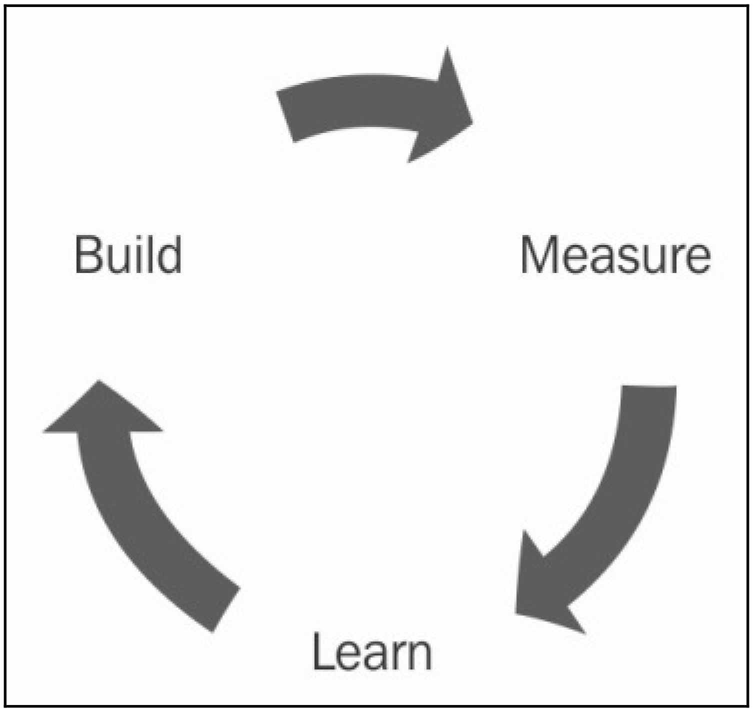
# 1
# Monitoring and Alerting

In previous chapters, we built a state of the art infrastructure and implemented a number of engineering best practices following the DevOps principles. One of the principle we haven't covered yet is the concept of measuring everything.

The core concept of measuring everything is the goal of collecting actionable feedback. We want to create the following feedback loop that will let us assess the impact of a change:



This idea isn't unique to DevOps. Most reputable companies will rely on similar systems to dynamically steer their teams in the right direction, as intuition and gut feeling isn't enough anymore when making most decisions and trying to stay competitive.

By applying this concept to our infrastructure and services, we can take them to the next level and implement a monitoring and alerting solution, which is, of course, a must to have for any production environment. In the first part of the chapter, we will make changes to our application to better expose how our application is behaving. Following this, we will do the same to our infrastructure. Thanks to our understanding of **infrastructure as code**, we will be able to add those crucial components by extending the different CloudFormation templates we created.

Finally, we will implement an alert functionality on some of the public key metrics indicators to help us improve the availability of our application. This chapter will contain the following sections:

- Instrumenting our application for monitoring
- Monitoring our infrastructure
- Creating alarms using CloudWatch and SNS

## Technical requirements

- AWS CloudWatch
- ELK (ElasticSearch, Logstash, and Kibana)
- Kinesis Firehose
- Ansible
- CloudFormation
- `https://github.com/yogeshraheja/Effective-DevOps-with-AWS/blob/master/Chapter08/EffectiveDevOpsTemplates/elasticsearch-cf-template.py`
- `https://github.com/yogeshraheja/Effective-DevOps-with-AWS/blob/master/Chapter08/EffectiveDevOpsTemplates/firehose-cf-template.py`

## Instrumenting our application for monitoring

In this section, we are going to make a couple of changes to our application to provide insight into what our code is doing and how it's behaving.

Because monitoring isn't as trivial as it may sound, there is no shortage of monitoring solutions. Since this book is focused on AWS, we will want to utilize what AWS provides as much as possible, starting with CloudWatch.

Furthermore, because of the rudimentary nature of the application, most of what we will implement won't be very meaningful, aside from demonstrating you different options, as well as the ideas behind the process.

## AWS CloudWatch

CloudWatch centralizes most essential functionalities for a monitoring solution. We used some of its functionalities previously when we created our Auto Scaling Groups and needed an alarm to trigger Auto Scaling Events, but CloudWatch can do a lot more.

In the world of infrastructure, data mostly comes in two forms—metrics and logs. CloudWatch supports both data types. In addition, they also have a third type of data called **events**.

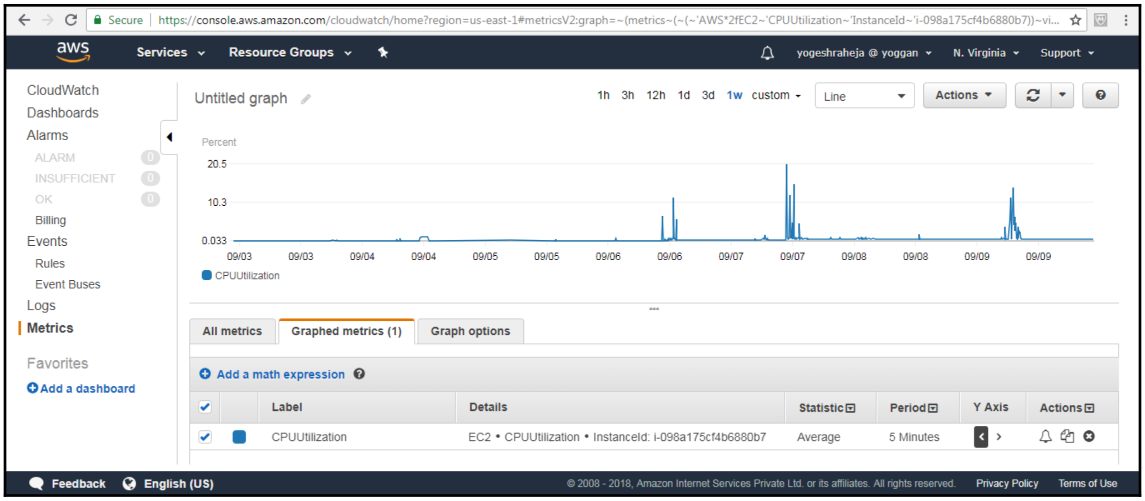
As with most services, you can access it using the web console, the command-line interface, and, of course, the API and various SDKs. We will first look at the different types of data.

### Metrics

Metrics are often used to monitor things that can be quantified, such as system metrics (CPU utilization, free memory, network consumption), page views, or HTTP status (the current error rate in my application). In CloudWatch, metrics are defined as tuples and contain the following:

- Resource ID
- Service name
- Metric name
- A metric value
- A timestamp

For example,



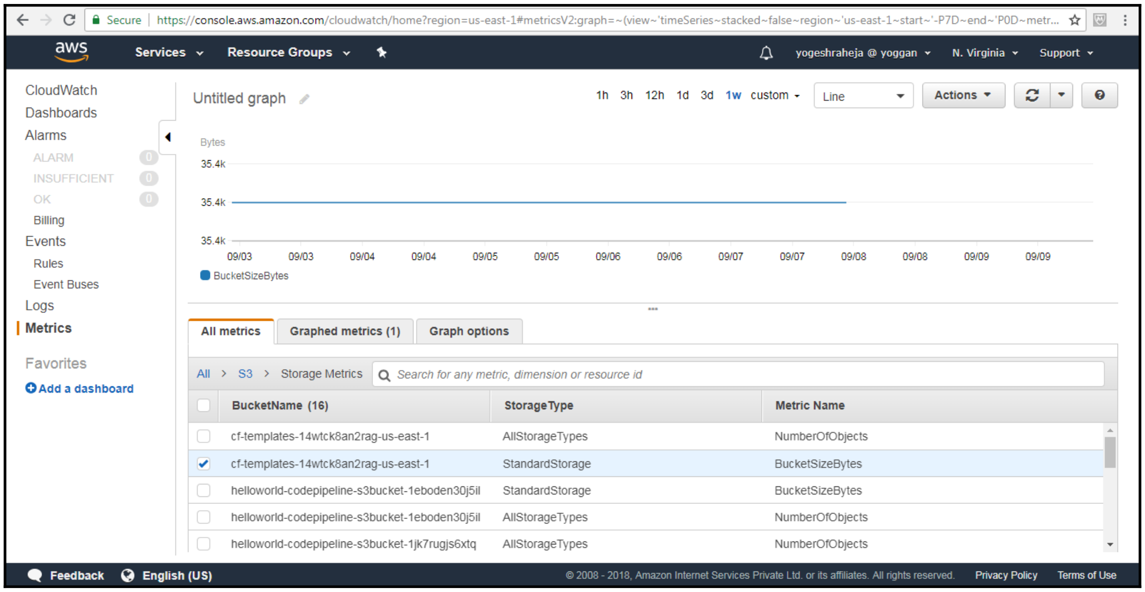
| ✓ | | Label | Details | Statistic | Period | Y Axis | Actions |
|---|---|---|---|---|---|---|---|
| ✓ | 🟦 | CPUUtilization | EC2 • CPUUtilization • InstanceId: i-098a175cf4b6880b7 | Average | 5 Minutes | < > | |

The above screenshot shows that the CPU utilization of the EC2 instance ID **i-098a175cf4b6880b7** was at Average.

Most of the AWS Services will get integrated natively with CloudWatch. By going to `https://console.aws.amazon.com/cloudwatch`, you can start browsing the different metrics already generated by the different services we used using the metrics menu on the left-hand side or the **Browse Metrics** button on the **Metrics Summary** page.

For example, we can display a metric representing how much data we have in our S3 bucket as follows:

1. From the CloudWatch dashboard, click on **Browse Metrics**.
2. Select the **S3** service from the **Namespaces** section.
3. Select **Storage Metrics**.
4. Find the bucket used to store artifacts and pick the metric **BucketSizeBytes**:



| | BucketName (16) | StorageType | Metric Name |
|---|---|---|---|
| | cf-templates-14wtck8an2rag-us-east-1 | AllStorageTypes | NumberOfObjects |
| ✓ | cf-templates-14wtck8an2rag-us-east-1 | StandardStorage | BucketSizeBytes |
| | helloworld-codepipeline-s3bucket-1eboden30j5il | StandardStorage | BucketSizeBytes |
| | helloworld-codepipeline-s3bucket-1eboden30j5il | AllStorageTypes | NumberOfObjects |
| | helloworld-codepipeline-s3bucket-1jk7rugjs6xtq | AllStorageTypes | NumberOfObjects |

## Logs

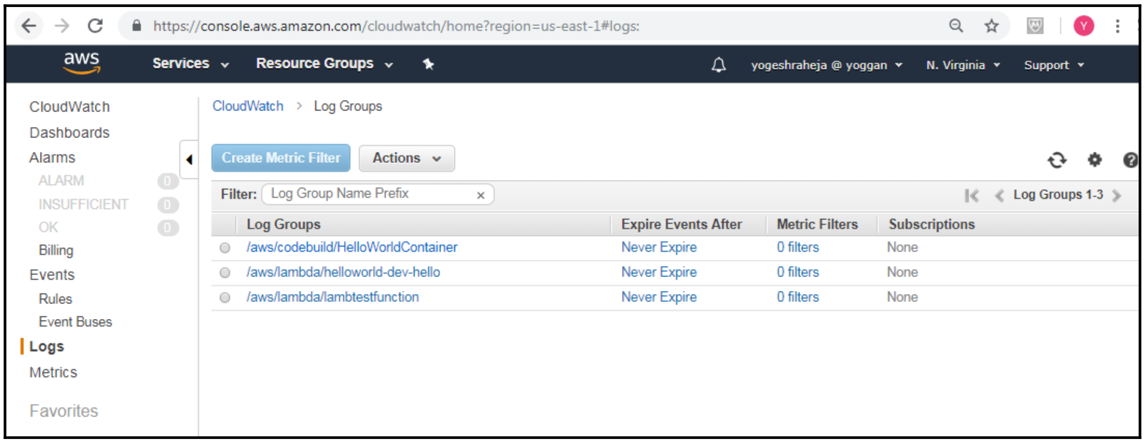
Log files are probably the most well-known way of monitoring systems. They are a great complement to metrics as they provide more flexibility. Because you aren't limited to a key-value pair system, like our metrics, you can use log files to provide very detailed information on events occurring in your application. For instance, you may capture, through your metric system, an increase in the error rate of your application, but to know what exactly is happening, you will want to access your application logs to see if there are exceptions, stack traces, or error messages that can help you troubleshoot that issue. The downside of logs is that they are much bigger than metrics. This means that they are more expensive to store, but also harder to index, search, and aggregate.

CloudWatch logs are organized around a few key concepts:

- Each log is called a log event and contains a raw message and a timestamp. The logs events produced by a unique source are grouped into a log stream.
- Log streams send their log event to log groups. Each log group has its own policy in terms of data retention (how many days you want to keep your log event for, who can access those logs, and so on).

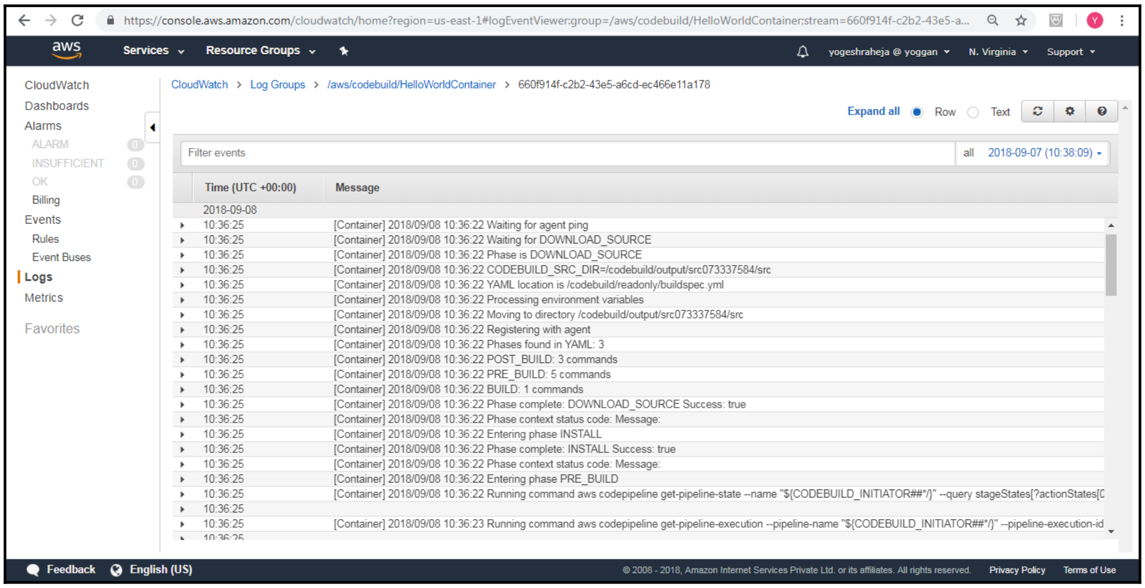
As an example of that, we can retrieve the events produced by our CodeBuild execution logs:

1. In your browser, open the CloudWatch service at `https://console.aws.amazon.com/cloudwatch`.
2. Click on **Logs** in the left-hand side menu.
3. From there, you can see the different log groups. Select one of the `/aws/codebuild/` groups to access the log streams, as shown in the following screenshot:



| Log Groups | Expire Events After | Metric Filters | Subscriptions |
|---|---|---|---|
| /aws/codebuild/HelloWorldContainer | Never Expire | 0 filters | None |
| /aws/lambda/helloworld-dev-hello | Never Expire | 0 filters | None |
| /aws/lambda/lambtestfunction | Never Expire | 0 filters | None |

4. Open one of the log streams to access the logs produced by CodeBuild:


https://console.aws.amazon.com/cloudwatch/home?region=us-east-1#logEventViewer:group=/aws/codebuild/HelloWorldContainer;stream=660f914f-c2b2-43e5-a...
Services
Resource Groups
yogeshraheja @ yoggan
N. Virginia
Support
CloudWatch
Dashboards
Alarms
ALARM
INSUFFICIENT
OK
Billing
Events
Rules
Event Buses
Logs
Metrics
Favorites
CloudWatch > Log Groups > /aws/codebuild/HelloWorldContainer > 660f914f-c2b2-43e5-a6cd-ec466e11a178
Expand all
Row
Text
Filter events
all
2018-09-07 (10:38:09)
Feedback
English (US)
© 2008 - 2018, Amazon Internet Services Private Ltd. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

```
Time (UTC +00:00)   Message
2018-09-08
10:36:25            [Container] 2018/09/08 10:36:22 Waiting for agent ping
10:36:25            [Container] 2018/09/08 10:36:22 Waiting for DOWNLOAD_SOURCE
10:36:25            [Container] 2018/09/08 10:36:22 Phase is DOWNLOAD_SOURCE
10:36:25            [Container] 2018/09/08 10:36:22 CODEBUILD_SRC_DIR=/codebuild/output/src073337584/src
10:36:25            [Container] 2018/09/08 10:36:22 YAML location is /codebuild/readonly/buildspec.yml
10:36:25            [Container] 2018/09/08 10:36:22 Processing environment variables
10:36:25            [Container] 2018/09/08 10:36:22 Moving to directory /codebuild/output/src073337584/src
10:36:25            [Container] 2018/09/08 10:36:22 Registering with agent
10:36:25            [Container] 2018/09/08 10:36:22 Phases found in YAML: 3
10:36:25            [Container] 2018/09/08 10:36:22 POST_BUILD: 3 commands
10:36:25            [Container] 2018/09/08 10:36:22 PRE_BUILD: 5 commands
10:36:25            [Container] 2018/09/08 10:36:22 BUILD: 1 commands
10:36:25            [Container] 2018/09/08 10:36:22 Phase complete: DOWNLOAD_SOURCE Success: true
10:36:25            [Container] 2018/09/08 10:36:22 Phase context status code: Message:
10:36:25            [Container] 2018/09/08 10:36:22 Entering phase INSTALL
10:36:25            [Container] 2018/09/08 10:36:22 Phase complete: INSTALL Success: true
10:36:25            [Container] 2018/09/08 10:36:22 Phase context status code: Message:
10:36:25            [Container] 2018/09/08 10:36:22 Entering phase PRE_BUILD
10:36:25            [Container] 2018/09/08 10:36:22 Running command aws codepipeline get-pipeline-state --name "${CODEBUILD_INITIATOR##*/}" --query stageStates[?actionStates[0
10:36:25
10:36:25            [Container] 2018/09/08 10:36:23 Running command aws codepipeline get-pipeline-execution --pipeline-name "${CODEBUILD_INITIATOR##*/}" --pipeline-execution-id
10:36:25
```

### Events

CloudWatch Events are a concept particular to AWS. You can see them as a hybrid of logs and metrics. Events have identifiers and context the same way metrics have a name and resources ID, but can also carry a payload with custom information. AWS uses it extensively in their infrastructure and services. Every time resources in your environment change, AWS creates an event that goes into a stream that the CloudWatch events service can subscribe to. You can create rules to match events of interest and either send the information to a service, such as SQS or SNS, or directly execute code using some pre-program functionalities or Lambda.

