16
Integrating ML Systems in Ecosystems

ML systems have gained a lot of popularity for two reasons – their ability to learn from data (which we've explored throughout this book), and their ability to be packaged into web services.

Packaging these ML systems into web services allows us to integrate them into workflows in a very flexible way. Instead of compiling or using dynamically linked libraries, we can deploy ML components that communicate over HTTP protocols using JSON protocols. We have already seen how to use that protocol by using the GPT-3 model that is hosted by OpenAI. In this chapter, we'll explore the possibility of creating a Docker container with a pre-trained ML model, deploying it, and integrating it with other components.

In this chapter, we're going to cover the following main topics:

- ML system of systems – software ecosystems
- Creating web services over ML models using Flask
- Deploying ML models using Docker
- Combining web services into ecosystems

Ecosystems

In the dynamic realm of software engineering, the tools, methodologies, and paradigms are in a constant state of evolution. Among the most influential forces driving this transformation is ML. While ML itself is a marvel of computational prowess, its true genius emerges when integrated into the broader software engineering ecosystems. This chapter delves into the nuances of embedding ML within an ecosystem. Ecosystems are groups of software that work together but are not connected at compile time. A well-known ecosystem is the PyTorch ecosystem, where a set of libraries work together in the context of ML. However, there is much more than that to ML ecosystems in software engineering.

From automated testing systems that learn from each iteration to recommendation engines that adapt to user behaviors, ML is redefining how software is designed, developed, and deployed. However, integrating ML into software engineering is not a mere plug-and-play operation. It demands a rethinking of traditional workflows, a deeper understanding of data-driven decision-making, and a commitment to continuous learning and adaptation.

As we delve deeper into the integration of ML within software engineering, it becomes imperative to discuss two pivotal components that are reshaping the landscape: web services and Docker containers. These technologies, while not exclusive to ML applications, play a crucial role in the seamless deployment and scaling of ML-driven solutions in the software ecosystem.

Web services, especially in the era of microservices architecture, provide a modular approach to building software applications. By encapsulating specific functionalities into distinct services, they allow for greater flexibility and scalability. When combined with ML models, web services can deliver dynamic, real-time responses based on the insights derived from data. For instance, a web service might leverage an ML model to provide personalized content recommendations to users or to detect fraudulent activities in real time.

Docker containers, on the other hand, have revolutionized the way software, including ML models, is packaged and deployed. Containers encapsulate an application along with all its dependencies into a standardized unit, ensuring consistent behavior across different environments. For ML practitioners, this means the painstaking process of setting up environments, managing dependencies, and ensuring compatibility is vastly simplified. Docker containers ensure that an ML model trained on a developer's machine will run with the same efficiency and accuracy on a production server or any other platform.

Furthermore, when web services and Docker containers are combined, they pave the way for ML-driven microservices. Such architectures allow for the rapid deployment of scalable, isolated services that can be updated independently without disrupting the entire system. This is especially valuable in the realm of ML, where models might need frequent updates based on new data or improved algorithms.

In this chapter, we'll learn how to use both technologies to package models and create an ecosystem based on Docker containers. After reading this chapter, we shall have a good understanding of how we can scale up our development by using ML as part of a larger system of systems – ecosystems.

Creating web services over ML models using Flask

In this book, we've mostly focused on training, evaluating, and deploying ML models. However, we did not discuss the need to structure them flexibly. We worked with monolithic software. Monolithic software is characterized by its unified, single code base structure where all the functionalities, from the user interface to data processing, are tightly interwoven and operate as one cohesive unit. This design simplifies initial development and deployment since everything is bundled together and they are compiled together. Any change, however minor, requires the entire application to be rebuilt and redeployed. This makes it problematic when the evolution of contemporary software is fast.

