Vehicle and Pedestrian Tracking with YOLOv8-Seg and Labellerr: An End-to-End Pipeline

Om Singh

September 23, 2025

Abstract

This report details the development of a complete, end-to-end computer vision pipeline for instance segmentation and multi-object tracking. The primary objective was to identify and track vehicles and pedestrians in challenging, real-world video streams. A custom dataset of difficult images was collected and annotated with polygon masks using the Labellerr platform. A YOLOv8-seg model was successfully fine-tuned on this data, achieving a respectable performance on the validation set. The trained model was then integrated with the ByteTrack algorithm to create a robust object tracker. The final system was deployed as an interactive web application that accepts a user-uploaded video and exports tracking results in JSON format. This project successfully demonstrates the entire machine learning lifecycle, from data creation and model training to final application deployment and quality assurance.

1 Data Collection and Preparation

Figure 1: BDD Image

As specified in the assignment, "pick a difficult dataset where publicly available models fail", we selected the Unannotated Berkeley DeepDrive (BDD100K) dataset, which constituted approximately 60–70% of our train–validation–test split. The remaining 30–40% comprised high-quality, complex images sourced from platforms such as Unsplash and Pexels. The dataset was partitioned into training, validation, and test subsets, ensuring that each subset maintained a consistent distribution of images from all sources.

Figure 2: Unplash Image

2 Data Annotation using Labellerr

Figure 3: Vehicle and Pedestrian Annotation

This was the first experience with the tool. The Labellerr interface was found to be intuitive and user-friendly. The training and validation datasets were imported, and separate projects were created. Polygon masking was primarily used for annotation, supported by AI-assisted segmentation, which handled most of the work. The main challenge was the precise labeling of intricate pedestrian and vehicle boundaries, which had to be refined manually using the clip tool. An initial mistake was made by repeatedly annotating the reference car frame boundary, which was later identified as static and unnecessary. After the annotations were completed, the data was converted into the YOLO-compatible JSON format containing polygon mask coordinates for each image, and subsequently exported for

YOLO model training.

Figure 4: Vehicle and Pedestrian Annotation

3 Model Training and Fine-Tuning

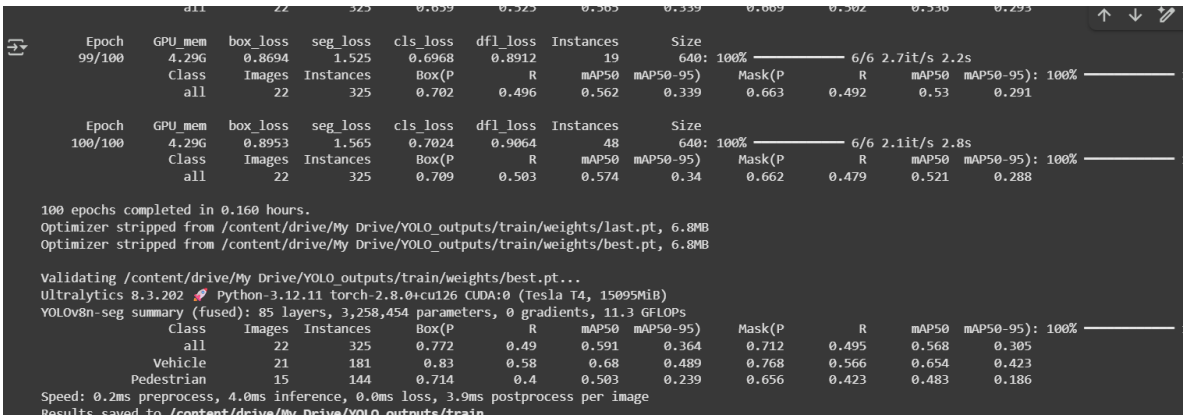

Figure 5: Running 100 Epochs

The primary objective of the model training phase was to fine-tune a state-of-the-art, pre-trained object segmentation model on the custom-annotated dataset of vehicles and pedestrians. This process of transfer learning is crucial for adapting a general-purpose model to a specialized task efficiently. To meet the high computational demands of this task, the project utilized a Google Colab notebook environment, which provided access to a free NVIDIA T4 GPU, as recommended by the assignment guidelines.

