Simple way to deploy machine learning models to cloud

Deploy your first ML model to production with a simple tech stack

Tanuj Jain Follow
Apr 14, 2019 · 11 min read

Photo by Randy Fath on Unsplash

The Machine learning world currently sees Data Scientists (DS) performing one or both of the following 2 prominent roles:

1. Where a DS receives a data dump, applies some Machine learning algo on the data and reports back the results in the form of some presentation or report.

2. Where the DS creates a usable piece of software for the stakeholders to consume the machine learning models.

In this blog post, I'm attempting to display an example approach to the second aspect of a DS's job i.e., creating some software that can be used by the stakeholders. Specifically, we would create a web-service that can be queried to obtain the predictions from a machine learning model. The post is mostly intended for machine learning practitioners who would like to go beyond only developing models.

Tech-stack: Python, Flask, Docker, AWS ec2

The workflow can be broken down into following basic steps:

1. Training a machine learning model on a local system.

2. Wrapping the inference logic into a flask application.

3. Using docker to containerize the flask application.

4. Hosting the docker container on an AWS ec2 instance and consuming the web-service.

DISCLAIMER: The system presented here is light years away from what a commercial production system should look like. The key takeaways from this blogpost should be the development workflow, acquaintance with the tech stack and getting the first taste of building an ML production system.

Let's start with the first step.

Training a machine learning model on a local system

We need **some** machine learning model that we can wrap in a web-service. For demo purpose, I chose a logistic regression model to do multiclass classification on iris dataset (Yep, super easy! #LazinessFTW). The model was trained on a local system using python 3.6.

Using the familiar scikit-learn, the above mentioned model can be trained quickly. For model development, refer the notebook 'Model_training.ipynb' in the github repo for this blog. There are only 2 important aspects of model development that I would like to highlight:

1. The model file generated after training is stored as a pickle file which is a serialized format for storing objects. (In the repo, the file is named 'iris_trained_model.pkl')
2. The inference call (.predict()) call requires 4 features per test sample in the form of a numpy array.

Wrapping the inference logic into a flask web service

Now that we have the trained model file, we are ready to query the model to get a class label for a test sample. The inference is as simple as calling a predict() function on the trained model with the test data. However, we would like to build the inference as a web-service. For this purpose, we would use Flask.

Flask is a powerful python microwebserver framework that allows us to build REST API based web-services quickly with minimum configuration hassle. Let's dive into the code:

a. First, let's define a simple function to load the trained model file.

```python
model = None

def load_model():
    global model
    # model variable refers to the global variable
    with open('iris_trained_model.pkl', 'rb') as f:
        model = pickle.load(f)
```

load_model.py hosted with ♡ by GitHub view raw

Here, we define a global variable called 'model' and populate it within the load_model() function. The purpose of using a global variable will become clear shortly.

b. Next, we instantiate a Flask object called 'app':

```python
app = Flask(__name__)
```

initialize_flask.py hosted with ♡ by GitHub view raw

c. Now, we define a home endpoint, which when hit, returns a 'Hello World!' message.

```python
@app.route('/')
def home_endpoint():
    return 'Hello World!'
```

home_endpoint.py hosted with ♡ by GitHub view raw

Notice the use of app.route decorator.

d. Now, we define a 'predict' endpoint. The endpoint accepts a 'POST' request wherein the test data on which we wish to get a prediction is received by the endpoint. Keeping things simple, the function works only when a single test sample needs to be predicted (won't work if multiple samples need to be predicted in a single call to the endpoint).

```python
@app.route('/predict', methods=['POST'])
def get_prediction():
    # Works only for a single sample
    if request.method == 'POST':
        data = request.get_json()  # Get data posted as a json
        data = np.array(data)[np.newaxis, :]  # converts shape from (4,) to (1, 4)
        prediction = model.predict(data)  # runs globally loaded model on the data
    return str(prediction[0])
```

predict_endpoint.py hosted with ♡ by GitHub view raw

Notice the direct call to the predict function through the 'model' variable.

e. Finally, declare the main function:

```python
if __name__ == '__main__':
    load_model()  # load model at the beginning once only
    app.run(host='0.0.0.0', port=80)
```

main.py hosted with ♡ by GitHub view raw

Here, a call to the load_model() function ensures that the variable 'model' is populated with the trained model attributes (and hence the need for a global model variable). So, there is no need to load the model repeatedly with every call to the predict endpoint. This allows the web-service to be quick. The response is returned as a string which is the predicted class label.

The complete flask specific code is as below:

```python
# Serve model as a flask application

import pickle
import numpy as np
from flask import Flask, request

model = None
app = Flask(__name__)


def load_model():
    global model
    # model variable refers to the global variable
    with open('iris_trained_model.pkl', 'rb') as f:
        model = pickle.load(f)


@app.route('/')
def home_endpoint():
    return 'Hello World!'


@app.route('/predict', methods=['POST'])
def get_prediction():
    # Works only for a single sample
    if request.method == 'POST':
        data = request.get_json()  # Get data posted as a json
        data = np.array(data)[np.newaxis, :]  # converts shape from (4,) to (1, 4)
        prediction = model.predict(data)  # runs globally loaded model on the data
    return str(prediction[0])


if __name__ == '__main__':
    load_model()  # load model at the beginning once only
    app.run(host='0.0.0.0', port=80)
```

flask_code.py hosted with ♡ by GitHub view raw

At this point, the web-service is ready to be run locally. Let's test this.

Execute the command `python app.py` from the terminal. Go to the browser and hit the url `0.0.0.0:80` to get a message `Hello World!` displayed. This corresponds to the home endpoint return message.

