SuperDemo

Setup

All of the following technologies are enabled in this environment. As time allows, I will be adding additional scripts for demonstration of specific features (linked to this ToC). In the interim, please feel free to explore on your own. All of the features iterated below are enabled in this demo.

- Agentic RCA
- Logs
- Metrics
- Tracing
- Workflows

Supporting slides (where available) can be found here .

Agentic RCA

Alert Correlation, Agentic Root Cause Analysis, and HIL Remediation

Goals

- Show fully agentic alert correlation and multi-signal RCA with context

Technical Setup

Perform these steps before you start the demo.

Steady-State

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Alerts`
3. Wait until there are no active alerts (service startup will trigger some failure alerts)

Generate Alerts

1. Open the button label="Trader" Instruqt tab
2. Navigate to `ERROR`
3. Open `DB`, select `Generate errors` and click `SUBMIT`
4. Open the button label="Elastic" Instruqt tab
5. Navigate to `Alerts`
6. Wait for new alerts to appear

Trigger Workflows

Be sure to wait until the new alerts (`APM Failure Rule`) fire before executing the `alert_process` workflow.

While you could trigger the `alert_process` workflow during the demo, I would recommend triggering it ahead of the demo rather than waiting for it to complete (which might take 5 minutes or so). During the demo, you can walk through the executed steps if the customer is interested in observing the workflows step-by-step.

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Workflows`
3. Open the `alert_process` workflow
4. Manually execute it

Once the `alert_process` completes alert correlation, it will automatically start `case_process` to perform the RCA.

Demo

Introduction

We have a set of microservices which implement a financial trading application: 1. Open the button label="Elastic" Instruqt tab 2. Navigate to `Applications` > `Service map`

Our database implements SQL data constraints which validate certain parameters, including that the number of shares being traded is a positive value: 1. Open the button label="K8s YAML" Instruqt tab 2. Navigate to `postgresql.yaml` 3. Note that `CREATE TABLE` (line 78) assigns constraints to specific fields

We are intentionally introducing errors into the system whereby all of the trades coming from the `EU` region will be trying to trade a negative number of shares and violate the database constraints.

Manual Debugging

Let's first debug this problem with minimal AI assistant as an SRE might do today:

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Alerts` (note the 3 possibly related alerts)
3. Choose one of the new alerts and select `View in app`
4. Scroll down to `Transactions` and select the `POST ...` transaction (it will be named differently depending on which service you selected)
5. Enter the following into the `Search transactions` bar at the top of the page:

```
status.code : "Error"
```

6. Scroll down to the waterfall graph and note the rippling error from database `INSERT` back up through the `trader` application
7. Click `View related error` on the failed `INSERT trades.trades` span
8. Click on the error message `ERROR: new row for relation "trades" violates check constraint "trades_share_price_ch`
9. Open `What's this error?` to show how the AI Assistant can help make sense of this error

While this was a helpful analysis, we still have 2 additional alerts to triage. They are possibly related, but we would have to repeat this exercise at least in part to confirm that hypothesis. Additionally, once we confirm these alerts are related, how can we communicate that association to other SREs.

Let's look at our dependency map to appreciate how a database error could trigger multiple alerts:

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Applications` > `Service map`
3. Note the dependency chain leading back from `postgresql`. Database validation errors will propagate backwards through `recorder-java`, `router`, and `trader`.

Agentic Alert Correlation

Imagine a larger system where a simple problem manifests in hundreds of alerts. To avoid triaging each alert individually, we first need to intelligently group related alerts to Cases. We will leverage Workflows, Agent Builder, and Cases.

(Optional) How does this work? We process each alert in a workflow (`alert_process`).

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Workflows`
3. Open the `alert_process` workflow

For each alert, we call the `alert_correlation` agent whose prompt and tools help it decide if an alert is related to an existing case or not.

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Agents`
3. Click `More` in the upper-right and select `View all agents`
4. Open the `Alert Correlation` agent

Note the custom instructions (prompt) which tell it how to correlate alerts.

5. Click the `Tools` tab (to the right of `Settings`)

Note that we give this agent access to only specific tools to help it focus on its task and execute in a definitive fashion.

Note that we leverage system topology (accessed via the `get_topologies` tool, derived from the `topology_builder` agent) to help understand dependencies when correlating alerts to Cases.

(Optional) Alert Correlation Workflow Walkthrough If the customer is interested in understanding OneWorkflow, you can optionally walk them through the execution steps. I would generally note here that we are working to bring alert correlation into the platform (e.g., we wouldn't expect a customer to write this workflow themselves).

