data.file_reader"
    },
    {
      "key": "span.kind",
      "type": "string",
      "value": "internal"
    },
    {
      "key": "internal.span.format",
      "type": "string",
      "value": "proto"
    }
  ],
  "logs": [
    {
      "timestamp": 1659199980429173,
      "fields": [
        {
          "key": "event",
          "type": "string",
          "value": "Opening data file."
        }
      ]
    },
    {
      "timestamp": 1659199980429301,
      "fields": [
        {
          "key": "event",
          "type": "string",
          "value": "Finished reading data file."
        }
      ]
    }
  ],
  "processID": "p1"
}
```

span with two events



Who creates spans ?

In OpenTelemetry, the Tracer creates the spans.

It's an object that tracks the currently active span while allowing you to create new spans.

As spans start and complete, the Tracer dispatches them to the back end you configured on the Collector.

Logs

A log is recorded as lines of text that describe a timestamped event and can be output in plain text, structured text (like JSON), or binary code.

They result from a code execution block and are convenient for troubleshooting systems.

It’s less prone to instrumentation (e.g., databases, load balancers).

OpenTelemetry assumes that any data that doesn't belong to a trace or metric must be a log.

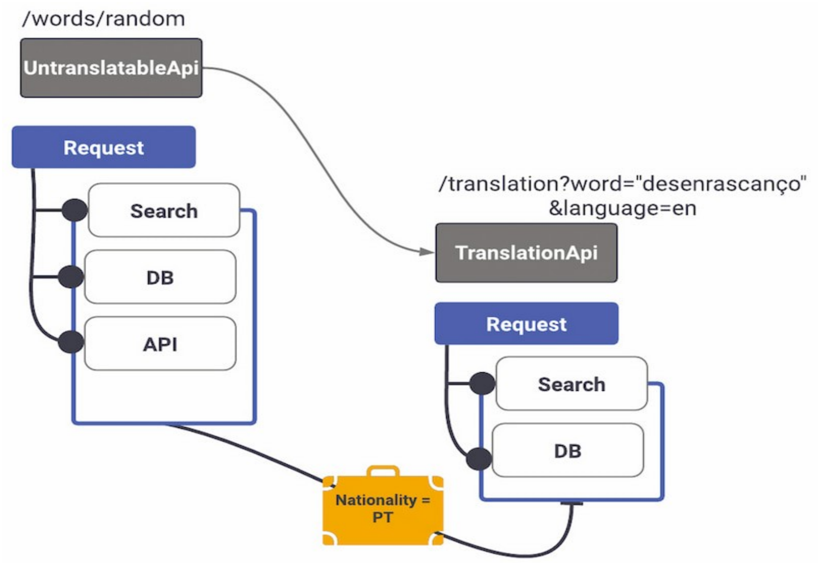
Baggage

Baggage refers to **contextual information passed on between spans**.

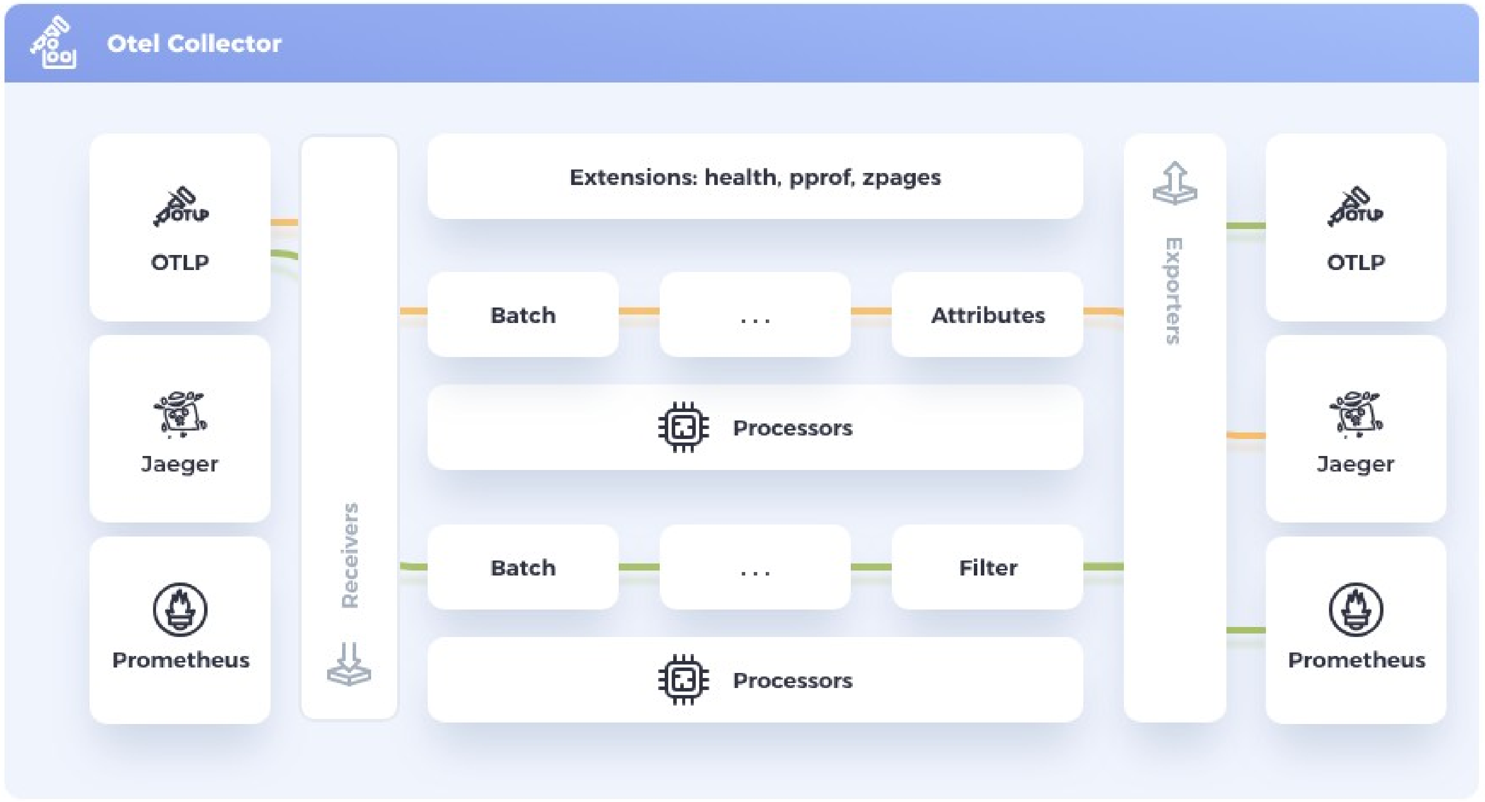
