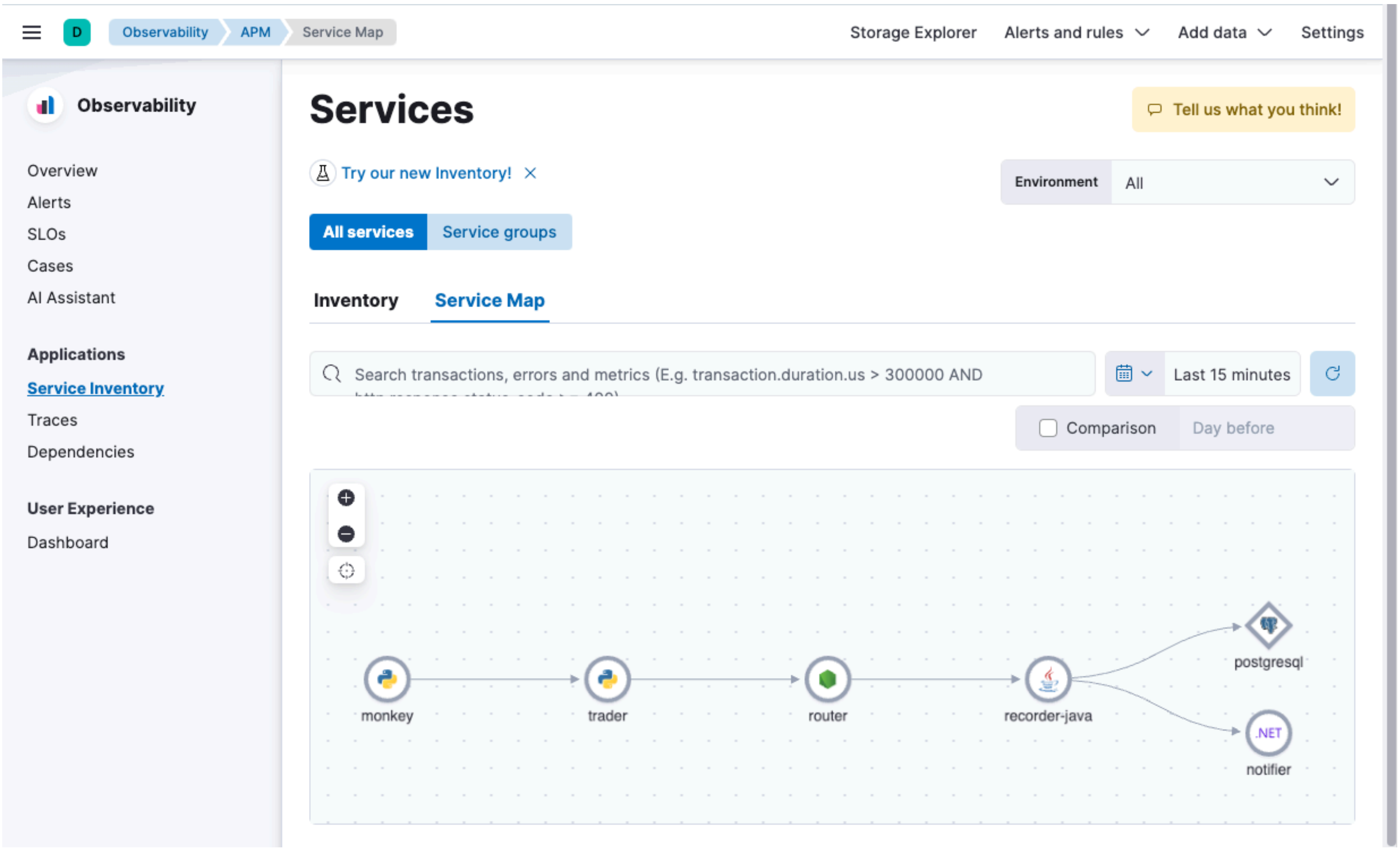
Observability 200: Adding attributes to traces and logs with OpenTelemetry

To help us better appreciate how OpenTelemetry is forever changing observability, we will be working with an example stock trading system, comprised of several services and their dependencies, all instrumented using OpenTelemetry.

