AN ABSTRACT OF THE MASTER'S PROJECT REPORT OF

Nandam Sriranga Chaitanya for the degree of Master of Science in Computer Science, Software Innovation Track presented on December 5, 2022.

Title: SmartPark

Abstract approved: ______________________________

Dr Will Braynen

SmartPark

by

Nandam Sriranga Chaitanya

A MASTER'S PROJECT REPORT

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 5, 2022
Commencement June 2022

Master of Science master's project report of Nandam Sriranga Chaitanya presented on December 5, 2022.

APPROVED:

Major Professor, representing Computer Science, Software Innovation Track

---

Director of the School of Computer Science

Dean of the Graduate School

I understand that my master's project report will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my master's project report to any reader upon request.

Nandam Sriranga Chaitanya, Author

TABLE OF CONTENTS

Page

TABLE OF CONTENTS (Continued)

Page

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Problem Statement and Scope of the Project

There is considerable vehicular traffic in universities these days. Vehicles of university staff, students, faculty, guests, visitors, and the local population of the city/town arrive at the university campus daily. The traffic trends vary based on the time and day of the week and peak on the occasion of events and games. As the traffic increases the process of searching for a parking lot becomes tedious and time-consuming which causes discomfort for the vehicle drivers. Identifying a parking lot that is the nearest, and finding the availability of empty parking space in it is challenging. Also, sign boards to the parking lots might not work as the space availability is still not known and may cause traffic congestion. Increased traffic also harms the environment due to pollution.

On event or game days, it becomes difficult as the demand for parking space raises exponentially, sometimes uncontrollably with most vehicles trying secure space on a few parking lots, that are nearer to the event or game location. Instead, Having the information on the parking lot location and the space upfront (in real time) leads to a better user experience, smoother traffic flow, and a healthier environment.

To meet this requirement, parking management organizations are trying to implement better and technologically advanced solutions. A cloud-based smart parking application will enable real-time parking availability monitoring providing better service to the end users as well as reducing the workload of the parking administrator or the parking management. Hence, it is intended to design an application that can track the parking lot occupancy information presenting the number of empty/full spaces in the parking lot through a cloud-based application.

Such an application is to be developed by implementing the core technology for tracking down the parking spaces occupancy and providing the information over RESTful APIs is to be Any client can consume the APIs to query for empty parking spaces in the nearby parking lot. The system logic should be intelligent enough to identify the nearest parking lot from the location of the mobile application user at the time of the query based on the

GPS location of the mobile application users at the time of the query.

The scope of the current project includes building a core technology module that processes the YouTube video and packages the information into RESTful APIs. These APIs form the back-end API layer enabling API queries for fetching the occupancy status of the parking lot at a given time substantiated with images of the parking lot. The APIs can be extended for advanced functionality and/or more APIs can be added along the same lines as the current API.

Chapter 2: Solution

2.1 Possible Solutions

In the below sections, I describe some possible solutions to this problem.

2.1.1 Security Guard

This can be one of the naive solutions given the evolution of technology to date. A security guard can be appointed with an hourly wage just to count the number of vacant parking lots every five minutes and update it in a registry. This registry can be used to know the number of vacant parking spaces in a parking lot at a given time. This solution is highly inefficient and requires very high manpower if considering the situations where the parking lot is very big, 24-hour monitoring of the lot, and unfavorable weather conditions.

2.1.2 Infrared Sensor System

Another solution might be there can exist an Infrared (IR) sensor system where two IR sensors are set up at the entry and exit points of the parking lot. So, whenever a vehicle enters the parking lot through the Entry point, the IR sensor will sense the event as vehicle entry and when the vehicle leaves the lot, the IR sensor at the exit point will sense it as a vehicle exit event. For each vehicle entry event, the total number of vehicles will be incremented by 1 and for each vehicle exit event, it is decreased by 1. So, this way we have the number of vehicles in the parking lot at a given point in time and we can calculate the number of vacancies by performing simple math. This solution seems to be good but is very cost-draining when it comes to building it on large scale with numerous parking lots as each parking lot needs at least two IR sensors and also the supporting technology to manage and maintain the IR sensor data. Also, huge investments to buy hardware have their challenges with the quality of the sensor, durability of the sensor in different weather conditions, and the accuracy of the sensor.

2.1.3 Ticketing System

A ticketing system for the parking lot, where a ticket is issued with a QR code on it upon entry into the parking lot, and the same QR code is scanned by a QR code scanner at the exit gate of the same parking lot and registered as a vehicle exit event. This solution is similar to the IR sensor system solution but a bit less cost draining when compared. Each parking lot at least needs One ticket printer machine and a QR code scanner, which again on a High scale needs a lot of hardware investment and maintenance.

2.1.4 CCTV Footage Monitoring System

Another way of solving this problem can be to monitor the CCTV footage and manually update the number of vacancies in the parking lot. But this solution needs human intervention and is similar to the security guard solution given the person who is monitoring doesn't need to be physically present in the parking lot.

2.1.5 Artificial Intelligence and Computer Vision

Probably, the best way to solve this problem is to build an application with the help of Artificial Intelligence (AI) and computer-vision techniques to get the parking lot occupancy information in real-time in a cost-effective manner. I have chosen this solution and built the application. The AI models can be trained to process the CCTV video like humans with precision to identify the number of vehicles present in the lot to calculate the empty spaces available. Also, machine learning algorithms can be used to identify the nearest parking lot from the location where the query is made from.

2.2 Existing Solutions

2.2.1 Smart Parking - Unabiz

This is a solution that has been implemented and deployed in Taipei in the Republic of China and some parts of Singapore [7]. Internet of Things (IoT) based sensors are used to detect the availability of each parking lot. Each parking lot has an IoT sensor installed which either detects the presence of the vehicle. Also, there are other types of sensors present that act as a barrier in the middle of the parking spaces and go down only if the payment for parking space is done and then the vehicle can be parked in that space. There is a mobile application built which can be used to navigate to a particular parking space in the lot and pay for the parking space.

2.2.2 Parking Management - Automatic Number Plate Recognition (ANPR) Software

This is a camera-based ANPR system [4] that detects the number plates of the vehicles upon entry into the parking lot and increases the count of the number of vehicles and updates the total count upon exit of the vehicle from the parking lot. This solution also provides a mobile application for payments.

2.2.3 Parking Guidance system (PGS) - PDX International Airport

PDX was one of the first airports in North America to implement a PGS. Their original system consisted of ultra-sonic sensors, now at the end of their life [3]. As technology has evolved, Park Assist's M4 PGS provides PDX with more comprehensive technology as it is an advanced, durable and intelligent camera-based sensor system. Original system prior to M4 PGS consisted an ultrasonic sensor above each parking lot to detect the presence of a vehicle whereas the new system consists of multiple M4 camera sensors with each sensor monitoring four parking spaces. The transition into a new system helped the PDX management to reduce the investment on hardware equipment.

2.3 My Solution

To solve this problem I am using deep learning based computer-vision AI to recognize vehicles in a given image with utmost accuracy in real time. The images captured from the live YouTube video stream are analyzed to infer vehicle presence and the logic around calculates the total number of vehicles in each image. This process repeats with 40 seconds time interval between consecutive executions. Further RESTful APIs are implemented to enable queries from the User Interface application (typically a Mobile Application) for fetching the occupancy status on demand.

