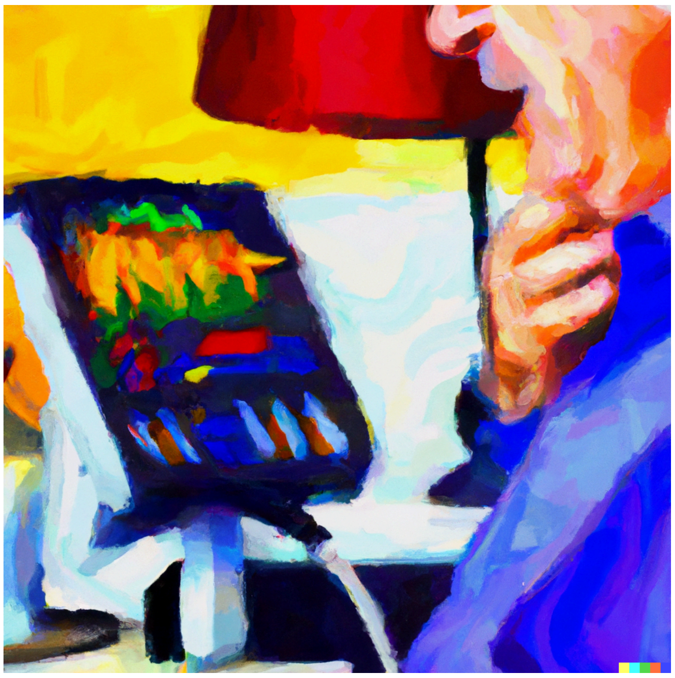
Machine Learning Fast Demo with Gradio and IBM Code Engine

Estimated time needed: 65 minutes

Do you want to deploy a serverless AI model like a software engineer using technologies such as Docker containers and Kubernetes? This guided project will show you how to deploy a trained model app in less than an hour. On the one hand, this project does not require knowledge of front-end and back-end development. On the other hand, the model deployment comes at no cost! You get free resources on IBM Cloud to deploy the AI model you like and share the app with others.



In this project, we will show you step-by-step how to use IBM Code Engine to deploy your AI application on IBM Cloud. IBM Code Engine is a fully managed, serverless platform which provides an abstraction for the underlying infrastructure required to deploy your apps and lets you focus on the source code only (such as the Python code).

You will also use Gradio, a Python framework that allows you to build demos or interactive apps of your Machine Learning models, APIs, or Data Science workflows and share them. Gradio lets you quickly generate a user interface for your application with just a few lines of code, which means even without knowledge of HTML or CSS, you can still create a simple UI by calling its powerful Python APIs.

Learning Objectives

- Know how to wrap a Machine Learning model inside Gradio’s interface.
- Understand containerization.
- Have hands-on experience with containerization.
- Become familiar with IBM Code Engine.
- Know how to use Code Engine to create and store container images on IBM Cloud.
- Deploy the Machine Learning app from the container image.
- Learn good practices and troubleshooting with IBM Code Engine.

Prerequisites (optional)

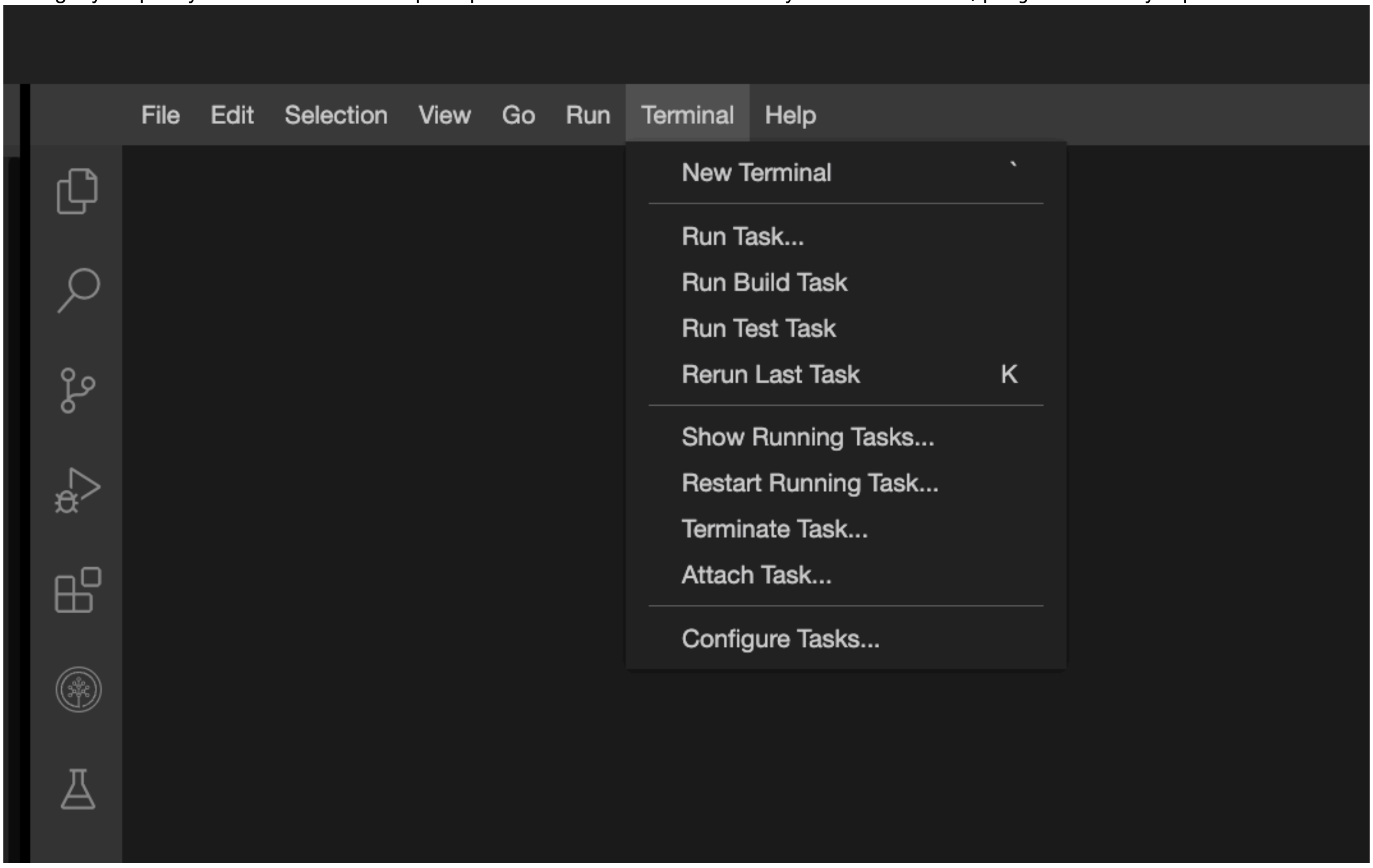
There is no prerequisite for this project. We will teach you all the Python code and the Code Engine CLI commands. We will make sure you understand everything as you go through the steps of the project. If you don’t have a finished AI model, don’t worry. We will also teach you some basics of deploying containerized applications.

