Protecting Data

We've gotten word from our customer service department that some users are unable to complete stock trades. We know that all of the REST API calls from our front end web application flow through a nginx reverse proxy, so that seems like a good place to start our investigation.

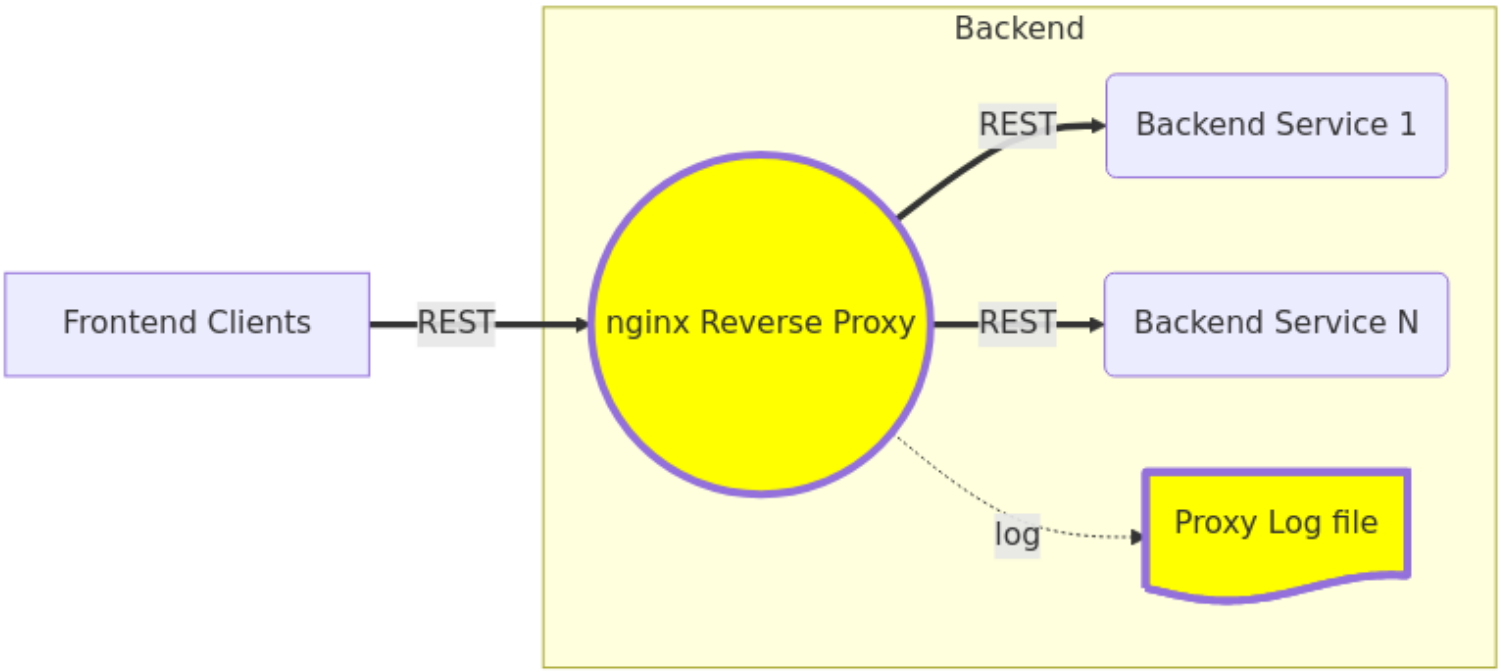


Figure 1: proxy_arch.mmd.png

Ingest vs. query-time parsing

We will also be pivoting back and forth between query-time parsing using ES|QL and ingest-time parsing using Streams. ES|QL lets us quickly test theories and look for possible tells in our log data. Once we've determined value in parsing our logs using ES|QL at query-time, we can shift that parsing to ingest-time using Streams. As we will see in this lab, ingest-time parsing allows for more advanced and complex parsing. Moving parsing to ingest-time also facilitates much faster query-time searches. Regardless of where the parsing is done, we will leverage ES|QL to perform aggregations, analysis, and visualization.

Getting started

We will start our investigation using ES|QL to interrogate our nginx reverse proxy logs. You can enter your queries in the pane at the top of the Elasticsearch tab. Set the time field to the last hour, then click "Refresh" to load the results.

Finding the errors

Let's have a look at the logs coming from our nginx reverse proxy.

Execute the following query:

```
FROM logs-proxy.otel-default
```

We can see that there are still transactions occurring, but we don't know if they are successful or failing. Before we spend time parsing our logs, let's just quickly search for common HTTP "500" errors in our nginx logs.

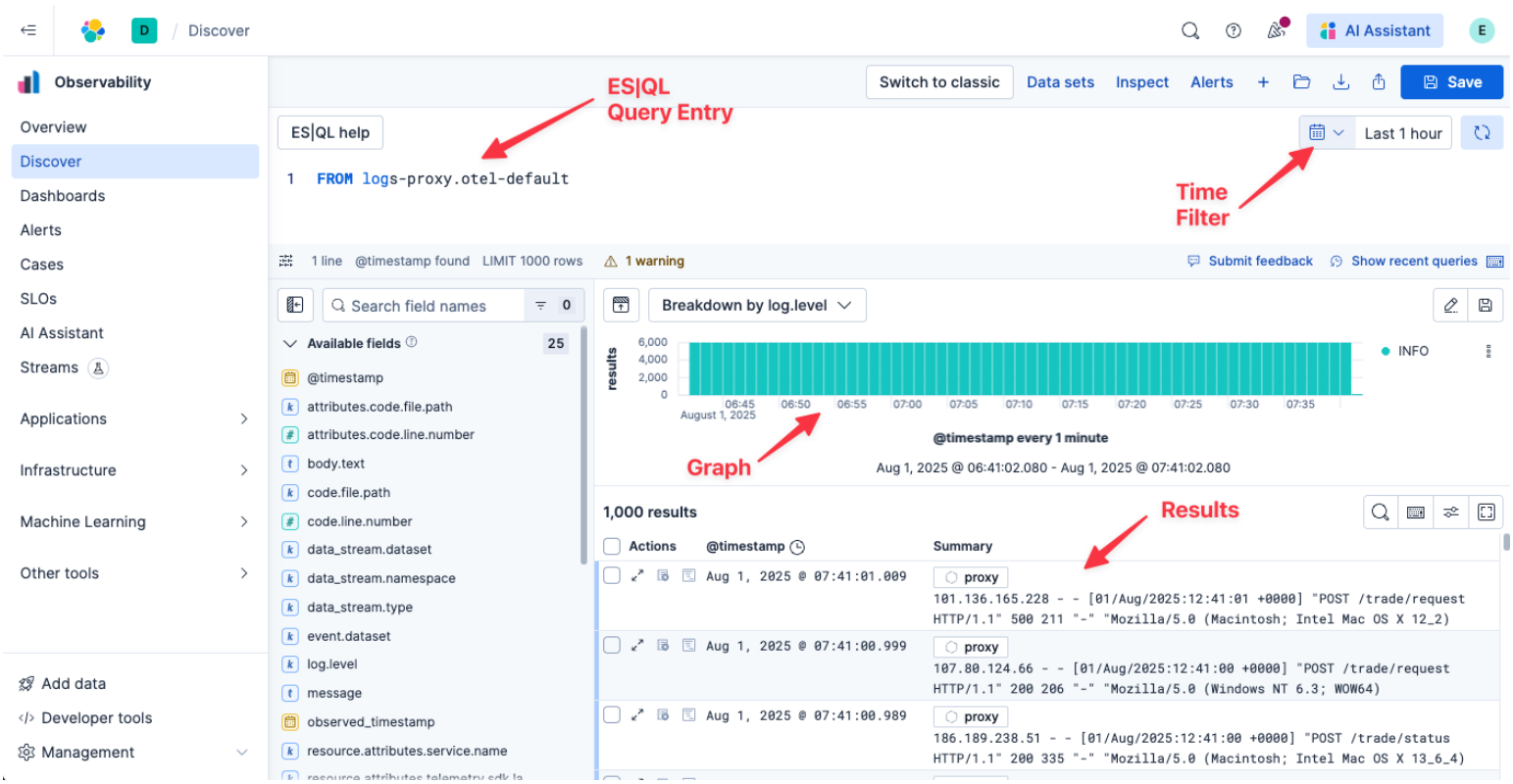


Figure 2: 1_discover.png

Execute the following query:

```
FROM logs-proxy.otel-default
| WHERE body.text LIKE "* 500 *" // look for messages containing " 500 " in the body
```

If we didn't find "500", we could of course add additional `LIKE` criteria to our `WHERE` clause, like `WHERE body.text LIKE "* 500 *" OR body.text LIKE "* 404 *"`. We will do a better job of handling more types of errors once we start parsing our logs. For now, though, we got lucky: indeed, we are clearly returning 500 errors for some users.

Is it affecting everyone?

The next thing we quickly want to understand is what percentage of users are experiencing 500 errors?

Execute the following query:

```
FROM logs-proxy.otel-default
| EVAL status = CASE(body.text LIKE "* 500 *", "bad", "good") // label messages containing " 500 " as "bad", else "good"
| STATS count = COUNT() BY status // count good and bad
```

Let's visualize this as a pie graph to make it a little easier to understand.

1. Click on the pencil icon to the right of the existing graph
2. Select `Pie` from the visualizations drop-down menu
3. Click `Apply and close`

This error appears to only be affecting a small percentage of our overall API queries.

Let's also confirm that we are still seeing a mix of 500 and 200 errors (e.g., the problem wasn't transitory and somehow fixed itself).

