CALIFORNIA STATE UNIVERSITY SAN MARCOS

PROJECT SIGNATURE PAGE

PROJECT SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE

MASTER OF SCIENCE

IN

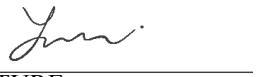
COMPUTER SCIENCE

PROJECT TITLE: A Design Pattern for Deploying Machine Learning Models to Production

AUTHOR: Runyu Xu

DATE OF SUCCESSFUL DEFENSE: 06/19/2020

THE PROJECT HAS BEEN ACCEPTED BY THE PROJECT COMMITTEE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN COMPUTER SCIENCE.

Dr. Yanyan Li		06/19
PROJECT COMMITTEE CHAIR	SIGNATURE	DATE
Dr. Xin Ye		06/19
PROJECT COMMITTEE MEMBER	SIGNATURE	DATE

A Design Pattern for Deploying Machine Learning Models to Production

Runyu Xu

xu026@cougars.csusm.edu

Computer Science and Information Systems
California State University San Marcus

Table of Contents

Abstract

Machine Learning (ML) becomes increasingly popular; industry spends billions of dollars building ML systems. Data scientists have come up with many good algorithms and trained models. However, putting those ML models into production is still in the early stage. The deployment process is distinct from that for traditional software applications; it is not yet well understood among data scientists and IT engineers in their roles and responsibilities, resulting in many anti-pattern practices [21]. The key issues identified by researchers at Google[40] include lack of production-like prototyping stack for data scientists, monolithic programs not fitted for component based ML system orchestration, and lack of best practices in system design. To find solutions, teams need to understand the inherent structure of ML systems and to find ML engineering best practices. This paper presents an abstraction of ML system design process, a design pattern named **Model-Service-Client + Retraining (MSC/R)** consisting of four main components: Model (data and trained model), Service (model serving infrastructure), Client (user interface), and Retraining (model monitoring and retraining). Data scientists and engineers can use this pattern as a discipline in designing and deploying ML pipelines methodically. They can separate concerns, modularize ML systems, and work in parallel. This paper also gives case studies on how to use MSC/R to quickly and reliably deploy two ML models -- *YOLOv3*, an object detection model, and *Stock Prediction* using Long Short-Term Memory (LSTM) algorithm onto AWS and GCP clouds. Two different implementation approaches are used: serving the model as a microservice RESTful API on AWS managed container platform ECS, and on GCP serverless platform Cloud Run. In the end, this paper gives analysis and discussion on how using the MSC/R design pattern helps to meet the objectives of implementing ML production systems and solve the common problems. It also provides insights and recommendations.

List of Abbreviations

AWS: Amazon Web Service

EC2: Elastic Cloud Computing Service

ECS: Elastic Container Service

ECR: Elastic Container Registry

ELB: Elastic Load Balancing

S3: Simple Storage Service

IAM role: Identity and Access Management

GCP: Google Cloud Platform

GCE: Google Computing Engine

GKE: Google Kubernetes Engine

REST: REpresentational State Transfer

API: Application Programming Interface

IoT: Internet of Things

MVC: Model-View-Controller

ML: Machine Learning

AI: Artificial Intelligence

DevOps: Development and Operations

YOLO: You Only Look Once. A new object detection approach

COCO dataset: Common Objects in Context

OpenCV: Open Computer Vision

LSTM: Long Short-Term Memory

RNN: Recurrent Neural Network

PaaS: Platform as a Service

ROI: Return on Investment

1. Introduction

According to the recently updated International Data Corporation (IDC) Worldwide Artificial Intelligence Systems Spending Guide, spending on AI systems will reach $97.9 billion in 2023, more than two and one half times the $37.5 billion that was spent in 2019. The compound annual growth rate (CAGR) for the 2018-2023 forecast period will be 28.4% [1]. However, reports show that a majority (up to 87%) of corporate AI initiatives are struggling to move beyond test stages [2]. Early evidence suggests that the technology can deliver real value to serious adopters. Those organizations that actually put AI and machine learning into production saw a 3-15% profit margin increase [3].

Commercial clouds provide Platform as a Service (PaaS) that offer end-to-end services to develop, design and run business applications in an agile and scalable environment [4]. This ability is good to host ML models to a large number of users and large volumes of data sets. However, to effectively navigate and configure the complex cloud services and find the optimum deployment architecture remains a challenge. No matter how well the model is, data scientists continue to have issues deploying to production often resulting in crippled projects.
One of the major obstacles preventing businesses from gaining returns on their investment (ROI) is that ML infrastructure and operations are different from those designed for traditional software applications; they are much more complex and dynamic. As explained in [5], both traditional and ML applications perform actions in response to inputs. But the way actions are codified differs greatly. Traditional software codifies actions as explicit rules. ML does not codify explicitly. Instead rules are indirectly set by capturing patterns from data. As a result, IT teams are ill prepared to deploy and operationalize the trained models as usable business applications, and data scientists are diverting their talent to sorting out infrastructure and operational issues.

The fact is ML applications introduce a new culture; their deployment and operations require new discipline and processes different from the existing DevOps practices [6]. The large investment with 87% failure to bring ML applications to production shows companies tried to solve the same design problems over and over again at great time and expense.

The failed ROI in the industry opened opportunities for finding best practices in the field. The concept of MLOps, short for Machine Learning and Operations, was introduced [42] in recent years as a new discipline and process of collaboration among data scientists, system engineers and business analysts to work together and build ML business applications effectively. The main success factors are outlined as follows.

1) Reduced time and difficulty to deploy ML models to production.
2) Capability to scale up/down horizontally and automatically.
3) Live model monitoring, tracking and retraining.

To achieve the goals, MLOps need to find solutions to the obstacles standing in the way. After analyzing over 100 cases, Google researchers identified three **common problems** listed below.

- Lack of environment that mirrors production for data scientists. Data scientists use local machine to develop models; the environment is completely different from production resulting the need to re-implement from scratch for production
- Programming style conflict. Data scientists tend to develop models with a monolithic program, not following software engineering best practices.
- System design anti-patterns. Glue code and pipeline jungles, causing integration issues.

The industry is beginning to understand the need for more engineering discipline around ML. As Professor Michael Jordan of UC Berkeley stated in his article *Artificial Intelligence - The revolution Hasn't Happened Yet*, "What we're missing is an engineering discipline with its principles of analysis and design."[45]

Design patterns have been used as a discipline and best practice for software development for many years. As an example, the widely used Model-View-Controller [7] pattern has greatly simplified web application development through separation of concerns, code reuse, and collaboration to quickly and reliably build and deploy web applications. This paper presents a design pattern named **Model-Service-Client + Retraining** or **MSC/R**; it's an abstraction to

guide ML system design and operation. MLOps can use this pattern as a discipline to deploy ML pipelines to production quickly and effectively.

This thesis contains following contributions:

- Section 3. Present the design pattern **MSC/R**. Explain the constructs, and how it can be used as guardrails and discipline by MLOps during the process of deploying ML models in production.
- Section 4. Present case study of using this pattern to quickly and reliably deploy two ML models, *YOLOv3 and Stock Prediction*, as RESTful APIs in public cloud AWS and GCP, using two implementation approaches: serving the models on AWS docker container platform ECS, and on Google Cloud Run serverless platform.
- Section 5. Give analysis on the use of MSC/R pattern to meet MLOps success objectives, discuss how the discipline helps solve those common problems described above. It also provides insight and recommendation for the ML system design process.

2. Technologies and Related Work

2.1 Brief Introduction of Machine Learning Models Used

2.1.1 YOLOv3 Object Detector

You Only Look Once (YOLO) is a state-of-the-art, real-time object detection system [8]. The latest version YOLOv3 published in 2018 is extremely fast and accurate. YOLOv3 uses a single neural network to the full image. The network divides the image into regions and predicts an objectness score for bounding boxes using logistic regression [9]. Based on YOLO team's experiments, YOLOv3 is 1000x faster than R-CNN and 100x faster than Fast R-CNN [8]. The code used in this project is from Zihao Zhang's GitHub repository [10].

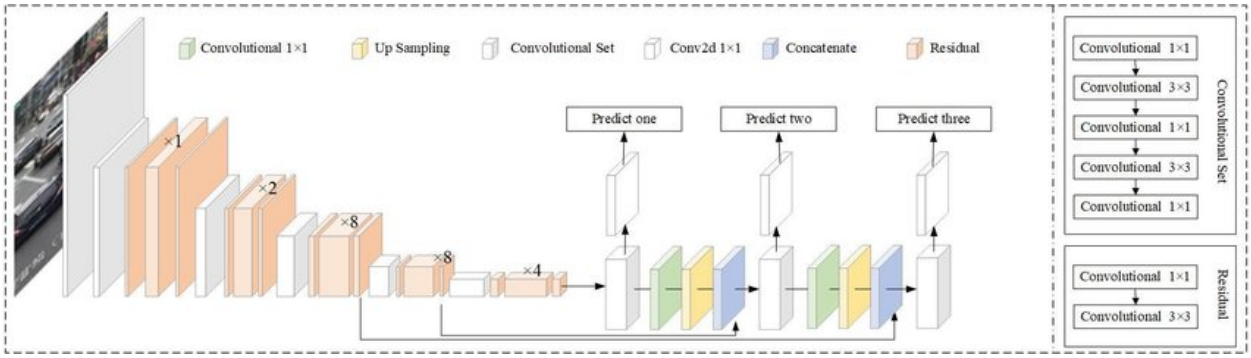


Figure 2.1 Structure detail of YOLOv3.It uses Darknet-53 as the backbone network.[36]

2.1.2 Stock Predictor Using Long Short-Term Memory (LSTM)

Another machine learning model used is a stock prediction model using Long Short-Term Memory (LSTM) designed by Dr. Sung Kim [11]. LSTM, first published in 1997, is a Recurrent Neural Network (RNN) algorithm designed to avoid the long-term dependency problem in standard RNN [12, 13]. A LSTM has the form of a chain of repeating modules of neural networks which is very powerful in sequence prediction problems. The details of LSTM are in **Figure 2.2** and **Figure 2.3**. In the Figures, X_t is the input, h_t is the output, C_t is the cell state, f_t is the forget value.

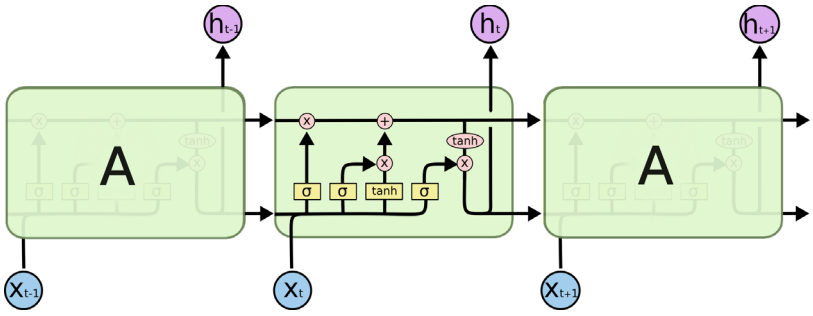
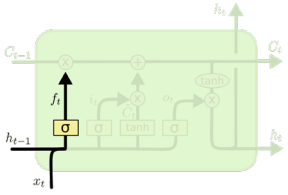
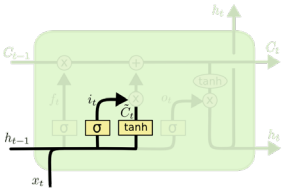


Figure 2.2 The repeating module in an LSTM contains four interacting layers [13]



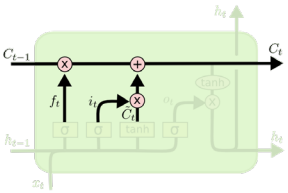
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

(a) removing information



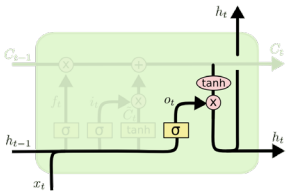
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

(b) create an update to the state



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

(c) update the cell state



$$o_t = \sigma\left(W_o \left[h_{t-1}, x_t\right] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

(d) generate the output

Figure 2.3 Steps of how LSTM works [13]

In the case study, historical stock price of Alphabet Inc. (GOOG) [14] is used to train and generate the stock prediction model. The model can be used to predict the next day's Close Price based on the data from the previous seven days.