NOTE: A permission error may be received at this point. In this case, change the port number to 5000 in `app.run()` command in `app.py`. (Port 80 is a privileged port, so change it to some port that isn't, eg: 5000)

Next, let's test if we can get predictions using this web-service using the following curl post request on the terminal:

```
curl -X POST \
   0.0.0.0:80/predict \
   -H 'Content-Type: application/json' \
   -d '[5.9,3.0,5.1,1.8]'
```

The curl request posts one test sample `[5.9,3.0,5.1,1.8]` to our web-server and returns a single class label.

Using docker to containerize the flask service

Up to this point, we have a web-service that runs locally. Our ultimate intention is to be able to run this piece of code on a cloud virtual machine.

In the Software Development world, there is a famous justification given by a developer whose code was found to be broken by a tester: '*But it worked on my machine!*'. The problem portrayed here can usually be attributed to a lack of consistent environment that runs the software across different machines. Ideally, our code itself should be independent of the underlying machine/OS that runs it. Containerization allows developers to provide such isolation.

How is it important here?

Our intention is to run our web-service on a cloud VM. The cloud VM itself may run any OS. Containerization of our web-server allows us to avoid the trouble of running into environment related issues. If the containerized code works on one machine, it will surely run on another irrespective of the characteristics of the machine. Docker is the most famous containerized technology out there at this point and we will be using the same here. For a quick tutorial on docker, check this link.

Let's dive into the Dockerfile that comprises a set of instructions for docker daemon to build the docker image.

```
FROM python:3.6-slim
COPY ./app.py /deploy/
COPY ./requirements.txt /deploy/
COPY ./iris_trained_model.pkl /deploy/
WORKDIR /deploy/
RUN pip install -r requirements.txt
EXPOSE 80
ENTRYPOINT ["python", "app.py"]
```

Dockerfile hosted with ♡ by GitHub view raw

We pull the base docker image from python dockerhub repo on which our specific build instructions are executed. The COPY commands are simply taking specific files from the current folder and copying them over to a folder called 'deploy' within the docker image we are trying to build. In addition to app.py and model file, we also need a requirements file that lists specific versions of python packages we use to run our code. The WORKDIR command changes the working directory to 'deploy/' within the image. We then issue a RUN command to install specific python packages using the requirements file. The EXPOSE command makes the port 80 accessible to the outside world (our flask service runs on port 80; we need this port inside the container to be accessible outside the container).

Issue the build command to end up with a docker image:

```
docker build -t app-iris .
```

(Don't forget the period at the end of the command).

Use command 'docker images' to see a docker image with a docker repository named 'app-iris' created. (Another repository named python will also be seen since it is the base image on top of which we build our custom image.)

Now, the image is built and ready to be run. We can do this using the command:

```
docker run –p 80:80 app–iris .
```

The above commands uses -p flag to map port 80 of the local system to the port 80 of the docker container for the redirection of traffic on local HTTP port 80 to port 80 of the container. (If you are using local port 5000 instead of port 80, change the port mapping part of the command to 5000:80).

Let's test if this works by hitting the URL: http://0.0.0.0:80 on the browser which should display 'Hello World!' which is the home endpoint output message (If port 5000 is used, modify the http port to 5000 in the url). Also, use the curl request mentioned earlier to check if the predicted class label is returned.

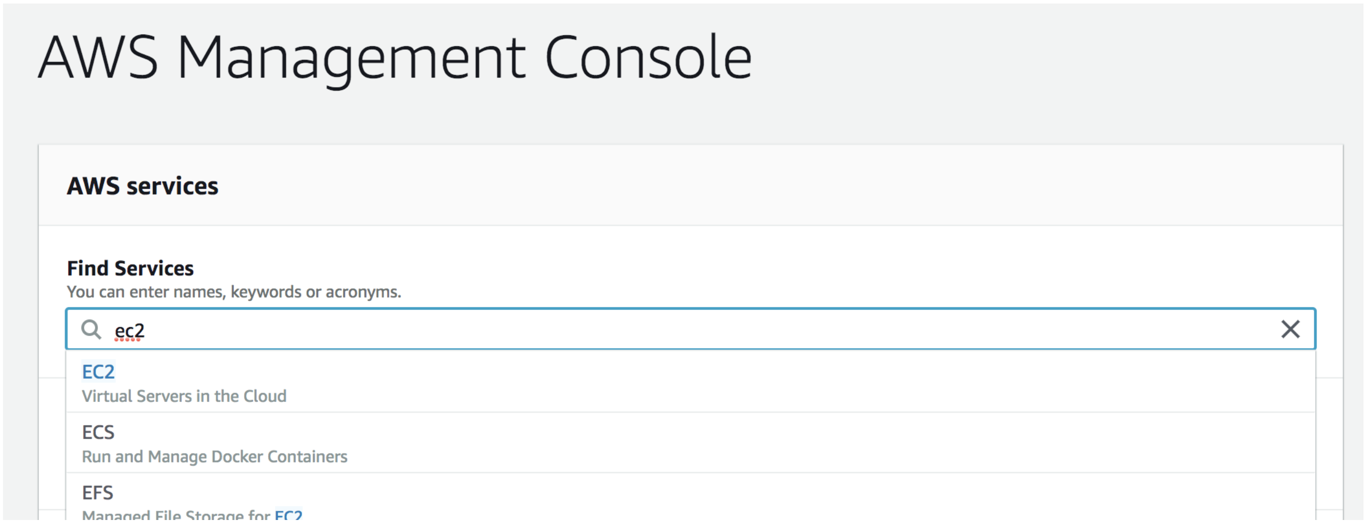
Hosting the docker container on an AWS ec2 instance

We already have a containerized application that works on our local system. Now, what if someone else wishes to consume the service? What happens if we need to build an architectural ecosystem around the service that needs to be available, automated and scalable? It's easy to see that having a web-service running locally would be a very bad idea. So, we wish to host the web-service somewhere on the internet to fulfil the requirements we listed. For this blog, we choose to host our service on an AWS ec2 instance.