The base model selected for this task was YOLOv8n-seg from the powerful Ultralytics library, chosen for its excellent balance of speed and accuracy, making it an ideal candidate for a real-time tracking application. The fine-tuning was configured to run for 100 epochs, a duration specified to ensure the model had sufficient time to learn the nuances of the custom dataset. The training script was pointed to the dataset's location and class definitions via the data.yaml file. The training was initiated by loading the pre-trained yolov8n-seg.pt weights and executing the .train() method

within the Colab environment. Throughout the process, the model's learning progress was actively monitored by observing the metrics printed to the console after each epoch. Key indicators of successful training were the consistent decrease in loss values (including box loss and segmentation loss) and, most importantly, the steady increase in the mean Average Precision (mAP) score on the unseen validation set, confirming that the model was generalizing rather than just memorizing the training data. The training completed successfully, yielding a final model that demonstrated significant improvement over its baseline performance.

The most critical outcome of this phase was the generation of thebest.pt file, which contains the weights of the model checkpoint that achieved the highest mAP score on the validation data. This final, custom-trained model artifact served as the core component for all subsequent evaluation and deployment tasks in the project pipeline.

4 Model Evaluation and Performance Metrics

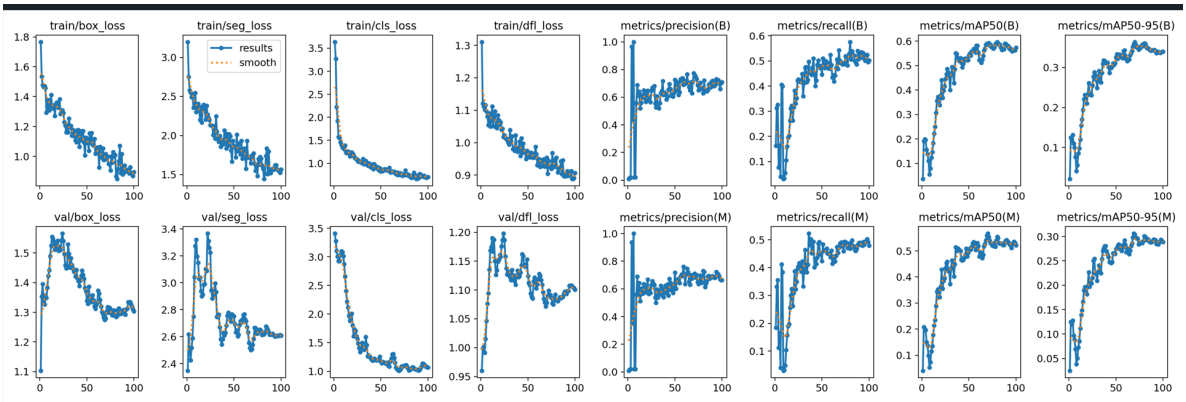

Figure 6: Performance Charts

The fine-tuned YOLOv8-seg model demonstrated a strong ability to learn from the custom-annotated dataset, achieving successful overall performance on the validation set. The primary evaluation metric, mean Average Precision for masks at an Intersection-over-Union (IoU) of 0.5 (**Mask mAP@0.5**), reached **0.568** for all classes combined.