## Using CloudWatch to monitor our helloworld application

Now that we know a bit more about the different monitoring functionalities that CloudWatch offers, we will make changes in our `helloworld` application to get the best out of CloudWatch. We will first look at producing better logs. Following this, we will add metrics and finally events. Once the changes are in place, we will then make some changes to our infrastructure and its permission to start collecting that data.

## Adding logs to our application

When we initially created our application, we added a console log to state that the application is running on the last line:

```
console.log('Server running')
```

As you might imagine, this is not enough. In order to improve this, we will create a new logger.

### Creating a custom logger for our application

To be useful, the logs need to be put in a context. In an environment where things are quickly changing, you want to provide some extra information in your messages, including the type of log (info, warning, critical, and so on), which version of your application produced it, and an accurate timestamp of when the error was produced. If your logs are all aggregated in the same place, you may also want to include the name of the service and the server that produced it. We will change our code to include that information.

When we first created the application, we kept the code to a bare minimum and avoided the use of extra packages. This made it easy to initially deploy the service. Now that we have more tooling around it, adding extra libraries will not be an issue. To improve our logging, we will rely on a library called **winston** (`https://www.npmjs.com/package/winston`). Winston is one of the most common logging libraries in JavaScript. The library has many features and lets you manipulate your logs in several powerful ways, as we will see shortly.

On your server, go to the root directory of the `helloworld` application:

```
$ cd helloworld
```

Then install the `winston` library, as follows:

```
$ npm install winston@2.4.4 --save-dev
```

Adding the `--save` option will make NPM include the package definition in the `package.json` file. With your text editor, open the `helloworld.js` file containing your application. After the initialization of the `http` variable, we will initialize the `winston` library as follows:

```
var http = require("http")
var winston = require("winston")
```

Next, we will create a new variable to specify the version of the code. One of the ways we can achieve this is by assuming that this information will be provided later through the use of environment variables. For now, we will simply import the values as follows:

```
var version = process.env.HELLOWORLD_VERSION
```

We will now create our custom logger. This will allow us to specify that we want a timestamp and provide the code to define that timestamp:

```
var logger = new winston.Logger({
 transports: [new winston.transports.Console({
 timestamp: function() {
 var d = new Date()
 return d.toISOString()
 },
 })]
})
```

To add the remaining context to all our logs, we will use a feature of the library called a **rewriter**, which allows us to modify the content of the meta.

Under the definition of the var logger variable definition, add the following:

```
logger.rewriters.push(function(level, msg, meta) {
  meta.version = version
  return meta
})
```

Lastly, we now need to use the logger we just created and configured. For that, we will select the following code fragment:

```
console.log("Server running")
```

We replace this code fragment with the following:

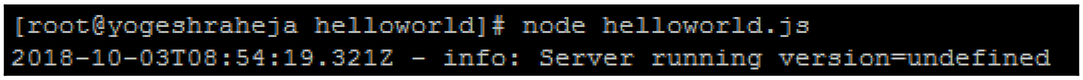
```
logger.info("Server running")
```

We can now test our application by running it locally:

```
$ node helloworld.js
```

As the application starts, it produces logs that look as follows:

```
2018-10-03T08:54:19.321Z - info: Server running version=undefined
```



```
[root@yogeshraheja helloworld]# node helloworld.js
2018-10-03T08:54:19.321Z - info: Server running version=undefined
```

Your `helloworld.js` file should look as shown at: `https://raw.githubusercontent.com/yogeshraheja/Effective-DevOps-with-AWS/master/Chapter08/helloworld/helloworld-part-1.js`. The collection of logs on EC2 and ECS are typically different. We will show how to make both collections starting with EC2.

At this point, the console log isn't captured anywhere on EC2. We need to make some changes to our upstart script to save the console log and set the `HELLOWORLD_VERSION` environment variable.

## Making changes to provide the version and save the console log

Our upstart configuration is located in the subdirectory scripts. With your code editor, open the file scripts`/helloworld.conf`.

We will edit the script section of the file as follows:

```
script
    set -a
    . /usr/local/helloworld/version mkdir -p /var/log/helloworld/
    chown ec2-user:ec2-user /var/log/helloworld/
    exec su --session-command="/usr/bin/node
/usr/local/helloworld/helloworld.js >> /var/log/helloworld/helloworld.log
2>&1" ec2-user
end script
```

The first set of changes will allow us to define the `HELLOWORLD_VERSION` environment variable. To get there, we will add a call to `set -a` to force our variables to be exported and source a new file, `/usr/local/helloworld/version`, which we will create later. The second part of the changes will allow us to log the console output onto the filesystem. To do that, we need to create a directory in `/var/log` and change the command that starts the application to save `stdout` and `stderr` into that new directory.

We now have a log file containing our console log. Whenever you add new logs, you should always think about how to rotate those logs. We will do that using `logrotate`.

We will create a new folder and configuration file for it. In the root of the `helloworld` directory, create a new folder `conf` and a new file called `logrotate`:

```
$ mkdir conf
$ touch conf/logrotate
```

We will now edit that file and put the following configuration in it:

```
/var/log/helloworld/*.log {
rotate 3
size=100M
copytruncate
nocompress
}
```

You may, of course, adjust it to your liking.

We will now address the creation of the `/usr/local/helloworld/version` file. Our goal is to generate a new version file every time new code is released with CodeDeploy. One of the functionalities that we didn't cover is that whenever CodeDeploy runs, it sets some environment variables of its own. We will use those to generate a version.

Create a new script in the script directory, call it `setversion.sh`, and set its permissions to be executable:

```
$ touch scripts/setversion.sh
$ chmod +x scripts/setversion.sh
```

Open the file and simply put the following:

```
#!/bin/sh
echo "HELLOWORLD_VERSION=${APPLICATION_NAME}-${DEPLOYMENT_GROUP_NAME}-
${DEPLOYMENT_GROUP_ID}-${DEPLOYMENT_ID}" > /usr/local/helloworld/version
```

We can now make changes to CodeDeploy to incorporate all our changes.

## Making changes to CodeDeploy to better handle logging

We will first make CodeDeploy generate the version file. To do that, we will open the `appspec.yml` file at the root of the `helloworld` application directory.

At the bottom of the file, after the `ValidateService` hook, we will use a new hook to trigger our `setversion` script. This operation will need to happen after the `helloworld` application is installed. As such, we will want to add the following hook:

```
AfterInstall:
  - location: scripts/setversion.sh
    timeout: 180
```

We now need to handle our new files. The first one is the `logrotate` configuration. In the files section at the top of our `appsec` file, add the following:

```
- source: conf/logrotate
  destination: /etc/logrotate.d/helloworld
```

Lastly, we need to handle the installation of our libraries now that we have some extra dependencies. There are a few ways to deal with dependencies. The most obvious one is to run `npm install` during the deployment. This allows you to keep your code base small and light. The downside is that you now rely on `https://www.npmjs.com/` to be working to get a deployment out, which can be risky.

Imagine the scenario where you have to deploy a fix for a major bug, but can't because the installation of your dependencies is failing. To avoid this, it is best practice to also commit the `node_modules` directory with your code.

After the previous configuration to deploy the `logrotate` configuration, add the following:

```
- source: node_modules
  destination: /usr/local/helloworld/node_modules
```

Your new `appsec` file should look like this `https://raw.githubusercontent.com/yogeshraheja/Effective-DevOps-with-AWS/master/Chapter08/helloworld/appspec.yml`.

We can now commit all the changes:

```
$ git add helloworld.js node_modules package.json conf
scripts/setversion.sh appsec.yml scripts/helloworld.conf
$ git commit -m "Adding logging to helloworld"
$ git push
```

Thanks to our pipelines, those changes will be deployed to our EC2 instances, and in no time, we will start seeing logs populated on the hosts.

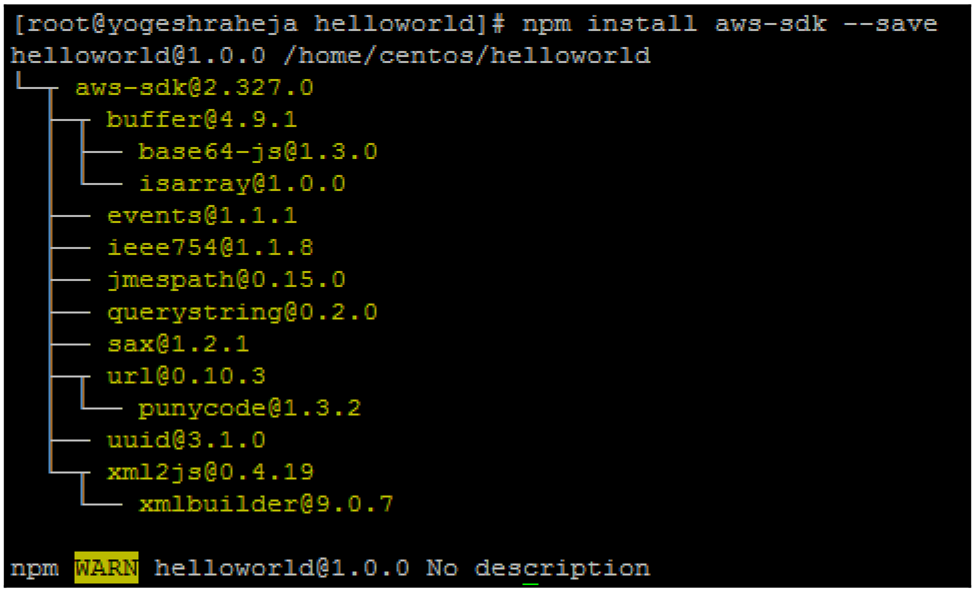
We will now add metrics and events to our application.

## Adding metrics and events to our application

As you might expect, there are several ways to add metrics. You can either opt for a generic protocol, such as StatsD, which will let you reuse your metrics across a variety of products, both SaaS and open source or, if you want to get something working quickly, use the AWS native SDK. Both options have pros and cons, but we will focus on the AWS native SDK.

We will once again start off from the root directory of our `helloworld` application. We will first start by installing the AWS SDK for JavaScript as follows:

```
$ npm install aws-sdk --save
```



```
[root@yogeshraheja helloworld]# npm install aws-sdk --save
helloworld@1.0.0 /home/centos/helloworld
└─┬ aws-sdk@2.327.0
  ├─┬ buffer@4.9.1
  │ ├── base64-js@1.3.0
  │ └── isarray@1.0.0
  ├── events@1.1.1
  ├── ieee754@1.1.8
  ├── jmespath@0.15.0
  ├── querystring@0.2.0
  ├── sax@1.2.1
  ├─┬ url@0.10.3
  │ └── punycode@1.3.2
  ├── uuid@3.1.0
  └─┬ xml2js@0.4.19
    └── xmlbuilder@9.0.7

npm WARN helloworld@1.0.0 No description
```

This will install into the `node_modules` directory the library and its dependencies and update the `package.json` file.

Once the library is installed, open the `helloworld.js` file with your editor. We will first instantiate the library.

At the top of the file, after the initialization of the `winston` library, add the following:

```
var http = require("http")
var winston = require("winston")
var AWS = require("aws-sdk")
```

The AWS object will let us interact with all AWS services. Our infrastructure is located in `us-east-1`. We will configure our application to use this region to access the different services. After the definition of the AWS variable, add the following:

```
AWS.config.update({region:"us-east-1"})
```

Next, we will create a variable to access the CloudWatch metric and event service:

```
var cwevents = new AWS.CloudWatchEvents({apiVersion: "2015-10-07"})
var cw = new AWS.CloudWatch({apiVersion: "2010-08-01"})
```

In a more realistic world, an application will have a different use case for events and metrics. You would add an event if you wish to run a Lambda function when a specific event happens, while you would add metrics extensively to measure everything that's happening in your application. Here, since our application is only doing one thing, we don't need that luxury, and we will create a metric and an event whenever someone accesses the application.

We will add an event inside the `http`. We will start by creating an event variable to define a simple event. We will join the content of the request header to that event:

```
http.createServer(function (request, response) {
  var event = {
    Entries: [{
      Detail: JSON.stringify(request.headers),
      DetailType: "hellworld application access request",
      Source: "helloworld.app"
    }]
  }
```

We will also create a metric. A lot of the time, you want your metrics to be associated with several dimensions. We will create a simple metric of `page_viewed`, but to illustrate the concept of dimensions better, we will associate the version of our application with the metric.

We can do that as follows:

```
var metric = {
  MetricData: [{
    MetricName: "page_viewwed",
    Dimensions: [{
      Name: "Version",
      Value: version
    }],
    Unit: "None",
    Value: 1.0
  }],
  Namespace: "Helloworld/traffic"
}
```

We will see the metrics used to monitor disk space on the root partition of our instances a bit later in this chapter. In that case, for example, the dimensions will be partitions, mount paths, and instance-id.

We will now emit those two events just after the `response.end("Hello World\n")` call, as follows:

```
response.end('Hello World\n')
   cwevents.putEvents(event, function(err, data) {
    if (err) {
      logger.error("error", "an error occurred when creating an event",
{error: err})
    } else {
      logger.info("created event", {entries: data.Entries})
    }
  })
   cw.putMetricData(metric, function(err, data) {
    if (err) {
      logger.error("an error occurred when creating a metric", {error:
err});
    } else {
      logger.info("created metric", {data: JSON.stringify(data)});
    }
  })
```

Note how we are relying on the logging capability to catch possible issues with putting metrics or events into CloudWatch.

The `helloworld.js` should look like `https://raw.githubusercontent.com/yogeshraheja/Effective-DevOps-with-AWS/master/Chapter08/helloworld/helloworld.js`.

**Log cardinality**

If you look closely at the logs we just produced, you will notice that our logs are made of constant strings. The extra information, such as the detail of the error or the event ID, is provided in a separate meta.
Having structured logs makes querying and indexing a lot better. For example, thanks to that format, we will be able to extract all logs matching **an error occurred when creating a metric** and group the error message that follows. This will, in turn, allow us to state the reason why we fail to create a metric by count. We will see that in action a bit later when we send logs to ElasticSearch.

At this point, if we were to run this code on our EC2 instances, we would see some errors in our logs because we don't have the permissions to send logs and metrics to AWS. In addition, we are producing our logs, but we aren't sending them to the CloudWatch logs yet. We will address those issues in the next 2 sections.

## Sending logs, events, and metrics to CloudWatch from EC2

Our application is now producing logs, metrics, and events. We are going to make changes to Ansible and CloudFormation to collect all those elements. We will add a new service to collect the logs and grant sufficient permissions to our EC2 instance roles.

### Creating an Ansible role for CloudWatch logs

To send our logs to CloudWatch, AWS provides a daemon called **awslogs**. We are going to install and configure it through Ansible.

Go into your **ansible/roles** directory:

```
$ cd ansible/roles
```

Create a new role called `awslogs`:

```
$ ansible-galaxy init awslogs
- awslogs was created successfully
```

We will first edit the task file `awslogs/tasks/main.yml`. Our first operation will be to install the package. For that, we will use the `yum` module:

```
---
# tasks file for awslogs

- name: install awslogs
  yum:
    name: awslogs
    state: present
```

We will want to configure the service dynamically with Ansible. For that, we will want to create a handler to restart `awslogs` when the configuration changes.

Edit the file `awslogs/handlers/main.yml` and add the following:

```
---
# handlers file for awslogs
- name: restart awslogs
  service:
    name: awslogs
    state: restarted
```

We can now configure the service. You can refer to `http://amzn.to/2qMhaEt` for the full documentation of the configuration of the service. In our case, we will keep it very simple. The service is configured through a set of INI files. The first one goes into `/etc/awslogs/awslogs.conf`.

We will create the file using the file module from Ansible. Create a new file in `awslogs/files/`, call it `awslogs.conf`, and put the following in it:

```
[general]
state_file = /var/lib/awslogs/agent-state
```

Now that the file is created, we are going to copy it to its target destination, `/etc/awslogs/awslogs.conf`. For that, we will use the copy module. Back in the `awslogs/tasks/main.yml` task file, we will add the following:

```
- name: copy global configuration
  copy:
    src: awslogs.conf
    dest: /etc/awslogs
  notify: restart awslogs
```

Thanks to the notify handler we created, if we were to change our configuration file, `awslogs` would automatically restart and, through that, load the new configuration.

We now want to use Ansible to configure which file needs to be collected. We will do that by creating new INI files inside the `/etc/awslogs/config` directory. To make it easy to operate, we will create one INI file per log file we want to collect. We will take advantage of the INI file module that Ansible provides to implement that. At the bottom of the task file, create a new command as follows:

```
- name: configure awslogs to collect {{ file }} ini_file:
```

We want our role to be generic and able to collect a variety of logs. As such, we will take advantage of the variable system that Ansible provides. The idea is that whenever we call this role, we will provide the information needed, such as the file to collect and its name.

The INI module requires us to provide the path of the INI file we want to configure. We will do that as follows:

```
path: "/etc/awslogs/config/{{ name }}.conf"
```

Here as well, we are calling the variable name that will be provided when we instantiate the module. The section of the configuration file will be the filename we want to collect:

```
section: "{{ file }}"
```

Now that the section is created, we are going to configure the different options. We will want to configure six different options. To keep the code dry, we will take advantage of the `with_items` keyword to iterate over the list of options and values:

```
option: "{{ item.option }}"
value: "{{ item.value }}"
```

Finally, we can list the different options and values as follows:

```
  with_items:
    - { option: file, value: "{{ file }}" }
    - { option: log_group_name, value: "{{ file }}" }
    - { option: log_stream_name, value: "{instance_id}" }
    - { option: initial_position, value: "start_of_file" }
    - { option: buffer_duration, value: "5000" }
    - { option: datetime_format, value: "{{ datetime_format | default('%b
%d %H:%M:%S') }}" }
```

Here too, we rely heavily on the fact that most values will be provided later. Note how we are making the `datetime_format` field optional. If it's not provided, we will try to read our logs as if they were formatted by the syslog. You can refer to the Python `datetime.strptime()` documentation for the full list of format variables.

