

Chapter 10: Serving Transformer Models



So far, we've explored many aspects surrounding Transformers, and you've learned how to train and use a Transformer model from scratch. You also learned how to fine-tune them for many tasks. However, we still don't know how to serve these models in production. Like any other real-life and modern solution, **Natural Language Processing (NLP)**-based solutions must be able to be served in a production environment. However, metrics such as response time must be taken into consideration while developing such solutions.

This chapter will explain how to serve a Transformer-based NLP solution in environments where CPU/GPU is available. **TensorFlow Extended (TFX)** for machine learning deployment as a solution will be described here. Also, other solutions for serving Transformers as APIs such as FastAPI will be illustrated. You will also learn about the basics of Docker, as well as how to dockerize your service and make it deployable. Lastly, you will learn how to perform speed and load tests on Transformer-based solutions using Locust.

We will cover the following topics in this chapter:

- fastAPI Transformer model serving

- Dockerizing APIs

- Faster Transformer model serving using TFX

- Load testing using Locust

Technical requirements

We will be using Jupyter Notebook, Python, and Dockerfile to run our coding exercises, which will require Python 3.6.0. The following packages need to be installed:

TensorFlow

PyTorch

Transformer >=4.00

fastAPI

Docker

Locust

Now, let's get started!

All the notebooks for the coding exercises in this chapter will be available at the following GitHub link: <https://github.com/PacktPublishing/Mastering-Transformers/tree/main/CH10>.

Check out the following link to see the Code in Action video: <https://bit.ly/375TOPO>

fastAPI Transformer model serving

There are many web frameworks we can use for serving. Sanic, Flask, and fastAPI are just some examples. However, fastAPI has recently gained so much attention because of its speed and reliability. In this section, we will use fastAPI and learn how to build a service according to its documentation. We will also use `pydantic` to define our data classes. Let's begin!

1. Before we start, we must install `pydantic` and fastAPI:

```
$ pip install pydantic
$ pip install fastapi
```

Copy

Explain

2. The next step is to make the data model for decorating the input of the API using `pydantic`. But before forming the data model, we must know what our model is and identify its input.

We are going to use a **Question Answering (QA)** model for this. As you know from [Chapter 6, Fine-Tuning Language Models for Token Classification](#), the input is in the form of a question and a context.

3. By using the following data model, you can make the QA data model:

```
from pydantic import BaseModel
class QADataModel(BaseModel):
    question: str
    context: str
```

[Copy](#)[Explain](#)

4. We must load the model once and not load it for each request; instead, we will preload it once and reuse it. Because the endpoint function is called each time we send a request to the server, this will result in the model being loaded each time:

```
from transformers import pipeline
model_name = 'distilbert-base-cased-distilled-squad'
model = pipeline(model=model_name, tokenizer=model_name,
                  task='question-answering')
```

[Copy](#)[Explain](#)

5. The next step is to make a fastAPI instance for moderating the application:

```
from fastapi import FastAPI
app = FastAPI()
```

[Copy](#)[Explain](#)

6. Afterward, you must make a fastAPI endpoint using the following code:

```
@app.post("/question_answering")
async def qa(input_data: QADataModel):
    result = model(question = input_data.question, context=input_data.context)
    return {"result": result["answer"]}
```

[Copy](#)[Explain](#)

7. It is important to use **async** for the function to make this function run in asynchronous mode; this will be parallelized for requests. You can also use the **workers** parameter to increase the number of workers for the API, as well as making it answer different and independent API calls at once.

8. Using **uvicorn**, you can run your application and serve it as an API. **Uvicorn** is a lightning-fast server implementation for Python-based APIs that makes them run as fast as possible. Use the following code for this:

```
if __name__ == '__main__':
    uvicorn.run('main:app', workers=1)
```