We will be working with a live Elasticsearch instance, displayed in the browser tab to the left. We are currently looking at Elastic's dynamically generated Service Map. It shows all of the services that comprise our system, and how they interact with one another.



Our trading system is composed of: * `trader`: a python application that trades stocks on orders from customers * `router`: a node.js application that routes committed trade records * `recorder-java`: a Java application that records trades to a PostgreSQL database * `notifier`: a .NET application that notifies an external system of completed trades

Finally, we have `monkey`, a python application we use for testing our system that makes periodic, automated trade requests on behalf of fictional customers.

> [!NOTE] You are welcome to explore each service and our APM solution by clicking on each service icon in the Service Map and selecting `Service Details`

When you are ready, click the `Next` button to continue. ___

Let's put ourselves in the shoes of an on-duty SRE. Assume you just were just assigned a ticket indicating that a customer with the id `l.hall` is experiencing problems.

Searching logs for user activity

Elastic is well known for its ability to search log records for arbitrary strings. This feature alone bring immense value to otherwise unstructured logs. Let's have a go at searching our application logs for lines related to `l.hall` using the tools available to us today.

Let's first just do a simplistic search for the string `l.hall` in any field within our logs: 1. Open the button label="Elasticsearch" tab 2. Copy `kql      l.hall` into the `Filter your data using KQL syntax` search bar toward the top of the Kibana window 3. Click on the refresh icon at the right of the time picker 4. Click on the `Patterns` tab

One immediate concern is that clearly customer id is expressed throughout our logs in unique ways. Already we see many patterns: * `172.18.0.13 - - [18/Jan/2025 16:27:27] "POST /trade/request?symbol=BAA&day_of_week=Tu&customer_id=l.hall`

`HTTP/1.1"`*`trade committed for l.hall`*`traded BAA on day Tu for l.hall`*`trading ZYX for l.hall on Tu from LATAM with latency 0, error_model=false, error_db=false, skew_market_factor=0, canary=false`

While Elasticsearch can easily surface any log line containing `l.hall`, it does make clear the reliance SREs have on developers when searching for clues related to a given user or session: searchable attributes exist (or don't exist) and are named often at the whim of the developer. This search only returns log lines where the value of the `customer_id` variable was intentionally inserted into the log message by the developer. Without further development effort across functions and microservices, many log lines related to the processing of `l.hall` requests will not be returned because the developer did not think to proactively include `customer_id` in the log message.

Parsing logs with ES|QL

Often, it is helpful to further perform some analysis on a given field. In this case, assume we want to graph customer id by region to see if `l.hall` is connecting to just one or to all of our regions.

Let's choose one of the patterns above and parse out customer id with ES|QL: 1. Click `Try ES|QL` in the upper taskbar 2. Copy `es|ql     FROM logs-*     | GROK message """(?:^|[?&])customer_id=(?<customer_id>[^&]*).*region=(?<region>[^&]*)""` `| WHERE customer_id is not null     | STATS COUNT(customer_id) BY customer_id, region` into the ES|QL input box (replacing whatever is there) 3. Click `Run`

This is incredibly powerful: we just taught Elasticsearch how to parse and make use of a new `customer_id` field. We were able to accomplish this "just in time" for search and analysis without resorting to ingest pipelines or other ingest-time parsing.

Like any string parsing exercise, however, this is fragile. If the pattern changes dramatically, this ES|QL will fail. And of course this GROK expression only covers one of the patterns in which customer id appears. Crafting ES|QL to accommodate all such patterns is of course possible, but more complex to write and maintain.

Ideally, of course, the log lines would all have a common, structured field like `customer_id`.

Searching traces for user activity

Conceptually, traces are an ideal way of tracking user activity through a system. They nicely capture execution flows across services, recording latency and outcomes at each step.

Let's look for traces related to `l.hall`: 1. Open the button label="Elasticsearch" tab 2. Navigate to `Observability` / `APM` / `Traces` 3. Click on the `Explorer` tab 4. Copy `kql     l.hall` into the `Filter your data using KQL syntax` search bar toward the top of the Kibana window 5. Click `Search`

This is actually surprisingly useful! We can obviously see and debug the database error that is plaguing `l.hall`.

