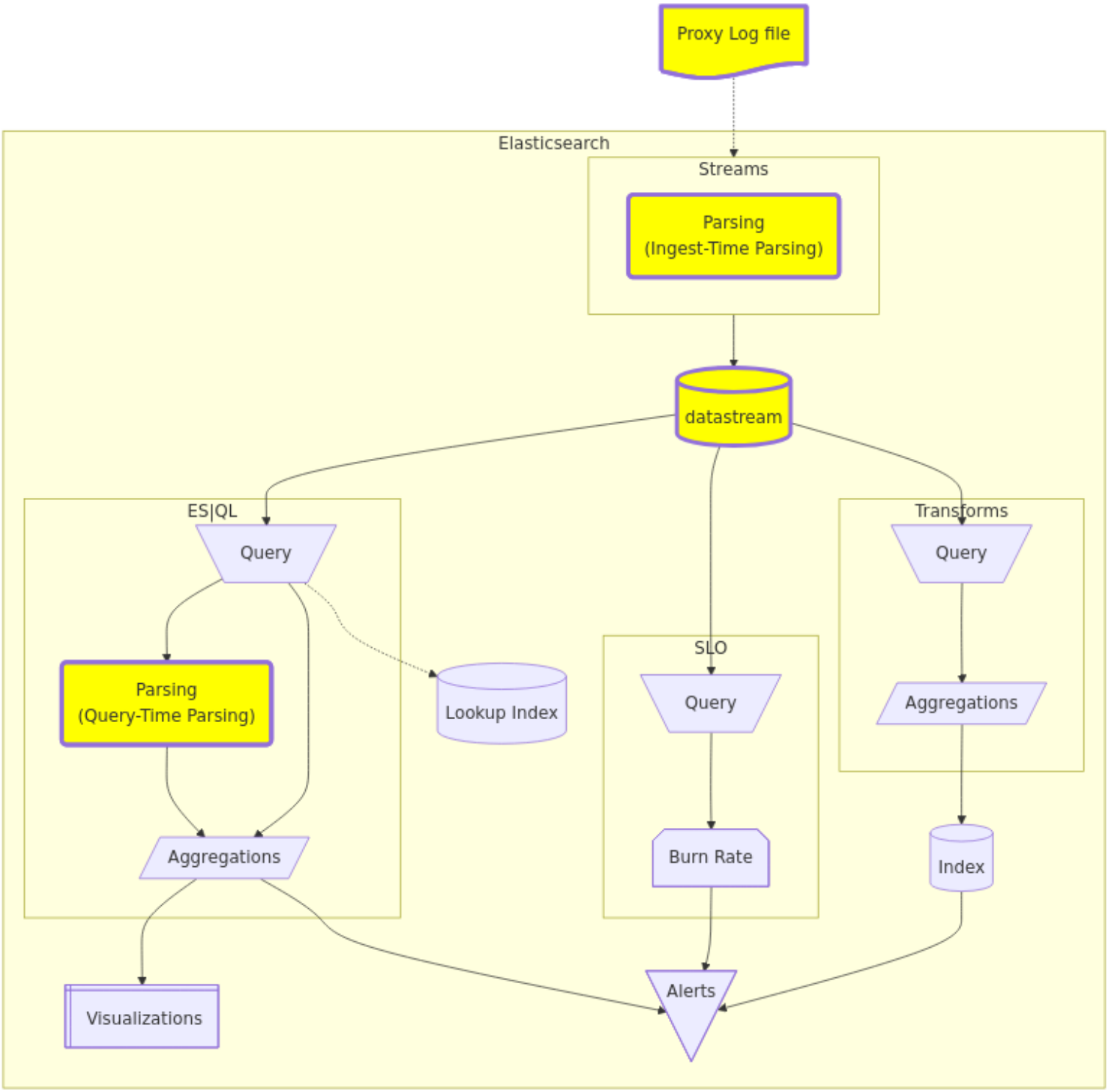
Observability 100: A Modern Logging Workflow

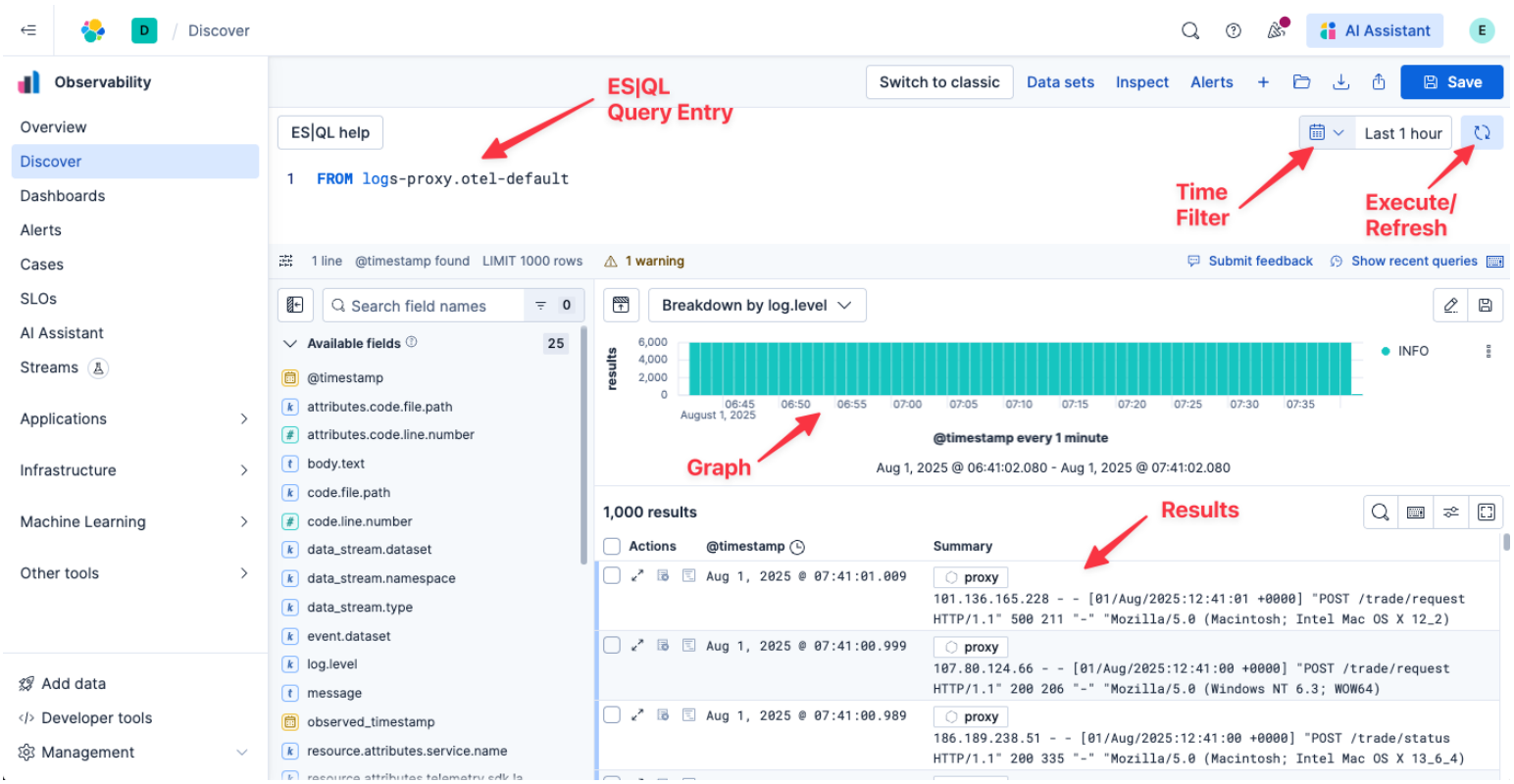
We've gotten word from our customer service department that some users are receiving an error when trying to use our web-based application. We will use this workshop to showcase Elastic's state-of-the-art logging workflow to get to the root cause of these errors.

Ingest vs. query-time parsing

Throughout this workshop, we will 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 workflows. Moving parsing to ingest-time also facilitates faster search results. Regardless of where the parsing is done, we will leverage ES|QL to perform aggregations, analysis, and visualization.

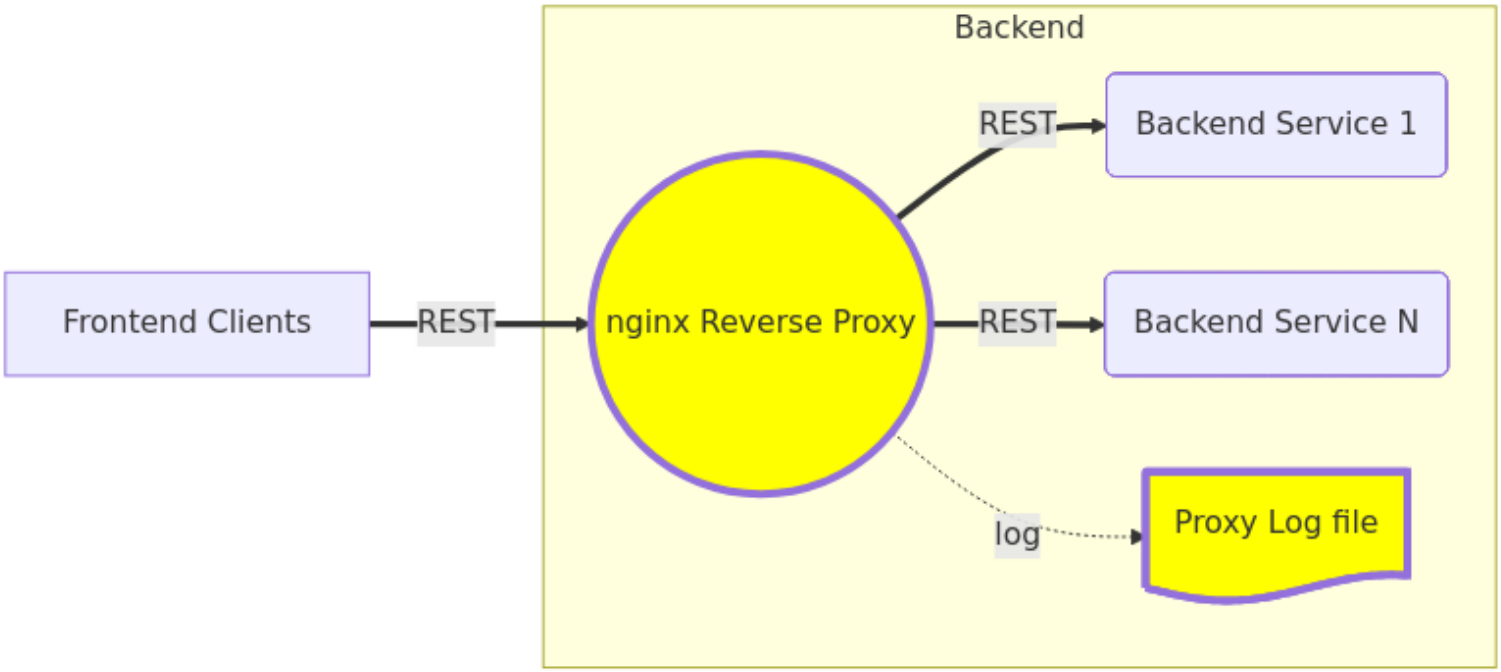


This workshop will heavily leverage ES|QL, Elastic's query-time language, to analyze our nginx reverse proxy logs. You can enter your queries in the pane at the top of the Elasticsearch tab. You can change the time window of your search using the Time Filter. To execute a search, click the Play/Refresh icon.



Getting started

We know that all of the REST API calls from our frontend web app flow through a nginx reverse proxy en route to our backend services; that seems like a good place to start our investigation.



Finding errors

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.

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 any 500 errors, 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 implicitly 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.

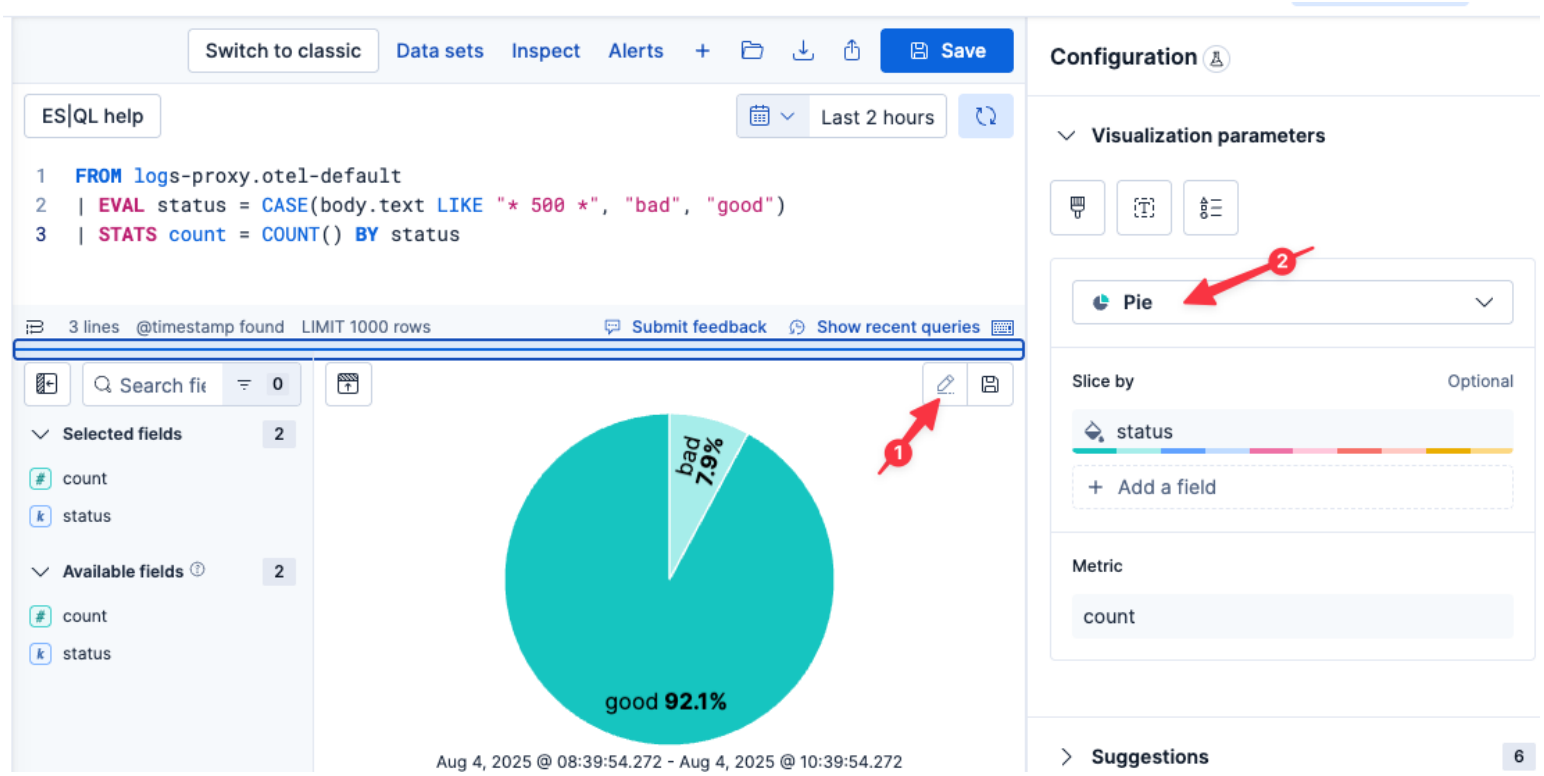
Are the errors affecting all requests?