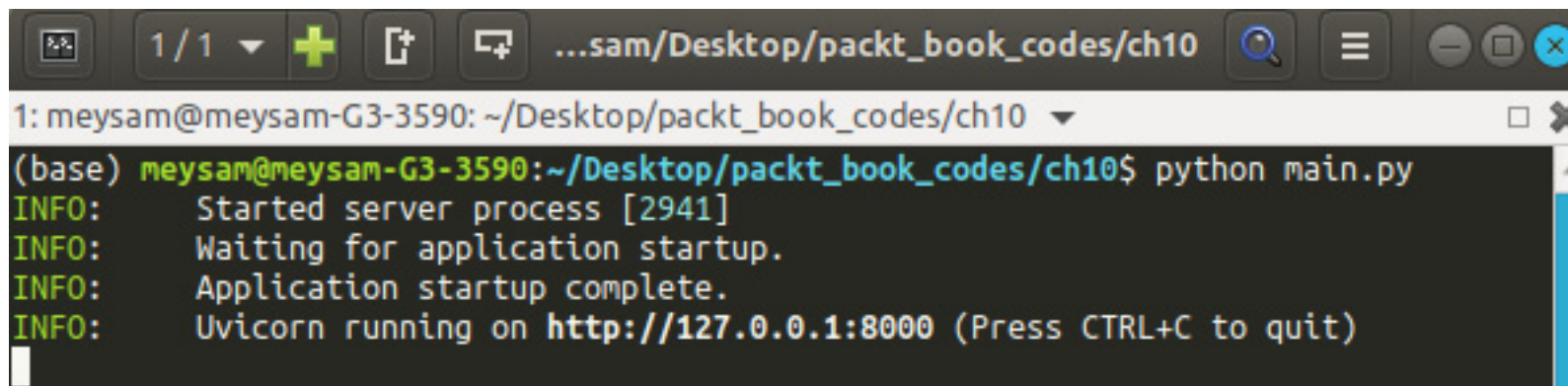
[Copy](#)[Explain](#)

9. It is important to remember that the preceding code must be saved in a `.py` file (`main.py`, for example). You can run it by using the following command:

```
$ python main.py
```

Copy Explain

As a result, you will see the following output in your terminal:



```
1: meysam@meysam-G3-3590: ~/Desktop/packt_book_codes/ch10
(base) meysam@meysam-G3-3590:~/Desktop/packt_book_codes/ch10$ python main.py
INFO: Started server process [2941]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

Figure 10.1 – fastAPI in action

10. The next step is to use and test it. There are many tools we can use for this but Postman is one of the best. Before we learn how to use Postman, use the following code:

```
$ curl --location --request POST 'http://127.0.0.1:8000/question_answering' \
--header 'Content-Type: application/json' \
--data-raw '{
    "question":"What is extractive question answering?",
    "context":"Extractive Question Answering is the task of extracting an answer
from a text given a question. An example of a question answering dataset is the
SQuAD dataset, which is entirely based on that task. If you would like to fine-
tune a model on a SQuAD task, you may leverage the `run_squad.py`.'"
}'
```

Copy Explain

As a result, you will get the following output:

```
{"answer":"the task of extracting an answer from a text given a question"}
```

Copy Explain

Curl is a useful tool but not as handy as Postman. Postman comes with a GUI and is easier to use compared to curl, which is a CLI tool. To use Postman, install it from <https://www.postman.com/downloads/>.

11. After installing Postman, you can easily use it, as shown in the following screenshot:

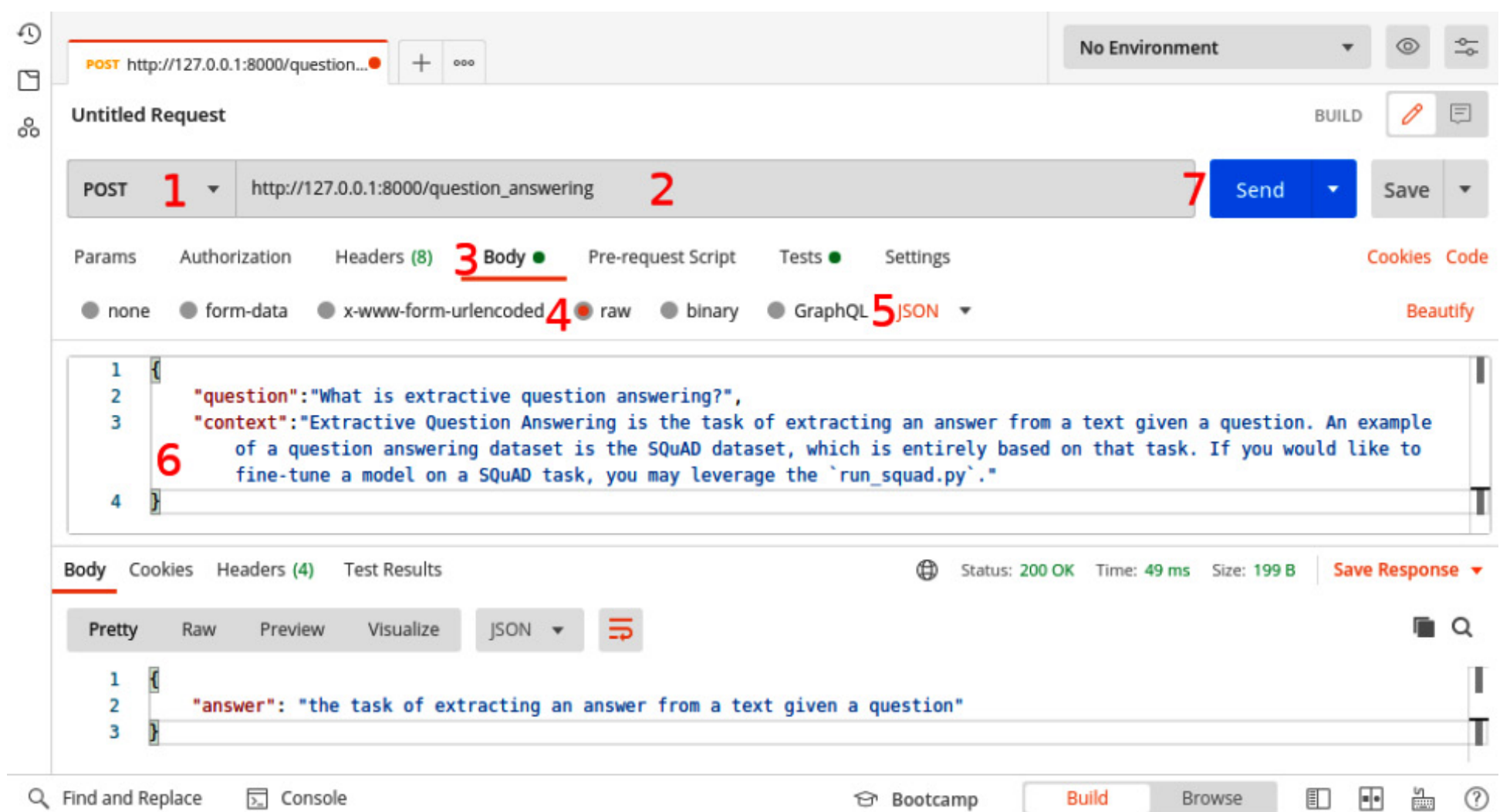


Figure 10.2 – Postman usage

12. Each step for setting up Postman for your service is numbered in the preceding screenshot. Let’s take a look at them:

1. Select **POST** as your method.
2. Enter your full endpoint URL.
3. Select **Body**.
4. Set **Body** to **raw**.
5. Select the **JSON** data type.
6. Enter your input data in JSON format.
7. Click **Send**.

You will see the result in the bottom section of Postman.

In the next section, you will learn how to dockerize your fastAPI-based API. It is essential to learn Docker basics to make your APIs packageable and easier for deployment.

Dockerizing APIs

To save time during production and ease the deployment process, it is essential to use Docker. It is very important to isolate your service and application. Also, note that the same code can be run anywhere, regardless of the underlying OS. To achieve this,

Docker provides great functionality and packaging. Before using it, you must install it using the steps recommended in the Docker documentation (<https://docs.docker.com/get-docker/>):

1. First, put the `main.py` file in the app directory.
2. Next, you must eliminate the last part from your code by specifying the following:

Copy Explain

```
if __name__ == '__main__':  
    uvicorn.run('main:app', workers=1)
```

3. The next step is to make a Dockerfile for your fastAPI; you made this previously. To do so, you must create a Dockerfile that contains the following content:

Copy Explain

```
FROM python:3.7  
RUN pip install torch  
RUN pip install fastapi uvicorn transformers  
EXPOSE 80  
COPY ./app /app  
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

4. Afterward, you can build your Docker container:

Copy Explain

```
$ docker build -t qaapi .  
And easily start it:  
$ docker run -p 8000:8000 qaapi
```

As a result, you can now access your API using port `8000`. However, you can still use Postman, as described in the previous section, *fastAPI Transformer model serving*.

So far, you have learned how to make your own API based on a Transformer model and serve it using fastAPI. You then learned how to dockerize it. It is important to know that there are many options and setups you must learn about regarding Docker; we only covered the basics of Docker here.

In the next section, you will learn how to improve your model serving using TFX.

Faster Transformer model serving using TFX

TFX provides a faster and more efficient way to serve deep learning-based models. But it has some important key points you must understand before you use it. The model must be a saved model type from TensorFlow so that it can be used by TFX Docker or the CLI. Let's take a look:

1. You can perform TFX model serving by using a saved model format from TensorFlow. For more information about TensorFlow saved models, you can read the official documentation at https://www.tensorflow.org/guide/saved_model. To make a saved model from Transformers, you can simply use the following code:

```
from transformers import TFBertForSequenceClassification
model = TFBertForSequenceClassification.from_pretrained("nateraw/bert-base-uncased-imdb", from_pt=True)
model.save_pretrained("tfx_model", saved_model=True)
```

