Real Time Traffic Flow and Anomaly Detection

Nathan Conger, Matthew Dulcich, James Frech,
Eleftherios Lymperopoulos, Thilak Mohan

***Abstract*—Urban traffic management presents significant challenges due to the high volume and complexity of data generated by live traffic camera feeds. This project addresses these challenges by developing a real-time, cloud-based system that integrates AWS services such as IVS, Lambda, S3, and SageMaker. The system processes live video streams, automates video format conversion, and employs deep learning models for speed estimation and the Random Cut Forest algorithm for anomaly detection. The solution includes an interactive, customer-facing front-end interface built with Gradio, enabling end-users to visualize traffic patterns, anomalies, and actionable insights in real time. Through an event-driven, serverless architecture, the system achieves scalability, cost efficiency, and reliability, demonstrating the viability of cloud-based tools for responsive and accurate urban traffic analysis.**

I. Problem Statement

Managing urban traffic and ensuring road safety requires accurate, real-time detection of traffic conditions, speed, and accidents. However, current systems often fall short due to their limited ability to handle the scalability and high data throughput required for processing live video streams from multiple traffic cameras. These systems usually lack responsiveness and fail to provide timely insights, making it difficult to address traffic congestion, identify accidents, or manage emergencies effectively. Furthermore, the lack of user-friendly front-end interfaces limits accessibility and usability for end-users who need actionable insights. There is a pressing need for a scalable, cloud-based solution to process large volumes of live video data in real time while presenting results through an intuitive, customer-facing front end.

II. Related Work

Accurate and fast traffic speed estimation is an important task for transportation systems. A commonly used method for the estimation of vehicles speeds is deep learning, in particular, the YOLO series based in convolutional neural networks are widely applied. While a brief overview of the evolution of the YOLO series can be found in [1], the most recent version, YOLOv11 has shown high performance with increased inference time over previous versions including YOLOv8 and YOLOv10 [1]. The YOLOv11 model does this in a few ways. It replaces the C2f block from YOLOv8 with the C3k2 block which uses two small convolution layers with a kernel size of 2 to reduce computational cost. In addition, YOLOv11 also contains a spatial attention mechanism to improve focus on the more important parts of the image. This model is shown to outperform both YOLOv8 and YOLOv10 with a higher mean average precision across all classes when trained on traffic data including cars, motorcycles, trucks, buses, and bicycles while simultaneously improving inference time to 290 fps compared to 280 fps in YOLOv10 and 260 fps in YOLOv8 [1]. As such, the model shows high value for object detection and speed estimation in real-time situations.

For the task of anomaly detection, machine learning is often implemented as well. Some machine learning algorithms such as clustering or classification construct a profile of normal instances, then identify instances that do not conform to the normal profile as anomalies [2]. As this approach is not optimized to detect anomalies and instead reconstruct normal instances, the isolation forest (IF) algorithm was created and instead uses the concept of isolation to detect anomalies. The IF constructing trees with the purpose of isolating every point and classifying observations that have the smallest average paths from the root of the trees as anomalies [2]. Since then, improvements have been made on the IF including the random cut forest (RCF) which has the improved capability of detecting anomalies in real-time data [3]. The RCF model in particular is hosted on AWS, making it a suitable choice for detecting anomalies in our real-time traffic analysis. A detailed description of the RCF model is available in section III C.

III. Solution and Significance

As running real-time inference and anomaly detection on many data streams can be very computationally expensive, cloud computing offers a solution to create a scalable system allowing for real-time analysis of traffic flow and anomaly detection. While we only had time and resources (due to free tier constraints) to create a pipeline for one video stream at a time, this project has the potential to scale up to running real-time inference on all provided Maryland traffic cameras providing a system for the state to analyze traffic patterns and where any issues may occur in real-time (i.e. stopped traffic for accident, traffic congestion on certain roads, etc.). Our pipeline is hosted on Amazon Web Services (AWS) and includes ingesting live video streams through AWS Interactive Video Service (IVS), automatically converting the videos from ts to mp4 via a Lambda function, automated speed estimation using YOLOv11 and gathering traffic statistics (average speed, number of vehicles) on hosted EC2 instances, performing anomaly detection of traffic statistics using the AWS SageMaker hosted Random Cut Forest model, and providing a user interface to see annotated real-time traffic streams and explore historical traffic data statistics. A flow chart of our AWS architecture is provided in Fig. 1.

