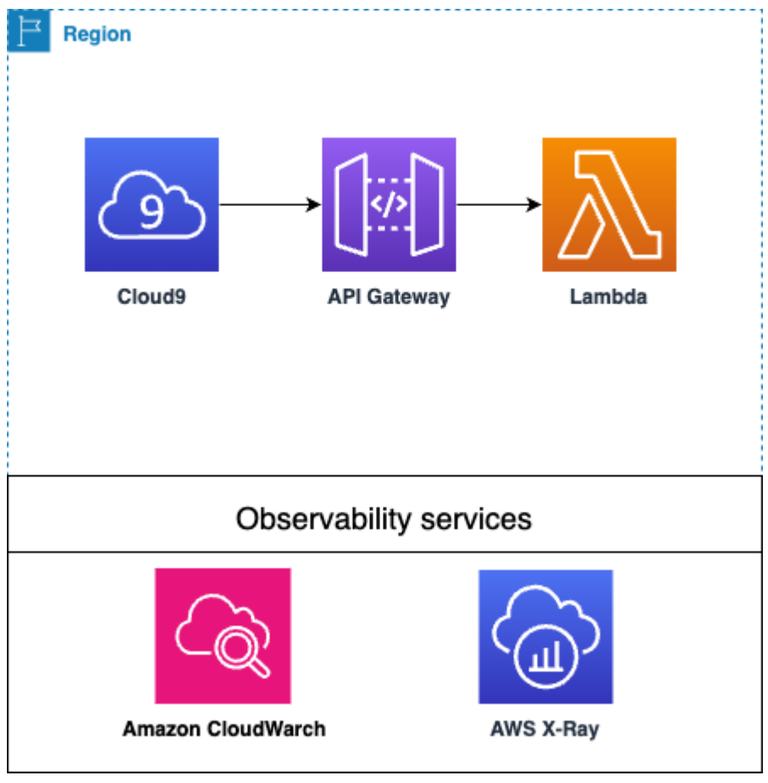
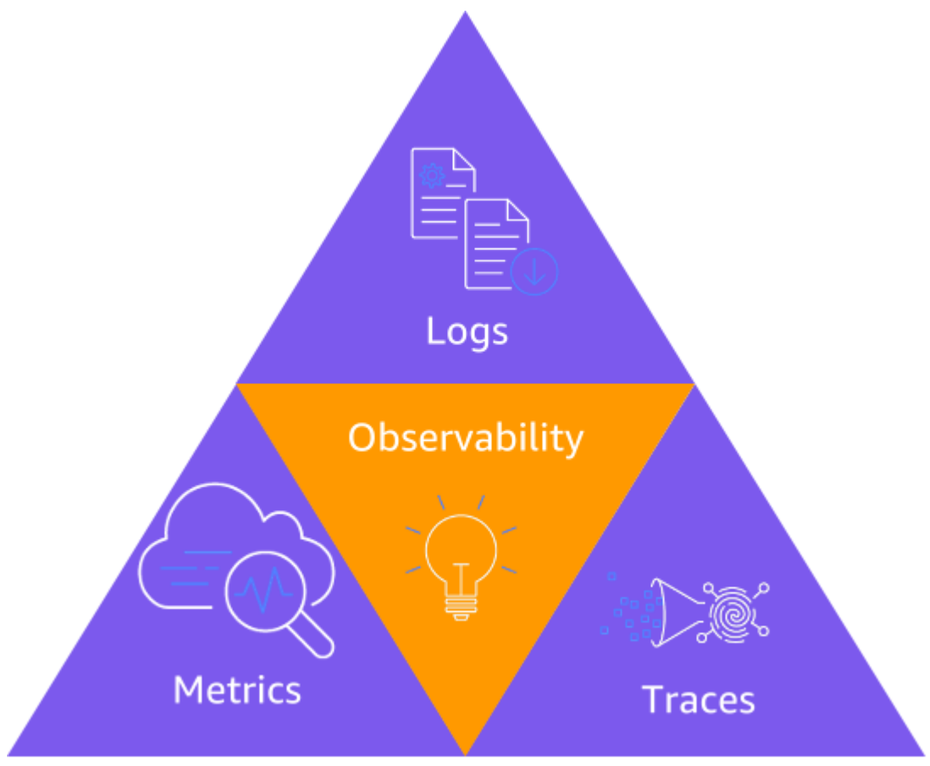
Module 4 - Observability in API Gateway



In this module you will learn how to leverage key metrics, logging and tracing features in order to gain observability into your API Gateway APIs. Gaining observability helps you detect, investigate, and remediate problems.