Execute the following query:

```
FROM logs-proxy.otel-default
| EVAL status = CASE(body.text LIKE "* 500 *", "bad", "good") // label messages containing " 500 " as "bad", else "good"
| STATS COUNT() BY minute = BUCKET(@timestamp, "1 min"), status
```

Then change the resulting graph to a bar graph over time:

1. Click on the pencil icon to the right of the existing graph
2. Select `Bar` from the visualizations drop-down menu
3. Click `Apply and close`

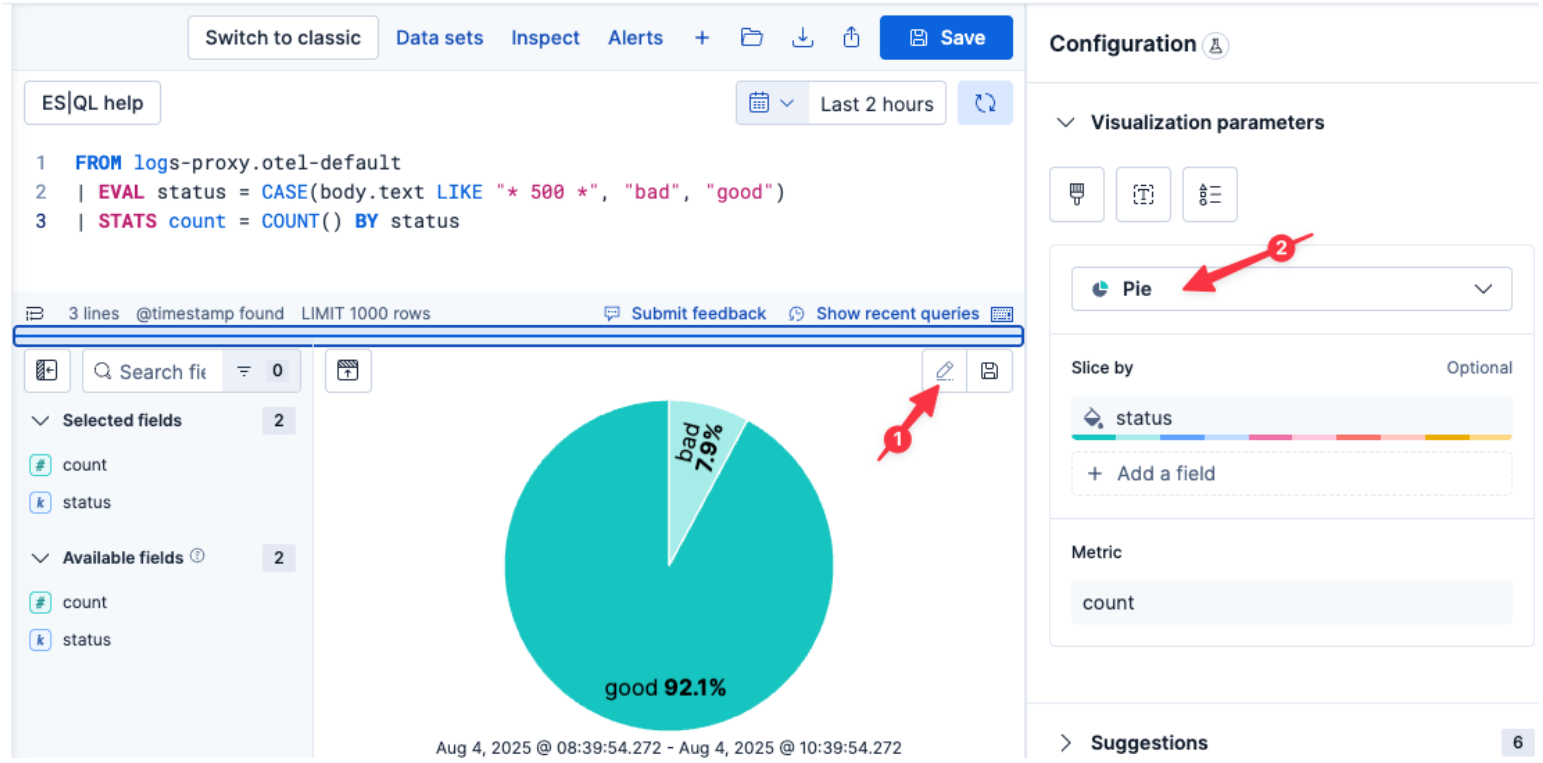


Figure 3: 1_pie.png

Indeed, we are still seeing a mix of 500 and 200 errors.

When did it start?

Let's see if we can find when the errors started occurring. Adjust the time field to show the last 2 hours of data.

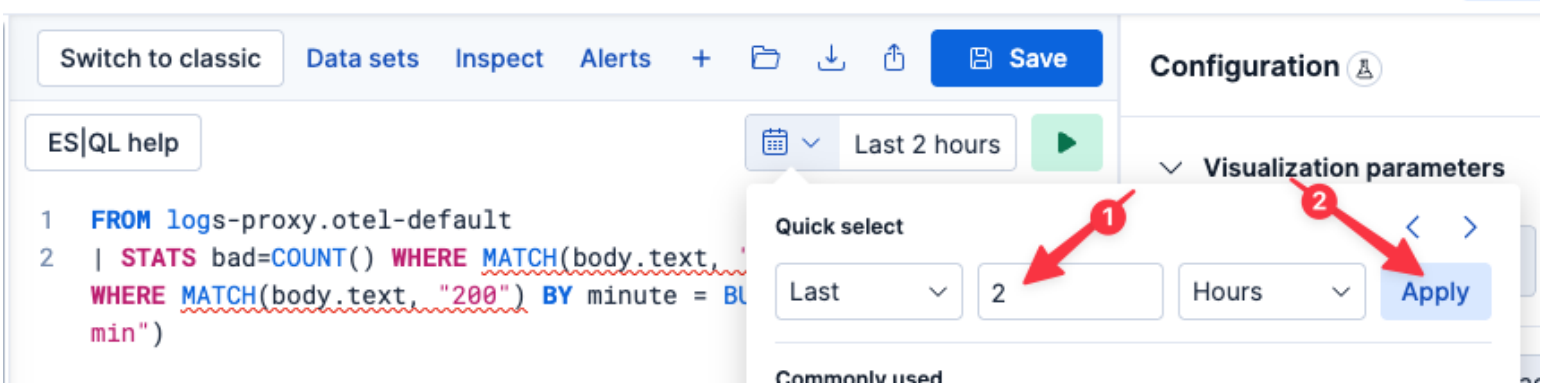


Figure 4: 1_time_field.png

Execute the following query:

```
FROM logs-proxy.otel-default
| EVAL status = CASE(body.text LIKE "* 500 *", "bad", "good") // label messages containing " 500 " as "bad", else "good"
| STATS COUNT() BY minute = BUCKET(@timestamp, "1 min"), status
```

Ok, it looks like this issue first started happening around 80 minutes ago. We can use `CHANGE_POINT` to narrow it down to a specific minute:

Execute the following query:

```
FROM logs-proxy.otel-default
| EVAL status = CASE(body.text LIKE "* 500 *", "bad", "good") // label messages containing " 500 " as "bad", else "good"
| STATS count = COUNT() BY minute = BUCKET(@timestamp, "1 min"), status
| CHANGE_POINT count ON minute AS type, pval // look for distribution change
| WHERE type IS NOT NULL
| KEEP type, minute
```

Let's take stock of what we know:

- a small percentage of users are experiencing 500 errors
- the errors started occurring around 80 minutes ago

Parsing with ES|QL

As you can see, simply searching for known error codes in our log lines will only get us so far. Maybe the error codes vary, or aren't specifically 500, but rather something in the 400 range?

Fortunately, nginx logs are semi-structured which makes them (relatively) easy to parse.

Some of you may be familiar with GROK expressions which provides a higher-level interface on top of regex; namely, GROK allows you define patterns. If you are well versed in GROK, you may be able to write a parsing pattern yourself for nginx logs, possibly using tools like GROK Debugger to help.

If you aren't well versed in GROK expressions, or you don't want to spend the time, you can leverage our AI Assistant to help! Click on the AI Assistant button in the upper-right and enter the following prompt:

```
can you write an ES|QL query to parse these nginx log lines?
```

> [!NOTE] The output should look something like the following. Notably, the AI Assistant may generate slightly different field names on each generating. Because we rely on those field names in subsequent analysis, please close the flyout and copy and paste the following ES|QL expression into the ES|QL query entry box.

```
FROM logs-proxy.otel-default
| GROK body.text "%{IPORHOST:client_ip} %{USER:ident} %{USER:auth} \\[%{HTTPDATE:timestamp}\\] \"%{WORD:http_method} %{NOTSPACE:request_path
| WHERE status_code IS NOT NULL
| EVAL @timestamp = DATE_PARSE("dd/MMM/yyyy:HH:mm:ss Z", timestamp)
| KEEP @timestamp, client_ip, http_method, request_path, status_code, user_agent
```

> [!NOTE] You'll note that our search has gotten a little slower when we added query-time GROK parsing. In our next challenge, we will show you how we can retain fast-search over long time windows WITH parsing using ingest-time parsing.

Is this affecting all APIs?

Let's make use of these parsed fields to break down `status_code` by `request_path` to see if this is affecting only a specific API?

Execute the following query:

```
FROM logs-proxy.otel-default
| GROK body.text "%{IPORHOST:client_ip} %{USER:ident} %{USER:auth} \\[%{HTTPDATE:timestamp}\\] \"%{WORD:http_method} %{NOTSPACE:request_path
| WHERE status_code IS NOT NULL
| STATS COUNT() BY status_code, request_path
```

Ok, it seems these errors are affecting all of our APIs.

Is this affecting all User Agents?