It's represented in OpenTelemetry by a **key-value** store that *lives in a Trace Context*, making those values available to all spans created within that trace.

OpenTelemetry uses Context Propagation to pass around Baggage and ***exists in all libraries***, so you don't have to implement it yourself.

Baggage is designed to be language agnostic so that it can travel through stacks.



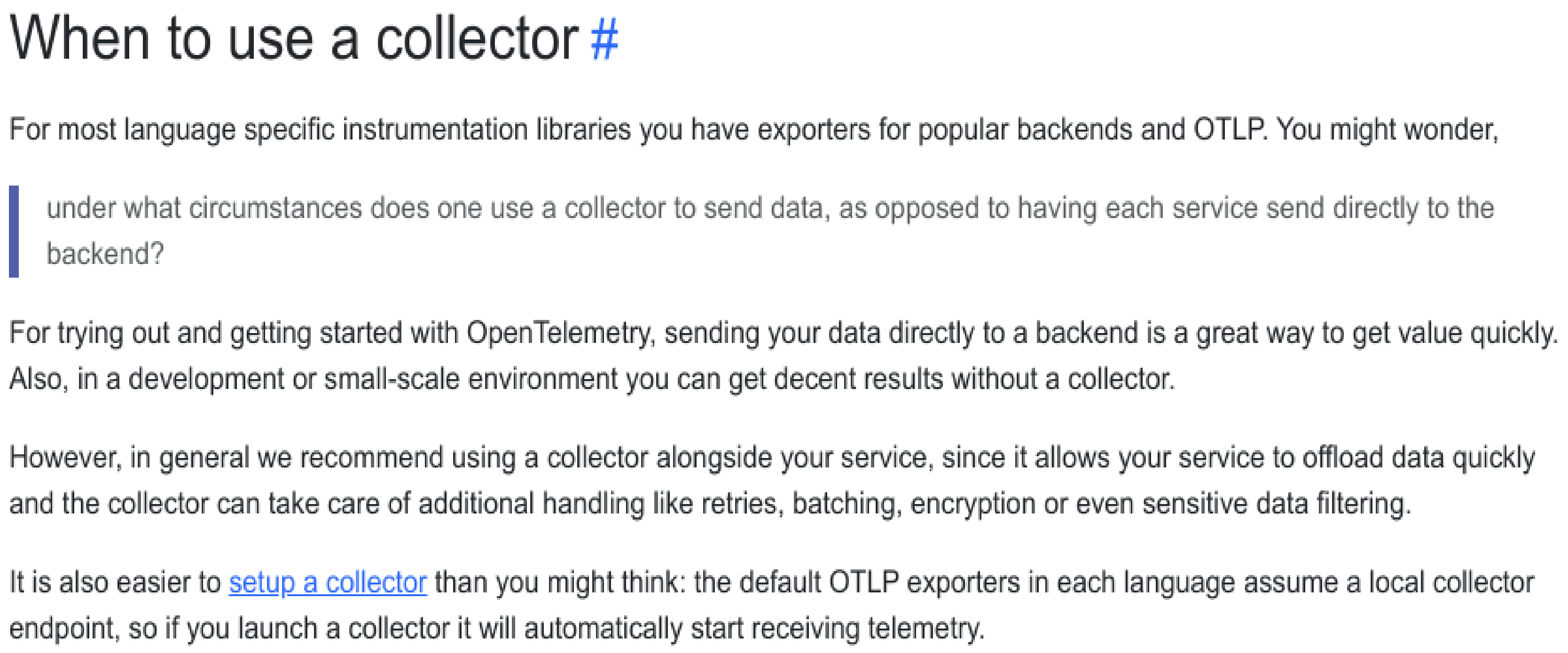
Getting Started With OpenTelemetry



OTEL COLLECTOR

Objectives

- *Usability*: Reasonable default configuration, supports popular protocols, runs and collects out of the box.
- *Performance*: Highly stable and performant under varying loads and configurations.
- *Observability*: An exemplar of an observable service.
- *Extensibility*: Customizable without touching the core code.
- *Unification*: Single codebase, deployable as an agent or collector with support for traces, metrics, and logs.



When to use a collector

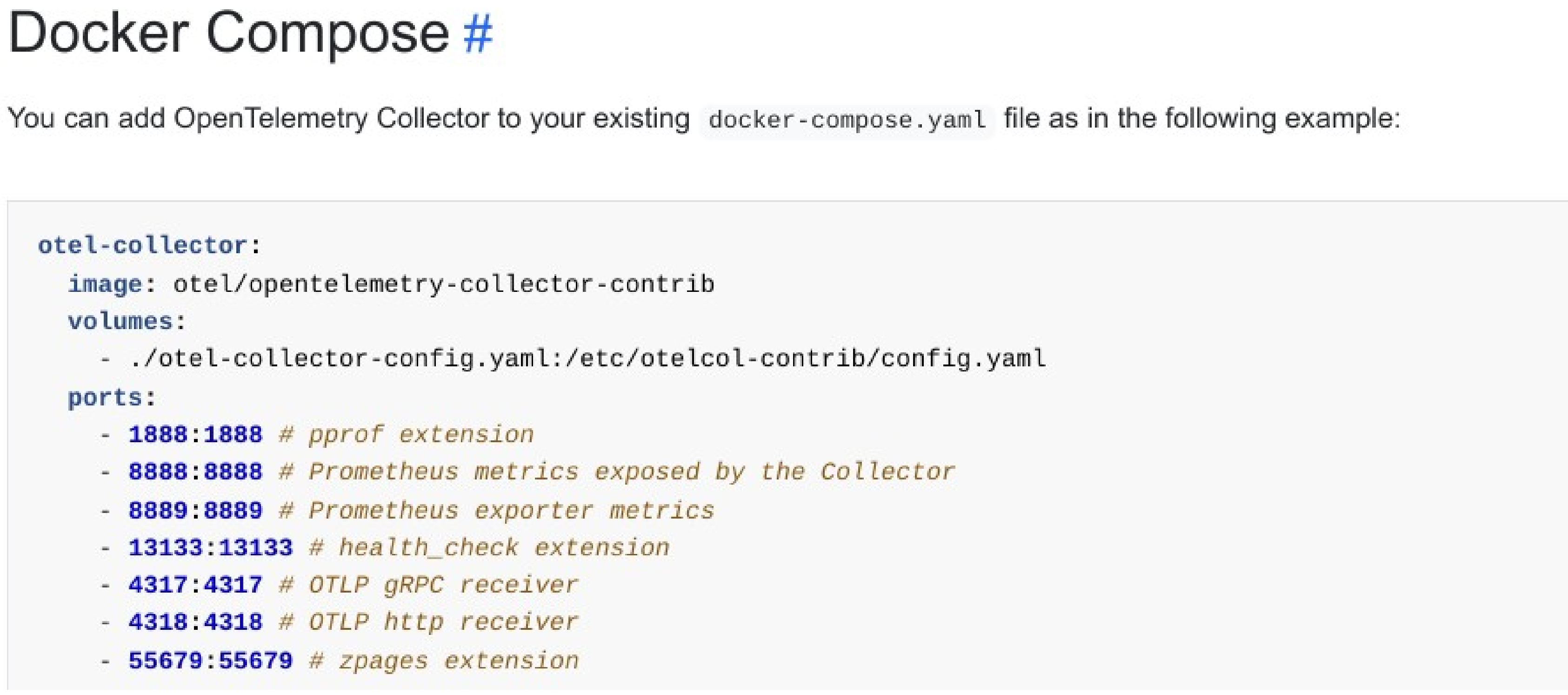
For most language specific instrumentation libraries you have exporters for popular backends and OTLP. You might wonder,

> under what circumstances does one use a collector to send data, as opposed to having each service send directly to the backend?

For trying out and getting started with OpenTelemetry, sending your data directly to a backend is a great way to get value quickly. Also, in a development or small-scale environment you can get decent results without a collector.