Let's see how Elasticsearch was able to surface this transaction without explicit labels: 1. Click on the first row `POST` 2. Scroll down in the `Transaction details` flyout to find the `url` section 3. Note that `l.hall` appears in the `url.full` as a request parameter

Using the same text search available with logs, Elasticsearch found the parent `POST` transaction from the text embedded in the `url`, and then using `trace.id`, it was able to assemble all related spans, including the one that ultimately ended in failure (the database transaction).

Why labels matter

While helpful, similar to logs, relying on arbitrary text strings is not ideal: * in this case, we got lucky in that our customer id happens to be captured in the request arguments, and that the auto instrumentation for flask records the request arguments in an attribute * there is no way for us to leverage Elastic's advanced OOTB analytics (which generally require discrete attributes) to help us quickly determine if this problem is happening only to `l.hall` or to all users.

1. Open the button label="Elasticsearch" tab
2. Navigate to `Observability` / `APM` / `Service Inventory`
3. Click on the `Service Map` tab
4. Click on the `trader` service
5. Click on `Service Details`
6. Under `Transactions`, click on `POST /trade/request`
7. Scroll down to `Trace samples`, and click on `Failed transaction correlations`

Note that the results aren't particularly helpful in determining if this problem is specific to `l.hall` or more systemic.

Adding labels to spans

Most/all existing APM frameworks do offer the ability to add labels to your traces, with some caveats: * adding labels generally means adding custom observability code to your service * prior to OpenTelemetry, each vendor had their own API for adding labels, so adding labels meant vendor lock-in * there was no way to automatically propagate labels through your services, so custom code would have to be added to each service in the calling chain leading to a similar problem as logs: each developer could name the `customer_id` label in a different fashion complicating search

Let's see how OpenTelemetry can help address these concerns!

Adding span attributes

Among its many virtues, OpenTelemetry's support for common attributes which span across observability signals and your distributed services not only provides a solution to the aforementioned problems, but will also fuel the ML and AI based analysis we will consider in subsequent labs.

With only a very small investment in manual instrumentation (really, just a few lines of code!) on top of auto-instrumentation, every log and trace emitted by your distributed microservices can be tagged with common, application-specific metadata, like a `customer_id`.

`trader` is at the front of our user-driven call stack and seems like an ideal place to add a `customer_id` attribute. Let's do it!

`trader` is a simple python app built on the flask web application framework. We are leveraging OpenTelemetry's rich library of python auto-instrumentation to generate spans when APIs are called without having to explicitly instrument service calls.

So what needs to be added on top of OpenTelemetry's auto-instrumentation to add attributes to a span? Not much!

1. Open the button label="VS Code" tab
2. Navigate to `src` / `trader` / `app.py`
3. Look for the python code around line 36: `python,nocopy customer_id = request.args.get('customer_id', default=None, type=str)`
4. Immediately thereafter, add this line to add customer_id as a span attribute: `python trace.get_current_span().set_attrib` `customer_id)` what does this do?
    - gets the current span (from context on the current thread)
    - sets customer_id (prefixed by `com.example` for best practice)
5. You should now have: `python,nocopy customer_id = request.args.get('customer_id', default=None, type=str) trace.get_current_span().set_attribute(f"{ATTRIBUTE_PREFIX}.customer_id", customer_id)`
6. Save the file (Command-S on Mac, Ctrl-S on Windows) or use the VS Code "hamburger" menu and select `File` / `Save`
7. Enter the following in the terminal pane of VS Code to recompile the `trader` service: `bash ./rebuild.sh -s trader`

After issuing the rebuild command, wait for the build to complete...

Taking stock of our work

Let's see how far this gets us:

1. Open the button label="Elasticsearch" tab
2. Navigate to `Observability` / `APM` / `Traces`
3. Click on the `Explorer` tab
4. Copy `kql attributes.com.example.customer_id : "l.hall"` into the `Filter your data using KQL syntax` search bar toward the top of the Kibana window
5. Click `Search`
6. Click on the parent transaction `POST /trade/request` toward the top of the waterfall graph
7. A flyout panel will open on the right, showing all of the metadata associated with the span. Note the presence of `attributes.com.example.customer_id`.

Ah-ha! This is clearly a much cleaner approach to definitively searching for customer id than relying on text matching.

