

# HOWTO: Implement the Consumption Framework

This HOWTO is intended to demonstrate one method of running the Consumption Framework (CF) in different setups:

- On a “bare metal” Linux OS, including inside a virtual machine.
  - As a native Docker container.
  - As a Kubernetes CronJob.
  - As a Lambda function on AWS.
- 

## BARE-METAL DEPLOYMENT

### Introduction

Many of our customers employ the concept of at least one “utility server” in their environment which is intended to run this exact nature of recurring task. This method of deploying the CF is suitable for them, or as a base from which they can craft their own specific to their Linux distribution, or for their own security requirements.

This procedure was developed on OpenSUSE Tumbleweed Server; it should translate fine to other **systemd** Linux distributions, such as RHEL / CentOS and derivatives / successors.

This setup procedure assumes superuser privileges (logged in as or **sudo** to the **root** user). Once the CF code is in place, the non-privileged **elastic-cf** user will run the code.

To import dashboards into Kibana, you will need the appropriate **.ndjson** file from inside the ZIP archive. Hence, place the ZIP archive on the system which will run it (for the procedure below) as well as unzipped on your local computer.

### HOWTO

First, copy the **consumption\_framework-main.zip** file into the root user’s home directory (**/root**).

Next, create a user to securely run the CF code.

Then, create a folder for the CF code, extract files from the ZIP archive and set up a Python virtual environment to contain the application dependencies and Python environment separate from the system itself.

Finally, have the new non-privileged user 'own' the new directory and files.

The steps to do all this save for copying the ZIP archive into the `/root` directory are in the Bash script below, which can be run (with `sudo` or as `root`) on the CF script host to automate this process:

```
#!/bin/bash
# add elastic-cf user
useradd --shell /usr/sbin/nologin elastic-cf
# create directory for CF code
mkdir /opt/elastic-cf
# create a Python virtual environment for dependencies
cd /opt/elastic-cf
python3 -m venv elastic-cf-venv
# activate the Python virtual environment
source elastic-cf-venv/bin/activate
# move the ZIP archive into place and unpack it
cp /root/consumption_framework-main.zip /opt/elastic-cf
unzip consumption_framework-main.zip
# install Python dependencies
pip3 install -r consumption_framework-main/requirements.txt
# set elastic-cf user & group to own the CF directory
chown -R elastic-cf:elastic-cf /opt/elastic-cf
```

Once the setup is complete, use the `su` command to switch to the `elastic-cf` user and initialize the `config.yml` file template:

```
su elastic-cf
cd ~
source /opt/elastic-cf/elastic-cf-venv/bin/activate
python3 /opt/elastic-cf/consumption_framework-main/main.py init
```

## CREATE CONFIG.YML

Once the above setup is complete, you will need to create the configuration file to reference where to get the data from, where to send the data to and all other credentials for the script to run properly. You can use the snippet provided in [Sample configuration](#) as a starting point.

Once all details are set in the configuration file, the Consumption Framework code can be used to analyze data on a monitoring cluster.

As API keys are stored in the `config.yml` file, it is important to secure this file. To secure this file for the user `elastic-cf`, use `chmod` to change the file permissions:

```
chmod 400 /home/elastic-cf/config.yml
```

This will give read-only access to only the `elastic-cf` user.

## RUN THE CONSUMPTION FRAMEWORK CODE

The Consumption Framework has the following basic usage:

```
# python3 main.py --help
# python3 main.py [OPTIONS] COMMAND [ARGS]...
```

Once the `config.yml` configuration file is set up in the `elastic-cf` user's home directory, run the CF code again with the `init` option to prime the destination cluster with the consumption results index template, ingest pipeline and ILM policy.:

```
# su elastic-cf
# cd ~
# source /opt/elastic-cf/elastic-cf-venv/bin/activate
# python /opt/elastic-cf/consumption_framework-main/main.py init
```

Edit the ILM policy for `consumption` at this point as appropriate; the default is to keep the consumption data in the hot tier for 365 days and then delete it.

**NOTE:** the documents created by the consumption framework are dated according to the monitoring data they are based on. If you move the data too fast to non-writeable tiers, you might not be able to ingest historical data.

Next, import initial Elasticsearch Service / Elastic Cloud billing data:

```
# python /opt/elastic-cf/consumption_framework-main/main.py
get-billing-data -config-file config.yml --lookbehind=12 --force
```

Finally, pull initial cluster usage data from the Monitoring Cluster:

```
# python /opt/elastic-cf/consumption_framework-main/main.py
consume-monitoring -config-file config.yml --lookbehind=12 --force
```

## SET UP CRON TASK

The **cron** scheduler is used to execute the CF code repeatedly on a schedule.

**NOTE:** Running the CF with a frequency greater than 1 hour will not bring any benefit. Further, the frequency should be higher than your monitoring cluster data retention rate, or there will be holes in the data. Finally, we recommend the **lookbehind** value being greater than your **cron** job frequency. A frequency of 3-12 hours is usually appropriate. To run every four hours, set a **cron** frequency of 4 hours (`0 */4 * * *`), set a **lookbehind** value of at least 5, and have at least 6 hours of historical data in your monitoring cluster.

**NOTE:** Your **lookbehind** time cannot be longer than a cluster or cloud account has existed.

**NOTE:** When running the CF repeatedly via **cron**, remove the `--force` parameter.

**NOTE:** When running via a **cron** job, the CF will look for its **config.yml** file in the **\$HOME** directory of the user. If running as the **elastic-cf** user, as in this example, the **config.yml** file should be placed in the **/home/elastic-cf** directory.

For example, to set up the Consumption Framework to update every hour at 29 minutes past the hour every fourth hour, run **crontab** to edit the **cron** configuration file for the **elastic-cf** user:

```
# su elastic-cf
# crontab -e
```

Alternatively, if logged in as **root**, you can edit the **elastic-cf** user's **crontab** file directly:

```
# crontab -e -u elastic-cf
```

While editing, add lines to cause the CF code to run repeatedly (no line breaks; three distinct lines; spaces between lines as shown here are acceptable):

Unset

```
# get Consumption Framework data at :29 past every 4 hours (when the hour is
divisible by 4)
```

```
29 */4 * * * source /opt/elastic-cf/elastic-cf-venv/bin/activate && python
/opt/elastic-cf/consumption_framework-main/main.py get-billing-data
--config-file config.yml --lookbehind=8

29 */4 * * * source /opt/elastic-cf/elastic-cf-venv/bin/activate && python
/opt/elastic-cf/consumption_framework-main/main.py consume-monitoring
--config-file config.yml --lookbehind=8
```

These **cron** jobs will launch a shell, activate the Python virtual environment and then, if that is successful, execute the CF code. If you have no cloud instances with billing data, you can skip the **get-billing-data** command usage and just run the CF with the **consume-monitoring** command.

Once you are through these steps, please refer to the [COMMON INSTALL STEPS](#) section.

## DOCKER DEPLOYMENT

### Introduction

For a self-managed environment, docker is a common deployment mechanism. This approach can also be used on ECE deployments, since they rely on docker being installed on the runners.

This procedure was developed on a MacOS with standard docker distribution, but should translate to other similar systems.