The three pillars of observability are:

- Metrics
- Logs
- Traces

Metrics are a numeric representation of data measured over intervals of time. Metrics provide data about the performance of your APIs and their backend integrations. You can think of a metric as a single variable used to monitor an aspect of your system. When you combine all these individual metrics, you are able to get insights as to how your application is performing over time.

Logs help you keep track of events that have occurred during the life cycle of an API request. Logs are time-stamped records that can include data such as request payloads, backend responses, errors, data transformations, or even who accessed your system at a certain time. Logs can be recorded in unstructured, semi-structured or structured formats.

Traces are representations of a series of causally related distributed events. Traces may encode end-to-end request flows through a distributed system.

Monitoring metrics, logs, and traces can help you understand the availability, performance and health of your APIs. AWS provides several tools you can use with API Gateway to monitor these three pillars of observability, this module will show you how to best utilise these.

Estimated Duration: 1 hour

Setup

1. Navigate to Cloud9 in your AWS console. Ensure that you are in the region where your cloudformation stack was launched.
2. Click **Open** on the **APIGatewayWorkshopWorkspace** environment.
3. The AWS Cloud9 environment comes with some AWS utilities pre-installed. Run the following command in your AWS Cloud9 terminal to verify that it contains an updated version of AWS SAM.

```
sam --version
```

4. Set up your AWS SAM Project Folder and Files for this module:

```
mkdir -p module-4-observability && cd module-4-observability/
```

5. Create the required files in this folder with the following command:

```
touch template.yaml openapi.yaml
```

- `template.yaml` - This file is the primary AWS SAM configuration file. AWS SAM templates are an extension of AWS CloudFormation templates, with some additional components that make them easier to work with. For the full reference for AWS CloudFormation templates, see AWS CloudFormation Template Reference in the AWS CloudFormation User Guide.
- `openapi.yaml` - This file is the OpenAPI definition file that will configure the structure of the API Gateway endpoint.

Create the Lambda function

In order to generate some metrics, logs and traces required by this module, we will be using an AWS Lambda Function as our backend integration.

1. In your `module-4-observability` folder, run the below command to create the folders and files needed to store our Lambda code.

```
mkdir -p handlers/code && cd handlers/code && touch generate-data.js
```

2. This Lambda function is what we will use to generate metric and log data. To do this, the code has a sleep timer at random intervals in order to force some requests to timeout, as the timeout value configured on the function will be set at 1 second. This will populate error metrics as well as success metrics. Review the code and then **copy/paste** it into the `generate-data.js` file.

```
exports.handler = async (event) => {
      //generate random number to set as sleep duration
      const randomDuration = getRandomArbitrary(250, 1500)
      //sleep function to force sporadic timeouts
      await new Promise(r => setTimeout(r, randomDuration));

      //if function executes successfully, return HTTP 200 back to client
      const response = {
        statusCode: 200,
        headers: {
           'Content-Type': 'application/json',
        },
        body: JSON.stringify({
           message: 'Hello from Lambda!'
        }),
      };
   console.log('Response:', JSON.stringify(response));
   return response;
};
//logic to generate the random value
function getRandomArbitrary(min, max) {
  return Math.random() * (max - min) + min;
}
```

Use AWS SAM and OpenAPI to create our API Gateway and backend integrations

1. Using AWS Cloud9 console, return to the root folder `module-4/observability`
2. This code belongs in your SAM template file `template.yaml`
3. Review the code and then **copy/paste** it into the `template.yaml` file.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: >
  module4-observability: Sample SAM Template for module4-observability


Resources:
    # Our API Gateway API
    ObservabilityAPI:
        Type: AWS::Serverless::Api
        Properties:
            StageName: dev
            OpenApiVersion: 3.0.3
            DefinitionBody: # an OpenApi definition
                "Fn::Transform":
                    Name: "AWS::Include"
                    Parameters:
                        Location: "openapi.yaml"
            EndpointConfiguration:
                Type: REGIONAL