[Copy](#)[Explain](#)

2. Before we understand how to use it to serve Transformers, it is required to pull the Docker image for TFX:

```
$ docker pull tensorflow/serving
```

[Copy](#)[Explain](#)

3. This will pull the Docker container of the TFX being served. The next step is to run the Docker container and copy the saved model into it:

```
$ docker run -d --name serving_base tensorflow/serving
```

[Copy](#)[Explain](#)

4. You can copy the saved file into the Docker container using the following code:

```
$ docker cp tfx_model/saved_model tfx:/models/bert
```

[Copy](#)[Explain](#)

5. This will copy the saved model files into the container. However, you must commit the changes:

Copy Explain

```
$ docker commit --change "ENV MODEL_NAME bert" tfx my_bert_model
```

6. Now that everything is ready, you can kill the Docker container:

Copy Explain

```
$ docker kill tfx
```

This will stop the container from running.

Now that the model is ready and can be served by the TFX Docker, you can simply use it with another service. The reason we need another service to call TFX is that the Transformer-based models have a special input format provided by tokenizers.

7. To do so, you must make a fastAPI service that will model the API that was served by the TensorFlow serving container. Before you code your service, you should start the Docker container by giving it parameters to run the BERT-based model. This will help you fix bugs in case there are any errors:

Copy Explain

```
$ docker run -p 8501:8501 -p 8500:8500 --name bert my_bert_model
```

8. The following code contains the content of the `main.py` file:

[Copy](#)[Explain](#)

```
import uvicorn
from fastapi import FastAPI
from pydantic import BaseModel
from transformers import BertTokenizerFast, BertConfig
import requests
import json
import numpy as np
tokenizer =\
    BertTokenizerFast.from_pretrained("nateraw/bert-base-uncased-imdb")
config = BertConfig.from_pretrained("nateraw/bert-base-uncased-imdb")
class DataModel(BaseModel):
    text: str
app = FastAPI()
@app.post("/sentiment")
async def sentiment_analysis(input_data: DataModel):
    print(input_data.text)
    tokenized_sentence = [dict(tokenizer(input_data.text))]
    data_send = {"instances": tokenized_sentence}
    response = \        requests.post("http://localhost:8501/v1/models/bert:predict",
data=json.dumps(data_send))
    result = np.abs(json.loads(response.text)["predictions"][0])
    return {"sentiment": config.id2label[np.argmax(result)]}
if __name__ == '__main__':
    uvicorn.run('main:app', workers=1)
```

9. We have loaded the **config** file because the labels are stored in it, and we need them to return it in the result. You can simply run this file using **python**:

[Copy](#)[Explain](#)

```
$ python main.py
```

Now, your service is up and ready to use. You can access it using Postman, as shown in the following screenshot:

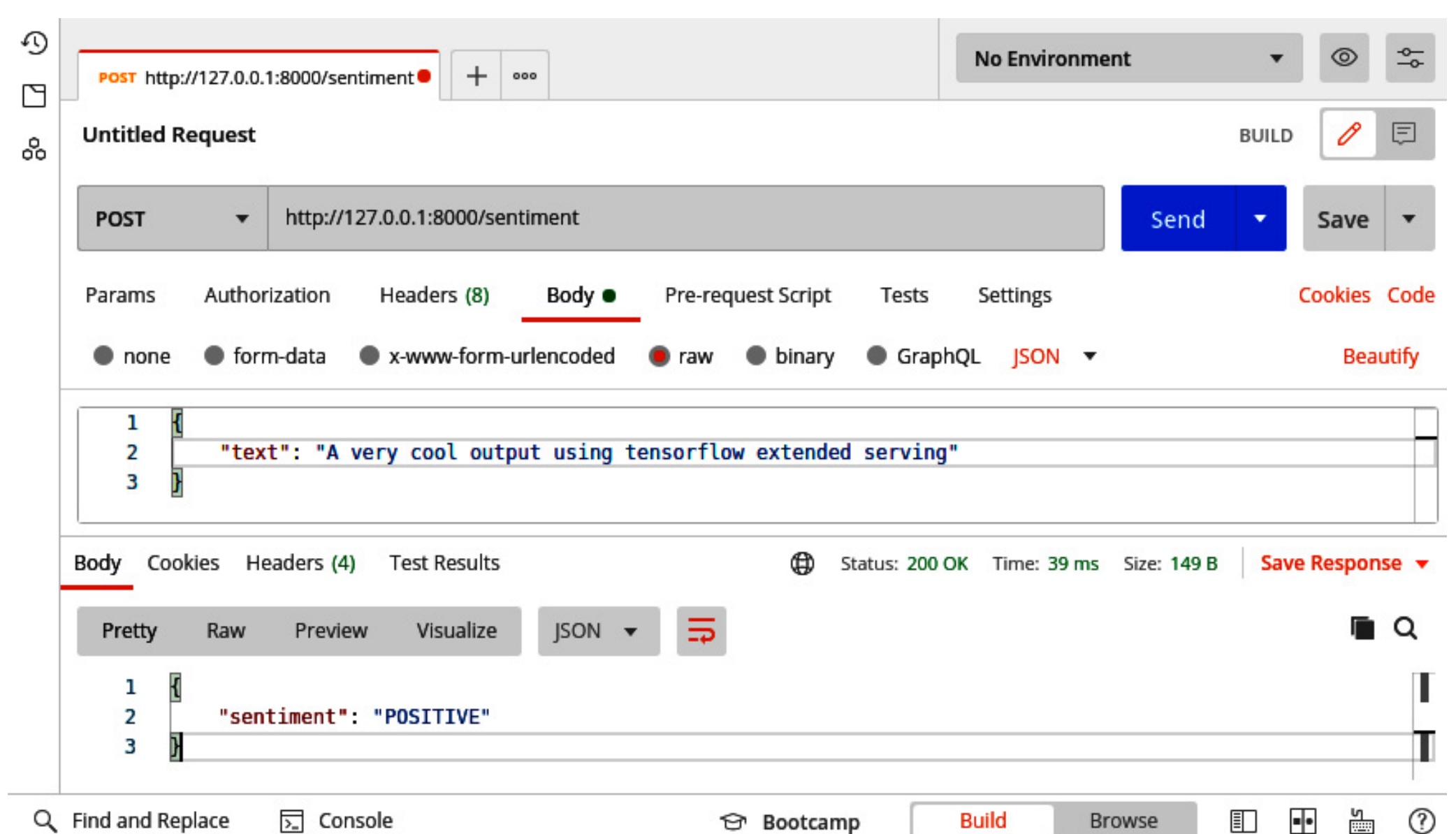


Figure 10.3 – Postman output of a TFX-based service

The overall architecture of the new service within TFX Docker is shown in the following diagram:

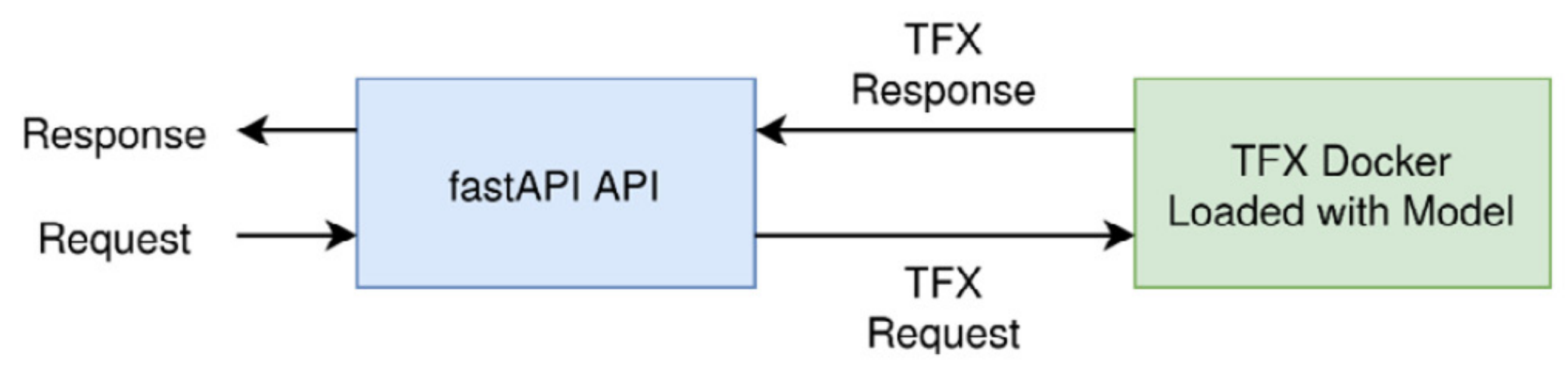


Figure 10.4 – TFX-based service architecture

So far, you have learned how to serve a model using TFX. However, you need to learn how to load test your service using Locust. It is important to know the limits of your service and when to optimize it by using quantization or pruning. In the next section, we will describe how to test model performance under heavy load using Locust.