A. Processing Real-Time Videos

The real-time video processing pipeline begins with ingesting live traffic camera feeds. The traffic camera data we have used is from the Maryland Department of Transportation

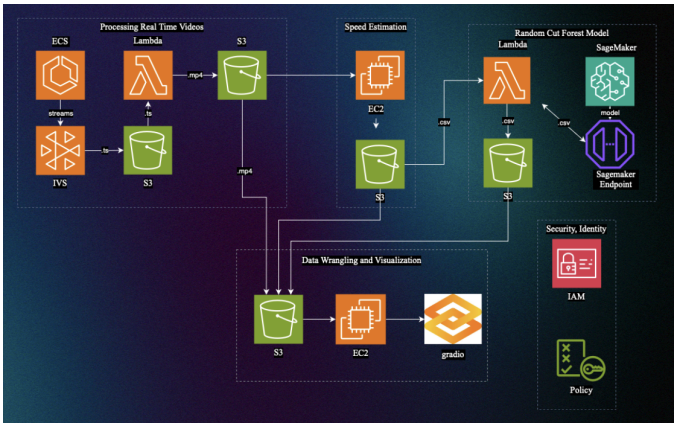

Fig. 1. Real time traffic flow and anomaly detection AWS Architecture

and can be viewed in [4] and a complete data table with their filenames can be found in [5]. To ingest the data, we firstly created an AWS Interactive Video Service (IVS) channel which was used to stream the video feed. To fetch that feed, we created a Docker container which uses the ffmpeg tool software, that gets a live feed using the RTMP protocol and streams it to the IVS channel. In order to be able to run this container indefinitely, we pushed it to an AWS Elastic Container Service (ECS) cluster and ran this container in the cluster. When the container is running, the live traffic camera feed gets streamed to the IVS channel, which was configured to store the video feed into an S3 bucket we created for this purpose. This video data is stored as 10-second long .ts files. Once a new .ts file is uploaded, a Lambda function is triggered to convert the file into .mp4 format using the ffmpeg library, ensuring compatibility with downstream tasks. The converted .mp4 files are prepared for further analysis, such as speed estimation and anomaly detection.

Implementing the Lambda function required additional optimizations to fit within AWS's lightweight resource constraints. For example, while a robust time conversion system using the pytz library was initially considered, the library exceeded Lambda's memory limits. Instead, a manual time conversion formula was implemented to handle timestamp adjustments from UTC to Eastern Time (ET). Additionally, setting up the base environment for the Lambda function was challenging, as it required obtaining a static build of ffmpeg and restructuring it to ensure feasibility within a Lambda-compatible folder structure. These measures ensured the function's performance while adhering to deployment size limitations, forming a critical foundation for the automated video processing workflow.

To enable the Lambda function to access files from Eleftherios' S3 bucket, an execution role was set up with permissions to read from the bucket. Additionally, a resource-based policy was created to allow the Lambda function to be invoked by an S3 event notification. This configuration ensured secure cross-account access and automated the triggering of the Lambda function upon the arrival of new .ts files, streamlining the workflow while maintaining proper security and access controls.

B. Speed Estimation

After the Maryland traffic camera data has been ingested through IVS and converted to mp4 via a lambda function, the data is ready to be processed for speed estimation. A pretrained YOLOv11n model from python's ultralytics package is implemented for object detection and speed estimation of vehicles that appear in the real-time traffic streams. The YOLOv11 model is chosen for its superior performance for the task compared to previous versions of YOLO including YOLOv8 and YOLOv10 [1]. This model is hosted on EC2 instances where it is automated to run on the most recent created mp4 files. EC2 was chosen as to run the model instead of other AWS services such as Lambda or SageMaker for a few reasons. The first reason is that the Lambda function, while having the functionality to automatically run inference on a mp4 video when it is uploaded to the S3 bucket, is too lightweight for the model. Lambda does not come with certain dependencies needed for running a YOLO model (ultralytics, pytorch, opencv) and would need to have them added as a custom layer uploaded in a zip file. However, Lambda has a maximum size of 50 mb for an uploaded zip file, and the dependencies needed for the model are too large. As such, Lambda is not a suitable platform for hosting our model. The second reason why we chose to use EC2 for our model is that it is cost-efficient. As hosting a model on SageMaker includes invoking endpoints to run the model which are not included in the free tier, EC2 provides a more economic alternative for deploying our speed estimation model.