On the other hand, web service-based software, which is often associated with microservices architecture, breaks down the application into smaller, independent services that communicate over the web, typically using protocols such as HTTP and REST. Each service is responsible for a specific functionality and operates independently. This modular approach offers greater flexibility. Services can be scaled, updated, or redeployed individually without affecting the entire system. Moreover, failures in one service don't necessarily bring down the whole application. *Figure 16.1* presents how the difference between these two types of software can be seen:

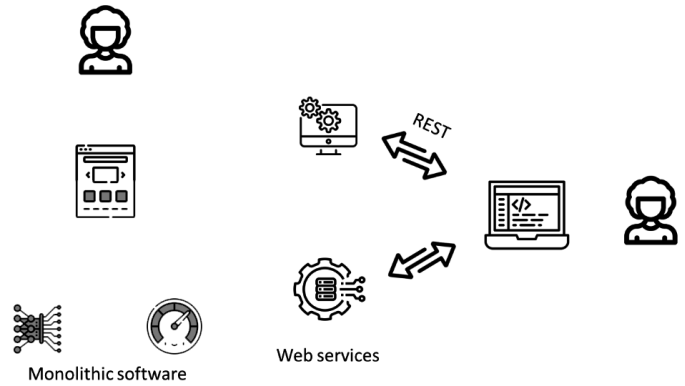

Figure 16.1 – Monolithic software versus web service-based software

On the left-hand side, we have all the components bundled together into one product. Users interact with the product through the user interface. Only one user can interact with the software, and more users require more installations of the software.

On the right-hand side, we have a decentralized architecture where each component is a separate web service. The coordination of these components is done by a thin client. If more users/clients want to use the same services, they can just connect them using the HTTP REST protocol (API).

Here is my first best practice.

Best practice #75

Use web services (RESTful API) when deploying ML models for production.

Although it takes additional effort to create web services, it is worth using them. They provide a great separation of concerns and asynchronous access and also provide great possibilities for load balancing. We can use different servers to run the same web service and therefore balance the load.

So, let's create the first web service using Flask.

Creating a web service using Flask

Flask is a framework that allows us to provide easy access to internal APIs via the REST interface over HTTP protocol. First, we need to install it:

```
pip install flask
pip install flask-restful
```

Once we've installed the interface, we can write our programs. In this example, our first web service calculates the lines of code and complexity of the program sent to it. The following code fragment exemplifies this:

```
from fileinput import filename
from flask import *
from radon.complexity import cc_visit
from radon.cli.harvest import CCHarvester

app = Flask(__name__)

# Dictionary to store the metrics for the file submitted
# Metrics: lines of code and McCabe complexity
metrics = {}


def calculate_metrics(file_path):
    with open(file_path, 'r') as file:
        content = file.read()

    # Count lines of code
    lines = len(content.splitlines())

    # Calculate McCabe complexity
    complexity = cc_visit(content)

    # Store the metrics in the dictionary
    metrics[file_path] = {
        'lines_of_code': lines,
        'mccabe_complexity': complexity
    }


@app.route('/')
def main():
    return render_template("index.html")
```

```
@app.route('/success', methods=['POST'])
def success():
    if request.method == 'POST':
        f = request.files['file']

        # Save the file to the server
        file_path = f.filename
        f.save(file_path)

        # Calculate metrics for the file
        calculate_metrics(file_path)

        # Return the metrics for the file
        return metrics[file_path]


@app.route('/metrics', methods=['GET'])
def get_metrics():
    if request.method == 'GET':
        return metrics

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

First, the code requires a few imports and then initializes the application in `app = Flask(__name__)`. Then, it creates the routes – that is, the places where the program will be able to communicate via the REST API:

- `@app.route('/')`: This is a decorator that defines a route for the root URL (`"/"`). When users access the root URL, it renders the `"index.html"` template.
- `@app.route('/success', methods=['POST'])`: This decorator defines a route for the `"/success"` URL, which expects HTTP POST requests. This route is used to handle file uploads, count lines of code, and calculate McCabe complexity.
- `@app.route('/metrics', methods=['GET'])`: This decorator defines a route for the `"/metrics"` URL, which expects HTTP GET requests. It is used to retrieve and display the metrics.
- `def main()`: This function is associated with the root (`"/"`) route. It returns an HTML template called `"index.html"` when users access the root URL.

- `def success()`: This function is associated with the `"/success"` route, which handles file uploads:
  - It checks if the request method is POST
  - It saves the uploaded file to the server
  - It counts the lines of code in the uploaded file
  - It calculates the McCabe complexity using the radon library
  - It stores the metrics (lines of code and McCabe complexity) in the metrics dictionary
  - It returns the metrics for the uploaded file as a JSON response
- `def get_metrics()`: This function is associated with the "/metrics" route:
  - It checks if the request method is GET.
  - It returns the entire metrics dictionary as a JSON response. This is used for debugging purposes to see the metrics for all files that are uploaded during the session.
- `if __name__ == '__main__':`: This block ensures that the web application is only run when this script is executed directly (not when it's imported as a module).
- `app.run(host='0.0.0.0', debug=True)`: This starts the Flask application in debug mode, allowing you to see detailed error messages during development.

Then, the application is executed – `app.run(debug=True)` starts the Flask application with debugging mode enabled. This means that any changes made to the code will automatically reload the server, and any errors will be displayed in the browser. Once we execute it, the following web page appears (note that the code of the page has to be in the `templates` subfolder and should contain the following code):