Ideally, we could also cross-reference the errors against the `user_agent` field to understand if it is affecting all browsers.

Execute the following query:

```
FROM logs-proxy.otel-default
| GROK body.text "%{IPORHOST:client_ip} %{USER:ident} %{USER:auth} \\[%{HTTPDATE:timestamp}\\] \"%{WORD:http_method} %{NOTSPACE:request_path
| WHERE status_code IS NOT NULL
| WHERE TO_INT(status_code) == 500
| STATS bad = COUNT() BY user_agent
```

Unfortunately, the unparsed `user_agent` field is too unstructured to really be useful for this kind of analysis. We could try to write a GROK expression to further parse `user_agent`, but in practice, it is too complicated (it requires translations and lookups in addition to parsing). Let's put a pin in this topic and revisit it in a bit when we have more tools at our disposal.

A better way to query

Let's redraw the time graph we drew before, but this time using `status_code` instead of looking for specific error codes.

Execute the following query:

```
FROM logs-proxy.otel-default
| GROK body.text "%{IPORHOST:client_ip} %{USER:ident} %{USER:auth} \\[%{HTTPDATE:timestamp}\\] \"%{WORD:http_method} %{NOTSPACE:request_path}
| WHERE status_code IS NOT NULL
| EVAL @timestamp = DATE_PARSE("dd/MMM/yyyy:HH:mm:ss Z", timestamp) // use embedded timestamp as record timestamp
| STATS status = COUNT() BY status_code, minute = BUCKET(@timestamp, "1 min")
```

> [!NOTE] If the resulting graph does not default to a bar graph plotted over time, click on the Pencil icon in the upper-right of the graph and change the graph type to `Bar`

This is a useful graph, and you can clearly see the advantage of parsing the log line vs. simply searching for specific error codes. Here, we can just generally graph by `status_code` and additionally split the data by, say, `request_path`.

This is a useful graph! Let's save it to a Dashboard for future use.

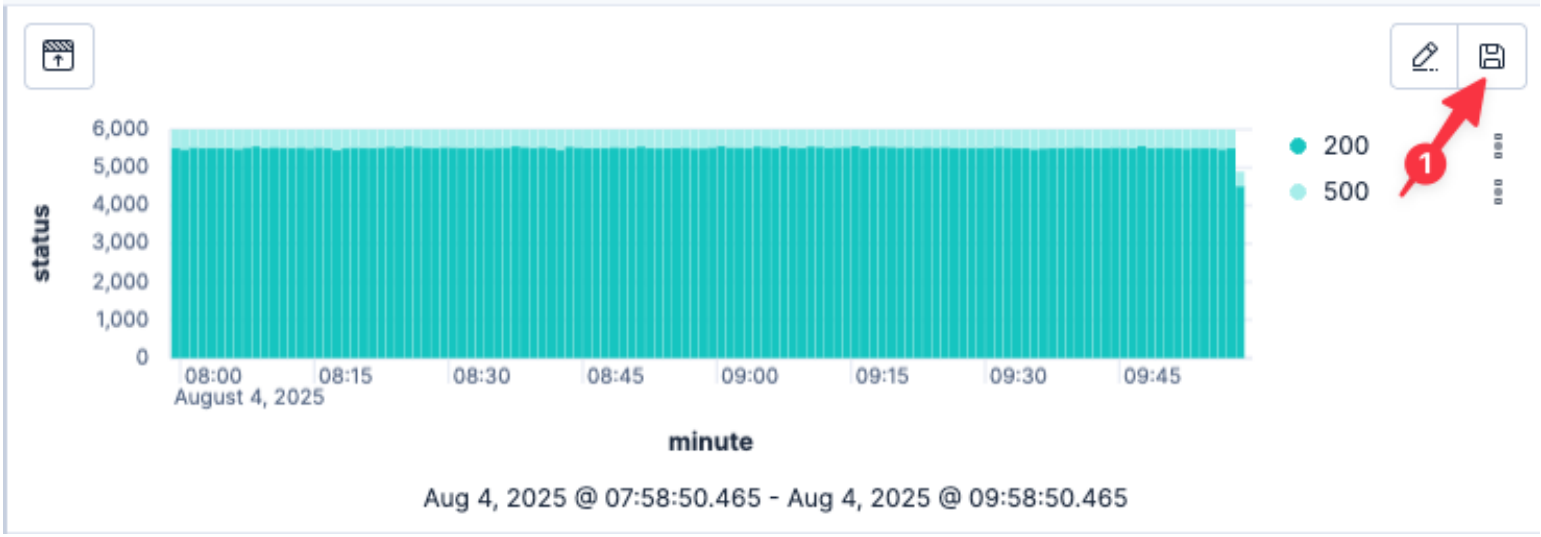


Figure 5: 1_save.png

1. Click on the Disk icon in the upper-left of the resulting graph
2. Name the visualization

```
Status Code Over Time (ESQL)
```

3. Select `New` under `Add to dashboard`
4. Click `Save and go to Dashboard`
5. Once the new dashboard has loaded, click the `Save` button in the upper-right
6. Enter the title of the new dashboard as

```
Ingress Proxy
```

7. Click `Save`

Setting up a simple alert

Navigate back to `Discover` using the left-hand navigation pane.

Let's create a simple alert to notify us whenever a `status_code` >= 400 is received:

Execute the following query:

```
FROM logs-proxy.otel-default
| GROK body.text "%{IPORHOST:client_ip} %{USER:ident} %{USER:auth} \\[%{HTTPDATE:timestamp}\\] \"%{WORD:http_method} %{NOTSPACE:request_path}
| WHERE status_code >= 400
```

1. Click Alerts in the taskbar
2. Select `Create search threshold rule`
3. Click `Test query`
4. Leave the defaults and click `Next`
5. Click `Next` on `Actions` tab
6. Set `Rule name` to

```
status_code >= 400
```

