O'REILLY®

Practical MLOps

Operationalizing Machine Learning Models

Noah Gift & Alfredo Deza

CHAPTER 7
MLOps for AWS

By Noah Gift

> Everybody was scared of him [Dr. Abbott (because he yelled at everyone)]. When I was attending, there was a new resident named Harris. Harris was still afraid of him [Dr. Abbott] even though he was the chief resident and had been there for 5 years. Later [Dr. Abbott] has a heart attack and then his heart stops. A nurse yells, "Quick, he just had an arrest, come in!" So Harris went in there…leaning on the sternum and breaking ribs and stuff. So Harris starts pumping on Abbott and he woke up. He woke up! And he looked up at Harris and he said, "You! STOP THAT!" So Harris stopped. And that was the last story about Abbott.
>
> —Dr. Joseph Bogen

One of the most common questions I get from students is, "which cloud do I pick?" I tell them the safe choice is Amazon. It has the widest selection of technology and the largest market share. Once you master the AWS cloud, it is easier to master other cloud offerings since they also assume you might know AWS. This chapter covers the foundations of AWS for MLOps and explores practical MLOps patterns.

I have a long, rich history of working with AWS. At a sports social network I ran as the CTO and General Manager, AWS was a secret ingredient that allowed us to scale to millions of users worldwide. I also worked as an SME (subject matter expert) on the AWS Machine Learning Certification from scratch in the last several years. I have been recognized as an AWS ML Hero, I am also a part of the AWS Faculty Cloud Ambassador program, and have taught thousands of students cloud certifications at UC Davis, Northwestern, Duke, and the University of Tennessee. So you can say I am a fan of AWS!

Due to the sheer size of the AWS platform, it is impossible to cover every single aspect of MLOps. If you want a more exhaustive coverage of all of the available AWS platform options, you can also check out *Data Science on AWS* by Chris Fregly and Antje Barth (O'Reilly), for which I was one of the technical editors.

Instead, this chapter focuses on higher-level services like AWS Lambda and AWS App Runner. More complicated systems like AWS SageMaker are covered in other chapters in this and the previously mentioned book. Next, let's get started building AWS MLOps solutions.

Introduction to AWS

AWS is the leader in cloud computing for several reasons, including its early start and corporate culture. In 2005 and 2006, Amazon launched Amazon Web Services, including MTurk (Mechanical Turk), Amazon S3, Amazon EC2, and Amazon SQS.

To this day, these core offerings are not only still around, but they get better by the quarter and year. The reason AWS continues to improve its offerings is due to its culture. They say it is always "Day 1" at Amazon, meaning the same energy and enthusiasm that happens on Day 1 should happen every day. They also place the customer at the "heart of everything" they do. I have used these building blocks to build systems scaled to thousands of nodes to do machine learning, computer vision, and AI tasks. Figure 7-1 is an actual whiteboard drawing of a working AWS Cloud Technical Architecture shown in a coworking spot in downtown San Francisco.

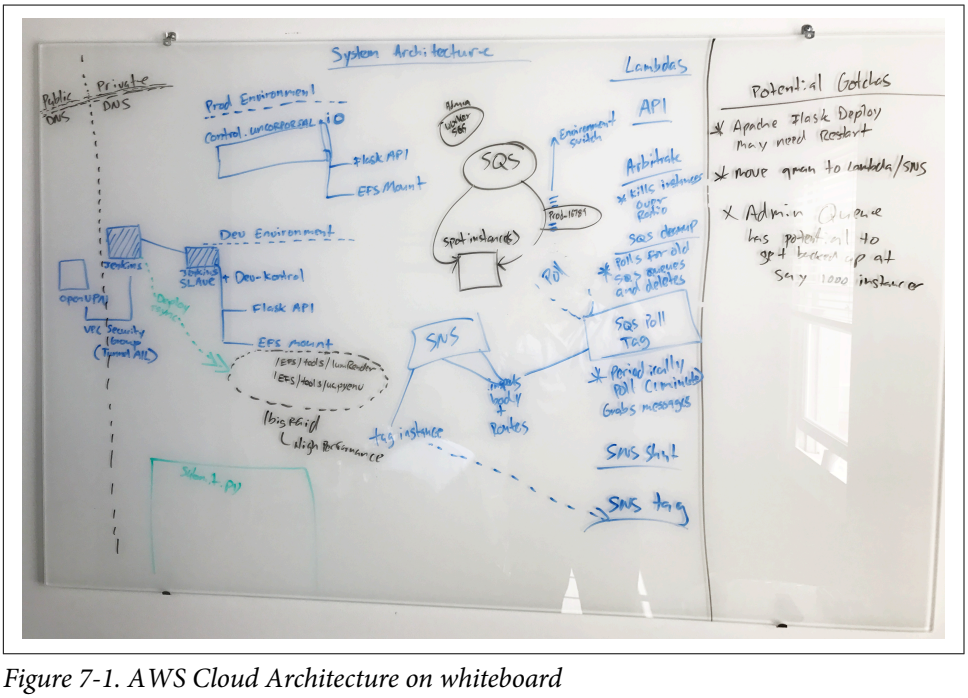

Figure 7-1. AWS Cloud Architecture on whiteboard

This type of culture is very different than Google, which has struggled in cloud computing. The Google culture is research-oriented with heavy academic hiring. The benefits of this culture are open source projects like Kubernetes and TensorFlow. Both are complex engineering marvels. The downside is that the customer is not first in the culture, which has hurt Google in the cloud computing market share. Many organizations balk at not buying professional services and supporting something as critical as cloud computing.

Next, let's take a look at getting started with AWS Services.

Getting Started with AWS Services

To get started with AWS initially requires only a Free Tier account. If you are at university, you can also use both AWS Academy and AWS Educate. AWS Academy provides hands-on certification material and labs. AWS Educate delivers sandboxes for classrooms.

Once you have an account, the next step is to experiment with the various offerings. One way to think about AWS is to compare it to a bulk wholesale store. For example, according to statista, Costco has 795 worldwide locations in multiple countries and approximately 1 in 5 Americans shop there. Similarly, according to the 2020 Amazon Web Services whitepaper, AWS provides an infrastructure that "powers hundreds of thousands of businesses in 190 countries around the world."

At Costco, there are three approaches considered relevant to an AWS analogy:

- A customer could walk in and order a very inexpensive but reasonable quality pizza in bulk in the first scenario.
- In a second scenario, a customer new to Costco needs to figure out all of the bulk items available to investigate the best way to use Costco. They will need to walk through the store and look at the bulk items like sugar to see what they could build from these raw ingredients.
- In a third scenario, when a customer of Costco knows about the premade meals (such as rotisserie chicken, pizza, and more), and they know about all of the raw ingredients they could buy, they could, in theory, start a catering company, a local deli, or a restaurant utilizing their knowledge of Costco and the services it provides.

A big box retailer the size of Costco charges more money for prepared food, and less money for the raw ingredients. The Costco customer who owns a restaurant may choose a different food preparation level depending on the maturity of their organization and the problem they aim to solve. Figure 7-2 illustrates how these Costco options compare to AWS options.

