# 6 Automating operational tasks with Lambda

This chapter covers

- Creating a Lambda function to perform periodic health checks of a website
- Triggering a Lambda function with EventBridge events to automate DevOps tasks
- Searching through your Lambda function's logs with CloudWatch
- Monitoring Lambda functions with CloudWatch alarms
- Configuring IAM roles so Lambda functions can access other services

This chapter is about adding a new tool to your toolbox. The tool we're talking about, AWS Lambda, is as flexible as a Swiss Army knife. You don't need a virtual machine to run your own code anymore, because AWS Lambda offers execution environments for C#/.NET Core, Go, Java, JavaScript/Node.js, Python, and Ruby. All you have to do is implement a function, upload your code, and configure the execution environment. Afterward, your code is executed within a fully managed computing environment. AWS Lambda is well integrated with all parts of AWS, enabling you to easily automate operations tasks within your infrastructure. We use AWS to automate our infrastructure regularly, such as to add and remove instances to a container cluster based on a custom algorithm and to process and analyze log files.

AWS Lambda offers a maintenance-free and highly available computing environment. You no longer need to install security updates, replace failed virtual machines, or manage remote access (such as SSH or RDP) for administrators. On top of that, AWS Lambda is billed by invocation. Therefore, you don't have to pay for idling resources that are waiting for work (e.g., for a task triggered once a day).

In our first example, you will create a Lambda function that performs periodic health checks for your website. This will teach you how to use the Management Console to get started with AWS Lambda quickly. In our second example, you will learn how to write your own Python code and deploy a Lambda function in an automated way using CloudFormation, which we introduced in chapter 4. Your Lambda function will automatically add a tag to newly launched EC2 instances. At the end of the chapter, we'll show you additional use cases like building web applications and Internet of Things (IoT) backends and processing data with AWS Lambda.

Examples are almost all covered by the Free Tier

The examples in this chapter are mostly covered by the Free Tier. There is one exception: AWS CloudTrail. In case you already created a trail in your account, additional charges—most likely just a few cents—will apply. For details, please see **https://aws.amazon.com/cloudtrail/pricing/**.

You will find instructions on how to clean up the examples at the end of each section.

You may be asking a more basic question: what exactly is AWS Lambda? Before diving into our first real-world example, let us introduce you, briefly, to Lambda and explain why it's often mentioned in the context of an architecture that's being called *serverless*.

## 6.1 Executing your code with AWS Lambda

Computing capacity is available at different layers of abstraction on AWS: virtual machines, containers, and functions. You learned about the virtual machines offered by Amazon's EC2 service in chapter 3. Containers offer another layer of abstraction on top of virtual machines; you will learn about containers in chapter 18. AWS Lambda provides computing power, as well, but in a fine-grained manner: it is an execution environment for small functions, rather than a full-blown operating system or container.

### 6.1.1 What is serverless?

When reading about AWS Lambda, you might have stumbled upon the term *serverless*. In his book *Serverless Architectures on AWS* (Manning, 2022; **http://mng.bz/wyr5**), Peter Sbarski summarizes the confusion created by this catchy and provocative phrase:

*[...] the word serverless is a bit of a misnomer. Whether you use a compute service such as AWS Lambda to execute your code, or interact with an API, there are still servers running in the background. The difference is that these servers are hidden from you. There's no infrastructure for you to think about and no way to tweak the underlying operating system. Someone else takes care of the nitty-gritty details of infrastructure management, freeing your time for other things.*

—Peter Sbarski

We define a serverless system as one that meets the following criteria:

- No need to manage and maintain virtual machines
- Fully managed service offering scalability and high availability

- Billed per request and by resource consumption

AWS Lambda certainly fits these definitions and is indeed a serverless platform in that AWS handles server configurations and management so the server is essentially invisible to you and takes care of itself. AWS is not the only provider offering a serverless platform. Google (Cloud Functions) and Microsoft (Azure Functions) are other competitors in this area. If you would like to read more about serverless, here is a free chapter from *Serverless Architectures on AWS,* second edition: **http://mng.bz/qoEx**.

### 6.1.2 Running your code on AWS Lambda

AWS Lambda is the basic building block of the serverless platform provided by AWS. The first step in the process is to run your code on Lambda instead of on your own server.

As shown in figure 6.1, to execute your code with AWS Lambda, follow these steps:

1. Write the code.
2. Upload your code and its dependencies (such as libraries or modules).
3. Create a function determining the runtime environment and configuration.
4. Invoke the function to execute your code in the cloud.

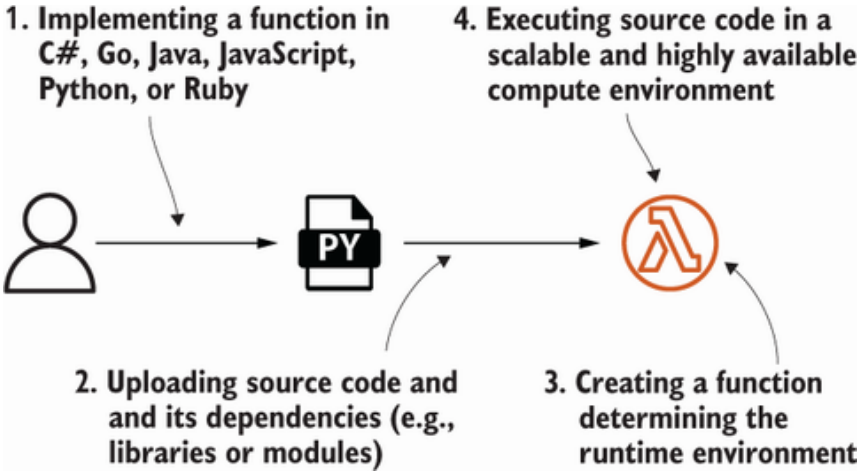


Figure 6.1 Executing code with AWS Lambda

Currently, AWS Lambda offers runtime environments for the following languages:

- C#/.NET Core
- Go
- Java
- JavaScript/Node.js

- Python
- Ruby

Besides that, you can bring your own runtime by using a custom runtime. In theory, a custom runtime supports any programming language. You need to bring your own container image and follow conventions for initializing the function and processing tasks. Doing so adds extra complexity, so we recommend going with one of the available runtimes.

Next, we will compare AWS Lambda with EC2 virtual machines.

### 6.1.3 Comparing AWS Lambda with virtual machines (Amazon EC2)

What is the difference between using AWS Lambda and virtual machines? First is the granularity of virtualization. Virtual machines provide a full operating system for running one or multiple applications. In contrast, AWS Lambda offers an execution environment for a single function, a small part of an application.

Furthermore, Amazon EC2 offers virtual machines as a service, but you are responsible for operating them in a secure, scalable, and highly available way. Doing so requires you to put a substantial amount of effort into maintenance. In contrast, when building with Lambda, AWS manages the underlying infrastructure for you and provides a production-ready infrastructure.

Beyond that, AWS Lambda is billed per execution, and not per second like when a virtual machine is running. You don't have to pay for unused resources that are waiting for requests or tasks. For example, running a script to check the health of a website every five minutes on a virtual machine would cost you about $3.71 per month. Executing the same health check with AWS Lambda will cost about $0.04 per month.

Table 6.1 compares AWS Lambda and virtual machines in detail. You'll find a discussion of AWS Lambda's limitations at the end of the chapter.

Table 6.1 AWS Lambda compared to Amazon EC2

| | AWS Lambda | Amazon EC2 |
|---|---|---|
| Granularity of virtualization | Small piece of code (a function). | An entire operating system. |
| Scalability | Scales automatically. A throttling limit prevents you from creating unwanted charges accidentally and can be increased by AWS support if needed. | As you will learn in chapter 17, using an Auto Scaling group allows you to scale the number of EC2 instances serving requests automatically, but configuring and monitoring the scaling activities is your responsibility. |
| High availability | Fault tolerant by default. The computing infrastructure spans multiple machines and data centers. | Virtual machines are not highly available by default. Nevertheless, as you will learn in chapter 13, it is possible to set up a highly available infrastructure based on EC2 instances as well. |
| Maintenance effort | Almost zero. You need only to configure your function. | You are responsible for maintaining all layers between your virtual machine's operating system and your application's runtime environment. |
| Deployment effort | Almost zero due to a well-defined API | Rolling out your application to a fleet of virtual machines is a challenge that requires tools and know-how. |
| Pricing model | Pay per request as well as execution time and | Pay for operating hours of the virtual machines, |

allocated memory billed per second

Looking for limitations and pitfalls of AWS Lambda? Stay tuned: you will find a discussion of Lambda's limitations at the end of the chapter.

That's all you need to know about AWS Lambda to be able to go through the first real-world example. Are you ready?

## 6.2 Building a website health check with AWS Lambda

Are you responsible for the uptime of a website or application? We do our best to make sure our blog **https://cloudonaut.io** is accessible 24/7. An external health check acts as a safety net, making sure we, and not our readers, are the first to know when our blog goes down. AWS Lambda is the perfect choice for building a website health check, because you do not need computing resources constantly but only every few minutes for a few milliseconds. This section guides you through setting up a health check for your website based on AWS Lambda.

In addition to AWS Lambda, we are using the Amazon CloudWatch service for this example. Lambda functions publish metrics to CloudWatch by default. Typically, you inspect metrics using charts and create alarms by defining thresholds. For example, a metric could count failures during the function's execution. On top of that, EventBridge provides events that can be used to trigger Lambda functions as well. Here we are using a rule to publish an event every five minutes.