The next thing we quickly want to understand is what percentage of requests to our backend services are resulting in 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 percentage of our overall requests. We don't yet have the tools to break this down further, but we will in a future exercise.

Are the errors still occurring?

Let's 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`

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

When did the errors start?

Let's see if we can find when the errors started occurring. Use the Time Filter to show the last 3 hours of data; this should automatically rerun the last query.

Ok, it looks like this issue first started happening roughly in the last 2 hours. We can use ES|QL's 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
| EVAL minutes_ago = DATE_DIFF("minute", minute, NOW())
```

A-ha! Using `CHANGE_POINT`, we can say that these errors clearly started occurring 60-80 minutes ago.

Parsing logs 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 code isn't just 500, or maybe we want to analyze status code vs. request URL, for example.

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

Some of you may already 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 to debug an expression yourself, you can leverage our AI Assistant to help!

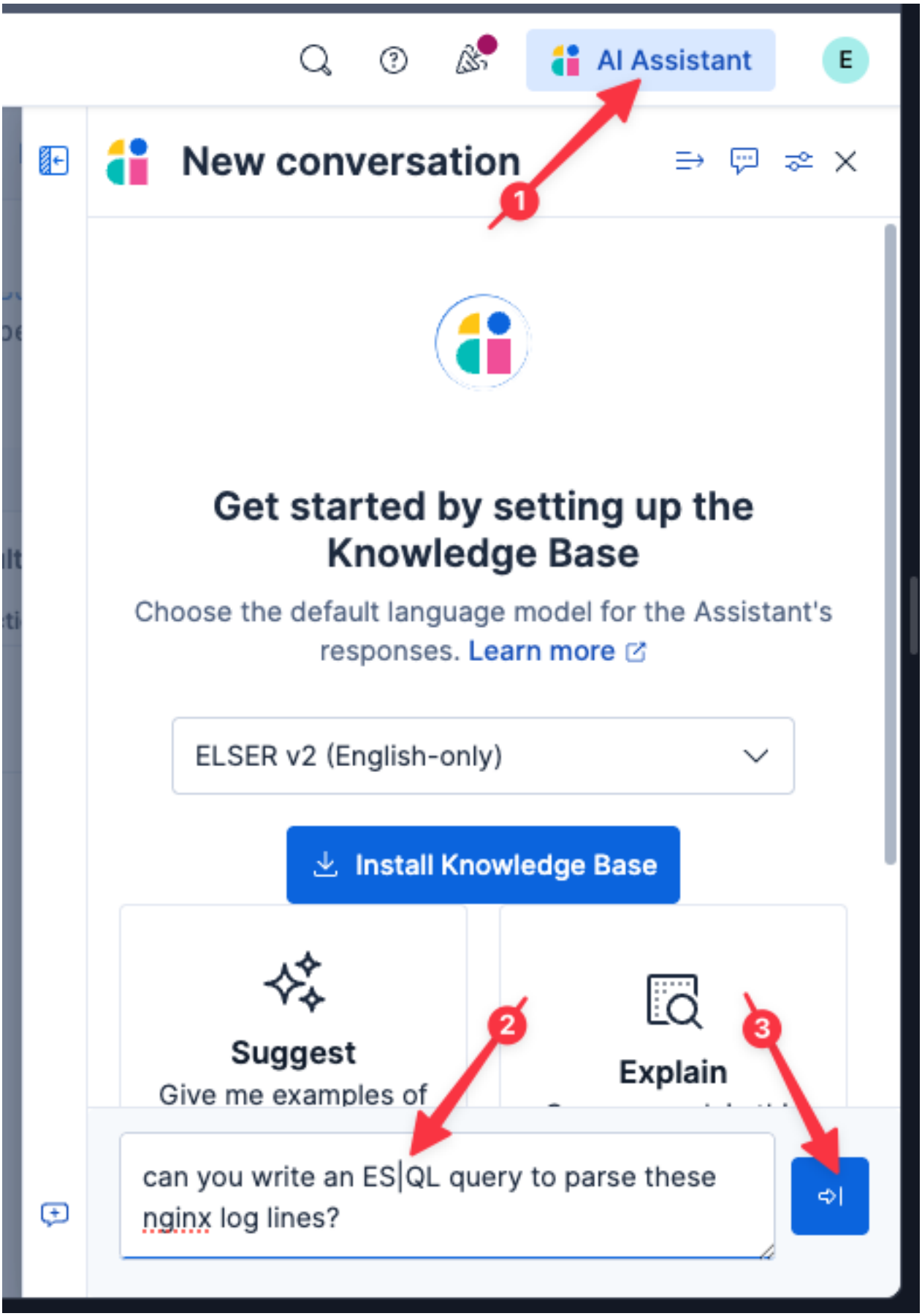
1. Execute the following query:

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

2. Click on the AI Assistant button in the upper-right.
3. Enter the following prompt in the `Send a message to the Assistant` field at the bottom of the fly-out.

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

4. Click the execute button



[!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 box and use the provided ES|QL expressions in the following exercises.

Making use of our parsed fields

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)
| STATS status_count = 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`

Now that we are graphing by `status_code`, we know definitively that we returning only 200 and 500 status codes.

> [!NOTE] You may notice that our search has gotten a little slower when we added query-time grok parsing. This is because Elasticsearch is now applying our grok pattern to *every* log line in the selected time window. 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 User Agents?

Our nginx access logs also include a User Agent field, which is a semi-structured field containing some information about the requesting browser. Ideally, we could also cross-reference the errors against this field to understand if it is affecting all browsers, or only some types of 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). Thankfully, Elastic provides tools to parse User Agent as you will see in an upcoming exercise.

Summary

Let's take stock of what we know:

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

And what we've done:

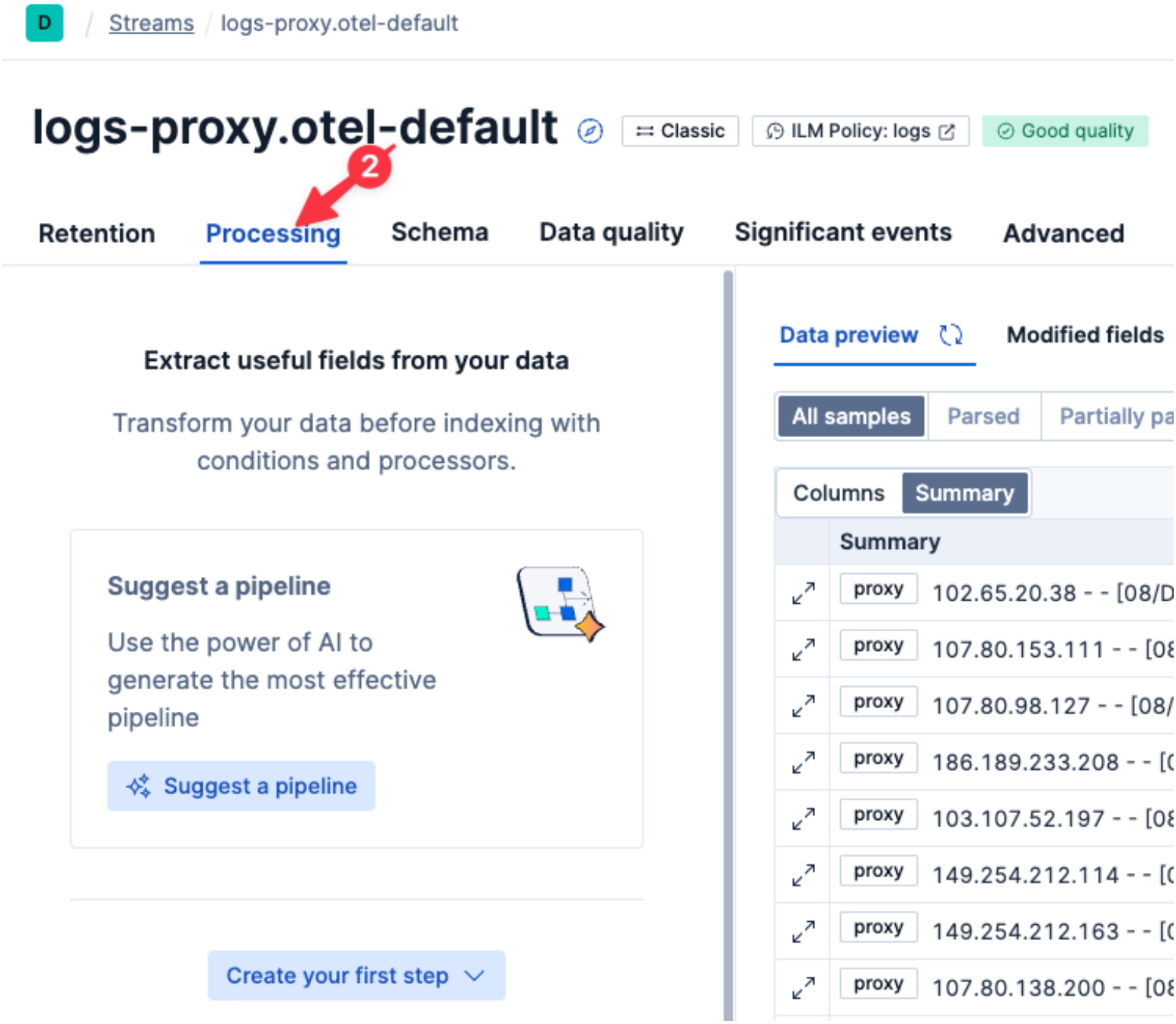
- Created several graphs to help quantify the extent of the problem ___

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 Elastic Streams which makes it easy to setup log parsing pipelines.

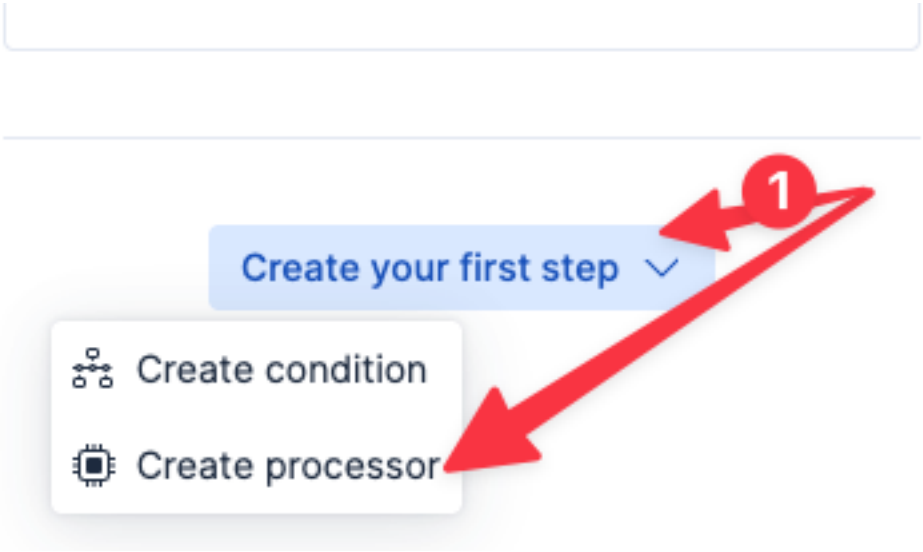
1. Select `logs-proxy.otel-default` from the list of data streams (if you start typing, Elasticsearch will help you find it)
2. Select the `Processing` tab



Parsing the log message

We can parse our nginx log messages at ingest-time using the Elastic Grok processor.

1. Select `Create processor` from the menu `Create your first step`



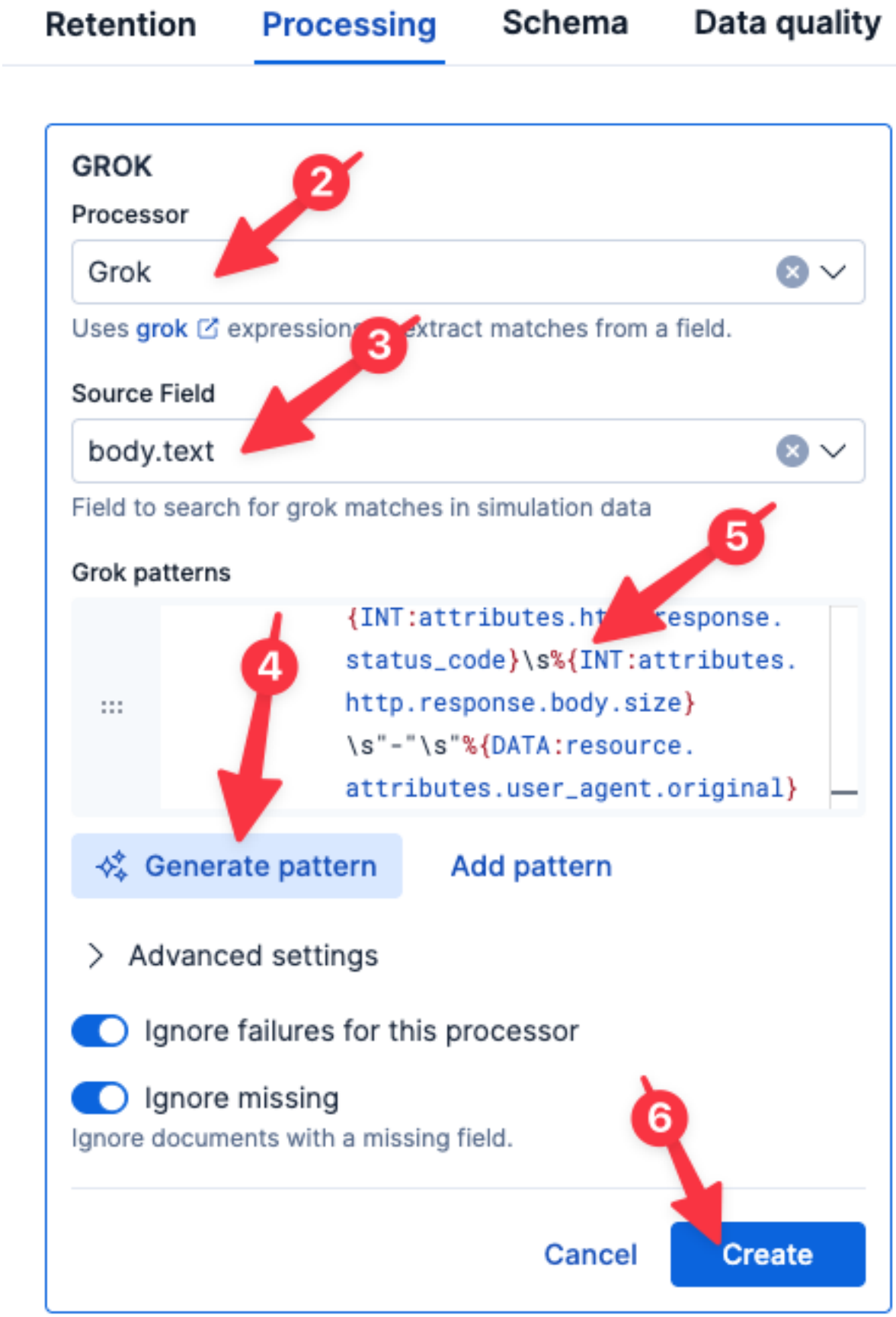
2. Select the `Grok` Processor (if not already selected)
3. Set the `Field` to