8. Scroll down and click on the failed `SELECT trades.trades` child span
9. A flyout panel will open on the right, showing all of the metadata associated with the span. Note the lack of the label `attributes.com.example.customer_id`.

Unfortunately, *this* span (which covers interactions with the database) *isn't* labeled with `com.example.customer_id`. We only labeled one span (`POST /trade/request`), and while that is ultimately a parent of this span, labels applied to the parent are not applied to their children.

Leveraging Elastic's analytics

We should now be able to leverage Elastic's advanced analytics to help us determine the scope of this issue:

1. Open the button label="Elasticsearch" tab
2. Navigate to `Observability` / `APM` / `Service Inventory`
3. Click on the `Service Map` tab
4. Click on the `trader` service
5. Click on `Service Details`
6. Under `Transactions`, click on `POST /trade/request`
7. Scroll down to `Trace samples`, and click on `Failed transaction correlations`

Nice - it appears all of our failed transactions are specific to `l.hall`! You can clearly start to see the value use of definitive labels brings to Elasticsearch.

What is good enough?

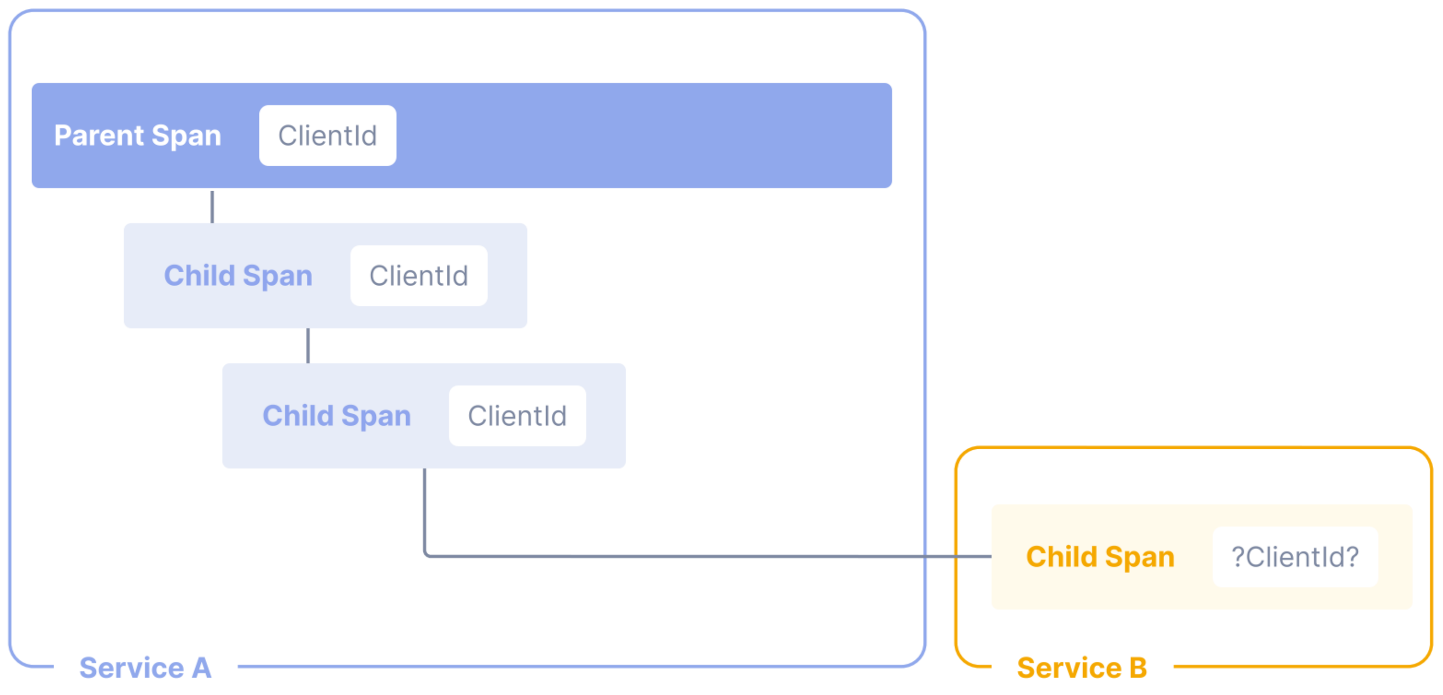
Is lack of definitive labels on spans problematic? Consider a few things: * perhaps you want to analyze specific child spans (like `span.type : "db"`) for errors * remember that we got "lucky" that our customer_id happened to be captured by auto-instrumentation

Also, don't forget the problems we encountered when searching logs (like missing log lines).