The setup for running our speed estimation model in EC2 included the following. First, our setup was run on t2.micro instances using the basic Amazon Linux 2023 AMI. While the accuracy of speed estimation of our model may depend on available resources and would perform better running on a GPU than the CPU available in a t2.micro instance, AWS denied access for creating a GPU instance as we would have to contact them to allow our account to create one. Had we been able to test out a GPU instance, we would've used a g6.xlarge instance that provides an NVIDIA L4 Tensor Core GPU and is compatible with the Deep Learning OSS Nvidia Driver AMI GPU PyTorch 2.5 (Amazon Linux 2023) AMI, which comes with pytorch installed along with dependencies needed to interface with the GPU in the instance. This would speed up our processing time for the model inference greatly compared to the t2.micro instances.

While the t2.micro instance with basic Amazon Linux 2023 AMI is not specifically set up for deep learning, we were able to accomplish a setup allowing for real-time processing. A bash script was developed to install dependencies needed to automate running our model including pip to install python libraries, cronie to run cronjobs, and python libraries necessary for our model. After setup was complete, a python script was added to the instance for speed estimation which is then scheduled to run at certain times with a cronjob. The script takes the mp4 files from the previous minute, check the

model did not already run on them, then process the videos frame by frame to estimate the speed of traffic and obtain the count of vehicles in each video. Results are output to csv for anomaly detection processing along with annotated videos to S3. IAM permissions are needed for the EC2 instance in the input/output S3 bucket policies in addition to an IAM role for the EC2 instance with access provided to the buckets through an attached IAM policy.

Due to slow inference time on the t2.micro instance, it would take longer than a minute to process a minute's worth of videos. As such, it wasn't possible to keep up with real-time data using just one t2.micro instance. To solve this issue, five EC2 instances were created and each was scheduled to run a cronjob for speed estimation once every five minutes, each instance offset by one minute to the others. This way, every minute an EC2 instance starts the process to run the speed estimation on the previous minute of data in a round-robin fashion. This increased the speed the model could run and avoided overutilizing the CPU of just one instance. As the model is scheduled to run at selected intervals, once set up, the models run constantly without any user input. As such, the only process to run the models once the full setup is complete is to start the EC2 instances and keep them running.

As all EC2 instances serve the same purpose, an instance template was created. The instances all share the same t2.micro instance type, Amazon Linux AMI, security group, EC2 instance role (with associated IAM policy), and user data. The user data is given as the setup bash script previously mentioned to install dependencies for the model in the instance upon initialization. For full automation, the python script could be added to an S3 bucket and the user data could pull it during initialization as well as set up the cronjob. However, for our purposes, it was simple enough to add the file and set up the cronjobs to start at a different minute manually. In addition, future work could include automating this process fully and potentially use an autoscaling group to scale the number of instances according to the number of video streams in use.

C. Random Cut Forest Anomaly Detection Pipeline

For the anomaly detection component of the project, we implemented the built-in Random Cut Forest (RCF) algorithm using SageMaker notebooks, endpoints, Lambda, and S3, to build an efficient and scalable pipeline. The solution was designed to process data in real-time and detect anomalies effectively across multiple datasets.

The pipeline started with data preprocessing and model training. Using SageMaker Jupyter Notebooks, we preprocessed the raw data, trained the RCF model, and deployed it as an endpoint for real-time inference. SageMaker managed the scaling, security, and deployment of the model, ensuring smooth operation throughout. Once trained, the model was deployed via SageMaker endpoints, enabling live inference through the API.