7. Set `Tags` to

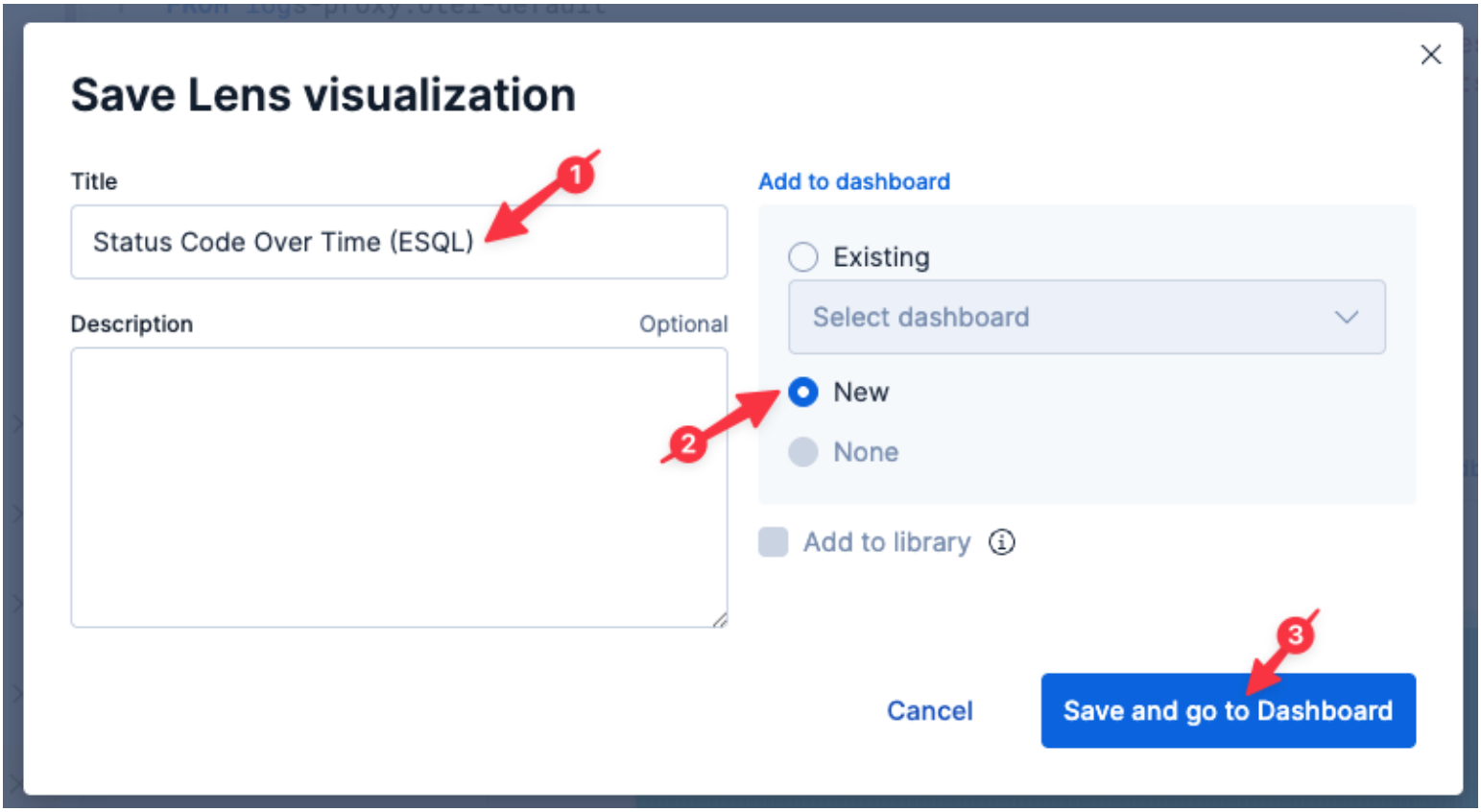


Figure 6: 1_dashboard.png

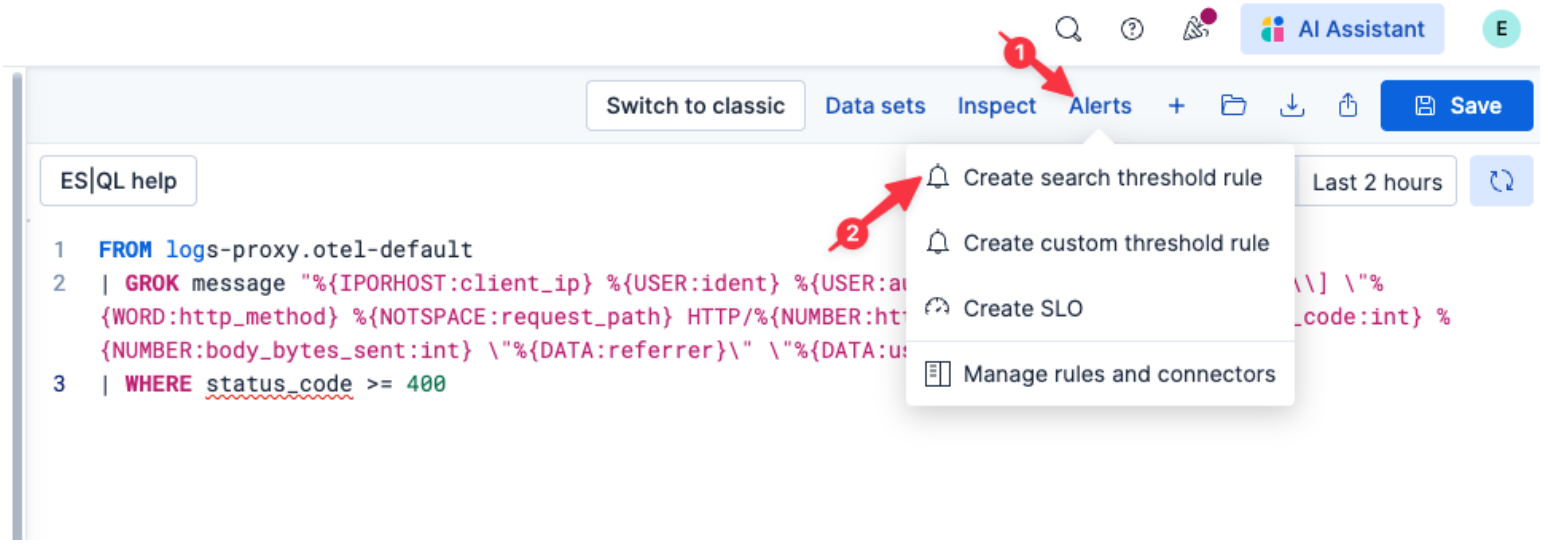


Figure 7: 1_alert.png

`ingress`

8. Click `Create rule` on `Details` tab

In practice, this alert is too simple. We probably are okay with a small percentage of non-200 errors for any large scale infrastructure. What we really want is to alert when we violate a SLO. We will come back to this in a bit.

Summary

Let's take stock of what we know:

- a small percentage of users are experiencing 500 errors
- the errors started occurring around 80 minutes ago
- the only error type seen is 500
- the errors occur over all APIs

And what we've done:

- Created a Dashboard showing ingress status
- Created a simple alert to let us know if we ever return non-200 error codes

---

So far, we've been using ES|QL to parse our proxy logs at query time. While incredibly powerful for quick analysis, we can do even more with our logs if we parse them at ingest-time.

Parsing with Streams

We will be working with the Elastic Streams interface which makes it easy to setup log parsing pipelines.

1. Select `logs-proxy.otel-default` from the list of data streams (if you start typing, Elasticsearch will help you find it)
2. Select the `Processing` tab
3. Click `Add a processor`
4. Select `Grok` for the `Processor` if not already selected
5. Set the `Field` to `body.text` if not already filled in
6. Click `Generate pattern`

Elastic will analyze your log lines and try to recognize a pattern.

The generated pattern should look similar to the following.

> [!NOTE] To ensure a consistent lab experience, please copy the following GROK expression and paste it into `Grok patterns`

```
%{IPV4:client.ip} - %{NOTSPACE:client.user} \[%{HTTPDATE:timestamp}\] "%{WORD:http.request.method} %{URIPATH:http.request.url.path} HTTP/%{NU
```

7. Click `Add processor`

The nginx log line includes a timestamp; let's use that as our record timestamp.

1. Click `Add a processor`
2. Select `Date`
3. Set `Field` to `timestamp`
4. Elastic should auto-recognize the format: `dd/MMM/yyyy:HH:mm:ss XX`
5. Click `Add processor`

Now save the Processing by clicking `Save changes`.

A faster way to query

Now let's jump back to Discover by clicking Discover in the left-hand navigation pane.

Execute the following query:

```
FROM logs-proxy.otel-default
| WHERE http.response.status_code IS NOT NULL
| KEEP @timestamp, client.ip, http.request.method, http.request.url.path, http.response.status_code, user_agent.original
```

> [!NOTE] If you get back `1,000 results` but the resulting columns are empty, remove the `Selected fields` (by clicking the `X` next to each), and then add each `Available field` (by clicking the `+` next to each).