To import dashboards into Kibana, you will need the appropriate **.ndjson** file from inside the ZIP archive. Hence, place the ZIP archive on the system which will run it (for the procedure below) as well as unzipped on your local computer.

### HOWTO

First, copy the **consumption\_framework-main.zip** file into your user's home directory.

## CREATE THE IMAGE

You will then need to build the image for use with docker. We have a convenient Makefile for this purpose, or you can also manually point to the local Dockerfile to tag the image as you please:

```
# make build
```

OR

```
# docker build -t docker.elastic.co/cloud/consumption_framework:latest  
.
```

The image uses the entrypoint.sh script as entrypoint, which will allow you to also use env variables to pass configuration parameters to the script, should it be better than using the raw configuration file.

Once the image is built, you can verify that everything works by running:

```
# docker run --rm docker.elastic.co/cloud/consumption_framework:latest  
--help
```

You should see a help message describing the available commands:

```
Usage: main.py [OPTIONS] COMMAND [ARGS]...
```

Options:

```
--help Show this message and exit.
```

Commands:

consume-monitoring	Consume monitoring data from an existing cluster
get-billing-data	Recover org-level billing data from ESS
init	Initialize the target cluster

## CREATE CONFIG.YML

Once the above setup is complete, you will need to create the configuration file to reference where to get the data from, where to send the data to and all other credentials for the script to run properly. You can use the snippet provided in [Sample configuration](#) as a starting point.

Once all details are set in the configuration file, the Consumption Framework code can be used to analyze data on a monitoring cluster.

## RUN THE CONSUMPTION FRAMEWORK CODE

Since the configuration file is not present in the Docker image, you will need to mount it at runtime using the `-v` option. You can then specify the command you'd like to run and any additional parameter.

We start by initializing the cluster with the necessary ingest pipeline and index template.

```
# docker run --rm -v $(pwd)/config.yml:/app/config.yml
docker.elastic.co/cloud/consumption_framework:latest init
--config-file /app/config.yml
gi
```

We can then run the get-billing-data (for cloud-based deployments) and consume-monitoring commands:

```
# docker run --rm -v $(pwd)/config.yml:/app/config.yml
docker.elastic.co/cloud/consumption_framework:latest get-billing-data
--config-file /app/config.yml --lookbehind 24
# docker run --rm -v $(pwd)/config.yml:/app/config.yml
docker.elastic.co/cloud/consumption_framework:latest
consume-monitoring --config-file /app/config.yml --lookbehind 24
```

Additional parameters can be specified, more information about that can be found in the [Configuration parameters](#) section.

## AUTOMATION / CRONJOB

As the configuration of a cronjob within a docker container arguably goes against the principle of containerized workloads, we recommend relying on external components for recurring runs.

You could for example configure cronjobs executing the docker commands rather than the direct python script, in that case using an approach similar to what is described in the [SET UP CRON TASK](#) section of this document:

Unset

```
# get Consumption Framework data at :29 past every 4 hours (when the hour is
divisible by 4)
```

```
29 */4 * * * docker run --rm -v $(pwd)/config.yml:/app/config.yml
docker.elastic.co/cloud/consumption_framework:latest get-billing-data
--config-file /app/config.yml --lookbehind 8
```

```
29 */4 * * * docker run --rm -v $(pwd)/config.yml:/app/config.yml
docker.elastic.co/cloud/consumption_framework:latest init --config-file
/app/config.yml --lookbehind 8
```

## ENV VARIABLES

A particularity of the docker image is that you can also pass environment variables to set configuration parameters. This is useful when you don't want to write sensitive information in the configuration file, or even the command line that is run.

Two prefix-based mechanisms are used to pass along parameters to the script from environment variables:

1. All environment variables starting with `CF_` are passed to the python script. Some examples are shown in the below table:

Environment variable	Script translation
<code>CF_DEBUG=' '</code>	<code>--debug</code>
<code>CF_THREADS=15</code>	<code>--threads 15</code>

2. All environment variables starting with `CFCONF_` are passed as additional `--config` flag to the script. Some examples are shown below:

Environment variable	Script translation
<code>CFCONF_ORGANIZATION_ID=1234</code>	<code>--config organization_id=1234</code>
<code>CFCONF_MONITORING_SOURCE_HOSTS=http://elasticsearch:9200</code>	<code>--config monitoring_source.hosts=http://elasticsearch:9200</code>

Once you are through these steps, please refer to the [COMMON INSTALL STEPS](#) section.

## KUBERNETES DEPLOYMENT

### Introduction

Running the Consumption Framework as a Kubernetes CronJob is a great way to automate data collection, if you already have a Kubernetes footprint.

This approach is very similar to the docker-based one, with the added benefit of using native Kubernetes scheduling for recurring jobs.

We strongly suggest you make yourself familiar with the [DOCKER DEPLOYMENT](#) section prior to continuing your read of this section, as the initial image build step is identical, and the same parametrization characteristics are offered.



## HOWTO

First, copy the `consumption_framework-main.zip` file into your user's home directory.

### CREATE & DISTRIBUTE THE IMAGE

The initial step is identical to the docker-based deployment, please refer to the [CREATE THE IMAGE](#) section.

You will then need to publish the image to your repository for use in your Kubernetes cluster.

### CREATE CONFIG.YML

Once the above setup is complete, you will need to create the configuration file to reference where to get the data from, where to send the data to and all other credentials for the script to run properly. You can use the snippet provided in [Sample configuration](#) as a starting point.

We are then going to create a Kubernetes secret containing this file.

```
# kubectl create secret generic cf-configuration
--from-file=config.yml
```

### INIT CONSUMPTION\_DESTINATION

Before we fully configure the Consumption Framework as CronJob, we will start a single one shot container to run the init command and create the necessary ingest pipeline and index template in the target cluster.

The below manifest will do just that:

```
Unset
apiVersion: v1
kind: Pod
metadata:
  name: cf-init
spec:
  containers:
  - name: cf-init
    image: docker.elastic.co/cloud/consumption_framework:latest
    args:
      - "init"
      - "--config-file=/app/config.yml"
  volumeMounts:
```

```

- name: cf-configuration
  mountPath: /app/config.yml
  subPath: config.yml
volumes:
- name: cf-configuration
  secret:
    secretName: cf-configuration
    items:
      - key: config.yml
        path: config.yml

```

```

# We apply the manifest
# kubectl apply -f pod.yml
# And then verify the execution was successful
# kubectl logs cf-init
2024-04-05 15:09:26,366 [MainThread] - INFO (consumption): Created
consumption object for None
2024-04-05 15:09:26,366 [MainThread] - INFO (consumption):
Initializing consumption package Elasticsearch assets

```

## CRONJOBS

Now that everything is set, we can create a cronjob for **get-billing-data** (if applicable) and another for **consume-monitoring**.

```

Unset
apiVersion: batch/v1
kind: CronJob
metadata:
  name: cf-get-billing-data-cronjob
spec:
  schedule: "0 */4 * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: cf-get-billing-data
              image: docker.elastic.co/cloud/consumption_framework:latest
              args:
                - "get-billing-data"