We will conclude this call with a notify to restart `awslogs` if the configuration changed:

```
notify: restart awslogs
```

The last call of our task file will be to start the service and enable it so that `awslogs` starts right away, and upon reboot:

```
- name: start awslogs and enable it
  service:
    name: awslogs
    state: started
    enabled: yes
```

Our role is ready, we will use it inside our `nodeserver.yml` file. We will collect `/var/log/messages` and `/var/log/helloworld/helloworld.log`. Each entry will have a unique name (`messages` and `helloworld`) and their full path. In addition, we need to specify the logging format that our `helloworld` application is using:

```
---
- hosts: "{{ target | default('localhost') }}"
  become: yes
  roles:
    - nodejs
    - codedeploy
    - { role: awslogs, name: messages, file: /var/log/messages }
    - {
        role: awslogs,
        name: helloworld,
        file: /var/log/helloworld/helloworld.log, datetime_format: "%Y-%m-
%dT%H:%M:%S.%f"
}
```

Your code should be similar to `https://raw.githubusercontent.com/yogeshraheja/Effective-DevOps-with-AWS/master/Chapter08/ansible/roles/awslogs/tasks/main.yml`.

Before we commit those changes, we are going to update our CloudFormation template to add proper permissions.

## Updating our CloudFormation template

The last time we worked on EC2 was in `Chapter 6`, *Scaling Your Infrastructure,* when we implemented the Auto Scaling Groups. We are going to edit the troposphere script we used for this and make the necessary changes.

Go to your template directory and with your text editor, open the file `nodeserver-cf-template.py` from our `EffectiveDevOpsTemplates` repository

Previously, we created a policy to allow access to S3, which we needed for CodeDeploy. We will add a second policy and grant access to CloudWatch, CloudWatch logs, and CloudWatch events. After the creation of the IAM policy `AllowS3`, add the following resource:

```
t.add_resource(IAMPolicy(
    "MonitoringPolicy",
    PolicyName="AllowSendingDataForMonitoring",
    PolicyDocument=Policy(
        Statement=[
            Statement(
                Effect=Allow,
                Action=[
                    Action("cloudwatch", "Put*"),
                    Action("logs", "Create*"),
                    Action("logs", "Put*"),
                    Action("logs", "Describe*"),
                    Action("events", "Put*"),
                ],
                Resource=["*"])
        ]
    ),
    Roles=[Ref("Role")]
))
```

We can save our template and generate the new CloudFormation template:

```
$ git add nodeserver-cf-template.py
$ git commit -m "Adding permissions to interact with CloudWatch Logs,
Events"
$ git push
```

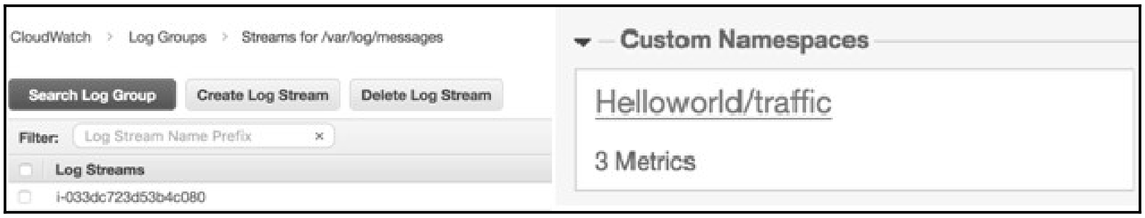
To update our existing stack, we are going to use the AWS CLI. In this particular instance, the main change is at the IAM level where we are creating the monitoring policy. The parameters we previously set when we initially created our stacks don't need to be changed. Instead of providing the same parameters again, we are going to use the `UsePreviousValue` option to update our `helloworld` stacks as follows:

```
$ python nodeserver-cf-template.py > nodeserver-cf.template
$ aws cloudformation update-stack \
    --capabilities CAPABILITY_IAM \
    --stack-name helloworld-staging \
    --template-body file://nodeserver-cf.template \
    --parameters \
         ParameterKey=InstanceType,UsePreviousValue=true \
         ParameterKey=KeyPair,,UsePreviousValue=true \
         ParameterKey=PublicSubnet,,UsePreviousValue=true \
            ParameterKey=ScaleCapacity,,UsePreviousValue=true \
            ParameterKey=VpcId,,UsePreviousValue=true

$ aws cloudformation update-stack \
    --capabilities CAPABILITY_IAM \
    --stack-name helloworld-production \
    --template-body file://nodeserver-cf.template \
    --parameters \
        ParameterKey=InstanceType,UsePreviousValue=true \
        ParameterKey=KeyPair,UsePreviousValue=true \
        ParameterKey=PublicSubnet,UsePreviousValue=true \
        ParameterKey=ScaleCapacity,UsePreviousValue=true \
        ParameterKey=VpcId,UsePreviousValue=true
```

Once the stack update is done, we can commit and merge our `ansible` changes. Your code should be similar to the one present at the following link: `https://raw.githubusercontent.com/yogeshraheja/Effective-DevOps-with-AWS/master/Chapter08/EffectiveDevOpsTemplates/nodeserver-cf-template-part-1.py`.

```
$ cd ansible
$ git add nodeserver.yml roles/awslogs
$ git commit -m "Adding awslogs role and permission to use CloudWatch"
$ git push

$ cd helloworld
$ git add .
$ git commit -m "Adding CloudWatch support to our application"
$ git push
```

Within a few minutes, you should be able to see your new log groups in CloudWatch under the **Logs** section, and inside them, the different log streams of our different hosts, and in the **Metrics** section, our `Helloworld/traffic` graph:



We now have an elegant solution to send logs from an EC2 instance into CloudWatch.

We won't cover this in the book, but CloudWatch has a dashboard feature that will let you create custom views to group some of the critical metrics. For example, if you are monitoring a web application, you may create a dashboard with your application error rate, latency, and **queries per second** (**QPS**).

**Average, 95th, and 99th percentile**

Averages can be misleading when looking at certain metrics. A classic example is latency. To monitor the latency of your application, you want to collect and graph the worst 95th and 99th percentiles as opposed to simply the mean or average. These two graphs will often tell you a different story of how some users are perceiving the latency on the site.

We now need to provide the same functionalities to ECS.

## Handling logs, events, and metrics in ECS

In the previous section, we added an extra step in our deployment process to identify and export the version for our application. In the case of EC2 and CodeDeploy, we created a version string using the deployment execution information. As such, we can easily correlate the logs produced by the deployment execution. In the case of ECS, what matters the most is to be able to identify the container ID within the ECR registry, as we are working with immutable containers. Therefore, we will update our code to use the container tag information as our application version.

In addition, we collected logs on EC2 instances using the `awslogs` agent. In the case of ECS, while we could do something similar by mounting the `/var/log` volume onto the ECS host and running the same agent, there is a much better way to do that.

ECS has many settings that we didn't explore, among them, the ability to configure environment variables and change how logs are managed. We will edit the troposphere script `helloworld-ecs-service-cf` created in the last chapter to send the logs produced in the console directly to the CloudWatch logs.

With your text editor, open the file `helloworld-ecs-service-cf-template.py`.

We will first add a new `troposphere.ecs` import as follows:

```
from troposphere.ecs import (
    TaskDefinition,
    ContainerDefinition,
    LogConfiguration,
    Environment,
)
```

We will use these classes inside the `TaskDefinition` section. Locate the

`TaskDefinition`, and after the port mapping definition, add the following to define our `HELLOWORLD_VERSION` variable and the logging configuration:

```
PortMappings=[ecs.PortMapping(
    ContainerPort=3000)],
Environment=[
    Environment(Name='HELLOWORLD_VERSION', Value=Ref("Tag"))
],
LogConfiguration=LogConfiguration(
    LogDriver="awslogs",
    Options={
        'awslogs-group': "/aws/ecs/helloworld",
        'awslogs-region': Ref("AWS::Region"),
    }
),
```

Once those changes are in place, we will create the volume group using the command-line interface:

```
$ aws logs create-log-group --log-group-name /aws/ecs/helloworld
```

In the last chapter, we created our cluster with all the permissions that we need to go through this chapter, therefore we won't need to do anything else to get our logs, events, and metrics sent to CloudWatch.

We can save the changes and commit them, as follows:

```
$ git commit -am "Configuring logging"
$ git push
```

You can then generate the new CloudFormation template and commit it to the template directory of our `helloworld` application.

```
$ cd helloworld
$ curl -L
https://raw.githubusercontent.com/yogeshraheja/Effective-DevOps-with-AWS/ma
ster/Chapter08/EffectiveDevOpsTemplates/helloworld-ecs-service-cf-
template.py| python > templates/helloworld-ecs- service-cf.template

$ git commit -am "Configuring logging"
$ git push
```

Thanks to our pipeline, a new version of the container will soon be deployed and you will be able to observe the logs and metrics produced by your container.

Our monitoring infrastructure is now looking good. We are collecting and indexing metrics, events, and logs. In most cases, this is enough to get started. We can improve our metrics by creating dashboards to display some of the key metrics and search in our logs for a particular event or timeframe. As applications get more complex, it is common for these types of monitoring architectures to reach their limits. Sometimes, you would like to be able to group logs to find out what type of errors are happening often, or do some complex queries. In addition, you may want to have a more hybrid approach to how you store your logs and keep them indexed for just a few days, but archive them on S3 for a much longer period. To do that, we will need a logging infrastructure made up of ElasticSearch, Kibana, and Kinesis Firehose.

**Creating a health check endpoint**

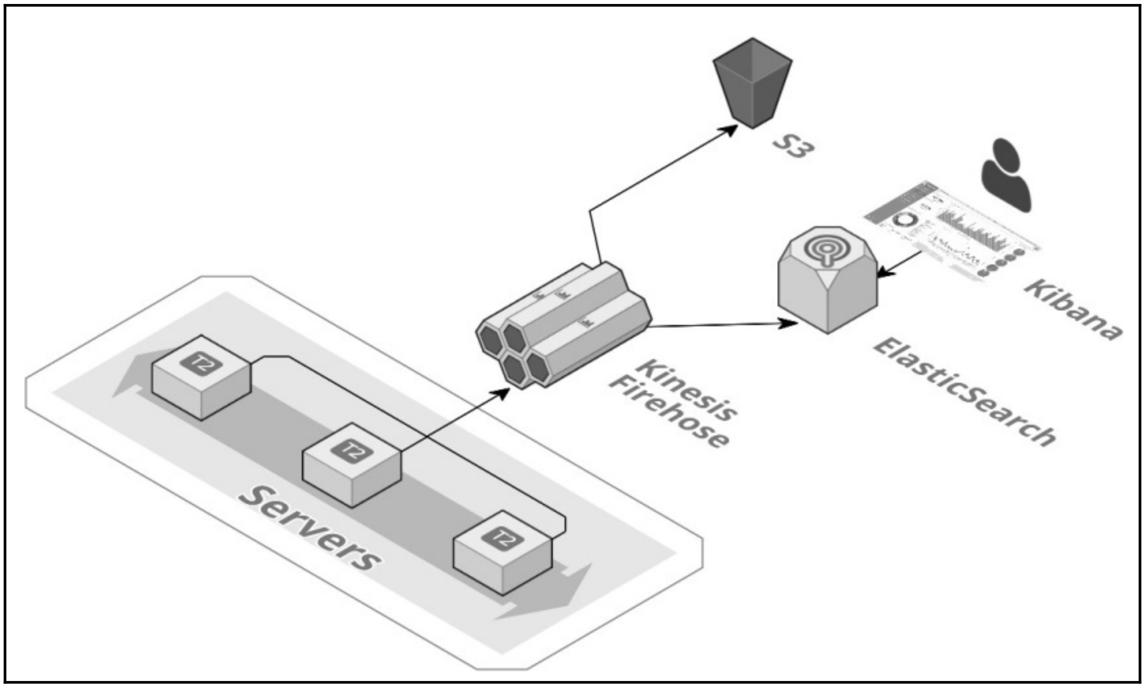
It is a good practice to create a route dedicated to monitoring in your application. This endpoint can then be used with your load balancers and ECS tasks to validate that the application is in a working state. The code behind that route will commonly check that the application can connect to your databases, storages, and other services that it depends on before returning an HTTP 200 (OK) to signal that the application is healthy.

## Advanced logging infrastructure with ElasticSearch, Kibana, and Firehose

In the world of telemetry, one of the favorite sets of tools that engineers like to use to store their logs is called the **ELK stack**. The ELK stack consists of ElasticSearch, Logstash, and Kibana. Logs are captured and filtered by Logstash, converted into JSON documents and sent to ElasticSearch, a distributed search and analytics engine. ElasticSearch is then queried via Kibana, which lets you visualize your data. You can look at this stack on `https://www.elastic.co/`.

AWS has a very similar system that you can use that also involves ElasticSearch and Kibana, but instead of Logstash, we will use Kinesis Firehose. That variation on the classic ELK stack is a very compelling option as you have even fewer services to manage, and potentially, the fact that Kinesis will retain information for up to five days makes it a better candidate than Logstash to transit your logs.

In addition, Kinesis will let us write our logs to both ElasticSearch and S3 such that if a log fails to be written to ElasticSearch, it will be saved to S3:



To create our stack, we will once again rely on CloudFormation templates and the troposphere. We will first create an ElasticSearch stack. AWS provides ElasticSearch as a service, and it comes with Kibana preinstalled and configured for your cluster.

Following this, we will create a Kinesis—the reasoning for Firehose stack. The reasoning for this is that you may want to use multiple Firehose streams for your different services, but also centralize all your logs into a single ElasticSearch cluster.

Once the new stack is in place, we will change our application a bit to deliver our logs to the Kinesis stream.

## Creating and launching an ElasticSearch cluster

As mentioned, AWS has a managed service for ElasticSearch. We will use it to create our cluster.

Create a new file and call it `elasticsearch-cf-template.py`. Our script will start almost like the `nodeserver-cf-template.py` file, but with a number of imports, including some for the `elasticsearch` service:

```
"""Generating CloudFormation template."""

from ipaddress import ip_network

from ipify import get_ip

from troposphere import (
    GetAtt,
    Join,
    Output,
    Export,
    Parameter,
    Ref,
    Template,
)

from troposphere.elasticsearch import (
    Domain,
    EBSOptions,
    ElasticsearchClusterConfig,
)
```

We will continue the script with the creation of the template and the extraction of the IP address. In the context of ElasticSearch, limiting who can access your cluster is very important, as there is no other authentication mechanism in place:

```
t = Template()
PublicCidrIp = str(ip_network(get_ip()))
```

We will now provide a brief description and collect the different parameters. The first parameter to select, the instance size. We will provide a few options here, but you can refer to `http://amzn.to/2s32Vvb` for the full list of available instance types:

```
t.add_description('Effective DevOps in AWS: Elasticsearch')

t.add_parameter(Parameter(
    "InstanceType",
    Type="String",
    Description="instance type",
    Default="t2.small.elasticsearch",
    AllowedValues=[
        "t2.small.elasticsearch",
        "t2.medium.elasticsearch",
        "m4.large.elasticsearch",
    ],
))
```

We will also provide the ability to set the number of instances present in our cluster. In the context of the book, we are assuming that the cluster will store just a few GB of logs. For bigger clusters, you may consider altering the template to also provide the ability to have dedicated master instances:

```
t.add_parameter(Parameter(
    "InstanceCount",
    Default="2",
    Type="String",
    Description="Number instances in the cluster",
))
```

The `t2` and `m4` instances don't come with any attached storage. We will use EBS volumes to store our logs. This next option will let us set the size of the EBS volumes:

```
t.add_parameter(Parameter( "VolumeSize", Default="10", Type="String",
Description="Size in Gib of the EBS volumes",
))
```

The different parameters we wish to configure are now all present. We can proceed with the creation of our ElasticSearch cluster. ElasticSearch clusters are called domains. We will create a domain resource and give it a name as follows:

```
t.add_resource(Domain(
    'ElasticsearchCluster',
    DomainName="logs",
```

We then configure which version of ElasticSearch to use. We will pick version 5.3 which is the most recent version of ElasticSearch released when this was published:

```
ElasticsearchVersion="5.3",
```

Next, we will configure our cluster. As mentioned earlier, we are assuming that the cluster will stay fairly small, and, therefore, we won't need dedicated master instances. For the same reason, we will also opt out of the zone awareness feature, which creates node replicas on the different AZ of the region the cluster is created in. Finally, we will reference the desired instance count and instance type from the parameters of the template:

```
ElasticsearchVersion="6.3",
    ElasticsearchClusterConfig=ElasticsearchClusterConfig(
        DedicatedMasterEnabled=False,
        InstanceCount=Ref("InstanceCount"),
        ZoneAwarenessEnabled=False,
        InstanceType=Ref("InstanceType"),
    ),
```

We will also want to specify a few advanced options as follows:

```
    AdvancedOptions={
           "indices.fielddata.cache.size": "",
           "rest.action.multi.allow_explicit_index": "true",
       },
```

After configuring the cluster, we will configure the EBS volume for our instances. Here too, we will reference our parameters to get the volume size of our volumes:

```
   EBSOptions=EBSOptions(EBSEnabled=True,
                              Iops=0,
                              VolumeSize=Ref("VolumeSize"),
                              VolumeType="gp2"),
```

We will conclude the creation of our domain with the configuration of the access policy:

```
    AccessPolicies={
           'Version': '2012-10-17',
           'Statement': [
               {
                   'Effect': 'Allow',
                   'Principal': {
                       'AWS': [Ref('AWS::AccountId')]
                   },
                   'Action': 'es:*',
                   'Resource': '*',
               },
               {
                   'Effect': 'Allow',
                   'Principal': {
                       'AWS': "*"
                   },
                   'Action': 'es:*',
                   'Resource': '*',
                   'Condition': {
                       'IpAddress': {
                           'aws:SourceIp': PublicCidrIp
                       }
                   }

               }
           ]
       },
   ))
```

Finally, we will conclude the creation of our template with two outputs and our final print statement. The output will be the Kibana URL and the `DomainArn` of our ElasticSearch domain, which we will use in the next section. To do so, we are going to export it under the name `LogsDomainArn`:

```
t.add_output(Output(
    "DomainArn",
    Description="Domain Arn",
    Value=GetAtt("ElasticsearchCluster", "DomainArn"),
    Export=Export("LogsDomainArn"),
))

t.add_output(Output(
    "Kibana",
    Description="Kibana url",
    Value=Join("", [
        "https://",
        GetAtt("ElasticsearchCluster", "DomainEndpoint"),
        "/_plugin/kibana/"
    ])
))

print t.to_json()
```

Our template is now completed. Your script should be similar to `http://bit.ly/2v3DHRG`. We can commit it, and create our ElasticSearch domain:

```
$ python elasticsearch-cf-template.py > elasticsearch-cf.template
$ git add elasticsearch-cf-template.py
$ git commit -m "Adding ElasticSearch template"
$ git push
$ aws cloudformation create-stack \
    --stack-name elasticsearch \
    --template-body file://elasticsearch-cf.template \
    --parameters \
        ParameterKey=InstanceType,ParameterValue=t2.small.elasticsearch \
        ParameterKey=InstanceCount,ParameterValue=2 \
        ParameterKey=VolumeSize,ParameterValue=10
```

Your script should look like `https://raw.githubusercontent.com/yogeshraheja/Effective-DevOps-with-AWS/master/Chapter08/EffectiveDevOpsTemplates/elasticsearch-cf-template.py`.