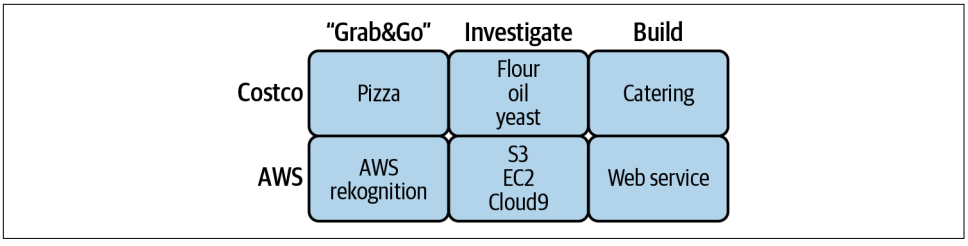

Figure 7-2. Costco versus AWS

Let's take the case of a local Hawaii Poke Bowl stand near a famous Hawaii beach. The owner could buy Costco's premade Poke in bulk and sell it at a price that is approximately twice their cost. But, on the other hand, a more mature BBQ restaurant in Hawaii, with employees who can cook and prepare food, Costco sells this uncooked food at a much lower price than the fully prepared Poke.

Much like Costco, AWS provides different levels of product and it's up to the customer to decide how much they're going to utilize. Let's dig into those options a bit.

Using the "No Code/Low Code" AWS Comprehend solution

The last example showed how using Costco can benefit different restaurant businesses, from one with little to no staff—like a pop-up stand—to a more extensive sit-down facility. The more work Costco does in preparing the food, the higher the benefit to the customer purchasing it, and also the higher the cost for the food. The same concept applies with AWS; the more work AWS does for you, the more you pay and the fewer people you need to maintain the service.

In economics, the theory of comparative advantage says that you shouldn't compare the cost of something directly. Instead, it would help if you compared the opportunity cost of doing it yourself. All cloud providers have this assumption baked in, since running a data center and building services on top of that data center is their specialization. An organization doing MLOps should focus on creating a product for its customers that generates revenue, not recreating what cloud providers do poorly.

With AWS, an excellent place to start is to act like the Costco customer who orders Costco pizza in bulk. Likewise, a Fortune 500 company may have essential requirements to add natural language processing (NLP) to its customer service products. It could spend nine months to a year hiring a team and then building out these capabilities, or it could start using valuable high-level services like AWS Comprehend for Natural Language Processing. AWS Comprehend also enables users to leverage the Amazon API to perform many NLP operations, including the following:

- Entity detection
- Key phrase detection
- PII
- Language detection
- Sentiment

For example, you can cut and paste text into the Amazon Comprehend Console, and AWS Comprehend will find all of the text's entities. In the example in Figure 7-3, I grabbed the first paragraph of LeBron James's Wikipedia bio, pasted it into the console, clicked Analyze, and it highlighted the entities for me.

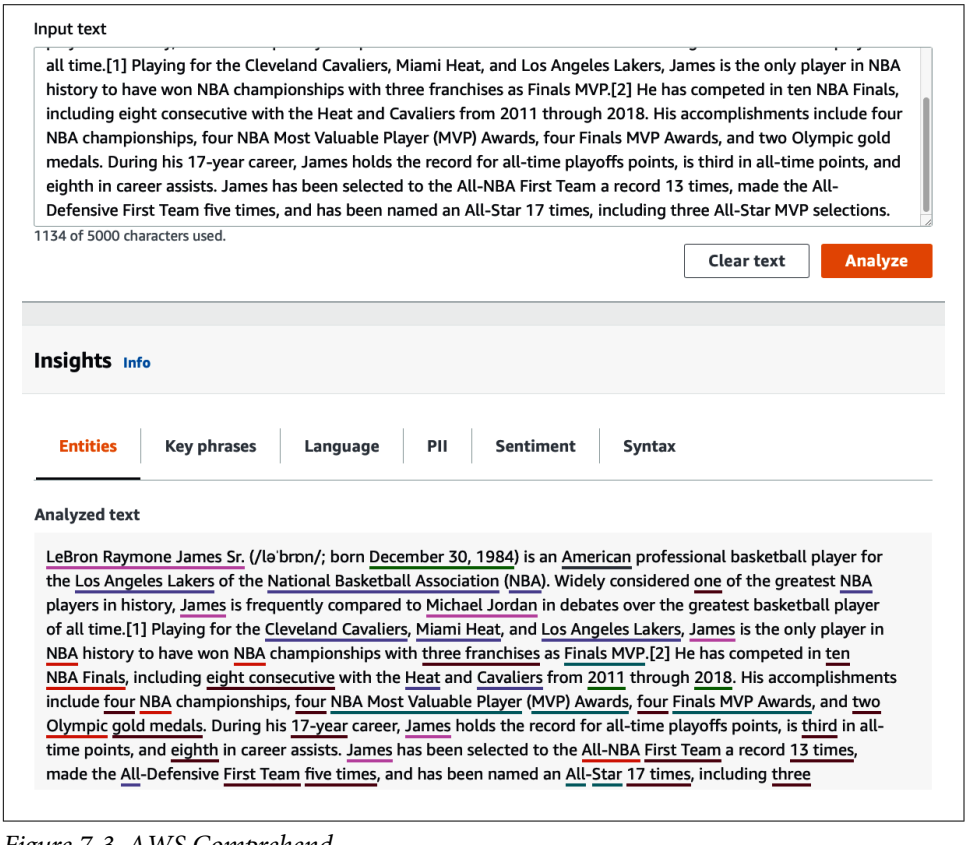

Figure 7-3. AWS Comprehend

Other use cases like reviewing medical records or determining the sentiment of a customer service response are equally straightforward with AWS Comprehend and the boto3 Python SDK. Next, let's cover a "hello world" project for DevOps on AWS, deploying a static website using Amazon S3.

Using Hugo static S3 websites

In the following scenario, an excellent way to explore AWS is to "walk around" the console, just as you would walk around Costco in awe the first time you visit. You do this by first looking at the foundational components of AWS, i.e., IaaS (infrastructure as code). These core services include AWS S3 object storage and AWS EC2 virtual machines.

Here, I'll walk you through deploying a Hugo website on AWS S3 static website hosting using "hello world" as an example. The reason for doing a hello world with Hugo is that it is relatively simple to set up and will give you a good understanding of host services using core infrastructure. These skills will come in handy later when you learn about deploying machine learning applications using continuous delivery.

It is worth noting that Amazon S3 is low cost yet highly reliable. The pricing of S3 approaches a penny per GB. This low-cost yet highly reliable infrastructure is one of the reasons cloud computing is so compelling.

You can view the whole project in the GitHub repo. Notice how GitHub is the source of truth for the website because the entire project consists of text files: markdown files, the Hugo template, and the build commands for the AWS Code build server. Additionally, you can walk through a screencast of the continuous deployment there as well. Figure 7-4 shows the high-level architecture of this project.

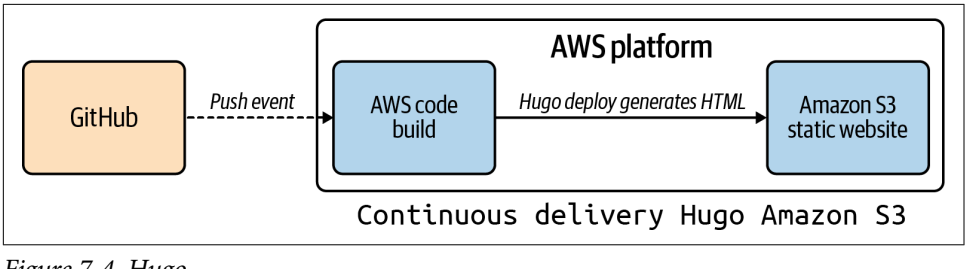

Figure 7-4. Hugo

The short version of how this project works is through the magic of the *buildspec.yml* file. Let's take a look at how this works in the following example. First, note that the `hugo` binary installs, then the `hugo` command runs to generate HTML files from the checked-out repo. Finally, the `aws` command `aws s3 sync --delete public s3://dukefeb1` is the entire deployment process due to the power of S3 bucket hosting:

```
version: 0.1

environment_variables:
  plaintext:
    HUGO_VERSION: "0.79.1"

phases:
  install:
    commands:
      - cd /tmp
      - wget https://github.com/gohugoio/hugo/releases/download/v0.80.0/\
 hugo_extended_0.80.0_Linux-64bit.tar.gz
      - tar -xzf hugo_extended_0.80.0_Linux-64bit.tar.gz
      - mv hugo /usr/bin/hugo
      - cd
      - rm -rf /tmp/*
  build:
    commands:
      - rm -rf public
      - hugo
  post_build:
    commands:
      - aws s3 sync --delete public s3://dukefeb1
      - echo Build completed on `date`
```

Another way of describing a build system file is that it is a recipe. The information in the build configuration files is a "how-to" to perform the same actions in an AWS Cloud9 development environment.