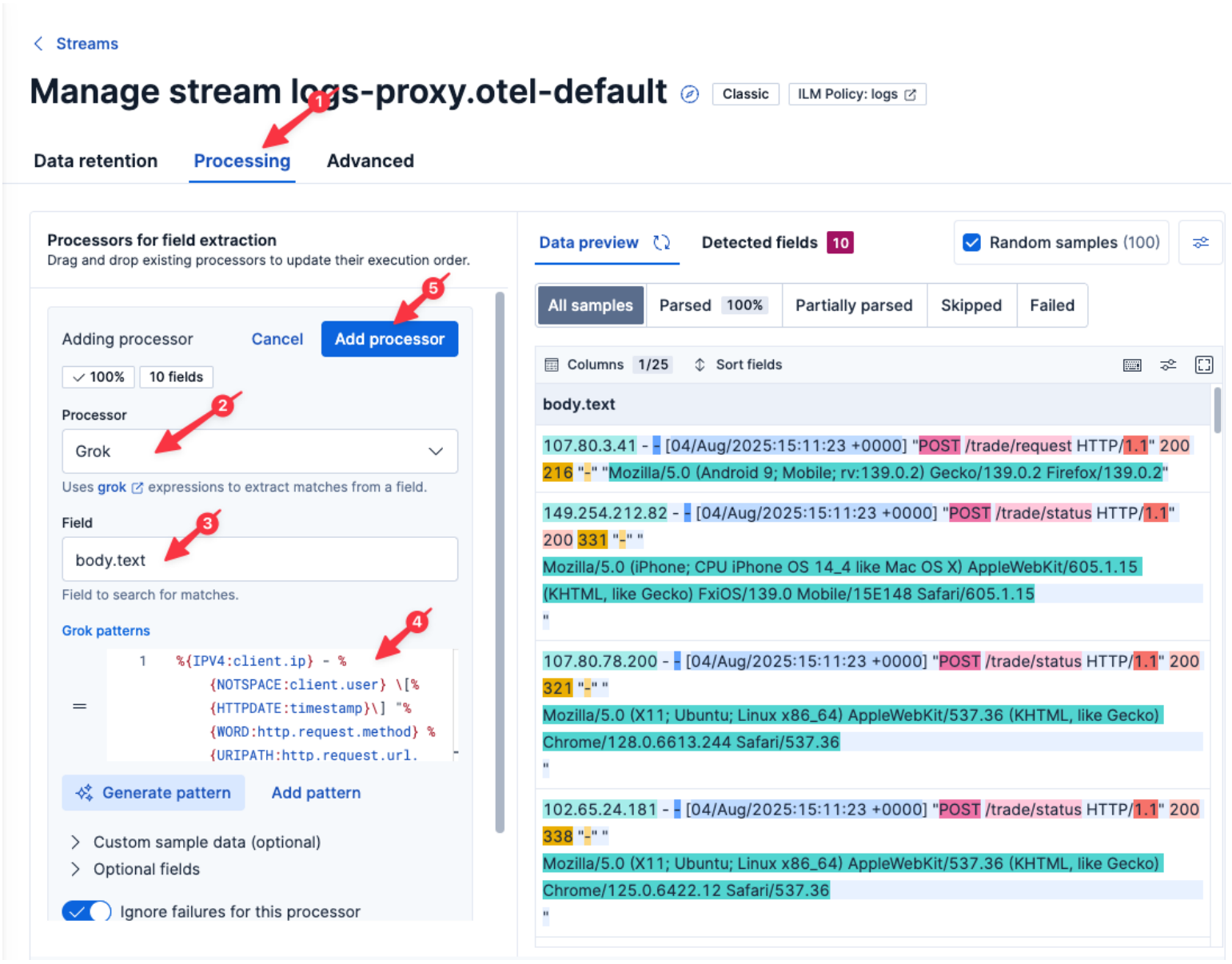


Figure 8: 2_grok.png

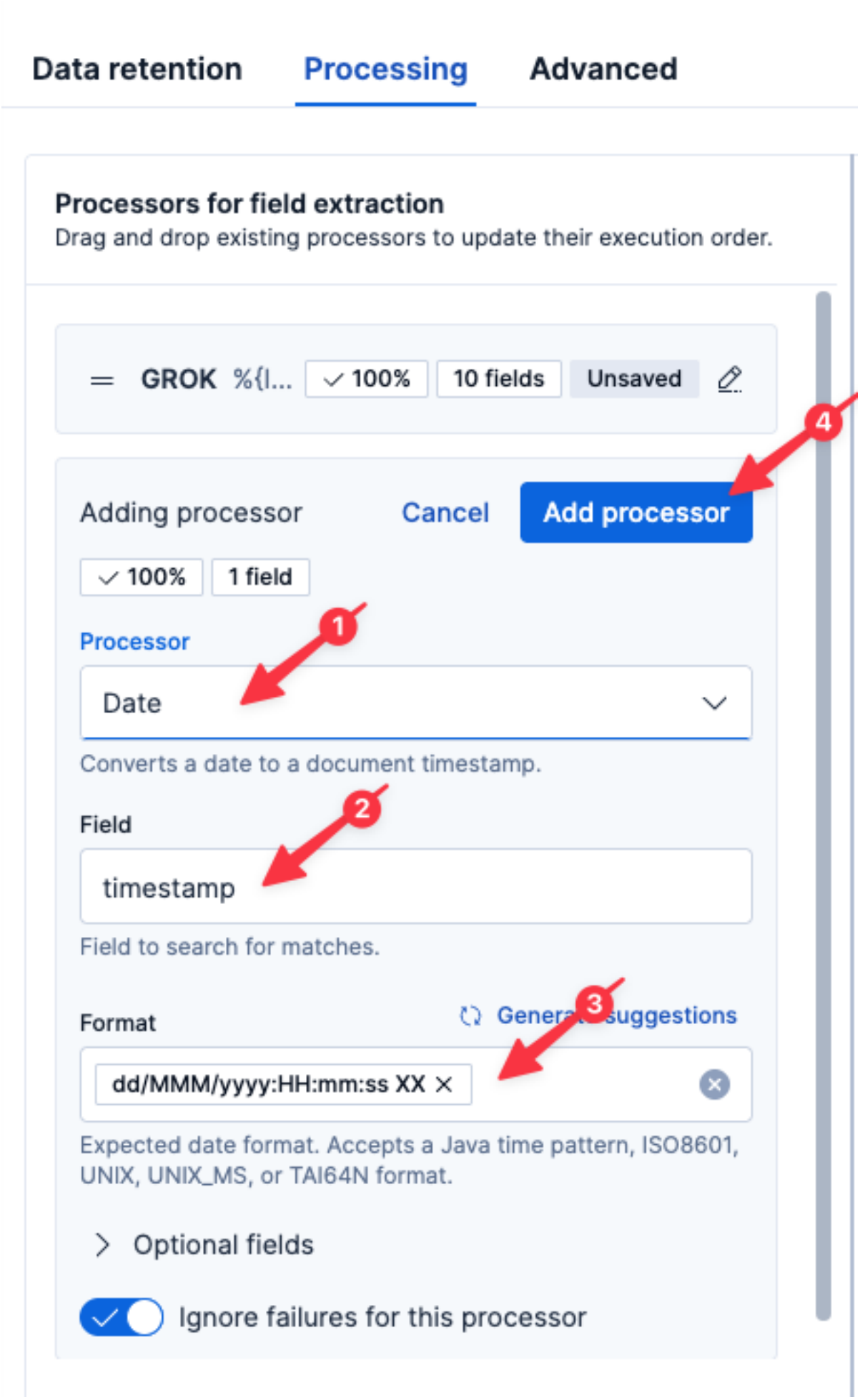


Figure 9: 2_date.png

Let's redraw our status code graph using our newly parsed field:

Execute the following query:

```
FROM logs-proxy.otel-default
| WHERE http.response.status_code IS NOT NULL
| STATS COUNT() BY TO_STRING(http.response.status_code), minute = BUCKET(@timestamp, "1 min")
```

Note that this graph, unlike the one we drew before, only has a few minutes of data. That is because it relies upon the fields we parsed in the Processing we just setup. Prior to that time, those fields didn't exist. Change the time field to `Last 15 Minutes` to see newly parsed data.

You'll also note how quickly this graph rendered compared to when we were parsing our log lines at query-time with ES|QL.

This is a useful graph! Let's save it to our Dashboard for future use.

1. Click on the Disk icon in the upper-left of the resulting graph
2. Name the visualization

```
Status Code Over Time (Streams)
```

3. Select `Existing` under `Add to dashboard`
4. Select the existing dashboard `Ingress Proxy`
5. Click `Save and go to Dashboard`
6. Once the dashboard has loaded, click the `Save` button in the upper-right

Creating a SLO

Remember that simple alert we created? Now that we are parsing these fields at ingest-time, we can create a proper SLO instead of a simple binary alert. With a SLO, we can allow for some percentage of errors over time (common in a complex system) before we get our support staff out of bed.

1. Click `SLOs` in the left-hand navigation
2. Click `Create SLO`
3. Select `Custom Query`
4. Set `Data view` to `logs-proxy.otel-default`
5. Set `Timestamp field` to `@timestamp`
6. Set `Good query` to `http.response.status_code < 400` (if this field isn't available, refresh the Instruqt virtual browser tab)
7. Set `Total query` to `http.response.status_code : *` (if this field isn't available, refresh the Instruqt virtual browser tab)
8. Set `Group by` to `http.request.url.path` (if this field isn't available, refresh the Instruqt virtual browser tab)
9. Set `SLO Name` to

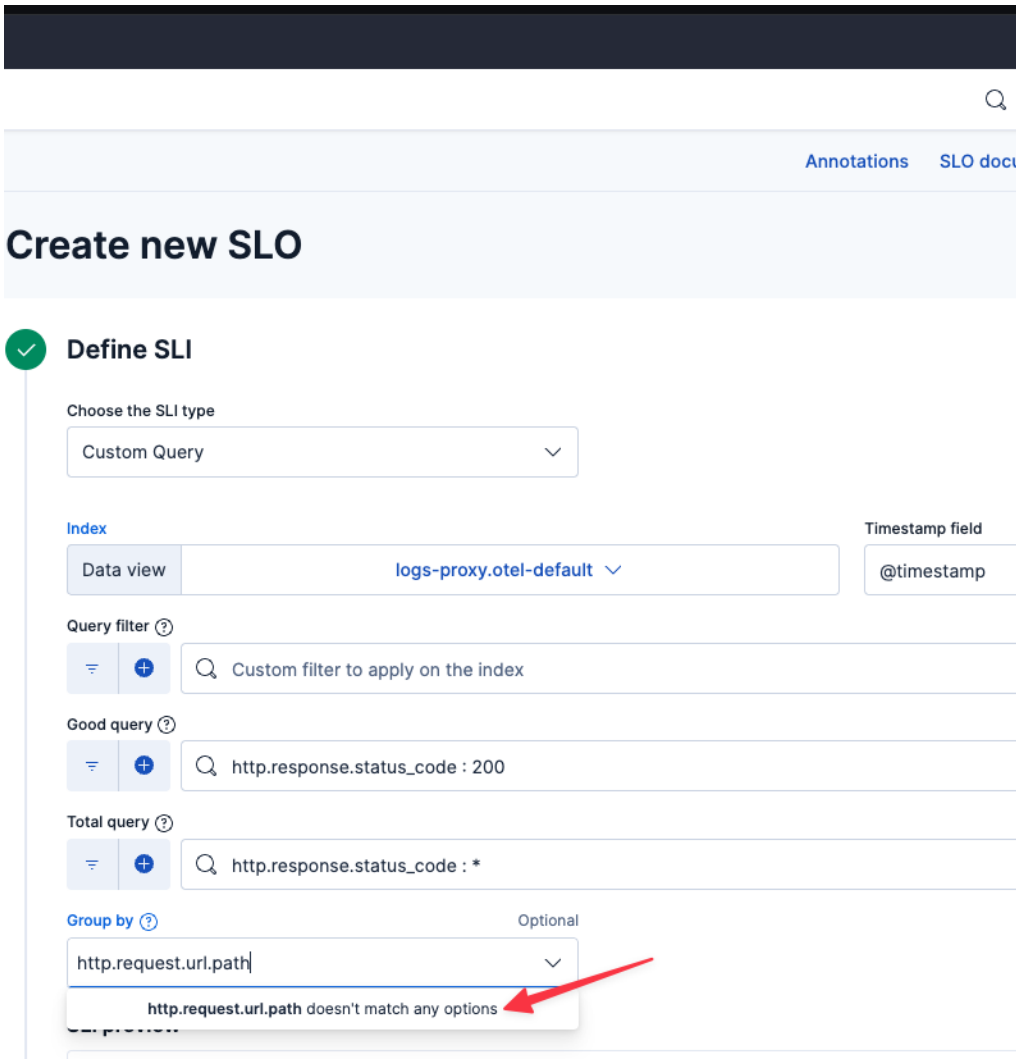
```
Ingress Status
```

10. Set `Tags` to

```
ingress
```

11. Click `Create SLO`

    [!NOTE] Because we are moving quickly, Elasticsearch may take some time to update field lists in the UI. If you encounter a situation where Elasticsearch doesn't recognize one of the fields we just parsed, click the Refresh icon in the upper-right of the



Instruqt tab and try again to create the SLO.

Now let's setup an alert that triggers when this SLO is breached.

1. Click on your newly created SLO `Ingress Status`
2. Under the `Actions` menu in the upper-right, select `Manage burn rate rule`

With burn rates, we can have Elastic dynamically adjust the escalation of a potential issue depending on how quickly it appears we will breach our SLO.

3. On the `Details` tab, set the `Rule name` to :

```
Ingress Status SLO
```

4. Set `Tags` to

```
ingress
```

5. Click `Save changes`
6. Click `Save rule` on the resulting pop-up

Now let's add the SLO monitor to our Dashboard.

1. Navigate to Dashboards and open `Ingress Status`
2. Click `Add panel`
3. Select `SLO Overview`
4. Select `Grouped SLOs`
5. Set `Group by` to `Tags`
6. Set `Tags` to `ingress`
7. Click `Save`

Let's also add our growing list of Alerts to our Dashboard.