What are you still waiting for? Let’s jump right in!

Quickstart Gradio

Setting up the environment and install Gradio

Let’s get you quickly started with Gradio. Open up a new terminal and make sure you are in the `home/project` directory. Open a terminal:



Create a python virtual environment and install Gradio using the following commands in the terminal:

```
pip3 install virtualenv
virtualenv my_env # create a virtual environment my_env
source my_env/bin/activate # activate my_env
```

Copied! Executed!

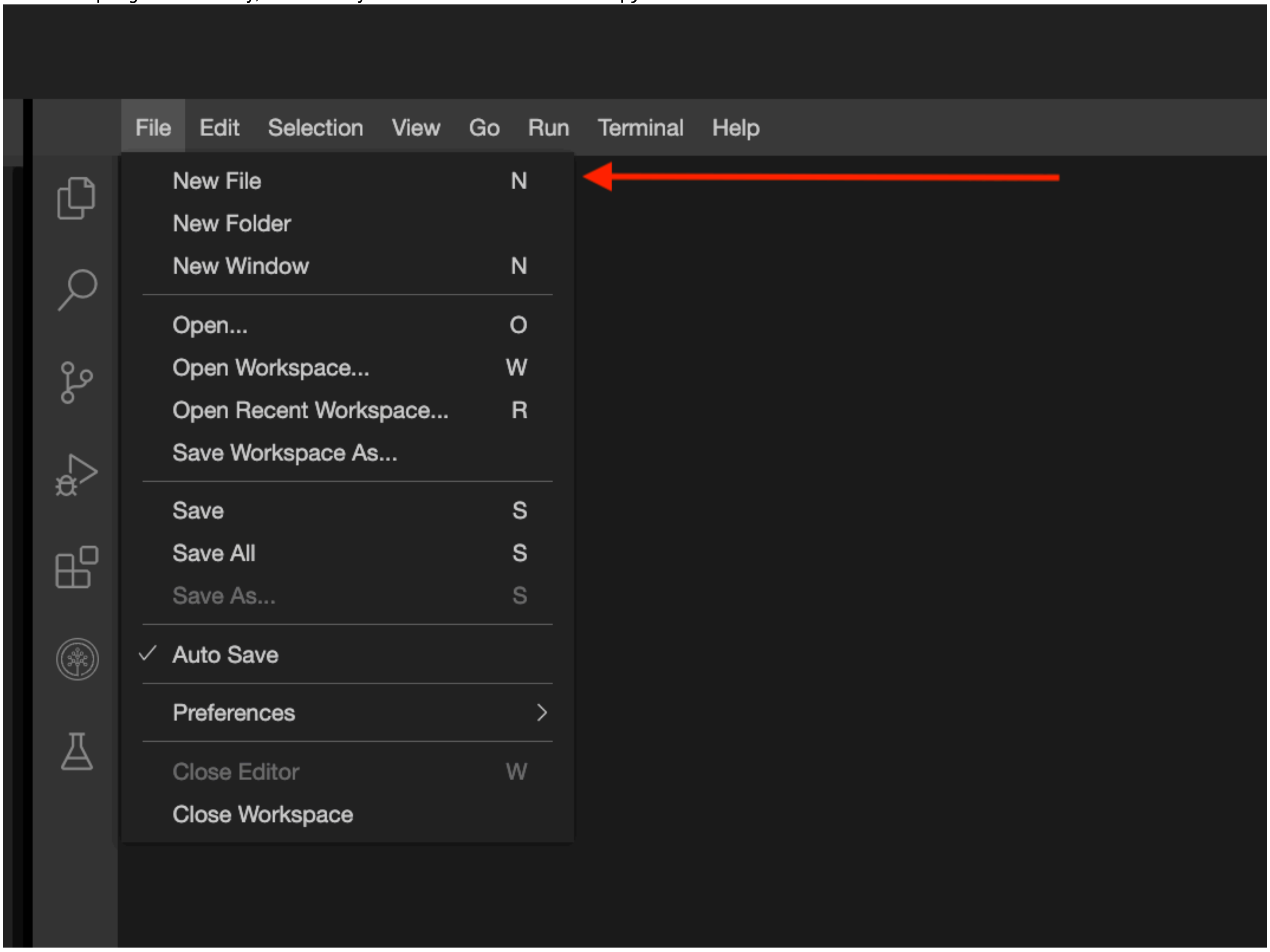
Then, install the required liberies in the environment:

```
# installing required libraries in my_env
pip3 install gradio
pip3 install cloudpickle
pip install -U scikit-learn
pip3 install requests
```

Copied! Executed!

Creating a simple demo

Still in the `project` directory, create a Python file and name it `hello.py`.



Open `hello.py`, paste the following Python code and save the file.

```
1
2
3
4
5
6
7
8
```

```
import gradio as gr

def greet(name):
    return "Hello " + name + "!"

demo = gr.Interface(fn=greet, inputs="text", outputs="text")

demo.launch()
```

Copied!

The above code creates a **gradio.Interface** called `demo`. It wraps the `greet` function with a simple text-to-text user interface that you could interact with.

The **gradio.Interface** class is initialized with 3 required parameters:

- fn: the function to wrap a UI around
- inputs: which component(s) to use for the input (e.g. “text”, “image” or “audio”)
- outputs: which component(s) to use for the output (e.g. “text”, “image” or “label”)

The last line `demo.launch()` launches a server to serve our `demo`.

Launching the demo app

Now go back to the terminal and make sure that the `my_env` virtual environment name is displayed at the begining of the line. Now run the following command to execute the Python script.

```
1. 1

1. python3 hello.py
```

Copied! Executed!

As the Python code is served by local host, click on the button below and you will be able to see the simple application we just created. Feel free to play around with the input and output of the web app!

Web Application

You should see the following, here we entered the name bob:

If you finish playing with the app and want to exit, **press `ctrl+c` in the terminal and close the application tab**.

You just had a first taste of the Gradio interface, it's easy right? If you wish to learn a little bit more about customization in Gradio, you are invited to take the guided project called **Bring your Machine Learning model to life with Gradio**. You can find it under **Courses & Projects** on cognitiveclass.ai!

Deploy a parkinson detector App with Code Engine

Now that we have Gradio as our friend to help us with fast-generating a user interface for an application, let's see how we can run applications on IBM Cloud and access it with a public url using IBM Code Engine.

Container images and Containers

Code Engine lets you run your apps in containers on IBM Cloud. You might wonder what are containers. A **container** is an isolated environment or place where an application can run independently. Containers can run anywhere, such as on operating systems, virtual machines, developer's machines, physical servers, etc. This allows the containerized application to run anywhere as well and the isolation mechanism makes sure that the running application will not interfere with the rest of the system.

How can we create these containers that are so convenient for deploying apps? Containers are all created from container images. A container image is basically a snapshot or a blueprint that tells what will be in a container when it runs. Therefore, to deploy a containerized parkinson app, we first need to create the app's container image.

Creating the container image

As we talked about, a container image tells us what will be in a container when it runs. If we are going to deploy a parkinson app in a container, what are the files that we need?