To preprocess the data we had to take a few steps, most of them are typical processing techniques, but we want to highlight one of the steps that involves handling the date-time features, which have a cyclical nature. Time-related features like hours of the day, days of the week, and months of the year repeat periodically, and directly using raw values would imply linear relationships between them. To properly capture this periodicity, we used cyclic encoding, transforming these features into sine and cosine values. For example, the hour feature was transformed using the formulas

$$\sin(\text{hour}) = \sin\left(2\pi\frac{\text{hour}}{24}\right),$$

$$\cos(\text{hour}) = \cos\left(2\pi\frac{\text{hour}}{24}\right),$$

ensuring that 23:00 and 00:00 were represented as adjacent points on the unit circle. Similarly, the day of the week and month features were transformed into sine and cosine values using similar methods. This encoding helped the RCF model recognize that, for instance, 23:00 (11 PM) and 00:00 (midnight) are closer to each other than 06:00 (6 AM), which is crucial for accurately detecting anomalies in time-series data with repeating patterns.

The Random Cut Forest (RCF) model is a machine learning algorithm used for anomaly detection. It works by constructing a forest of trees, where each tree is built from random cuts of the data, dividing the data into partitions that represent distinct subsets of the feature space. Each tree is created by recursively partitioning the data, resulting in a collection of decision trees that help capture the underlying structure of the data. Outliers are more likely to be isolated in these trees, and their anomaly score is calculated based on how far they are from their neighbors. A higher score indicates that a point is an anomaly.

The algorithm also includes a pruning step during tree construction, where branches that no longer contribute to the model's predictive power are removed, improving performance and efficiency. This pruning helps reduce tree complexity, maintaining relevance and allowing the model to scale efficiently with large datasets. RCF's ability to handle high-dimensional data and provide real-time inference makes it particularly suitable for projects like ours, which require fast, accurate processing of large volumes of data.

While we initially considered the Prophet model, which can only process data from one camera at a time, we ultimately chose the RCF model because of its ability to handle input from multiple cameras simultaneously. To get proper predictions, the RCF model must be retrained now and then, and the endpoint must be redeployed. To do this, we just run the SageMaker notebook again and update the endpoint call in the Lambda function because all the data is saved in the S3 bucket.

For the real-time process, we implemented a Lambda function to preprocess CSV data uploaded to an S3 input bucket. The Lambda function then sent the preprocessed data to the SageMaker endpoint for anomaly detection. After inferring, the resulting data, along with the original data, was stored back in the output S3 in a single CSV file, allowing for the

automated process for the data. We had to make sure to add the `python 3.10 pandas` layer to the Lambda function as well as updating the memory it was allowed to use per run.

We encountered several challenges during the project. One major hurdle was correctly setting up the SageMaker environment in the Lambda function, including importing the necessary libraries, such as the SageMaker SDK and ensuring the model could be invoked through the API. This was challenging because Lambda functions are designed to be lightweight. To overcome this, we used the built-in `boto3` library instead of `sagemaker`. The default `boto3` import has a function, `invoke_endpoint`, for invoking the endpoint, while the `sagemaker` import offers a similar function, `predict`. Both functions yielded the same results, allowing us to maintain the lightweight nature of our Lambda function.

Another challenge arose during model training. Initially, we used the `ml.t4.medium` instance type, but it was too small to meet the computational requirements. Switching to an `ml.m4.xlarge` or `ml.m5.xlarge` instance provided the necessary resources for accurate training. While this change increased resource usage, it was essential to ensure the model performed optimally. Additionally, the Lambda function required specific IAM policies to interact with the S3 buckets and invoke the SageMaker endpoint. The policies we configured allowed Lambda to invoke the SageMaker endpoint, SageMaker to access the necessary S3 bucket for data retrieval, and both services to maintain secure access to the data.

AWS SageMaker provided the infrastructure for training and deploying the RCF model for real-time inference. By hosting the trained model as a SageMaker endpoint, we were able to create an API that efficiently processed real-time inference requests. This approach allowed us to seamlessly handle incoming data and deliver anomaly detection results without the need for retraining the model with each new data point. SageMaker's and Lambda's built-in models and features ensured the smooth operation of the pipeline, making it an ideal solution for our real-time anomaly detection needs.