2.2 Brief Introduction of Cloud Technologies

2.2.1 Amazon Web Services (AWS)

Amazon Web Service (AWS) is a cloud computing service provided by Amazon. It offers reliable, scalable, and inexpensive cloud computing services [15], such as Elastic Cloud Computing Service (EC2), Simple Storage Service (S3), etc. With its rich ecosystem, AWS allows users to develop, test, deploy, maintain their application without hardware worries.

2.2.2 Google Cloud Platform (GCP)

Google Cloud Platform (GCP) is a cloud computing service provided by Google. It is a suite of cloud computing services that runs on the same infrastructure Google uses internally for its end-user products, such as Google Search, Gmail and YouTube [16]. The services on GCP such as Google Computing Engine (GCE), Google Kubernetes Engine (GKE), etc have less configurations compared to AWS, making it easy for users to develop and deploy their applications.

2.2.3 Other Technologies Used

Docker

Docker is an open platform for developing, shipping, and running applications. It provides the ability to package and run an application in a loosely isolated environment called a container [17]. With container technology, applications can be portable, scalable, and lightweight running on different environments.

Kubernetes (K8s)

Kubernetes is an open-source production-grade container orchestration system for automating deployment, scaling, and management of containerized applications [18]. It is now the most popular container orchestration platform that provides the functionalities to take care of scaling and failover, and make it easy to orchestrate and manage applications.

RESTful API

REST is an acronym for **REpresentational State Transfer.** It is an architecture style for distributed hypermedia systems and was first presented by Roy Fielding in 2000 in his famous dissertation [19]. RESTful APIs are built on this architecture style and work best on the Web. It is stateless, lightweight, simple, and fast. In RESTful APIs, resources such as data and functionality are accessible using URIs. In industry practices, HTTP GET/PUT/POST/DELETE methods are used as the Resource methods in RESTful API.

Serverless Computing

Serverless computing is a cloud computing execution model in which the cloud provider runs the server, and dynamically manages the allocation of machine resources [20]. With serverless computing, users can build and run applications without worrying about commission servers. It simplifies the process of deploying applications into production, improves the resource utilization, and is high availability and scalability. Serverless computing is cost-efficient, only the actual amount of resources used by a service is charged. It is now a hot trend in industry.

2.3 Related Work

Deploying Machine Learning service into production is still new. Industry leaders such as Google, IBM, AWS, Microsoft spend a lot of resources to do research and try to capture the best practices that can be implemented widely. O'Leary and M. Uchida [21] worked with over 100 participants in industry on their ML pipelines and identified three common problems as: 1) The environment for prototyping ML models should be designed to prevent the need to re-implement from scratch for production, 2) ML pipelines should provide a framework of pre-defined canonical unit of operations as components such that ML code can follow ML engineering best practices, as opposed to free-form flexibility, 3) Interfaces between components—both code and data—should be made explicit and simple enough so that implementing such interfaces is easy to use for ML code authors.

H. Washizaki, H. Uchida, F. Khomh and Y. Guéhéneuc [39] has conducted a study on the use of design patterns in ML application development. It concludes that software developers are concerned by the complexity of ML systems and their lack of knowledge of the architecture and design (anti-)patterns that could help them. The authors identified a list of patterns, such as "ML Versioning", "Daisy Architecture" loosely applied in the process of developing ML pipeline, but there remains work to identify design patterns for ML applications. In fact, according to [40], ML systems have a special capacity for incurring technical debt, because they have all of the maintenance problems of traditional code plus an additional set of ML-specific issues; and it is unfortunately common for systems that incorporate machine learning methods to end up with many anti-patterns.

3. A Design Pattern for Deploying ML Models

3.1 Concept of Software Design Pattern

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code or actions. It is a description or template for how to solve a problem that can be used in many different situations [22]. Software architects resort to design patterns to summarize best practices that identify principles, abstractions of reusable/repeatable paradigms, collaboration approach and guidelines for separation of concerns.

Patterns also provide a language and principles for team collaboration. Effective system design requires considering issues that may not become visible until later in the implementation. Therefore, in MLOps context engineers can use design patterns to speed up their design and not repeat the common mistakes.

3.2 What is MLOps

MLOps (a compound of "machine learning" and "operations") [46] is a new engineering practice for collaboration and communication between data scientists and engineers to manage production ML system lifecycle. Similar to DevOps, MLOps has two main responsibilities:

1. **Bringing ML applications to production quickly and reliably, and**
2. **Ensure ML applications operational 24x7 while meeting all the functional and non-functional requirements.**

Examples of specific tasks include:

- build ML development and production infrastructure
- software packaging, orchestration and deployment
- monitoring, diagnostics and mitigation while ML system in production state
- performance solutions: availability, scalability, SLAs (service level agreement)
- build and maintain best practice CI/CD/CT pipeline: continuous integration, continuous deploy and continuous training; and so on.

Current MLOps Challenges

Despite the promise of ML and AI technology, more than 80% data science projects never made it to production. As [47] stated, it's not just the right ML models and services that allow you to do Machine Learning at scale the way you want to; it's being able to place them in the right secure, operationally performant, fully featured, cost-effective system, with the right access controls, that allows you to gain the business results you desire.

The barriers include difficulty to operationalize trained ML models in an enterprise production environment. It's a common issue that data scientists and MLOps engineers have different understanding regarding their roles and responsibilities, as well as the boundaries and interfaces of pipeline components. The lack of discipline and processes on how teams are expected to collaborate resulted in many anti-pattern practices such as "glue code", "pipeline jungles", and "dead experimental codepath" [40].

With MLOps, the goal is to make model deployment easy. ML engineers, not data scientists, can deploy models written in a variety of modern programming languages like Python and R onto modern runtime platforms in the cloud. By adopting best practices and improving collaboration, MLOps engineers can help deliver and maintain successful ML solutions in production environments.

3.3 Model-Service-Client + Retraining (MSC/R) - A Design Pattern for MLOps

Figure 3.1 below is a graphical view of the proposed design pattern **Model-Service-Client + Retraining (MSC/R).** It has 4 layers (Model, Service, Client, Retraining) and 4 connectors (MS, SR, MR and SC). The aim of this pattern is to capture common principles in building production environments of a ML system, captures abstraction of reusable/repeatable processes, and provides guidelines for separation of concerns.

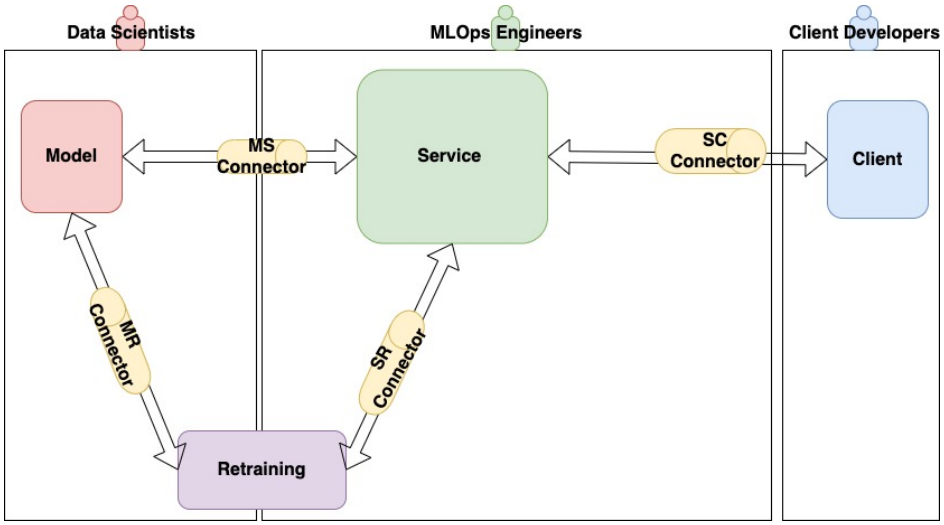


Figure 3.1: the Model-Service-Client + Retraining (MSC/R) design pattern

The labels at the top mark team members' roles, responsibilities and interfaces: data scientists with focus on developing and refining models, MLOps engineers with focus on building and tuning the runtime infrastructure, frontend engineers with focus on Client development. The connectors provide guidelines for collaborations as detailed below under Connectors. With this paradigm, members in ML teams can separate responsibilities and improve efficiency by working in parallel on non-dependent tasks.

Model

The **Model** layer is a Pipeline pattern by itself, used for data preparation and ML model generation. In a ML project, it contains the workflow of data scientists: data collection → data cleaning → feature engineering → model training.

Service

The Service layer is a Composite pattern; its main purpose is serving ML models and meeting all the functional and non-functional system requirements. The subcomponents consist of Front Controller, Model-Serving, and Dynamic Infrastructure Platform [43].

In Service, MLOps concentrate on building and maintaining the service infrastructure such as hosting platforms, scaling mechanism, data collection mechanism, and model-serving functions to provide best performance. Overall, it provides the platforms and tools to meet functional requirements such as handling client requests, producing responses, and coordinating with Model. It also needs to satisfy non-functional requirements such as availability, scalability, performance, governance, etc.

Client

The Client is a Facade pattern, providing user interface to ML service. The implementation may be a web site with user interaction logic, a client to display predictions, API endpoint, mobile front end, IoT edge device. It communicates with the Service's front controller to access ML service.

Retraining

The Retraining layer is a Composite pattern consisting of an Observer and a Trigger. In the Retraining, Data scientists and MLOps engineers work together to determine the performance metrics of ML models, and the threshold used to trigger retraining for the next generation/version of the model. MLOps engineers create the Observer to monitor the performance of the model. Data scientists provide executable retraining code.

Connector

A Connector is an Interface Pattern. It defines data exchange and communication protocol between two entities; it also defines the collaboration methods between the two roles (owners of entities). The contract is negotiated between the roles. There are 4 connectors in MSC/R.

MS Connector: interface between Model and Service. It requires collaboration between data scientists and MLOps to define the type and format of artifacts passed from data scientists to MLOps for deployment. It also defines the deliverables from MLOps to data scientists, for example the development stack, model training pipeline.

MR Connector: interface between Model and Retraining. It requires collaboration between data scientists and MLOps to define the rules of how retrained models are to be tested, versioned and released.

SR Connector: interface between Service and Retraining. It requires collaboration between data scientists and MLOps to define the metrics for ML model performance monitoring, retraining threshold, and retraining data source and code.

SC Connector: interface between Service and Client. It requires collaboration between MLOps and Client developers to define the type, format and protocol of data exchange between Client application and Service entry point.

The Connector pattern is an important construct in the big picture. It is intended to solve the overly separated "research" and "engineering" roles by providing discipline for the two teams to design contracts between them. Well-designed connectors can reduce anti-patterns such as "glue code" and "pipeline jungles" to improve integration and collaboration experience.

3.4 Using MSC/R in ML System Development Process

Similar to software development, ML applications have a development life cycle characterized as following 4 stages [47]: **Research → Development → Deployment → Production**. Details about each stage are given below; for this project the major focus is on Development and Deployment stages.

MLOps play critical roles in the later 3 stages. MLOps can use MSC/R as guardrails and engineering discipline in the development and deployment stages to produce quality work and ease the path of bringing ML models to production.

More about the ML development stages
Research - Data scientists try many approaches and analyses; many options are discarded. Typically, data scientists scratch out some code on their laptops.
Development - Now there are requirements produced as a result of the research stage. Project team is assembled that includes data scientists, MLOps and client developers.