- First of all, let me remind you that we can make use of the Gradio framework for generating the user interface of the app, so let's call the Python script which contains the code that creates and launches the **gradio.Interface** `demo.py`.
- Second, the source code of the app has its dependencies, such as libraries that the code uses. Hence, we need a `requirements.txt` file that specifies all the libraries the source code depends on.
- Third, and most importantly, we need a file that tells the container runtime the steps for assembling the container image. We call it `Dockerfile`.

Let's create the three files. Open up a terminal and while you are in the `home/project` directory, make a new directory `myapp` for storing the files and get into the directory with:

```
1. 1
2. 2

1. mkdir myapp
2. cd myapp
```

Copied! Executed!

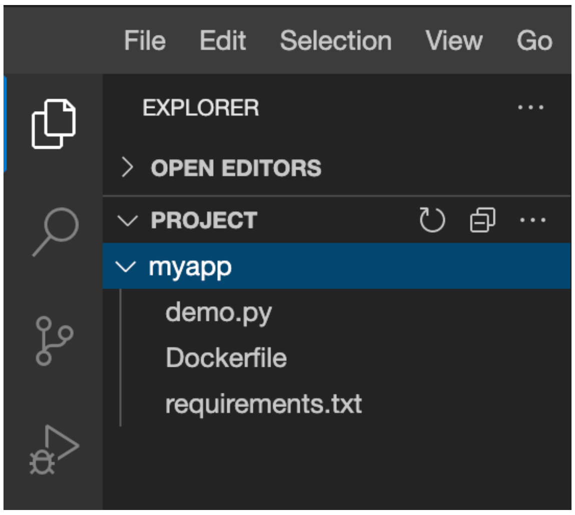
Now that you are in the `myapp` directory, run the following command to create the files:

```
1. 1

1. touch demo.py Dockerfile requirements.txt
```

Copied! Executed!

If you open the file explorer, you will see the files you just created.



Next, we will take a closer look at what should be included in each of these three files.

Step 1: Creating requirements.txt

If you are a Data Scientist, you may be very familiar with the `pip3 install <library-name>` command for installing libraries. By using a `requirements.txt` file that consists of all libraries you need, you can install all of them into your environment at once with the command `pip3 install -r requirements.txt`.

Our goal is to deploy the app in a container, thus all the dependencies need to go into the container as well.

Let’s create a `requirements` file to cover all the libraries and dependencies your app may need.

Open the `requirements.txt` file in `/myapp` and paste the following library names into the file.

```
gradio
scikit-learn
cloudpickle
requests
```

Copied!

Testing requirements.txt locally

Since we have already installed the required packages in the “QuickStart Gradio” section, we don’t need to install them again here. However, it’s important to test the application locally before launching it through Docker to ensure that it runs smoothly without any errors.

You can test to see if the file works correctly by executing:

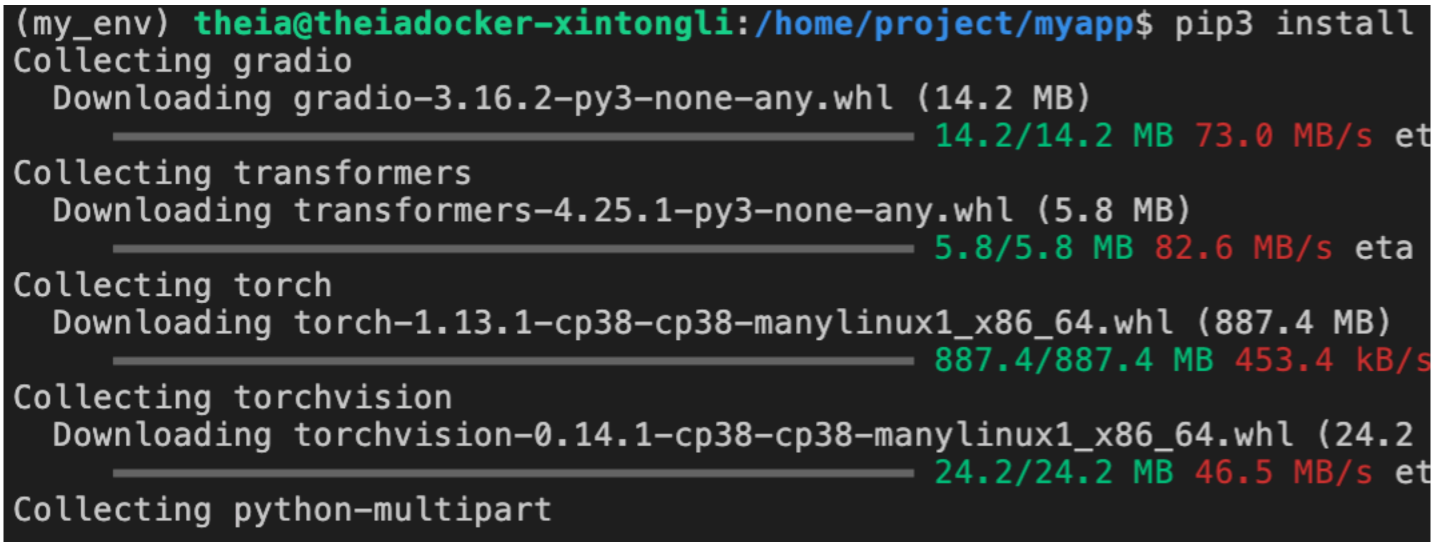
```
pip3 install -r requirements.txt
```

Copied! Executed!

in the terminal of your current CloudIDE working environment.

Note: Make sure you are in the `myapp` directory and your virtual environment `my_env` created previously is activated. This allows the libraries to be installed into your virtual environment only.

You should see the following:



```
(my_env) theia@theiadocker-xintongli:/home/project/myapp$ pip3 install
Collecting gradio
  Downloading gradio-3.16.2-py3-none-any.whl (14.2 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 14.2/14.2 MB 73.0 MB/s et
Collecting transformers
  Downloading transformers-4.25.1-py3-none-any.whl (5.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5.8/5.8 MB 82.6 MB/s eta
Collecting torch
  Downloading torch-1.13.1-cp38-cp38-manylinux1_x86_64.whl (887.4 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 887.4/887.4 MB 453.4 kB/s
Collecting torchvision
  Downloading torchvision-0.14.1-cp38-cp38-manylinux1_x86_64.whl (24.2
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.2/24.2 MB 46.5 MB/s et
Collecting python-multipart
```

Now that we have the libraries we need, let’s go to writing the Python code for the text-generation model demo.