```
body.text
```

4. Click `Generate pattern`. Elasticsearch will analyze your log lines and try to determine a suitable grok pattern.
5. To ensure a consistent lab experience, copy the following grok expression and paste it into the `Grok patterns` field (*do not* click the `Accept` button next to the generated pattern)

```
%{IPV4:attributes.client.ip}\s-\s-\s\[%{HTTPDATE:attributes.custom.timestamp}\]\s"%{WORD:attributes.http.request.method_original} /(?<attribu
```

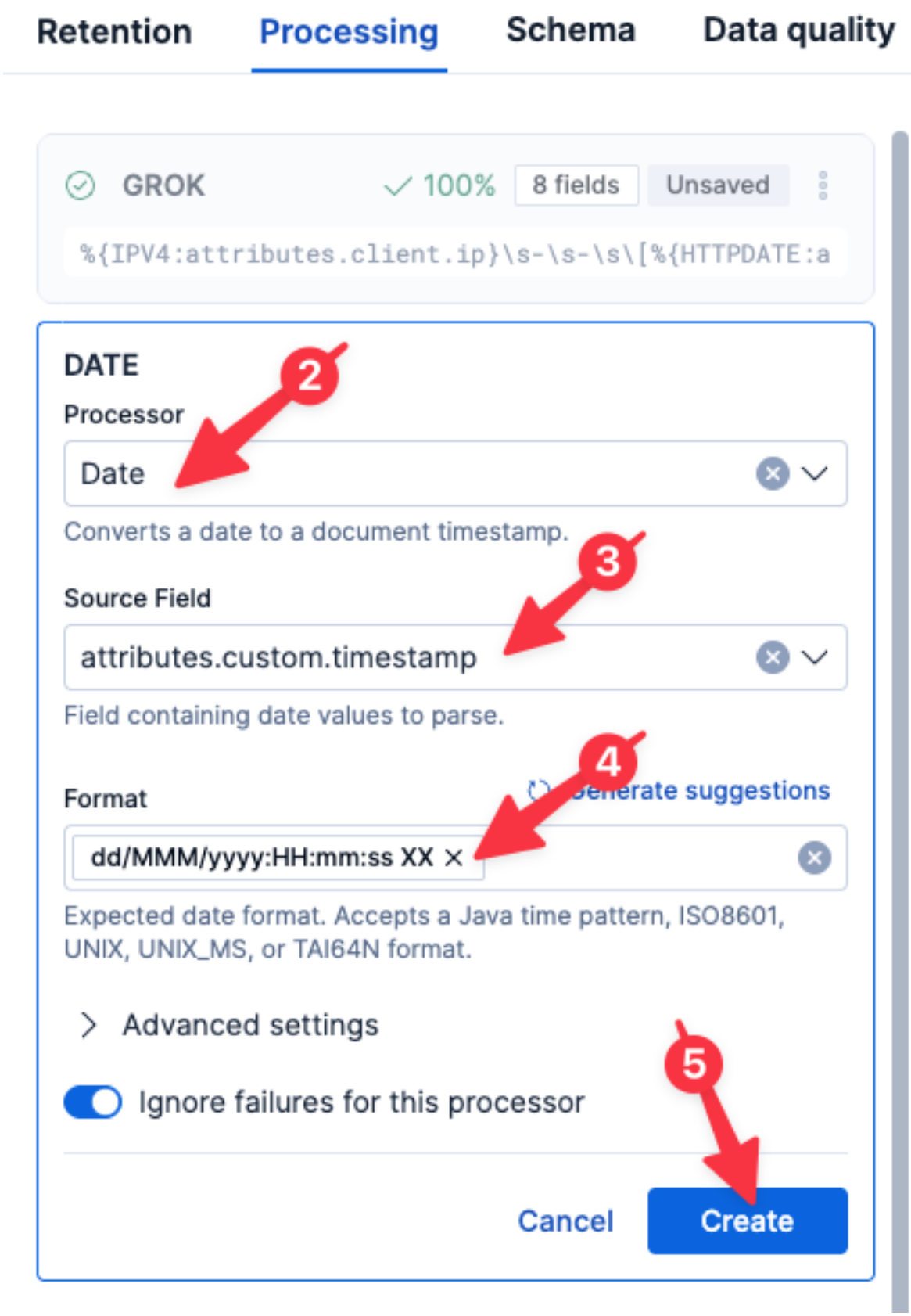
6. Wait until the sample `body.text` on the right shows highlighting, then click `Create`



Parsing the timestamp

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

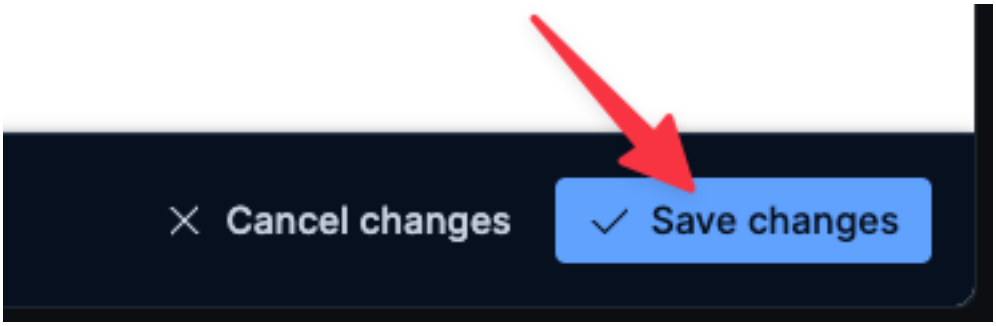
1. Select `Create processor` from the menu `Create`
2. Select the `Date` Processor
3. Set `Field` to `attributes.custom.timestamp`
4. Elastic should auto-recognize the format: `dd/MMM/yyyy:HH:mm:ss XX`
5. Click `Create`



Saving our processors

Now let's save our Processing chain.

1. Click `Save changes` in the bottom-right
2. Click `Confirm changes` in the resulting dialog



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, attributes.client.ip, attributes.http.request.method_original, attributes.url.path, attributes.http.response.status_code,
```

> [!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).

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

Execute the following query:

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

Note that this graph, unlike the one we drew before, currently shows only 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 5 Minutes` to zoom in on the newly parsed data.

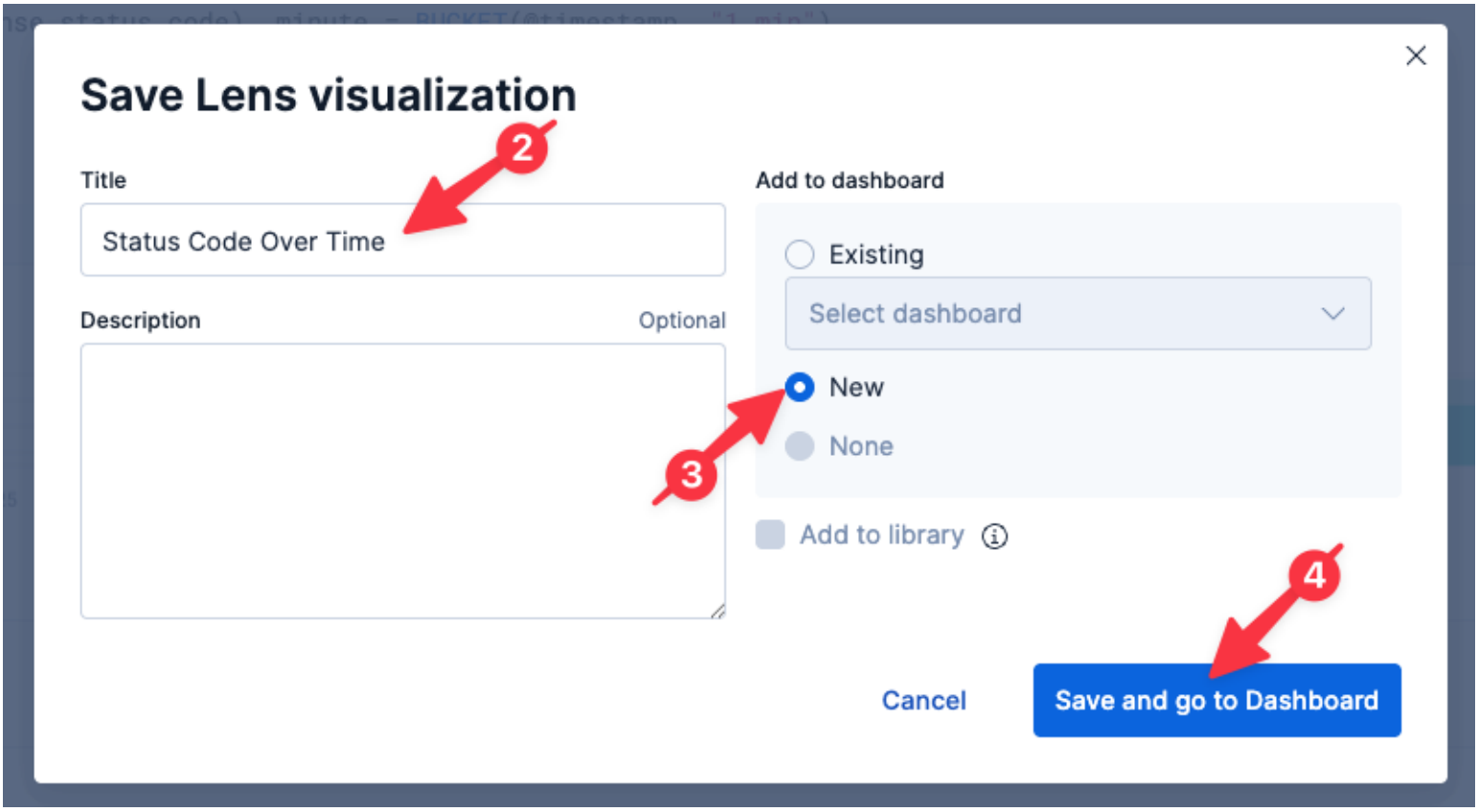
Saving our visualization to a dashboard

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

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

```
Status Code Over Time
```

3. Select `New` under `Add to dashboard`
4. Click `Save and go to Dashboard`



You will be taken to a new dashboard. Let's save it for future reference.