During this stage, teams collaborate to analyze requirements, design system architecture, identify domain tasks and dependencies, and agree on collaboration processes.

Data scientists are typically responsible for the model layer; their work in this stage is illustrated in (**Figure 3.2**):

Figure 3.2 Data Scientists Model Development Phase

Client developers are responsible for developing the front end for users to access the ML model. Common front-end clients include web browser, mobile device, IoT devices, or front end applications such as business dashboards. There can be other requirements such as synchronous or asynchronous response, etc.

MLOps engineers are responsible for building and operationalizing the infrastructure for serving models.

In addition to functional requirements, building the production grade infrastructure involves many non-functional requirements [48] that include availability, scalability, security, governance, automation, etc. Many of the work in these areas can be done in parallel with other teams, i.e. while the ML models and client front ends are being developed.

During this stage, many engineering best practices and methodologies should be adopted to produce quality results. For instance, teams can apply principles of modularization, separation of concerns, and well-defined interfaces between modules. MSC/R serves as a blueprint for MLOps to follow best practices.

Figure 3.3 is an example of high-level architecture after requirement analysis in the development stage.

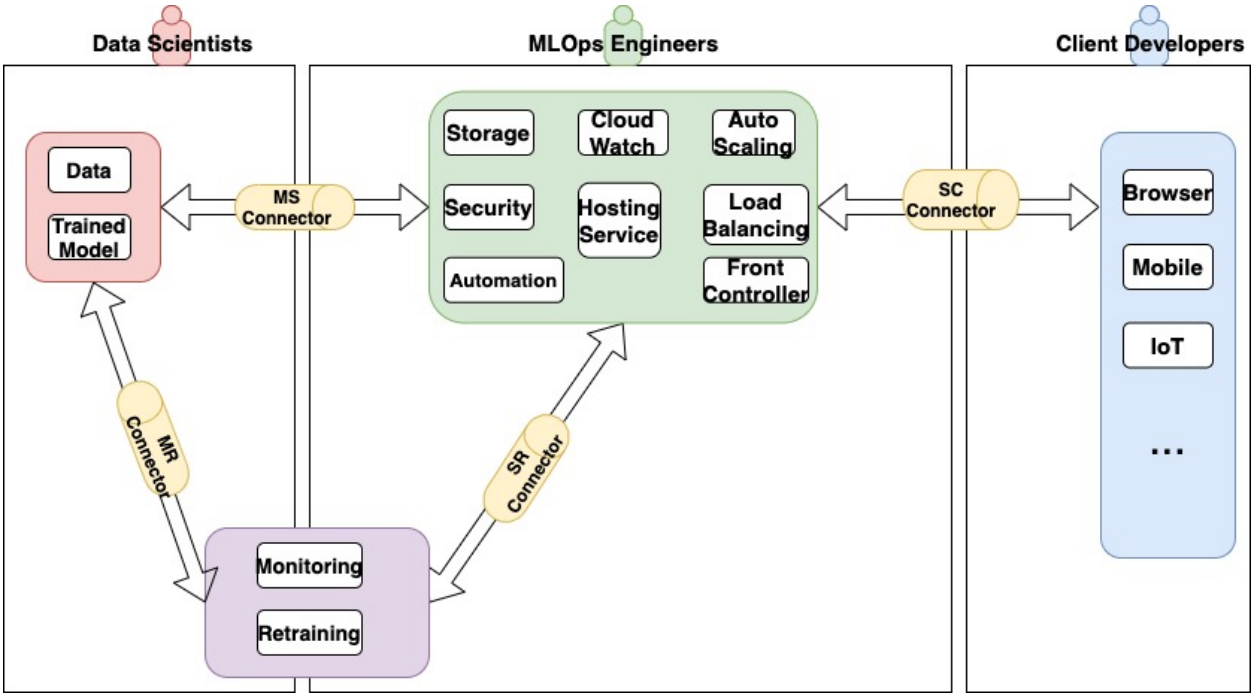


Figure 3.3: MSC/R based ML System Architecture

Deployment - ML model is ready to be deployed and tested.

MLOps is the major player in this stage. Its work includes packaging the code, deploying ML models, configuring and tuning the environment, testing, and working with other teams to re-engineer as needed. Traditionally, this is the point at which many companies struggle [47].

Following MSC/R in the Development stage, the 3 main issues identified by [21] can be greatly reduced or eliminated, thus, to increase success rate in this stage significantly.

Production - While the ML model is in production serving clients, MLOps is responsible for operational tasks such as monitoring, performance tuning, continuous integration and testing etc. This is not the focus of this project.

4. MSC/R Implementation Case Study

The Models

- YOLOv3 for image detection using COCO dataset [8,10]. The COCO dataset contains 330K images with 80 categories of objects.
- Stock Prediction [11] trained with the historical stock price of Alphabet Inc. (GOOG) from Aug 18,2004 to May 22,2020 [14] to predict 20-day stock price. The data contains Open, High, Low, Volume and Close data.

Implementation Environment

The implementations are conducted using the free tier services provided by Amazon cloud AWS and Google cloud GCP.

Availability and Load testing are also conducted to demonstrate the reliability, scalability, and robustness.

Goal

Deploy models to serve as RESTful APIs.

A common best practice of serving ML models is to expose them as RESTful APIs for the benefit of platform independence and service evolution. [49]

Note: this project uses pretrained models, assuming the research and development work by data scientists have been completed beforehand and therefore not included in here. The main focus is on MLOps work during development and deployment stages.

4.1 Case-1: ML Service on Amazon Elastic Container Service (ECS)

4.1.1 Development Process

Following MSC/R as blueprint, I identify three main tracks of work in this stage

Track-1: Design infrastructure for the Service layer based one functional, non-functional requirements

Track-2: Design all the connectors that interface with Service

Track-3: Design retraining pipeline

Note: In real world projects at this stage, data scientists focus on the data and training the models. MLOps focus on building the infrastructure. They collaborate on the connectors part.

Track-1

This track is to design system infrastructure required to host and support ML model deployment. *Most of the work here can be done in parallel with other teams' work.*

The system component diagram is shown as in **Figure 4.1.**

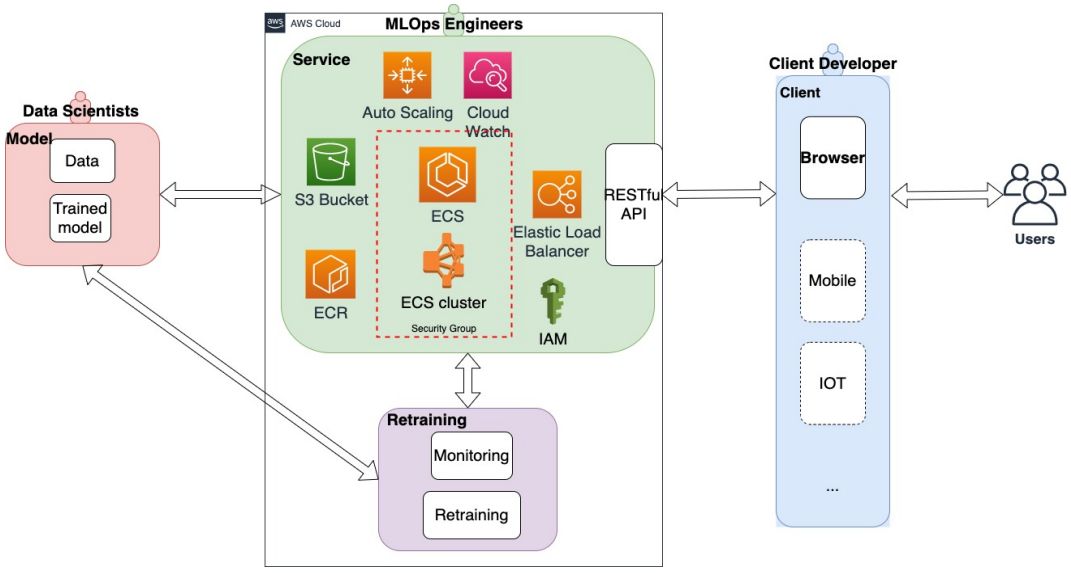


Figure 4.1: Architecture to deploy ML service using AWS ECS cluster

Hosting service -- ECS provides managed container service.

Storage -- Amazon S3 bucket is used to store ML artifacts, Elastic Container Registry (ECR) is used to store ML container images

Security -- IAM is used to assign permissions to access AWS services, Security Group is used to control inbound and outbound traffic

Auto Scaling -- Auto Scaling Group manages the auto scaling

CloudWatch -- AWS CloudWatch is used to monitor the ML infrastructure

Load Balancing -- Elastic Load Balancer is used to balance network traffic and provides URL

Front Controller -- RESTful API

Configuration

- ECS
    - EC2 instances in the cluster are t2.large (2 vCPUs, 8 GiB memory)
    - ECS task definition using existing container image
- Security
    - IAM role with full access to ECS, EC2, ECR, S3
    - security group with inbound rules on port 80 and 5000
- Auto scaling
    - minimum 3 instances, maximum 10 instances
- Elastic Load Balancing
    - Round-robin

Reusability and Templatization

This service infrastructure and configurations can be saved as a template for reuse. In this case study, the same infrastructure and configurations are used for both Stock Prediction and YOLOv3. The template can be easily customized to fit more complex models.

Track-2

This track is to design all the 4 connectors that **Service** interfaces with other layers. This requires collaborating and integrating with other teams. The best practices for components to integrate is through modularization, well defined interfaces, separation of concerns, and design by contracts. [50]

Design MS Connector

The main task is to define the interface with the Model layer. In this case, the standards for the ML model code. Typically, the two parties also need to agree on versioning, code repository, release processes etc. For this project, I will just use GitHub as a code repository, and use the latest version available.

Convert to Microservices

The Stock Prediction code downloaded is an example of monolithic code that packed data access, model black box, and test drivers in one program file. A better approach is to expose the

model as a microservice, a paradigm for code reuse and continuous delivery. Therefore, the program needs to be separated into functions or microservices to take advantage of container technology.

In summary, the connector pattern provides a discipline in the design process: the two teams should either define a protocol that code handed over by data scientists are microservice-enabled, or have an agreement that MLOps are responsible for the conversion.

Figure 4.2 is the original ML code of Stock Prediction. So I converted the program into separate functions or microservices: data input/output - to access S3 bucket (Figure 4.4), model prep - to load model/data (Figure 4.5), and API-enablement - to add API methods (Figure 4.6).

```
'''
This script shows how to predict stock prices using a basic RNN
'''
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

def MinMaxScaler(data):
    ''' Min Max Normalization
    Parameters
    ----------
    data : numpy.ndarray
        input data to be normalized
        shape: [Batch size, dimension]
    Returns
    ----------
    data : numpy.ndarry
        normalized data
        shape: [Batch size, dimension]
    References
    ----------
    .. [1] http://sebastianraschka.com/Articles/2014_about_feature_scaling.html
    '''
    numerator = data - np.min(data, 0)
    denominator = np.max(data, 0) - np.min(data, 0)
    # noise term prevents the zero division
    return numerator / (denominator + 1e-7)

# train Parameters
seq_length = 7
data_dim = 5
output_dim = 1
learning_rate = 0.01
iterations = 500

# Open, High, Low, Volume, Close
xy = np.loadtxt('data-02-stock_daily.csv', delimiter=',')
xy = xy[::-1]  # reverse order (chronically ordered)

# train/test split
train_size = int(len(xy) * 0.7)
train_set = xy[0:train_size]
test_set = xy[train_size - seq_length:]  # Index from [train_size - seq_length] to utilize past sequence
# Scale each
train_set = MinMaxScaler(train_set)
test_set = MinMaxScaler(test_set)

```

```python
# build datasets
def build_dataset(time_series, seq_length):
    dataX = []
    dataY = []
    for i in range(0, len(time_series) - seq_length):
        x = time_series[i:i + seq_length, :]
        y = time_series[i + seq_length, [-1]]  # Next close price
        print(x, "->", y)
        dataX.append(x)
        dataY.append(y)
    return np.array(dataX), np.array(dataY)

trainX, trainY = build_dataset(train_set, seq_length)
testX, testY = build_dataset(test_set, seq_length)

print(trainX.shape)  # (505, 7, 5)
print(trainY.shape)

tf.model = tf.keras.Sequential();
tf.model.add(tf.keras.layers.LSTM(units=1, input_shape=(seq_length, data_dim)))
tf.model.add(tf.keras.layers.Dense(units=output_dim, activation='tanh'))
tf.model.summary()

tf.model.compile(loss='mean_squared_error', optimizer=tf.keras.optimizers.Adam(lr=learning_rate))
tf.model.fit(trainX, trainY, epochs=iterations)

# Test step
test_predict = tf.model.predict(testX)
print("----------------------PRINT TEST PREDCIT---------------------- ")
print(test_predict)
# Plot predictions
plt.plot(testY)
plt.plot(test_predict)
plt.title("Alphabet Inc.")
plt.xlabel("Days")
plt.ylabel("Stock Price")
plt.savefig('static/prediction.png')
plt.show()
```

Figure 4.2 The original code of Stock Prediction [11]

Containerization

Containerization allows IT professionals to deploy software packaged as containers across environments with little or no modification [51]. In short containers offer the benefits of isolation, portability, agility, scalability, and fast deployment. It also raises new challenges; each service runs in its own process and communicates with other processes using protocols such as HTTP or AMQP.