The solution architecture involves two tasks running in parallel. I refer to the first task as the ingress pipeline and to the second task as the egress pipeline.

The ingress pipeline is designed to fetch the images from the camera's YouTube video feed, preprocess the fetched images, and feed them to a computer-vision vehicle-detection model for predicting the vehicle presence and returning the vehicle count and image with detected vehicle annotations on it. In the next step of the pipeline the annotated images are stored in the cloud storage, and the corresponding storage URL is stored in the local database. In addition, GPS location information is used to identify the nearest parking lot.

Task 2 involves the REST API design that responds to GET HTTP requests from the mobile application. The API has two GET methods that are designed to fetch the data of the latest processed image of the parking lot and send it to the application as a response.

The ingress pipeline fetches images from the university camera's live YouTube feed of the parking lot, while the egress pipeline executes on-demand responding to the requests from the application as applicable. This design approach is taken to optimize the cloud resources' engagement and utilization.

The REST API which is a part of the egress pipeline, is designed to provide the occupancy status, images (as metadata) and the nearest parking lot ID, and the location GPS. The system design allows the implementation of additional APIs to fetch occupancy reports for a given duration, extending the capability to identify the type of vehicles and duration of occupancy of a specific car, in the future.

2.4 Advantages of My Solution

The existing solutions use a lot of hardware equipment such as Internet of Things (IoT) sensors, sensor-based barriers, cameras and ticket generators. My solution needs minimal hardware equipment: camera and its accessories. The number of cameras required is determined only by evaluating the size of the parking lot according to the coverage of each camera. In comparison, my solution is cost effective in terms of investment into hardware equipment and also takes minimal effort and time for deployment and scaling.

If open parking lots with multiple entries and exits are considered the existing solutions are very complex to make use of. Whereas my solution is very simple to be deployed and can be apt for such parking lots as all it needs is the images of the parking lot from a camera.

2.5 Technical Constraints and Shortcomings

2.5.1 Camera View and Coverage

The view from the existing camera limits the coverage of the parking lot giving a clear view of a section of the parking lot. This limits the analytics coverage to a lesser number of parking spaces. The cameras in parking garages or underground parking lots can capture only a certain region as the ground clearance of the cameras might be between 15 to 20 feet which is not adequate to cover a large region. The position of the camera is fixed and cannot be moved because the masking techniques of the current solution are built specific to one parking lot. These masking techniques have to be re-worked for every new parking lot. If the camera's position is changed, it changes the view of the camera and affects the upcoming stages of the solution by ingesting a improper image.

2.5.2 Climatic Conditions

Rain or dust can blur the camera's view and impact the efficacy of the vision model's predictions leading to erroneous results. The image pre-processing implements the best possible cropping to mitigate this risk.

2.5.3 Occlusions

Further, the camera view is blocked by the trees present in the parking lot eliminating the spaces behind the tree view and disabling the model from detecting any present vehicles.

Chapter 3: The REST API

This solution provides a REST API for a client to consume and retrieve the occupancy information of a parking lot or a set of parking lots. A UUID stands for Universally Unique Identifier. All the parking lots are assigned with a UUID. Client applications can call this API to get a list of available parking lots and their respective UUIDs and also to get occupancy information of a particular parking lot using its UUID.

This API consists of two endpoints: `GET /parking_lots` and `GET /parking_lots/{parking_lot_UUID}`. Both endpoints return data in JSON format.

3.1 The `GET /parking_lots`endpoint

This GET method on route `GET /parking_lots` vends a list of available active parking lots along with their UUIDs.

The request to the API should send the location as part of the request headers:

```
{
 "location": "Oregon State University Campus",
 "City": "Corvallis",
 "State": "Oregon",
 "ZIP": "97333"
}
```

The sample API response containing the list of available parking lots:

```
{
 "parking_lots": [
    {
     "UUID": "123e4567-e89b-12d3-a456-426614174000",
     "name": "Tabaeu Hall"
    },
    {
     "UUID": "00112233-4455-6677-c899-aabbccddeeff",
     "name": "Wilson Hall"
    }
]
}
```

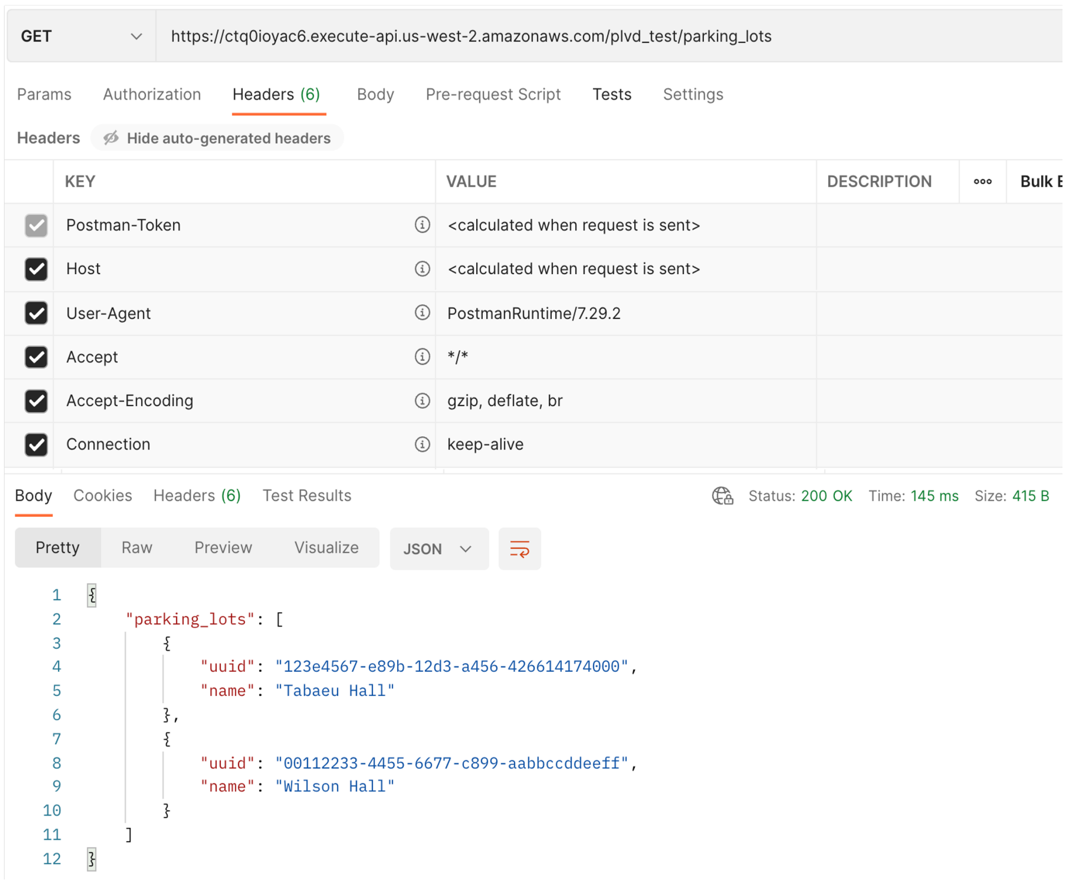
3.2 The `GET /parking_lots/{parking_lot_UUID}` endpoint