However, in general we recommend using a collector alongside your service, since it allows your service to offload data quickly and the collector can take care of additional handling like retries, batching, encryption or even sensitive data filtering.

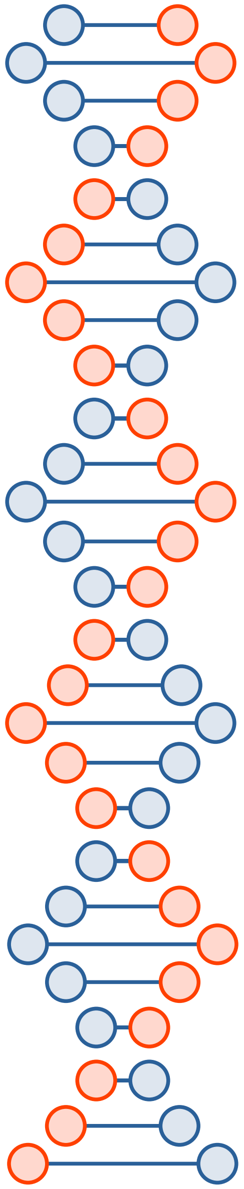
It is also easier to setup a collector than you might think: the default OTLP exporters in each language assume a local collector endpoint, so if you launch a collector it will automatically start receiving telemetry.

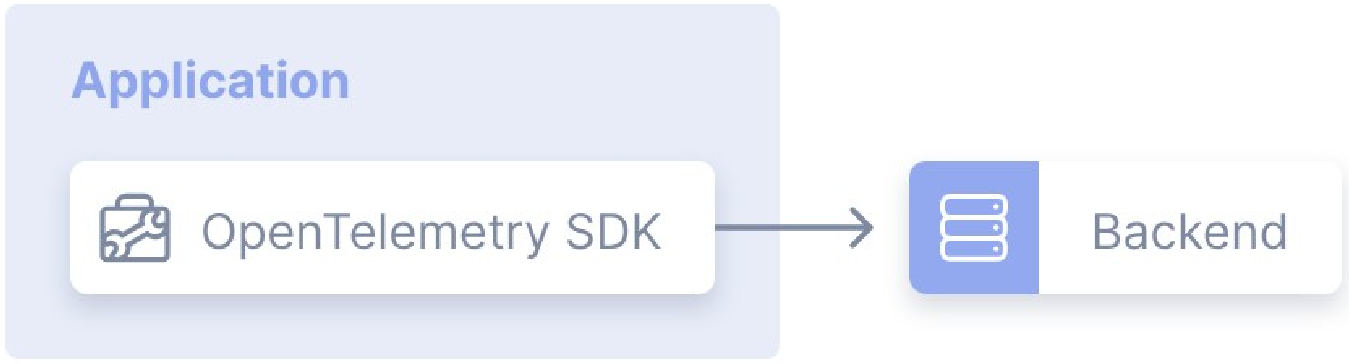


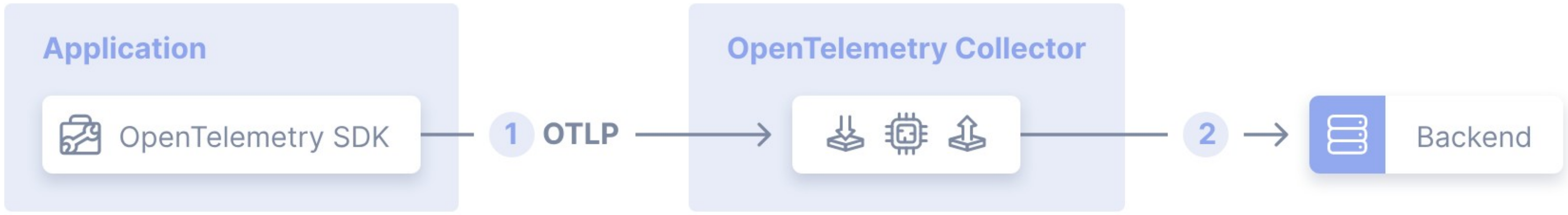
Docker Compose

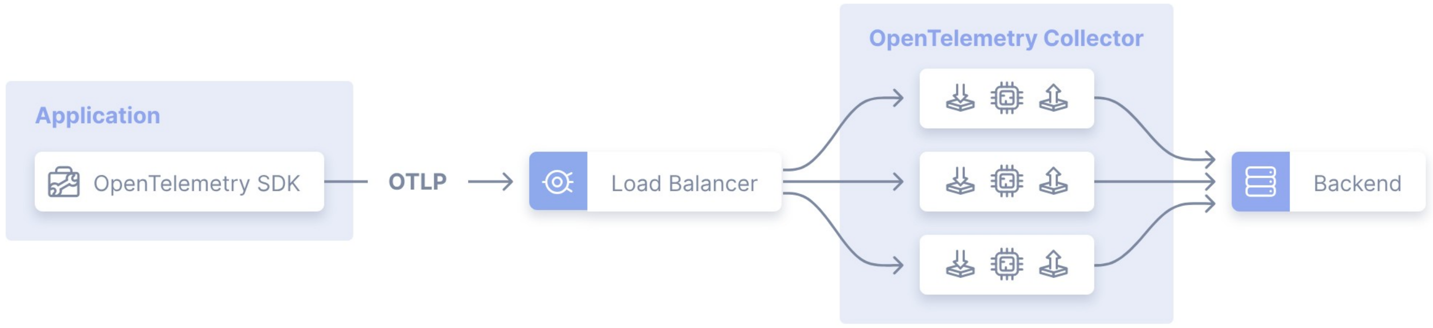
You can add OpenTelemetry Collector to your existing `docker-compose.yaml` file as in the following example:

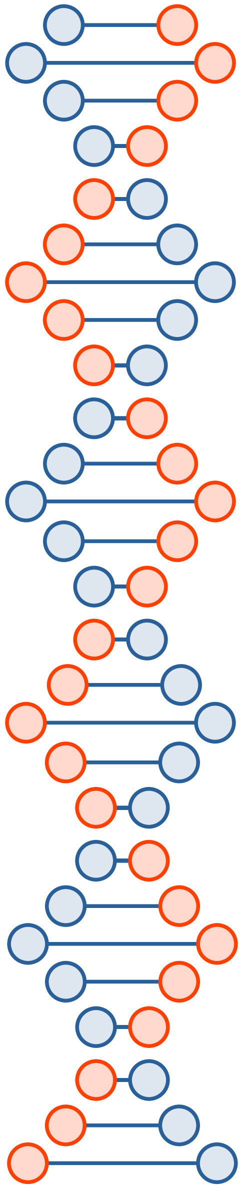
```
otel-collector:
  image: otel/opentelemetry-collector-contrib
  volumes:
    - ./otel-collector-config.yaml:/etc/otelcol-contrib/config.yaml
  ports:
    - 1888:1888 # pprof extension
    - 8888:8888 # Prometheus metrics exposed by the Collector
    - 8889:8889 # Prometheus exporter metrics
    - 13133:13133 # health_check extension
    - 4317:4317 # OTLP gRPC receiver
    - 4318:4318 # OTLP http receiver
    - 55679:55679 # zpages extension
```

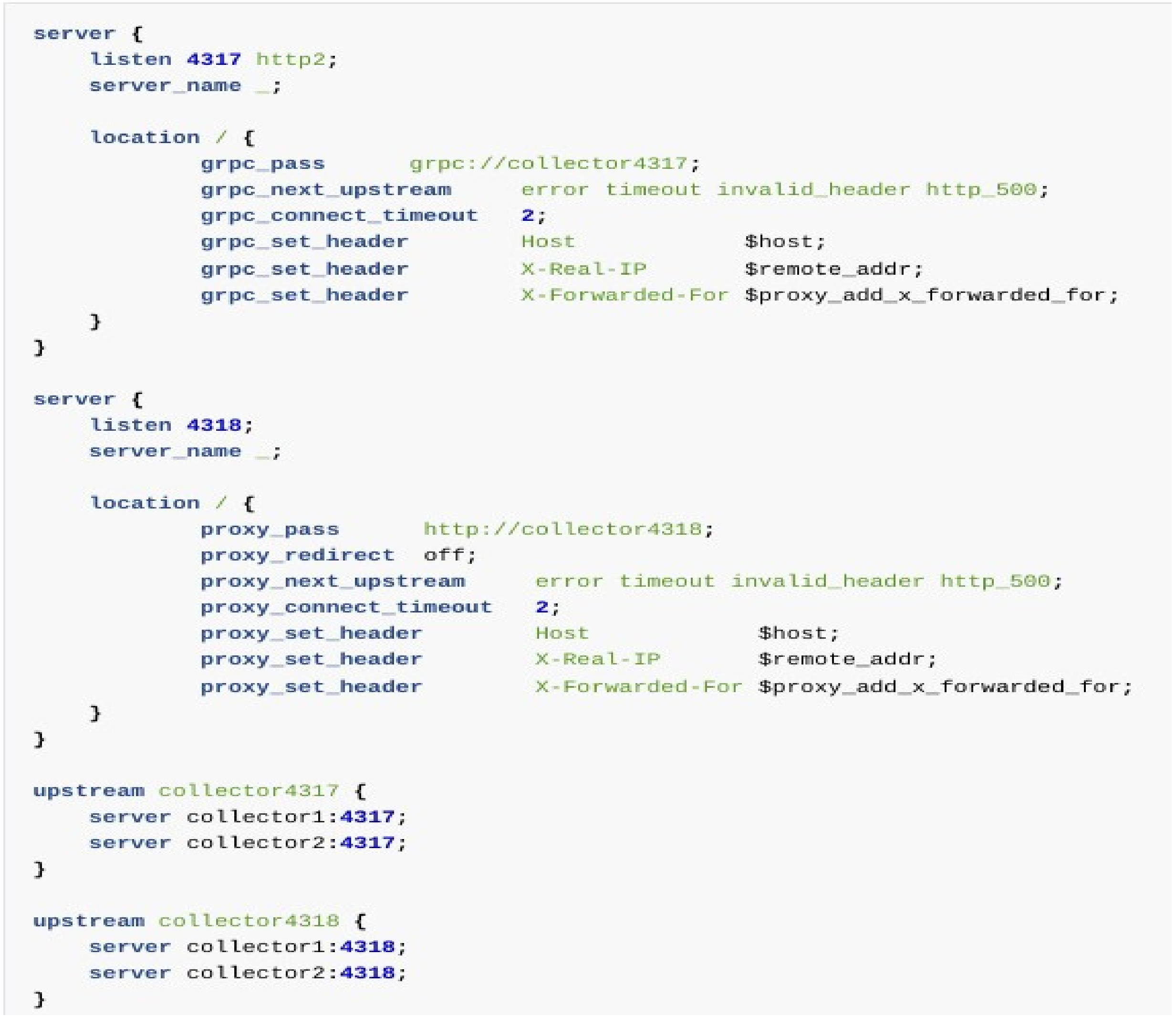



Application
OpenTelemetry SDK
Backend

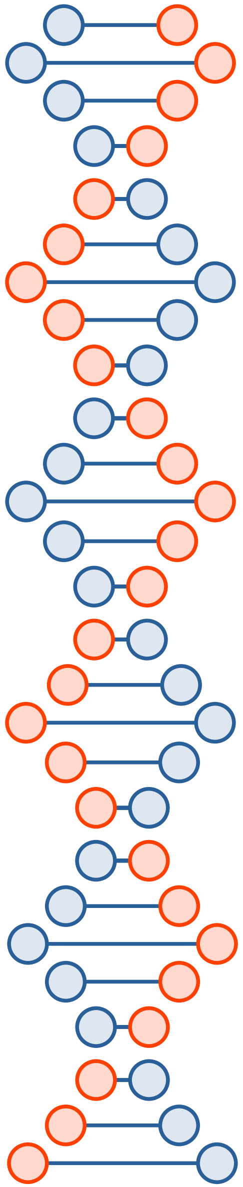

Application
OpenTelemetry SDK
1
OTLP
OpenTelemetry Collector
2
Backend


Application
OpenTelemetry SDK
OTLP
Load Balancer
OpenTelemetry Collector
Backend



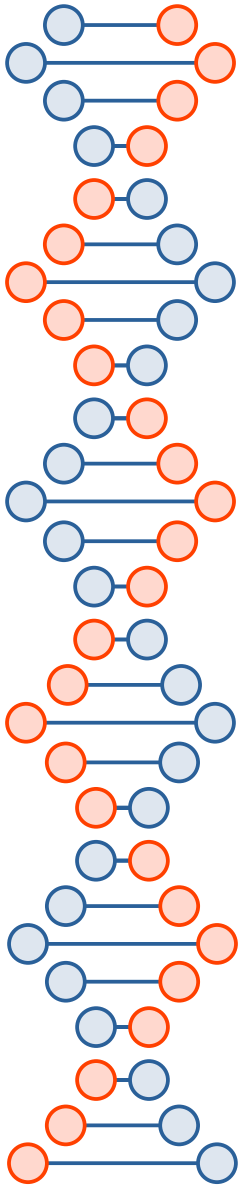