```

```

      - "--config-file=/app/config.yml"
      - "--lookbehind=8"
    volumeMounts:
      - name: cf-configuration
        mountPath: /app/config.yml
        subPath: config.yml
    volumes:
      - name: cf-configuration
        secret:
          secretName: cf-configuration
          items:
            - key: config.yml
              path: config.yml

```

Unset

```

apiVersion: batch/v1
kind: CronJob
metadata:
  name: cf-consume-monitoring-cronjob
spec:
  schedule: "0 */4 * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: cf-consume-monitoring
              image: docker.elastic.co/cloud/consumption_framework:latest
              args:
                - "consume-monitoring"
                - "--config-file=/app/config.yml"
                - "--lookbehind=8"
              volumeMounts:
                - name: cf-configuration
                  mountPath: /app/config.yml
                  subPath: config.yml
          volumes:
            - name: cf-configuration
              secret:
                secretName: cf-configuration
                items:

```

```
- key: config.yml
  path: config.yml
```

Similar to docker-based runs, you can also pass parameters using environment variables. You can find more information about this in the [ENV VARIABLES](#) section of this document.

Once you are through these steps, please refer to the [COMMON INSTALL STEPS](#) section.

## AWS LAMBDA DEPLOYMENT

### Introduction

In a cloud-native environment, AWS lambda is definitely a great choice for Consumption Framework runs. You can simply build your container image and ask the functions to be run on a predefined schedule.

A couple prerequisites are required to run the steps listed below:

- The [AWS CLI tool](#) should be installed and configured to your region of choice.
- The Lambda is created for python 3.11 (and this version **only**), make sure you have the correct one (possibly using [virtual environment](#)).
- You need to be able to run [GNU make](#).

### HOWTO

First, copy the `consumption_framework-main.zip` file into your user's home directory.

### CREATE THE ZIP

A convenient Makefile target is available to create two zip archives for the Lambda function:

```
# make lambda-layers
```

This will create the dependency layer and the main application code, both for arm64 architecture.

### CREATE SECRET

We will be storing sensitive information inside of AWS Secrets Manager:

1. Go to the [AWS Console > Secrets Manager > Store a new secret.](#)
2. Select **Other type of secret.**
3. You will then want to use the same configuration values as what is documented in the [Configuring the CF](#) section, using "flat" keys (`{"foo": {"bar": "baz"}}` is literally `{"foo.bar": "baz"}`). You can use the key/value pairs view to verify things are properly configured.

Example:

Key/value	Plaintext
Secret key	Secret value
organization_name	todds
organization_id	28[REDACTED]
billing_api_key	essu[REDACTED]
monitoring_source.hosts	https://my-ccs-source.es.us-east-1.aws.found.io:443
monitoring_source.api_key	[REDACTED]
monitoring_source.retry_on_timeout	true
consumption_destination.hosts	https://my-ccs-source.es.us-east-1.aws.found.io:443
consumption_destination.api_key	[REDACTED]
monitoring_source.request_timeout	60

4. Make sure you write down the resulting ARN of the secret, as you will need to pass it to the Lambda function.

## CREATE LAMBDA FUNCTION

Another Makefile target can be used to create the Lambda function, it will sequentially perform the following actions:

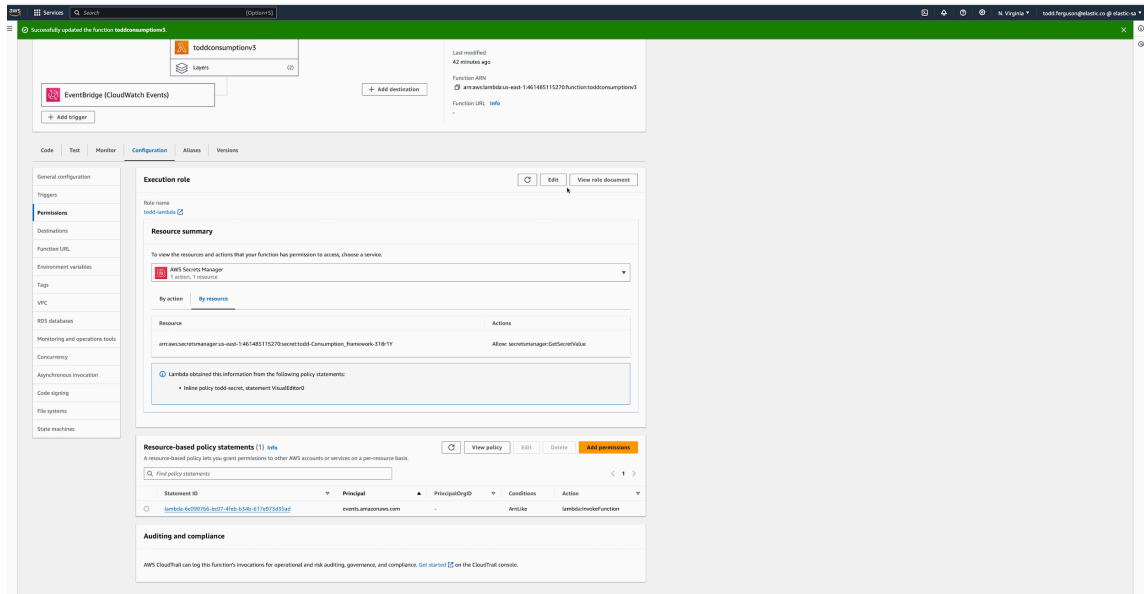
- Create a basic role for the Lambda function.
- Push the dependency layer to AWS.
- Create the Lambda function, using the previously created role and dependency layer. The main application code zip is used as content.
- The `AWSLambdaBasicExecutionRole` role is added to the execution policy of the Lambda, which will allow Secrets consumption

**# make lambda-release**

Additional actions need to be performed manually, since they are not currently included in the Makefile target:

1. Addition of the `AWS-Parameters-and-Secrets-Lambda-Extension-Arm64` Lambda layer, in **Layers > Add a layer > AWS Layers.**
2. Addition of `Allow: secretsmanager:GetSecretValue` action in the role policy on the previously created secret. In **Configuration > Permissions > consumption\_framework\_role > Add permissions > Create inline policy.** You can then fill the sequential dropdown menus with `SecretsManager,`

`GetSecretValue` and finally the ARN of the configuration file from the previous step. Don't forget to save the new policy, with any name of your choosing.



3. Addition of the necessary environment variables in **Configuration > Environment variables**:

- a. **config\_arn** - set this to the ARN of the secret you previously created.
- b. **command** - either get-billing-data or consume-monitoring.
  - i. Get-billing-data
  - ii. Consume-monitoring
  - iii. both
- c. Any other [non-config related command line parameters](#).

Environment variables (3)		Edit
The environment variables below are encrypted at rest with the default Lambda service key.		
<input type="text" value="Find environment variables"/>		
Key	Value	
command	both	
config_arn	arn:aws:secretsmanager:us-east-1:██████████:secret:VisualEditor	
lookbehind	12	

4. Creation of the trigger to automate the Lambda execution: **Add trigger > EventBridge > Create a new rule**, then use an expression like **rate (6 hours)** or **rate (1 day)**.