This GET method on the route `GET /parking_lots/{parking_lot_UUID}` returns the occupancy information of a particular parking lot using its UUID. The parking lot occupancy information constitutes the parking lot name, parking lot UUID, number of vehicles in the lot, number of vacancies in the parking lot, timestamp, and a link to the detection image on the S3 bucket. For example, a client call to this API might return a response like the below:

```
{
  "parking_lot_name": "Tabaeu Hall",
  "parking_lot_UUID": "123e4567-e89b-12d3-a456-426614174000",
  "number_of_vehicles": 12,
  "number_of_empty_parking_slots": 29,
  "timestamp": "2022-11-10T01:04:51.640824",
  "image_s3_http_url": "https://s3-us-west-2.amazonaws.com/
     detectionlog/prediction_images/2022-11-10T01:04:51.640824
     _nxp5w8p.jpg"
}
```

3.3 Test Results

I tested the SmartPark API using Postman. Postman is an HTTP client that is commonly used for testing of REST APIs. Figure 3.1 shows test result for the `GET /parking_lots` endpoint. Figure 3.2 shows test result for the `GET /parking_lots/{parking_lot_UUID}` endpoint.



GET https://ctq0ioyac6.execute-api.us-west-2.amazonaws.com/plvd_test/parking_lots

Headers

KEY	VALUE	DESCRIPTION
Postman-Token	<calculated when request is sent>	
Host	<calculated when request is sent>	
User-Agent	PostmanRuntime/7.29.2	
Accept	*/*	
Accept-Encoding	gzip, deflate, br	
Connection	keep-alive	

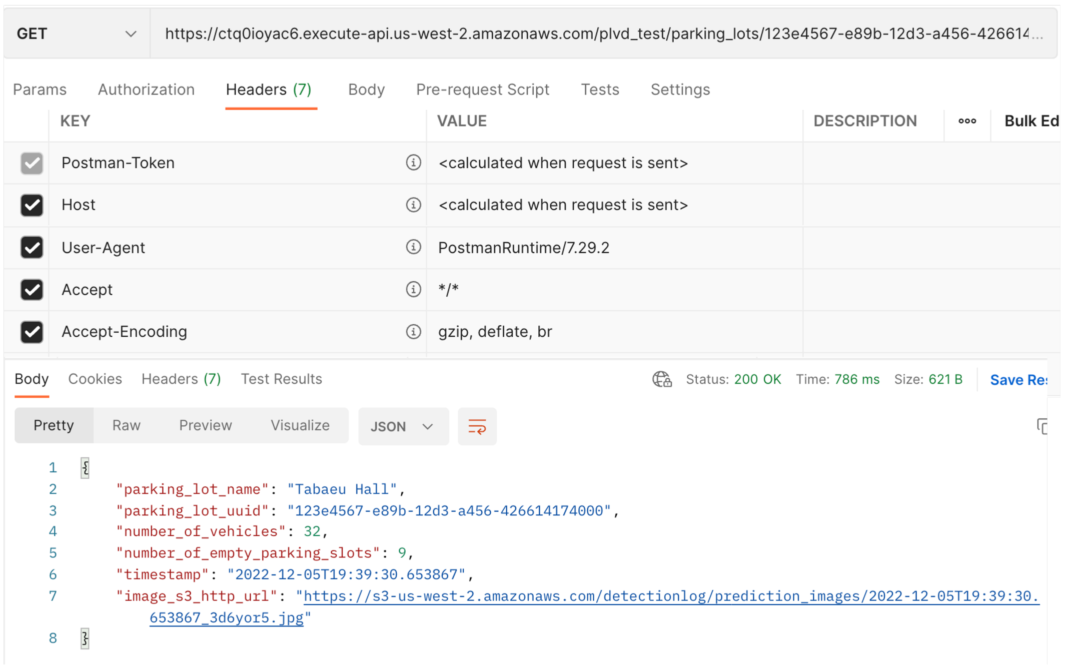
Status: 200 OK Time: 145 ms Size: 415 B

```
{
    "parking_lots": [
        {
            "uuid": "123e4567-e89b-12d3-a456-426614174000",
            "name": "Tabaeu Hall"
        },
        {
            "uuid": "00112233-4455-6677-c899-aabbccddeeff",
            "name": "Wilson Hall"
        }
    ]
}
```

Figure 3.1: GET /parking_lots Endpoint Test Results

3.4 Swagger Documentation

I have documented my REST API using the OpenAPI specification (version 3.0), the REST API: documentation for which can be found at https://app.swaggerhub.com/


GET https://ctq0ioyac6.execute-api.us-west-2.amazonaws.com/plvd_test/parking_lots/123e4567-e89b-12d3-a456-426614...

Params Authorization Headers (7) Body Pre-request Script Tests Settings

KEY	VALUE	DESCRIPTION
Postman-Token	<calculated when request is sent>	
Host	<calculated when request is sent>	
User-Agent	PostmanRuntime/7.29.2	
Accept	*/*	
Accept-Encoding	gzip, deflate, br	

Body Cookies Headers (7) Test Results — Status: 200 OK Time: 786 ms Size: 621 B

Pretty Raw Preview Visualize JSON

```
{
    "parking_lot_name": "Tabaeu Hall",
    "parking_lot_uuid": "123e4567-e89b-12d3-a456-426614174000",
    "number_of_vehicles": 32,
    "number_of_empty_parking_slots": 9,
    "timestamp": "2022-12-05T19:39:30.653867",
    "image_s3_http_url": "https://s3-us-west-2.amazonaws.com/detectionlog/prediction_images/2022-12-05T19:39:30.
        653867_3d6yor5.jpg"
}
```

Figure 3.2: GET /parking_lots/{parking_lot_UUID} Endpoint Test Results

Chapter 4: Solution Design and Architecture

4.1 System Components

SmartPark consists of an ingress pipeline and an egress pipeline (as described in Chapter 2). The ingress uses a machine learning model to detect cars in parking-lot image. The egress pipeline includes a REST API (described in Chapter 4) that vends the results from the ingress pipeline. The machine learning model used by the ingress pipeline is a YOLOv5 vehicle-detection model. This model was built by [2] using a machine learning technique called *deep learning*.

4.2 Design Strategy

A HD camera was set up by University Housing and Dining Services (UHDS) on top of the "Tabeau Hall" at Oregon State University to monitor the progress of the "Arts and Education Complex (AEC)" project. This camera feed is used in three different ways:

- A "Box" storage is set up to store images that are captured every five minutes. The images are stored in folders where each folder has images related to one single day.
- A website that displays the image captured from the camera every minute.
- A Live YouTube stream.

The design is split into two tasks running in parallel. Task 1 is called the ingress pipeline and the task 2 is called the egress pipeline. The ingress pipeline executes on a virtual server, an "EC2 instance" on Amazon Web Services (AWS). The ingress pipeline runs every 40 seconds. I will refer to this time interval as *execution interval*. The egress pipeline runs on an AWS service called "Lambda". The execution of the pipeline is triggered by a call to the REST API.