As shown in figure 6.2, your website health check will consist of the following three parts:

1. *Lambda function*—Executes a Python script that sends an HTTP request to your website (e.g., `GET https:/ /cloudonaut.io`) and verifies that the response includes specific text (such as `cloudonaut`).
2. *EventBridge rule*—Triggers the Lambda function every five minutes. This is comparable to the cron service on Linux.
3. *Alarm*—Monitors the number of failed health checks and notifies you via email whenever your website is unavailable.

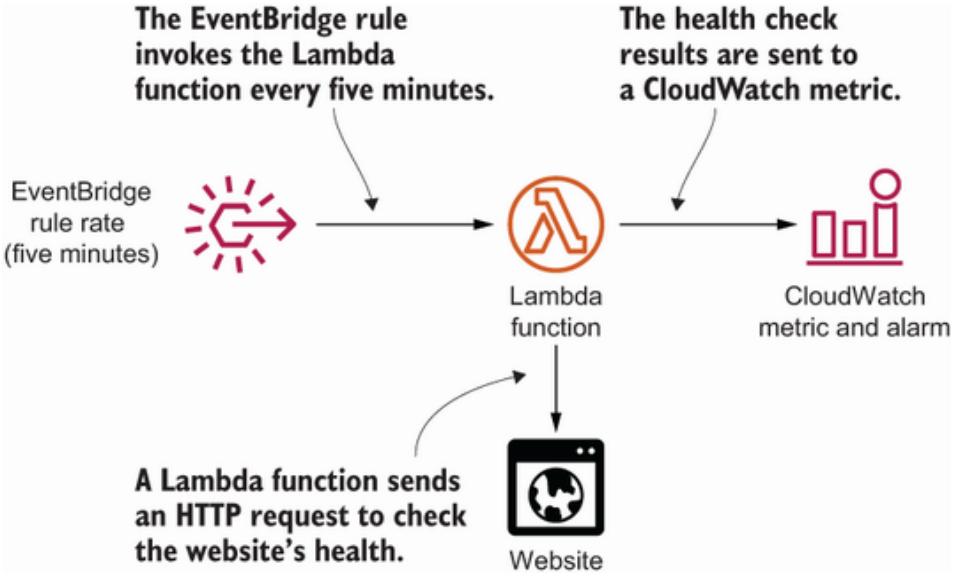


Figure 6.2 The Lambda function performing the website health check is executed every five minutes by a scheduled event. Errors are reported to CloudWatch.

You will use the Management Console to create and configure all the necessary parts manually. In our opinion, this is a simple way to get familiar with AWS Lambda. You will learn how to deploy a Lambda function in an automated way in section 6.3.

### 6.2.1 Creating a Lambda function

The following step-by-step instructions guide you through setting up a website health check based on AWS Lambda. Open AWS Lambda in the Management Console: **https://console.aws.amazon.com/lambda/home**. Click Create a Function to start the Lambda function wizard, as shown in figure 6.3.


Figure 6.3 Welcome screen: Ready to create your first Lambda function

AWS provides blueprints for various use cases, including the code and the Lambda function configuration. We will use one of these blueprints to create a website health check. Select Use a Blueprint, and search for canary. Next, choose the lambda-canary blueprint. Figure 6.4 illustrates the details.

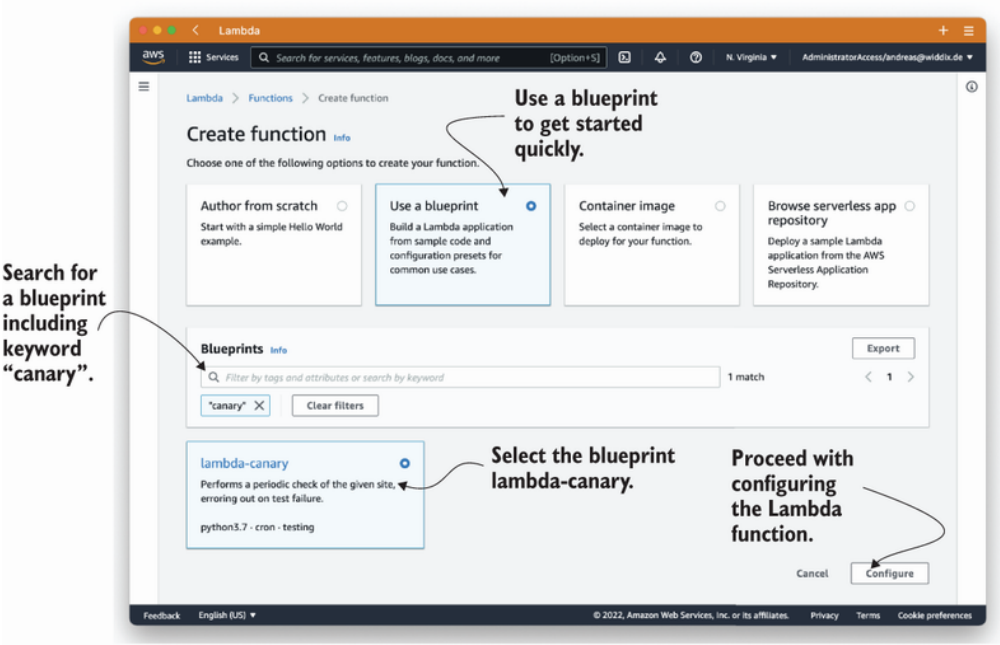


Figure 6.4 Creating a Lambda function based on a blueprint provided by AWS

In the next step of the wizard, you need to specify a name for your Lambda function, as shown in figure 6.5. The function name needs to be unique within your AWS account and the current region US East (N. Virginia). In addition, the name is limited to 64 characters. To invoke a function via the API, you need to provide the function name. Type in website-health-check as the name for your Lambda function.

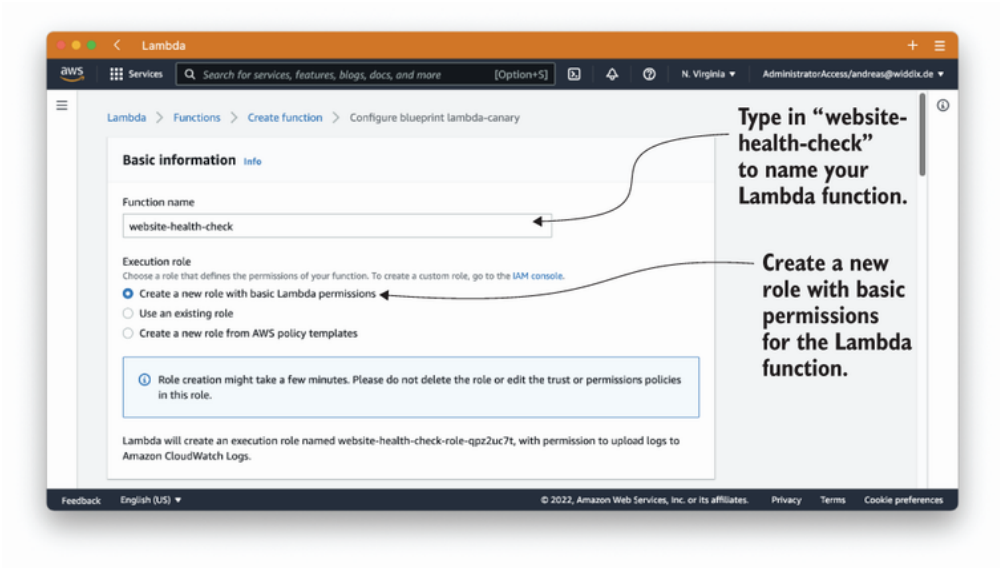


Figure 6.5 Creating a Lambda function: Choose a name and define an IAM role.

Continue with creating an IAM role. Select Create a New Role with Basic Lambda Permissions, as shown in figure 6.5, to create an IAM role for your Lambda function. You will learn how your Lambda function uses the IAM role in section 6.3.

Next, configure the scheduled event that will trigger your health check repeatedly. We will use an interval of five minutes in this example. Figure

6.6 shows the settings you need:

1. Select Create a New Rule to create a scheduled event rule.
2. Type in website-health-check as the name for the rule.
3. Enter a description that will help you to understand what is going on if you come back later.
4. Select Schedule Expression as the rule type. You will learn about the other option, Event Pattern, at the end of this chapter.
5. Use rate(5 minutes) as the schedule expression.