Step 2: Creating demo.py

Open the `demo.py` file in `/myapp` and fill the empty script with the following code.

```python
import gradio as gr
import sklearn
import requests
import cloudpickle as cp
import gradio as gr
import pandas as pd

# setting up the feature names:
feature_names = ['MDVP:Fo(Hz)', 'MDVP:Fhi(Hz)', 'MDVP:Flo(Hz)', 'MDVP:Jitter(%)', 'MDVP:Jitter(Abs)', 'MDVP:RA

#load the trained model
# set the URL of the pickle file to download the trained model
url = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMSkillsNetwork-GPXX0N9TEN/model/rf_

# use the requests module to download the pickle file
response = requests.get(url)

# raise an exception if the download fails
response.raise_for_status()

# load the pickle object from the response content
rf = cp.loads(response.content)


def detect_Parkinson(variables):

   # Split the variables string into a list of strings
    variables = variables.strip().split("\n")

    # Convert the list of strings to a pandas DataFrame
    df_temp = pd.DataFrame([variables], columns=feature_names)

    # Make a prediction based on the inputs
    prediction = rf.predict(df_temp)
    if prediction==1:
        return "Predicted label: 1 --> 🔴 Parkinson Detected"
    else:
        return "Predicted label: 0 --> 🟢 Parkinson Not Detected"

inputs = [gr.inputs.Textbox(lines=22, default="\n".join(feature_names))]

outputs = [gr.outputs.Textbox()]

demo = gr.Interface(detect_Parkinson, inputs, outputs)

demo.launch(server_name="0.0.0.0", server_port= 7860)
```

Copied!

What does the script do?

Let me explain to you what we are doing in this Python script.

This code sets up a Gradio interface for detecting Parkinson's disease using a pre-trained random forest model. Here's a brief explanation of each step:

1. The required libraries are imported, including Gradio for building the user interface, pandas for handling data, requests for downloading the pre-trained model file, and cloudpickle for loading the pickled model.
2. The `feature_names` list is defined, which contains the names of the 22 features used in the pre-trained model.
3. The URL of the pre-trained model file is set.
4. The `requests` module is used to download the model file from the URL.
5. The `cloudpickle` module is used to load the pickled model from the response content.
6. The `detect_Parkinson` function is defined, which takes the user's input as a string of values separated by newlines. The function then converts the input into a Pandas DataFrame and makes a prediction based on the pre-trained random forest model.
7. An input textbox with 22 lines is created for the user to enter the values for each feature.
8. An output textbox is created to display the prediction made by the model.
9. A Gradio interface is created with the `detect_Parkinson` function as the main function, the inputs and outputs lists defining the input and output types, and the `demo.launch()` method launching the interface.

When the user inputs the feature values and clicks the "Submit" button, the `detect_Parkinson` function is called to make a prediction based on the pre-trained model. The predicted label is then displayed in the output textbox of the Gradio interface.

Let's test our application and make sure it can run properly.

Testing demo.py locally

Open your terminal and `cd myapp` to enter the correct directory of files.

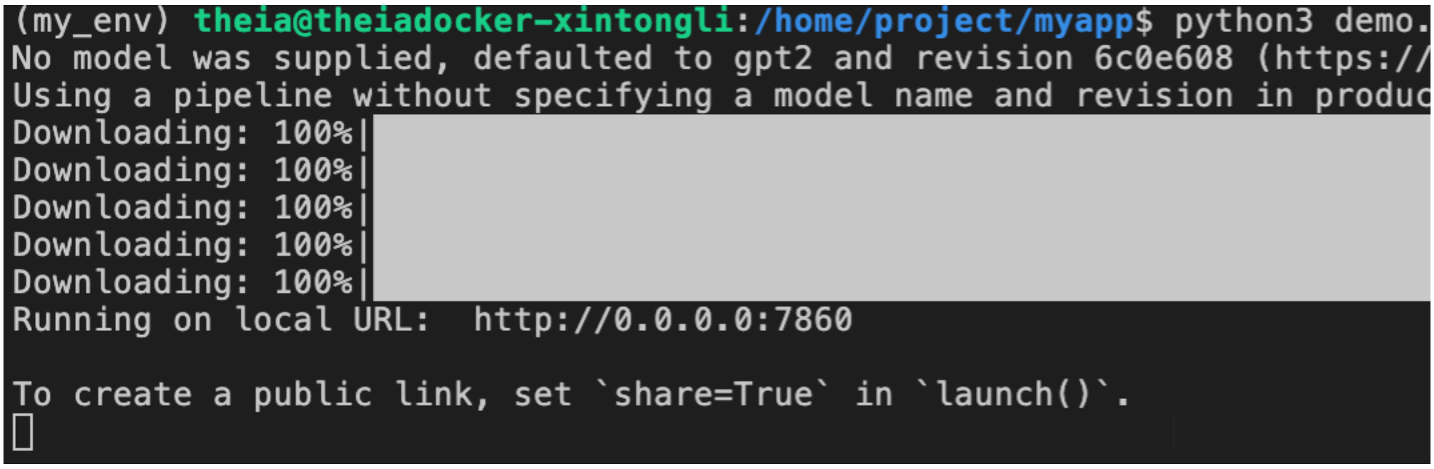
If you have run `pip3 install -r requirements.txt` in the previous step and it was all successful, you are good to go . If not, please run the command now to have the required libraries installed in your working environment.

Run the following command in the terminal:

```
python3 demo.py
```

Copied! Executed!

If you run this script properly, you should see the following result in your terminal:



```
(my_env) theia@theiadocker-xintongli:/home/project/myapp$ python3 demo.
No model was supplied, defaulted to gpt2 and revision 6c0e608 (https://
Using a pipeline without specifying a model name and revision in produc
Downloading: 100%|
Downloading: 100%|
Downloading: 100%|
Downloading: 100%|
Downloading: 100%|
Running on local URL:  http://0.0.0.0:7860

To create a public link, set `share=True` in `launch()`.
```

The result shows that the parkinson detection model pipeline is downloaded and the app is running on http://0.0.0.0:7860/. Click on the button below to access the web app hosted in the CloudIDE:

Web Application

In the app, every feature is writen and you can insert these numbers for specific feature. Here is an example:

```
1.137150e+02
1.164430e+02
9.691300e+01
3.490000e-03
3.000000e-05
1.710000e-03
2.030000e-03
5.140000e-03
1.472000e-02
1.330000e-01
7.480000e-03
9.050000e-03
1.148000e-02
2.245000e-02
```

```
4.780000e-03
2.654700e+01
3.802530e-01
7.667000e-01
-5.943501e+00
1.921500e-01
1.852542e+00
1.796770e-01
```

Copied!

You can press `ctrl+c` to shut down the application.

Now let’s proceed to creating the `Dockerfile` which tells the container runtime what to do with our files for constructing the container image.

Step 3: Creating Dockerfile

We create the `Dockerfile` which is the blueprint for assembling a container image.

Open the `Dockerfile` in `/myapp` and paste following commands into the file.

```
FROM python:3.10

WORKDIR /app
COPY requirements.txt requirements.txt
RUN pip3 install --no-cache-dir -r requirements.txt

COPY . .

CMD ["python", "demo.py"]
```

Copied!

What does the Dockerfile do?

FROM python:3.10

Docker images can be inherited from other images. Therefore, instead of creating our own base image, we’ll use the official Python image `python:3.10` that already has all the tools and packages that we need to run a Python application.

WORKDIR /app

To make things easier when running the rest of our commands, we create a working directory `/app`. This instructs Docker to use this path as the default location for all subsequent commands. By doing this, we do not have to type out full file paths but can use relative paths based on the working directory.