As a prerequisite, one needs to have an AWS account for using the ec2 instance. For new users, there are several AWS resources that are available for free for a period of 1 year (usually up to some limit). In this blog, I would be using a '*t2.micro*' ec2 instance type which is free tier eligible. For users who have exhausted their AWS free-tier period, this instance costs around 1 cent(USD) per hour at the time of writing this blog; a super negligible amount to pay.

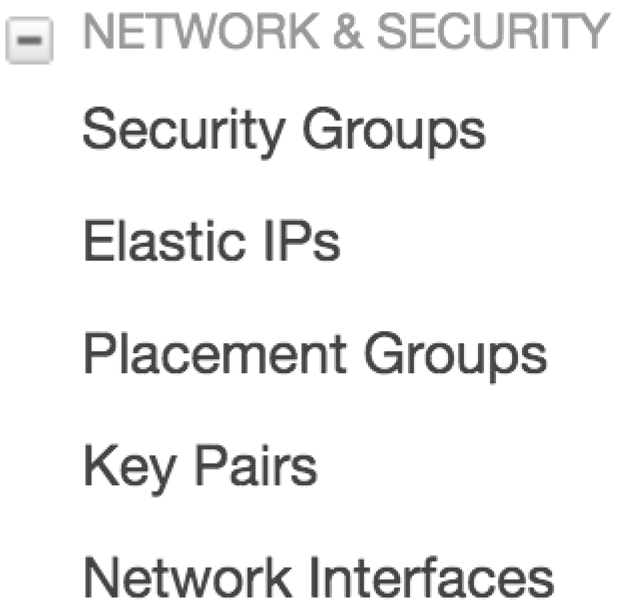
Let's start with the process.

Log into the AWS management console and search for ec2 in the search bar to navigate to EC2 dashboard.


AWS Management Console
AWS services
Find Services
You can enter names, keywords or acronyms.
ec2
EC2
Virtual Servers in the Cloud
ECS
Run and Manage Docker Containers
EFS
Managed File Storage for EC2

Search for ec2 on the aws management console

Look for the below pane, select 'Key Pairs' and create one.



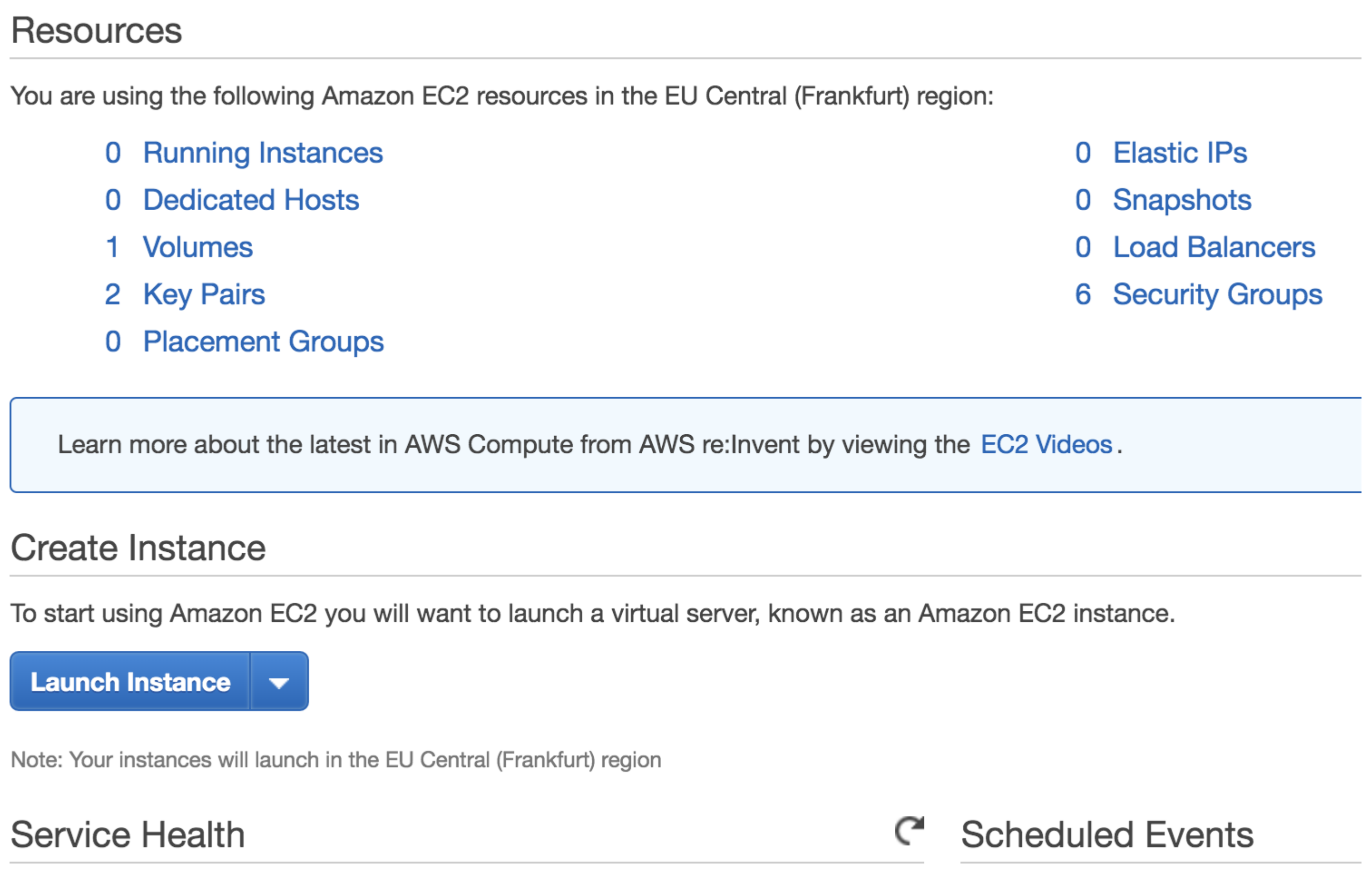
Select Key Pairs for looking at existing key pairs and creating new ones