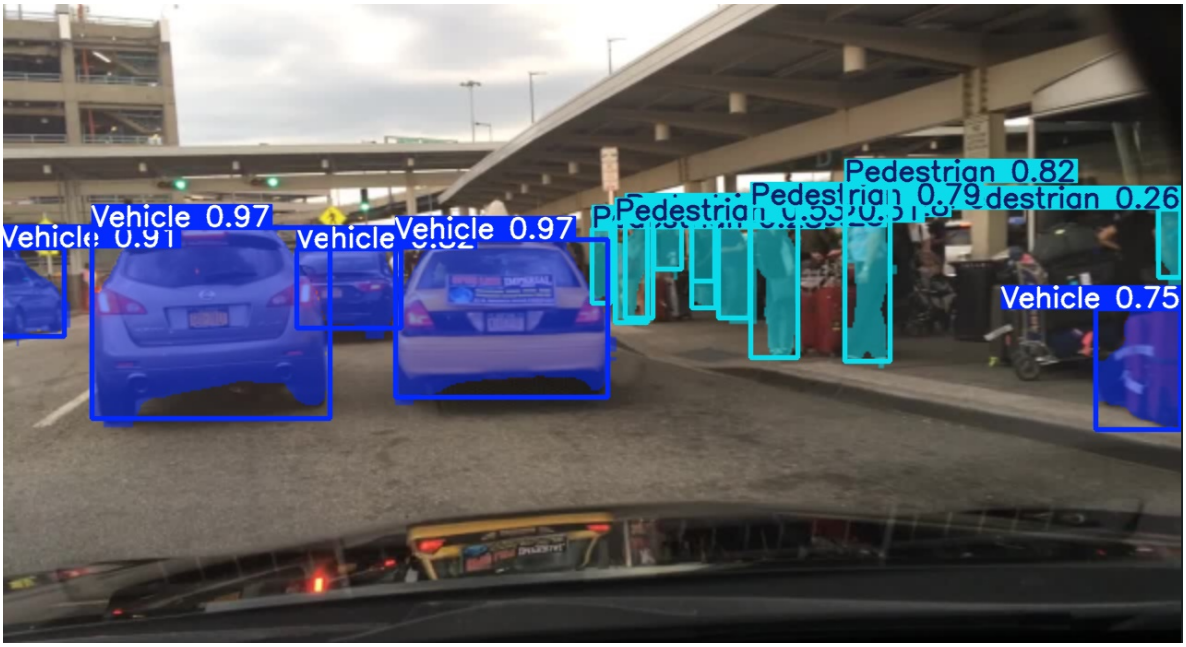

Figure 7: Model Result

A detailed breakdown of the results, as shown in the Precision–Recall Curve for segmentation, reveals a clear performance difference between the two classes. The model was significantly more adept at segmenting *vehicles*, achieving a mAP@0.5 of **0.654**, while the more challenging *pedestrian* class scored only **0.483**.

This performance gap is also reflected in the F1–Confidence Curve, where the *Vehicle* curve consistently remains above the *Pedestrian* curve. The Normalized Confusion Matrix provides further insight, showing that the model correctly predicted pedestrians **41%** of the time, but also misidentified pedestrians as background **58%** of the time. This highlights the difficulty of segmenting smaller, more occluded objects. Conversely, the model was much more confident with vehicles, correctly predicting them **58%** of the time with minimal confusion.

The dataset analysis suggests a potential reason for this discrepancy, showing a higher instance count for vehicles (**567**) compared to pedestrians (**304**), which may have provided the model with more robust training examples for the vehicle class.