The ingress pipeline uses the live YouTube stream from the university camera to fetch images. The task of this pipeline is to continuously process the parking lot images fetched from the live YouTube stream, analyze, predict and count the vehicles present in the parking lot then and store the results (annotated images) in Amazon S3 storage and store the corresponding annotated image URL and Occupancy information along with timestamp in MongoDB.

For the given architecture design of the application, the system breaks if the "Execution Interval" is decreased. In the future, there can be two ways to solve this problem. One is to implement the Amazon Simple Queue Service (SQS) where all the fetched images are placed in a queue and processed sequentially. The other way would be to use the autoscaling feature of AWS EC2 which allocates the cloud resources on demand and de-allocates them when there is no requirement or usage. If both the solutions are compared the SQS solution doesn’t serve the purpose of the application which is to retrieve the real-time occupancy status of the parking lots as queues process the data in sequential order. But the autoscaling method works as it is adaptive in terms of allocation and deallocation of cloud resources based on the need.

The video and image analytics and subsequent data and metadata storage involving videos, images, and reports, it is intended to use a NoSQL database like MongoDB and blob storage like Amazon S3. Amazon S3 allows direct access to large files via the browser. Also, as data may be required to be stored for a longer period for use by analytical services, it makes sense to store it in S3 as it helps keep the storage costs low.

4.3 Pre-processing Strategies

In the pre-processing stage of the ingress pipeline, two types of experiments were performed based on two different approaches, they are:

General Approach: *Images were cropped* $\rightarrow$ *converted to grayscale* $\rightarrow$ *pushed for prediction*.

Mask-Based Approach: *Images were cropped → converted to grayscale → masked unwanted regions → pushed for prediction.*

The image is cropped to consider only the parking lot part of the image and eliminate the unwanted parts.

The image transformation through the ingress pipeline in general approach is shown in the below figure 4.1.

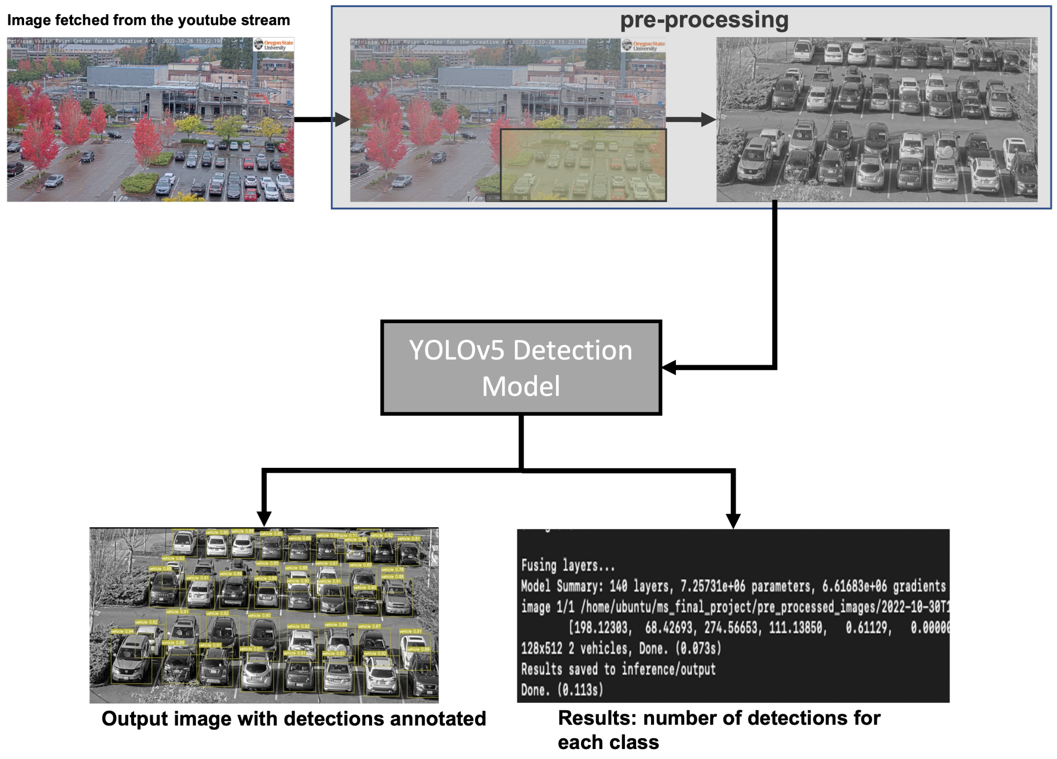


Image fetched from the youtube stream
pre-processing
YOLOv5 Detection Model
Output image with detections annotated
Results: number of detections for each class

```
Fusing layers...
Model Summary: 140 layers, 7.25731e+06 parameters, 6.61683e+06 gradients
image 1/1 /home/ubuntu/ms_final_project/pre_processed_images/2022-10-30T1
        [198.12303,  68.42693, 274.56653, 111.13850,   0.61129,   0.0000
128x512 2 vehicles, Done. (0.073s)
Results saved to inference/output
Done. (0.113s)
```

Figure 4.1: Image transformation through the ingress pipeline using general approach

The image transformation through the ingress pipeline in a mask-based approach is shown in the below figure 4.2.

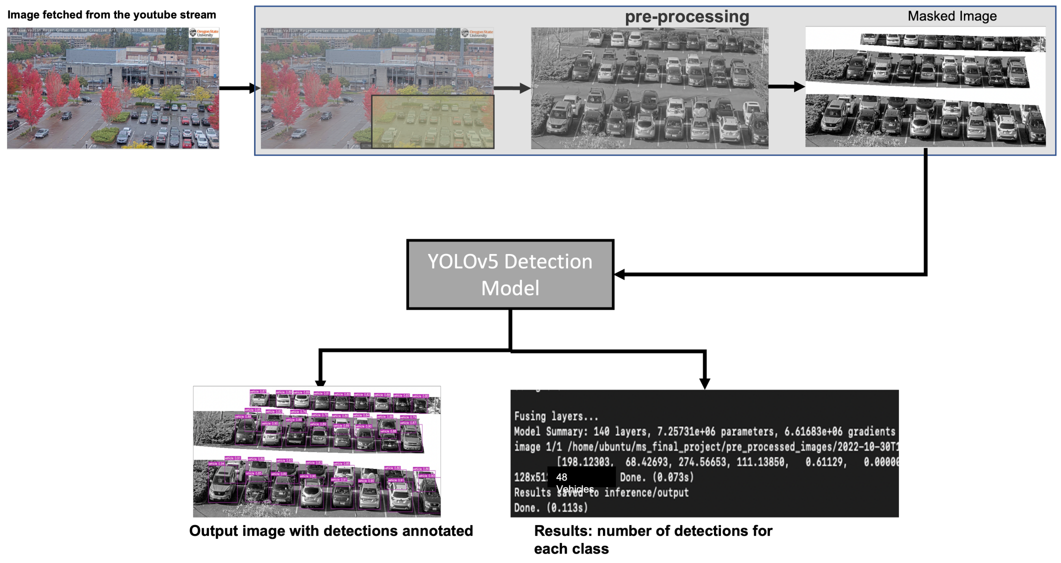


Image fetched from the youtube stream
pre-processing
Masked Image
YOLOv5 Detection Model
Output image with detections annotated
Results: number of detections for each class
48 Vehicles

```
Fusing layers...
Model Summary: 140 layers, 7.25731e+06 parameters, 6.61683e+06 gradients
image 1/1 /home/ubuntu/ms_final_project/pre_processed_images/2022-10-30T1
        [198.12303,  68.42693, 274.56653, 111.13850,   0.61129,   0.0000
128x51                 Done. (0.073s)
Results saved to inference/output
Done. (0.113s)
```