For this case study, I packaged all 3 microservices into one container and deployed to the cloud. This is simple to implement, but it's not a best practice because the base unit of auto scaling is at container level; this means when client calls increase to make prediction, the other two microservices and libraries packaged in the same container are also duplicated unnecessarily. A better approach is to put each microservice in a separate container and use HTTP or AMQP to exchange parameters between the services. This approach will be implemented in future work.

Using microservices adds portability and agility. In this case, the same container template can be used for deploying to different clouds. Also, when a new version of the model is available, it can be easily deployed with minimum impact of the ML service.

Design SC Connector

MLOps works with Client developers to determine the protocol between Front Controller and client endpoint. In this case, the ML model is exposed as a RESTful API, the client needs to invoke the service by sending HTTP requests and get responses in json file format. For testing purposes, I acted as a client developer and wrote a HTML web page where a user can submit a request to the REST API and the page then displays the result on the browser.

In business projects, a client may be a streaming device, MLOps would need to use different protocols such as Real-Time Messaging Protocol (RTMP) in Service-Client connector.

Design MR Connector

This interface between Model and Retraining requires collaboration between data scientists and MLOps to define the rules of how retrained models are to be tested, versioned and released. For this project with limited resources, I chose to use AWS S3 to store a retrained model. The auto deploying script automatically picks up the latest model in the directory.
In business, when a new version is produced and deposited to GitHub repository, it can trigger CI/CD process before the model is deployed to the cloud.

Design SR Connector

This interface between Service and Retraining requires collaboration between data scientists to decide how to initiate retraining, and what parameters to dynamically set at runtime. For example, you can change the training data path and get data from different sources.
For this case study, I used a static data path pointing to an AWS S3 bucket. I used a system scheduler as a trigger.

Track-3

ML model drift can occur, and a model needs to be retrained when data distribution deviates from the original training set, new data is available, or the model performance is degraded.

Model monitoring and continuous retraining are critical parts of a ML system. The model needs to be automatically retrained in production using fresh data. A typical retraining action is triggered by the model performance monitor; AWS CloudWatch and Lambda functions are great tools for this implementation.

In this case study, I use a task scheduler, a linux cron job to launch retraining every Monday to regenerate a new version of the model on the latest data set stored in an AWS S3 bucket. In future work I plan to implement the performance metrics and monitoring scripts and add a data input channel to fully automate and complete the pipeline.

4.1.2 Implementation

Project directory

GitHub artifacts for the Stock Prediction model & YOLOv3 model are in **Figure 4.3**.

```
stock-prediction-api/
├── datasets/
│   ├── train/ # Datasets used for training
│   └── test/  # Datasets used for testing
├── saved_model/ # Directory of trained model
│   └── my_model/ # trained model
│       ├── assets/
│       ├── variables/
│       └── saved_model.pb
├── api.py  # Script to convert the ML service to RESTful API
├── data.py   # Script to download & upload objects that stored on cloud
├── model.py  # Script to train and build ML model
├── retraining.py  # Script to run the retraining
└── wsgi.py  # Script of server interface for Python

yolo-api/
├── checkpoints/ # Directory of all checkpoints of the Model
├── data/ # Directory of data
│   ├── training/ # training data
│   └── test/ # images used for testing
├── tools/ # Directory of tool files
│   ├── export_tfserving.py # Script to export model to tf serving
│   ├── visualize_dataset.py # Script to visualize data records
│   └── voc2012.py # Script to test VOC2012 dataset
```

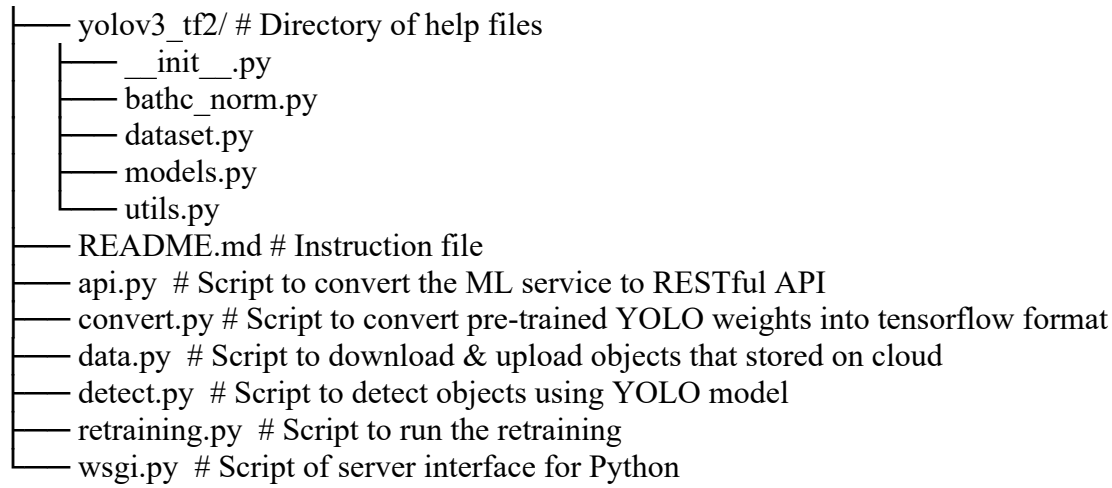


Figure 4.3: The RESTful API project directories
-- Left is Stock prediction, right is YOLO

In the *stock-prediction* directory, the *datasets* folder is used to store training and testing datasets. The trained model is stored in the *saved_model* folder. The *static* folder is to store prediction results. All the web pages are under the *templates* folder. The *data.py* is used to access objects (e.g. data file, model file, etc). The trained model is generated by *model.py* using the training dataset. The *api.py* is to make the ML service as a RESTful API. To serve the ML model, the *wsgi.py* is used. The *retraining.py* is used to retrain current ML model and generate a new version of ML model.

In the *yolo* directory, the *checkpoints* folder stores the pre-trained model. The *data* folder is used to store training and testing datasets. The *static* folder is to store prediction results. All the web pages are under the *templates* folder. The *tools* folder stores some tools files such as *visualize_dataset.py* (visualize the data records). The *yolov3_tf2* hosts the help files such as utils.py. The *convert.py* is used to convert pre-trained YOLOv3 weights into tensorflow format. The *data.py* is used to access objects. The *detect.py* is used to test the YOLO model. The *api.py* is used to make the ML service as a RESTful API. The *wsgi.py* is used to serve the ML model.

Code Containerization

The code artifacts include system runtime and the following customer files: *requirements.txt, data.py, model.py, service.py* and *Dockerfile*

The data input/output code is in *data.py* (**Figure 4.4**), it is used to access S3 buckets.

```
import boto3
## method to get artifacts in a S3 bucket
# bucketName -- the S3 bucket name
# fileName -- the local file name
# s3objectName -- the object name in the S3 bucket
##
def getS3object(bucket_name, fileName, s3objectName):
# create a connection to s3
s3client = boto3.client('s3')
# download file from the S3 bucket
s3client.download_file(bucketName,fileName, s3objectName)

## method to upload an object into a S3 bucket
# bucketName -- the S3 bucket name
# fileName -- the local file name
# s3objectName -- the object name in the S3 bucket
##
def uploadS3object(bucketName, fileName, s3objectName):
s3 = boto3.resource('s3')
s3.Bucket(bucketName).upload_file(fileName,s3objectName, ExtraArgs={'ACL':'public-read'})
```

Figure 4.4: Functions in *data.py*

model.py loads the pre-trained model (**Figure 4.5)**

```
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np


# train Parameters
seq_length = 7
data_dim = 5
output_dim = 1
learning_rate = 0.01
iterations = 500

def MinMaxScaler(data):
    numerator = data - np.min(data, 0)
    denominator = np.max(data, 0) - np.min(data, 0)
    # noise term prevents the zero division
    return numerator / (denominator + 1e-7)

# build datasets
def build_dataset(time_series, seq_length):
    dataX = []
    dataY = []
    for i in range(0, len(time_series) - seq_length):
        x = time_series[i:i + seq_length, :]
        y = time_series[i + seq_length, [-1]]  # Next close price
        print(x, "->", y)
        dataX.append(x)
        dataY.append(y)
    return np.array(dataX), np.array(dataY)

def load_model(fileName):
    new_model = tf.keras.models.load_model(fileName)
    new_model.summary()
    return new_model

def load_data(dataName):
    xy = np.loadtxt(dataName, delimiter=',')
    train_size = int(len(xy) * 0.0)
    test_set = xy[train_size - seq_length:]  # Index from [train_size - seq_length] to utilize past sequence
    test_set = MinMaxScaler(test_set)
    testX, testY = build_dataset(test_set, seq_length)

    return testX,testY

def test(model,testX,testY,outputName):
    test_predict = model.predict(testX)
```

```
print("--------------------PRINT TEST PREDCIT---------------------- ")
#print(test_predict)
# Plot predictions
plt.plot(testY, label = 'Actual price')
plt.plot(test_predict, label = 'Predicted price')
plt.title("Alphabet Inc.")
plt.xlabel("Week")
plt.ylabel("Stock Price ($)")
plt.legend()
if not os.path.isdir("static"):
    os.mkdir("static")

plt.savefig(outputName)
plt.show()
```

Figure 4.5: load_model() and load_data() functions in *model.py*

In the *api.py*, the Flask-RESTful library is used to create RESTful API methods (Figure 4.6).

```
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np

import boto3
import flask
from flask_restful import reqparse, Api, Resource
import io
import os
from data import getObject, uploadObject
from model import MinMaxScaler, build_dataset,load_model,load_data,test

app = flask.Flask(__name__)
api = Api(app)

# argument parsing
parser = reqparse.RequestParser()
parser.add_argument('query', type=str)

class PredictStockPrice(Resource):
    def get(self):
        # use parser and find the user's query
        args = parser.parse_args()
        user_query = args['query']
        file_name = user_query
        model_path = 'saved_model/my_model'
        bucket = "elasticbeanstalk-us-west-1-019024743397"
        getObject(bucket, file_name, file_name)
        model = load_model(model_path)
```

```
        dataX,dataY = load_data(file_name)
        image_name = 'static/test.png'
        object_name = 'prediction.png'
        test(model,dataX,dataY,image_name)
        uploadObject(bucket, image_name, object_name)
        url = 'http://' + bucket + '.s3.amazonaws.com/' + object_name
        # format a json file as output
        output = {'Prediction chart name': url}

        return output

# add API endpoint
api.add_resource(PredictStockPrice,'/')

if __name__ == '__main__':
    app.run(host='0.0.0.0', port= 5000)
```