1. Click `Add panel`
2. Select `Alerts`
3. Set `Filter by` to `Rule tags`
4. Set `Rule tags` to `ingress`

5. Click `Save`

Summary

Let's take stock of what we know:

- a small percentage of users are experiencing 500 errors
- the errors started occurring around 80 minutes ago
- the only error type seen is 500
- the errors occur over all APIs

And what we've done:

- Created a Dashboard showing ingress status
- Created a simple alert to let us know if we ever return non-200 error codes
- Parsed the logs for quicker and more powerful analysis
- Create a SLO to let us know if we ever return non-200 error codes over time

---

We still don't know why some requests are failing. Now that we are parsing the logs, however, we have access to a lot more information.

Is this affecting every region?

Let's analyze our clients by `client.ip` to look for possibly geographic patterns. We can easily do that with the Elastic `GeoIP` processor.

1. Select `logs-proxy.otel-default` from the list of Streams.
2. Select the `Processing` tab
3. Click `Add a processor`
4. Select `GeoIP`
5. Set the `Field` to

```
client.ip
```

6. Open `Optional fields` and set `Target field` to

```
client.geo
```

7. Set `Ignore missing` to true
8. Click `Add processor`
9. Click `Save changes`

Streams

Manage stream logs-proxy.otel-d

Data retention | Processing | Advanced

Processors for field extraction

Drag and drop existing processors to update their execution order.

DATE timestamp • dd/MMM/yyyy:HH:m...

Adding processor | Cancel | Add processor

✓ 100% | 8 fields

Processor

1

GeoIP

Adds information about the geographical location of an IPv4 or IPv6 address.

Field

2

client.ip

3

The field to get the IP address from for the geographical lookup.

Optional fields

Target field

4

client.geo

The field that will hold the geographical information looked up from the database.

Data

All s

clien

186.

101.

186.

107.

149.

149.

149.

102.

101.

149.

Mana

Data reter

Processo

Drag and d

GR

DA

Adding

100

Processo

GeoIP

Adds info

or IPv6 a

Field

client.

The field

lookup.

Opt

Ignore do

Let's jump back to Discover by clicking Discover in the left-hand navigation pane.

Adjust the time field to show the last 2 hours of data.

Execute the following query:

```
FROM logs-proxy.otel-default
| WHERE client.geo.country_iso_code IS NOT NULL AND http.response.status_code IS NOT NULL
| STATS COUNT() BY http.response.status_code, client.geo.country_iso_code
| SORT http.response.status_code DESC
```

Let's make this a pie chart to allow for more intuitive visualization.

1. Click the pencil icon to the right of the graph
2. Select `Pie` from the dropdown menu

So it looks like all of our 500 errors are contained in the `TH` (Thailand) region. That is interesting, and without more information, we might be tempted to stop our RCA analysis here. There is always more to the story, as you will see.

In the meantime, this is a useful graph! Let's save it to a Dashboard for future use.

1. Click on the Disk icon in the upper-left of the resulting graph
2. Name the visualization

```
Status by Region
```

3. Select `Existing` under `Add to dashboard`
4. Select the existing dashboard `Ingress Proxy`
5. Click `Save and go to Dashboard`
6. Once the dashboard has loaded, click the `Save` button in the upper-right

Visualizing with Maps

Sometimes it is helpful to visualize geography on a map. Fortunately, Elastic has a built-in Map visualization we can readily use!

1. Navigate to `Other tools` > `Maps`
2. Click `Add layer`
3. Select `Elasticsearch`
4. Select `Documents`
5. Select `Data view` to `logs-proxy.otel-default`
6. Set `Geospatial field` to `client.geo.location` (if this field isn't available, refresh the Instruqt virtual browser tab)
7. Click `Add and continue`
8. Scroll down to `Layer style`
9. Set `Fill color` to `By value`
10. Set `Select a field` to `http.response.status_code`
11. Set `As number` to `As category`
12. Set `Symbol Size` to `By value`
13. Set `Select a field` to `http.response.status_code`
14. Click `Keep changes`
15. Click `Save`
16. Name the Map

```
Status Code by Location
```

15. Select existing dashboard `Ingress Status`
16. Click `Save and go to dashboard`
17. Once the dashboard has loaded, click the `Save` button in the upper-right

    [!NOTE] Because we are moving quickly, Elasticsearch may take some time to update field lists in the UI. If you encounter a situation where Elasticsearch doesn't recognize one of the fields we just parsed, click the Refresh icon in the upper-right of the Instruqt tab and try again to create the Map.

Summary

Let's take stock of what we know:

- a small percentage of users are experiencing 500 errors
- the errors started occurring around 80 minutes ago
- the only error type seen is 500
- the errors occur over all APIs

- the errors occur only in the `TH` (Thailand) region

And what we've done:

- Created a Dashboard showing ingress status
- Created a simple alert to let us know if we ever return non-200 error codes
- Parsed the logs for quicker and more powerful analysis
- Create a SLO to let us know if we ever return non-200 error codes over time
- Created a Map to help us visually geo-locate the errors

We know that errors appear to be localized to a specific region. But maybe there is more to the story?

Is this affecting every browser type?

Let's parse that User Agent string to look for correlation. While difficult/impossible with a simple GROK expression, you can easily do this with the Elastic `User agent` processor.

1. Select `logs-proxy.otel-default` from the list of Streams.
2. Select the `Processing` tab
3. Click `Add a processor`
4. Select `User agent`
5. Set the `Field` to

`user_agent.original`

6. Set `Ignore missing` to true
7. Click `Add processor`

In addition to the fields produced by the User Agent processor, we also want a simplified combination of browser name and version. We can easily craft one using the Set processor.

1. Click `Add a processor`
2. Click `Set`
3. Set `Field` to

`user_agent.full`

4. Set `Value` to

`{{user_agent.name}} {{user_agent.version}}`

5. Click `Ignore failures for this processor`
6. Click `Add processor`
7. Click `Save changes`

Now let's jump back to Discover by clicking Discover in the left-hand navigation pane.

Execute the following query:

```
FROM logs-proxy.otel-default
| WHERE user_agent.full IS NOT NULL
| STATS good = COUNT(http.response.status_code < 400 OR NULL), bad = COUNT(http.response.status_code >= 400 OR NULL) BY user_agent.full
| SORT bad DESC
```

Ah-ha, there is more to the story! It appears our errors may be isolated to a specific browser version. Let's break this down by `user_agent.version`.

```
FROM logs-proxy.otel-default
| WHERE user_agent.full IS NOT NULL
| STATS good = COUNT(http.response.status_code < 400 OR NULL), bad = COUNT(http.response.status_code >= 400 OR NULL) BY user_agent.version
| SORT bad DESC
```

Indeed, it appears we might have a problem with version 136 of the Chrome browser!

Correlating with region

So what's the correlation with the geographic area we previously saw?

Execute the following query:

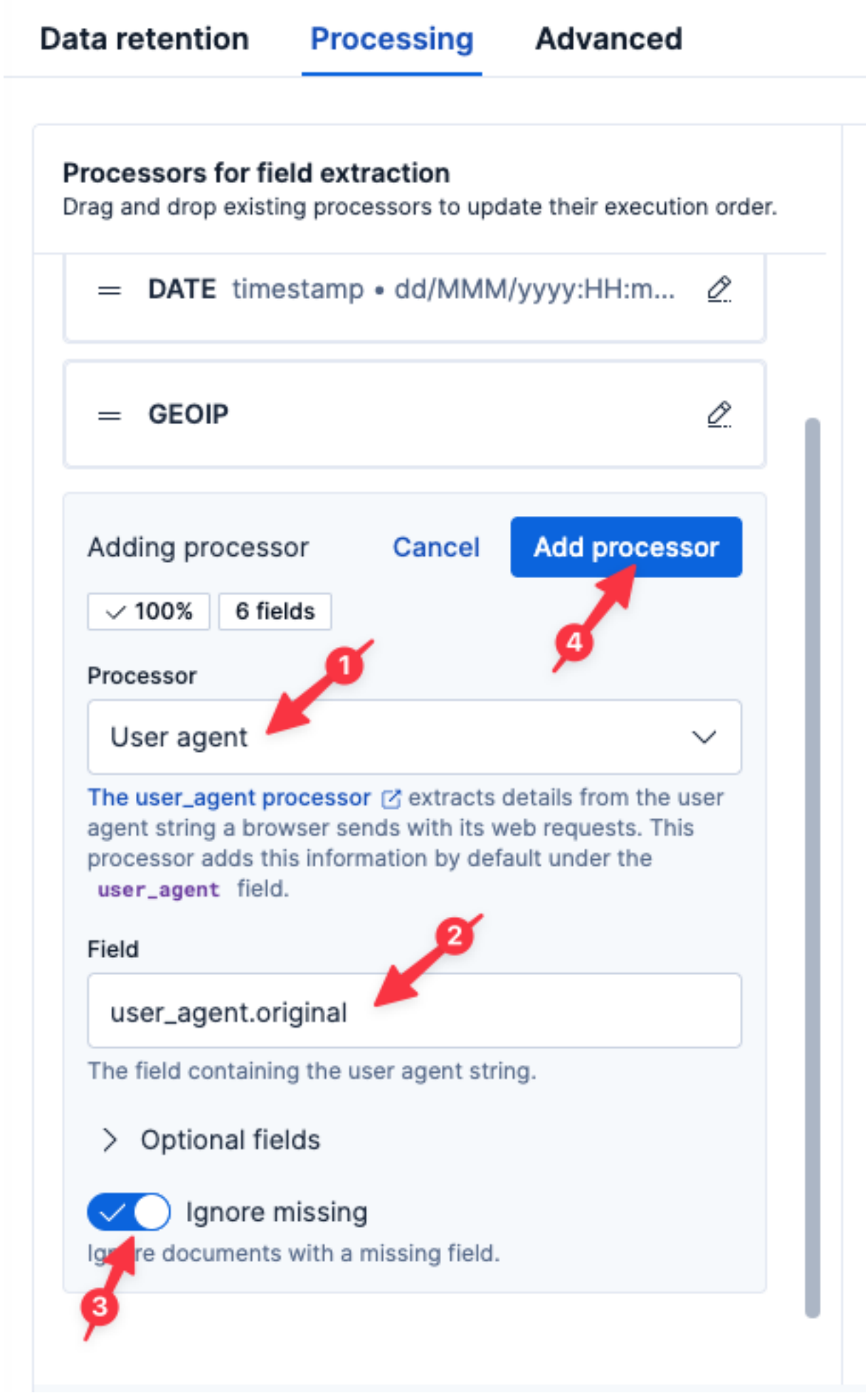


Figure 10: 4 ua1.png

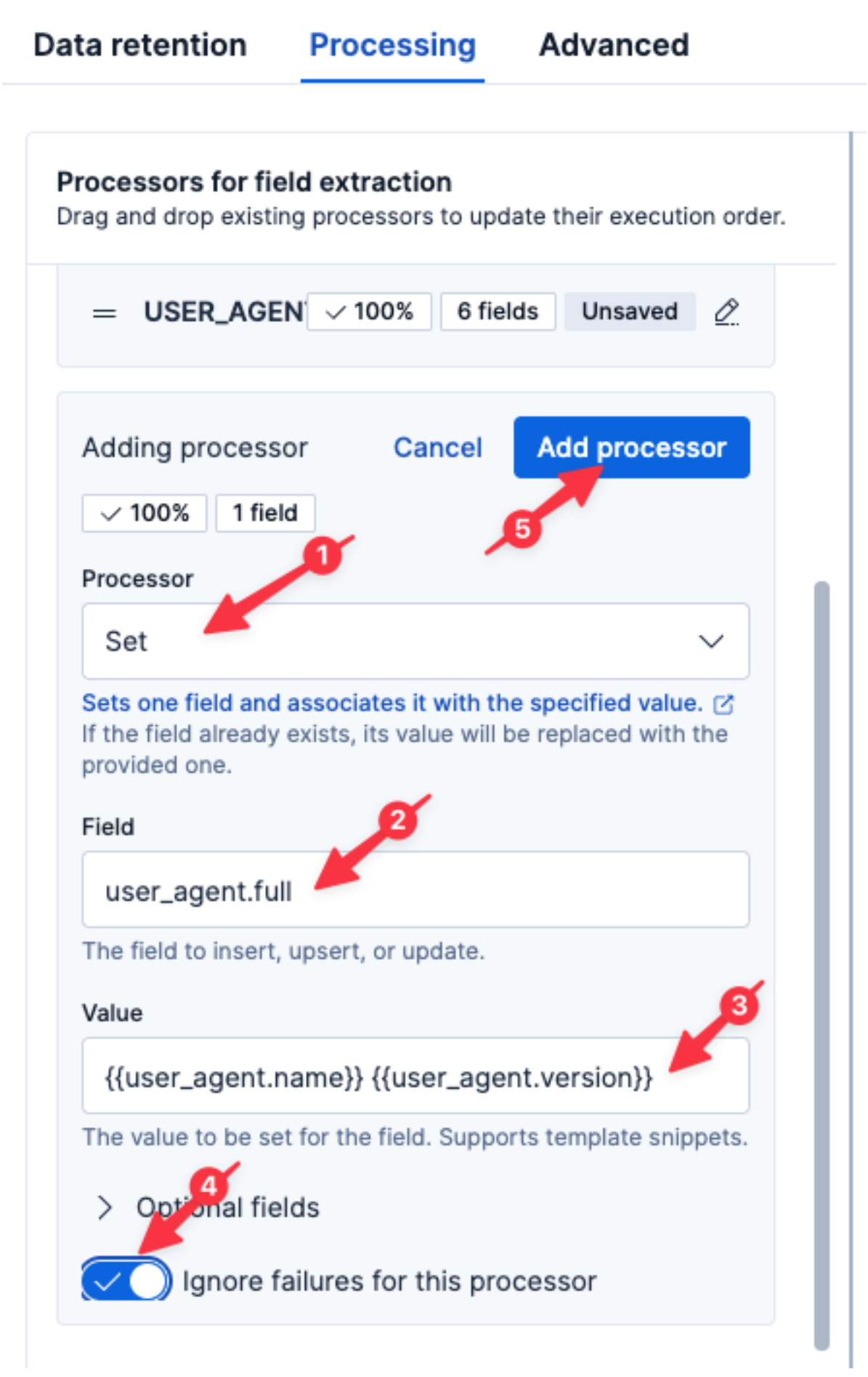


Figure 11: 4_ua2.png

```
FROM logs-proxy.otel-default
| WHERE client.geo.country_iso_code IS NOT NULL AND user_agent.version IS NOT NULL AND http.response.status_code IS NOT NULL
| EVAL version_major = SUBSTRING(user_agent.version,0,LOCATE(user_agent.version, ".")-1)
| WHERE user_agent.name == "Chrome"
| WHERE TO_INT(version_major) == 136
| STATS COUNT() BY client.geo.country_iso_code
```

Ah! It appears that this specific version of the Chrome browser has only been seen in the `TH` region! Quite possibly, Google has rolled out a specialized or canary version of their browser first in the `TH` region.

Congratulations! We found our problem! In the next challenge, we will setup a way to catch new User Agents in the future.

Summary

Let's take stock of what we know:

- a small percentage of users are experiencing 500 errors
- the errors started occurring around 80 minutes ago
- the only error type seen is 500
- the errors occur over all APIs
- the errors occur only in the `TH` region
- the errors occur only with browsers based on Chrome v136

And what we've done:

- Created a Dashboard showing ingress status
- Created a simple alert to let us know if we ever return non-200 error codes
- Parsed the logs for quicker and more powerful analysis
- Create a SLO to let us know if we ever return non-200 error codes over time
- Created a Map to help us visually geo-locate the errors

Now that we know what happened, let's try to be sure this never happens again.

Generating a breakdown of user agents

In general, it would be nice to have a graphical breakdown of the makeup of our clients.

We can accomplish this using our parsed User Agent string and ES|QL.

Execute the following query:

```
FROM logs-proxy.otel-default
| WHERE user_agent.os.name IS NOT NULL
| STATS COUNT() by user_agent.os.name, user_agent.os.version
```

1. Click on the pencil icon to the right of the existing graph
2. Select `Treemap` from the visualizations drop-down menu
3. Click `Apply and close`

Let's save it to our Dashboard for future use.

1. Click on the Disk icon in the upper-left of the resulting graph
2. Name the visualization

```
Client OSs
```

3. Select `Existing` under `Add to dashboard`
4. Select the existing dashboard `Ingress Proxy`
5. Click `Save and go to Dashboard`
6. Once the dashboard has loaded, click the `Save` button in the upper-right

Let's also create a chart depicting the overall breakdown of Browsers.

Navigate back to Discover.

Execute the following query:

```
FROM logs-proxy.otel-default
| WHERE user_agent.name IS NOT NULL
| STATS COUNT() by user_agent.name
```

1. Click on the pencil icon to the right of the existing graph

2. Select `Pie` from the visualizations drop-down menu
3. Click `Apply and close`

Let's save it to our Dashboard for future use.

1. Click on the Disk icon in the upper-left of the resulting graph
2. Name the visualization

```
Client Browsers
```

3. Select `Existing` under `Add to dashboard`
4. Select the existing dashboard `Ingress Proxy`
5. Click `Save and go to Dashboard`
6. Once the dashboard has loaded, click the `Save` button in the upper-right

Generating a table of user agents

It would also be helpful is to keep track of new User Agents as they appear in the wild.

Execute the following query:

```
FROM logs-proxy.otel-default
| WHERE user_agent.full IS NOT NULL
| STATS @timestamp.min = MIN(@timestamp), @timestamp.max = MAX(@timestamp) BY user_agent.full
```

This is good, but it would also be helpful, based on what we saw, to know the first country that a given User Agent appeared in.

Execute the following query:

```
FROM logs-proxy.otel-default
| WHERE user_agent.full IS NOT NULL
| STATS @timestamp.min = MIN(@timestamp), @timestamp.max = MAX(@timestamp) BY user_agent.full, client.geo.country_iso_code
| SORT @timestamp.min ASC // sort first seen to last seen
| STATS first_country_iso_code = TOP(client.geo.country_iso_code , 1, "asc"), first_seen = MIN(@timestamp.min), last_seen = MAX(@timestamp.ma
| SORT user_agent.full, first_seen, last_seen, first_country_iso_code
```

Fabulous! Now we can see every User Agent we encounter, when we first encountered it, and in what region it was first seen.

Say you also wanted to know when a given User Agent was released by the developer?