Figure 4.2: Image transformation through the ingress pipeline using mask-based approach

4.4 Model Inference Results

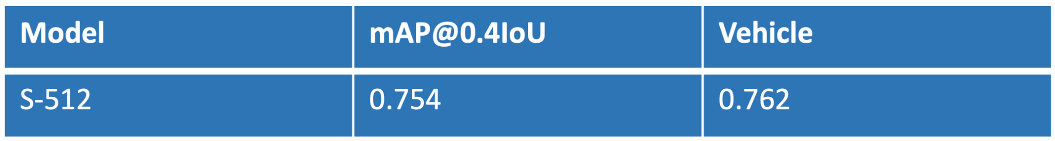
The figures 4.3 and 4.4 show the validation results of the detector for both mentioned approaches respectively in terms of mAP. The mAP stands mean Average Precision which one of the popular evaluation metric for object detection models. Average Precision is calculated as the area under a precision vs recall curve for a set of predictions. The recall is calculated as the ratio of the total predictions made by the model under a class with a total of existing labels for the class. On the other hand precision refers to the ratio of true positives to the total predictions made by the model. The area under the precision vs recall curve gives us the Average Precision per class for the model. The average of this value, taken over all classes, is termed as mean Average Precision. In object detection, precision and recall are not for class predictions, but for predictions of boundary boxes for measuring the decision performance. An IoU value greater than 0.5 is taken as a positive prediction, while an IoU value less than 0.5 is a negative prediction.

The egress pipeline involves responding to the client API requests (RESTful API) querying for parking lot occupancy information to know if the parking lot is empty/full and gets the following information:

- The number of vacant parking spaces.
- The number of vehicles in the lot.
- Parking lot name.
- Parking lot UUID.
- S3 bucket link to the image with vehicle detections annotated.

The design approach leverages the on-demand compute service (serverless computing) of AWS, AWS Lambda for fetching the data from the MongoDB that is deployed as a Docker service on the EC2 instance through REST API GET methods. The AWS API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. API Gateway is used to build REST API with GET methods which trigger an AWS Lambda when tried to access.

The pre-processing, post-processing and predictor modules are kept separate to ensure the predictor module can scale on a need basis namely. AI models reconfiguration, fine-tuning, retraining, and redeployment or replacing the model. This approach helps leverage the best models available at any given point in time as the models are evolving and the efficacy of models keeps evolving.



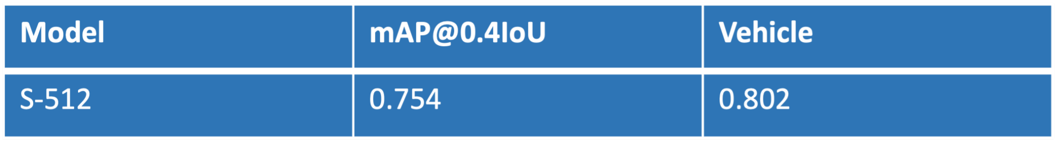
Model	mAP@0.4IoU	Vehicle
S-512	0.754	0.762

Figure 4.3: Validation results of the detector using the general approach for Vehicle class

Solution Architecture is shown in figure 4.5

4.5 Cloud Implementation Strategy

It is proposed to engage EC2 and serverless (Lambda) for computing, open source MongoDB on Docker or Amazon S3 storage for large file storage, and



Model	mAP@0.4IoU	Vehicle
S-512	0.754	0.802

Figure 4.4: Validation results of the detector using the mask-based approach for Vehicle class

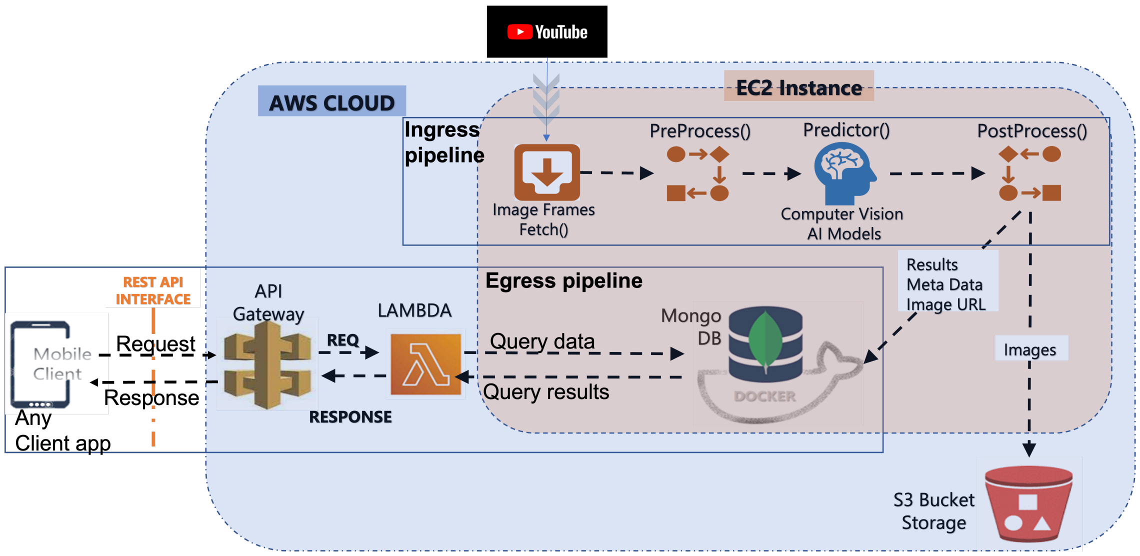


Figure 4.5: Solution Architecture

API gateway for client-side API service. Detailed justification for engaging these services is given below.

4.5.1 Compute Instances – EC2 and Serverless AWS Lambda

EC2 Instances on AWS are like servers in physical. The cost of EC2 service depends on the configuration of the EC2 and the duration of the EC2 run. As the application is required to be running 24 hours and 7 days a week (24/7) and is hosted on an EC2 instance, it is important to identify only those components required to run 24 hours and 7 days a week (24/7) to be hosted on EC2. The ingress Pipeline involves parking lot YouTube video processing, Image analytics, and predictions using AI Models and post-processing of the AI model prediction result, and the database is hosted on EC2 and runs 24 hours and 7 days a week (24/7).

The client-side API requests are on demand hence AWS LAMBDA service is engaged to respond to the requests on demand and avoid using EC2 kind of compute instance running unnecessarily.

4.5.2 Containerization - Docker

The solution is hosted on the AWS cloud. For the application to work in SaaS(Software as a Service) mode and be implemented across multiple parking lots, microservices architecture and containerization of solution components are used. This implementation approach allows scalability and performance when a large number of API requests hit the system and portability, manageability, and implementation of modifications and upgrades without disturbing the rest of the solution architecture. The ingress pipeline from the image fetcher stage to the post-processing stage can be containerized and can be deployed wherever needed to repeat the same process.