This will download a '.pem' file that is the key. Save this file somewhere safely. Now navigate to the location of this file on your system and issue the below command with key file name replaced by yours:

```
chmod 400 key-file-name.pem
```

This commands changes permissions on your key pair file to private. The use of key pairs will be explained later.

Next, click ‘Launch Instance’ on the EC2 dashboard:



Resources

You are using the following Amazon EC2 resources in the EU Central (Frankfurt) region:

0 Running Instances	0 Elastic IPs
0 Dedicated Hosts	0 Snapshots
1 Volumes	0 Load Balancers
2 Key Pairs	6 Security Groups
0 Placement Groups	

Learn more about the latest in AWS Compute from AWS re:Invent by viewing the EC2 Videos.

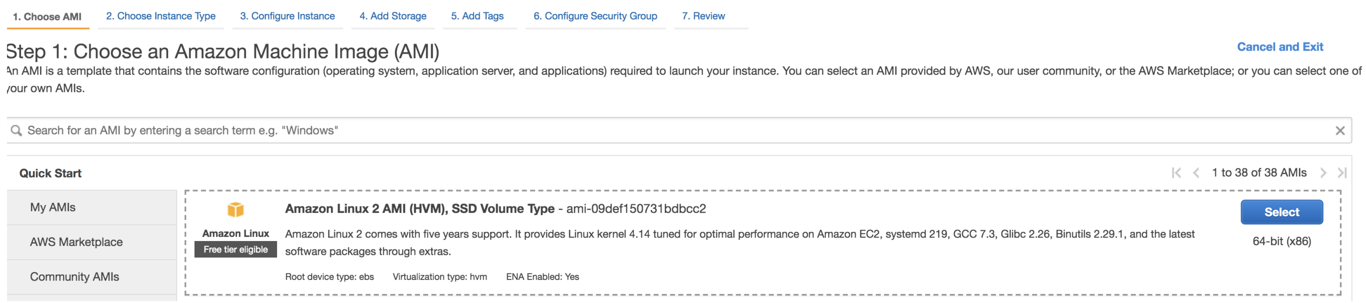
Create Instance

To start using Amazon EC2 you will want to launch a virtual server, known as an Amazon EC2 instance.

Note: Your instances will launch in the EU Central (Frankfurt) region

Launch ec2 instance

Choose the Amazon Machine Instance (AMI) from the list of options. An AMI determines the OS that the VM will be running (plus some other stuff we don’t care about at this point). For this blog, I chose ‘Amazon Linux 2 AMI’ which was the default selection.



Step 1: Choose an Amazon Machine Image (AMI)

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. You can select an AMI provided by AWS, our user community, or the AWS Marketplace; or you can select one of your own AMIs.

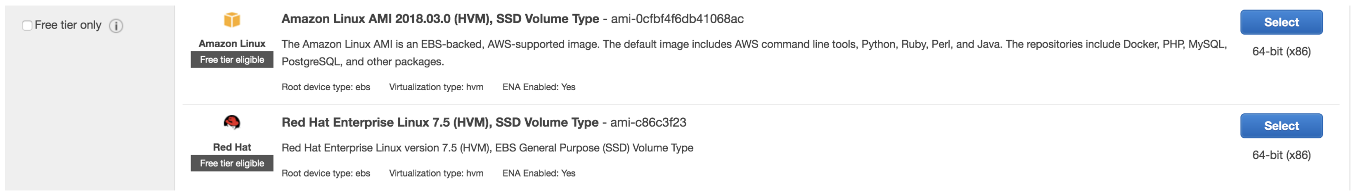
Search for an AMI by entering a search term e.g. "Windows"

Amazon Linux 2 AMI (HVM), SSD Volume Type - ami-09def150731bdbcc2

Amazon Linux
Free tier eligible

Amazon Linux 2 comes with five years support. It provides Linux kernel 4.14 tuned for optimal performance on Amazon EC2, systemd 219, GCC 7.3, Glibc 2.26, Binutils 2.29.1, and the latest software packages through extras.