## Creating and launching a Kinesis Firehose stream

Within a few minutes, our ElasticSearch cluster should be up and running. We will now focus on the Kinesis Firehose component of our stack, which will let us feed data into ElasticSearch.

We will create a new script and call it `firehose-cf-template.py`.

The script starts as usual with several imports, the creation of a template variable, and a brief description:

```
"""Generating CloudFormation template."""

from troposphere import (
    GetAtt,
    Join,
    Ref,
    Template,
    ImportValue
)

from troposphere.firehose import (
    BufferingHints,
    CloudWatchLoggingOptions,
    DeliveryStream,
    S3Configuration,
    ElasticsearchDestinationConfiguration,
    RetryOptions,
)

from troposphere.iam import Role

from troposphere.s3 import Bucket

t = Template()

t.add_description('Effective DevOps in AWS: Kinesis Firehose Stream')
```

- `t.a dd_description('Effective DevOps in AWS: Kinesis Firehose Stream')`

The first resource we will create is an S3 bucket:

```
t.add_resource(Bucket(
    "S3Bucket",
    DeletionPolicy="Retain"
))
```

Following this, we will create a new role to give the permissions to our Firehose stream to communicate with ElasticSearch and S3. To save a bit of time, we are going to use some of the managed policies that AWS provides. In a production environment, these policies might be too open for your liking, and you may instead opt for writing your own:

```
t.add_resource(Role(
    'FirehoseRole',
    ManagedPolicyArns=[
        'arn:aws:iam::aws:policy/AmazonS3FullAccess',
        'arn:aws:iam::aws:policy/AmazonESFullAccess',
    ],
    AssumeRolePolicyDocument={
        'Version': '2012-10-17',
        'Statement': [{
            'Action': 'sts:AssumeRole',
            'Principal': {'Service': 'firehose.amazonaws.com'},
            'Effect': 'Allow',
        }]
    }
))
```

Finally, we will create our Firehose stream. We will create a new resource of the type `DeliveryStream` and give it the name `FirehoseLogs`:

```
    t.add_resource(DeliveryStream(
        'FirehoseLogs',
        DeliveryStreamName='FirehoseLogs',
```

DeliveryStreams can be used to deliver data to several services. In our case, we want it to deliver data to ElasticSearch. For that, we will create an `ElasticSearchDestinationConfiguration` parameter as follows:

```
    ElasticsearchDestinationConfiguration=ElasticsearchDestinationConfiguration
    (
```

The first piece of information we need to provide is the identifier of the ElasticSearch domain. We will do that by referencing the `LogsDomainArn` variable that we exported in the previous section:

```
    DomainARN=ImportValue("LogsDomainArn"),
```

Next, we will reference the IAM role we just defined before the creation of our `DeliveryStream`:

```
    RoleARN=GetAtt("FirehoseRole", "Arn"),
```

We will then specify the index name. You can picture that as the name of the database you want your logs to be in. We call our index logs to keep things simple:

```
    IndexName="logs",
```

In addition, in ElasticSearch, indices contain documents of different types (each type has its own name and mapping). We will name ours `Logs`:

```
    TypeName="Logs",
```

One of the common ways to shared data across an ElasticSearch cluster is to use temporal sharding. In our case, we will pick a daily rotation. For instance, an index containing the logs of March 24, 2020 will be called **logs-2020.03.24**. To do that, we will configure the `IndexRotationPeriod` to rotate logs every day:

```
    IndexRotationPeriod="OneDay",
```

Occasionally, ElasticSearch might get congested and won't reply right away. We will configure our stream to retry delivering data for 5 minutes:

```
    RetryOptions=RetryOptions( DurationInSeconds="300" ),
```

Kinesis Firehose works by buffering data until you reach a certain duration or a certain size. We will set them to do the minimum which, is 1 minute and 1 MB:

```
    BufferingHints=BufferingHints( IntervalInSeconds=60, SizeInMBs=1 ),
```

At this point, Kinesis Firehose has all the information it needs to send data to ElasticSearch. We will now configure it to also store all those logs on S3. We will first configure the stream to back up all documents (the alternative is backing up only the ones that failed being inserted into ElasticSearch):

```
    S3BackupMode="AllDocuments",
```

Following this, we will configure the S3. This will involve configuring the buffering like we did for ElasticSearch, referencing the bucket we created at the beginning of the template, stating whether we want to compress those logs, referencing the prefix for our files and, finally entering the role to use for these operations:

```
     S3Configuration=S3Configuration(
                BufferingHints=BufferingHints(
                    IntervalInSeconds=300,
                    SizeInMBs=5
                ),
                BucketARN=Join("", [
                    "arn:aws:s3:::", Ref("S3Bucket")
                ]),
                CompressionFormat='UNCOMPRESSED',
                Prefix='firehose-logs',
                RoleARN=GetAtt("FirehoseRole", "Arn"),
            ),
        )
    ))
```

We will conclude our script by printing the JSON output of our template:

```
print t.to_json()
```

Your script should look as follows `https://raw.githubusercontent.com/yogeshraheja/Effective-DevOps-with-AWS/master/Chapter08/EffectiveDevOpsTemplates/firehose-cf-template.py`.

We can now commit the script, create the template and launch it:

```
$ git add firehose-cf-template.py
$ git commit -m "Adding Firehose template"
$ git push
$ python firehose-cf-template.py > firehose-cf.template
$ aws cloudformation create-stack \
    --stack-name firehose \
    --template-body file://firehose-cf.template \
    --capabilities CAPABILITY_IAM
```

At this point, we have a working Firehose-to-ElasticSearch pipeline (also known as the **EKK stack**—Amazon ElasticSearch Service, Amazon Kinesis, and Kibana). We now need to circle back to our application and make some changes to deliver our logs to it.

## Updating our application to send logs to the Firehose endpoint

Our options to send logs to Kinesis Firehose are very similar to CloudWatch logs. We could use the `aws-kinesis-agent` provided in the `yum` repo as a replacement for the `awslogs` agent, but instead, we will demonstrate another way to do that. We are going to make our application send its logs directly to Kinesis instead of writing them on disk first.

Before making those changes, we will adjust the permissions to allow our instances to interact with Firehose.

## Adding permissions to EC2 to communicate with Firehose

We will once again edit our `nodeserver-cf-template.py` script. Open the file with your editor and in the `MonitoringPolicy` policy, add the following to allow our EC2 instance to communicate with Firehose and put a record into the stream:

```
t.add_resource(IAMPolicy(
      "MonitoringPolicy",
       PolicyName="AllowSendingDataForMonitoring",
       PolicyDocument=Policy(
         Statement=[
            Statement(
                Effect=Allow,
                Action=[
                    Action("cloudwatch", "Put*"),
                    Action("logs", "Create*"),
                    Action("logs", "Put*"),
                    Action("logs", "Describe*"),
                    Action("events", "Put*"),
                    Action("firehose", "Put*"),
                  ],
                  Resource=["*"])
]
),
Roles=[Ref("Role")],
))
```

Save the new script, commit your changes, and, following the same step as before, deploy the new version of the template. Your new template should look like `https://raw.githubusercontent.com/yogeshraheja/Effective-DevOps-with-AWS/master/Chapter08/EffectiveDevOpsTemplates/nodeserver-cf-template-part-2.py`.

```
$ git add nodeserver-cf-template.py
$ git commit -m "Allowing our application to send logs to Firehose"
$ git push
$ python nodeserver-cf-template.py > nodeserver-cf.template
$ aws cloudformation create-stack \
    --capabilities CAPABILITY_IAM \
    --stack-name helloworld-staging \
    --template-body file://nodeserver-cf.template \
    --parameters \
        ParameterKey=InstanceType,ParameterValue=t2.micro \
        ParameterKey=KeyPair,ParameterValue=EffectiveDevOpsAWS \
        ParameterKey=PublicSubnet,ParameterValue=subnet-e67190bc\\,subnet-
        658b6149\\,subnet-
d890d3e4\\,subnet-6fdd7927\\,subnet-4c99c229\\,subnet-
        b03baebc \
        ParameterKey=ScaleCapacity,ParameterValue=3 \
        ParameterKey=VpcId,ParameterValue=vpc-4cddce2a

$ aws cloudformation create-stack \
    --capabilities CAPABILITY_IAM \
    --stack-name helloworld-production \
    --template-body file://nodeserver-cf.template \
    --parameters \
        ParameterKey=InstanceType,ParameterValue=t2.micro \
        ParameterKey=KeyPair,ParameterValue=EffectiveDevOpsAWS \
        ParameterKey=PublicSubnet,ParameterValue=subnet-e67190bc\\,subnet-
        658b6149\\,subnet-
d890d3e4\\,subnet-6fdd7927\\,subnet-4c99c229\\,subnet-
        b03baebc \
        ParameterKey=ScaleCapacity,ParameterValue=3 \
        ParameterKey=VpcId,ParameterValue=vpc-4cddce2a
```

If you have `helloworld-staging` and `helloworld-production` stack already present, then run the following command to update the stack.

```
$ aws cloudformation update-stack \
    --capabilities CAPABILITY_IAM \
    --stack-name helloworld-staging \
    --template-body file://nodeserver-cf.template \
    --parameters \
        ParameterKey=InstanceType,UsePreviousValue=true \
        ParameterKey=KeyPair,,UsePreviousValue=true \
        ParameterKey=PublicSubnet,,UsePreviousValue=true \
        ParameterKey=ScaleCapacity,,UsePreviousValue=true \
        ParameterKey=VpcId,,UsePreviousValue=true
$ aws cloudformation update-stack \
    --capabilities CAPABILITY_IAM \
    --stack-name helloworld-production \
    --template-body file://nodeserver-cf.template \
    --parameters \
        ParameterKey=InstanceType,UsePreviousValue=true \
        ParameterKey=KeyPair,UsePreviousValue=true \
        ParameterKey=PublicSubnet,UsePreviousValue=true \
        ParameterKey=ScaleCapacity,UsePreviousValue=true \
        ParameterKey=VpcId,UsePreviousValue=true
```

In the case of ECS, we already added the proper permissions in the last chapter when we created our clusters.

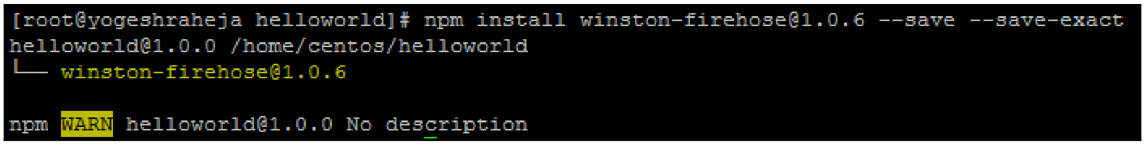
Now that this is in place, we will make changes to our code.

## Changing the logging transport to send logs to Firehose

Our logging library, `winston`, has a system to extend some of its functionalities, including its transport system. We will install a new transport system that can talk to Kinesis Firehose.

In your terminal, go to the root directory of your `helloworld` application and run the following command:

```
$ npm install winston-firehose@1.0.6 --save --save-exact
```



```
[root@yogeshraheja helloworld]# npm install winston-firehose@1.0.6 --save --save-exact
helloworld@1.0.0 /home/centos/helloworld
└── winston-firehose@1.0.6

npm WARN helloworld@1.0.0 No description
```

This will install specifically version 1.0.6 of the `winston-firehose` package and update the `package.json` accordingly. We need to enforce this version because our EC2 instance is running an old version of Node.js.

Then, open the `helloworld.js` file with your code editor. After the declaration of our `winston` variable, we will define a new variable as follows:

```
var WFirehose = require('winston-firehose')
```

When we used the `awslogs` agent or the ECS logging driver, the service was able to specify the hostname or container ID that the logs were coming from. In addition, each log file was in its own log group, which made it easy to identify what service and what instance of this service emitted a given log. The new architecture we are migrating to doesn't offer this. We will make some changes to our code to expose the service name and the host that produced the logs.

After the creation of the version variable, add the following:

```
var hostname = process.env.HOSTNAME
```

This will get the hostname of the server from our environment. A bit below, we will edit the rewriter to include this extra information as follows:

```
logger.rewriters.push(function(level, msg, meta) {
    meta.version = version
    meta.hostname = hostname
    meta.appname = "helloworld"
    return meta
})
```

Lastly, we will replace the logger variable definition. Find the following code:

```
var logger = new winston.Logger({
    transports: [new winston.transports.Console({
    timestamp: function() {
    var d = new Date();
    return d.toISOString()
},
})]
})
```

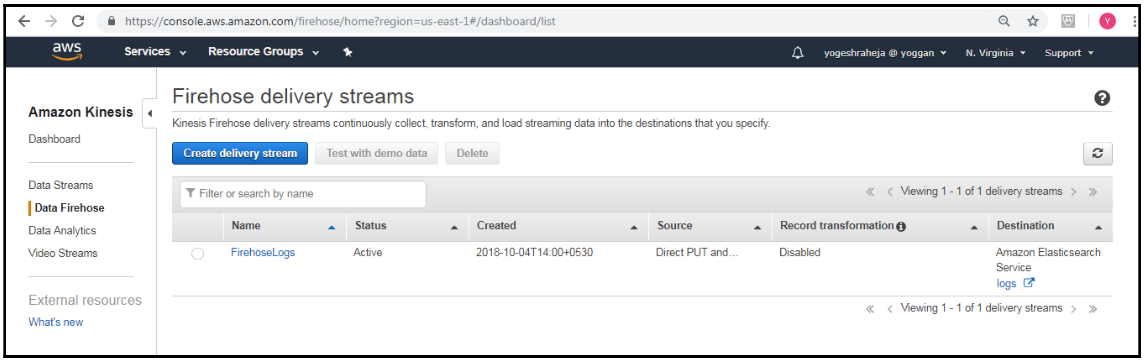
Replace this code with the reference to our Firehose endpoint:

```
var logger = new (winston.Logger)({
    transports: [new WFirehose({
    'streamName': 'FirehoseLogs',
    'firehoseOptions': {
        'region': 'us-east-1'
}
})]
})
```

Once those changes are in place, we can add the new module to `git`, `commit`, and `push` the changes:

```
$ git add helloworld.js package.json node_modules
$ git commit -m "Sending logs to Firehose directly"
$ git push
```

Within a few minutes, CloudWatch logs should stop receiving logs, while your Firehose delivery service should start seeing traffic. You can open `https://console.aws.amazon.com/firehose/home?region=us-east-1#/details/Firehose Logs?edit=false` to verify that the last changes are working:



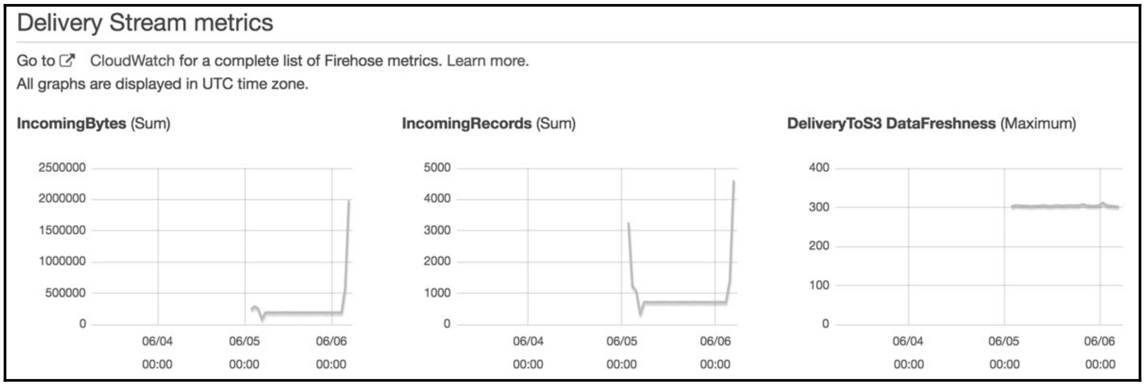
Firehose delivery streams

Kinesis Firehose delivery streams continuously collect, transform, and load streaming data into the destinations that you specify.

Viewing 1 - 1 of 1 delivery streams

| Name | Status | Created | Source | Record transformation | Destination |
|---|---|---|---|---|---|
| FirehoseLogs | Active | 2018-10-04T14:00+0530 | Direct PUT and... | Disabled | Amazon Elasticsearch Service logs |

Viewing 1 - 1 of 1 delivery streams

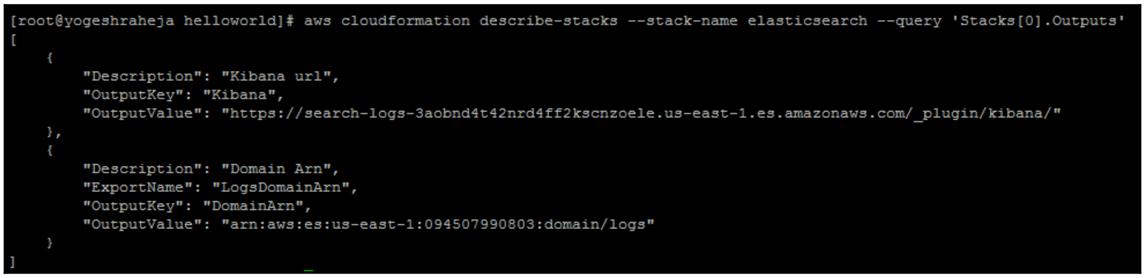


We can now look at our logs in Kibana.