⚠ Make sure that the rate of execution is greater than the **lookbehind** parameter of the script, otherwise you will have holes in your data.

## INITIALIZE DESTINATION CLUSTER

▲ The lambda container image doesn't contain the initialization code, since this is a one-time operation. As a result, it is recommended to follow one of the alternative methods to configure the destination cluster:

- [Bare metal](#)
- [Docker container](#)

Once you are through these steps, please refer to the [COMMON INSTALL STEPS](#) section.

# AWS LAMBDA DEPLOYMENT: Container Image

## Introduction

In a cloud-native environment, AWS lambda is definitely a great choice for Consumption Framework runs. You can simply build your container image and ask the functions to be run on a predefined schedule.

A couple prerequisites are required to run the steps listed below:

- The [AWS CLI tool](#) should be installed and configured to your region of choice.
- The Lambda is using containers, you need a container runtime such as docker to build the image. [Docker](#)
- You need to be able to run [GNU make](#).

## HOWTO

First, copy the `consumption_framework-main.zip` file into your user's home directory.

## Create the Container Image

```
# make build-lambda-image
```

```

@krol_valencia $ make build-lambda-image
docker build -f Dockerfile.lambda --platform linux/amd64 -t my-consumption:aeb78469320d2b3b0772d1cea26f2ac5592677dc .
[+] Building 1.7s (11/11) FINISHED                                docker:orbstack
=> [internal] load build definition from Dockerfile.lambda        0.0s
=> => transferring dockerfile: 449B                               0.0s
=> [internal] load metadata for public.ecr.aws/lambda/python:3.11 1.5s
=> [internal] load .dockerignore                                  0.0s
=> => transferring context: 2B                                     0.0s
=> [1/6] FROM public.ecr.aws/lambda/python:3.11@sha256:1816be4e08ecf134370c69d05ad3c9c72c7541d8e47ef838501397abb3e83 0.0s
=> [internal] load build context                                  0.0s
=> => transferring context: 11.54kB                                0.0s
=> CACHED [2/6] COPY requirements_lambda.txt /var/task            0.0s
=> CACHED [3/6] RUN pip install -r requirements_lambda.txt        0.0s
=> CACHED [4/6] COPY config.yml .                                  0.0s
=> CACHED [5/6] ADD consumption consumption                       0.0s
=> [6/6] COPY lambda_function.py /var/task                        0.1s
=> exporting to image                                             0.0s
=> => exporting layers                                             0.0s
=> => writing image sha256:9adfca93a53f887697218f6d39cf4585b30d94cec171dd91bb3c199258166613 0.0s
=> => naming to docker.io/library/my-consumption:aeb78469320d2b3b0772d1cea26f2ac5592677dc 0.0s
@krol_valencia $ make push-lambda-image

```

## Push the Container Image to the ECR

# make push-lambda-image

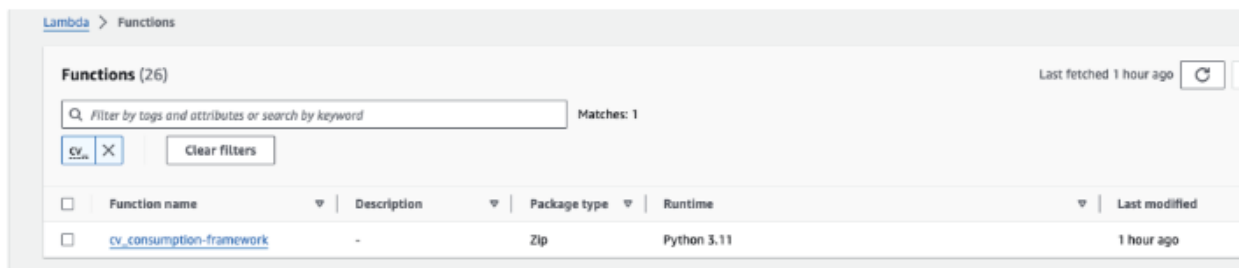
```

@krol_valencia $ make push-lambda-image
aws ecr get-login-password --region us-east-2 | docker login --username AWS --password-stdin [REDACTED]:ecr.us-east-2.
amazonaws.com
Login Succeeded
docker tag my-consumption:aeb78469320d2b3b0772d1cea26f2ac5592677dc [REDACTED]:dkr.ecr.us-east-2.amazonaws.com/my-consumption:aeb78469320d2b3b0772d1cea26f2ac5592677dc
docker push [REDACTED]:dkr.ecr.us-east-2.amazonaws.com/my-consumption:aeb78469320d2b3b0772d1cea26f2ac5592677dc
The push refers to repository [REDACTED]:dkr.ecr.us-east-2.amazonaws.com/my-consumption]
3aec7c3f49c3: Pushed
44636d57fcfe: Pushed
b32f66813bde: Pushed
e0ee4d82418e: Pushed
dbe0e2bdb196: Pushed
786eadf9f751: Pushed
5b8162a25aac: Pushed
e073f5919ae5: Pushed
b0063523a7bf: Pushed

```

## Create Lambda Function using ECR image

# make create-lambda-image-function

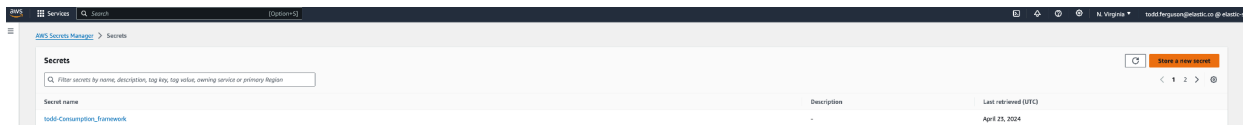


## Create Secret

1. Go to Secrets Manager



2. Click on store a new secret:



3. Click on Other Type Secret, and fill out like so:

## Choose secret type

**Secret type** [Info](#)

☐ Credentials for Amazon RDS database

☐ Credentials for Amazon DocumentDB database

☐ Credentials for Amazon Redshift data warehouse

☐ Credentials for other database

☒ Other type of secret  
API key, OAuth token, other.