```
<html>
<head>
    <title>Machine learning best practices for software engineers:
Chapter 16 - Upload a file to make predictions</title>
</head>
<body>
    <h1>Machine learning best practices for software engineers -
Chapter 16</h1>

    <p>This page allows to upload a file to a web service that has
been written using Flask. The web application behind this interface
calculates metrics that are important for the predictions. It returns
a JSON string with the metrics.  </p>

    <p>We need another web app that contains the model in order to
```

```
actually obtain predictions if the file can contain defects. </p>

    <h1>Upload a file to make predictions</h1>
    <p>The file should be a .c or .cpp file</p>

    <form action = "/success" method = "post" enctype="multipart/form-
data">
        <input type="file" name="file" />
        <input type = "submit" value="Upload">
    </form>

    <p>Disclaimer: the container saves the file it its local folder,
so don't send any sensitive files for analysis.</p>

    <p>This is a research prototype</p>
</body>
</html>
```

The page contains a simple form that allows us to upload the file to the server:

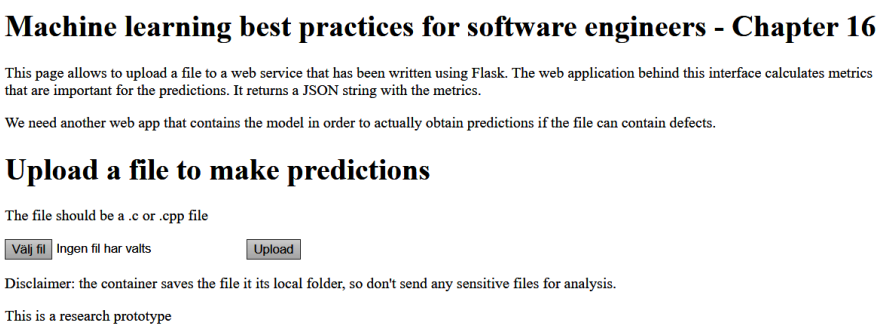

Figure 16.2 – The web page where we can send the file to calculate lines of code. This is only one way of sending this information to the web service

After uploading the file, we get the results:

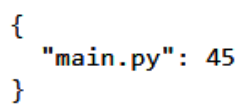

Figure 16.3 – Results of calculating the lines of code

The majority of the transmission is done by the Flask framework, which makes the development a really pleasant experience. However, just counting lines of code and complexity is not a great ML model. Therefore, we need to create another web service with the code of the ML model itself.

Therefore, my next best practice is about dual interfaces.

> **Best practice #76**
> Use both a website and an API for the web services.

Although we can always design web services so that they only accept JSON/REST calls, we should try to provide different interfaces. The web interface (presented previously) allows us to test the web service and even send data to it without the need to write a separate program.

Creating a web service that contains a pre-trained ML model

The code of the ML model web service follows the same template. It uses the Flask framework to provide the REST API of the web service for the software. Here is the code fragment that shows this web service:

```
#
# This is a flask web service to make predictions on the data
# that is sent to it. It is meant to be used with the measurement
instrument
#
from flask import *
from joblib import load
import pandas as pd

app = Flask(__name__)   # create an app instance

# entry point where we send JSON with two parameters:
# LOC and MCC
# and make prediction using make_prediction method
@app.route('/predict/<loc>/<mcc>')
def predict(loc,mcc):
    return {'Defect': make_prediction(loc, mcc)}

@app.route('/')
def hello():
    return 'Welcome to the predictor! You need to send a GET request
with two parameters: LOC (lines of code) and MCC (McCabe complexity))'