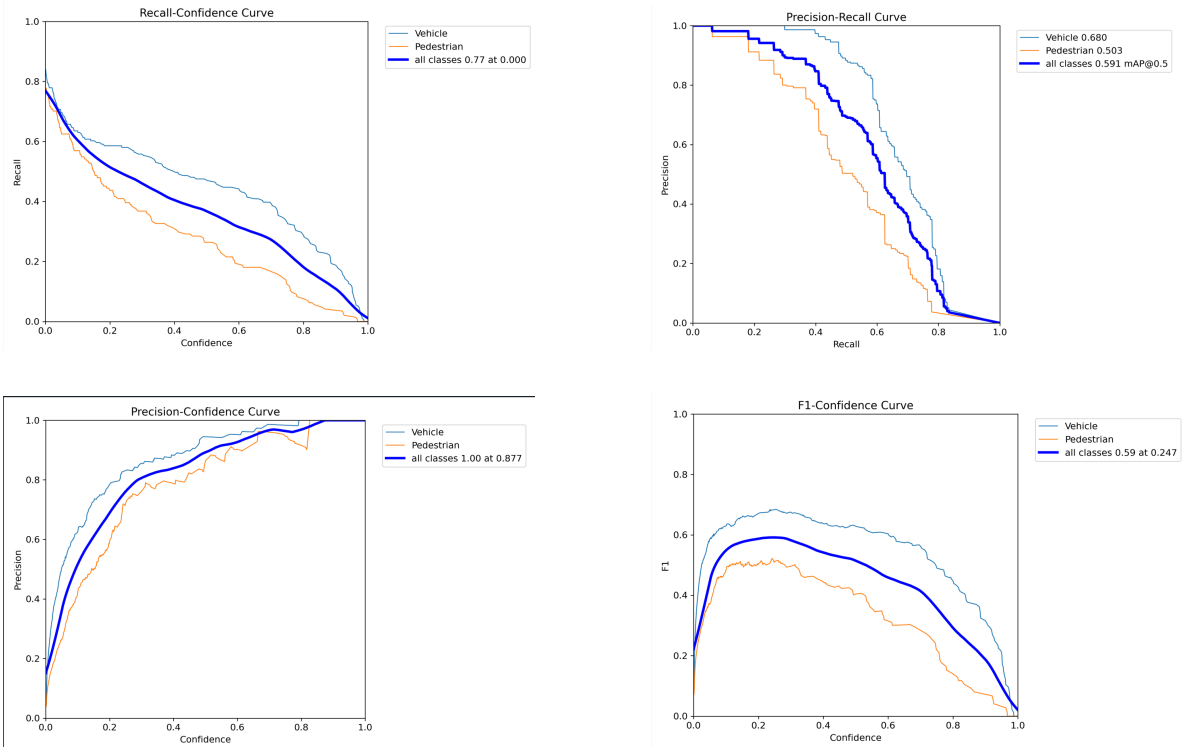

Figure 8: Performance Charts

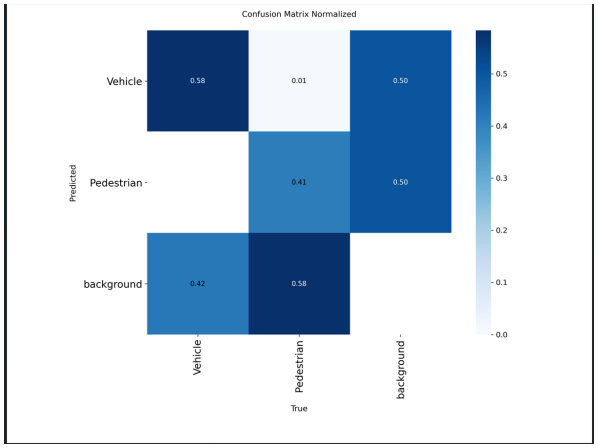

Figure 9: Confusion Matrix

Class mAP50-95	Images	Instances	Box(P)	R	mAP50	mAP50-95	Mask(P)	R	mAP50
all 0.305	22	325	0.772	0.49	0.591	0.364	0.712	0.495	0.568
Vehicle 0.423	21	181	0.83	0.58	0.68	0.489	0.768	0.566	0.654
Pedestrian 0.186	15	144	0.714	0.40	0.503	0.239	0.656	0.423	0.483

Table 1: YOLO Model Performance Metrics

5 Quality Assurance: The Labellerr Review Loop

Following the successful training and validation of the model, the next crucial step was to perform a quality assurance (QA) review, simulating a real-world MLOps workflow as required by the assignment. The objective of this phase was to use the trained model to pre-annotate the unseen test set and then verify these predictions within the Labellerr platform. To accomplish this, a new project was created on Labellerr, and the 50 original test images were uploaded.

Figure 10: Predicition of boundaries of objects

A Python script was then developed using the Labellerr SDK to programmatically upload the model's predictions. This script read the YOLO-format `.txt` files generated during inference, converted the normalized polygon coordinates for each detected object into the required `LABELLERR_JSON` format, and then called the `upload_preannotation_by_project_id` SDK function.