    #Lambda Function used to generate the required data points
    FunctionForDataPoints:
        Type: 'AWS::Serverless::Function'
        Properties:
            CodeUri: ./handlers/code
            Handler: generate-data.handler
            Timeout: 1
            Runtime: nodejs18.x
    # Execution Role for lambda functions
    LambdaExecutionRole:
        Type: AWS::IAM::Role
        Properties:
            AssumeRolePolicyDocument:
                Version: "2012-10-17"
                Statement:
                  - Effect: Allow
                    Principal:
                      Service:
                        - apigateway.amazonaws.com
                    Action:
                      - 'sts:AssumeRole'
            Policies:
                - PolicyName: AllowLambdaExec
                  PolicyDocument:
                    Version: "2012-10-17"
                    Statement:
                      - Effect: Allow
                        Action: 'lambda:InvokeFunction'
                        Resource: [
                            !GetAtt FunctionForDataPoints.Arn,
                        ]

Outputs:
    APIGatewayInvokeURL:
        Description: API Gateway Endpoint for logging example
        Value:
            Fn::Sub: https://${ObservabilityAPI}.execute-api.${AWS::Region}.amazonaws.com/dev/generate-data
    LogInsightsKeyword:
        Description: Keyword needed to filter for this api logs
        Value:
          Fn::Sub: API-Gateway-Execution-Logs_${ObservabilityAPI}/dev
```

Within the above SAM template, we are creating 3 AWS resources. These include our API Gateway API, the Lambda function needed to generate data points, and an execution role used by API Gateway to invoke the Lambda function. After a successful deployment, the template will output the invoke URL needed as part of this module, as well as the keyword needed for the CloudWatch logs insights section. Keep a note of these values.

The below code belongs in your OpenAPI definiton file `openapi.yaml`. Review the code then **copy/paste** it into the `openapi.yaml` file

```
1   openapi: "3.0.1"
2   info:
3     title: "module-4-observability"
4     version: "2023-04-10T16:25:39Z"
5   servers:
6   - url: "https://${ObservabilityAPI}.execute-api.${AWS::Region}.amazonaws.com/dev/generate-data"
7
8   paths:
9     /generate-data:
10      get:
11        x-amazon-apigateway-integration:
12                  httpMethod: "POST"
13                  credentials:
14                      Fn::GetAtt: [LambdaExecutionRole, Arn]
15                  type: "aws_proxy"
16                  uri:
17                      Fn::Sub: "arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-31/functions/${FunctionForDataPoints.Arn}/invocations"
18                  passthroughBehavior: "when_no_match"
```

This is a relatively simple Open API spec as we are only creating 1 API with a single resource/method. Line **(9)** is where we are defining the name for the resource path, line **(10)** for the method, and lines **(11-18)** is where we are defining the Lambda function as the integration.

Deploy the project

1. To deploy the required API Gateway resources to your AWS account, run the following commands from the application root `module-4/observability`, where the `template.yaml` file for the sample application is located:

```
sam build && sam deploy --guided
```

The first time that you run the `sam deploy --guided` command, AWS SAM starts an AWS CloudFormation deployment. In this case, you need to say what are the configurations that you want SAM to have in order to get the guided deployment. You can configure it as below.

- **Stack Name**: `module-4-observability`
- **AWS Region**: Put the chosen region to run the workshop. **e.g.** `us-east-1`
- **Confirm changes before deploy**: `Y`
- **Allow SAM CLI IAM role creation**: `Y`
- **Disable rollback**: `N`
- **Save arguments to configuration file**: `Y`
- **SAM configuration file** and **SAM configuration environment** leave blank

```
Setting default arguments for 'sam deploy'
=========================================
Stack Name [sam-app]: module-4-observability
AWS Region [eu-west-1]:
#Shows you resources changes to be deployed and require a 'Y' to initiate deploy
Confirm changes before deploy [y/N]: Y
#SAM needs permission to be able to create roles to connect to the resources in your template
Allow SAM CLI IAM role creation [Y/n]: Y
#Preserves the state of previously provisioned resources when an operation fails
Disable rollback [y/N]: N
Save arguments to configuration file [Y/n]: Y
SAM configuration file [samconfig.toml]:
SAM configuration environment [default]:
```

2. After configuring the deployment, AWS SAM will display assets that will be created. But first, it will automatically upload the template to a temporary bucket it creates. Then, it will ask you to confirm the changes. Type `y` to confirm.
3. Once the deployment has been successful, you will see an **'Outputs'** section that contains the API Gateway invoke URL and the log group name needed for the module - **take note of both of these values** as you will need them throughout the module.