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Workflows`
3. Open the `alert_process` workflow
4. Click `Executions`
5. Select the latest execution
6. Open `is_workflow_running` > `false` > `foreach_alert` > `0` > `correlate`
7. Click on `Input` and note that we are passing an alert to the `alert_correlation` Agent (`body.input`)
8. Click on `Output` and note for the first alert that it couldn't find an existing case to correlate to (`response.message`)
9. Open `is_workflow_running` > `false` > `foreach_alert` > `0` > `new_or_update_case` > `true` > `create_new_case`
10. Note that we are creating a new case for this alert (`description`)
11. Open `is_workflow_running` > `false` > `foreach_alert` > `0` > `add_alert_to_case`
12. Note that we are adding this alert to the new case

The above creates the case and adds the first alert to it. Now let's look at how additional alerts might be correlated to this alert.

13. Open `is_workflow_running` > `false` > `foreach_alert` > `1` > `correlate`
14. Click on `Input` and note that we are passing an alert to the `alert_correlation` Agent (`body.input`)
15. Click on `Output` and note for the second alert that the Agent correlated this alert to the case we created above (`response.message`)
16. Open `is_workflow_running` > `false` > `foreach_alert` > `1` > `add_comment_to_case`
17. Click on `Input` and note that we are adding this alert to an existing case (`body.comment`)

In the next step, we will have the opportunity to see the reasoning steps the Agent took to correlate these alerts to the same case.

View alert correlation results

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Cases`
3. Open the newly created case
4. Click `Attachments` and note that all 3 related alerts were correlated to the same case
5. Click `Activity` and note that when each alert was added, a comment was automated appended indicating a summary of the alert and why it was correlated to this case
6. Click on the `Conversation` link in the comment for an added alert
7. In the AgentBuilder conversation, click on `Completed reasoning` to see all of the steps the agent took to correlate this alert to this case
8. Open up a tool call request/response to better understand the data the agent used to correlate the alert

Agentic Root Cause Analysis

Now that we've grouped related alerts to Cases, we can perform Root Cause Analysis on the case. We will leverage Workflows, Agent Builder, and Cases.

(Optional) How does this work? We process each case in a workflow (`case_process`).

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Workflows`
3. Open the `case_process` workflow

For each case, we call the `rca` agent whose prompt and tools help it perform root cause analysis of an issue.

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Agents`
3. Click `More` in the upper-right and select `View all agents`
4. Open the `rca` agent

Note the custom instructions (prompt) which tell it how to perform root cause analysis.

5. Click the `Tools` tab (to the right of `Settings`)

Note that we give this agent access to only specific tools to help it focus on its task and execute in a definitive fashion.

Note that we leverage knowledge (accessed via the `search_knowledgebase` tool) to understand if this issue is known. Note that we leverage lots of OOTB platform and observability tools to gather evidence.

(Optional) Root Cause Analysis Workflow Walkthrough If the customer is interested in understanding OneWorkflow, you can optionally walk them through the execution steps. I would generally note here that we are working to bring agentic RCA into the platform (e.g., we wouldn't expect a customer to write this workflow themselves).

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Workflows`
3. Open the `case_process` workflow
4. Click `Executions`
5. Select the latest execution
6. Open `is_workflow_running > false > foreach_case > 0 > is_case_unstable > false > if_needs_update > true > rca`
7. Click on `Input` and note that we are passing the case to the `rca` Agent (`body`)
8. Click on `Output` and note that we are getting back the root cause analysis (`response.message`)
9. Open `is_workflow_running > false > foreach_case > 0 > is_case_unstable > false > if_needs_update > true > add_response_to_existing_case`
10. Click on `Input` and note that we are appending the root cause analysis to the case (`body.comment`)

In the next step, we will have the opportunity to see the reasoning steps the Agent took to perform the root cause analysis.

View RCA results

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Observability > Cases`
3. Open the case
4. Click `Attachments` and note that all 3 related alerts were correlated to the same case
5. Click `Activity`
6. Click `Show more` if needed to show all of the case comments
7. Note that we appended each step of the reasoning of the RCA Agent to the case as a comment
8. Scroll down to the last comments in the case
9. Note the detailed Root Cause Analysis w/ dependency map, explanation of correlation, and analysis of logs, traces, and metrics
10. Note that the Agent matched this problem with a known issue
11. Where did this issue come from? Open the button label="GitHub Issues" Instruqt tab and open the corresponding GitHub issue.
12. Note also that the Agent recommended for us to restart the monkey service to resolve this issue

Continue the Investigation

Once the initial triage is done, SREs may want to deep-dive into the available telemetry to validate the findings of the Agent.