## Using Kibana to visualize logs

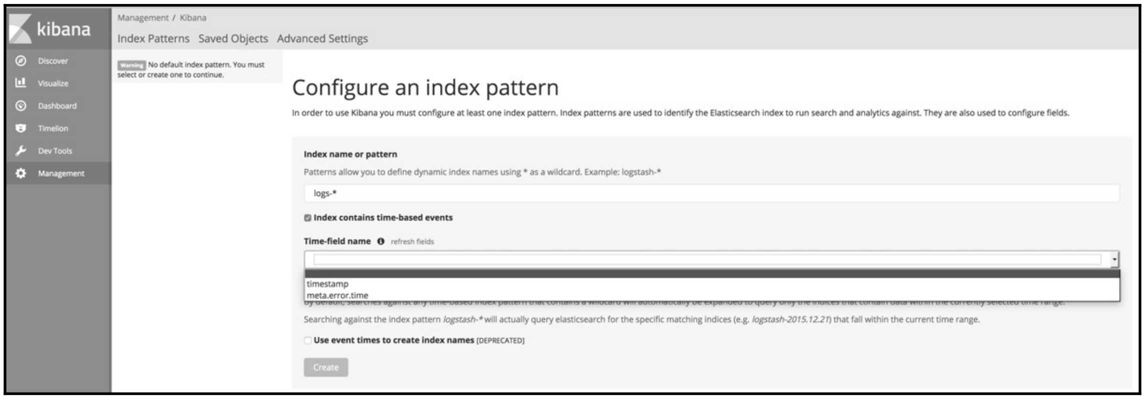
At this point, our application logs are going to ElasticSearch via Kinesis Firehose. One of the best ways to access our logs now is to use Kibana. You can find the URL of your Kibana instance by looking at the output of the ElasticSearch CloudFormation stack we launched earlier:

```
$ aws cloudformation describe-stacks \
    --stack-name elasticsearch \
    --query 'Stacks[0].Outputs'
```



```
[root@yogeshraheja helloworld]# aws cloudformation describe-stacks --stack-name elasticsearch --query 'Stacks[0].Outputs'
[
    {
        "Description": "Kibana url",
        "OutputKey": "Kibana",
        "OutputValue": "https://search-logs-3aobnd4t42nrd4ff2kscnzoele.us-east-1.es.amazonaws.com/_plugin/kibana/"
    },
    {
        "Description": "Domain Arn",
        "ExportName": "LogsDomainArn",
        "OutputKey": "DomainArn",
        "OutputValue": "arn:aws:es:us-east-1:094507990803:domain/logs"
    }
]
```

Using your browser, open the Kibana URL:



Warning No default index pattern. You must select or create one to continue.

Configure an index pattern

In order to use Kibana you must configure at least one index pattern. Index patterns are used to identify the Elasticsearch index to run search and analytics against. They are also used to configure fields.

**Index name or pattern**

Patterns allow you to define dynamic index names using * as a wildcard. Example: logstash-*

logs-*

☑ **Index contains time-based events**

**Time-field name** refresh fields

timestamp
meta.error.time

Searching against the index pattern *logstash-** will actually query elasticsearch for the specific matching indices (e.g. *logstash-2015.12.21*) that fall within the current time range.

☐ **Use event times to create index names** [DEPRECATED]

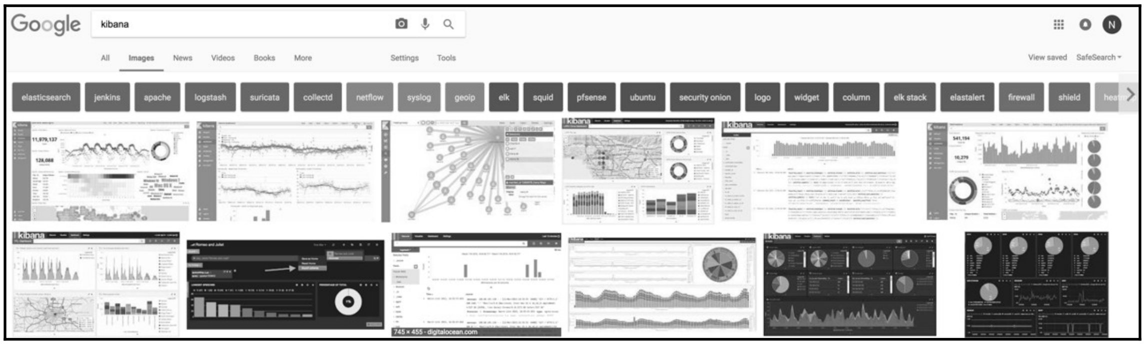
Create

This brings you to a screen resembling the one shown in the preceding screenshot where you will have to do the initial configuration:

- Set the value of the **Index name or pattern** field to `logs-*`.
- Doing so will make Kibana analyze your logs to find out the possible time field names. We will select the one called **timestamp**.
- Once you have selected the proper values, click on **Create**.

This will lead you to the management screen of the index pattern we just created. As you will see, each meta has been analyzed. At that point, you can click on **Discover** to see all your logs and explore the different visualization options to create dashboards for your logs.

You can Google `kibana` to get some inspiration on what you can do and what to put in your dashboards:



Google
kibana
All Images News Videos Books More Settings Tools
View saved SafeSearch
elasticsearch jenkins apache logstash suricata collectd netflow syslog geoip elk squid pfsense ubuntu security onion logo widget column elk stack elastalert firewall shield

**Deleting old logs using Curator**

Unlike CloudWatch logs, the deletion of old logs isn't a built-in feature of this stack. To delete old logs, look at Elastic Curator at `http://bit.ly/2rFHzUT`. You can easily deploy it using a Lambda function that will run once a day, for example.

Over the course of the last few pages, we went over several concepts to improve logging. With the help of several AWS services, we were able to collect logs, events, and metrics from our application and send them to different services, including CloudWatch, S3, and ElasticSearch. Of course, logging doesn't stop here. The monitoring stack we explored can be reused to monitor virtually anything that would matter to you and your company.

Among the other functionalities worth exploring is how to monitor the rest of our AWS infrastructure.

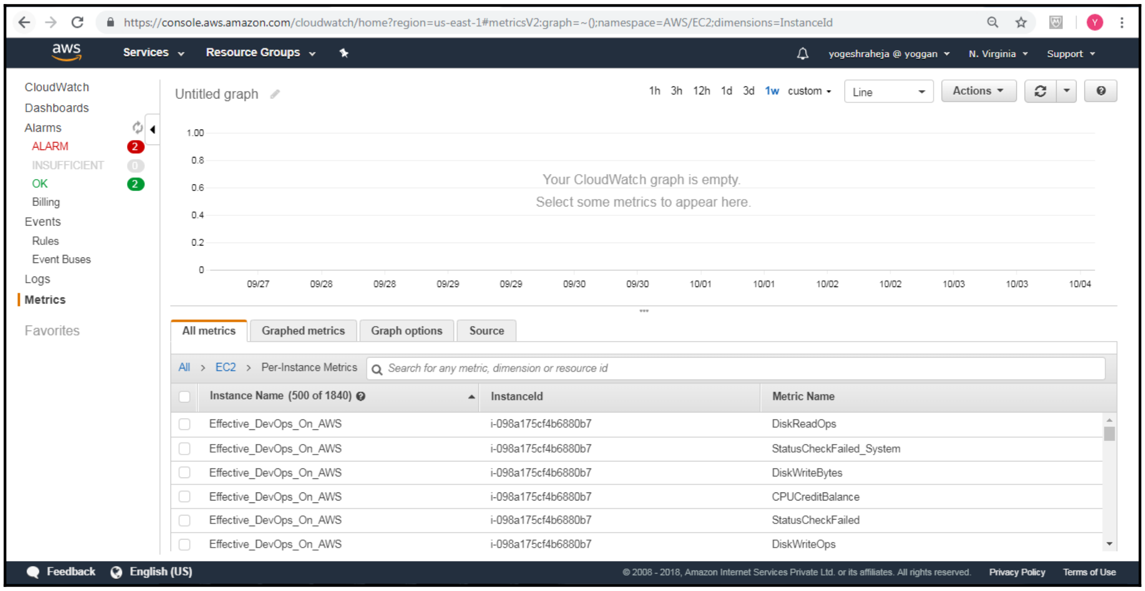
## Monitoring our infrastructure

Monitoring is one of those tasks with no finish line. There is always something that you could add or improve. Because of that, it is important to prioritize the areas on which you will focus most of your efforts, especially at the beginning, or when new services are released. In addition, different services require different levels of attention. For instance, services such as Lambda, S3, or DynamoDB are considered serverless. AWS takes care of almost everything. You don't need to handle failures, security patches, scaling, high availability, and so on. At the opposite end of the spectrum, you have services such as EC2 where, aside from the hardware itself, you control every aspect of the instance, and therefore, you need to also invest time and effort in monitoring. Lucky for us, most services that AWS creates have a native integration with CloudWatch. Rather than going down the full list of services we used so far and seeing how monitoring is done, we will look at the different ways to add monitoring to our existing templates.

## Monitoring EC2

EC2 instances are created on top of a hypervisor, which is controlled by AWS. Thanks to the hypervisor, AWS is able to extract several metrics and feed them into CloudWatch metrics. This includes CPU utilization, network utilization (NetworkIn and NetworkOut), and disk performance (DiskReadOps, DiskWriteOps, DiskReadBytes, DiskWriteBytes).

Those metrics are available in the EC2 console when you select your instances and also in the CloudWatch metrics menu:


https://console.aws.amazon.com/cloudwatch/home?region=us-east-1#metricsV2:graph=~();namespace=AWS/EC2;dimensions=InstanceId
Services
Resource Groups
yogeshraheja @ yoggan
N. Virginia
Support
CloudWatch
Dashboards
Alarms
ALARM 2
INSUFFICIENT 0
OK 2
Billing
Events
Rules
Event Buses
Logs
Metrics
Favorites
Untitled graph
1h 3h 12h 1d 3d 1w custom
Line
Actions
Your CloudWatch graph is empty.
Select some metrics to appear here.
All metrics
Graphed metrics
Graph options
Source
All > EC2 > Per-Instance Metrics
Search for any metric, dimension or resource id
Feedback
English (US)
© 2008 - 2018, Amazon Internet Services Private Ltd. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

| Instance Name (500 of 1840) | InstanceId | Metric Name |
|---|---|---|
| Effective_DevOps_On_AWS | i-098a175cf4b6880b7 | DiskReadOps |
| Effective_DevOps_On_AWS | i-098a175cf4b6880b7 | StatusCheckFailed_System |
| Effective_DevOps_On_AWS | i-098a175cf4b6880b7 | DiskWriteBytes |
| Effective_DevOps_On_AWS | i-098a175cf4b6880b7 | CPUCreditBalance |
| Effective_DevOps_On_AWS | i-098a175cf4b6880b7 | StatusCheckFailed |
| Effective_DevOps_On_AWS | i-098a175cf4b6880b7 | DiskWriteOps |

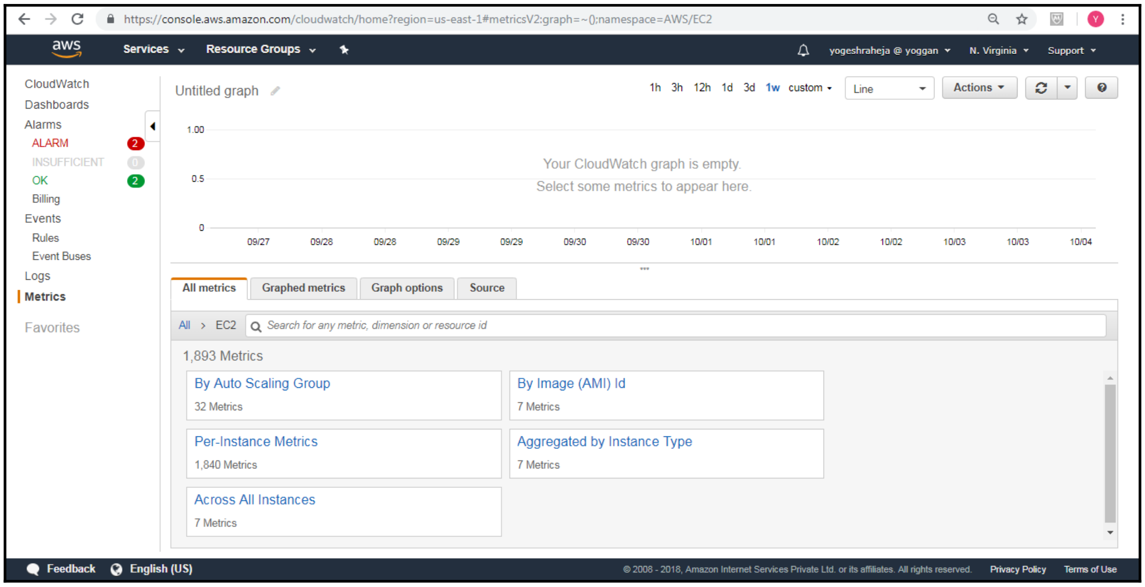
By default, the resolution of those metrics is 5 minutes. This means that new data points for your EC2 instances are collected every 5 minutes. If you have critical hosts where you wish to have a higher resolution, you can enable a feature called **detailed monitoring** on a per instance basis, which will increase the frequency of those metrics to a 1 minute resolution.

Unfortunately, only relying on the hypervisor metrics isn't enough. As you might have noticed, CloudWatch has information on how your disk performs, for example, but not on how much disk space you have left. That's because the hypervisor doesn't have visibility of the operating system. To supply that information, we will need to install an agent on our EC2 instances to collect those extra metrics.

## Providing custom metrics to CloudWatch

We are going to complement the metrics we already have with OS-specific information. This will include more detailed information about the CPU, disk usage, and memory.

When browsing the EC2 metrics, you will notice that you can query those metrics by Auto Scaling Group:



It is important to retain this ability to query a set of metrics by specific criteria, especially by Auto Scaling Groups. To provide the same capability to our custom metrics, we need to make a change in our CloudFormation template to allow our EC2 instances to get the Auto Scaling Group information.

## Updating our CloudFormation template

Open the `nodeserver-cf-template.py` script.

After the first IAM policy, add the following `MonitoringPolicy` IAM policy. Simply add the following action to the statement:

```
t.add_resource(IAMPolicy(
    "MonitoringPolicy",
    PolicyName="AllowSendingDataForMonitoring",
    PolicyDocument=Policy(
        Statement=[
            Statement(
                Effect=Allow,
                Action=[
                    Action("cloudwatch", "Put*"),
                    Action("logs", "Create*"),
                    Action("logs", "Put*"),
                    Action("logs", "Describe*"),
                    Action("events", "Put*"),
                    Action("firehose", "Put*"),
                    Action("autoscaling", "DescribeAutoScalingInstances"),
                ],
                Resource=["*"])
        ]
    ),
    Roles=[Ref("Role")]
))
```

Save the file. It should be similar to `https://raw.githubusercontent.com/yogeshraheja/Effective-DevOps-with-AWS/master/Chapter08/EffectiveDevOpsTemplates/nodeserver-cf-template-part-3.py`.

Then, once again, update the stacks:

```
$ git add nodeserver-cf-template.py
$ git commit -m "Allowing the instance to describe the ASG instances"
$ git push
$ python nodeserver-cf-template.py > nodeserver-cf.template
$ aws cloudformation update-stack \
    --capabilities CAPABILITY_IAM \
    --stack-name helloworld-staging \
    --template-body file://nodeserver-cf.template \
    --parameters \
      ParameterKey=InstanceType,UsePreviousValue=true \
      ParameterKey=KeyPair,UsePreviousValue=true \
      ParameterKey=PublicSubnet,UsePreviousValue=true \
      ParameterKey=ScaleCapacity,UsePreviousValue=true \
      ParameterKey=VpcId,UsePreviousValue=true

$ aws cloudformation update-stack \
    --capabilities CAPABILITY_IAM \
    --stack-name helloworld-production \
    --template-body file://nodeserver-cf.template \
    --parameters \
       ParameterKey=InstanceType,UsePreviousValue=true \
       ParameterKey=KeyPair,UsePreviousValue=true \
       ParameterKey=PublicSubnet,UsePreviousValue=true \
       ParameterKey=ScaleCapacity,UsePreviousValue=true \
           ParameterKey=VpcId,UsePreviousValue=true
```

Next, we are going to create a new role in our Ansible repository to install and configure a tool to emit those metrics.

## Creating a CloudWatch role in Ansible

We first need to go into the roles directory of our Ansible repository:

```
$ cd roles
```

We will use the `ansible-galaxy` command to generate our new role:

```
$ ansible-galaxy init cloudwatch
- cloudwatch was created successfully
```

We will create a minimal role that allows us to report some of those missing stats. With your text editor, open the file `cloudwatch/tasks/main.yml`.

We will use an open source tool called **cloudwatchmon**. You can access its code source and documentation on the GitHub page of the project at `http://bit.ly/2pYjhI9`. The tool is written in Python and is available through `pip`. To install PIP packages, Ansible provides a `pip` module. After the initial comment of the task, add the following:

```
---
# tasks file for cloudwatch
- name: Installing cloudwatchmon
  pip:
    name: cloudwatchmon
```

This tool works through the intermediary of cronjob. We will use the `cron` module of Ansible to create what we will call `cloudwatchmon`.

After the call to the `pip` module, call the `cron` module as follows:

```
- name: Execute cloudwatchmon every 5min
  cron:
    name: "cloudwatchmon"
    minute: "*/5"
    job: /usr/local/bin/mon-put-instance-stats.py --auto-scaling --loadavg-
percpu --
    mem-util --disk-space-util --disk-path=/ --from-cron"
```

In this case, we are configuring our job to trigger the `mon-put-instance-stats.py` every 5 minutes. We are also specifying the list of metrics we want to collect in the command. The `mem-util` option will provide the percentage of memory utilization while `disk-space-util` will do the same, but referring to the disk space on the `/` partition. You can refer to the documentation of the script to check the full list of options available.

**Percentage versus raw values**

There are two ways to report these resource usages. You can provide the utilization percentages (for example, the partition is full at 23%) or look at the exact value (for example, there are 2 GB free on that partition). For our purposes, suffice it to say that monitoring infrastructures using percentages tends to speed up iteration time as you can create more generic alerts. This tends to change over time, as your different applications will often have different constraints requiring different types of hardware.