D. Data Wrangling and Visualization

Structure of Visualization and Data Wrangling The pipeline for data wrangling involves pulling in annotated `.avi` videos for object and speed detection and `predictions.csv`, which contains input parameters and model outputs concatenated together, from separate S3 buckets into a data ingestion bucket. This data is read using an EC2 instance associated with an Elastic IP and security group permissions allowing inbound traffic from the internet gateway. The visualization is driven by gradio which is an open source library used to create web applications for machine learning. Data ingestion and data passing are managed through use of watchdog, a library used for local file systems monitoring and modification.

Video Handling and visualization

Since the watchdog library cannot remotely monitor the s3 bucket's file systems, a gradio timer is used to periodically look into the bucket and assess if any new videos are added. If one exists, it pulls the `.avi` video into the EC2 instance `avi_videos` folder. The avi video is then observed by watchdog and converted into a `.mp4` file and placed in the videos folder. Watchdog detects that a new video has been added and pushes the video up into a queue for the gradio video object to see and then present.

Data Handling and Visualization

Similar to the video handling, gradio uses a timer to periodically check if the `predictions.csv` file is modified in the S3 bucket by periodically saving the last modified time of the current comparison with the last saved modified time of the last comparison. If a change is detected, the predictions file is pulled into the EC2 instance and the data appended to the `master.csv` file. Upon detecting the change, gradio updates the scatterplot reflecting the new data. Dropdown lists are utilized to modify the axis types of the scatterplot so users can compare trends through variables.

IV. Evaluation Results

Our project results in a workflow that processes raw live feeds from traffic cameras, estimates vehicle speeds, and detects anomalies in traffic flow. The YOLO model demonstrates strong performance during the daytime (Fig. 2), providing realistic speed estimates out of the box without requiring additional training.

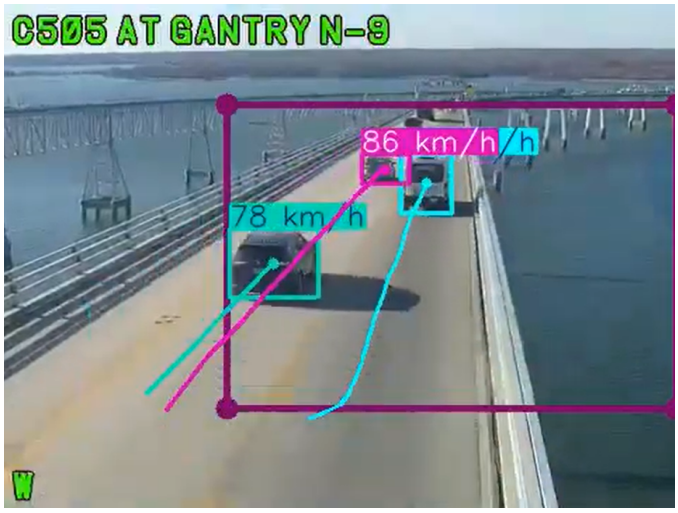

Fig. 2. Real-time speed estimation for one Maryland traffic camera near Annapolis.

In order to ensure that running the model would not have issues with overworking our t2.micro EC2 instances, we checked their CPU utilization through EC2 monitoring provided by AWS. Results show that when running the five EC2 instances together, the CPU utilization maxed out at ∼50% on each instance (Fig. 3). This shows the round-robin scheduling with five EC2 instances is efficient without overworking any of the instances and provides a working solution to our issue of only one t2.micro instance not having the processing power to keep up with the workload.

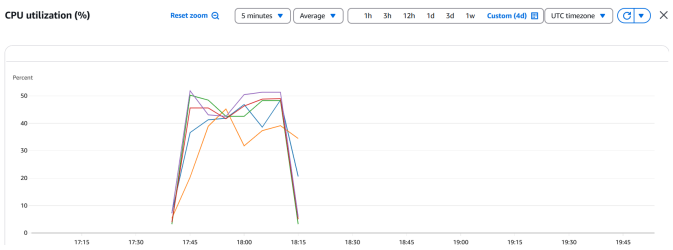

Fig. 3. CPU utilization of our EC2 instances running real-time speed estimation.

In addition to speed estimation, our system relies on AWS Lambda for automated video preprocessing, including converting .ts files to .mp4. To evaluate its performance, we focused on December 3rd, the day of our presentation, which had the highest utilization of the pipeline (Fig. 4 and Fig. 5). This day provides insights into how the system performs under peak workloads.