As discussed in Chapter 2, AWS Cloud9 holds a special place in my heart because it solves a particular problem. Cloud-based development environments let you develop in the exact location where all of the action takes place. The example shows how powerful this concept is. Check out the code, test it in the cloud, and verify the same tool's deployment. In Figure 7-5 the AWS Cloud9 environment invokes a Python microservice.

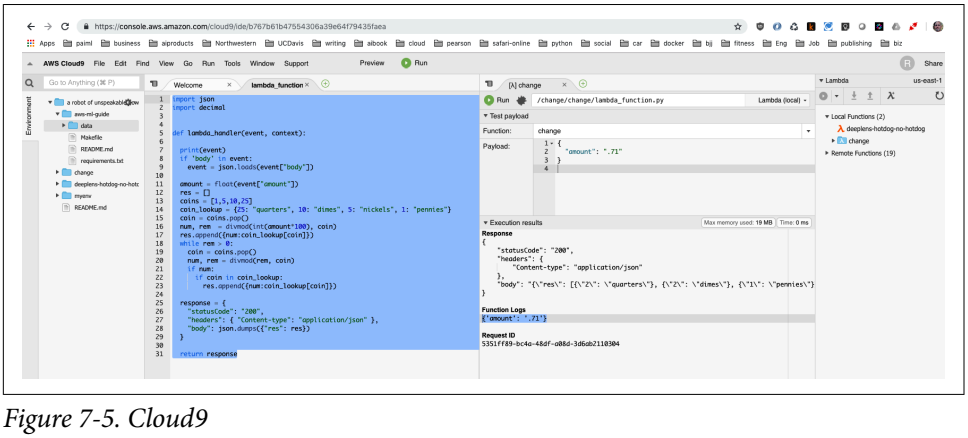

Figure 7-5. Cloud9

You can watch a walkthrough of Hugo deployment on AWS on the O'Reilly platform, as well follow an additional, more detailed guide on the Pragmatic AI Labs website.

With the foundations of continuous delivery behind us, let's get into serverless on the AWS Platform.

Serverless Cookbook

Serverless is a crucial methodology for MLOps. In Chapter 2, I brought up the importance of Python functions. A Python function is a unit of work that can both take an input and optionally return an output. If a Python function is like a toaster, where you put in some bread, it heats the bread, and ejects toast, then serverless is the source of electricity.

A Python function needs to run somewhere, just like a toaster needs to plug into something to work. This concept is what serverless does; it enables code to run in the cloud. The most generic definition of serverless is code that runs without servers. The servers themselves are abstracted away to allow a developer to focus on writing functions. These functions do specific tasks, and these tasks could be chained together to build more complex systems, like servers that respond to events.

The function is the center of the universe with cloud computing. In practice, this means anything that is a function could map into a technology that solves a problem: containers, Kubernetes, GPUs, or AWS Lambda. As you can see in Figure 7-6, there is a rich ecosystem of solutions in Python that map directly to a function.

The lowest level service that performs serverless on the AWS platform is AWS Lambda. Let's take a look at a few examples from this repo.

First, one of the simpler Lambda functions to write on AWS is a Marco Polo function. A Marco Polo function takes in an event with a name in it. So, for example, if the event name is "Marco," it returns "Polo." If the event name is something else, it returns "No!"

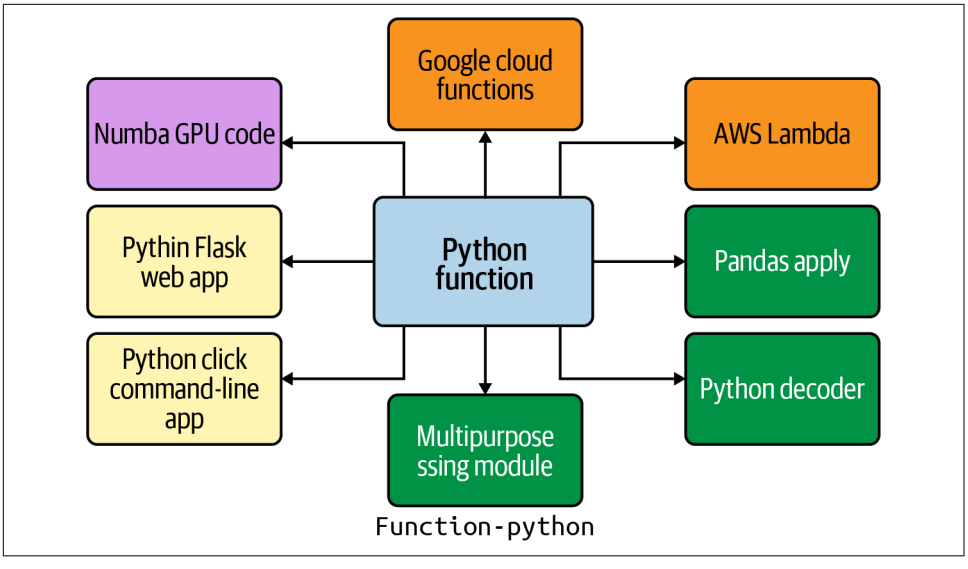

Figure 7-6. Python functions

Growing up as a teenager in the 1980s and 1990s, Marco Polo was a typical game to play in the summer swimming pool. When I worked as a camp counselor at a pool near my house it was a favorite of the kids I supervised. The game works by everyone getting into a pool and one person closing their eyes and yelling "Marco." Next, the other players in the pool must say, "Polo." The person with their eyes closed uses sound to locate someone to tag. Once someone is tagged, then they are "It."

Here is the AWS Lambda Marco Polo code; note that an `event` passes into the `lambda_handler`:

```
def lambda_handler(event, context):
    print(f"This was the raw event: {event}")
    if event["name"] == "Marco":
        print(f"This event was 'Marco'")
        return "Polo"
    print(f"This event was not 'Marco'")
    return "No!"
```

With serverless cloud computing, think about a lightbulb in your garage. A lightbulb can be turned on many ways, such as manually via the light switch or automatically via the garage door open event. Likewise, an AWS Lambda responds to many signals as well.