```
CloudFormation outputs from deployed stack
------------------------------------------------------------------------------------
Outputs
------------------------------------------------------------------------------------
Key                 LogInsightsKeyword
Description         Keyword needed to filter for this api logs
Value               API-Gateway-Execution-Logs_mfwerierj7/dev

Key                 APIGatewayInvokeURL
Description         API Gateway Endpoint for logging example
Value               https://mfwerierj7.execute-api.eu-west-1.amazonaws.com/dev/generate-data
------------------------------------------------------------------------------------
```

Success! You have successfully deployed the required AWS resources needed to complete the module. Click Next to begin.

Monitor API executions with Amazon CloudWatch Metrics

Within this section, we will learn how to understand and interpret the API Gateway CloudWatch metrics in order to gain visibility into how our API is performing over a period of time. By configuring the IAM role in the initial setup module of this workshop, we gave API Gateway permission to publish both metrics and logs to CloudWatch.

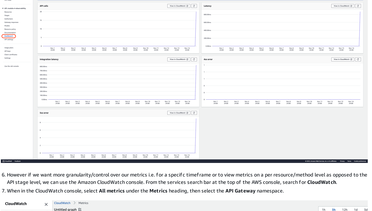
Generate metrics for our API

1. Open a terminal in your Cloud9 environment
2. From the outputs section of the SAM deployment, grab the URL from **API Gateway Invoke URL**.
3. Run the below command to send some requests to your API. Be sure to replace **<API Gateway Endpoint#>** with the value retrieved above.

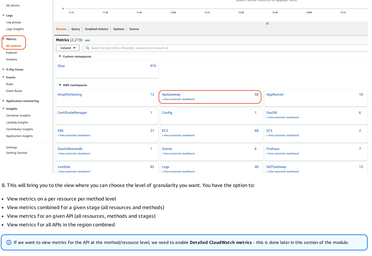
> This command will send 20 requests to your API in order to generate multiple different datapoints. The output should be a mix of 200 and 502 status codes to simulate success and failure data points. This command will take approximately **20-30** seconds to complete.

