Machine Learning Pipeline End-to-End Solution

ML implementations tend to get complicated quickly. This article will explain how ML system can be split into as few services as possible.

Andrej Baranovskij

Jun 15·6 min read

Author: Andrej Baranovskij

Introduction

After implementing several ML systems and running them in production, I realized there is a significant maintenance overload for monolithic ML apps. ML app code complexity grows exponentially. Data processing and preparation, model training, model serving — these things could look straightforward, but they are not, especially after moving to production.

Data structures are changing, this requires adjusting data processing code. New data types are appearing, this requires maintaining the model up to date and re-train it. These changes could

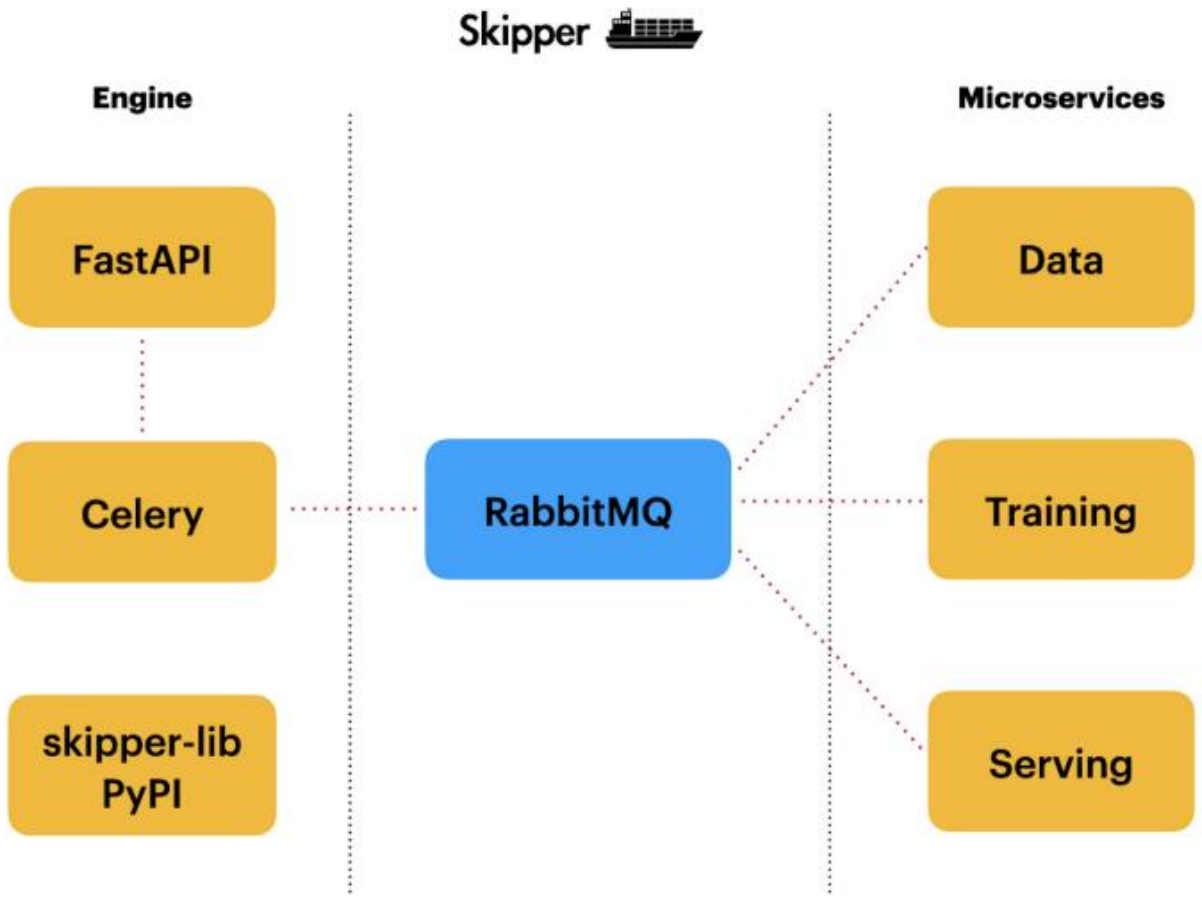
lead to model serving updates. When all this runs as a monolith, it becomes so hard to fix one thing, without breaking something else.

Performance is another important point. When the system is split into different services, it becomes possible to run these services on different hardware. For example, we could run training services on TensorFlow Cloud with GPU, while data processing service could run on local CPU VM.