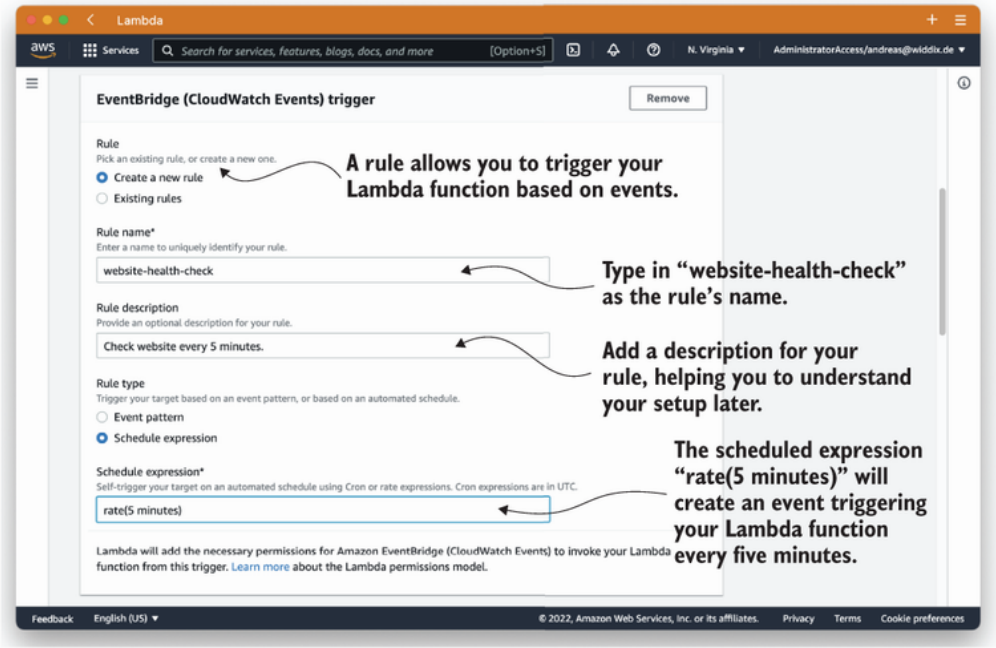


Figure 6.6 Configuring a scheduled event triggering your Lambda function every five minutes

Define recurring tasks using a *schedule expression* in form of `rate($value $unit)`. For example, you could trigger a task every five minutes, every hour, or once a day. `$value` needs to be a positive integer value. Use `minute`, `minutes`, `hour`, `hours`, `day`, or `days` as the unit. For example, instead of triggering the website health check every five minutes, you could use `rate(1 hour)` as the schedule expression to execute the health check every hour. Note that frequencies of less than one minute are not supported.

It is also possible to use the crontab format, shown here, when defining a schedule expression:

```
cron($minutes $hours $dayOfMonth $month $dayOfWeek $year)

 # Invoke a Lambda function at 08:00am (UTC) everyday
 cron(0 8 * * ? *)

 # Invoke a Lambda function at 04:00pm (UTC) every Monday to Friday
 cron(0 16 ? * MON-FRI *)
```

See “Schedule Expressions Using Rate or cron” at http://mng.bz/7ZnQ for more details.

Your Lambda function is missing an integral part: the code. Because you are using a blueprint, AWS has inserted the Python code implementing the website health check for you, as shown in figure 6.7.


Figure 6.7 The predefined code implementing the website health check and environment variables to pass settings to the Lambda function

The Python code references two environment variables: `site` and `expected`. Environment variables are commonly used to dynamically pass settings to your function. An environment variable consists of a key and a value. Specify the following environment variables for your Lambda function:

- `site`—Contains the URL of the website you want to monitor. Use https://cloudonaut.io if you do not have a website to monitor yourself.

- `expected`—Contains a text snippet that must be available on your website. If the function doesn't find this text, the health check fails. Use `cloudonaut` if you are using **https://cloudonaut.io** as your website.

The Lambda function is reading the environment variables during its execution, as shown next:

```
SITE = os.environment['site']
 EXPECTED = os.environment['expected']
```

After defining the environment variables for your Lambda function, click the Create Function button at the bottom of the screen.

Congratulations—you have successfully created a Lambda function. Every five minutes, the function is invoked automatically and executes a health check for your website.

You could use this approach to automate recurring tasks, like checking the status of a system, cleaning up unused resources like EBS snapshots, or to send recurring reports.

So far, you have used a predefined template. Lambda gets much more powerful when you write your own code. But before we look into that, you will learn how to monitor your Lambda function and get notified via email whenever the health check fails.

### 6.2.2 Use CloudWatch to search through your Lambda function's logs

How do you know whether your website health check is working correctly? How do you even know whether your Lambda function has been executed? It is time to look at how to monitor a Lambda function. You will learn how to access your Lambda function's log messages first. Afterward, you will create an alarm notifying you if your function fails.

Open the Monitor tab in the details view of your Lambda function. You will find a chart illustrating the number of times your function has been invoked. Reload the chart after a few minutes, in case the chart isn't showing any invocations. To go to your Lambda function's logs, click View Logs in CloudWatch, as shown in figure 6.8.

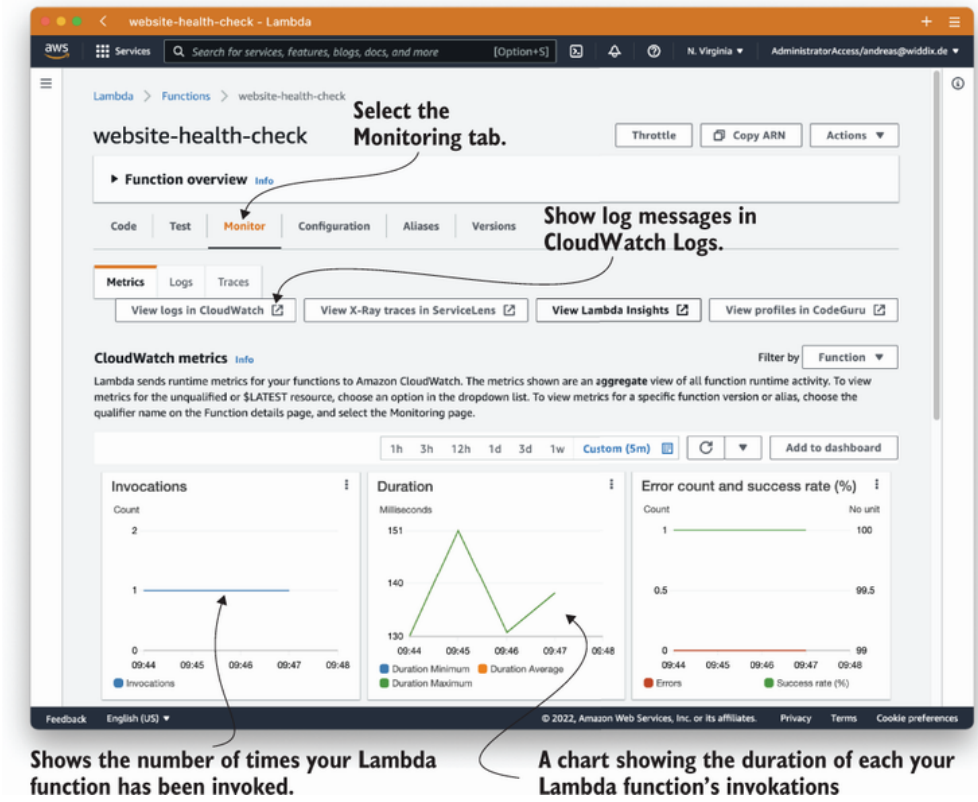


Figure 6.8 Monitoring overview: Get insights into your Lambda function's invocations.

By default, your Lambda function sends log messages to CloudWatch. Figure 6.9 shows the log group named /aws/lambda/website-health-check that was created automatically and collects the logs from your function. Typically, a log group contains multiple log streams, allowing the log group to scale. Click Search Log Group to view the log messages from all streams in one view.

Figure 6.9 A log group collects log messages from a Lambda function stored in multiple log streams.

All log messages are presented in the overview of log streams, as shown in figure 6.10. You should be able to find a log message `Check passed!`, indicating that the website health check was executed and passed successfully, for example.

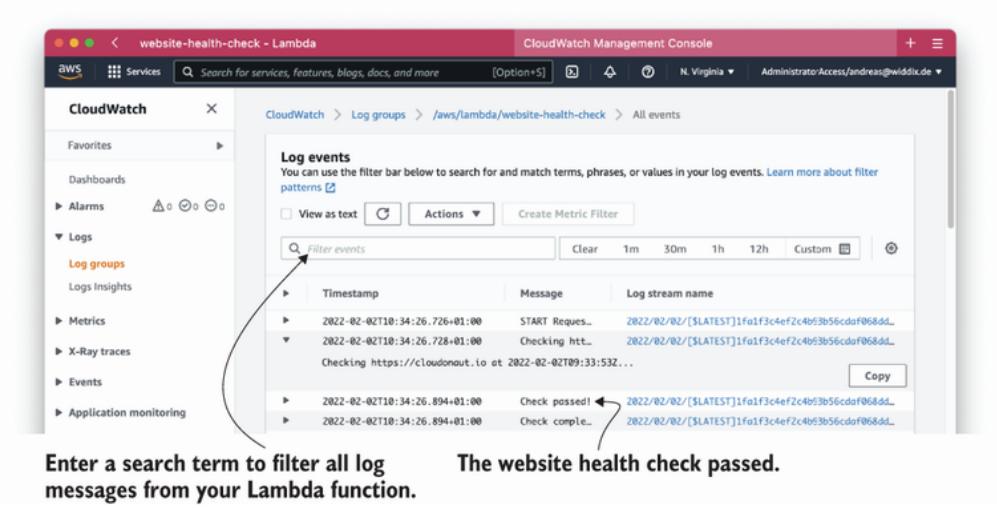


Figure 6.10 CloudWatch shows the log messages of your Lambda function.

The log messages show up after a delay of a few minutes. Reload the table if you are missing any log messages.

Being able to search through log messages in a centralized place is handy when debugging Lambda functions, especially if you are writing your own code. When using Python, you can use `print` statements or use the `logging` module to send log messages to CloudWatch.

### 6.2.3 Monitoring a Lambda function with CloudWatch metrics and alarms

The Lambda function now checks the health of your website every five minutes, and a log message with the result of each health check is written to CloudWatch. But how do you get notified via email if the health check fails? Each Lambda function publishes metrics to CloudWatch by default. Table 6.2 shows important metrics. Check out **http://mng.bz/m298** for a complete list of metrics.