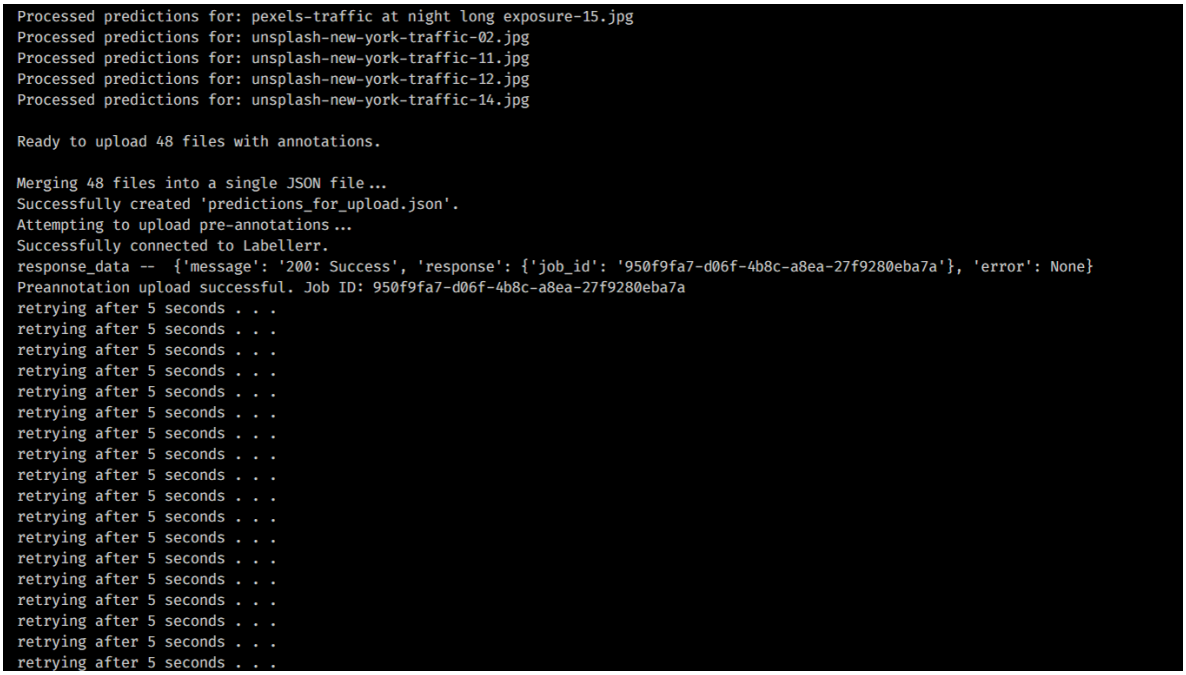

Figure 11: Successful upload of pre-annotations to server

Upon successful execution, a final verification was performed by opening the test project in the Labellerr UI, which confirmed that the previously unlabelled images were now populated with the model's predictions, ready for a human reviewer to approve or correct. This successfully closed the loop between model training and data annotation, demonstrating a key component of an iterative machine learning pipeline.

6 Application: YOLO-Seg and ByteTrack Integration

The core of the final application is the integration of the custom-trained **YOLOv8-seg** model with the **ByteTrack** algorithm to achieve robust multi-object tracking in video streams. The YOLOv8 model served as the high-performance detector, analyzing each video frame to identify instances of vehicles and pedestrians. These detections, including bounding boxes, class labels, and confidence scores, were then passed to the ByteTrack algorithm.