COPY requirements.txt requirements.txt

Before we can run `pip3 install`, we need to get our requirements.txt file into our image. We’ll use the `COPY`command to do this. The `COPY`command takes two parameters. The first parameter tells Docker what file(s) you would like to copy into the image. The second parameter tells Docker where you want that file(s) to be copied to. We’ll copy the `requirements.txt` file into our working directory `/app`.

RUN pip3 install –no-cache-dir -r requirements.txt

Once we have our `requirements.txt` file inside the image, we can use the `RUN` command to execute the command `pip3 install --no-cache-dir -r requirements.txt`. This works exactly the same as if we were running the command locally on our machine, but this time the modules are installed into the image.

COPY . .

At this point, we have an image that is based on Python version 3.10 and we have installed our dependencies. The next step is to add our source code into the image. We'll use the `COPY` command just like we did with our `requirements.txt` file above to copy everything in our current working directory to the file system in the container image.

CMD ["python", "demo.py"]

Now, all we have to do is to tell Docker what command we want to run when our image is executed inside a container. We do this using the CMD command. Docker will run the `python demo.py` command to launch our app inside the container.

Now that all three files have been created, let's bring in Code Engine for building the container image.

IBM Code Engine Project

IBM Cloud® Code Engine is a fully managed, serverless platform that runs your containerized workloads, including web apps, micro-services, event-driven functions, or batch jobs. Code Engine even builds container images for you from your source code. All these workloads can seamlessly work together because they are all hosted within the same Kubernetes infrastructure. The Code Engine experience is designed so that you can focus on writing code and not on the infrastructure that is needed to host it.

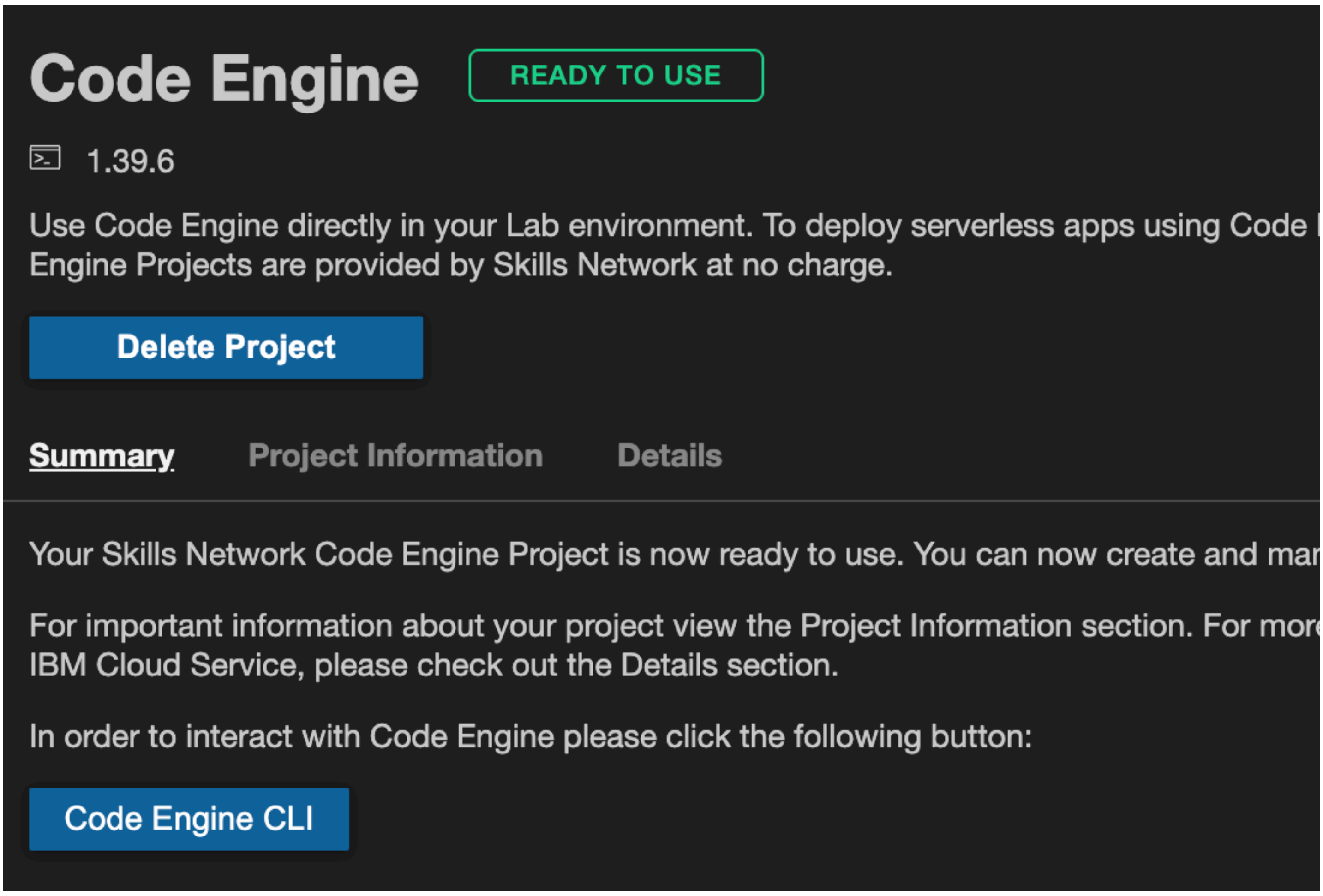
Creating your Code Engine (CE) project

To deploy serverless apps using Code Engine you'll need a project. A **project** is a grouping of Code Engine entities such as applications, jobs, and builds. Projects are used to manage resources and provide access to its entities.

As you started this guided project, we already have a CE project set up for you so you don't need to go through creating and configuring the project parameters yourself (Thanks to the Skills Network developers!). Now it's time to click on the button below to create your project!

Create Code Engine Project in IDE

Wait for 3 to 5 minutes and you should see the following indicating that your Code Engine project is ready to use.


Code Engine READY TO USE

1.39.6

Use Code Engine directly in your Lab environment. To deploy serverless apps using Code
Engine Projects are provided by Skills Network at no charge.