I did research and checked what options are available to implement ML microservices. There are various solutions, but most of them looked over complicated to me. I decided to implement my own open-source product, which would rely on Python, FastAPI, RabbitMQ, and Celery for communication between services. I called it Skipper, it is on GitHub. The product is under active development, nevertheless, it can be used already.

The core idea of Skipper is to provide a simple and reliable workflow for ML microservices implementation, with Web API interface in the front. In the next phases, services will be wrapped into Docker containers. We will provide support to run services on top of Kubernetes.

Solution Architecture



Skipper architecture, author: Andrej Baranovskij

There are two main blocks — engine and microservices. The engine can be treated as a microservice on its own, but I don't call it a microservice for a reason. The engine part is responsible to provide Web API access, which is called from the outside. It acts as a gateway to a group of microservices. Web API is implemented with FastAPI.

Celery is used to handle long-running tasks submitted through Web API. We start a long-running async task with Celery, the result is retrieved through another endpoint, using task ID.

Common logic is encapsulated into Python library, which is published on PyPI — skipper-lib. The idea of this library is to allow generic event publishing/receiving with RabbitMQ task broker. The same library is used in Web API engine and in microservices.

Microservices block is more like an example. It implements a sample service for data processing, model training, and finally model serving. There is communication through RabbitMQ queue between data processing model training services. The idea is that you could plug your own services into the workflow.

The core element — RabbitMQ broker. Skipper is using RabbitMQ RPC calls to send messages between the services. There is no orchestrator to manage communication. Communication runs based on events, which are sent and received by the services.

Engine and Web API Service

Web API is implemented with FastAPI. There is a *router.py* script, where endpoints are implemented.

Long-running tasks, such as model training are started in async mode, use the Celery distributed task queue. We call *process_worflow* task and get its ID:

```
@router_tasks.post('/execute_async',
                   response_model=WorkflowTask,
                   status_code=202)
def exec_workflow_task_async(workflow_task_data: WorkflowTaskData):
    payload = workflow_task_data.json()

    task_id = process_workflow.delay(payload)

    return {'task_id': str(task_id),
            'task_status': 'Processing'}
```

Task status is checked through another endpoint, where we send the task ID and query Celery API to get the status:

```
@router_tasks.get('/workflow/{task_id}',
                  response_model=WorkflowTaskResult,
                  status_code=202,
```

```
                  responses={202:
                           {'model': WorkflowTask,
                           'description': 'Accepted: Not Ready'}})
async def exec_workflow_task_result(task_id):
    task = AsyncResult(task_id)
    if not task.ready():
        return JSONResponse(status_code=202,
                           content={'task_id': str(task_id),
                                    'task_status': 'Processing'})
    result = task.get()
    return {'task_id': task_id,
            'task_status': 'Success',
            'outcome': str(result)}
```

Tasks that are supposed to complete quickly—for example, predict task—are executed directly, without starting Celery task. The event is sent to RabbitMQ broker and the result is returned in synch mode:

```
@router_tasks.post('/execute_sync',
                   response_model=WorkflowTaskResult,
                   status_code=202,
                   responses={202:
                           {'model': WorkflowTaskCancelled,
                            'description': 'Accepted: Not Ready'}})
def exec_workflow_task_sync(workflow_task_data: WorkflowTaskData):
    payload = workflow_task_data.json()

    queue_name = None
    if workflow_task_data.task_type == 'serving':
        queue_name = 'skipper_serving'

    if queue_name is None:
        return JSONResponse(status_code=202,
                           content={'task_id': '-',
                                  'task_status': 'Wrong task type'})

    event_producer = EventProducer(username='skipper',
                                   password='welcome1',
                                   host='localhost',
                                   port=5672)
    response = json.loads(event_producer.call(queue_name, payload))

    return {'task_id': '-',
            'task_status': 'Success',
            'outcome': str(response)}
```

The Celery task is implemented in *tasks.py* script. Its job is to submit a new event to RabbitMQ and wait for the response.

Skipper Library

The library helps to encapsulate common logic without repeating the same code. I have built and published a library on PyPI with a tool called Poetry.

The library helps to simplify communication with RabbitMQ. It implements the event producer and receiver.