Before committing our change, we are going in go one directory up and edit the file `nodeserver.yml`:

```
$ cd ..
```

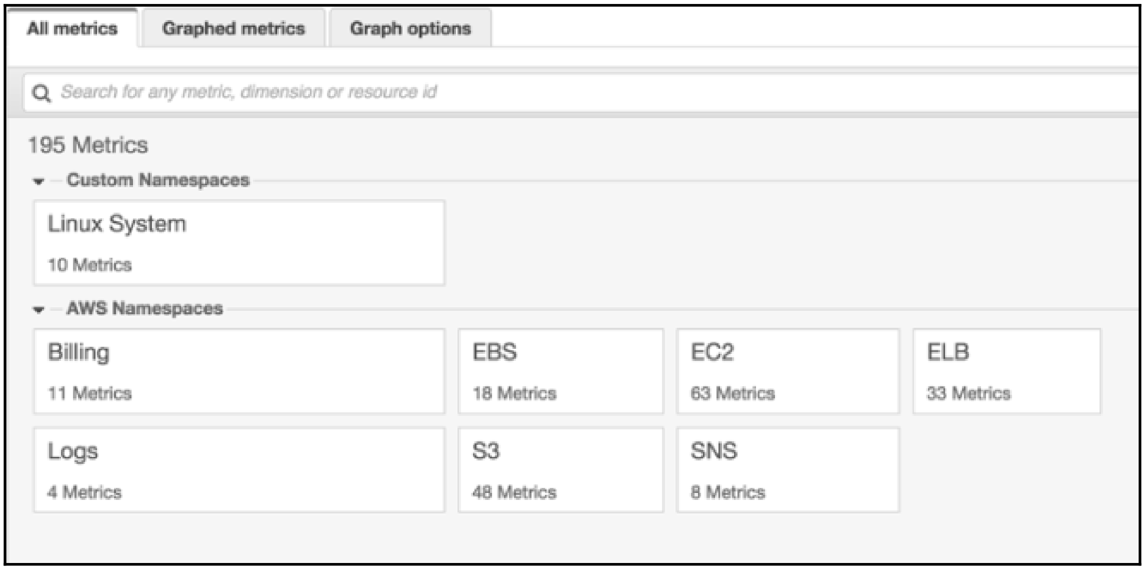
We need to include the new role we just created to our service. We can do that simply by adding a new entry to the `roles` section, as shown here:

```
---
- hosts: "{{ target | default('localhost') }}"
  become: yes
  roles:
    - nodejs
    - codedeploy
    - cloudwatch
    - { role: awslogs, name: messages, file: /var/log/messages }
    - {
        role: awslogs, name: helloworld,
        file: /var/log/helloworld/helloworld.log,
        datetime_format: "%Y-%m-%dT%H:%M:%S.%f"
      }
```

We can save all the changes and commit them:

```
$ git add roles/cloudwatch nodeserver.yml
$ git commit -m "Adding new role for CloudWatch monitoring"
$ git push
```

Since Ansible pulls changes every 10 minutes, within 15 minutes at most we should start seeing a new section in CloudWatch called **Linux System**, containing the new metrics of our hosts:



195 Metrics

Custom Namespaces

Linux System
10 Metrics

AWS Namespaces

Billing
11 Metrics

EBS
18 Metrics

EC2
63 Metrics

ELB
33 Metrics

Logs
4 Metrics

S3
48 Metrics

SNS
8 Metrics

Now that we have all the visibility we need on our EC2 instances, we can put some alarms in place.

In many cases, especially with applications exposed to the internet, you tend to observe occasional strange behavior, but can't easily understand how the application gets in that state. One of the most useful pieces of information to have in those cases is the access logs of your load balancer.

Currently, our load balancer exposes several metrics in CloudWatch, but can't tell us the full story. What routes are causing a 5xx error? What is the latency? Where are the users coming from? How aggressively are they using your application? To gain access to those insights, we make a few changes to our ELB and ALB instances.

## Monitoring ECS clusters

When it comes to monitoring, EC2 and ECS are very similar. We can slice ECS into three components: the ECS hosts, the ECS service, and the containers.

## Monitoring ECS hosts

ECS runs on top of EC2. Therefore, everything that can be done for EC2 can and should be done for ECS, including disk space monitoring. The main difference is that you have more options on how to implement it:

1. Your first option consists of duplicating what we did with EC2 and have the `cloudwatchmon` run on every EC2 instance running ECS. To implement this, you can create a new Ansible role for the ECS host and add the installation and execution of Ansible to the `UserData` variable of the `ecs-cluster-cf-template.py`, the way we did it in `nodeserver-cf-template.py`.
2. Your second option also relies on the `UserData` field, but this time, you create a new container that runs `cloudwatchmon` and creates a task for it. The `UserData` file would end with something like:

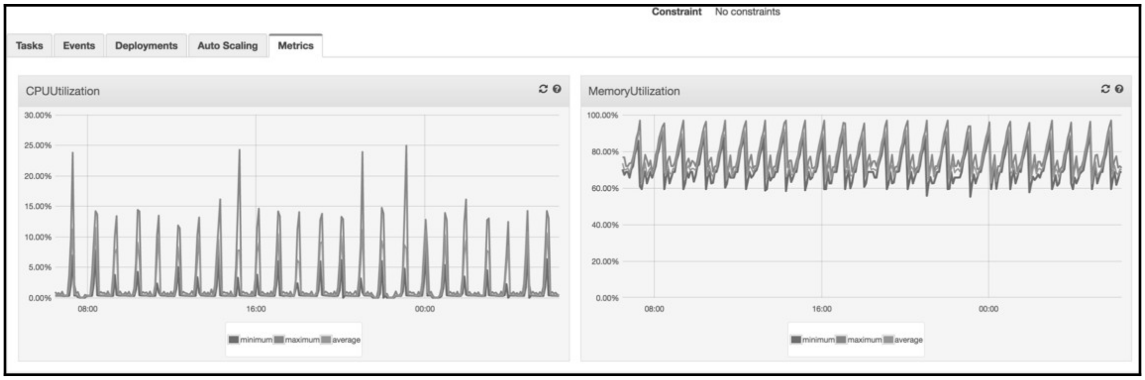
```
$ aws ecs start-task \
    --cluster $cluster \
    --task-definition cloudwatchmon:1 \
    --container-instances $instance_arn \
    --region $region
```

3. Your last option is to take advantage of the task placement feature that ECS offers. This will let you run your containers using a spread strategy such that, by launching as many containers as you have ECS hosts, you will be able to collect the stats of each ECS server. You can read more about task placement at `http://amzn.to/2kn2OXO`.

Once you have that part under control, you can look at the ECS service itself.

## Monitoring the ECS service

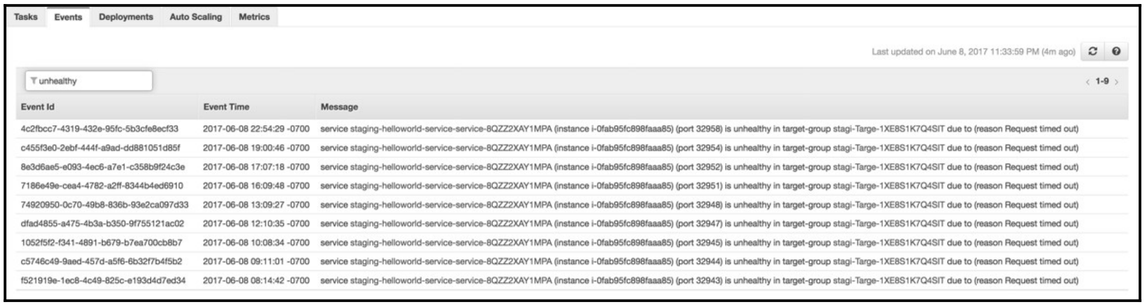
Often with managed services, AWS natively provides the metrics you need to care about. If you go into the console and look at the monitoring tabs, you will see two graphs. The first graph focuses on CPU allocation and the second one on memory utilization. Because we configured Auto Scaling Groups, the usage shown in these graphs should always stay within the thresholds we set:



Lastly, you can monitor your tasks and containers.

## Monitoring your containers

We containerized our applications, but, fundamentally, nothing about the application really changed. The best way to monitor your application is through the creation of metrics and logs, as we did in the earlier part of this chapter. That said, if you experience some issues with your containers, you may want to search for unhealthy events in the ECS clusters menu:



| Event Id | Event Time | Message |
|---|---|---|
| 4c2fbcc7-4319-432e-95fc-5b3cfe8ecf33 | 2017-06-08 22:54:29 -0700 | service staging-helloworld-service-service-8QZZ2XAY1MPA (instance i-0fab95fc898faaa85) (port 32958) is unhealthy in target-group stagi-Targe-1XE8S1K7Q4SIT due to (reason Request timed out) |
| c455f3e0-2ebf-444f-a9ad-dd881051d85f | 2017-06-08 19:00:46 -0700 | service staging-helloworld-service-service-8QZZ2XAY1MPA (instance i-0fab95fc898faaa85) (port 32954) is unhealthy in target-group stagi-Targe-1XE8S1K7Q4SIT due to (reason Request timed out) |
| 8e3d6ae5-e093-4ec6-a7e1-c358b9f24c3e | 2017-06-08 17:07:18 -0700 | service staging-helloworld-service-service-8QZZ2XAY1MPA (instance i-0fab95fc898faaa85) (port 32952) is unhealthy in target-group stagi-Targe-1XE8S1K7Q4SIT due to (reason Request timed out) |
| 7186e49e-cea4-4782-a2ff-8344b4ed6910 | 2017-06-08 16:09:48 -0700 | service staging-helloworld-service-service-8QZZ2XAY1MPA (instance i-0fab95fc898faaa85) (port 32951) is unhealthy in target-group stagi-Targe-1XE8S1K7Q4SIT due to (reason Request timed out) |
| 74920950-0c70-49b8-836b-93e2ca097d33 | 2017-06-08 13:09:27 -0700 | service staging-helloworld-service-service-8QZZ2XAY1MPA (instance i-0fab95fc898faaa85) (port 32948) is unhealthy in target-group stagi-Targe-1XE8S1K7Q4SIT due to (reason Request timed out) |
| dfad4855-a475-4b3a-b350-9f755121ac02 | 2017-06-08 12:10:35 -0700 | service staging-helloworld-service-service-8QZZ2XAY1MPA (instance i-0fab95fc898faaa85) (port 32947) is unhealthy in target-group stagi-Targe-1XE8S1K7Q4SIT due to (reason Request timed out) |
| 1052f5f2-f341-4891-b679-b7ea700cb8b7 | 2017-06-08 10:08:34 -0700 | service staging-helloworld-service-service-8QZZ2XAY1MPA (instance i-0fab95fc898faaa85) (port 32945) is unhealthy in target-group stagi-Targe-1XE8S1K7Q4SIT due to (reason Request timed out) |
| c5746c49-9aed-457d-a5f6-6b32f7b4f5b2 | 2017-06-08 09:11:01 -0700 | service staging-helloworld-service-service-8QZZ2XAY1MPA (instance i-0fab95fc898faaa85) (port 32944) is unhealthy in target-group stagi-Targe-1XE8S1K7Q4SIT due to (reason Request timed out) |
| f521919e-1ec8-4c49-825c-e193d4d7ed34 | 2017-06-08 08:14:42 -0700 | service staging-helloworld-service-service-8QZZ2XAY1MPA (instance i-0fab95fc898faaa85) (port 32943) is unhealthy in target-group stagi-Targe-1XE8S1K7Q4SIT due to (reason Request timed out) |

Those issues are often caused by misconfiguration of the tasks, when you don't provide enough CPU and memory reservation, or by bugs or bad user behavior. For those issues, it is often hard to diagnose them by solely looking at metrics. To help with the diagnosis of these issues, we can turn on logging on our load balancers.

## Monitoring ALB and ELB instances

ALB and ELB both provide a fair amount of top-level metrics, giving you a sense of how your services are behaving, but sometimes, metrics aren't enough. You want to produce an access log and track the detail of each request hitting your services. Both ELB and ALB provide the ability to generate an access log and store it on S3. We will illustrate how to turn on this feature by making changes to our CloudFormation templates. We will take the example of our ALB template to turn on logging.

With your editor, open the file `helloworld-ecs-alb-cf-template.py` located in the `EffectiveDevOpsTemplates` repository.

In order to create the access log, we will need to create an S3 bucket and provide it a special policy so that AWS can access our bucket. This will require including a few extra classes. In the import section, add the following:

```
from awacs.aws import Allow, Policy, Principal, Statement

from awacs.s3 import PutObject, ARN

from troposphere.s3 import (
    Bucket,
    BucketPolicy,)
```

Next, we will create our S3 bucket. After the creation of the template variable and the addition of its description, add the following resource:

```
t.add_resource(Bucket(
    "S3Bucket",
    DeletionPolicy="Retain",
))
```

We are setting a deletion policy such that if we delete the CloudFormation template, the S3 bucket will remain and the logs will still be available. The next resource we are going to create is the special policy for that bucket. The policy will start by referencing the bucket we just created:

```
t.add_resource(BucketPolicy(
    'BucketPolicy',
    Bucket=Ref("S3Bucket"),
```

The next part is the creation of the policy. The policy contains a statement that tells the bucket that the AWS account `127311923021` is allowed to put object operations into the `/AWSLogs/511912822958/` prefix.

The account `127311923021` is a special account that AWS operates. You can refer to `http://amzn.to/2r8AqPI` for the list of account IDs in case your bucket isn't in `us-east-1`. In addition `511912822958` needs to be replaced with your own AWS account ID:

```
        PolicyDocument=Policy(
             Version='2012-10-17',
             Statement=[
                 Statement(
                     Action=[PutObject],
                     Effect=Allow,
                     Principal=Principal("AWS", ["127311923021"]),
                     Resource=[Join('',
                                    [ARN(''),
                                     Ref("S3Bucket"),
                                        "/AWSLogs/511912822958/*"])],
                 )
             ]
         )
     ))
```

Now that the bucket is created and contains the specific policy, we can turn on the access log in our ALB resource as follows:

```
     t.add_resource(elb.LoadBalancer(
         "LoadBalancer",
         Scheme="internet-facing",
         Subnets=Split(
             ',',
             ImportValue(
                 Join("-",
                      [Select(0, Split("-", Ref("AWS::StackName"))),
                       "cluster-public-subnets"]
                      )
             )
         ),
         SecurityGroups=[Ref("LoadBalancerSecurityGroup")],
         LoadBalancerAttributes=[
             elb.LoadBalancerAttributes(
                 Key="access_logs.s3.enabled",
                 Value="true",
             ),
             elb.LoadBalancerAttributes(
                 Key="access_logs.s3.bucket",
                 Value=Ref("S3Bucket"),
             )
         ],
     ))
```

Once those changes are in place, you can save and commit your changes, generate the new template, and update your stack. The code should be similar to `https://raw.githubusercontent.com/yogeshraheja/Effective-DevOps-with-AWS/master/Chapter08/EffectiveDevOpsTemplates/helloworld-ecs-alb-cf-template-part-1.py`.

```
$ git add helloworld-ecs-alb-cf-template.py
$ git commit -m "Sending ALB logs to S3"
$ git push
$ python helloworld-ecs-alb-cf-template.py > helloworld-ecs-alb-cf.template
$ aws cloudformation update-stack \
    --stack-name staging-alb \
    --template-body file://helloworld-ecs-alb-cf.template
$ aws cloudformation update-stack \
    --stack-name production-alb \
    --template-body file://helloworld-ecs-alb-cf.template
```

Logs will now be automatically uploaded every 5 minutes to the S3 bucket.

**Using AWS Athena to efficiently retrieve logs**

Once your logs are in S3, you can either download them and analyze them locally or, if you are looking for specific information, you can use AWS Athena (`http://amzn.to/2rSsrn7`) to run SQL queries against your logs. For example, to get the list of the most active IP addresses, you can run the following query:

```
SELECT client_ip, COUNT(*) as count
FROM logs.alb
GROUP BY client_ip
ORDER BY COUNT(*) DESC LIMIT 100;
```

As you would expect, each service AWS releases comes with documentation that covers every aspect of its monitoring. You can refer to it to see what you need to expose and implement it with code using one of the strategies we showed in this section.

The last part of adding a monitoring solution is to create alarms to automatically notify engineers when something abnormal is happening. We will use CloudWatch in conjunction with SNS to create those alarms.

## Creating alarms using CloudWatch and SNS

Up to this point, we have focused on exposing metrics to better understand what is happening around us. We can now access the data and create nice visualizations of it, but that is not enough. **Meantime to discover** (**MTD**) and **Mean time to recover** (**MTTR**) are two very common metrics used to see how the operations team, and by extension the DevOps team, is performing. To keep those two metrics as low as possible, automated alerts are essential. A good alerting system will often help to rapidly identify issues in your systems and help minimize service degradation and disruption. That said, creating the proper alarms isn't always as easy as it sounds.

What should we be alerted about? Measuring everything doesn't mean being alerted about everything. As a rule of thumb, aim at creating alerts about symptoms rather than causes, and be mindful of when to page someone versus sending a less distributive email or message (like slack) notification. You want to avoid alert fatigue as much as possible. This is when on-call engineers become numb to certain alerts that occur too often. In addition, you want to avoid flooding the on-call engineer with a sea of noisy alerts.

Alerts, and in particular the ones that create pages, should always be timely and actionable:

Think about limiting the scope of what your alerts are covering to important resources, such as your production environment, only. Make sure that planned maintenance is also factored into your alerting policy. We won't show that in this book, but you might extend the work done in the AWS health section of this chapter to disable the alarms of the services impacted by some of the planned maintenance around EC2.

As your infrastructure grows and the number of EC2 instances needed to run a service increases, you may want to avoid sending a page of information if only a small portion of your infrastructure is having issues. For instance, the architectures we used in this book put our EC2 instances behind load balancers. If one of your instances stops working, the user impact will be minimal and paging someone is likely not required.

To create our alerts, we will once again turn to CloudWatch. In addition to its capacity to log metrics, create logs, and trigger events, CloudWatch also features many functionalities to watch metrics. We already used some of its features in `Chapter 6`, *Running Containers in AWS*, when we configured the scaling component of our Auto Scaling Groups in EC2 and ECS. We will use it here in conjunction with SNS.

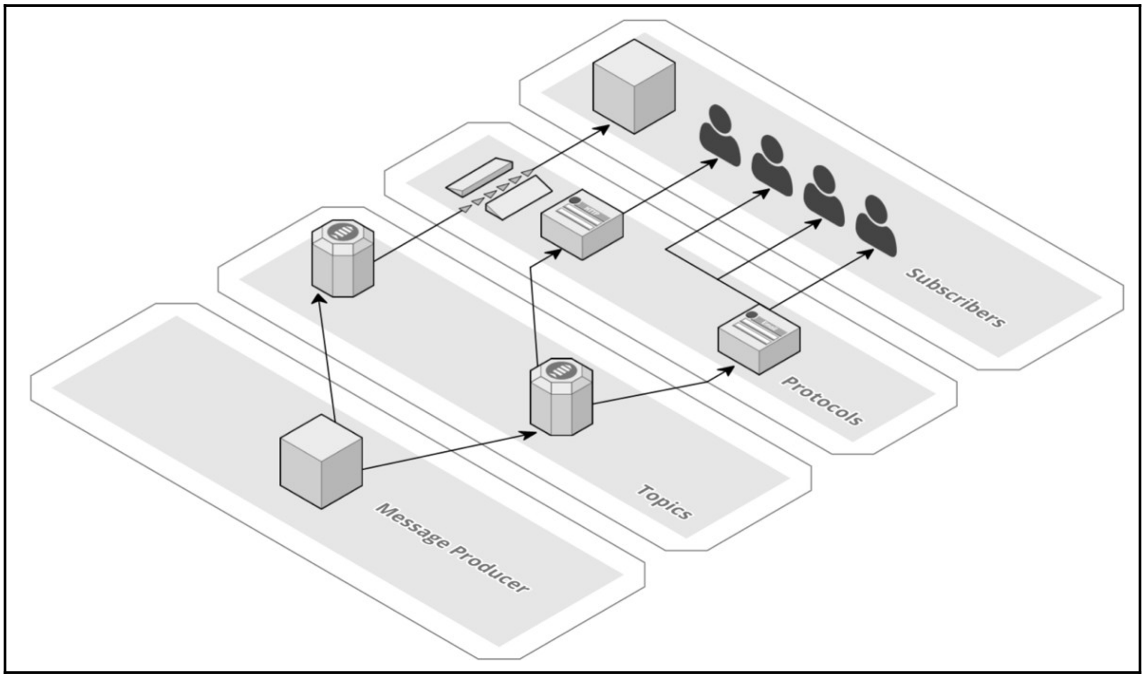
## AWS Simple Notification Service (SNS)

SNS is a web service that enables applications, end users, and devices to send and receive notifications. We already briefly used it in our deployment pipeline to receive email notifications to approve production deployment. We will use SNS to notify us about important events.