In the current implementation of MongoDB database is hosted in a docker and the database queries are exposed as APIs. There is scope for implementing ingress Pipeline (as shown in the architecture diagram above in figure 4.5) also in docker or just the computer-vision AI Models and associated logic inside of Docker.

4.5.3 API Gateway

Keeping in view that several APIs will be implemented to meet functional and analytical requirements in the future, the implementation engages the API gateway service of AWS. This ensures all API requests from various clients are serviced in real-time without impacting the performance and user experience.

4.6 Custom Trained YOLOv4 Parking Space Detector

In this section I am going to describe about an approach that failed and was not used in the development of this application. Initially I trained a pre-trained YOLOv4 object detection model on a custom-built dataset for parking space detection. In this method, the detector detects the parking spaces if they are empty, occupied (which is classified as a vehicle), occluded, or occupied but occluded. So, that is the reason it is referred to as a parking space detector. The details on how the custom dataset is built are in - [1]. But later due to the following reasons, I dropped the the idea of using this detector.

- High Inference time.
- The model needed additional training to achieve better accuracy.
- The model was a very heavy-weight model.
- A better approach was designed with the help of the vehicle detection technique.

I have used darknet library for custom training and inferencing of the parking space detector by following this article [6]

4.6.1 Pre-trained YOLOv5 Vehicle Detector

A pre-trained YOLOv5 model for vehicle detection was chosen from repository:[2] which was developed by a Canadian company "Ville de Montréal" as part of the traffic management system. This repository has three different versions of YOLOv5 models they are:

- S-512: YOLOv5 'Small' with input image size of 512x512 trained on the CGMU training set + MIO-TCD dataset.
- X-512: YOLOv5 'X-Large' with input image size of 512x512 trained on the CGMU training set + MIO-TCD dataset.
- X-704: YOLOv5 'X-Large' with input image size of 704x704 trained on the CGMU training set only.

The above models were trained on the CGMU dataset (The Centre de Gestion de la Mobilité Urbaine (CGMU) is the heart and brain of intelligent transport systems in Montreal) as well as an additional 11000+ images from the MIO-TCD dataset[5]. The CGMU dataset is not accessible to everyone. Different stages of the training pipeline are shown in figure 4.6. There were 13 different classes labeled in these datasets, they are:

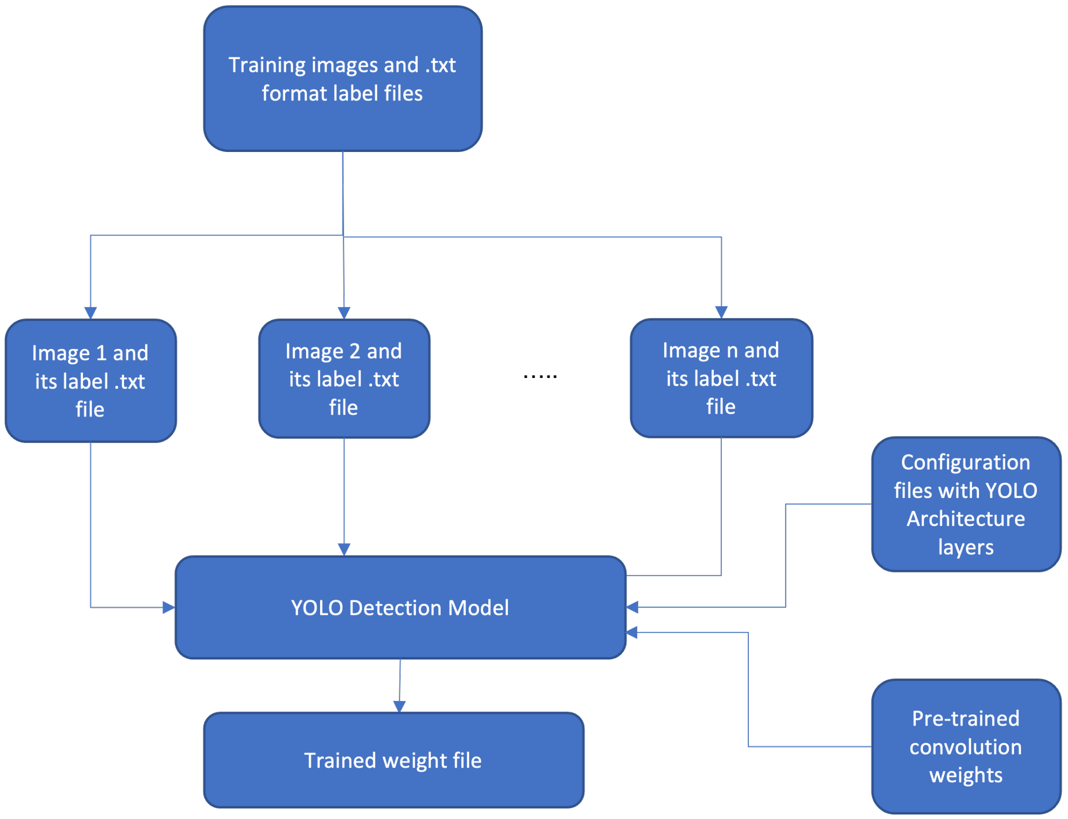
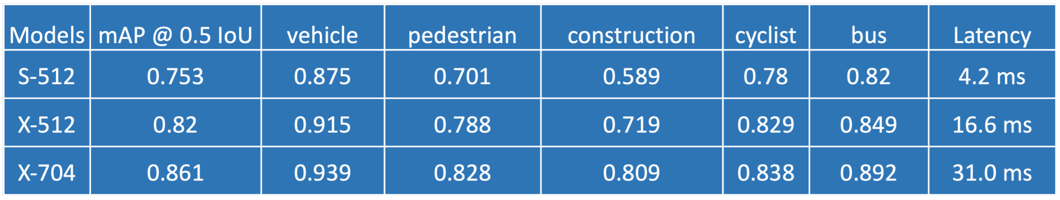

Training images and .txt format label files
Image 1 and its label .txt file
Image 2 and its label .txt file
.....
Image n and its label .txt file
Configuration files with YOLO Architecture layers
YOLO Detection Model
Trained weight file
Pre-trained convolution weights

Figure 4.6: Model training

- Pole
- Traffic Sign
- Vehicle
- Vegitation
- Median Strip
- Building
- Private
- Sidewalk

- Road
- Pedestrian
- Structure
- Void
- Construction

The figure 4.7 [2] shows the total mean Average Precision (mAP) for each model variant, as well as the mAP for each class, on the CGMU test set:



Models	mAP @ 0.5 IoU	vehicle	pedestrian	construction	cyclist	bus	Latency
S-512	0.753	0.875	0.701	0.589	0.78	0.82	4.2 ms
X-512	0.82	0.915	0.788	0.719	0.829	0.849	16.6 ms
X-704	0.861	0.939	0.828	0.809	0.838	0.892	31.0 ms

Figure 4.7: YOLOv5 Model Training results [2]

These three fully trained models S-512, X-512, and X-704 have their respective checkpoints that are available in the repository for inference. These specific checkpoints (weights) can be used to infer from new images or videos with the help of a detection script from the repository. Below figure 4.8 shows how the model inference happens.