Table 6.2 The CloudWatch metrics published by each Lambda function

| Name | Description |
|---|---|
| Invocations | Counts the number of times a function is invoked. Includes successful and failed invocations. |
| Errors | Counts the number of times a function failed due to errors inside the function, for example, exceptions or timeouts. |
| Duration | Measures how long the code takes to run, from the time when the code starts executing to when it stops executing. |
| Throttles | As discussed at the beginning of the chapter, there is a limit for how many copies of your Lambda function can run at one time. This metric counts how many invocations have been throttled due to reaching this limit. Contact AWS support to increase the limit, if needed. |

Whenever the website health check fails, the Lambda function returns an error, which increases the count of the Errors metric. You will create an alarm notifying you via email whenever this metric counts more than zero errors. In general, we recommend creating an alarm on the Errors and Throttles metrics to monitor your Lambda functions.

The following steps guide you through creating a CloudWatch alarm to monitor your website health checks. Your Management Console still shows the CloudWatch service. Select Alarms from the subnavigation menu. Next, click Create Alarm, as shown in figure 6.11.

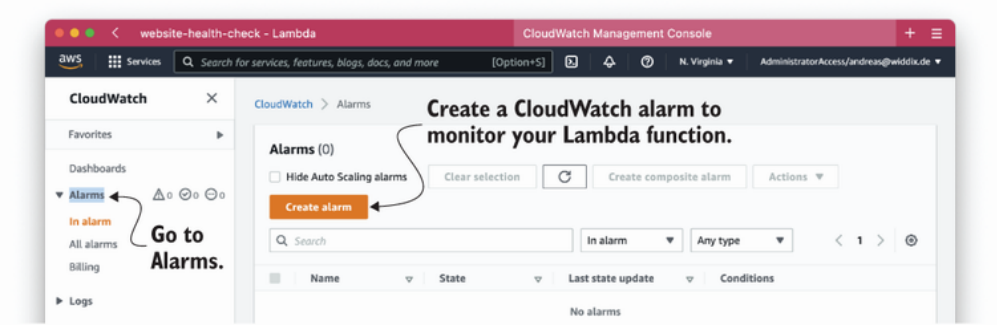


Figure 6.11 Starting the wizard to create a CloudWatch alarm to monitor a Lambda function

The following three steps guide you through the process of selecting the Error metric of your `website-health-check` Lambda function, as illus-

trated in figure 6.12:

1. Click Select Metric.
2. Choose the Lambda namespace.
3. Select metrics with dimension Function Name.

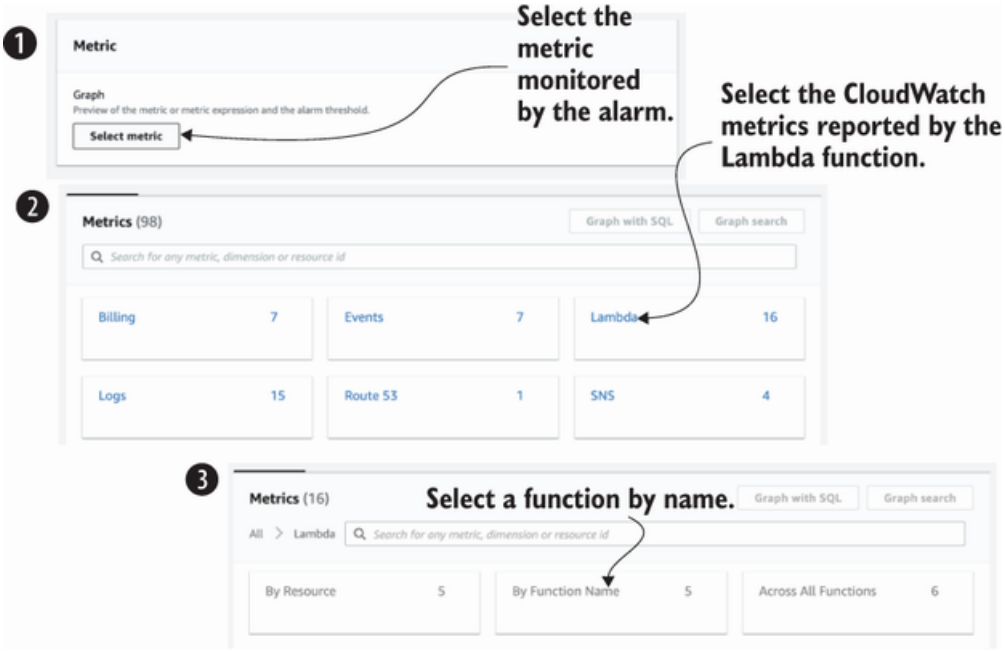


Figure 6.12 Searching the CloudWatch metric to create an alarm

Last but not least, select the Error metric belonging to the Lambbda function `website-health-check` as shown in figure 6.13. Proceed by clicking the Select Metric button.

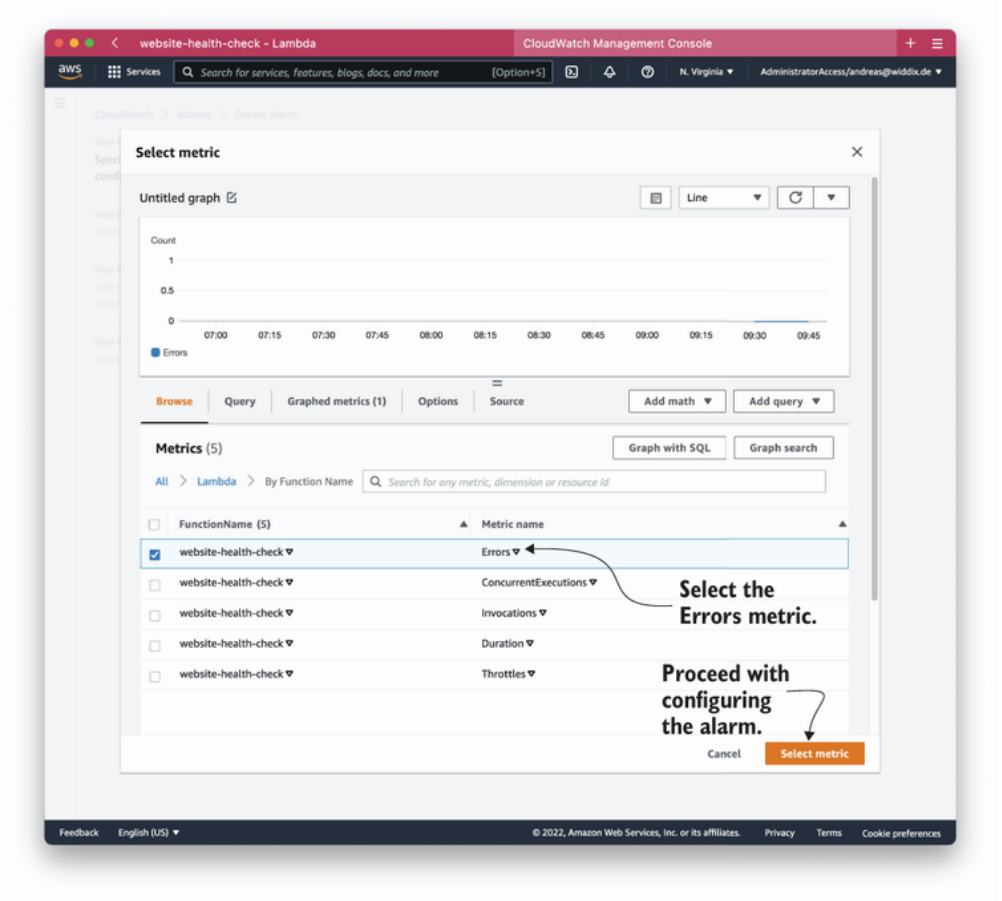


Figure 6.13 Selecting the Error metric of the Lambda function

Next, you need to configure the alarm, as shown in figure 6.14. To do so, specify the statistic, period, and threshold as follows:

1. Choose the statistic Sum to add up all the errors that occurred during the evaluation period.
2. Define an evaluation period of five minutes.
3. Select the Static Threshold option.
4. Choose the Greater operator.
5. Define a threshold of zero.
6. Click the Next button to proceed.