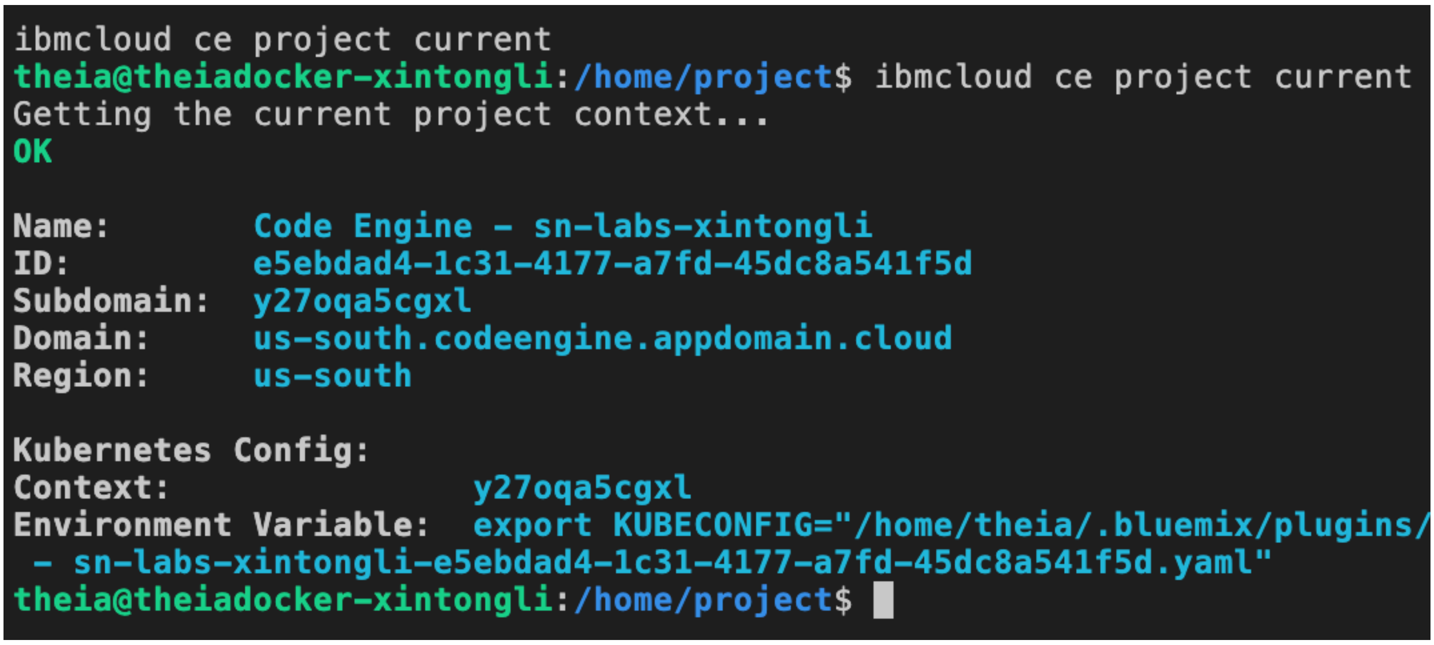
Your Skills Network Code Engine Project is now ready to use. You can now create and mar

For important information about your project view the Project Information section. For more
IBM Cloud Service, please check out the Details section.

In order to interact with Code Engine please click the following button:

Launching the Code Engine (CE) CLI

Once your project is ready, click on the **Code Engine CLI** button to launch your project in the terminal. The new terminal that just opened should show information about your current CE project, such as the name and ID of your project and the region where your project is deployed to.



```
ibmcloud ce project current
theia@theiadocker-xintongli:/home/project$ ibmcloud ce project current
Getting the current project context...
OK

Name:        Code Engine - sn-labs-xintongli
ID:          e5ebdad4-1c31-4177-a7fd-45dc8a541f5d
Subdomain:   y27oqa5cgxl
Domain:      us-south.codeengine.appdomain.cloud
Region:      us-south

Kubernetes Config:
Context:               y27oqa5cgxl
Environment Variable:  export KUBECONFIG="/home/theia/.bluemix/plugins/
 - sn-labs-xintongli-e5ebdad4-1c31-4177-a7fd-45dc8a541f5d.yaml"
theia@theiadocker-xintongli:/home/project$ █
```

As you created your current project, you have resources on IBM Cloud® allocated to the project, such as CPU runtime, memory, storage, etc. You can check for limits and quota usage of this project's allocated resources by running this command:

You can access the information of your project in cloud ide by clicking on the "Code engine" page and select "Project Summary".

Note: surround your project name with double quotes if it has space in the characters.

```
1. 1
```

```
1. ibmcloud ce project get --name PROJECT_NAME
```

Copied! Executed!

Next, we are going to use these resources to deploy our app.

Building a Container Image with Code Engine

In a Code Engine build, you need to define a source that points to a place where your source code and Dockerfile reside, it could be a public or private Git repository, or a local source if you want to run the build using the files in your working directory.

Since we have created all the files in the `myapp` directory, we are going to build the image from local source, and then upload the image to your container registry with the registry access that you provide. Let's first change the working directory to `/myapp` where we have the source code and other files.

if you are not in the myapp folder change the working directory to it:

```
1. 1
```

```
1. cd myapp
```

Copied! Executed!

Container Registries

A container registry, or registry, is a service that stores container images. For example, IBM Cloud Container Registry and Docker Hub are container registries.

Images that are used by IBM Cloud® Code Engine are typically stored in a registry that can either be accessible by the public (public registry) or set up with limited access for a small group of users (private registry).

Code Engine requires access to container registries to complete the following actions:

- To store and retrieve local files when a build is run from local source
- To store a newly created container image as an output of an image build
- To pull a container image to run an app or job

Your IBM Cloud Container Registry

When you created the Code Engine project, Code Engine has also created a registry access secret for you to a private IBM Cloud Container Registry namespace. You can find the provided ICR namespace and the registry secret in the **Code Engine** tab.

Note that in order to use the provided ICR namespace to store docker images you will need to include `--registry-secret icr-secret` in most of your CE commands. **Code Engine handles many of the underlying details of the interactions between the system and your registry.**

Creating the Build Configuration

To create a build configuration that pulls code from a local directory, use the `build create` command and specify the `build-type` as `local`.

Since you also need Code Engine to store the image in IBM Cloud Container Registry for us, you need to provide your registry access secret so that Code Engine can access and push the build result to the registry in your namespace.

Run the following command in your Code Engine CLI to create a build configuration:

```
ibmcloud ce build create --name build-local-dockerfile1 \
                        --build-type local --size large \
                        --image us.icr.io/${SN_ICR_NAMESPACE}/myapp1 \
                        --registry-secret icr-secret
                        /
```

Copied! Executed!

With the `build create` command, we create a build configuration called `build-local-dockerfile1` and we specify `local` as the value for `--build-type`. The size of the build defines how much resources such as CPU cores, memory, and disk space are assigned to the build. We specify `size` as `large` in the case our model pipeline that will be downloaded into a container requires lots of resources to run.

We also provide the location of the image registry, which is the namespace `us.icr.io/${SN_ICR_NAMESPACE}` in the IBM Cloud Container Registry. Remember to replace `${SN_ICR_NAMESPACE}` with your ICR namespace provided by Code Engine. Your ICR namespace can be seen in *Code Engine page -> project information*.

The container image will be tagged (named) as `myapp1`. We specify the `--registry-secret` option to access the registry with the `icr-secret`.

Submitting and Running the Build Configuration

To submit a build run from a build configuration with the CLI that pulls source from a local directory, use the `buildrun submit` command. This command requires the name of the build configuration, and the path to your local source. Other optional arguments can be specified.

When you submit a build that pulls code from a local directory, your source code is packed into an archive file and uploaded to your IBM Cloud Container Registry instance. Note that you can only target IBM Cloud Container Registry for your local builds. The source image is created in the same namespace as your build image.