Root device type: ebs Virtualization type: hvm ENA Enabled: Yes

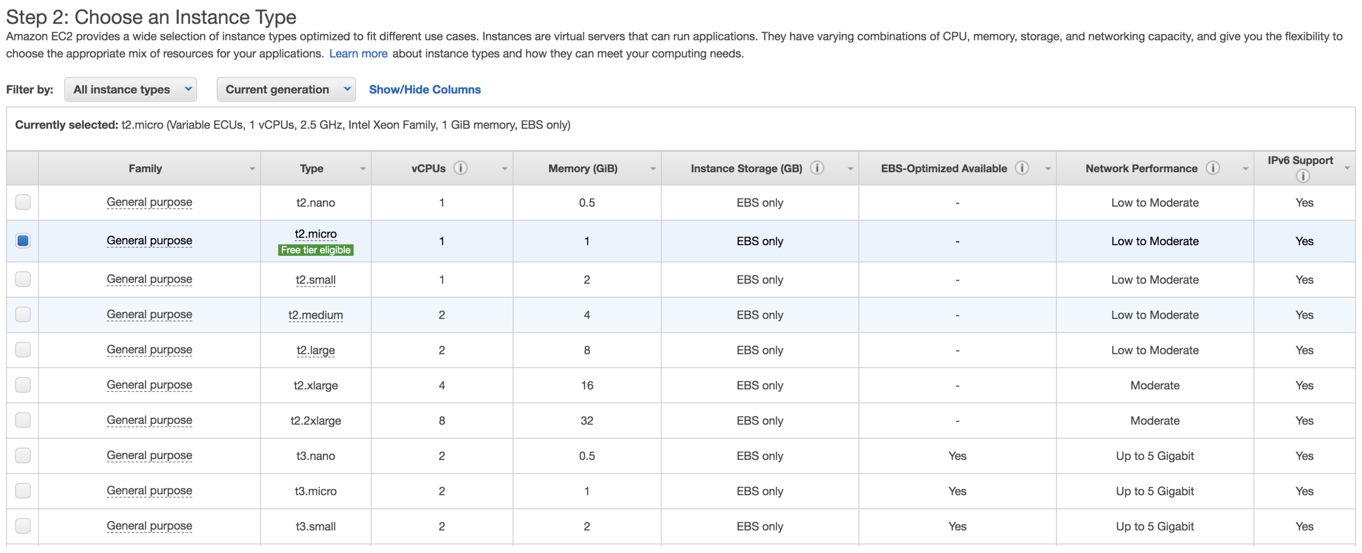


☐ Free tier only ⓘ

Amazon Linux Free tier eligible	**Amazon Linux AMI 2018.03.0 (HVM), SSD Volume Type** - ami-0cfbf4f6db41068ac The Amazon Linux AMI is an EBS-backed, AWS-supported image. The default image includes AWS command line tools, Python, Ruby, Perl, and Java. The repositories include Docker, PHP, MySQL, PostgreSQL, and other packages. Root device type: ebs Virtualization type: hvm ENA Enabled: Yes	Select	64-bit (x86)
Red Hat Free tier eligible	**Red Hat Enterprise Linux 7.5 (HVM), SSD Volume Type** - ami-c86c3f23 Red Hat Enterprise Linux version 7.5 (HVM), EBS General Purpose (SSD) Volume Type Root device type: ebs Virtualization type: hvm ENA Enabled: Yes	Select	64-bit (x86)

Choosing AMI

The next screen allows you to select the instance type. This is where the hardware part of the VM can be selected. As mentioned previously, we will work with ‘t2.micro’ instance.



Step 2: Choose an Instance Type

Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. Instances are virtual servers that can run applications. They have varying combinations of CPU, memory, storage, and networking capacity, and give you the flexibility to choose the appropriate mix of resources for your applications. Learn more about instance types and how they can meet your computing needs.

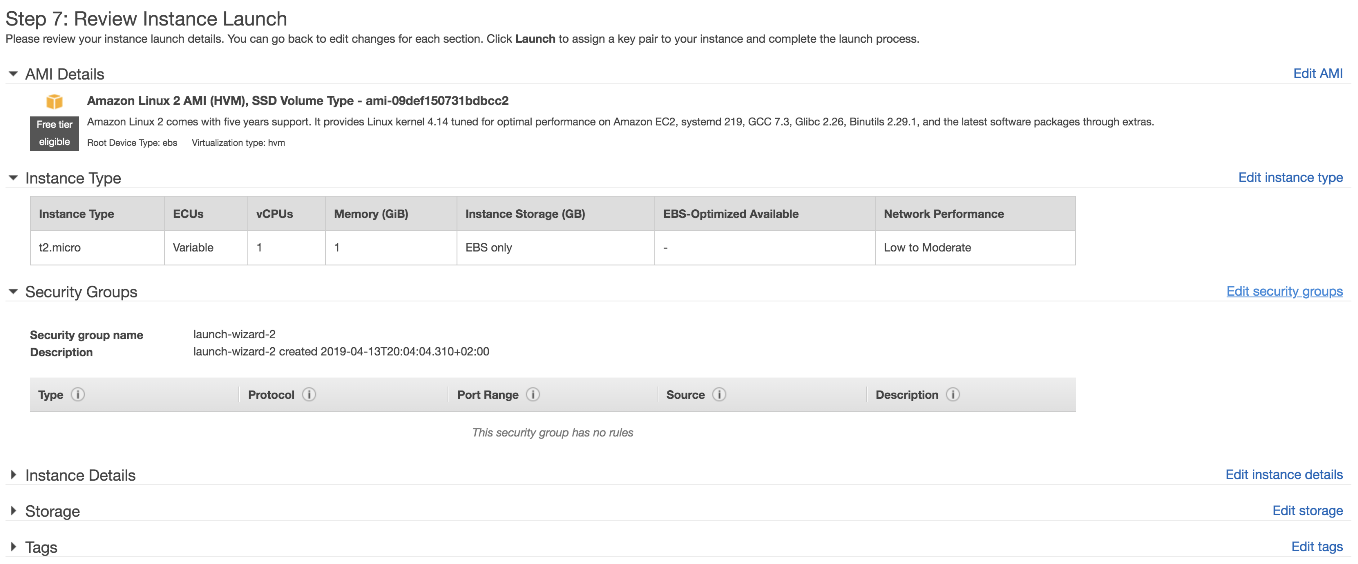
Filter by: All instance types | Current generation | Show/Hide Columns