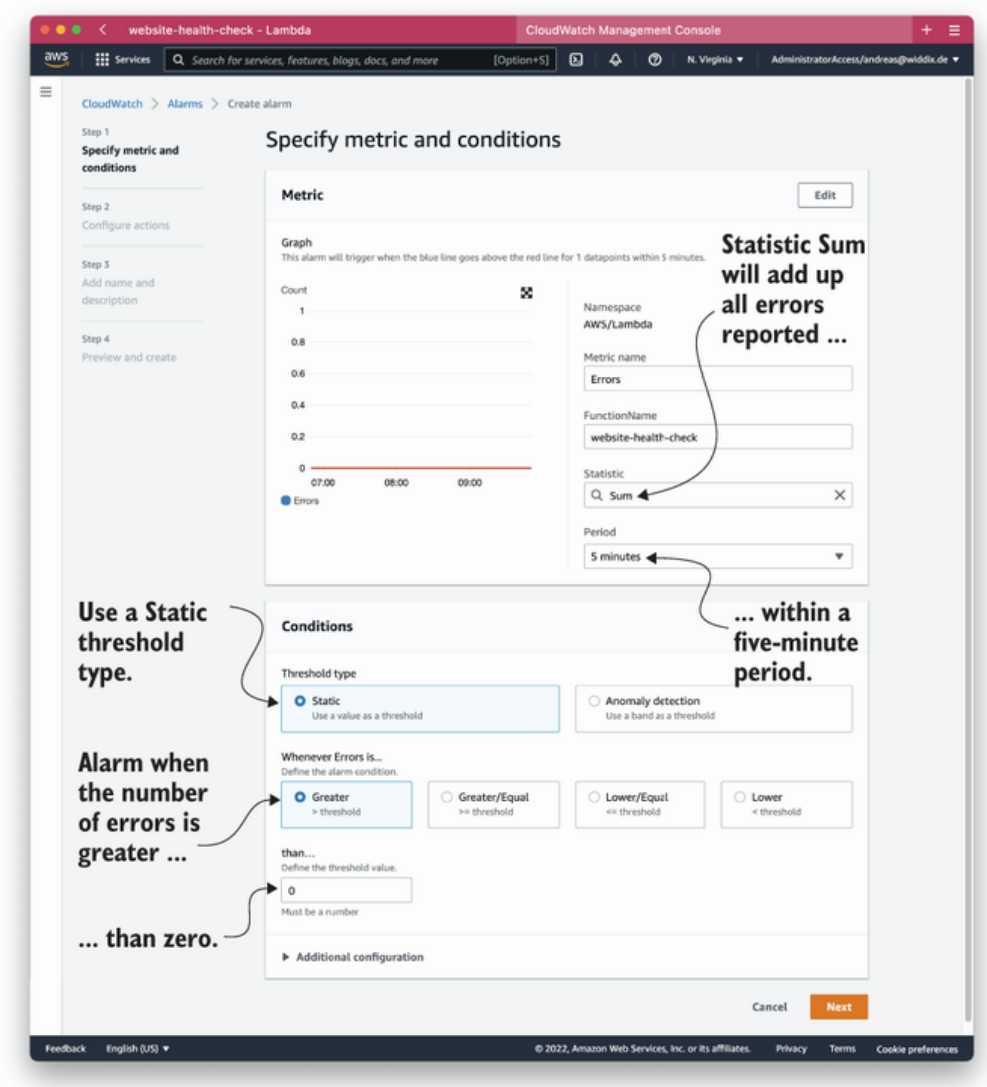


Figure 6.14 Selecting and preparing the metric view for the alarm

In other words, the alarm state will change to `ALARM` when the Lambda function reports any errors during the evaluation period of five minutes. Otherwise, the alarm state will be `OK`.

The next step, illustrated in figure 6.15, configures the alarm actions so you will be notified via e-mail:

1. Select In Alarm as the state trigger.
2. Create a new SNS topic by choosing Create New Topic.
3. Type in website-health-check as the name for the SNS topic.
4. Enter your email address.
5. Click the Next button to proceed.

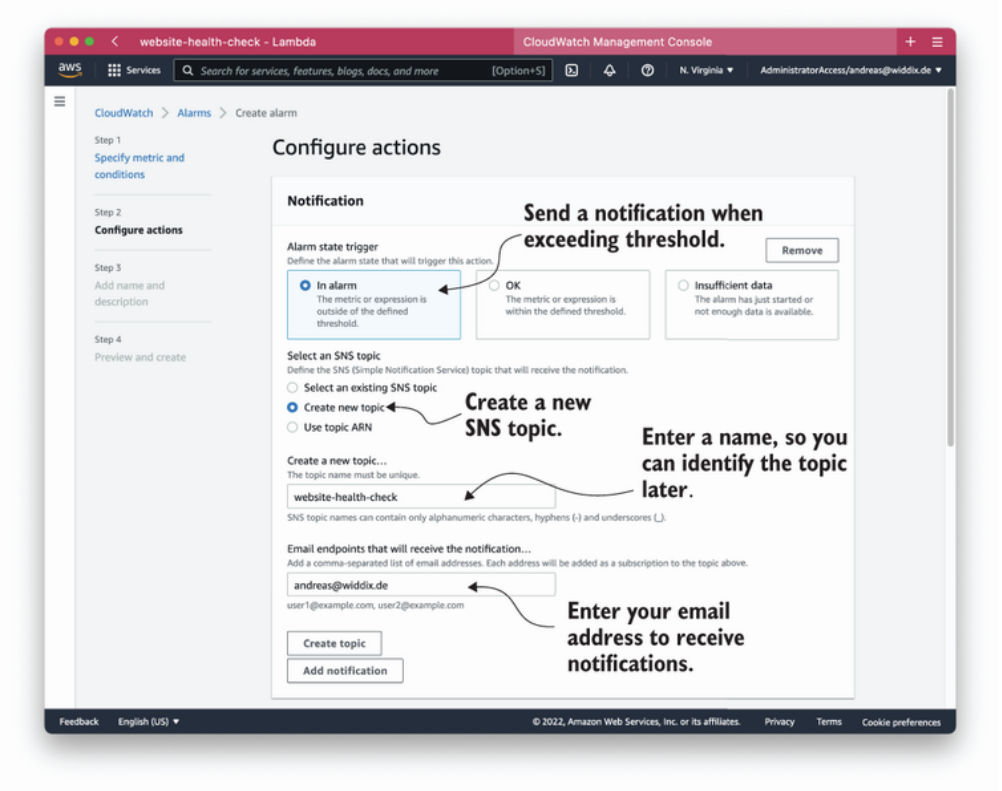


Figure 6.15 Creating an alarm by defining a threshold and defining an alarm action to send notifications via email

In the following step, you need to enter a name for the CloudWatch alarm. Type in website-health-check-error. Afterward, click the Next button to proceed.

After reviewing the configuration, click the Create Alarm button. Shortly after creating the alarm, you will receive an email including a confirmation link for SNS. Check your inbox and click the link to confirm your subscription to the notification list.

To test the alarm, go back to the Lambda function and modify the environment variable `expected`. For example, modify the value from `cloudonaut` to `FAILURE`. This will cause the health check to fail, because the word `FAILURE` does not appear on the website. It might take up to 15 minutes before you receive a notification about the failed website health check via email.

Cleaning up

Open your Management Console and follow these steps to delete all the resources you have created during this section:

1. Go to the AWS Lambda service and delete the function named website-health-check.
2. Open the AWS CloudWatch service, select Logs from the subnavigation options, and delete the log group /aws/lambda/website-health-check.

3. Go to the EventBridge service, select Rules from the subnavigation menu, and delete the rule website-health-check.
4. Open the CloudWatch service, select Alarms from the subnavigation menu, and delete the alarm website-health-check-error.
5. Jump to the AWS IAM service, select Roles from the subnavigation menu, and delete the role whose name starts with website-health-check-role-.
6. Go to the SNS service, select Topics from the subnavigation menu, and delete the rule website-health-check.

### 6.2.4 Accessing endpoints within a VPC

As illustrated in figure 6.16, by default Lambda functions run outside your networks defined with VPC. However, Lambda functions are connected to the internet and, therefore, are able to access other services. That's exactly what you have been doing when creating a website health check: the Lambda function was sending HTTP requests over the internet.

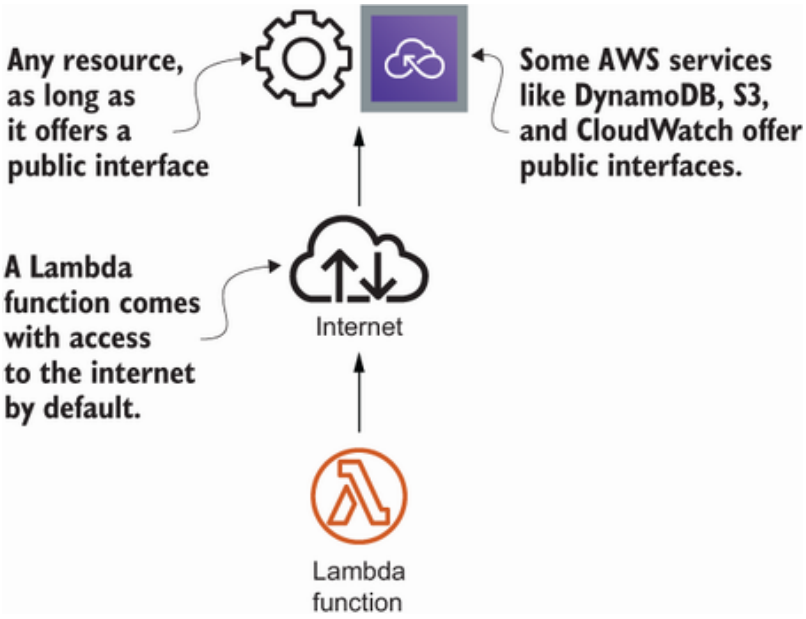


Figure 6.16 By default a Lambda function is connected to the internet and running outside your VPCs.

So, what do you do when you have to reach a resource running in a private network within your VPC, for example, if you want to run a health check for an internal website? If you add network interfaces to your Lambda function, the function can access resources within your VPCs, as shown in figure 6.17.