Validate the Investigation

1. Starting from the last step, at the bottom of the case, you will find a link to continue investigation or take remedial action
2. In the AgentBuilder conversation, click on `Completed reasoning` (at the top) to see all of the steps the agent took to correlate this alert to this case
3. Open up a tool call request/response to better understand the data the agent used to perform root cause analysis

Dig for further evidence

1. Enter the following into the `Ask anything` box at the bottom of the Agent

```
was there a spike in log rate when this issue started occurring?
```

2. Enter the following into the `Ask anything` box at the bottom of the Agent

```
can you update the case with the results of our log rate analysis?
```

After the agent is done, let's verify that the case was updated: 1. Open the button label="Elastic" Instruqt tab 2. Navigate to `Observability > Cases` 3. Open the case 4. Scroll down to the last comments in the case and note that it now includes our log rate analysis

Remediation

We intentionally make remediation a Human-In-the-Loop (HIL) activity, though of course we could have told our agent to automatically take this step if we wanted. You'll recall that the RCA analysis suggested we could remediate the issue by restarting the `monkey` service. Let's do it!

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Observability` > `Cases`
3. Open the case
4. Scroll down toward the bottom of the case and look for the link to continue investigation or take remedial action
5. Click on the link to enter the Agent
6. Enter the following into the `Ask anything` box at the bottom of the Agent

```
can you restart the monkey service?
```

After the agent has completed the task, let's verify that the service was restarted

1. Open the button label="Services Host" Instruqt tab
2. Enter the following `kubectl` command:

```
kubectl -n trading-na get pods
```

3. Note that the `monkey` pod was recently restarted

(Optional) How does this work? Remediation can be achieved by teaching the RCA Agent how to call a Workflow that in turn can call a remote command.

Let's have a look at the RCA Agent: 1. Open the button label="Elastic" Instruqt tab 2. Navigate to `Agents` (it may be hidden under the `...` menu) 3. Click on `More` in the upper-right and select `View all tools` 4. Enter the following into the search bar

```
remediation
```

5. Click on the `remediation_service_action` tool and note that it calls the `remediation_service_action` workflow

Let's have a look at the `remediation_service_action` workflow: 1. Open the button label="Elastic" Instruqt tab 2. Navigate to `Workflows` 3. Enter the following into the search bar

```
remediation
```

4. Click on the `remediation_service_action` workflow
5. Note that it makes a HTTP call into our services cluster
6. Click on `Executions`
7. Open the most recent execution
8. Click on `call_remote` and select `Input`
9. Note the http call to restart the `monkey` service

Logs

SQL Commentor

Typically, database audit logs cannot easily be associated with traces. Using SQL Commentor, however, it becomes possible to follow a trace all the way from user interaction down to the database audit log.

SQL Commentor is a library that can be used by Java applications making SQL calls (here, `recorder-java`). SQL Commentor will look for an active OpenTelemetry trace and append the appropriate `traceparent` header as a comment to the SQL query. Most SQL databases (including postgresql) will output the comment as part of the audit log!

1. Open the button label="Elasticsearch" tab
2. Navigate to `Applications` > `Service map`
3. Select `recorder-java`
4. Select transaction `POST /record`
5. Click on the `Logs` tab under `Trace sample`
6. Note the inclusion of logs from `postgresql`

How does this work?

We are loading the `recorder-java` service with a custom build of the java SQL commentor library.

1. Open the button label="K8s YAML" Instruqt tab

2. Navigate to `recorder-java.yaml`

The SQL commentor library will automatically add `trace.id` (if its available on the current context) to SQL commands as a comment. We are then parsing that `trace.id` out of the comments and putting it into a first-class `trace.id` field.

1. Open the button label="K8s YAML" Instruqt tab
2. Navigate to `postgres.yaml`

Note the `regex_parser` operator specific in the `io.opentelemetry.discovery.logs.postgresql/config` annotation.

OTTL Parsing

Our `router` service emits logs in JSON format.

1. Open the button label="Code" Instruqt tab
2. Navigate to `router/app.ts`

```
import { Logger } from "tslog";
const logger = new Logger({ name: "router", type: "json" });
```

Logs are emitted to stdout and written to disk by k8s. Let's have a look:

1. Open the button label="Services Host" Instruqt tab
2. Enter the following `kubectl` command:

```
kubectl -n trading-1 get pods
```

3. Note the `router` pod name as (used in the next step)
4. Enter the following `kubectl` command:

```
kubectl -n trading-1 logs <router-pod-name>
```

Note that we are writing out logs in JSON format.

We are then using OTTL to parse out the fields in these JSON logs:

1. Open the button label="OTel Operator YAML" Instruqt tab
2. Navigate to `apm/serverless.yaml`
3. Search for `transform/parse_json_body`

Note the OTTL used to parse the JSON.

Let's examine the results.

1. Open the button label="Elasticsearch" tab
2. Navigate to `Discover` > `ES|QL`
3. Search for:

```
FROM logs-*
| WHERE service.name == "router"
```

4. Open a record to examine it Metrics ===

Metrics Discovery

The goal of this demo is to demonstrate that the metrics experience in Elastic is optimized to derive value with few clicks.