```
server {
    listen 4317 http2;
    server_name _;

    location / {
            grpc_pass       grpc://collector4317;
            grpc_next_upstream      error timeout invalid_header http_500;
            grpc_connect_timeout    2;
            grpc_set_header         Host            $host;
            grpc_set_header         X-Real-IP       $remote_addr;
            grpc_set_header         X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}

server {
    listen 4318;
    server_name _;

    location / {
            proxy_pass       http://collector4318;
            proxy_redirect   off;
            proxy_next_upstream      error timeout invalid_header http_500;
            proxy_connect_timeout    2;
            proxy_set_header         Host            $host;
            proxy_set_header         X-Real-IP       $remote_addr;
            proxy_set_header         X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}

upstream collector4317 {
    server collector1:4317;
    server collector2:4317;
}

upstream collector4318 {
    server collector1:4318;
    server collector2:4318;
}
```



otelcol --config=customconfig.yaml

```
otelcol --config=env:MY_CONFIG_IN_AN_ENVVAR --config=https://server/config.yaml
otelcol --config="yaml:exporters::debug::verbosity: normal"
```



- Receivers
- Processors
- Exporters
- Connectors

```
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4317
      http:
        endpoint: 0.0.0.0:4318
processors:
  batch:

exporters:
  otlp:
    endpoint: otelcol:4317

extensions:
  health_check:
  pprof:
  zpages:
```

```
service:
  extensions: [health_check, pprof, zpages]
  pipelines:
    traces:
      receivers: [otlp]
      processors: [batch]
      exporters: [otlp]
    metrics:
      receivers: [otlp]
      processors: [batch]
      exporters: [otlp]
    logs:
      receivers: [otlp]
      processors: [batch]
      exporters: [otlp]
```

1. Receivers Section

- `` `otlp` ``: The OTLP receiver is configured to receive telemetry data over gRPC and HTTP. The endpoints `` `0.0.0.0:4317` `` and `` `0.0.0.0:4318` `` are standard ports for OTLP over gRPC and HTTP, respectively.

2. Processors Section

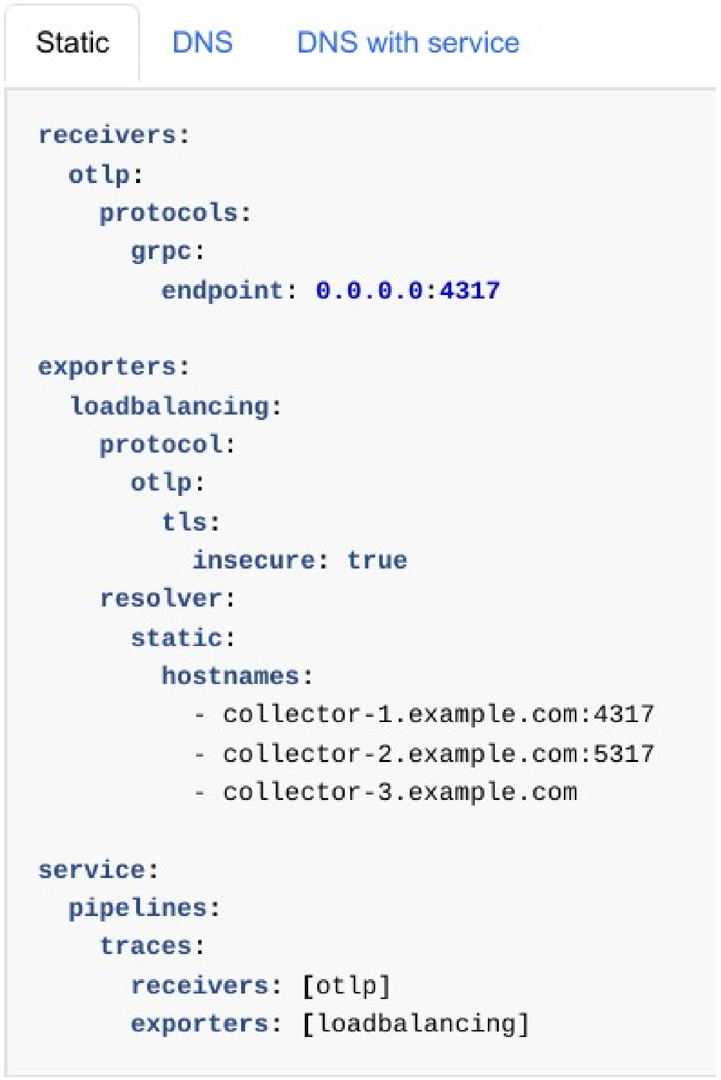
- `` `batch` ``: The batch processor is configured to batch telemetry data before sending it to exporters. This helps in improving efficiency and reducing the number of outgoing requests:
  - `` `timeout: 5s` ``: The processor waits for up to 5 seconds to collect a batch before sending it.
  - `` `send_batch_size: 1024` ``: This is the size of the batch that will be sent.
  - `` `send_batch_max_size: 8192` ``: This is the maximum size of the batch that can be sent. If the batch exceeds this size, it will be split.

3. Exporters Section

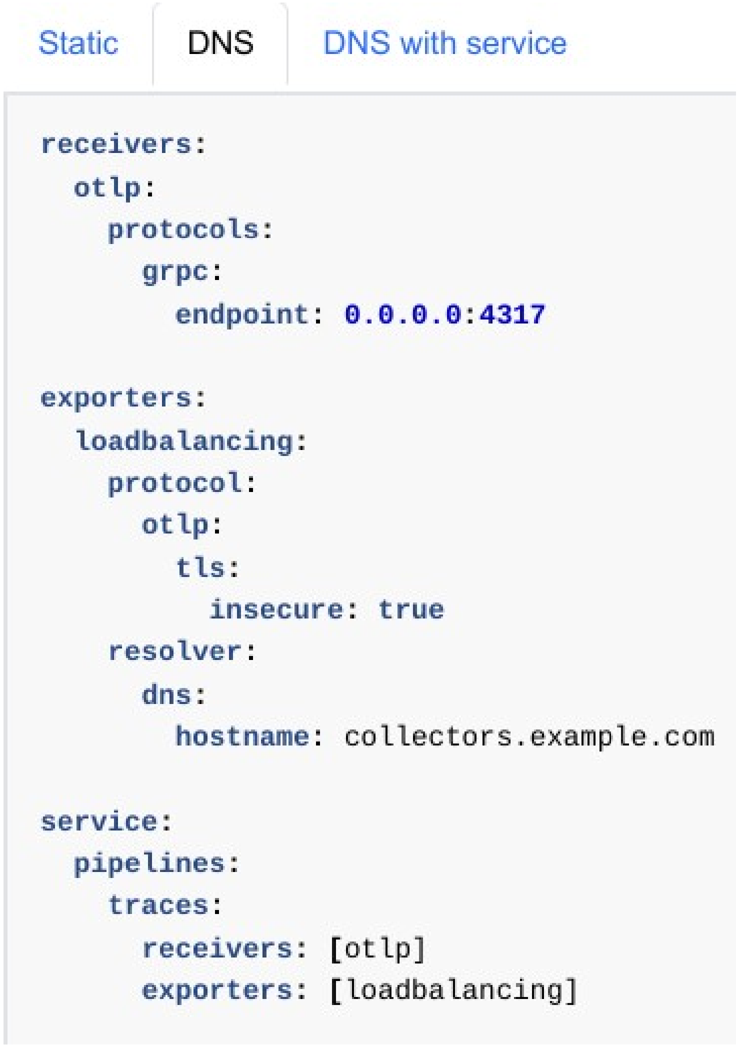
- `logging`: A logging exporter is configured to output telemetry data to the logs, useful for debugging.
- `otlp`: The OTLP exporter is configured to send data to an OTLP endpoint. The `tls` configuration with `insecure: true` disables TLS verification, useful for local testing.

4. Service Section

- **Pipelines**: Three pipelines are defined for traces, metrics, and logs:
  - `traces`: Receives traces via the OTLP receiver, processes them in batches, and exports to logging and OTLP exporters.
  - `metrics`: Similar setup for metrics.
  - `logs`: Similar setup for logs.

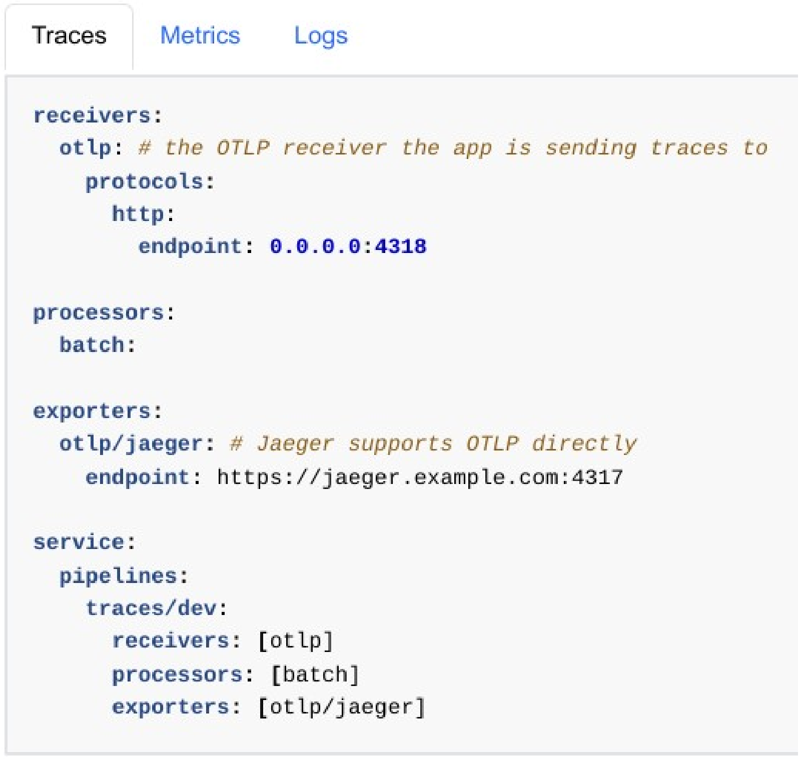