Let's enumerate the ways both lightbulbs and lambdas can trigger:

- Lightbulbs
  - — Manually flip on the switch
  - — Via the garage door opener
  - — Nightly security timer that turns on the light at midnight until 6 a.m.
- AWS Lambda
  - — Manually invoke via console, AWS command line, or AWS Boto3 SDK
  - — Respond to S3 events like uploading a file to a bucket
  - — Timer invokes nightly to download data

What about a more complex example? With AWS Lambda, it is straightforward to integrate an S3 Trigger with computer vision labeling on all new images dropped in a folder, with a trivial amount of code:

```
import boto3
from urllib.parse import unquote_plus

def label_function(bucket, name):
    """This takes an S3 bucket and a image name!"""
    print(f"This is the bucketname {bucket} !")
    print(f"This is the imagename {name} !")
    rekognition = boto3.client("rekognition")
    response = rekognition.detect_labels(
        Image={"S3Object": {"Bucket": bucket, "Name": name,}},
    )
    labels = response["Labels"]
    print(f"I found these labels {labels}")
    return labels


def lambda_handler(event, context):
    """This is a computer vision lambda handler"""

    print(f"This is my S3 event {event}")
    for record in event['Records']:
        bucket = record['s3']['bucket']['name']
        print(f"This is my bucket {bucket}")
        key = unquote_plus(record['s3']['object']['key'])
        print(f"This is my key {key}")

    my_labels = label_function(bucket=bucket,
        name=key)
    return my_labels
```

Finally, you can chain multiple AWS Lambda functions together via AWS Step Functions:

```
{
  "Comment": "This is Marco Polo",
  "StartAt": "Marco",
  "States": {
    "Marco": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:561744971673:function:marco20",
      "Next": "Polo"
    },
    "Polo": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:561744971673:function:polo",
      "Next": "Finish"
    },
    "Finish": {
      "Type": "Pass",
      "Result": "Finished",
      "End": true
    }
  }
}
```

You can see this workflow in action in Figure 7-7.

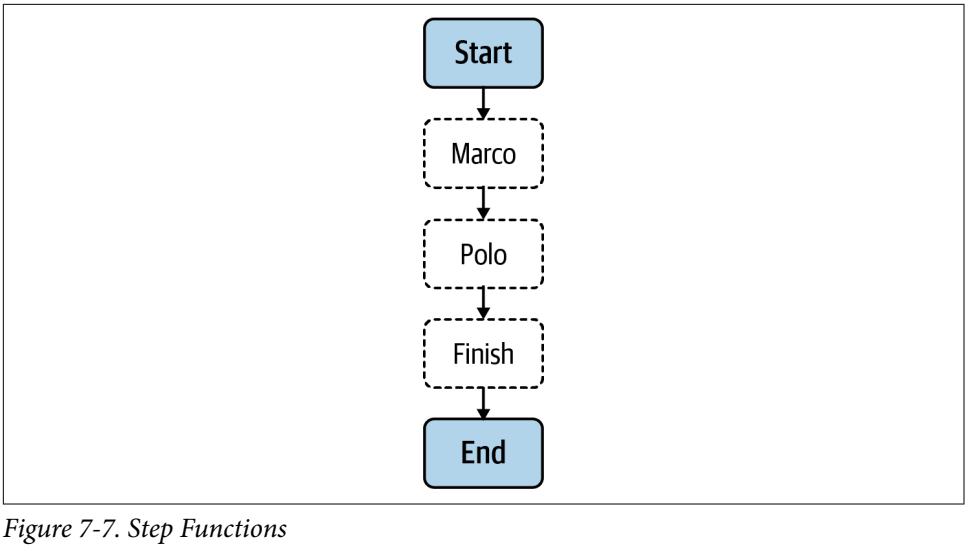

Figure 7-7. Step Functions

Even more fun is the fact that you can call AWS Lambda functions via a CLI. Here is an example:

```
aws lambda invoke \
   --cli-binary-format raw-in-base64-out \
  --function-name marcopython \
  --payload '{"name": "Marco"}' \
  response.json
```

It is important to always refer to the latest AWS documentation for the CLI as it is an actively moving target. As of the writing of this book, the current CLI version is V2, but you may need to adjust the command line examples as things change in the future. You can find the latest documentation at the AWS CLI Command Reference site.

The response of the payload is as follows:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
(.venv) [cloudshell-user@ip-10-1-14-160 ~]$ cat response.json
"Polo"(.venv) [cloudshell-user@ip-10-1-14-160 ~]$
```

For a more advanced walkthrough of AWS Lambda using Cloud9 and the AWS SAM (Serverless Application Model), you can view a walkthrough of a small Wikipedia microservice on the Pragmatic AI Labs YouTube Channel or the O'Reilly Learning Platform.

AWS Lambda is perhaps the most valuable and flexible type of computing you can use to serve out predictions for machine learning pipelines or wrangle events in the service of an MLOps process. The reason for this is the speed of development and testing. Next, let's talk about a couple of CaaS, or container as a service, offerings.

AWS CaaS

Fargate is a container as a service offering from AWS that allows developers to focus on building containerized microservices. For example, in Figure 7-8, when this microservice works in the container, the entire runtime, including the packages necessary for deployment, will work in a new environment. The cloud platform handles the rest of the deployment.

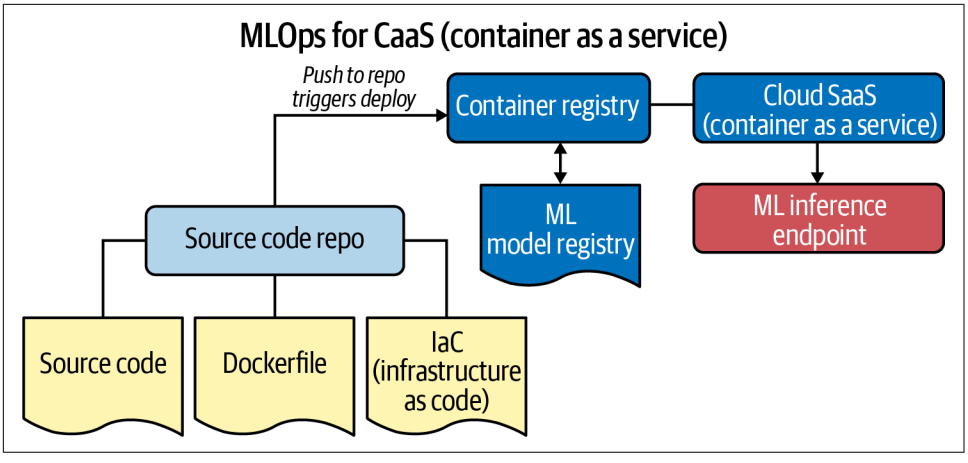

Figure 7-8. MLOps for CaaS

Containers solve many problems that have plagued the software industry. So as a general rule, it is a good idea to use them for MLOps projects. Here is a partial list of the advantages of containers in projects:

- Allows the developer to mimic the production service locally on their desktop
- Allows easy software runtime distribution to customers through public container registries like Docker Hub, GitHub Container Registry, and Amazon Elastic Container Registry
- Allows for GitHub or a source code repo to be "source of truth" and contain all aspects of microservice: model, code, IaC, and runtime
- Allows for easy production deployment via CaaS services

Let's look at how you can build a microservice that returns the correct change using Flask. Figure 7-9 shows a development workflow on AWS Cloud9. Cloud9 is the development environment; a container gets built and pushed to ECR. Later that container runs in ECS.

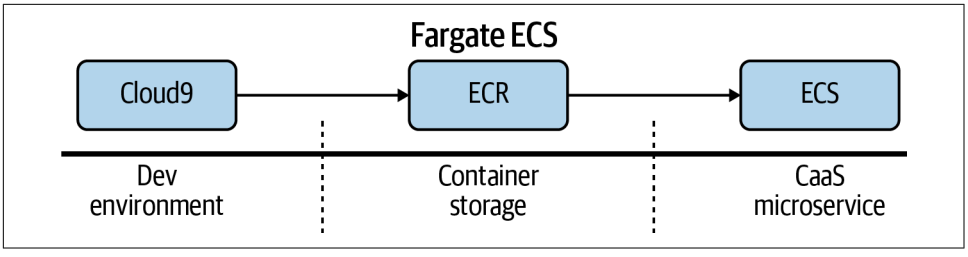

Figure 7-9. ECS workflow

The following is the Python code for *app.py*:

```
from flask import Flask
from flask import jsonify
app = Flask(__name__)

def change(amount):
    # calculate the resultant change and store the result (res)
    res = []
    coins = [1,5,10,25] # value of pennies, nickels, dimes, quarters
    coin_lookup = {25: "quarters", 10: "dimes", 5: "nickels", 1: "pennies"}

    # divide the amount*100 (the amount in cents) by a coin value
    # record the number of coins that evenly divide and the remainder
    coin = coins.pop()
    num, rem  = divmod(int(amount*100), coin)
    # append the coin type and number of coins that had no remainder
    res.append({num:coin_lookup[coin]})

    # while there is still some remainder, continue adding coins to the result
    while rem > 0:
        coin = coins.pop()
        num, rem = divmod(rem, coin)
        if num:
            if coin in coin_lookup:
                res.append({num:coin_lookup[coin]})
    return res

@app.route('/')
def hello():
    """Return a friendly HTTP greeting."""
    print("I am inside hello world")
    return 'Hello World! I can make change at route: /change'

@app.route('/change/<dollar>/<cents>')
def changeroute(dollar, cents):
    print(f"Make Change for {dollar}.{cents}")
    amount = f"{dollar}.{cents}"
    result = change(float(amount))
    return jsonify(result)
```

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080, debug=True)
```

Notice that Flask web microservice responds to change requests via web requests to the URL pattern `/change/<dollar>/<cents>`. You can view the complete source code for this Fargate example in the following GitHub repo. The steps are as follows:

1. Setup app: virtualenv + `make all`
2. Test app local: `python app.py`
3. Curl it to test: `curl localhost:8080/change/1/34`
4. Create ECR (Amazon Container Registry)

   In Figure 7-10, an ECR repository enables later Fargate deployment.

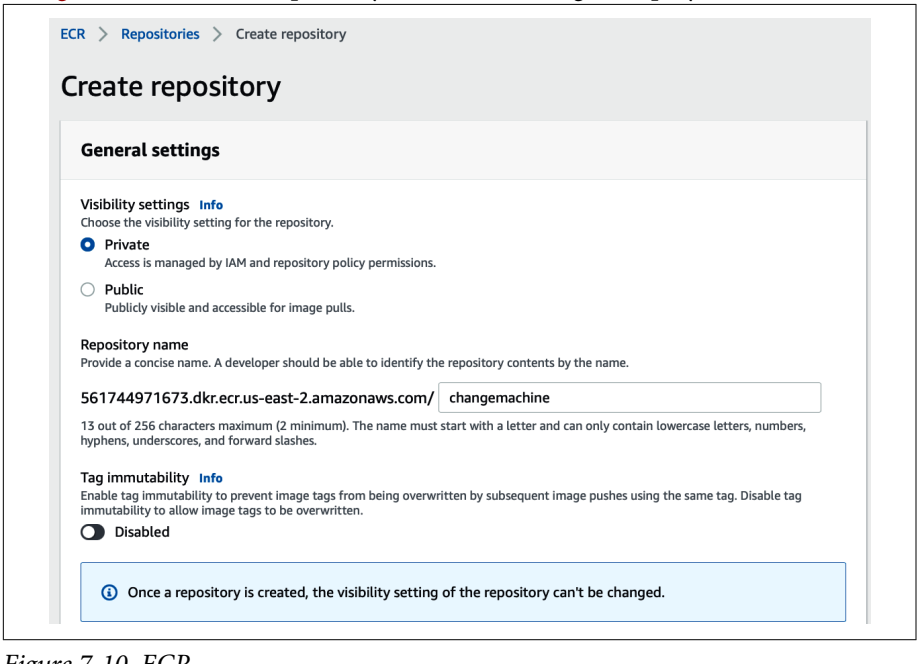

Figure 7-10. ECR

5. Build container
6. Push container
7. Run docker local: `docker run -p 8080:8080 changemachine`
8. Deploy to Fargate
9. Test the public service

Any cloud service will have rapid, constant changes in functionality, so it is best to read the current documentation. The current Fargate documentation is a great place to read more about the latest ways to deploy to the service.

You can also, optionally, watch a complete walkthrough of a Fargate deployment on the O'Reilly platform.

Another option for CaaS is AWS App Runner, which further simplifies things. For example, you can deploy straight from source code or point to a container. In Figure 7-11, AWS App Runner creates a streamlined workflow that connects a source repo, deploy environment, and a resulting secure URL.

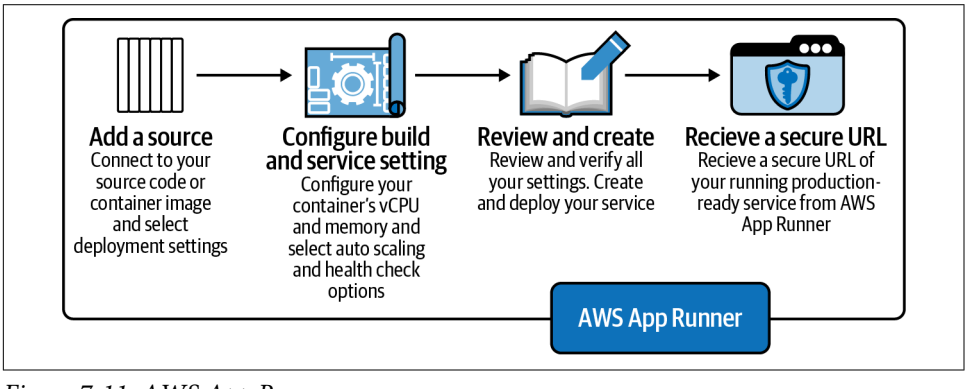

Figure 7-11. AWS App Runner

This repository can easily be converted to an AWS App Runner method in the AWS wizard with the following steps:

1. To build the project, use the command: `pip install -r requirements.txt`.
2. To run the project, use: `python app.py`.
3. Finally, configure the port to use: `8080`.

A key innovation, shown in Figure 7-12, is the ability to connect many different services on AWS, i.e., core infrastructure, like AWS CloudWatch, Load Balancers, Container Services, and API Gateways into one complete offering.

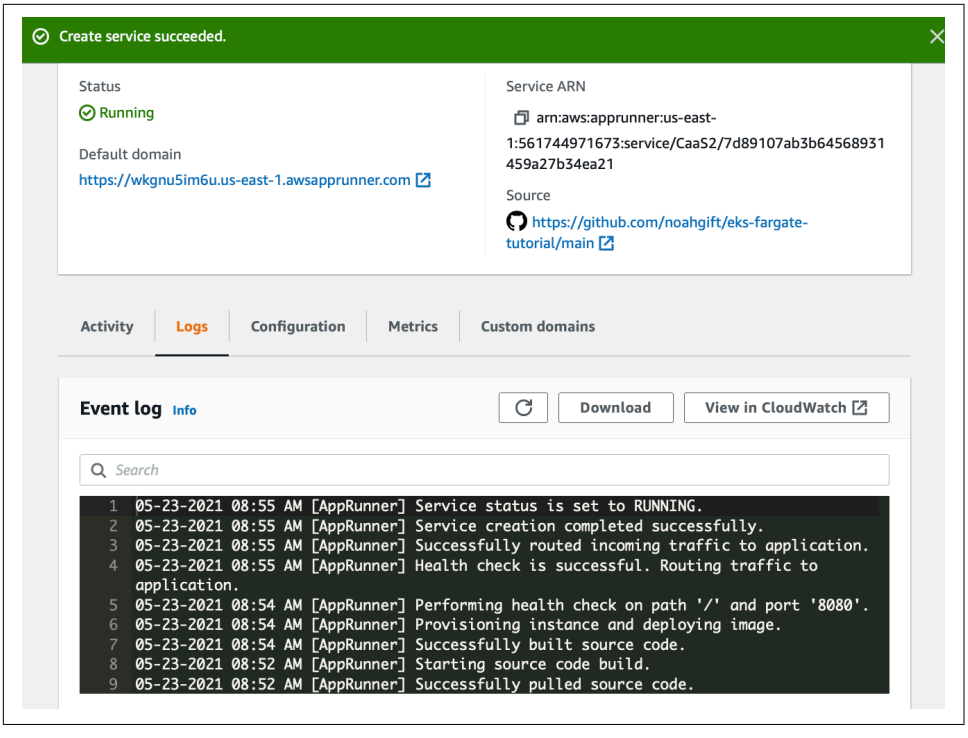

Figure 7-12. AWS App Runner service created

The final deployed service in Figure 7-13 shows a secure URL and the ability to invoke the endpoint and return correct change.

