Parsing with Streams

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.

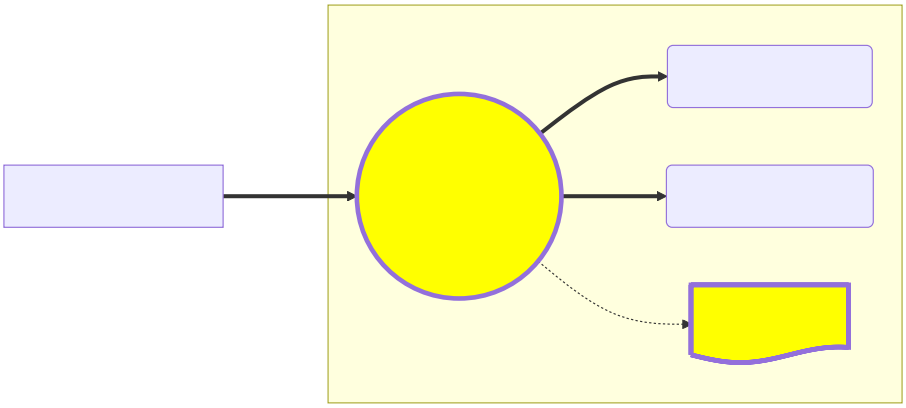

Figure 1: proxy_arch.mmd.svg

If you are generally familiar with SQL or other piped query languages, you will be right at home with ES|QL, Elastic's modern piped query language.

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 happening, but we don't know if they are successful or failing. Before we spend time parsing our logs, let's just quickly

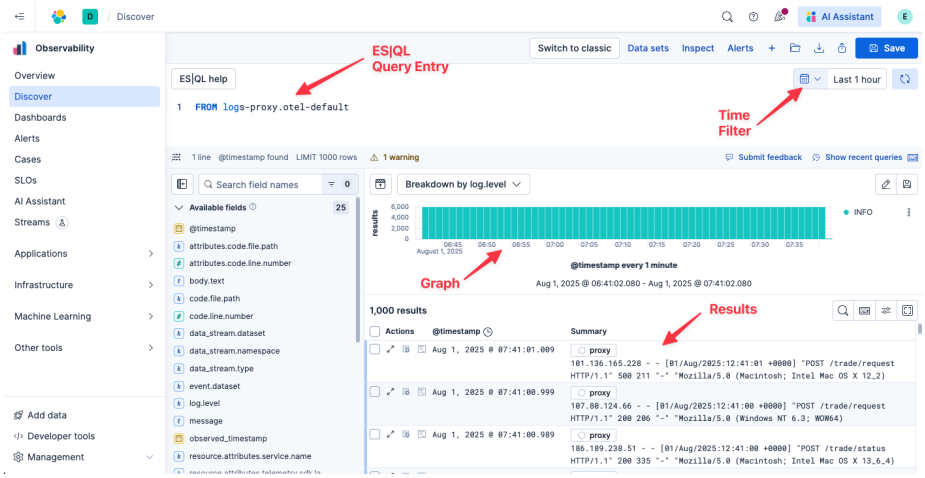

Figure 2: discover.svg

check for common “500” errors in our nginx logs.

Execute the following query:

```
FROM logs-proxy.otel-default
| WHERE MATCH(body.text, "500")
```

If we didn’t find “500”, we could of course add additional MATCH statements, like `OR MATCH(body.text, "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.

Does it affect 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
| STATS bad=COUNT() WHERE MATCH(body.text, "500"), good=COUNT() WHERE MATCH(body.text, "200"
```

That’s good: clearly this issue is affecting only some users.

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.

Execute the following query:

```
FROM logs-proxy.otel-default
| STATS bad=COUNT() WHERE MATCH(body.text, "500"), good=COUNT() WHERE MATCH(body.text, "200"
```

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
| STATS bad=COUNT() WHERE MATCH(body.text, "500"), good=COUNT() WHERE MATCH(body.text, "200"
| EVAL bad = COALESCE(TO_INT(bad), 0) // set bad=0 for time buckets with no bad entries
| SORT minute ASC
| CHANGE_POINT bad ON minute
| WHERE type IS NOT NULL
| KEEP type, minute, bad
```

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-structure 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 https://grokdebugger.com to help.

If you aren't well versed, 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. Because the AI Assistant may use slightly different field names, please copy and use the following instead of what was generated to ensure consistency throughout this workshop.

```
FROM logs-proxy.otel-default
| GROK message "%{IPORHOST:client_ip} %{USER:ident} %{USER:auth} \\[%{HTTPDATE:timestamp}\\]
| KEEP timestamp, client_ip, http_method, request_path, status_code, user_agent
```

Now close the flyout and execute the generated ES|QL.

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 message "%{IPORHOST:client_ip} %{USER:ident} %{USER:auth} \\[%{HTTPDATE:timestamp}\\]
| WHERE status_code IS NOT NULL
| EVAL @timestamp = DATE_PARSE("dd/MMM/yyyy:HH:mm:ss Z", timestamp)
| 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 against the `user_agent` field.