```
for i in {1..20}; do curl -s -o /dev/null -w "%{http_code}\n" <API Gateway Endpoint#>; done
```

4. The API Gateway console provides a built-in dashboard that can be used to get a quick glance into how your API is performing. This dashboard will show metrics from the past 2 weeks of activity and provides information such as the number of API calls, average latency of requests, the average latency of your integration and finally the number of 4XX and 5XX errors. You can view these metrics for each stage of your API.
5. To access this dashboard, navigate to the API Gateway console and select your API that was created as part of the SAM deployment. This should be named `module-4-observability`. From the menu on the left select **Dashboard**, from this you will see a number of graphs showcasing these metrics.



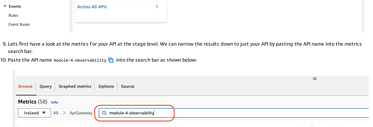
6. However if we want more granularity/control over our metrics i.e. for a specific timeframe or to view metrics on a per resource/method level as opposed to the API Usage level, we can use the Amazon CloudWatch console. From the services search bar at the top of the AWS console, search for **CloudWatch**.
7. When in the CloudWatch console, select **All metrics** under the **Metrics** heading, then select the **API Gateway** namespace.



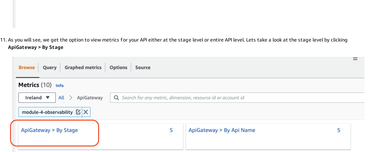
8. This will bring you to the view where you can choose the level of granularity you want. You have the option to:

- View metrics on a per resource per method level
- View metrics combined for a given stage (all resources and methods)
- View metrics for an given API (all resources, methods and stages)
- View metrics for all APIs in the region combined

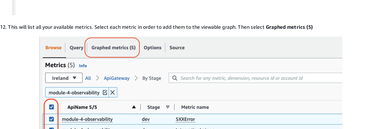
> If we want to view metrics for the API at the method/resource level, we need to enable **Detailed CloudWatch metrics** - this is done later in this section of the module.



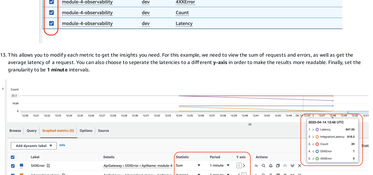
9. Lets first have a look at the metrics for your API at the stage level. We can narrow the results down to just your API by pasting the API name into the metrics search bar.
10. Paste the API name `module-4-observability` into the search bar as shown below:



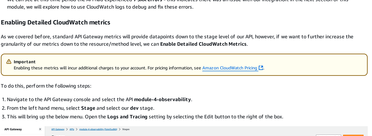
11. As you will see, we get the option to view metrics for your API either at the stage level or entire API level. Lets take a look at the stage level by clicking **ApiGateway > By Stage**



12. This will list all your available metrics. Select each metric in order to add them to the viewable graph. Then select **Graphed metrics (5)**



13. This allows you to modify each metric to get the insights you need. For this example, we need to view the sum of requests and errors, as well as get the average latency of a request. You can also choose to seperate the latencies to a different **y-axis** in order to make the results more readable. Finally, set the granularity to be **1 minute** intervals.



14. As you can see, if you hover your cursor over a specific interval on the graph, in this case the time **12:48 UTC**, we can gain observability into that period of time. We can see at this time period the API had experienced **7 5XX errors** - this indicates there was an issue with our integration. In the next section of this module, we will explore how to use CloudWatch logs to debug and fix these errors.

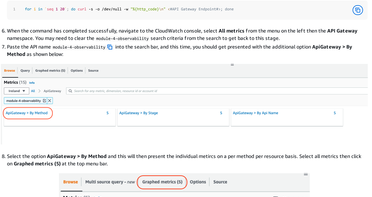
Enabling Detailed CloudWatch metrics

As we covered befor, standard API Gateway metrics will provide datapoints down to the stage level of our API, however, if we want to further increase the granularity of our metrics down to the resource/method level, we can **Enable Detailed CloudWatch Metrics.**

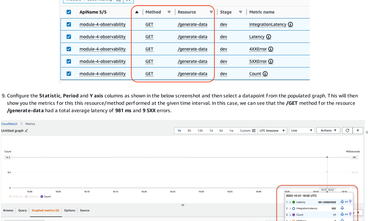
> **Important**
> Enabling these metrics will incur additional charges to your account. For pricing information, see Amazon CloudWatch Pricing

To do this, perform the following steps:

1. Navigate to the API Gateway console and select the API **module-4-observability**.
2. From the left hand menu, select **Stage** and select our **dev** stage.
3. This will bring up the below menu. Open the **Logs and Tracing** setting by selecting the Edit button to the right of the box.



4. Select the tick box **Detailed metrics** and be sure to save your changes.



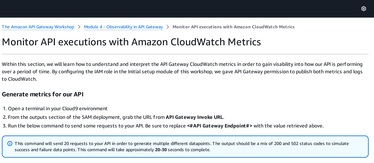
5. Once these additional metrics have been enabled, navigate back to your Cloud9 environment and run the same script as before to invoke your API and populate these metrics into CloudWatch.