Load testing using Locust

There are many applications we can use to load test services. Most of these applications and libraries provide useful information about the response time and delay of the service. They also provide information about the failure rate. Locust is one of the best tools for this purpose. We will use it to load test three methods for serving a Transformer-based model: using fastAPI only, using dockerized fastAPI, and TFX-based serving using fastAPI. Let's get started:

1. First, we must install Locust:

```
$ pip install locust
```

[Copy](#)[Explain](#)

This command will install Locust. The next step is to make all the services serving an identical task use the same model. Fixing two of the most important parameters of this test will ensure that all the services have been designed identically to serve a single purpose. Using the same model will help us freeze anything else and focus on the deployment performance of the methods.

2. Once everything is ready, you can start load testing your APIs. You must prepare a **locustfile** to define your user and its behavior. The following code is of a simple **locustfile**:

```
from locust import HttpUser, task
from random import choice
from string import ascii_uppercase
class User(HttpUser):
    @task
    def predict(self):
        payload = {"text": ''.join(choice(ascii_uppercase) for i in range(20))}
        self.client.post("/sentiment", json=payload)
```

[Copy](#)[Explain](#)

By using **HttpUser** and creating the **User** class that's inherited from it, we can define an **HttpUser** class. The **@task** decorator is essential for defining the task that the user must perform after spawning. The **predict** function is the actual task that the user will perform repeatedly after spawning. It will generate a random string that's **20** in length and send it to your API.

3. To start the test, you must start your service. Once you've started your service, run the following code to start the Locust load test:

```
$ locust -f locust_file.py
```

[Copy](#)[Explain](#)

Locust will start with the settings you provided in your `locustfile`. You will see the following in your Terminal:

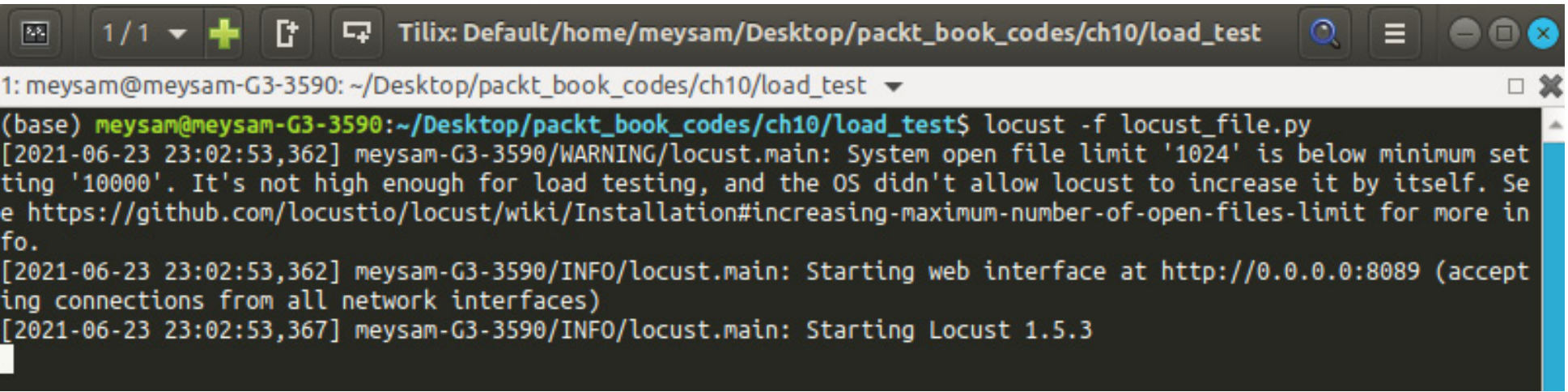


Figure 10.5 – Terminal after starting a Locust load test

As you can see, you can open the URL where the load web interface is located; that is, `http://0.0.0.0:8089`.

- 4. After opening the URL, you will see an interface, as shown in the following screenshot:

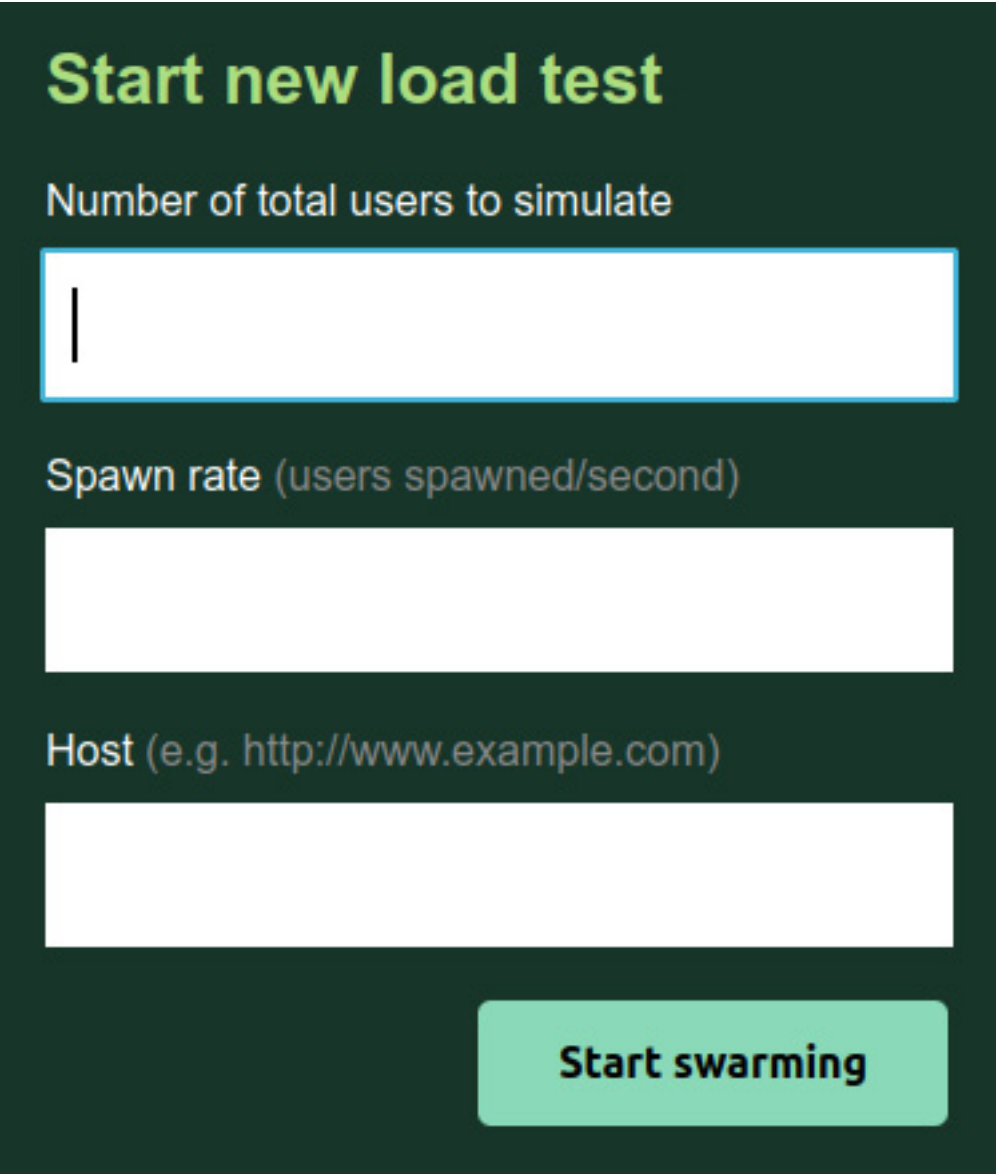


Figure 10.6 – Locust web interface

- 5. We are going to set **Number of total users to simulate** to **10**, **Spawn rate** to **1**, and **Host** to `http://127.0.0.1:8000`, which is where our service is running. After setting these parameters, click **Start swarming**.
- 6. At this point, the UI will change, and the test will begin. To stop the test at any time, click the **Stop** button.
- 7. You can also click the **Charts** tab to see a visualization of the results:

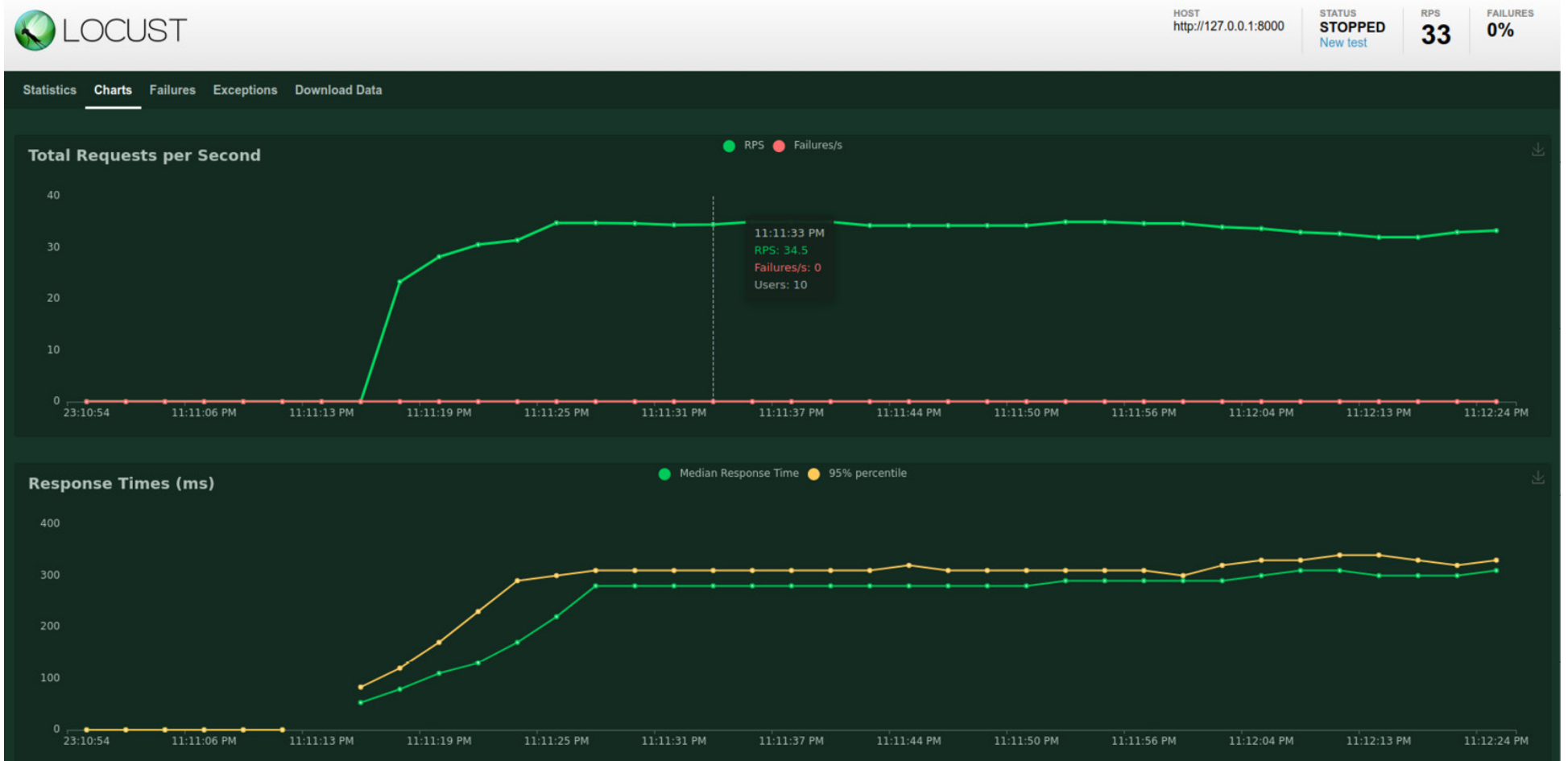


Figure 10.7 – Locust test results from the Charts tab

8. Now that the test is ready for the API, let's test all three versions and compare the results to see which one performs better. Remember that the services must be tested independently on the machine where you want to serve them. In other words, you must run one service at a time and test that, close the service, run the other one and test it, and so on.
- The results are shown in the following table:

	TFX-based FastAPI	FastAPI	Dockerized FastAPI
RPS	38.5	33	34
Average RT (ms)	237	275	270

Table 1 – Comparing the results of different implementations

In the preceding table, **Requests Per Second (RPS)** means the number of requests per second that the API answers, while the **Average Response Time (RT)** means the milliseconds that service takes to respond to a given call. These results shows that the TFX-based fastAPI is the fastest. It has a higher RPS and a lower average RT. All these tests were performed on a machine with an Intel(R) Core(TM) i7-9750H CPU with 32 GB RAM, and GPU disabled.

In this section, you learned how to test your API and measure its performance in terms of important parameters such as RPS and RT. However, there are many other stress tests a real-world API can perform, such as increasing the number of users to make them behave like real users. To perform such tests and report their results in a more realistic way, it is important to read Locust's documentation and learn how to perform more advanced tests.

Summary

In this chapter, you learned the basics of serving Transformer models using fastAPI. You also learned how to serve models in a more advanced and efficient way, such as by using TFX. You then studied the basics of load testing and creating users. Making these users spawn in groups or one by one, and then reporting the results of stress testing, was another major topic of this chapter. After that, you studied the basics of Docker and how to package your application in the form of a Docker container. Finally, you learned how to serve Transformer-based models.

In the next chapter, you will learn about Transformer deconstruction, the model view, and monitoring training using various tools and techniques.

References

Locust documentation: <https://docs.locust.io>

TFX documentation: <https://www.tensorflow.org/tfx/guide>

FastAPI documentation: <https://fastapi.tiangolo.com>

Docker documentation: <https://docs.docker.com>

HuggingFace TFX serving: <https://huggingface.co/blog/tf-serving>

Previous Chapter

Next Chapter