Figure 7-13. AWS App Runner deployed

Why is this helpful magic? In a nutshell, all of the logic, including perhaps a machine learning model, are in one repo. Thus, this is a compelling style for building MLOps-friendly products. In addition, one of the more complex aspects of delivering a machine learning application to production is microservice deployment. AWS App Runner makes most of the complexity disappear, capturing time for other parts of the MLOps problem. Next, let's discuss how AWS deals with computer vision.

Computer vision

I teach an applied computer vision course at Northwestern's graduate data science program. This class is delightful to teach because we do the following:

- Weekly video demos
- Focus on problem-solving versus coding or modeling
- Use high-level tools like AWS DeepLens, a deep learning–enabled video camera

In practice, this allows a rapid feedback loop that focuses on the problem versus the technology to solve the problem. One of the technologies in use is the AWS Deep Lens device as shown in Figure 7-14. The device is a complete computer vision hardware developer kit in that it contains a 1080p camera, operating system, and wireless capabilities. In particular, this solves the prototyping problem of computer vision.

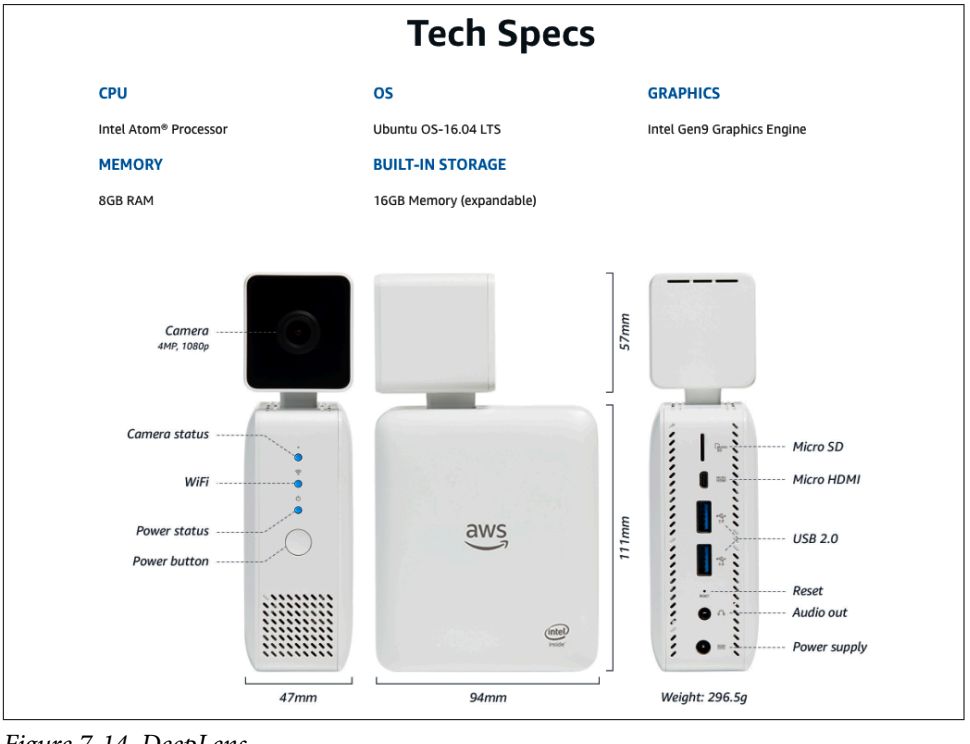

Figure 7-14. DeepLens

Once AWS DeepLens starts capturing video, it splits the video into two streams. The Project Stream shown in Figure 7-15 adds real-time annotation and sends the packets to the MQ Telemetry Transport (MQTT) service, which is a publish-subscribe network protocol.

Figure 7-15. Detect

In Figure 7-16, the MQTT packets arrive in real time as the objects get detected in the stream.

The DeepLens is a "plug and play" technology since it tackles perhaps the most challenging part of the problem of building a real-time computer vision prototyping system—capturing the data, and sending it somewhere. The "fun" factor of this is how easy it is to go from zero to one building solutions with AWS DeepLens. Next, let's get more specific and move beyond just building microservices and into building microservices that deploy machine learning code.

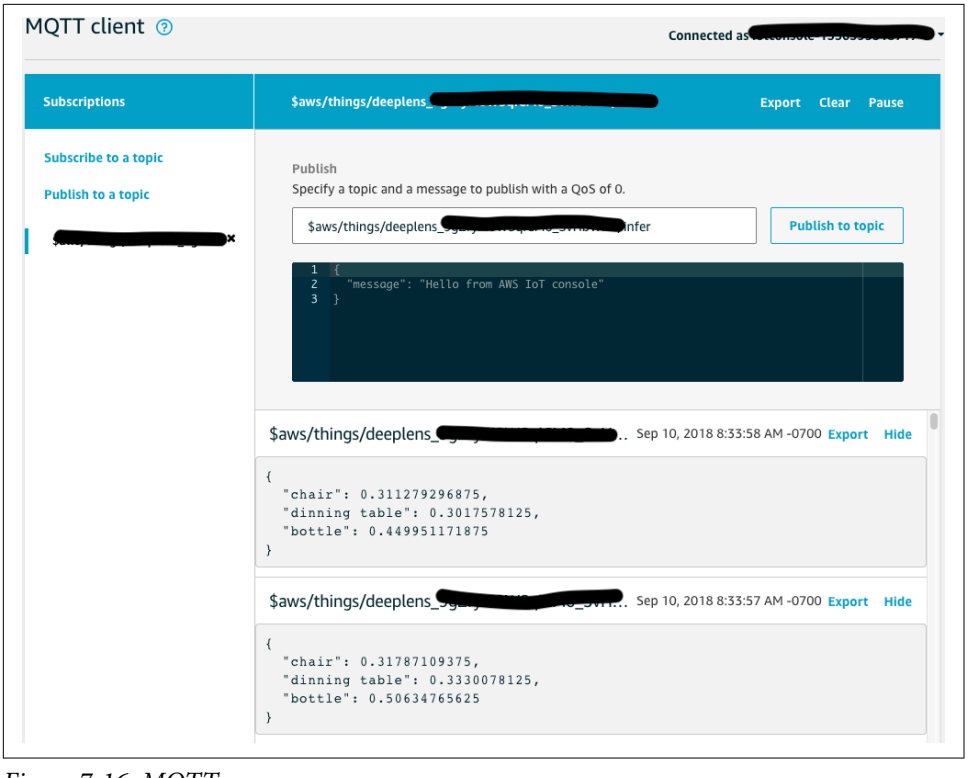

Figure 7-16. MQTT

MLOps on AWS

One way to get started with MLOps on AWS is to consider the following question. When given a machine learning problem with three constraints—prediction accuracy, explainability, and operations, which two would you focus on to achieve success, and in what order?