```
for i in {1..20}; do curl -s -o /dev/null -w "%{http_code}\n" <API Gateway Endpoint#>; done
```

6. When the command has completed successfully, navigate to the CloudWatch console, select **All metrics** from the menu on the left then the **API Gateway** namespace. You may need to clear the `module-4-observability` search criteria from the search to get back to this stage.
7. Paste the API name `module-4-observability` into the search bar, and this time, you should get presented with the additional option **ApiGateway > By Method** as shown below:

8. Select the option **ApiGateway > By Method** and this will then present the individual metrics on a per method per resource basis. Select all metrics then click on **Graphed metrics (5)** at the top menu bar.

9. Configure the **Statistic, Period** and **Y axis** columns as shown in the below screenshot and then select a datapoint from the populated graph. This will then show you the metrics for this this resource/method performed at the given time interval. In this case, we can see that the **/GET** method for the resource **/generate-data** had a total average latency of **981 ms** and **5 5XX** errors.



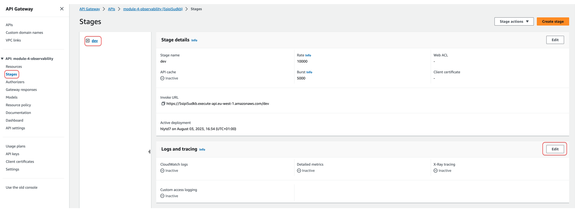
Record API execution history with Amazon CloudWatch Logs

Within this section of the module, we will focus on the logging pillar to understand how we can use logs generated by API Gateway to get insights into the data flowing through our API, as well as use these to troubleshoot issues our API may experience.

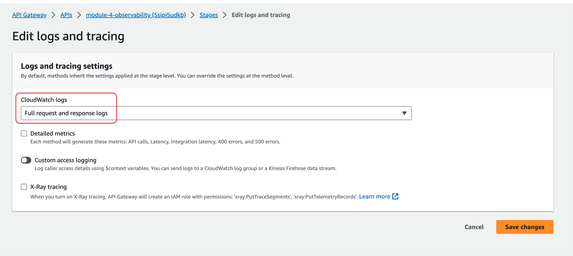
Enable logging on our API

Before we can view logs produced by our API, we need to first enable them. To do this:

1. Navigate to the API Gateway console, and select the API **module-4-observability**
2. From the left hand menu, select **Stages** and select our **dev** stage.
3. This will bring up the below menu. Open the **Logs and Tracing** setting by selecting the Edit button to the right of the box.



4. Then, from the **CloudWatch logs** dropdown menu, select **Full Request and Response Logs**. This is the most verbose level of logging as it will log all request/response payloads as well as any data transformations for every request. Be sure to save your changes.



5. Be sure to re-deploy the API to ensure all changes are reflected.

Generate logs for our API

Now that we have logging enabled, its time to generate some logs. For this, we can use the same command and endpoint that we used to generate metrics.

You can get the endpoint from the outputs section of the SAM deployment for this module, it should be under **APIGatewayInvokeURL**. Be sure to replace **<#API Gateway Endpoint#>** with the value retrieved above.

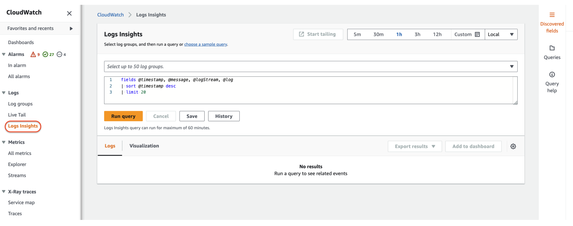
```
for i in `seq 1 20` ; do curl -s -o /dev/null -w "%{http_code}\n" <#API Gateway Endpoint#>; done
```

As seen previousy, this command will generate a number of 200 and 502 HTTP status codes, and so we can use these to better understand why some requests are succeeding and others are not.

View generated execution logs

Once the curl command has been sucessfully executed, navigate to the Amazon CloudWatch console. From here there is 2 options. The first is to go to the log group where all execution logs reside. Within this, you can look through each log 1 by 1 until you find a log which returns the 502 status code. However, a more efficient approach is to use a CloudWatch Insights query in order to filter through all log streams and only return the logs which produced a 502 HTTP response. In order to run a CloudWatch insights query:

- Navigate to the Amazon CloudWatch console
- From the menu on the left, under the **Logs** heading, select **Logs Insights**



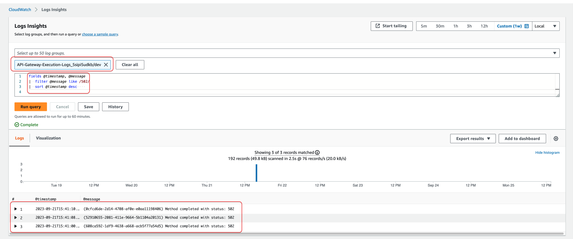
This will present you with an interface where you can select the log groups you want to query, as well as a query interface you can use to filter the data within the logs.

To filter the log group associated with your API Gateway for this module, you can grab the value from the Outputs section of the CloudFormation stack named **module-4-observability** under the **LogInsightKeyword** heading.

Then, paste the below insights query that will filter our logs for any log lines that produced a 502 status code.

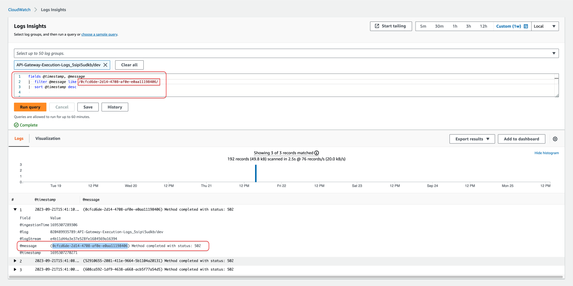
```
fields @timestamp, @message
| filter @message like /Method completed with status: 502/
| sort @timestamp desc
```

The results of this query should return a number of log lines that shows our API errored with a 502 response.



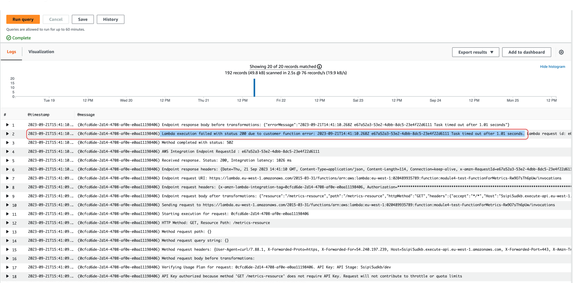
In order to dive deeper into these errors and understand what caused the 502 response, copy the request ID from one of the log lines as shown and run a second query to show all log lines associated with this failed API request. Be sure to replace **Request_ID** with the value from your returned log.