What if we could apply labels *once* and have them automatically propagate through every related child span? Amazingly, this is possible with OpenTelemetry using Baggage!

OpenTelemetry Baggage

OpenTelemetry Baggage lets you add arbitrary key/value pairs to contextual "baggage" which is automatically ferried between distributed services on remote API calls (gRPC, REST, ...), alongside a trace and span ID (to establish distributed tracing).



As you might suspect, we can leverage OpenTelemetry Baggage to carry our span attributes between spans and services!

Adding elements to baggage is trivial.

Let's return to the `trader` code: 1. Open the button label="VS Code" tab 2. Navigate to `src` / `trader` / `app.py` 3. Go back to the python code around line 36 where we previously added the customer_id attribute: `python,nocopy     customer_id = request.args.get('customer_id', default=None, type=str)     trace.get_current_span().set_attribute(f"{ATTRIBUTE_PR` `customer_id)` 4. Now let's put that attribute we just created in baggage by adding the following line `python      context.attach(ba` `customer_id))` 5. You should now have: `python,nocopy     customer_id = request.args.get('customer_id', default=None, type=str)     trace.get_current_span().set_attribute(f"{ATTRIBUTE_PREFIX}.customer_id", customer_id)     context.a` `customer_id))` 6. Save the file (Command-S on Mac, Ctrl-S on Windows) or use the VS Code "hamburger" menu and select `File` / `Save`

> [!NOTE] OpenTelemetry Baggage key/value pairs are *always* strings. You will need to keep this in mind if you are trying to propagate an integer attribute, for example. This is particularly important to note if you are manually applying attributes in a parent span as an integer, and then passing that attribute to child spans and services via baggage. If added automatically to child spans, the attribute *must* have the same type (string) as the manually added attribute in the parent span.

Notably, while this will add this attribute to baggage, that in and of itself won't actually do anything. Baggage is carried from span to span (via thread context) and from service to service (via REST or gRPC). But by default, nothing is actually *done* with that baggage. We need to tell OpenTelemetry what to do with it.

Propagating Baggage with Extensions

Notably, OpenTelemetry Baggage is merely a transport for contextual key/value pairs between spans and services. So how do we get the attributes in baggage to be automatically applied to every child span without having to introduce new code into our applications? Fortunately, OpenTelemetry supports a concept of extensions; namely, loadable modules which can hook into various parts of OpenTelemetry. There already exists an extension (`BaggageSpanProcessor`) for most popular languages which will apply attributes in baggage to the current span.

Adding the BaggageSpanProcessor to Java

For Java, the BaggageSpanProcessor can be added purely at the orchestration layer:

1. Open the button label="VS Code" tab
2. Right-click on
3. Create a new file in the folder `operator` with the following contents `yaml  apiVersion: opentelemetry.io/v1alpha1 kind: Instrumentation  metadata:  name: elastic-instrumentation  spec:  java:      image: docker.elastic.co/o` `extensions:      - image: us-central1-docker.pkg.dev/elastic-sa/tbekiares/baggage-processor          dir: /extensions      env:      - name: OTEL_JAVA_TEST_SPAN_ATTRIBUTES_COPY_FROM_BAGGAGE_INCLUDE          value: '*'`
4. Save the file as `operator/java.yaml` (Command-S on Mac, Ctrl-S on Windows) or use the VS Code "hamburger" menu and select `File` / `Save`
5. Redeploy the instrumentation CRD by issuing the following in the VSCode Terminal: `kubectl -n opentelemetry-operator-sy` `apply -f java.yaml`
6. Redeploy the `recorder-java` service by issuing the following in the VSCode Terminal: `kubectl -n trading rollout restart deployment/recorder-java`

Let's test

1. Open the button label="Elasticsearch" tab
2. Navigate to `Observability` / `APM` / `Traces`
3. Click on the `Explorer` tab
4. Copy `kql  attributes.com.example.customer_id : "l.hall"` into the `Filter your data using KQL syntax` search bar toward the top of the Kibana window
5. Click `Search`
6. Click on the child spans inside `recorder-java`; note in the flyout for each that the `attributes.com.example.customer_id` exists
7. Scroll down and click on the failed `SELECT trades.trades` span
8. Note that our label is *also* applied to the SQL spans!