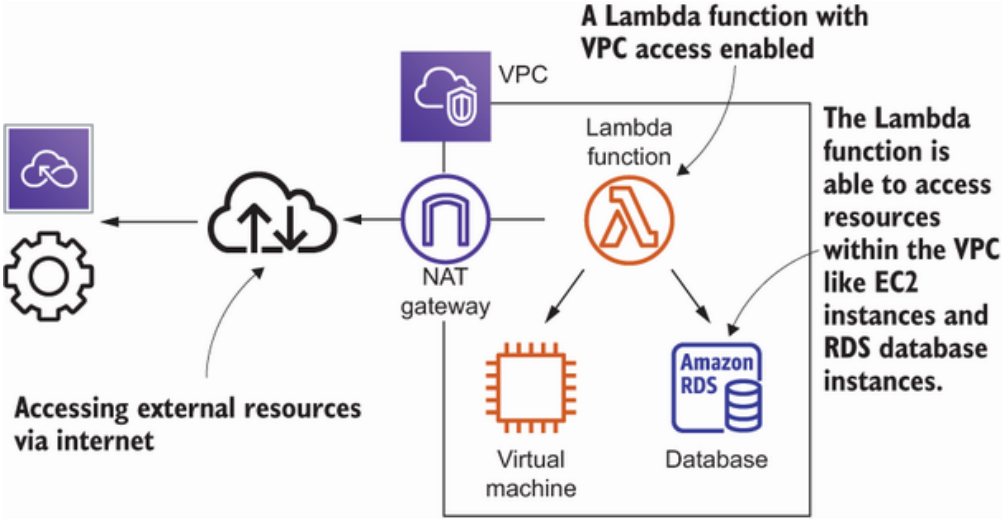


Figure 6.17 Deploying a Lambda function into your VPC allows you to access internal resources (such as database, virtual machines, and so on).

To do so, you have to define the VPC and the subnets, as well as security groups for your Lambda function. See "Configuring a Lambda Function to Access Resources in an Amazon VPC" at **http://mng.bz/5m67** for more details. We have been using the ability to access resources within a VPC to access databases in various projects.

We do recommend connecting a Lambda function to a VPC only when absolutely necessary because it introduces additional complexity. However, being able to connect with resources within your private networks is very interesting, especially when integrating with legacy systems.

## 6.3 Adding a tag containing the owner of an EC2 instance automatically

After using one of AWS's predefined blueprints to create a Lambda function, you will implement the function from scratch in this section. We are strongly focused on setting up your cloud infrastructure in an automated way. That's why you will learn how to deploy a Lambda function and all its dependencies without needing the Management Console.

Are you working in an AWS account together with your colleagues? Have you ever wondered who launched a certain EC2 instance? Sometimes you need to find out the owner of an EC2 instance for the following reasons:

- Double-checking whether it is safe to terminate an unused instance without losing relevant data
- Reviewing changes to an instance's configuration with its owner (such as making changes to the firewall configuration)
- Attributing costs to individuals, projects, or departments
- Restricting access to an instance (e.g., so only the owner is allowed to terminate an instance)

Adding a tag that states who owns an instance solves all these use cases. A tag can be added to an EC2 instance or almost any other AWS resource and consists of a key and a value. You can use tags to add information to a resource, filter resources, attribute costs to resources, and restrict access. See “Tag your Amazon EC2 resources” at **http://mng.bz/69oR** for more details.

It is possible to add tags specifying the owner of an EC2 instance manually. But, sooner or later, someone will forget to add the owner tag. There is a better solution! In the following section, you will implement and deploy a Lambda function that automatically adds a tag containing the name of the user who launched an EC2 instance. But how do you execute a Lambda function every time an EC2 instance is launched so that you can add the tag?

### 6.3.1 Event-driven: Subscribing to EventBridge events

EventBridge is an event bus used by AWS, third-party vendors, and customers like you, to publish and subscribe to events. In this example, you will create a rule to listen for events from AWS. Whenever something changes in your infrastructure, an event is generated in near real time and the following things occur:

- CloudTrail emits an event for every call to the AWS API if CloudTrail is enabled in the AWS account and region.
- EC2 emits events whenever the state of an EC2 instances changes (such as when the state changes from `Pending` to `Running`).
- AWS emits an event to notify you of service degradations or downtimes.

Whenever you launch a new EC2 instance, you are sending a call to the AWS API. Subsequently, CloudTrail sends an event to EventBridge. Our goal is to add a tag to every new EC2 instance. Therefore, we are executing a function for every event that indicates the launch of a new EC2 instance. To trigger a Lambda function whenever such an event occurs, you need a rule. As illustrated in figure 6.18, the rule matches incoming events and routes them to a target, a Lambda function in our case.

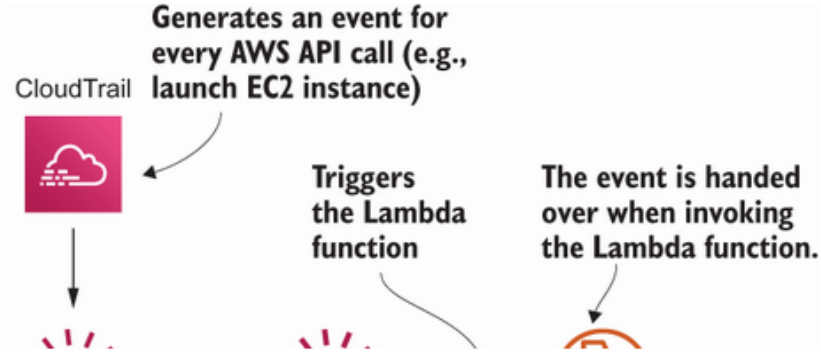


Figure 6.18 CloudTrail generates an event for every AWS API call; a rule routes the event to the Lambda function.

Listing 6.1 shows some of the event details generated by CloudTrail whenever someone launches an EC2 instance. For our case, we're interested in the following information:

- `detail-type` —The event has been created by CloudTrail.
- `source` —The EC2 service is the source of the event.
- `eventName` —The event name `RunInstances` indicates that the event was generated because of an AWS API call launching an EC2 instance.
- `userIdentity` —Who called the AWS API to launch an instance?
- `responseElements` —The response from the AWS API when launching an instance. This includes the ID of the launched EC2 instance; we will need to add a tag to the instance later.

**Listing 6.1 Event generated by CloudTrail when launching an EC2 instance**

```
{
  "version": "0",
  "id": "2db486ef-6775-de10-1472-ecc242928abe",
  "detail-type": "AWS API Call via CloudTrail",          ①
  "source": "aws.ec2",                                   ②
  "account": "XXXXXXXXXXXX",
  "time": "2022-02-03T11:42:25Z",
  "region": "us-east-1",
  "resources": [],
  "detail": {
    "eventVersion": "1.08",
    "userIdentity": {                                    ③
      "type": "IAMUser",
      "principalId": "XXXXXXXXXXXX",
      "arn": "arn:aws:iam::XXXXXXXXXXXX:user/myuser",    ④
```

```
        "accountId": "XXXXXXXXXXXX",
        "accessKeyId": "XXXXXXXXXXXX",
        "userName": "myuser"
      },
      "eventTime": "2022-02-03T11:42:25Z",
      "eventSource": "ec2.amazonaws.com",
      "eventName": "RunInstances",                                    ⑤
      "awsRegion": "us-east-1",
      "sourceIPAddress": "109.90.107.17",
      "userAgent": "aws-cli/2.4.14 Python/3.9.10 Darwin/21.2.0
                ➥ source/arm64 prompt/off command/ec2.run-instances",
      "requestParameters": {
        [...]
      },
      "responseElements": {                                           ⑥
        "requestId": "d52b86c7-5bf8-4d19-86e8-112e59164b21",
        "reservationId": "r-08131583e8311879d",
        "ownerId": "166876438428",
        "groupSet": {},
        "instancesSet": {
          "items": [
            {
              "instanceId": "i-07a3c0d78dc1cb505",                    ⑦
              "imageId": "ami-01893222c83843146",
              [...]
            }
          ]
        }
      },
      "requestID": "d52b86c7-5bf8-4d19-86e8-112e59164b21",
      "eventID": "8225151b-3a9c-4275-8b37-4a317dfe9ee2",
      "readOnly": false,
      "eventType": "AwsApiCall",
      "managementEvent": true,
      "recipientAccountId": "XXXXXXXXXXXX",
      "eventCategory": "Management",
      "tlsDetails": {
        [...]
      }
    }
  }
```

① CloudTrail generated the event.

② Someone sent a call to the AWS API, affecting the EC2 service.

③ Information about the user who launched the EC2 instance

④ ID of the user who launched the EC2 instance

⑤ Event was generated because a RunInstances call (used to launch an EC2 instance) was processed by the AWS API.

⑥ Response of the AWS API when launching the instance

⑦ ID of the launched EC2 instance

A rule consists of an event pattern for selecting events, along with a definition of one or multiple targets. The following pattern selects all events from CloudTrail generated by an AWS API call affecting the EC2 service. The pattern matches four attributes from the event described in the next listing: `source`, `detail-type`, `eventSource`, and `eventName`.

Listing 6.2 The pattern to filter events from CloudTrail

```
{
  "source": [
    "aws.ec2"                          ①
  ],
  "detail-type": [
    "AWS API Call via CloudTrail"      ②
  ],
  "detail": {
    "eventSource": [
      "ec2.amazonaws.com"              ③
    ],
    "eventName": [
      "RunInstances"                   ④
    ]
  }
}
```

① Filters events from the EC2 service

② Filters events from CloudTrail caused by AWS API calls

③ Filters events from the EC2 service

④ Filters events with event name RunInstances, which is the AWS API call to launch an EC2 instance

Defining filters on other event attributes is possible as well, in case you are planning to write another rule in the future. The rule format stays the same.

When specifying an event pattern, we typically use the following fields, which are included in every event:

- `source`—The namespace of the service that generated the event. See "Amazon Resource Names (ARNs)" at **http://mng.bz/o5WD** for details.
- `detail-type`—Categorizes the event in more detail.