YOLO – api.py

```
import time
from absl import flags, logging
from absl.flags import FLAGS
import cv2
import flask
import numpy as np
import tensorflow as tf
import boto3
from flask_restful import reqparse, Api, Resource
from yolov3_tf2.models import (
    YoloV3, YoloV3Tiny
)
from yolov3_tf2.dataset import transform_images, load_tfrecord_dataset
from yolov3_tf2.utils import draw_outputs
from data.py import getObject,uploadObject


flags.DEFINE_string('classes', './data/coco.names', 'path to classes file')
flags.DEFINE_string('weights', './checkpoints/yolov3.tf',
                    'path to weights file')
flags.DEFINE_boolean('tiny', False, 'yolov3 or yolov3-tiny')
flags.DEFINE_integer('size', 416, 'resize images to')
flags.DEFINE_string('image', './data/girl.png', 'path to input image')
flags.DEFINE_string('tfrecord', None, 'tfrecord instead of image')
flags.DEFINE_string('output', './static/output.jpg', 'path to output image')
flags.DEFINE_integer('num_classes', 80, 'number of classes in the model')
```

```
def detect(fileName):
    physical_devices = tf.config.experimental.list_physical_devices('GPU')
    if len(physical_devices) > 0:
        tf.config.experimental.set_memory_growth(physical_devices[0], True)

    #if FLAGS.tiny:
    #    yolo = YoloV3Tiny(classes=FLAGS.num_classes)
    #else:
    yolo = YoloV3(classes=80)

    yolo.load_weights('./checkpoints/yolov3.tf').expect_partial()
    logging.info('weights loaded')

    class_names = [c.strip() for c in open('./data/coco.names').readlines()]
    logging.info('classes loaded')

    #if FLAGS.tfrecord:
    #    dataset = load_tfrecord_dataset(
    #        FLAGS.tfrecord, FLAGS.classes, FLAGS.size)
    #    dataset = dataset.shuffle(512)
    #    img_raw, _label = next(iter(dataset.take(1)))
    #else:
    img_raw = tf.image.decode_image(
        open(fileName, 'rb').read(), channels=3)

    img = tf.expand_dims(img_raw, 0)
    img = transform_images(img, 416)

    t1 = time.time()
    boxes, scores, classes, nums = yolo(img)
    t2 = time.time()
    logging.info('time: {}'.format(t2 - t1))

    logging.info('detections:')
    for i in range(nums[0]):
        logging.info('\t{}, {}, {}'.format(class_names[int(classes[0][i])],
                                           np.array(scores[0][i]),
                                           np.array(boxes[0][i])))

    img = cv2.cvtColor(img_raw.numpy(), cv2.COLOR_RGB2BGR)
    img = draw_outputs(img, (boxes, scores, classes, nums), class_names)
    cv2.imwrite('./static/output.jpg', img)
    logging.info('output saved to: {}'.format('./static/output.jpg'))


app = flask.Flask(__name__)
api = Api(app)
```

```
# argument parsing
parser = reqparse.RequestParser()
parser.add_argument('query', type=str)

class YOLODetection(Resource):
    def post(self):
        args = parser.parse_args()
        user_query = args['query']
        file_name = user_query
        bucket = "elasticbeanstalk-us-west-1-019024743397"
        object_name = file_name
        getObject(bucket, file_name, object_name)
        detect(file_name)
        image_name = './static/output.jpg'
        upload_name = 'yolo/detection.png'
        url = uploadObject(bucket,image_name,upload_name)
        output = {'Detection result': url}
        return output


api.add_resource(YOLODetection,'/')

if __name__ == '__main__':
    app.run(host='0.0.0.0', port= 5000)
```

Figure 4.6: RESTful API code in *api.py*

After the ML models are web enabled, MLOps package and dockerize them using the *reqduirements.txt* in **Figure 4.7** and *Dockerfile* in **Figure 4.8.**

Stock prediction – requirements.txt

```
## requirements.txt for stock prediction

tensorflow==2.1.0rc1
numpy==1.17.1
matplotlib==3.1.1
Keras==2.3.1
flask==1.1.1
gunicorn==20.0.4
boto3==1.12.40
```

YOLO – requirements.txt

```
## requirements file for YOLO

tensorflow==2.1.0rc1
numpy==1.17.1
flask==1.1.1
gunicorn==20.0.4
boto3==1.12.40
opencv-python
lxml
tqdm
absl-py
```

Figure 4.7: The *requirements.txt* that list all the required libraries.

--left is stock prediction, right is YOLOv3

Stock prediction – Dockerfile

```
## Dockerfile for stock prediction

# Import a base image
FROM ubuntu:latest
MAINTAINER Jay

# Copy all necessary files
COPY requirements.txt requirements.txt
COPY saved_model saved_model
COPY templates templates
COPY data.py data.py
COPY service.py service.py
COPY wsgi.py wsgi.py

# Install dependencies
RUN apt-get update \
  && apt-get install -y python3-pip python3-dev \
  && cd /usr/local/bin \
  && ln -s /usr/bin/python3 python \
  && pip3 install -r requirements.txt

# Start the service
CMD ["gunicorn","service:app","--bind","0.0.0.0:6001"]
```

YOLO – Dockerfile

```
## Dockerfile for YOLO

# Import a base image
```

```
FROM ubuntu:latest

ENV DEBIAN_FRONTEND=noninteractive

# Copy all necessary files
COPY requirements.txt requirements.txt
COPY checkpoints checkpoints
COPY yolov3_tf2 yolov3_tf2
COPY templates templates
COPY data.py data.py
COPY detect.py detect.py
COPY service.py service.py
COPY wsgi.py wsgi.py

# Install dependencies
RUN apt-get update \
    && apt-get install -y python3-pip python3-dev \
    && apt-get install -y libglib2.0-dev libsm6 libxext6 libxrender-dev \
    && cd /usr/local/bin \
    && ln -s /usr/bin/python3 python \
    && pip3 install -r requirements.txt

# Start the service
CMD ["gunicorn", "service:q:app", "--bind", "0.0.0.0:5000"]
```

Figure 4.8: The *Dockerfile* to containerize ML services

-- Above is Stock Prediction

Deployment Steps

To build the infrastructure to host the ML model, follow steps on ECS:

- Push the container image to Amazon ECR
- Create an IAM role with full access to ECS, EC2, ECR, S3
- Create a security group with inbound rules on port 80 and 5000
- Create an ECS cluster, the EC2 instances in the cluster are t2.large (2 vCPUs, 8 GiB memory)
- Configure auto scaling group: minimum 3 instances, desired 3 instances, and maximum 10 instances.
- Create an ECS task definition using the container image
- Test this ECS task definition on single host (EC2 instance) in the cluster

- Create an Elastic Load Balancer (ELB) listening on port 80
- Create an ECS service using the above task definition
- Wait for the ECS cluster to be launched

Test

A client HTML page is used for testing as given in **Figure 4.9, Figure 4.10** for Stock Prediction and YOLOv3 respectively. Following are the URLs for each application.

- Stock prediction service: ***http://jay-stock-214955919.us-west-1.elb.amazonaws.com***
- YOLO service: ***http://jay-yolo-820405127.us-west-1.elb.amazonaws.com***

Testing Stock Prediction

The test dataset *GOOG.csv*, a historical stock price dataset of Alphabet Inc. is used to test the Stock prediction service. The test dataset contains 27 days prices, only the last 20 days prices are tested. When the Stock Prediction service receives the test data, the trained model uses the first 7 days prices to predict the price of day 8. Then the price from day 2 to day 8 is used to predict the price of day 9. Following this step, the final result is an array of 20 day predicted prices. Next, a plot image is generated using the predicted prices array and the actual prices that is the result in **Figure 4.9**.

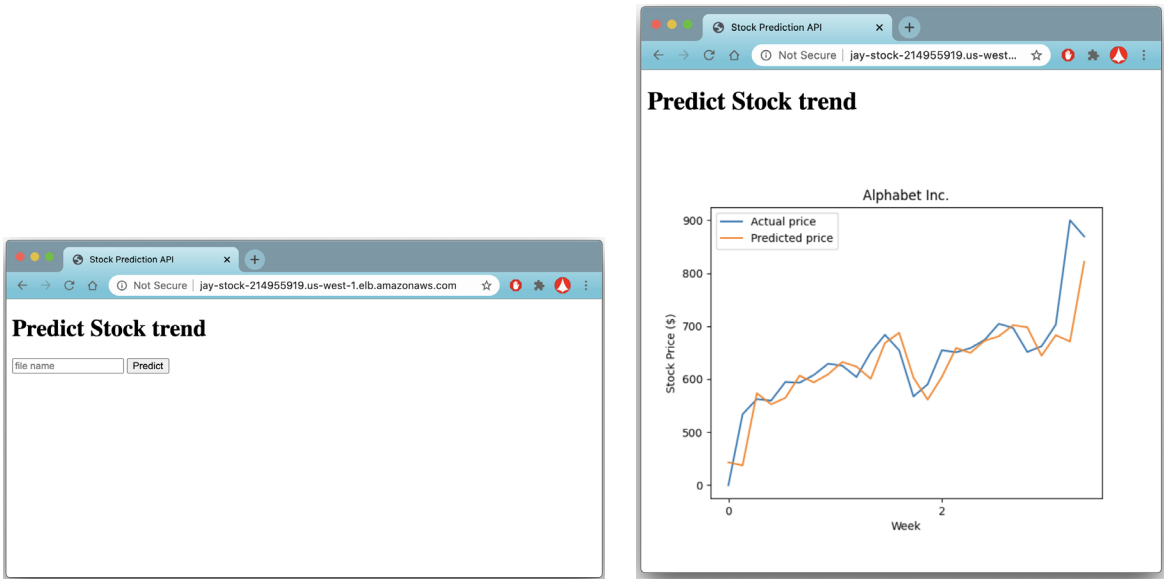


Figure 4.9: Access and get prediction from the Stock prediction service on ECS

Testing YOLOv3

Using the image *street.jpg* to test the YOLO service. When the test image *street.jpg* is passed to the service, the YOLO model is run to detect objects in the image. Result is in **Figure 4.10.** It is observed that not all the objects such as the streetlamp on the left can be detected. The reason is that the COCO dataset only has 80 classes of objects.

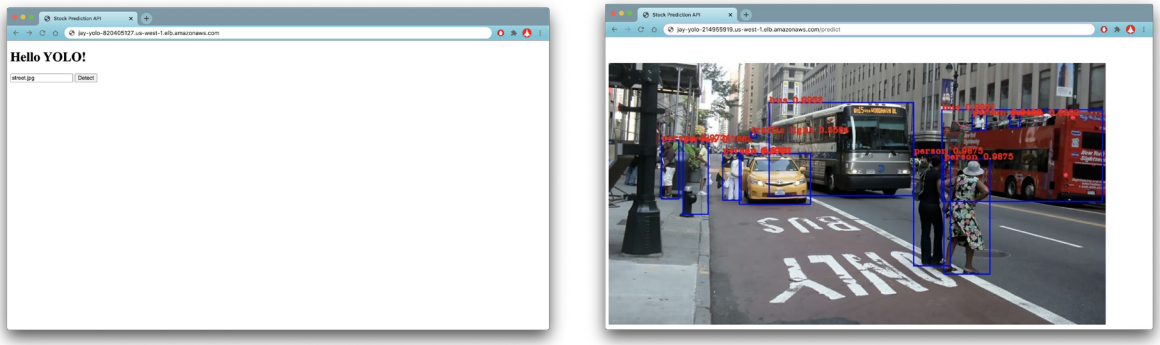


Figure 4.10: Access and get result from the YOLO service on ECS

Retraining

The trigger for retraining the Stock Prediction model is a timer implemented by a Linux cron job (**Figure 4.11**), a task scheduler that can automatically launch the *retraining.py* in the *stock-prediction* directory.

```
# cron job for retraining

0 12 * * 1 /usr/bin/python3 /Users/jayxu/Downloads/project/Stock/retraining.py >> ~/cron.log 2>&1
```

Figure 4.11 cron job to trigger retraining at 12:00 every Monday

In business projects, this component is built by MLOps and data scientists. MLOps engineers implement the connector between Service and Retraining.