```
fields @timestamp, @message
| filter @message like /Request_ID/
| sort @timestamp desc
```



The output of this will be the full log for this API request. As you can see there is a lot of information within this log and it gives us great visability into the lifecycle of this request, as well as how data is being transformed at the API Layer.

However, the most important information here is what caused the API request to fail and we can see from the second line at the top, that this is due to the Lambda function execution failing due to a timeout error. This will allow you to then focus your efforts in troubleshooting the Lambda function and understand why its timing out.



Challenge

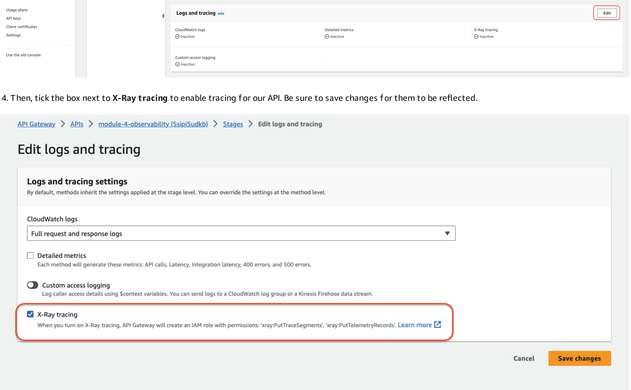
Debug failed executions with AWS X-Ray traces

Within this final section of the observability module, we will focus on the tracing pillar and how we can use AWS X-Ray to trace and analyze user requests as they travel through your Amazon API Gateway REST APIs to the underlying downstream services of your application.

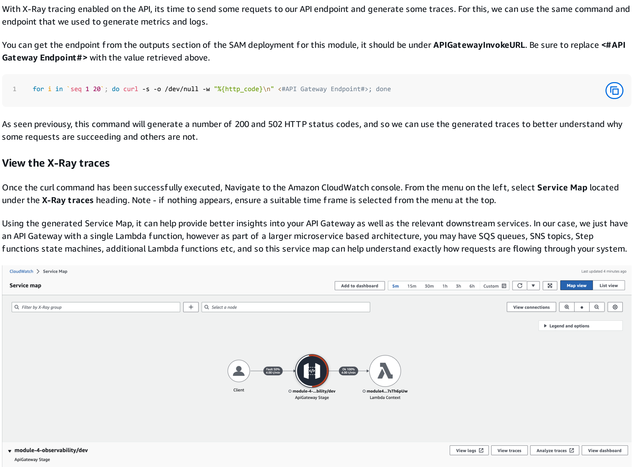
Enable tracing on your API

By default, X-Ray tracing is not enabled on our API stage, and so to be able to analyze and debug failed requests using traces, we first need to enable it. To do this:

1. Navigate to the API Gateway console, and select the API **module-4-observability**
2. From the left hand menu, select **Stages** and select our **dev** stage.
3. This will bring up the below menu. Open the **Logs and Tracing** setting by selecting the Edit button to the right of the box.



4. Then, tick the box next to **X-Ray tracing** to enable tracing for our API. Be sure to save changes for them to be reflected.



Generate X-Ray traces

With X-Ray tracing enabled on the API, its time to send some requests to our API endpoint and generate some traces. For this, we can use the same command and endpoint that we used to generate metrics and logs.

You can get the endpoint from the outputs section of the SAM deployment for this module, it should be under **APIGatewayInvokeURL**. Be sure to replace **<#API Gateway Endpoint#>** with the value retrieved above.

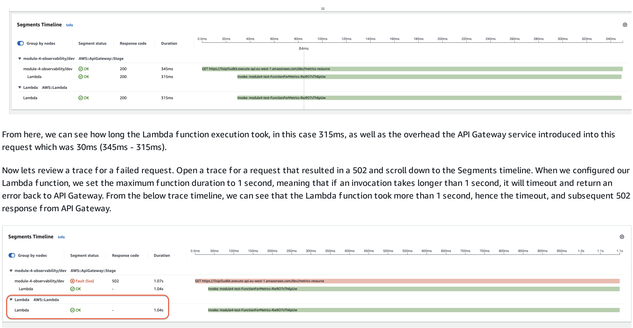
```
for i in `seq 1 20`; do curl -s -o /dev/null -w "%{http_code}\n" <#API Gateway Endpoint#>; done
```

As seen previously, this command will generate a number of 200 and 502 HTTP status codes, and so we can use the generated traces to better understand why some requests are succeeding and others are not.

View the X-Ray traces

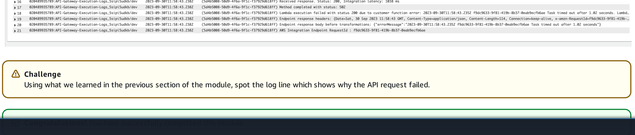
Once the curl command has been successfully executed, Navigate to the Amazon CloudWatch console. From the menu on the left, select **Service Map** located under the **X-Ray traces** heading. Note - if nothing appears, ensure a suitable time frame is selected from the menu at the top.