Execute the following query:

```
FROM logs-proxy.otel-default
| GROK message "%{IPORHOST:client_ip} %{USER:ident} %{USER:auth} \\[%{HTTPDATE:timestamp}\\]
| WHERE status_code IS NOT NULL
| EVAL @timestamp = DATE_PARSE("dd/MMM/yyyy:HH:mm:ss Z", timestamp)
| WHERE TO_INT(status_code) == 500
| STATS bad = COUNT() BY user_agent
```

Unfortunately, the unparsed user_agent field is too noisy to really be useful for this kind of analysis. We could try to write a GROK expression to parse `user_agent`, but in practice, it is too complicated (it requires translation in addition to parsing). Let's put a pin in this topic and revisit it in a bit.

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 message "%{IPORHOST:client_ip} %{USER:ident} %{USER:auth} \\[%{HTTPDATE:timestamp}\\]
| WHERE status_code IS NOT NULL
| EVAL timestamp = DATE_PARSE("dd/MMM/yyyy:HH:mm:ss Z", timestamp)
| STATS status = COUNT() BY status_code, minute = BUCKET(timestamp, "1 min")
```

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.

1. Click on the Disk icon in the upper-left of the resulting graph
2. Add to a new dashboard
3. Save the new dashboard as `Ingress Proxy`

Setting up a simple alert

We could at this point create a simple alert to notify us whenever a `status_code` >= 400 is received:

Execute the following query:

```
FROM logs-proxy.otel-default
| GROK message "%{IPORHOST:client_ip} %{USER:ident} %{USER:auth} \\[%{HTTPDATE:timestamp}\\]
| 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. 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 status code over time * 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. Notably, this is a pretty typical workflow with Elastic: start parsing your logs on-demand with ES|QL and when you find a consistent pattern which you believe will be of greater value if always parsed, move to ingest-time parsing. This, as you will see, unlocks another set of powerful Elastic log analytics.

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 Streams
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.m
```

7. Click `Accept`
8. 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 better 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.stat
```

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.

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. Add to a existing dashboard "Ingress Proxy"
3. Click `Save and go to dashboard`
4. Click `Save` in the upper-right

Creating a SLO

Remember that alert we created? Now that we are parsing these fields at ingest-time, we can create a proper SLO which allows for a small amount of errors before we get someone 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 : 200` (if this field isn't available, refresh the Instruqt virtual browser tab)
7. Set `Total query` to `http.response.status_code :*`
8. Set `Group by` to `http.request.url.path`
9. Set `Duration` to `7 days`
10. Set `Target / SLO (%)` to `99.999`
11. Set `SLO Name` to `Ingress Status`
12. Click `Create SLO`
13. Click on your newly created SLO `Ingress Status`
14. Under the `Actions` menu in the upper-right, select `Create new alert 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.

13. Click `Next`
14. (could add an action)
15. Click `Next`
16. Click `Create rule`

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 status code over time * 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 — slug: geo id: wex8hob0j9kz type: challenge title: Analyzing by GEO tabs: - id: eqsvxidyavvw title: Elasticsearch type: service hostname: kubernetes-vm path: /app/streams port: 30001 - id: enj1oldehuby title: Terminal type: terminal hostname: kubernetes-vm difficulty: basic timelimit: 600 enhanced_loading: false — 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`

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 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
```

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 `TW` (Taiwon) 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. Add to a existing dashboard "Ingress Proxy"
3. Click `Save and go to dashboard`
4. Click `Save` 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`
17. Select existing dashboard `Ingress Status`
18. Click `Save and go to dashboard`

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 `TW` region

And what we've done: * Created a Dashboard showing status code over time * 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 — slug: ua id: y89elnycbugj type: challenge title: Analyzing by User Agent tabs: - id: b6ohgb7enbox title: Elasticsearch

type: service hostname: kubernetes-vm path: /app/streams port: 30001 - id: tphygyq6udbv title: Terminal type: terminal hostname: kubernetes-vm difficulty: basic timelimit: 600 enhanced_loading: false — 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 == 200 OR NULL), bad = COUNT(http.response.st
| 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 == 200 OR NULL), bad = COUNT(http.response.st
| 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:

```
FROM logs-proxy.otel-default
| WHERE client.geo.country_iso_code IS NOT NULL AND user_agent.version IS NOT NULL AND http.
| EVAL version_major = SUBSTRING(user_agent.version,0,LOCATE(user_agent.version, ".")-1)
| 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 `TW` region! Quite possibly, Google has rolled out a specialized or canary version of their browser first in the `TW` 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 `TW` region * the errors occur only with browsers based on Chrome v136

And what we've done: * Created a Dashboard showing status code over time * 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 — slug: reporting id: lxkd6xcqmk0m type: challenge title: Reporting tabs: - id: ijfyyxmq23sz title: Elasticsearch type: service hostname: kubernetes-vm path: /app/discover#/?_g=(filters:!(),query:(language:kuery,query:''),refreshInterval:(pause:!t,value:60000),time:(from 1h,to:now))&_a=(breakdownField:log.level,columns:!(),dataSource:(type:esql),filters:!(),hideChart:!f,interval:auto proxy.otel-default'),sort:!(!('@timestamp',desc))) port: 30001 - id: kkeiiypxhfht title: Terminal type: terminal hostname: kubernetes-vm difficulty: basic

timelimit: 600 enhanced_loading: false — Now that we know what happened, let's try to be sure this never happens again.