4.2 Case 2: ML Service on Google Cloud Run - a Serverless Platform

4.2.1 Serverless Platform Overview

AWS Lambda is a serverless service, but it has limitations on the size of deployment package (250 MB, unzipped) and local disk size (512MB) [31], both Stock Prediction and YOLOv3 models are not suited for AWS Lambda function because of the limitations.

Google Cloud Run is a serverless platform that enables users to run stateless containers invocable via HTTP requests. It's a fully managed, pay-only-for-what-you-use platform that automatically scales containers [27]. Unlike Cloud Functions, an event-driven serverless platform with max deployment size 100 MB (compressed) / 500 MB (uncompressed)[28, 37], Cloud Run is used to build serverless containers with no direct limit for container image size [38].

4.2.2 Development Process

Following MSC/R as blueprint, the three main tracks of work are the same as the one in 4.1.1

Track-1

This track is to design system infrastructure required to host and support ML model deployment. *Most of the work here can be done in parallel with other teams' work.*

The system component diagram is shown as in Figure 4.11.

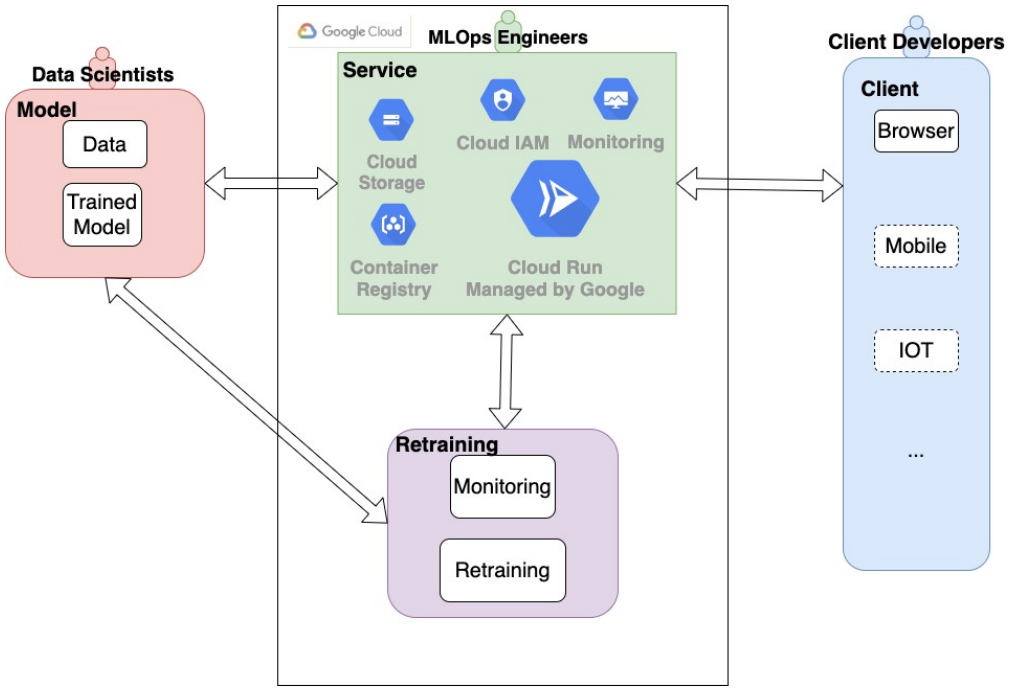


Figure 4.11: Architecture to deploy ML service on Google Cloud Run

Hosting service -- Google Cloud Run a serverless platform is used

Storage -- Cloud Storage is used to store ML artifacts, Cloud Container Registry is used to store ML container images

Security -- Cloud IAM is used to set permissions to access Google Cloud services

Auto Scaling -- Google managed auto scaling service

CloudWatch -- Google managed Cloud Monitoring is used to monitor the ML infrastructure

Load Balancing -- Google managed Load Balancer

Configuration

- Cloud Run:
    - Resource allocated: 2 vCPUs, 2 GiB memory
- Security
    - IAM service account with full access to all GCP services
- Auto scaling
    - size: maximum 1000 instances
- Elastic Load Balancing
    - Round-robin

Track-2

The design of all the 4 connectors are the same as the described in Section 4.1.1 Track-2.

Track-3

The retraining design is similar with the one in Section 4.1.1 Track-3. Instead of using AWS CloudWatch and Lambda functions, GCP Cloud Monitoring and Cloud functions are used.

4.2.3 Implementation

Project directory

In this case, GitHub artifacts for Stock Prediction & YOLOv3 are the same as given in Section 4.1.2.

Code Containerization

The data access function in *data.py* needs to be switched to GCP version (**Figure 4.12**).

```
from google.cloud import storage

def getCloudobject(certentialName,bucketName,objectName,fileName):
  client = storage.Client.from_service_account_json(certentialName)
  bucket = client.get_bucket(bucketName)
  blob = bucket.get_blob(objectName)
  blob.download_to_filename(fileName)
  print("download successful!")

def uploadObject(certentialName,bucketName,objectName,fileName):
   client = storage.Client.from_service_account_json(certentialName)
   bucket = client.get_bucket(bucketName)
   blob = bucket.blob(objectName)
   blob.upload_from_filename(filename=fileName)
   url = "http://storage.cloud.google.com/" + bucketName + objectName
   return url
```

Figure 4.12: GCP version's *data.py*

The *model.py* and *api.py* are the same as given in Section 4.1.2.

The same container template in Section 4.1.2 can be used to generate the container image for Cloud Run.

Deployment Steps

- Create the ML service in Cloud Run, in this implementation, specify the capacity of CPU allocated to 2, Memory allocated to 2 GiB.
- Other infrastructures such as Load Balancing, HTTP endpoints, auto scaling are full managed by Google
- Wait for Google Cloud Run launches the service

Test

The accessible URL of the ML service can be gotten from Cloud Run → Service details:

Stock Prediction: https://stock-predict-jay-rakei3ugwq-uc.a.run.app

YOLO: https://yolo-detection-jay-rakei3ugwq-ue.a.run.app

The test details for both Stock Prediction and YOLO are similar to the description in Section 4.1.2. The result of Stock Prediction is in **Figure 4.13** and the Result of YOLO is in **Figure 4.14**

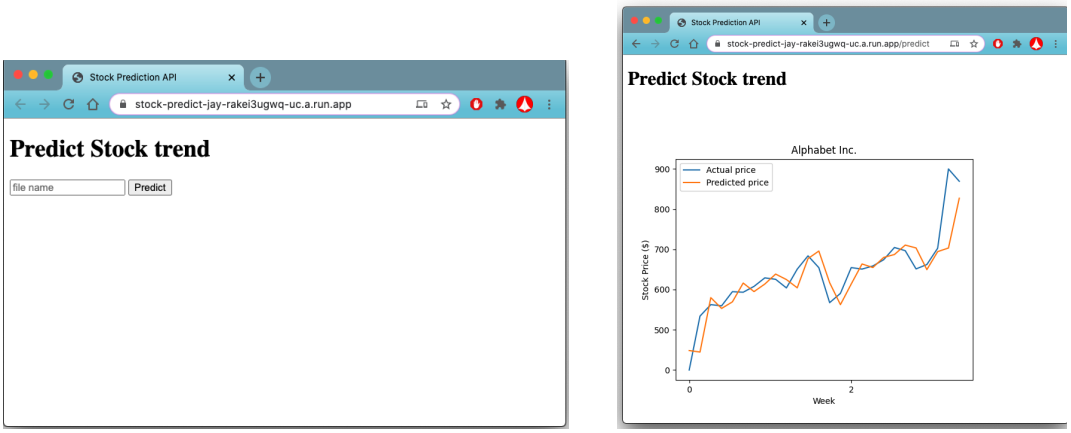


Figure 4.13 Access the stock prediction on Cloud Run via URL

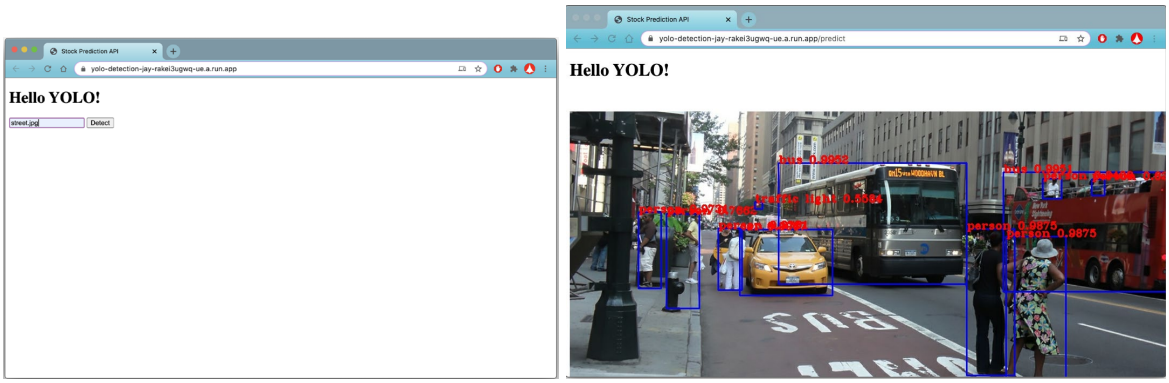


Figure 4.14: Access YOLO service on Cloud Run via URL

Retraining

To retrain the ML model on GCP, the same methods are used as in Section 4.1.

4.3 Performance Testing

This section tests the performance for 3 different deployment approaches. The metrics for testing is availability, scalability, and failover.

The test evidence below is from testing on the Stock Prediction model only. The test results on YOLOv3 models are consistent but not listed here.

Availability

A Linux test script (**Figure 4.15**) is scheduled to call Stock Prediction service every hour for 7 days. The hosting machine logs each access event to system file ****_log.txt*. It shows Stock Prediction service is 100% available for all 24*7 times.

```
# cron job for availability

0 * 1-7 6 * curl http://jay-stock-214955919.us-west-1.elb.amazonaws.com >>
/Users/jayxu/ecs_log.txt
0 * 1-7 6 * curl http://34.66.139.13 >> /Users/jayxu/gke_log.txt
0 * 1-7 6 * curl http://JayStockPrediction-env-1.eba-5u3sjzpp.us-west-1.elasticbeanstalk.com >>
/Users/jayxu/eb_log.txt
0 * 1-7 6 * curl http://34.71.117.126 >> /Users/jayxu/ae_log.txt
0 * 1-7 6 * curl http://stock-predict-jay-rakei3ugwq-uc.a.run.app >> /Users/jayxu/cr_log.txt
```

Figure 4.15: The Cron-job script to send request every hour for 7 days

Load Testing and Autoscaling

Load test application **Locust** is used. It is an open source load testing tool that defines user behavior with Python code, and swarms the target system with millions of simultaneous users [29].