Currently selected: t2.micro (Variable ECUs, 1 vCPUs, 2.5 GHz, Intel Xeon Family, 1 GiB memory, EBS only)

	Family	Type	vCPUs ⓘ	Memory (GiB)	Instance Storage (GB) ⓘ	EBS-Optimized Available ⓘ	Network Performance ⓘ	IPv6 Support ⓘ
☐	General purpose	t2.nano	1	0.5	EBS only	-	Low to Moderate	Yes
☑	General purpose	t2.micro Free tier eligible	1	1	EBS only	-	Low to Moderate	Yes
☐	General purpose	t2.small	1	2	EBS only	-	Low to Moderate	Yes
☐	General purpose	t2.medium	2	4	EBS only	-	Low to Moderate	Yes
☐	General purpose	t2.large	2	8	EBS only	-	Low to Moderate	Yes
☐	General purpose	t2.xlarge	4	16	EBS only	-	Moderate	Yes
☐	General purpose	t2.2xlarge	8	32	EBS only	-	Moderate	Yes
☐	General purpose	t3.nano	2	0.5	EBS only	Yes	Up to 5 Gigabit	Yes
☐	General purpose	t3.micro	2	1	EBS only	Yes	Up to 5 Gigabit	Yes
☐	General purpose	t3.small	2	2	EBS only	Yes	Up to 5 Gigabit	Yes

Selecting instance type

You can select ‘Review and Launch’ that takes you to ‘Step 7: Review Instance Launch’ screen. Here, you need to click the ‘Edit Security Groups’ link:



Step 7: Review Instance Launch

Please review your instance launch details. You can go back to edit changes for each section. Click **Launch** to assign a key pair to your instance and complete the launch process.

▾ AMI Details — Edit AMI

Amazon Linux 2 AMI (HVM), SSD Volume Type - ami-09def150731bdbcc2

Free tier eligible

Amazon Linux 2 comes with five years support. It provides Linux kernel 4.14 tuned for optimal performance on Amazon EC2, systemd 219, GCC 7.3, Glibc 2.26, Binutils 2.29.1, and the latest software packages through extras.

Root Device Type: ebs Virtualization type: hvm

▾ Instance Type — Edit instance type

Instance Type	ECUs	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance
t2.micro	Variable	1	1	EBS only	-	Low to Moderate

▾ Security Groups — Edit security groups

Security group name	launch-wizard-2
Description	launch-wizard-2 created 2019-04-13T20:04:04.310+02:00

Type ⓘ	Protocol ⓘ	Port Range ⓘ	Source ⓘ	Description ⓘ

This security group has no rules

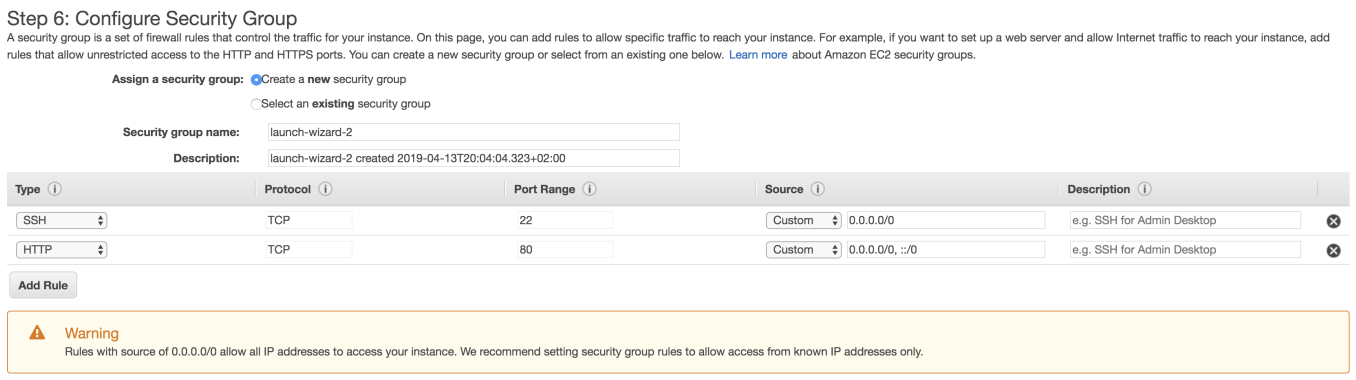
▸ Instance Details — Edit instance details

▸ Storage — Edit storage

▸ Tags — Edit tags

Security Groups

You now have to modify the security group to allow HTTP traffic on port 80 of your instance to be accessible by the outside world. This can be done by creating a rule. At the end, you should end up with such a screen:



Step 6: Configure Security Group

A security group is a set of firewall rules that control the traffic for your instance. On this page, you can add rules to allow specific traffic to reach your instance. For example, if you want to set up a web server and allow Internet traffic to reach your instance, add rules that allow unrestricted access to the HTTP and HTTPS ports. You can create a new security group or select from an existing one below. Learn more about Amazon EC2 security groups.