The service is logically broken up into four parts: producers, topics, protocols, and subscriptions.

The message producers are the applications and services producing the messages, and those messages are organized around topics. Topics are like access points. As you create your SNS topics, AWS will associate them with an ARN that other services can subscribe to.

Once the topic is created, other systems or end users can subscribe (or be subscribed) to those topics via a number protocols, such as HTTP(S), email, Amazon SQS, Amazon Lambda, SMS, or the mobile endpoint (mobile application and devices):



We will type of notifications, one for non-urgent issues. In this case, we will simply send emails. The second one will be used to notify us of critical and time-sensitive issues. For the latter, we will rely on SMS notifications.

We will use the AWS command-line interface to create them. The first step will be to create the two topics as follows:

```
$ aws sns create-topic --name alert-email
{
    "TopicArn": "arn:aws:sns:us-east-1:511912822958:alert-email"
}
$ aws sns create-topic --name alert-sms
```

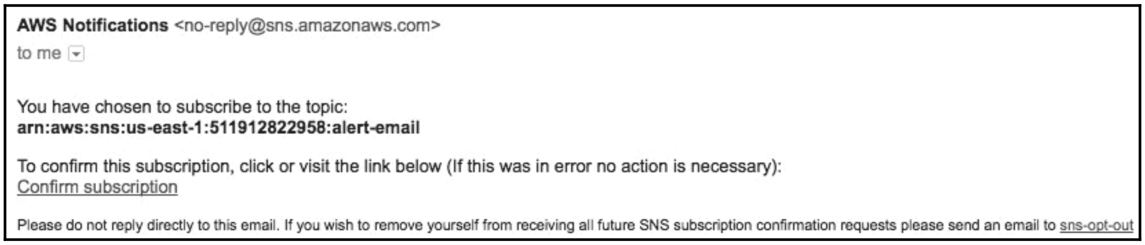
```
{
    "TopicArn": "arn:aws:sns:us-east-1:511912822958:alert-sms"
}
```

This part is straightforward: the only difference is the same for those topics. We will now put our protocols in place, starting with the email protocol. To do that, we will use the `sns subscribe` option and specify the following:

- The `topic-arn` returned by the previous command
- The protocol, in this case email
- The `notification-endpoint`, which should be your email address:

```
$ aws sns subscribe \
    --topic-arn arn:aws:sns:us-east-1:511912822958:alert-email \
    --protocol email \
    --notification-endpoint email@domain.com
{
    "SubscriptionArn": "pending confirmation"
}
```

At this point, you should check your inbox and confirm that you want to subscribe:



**AWS Notifications** <no-reply@sns.amazonaws.com>
to me

You have chosen to subscribe to the topic:
**arn:aws:sns:us-east-1:511912822958:alert-email**

To confirm this subscription, click or visit the link below (If this was in error no action is necessary):
Confirm subscription

Please do not reply directly to this email. If you wish to remove yourself from receiving all future SNS subscription confirmation requests please send an email to sns-opt-out

The SMS subscription is very similar to the email subscription, with the addition of an extra step of setting a `DisplayName` attribute, which is required for the SMS protocol subscription:

```
$ aws sns set-topic-attributes \
    --topic-arn arn:aws:sns:us-east-1:511912822958:alert-sms2 \
    --attribute-name DisplayName \
    --attribute-value helloworld
```

The notification endpoint should be your cell phone number prefixed with the country code (for example, in the US, if your number is +1 (222) 333-4444, you will need to use `12223334444`):

```
$ aws sns subscribe \
    --topic-arn arn:aws:sns:us-east-1:511912822958:alert-sms2 \
    --protocol sms \
    --notification-endpoint 12223334444
```

At that point, the first time around, you will receive an SMS to confirm your subscription.

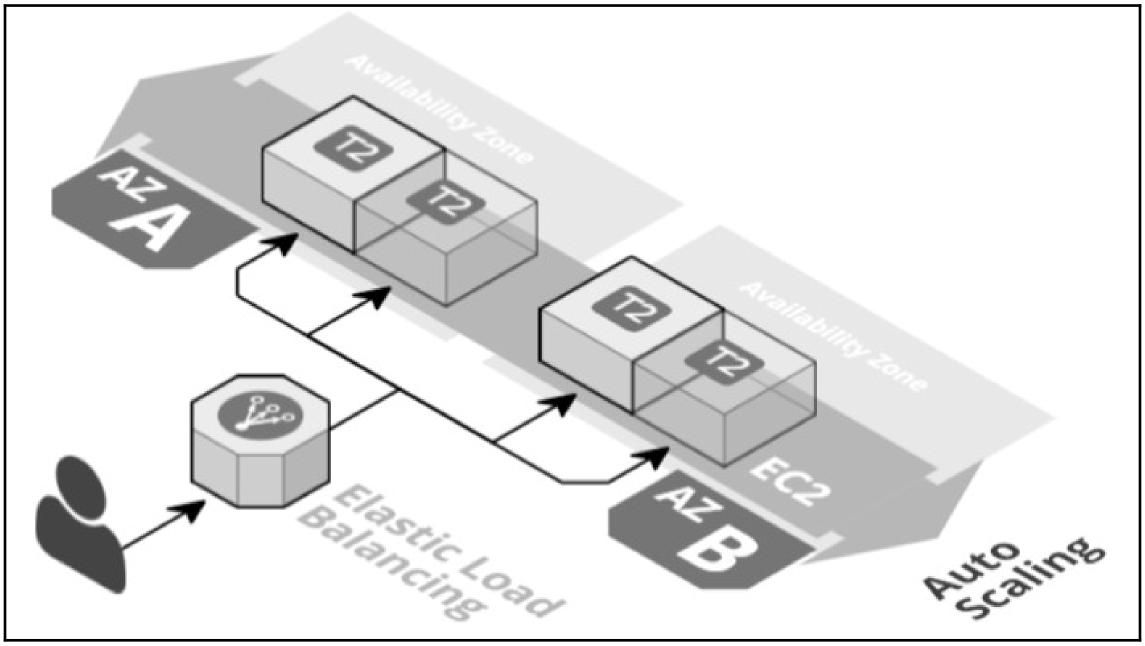
**Integrating with PagerDuty, Opsgenie, or Victorops**

We are showing in this book a minimum viable solution to send SMS notifications for important issues using only SNS. If you want to use a more feature-rich solution, such as PagerDuty, Opsgenie, or VictoOps, to name a few, you will simply need to change the type of subscription in the last command. Instead of using SMS, you will use the HTTPS protocol and provide the WebHook URL of your service provider.

Now that our notification system is in place, we can start feeding notifications to our topics. For that, we will turn to AWS CloudWatch.

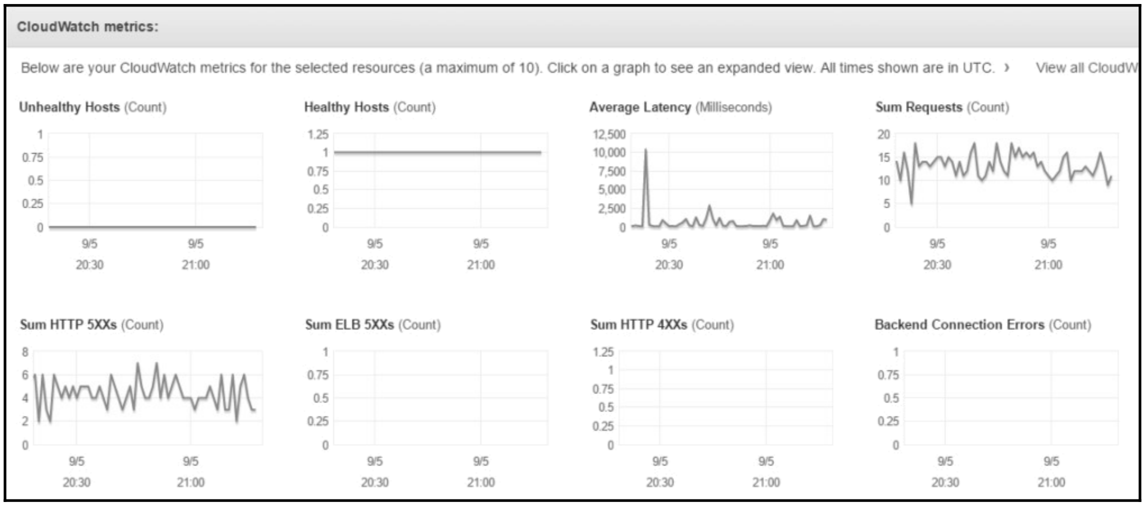
## Creating an alert of an elevated error rate in our application

As a reminder, our `helloworld` application has a very simple design:



All user traffic is going through an ELB instance. Since we use the HTTP to communicate, we can easily identify when something unexpected is happening.

If you take a closer look at the monitoring tab of one of your load balancer instances (`http://amzn.to/2rsEaLY`), you will see some of the top-level metrics you should care about:


CloudWatch metrics:
Below are your CloudWatch metrics for the selected resources (a maximum of 10). Click on a graph to see an expanded view. All times shown are in UTC.
View all CloudW
Unhealthy Hosts (Count)
Healthy Hosts (Count)
Average Latency (Milliseconds)
Sum Requests (Count)
Sum HTTP 5XXs (Count)
Sum ELB 5XXs (Count)
Sum HTTP 4XXs (Count)
Backend Connection Errors (Count)

While each application has its own behavior, an increase in latency or HTTP **5XXs** is usually a good signal that someone needs to take a closer look at the service.

We will incorporate the monitoring of those two metrics in our template. We will reopen our troposphere script `helloworld-ecs-alb-cf-template.py`. We will first add some new import as such:

```
from troposphere.cloudwatch import (
    Alarm,
    MetricDimension,
)
```

Then, we will go to the bottom of the file, where we already created the alarms CPU too low

and CPU too high.

Just before the last print statement, we will add a new `Alarm` resource as follows:

```
t.add_resource(Alarm(
    "ELBHTTP5xxs",
        AlarmDescription="Alarm if HTTP 5xxs too high",
```

We are giving it a reference and a description. To target the proper metric, we need to specify the namespace of the ELB service and the name of the metric, as shown here:

```
    Namespace="AWS/ELB", MetricName="HTTPCode_Backend_5XX",
```

We want our alert to target the load balancer instance created with that template. For that, we'll reference the load balancer resource in the metric dimension as follows:

```
     Dimensions=[
            MetricDimension(
                Name="LoadBalancerName",
                Value=Ref("LoadBalancer")
            ),
        ],
```

We want the alert to trigger if the number of HTTP 5xx is, on average, greater than 30 for three consecutive periods of 1 minute. This is done using the following properties:

```
     Statistic="Average",
        Period="60",
        EvaluationPeriods="3",
        Threshold="30",
        ComparisonOperator="GreaterThanOrEqualToThreshold",
```

The last part of the alert consists of selecting the action to perform when the alert triggers. In our case, we will want to send a message with the `alert-sms` SNS topic. In order to do that, we need to get this topic's ARN. We can do it using the following command:

```
    $ aws sns list-topics
```

Once you have the information, you can specify the information in `AlarmActions` and `OKActions`. Additionally, we will leave the `InsufficientDataActions` action empty as this metric is what we call a sparse metric, meaning that if no 5xxs are emitted, the service will not produce any data points, as opposed to creating an entry with a value of `0`. The `OKActions` is also somewhat optional and is more a question of taste. Configured as such, CloudWatch will emit another SMS when the alert resolves:

```
    AlarmActions=["arn:aws:sns:us-east-1:511912822958:alert-sms"],
        OKActions=["arn:aws:sns:us-east-1:511912822958:alert-sms"],
        InsufficientDataActions=[],
```

This concludes the creation of that alarm. We can close our open parenthesis:

```
    ))
```

After that new alarm, we will create the alarm to target the latency. Almost everything is identical. We are going to create a new resource and give its identifier and description as follows:

```
t.add_resource(Alarm(
    "ELBHLatency",
    AlarmDescription="Alarm if Latency too high",
```

We will use the same namespace, but with a different metric name:

```
Namespace="AWS/ELB",
MetricName="Latency",
```

The dimensions are the same as before:

```
Dimensions=[
        MetricDimension(
            Name="LoadBalancerName",
            Value=Ref("LoadBalancer")
        ),
    ],
```

For the case of latency, we are looking at five evaluations of 1 minute and a threshold of 500 ms to trigger the alarm:

```
 Statistic="Average",
    Period="60",
    EvaluationPeriods="5",
    Threshold="0.5",
    ComparisonOperator="GreaterThanOrEqualToThreshold",
    AlarmActions=["arn:aws:sns:us-east-1:511912822958:alert-sms"],
    OKActions=["arn:aws:sns:us-east-1:511912822958:alert-sms"],
    InsufficientDataActions=[],
```

This concludes the creation of our alarms. Your new template should look as follows `https://raw.githubusercontent.com/yogeshraheja/Effective-DevOps-with-AWS/master/Chapter08/EffectiveDevOpsTemplates/helloworld-ecs-alb-cf-template.py`

You can commit the changes, generate the new CloudFormation template, and deploy it using the usual steps.

```
$ git add helloworld-ecs-alb-cf-template.py
$ git commit -m "Creating SNS alarms"
$ git push
$ python helloworld-ecs-alb-cf-template.py > helloworld-ecs-alb-cf.template
$ aws cloudformation update-stack \
    --stack-name staging-alb \
    --template-body file://helloworld-ecs-alb-cf.template
```

```
$ aws cloudformation update-stack \
    --stack-name production-alb \
    --template-body file://helloworld-ecs-alb-cf.template
```

**Blameless post-mortems**

To close our feedback loop, we need to talk about learning. When failures happen, one of the best approaches to building that learning component is to create post-mortem documents that describe the incident, the timeline, the root cause, and how it was resolved. John Allspaw, one of the founding fathers of the DevOps movement, did some extensive thinking in that area and created the concept of blameless post-mortems, which describe in more detail this approach that emphasizes learning over finger-pointing.

One of the restrictions that CloudWatch has is the notion of alarm dimensions. In our last example, the ELB represents only one resource, which made it easy to create our alert as we could reference the resource name. For more dynamic resources, such as our EC2 instances, we might want to monitor a resource that's not exposed at the load balancer level.

To accomplish such things, we need to look at CloudWatch events.

## Using CloudWatch events and Lambda to create alerts on custom metrics

In the previous section, we added two alarms to our CloudFormation template. Whenever possible, keeping your monitoring information with the resources they are monitoring is good practice. Unfortunately, it isn't always easy to do. For instance, we are keeping track of the disk space usage of our EC2 instances. Those EC2 instances are created by our Auto Scaling Group. Because of that, adding alerts for that metric in our troposphere code is a lot more complicated as we don't have some of the critical information, such as the instance ID. To get around that issue, we are going to see how to create alerts based on infrastructure changes.

As we saw earlier, whenever a change occurs in your AWS infrastructure, the event is emitted in real time to a CloudWatch event. This includes the creation of EC2 instances. We will create a rule to capture those events and send that information to a Lambda function that will create our alarms.

We will implement that using the serverless framework (`https://serverless.com/`) that we looked at in `Chapter 6`, *Scaling Your Infrastructure*.

We will first create a new serverless application. In `Chapter 6`, *Scaling Your Infrastructure*, we demonstrated how to create a `helloworld` application using Node.js. Lambda and serverless are also both able to handle other languages, including Python. We will use Python and the Boto library to manage the creation of our alarms. To get started, we need to create a new application using the following command:

```
serverless create --template aws-python \
    --name disk-free-monitoring \
    --path disk-free-monitoring
```

This will create all the boilerplate we need inside a directory called disk-free- monitoring:

```
$ cd disk-free-monitoring
```

The directory contains two files, `handler.py` and `serverless.yml`. The handler file will contain the code of our Lambda function while `serverless.yml` will have the information about how to deploy and configure our function. We will start there.

With your text editor, open the `serverless.yml` file. The file is broken up into different sections.

The first change we will do is to add IAM permissions to our function. We want our function to be able to create and delete alarms. For that, find the provider block in the configuration file and add the following:

```
provider:
    name: aws
    runtime: python2.7
    iamRoleStatements:
        - Effect: "Allow"
          Action:
            - "cloudwatch:PutMetricAlarm"
            - "cloudwatch:DeleteAlarms"
          Resource: "*"
```

Toward the middle of the file, a section defines the name of the handler:

```
functions:
    hello:
        handler: handler.hello
```

While ultimately we could create a function and call it hello, we can also come up with something more descriptive about the action. We will change the name to alarm as follows:

```
functions:
    alarm:
        handler: handler.alarm
```

Lastly, we need to define how our function will get triggered. After the handler definition, add the following (events and handler are aligned):

```
events:
    - cloudwatchEvent:
      event:
        source:
            - "aws.ec2"
        detail-type:
            - "EC2 Instance State-change Notification"
        detail:
            state:
                - running
                - stopping
                - shutting-down
                - stopped
                - terminated
```

We will now edit the `handler.py` file.

When you first open the file, it shows a basic `hello` function. We won't keep any of it. As a first step, delete everything in that file. We will start our file with the import and initialization of the `boto3` library.