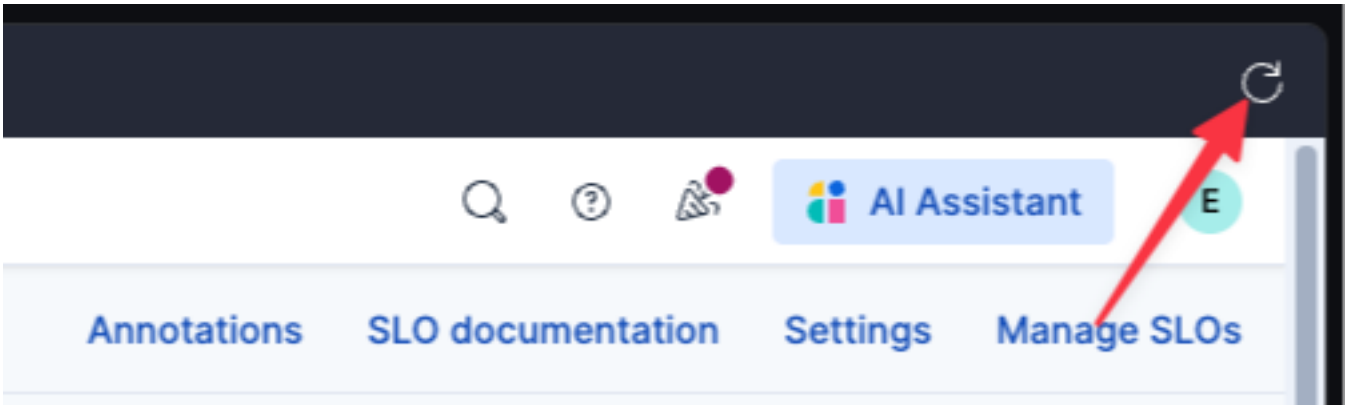
1. Click the `Save` button in the upper-right
2. Enter the title of the new dashboard as

```
Ingress Status
```

3. Click `Save`

Creating a 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 that we are parsing out specific fields at ingest-time, we can create a SLO to monitor HTTP status over time. 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 pane
2. Click `Create SLO`
3. Select `Custom Query` (if not already selected)
4. Set `Data view` to `logs-proxy.otel-default`
5. Set `Timestamp field` to `@timestamp` (if not already selected)

6. Set `Good query` to

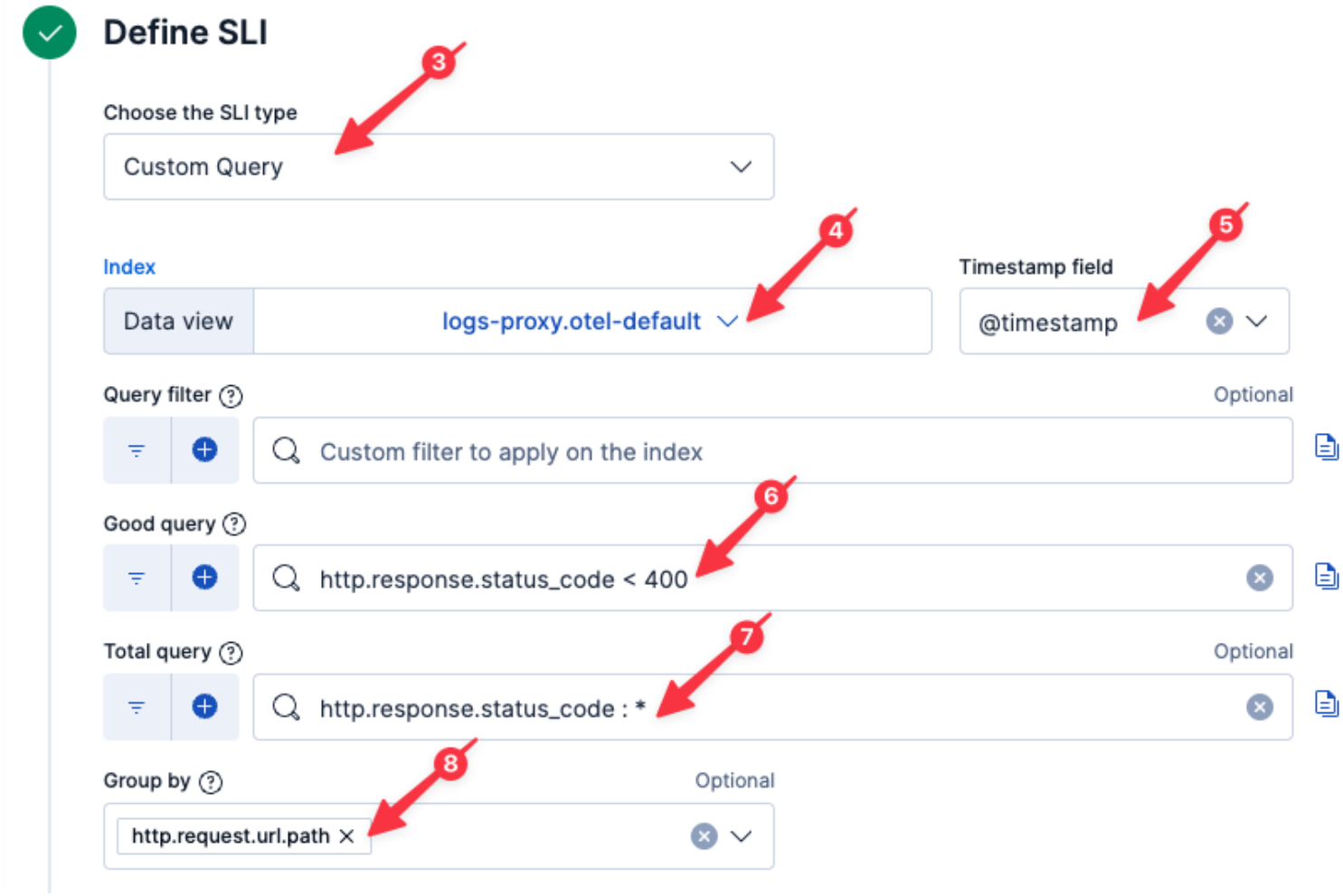
```
attributes.http.response.status_code < 400
```

7. Set `Total query` to

```
attributes.http.response.status_code : *
```

8. Set `Group by` to

```
attributes.url.path
```



9. Set `SLO Name` to

```
Ingress Status
```

10. Set `Tags` to

```
ingress
```

11. Click `Create SLO`

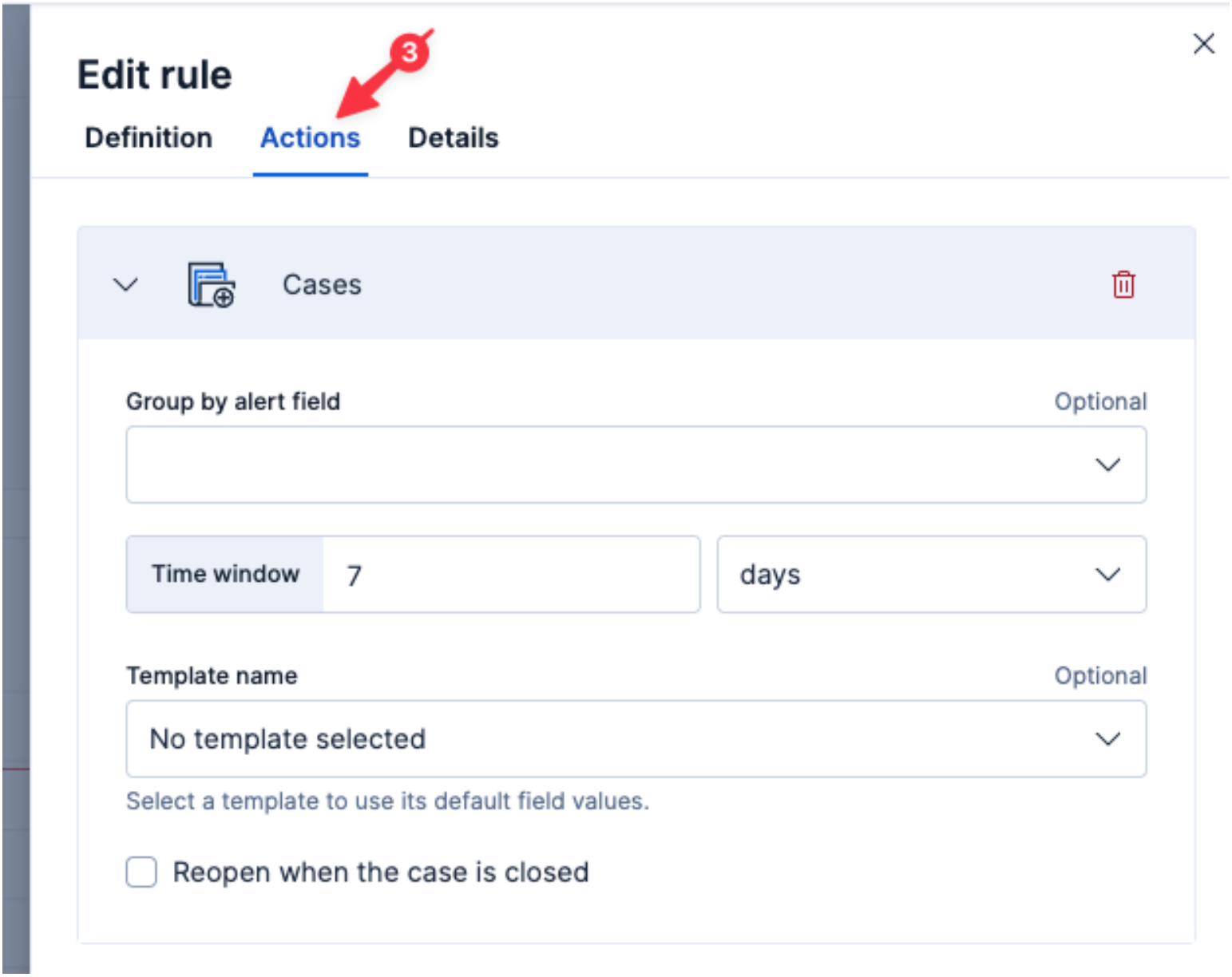
Alerting on a SLO

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. Click on the `Actions` tab of the fly-out
4. Click `Add action`
5. Select `Cases` (this will automatically open a case/ticket when this SLO is breached)



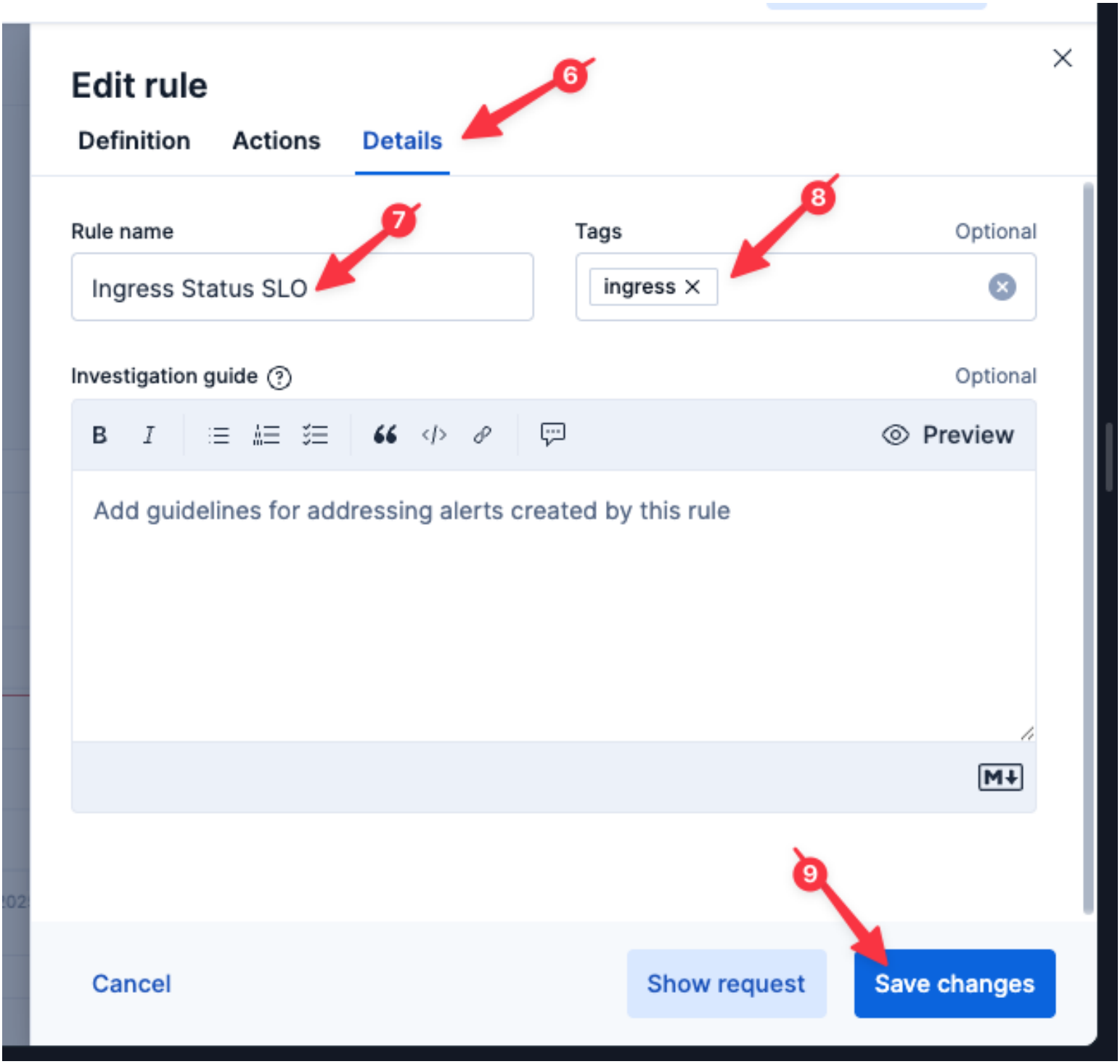
6. Click on the `Details` tab of the fly-out
7. Set the `Rule name` to :

```
Ingress Status SLO
```