December 3rd, 2024 (date of our presentation) showed the highest utilization of our pipeline. Fig. 4 highlights the invocation count for the Lambda function, reflecting consistent workloads with peaks during periods of high video ingestion.

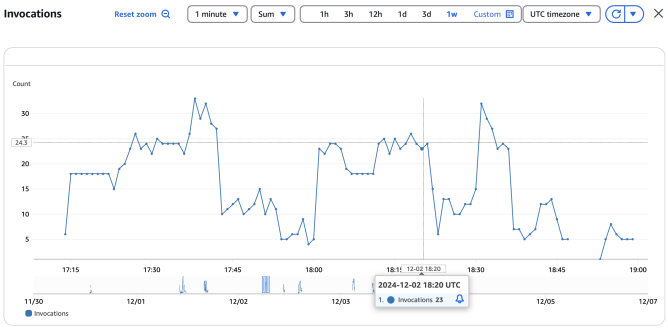

Fig. 4. Invocation count of the Lambda function on December 3rd, during peak utilization.

Fig. 5 displays the execution duration of the Lambda function. Some invocations approached the timeout threshold due to large .ts file sizes, impacting real-time responsiveness slightly.

Fig. 5. Execution duration of the Lambda function on December 3rd, showing the impact of high workload.

The high utilization metrics on this day showcases the robustness of our architecture in handling peak workloads. However, they also highlight opportunities for further optimization to reduce processing times and improve real-time responsiveness, ensuring the system scales effectively without compromising performance.

V. Potential Improvements in the Pipeline

- **Architecture:** Our current architecture was intentionally designed to optimize costs by leveraging AWS's free tier limits. By distributing resources such as S3 buckets and EC2 instances across separate accounts, we avoided exceeding free tier restrictions. This approach ensured that the large number of requests to S3 buckets and the computational demands of the speed estimation pipeline remained within the free tier limits. While effective in reducing costs, this distributed setup introduced some challenges in coordination and management.
  To improve the architecture, we could focus on enhancing the organization and automation of services. For example, implementing standardized naming conventions or tagging across S3 buckets and EC2 instances would simplify tracking and debugging while retaining the cost benefits of distributed resources. Additionally, using infrastructure-as-code tools like AWS CloudFormation or Terraform could automate deployment and configuration across accounts, reducing manual effort and ensuring consistency. These improvements would streamline the system without compromising its cost-effectiveness.

- **Speed Estimation Accuracy:** While the speed estimation model generates realistic results, we do not have a way to determine the accuracy on the real-time Maryland traffic cameras as they do not have any historical data associated with them that contain associated vehicle speeds for us to verify the model. If we were able to obtain this data, we could more accurately assess our model, and in addition, fine tune it to work perform better on specific traffic cameras.
  In addition to not being able to assess the performance of the speed estimation, we found the model struggled to estimate speeds at all during the night. We found that when applying the model at night, all speeds were estimated to be 0 mph (Fig. 6). In addition to not predicting speeds, the model had a harder time detecting vehicles at night due to glare from headlights and darker silhouettes. However, some vehicles were still detected showing that the object detection does a decent job at night, just not as good during the day time. Future work could focus on finding ways to fix this issue, perhaps by training the model on traffic datasets at night, or perhaps finding appropriate preprocessing techniques on the videos to allow for better inference.

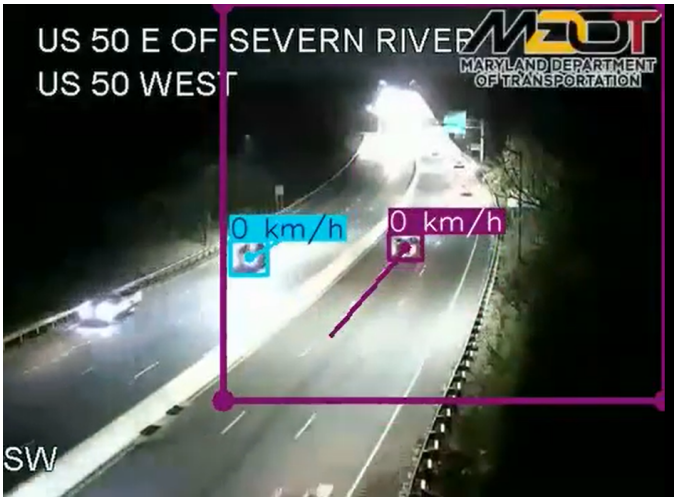

Fig. 6. Speed estimation at night using pretrained YOLOv11.