See "EventBridge Event Patterns" at **http://mng.bz/nejd** for more detailed information.

You have now defined the events that will trigger your Lambda function. Next, you will implement the Lambda function.

### 6.3.2 Implementing the Lambda function in Python

Implementing the Lambda function to tag an EC2 instance with the owner's user name is simple. You will need to write no more than 10 lines of Python code. The programming model for a Lambda function depends on the programming language you choose. Although we are using Python in our example, you will be able to apply what you've learned when implementing a Lambda function in C#/.NET Core, Go, Java, JavaScript/Node.js, Python, or Ruby. As shown in the next listing, your function written in Python needs to implement a well-defined structure.

Listing 6.3 Lambda function written in Python

```
def lambda_handler(event, context):   ①
   # Insert your code                 ②
   return                             ③
```

① The name of the Python function, which is referenced by the AWS Lambda as the function handler. The event parameter is used to pass the CloudWatch event, and the context parameter includes runtime information.

② It is your job to implement the function.

③ Use return to end the function execution. It is not useful to hand over a value in this scenario, because the Lambda function is invoked asynchronously by a CloudWatch event.

Where is the code located?

As usual, you'll find the code in the book's code repository on GitHub: **https://github.com/AWSinAction/code3**. Switch to the chapter06 directory, which includes all files needed for this example.

Time to write some Python code! Listing 6.4 for `lambda_function.py` shows the function, which receives an event from CloudTrail indicating

that an EC2 instance has been launched recently, and adds a tag including the name of the instance's owner. The AWS SDK for Python 3.9, named `boto3`, is provided out of the box in the Lambda runtime environment for Python 3.9. In this example, you are using the AWS SDK to create a tag for the EC2 instance `ec2.create_tags(...)`. See the Boto3 documentation at **https://boto3.readthedocs.io/en/latest/index.html** if you are interested in the details of `boto3`.

Listing 6.4 Lambda function adding a tag to EC2 instance

```
import boto3
ec2 = boto3.client('ec2')                                          ①

def lambda_handler(event, context):                                ②
  print(event)                                                     ③
  if "/" in event['detail']['userIdentity']['arn']:                ④
      userName = event['detail']['userIdentity']['arn'].split('/')[1]
  else:
      userName = event['detail']['userIdentity']['arn']
  instanceId = event['detail']['responseElements']['instancesSet']
  ➥ ['items'][0]['instanceId']                                    ⑤
  print("Adding owner tag " + userName + " to instance " + instanceId +
  ec2.create_tags(Resources=[instanceId,],
  ➥ Tags=[{'Key': 'Owner', 'Value': userName},])                  ⑥
  return
```

① Creates an AWS SDK client for EC2

② The name of the function used as entry point for the Lambda function

③ Logs the incoming event for debugging

④ Extracts the user's name from the CloudTrail event

⑤ Extracts the instance's ID from the CloudTrail event

⑥ Adds a tag to the EC2 instance using the key owner and the user's name as value

After implementing your function in Python, the next step is to deploy the Lambda function with all its dependencies.

### 6.3.3 Setting up a Lambda function with the Serverless Application Model (SAM)

You have probably noticed that we are huge fans of automating infrastructures with CloudFormation. Using the Management Console is a per-

fect way to take the first step when learning about a new service on AWS. But leveling up from manually clicking through a web interface to fully automating the deployment of your infrastructure should be your second step.

AWS released the *Serverless Application Model* (SAM) in 2016. SAM provides a framework for serverless applications, extending plain CloudFormation templates to make it easier to deploy Lambda functions. The next listing shows how to define a Lambda function using SAM and a CloudFormation template.

Listing 6.5 Defining a Lambda function with SAM within a CloudFormation template

```
---
AWSTemplateFormatVersion: '2010-09-09'                          ①
Transform: AWS::Serverless-2016-10-31                           ②
Description: Adding an owner tag to EC2 instances automatically
Resources:
  # [...]
  EC2OwnerTagFunction:
    Type: AWS::Serverless::Function                             ③
    Properties:
      Handler: lambda_function.lambda_handler                   ④
      Runtime: python3.9                                        ⑤
      Architectures:                                            ⑥
      - arm64
      CodeUri: '.'                                              ⑦
      Policies:                                                 ⑧
      - Version: '2012-10-17'
        Statement:
        - Effect: Allow
          Action: 'ec2:CreateTags'
          Resource: '*'
      Events:
        EventBridgeRule:                                        ⑨
          Type: EventBridgeRule                                 ⑩
          Properties:
            Pattern:                                            ⑪
              source:
              - 'aws.ec2'
              detail-type:
              - 'AWS API Call via CloudTrail'
              detail:
                eventSource:
                - 'ec2.amazonaws.com'
                eventName:
                - 'RunInstances'
```

① The CloudFormation template version, not the version of your code

② Transforms are used to process your template. We're using the SAM transformation.

③ A special resource provided by SAM allows us to define a Lambda function in a simplified way. CloudFormation will generate multiple resources out of this declaration during the transformation phase.

④ The handler is a combination of your script's filename and Python function name.

⑤ Uses the Python 3.9 runtime environment

⑥ Choose the ARM architecture for better price performance.

⑦ The current directory will be bundled, uploaded, and deployed. You will learn more about that soon.

⑧ Authorizes the Lambda function to call other AWS services (more on that next)

⑨ The definition of the event invoking the Lambda function

⑩ Creates an EventBridge rule

⑪ Configures the filter pattern

Please note: this example uses CloudTrail, which records all the activity within your AWS account. The CloudFormation template creates a trail to store an audit log on S3. That's needed because the EventBridge rule does not work without an active trail.

### 6.3.4 Authorizing a Lambda function to use other AWS services with an IAM role

Lambda functions typically interact with other AWS services. For instance, they write log messages to CloudWatch allowing you to monitor and debug your Lambda function. Or they create a tag for an EC2 instance, as in the current example. Therefore, calls to the AWS APIs need to be authenticated and authorized. Figure 6.19 shows a Lambda function assuming an IAM role to be able to send authenticated and authorized requests to other AWS services.

Figure 6.19 A Lambda function assumes an IAM role to authenticate and authorize requests to other AWS services.

Temporary credentials are generated based on the IAM role and injected into each invocation via environment variables (such as `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ ACCESS_KEY`, and `AWS_SESSION_TOKEN`). Those environment variables are used by the AWS SDK to sign requests automatically.

You should follow the least-privilege principle: your function should be allowed to access only services and actions that are needed to perform the function's task. You should specify a detailed IAM policy granting access to specific actions and resources.

Listing 6.6 shows an excerpt from the Lambda function's CloudFormation template based on SAM. When using SAM, an IAM role is created for each Lambda function by default. A managed policy that grants write access to CloudWatch logs is attached to the IAM role by default as well. Doing so allows the Lambda function to write to CloudWatch logs.

So far the Lambda function is not allowed to create a tag for the EC2 instance. You need a custom policy granting access to the `ec2:CreateTags`.

Listing 6.6 A custom policy for adding tags to EC2 instances

```
# [...]
 EC2OwnerTagFunction:
   Type: AWS::Serverless::Function
   Properties:
     Handler: lambda_function.lambda_handler
     Runtime: python3.9
     CodeUri: '.'
     Policies:                          ①
     - Version: '2012-10-17'
       Statement:
       - Effect: Allow                  ②
         Action: 'ec2:CreateTags'       ③
         Resource: '*'                  ④
     # [...]
```

① Defines a custom IAM policy that will be attached to the Lambda function's IAM role

② The statement allows ...

③ ...creating tags...

④ ...for all resources.

If you implement another Lambda function in the future, make sure you create an IAM role granting access to all the services your function needs to access (e.g., reading objects from S3, writing data to a DynamoDB database). Revisit section 5.3 if you want to recap the details of IAM.

### 6.3.5 Deploying a Lambda function with SAM

To deploy a Lambda function, you need to upload the deployment package to S3. The deployment package is a zip file including your code as well as additional modules. Afterward, you need to create and configure the Lambda function as well as all the dependencies (the IAM role, event rule, and so on). Using SAM in combination with the AWS CLI allows you to accomplish both tasks.

First, you need to create an S3 bucket to store your deployment packages. Use the following command, replacing `$yourname` with your name to avoid name conflicts with other readers:

```
$ aws s3 mb s3://ec2-owner-tag-$yourname
```

Execute the following command to create a deployment package and upload the package to S3. Please note: the command creates a copy of your template at output.yaml, with a reference to the deployment package uploaded to S3. Make sure your working directory is the code directory chapter06 containing the template.yaml and lambda_ function.py files:

```
$ aws cloudformation package --template-file template.yaml \
 ➥ --s3-bucket ec2-owner-tag-$yourname --output-template-file output.yam
```

By typing the following command in your terminal, you are deploying the Lambda function. This results in a CloudFormation stack named `ec2-owner-tag`:

```
$ aws cloudformation deploy --stack-name ec2-owner-tag \
 ➥ --template-file output.yaml --capabilities CAPABILITY_IAM
```

You are a genius! Your Lambda function is up and running. Launch an EC2 instance, and you will find a tag with your username `myuser` attached after a few minutes.