Generating a table of user agents

One thing that would be helpful is to keep track of new User Agents as they appear in the wild.

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.full IS NOT NULL
| STATS @timestamp.min = MIN(@timestamp), @timestamp.max = MAX(@timestamp) BY user_agent.ful
```

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.ful
| EVAL first_ts = LEAST(@timestamp.min)
| STATS client.geo.country_iso_code = TOP(client.geo.country_iso_code, 1, "desc"), user_agen
| SORT @timestamp.min DESC
| KEEP client.geo.country_iso_code, user_agent.full, @timestamp.min, @timestamp.max
```

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 = CONCAT(user_agent.name, " ", SUBSTRING(user_agent.versio
| STATS @timestamp.min = MIN(@timestamp), @timestamp.max = MAX(@timestamp) BY user_agent.nam
| EVAL first_ts = LEAST(@timestamp.min)
```

```
| STATS client.geo.country_iso_code = TOP(client.geo.country_iso_code, 1, "desc"), user_agen
| SORT @timestamp.min DESC
| LOOKUP JOIN ua_lookup ON user_agent.name_and_vmajor
| WHERE release_date IS NOT NULL
| KEEP release_date, user_agent.name_and_vmajor, client.geo.country_iso_code, @timestamp.min
```

Wow! This is great! But maintaining that `ua_lookup` index looks like a lot of work. Fortunately, Elastic makes it possible to leverage an external Large Language Model 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
| EVAL first_ts = LEAST(@timestamp.min)
| STATS client.geo.country_iso_code = TOP(client.geo.country_iso_code, 1, "desc"), user_agen
| SORT @timestamp.min DESC
| LIMIT 25
| EVAL prompt = CONCAT(
   "when did this version of this browser come out? output only a version of the format mm/d
   "browser: ", user_agent.full
  ) | COMPLETION release_date = prompt WITH openai_completion
| EVAL release_date = DATE_PARSE("MM/dd/YYYY", release_date)
| KEEP release_date, client.geo.country_iso_code, user_agent.full, @timestamp.min, @timestam
```

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

Yes! 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`

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. Click `Schedule exports`
3. Click `Schedule export`

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. Set the `Transform ID` to `user_agents` 9. Set `Time field` to `@timestamp.min` 10. Set `Continuous mode` 11. Click `Next` 12. Click `Create and start`

Let's create a new alert which will fire when

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 `Related dashboards` to `Ingress Proxy`
11. 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`
4. Look at the table of UAs that we added and note the addition of Chrome 137!

Let's see if we fired an alert:

1. Navigate to `Alerts`
2. Note the active alert

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 `TW` region * the errors occur only with browsers based on Chrome v136

And what we've done: * Created a Dashboard showing status code over time * 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 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— slug: pii id: nxo4fxk0dray type: challenge title: Protecting Data tabs: - id: anstrmekrtc2 title: Elasticsearch type: service hostname: kubernetes-vm path: /app/discover#/?_g=(filters:!(),query:(language:kuery,query:''),refreshInterval:(pause:!t,value:60000),time:(from 1h,to:now))&_a=(breakdownField:log.level,columns:!(),dataSource:(type:esql),filters:!(),hideChart:!f,interval:aut proxy.otel-default'),sort:!(!('@timestamp',desc))) port: 30001 - id: qxodixi0jlll title: Elasticsearch (Limited) type: service hostname: kubernetes-vm path: /app/discover#/?_g=(filters:!(),query:(language:kuery,query:''),refreshInterval:(pause:!t,value:60000),time:(from 1h,to:now))&_a=(breakdownField:log.level,columns:!(),dataSource:(type:esql),filters:!(),hideChart:!f,interval:aut proxy.otel-default'),sort:!(!('@timestamp',desc))) port: 30002 custom_request_headers: - key: Authorization value: Basic bGltaXRlZF91c2VyOmVsYXN0aWM= - id: poj3poztpezq title: Terminal type: terminal hostname: kubernetes-vm difficulty: basic timelimit: 600 enhanced_loading: false — Sometimes our data contains PII information which needs to be kept to a need-to-know basis and only for a given 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. 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 `TW` region * the errors occur only with browsers based on Chrome v136

And what we've done: * Created a Dashboard showing status code over time * 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 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