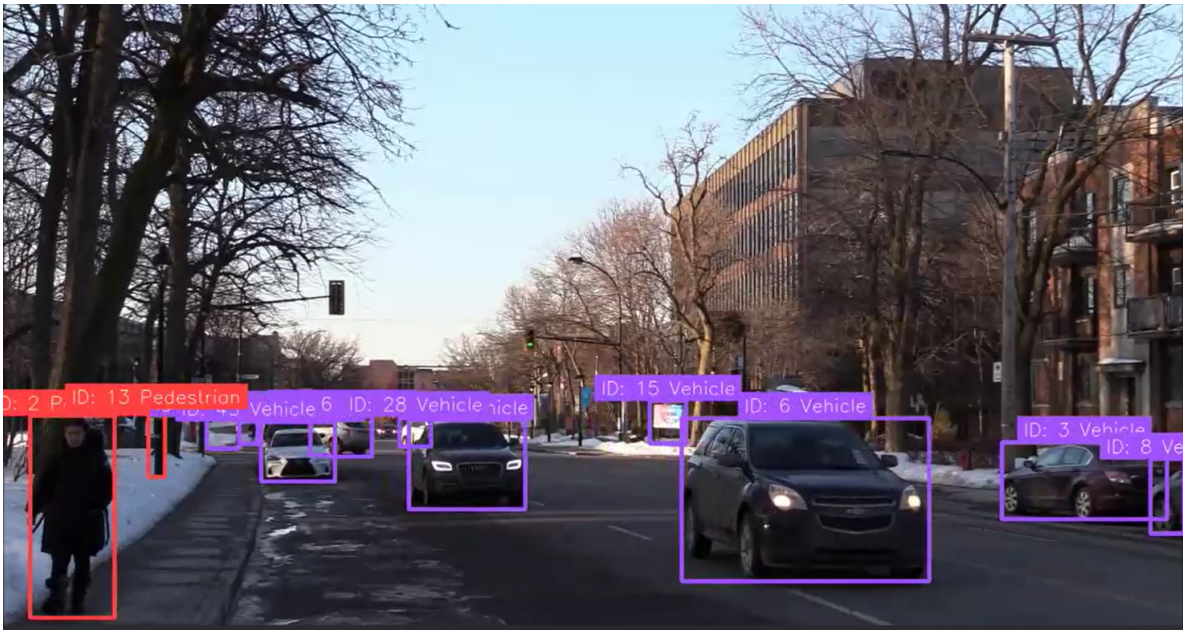

Figure 12: Segmentation and Live Tracking

ByteTrack's key advantage is its ability to handle occlusions by associating both high-confidence and low-confidence detections, which prevents the tracker from losing objects that are temporarily obscured. The integration was streamlined by using the `supervision` library, which provides a robust implementation of ByteTrack that interfaces seamlessly with the `ultralytics` YOLO library.

The final script processes a video frame-by-frame, passing detections to the tracker, which then assigns a persistent `track_id` to each unique object. This process results in two primary outputs:

1. A processed video file with annotated bounding boxes and tracking IDs.
2. A `results.json` file containing detailed, structured data for each tracked object in every frame.

7 Live Demo Application

To provide a user-friendly interface for the tracking pipeline, a web application was developed and deployed, fulfilling the "Demo Live link" deliverable. Initially built with **Gradio**, the application was later migrated to **HuggingFace Spaces** to resolve performance and deployment challenges.

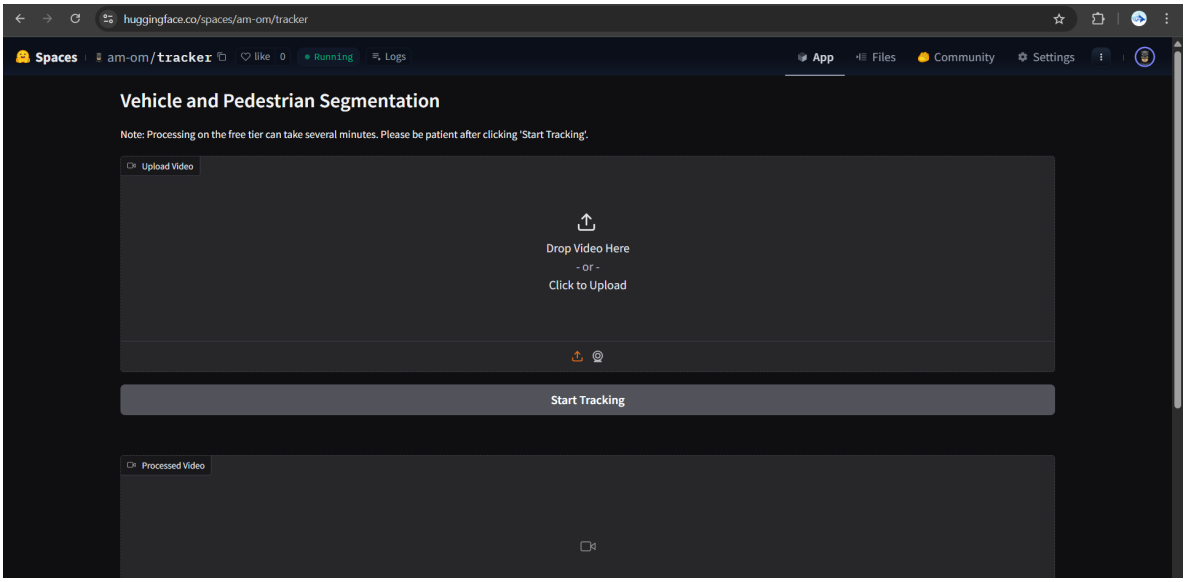

Figure 13: Live Demo

A significant architectural decision was made to decouple the application into a lightweight **frontend** and a powerful **backend API** to ensure a fast and responsive user experience. The frontend, built with Gradio, provides a simple interface for video uploads. Upon submission, it sends the video to a separate backend API built with **FastAPI**. The backend service, which hosts the `best.pt` model, performs all the computationally intensive video processing and returns the final result.