Metrics Discovery in Grafana (optional)

> [!NOTE] Typically, you would demo the Grafana comparison only if the customer is an existing Prometheus/Grafana user

1. Open the button label="Grafana" tab
2. Navigate to `Drilldown` > `Metrics`
3. Enter the following in the `Search metric` bar

```
http_requests_total
```

4. Click `Select`
5. Set `Breakdown` > `By label` to `region`
6. Click the `Add to dashboard` control in the upper-right of the `http_requests_total` graph
7. Click `Open dashboard` in the `Add to dashboard` dialog

(add alert rule?)

Metrics Discovery in Kibana

Now let's mirror that experience in Kibana.

1. Open the button label="Elasticsearch" tab
2. Navigate to `Discover`
3. Enter `ES|QL` mode (if Discover is not yet in `ES|QL` mode)
4. Execute the following ES|QL:

```
TS metrics-*
```

Note how easy it is to quickly visualize a large set of metrics.

Now let's find our metric: 1. Enter the following in the `Search metric` bar

```
http_requests_total
```

Note that Kibana detected that this is a monotonically increasing counter and automatically applied a rate (derivative) transformation.

2. Click `No dimension selected` and select `region`
3. Click on the three vertical dots in the upper right of one of the `http_requests_total` graphs
4. Select `Copy to dashboard`
5. Click `New` in the `Save Lens visualization` dialog
6. Click `Save and go to Dashboard`

(add alert rule?)

PROMQL

(generally only applicable if customer is coming from prometheus/grafana)

PROMQL in Grafana

1. Open the button label="Grafana" tab
2. Navigate to `Explore`
3. Click `Code` toggle (on the right)
4. Enter the following into the `Enter a PromQL query...` field

```
sum by (region) (rate(http_requests_total[5m]))
```

5. Click `Run query`

PROMQL in Kibana

1. Open the button label="Elasticsearch" tab
2. Navigate to `Discover`
3. Enter `ES|QL` mode (if Discover is not yet in `ES|QL` mode)
4. Execute the following ES|QL:

```
PROMQL index=metrics-* start=?_tstart end=?_tend step=5m sum by (region) (rate(http_requests_total[5m]))
```

(note, you can copy and paste `sum by (region) (rate(http_requests_total[5m]))` from Grafana into Kibana for greater effect)

Note the same result as in Grafana.

OOTB OTel Dashboards

PostgreSQL

Dashboard

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Dashboards`
3. Open dashboard `[Metrics PostgreSQL OTel] Database Overview`

How does this work?

1. Open the button label="K8s YAML" Instruqt tab
2. Navigate to `postgresql.yaml`

Note the OTel Collector configuration with the `postgresql` receiver.

MySQL

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Dashboards`
3. Open dashboard `[MySQL OTel] Overview`

How does this work?

To collect mysql metrics, we use a sidecar vanilla OTel Collector configured with the `mysql` receiver.

1. Open the button label="K8s YAML" Instruqt tab
2. Navigate to `mysql.yaml`

Note the OTel Collector configuration with the `mysql` receiver.

nginx

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Dashboards`
3. Open dashboard `[Metrics Nginx OTEL] Overview`

How does this work?

To collect mysql metrics, we use a sidecar vanilla OTel Collector configured with the `mysql` receiver.

1. Open the button label="K8s YAML" Instruqt tab
2. Navigate to `proxy.yaml`

Note that we use the creator receiver pattern to tell the daemonet OTel Collector to invoke the `nginx` receiver

ES|QL Analytics

Tracing

OTel Profiling

Setup

Do this before you begin the demo.

1. Open the button label="Trader" Instruqt tab
2. Navigate to `Test`
3. Open `Flags`, select `Test New Hashing Algorithm` and click `SUBMIT`

Dashboard

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Infrastructure` > `Hosts`
3. Select host `k3s`
4. Select `Dashboards` tab

How does this work?

1. Open the button label="OTel Operator YAML" Instruqt tab
2. Navigate to `profiling/profiler.yaml`

Note the OTel Collector configuration with the `profiling` receiver and `profilingmetrics` connector.Workflows

Basic

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Workflows`
3. Open the `hello_world` workflow
4. Run it
5. Walk through the execution steps

Calling an external REST API

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Workflows`
3. Open the `ip_geolocator` workflow
4. Run it
5. Walk through the execution steps

Calling an external REST API w/ conditionals and retries

1. Open the button label="Elastic" Instruqt tab
2. Navigate to `Workflows`
3. Open the `ip_geolocator_advanced` workflow
4. Run it
5. Walk through the execution steps