## Key/value pairs

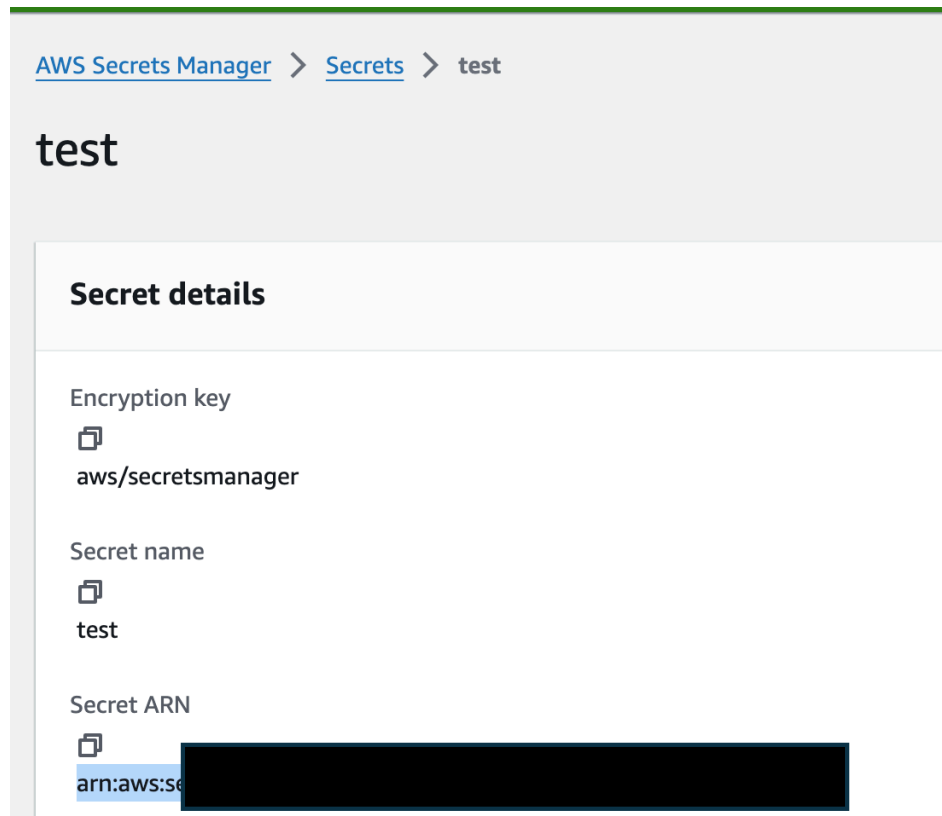
[Key/value](#) | [Plaintext](#)

organization_name	todds	<a href="#">Remove</a>
organization_id	121212122	<a href="#">Remove</a>
<a href="#">+ Add row</a>		

Required fields:

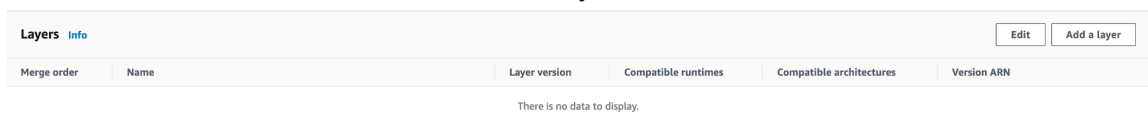
- organization\_name
  - organization\_id (This is the org ID in your ESS console)
  - billing\_api\_key
  - monitoring\_source.hosts
  - monitoring\_source.api\_key
  - monitoring\_source.retry\_on\_timeout (recommended value true)
  - consumption\_destination.hosts
  - Consumption\_destination.api\_key
  - monitoring\_source.request\_timeout (recommended value 60)
4. Do not enable Automatic Rotation.

5. After it is created Make sure you grab the Secret ARN

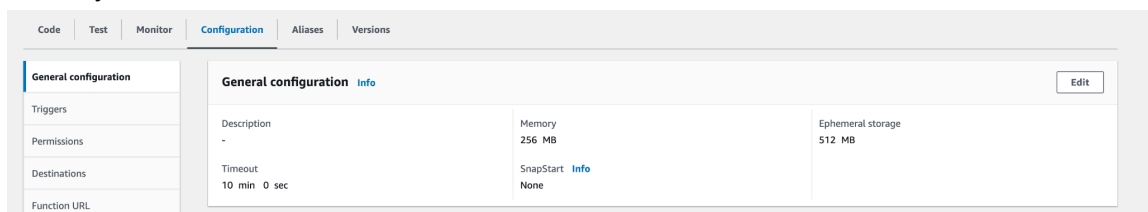


## Setting Lambda Function

1. Go to Lambda > Functions > My Function
- 2.
3. After the function is created scroll down to layers:



4. Go to Configuration > General Configuration. Modify Timeout to **10 minutes** and Memory to **256MB**



5. Go to Configuration > Environment Variables and add these.

Environment variables (3)		Edit
The environment variables below are encrypted at rest with the default Lambda service key.		
<input type="text" value="Find environment variables"/>		< 1 >
Key	Value	
command	both	
config_arn	arn:aws:secretsmanager:us-east-1: [REDACTED]	
lookbehind	12	

- config\_arn (Is the secret ARN from the secret created above)
  - lookbehind (is how far back the consume monitoring goes)
  - command is which function to run (generally recommended for both)
6. Go to Configuration > Permissions and click on the role name.

Code

Test

Monitor

Configuration

Aliases

Versions

General configuration

Triggers

Permissions

Destinations

Function URL

Execution role

Role name

Consumption-Framework-Doc-role-py05wkr7

Resource summary

To view the resources and actions that your function has permission to access, choose a service.

7. In the IAM > Roles > rolename, click on add permission > Create inline policy

IAM

Roles

Consumption-Framework-Doc-role-py05wkr7

Consumption-Framework-Doc-role-py05wkr7

Summary

Creation date  
April 21, 2024, 14:27 (UTC-05:00)

ARN  
arn:aws:iam::461485113270:role/service-role/Consumption-Framework-Doc-role-py05wkr7

Last activity  
-

Maximum session duration  
1 hour

Permissions

Permissions policies (1)

You can attach up to 10 managed policies.

Search

Filter by Type

All types

Attach policies

Create inline policy

☐

Policy name

Type

Attached entities

☐

205x.ambdatausdetectionrole-Ofdsaa17-243F-4B3a-b4f8-7d82719bcaad

Customer managed

1

8. Type Secrets Manager, and then select GetSecretValue (Do not click next)

# Specify permissions [Info](#)

Add permissions by selecting services, actions, resources, and conditions. Build permissions

## Policy editor

### ▼ Secrets Manager

**Allow** 1 Action

Specify what actions can be performed on specific resources in [Secrets Manager](#).

#### ▼ Actions allowed

Specify actions from the service to be allowed.

Manual actions | [Add actions](#)

☐ All Secrets Manager actions (secretsmanager:\*)

Access level

► List (2)

▼ Read (**Selected 1/5**)

☐ All read actions

☐ DescribeSecret [Info](#)

☒ GetSecretValue [Info](#)

► Write (11)

► Permissions management (3)

► Tagging (2)

### ▼ Resources

Image | Test | Monitor | **Configuration** | Aliases | Versions

General configuration

Triggers

**Permissions**

Destinations

Function URL

Environment variables

Tags

VPC

RDS databases

Monitoring and operations tools

Concurrency and recursion detection

Asynchronous invocation

Code signing

File systems


State machines

### Execution role

Role name  
cv-lambda-default [↗](#)

### Resource summary

To view the resources and actions that your function has permission to access, choose a service.

 **AWS Secrets Manager**  
1 action, 1 resource

By action | **By resource**

Resource	Actions
All resources	Allow: secretsmanager:GetSecretValue

① Lambda obtained this information from the following policy statements:

- Inline policy cv-secrets-all-v1-2024-09-policy, statement VisualEditor0

- Click on Add Arns to restrict access, and then paste your full arn on the bottom row. Hit Add Arn

### Specify ARN(s) ✕

**Visual** | Text

Resource in  
☒ This account ☐ Any account ☐ Other account

Resource region  
 ☐ Any region

Resource secret  
 ☐ Any secret

Resource ARN

Cancel **Add ARNs**

- Hit next, and put in a policy name and then create policy.