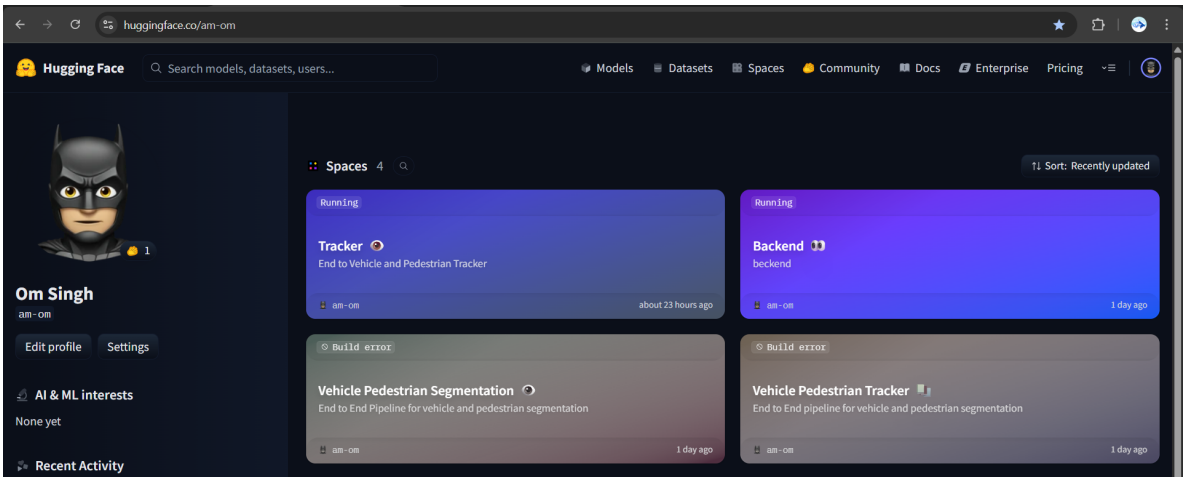

Figure 14: Deployment

Both services were deployed independently on **Hugging Face Spaces**, with the backend leveraging an upgraded CPU hardware tier for faster inference. This decoupled architecture ensures the user-facing application loads instantly, providing a professional and seamless demonstration of the project's capabilities.

The key challenge and learning experience was transitioning from local host deployment to server-level deployment. I encountered multiple dependency and build errors, but after several iterations and adjustments, I was able to successfully resolve them.

This project successfully demonstrated the creation of an end-to-end computer vision pipeline, covering the entire machine learning lifecycle from data collection to a deployed application. A custom **YOLOv8-seg** model was fine-tuned on a challenging, hand-annotated dataset, achieving a respectable baseline performance. This model was then integrated with **ByteTrack** to build a functional multi-object tracker, which was wrapped in a responsive, two-part web application. The process highlighted

proficiency in data management, model training, performance evaluation, and modern deployment practices.

8 Future Work

Several avenues could be explored to improve upon this baseline:

- **Dataset Expansion:** Enhance model accuracy by significantly expanding the training dataset with more diverse and challenging scenarios.
- **Model Improvements:** Experiment with larger YOLOv8 backbones (e.g., `yolov8m-seg.pt`) and perform systematic hyperparameter tuning to achieve higher performance.
- **Scalability:** Re-architect the pipeline for large-scale datasets (millions of images) by leveraging cloud storage (e.g., AWS S3), a data versioning tool (e.g., DVC), and distributed training frameworks (e.g., PyTorch DistributedDataParallel) on multi-GPU cloud instances.
- **UI and Deployment :** The application's UI and deployment can be further improved, as the container build process encountered multiple dependency errors. Additionally, increasing backend compute resources would enable faster processing and more efficient results