We could try to maintain our own User Agent lookup table and use ES|QL `LOOKUP JOIN`s to match browser versions to release dates:

Execute the following query:

```
FROM ua_lookup
```

We built this table by hand; it is far from comprehensive. Now let's use `LOOKUP JOIN` to do a real-time lookup for each row:

Execute the following query:

```
FROM logs-proxy.otel-default
| WHERE user_agent.full IS NOT NULL
| EVAL user_agent.name_and_vmajor = SUBSTRING(user_agent.full, 0, LOCATE(user_agent.full, ".")-1)
| STATS @timestamp.min = MIN(@timestamp), @timestamp.max = MAX(@timestamp) BY user_agent.name_and_vmajor, client.geo.country_iso_code
| SORT @timestamp.min ASC // sort first seen to last seen
| STATS first_country_iso_code = TOP(client.geo.country_iso_code , 1, "asc"), first_seen = MIN(@timestamp.min), last_seen = MAX(@timestamp.ma
| SORT user_agent.name_and_vmajor, first_seen, last_seen, first_country_iso_code
| LOOKUP JOIN ua_lookup ON user_agent.name_and_vmajor
| KEEP release_date, user_agent.name_and_vmajor, first_country_iso_code, first_seen, last_seen
```

We can quickly see the problem with maintaining our own `ua_lookup` index. It would take a lot of work to truly track the release date of every Browser version in the wild.

Fortunately, Elastic makes it possible to leverage an external Large Language Model (LLM) to lookup those browser release dates for us!

Execute the following query:

```
FROM logs-proxy.otel-default
| WHERE user_agent.full IS NOT NULL
| STATS @timestamp.min = MIN(@timestamp), @timestamp.max = MAX(@timestamp) BY user_agent.full, client.geo.country_iso_code
| SORT @timestamp.min ASC // sort first seen to last seen
| STATS first_country_iso_code = TOP(client.geo.country_iso_code , 1, "asc"), first_seen = MIN(@timestamp.min), last_seen = MAX(@timestamp.ma
| SORT first_seen DESC
| LIMIT 10 // intentionally limit to top 10 first_seen to limit LLM completions
| EVAL prompt = CONCAT(
   "when did this version of this browser come out? output only a version of the format mm/dd/yyyy",
   "browser: ", user_agent.full
  ) | COMPLETION release_date = prompt WITH openai_completion
```

```
| EVAL release_date = DATE_PARSE("MM/dd/YYYY", release_date)
| KEEP release_date, first_country_iso_code, user_agent.full, first_seen, last_seen
```

> [!NOTE] If this encounters a timeout, try executing the query again.

You'll note that we are limiting our results to only the top 10 last seen User Agents. This is intentional to limit the number of `COMPLETION` commands executed, as each one will result in a call to our configured external Large Language Model (LLM). Notably, the use of the `COMPLETION` command is in Tech Preview; future revisions of ES|QL may include a means to more practically scale the use of the `COMPLETION` command.

Let's save this search for future reference:

1. Click `Save`
2. Set `Title` to

```
ua_release_dates
```

Now let's add this as a table to our dashboard

1. Navigate to Dashboards and open `Ingress Status`
2. Click `Add from library`
3. Find `ua_release_dates`
4. Click `Save`

Scheduling a report

The CIO is concerned about us not testing new browsers sufficiently, and for some time wants a nightly report of our dashboard. No problem!

1. Click `Download` icon
2. Select `Schedule exports`
3. Click `Schedule exports`

Alert when a new UA is seen

Ideally, we can send an alert whenever a new User Agent is seen. To do that, we need to keep state of what User Agents we've already seen. Fortunately, Elastic Transforms makes this easy!

Create transform: 1. Navigate to `Management` > `Stack Management` > `Transforms` 2. Click `Create a transform` 3. Select `logs-proxy.otel-default` 4. Select `Pivot` 5. Set `Search filter` to `user_agent.full :*` (if this field isn't available, refresh the Instruqt virtual browser tab) 5. Set `Group by` to `terms(user_agent.full)` 6. Add an aggregation for `@timestamp.max` 7. Add an aggregation for `@timestamp.min` 8. Click `> Next` 9. Set the `Transform ID` to `user_agents` 10. Set `Time field` to `@timestamp.min` 11. Set `Continuous mode` 12. Open `Advanced settings` and set the Frequency to `5s` 13. Click `Next` 14. Click `Create and start`

> [!NOTE] Because we are moving quickly, Elasticsearch may take some time to update field lists in the UI. If you encounter a situation where Elasticsearch doesn't recognize one of the fields we just parsed, click the Refresh icon in the upper-right of the Instruqt tab and try again to create the Map.

Let's create a new alert which will fire whenever a new User Agent is seen.

1. Navigate to `Alerts`
2. Click `Manage Rules`
3. Click `Create Rule`
4. Select `Custom threshold`
5. Set `DATA VIEW` to `user_agents`
6. Set `IS ABOVE` to `1`
7. Set `FOR THE LAST` to `5 minutes`
8. Set `Rule schedule` to `5 seconds`
9. Set `Rule name` to `New UA Detected`
10. Set `Tags` to

```
ingress
```

11. Set `Related dashboards` to `Ingress Proxy`
12. Click `Create rule`

Let's test it

1. Navigate to the button label="Terminal" tab

2. Run the following command:

```
curl -X POST http://kubernetes-vm:32003/err/browser/chrome
```

This will create a new Chrome UA 137. Let's go to our dashboard and see if we can spot it.

1. Navigate to the button label="Elasticsearch" tab
2. Navigate to Dashboards
3. Select `Ingress Proxy`

Look at the table of UAs that we added and note the addition of Chrome 137! You'll also note a new active alert `New UA Detected`!

Summary

Let's take stock of what we know:

- a small percentage of users are experiencing 500 errors
- the errors started occurring around 80 minutes ago
- the only error type seen is 500
- the errors occur over all APIs
- the errors occur only in the `TH` region
- the errors occur only with browsers based on Chrome v136

And what we've done:

- Created a Dashboard showing ingress status
- Created a simple alert to let us know if we ever return non-200 error codes
- Parsed the logs for quicker and more powerful analysis
- Create a SLO to let us know if we ever return non-200 error codes over time
- Created a Pie Graph showing errors by region
- Created a Map to help us visually geo-locate the errors
- Created graphs in our dashboard showing the breakdown of User Agents
- Created a table in our dashboard iterating seen User Agents
- Created a nightly report to snapshot our Dashboard
- Created an alert to let us know when a new User Agent string appears _____

Sometimes our data contains PII information which needs to be restricted to a need-to-know basis and kept only for a limited time.

Limiting access

With Elastic's in-built support for RBAC, we can limit access at the index, document, or field level.

In this example, we've created a limited_user with a limited_role which restricts access to the `client.ip` and `body.text` fields (to avoid leaking the `client.ip`).

In the Elasticsearch tab, we are logged in as a user with full privileges. Let's check our access. 1. Open the button label="Elasticsearch" tab 2. Open a log record and click on the `Table` tab in the flyout 3. Note access to the `client.ip` and `body.text` fields

In the Elasticsearch (Limited) tab, we are logged in as a user with full privileges. Let's check our access.

1. Open the button label="Elasticsearch (Limited)" tab
2. Open a log record and click on the `Table` tab in the flyout
3. Note that `client.ip` and `body.text` fields don't exist

Let's change permissions and see what happens:

1. Open the button label="Elasticsearch" tab
2. Navigate to `Management` > `Stack Management` > `Security` > `Roles`
3. Select `limited_viewer`
4. For Indices `logs-proxy.otel-default` click `Grant access to specific fields`
5. Update `Denied fields` to be only `client.ip`, but remove `body.text`
6. Click `Update role`
7. Open the button label="Elasticsearch (Limited)" tab
8. Close the open log record flyout
9. Run the search query again

10. Open a log record
11. Note that `client.ip` doesn't exist, but `body.text` now does!

Limiting retention

Say your records department requires you to keep these logs generally accessible only for a very specific period of time. We can ask Elasticsearch to automatically delete them after some number of days.

1. Open the button label="Elasticsearch" tab
2. Navigate to `Streams`
3. Select `logs-proxy.otel-default` from the list of Streams
4. Click on the `Data retention` tab
5. Click `Edit data retention`
6. Select `Set specific retention days`
7. Set to `30` days

Elasticsearch will now remove this data from its online indices after 30 days. At that time, it will only be available in backups.

Summary

Let's take stock of what we know:

- a small percentage of users are experiencing 500 errors
- the errors started occurring around 80 minutes ago
- the only error type seen is 500
- the errors occur over all APIs
- the errors occur only in the `TH` region
- the errors occur only with browsers based on Chrome v136

And what we've done:

- Created a Dashboard showing ingress status
- Created a simple alert to let us know if we ever return non-200 error codes
- Parsed the logs for quicker and more powerful analysis
- Create a SLO to let us know if we ever return non-200 error codes over time
- Created a Pie Graph showing errors by region
- Created a Map to help us visually geo-locate the errors
- Created graphs in our dashboard showing the breakdown of User Agents
- Created a table in our dashboard iterating seen User Agents
- Created a nightly report to snapshot our Dashboard
- Created an alert to let us know when a new User Agent string appears
- Setup RBAC to restrict access to `client.ip`
- Setup retention to keep the logs online for only 30 days

Wrap-Up

Over the course of this lab, we learned about:

- Using ES|QL to search logs
- Using ES|QL to parse logs at query-time
- Using ES|QL to do advanced aggregations, analytics, and visualizations
- Creating a dashboard
- Using ES|QL to create Alerts
- Using AI Assistant to help write ES|QL queries
- Using Streams to setup ingest-time log processing pipeline (GROK parsing, geo-location, User Agent parsing)
- Setting up SLOs
- Using Maps to visualize geographic information
- Scheduling dashboard reports
- Setting up a Pivot Transform
- Setting up RBAC
- Setting up data retention

We put these technologies to use in a practical workflow which quickly took us from an unknown problem to a definitive Root Cause. Furthermore, we've setup alerts to ensure we aren't caught off-guard in the future. Finally, we built a really nice custom Dashboard to

help us monitor the health of our Ingress Proxy.

All of this from just a lowly nginx access file. That's the power unlocked by Elastic. ____