- Add Trigger : AWS Lambda Triggers

Creation of the trigger to automate the Lambda execution: \*\*Add trigger > EventBridge > Create a new rule\*\*, then use an expression like \*\*rate(6 hours)\*\* or \*\*rate(1 day)\*\*.

[Lambda](#) > Add triggers

## Add trigger

**Trigger configuration** [Info](#)

 **EventBridge (CloudWatch Events)**  
aws asynchronous schedule management-tools

**Rule**  
Pick an existing rule, or create a new one.  
☒ Create a new rule  
☐ Existing rules

**Rule name**  
Enter a name to uniquely identify your rule.

**Rule description**  
Provide an optional description for your rule.

**Rule type**  
Trigger your target based on an event pattern, or based on an automated schedule.  
☐ Event pattern  
☒ Schedule expression

**Schedule expression**  
Self-trigger your target on an automated schedule using [Cron or rate expressions](#). Cron expressions are in UTC.  
  
e.g. rate(1 day), cron(0 17 ? \* MON-FRI \*)

Lambda will add the necessary permissions for Amazon EventBridge (CloudWatch Events) to invoke your Lambda function from this trigger. [Learn more](#) about the Lambda permissions model.

[Cancel](#) [Add](#)

# GENERAL NOTES

## COMMON INSTALL STEPS

### INSTALL DASHBOARDS

In the `kibana_exports` folder of the ZIP archive, there are `.ndjson` files for Kibana; these are saved objects intended for upload.

In Kibana, go to **Stack Management** -> **Saved Objects** and click **Import** (upper right, as of 8.12.1). Next, select **Import** again and then choose the `.ndjson` file (e.g. `8.11.2.ndjson`) from your local filesystem. Click **Done** to close the sidebar window.

### GET RESULTS

In Kibana, you should see five new dashboards:

**Organization overview**  
**Deployment analysis**  
**Tier analysis**  
**Datastream analysis**  
**Usage analysis**

Open up the time selector in Kibana to view data historically (or if you see no data on first open with the default “last 15 minutes” time range).

### DEBUG LOGGING

To enable DEBUG level logging, add the `--debug` command line parameter when launching the CF Python code. This will emit a high verbosity of log entries.

By default, logs are sent to **STDOUT** (the console). If you want to capture these logs in a file, redirect output to the filesystem. For example if you used a bare-metal deployment:

```
# python main.py consume-monitoring --lookbehind=6 --debug > /path/to/logfile
```



If using the filesystem paths in the examples above, `/opt/elastic-cf/elastic-cf.log` would be an appropriate location to write the log file. This log file can then be read and entries forwarded by the Elastic Agent or Filebeat.

Rotation of this logfile must be done externally to the CF, such as with `logrotated`.

To use `logrotated` to rotate this log file, create a file named `elastic-cf` with the following contents and place it in the `/etc/logrotate.d/` directory:

```
Unset
/opt/elastic-cf/elastic-cf.log {
    daily
    compress
    dateext
    maxage 365
    rotate 99
    size +2048k
    notifempty
    missingok
    copytruncate
}
```

## MULTIPLE SOURCE MONITORING CLUSTERS

If multiple Monitoring Clusters are employed, use multiple `config.yml` files (one for each) and set up multiple execution jobs. For example, with a bare-metal approach, you could have multiple `crontab` entries:

Example:

```
Unset
29 */4 * * * python main.py get-billing-data --lookbehind=5 --config-file
config1.yml
29 */4 * * * python main.py get-billing-data --lookbehind=5 --config-file
config2.yml
```

## API KEY / SECRETS PROTECTION

Secrets (such as your API keys) can be further protected by not including them in the `config.yml` file.

Any parameters in the `config.yml` file can also be passed as single command-line arguments using the `--config` option; doing so will override any value also defined in the static `config.yml` file.

For example, to pass API keys held in environment variables as runtime configuration parameters, `main.py` could be launched with the following parameters specified on the command line as follows:

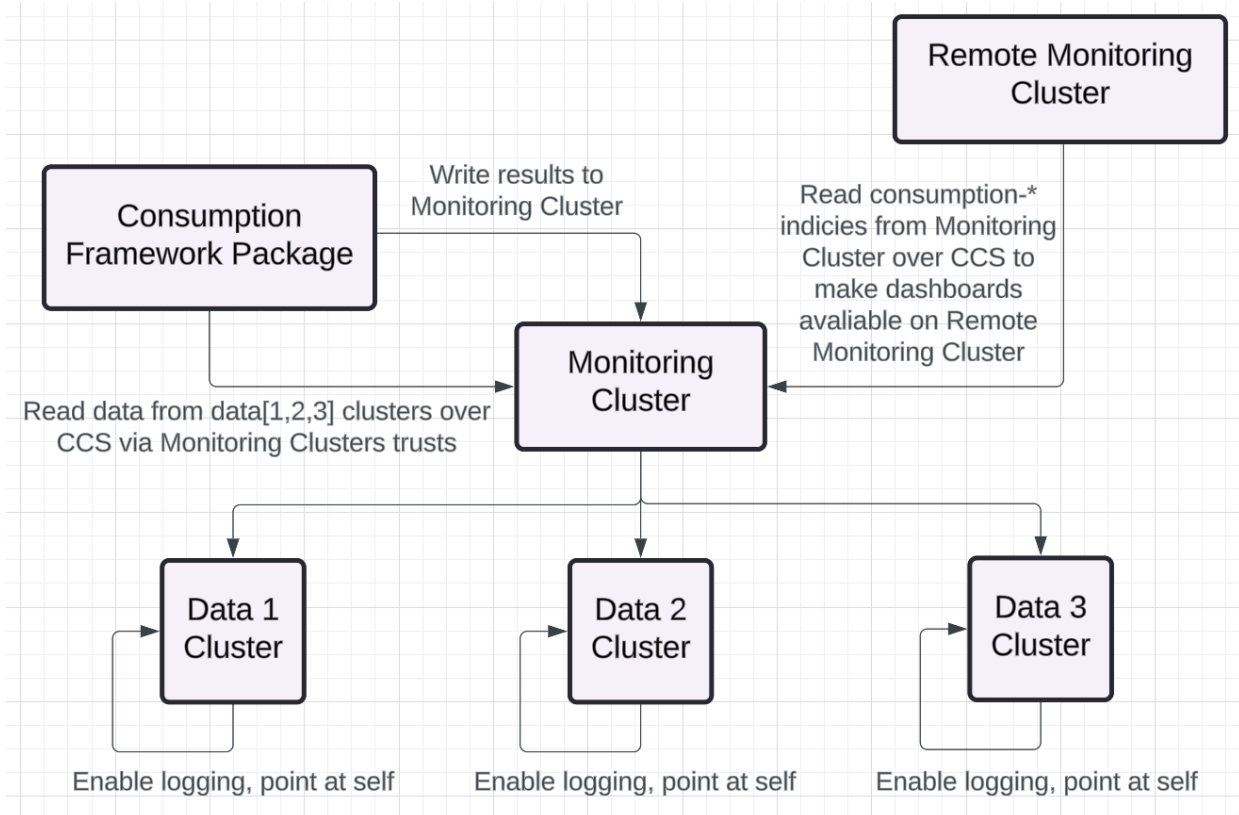
```
# python3 main.py get-billing-data --config-file config.yml
  --lookbehind=72 --debug \
  --config consumption_destination.api_key=$DEST_API_KEY \
  --config monitoring_source.api_key=$SOURCE_API_KEY
```

As shown above, if passing multiple configuration parameters as individual command-line options, multiple `--config` options should be declared.