4.6.2 Choice of the ML model

The YOLOv5 Vehicle detection model is chosen over the custom-trained YOLOV4 parking space detector. The reason behind the decision is that YOLOv5 is lighter and has less inference time when compared with YOLOv4. Also, the custom-trained parking space detector needs to be improved for better results.

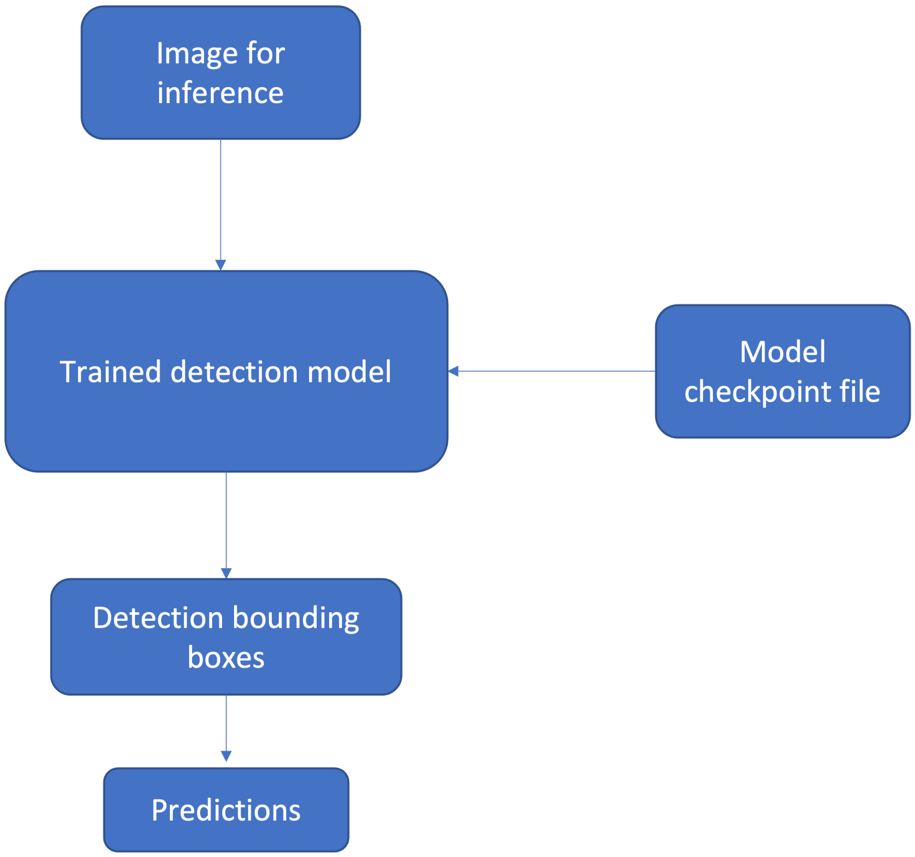


Figure 4.8: Model Inference

4.7 Functional Description

4.7.1 Ingress Pipeline

YouTube video Fetch and image extraction, the images extracted are normalized at the pre-processing module. The image is cropped to extract the visible section of the parking lot and converted to grayscale for uniformity across day and night visuals. The cropped image is fed to the Predictor module for running YOLO detection algorithm on the images to detect the vehicles in it and pass on the results to the postprocessing module for estimating the vehicle count, extracting metadata like parking lot ID, time of the day, parking lot name, number of empty spaces. In addition to that the prediction module also returns an image with detections which I call as "Annotated Image". The post-processing module performs two tasks in a sequential manner, first task is to insert the generated annotated image into AWS S3 storage bucket named "detectionlog" and the second task is to insert parking lot name, parking UUID, timestamp, number of vehicles, number of empty parking spaces and public URL of the annotated image that has been inserted onto S3 as part of the first task as a document into the MongoDB collection.

4.7.2 Database

The database chosen is MongoDB (Non-RDBMS) to handle unstructured data in the future. The database consists of a database named “plv_detection_data”, which consists of a collection named **"ResultsAndImageLinks"** in it. The data schema of the collection “ResultsAndImageLinks” is shown below.

Table 4.1:

Results and Image Links
Parking_lot_name : string
Parking_lot_UUID : string
number_of_vehicles : int
number_of_empty_parking_slots : int
image_s3_http_URL : string
Timestamp: ISODate

The ingress pipeline populates the database and the egress pipeline con-

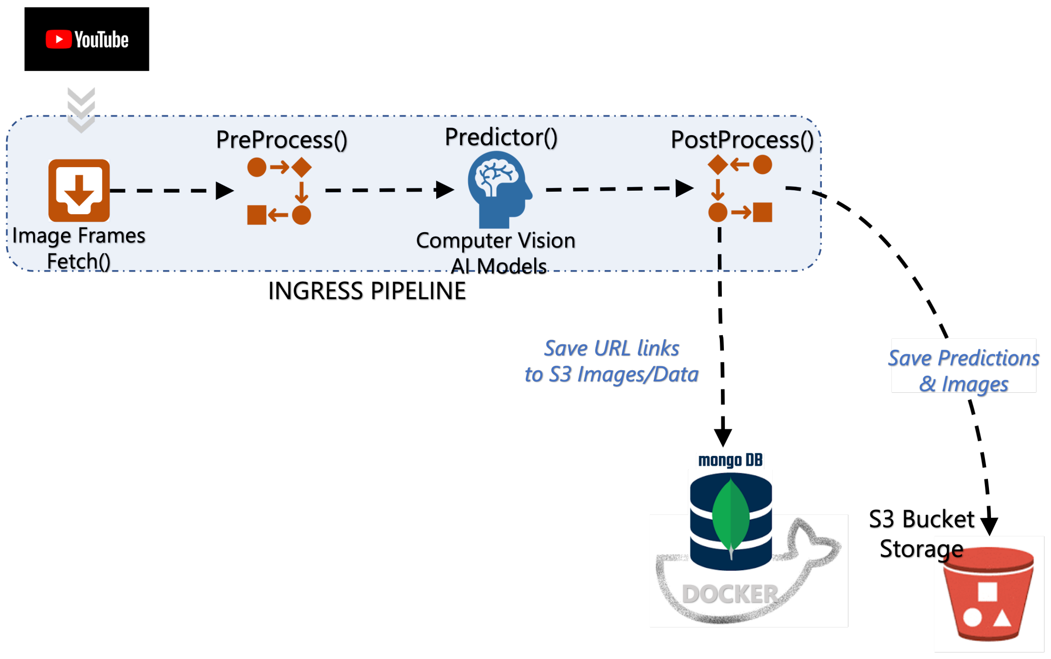


Figure 4.9: Ingress Pipeline

sumes the populated data. A package called "PyMongo" is used to operate the database. "PyMongo" helps in establishing a client connection to the database either running locally on localhost or any remote server. The post-processing script of the ingress pipeline builds a document by collecting all the required data and calls a simple method called "insert" to insert the document into the "ResultsAndImageLinks" collection. The Lambda function in the egress pipeline performs a query to fetch the latest available document from the "ResultsAndImageLinks" collection of the database.