VI. Members Contribution

This section describes the members contributions. We all used IAM Roles and Policies to connect our accounts.

i. **Nathan Conger:** Implemented data visualization and management within EC2 instance using gradio and watchdog libraries.
ii. **Matthew Dulcich:** Implemented a Sagemaker Jupyter notebook to develop the Random Cut Forest Model. Deployed the Sagemaker endpoint so we could use the model for inference. Implemented a lambda function to preprocess the CSV data from James bucket, send it to the inference endpoint, and save the results along with the original data as a CSV file in my S3 bucket.
iii. **James Frech:** Automated speed estimation and generated statistics of real-time Maryland traffic cameras using YOLOv11 hosted on EC2 instances.
iv. **Eleftherios Lymperopoulos:** Created ECS cluster and the Docker container which is run on the cluster. Created the IVS channel where the live video feed is being streamed. Also configured the IVS channel to store the stream into an S3 bucket.
v. **Thilak Mohan:** Designed and implemented a Lambda function to process .ts video files triggered by S3 bucket events. The function converts .ts files to .mp4 using ffmpeg and stores them in a designated output folder in S3. Addressed challenges in file naming conventions, such as timestamp conversion from UTC to ET, and optimized deployment size by avoiding libraries like pytz. Integrated resource-based policies and execution roles to enable secure cross-account S3 operations. Additionally, assisted in setting up Matthew's Lambda function to trigger when James' S3 bucket received a new file, mirroring the setup for Eleftherios' S3 bucket, where Thilak's Lambda function would process uploaded files.

VII. Implementation Tools

AWS services and other tools used for this project include:

i. **AWS ECS:** Amazon Elastic Container Service for deploying and managing containers.
ii. **AWS IVS:** Interactive Video Service used for low-latency live streaming.
iii. **AWS S3:** Amazon Simple Storage Service for storing and retrieving large datasets.
iv. **AWS Lambda:** Serverless compute service for running code in response to triggers.
v. **AWS EC2:** Elastic Compute Cloud for virtual server hosting.
vi. **AWS SageMaker Notebooks:** Integrated Jupyter notebooks for developing machine learning models.
vii. **AWS SageMaker Endpoints:** Deployed machine learning models for real-time inference.
viii. **IAM Roles:** Used for defining permissions for services and users.
ix. **IAM Policies:** Policies to manage access control across AWS resources.
x. **Gradio:** Open-source python library for creating ML model interfaces.

Code Availability

To ensure reproducibility, all the source code have been made available in a publicly accessible GitHub repository [6].

References

[1] M. A. R. Alif, "Yolov11 for vehicle detection: Advancements, performance, and applications in intelligent transportation systems," 2024. [Online]. Available: https://arxiv.org/abs/2410.22898
[2] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *2008 Eighth IEEE International Conference on Data Mining*, 2008, pp. 413–422.
[3] S. Yeom and J.-H. Jung, "Weighted isolation and random cut forest algorithms for anomaly detection," 2024. [Online]. Available: https://arxiv.org/abs/2202.01891
[4] Maryland Department of Transportation, "Maryland traffic cameras," https://chart.maryland.gov/TrafficCameras/GetTrafficCameras, 2024, accessed: 2024-12-06.
[5] MD Department of Transportation, "Maryland traffic cameras dataset," https://data.imap.maryland.gov/datasets/maryland::maryland-traffic-cameras-traffic-cameras/explore?showTable=true', 2024, accessed: 2024-12-06.
[6] N. Conger, M. Dulcich, J. Frech, E. Lymperopoulos, and T. Mohan, "Aws real-time traffic flow and anomaly detection," 2024, accessed: 2024-12-06. [Online]. Available: https://github.com/MatthewDulcich/AWS-Real-Time-Traffic-Flow-and-Anomaly-Detection