# the main method for making the prediction
```

```
# using the model that is stored in the joblib file
def make_prediction(loc, mcc):
    # now read the model from the joblib file
    # and predict the defects for the X_test data

    dt = load('dt.joblib')

    # input data to the model
    input = {'LOC': loc,
             'MCC': mcc}

    # convert input data into dataframe
    X_pred = pd.DataFrame(input, index=[0])

    # make prediction
    y_pred = dt.predict(X_pred)

    # return the prediction
    # as an integer
    return int(y_pred[0])

# run the application
if __name__ == '__main__':
    app.run(debug=True)
```

The main entry point to this web service takes two parameters: `@app.route('/predict/<loc>/<mcc>')`. It uses these two parameters as parameters of the method that instantiates the models and uses it to make a prediction – `make_prediction(loc, mcc)`. The `make_prediction` method reads a model from a `joblib` file and uses it to predict whether the module will contain a defect or not. I use `joblib` for this model as it is based on a NumPy array. However, if a model is based on a Python object (for example, when it is an estimator from a scikit library), then it is better to use pickle instead of `joblib`. It returns the JSON string containing the result. *Figure 16.4* illustrates how we can use a web browser to invoke this web service – not the address bar:

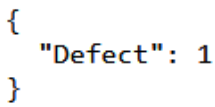

Figure 16.4 – Using the prediction endpoint to get the predicted number of defects in this module

The address bar sends the parameters to the model and the response is a JSON string that says that this module will, most probably, contain a defect. Well, this is not a surprise as we say that the module has 10 lines of code and a complexity of 100 (unrealistic, but possible).

These two web services already give us an example of how powerful the REST API can be. Now, let's learn how to package that with Docker so that we can deploy these web services even more easily.

Deploying ML models using Docker

To create a Docker container with our newly created web service (or two of them), we need to install Docker on our system. Once we've installed Docker, we can use it to compile the container.

The crucial part of packaging the web service into the Docker container is the Dockerfile. It is a recipe for how to assemble the container and how to start it. If you're interested, I've suggested a good book about Docker containers in the *Further reading* section so that you can learn more about how to create more advanced components than the ones in this book.

In our example, we need two containers. The first one will be the container for the measurement instrument. The code for that container is as follows:

```
FROM alpine:latest
RUN apk update
RUN apk add py-pip
RUN apk add --no-cache python3-dev
RUN pip install --upgrade pip
WORKDIR /app
COPY . /app
RUN pip --no-cache-dir install -r requirements.txt
CMD ["python3", "main.py"]
```

This Dockerfile is setting up an Alpine Linux-based environment, installing Python and the necessary development packages, copying your application code into the image, and then running the Python script as the default command when the container starts. It's a common pattern for creating Docker images for Python applications. Let's take a closer look:

- `FROM alpine:latest`: This line specifies the base image for the Docker image. In this case, it uses the Alpine Linux distribution, which is a lightweight and minimalistic distribution that's often used in Docker containers. `latest` refers to the latest version of the Alpine image available on Docker Hub.
- `RUN apk update`: This command updates the package index of the Alpine Linux package manager (`apk`) to ensure it has the latest information about available packages.
- `RUN apk add py-pip`: Here, it installs the `py-pip` package, which is the package manager for Python packages. This step is necessary to be able to install Python packages using `pip`.
- `RUN apk add --no-cache python3-dev`: This installs the `python3-dev` package, which provides development files for Python. These development files are often needed when compiling or building Python packages that have native code extensions.

- `RUN pip install --upgrade pip`: This upgrades the `pip` package manager itself to the latest version.
- `WORKDIR /app`: This sets the working directory for the subsequent commands to `/app`. This directory is where the application code will be copied, and it becomes the default directory for running commands.
- `COPY . /app`: This copies the contents of the current directory (where the Dockerfile is located) into the `/app` directory in the Docker image. This typically includes the application code, including `requirements.txt`.
- `RUN pip --no-cache-dir install -r requirements.txt`: This installs Python dependencies specified in the `requirements.txt` file. The `--no-cache-dir` flag is used to ensure that no cache is used during the installation, which can help reduce the size of the Docker image.
- `CMD ["python3", "main.py"]`: This specifies the default command to run when a container is started from this image. In this case, it runs the `main.py` Python script using `python3`. This is the command that will be executed when we run a container based on this Docker image.

In *Step 8*, we need the `requirements.txt` file. In this case, the file does not have to be too complex – it needs to use the same imports as the web service script:

```
flask
flask-restful
```

Now, we are ready to compile the Docker container. We can do that with the following command from the command line:

```
docker build -t measurementinstrument .
```

Once the compilation process is complete, we can start the container:

```
docker run -t -p 5000:5000 measurementinstrument
```

The preceding command tells the Docker environment that we want to start the container, `measurementinstrument`, and map the port of the web service (`5000`) to the same port in `localmachine`. Now, if we navigate to the address, we can upload the file just like we could when the web service was running without the Docker containers.

Best practice #77
Dockerize your web services for both version control and portability.

Using Docker is one way we can ensure the portability of our web services. Once we package our web service into the container, we can be sure that it will behave the same on every system that is capable of running Docker. This makes our lives much easier even more than using `requirements.txt` files to set up Python environments.

Once we have the container with the measurement instrument, we can package the second web service – with the prediction model – into another web service. The following Dockerfile does this:

```
FROM ubuntu:latest
RUN apt update && apt install python3 python3-pip -y
WORKDIR /app
COPY . /app
RUN pip --no-cache-dir install -q -r requirements.txt
CMD ["python3", "main.py"]
```

This Dockerfile sets up an Ubuntu-based environment, installs Python 3 and `pip`, copies your application code into the image, installs Python dependencies from `requirements.txt`, and then runs the Python script as the default command when the container starts. Please note that we use Ubuntu here and not Alpine Linux. This is no accident. There is no scikit-learn package for Alpine Linux, so we need to use Ubuntu (for which that Python package is available):

- `FROM ubuntu:latest`: This line specifies the base image for the Docker image. In this case, it uses the latest version of the Ubuntu Linux distribution as the base image. `latest` refers to the latest version of the Ubuntu image available on Docker Hub.
- `RUN apt update && apt install python3 python3-pip -y`: This command is used to update the package index of the Ubuntu package manager (`apt`) and then install Python 3 and Python 3 `pip`. The `-y` flag is used to automatically answer "yes" to any prompts during the installation process.
- `WORKDIR /app`: This sets the working directory for the subsequent commands to `/app`. This directory is where the application code will be copied, and it becomes the default directory for running commands.
- `COPY . /app`: This copies the contents of the current directory (where the Dockerfile is located) into the `/app` directory in the Docker image.
- `RUN pip --no-cache-dir install -q -r requirements.txt`: This installs Python dependencies specified in the `requirements.txt` file using pip. The flags that are used here are as follows:
  - `--no-cache-dir`: This ensures that no cache is used during the installation, which can help reduce the size of the Docker image
  - `-q`: This flag runs `pip` in quiet mode, meaning it will produce less output, which can make the Docker build process less verbose

- `CMD ["python3", "main.py"]`: This specifies the default command to run when a container is started from this image. In this case, it runs the `main.py` Python script using `python3`. This is the command that will be executed when we run a container based on this Docker image.

In this case, the requirements code is a bit longer, although not extremely complex:

```
scikit-learn
scipy
flask
flask-restful
joblib
pandas
numpy
```

We compile the Docker container with a similar command:

```
docker build -t predictor .
```

We execute it with a similar command:

```
docker run -t -p 5001:5000 predictor
```

Now, we should be able to use the same browser commands to connect. Please note that we use a different port so that this new web service does not collide with the previous one.

Combining web services into ecosystems

Now, let's develop the software that will connect these two web services. For this, we'll create a new file that will send one file to the first web service, get the data, and then send it to the second web service to make predictions:

```
import requests

# URL of the Flask web service for file upload
upload_url = 'http://localhost:5000/success'  # Replace with the
actual URL

# URL of the Flask web service for predictions
prediction_url = 'http://localhost:5001/predict/'  # Replace with the
actual URL


def upload_file_and_get_metrics(file_path):
    try:
```

```
        # Open and read the file
        with open(file_path, 'rb') as file:
            # Create a dictionary to hold the file data
            files = {'file': (file.name, file)}

            # Send a POST request with the file to the upload URL
            response = requests.post(upload_url, files=files)
            response.raise_for_status()