8. Set `Tags` to

```
ingress
```

9. Click `Save changes`

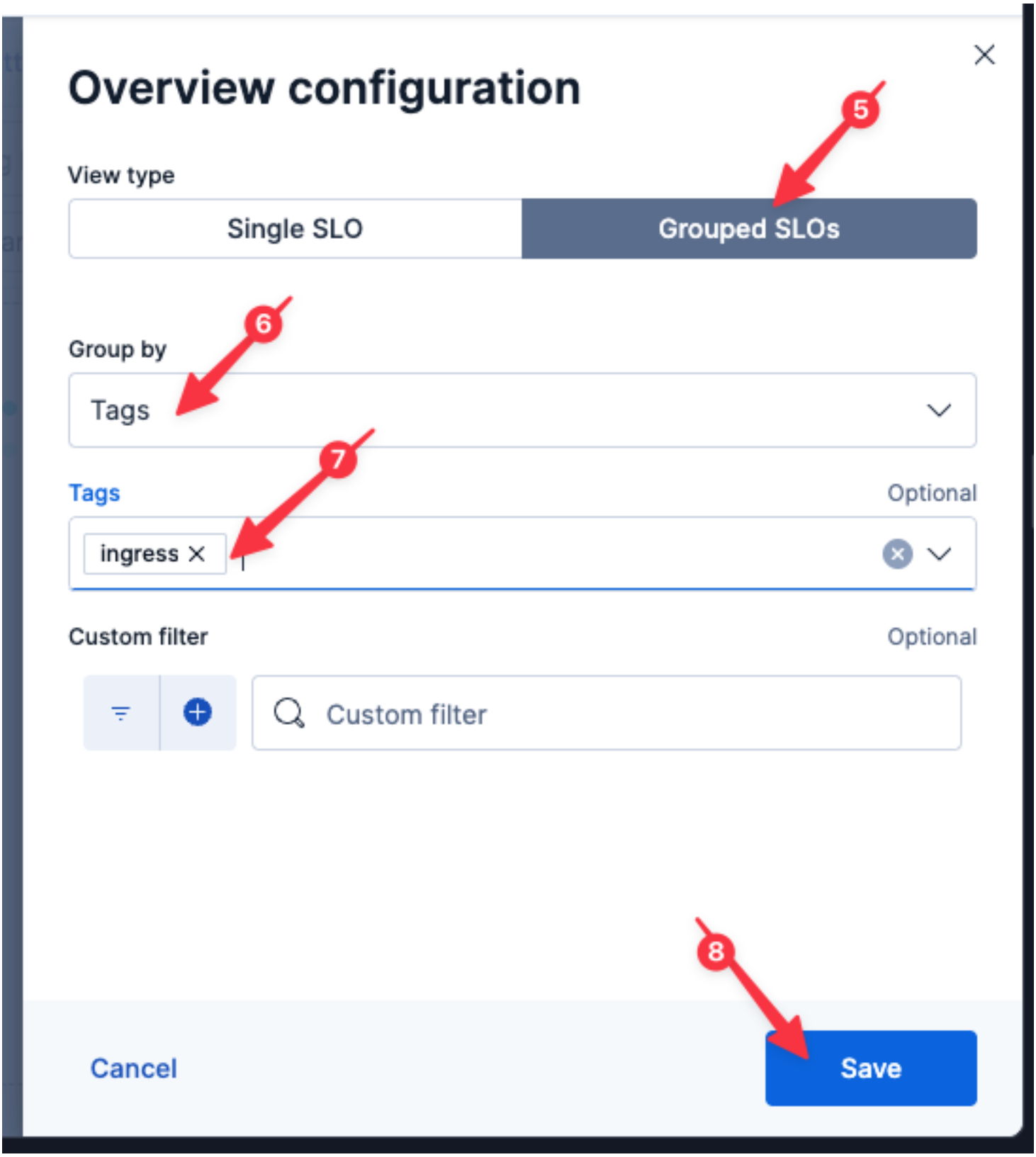


We now have a SLO that will tolerate a configurable percentage of errors, governed by a contractual goal. When it looks like we may violate that SLO, an alert will fire, which in turn will automatically open a case in Elastic! Moreover, if configured, Elastic can synchronize cases with your existing ticket management systems.

Adding SLO monitors to our dashboard

Now let's add the SLO monitor to our dashboard to help us find it in the future.

1. Click `Dashboards` in the left-hand navigation pane
2. Open the `Ingress Status` dashboard (if not already open)
3. Click the `Add` button in the upper-right
4. Select `New panel` from the dropdown menu
5. Select `SLO Overview`
6. Select `Grouped SLOs`
7. Set `Group by` to `Tags`
8. Set `Tags` to `ingress`
9. Click `Save`



Note that we are dynamically adding SLOs by tag. Any additional SLOs tagged with `ingress` will also appear here.

Adding alerts to our dashboard

Let's also add our growing list of alerts to our dashboard.

1. Click the `Add` button in the upper-right
2. Select `New panel` from the dropdown menu
3. Select `Alerts`
4. Set `Solution` to `Observability`
5. Set `Filter by` to `Rule tags`
6. Set `Rule tags` to `ingress`

7. Click `Save`

Note that we are dynamically adding alerts by tag. Any additional alerts tagged with `ingress` will also appear here.

Now save the changes to our dashboard by clicking the `Save` button in the upper-right.

Organizing our dashboard

As we are adding panels to our dashboard, we can group them into collapsible sections.

1. Click the `Add` button in the upper-right
2. Select `Collapsible Section` from the dropdown menu
3. Click on the Pencil icon to the right of the name of the new collapsible section
4. Name the collapsible section

```
Alerts
```

5. Click the green check box next to the name of the collapsible section
6. Open the collapsible section (if it isn't already) by clicking on the open/close arrow to the left of the collapsible section name
7. Drag `SLO Overview` and the Alerts panel we just setup into the body below the `Alerts` collapsible section
8. Click `Save` to save the dashboard

Summary

Let's take stock of what we know:

- a small percentage of requests are experiencing 500 errors
- the errors started occurring around 80 minutes ago
- the only error type seen is 500

And what we've done:

- Created several graphs to help quantify the extent of the problem
- Parsed the logs at ingest-time for quicker and more powerful analysis
- Created a dashboard to monitor our ingress proxy
- Create a SLO (with alert) to let us know if we ever return a significant number of 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.

Adding the GeoIP processor

We can add the Elastic GeoIP processor to geo-locate our clients based on their client IP address.

1. Select `logs-proxy.otel-default` from the list of Streams.
2. Select the `Processing` tab
3. Select `Create processor` from the menu `Create`
4. Select the `Manual pipeline configuration` Processor
5. Set the `Ingest pipeline processors` field to:

```
[
  {
    "geoip": {
      "field": "attributes.client.ip",
      "target_field": "client.geo",
      "ignore_missing": true
    }
  }
]
```

6. Click `Create`
7. Click `Save changes` in the bottom-right

8. Click `Confirm changes` in the resulting dialog

Retention | Processing | Schema | Data quality

GROK

%{IPV4:attributes.client.ip}\s-\s-\s\[%{HTTPDATE:att

DATE

attributes.custom.timestamp • dd/MMM/yyyy:HH:mm:ss X

MANUAL_INGEST_PIPELINE

Processor

Manual pipeline configuration

Specify an array of ingest pipeline processors using JSON.

Ingest pipeline processors

```
[
  {
    "geoip": {
      "field": "attributes.client.ip",
      "target_field": "client.geo",
      "ignore_missing": true
    }
  }
]
```

A JSON-encoded array of ingest pipeline processors. Conditions defined in the processor JSON take precedence over conditions defined in "Optional fields".

Advanced settings

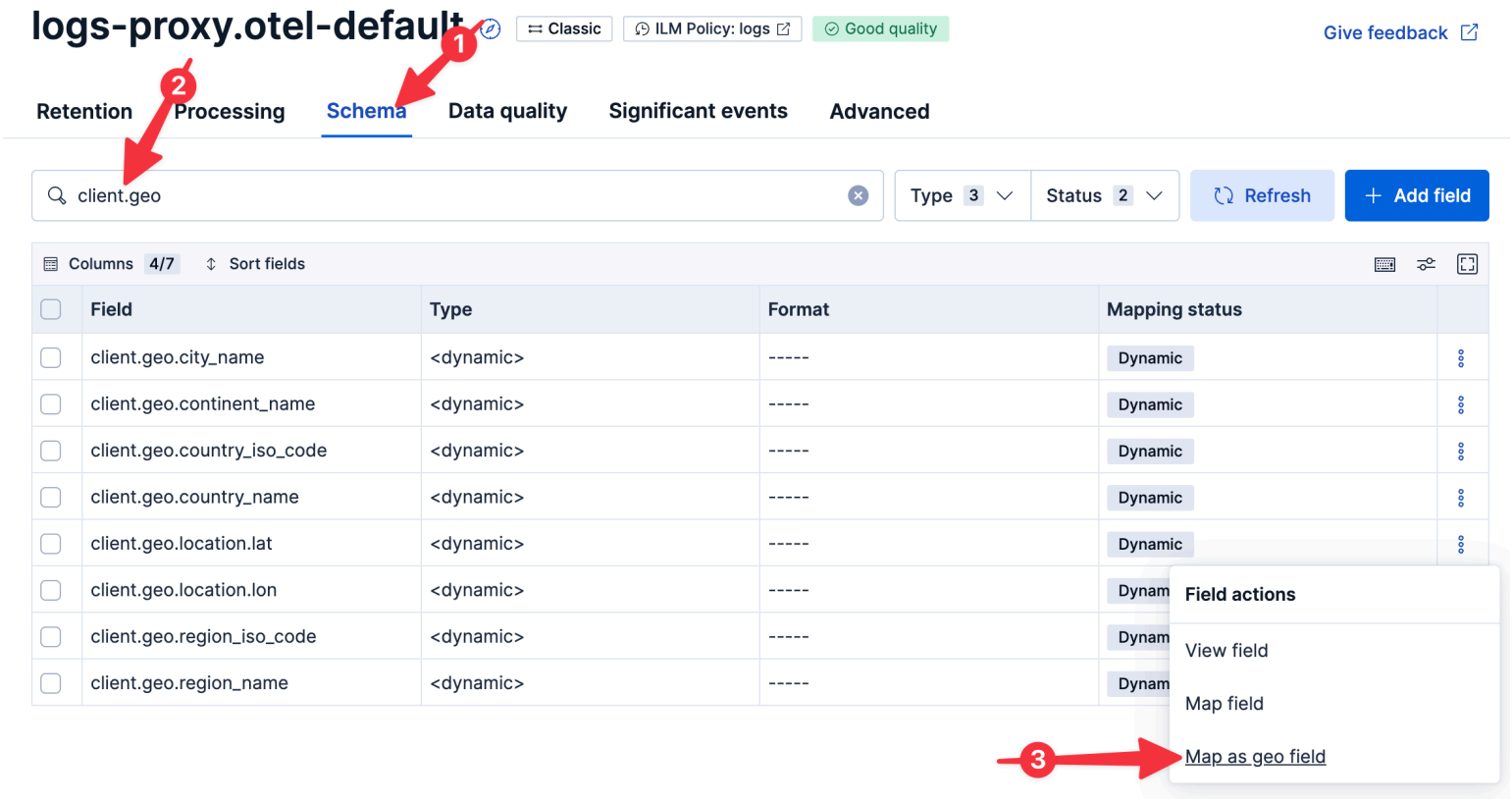
Ignore failures for this processor

Cancel | Create

Setting field mappings

Now we need to map several of our new fields to the proper field type.

1. Select the `Schema` tab
2. Search for field `client.geo`
3. Click on the ellipse on the right-hand side of the `client.geo.location.lat` row and select `Map as geo field`
4. Click `Stage changes` in the resulting dialog
5. Click on the ellipse on the right-hand side of the `client.geo.country_iso_code` row and select `Map field`
6. Set `Type` to `Keyword`
7. Click `Stage changes` in the resulting dialog
8. Click `Submit changes` in the bottom-right
9. Click `Confirm changes` in the resulting dialog



Analyzing with Discover

Jump back to Discover by clicking `Discover` in the left-hand navigation pane.

Adjust the time field to show the last 3 hours of data.