Many academic-focused data scientists immediately jump to prediction accuracy. Building ever-better prediction models is a fun challenge, like playing Tetris. Also, the modeling is glamorous and a coveted aspect of the job. Data scientists like to show how accurate they can train an academic model using increasingly sophisticated techniques.

The entire Kaggle platform works on increasing prediction accuracy, and there are monetary rewards for the precise model. A different approach is to focus on operationalizing the model. This approach's advantage is that later, model accuracy can improve alongside improvements in the software system. Just as the Japanese automobile industry focused on Kaizen or continuous improvement, an ML system can focus on reasonable initial prediction accuracy and improve quickly.

The culture of AWS supports this concept in the leadership principles of "Bias for Action" and "Deliver Results." "Bias for Action" refers to defaulting to speed and delivery results, focusing on the critical inputs to a business, and delivering results quickly. As a result, the AWS products around machine learning, like AWS SageMaker, show the spirit of this culture of action and results.

Continuous delivery (CD) is a core component in MLOps. Before you can automate delivery for machine learning, the Microservice itself needs automation. The specifics change depending on the type of AWS service involved. Let's start with an end-to-end example.

In the following example, an Elastic Beanstalk Flask app continuously deploys using all AWS technology from AWS Code Build to AWS Elastic Beanstalk. This "stack" is also ideal for deploying ML models. Elastic Beanstalk is a platform as a service technology offered by AWS that streamlines much of the work of deploying an application.

In Figure 7-17, notice that AWS Cloud9 is a recommended starting point for development. Next, a GitHub repository holds the source code for the project, and as change events occur, it triggers the cloud native build server, AWS CodeBuild. Finally, the AWS Code Build process runs continuous integration, tests for the code, and provides continuous delivery to AWS Elastic Beanstalk.

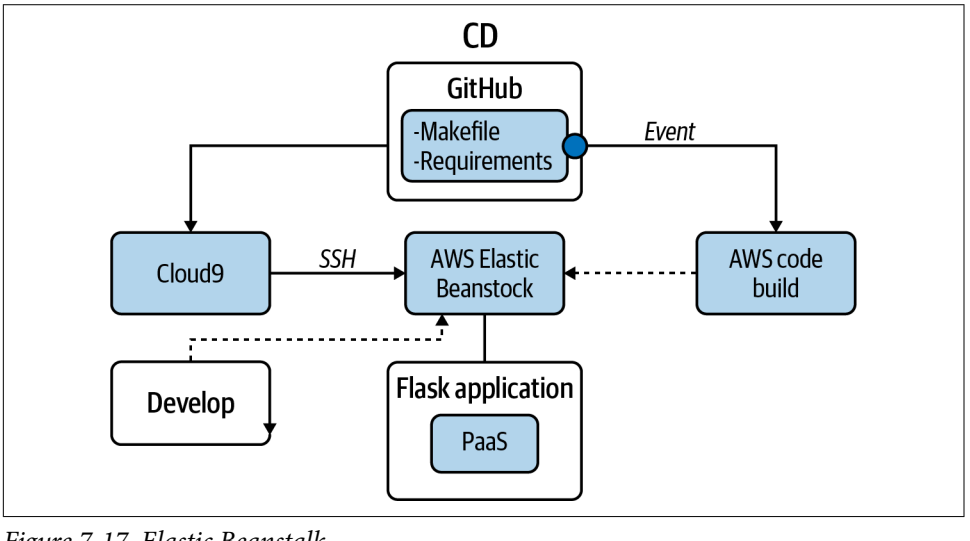

Figure 7-17. Elastic Beanstalk

The source code and a walkthrough of this example are at the following links:

- Source Code
- O'Reilly Platform Video Walkthrough

To replicate this exact project, you can do the following steps:

1. Check out the repository in AWS Cloud9 or AWS CloudShell if you have strong command line skills.
2. Create a Python virtualenv and source it and run `make all`:

```
python3 -m venv ~/.eb
source ~/.eb/bin/activate
make all
```

   Note that `awsebcli` installs via requirements, and this tool controls Elastic Beanstalk from the CLI.

3. Initialize new `eb` app:

```
eb init -p python-3.7 flask-continuous-delivery --region us-east-1
```

   Optionally, you use `eb init` to create SSH keys to shell into the running instances.

4. Create remote eb instance:

```
eb create flask-continuous-delivery-env
```

5. Setup AWS Code Build Project. Note your Makefile needs to reflect your project names:

```
version: 0.2

phases:
  install:
    runtime-versions:
      python: 3.7
  pre_build:
    commands:
      - python3.7 -m venv ~/.venv
      - source ~/.venv/bin/activate
      - make install
      - make lint

  build:
    commands:
      - make deploy
```

After you get the project working with continuous deployment, you are ready to move to the next step, deploying an ML model. I highly recommend getting a "hello world" type project working with continuous deployment like this one before you proceed directly into a complex ML project when you are learning new technology. Next, let's look at an intentionally simple MLOps Cookbook that is the foundation for many new AWS Services deployments.

MLOps Cookbook on AWS

With the foundational components out of the way, let's look at a basic machine learning recipe and apply it to several scenarios. Notice that this core recipe is deployable to many services on AWS and many other cloud environments. This following MLOps Cookbook project is intentionally spartan, so the focus is on deploying machine learning. For example, this project predicts height from a weight input for Major League Baseball players.

In Figure 7-18, GitHub is the source of truth and contains the project scaffolding. Next, the build service is GitHub Actions, and the container service is GitHub Container Registry. Both of these services can easily replace any similar offering in the cloud. In particular, on the AWS Cloud, you can use AWS CodeBuild for CI/CD and AWS ECR (Elastic Container Registry). Finally, once a project has been "containerized," it opens up the project to many deployment targets. On AWS, these include AWS Lambda, AWS Elastic Beanstalk, and AWS App Runner.

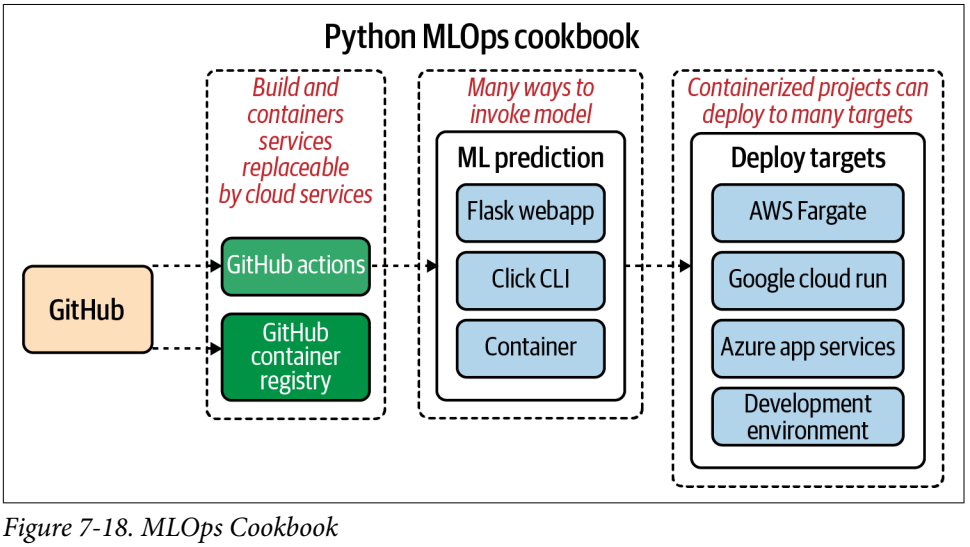

Figure 7-18. MLOps Cookbook

The following files are all useful to build solutions in many different recipes:

Makefile
: The Makefile is both a list of recipes and a way to invoke those recipes. View the Makefile in the example GitHub project.

requirements.txt
: The requirements file contains the list of Python packages for the project. Typically these packages are "pinned" to a version number, which limits unexpected package dependencies. View the requirements.txt in the example GitHub project.

cli.py
: This command line shows how an ML library can also be invoked from the CLI, not just via a web application. View the cli.py in the example GitHub project.

utilscli.py
: The utilscli.py is a utility that allows the user to invoke different endpoints, i.e., AWS, GCP, Azure, or any production environment. Most machine learning algorithms require data to be scaled. This tool simplifies scaling the input and scaling back out the output.View the utilscli.py in the example GitHub project.

app.py
: The application file is the Flask web microservice that accepts and returns a JSON prediction result via the `/predict` URL endpoint. View the app.py in the example GitHub project.

mlib.py
: The model handling library does much of the heavy lifting in a centralized location. This library is intentionally very basic and doesn't solve more complicated issues like caching loading of the model or other production issues unique to production deployment. View the mlib.py in the example GitHub project.

htwtmlb.csv
: A CSV file is helpful for input scaling. View the htwtmlb.csv in the example GitHub project.

model.joblib
: This model is exported from sklearn but could easily be in another format such as ONNX or TensorFlow. Other real-world production considerations could be keeping this model in a different location like Amazon S3, in a container, or even hosted by AWS SageMaker. View the model.joblib in the example GitHub project.

Dockerfile
: This file enables project containerization and, as a result, opens up many new deployment options, both on the AWS platform as well as other clouds. View the Dockerfile in the example GitHub project.

Baseball_Predictions_Export_Model.ipynb
: The Jupyter notebook is a crucial artifact to include in a machine learning project. It shows another developer the thinking behind creating the model and provides valuable context for maintaining the project in production. View the Baseball_Predictions_Export_Model.ipynb in the example GitHub project.

These project artifacts are helpful as an educational tool in explaining MLOps but may be different or more complex in a unique production scenario. Next, let's discuss how CLI (command line interface) tools help operationalize a machine learning project.

CLI Tools

In this project, there are two CLI tools. First, the main *cli.py* is the endpoint that serves out predictions. For example, to predict the height of an MLB player, you use the following command to create a forecast: `./cli.py --weight 180`. Notice in Figure 7-19 that the command line option of `--weight` allows the user to test out many new prediction inputs quickly.

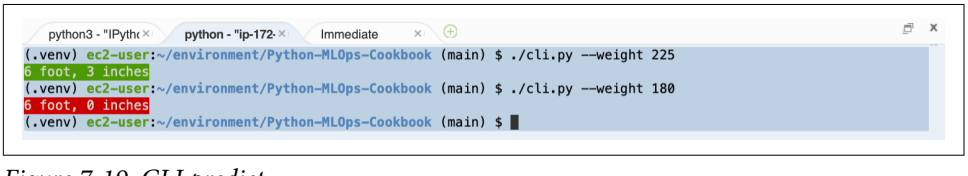

Figure 7-19. CLI predict

So how does this work? Most of the "magic" is via a library that does the heavy lifting of scaling the data, making the prediction, then doing an inverse transform back:

```
"""MLOps Library"""

import numpy as np
import pandas as pd
from sklearn.linear_model import Ridge
import joblib
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import logging

logging.basicConfig(level=logging.INFO)

import warnings
```

```
warnings.filterwarnings("ignore", category=UserWarning)


def load_model(model="model.joblib"):
    """Grabs model from disk"""

    clf = joblib.load(model)
    return clf


def data():
    df = pd.read_csv("htwtmlb.csv")
    return df


def retrain(tsize=0.1, model_name="model.joblib"):
    """Retrains the model

    See this notebook: Baseball_Predictions_Export_Model.ipynb
    """
    df = data()
    y = df["Height"].values  # Target
    y = y.reshape(-1, 1)
    X = df["Weight"].values  # Feature(s)
    X = X.reshape(-1, 1)
    scaler = StandardScaler()
    X_scaler = scaler.fit(X)
    X = X_scaler.transform(X)
    y_scaler = scaler.fit(y)
    y = y_scaler.transform(y)
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=tsize, random_state=3
    )
    clf = Ridge()
    model = clf.fit(X_train, y_train)
    accuracy = model.score(X_test, y_test)
    logging.debug(f"Model Accuracy: {accuracy}")
    joblib.dump(model, model_name)
    return accuracy, model_name


def format_input(x):
    """Takes int and converts to numpy array"""

    val = np.array(x)
    feature = val.reshape(-1, 1)
    return feature


def scale_input(val):
    """Scales input to training feature values"""
```

```
    df = data()
    features = df["Weight"].values
    features = features.reshape(-1, 1)
    input_scaler = StandardScaler().fit(features)
    scaled_input = input_scaler.transform(val)
    return scaled_input


def scale_target(target):
    """Scales Target 'y' Value"""

    df = data()
    y = df["Height"].values  # Target
    y = y.reshape(-1, 1)  # Reshape
    scaler = StandardScaler()
    y_scaler = scaler.fit(y)
    scaled_target = y_scaler.inverse_transform(target)
    return scaled_target


def height_human(float_inches):
    """Takes float inches and converts to human height in ft/inches"""

    feet = int(round(float_inches / 12, 2))  # round down
    inches_left = round(float_inches - feet * 12)
    result = f"{feet} foot, {inches_left} inches"
    return result


def human_readable_payload(predict_value):
    """Takes numpy array and returns back human readable dictionary"""

    height_inches = float(np.round(predict_value, 2))
    result = {
        "height_inches": height_inches,
        "height_human_readable": height_human(height_inches),
    }
    return result


def predict(weight):
    """Takes weight and predicts height"""

    clf = load_model()  # loadmodel
    np_array_weight = format_input(weight)
    scaled_input_result = scale_input(np_array_weight)
    scaled_height_prediction = clf.predict(scaled_input_result)
    height_predict = scale_target(scaled_height_prediction)
    payload = human_readable_payload(height_predict)
    predict_log_data = {
        "weight": weight,
        "scaled_input_result": scaled_input_result,
```

```
        "scaled_height_prediction": scaled_height_prediction,
        "height_predict": height_predict,
        "human_readable_payload": payload,
    }
    logging.debug(f"Prediction: {predict_log_data}")
    return payload
```

Next, the Click framework wraps the library calls to *mlib.py* and makes a clean interface to serve out predictions. There are many advantages to using command line tools as the primary interface for interacting with machine learning models. The speed to develop and deploy a command line machine learning tool is perhaps the most important:

```
#!/usr/bin/env python
import click
from mlib import predict


@click.command()
@click.option(
    "--weight",
    prompt="MLB Player Weight",
    help="Pass in the weight of a MLB player to predict the height",
)
def predictcli(weight):
    """Predicts Height of an MLB player based on weight"""

    result = predict(weight)
    inches = result["height_inches"]
    human_readable = result["height_human_readable"]
    if int(inches) > 72:
        click.echo(click.style(human_readable, bg="green", fg="white"))
    else:
        click.echo(click.style(human_readable, bg="red", fg="white"))


if __name__ == "__main__":
    # pylint: disable=no-value-for-parameter
    predictcli()
```

The second CLI tool is *utilscli.py*, which performs model retraining and could serve as the entry point to do more tasks. For example, this version doesn't change the default `model_name`, but you could add that as an option by forking this repo:

```
./utilscli.py retrain --tsize 0.4
```

Notice that the *mlib.py* again does the heavy lifting, but the CLI provides a convenient way to do rapid prototyping of an ML model:

```
#!/usr/bin/env python
import click
import mlib
```