            # Parse the JSON response
            json_result = response.json()

            # Extract LOC and mccabe_complexity from the JSON result
            loc = json_result.get('lines_of_code')
            mccabe_complexity = json_result.get('mccabe_complexity')
[0][-1]

            if loc is not None and mccabe_complexity is not None:
                print(f'LOC: [3], McCabe Complexity: {mccabe_
complexity}')
                return loc, mccabe_complexity
            else:
                print('LOC or McCabe Complexity not found in JSON
result.')

    except Exception as e:
        print(f'Error: {e}')

def send_metrics_for_prediction(loc, mcc):
    try:
        # Create the URL for making predictions
        predict_url = f'{prediction_url}[3]/[4]'

        # Send a GET request to the prediction web service
        response = requests.get(predict_url)
        response.raise_for_status()

        # Parse the JSON response to get the prediction
        prediction = response.json().get('Defect')

        print(f'Prediction: {prediction}')

    except Exception as e:
        print(f'Error: {e}')
```

```
if __name__ == '__main__':
    # Specify the file path you want to upload
    file_path = './main.py'  # Replace with the actual file path

    # Upload the specified file and get LOC and McCabe Complexity
    loc, mcc = upload_file_and_get_metrics(file_path)

    send_metrics_for_prediction(loc, mcc)
```

This code demonstrates how to upload a file to a Flask web service to obtain metrics and then send those metrics to another Flask web service for making predictions. It uses the `requests` library to handle HTTP requests and JSON responses between the two services:

- `import requests`: This line imports the `requests` library, which is used to send HTTP requests to web services.
- `upload_url`: This variable stores the URL of the Flask web service for file upload.
- `prediction_url`: This variable stores the URL of the Flask web service for predictions.
- `upload_file_and_get_metrics`:
  - This function takes `file_path` as input, which should be the path to the file you want to upload and obtain metrics for.
  - It sends a POST request to `upload_url` to upload the specified file.
  - After uploading the file, it parses the JSON response received from the file upload service.
  - It extracts the `"lines_of_code"` and `"mccabe_complexity"` fields from the JSON response.
  - The extracted metrics are printed, and the function returns them.
- `send_metrics_for_prediction`:
  - This function takes `loc` (lines of code) and `mcc` (McCabe complexity) values as input.
  - It constructs the URL for making predictions by appending the `loc` and `mcc` values to `prediction_url`.
  - It sends a GET request to the prediction service using the constructed URL.
  - After receiving the prediction, it parses the JSON response to obtain the `"Defect"` value.
  - The prediction is printed to the console.

- `if __name__ == '__main__'`: This block specifies the file path (`file_path`) that you want to upload and obtain metrics for. It calls the `upload_file_and_get_metrics` function to upload the file and obtain the metrics (lines of code and McCabe complexity). Then, it calls the `send_metrics_for_prediction` function to send these metrics for prediction and prints the prediction.

This program shows that we can package our model into a web service (with or without a container) and then use it, just like *Figure 16.1* suggested. This way of designing the entire system allows us to make the software more scalable and robust. Depending on the usage scenario, we can adapt the web services and deploy them on several different servers for scalability and load balancing.

Summary

In this chapter, we learned how to deploy ML models using web services and Docker. Although we only deployed two web services, we can see that it can become an ecosystem for ML. By separating predictions and measurements, we can separate the computational-heavy workloads (prediction) and the data collection parts of the pipeline. Since the model can be deployed on any server, we can reuse the servers and therefore reduce the energy consumption of these models.

With that, we have come to was last technical chapter of this book. In the next chapter, we'll take a look at the newest trends in ML and peer into our crystal ball to predict, or at least guess, the future.

References

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- *Masse, M., REST API design rulebook: designing consistent RESTful web service interfaces. 2011: " O'Reilly Media, Inc.".*
- *Raj, P., J.S. Chelladhurai, and V. Singh, Learning Docker. 2015: Packt Publishing Ltd.*
- *Staron, M., et al. Robust Machine Learning in Critical Care—Software Engineering and Medical Perspectives. In 2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN). 2021. IEEE.*
- *McCabe, T.J., A complexity measure. IEEE Transactions on Software Engineering, 1976(4): p. 308-320.*