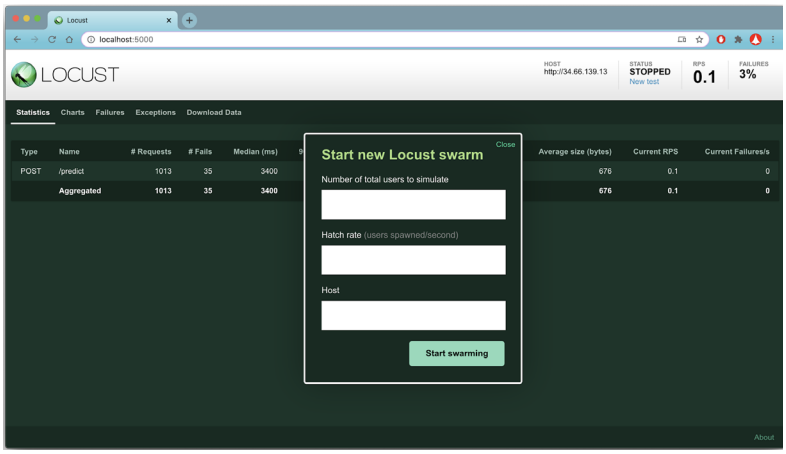


Figure 4.16: Locust interface

The **specs** used are:

Number of users: 10

Total number of requests: 1,000 (10 users sent requests randomly)

Intervals between requests: 15 seconds, 1 minute, and 5 minutes.

Script: *locustfile.py*

ML model: Stock prediction service

Test environment: Run Locust on a MacBook to act as a client outside the cloud.

```
from locust import HttpLocust, TaskSet, task, between
import time

# create a task to send request to ML service every 5 minutes
class WebsiteTasks(TaskSet):
    @task
    def predict(self):
        self.client.post("/predict",{'fileName': 'GOOG.csv'})
        #wait for 5 minutes
        time.sleep(300.0)

#configure the user settings in Locust Service
class WebsiteUser(HttpLocust):
    task_set = WebsiteTasks
    wait_time = between(0, 1)
```

Figure 4.17: *locustfile.py* for interval 5 minutes

Case	Hosting Service	Failure rate		
		every 15 seconds	every 1 minutes	every 5 minutes
Case 1: RESTful API on container platform	**RESTful API on AWS**	13%	4%	2%
	RESTful API on GCP	12%	4%	3%
Case 2: Serverless	**Google Cloud Run**	13%	10%	4%

Table 4.1 The failures rates of sending total 1000 requests in different time intervals

From the result in **Table 4.1**, for all implementations, when interval span between requests increases, the failure rate goes down.

This can be explained as two possibilities:

- Cold start latency of new host during auto scaling, due to warm up time. With requests frequency increasing, more hosts are being launched. Host launching time is around one minutes.
- Each host can handle 5-8 clients. As requests swarm at the same time, the web server gives HTTP 500 error for memory shortage (**Figure 4.18)**. This can possibly be mitigated

at application level by counting how many clients each host can serve and adjust the cluster size in the beginning.

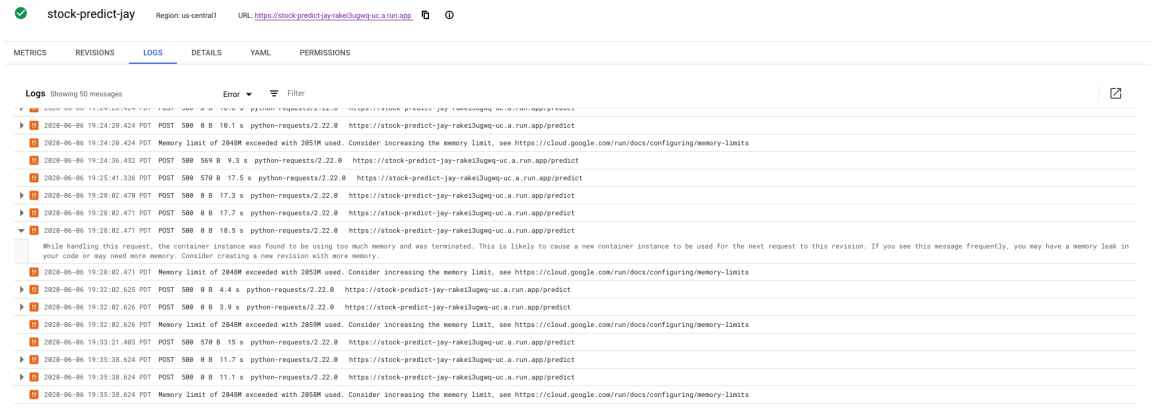


Figure 4.18 Details of HTTP 500 Error in Cloud Run

The suggestions are

- increase number of nodes in the cluster at the start if anticipate large number of concurrent requests, and
- increase auto scaling upper limit.

The conclusion is that through system configuration and tuning, the ML system can be set up to support a large number of users and service requests.

Host Failover

Host failover is tested by manually terminating a running host. **Figure 4.19** and **Figure 4.20** show that both AWS and GCP can detect a terminated instance and spin up a new instance in less than 40 seconds.

Figure 4.19: The activity history of auto scaling group in the ECS cluster

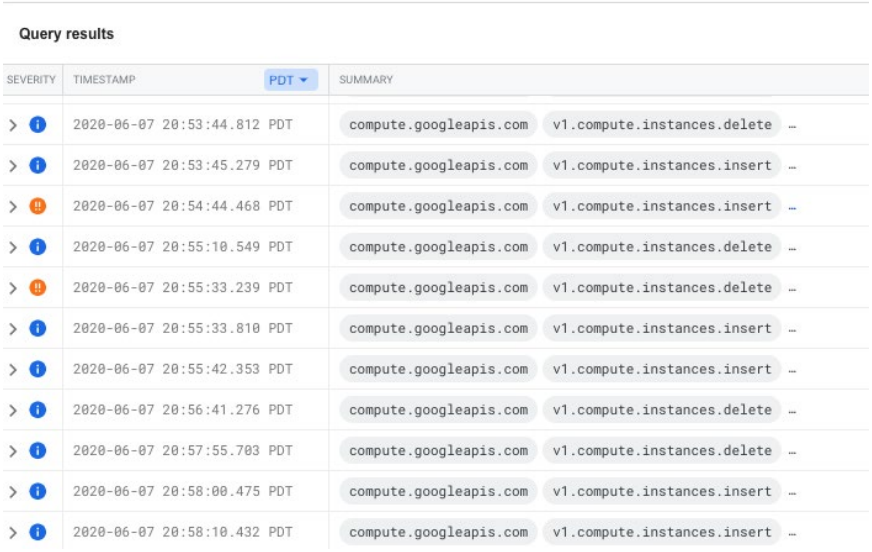


Figure 4.20: The log history of auto scaling in the GKE cluster

5. Results and Discussions

This section discusses the use of MSC/R design pattern in case studies to methodically design complex ML systems quickly and reliably and address the common problems.

5.1 Effectiveness in Aiding ML System Design

Separation of Concerns

MSC/R provides good abstraction and insight of ML system structure that help teams to quickly identify their tasks, roles and responsibilities. Each layer clearly marks the center of focus. With separation of duties, teams can work in parallel and improve project throughput. For example, Data scientists can focus on producing quality models, without worrying about setting up the environment, or about the discrepancy between the development and production environments. MLOps engineers can quickly build a prototyping stack without waiting for the ML model from data scientists. The prototype stack can be evolved into a production replica based on the experiment data scientists are doing. At the same time, client developers can concentrate on customer use cases.

Design by Contract

The connectors concept in the MSC/R keeps teams disciplined to define interfaces and collaboration protocols. This enhanced quality of design keeps components modular, and greatly enhances collaboration.

Reusability

Through isolation of concerns, design by contract and close collaboration, MLOps are free to practice best software engineering practices. For example, container technology has proven to be very effective to support distributed computing. MLOps capability to identify infrastructure abstraction and create templates greatly improves team's effectiveness by leveraging the cutting edge cloud technology.

5.2 Solving common problems

Problem-1

Lack of prototype stack mirroring production environment

With separation of concerns, data scientists can rely on MLOps to build the prototype stack that mirrors the production environment. MLOps is the domain experts in IT technology, system design and cloud technology. With proper requirements, standing up a ML development environment in public clouds in today's world is relatively simple and cost effective. Data scientists no longer need to struggle with using personal laptops, trying to piece together different tools and utilities to end up with “pipeline jungle.”

Problem-2

Programming style conflict. Data scientists tend to develop models with a monolithic program, not following software engineering best practices.

The connector components in MSC/R provide guidelines for teams to follow best practice to create well defined interface, to design by contract, and to heed modularization. For instance, it enforces data scientists to compose code in the form of functions or microservices to be compatible with the production ecosystem.

Problem-3

System design anti-patterns. Glue code and pipeline jungles, causing integration issues.

Following the discipline of adopting system design best practices, such as separation of concerns, design by contract to produce more modularized code. For example, in order to leverage container technology, ML code needs to be cleaned to get rid of glue code, and pipeline jungles. With close collaboration with MLOps, data scientists and teams can test more often, get rid of experimental code and dead code paths, and deal with technical debt and anti-pattern practice quickly to reduce integration issues.

5.3 Meeting Success Criteria

Criteria-1

Reduced time and difficulty to deploy ML models to production

The prototyping ML pipelines built in section 4 are relatively simple, they have all the components required for the production infrastructure. With proper upgrades in memory and computing power (e.g. using GPUs or TPUs), and changes in configuration the system can be converted to production grade in a short time.

Criteria-2

Capability to scale up/down horizontally and automatically.

The top public clouds such as AWS, GCP and Azure all have very mature mechanisms for horizontal scalability. As shown in the case study, auto scaling is seamless and very easy to configure.

Criteria-3

Live model monitoring, tracking and retraining

All the required technologies, such as AWS CloudWatch, are readily available to implement model monitoring. Serverless functions are effective tools for event triggers and event handling. The key is collaboration among teams to agree on a design solution. MSC/R also serves as a discipline and reminder that this task sits squarely on both teams.

5.4 Thoughts on using Serverless platform, Fully managed platform SageMaker

For ML systems running fully managed and serverless ML platforms, such as AWS SageMaker, and Google CloudRun, there are still separation of concerns and the interfaces among teams as described in the Connector pattern. Those platforms provide tools for better pipeline automation in production, and shorter development lifecycle, but the work and responsibility remain and have to be shared divided among teams.

For example, once a prototype pipeline is configured, AWS SageMaker enables developers and data scientists to build and train models using Notebook, and immediately deploy the model for integration and testing. Therefore, it enhances the CI-CD-CT process (continuous integration, continuous delivery, continuous testing). It comes with cost, for example, the price comparison between SageMaker and EC2 is in Table 5.2. The pick-and-choose approach used for this project provides a good opportunity to understand how different tools work together. It also provides flexibility for MLOps to customize and tune individual components, such as choosing different CPU/GPU, data storage on the pipeline and make the pipeline more cost effective.

Both AWS and GCP offer serverless options for deploying machine learning models.

Pros	Cons
Reduce the workloads and steps	Resource Limitation
Reduce the cost	Lock-in to cloud provider, hard to cross platform
Full managed by cloud provider	May lose certain critical insight into their services
Easy deployments and scalable	Latency of warm start

Table 5.2 Pros and Cons of Serverless

By using serverless, it reduces the workload of infrastructure administration and operation, and makes it easy to deploy a service. In AWS Lambda, it only needs 4 steps to deploy a Lambda Function. In Google Cloud Run, service can be deployed in only 1 step -- fill basic info and launch. It could potentially lower the cost of operations because of the pay-per-execution model. It allows developers to focus on application development without worrying management. However, the runtime environment is a blackbox. For large and complex systems, it's difficult to gain insight and troubleshoot problems.

Specifically, it also comes with drawbacks:

- Users may lose certain critical insight into their services, such as stack trace.
- Current serverless computing platforms have many limits on the computing and storage resources. AWS Lambda service has limitations on the deployment package size (250 MB, unzipped) and local disk size (512MB), that may prohibit it from being used for running deep learning algorithms with large models [31]. In fact, both models failed to deploy as a Lambda function due to the package size limitation. GCP also has limitations on CPU and memory size, in Cloud Functions, it has restrictions such as max deployment size 100 MB (compressed) / 500 MB (uncompressed) [37]. Therefore, serverless is not suited for heavyweight data processing currently.
- It totally relies on the cloud-provider's ecosystem, and users are locked into the vendor.
- Serverless also has the latency issue due to cold starts. As **Table 5.1** shows the failure rates on GCP Cloud Run are higher compared with non-serverless approaches.