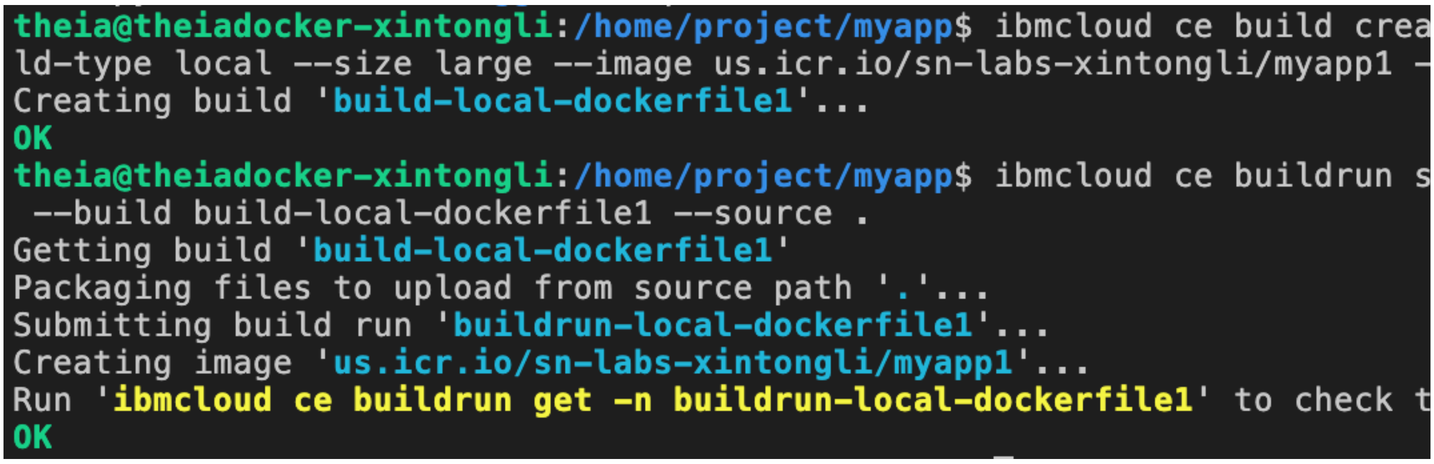
Run the following command in your Code Engine CLI to submit and run the build configuration:

```
ibmcloud ce buildrun submit --name buildrun-local-dockerfile1 \
                            --build build-local-dockerfile1 \
                            --source .
                            /
```

Copied! Executed!

From the directory where our source code resides, i.e: `/myapp`, submit the build run. The above command runs a build that is called `buildrun-local-dockerfile1` and uses the `build-local-dockerfile1` build configuration that we just created. The `--source` option specifies the path to the source on the local workstation.

After the two commands, the following result should be displayed in the terminal:



```
theia@theiadocker-xintongli:/home/project/myapp$ ibmcloud ce build crea
ld-type local --size large --image us.icr.io/sn-labs-xintongli/myapp1 -
Creating build 'build-local-dockerfile1'...
OK
theia@theiadocker-xintongli:/home/project/myapp$ ibmcloud ce buildrun s
 --build build-local-dockerfile1 --source .
Getting build 'build-local-dockerfile1'
Packaging files to upload from source path '.'...
Submitting build run 'buildrun-local-dockerfile1'...
Creating image 'us.icr.io/sn-labs-xintongli/myapp1'...
Run 'ibmcloud ce buildrun get -n buildrun-local-dockerfile1' to check t
OK
```

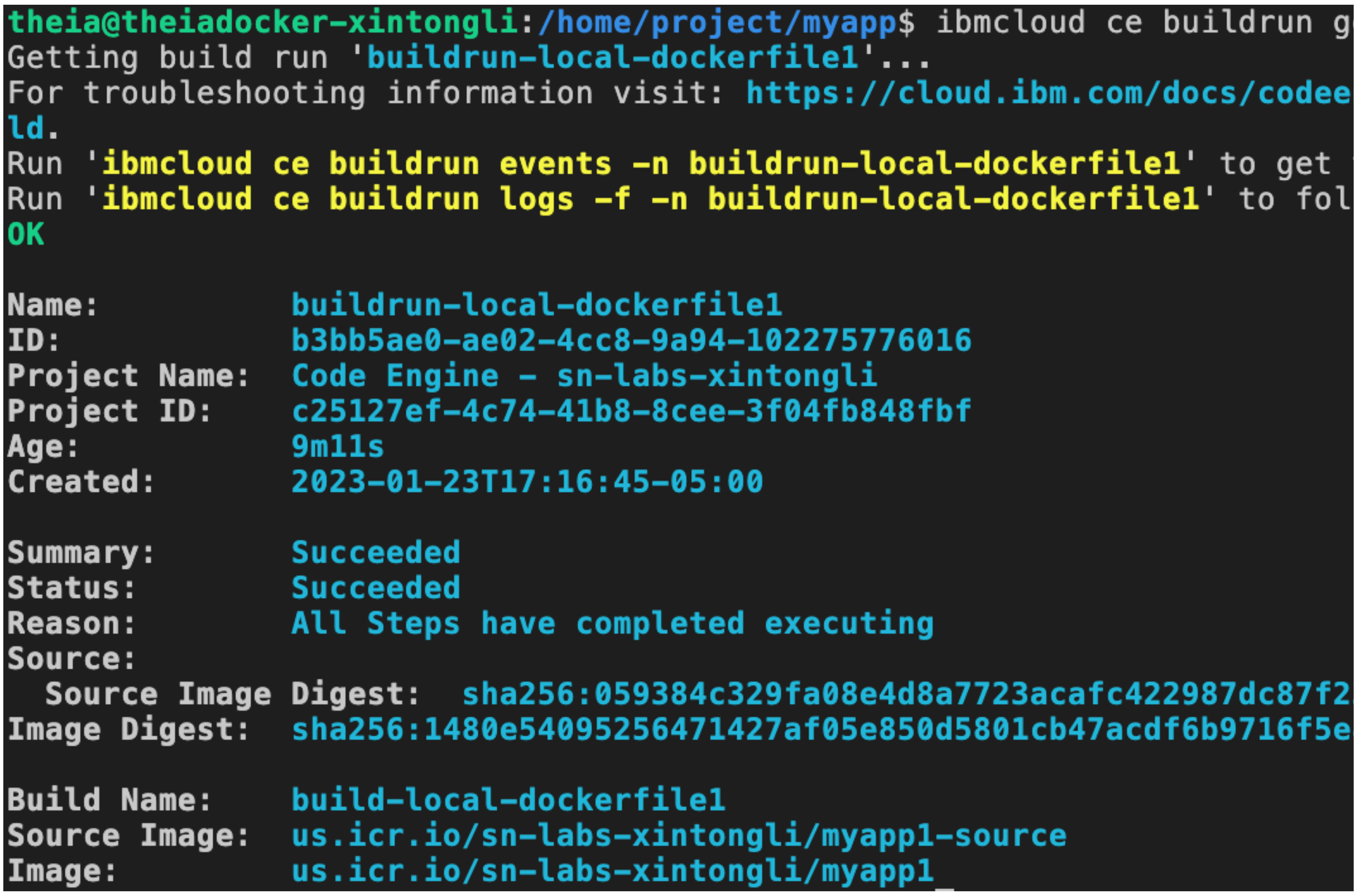
It will take ~3-5 minutes for the all the steps in a buildrun to finish running. To monitor the progress of the buildrun, use the following command:

1. 1

1. `ibmcloud ce buildrun get -n buildrun-local-dockerfile1`

Copied! Executed!