As you've seen, OpenTelemetry Baggage, with the help of OpenTelemetry Extensions, makes it possible to add contextual attributes across spans and services: * *without* requiring explicit code to add attributes to child functions * *without* requiring explicit code to transfer attributes between services ___

Attributes add context to logs!

So we've successfully added lots of rich attributes to our spans, but let's return to our log search:

1. Open the button label="Elasticsearch" tab
2. Copy `kql   l.hall` into the `Filter your data using KQL syntax` search bar toward the top of the Kibana window
3. Click on the refresh icon at the right of the time window picker
4. Click on the `Patterns` tab

Wouldn't it be great if we could also propagate our span attributes to our logs as well?

With common, automatically applied, attributes across your logs, you can focus your message text on **meaningful content that will help you perform RCA**, rather than manually adding context in a bespoke fashion that requires parsing and can obfuscate the actual message.

While most OpenTelemetry SDK distributions today include extensions to automatically apply attributes from baggage to spans, there does not currently exist extensions to automatically apply attributes from baggage to logs.

Fortunately, it is easy to author our own OpenTelemetry extensions!

Java

And like the BaggageSpanProcessor, we can install it purely at the orchestration layer:

7. Navigate to `operator` / `values.yaml`
8. Scroll to the bottom and find the `instrumentation` config for `java`
9. Find `yaml   java:       image: docker.elastic.co/observability/elastic-otel-javaagent:1.3.0`
10. Immediately thereafter, add these lines:

```
    env:
    - name: OTEL_JAVA_TEST_LOG_ATTRIBUTES_COPY_FROM_BAGGAGE_INCLUDE
        value: '*'
```

11. You should have:

```
apiVersion: opentelemetry.io/v1alpha1
kind: Instrumentation
metadata:
name: elastic-instrumentation
spec:
java:
    image: docker.elastic.co/observability/elastic-otel-javaagent:1.3.0
    extensions:
    - image: us-central1-docker.pkg.dev/elastic-sa/tbekiares/baggage-processor
        dir: /extensions
    env:
    - name: OTEL_JAVA_TEST_SPAN_ATTRIBUTES_COPY_FROM_BAGGAGE_INCLUDE
        value: '*'
    - name: OTEL_JAVA_TEST_LOG_ATTRIBUTES_COPY_FROM_BAGGAGE_INCLUDE
        value: '*'
```

12. Save the file (Command-S on Mac, Ctrl-S on Windows) or use the VS Code "hamburger" menu and select `File` / `Save`
13. Redeploy the instrumentation CRD by issuing the following in the VSCode Terminal: `kubectl -n opentelemetry-operator-sy` `apply -f java.yaml`
14. Redeploy the `recorder-java` service by issuing the following in the VSCode Terminal: `kubectl -n trading rollout restart deployment/recorder-java`

Let's test

1. Enter the following in the command line pane of VS Code to recompile: `bash   docker compose up -d --build`

Wait for the build to complete....

1. Open the button label="Elasticsearch" tab
2. Copy `kql   attributes.com.example.customer_id : "l.hall"` into the `Filter your data using KQL syntax` search bar toward the top of the Kibana window
3. Click on the refresh icon at the right of the time picker

Nice! We now have a common set of attributes, added *once* in our `trader` service, automatically propagated throughout our stack with minimal work!

> [!NOTE] In a subsequent lab, we will explore adding attributes from baggage to log lines where we cannot rely on the OpenTelemetry logging framework.

Summary

Behold: minimal effort for maximum effect. No more fragile regex expressions. Straightforward and definitive searching of traces and logs indexed by *your* metadata. Imagine your SREs being able to quickly land definitively search for all logs or traces related to a given customer. ___