```
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4317

exporters:
  loadbalancing:
    protocol:
      otlp:
        tls:
          insecure: true
    resolver:
      static:
        hostnames:
          - collector-1.example.com:4317
          - collector-2.example.com:5317
          - collector-3.example.com

service:
  pipelines:
    traces:
      receivers: [otlp]
      exporters: [loadbalancing]
```

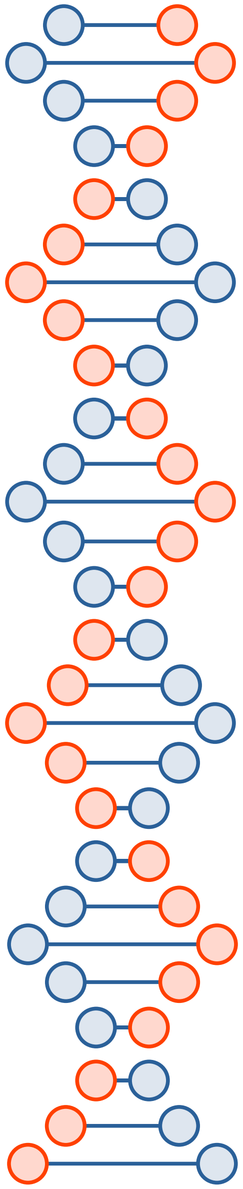


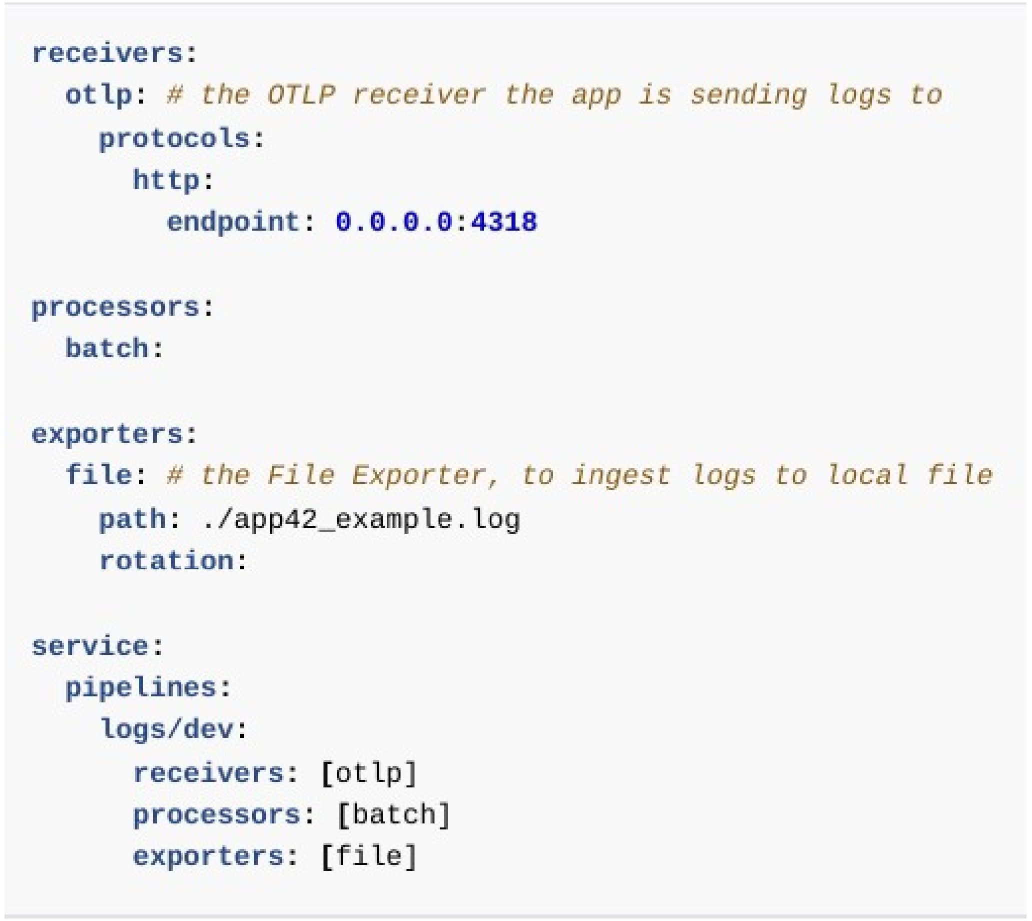
```
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4317

exporters:
  loadbalancing:
    protocol:
      otlp:
        tls:
          insecure: true
    resolver:
      dns:
        hostname: collectors.example.com

service:
  pipelines:
    traces:
      receivers: [otlp]
      exporters: [loadbalancing]
```

export OTEL_EXPORTER_OTLP_ENDPOINT=http://collector.example.com:4318



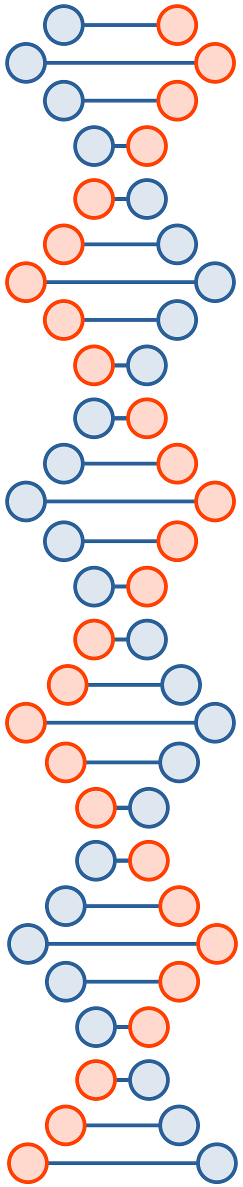
```yaml
receivers:
  otlp: # the OTLP receiver the app is sending traces to
    protocols:
      http:
        endpoint: 0.0.0.0:4318

processors:
  batch:

exporters:
  otlp/jaeger: # Jaeger supports OTLP directly
    endpoint: https://jaeger.example.com:4317

service:
  pipelines:
    traces/dev:
      receivers: [otlp]
      processors: [batch]
      exporters: [otlp/jaeger]
```





```yaml
receivers:
  otlp: # the OTLP receiver the app is sending logs to
    protocols:
      http:
        endpoint: 0.0.0.0:4318

processors:
  batch:

exporters:
  file: # the File Exporter, to ingest logs to local file
    path: ./app42_example.log
    rotation:

service:
  pipelines:
    logs/dev:
      receivers: [otlp]
      processors: [batch]
      exporters: [file]
```

```yaml
receivers:
  otlp: # the OTLP receiver the app is sending metrics to
    protocols:
      http:
        endpoint: 0.0.0.0:4318

processors:
  batch:

exporters:
  prometheusremotewrite: # the PRW exporter, to ingest metrics to backend
    endpoint: https://prw.example.com/v1/api/remote_write

service:
  pipelines:
    metrics/prod:
      receivers: [otlp]
      processors: [batch]
      exporters: [prometheusremotewrite]
```

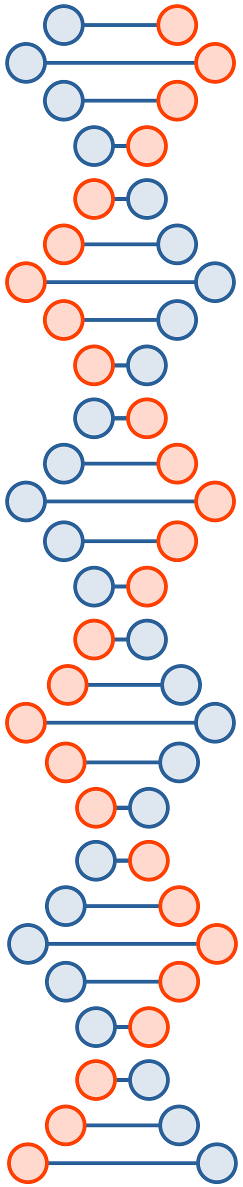


Receivers collect telemetry from one or more sources.

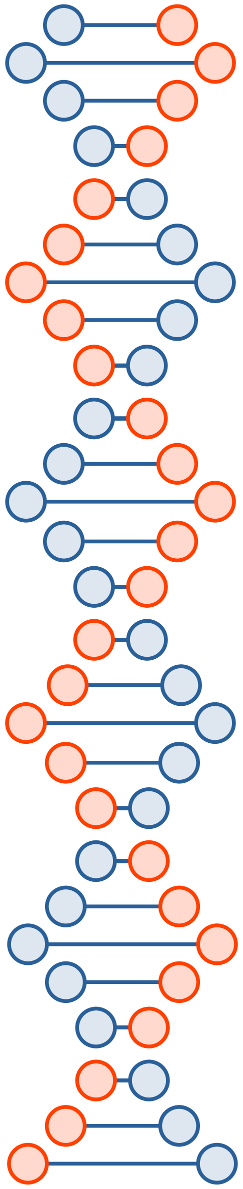
They can be pull or push based, and may support one or more data sources.

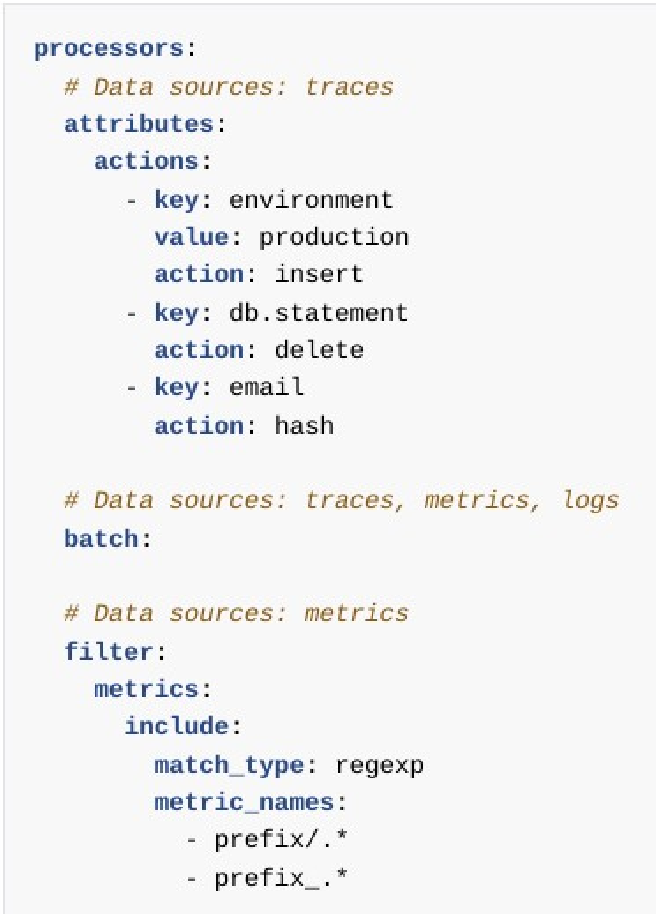
```yaml
receivers:
  # Data sources: logs
  fluentforward:
    endpoint: 0.0.0.0:8006