5.5 Recommendation (from MSC/R perspective)

Following the **MSC/R** pattern, MLOps team can have a clear guideline to plan and deploy trained machine learning models.

Below is a list of recommendations based on experience from this project to design and implement a ML pipeline in a commercial cloud.

- Understand and Choose ML models

  Machine learning algorithms classify to two groups: supervised learning and unsupervised learning. Under supervised learning there are two common types of models based on classification algorithms, e.g. used for image classification, and regression algorithms, e.g. used for stock prediction; these models are freely available on GitHub and are good candidates to start with. There are several repositories of already trained models available in GitHub; a common one is given at [32]
- Learn and choose ML platforms and libraries, such as TensorFlow, PyTorch, Keras, etc.
- Knowledge of Cloud Computing Services (from AWS, GCP, AZURE): Container, Scaling, Protocols, etc.
- Research on what hardware components and spec needed to train the models. Can build your own, best to use Cloud service.
- Prototype

  Python is a popular programming and scripting language for machine learning projects; it is commonly used by data scientists, software developers and system engineers. Python has a large repository of modules and libraries readily freely available online. For this project, it has proven to be very practical for writing front end, mid-tier integration and model training code.

6. Conclusions and Future Work

6.1 Conclusions

Case study shows that by following MSC/R pattern, MLOps can quickly create a pipeline for data scientists to use for their prototyping and testing and replicate the pipeline as a template for designing large scale environments that mirror production quickly and effectively.
The tasks can be better organized, and the collaboration is much easier with well-defined contracts and interfaces among teams. In the case study, a prototype ML stack can be configured and deployed in a matter of a few days in the public cloud. Load testing also shows the auto scaling and load balance technology in those clouds are very mature and reliable. The three success factors are very achievable if the process is guided by discipline and best practices. Leveraging infrastructure as code practice, the prototyping stack can be easily converted into a template; by upgrading the specification of certain components such as CPUs and memories, cluster size, it can be turned into production grade without huge effort; the template can be used to stand up more environment for testing staging with same architecture, significantly reduce the portability issue.

Using design patterns in software development helps address design complexity and quality. It can also adapt to ML service development. In this paper, an abstract Model-Service-Client + Retraining **(MSC/R)** is summarized from existing ML pipeline practices and simply implemented using 3 popular approaches. It is quite obvious that with the **MSC/R** pattern, the ML pipeline can be separated of concerns and reusable. This may provide a good suggestion in industry practising.

6.2 Future Work

End to End Pipeline Case Study

Collaborate with data scientists to conduct case study to create pipeline that start from data collection to model serving

More Patterns as Discipline and Best Practice

To further develop the **MSC/R** design pattern, more ML application patterns, classifying patterns and identifying anti-patterns will be collected. More detailed insight of each component in MSC/R and the connectors between them will be developed, such as Data infrastructure that produce live training data; Model Serving Infrastructure to differentiate Client requests, such as batch or real-time; Security infrastructure that protect the sensitive data and services.

7. References

1. International Data Corporation (2019). “Worldwide Spending on Artificial Intelligence Systems Will Be Nearly $98 Billion in 2023, According to New IDC Spending Guide.” Retrieved from https://www.idc.com/getdoc.jsp?containerId=prUS45481219
2. “Why do 87% of data science projects never make it into production” Retrieved from https://venturebeat.com/2019/07/19/why-do-87-of-data-science-projects-never-make-it-into-production/?ref=hackernoon.com
3. J. Bughin, E. Hazan, S. Ramaswamy, M. Chui, T. Allas, … M. Trench. (2017) "Artificial Intelligence The Next Digital Frontier?". *McKinsey*. McKinsey Global Institute. https://www.mckinsey.com/~/media/McKinsey/Industries/Advanced%20Electronics/Our%20Insights/How%20artificial%20intelligence%20can%20deliver%20real%20value%20to%20companies/MGI-Artificial-Intelligence-Discussion-paper.ashx
4. Mitesh Soni. (2014). “Cloud computing basics—platform as a service (PaaS)”, *Linux Journal February 2014*. Retrieved from https://www.linuxjournal.com/content/cloud-computing-basics%E2%80%94platform-service-paas
5. Ryan Dawson (2019). “why is devops for machine learning so different”, Retrieved from https://hackernoon.com/why-is-devops-for-machine-learning-so-different-384z32f1
6. DevOps(n.d.). Retrieved from https://en.wikipedia.org/wiki/DevOps
7. Model-view-controller (n.d.). Retrieved from https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
8. “YOLO: Real-Time Object Detection”. Retrieved from https://pjreddie.com/darknet/yolo/
9. J. Redmon and A. Farhadi, “YOLOv3: An Incremental Improvement”. Retrieved from https://pjreddie.com/media/files/papers/YOLOv3.pdf
10. Z. Zhang’s GitHub, Retrieved from https://github.com/zzh8829
11. S. Kim’s GitHub, Retrieved from https://github.com/hunkim/DeepLearningZeroToAll
12. S. Hochreiter and J. Schmidhuber. (1997). “LONG SHORT-TERM MEMORY”, *Neural Computation 9: 1735-1780, 1997,* Retrieved from https://www.researchgate.net/publication/13853244_Long_Short-term_Memory
13. C. Olah. (2015). “Understanding LSTM Networks” , Retrieved from https://colah.github.io/posts/2015-08-Understanding-LSTMs/

14. "Alphabet Inc. (GOOG) Stock price", Retrieved from https://finance.yahoo.com/quote/GOOG/history?p=GOOG
15. Amazon Web Service, Retrieved from https://aws.amazon.com
16. Google Cloud Platform, Retrieved from https://en.wikipedia.org/wiki/Google_Cloud_Platform
17. Docker overview, Retrieved from https://docs.docker.com/get-started/overview/
18. Kubernetes, Retrieved from https://kubernetes.io/
19. "What is REST", Retrieved from https://restfulapi.net/
20. serverless computing. Retrieved from https://en.wikipedia.org/wiki/Serverless_computing
21. K. O'Leary and M. Uchida. "Common Problems with Creating Machine Learning Pipelines from Existing Code". Retrieved from https://storage.googleapis.com/pub-tools-public-publication-data/pdf/b50bc83882bbd29c50250d1e59fbc3afda3fb5e5.pdf
22. Design Patterns, Retrieved from https://sourcemaking.com/design_patterns
23. Amazon Elastic Container Service, Retrieved from https://aws.amazon.com/ecs
24. Google Kubernetes Engine, Retrieved from https://cloud.google.com/kubernetes-engine
25. AWS Elastic Beanstalk, Retrieved from https://aws.amazon.com/elasticbeanstalk/
26. Google App Engine Documentation, Retrieved from https://cloud.google.com/appengine/docs
27. Google Cloud Run, Retrieved from https://cloud.google.com/run
28. Choosing a Serverless Option, Retrieved from https://cloud.google.com/serverless-options
29. Locust - A modern load testing framework. Retrieved from https://locust.io/
30. "MLOps: Continuous delivery and automation pipelines in machine learning", Retrieved from https://cloud.google.com/solutions/machine-learning/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning#mlops_level_0_manual_process
31. M. Zhang, Y. Zhu, C. Zhang, J. Liu. (2019). "Video Processing with Serverless Computing: A Measurement Study". *In NOSSDAV '19: Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video, 2019. pp. 61-66*, Retrieved from https://dl.acm.org/doi/10.1145/3304112.3325608

32. Tensorflow model GitHub, Retrieved from https://github.com/tensorflow/models/tree/master/official#object-detection-and-segmentation
33. Amazon EC2 Pricing, Retrieved from https://aws.amazon.com/ec2/pricing/on-demand/
34. Amazon SageMaker Pricing, Retrieved from https://aws.amazon.com/sagemaker/pricing/
35. matplotlib, Retrieved from https://matplotlib.org/
36. Mao, Qi-Chao & Sun, Hong-Mei & Liu, Yan-Bo & Jia, Rui-Sheng. (2019). Mini-YOLOv3: Real-Time Object Detector for Embedded Applications. *IEEE Access. PP. 1-1.* 2019, Retrieved from https://www.researchgate.net/publication/335865923_Mini-YOLOv3_Real-Time_Object_Detector_for_Embedded_Applications
37. Google Cloud Functions, Retrieved from https://cloud.google.com/functions/quotas
38. Google Cloud Run, Retrieved from https://cloud.google.com/run/quotas
39. H. Washizaki, H. Uchida, F. Khomh and Y. Guéhéneuc, "Studying Software Engineering Patterns for Designing Machine Learning Systems," *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, Tokyo, Japan, 2019, pp. 49-495. Retrieved from https://ieeexplore.ieee.org/document/8945075
40. D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, … D.Dennison. (2015) "Hidden Technical Debt in Machine Learning Systems." In *Advances in Neural Information Processing Systems 28 (NIPS 2015)* https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems
41. "What are Azure Machine Learning pipelines?". Retrieved from https://docs.microsoft.com/en-us/azure/machine-learning/concept-ml-pipelines
42. "Why MLOps (and not just ML) is your Business' New Competitive Frontier". Retrieved from https://www.aitrends.com/machine-learning/mlops-not-just-ml-business-new-competitive-frontier/
43. "Introduction to Infrastructure Patterns". Retrieved from https://infrastructure-as-code.com/patterns/
44. "Rules of Machine Learning:Best Practices for ML Engineering". Retrieved from https://developers.google.com/machine-learning/guides/rules-of-ml/#training-serving_skew

45. M. Jordan, “Artificial Intelligence — The Revolution Hasn’t Happened Yet”. Retrieved from https://medium.com/@mijordan3/artificial-intelligence-the-revolution-hasnt-happened-yet-5e1d5812e1e7
46. MLOps, Retrieved from https://en.wikipedia.org/wiki/MLOps
47. V.M Megler, Managing Machine Learning Projects. Retrieved from https://d1.awsstatic.com/whitepapers/aws-managing-ml-projects.pdf
48. Non-Functional Requirements: Scalability. Retrieved from https://www.modernanalyst.com/Resources/Articles/tabid/115/ID/4968/Non-Functional-Requirements-Scalability.aspx
49. Web API design, Retrieved from https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design
50. Retrieved from http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/interface.html
51. Microservices architecture. Retrieved from http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/interface.html

8. Appendix

Artifacts:

GitHub Repository: https://github.com/jayxu96/Jay-Master-Project

Table 8.1 below lists underlying cloud services used.

Cloud Service	AWS	GCP
Container orchestration	Elastic Container Service (ECS)	Google Kubernetes Engine (GKE)
Container Registry	Elastic Container Registry (ECR)	Google Container Registry
Computing	Elastic Cloud Computing Service (EC2)	Google Computing Engine (GCE)
Storage	Simple Storage Service (S3)	Google Cloud Storage
Auto Scaling	Auto Scaling Group	Auto Scaling
Load Balancing	Elastic Load Balancer (ELB)	Cloud Load Balancing
Monitoring	CloudWatch	Cloud Monitoring
Application deployment	Elastic Beanstalk (EB)	Google App Engine (GAE)
Serverless	Amazon Lambda Function	Google Cloud Run

Table 8.1: The Cloud services used in implementations

Software and Libraries Used:

- Python 3.7
- Docker
- AWS CLI: AWS Command Line Interface to manage AWS services
- Google SDK: Google Cloud Command Line tools to manage GCP services
- Tensorflow 2.1: an open source machine learning library
- Numpy: a Python library for scientific computing
- matplotlib: a comprehensive library for creating static, animated, and interactive visualizations in Python [35]

- OpenCV: an open- source library used for real-time computer vision
- Flask: a light-weight Web Server Gateway Interface web framework in Python
- Flask-RESTful: an extension of Flask to build REST APIs
- Gunicorn: a Python Web Server Gateway Interface HTTP server
- Boto3: an AWS SDK for Python to access various AWS services such as S3, EC2.