Once you see the status showing `Succeeded` (same as the following screenshot), that means your container image has been created successfully and pushed to the registry under your namespace.



```
theia@theiadocker-xintongli:/home/project/myapp$ ibmcloud ce buildrun g
Getting build run 'buildrun-local-dockerfile1'...
For troubleshooting information visit: https://cloud.ibm.com/docs/codee
ld.
Run 'ibmcloud ce buildrun events -n buildrun-local-dockerfile1' to get
Run 'ibmcloud ce buildrun logs -f -n buildrun-local-dockerfile1' to fol
OK

Name:          buildrun-local-dockerfile1
ID:            b3bb5ae0-ae02-4cc8-9a94-102275776016
Project Name:  Code Engine - sn-labs-xintongli
Project ID:    c25127ef-4c74-41b8-8cee-3f04fb848fbf
Age:           9m11s
Created:       2023-01-23T17:16:45-05:00

Summary:       Succeeded
Status:        Succeeded
Reason:        All Steps have completed executing
Source:
  Source Image Digest:  sha256:059384c329fa08e4d8a7723acafc422987dc87f2
Image Digest:  sha256:1480e54095256471427af05e850d5801cb47acdf6b9716f5e

Build Name:    build-local-dockerfile1
Source Image:  us.icr.io/sn-labs-xintongli/myapp1-source
Image:         us.icr.io/sn-labs-xintongli/myapp1
```

Now that the container image is ready, what's left for us is to pull the image from the Container Registry and deploy a containerized application using the image!

Deploying a Containerzied App using Code Engine

In the previous step, you have created and pushed the app's image to your namespace in the Container Registry. Let's deploy the app by referencing the `us.icr.io/${SN_ICR_NAMESPACE}/myapp1` image in Container Registry.

Creating your application

Code Engine lets you deploy an application that uses an image stored in IBM Cloud® Container Registry with the `ibmcloud app create` command.

Run the following command in your Code Engine CLI to deploy the app:

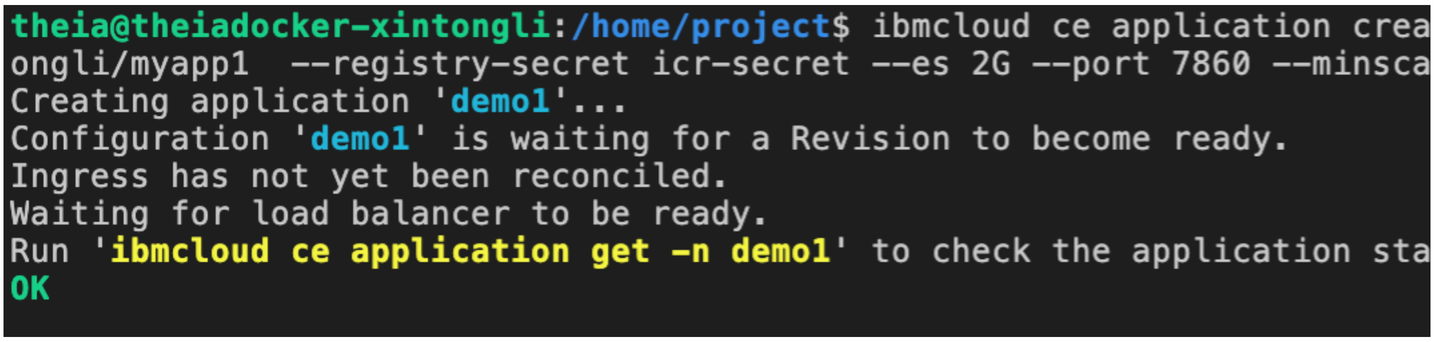
```
ibmcloud ce application create --name demo1 \
                             --image us.icr.io/${SN_ICR_NAMESPACE}/myapp1  \
                             --registry-secret icr-secret --es 2G \
                             --port 7860 --minscale 1
```

Copied! Executed!

There are some important arguments specified in the command:

- The deployed app will be called `demo1`.
- It references (uses) the image `us.icr.io/sn-labs-xintongli/myapp1`.
- It uses `2G` of ephemeral storage in the container. For smaller applications, a default value for the ephemeral storage, i.e. `--es 400MB`, might be enough. We need 2G because of the size of the GPT2 model.
- We need to specify the port number `7860` for the application so that the public network connections and requests can be directed to the application when they arrive at the server.
- We set `--minscale 1` to make sure that our app will keep running even though there are no continuous requests. This is important because otherwise we would need to wait for the app to start running everytime we access its URL.

After you run the command, it should take approximately 3-5 minutes for you to see the following message in the CLI:



```
theia@theiadocker-xintongli:/home/project$ ibmcloud ce application crea
ongli/myapp1  --registry-secret icr-secret --es 2G --port 7860 --minsca
Creating application 'demo1'...
Configuration 'demo1' is waiting for a Revision to become ready.
Ingress has not yet been reconciled.
Waiting for load balancer to be ready.
Run 'ibmcloud ce application get -n demo1' to check the application sta
OK
```

Accessing your application

This means your app has been deployed and you can access it now! To obtain the URL of your app, run `ibmcloud ce app get --name demo1 --output url`. Click on the URL returned, and you should be able to see your app running in your browser!

You can also create a custom domain and assign it to your app. For information about deploying an app with a custom domain through Cloudflare, see the Configuring a Custom Domain for Your Code Engine Application.

Conclusion

Congratulations on completing this guided project! You have now mastered a new Machine Learning model deployment skill using Gradio and IBM Code Engine. The URL of your deployed app can be accessed anytime by anybody as long as you didn't shut down the server in the Code Engine CLI.

Next Steps

In this guided project, you deployed an application to a Kubernetes cluster using IBM Code Engine. You used a shared cluster provided to you by the IBM Developer Skills Network. If you wish to deploy your containerized app and get a permanent URL of the app outside of the Code Engine CLI on your local machine, you can learn more about Kubernetes and containers. You can get your own free Kubernetes cluster and your own free IBM Container Registry.

Author(s)

Sina Nazeri (Linkedin profile)

> *As a data scientist in IBM, I have always been passionate about sharing my knowledge and helping others learn about the field. I believe that everyone should have the opportunity to learn about data science, regardless of their background or experience level. This belief has inspired me to become a learning content provider, creating and sharing educational materials that are accessible and engaging for everyone.]*

Roxanne Li

Change log

Date	Version	Changed by	Change Description
2023-03-15	0.1	Sina Nazeri	Created project first draft