You can find additional information about configuration mechanisms in the [Configuration definition](#) section.

## CROSS CLUSTER SEARCH (CCS) SCENARIO

The CCS feature of the Elastic Stack can be leveraged by the CF to query monitoring data from multiple clusters from a single endpoint. Below is a mock scenario to demonstrate the configuration requirements.



- Enable Logging on each of the **Data Clusters**. In this scenario the data clusters store their logs internally, not recommended, but sometimes is requested by clients.
- Configure each of the **Data Clusters** as a Remote Cluster on the **Monitoring Cluster**.
  - Validate CCS functionality with the **\_search** endpoint on the **Monitoring Cluster** via Dev Tools.  
I.e. (GET \*:monitoring-es-8\*/\_search)
  - Review the commands output and ensure the results include all the expected remote clusters
- Create API keys on the **Monitoring Cluster**, these will grant the CF permissions to Read and Write to the **Monitoring Cluster**
  - See [API Keys Permissions](#)
- Deploy the Consumption Framework, there are several runtime environments the package can be deployed into, see [Implement the Consumption Framework](#)
  - Update the **config.yml** parameter **monitoring\_index\_pattern** to query remote clusters. Adding "\*" in front of the index pattern will allow the CF to query the remote data clusters indices
    - I.e. (monitoring\_index\_pattern: '\*:monitoring-es-8\*')

```
# Uncomment and modify this if you want to read indices other than the default monitoring ones
# (for example using cross-cluster search)
monitoring_index_pattern: '*:monitoring-es-8*'

```

- Upload Consumption Dashboards to the **Monitoring Cluster**.
  - `./kibana_exports/${version}.ndjson`

- Test run the CF

Unset


```
python3 main.py consume-monitoring --lookbehind=12 --force --config-file config.yml
--compute-usages


```

- Review CF dashboards on the **Monitoring Cluster** and confirm you have data sourced from each of the **Data Clusters**.
- Configure the **Remote Monitoring Cluster** to read data from the **Monitoring Cluster**. This can be done by connecting the **Monitoring Cluster** to the **Remote Monitoring Cluster** as a remote cluster. We can then use CCS to read data from the **"consumption-\*"** indices stored on the **Monitoring Cluster** from the **Remote Monitoring Cluster**
  - Validate CCS from **Remote Monitoring Cluster** to the **Monitoring Cluster**
    - I.e. Run (`GET *:consumption*/_search`) on the **Remote Monitoring Cluster**
  - Upload CF Dashboards to the **Remote Monitoring Cluster**.
  - Modify CF Data Views to read from Remote Clusters
    - I.e. (`*:consumption*`)

**Edit data view**

Name  
consumption

Index pattern  
\*:consumption\* 

Timestamp field  
@timestamp 

Select a timestamp field for use with the global time filter.

[Show advanced settings](#)

- Depending on the version of the CF package used there could be a single or multiple Data Views in use. Be sure to update all of them.
  - Browse to CF Dashboards on the **Remote Monitoring Cluster** and confirm monitoring information is available from all **Data Clusters**

## Configuring the CF

Regardless of the chosen deployment method, you will be manipulating configuration parameters for the CF script. Depending on the configuration approach, you can use inline JSON, written YAML or individual inline key/value pairs.

### Sample configuration

The below yaml can be used as a starting point for your own configuration file.

```
Unset
---
# The name of your organization, this will be populated on every single
resulting document
organization_name: "Change me"

# The organization ID from Elastic Cloud, for ESS deployments.
# You can comment this out for on-premises deployments
organization_id: "12345"

# The billing API key for your organization
# You can comment this out for on-premises deployments
billing_api_key: "essu_XXXX"

# Where we'll read the monitoring data from
monitoring_source:
hosts: 'https://where-i-want-to-get-the-data:443'
  api_key: 'ZZZZZZZZZZ'
  retry_on_timeout: true
  request_timeout: 60

# Where the consumption data (result of the script's run) will be sent
# This can be the same cluster as the source.
consumption_destination:
  hosts: 'https://where-i-want-the-data-to-be-indexed:443'
  api_key: 'YYYYYYYYYYYY'

# For on-premises deployment, you'll need to specify the costs of your tier per
1 GB RAM per hour
# on_prem_costs:
#   hot: 1.0
#   warm: 0.5
#   cold: 0.25
#   frozen: 0.1
```

```
# Uncomment and modify this if you want to read indices other than the default
monitoring ones
# (for example using cross-cluster search)
# monitoring_index_pattern: '.monitoring-es-8*'

# Uncomment and modify this for Federal customers using a non-standard ESS
endpoint
# api_host: 'api.elastic-cloud.com'
```

You will find more details about each configuration parameter and its usage context in the section [Configuration parameters](#).

## Command line [Configuration parameters](#)

A few command line flags are available to alter the behavior of the script.








Generally speaking, you will run the command as follows:

```
python main.py [command] [command line flags]
```

With [command] being one of:

- `init`
- `get-billing-data`
- `consume-monitoring`

The table below presents the most common flags and describes their usage and impact.

Parameter	Description	init	get-billing-data	consume-monitoring
<code>--config-file TEXT</code>	See dedicated <a href="#">Configuration definition</a> section down below.			
<code>--inline-config TEXT</code>				
<code>--config TEXT</code>				
<code>--lookbehind INTEGER</code>	The amount of hours in the past to process, from current full hour. Defaults to 24.			
<code>--threads INTEGER</code>	Control the amount of parallel threads to use to process billing / monitoring			

	information. Defaults to 5.			
<b>--force</b>	Re-compute all data in the <code>lookbehind</code> scope. By default the script only processes missing chunks, rather than the full range. No duplicate information is generated, thanks to document ID control.	✗	✓	✓
<b>--debug</b>	Output debug information.	✓	✓	✓
<b>--vpn</b>	Retrieve billing information through Elastic VPN. This only makes sense for Elastic employees with an admin API key.	✗	✓	✓
<b>--help</b>	Display a help message.	✓	✓	✓

## Configuration definition

The recommended approach to pass configuration to the script is to write a `conf.yml` file containing the required parameters, and pass this file to the script using the `--config-file=conf.yml` flag.

Depending on your deployment means, it is sometimes not possible or advised to create a local configuration file with sensitive information.

In that case, you can either convert the yaml file to JSON and pass the entire string inline using the `--inline-config={ [ . . . ] }` flag, or pass individual sensitive values with the `--config key=val` flag (you can pass multiple of these).