Execute the following query:

```
FROM logs-proxy.otel-default
| WHERE client.geo.country_iso_code IS NOT NULL AND attributes.http.response.status_code IS NOT NULL
| STATS COUNT() BY attributes.http.response.status_code, client.geo.country_iso_code
| SORT attributes.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
3. Click `Apply and close`

Wow! It looks like all of our 500 errors are occurring in the `TH` (Thailand) region. That is really interesting; without more information, we might be tempted to stop our RCA analysis here. However, there is often more to the story, as we will see.

Saving our visualization to a dashboard

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

1. Click on the Disk icon in the upper-right of the resulting graph

2. Name the visualization

```
Status by Region
```

3. Select `Existing` under `Add to dashboard`
4. Select the existing dashboard `Ingress Status` (you will need to start typing `Ingress` in the `Search dashboards...` field)
5. Click `Save and go to Dashboard`
6. Once the dashboard has loaded, click the `Save` button in the upper-right

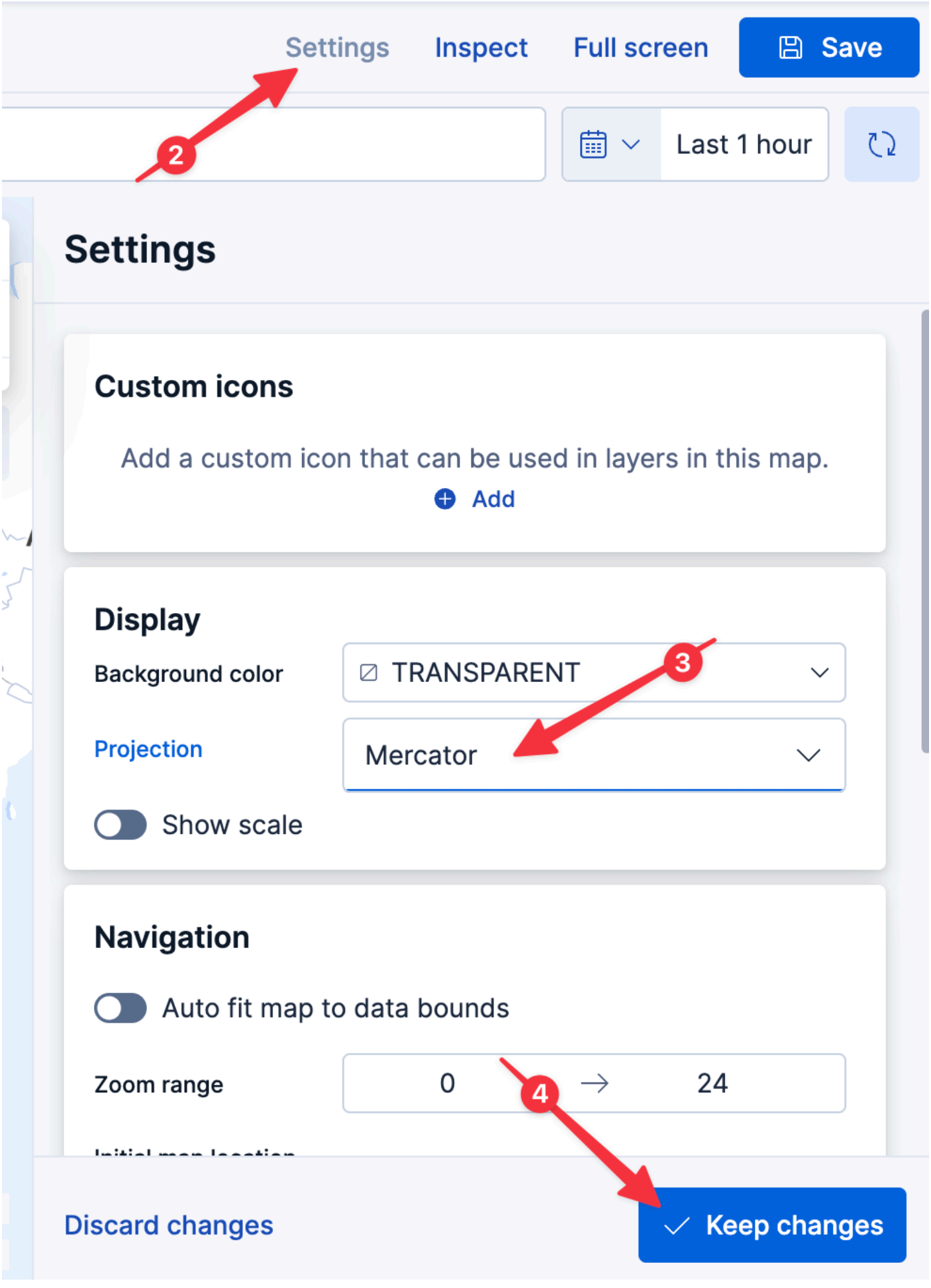
Visualizing with Maps

> [!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.

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

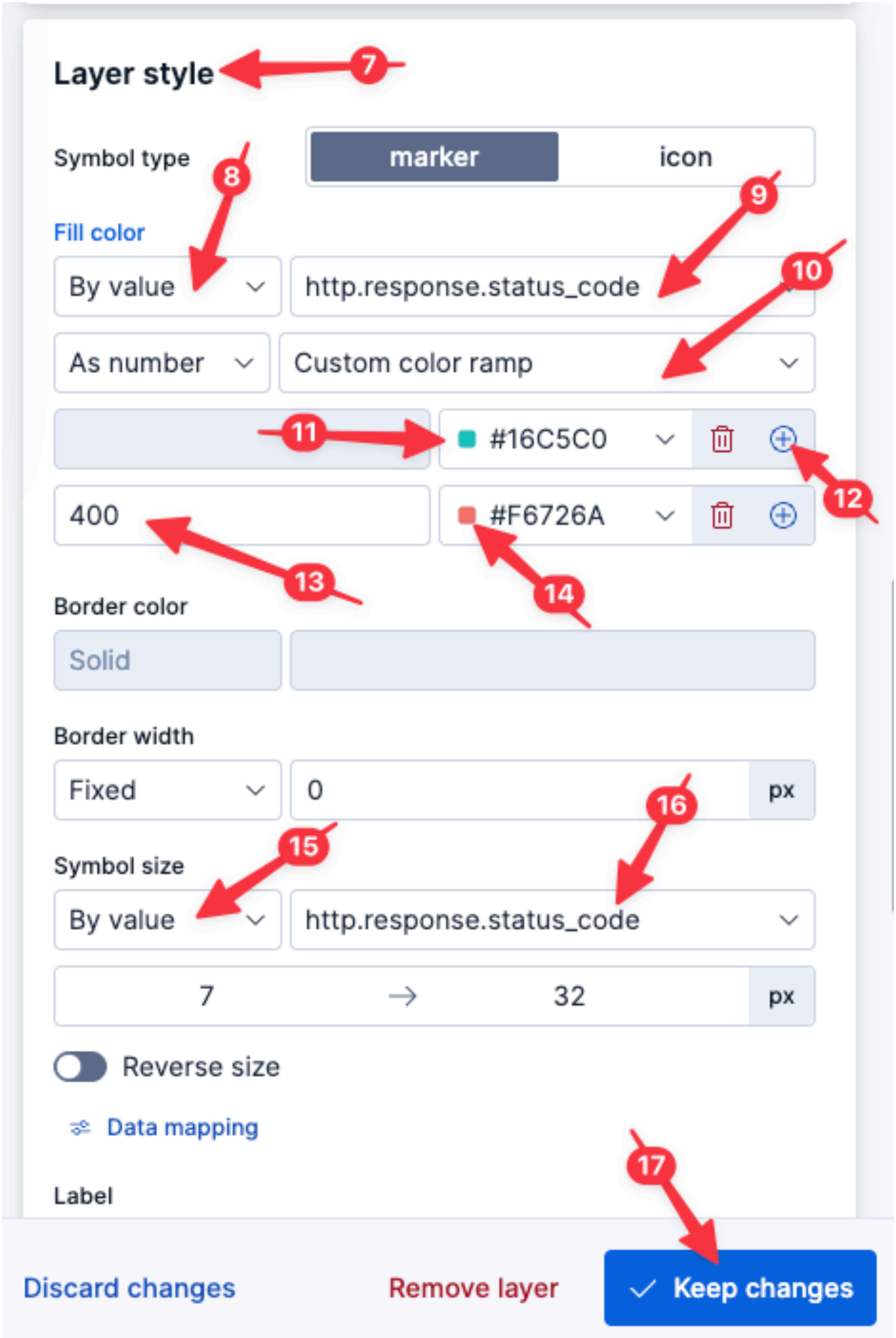
First, let's change the map projection from 3D to 2D:

1. Go to `Other tools` > `Maps` using the left-hand navigation pane
2. Click `Settings` in the upper-right
3. Under `Display`, set `Projection` to `Mercator`
4. Click `Keep changes` in the settings fly-out


Settings
Inspect
Full screen
Save
2
Last 1 hour
Settings
Custom icons
Add a custom icon that can be used in layers in this map.
Add
Display
Background color
TRANSPARENT
3
Projection
Mercator
Show scale
Navigation
Auto fit map to data bounds
Zoom range
0
24
4
Discard changes
Keep changes

Now let's visualize client access by location and status code:

1. Click `Add layer`
2. Click the `Elasticsearch` tab under `Add layer`
3. Select `Documents`
4. Select `Data view` to `logs-proxy.otel-default`
5. Set `Geospatial field` to `client.geo.location` (if this field isn't available, refresh the Instruqt virtual browser tab)
6. Click `Add and continue`
7. Scroll down to `Layer style`
8. Set `Fill color` to `By value`
9. Set `Select a field` to `http.response.status_code`
10. Select `Custom color ramp` in the field next to `As number`
11. Select a greenish color for the first number row (if not already selected)
12. Click the + button to the right of the first number row
13. Enter `400` in the second number row
14. Select a reddish color for the `400` number row
15. Set `Symbol Size` to `By value`
16. Set `Select a field` to `http.response.status_code`
17. Click `Keep changes`



Feel free to scroll around the globe and note the intuitive visualization of client locations and status codes.

Saving our map to a dashboard

Let's add our map to our dashboard for future reference.

1. Click the `Save` button in the upper-right
2. Set `Title` to

```
Status Code by Location
```

3. Select existing dashboard `Ingress Status` (you will need to start typing `Ingress` in the `Search dashboards...` field)
4. Click `Save and go to dashboard`

Now save the dashboard by clicking on the `Save` button in the upper-right.

Summary

Let's take stock of what we know:

- a small percentage of requests are experiencing 500 errors
- the errors started occurring around 80 minutes ago
- the only error type seen is 500
- the errors occur only in the `TH` (Thailand) region

And what we've done:

- Created several graphs to help quantify the extent of the problem
- Parsed the logs at ingest-time for quicker and more powerful analysis
- Created a dashboard to monitor our ingress proxy
- Create a SLO (with alert) to let us know if we ever return a significant number of non-200 error codes over time
- Geocoded client IP to associate location information with clients
- Created visualizations to help us visually locate clients and 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 type of browser?

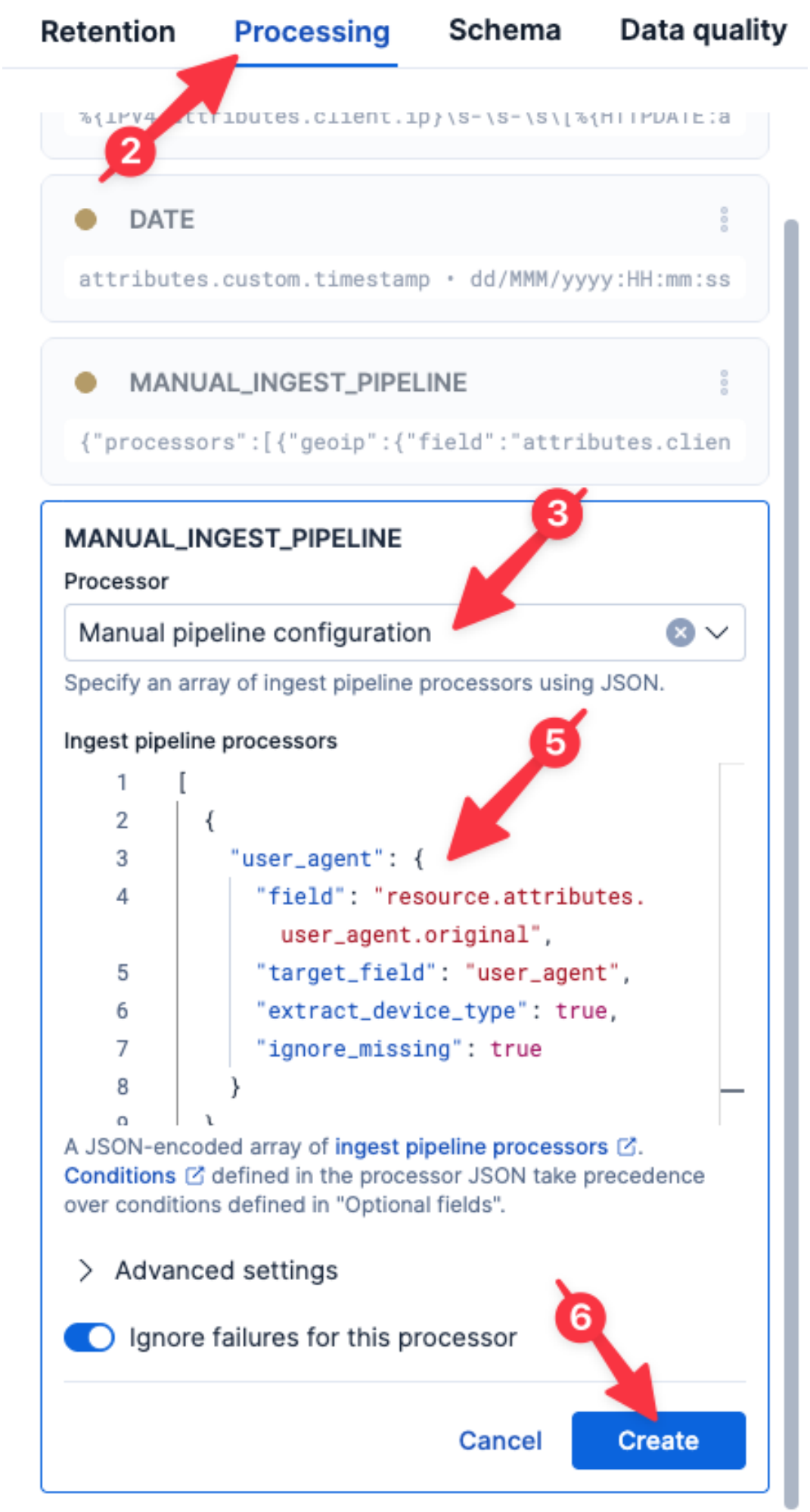
Remember the User Agent string we tried to group by and failed using ES|QL? While nearly impossible to parse with a simple grok expression, we can easily parse the User Agent string using the Elastic User agent processor.

Adding the User Agent processor

1. Select `logs-proxy.otel-default` from the list of Streams.
2. Select the `Processing` tab
3. Select `Create processor` from the menu `Create`
4. Select the `Manual pipeline configuration` Processor
5. Set the `Ingest pipeline processors` field to:

```
[
  {
    "user_agent": {
      "field": "resource.attributes.user_agent.original",
      "target_field": "user_agent",
      "extract_device_type": true,
      "ignore_missing": true
    }
  }
]
```

6. Click `Create`


Retention
Processing
Schema
Data quality
2
DATE
attributes.custom.timestamp • dd/MMM/yyyy:HH:mm:ss
MANUAL_INGEST_PIPELINE
{"processors":[{"geoip":{"field":"attributes.clien
3
MANUAL_INGEST_PIPELINE
Processor
Manual pipeline configuration
Specify an array of ingest pipeline processors using JSON.
Ingest pipeline processors
5
[
{
"user_agent": {
"field": "resource.attributes.
user_agent.original",
"target_field": "user_agent",
"extract_device_type": true,
"ignore_missing": true
}
A JSON-encoded array of ingest pipeline processors.
Conditions defined in the processor JSON take precedence over conditions defined in "Optional fields".
Advanced settings
Ignore failures for this processor
6
Cancel
Create

Setting field mappings

1. Click the `Modified fields` tab
2. Click the ellipse on the far right of the `user_agent.name` row
3. Select `Map field`
4. Set the type to `Keyword`
5. Click the ellipse on the far right of the `user_agent.version` row
6. Select `Map field`
7. Set the type to `Keyword`
8. Click `Stage changes`

Saving our processors

Now let's save our Processing chain.

1. Click `Save changes` in the bottom-right
2. Click `Confirm changes` in the resulting dialog

Analyzing with Discover

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
| EVAL user_agent.full = CONCAT(user_agent.name, " ", user_agent.version)
| 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.version 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 identified as being associated with the errors we saw?

Execute the following query:

```
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 LIKE "*Chrome*" AND TO_INT(version_major) == 136
| STATS COUNT() BY client.geo.country_iso_code
```

A-ha! It appears that this specific version of the Chrome browser (v136) 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. That would explain why we saw errors only 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 requests are experiencing 500 errors
- the errors started occurring around 80 minutes ago
- the only error type seen is 500
- the errors occur only in the `TH` region
- the errors occur only with browsers based on Chrome v136

And what we've done:

- Created several graphs to help quantify the extent of the problem
- Parsed the logs at ingest-time for quicker and more powerful analysis
- Created a dashboard to monitor our ingress proxy
- Create a SLO (with alert) to let us know if we ever return a significant number of non-200 error codes over time
- Geocoded client IP to associate location information with clients
- Created visualizations to help us visually locate clients and errors
- Parsed the user agent string to associate browser information with clients
- Determined, using client location and browser information, the root cause of our problem ___

Now that we know what happened, let's try to be sure this never happens again by building out more reporting and alerting.

Generating a table of user agents

Let's keep track of new User Agents as they appear in the wild.

Jump back to Discover by clicking `Discover` in the left-hand navigation pane.

Execute the following query:

```
FROM logs-proxy.otel-default
| EVAL user_agent.full = CONCAT(user_agent.name, " ", user_agent.version)
| 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.

Using LOOKUP JOIN to determine release date

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

We could try to maintain our own User Agent lookup table and use ES|QL LOOKUP JOIN 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
| EVAL user_agent.full = CONCAT(user_agent.name, " ", user_agent.version)
| WHERE user_agent.full IS NOT NULL
| EVAL user_agent.name_and_vmajor = SUBSTRING(user_agent.full, 0, LOCATE(user_agent.full, ".")-1) // simplify user_agent
| 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 // lookup release_date from ua_lookup using user_agent.name_and_vmajor key
| 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.

Using COMPLETION to determine release date

Fortunately, Elastic makes it possible to leverage an external Large Language Model (LLM) as part of an ES|QL query using the COMPLETION command. In this case, we can pipe each browser to the LLM and ask it to return the release date.

Execute the following query:

```
FROM logs-proxy.otel-default
| EVAL user_agent.full = CONCAT(user_agent.name, " ", user_agent.version)
| 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 {"inference_id" : "openai_completion"} // call out to LLM for each record
| 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 the `Save` button in the upper-right
2. Set `Title` to

```
ua_release_dates
```

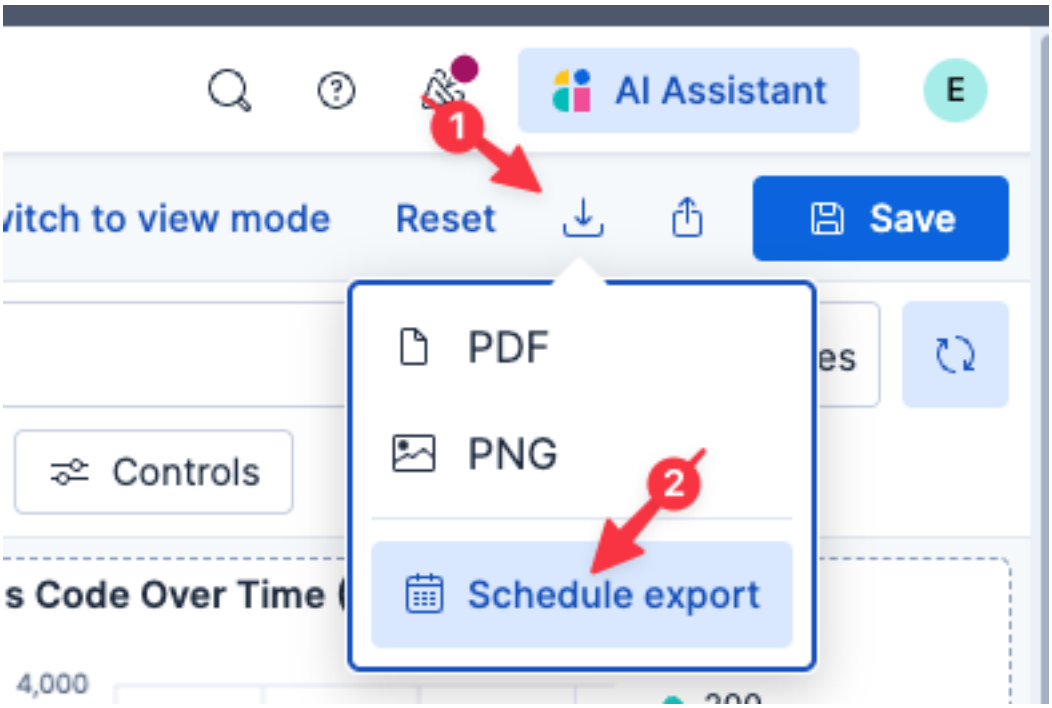
3. Click `Save`

Saving an ES|QL query allows others on our team to easily re-run it on demand.

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 on `Export` icon
2. Select `Schedule exports`
3. Click `Schedule exports` at the bottom-right of the resulting fly-out



> [!NOTE] In practice, you would setup an email Connector to allow Elasticsearch to automatically email the dashboard to the CIO on the schedule you defined.

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!

Transforms run asynchronously in the background, querying data, aggregating it, and writing the results to a new index. In this case, we can use a Pivot transform to read from our parsed proxy logs and pivot based on `user_agent.full`. This will create a new index with one record per `user_agent.full`. We can then alert whenever a new record is added to this index, indicating a new User Agent!

Creating a transform

> [!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 Transform.

1. Go to `Data management` > `Transforms` using the left-hand navigation pane
2. Click `Create a transform`
3. Select `logs-proxy.otel-default` as the data source
4. Select `Pivot` (if not already selected)
5. Set `Search filter` to