Assign a security group: [✓ Create a **new** security group] Select an **existing** security group

Security group name: launch-wizard-2

Description: launch-wizard-2 created 2019-04-13T20:04:04.323+02:00

Type	Protocol	Port Range	Source		Description
SSH	TCP	22	Custom	0.0.0.0/0	e.g. SSH for Admin Desktop
HTTP	TCP	80	Custom	0.0.0.0/0, ::/0	e.g. SSH for Admin Desktop

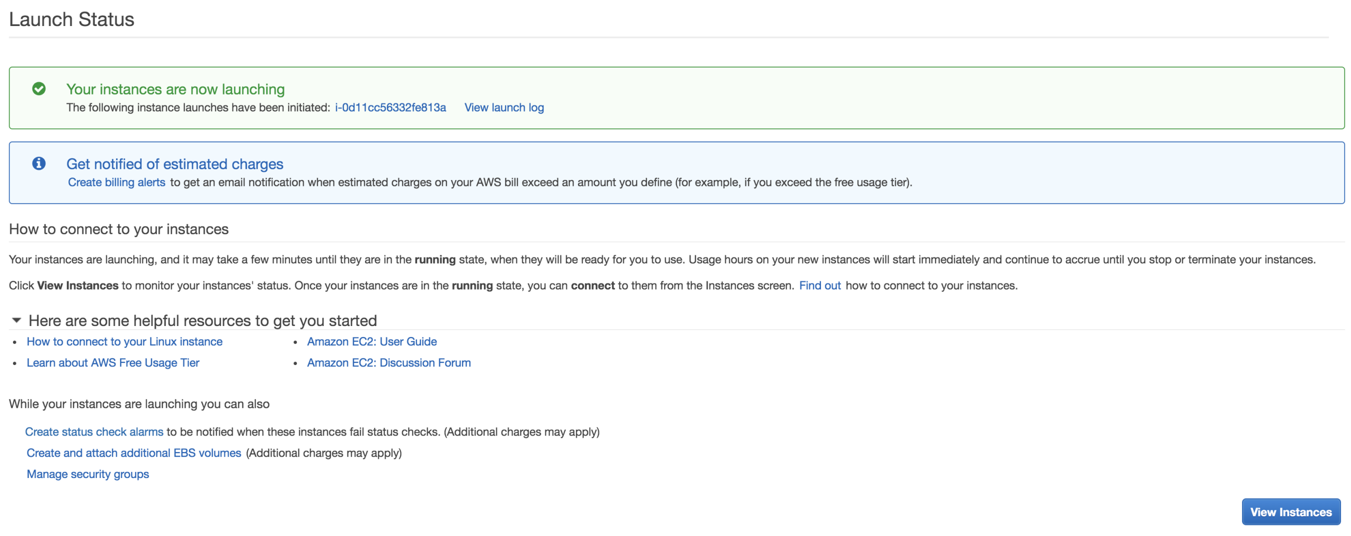
Warning

Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only.

Adding HTTP rule to security group

In the absence of this rule, your web-service will never be reachable. For more on security groups and configuration, refer AWS documentation. Clicking on the 'Launch' icon will lead to a pop up seeking a confirmation on having a key-pair. Use the name of the key pair that was generated earlier and launch the VM.

You would be redirected to a Launch screen:



Launch Status

Your instances are now launching
The following instance launches have been initiated: i-0d11cc56332fe813a View launch log

Get notified of estimated charges
Create billing alerts to get an email notification when estimated charges on your AWS bill exceed an amount you define (for example, if you exceed the free usage tier).

How to connect to your instances

Your instances are launching, and it may take a few minutes until they are in the **running** state, when they will be ready for you to use. Usage hours on your new instances will start immediately and continue to accrue until you stop or terminate your instances.

Click **View Instances** to monitor your instances' status. Once your instances are in the **running** state, you can **connect** to them from the Instances screen. Find out how to connect to your instances.

Here are some helpful resources to get you started

- How to connect to your Linux instance
- Learn about AWS Free Usage Tier
- Amazon EC2: User Guide
- Amazon EC2: Discussion Forum

While your instances are launching you can also

Create status check alarms to be notified when these instances fail status checks. (Additional charges may apply)
Create and attach additional EBS volumes (Additional charges may apply)
Manage security groups

Launch Status of ec2 instance

Use the ‘View Instance’ button to navigate to a screen that displays the ec2 instance being launched. When the instance state turns to ‘running’, then it is ready to be used.

We will now ssh into the ec2 machine from our local system terminal using the command with the field public-dns-name replaced with your ec2 instance name (of the form: ec2–x–x–x–x.compute-1.amazonaws.com) and the path of the key pair pem file you saved earlier.

```
ssh -i /path/my-key-pair.pem ec2-user@public-dns-name
```

This will get us into the prompt of our instance where we’ll first install docker. This is required for our workflow since we will build the docker image within the ec2 instance (There are better, but slightly complicated alternatives to this step). For the AMI we selected, the following bunch of commands can be used:

```
sudo amazon-linux-extras install docker
sudo yum install docker
sudo service docker start
sudo usermod -a -G docker ec2-user
```

For an explanation of the commands, check the documentation.

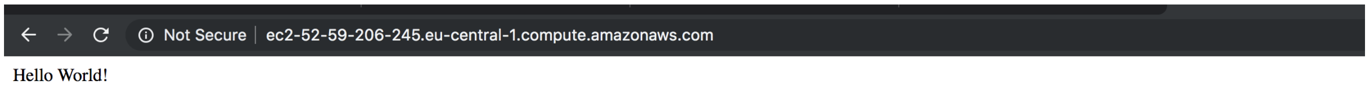
Log out of the ec2 instance using the ‘exit’ command and log back in again using the ssh command. Check if docker works by issuing the ‘docker info’ command. Log out again or open another terminal window.