4.7.3 Egress Pipeline

This pipeline consists of REST API configured with the AWS API Gateway service triggering an AWS Lambda function which fetches data from MongoDB running on the EC2 instance. The REST API interface is implemented and can be consumed by any type of client. As already mentioned this API interface provides two endpoints.

- GET /parking_lots.
- GET /parking_lots/{parking_lot_UUID}.

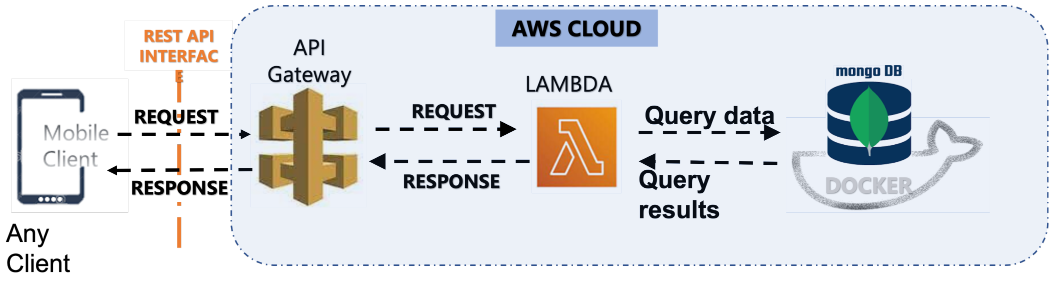


Figure 4.10: Egress Pipeline

The first method returns a list of parking lots with their names and corresponding UUIDs. Whereas the second method returns information on the parking lot occupancy status. This API /parking_lots/parking_lot_UUID triggers a Lambda function call which connects to the dockerized MongoDB on an EC2 instance and retrieves the latest inserted document from the collection “ResultsAndImageLinks”. Once the document is retrieved from the MongoDB it is sent to the client as part of the response body. This way the egress pipeline fetches data from MongoDB on the EC2 instance.

4.7.4 Detection Model Implementation

This is the part of predict module of the ingress pipeline, where the image is fetched from the live YouTube stream, pre-processed, and sent to the detection script (predict) as input. The detection script uses the trained model weights stored in the checkpoint files (S-512.pt/X-512.pt/X-704.pt) and returns an image with the detections annotated on it along with printing the number of detections for each class as shown in the figure 4.1. By default, the detection script uses S-512.pt weights for inference and can be changed by editing the configurations.

Upon observing the model inference results described in section 5.1.1.1, The mask-based approach has been adopted as it is providing better Mean

Average Precision (mAP) with an improvement of up to four percent when compared to the general approach. The current version of SmartPark follows the masking-based approach.

Chapter 5: Conclusion and Future Steps

5.1 Conclusion

The parking lot tracking system backend system has been developed. It provisions a REST API through which any client application can fetch occupancy information of nearby parking lots based on current GPS location individually. A pre-trained vehicle detection model has been chosen for inference, testing, and validation. A parking space detection model has been developed by training a YOLOv4 object detection model on a custom dataset and then tested and validated. Upon observation of results from both models, a decision was made to use the pre-trained vehicle detection model for the proof of concept (POC) and in the future, it is intended to fine-tune the vehicle detection model by custom training or use advanced computer-vision techniques like segmentation and tracking to re-develop the detection model.

The REST API using AWS API Gateway service has been developed, involving two GET methods on two different routes to retrieve the nearby available parking lots and occupancy information of individual parking lot respectively. These APIs have been tested and validated and the results are shown in section 4.3. The Image fetcher, pre-processing, prediction, and post-processing stages of the ingress pipeline and a MongoDB docker instance for storing the detection results have been deployed on an AWS EC2 instance. An AWS S3 storage bucket has been created for storing the resultant predicted images with annotated detections from the prediction stage. The serverless computing service, AWS Lambda function has been defined to fetch parking lot occupancy data from the MongoDB deployed on an EC2 instance on demand. It is also defined as a trigger to the REST API call.

The ingress pipeline and the egress pipeline are configured to run asynchronously to optimize the cloud resources' engagement and utilization. The solution is hosted on the AWS cloud and has been tested for functionality and performance. All the AWS resources were configured cost-effectively.

As of now the solution is built considering only one parking lot, In the future, it is intended to scale the solution for multiple parking lots at Oregon State

University. An insight into the future development of the detection model is detailed in chapter 7.

5.2 Future Steps

This chapter illustrates different strategies which can be followed in order to develop a better detection model.

5.2.1 Building a training pipeline with integration of MLFlow

A training pipeline has to be set up for regulating the training cycles in the future. This is essential because if the same model or a different model has to be trained on the same or different dataset then different models with different versions get generated. It becomes convoluted if these models and their versions are not maintained properly. So, ML Flow is a framework that helps in maintaining models with different versions which can be auto-logged onto a remote repository directly from the code base. ML Flow framework can be easily embedded into the code which makes its development comfortable.

5.2.2 Custom Training

A custom dataset has to be built with nearly 500 images of the parking lot from the camera that has already been set up. The dataset has to be labeled for vehicles in the image and the pre-trained vehicle detector has to be trained on this dataset. Then the model is expected to perform much better.

5.2.3 Parking Space Segmentation and Vehicle Tracking

A detection model has to be developed, which has to be trained for the detection of vehicles in the parking lots. The parking lot should be segmented, where each parking space is segmented into a unique color and acts as a trigger. Whenever a vehicle is parked in a parking space the trigger for that particular parking space activates and considers the parking lot to be occupied. In this solution the vehicle detection and parking space segmentation triggering are two different operations that execute in parallel. A company called "CVEDIA" has already done something similar to this and this strategy is inspired by the "Smart Parking" app they built.

Bibliography

[1] Nandam Sriranga Chaitanya. *Building a Custom Dataset*. URL: `https://oregonstate-innovationlab.atlassian.net/wiki/spaces/PLVD/pages/34635908/Building+a+Custom+Dataset`. (accessed: 11.29.2022).

[2] Ville de Montréal. *Urban-detection*. `https://github.com/VilledeMontreal/urban-detection`.

[3] Joe Petrie. *2020 Airport Business Project of the Year: Portland International Airport's Parking Guidance System*. URL: `https://www.aviationpros.com/airports/airport-revenue/parking-systems/article/21138052/2020-airport-business-project-of-the-year-portland-international-airports-parking-guidance-system`. (accessed: 12.12.2022).

[4] Plate.Vision. *Parking Management ANPR Software*. URL: `http://plate.vision/project/parking-management-anpr-software/`. (accessed: 12.12.2022).

[5] Igor Ryzhkov. *MIO-TCD dataset :Vehicle classification*. URL: `https://www.kaggle.com/datasets/yash88600/miotcd-dataset-50000-imagesclassification`. (accessed: 11.29.2022).

[6] Haobin Tan. *YOLOv4: Train on Custom Dataset*. URL: `https://haobin-tan.netlify.app/ai/computer-vision/object-detection/train-yolo-v4-custom-dataset/`. (accessed: 11.29.2022).

[7] Unabiz. *Smart Parking*. URL: `https://www.unabiz.com/use_case/smart-parking/`. (accessed: 12.12.2022).