  # Data sources: metrics
  hostmetrics:
    scrapers:
      cpu:
      disk:
      filesystem:
      load:
      memory:
      network:
      process:
      processes:
      paging:

  # Data sources: traces
  jaeger:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4317
      thrift_binary:
      thrift_compact:
      thrift_http:
```

```yaml
  # Data sources: traces, metrics, logs
  kafka:
    protocol_version: 2.0.0

  # Data sources: traces, metrics
  opencensus:

  # Data sources: traces, metrics, logs
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4317
      http:
        endpoint: 0.0.0.0:4318

  # Data sources: metrics
  prometheus:
    config:
      scrape_configs:
        - job_name: otel-collector
          scrape_interval: 5s
          static_configs:
            - targets: [localhost:8888]

  # Data sources: traces
  zipkin:
```

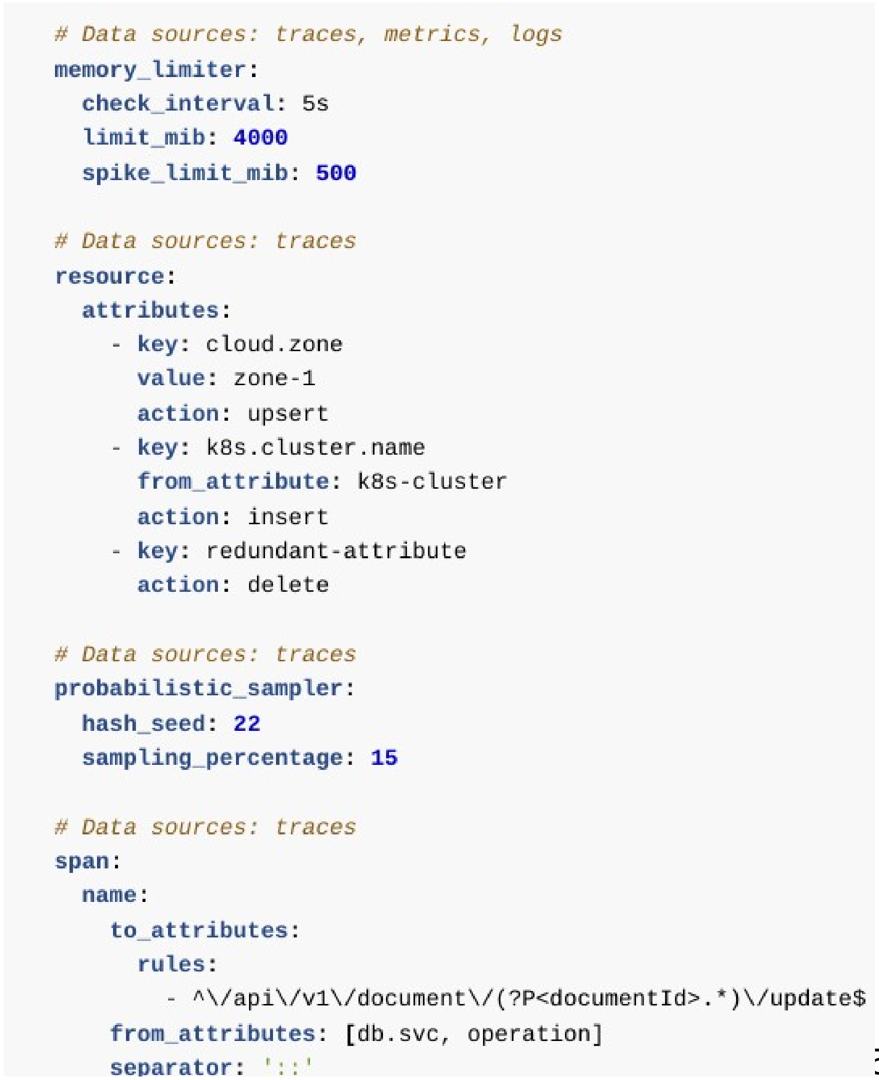


Processors take the data collected by receivers and modify or transform it before sending it to the exporters.





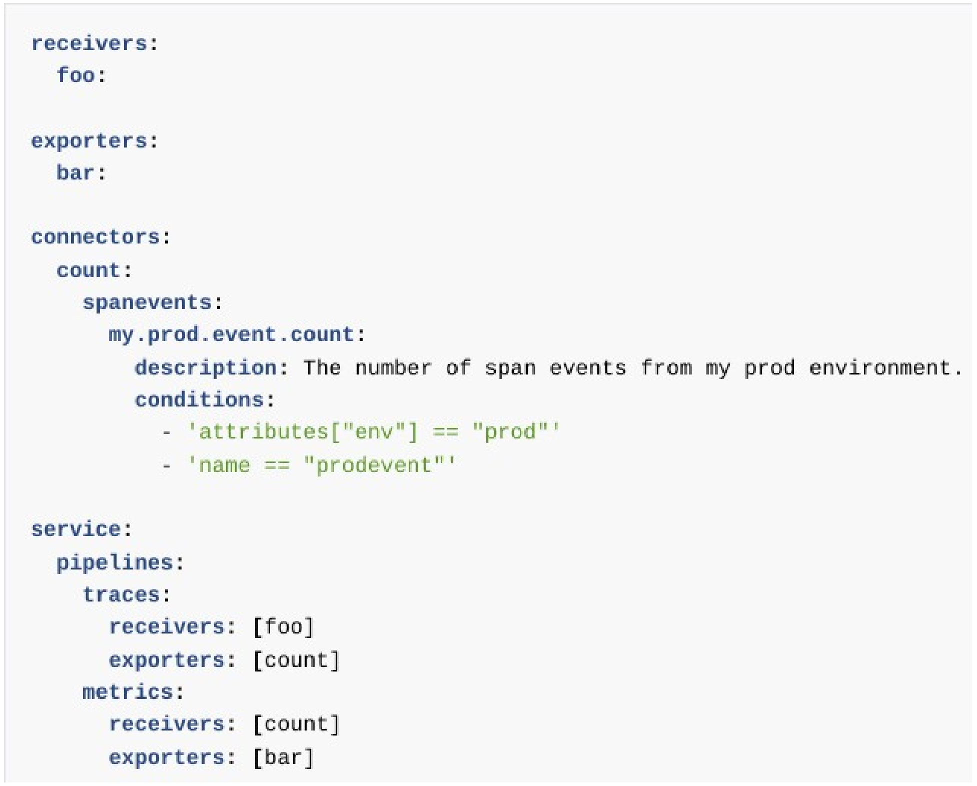
```yaml
processors:
  # Data sources: traces
  attributes:
    actions:
      - key: environment
        value: production
        action: insert
      - key: db.statement
        action: delete
      - key: email
        action: hash

  # Data sources: traces, metrics, logs
  batch:

  # Data sources: metrics
  filter:
    metrics:
      include:
        match_type: regexp
        metric_names:
          - prefix/.*
          - prefix_.*
```



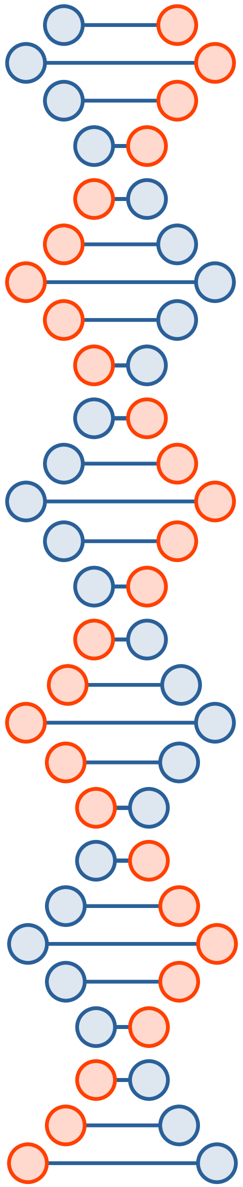
```yaml
# Data sources: traces, metrics, logs
memory_limiter:
  check_interval: 5s
  limit_mib: 4000
  spike_limit_mib: 500

# Data sources: traces
resource:
  attributes:
    - key: cloud.zone
      value: zone-1
      action: upsert
    - key: k8s.cluster.name
      from_attribute: k8s-cluster
      action: insert
    - key: redundant-attribute
      action: delete

# Data sources: traces
probabilistic_sampler:
  hash_seed: 22
  sampling_percentage: 15

# Data sources: traces
span:
  name:
    to_attributes:
      rules:
        - ^\/api\/v1\/document\/(?P<documentId>.*)\/update$
    from_attributes: [db.svc, operation]
    separator: '::'
```