The event producer submits the task to the queue using RPC and waits for the response:

```
def call(self, queue_name, payload):
    self.response = None
    self.corr_id = str(uuid.uuid4())
    self.channel.basic_publish(
        exchange='',
        routing_key=queue_name,
        properties=pika.BasicProperties(
            reply_to=self.callback_queue,
            correlation_id=self.corr_id
        ),
        body=payload
    )
    while self.response is None:
        self.connection.process_data_events()
    return self.response
```

The event receiver listens for the messages from the RabbitMQ queue, calls the service, and returns the response:

```
def on_request(self, ch, method, props, body):
    service_instance = self.service_worker()
    response, task_type = service_instance.call(body)

    ch.basic_publish(exchange='',
                     routing_key=props.reply_to,
                     properties=pika.BasicProperties(
                                 correlation_id=props.correlation_id
                                ),
                     body=response)
    ch.basic_ack(delivery_tag=method.delivery_tag)

    print('Processed request:', task_type)
```

Services

The main goal of the implemented services is to provide an example, how to use the workflow. You can plugin your own services if you are using skipper-lib for event communication.

All services follow the same code structure.

Training Service

This service is responsible to run model training as its name suggests.

Service logic is implemented in *training_service.py* script. There is a method named *call*. This method is automatically executed by the event receiver from *skipper-lib*. In this method you are supposed to read input data, call model training logic and return the result:

```
def call(self, data):
    data_json = json.loads(data)

    self.run_training(data)

    payload = {
        'result': 'TASK_COMPLETED'
    }
    response = json.dumps(payload)

    return response, data_json['task_type']
```

This service sends events to request the data. We are using *skipper-lib* for that too:

```
def prepare_datasets(self, data):
    event_producer = EventProducer(username='skipper',
                                   password='welcome1',
                                   host='localhost',
                                   port=5672)
    response = event_producer.call('skipper_data', data)
```

To start the service, run *main.py* script:

```
from skipper_lib.events.event_receiver import EventReceiver
from app.training_service import TrainingServiceevent_receiver =
EventReceiver(username='skipper',
                               password='welcome1',
                               host='localhost',
                               port=5672,
                               queue_name='skipper_training',
                               service=TrainingService)
```

Training service is configured to listen for *skipper_training* queue.

Data Service

This service receives events to prepare data and returns it to the caller. Multiple datasets are returned at once, for example, training and validation data, target values. Numpy arrays are converted to lists and serialized with *json.dump:*

```
data = [norm_train_x.tolist(),
        norm_test_x.tolist(),
        norm_val_x.tolist(),
        train_y,
        test_y,
        val_y]
```

```
response = json.dumps(data)
```

Training service will deserialize the data with *json.loads* function and then it will create Numpy arrays structure again. This way only a single call is made to the data service and all data structures are transferred in a single call.

To start the service, run *main.py* script:

```
event_receiver = EventReceiver(username='skipper',
                               password='welcome1',
                               host='localhost',
                               port=5672,
                               queue_name='skipper_data',
                               service=DataService)
```

Data service is configured to listen for *skipper_data* queue.

Serving Service

This service loads the model, which was saved by the training service. It loads numbers for data normalization, saved by data service. We need to normalize data, based on the same stats as the ones used for model training.

Service method implementation:

```
def call(self, data):
    data_json = json.loads(data)
    payload = pd.DataFrame(data_json['data'], index=[0, ])
    payload.columns = [x.upper() for x in payload.columns]

    train_stats = pd.read_csv(
                 '../models/train_stats.csv',
                 index_col=0)
    x = np.array(self.norm(payload, train_stats))

    models = self.get_immediate_subdirectories('../models/')
    saved_model = tf.keras.models.load_model(
                     '../models/' + max(models))

    predictions = saved_model.predict(x)

    result = {
        'price': str(predictions[0][0][0]),
        'ptratio': str(predictions[1][0][0])
    }
    response = json.dumps(result)
    return response, data_json['task_type']
```

To start the service, run *main.py* script:

```
event_receiver = EventReceiver(username='skipper',
                               password='welcome1',
                               host='localhost',
                               port=5672,
                               queue_name='skipper_serving',
                               service=ServingService)
```

The serving service is configured to listen for *skipper_serving* queue.

Conclusion

The main idea of this post is to explain how you could split ML implementation into different services. This will help to manage complexity and will improve solution scalability. Hopefully, you will find Skipper useful for your own implementations. Enjoy!

Source code

- GitHub repo. Follow readme for setup instructions

Youtube — watch the demo - https://youtu.be/TVkQCmIGR6w

Reference: https://towardsdatascience.com/ml-pipeline-end-to-end-solution-5889690abbd8