```
user_agent.name :*
```

6. Set `Group by` to `terms(user_agent.name)` and `terms(user_agent.version)`
7. Add an aggregation for `@timestamp.min`
8. Click `> Next`

Time range

Last 1 hour

Use full data

Search filter

user_agent.full :*

Edit JSON query

Runtime fields

No runtime field

Edit runtime fields

Source documents

Results are limited to a maximum of 10000 for preview purposes

Columns 10/57 Sort fields Histogram charts

1755095800000 - 175... @timestamp	1 category attributes.code.file.path	Chart not supported. attributes.code.file.path....	0 attributes.code.line.num...	Chart not sup body.text
Aug 13, 2025 @ 09:37:5...	proxy.py	proxy.py	0	107.80.111.1
Aug 13, 2025 @ 09:37:5...	proxy.py	proxy.py	0	103.107.52.5
Aug 13, 2025 @ 09:37:5...	proxy.py	proxy.py	0	102.65.25.10
Aug 13, 2025 @ 09:37:5...	proxy.py	proxy.py	0	103.107.52.2
Aug 13, 2025 @ 09:37:5...	proxy.py	proxy.py	0	149.254.212

Rows per page: 5

1 2 3 4 5 ... 2000

Transform configuration

Group by

user_agent.full

Edit JSON config

Add a group by field ...

Aggregations

@timestamp.min

Add an aggregation ...

Preview

Columns 2 Sort fields

user_agent.full	@timestamp.min
Chrome 126.0.6478.222	August 13th 2025, 09:3...
Chrome 127.0.6533.37	August 13th 2025, 09:3...
Chrome 128.0.6613.59	August 13th 2025, 09:3...
Chrome 131.0.6778.27	August 13th 2025, 09:3...
Chrome 133.0.6943.116	August 13th 2025, 09:3...

Rows per page: 5

1 2 3 4 6 7

Next

9. Set the `Transform ID` to

`user_agents`

10. Set `Time field` to `min(@timestamp)` (if not already selected)
11. Set `Continuous mode` on
12. Set `Delay` under `Continuous mode` to `0s`
13. Open `Advanced settings`
14. Set the Frequency to `1s` under `Advanced Settings`
15. Click `Next`

Transform ID

user_agents

9

Transform description

Description (optional)

Use transform ID as destination index name

Destination ingest pipeline

Select an ingest pipeline (optional)

Create data view

10

Time field for data view

@timestamp.min

Select a primary time field for use with the global time filter.

11

Continuous mode

Date field for continuous mode

@timestamp

Select the date field that can be used to identify new documents.

12

Delay

0s

Time delay between current time and latest input data time.

Retention policy

13

Advanced settings

14

Frequency

1s

The interval to check for changes in source indices when the transform runs continuously.

Maximum page search size

500

The initial page size to use for the composite aggregation for each checkpoint.

Number of failure retries

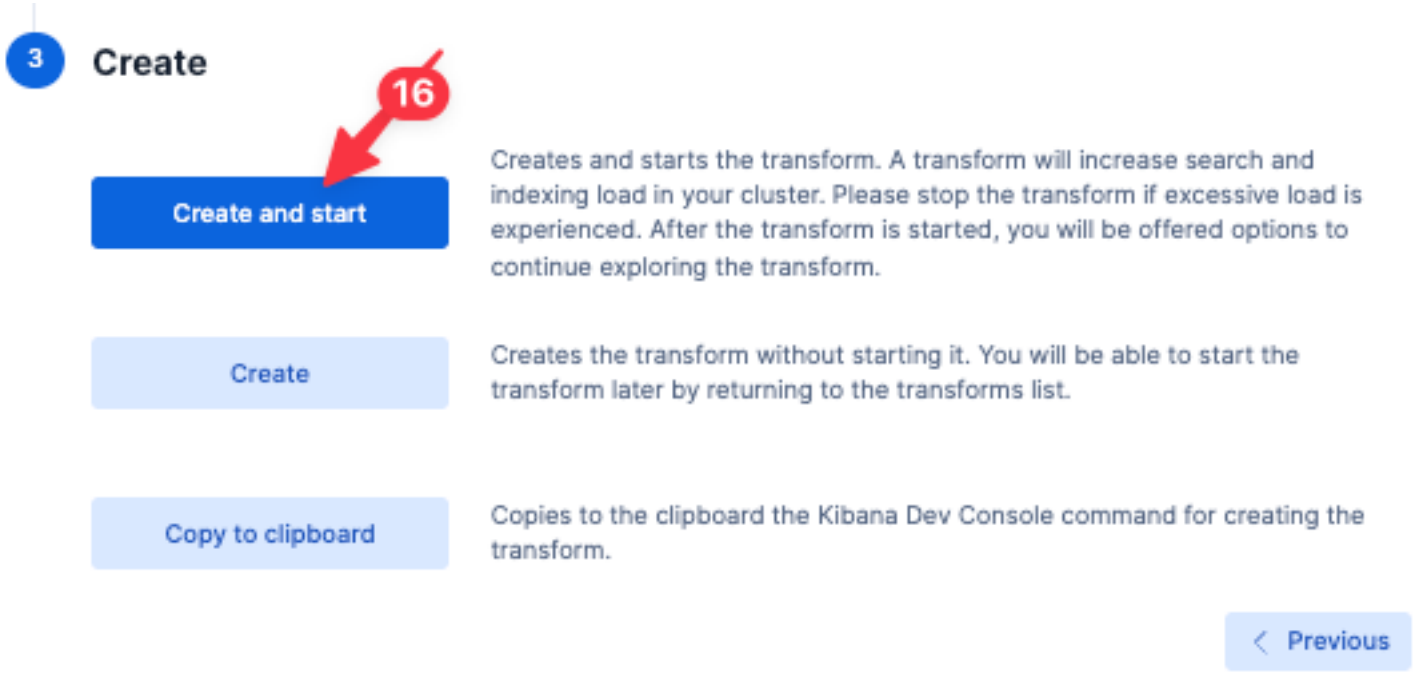
The number of retries on a recoverable failure before the transform task is marked as failed. Set it to -1 for infinite retries.

15

Previous

Next

16. Click `Create and start`



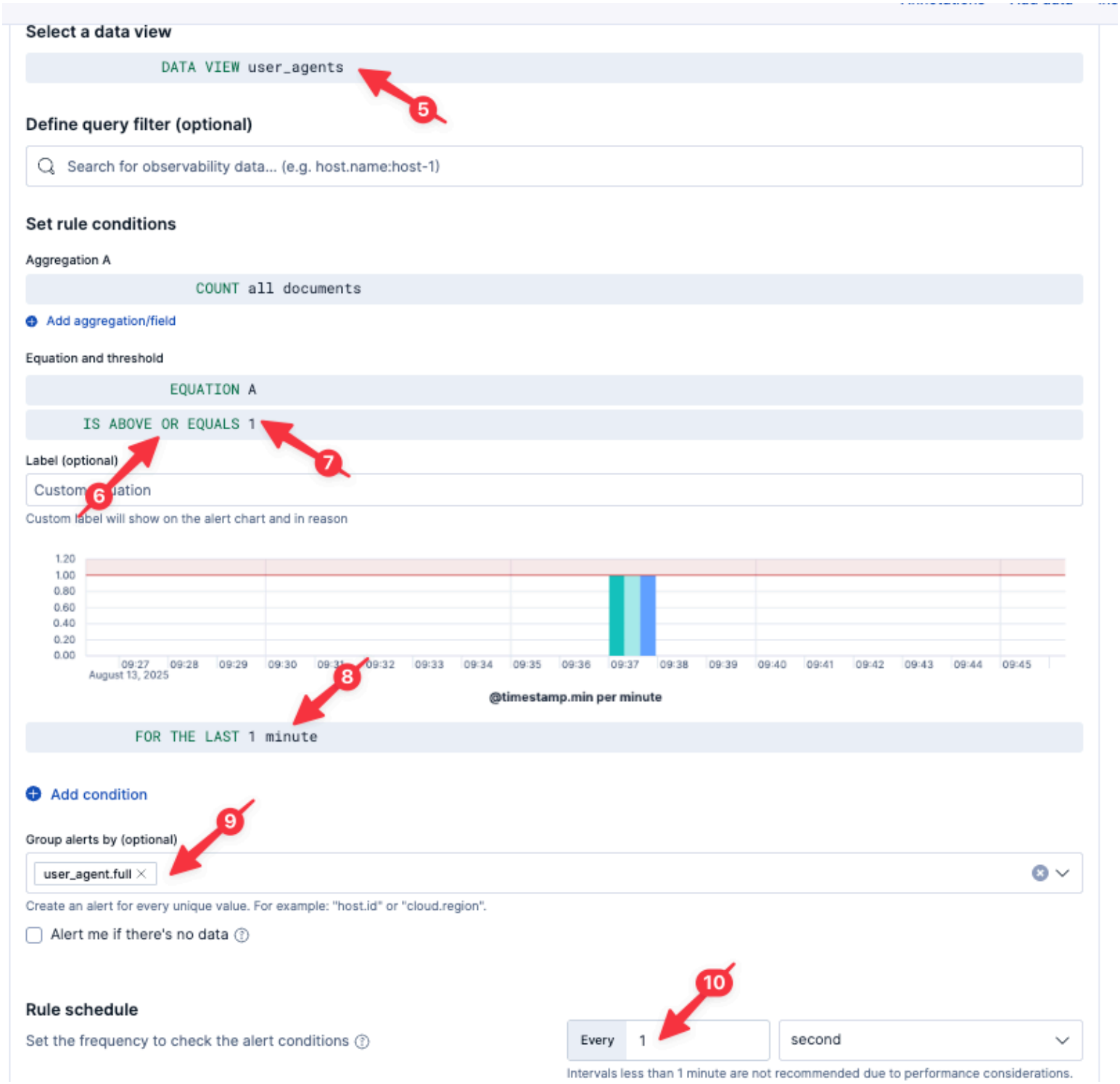
[!NOTE] We are intentionally choosing very aggressive settings here strictly for demonstration purposes (e.g., to quickly trigger an alert). In practice, you would use more a more practical frequency, for example.

Our transform is now running every second looking for new User Agents in the `logs-proxy.otel-default` datastream. It is smart enough to only look for new User Agents across log records which have arrived since the last run of the transform. When a new User Agent is seen, a corresponding record is written to the `user_agents` index.

Creating an alert

Let's create a new alert which will fire whenever a new User Agent is seen. We specifically want to alert whenever a new record is written to the `user_agents` index, which in turn is maintained by the transform we just created.

1. Go to `Alerts` using the left-hand navigation pane
2. Click `Manage Rules`
3. Click `Create Rule`
4. Select `Custom threshold`
5. Set `DATA VIEW` to `user_agents`
6. Change `IS ABOVE` to `IS ABOVE OR EQUALS`
7. Set `IS ABOVE OR EQUALS` to `1`
8. Set `FOR THE LAST` to `1 minute`
9. Set `Group alerts by (optional)` to `user_agent.name` and `user_agent.version`
10. Set `Rule schedule` to `1 seconds`



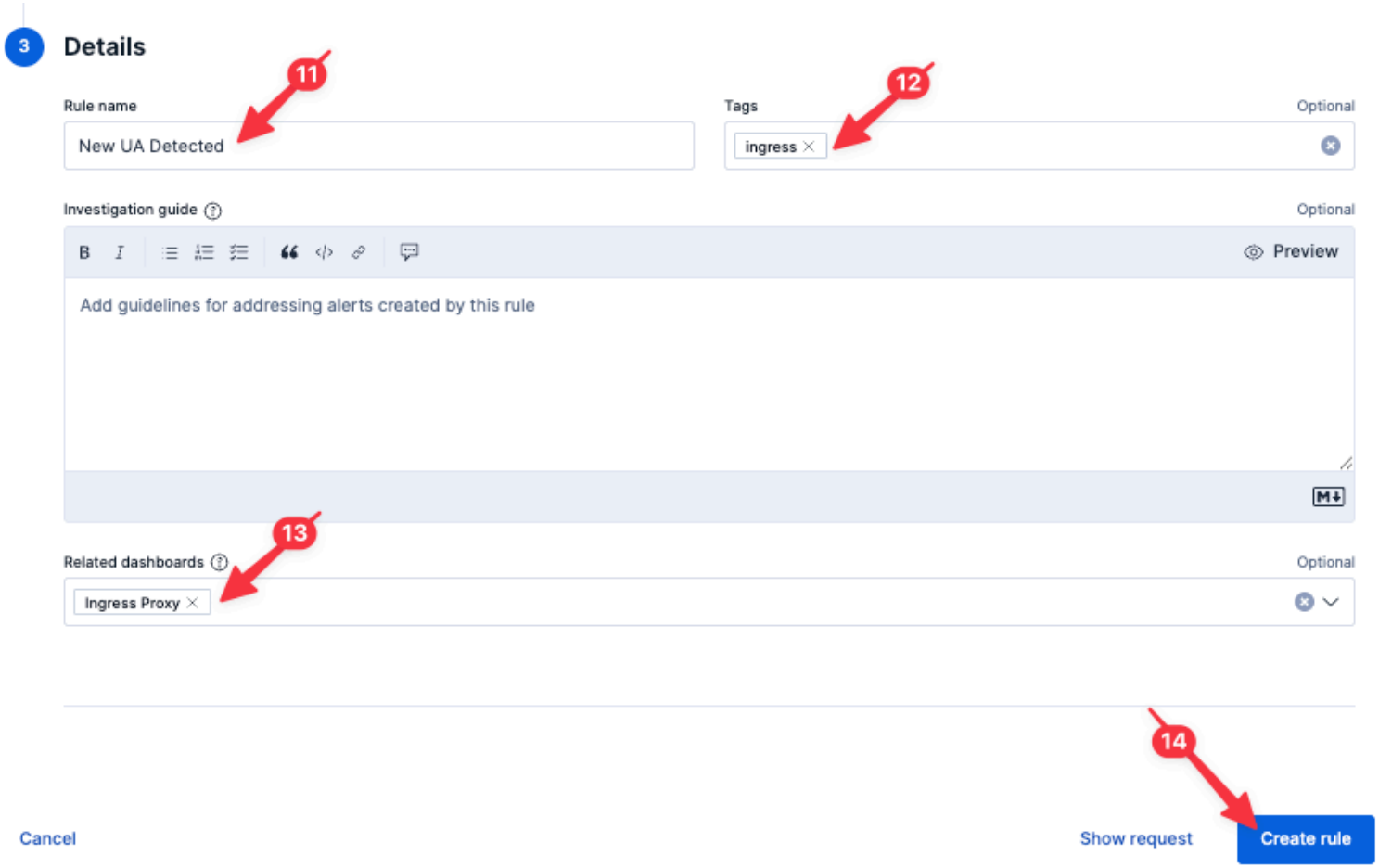
11. Set `Rule name` to

```
New UA Detected
```

12. Set `Tags` to

```
ingress
```

13. Set `Related dashboards` to `Ingress Status`
14. Click `Create rule`
15. Click `Save rule` in the resulting pop-up



[!NOTE] We are intentionally choosing very aggressive settings here strictly for demonstration purposes (e.g., to quickly trigger an alert). In practice, you would use more a more practical frequency, for example.

Testing our alert

1. Open the button label="Terminal" Instruqt tab
2. Run the following command:

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

This will create a new Chrome UA v137. Let's go to our dashboard and see if we can spot it.

1. Open the button label="Elasticsearch" Instruqt tab
2. Go to `Dashboards` using the left-hand navigation pane
3. Open `Ingress Status` (if it isn't already open)

Look at the table of UAs that we added and note the addition of Chrome v137! You'll also note a new active alert `New UA Detected`!

Summary

Let's take stock of what we know:

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

And what we've done:

- Created several graphs to help quantify the extent of the problem
- Parsed the logs at ingest-time for quicker and more powerful analysis
- Created a dashboard to monitor our ingress proxy
- Create a SLO (with alert) to let us know if we ever return a significant number of non-200 error codes over time

- Geocoded client IP to associate location information with clients
- Created visualizations to help us visually locate clients and errors
- Parsed the user agent string to associate browser information with clients
- Determined, using client location and browser information, the root cause of our problem
- Created a table in our dashboard iterating User Agents in the wild
- 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 `attributes.client.ip` and `body.text` fields (to avoid implicitly leaking the `attributes.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 the first log record by clicking on the double arrow icon under `Actions` 3. Click on the `Table` tab in the flyout 3. Note that the `attributes.client.ip` and `body.text` fields are accessible and shown

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

1. Open the button label="Elasticsearch (Limited)" tab
2. Open the first log record by clicking on the double arrow icon under `Actions`
3. Click on the `Table` tab in the flyout
4. Note that the `attributes.client.ip` and `body.text` fields are not accessible and not shown

Let's change permissions and see what happens:

1. Open the button label="Elasticsearch" tab
2. Go to `Management` > `Stack Management` > `Security` > `Roles` using the left-hand navigation pane
3. Find the role `limited_viewer` and click on the pencil icon to the right of that row to edit it
4. For Indices `logs-proxy.otel-default`, update `Denied fields` to remove `body.text` (it should only contain `attributes.client`
5. Click `Update role`

Now let's ensure our limited user has access to a redacted `body.text`.

1. Open the button label="Elasticsearch (Limited)" Instruqt tab
2. Close the open log record flyout
3. Run the search query again
4. Open the first log record by clicking on the double arrow icon under `Actions`
5. Note that `attributes.client.ip` is not accessible, but `body.text` is

Redaction

While we've removed access to the `attributes.client.ip` field for the limited viewer, the client's IP address is still visible in `body.text`. Fortunately, Elasticsearch has a processor easily and automatically redact such PII information.

1. Open the button label="Elasticsearch" tab
2. Go to `Streams` using the left-hand navigation pane
3. Select `logs-proxy.otel-default` from the list of Streams.
4. Select the `Processing` tab
5. Select `Create processor` from the menu `Create`
6. Select the `Manual pipeline configuration` Processor
7. Set the `Ingest pipeline processors` field to:

```
[
  {
    "redact": {
      "field": "body.text",
      "patterns": [
        "%{IP:client_ip}"
      ],
      "ignore_missing": true,
      "ignore_failure": true
    }
```

```
  }
]
```

8. Click `Create`
9. Click `Save changes` in the bottom-right
10. Click `Confirm changes` in the resulting dialog

    [!NOTE] This redaction will apply to ALL roles, not just the limited viewer

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
```

1. Open the first log record by clicking on the double arrow icon under `Actions`
2. Click on the `Table` tab in the flyout
3. Note that any presence of the client's ip address in `body.text` has been redacted as `<client_ip>`, yet non-limited viewers will still be able to see client ip explicitly in the field `attributes.client.ip`

Summary

Let's take stock of what we know:

- a small percentage of requests 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 several graphs to help quantify the extent of the problem
- Parsed the logs at ingest-time for quicker and more powerful analysis
- Created a dashboard to monitor our ingress proxy
- Create a SLO (with alert) to let us know if we ever return a significant number of non-200 error codes over time
- Geocoded client IP to associate location information with clients
- Created visualizations to help us visually locate clients and errors
- Parsed the user agent string to associate browser information with clients
- Determined, using client location and browser information, the root cause of our problem
- Created a table in our dashboard iterating User Agents in the wild
- 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 Redaction to remove IP addresses from the log message body

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 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 and alerts
- Using Maps to visualize geographic information
- Scheduling dashboard reports
- Setting up a Pivot Transform and Alert
- Setting up RBAC

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 Status.

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