Parameter	<code>--config-file=X.yml</code>	<code>--inline-config={Y}</code>	<code>--config foo=bar</code>
Format	YAML file	Inline JSON	key/value pairs
Description	Recommended default approach. Create a local configuration and pass it to the script	Pass the entire configuration as a JSON string inline to the script. When set, this takes precedence over the <code>--config-file</code> parameter.	Override of individual keys. This is useful to pass "secrets" in conjunction with the config file. This will override any configuration key passed with the other two options.

## Configuration parameters

The table below presents what each configuration parameter is, and in what context / execution mode it can be used.

▲ Don't mistake these parameters for the command line flags that can be given to the script. These are covered in the section [Command line](#).

Parameter	Explanation	On-prem	ESS
<code>organization_name</code>	This is the Elastic Cloud Organization Name. This is also required for on-prem clusters, for compatibility reasons.	✓	✓
<code>organization_id</code>	This is the Elastic Cloud Organization ID	✗	✓
<code>billing_api_key</code>	This is an Elastic Cloud API key with <b>Billing Admin</b> rights. You can create one <a href="#">here</a> .	✗	✓
<code>monitoring_source.cloud_id</code>	This is the Cloud ID of the monitoring cluster.	✗	✓
<code>monitoring_source.hosts</code>	The endpoint protocol, hostname and port of the monitoring cluster (e.g. <code>https://hostname:443</code> )	✓	✓
<code>monitoring_source.api_key</code>	This API key is used to read from the monitoring cluster; it requires <code>read</code> privileges to the <code>.monitoring-es-8*</code> index names.	✓	✓
<code>monitoring_source.*</code>	You can pass <a href="#">additional parameters to the Elasticsearch python client</a> by specifying these under the <code>monitoring_source</code> namespace.	✓	✓
<code>consumption_destination.cloud_id</code>	This is the Cloud ID of the cluster to store the Consumption Framework data; dashboards will be available for Kibana in this cluster.	✗	✓
<code>consumption_destination.hosts</code>	The endpoint protocol, hostname and port of the cluster to store the CF data (e.g. <code>https://hostname:443</code> ) This can be used in place of the <code>cloud_id</code> for on-prem clusters, or in ESS if you don't want to use the <code>cloud_id</code> .	✓	✓
<code>consumption_destination.api_key</code>	This API key is used to write to the target cluster where the Consumption Framework data will be stored; it requires <code>read</code> , <code>view_index_metadata</code> , <code>index</code> and <code>auto_configure</code> privileges to the <code>consumption*</code> index pattern, as well as <code>manage_ingest_pipelines</code> , <code>manage_ilm</code> and <code>manage_index_templates</code> cluster privileges.	✓	✓
<code>consumption_destination.*</code>	You can pass <a href="#">additional parameters to the</a>	✓	✓



	<a href="#">Elasticsearch python client</a> by specifying these under the <code>consumption_destination</code> namespace.		
<code>monitoring_index_pattern</code>	This allows you to override the index pattern of the source monitoring data. This is useful in CCS setups if you want to use a single endpoint for multiple backing monitoring clusters.	✓	✓
<code>api_host</code>	This allows you to override the standard <code>api.elastic-cloud.com</code> endpoint used to communicate with ESS API. This is useful for private cloud (Federal) customers.	✗	✓
<code>on_prem_costs</code>	For on-premises deployment, this is required to define your cost basis for each data tier, under the format: <code>on_prem_costs:</code> <code>hot: 0.0532</code> <code>warm: 0.0681</code> <code>cold: 0.0681</code> <code>frozen: 0.0547</code> Where each number represents a price per GB of RAM per hour.	✓	✗

## API keys permissions

The API keys for the monitoring cluster and the consumption cluster (the destination of the Consumption Framework data) can be created using API calls with Dev Tools in Kibana. In the configuration file, the **encoded** version of the API keys should be used.

To create the consumption cluster API key, execute the following API call on the consumption cluster (the cluster where the CF results should be sent):

```
Unset
POST /_security/api_key
{
  "name": "consumption_framework_destination",
  "role_descriptors": {
    "consumption_framework": {
      "indices": [
        {
          "names": [
            "consumption*"
          ],

```

```

        "privileges": [
            "read",
            "view_index_metadata",
            "index",
            "auto_configure"
        ]
    },
],
"cluster": [
    "manage_ingest_pipelines",
    "manage_ilm",
    "manage_index_templates",
    "monitor"
]
}
}
}
}

```

To create the API key for the monitoring cluster, execute the following API call on the monitoring cluster:

```

Unset
POST /_security/api_key
{
  "name": "consumption_framework_source",
  "role_descriptors": {
    "consumption_framework": {
      "indices": [
        {
          "names": [
            ".monitoring-es-8*"
          ],
          "privileges": [
            "read"
          ]
        }
      ]
    }
  }
}
}
}
}

```

## Migrating from v1

The active version of the consumption framework is version 2, which was released in early May 2024. It uses the “new” billing API exposed by Elastic which is more efficient than the old one, but gives slightly different results. As a result, it is necessary to run a few commands to make the v1 data compatible with v2 dashboards. The below steps describe how to achieve this.

### Important notes:

- This makes v1 data compatible with v2 dashboards and not the other way around.
- If you customized the dashboards, you will need to apply these modifications to the new dashboards as well.
- The largest change affects the organization overview tab (which is effectively fully populated by the billing data). For the other tabs, you only need to verify that the aliases are set properly (which is the case on late v1 versions of the CF).

### Re-init

Regardless of your setup, and whether you use Elastic billing API, make sure you re-run the init command of the CF to update the index templates and ingest pipelines to the latest versions.

### Aliases modification

The below commands will make sure that all **consumption** indices have the proper aliases set. This is necessary for late v1 and v2 dashboards that rely on specific subsets of the consumption indices to display data properly.

```
Unset
POST /_aliases
{
  "actions": [
    {
      "add": {
        "index": "consumption-*",
        "alias": "consumption-deployment",
        "filter": {
          "term": {
            "dataset": "deployment"
          }
        }
      }
    }
  ]
},
```

```

{
  "add": {
    "index": "consumption-*",
    "alias": "consumption-node",
    "filter": {
      "term": {
        "dataset": "node"
      }
    }
  }
},
{
  "add": {
    "index": "consumption-*",
    "alias": "consumption-datastream",
    "filter": {
      "term": {
        "dataset": "datastream"
      }
    }
  }
},
{
  "add": {
    "index": "consumption-*",
    "alias": "consumption-datastream_usage",
    "filter": {
      "term": {
        "dataset": "datastream_usage"
      }
    }
  }
}
]
}

```

## Re-feeding billing data

▲ This step is only necessary for ESS deployments using the **get-billing-data** command.

As the format changes, the easiest way to get “properly formatted v2 billing data is to simply delete the existing data and feed it back:

Unset

```
POST consumption-*/_delete_by_query?wait_for_completion=false
```

```
{
  "query": {
    "term": {
      "dataset": "deployment"
    }
  }
}
```

```
GET /_tasks/[TASK_ID]
```

## Resources

The CF is a lightweight script that doesn't require much client-side resources. Less than 4 GB of RAM and 2-4 CPU cores are enough for the script to run. Most of the load will be created server-side on the monitoring cluster, as the script relies on some extensive composite aggregations running against the monitoring indices.