```yaml
exporters:
  # Data sources: traces, metrics, logs
  file:
    path: ./filename.json

  # Data sources: traces
  otlp/jaeger:
    endpoint: jaeger-server:4317
    tls:
      cert_file: cert.pem
      key_file: cert-key.pem

  # Data sources: traces, metrics, logs
  kafka:
    protocol_version: 2.0.0

  # Data sources: traces, metrics, logs
  # NOTE: Prior to v0.86.0 use `logging` instead of `debug`
  debug:
    verbosity: detailed

  # Data sources: traces, metrics
  opencensus:
    endpoint: otelcol2:55678
```

```yaml
  # Data sources: traces, metrics, logs
  otlp:
    endpoint: otelcol2:4317
    tls:
      cert_file: cert.pem
      key_file: cert-key.pem

  # Data sources: traces, metrics
  otlphttp:
    endpoint: https://otlp.example.com:4318

  # Data sources: metrics
  prometheus:
    endpoint: 0.0.0.0:8889
    namespace: default

  # Data sources: metrics
  prometheusremotewrite:
    endpoint: http://prometheus.example.com:9411/api/prom/push
    # When using the official Prometheus (running via Docker)
    # endpoint: 'http://prometheus:9090/api/v1/write', add:
    # tls:
    #   insecure: true

  # Data sources: traces
  zipkin:
    endpoint: http://zipkin.example.com:9411/api/v2/spans
```



```yaml
receivers:
  foo:

exporters:
  bar:

connectors:
  count:
    spanevents:
      my.prod.event.count:
        description: The number of span events from my prod environment.
        conditions:
          - 'attributes["env"] == "prod"'
          - 'name == "prodevent"'

service:
  pipelines:
    traces:
      receivers: [foo]
      exporters: [count]
    metrics:
      receivers: [count]
      exporters: [bar]
```



Connectors join two pipelines, acting as both exporter and receiver.

A connector consumes data as an exporter at the end of one pipeline and emits data as a receiver at the beginning of another pipeline.

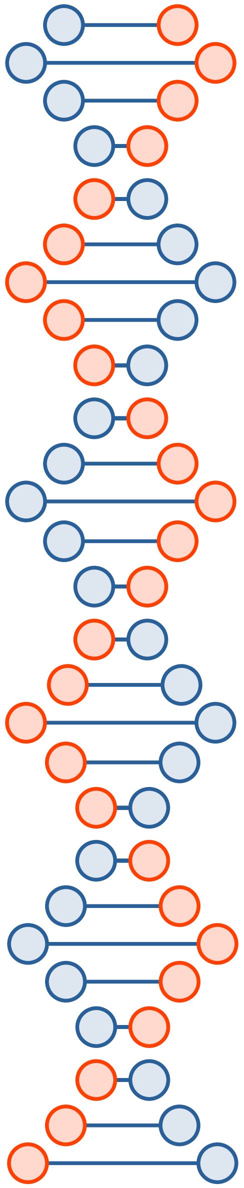
The service section consists of three subsections:

Extensions

Pipelines

Telemetry