Cleaning up

If you have launched an EC2 instance to test your Lambda function, don't forget to terminate the instance afterward. Otherwise, it is quite simple to delete the Lambda function and all its dependencies. Just execute the following command in your terminal. Replace `$yourname` with your name:

```
$ CURRENT_ACCOUNT=$(aws sts get-caller-identity --query Account \
➥ --output text)
$ aws s3 rm --recursive s3://ec2-owner-tag-${CURRENT_ACCOUNT}/
$ aws cloudformation delete-stack --stack-name ec2-owner-tag
$ aws s3 rb s3://ec2-owner-tag-$yourname --force
```

## 6.4 What else can you do with AWS Lambda?

In the last part of the chapter, we would like to share what else is possible with AWS Lambda, starting with Lambda's limitations and insights into the serverless pricing model. We will end with three use cases for serverless applications we have built for our consulting clients.

### 6.4.1 What are the limitations of AWS Lambda?

Executing a Lambda function cannot exceed 15 minutes. This means the problem you are solving with your function needs to be small enough to fit into the 900-second limit. It is probably not possible to download 10 GB of data from S3, process the data, and insert parts of the data into a database within a single invocation of a Lambda function. But even if your use case fits into the 900-second constraint, make sure that it will fit under all circumstances. Here's a short anecdote from one of our first serverless projects: we built a serverless application that preprocessed analytics data from news sites. The Lambda functions typically processed the data within less than 180 seconds. But when the 2017 US elections came, the volume of the analytics data exploded in a way no one expected. Our Lambda functions were no longer able to complete within 300 seconds—which was the maximum back then. It was a show-stopper for our serverless approach.

AWS Lambda provisions and manages the resources needed to run your function. A new execution context is created in the background every time you deploy a new version of your code, go a long time without any invocations, or when the number of concurrent invocations increases. Starting a new execution context requires AWS Lambda to download your code, initialize a runtime environment, and load your code. This process is called a *cold start*. Depending on the size of your deployment

package, the runtime environment, and your configuration, a cold start could take from a few milliseconds to a few seconds.

In many scenarios, the increased latency caused by a cold start is not a problem at all. The examples demonstrated in this chapter—a website health check and automated tags for EC2 instances—are not negatively affected by a cold start. To minimize cold-start times, you should keep the size of your deployment package as small as possible, provision additional memory, and use a runtime environment like JavaScript/ Node.js or Python.

However, when processing real-time data or user interactions, a cold start is undesirable. For those scenarios, you could enable provisioned concurrency for a Lambda function. With provisioned concurrency, you tell AWS to keep a certain amount of execution contexts warm, even when the Lambda function is not processing any requests. As long as the provisioned concurrency exceeds the required number of execution contexts, you will not experience cold starts. The downside is you pay $0.0000041667 for every provisioned GB per second, whether or not you use the capacity. However, you will get a discount on the cost incurred for the actual term of the Lambda function.

Another limitation is the maximum amount of memory you can provision for a Lambda function: 10,240 MB. If your Lambda function uses more memory, its execution will be terminated.

It is also important to know that CPU and networking capacity are allocated to a Lambda function based on the provisioned memory as well. So, if you are running computing- or network-intensive work within a Lambda function, increasing the provisioned memory will probably improve performance.

At the same time, the default limit for the maximum size of the compressed deployment package (zip file) is 250 MB. When executing your Lambda function, you can use up to 512 MB nonpersistent disk space mounted to `/tmp`. Look at “Lambda Quotas” at **http://mng.bz/vXda** if you want to learn more about Lambda’s limitations.

### 6.4.2 Effects of the serverless pricing model

When launching a virtual machine, you have to pay AWS for every operating hour, billed in second intervals. You are paying for the machines whether or not you are using the resource they provide. Even when nobody is accessing your website or using your application, you are paying for the virtual machine.

That's totally different with AWS Lambda. Lambda is billed per request. Costs occur only when someone accesses your website or uses your application. That's a game changer, especially for applications with uneven access patterns or for applications that are used rarely. Table 6.3 explains the Lambda pricing model in detail. Please note: when creating a Lambda function, you select the architecture. As usual, the Arm architecture based on AWS Graviton is the cheaper option.

Table 6.3 AWS Lambda pricing model

| | Free Tier | x86 | Arm |
|---|---|---|---|
| Number of Lambda function invocations | First 1 million requests every month are free. | $0.0000002 per request | $0.0000002 per request |
| Duration billed in 1 ms increments based on the amount of memory you provisioned for your Lambda function | Using the equivalent of 400,000 seconds of a Lambda function with 1 GB is free of charge provisioned memory every month. | $0.0000166667 for using 1 GB for one second | $0.0000133334 for using 1 GB for one second |

Free Tier for AWS Lambda

The Free Tier for AWS Lambda does not expire after 12 months. That's a huge difference compared to the Free Tier of other AWS services (such as EC2) where you are eligible for the Free Tier only within the first 12 months, after creating an AWS account.

Sounds complicated? Figure 6.20 shows an excerpt of an AWS bill. The bill is from November 2017 and belongs to an AWS account we are using to run a chatbot (see **https://marbot.io**). Our chatbot implementation is 100% serverless. The Lambda functions were executed 1.2 million times in November 2017, which results in a charge of $0.04. All our Lambda functions are configured to provision 1536 MB memory. In total, all our Lambda functions have been running for 216,000 seconds, or around 60 hours, in November 2017. That's still within the Free Tier of 400,000 seconds with 1 GB provisioned memory every month. So, in total we had to

pay $0.04 for using AWS Lambda in November 2017, which allowed us to serve around 400 customers with our chatbot.

Figure 6.20 Excerpt from our AWS bill from November 2017 showing costs for AWS Lambda

This is only a small piece of our AWS bill, because other services we used together with AWS Lambda—for example, to store data—add more significant costs to our bill.

Don't forget to compare costs between AWS Lambda and EC2. Especially in a high-load scenario with more than 10 million requests per day, using AWS Lambda will probably result in higher costs compared to using EC2. But comparing infrastructure costs is only one part of what you should be looking at. Consider the total cost of ownership (TOC), including costs for managing your virtual machines, performing load and resilience tests, and automating deployments. Our experience has shown that the total cost of ownership is typically lower when running an application on AWS Lambda compared to Amazon EC2.

The last part of the chapter focuses on additional use cases for AWS Lambda besides automating operational tasks, as you have done thus far.

### 6.4.3 Use case: Web application

A common use case for AWS Lambda is building a backend for a web or mobile application. As illustrated in figure 6.21, an architecture for a serverless web application typically consists of the following building blocks:

- *Amazon API Gateway*—Offers a scalable and secure REST API that accepts HTTPS requests from your web application's frontend or mobile application.
- *AWS Lambda*—Lambda functions are triggered by the API gateway. Your Lambda function receives data from the request and returns the data for the response.
- *Object store and NoSQL database*—For storing and querying data, your Lambda functions typically use additional services offering object storage or NoSQL databases, for example.

Figure 6.21 A web application build with API Gateway and Lambda

Do you want to get started building web applications based on AWS Lambda? We recommend *AWS Lambda in Action* from Danilo Poccia (Manning, 2016).

### 6.4.4 Use case: Data processing

Another popular use case for AWS Lambda follows: event-driven data processing. Whenever new data is available, an event is generated. The event triggers the data processing needed to extract or transform the data. Figure 6.22 shows an example:

1. The load balancer collects access logs and uploads them to an object store periodically.
2. Whenever an object is created or modified, the object store triggers a Lambda function automatically.
3. The Lambda function downloads the file, including the access logs, from the object store and sends the data to an Elasticsearch database to be available for analytics.



Figure 6.22 Processing access logs from a load balancer with AWS Lambda

We have successfully implemented this scenario in various projects. Keep in mind the maximum execution limit of 900 seconds when implementing data-processing jobs with AWS Lambda.

### 6.4.5 Use case: IoT backend

The AWS IoT service provides building blocks needed to communicate with various devices (things) and build event-driven applications. Figure 6.23 shows an example. Each thing publishes sensor data to a message broker. A rule filters the relevant messages and triggers a Lambda function. The Lambda function processes the event and decides what steps are needed based on the business logic you provide.



Figure 6.23 Processing events from an IoT device with AWS Lambda

We built a proof of concept for collecting sensor data and publishing metrics to a dashboard with AWS IoT and AWS Lambda, for example.

We have gone through three possible use cases for AWS Lambda, but we haven't covered all of them. AWS Lambda is integrated with many other

services as well. If you want to learn more about AWS Lambda, we recommend the following books:

- *AWS Lambda in Action* by Danilo Poccia (Manning, 2016) is an example-driven tutorial that teaches how to build applications using an event-based approach on the backend.
- *Serverless Architectures on AWS*, second edition, by Peter Sbarski (Manning, 2022) teaches how to build, secure, and manage serverless architectures that can power the most demanding web and mobile apps.

## Summary

- AWS Lambda allows you to run your C#/.NET Core, Go, Java, JavaScript/ Node.js, Python and Ruby code within a fully managed, highly available, and scalable environment.
- The Management Console and blueprints offered by AWS help you to get started quickly.
- By using a schedule expression, you can trigger a Lambda function periodically. This is comparable to triggering a script with the help of a cron job.
- CloudWatch is the place to go when it comes to monitoring and debugging your Lambda functions.
- The Serverless Application Model (SAM) enables you to deploy a Lambda function in an automated way with AWS CloudFormation.
- Many event sources exist for using Lambda functions in an event-driven way. For example, you can subscribe to events triggered by CloudTrail for every request you send to the AWS API.
- The most important limitation of a Lambda function is the maximum duration of 900 seconds per invocation.
- AWS Lambda can be used to built complex services as well, from typical web applications, to data analytics, and IoT backends.