Now let’s copy the files we need to build the docker image within the ec2 instance. Issue the command from your local terminal (not from within ec2):

```
scp -i /path/my-key-pair.pem file-to-copy ec2-user@public-dns-
name:/home/ec2-user
```

We would need to copy requirements.txt, app.py, trained model file and Dockerfile to build the docker image as was done earlier. Log back into the ec2 instance and issue '*ls*' command to see if the copied files exist. Next, build and run the docker image using the exact same commands that were used in the local system (Use port 80 at all locations in the code/commands this time).

Hit the home endpoint from your browser using the public dns name to see the familiar '*Hello World!*' message:


Not Secure | ec2-52-59-206-245.eu-central-1.compute.amazonaws.com

Hello World!

Home endpoint works from the browser (I used my ec2 public-dns-name in the address bar)

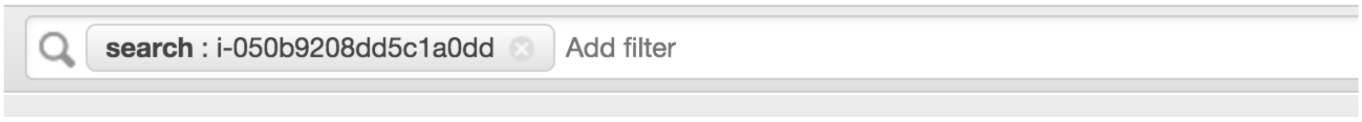
Now send a curl request to your web-service from local terminal with your test sample data after replacing the public-dns-name by yours:

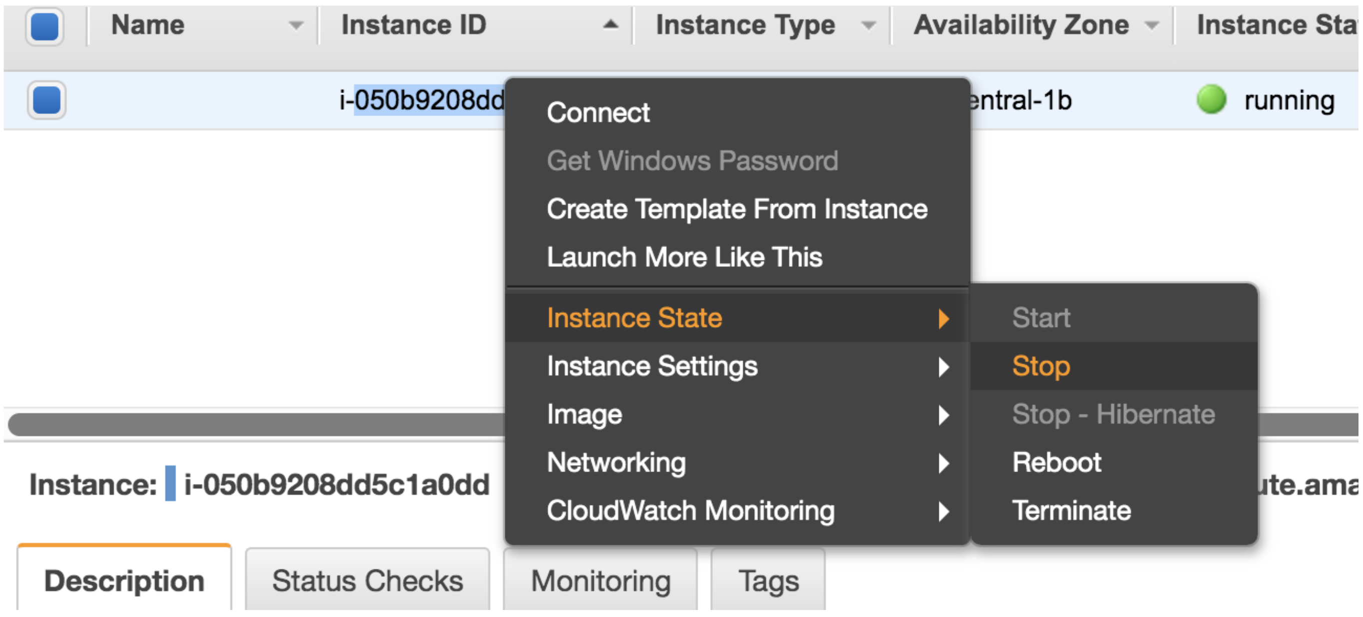
```
curl -X POST \
public-dns-name:80/predict \
-H 'Content-Type: application/json' \
-d '[5.9,3.0,5.1,1.8]'
```

This should get you the same predicted class label as the one you got locally.

And you are done! You can now share this curl request with anyone who wishes to consume your web-service with their test samples.

When you no longer need the web-service, **do not forget to stop or terminate the ec2 instance**:





Stop or terminate the ec2 instance to avoid getting charged

Some additional thoughts

This is a super basic workflow intended for ML practitioners itching to go beyond model development. A huge number of things need to be changed to make this system into one that is more suited to a real production system. Some suggestions (far from complete):

1. Use a Web Server Gateway Interface (WSGI) such as gunicorn in the flask app. Bonus points for using nginx reverse proxy and async workers.

2. Improve security of the ec2 instance: The service is currently open to the entire world. Suggestions: Restrict access to a set of IPs.

3. Write test cases for the app: Software without testing = Shooting yourself in the leg and then throwing yourself in a cage full of hungry lions all the while being pelted with stones. (Moral: Do not float a production software without thoroughly testing first)

A lot more could be added to the above list, but maybe that's something for another blog post.

Github repo: https://github.com/tanujjain/deploy-ml-model

It would be great to hear your feedback!

Docker Machine Learning AWS Data Science Artificial Intelligence