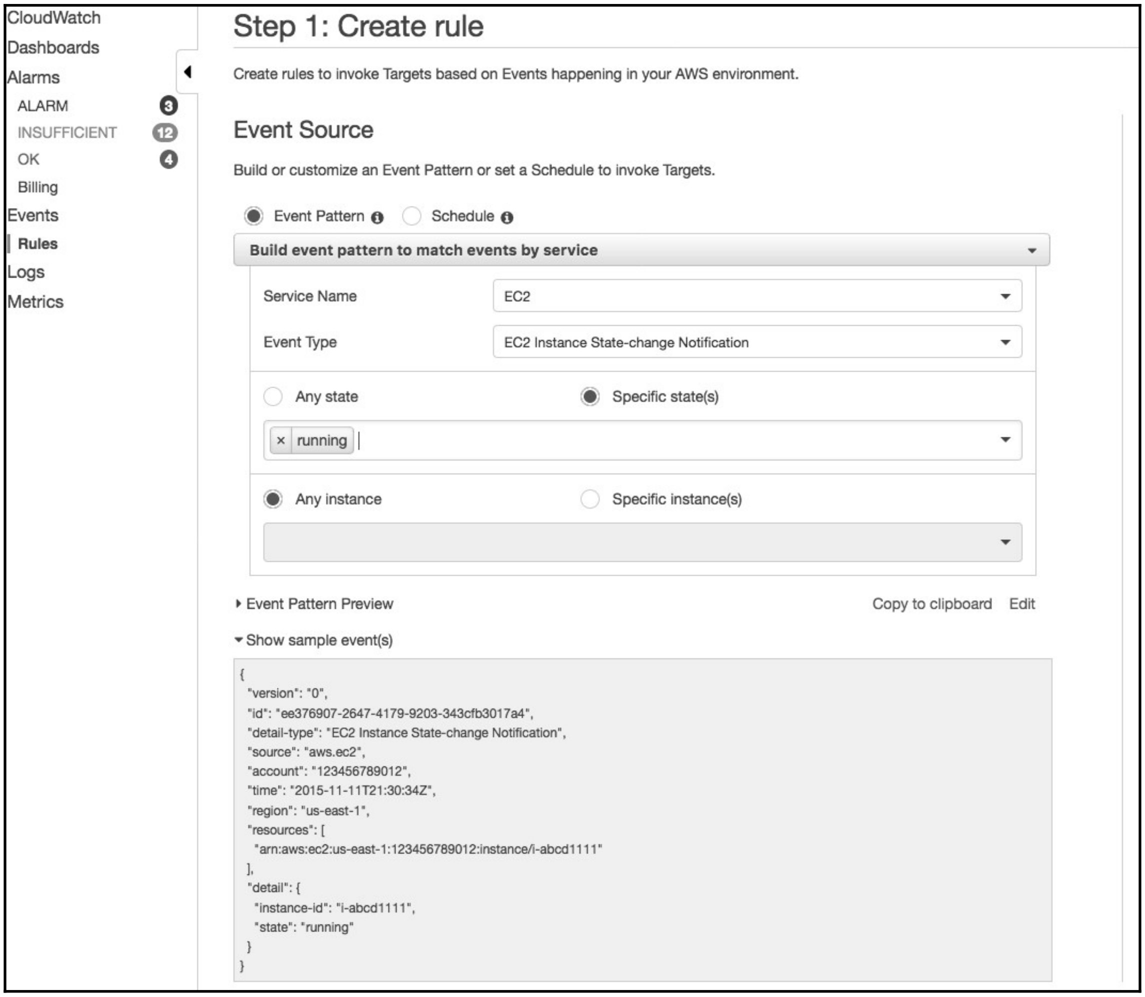
```
import boto3
client = boto3.client('cloudwatch')
```

We will now create a function and call it alarm in reference to the handler value defined in our last file (`handler.alarm`). The function takes two arguments, event and context:

```
def alarm(event, context):
```

The event will contain a JSON with the information that the EC2 instance state change received. You can see sample events by using the CloudWatch event web interface.

With your browser, open `https://console.aws.amazon.com/cloudwatch/home?region=us-east-1#rules:action=create` and then provide the new information of the event you want to match, as shown in this screenshot:



Step 1: Create rule

Create rules to invoke Targets based on Events happening in your AWS environment.

Event Source

Build or customize an Event Pattern or set a Schedule to invoke Targets.

[✓ Event Pattern] Schedule

Build event pattern to match events by service

Service Name: EC2

Event Type: EC2 Instance State-change Notification

Any state [✓ Specific state(s)]

running

[✓ Any instance] Specific instance(s)

Event Pattern Preview — Copy to clipboard — Edit

Show sample event(s)

```
{
  "version": "0",
  "id": "ee376907-2647-4179-9203-343cfb3017a4",
  "detail-type": "EC2 Instance State-change Notification",
  "source": "aws.ec2",
  "account": "123456789012",
  "time": "2015-11-11T21:30:34Z",
  "region": "us-east-1",
  "resources": [
    "arn:aws:ec2:us-east-1:123456789012:instance/i-abcd1111"
  ],
  "detail": {
    "instance-id": "i-abcd1111",
    "state": "running"
  }
}
```

In our case, we want to extract two pieces of information, the `instance-id` and the state. We will do that as follows:

```
instance = event['detail']['instance-id']
state = event['detail']['state']
```

We want to create alarms when an instance is running and delete them when they are in one of the other states listed in the `serverless.yml` file (stopping, shutting-down, stopped, terminated). We will create two alarms: a warning email alert when the partition is filled to 60% and a page for when we reach 80%.

We will do that by creating two functions, `put_alarm` and `delete_alarms`. For now, we will simply call them as follows:

```
if state == "running":
    warning = put_alarm(instance, 60, 'alert-email')
    critical = put_alarm(instance, 80, 'alert-sms')
    return warning, critical
else:
    return delete_alarms(instance)
```

We can now define our two functions, starting with the `put_alarm` function:

```
def put_alarm(instance, threshold, sns):
```

The function takes three arguments, the instance ID, the threshold of the alarm, and the topic information.

We will first define the `sns_prefix` information. We can get that value using the following command:

```
$ aws sns list-topics \
    sns_prefix = 'arn:aws:sns:us-east-1:511912822958:'
```

The next step will be to create the alarm. We will want to store the response so that we can return that to the Lambda execution:

```
response = client.put_metric_alarm(
```

We now need to provide all the information needed to create the alarm, starting with its name. The name of the alarm has to be unique to the AWS account. We will make sure this is the case by using the instance ID and `sns` suffix to generate the alarm name:

```
AlarmName='DiskSpaceUtilization-{}-{}'.format(instance, sns),
```

We now need to provide the details of the metric to monitor as follows. We will first provide the metric name and namespace followed by the dimensions. In the dimensions section, we are able to limit the monitoring to only our instance ID thanks to the information provided by CloudWatch through the event variable:

```
MetricName='DiskSpaceUtilization',
Namespace='System/Linux',
Dimensions=[
{
    "Name": "InstanceId",
    "Value": instance
},
{
    "Name": "Filesystem",
    "Value": "/dev/xvda1"
},
{
    "Name": "MountPath",
    "Value": "/"
}
],
```

We are going to define the threshold information as follows:

```
    Statistic='Average',
    Period=300, Unit='Percent',
    EvaluationPeriods=2,
    Threshold=threshold,
    ComparisonOperator='GreaterThanOrEqualToThreshold',
    TreatMissingData='missing',
```

In this particular case, we want to have two consecutive executions of 5 minutes where the average disk usage is higher than 60 or 80% to trigger the alarms. Finally, we are going to specify the topics to send the message to when the alert triggers and recovers:

```
    AlarmActions=[
           sns_prefix + sns,
    ],
    OKActions=[
           sns_prefix + sns,
    ]
    )
    return response
```

The function finishes with the return of the response. We will now create the function that deletes them. For that, we will create the function and call it `delete_alarms`. The code to delete the alarm is a lot simpler. We simply need to call the `boto` function, `delete_alarms`, and provide it an array with the two names of the alert we created:

```
    def delete_alarms(instance):
        names = [
            'DiskSpaceUtilization-{}-alert-email'.format(instance),
            'DiskSpaceUtilization-{}-alert-sms'.format(instance)
    ]
    return client.delete_alarms(AlarmNames=names)
```

The `handler.py` is done, but in order to make this code work, we need to create a few extra files. The first file we want to add is `requirements.txt`. This file defines the libraries required by our Python code to run. In our case, we need `boto`.

In the same directory as `handler.py` and `serverless.yml`, create a file and call it requirements.txt. In it, add boto3==1.4.4.

The serverless file doesn't automatically handle those requirement files. In order to handle them, we need to create a package.json file in the same directory as the other files and put the following in it:

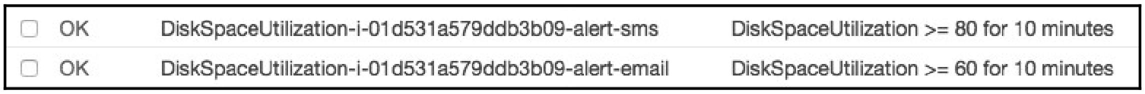
```
{
"name": "disk-free-monitoring",
"version": "1.0.0",
"description": "create cloudwatch alarms for disk space",
"repository": "tbd",
"license": "ISC",
"dependencies": {
    "serverless-python-requirements": "^2.3.3"
}
}
```

We now can run the command npm install.

With those two extra files created, we are ready to deploy our application as follows:

```
$ serverless deploy
Serverless: Packaging service... Serverless: Creating Stack...
Serverless: Checking Stack create progress...
.....
Serverless: Stack create finished...
Serverless: Uploading CloudFormation file to S3...

Serverless: Uploading artifacts...
Serverless: Uploading service .zip file to S3 (1.17 KB)... Serverless:
Updating Stack...
Serverless: Checking Stack update progress...
.....................
Serverless: Stack update finished...
Service Information
service: disk-free-monitoring stage: dev
region: us-east-1 api keys:
None endpoints:
None functions:
alarm: disk-free-monitoring-dev-alarm
```

From that point on, any EC2 instance that gets created in us-east-1 will automatically get two dedicated alarms while the instances are running:



| | | | |
|---|---|---|---|
| ☐ | OK | DiskSpaceUtilization-i-01d531a579ddb3b09-alert-sms | DiskSpaceUtilization >= 80 for 10 minutes |
| ☐ | OK | DiskSpaceUtilization-i-01d531a579ddb3b09-alert-email | DiskSpaceUtilization >= 60 for 10 minutes |

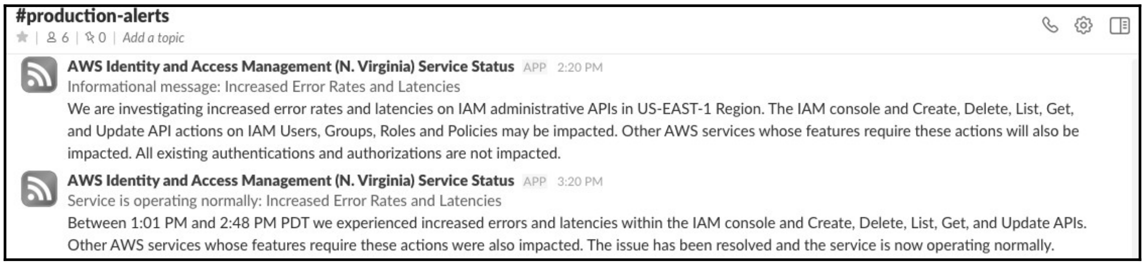
We won't show it in the book, but there are many things you can improve in this script, including looking at the EC2 tags of your instances to see if it's a production system or not.

Lastly, we will take a closer look at a service that AWS calls personal health.

## Monitoring and alerting with AWS health

While AWS is mostly stable, and outages are rare, it is not exempt from occasional service degradation. To check on the general health of their service, you go to their main dashboard at `https://status.aws.amazon.com`.

Note that this dashboard also provides an RSS feed, which can be integrated with most communication services, such as Slack:



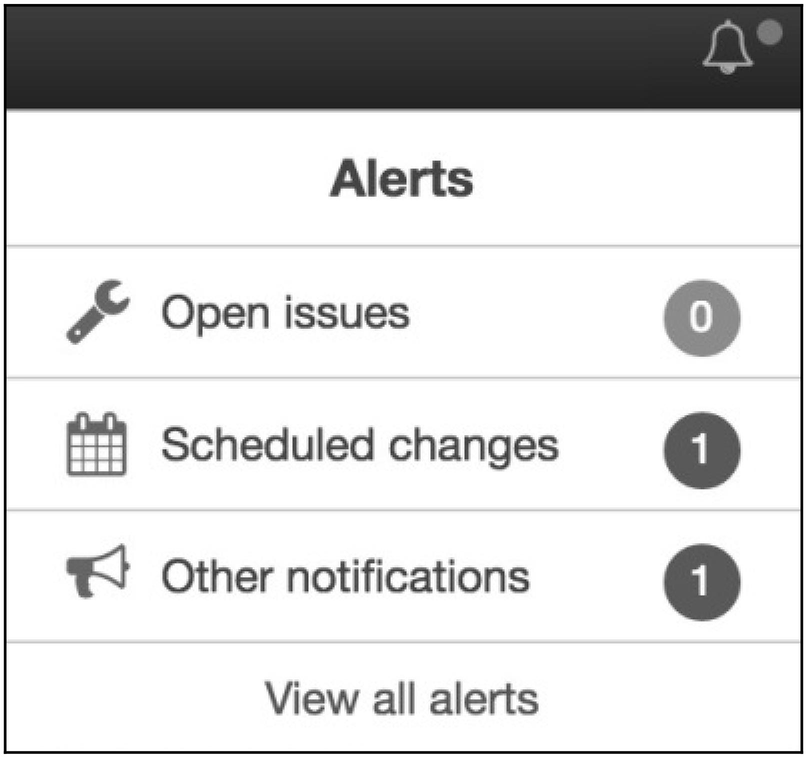
**#production-alerts**

6 | 0 | *Add a topic*

**AWS Identity and Access Management (N. Virginia) Service Status** APP 2:20 PM
Informational message: Increased Error Rates and Latencies
We are investigating increased error rates and latencies on IAM administrative APIs in US-EAST-1 Region. The IAM console and Create, Delete, List, Get, and Update API actions on IAM Users, Groups, Roles and Policies may be impacted. Other AWS services whose features require these actions will also be impacted. All existing authentications and authorizations are not impacted.

**AWS Identity and Access Management (N. Virginia) Service Status** APP 3:20 PM
Service is operating normally: Increased Error Rates and Latencies
Between 1:01 PM and 2:48 PM PDT we experienced increased errors and latencies within the IAM console and Create, Delete, List, Get, and Update APIs. Other AWS services whose features require these actions were also impacted. The issue has been resolved and the service is now operating normally.

In addition to that global status page, you can also access a personalized health dashboard in the AWS console by clicking on the bell icon:



You can also access the dashboard directly by opening `https://phd.aws.amazon.com` in your browser. The personalized health dashboard will display information affecting all customers in the region and also notifications that are specific to your account, such as when one of your instances is scheduled for maintenance and reboot. The personalized health dashboard doesn't have an RSS feed, but instead is integrated into the CloudWatch event.

We are going to create a new rule in the CloudWatch event to send us email notifications of the different alerts.

We will do that using the command-line interface:

1. The first step will be to create a rule that matches all events coming from the endpoint `aws.health`. We will do that with the following command:

```
$ aws events put-rule \
    --name AWSHealth \
    --event-pattern '{"source":["aws.health"]}' \
    --state ENABLED
{
    "RuleArn": "arn:aws:events:us-
east-1:511912822958:rule/AWSHealth"
}
```

2. Next, we will get the information of our target. In our case, the target is the SNS topic created earlier in the chapter. We will need to get the `TopicARN`, which you can get with the following command:

```
$ aws sns list-topics | grep alert-email
    "TopicArn": "arn:aws:sns:us-east-1:511912822958:alert-email"
```

3. Finally, we can tie the two together. The target's command expects a JSON entry, which we provide here using the following shorthand syntax:

```
aws events put-targets \
    --rule AWSHealth\
    --targets Id=1,Arn=arn:aws:sns:us- east-1:511912822958:alert-
email
```

Throughout the course of this section, we explored how to create alerts and applied this method to a few of our key public indicators. If you wish, you can continue that exercise, reusing some of the techniques we explored to put in place more alarms to make sure you don't miss any important events.

**Documentation**

All the work done so far will only be useful if you create good documentation to go with it. At the very least, your documentation should cover the different failure scenarios and how to recover from them.

## Summary

In this chapter, we explored several ways to add monitoring and alerting to our application and infrastructure. We could do it reasonably well by taking advantage of some of the services AWS provides, including CloudWatch, ElasticSearch, and SNS. You can now continue the work of measuring everything. Ultimately, measurement needs to become part of the company culture.

There are several areas to explore, including the following:

- At the infrastructure level, you can start tracking AWS costs and create budgets. You can also put monitoring around your backups to make sure that you aren't missing backup failures.

- At the service level, you can look at X-Ray, the distributed request tracking service, to monitor performance.
- At the build and release pipelines level, with a couple of changes to CloudFormation, CodeDeploy, and CodePipeline, you can start tracking the frequency of deployments and rollbacks. In addition, you can do the same to Ansible and create an alert when Ansible returns an error when it runs.

Quality can also get this kind of treatment. You can start collecting test code coverage information to make sure you don't push new code without unit testing. It is also interesting to compare outage frequency and bugs/tickets. You can sometimes find a correlation between quality going down and outages going up.

While it may not seem obvious at times, the DevOps movement is first and foremost about people. All those improvements in our process and the adoption of new technologies are means to that end. For that reason, you also want to find ways to track the impact all those changes have on the different people in the company.

In `Chapter 8`, *Hardening the Security of Your AWS Environment*, we will continue using some of the components we built in this chapter, but this time from the perspective of security.

## Questions

1. What is AWS CloudWatch?

   CloudWatch is a native monitoring solution for all most all AWS services. It supports three different type of data stream named metrics, logs and events. We can access CloudWatch using the web console, the command-line interface, and, of course, the API and various SDKs.

2. What is ELK stack?

   ELK is the acronym for three open source projects: Elasticsearch, Logstash, and Kibana. Elasticsearch is a search and analytics engine. Logstash is a server-side data processing pipeline that ingests data from multiple sources simultaneously, transforms it, and then sends it to a stash like Elasticsearch. Kibana lets users visualize data with charts and graphs in Elasticsearch.

3. What is EKK stack?

   The EKK solution eliminates the undifferentiated heavy lifting of deploying, managing, and scaling your log aggregation solution, which means you can remove Logstash with AWS Kinesis Solution. With the EKK stack, you can focus on analyzing logs and debugging your application, instead of managing and scaling the system that aggregates the logs.

## Further reading

- **winston logger**: `https://www.npmjs.com/package/winston`
- **ELK Information**: `https://www.elastic.co/`
- **Amazon ElasticSearch Service**: `https://aws.amazon.com/elasticsearch-servic`