```
service:
  pipelines:
    metrics:
      receivers: [opencensus, prometheus]
      processors: [batch]
      exporters: [opencensus, prometheus]
    traces:
      receivers: [opencensus, jaeger]
      processors: [batch, memory_limiter]
      exporters: [opencensus, zipkin]
```



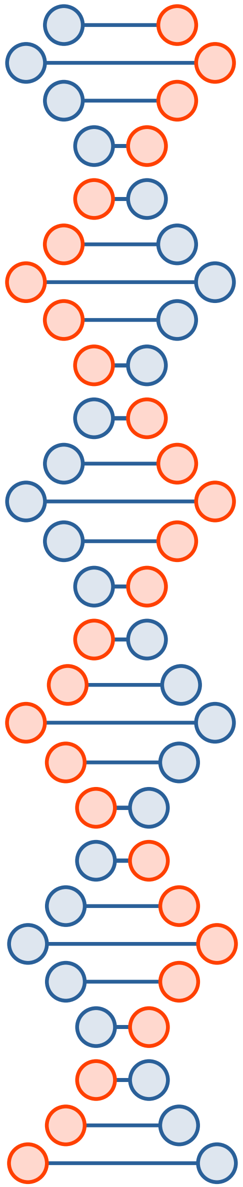
How to manage your OpenTelemetry collector deployment at scale

Example Scenario

Consider an organization that has hundreds of microservices deployed across multiple cloud environments.

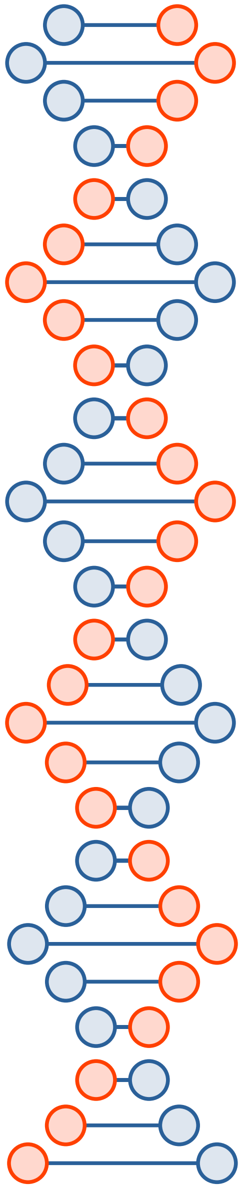
Each microservice is instrumented with an OpenTelemetry agent to collect traces and metrics.

Without OpAMP, every time the telemetry strategy changes (e.g., changing the sampling rate or adding new exporters), an operator would need to manually update each agent's configuration and restart it.



OpenTelemetry Agent Management Protocol (OTAMP), often referred to as **OpAMP**, is a protocol designed to manage the configuration, lifecycle, and status of OpenTelemetry agents and collectors.

It enables centralized control over distributed agents, allowing you to dynamically configure, monitor, and update them in a scalable manner without needing manual intervention on each agent.

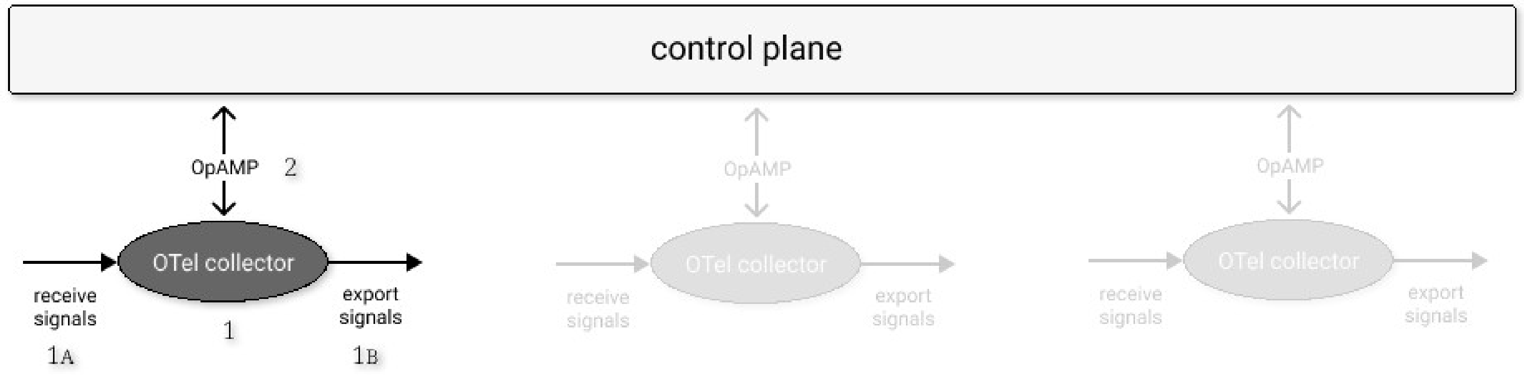


How OpAMP Works

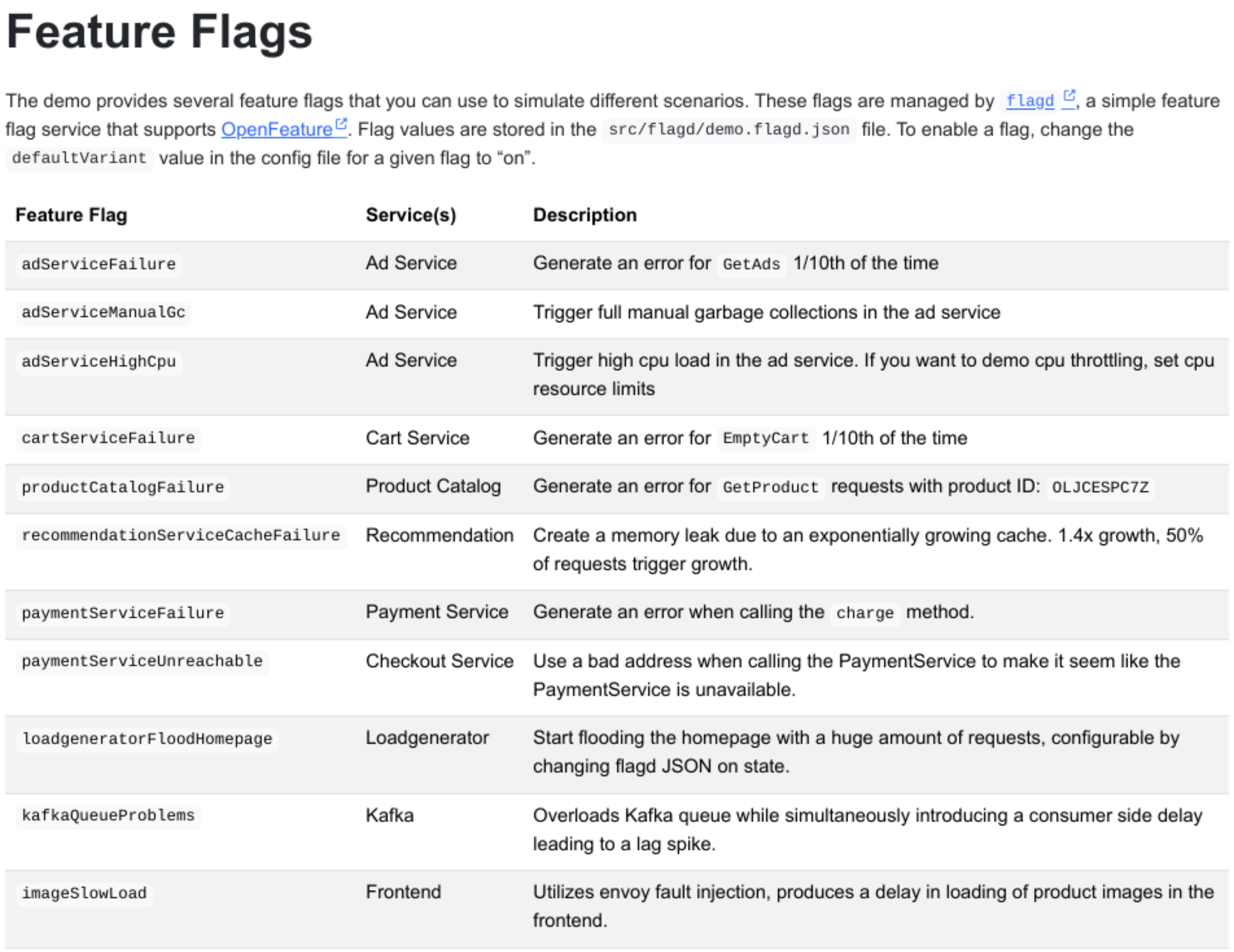
OpAMP works in a client-server model:

OpAMP Server: This is the centralized component that manages all the OpenTelemetry agents or collectors. The server holds the configurations, agent versions, and other control settings.

OpAMP Clients: These are OpenTelemetry agents or collectors deployed on various machines or environments. They communicate with the OpAMP server, receive configuration updates, and report their status.


control plane
OpAMP
2
OTel collector
receive signals
export signals
1
1A
1B
OpAMP
OTel collector
receive signals
export signals
OpAMP
OTel collector
receive signals
export signals

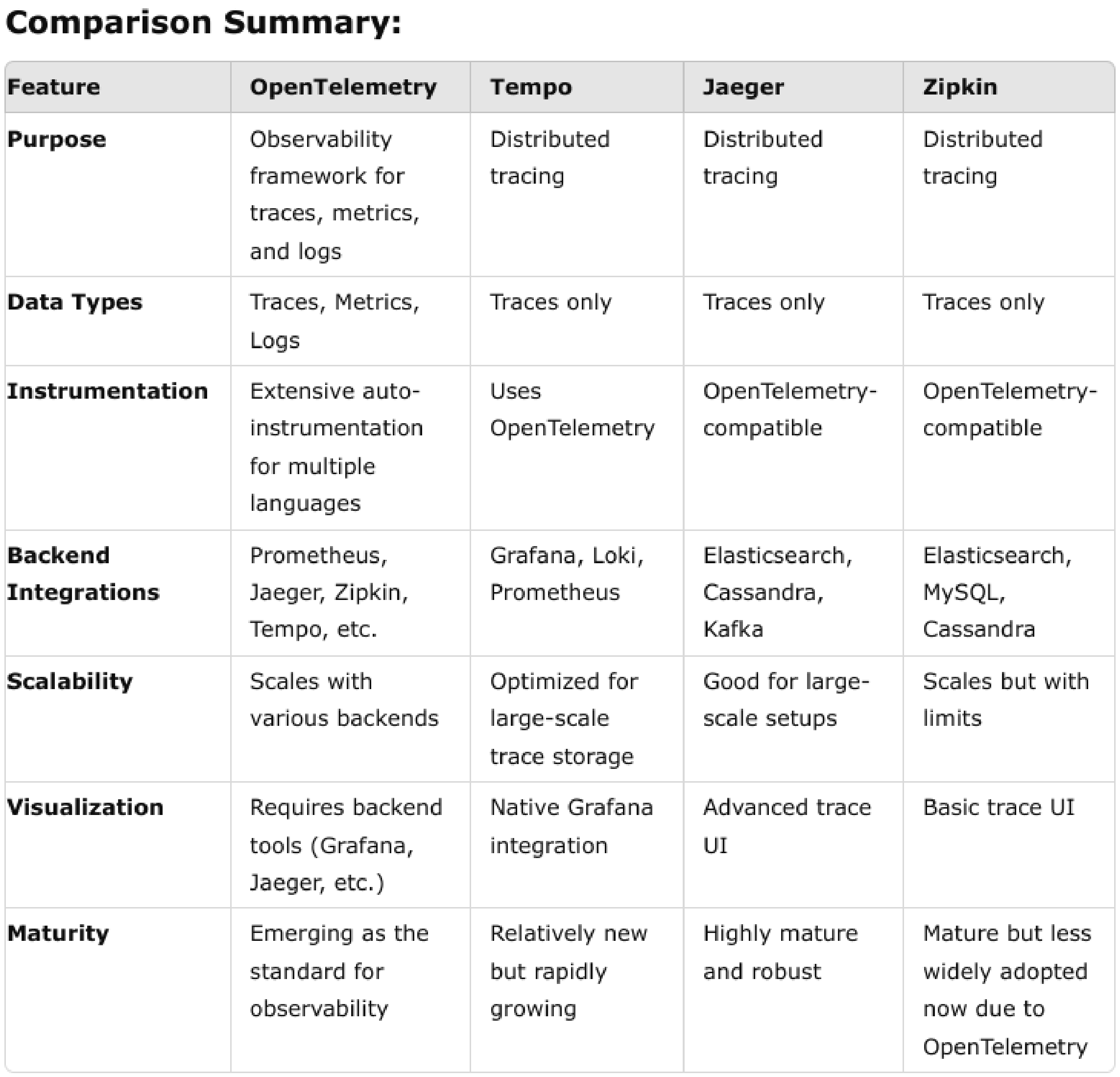
https://opentelemetry.io/docs/demo/



Feature Flags

The demo provides several feature flags that you can use to simulate different scenarios. These flags are managed by `flagd`, a simple feature flag service that supports OpenFeature. Flag values are stored in the `src/flagd/demo.flagd.json` file. To enable a flag, change the `defaultVariant` value in the config file for a given flag to "on".

Feature Flag	Service(s)	Description
`adServiceFailure`	Ad Service	Generate an error for `GetAds` 1/10th of the time
`adServiceManualGc`	Ad Service	Trigger full manual garbage collections in the ad service
`adServiceHighCpu`	Ad Service	Trigger high cpu load in the ad service. If you want to demo cpu throttling, set cpu resource limits
`cartServiceFailure`	Cart Service	Generate an error for `EmptyCart` 1/10th of the time
`productCatalogFailure`	Product Catalog	Generate an error for `GetProduct` requests with product ID: `OLJCESPC7Z`
`recommendationServiceCacheFailure`	Recommendation	Create a memory leak due to an exponentially growing cache. 1.4x growth, 50% of requests trigger growth.
`paymentServiceFailure`	Payment Service	Generate an error when calling the `charge` method.
`paymentServiceUnreachable`	Checkout Service	Use a bad address when calling the PaymentService to make it seem like the PaymentService is unavailable.
`loadgeneratorFloodHomepage`	Loadgenerator	Start flooding the homepage with a huge amount of requests, configurable by changing flagd JSON on state.
`kafkaQueueProblems`	Kafka	Overloads Kafka queue while simultaneously introducing a consumer side delay leading to a lag spike.
`imageSlowLoad`	Frontend	Utilizes envoy fault injection, produces a delay in loading of product images in the frontend.



Comparison Summary:

Feature	OpenTelemetry	Tempo	Jaeger	Zipkin
Purpose	Observability framework for traces, metrics, and logs	Distributed tracing	Distributed tracing	Distributed tracing
Data Types	Traces, Metrics, Logs	Traces only	Traces only	Traces only
Instrumentation	Extensive auto-instrumentation for multiple languages	Uses OpenTelemetry	OpenTelemetry-compatible	OpenTelemetry-compatible
Backend Integrations	Prometheus, Jaeger, Zipkin, Tempo, etc.	Grafana, Loki, Prometheus	Elasticsearch, Cassandra, Kafka	Elasticsearch, MySQL, Cassandra
Scalability	Scales with various backends	Optimized for large-scale trace storage	Good for large-scale setups	Scales but with limits
Visualization	Requires backend tools (Grafana, Jaeger, etc.)	Native Grafana integration	Advanced trace UI	Basic trace UI
Maturity	Emerging as the standard for observability	Relatively new but rapidly growing	Highly mature and robust	Mature but less widely adopted now due to OpenTelemetry

Best Fit:

OpenTelemetry:

For complete observability (traces, metrics, logs) and compatibility with different backends.

Tempo:

For highly scalable, cost-effective, and simple trace collection in a Grafana-centric environment.

Jaeger:

For robust distributed tracing with advanced features, particularly in microservices systems.

Zipkin:

For simpler tracing setups and legacy environments, though OpenTelemetry may replace it in new projects.