Using the generated Service Map, it can help provide better insights into your API Gateway as well as the relevant downstream services. In our case, we just have an API Gateway with a single Lambda function, however as part of a larger microservice based architecture, you may have SQS queues, SNS topics, Step functions state machines, additional Lambda functions etc, and so this service map can help understand exactly how requests are flowing through your system.



The service map shows that a certain percentage of our requests have resulted in an error, and so we can now dive deeper into the individual traces to understand why some requests are suceeding and others are failing. To do this, select the **ApiGateway Stage** node and then click the **View traces** button under the service map.

This will populate a query for you to show all traces associated with our API Gateway stage we have been invoking. You can of course edit this to get more granular into a specific resource or method on the API. Select the **Run query** button and it should retrive a number of traces for us to investigate. Again, if no traces appear, ensure you have selected a suitable time filter from the menu at the top.



As our test curl command resulted in a mix of 200, and 502 errors, this should be reflected within the returned traces as shown below. As part of the module, we are going to review first a trace that resulted in a successful request and also one that resulted in failure.

If we first select a trace that resulted in a 200, this will show us a much more detailed picture about the request and the latencies introduced by each service component of the microservice architecture. Once the trace is opened, scroll to below the generated trace map and review the **Segments timeline**.

From here, we can see how long the Lambda function execution took, in this case 315ms, as well as the overhead the API Gateway service introduced into this request which was 30ms (345ms - 315ms).

Now lets review a trace for a failed request. Open a trace for a request that resulted in a 502 and scroll down to the Segments timeline. When we configured our Lambda function, we set the maximum function duration to 1 second, meaning that if an invocation takes longer than 1 second, it will timeout and return an error back to API Gateway. From the below trace timeline, we can see that the Lambda function took more than 1 second, hence the timeout, and subsequent 502 response from API Gateway.

A great feature of X-Ray traces is that if you have execution logs enabled on the API Gateway, you can see the related logs associated with this trace and it can assist in debugging why the request failed. Because we enabled full request and response logs in the previous section, you will see the full execution log below the Segements timeline we just analyzed.

Challenge
Using what we learned in the previous section of the module, spot the log line which shows why the API request failed.