DEEPSTREAM 3.0 ANALYTICS APPLICATIONS

SWE-SWDOCDPSTR-001-APNT | November 19, 2018
Advance Information | Subject to Change

Application Note

DOCUMENT CHANGE HISTORY

Date	Author	Revision History
Nov. 19, 2018	Bhushan Rupde, Jonathan Sachs	Initial release.

TABLE OF CONTENTS

1.0 Architecture 7
1.1 Edge To Cloud Integration 8
1.2 Streaming Pipeline 8
1.3 Batch Pipeline 9
2.0 Docker Deployment 11
2.1 Clone the Smart Parking Application Repository 12
2.2 Deploying the Analytics Server 12
2.3 Perception Server Deployment 16
3.0 Perception Layer: 360-D Smart Parking Application 19
3.1 The Gst-uridecodebin Plugin 20
3.2 The Gst-nvdewarper Plugin 20
3.2.1 Core Dewarping 20
3.2.2 Preprocessing 21
3.3 The Gst-nvstreammux Plugin 22
3.4 The Gst-nvinfer Plugin 22
3.5 The Gst-nvbboxfilter Plugin 22
3.6 The Gst-nvtracker Plugin 23
3.7 The Gst-nvaislemetadata Plugin 23
3.8 The Gst-nvspot Plugin 23
3.9 The Gst-nvmsgconv Plugin 24
3.10 The Gst-nvmsgbroker Plugin 24
4.0 Calibration and Regions of Interest 25
4.1 Types of Calibration 25
4.2 Design 27
4.2.1 Spot Calibration 27
4.2.2 Aisle Calibration 28
4.3 Detailed Methodology 29
4.3.1 Annotating maps 29
4.3.2 Annotating images 30
4.3.2.1 Step 1: Capture Image Snapshots from Cameras 30
4.3.2.2 Step 2: Blueprint/CAD Image 30
4.3.2.3 Step 3: Georeferencing 31
4.3.3 Polygon Drawing for Aisle Calibration 32
4.3.4 Polygon Drawing for Spot Calibration 36
4.3.5 Transferring CSV to the DeepStream Server 37
5.0 Metadata 38
5.1 About Metadata Elements 38
5.2 Sensor 39
5.3 Video Path 39
5.4 Analytics Module 39
5.5 Place 40
5.6 Object 41
5.7 Event 42

5.8 Units 43
6.0 Multi-Camera Tracking 44
6.1 System Overview 46
6.1.1 Poller 47
6.1.2 Filters 47
6.1.3 Clustering 47
6.1.4 Matching 49
6.2 Configuration 50
6.3 Running the tracker 51
7.0 Stateful Stream Processing 53
7.1 Installation 54
8.0 Database Schema 56
9.0 API 58
9.1 Getting Started 58
9.1.1 Environment Variables 58
9.1.2 Installation 59
9.2 Endpoints exposed 59
9.2.1 Configuration-Related Endpoints 59
9.2.2 Garage-Related Endpoints 61
9.3 Websocket exposed 62
9.4 Configuration File 63
10.0 User Interface 65
10.1 User Interface Components 66
10.2 Getting Started 71
10.2.1 Dependencies 71
10.2.2 Environment Variables 71
10.2.3 Installation 71
10.2.4 Deployment 72
10.2.5 Configuration 72
10.2.5.1 Home Page 72
10.2.5.2 Garage Page 73
10.2.5.3 Live versus Playback 74
11.0 Dashboard 76
12.0 Appendix A: 360-D Smart Parking Application Configuration Details 77
12.1 About Configuration Groups 77
12.2 Application Group 78
12.3 Tiled-Display Group 79
12.4 Spot Group 79
12.5 Aisle Group 80
12.6 Message-Broker Group 80
12.7 Dewarper Group 81
12.8 Source Groups 81
12.9 Streammux Group 83
12.10 Primary Group 84
12.11 Tracker Group 86
12.12 OSD Group 86
12.13 Sink Group 88

12.14 Tests Group 89

LIST OF FIGURES

Figure 1. DeepStream 3.0 architecture 7
Figure 2. Use of protocols 8
Figure 3. Streaming data pipeline... 9
Figure 4. Batch data pipeline 10
Figure 5. Perception server deployment 11
Figure 6. The page at the Kibana URL 15
Figure 7. Create Elasticsearch anomaly-Index 15
Figure 8. Test by opening http://<IP_ADDRESS>... 16
Figure 9. The 360-D application's architecture 19
Figure 10. Typical output frame of decoder (1472×1384) 20
Figure 11. Sample output of the dewarper algorithm....................................... 21
Figure 12. Output of scaled surfaces 22
Figure 13. Distorted image from a 360° camera 26
Figure 14. Aisle A0: Dewarped image with RoI (yellow box) and car (blue box) 26
Figure 15. Aisle A1: Dewarped image with RoI 27
Figure 16. Spot S0: Dewarped image with detection lines (yellow) and car (red) 27
Figure 17. Spot S1: Dewarped image with detection lines and car........................ 27
Figure 18. Overview of spot calibration process 28
Figure 19. Overview of aisle calibration process... 29
Figure 20. Example parking area image (${GIS_DIR}/parking.png)........................ 31
Figure 21. Global map and camera image for camera A 33
Figure 22. Polygons drawn in an example parking area..................................... 35
Figure 23. Spot calibration by drawing spot polygons and lines 37
Figure 24. Example parking location with camera and fields of view 45
Figure 25. System overview of a multi-camera tracker..................................... 46
Figure 26. Stateful stream processing in the reference application 54
Figure 27. The React component hierarchy 66
Figure 28. How components are placed on the Garage page 69
Figure 29. Home page with a garage marker 70

Figure 30. The garage overlay display 70
Figure 31. Analytics dashboard built with Kibana 76

LIST OF TABLES

Table 1. Aisle calibration CSV file.. 36

1.0 ARCHITECTURE

This document describes the full end to end capability that is available with DeepStream 3.0. The below architecture provides a reference to build distributed and scalable DeepStream applications.

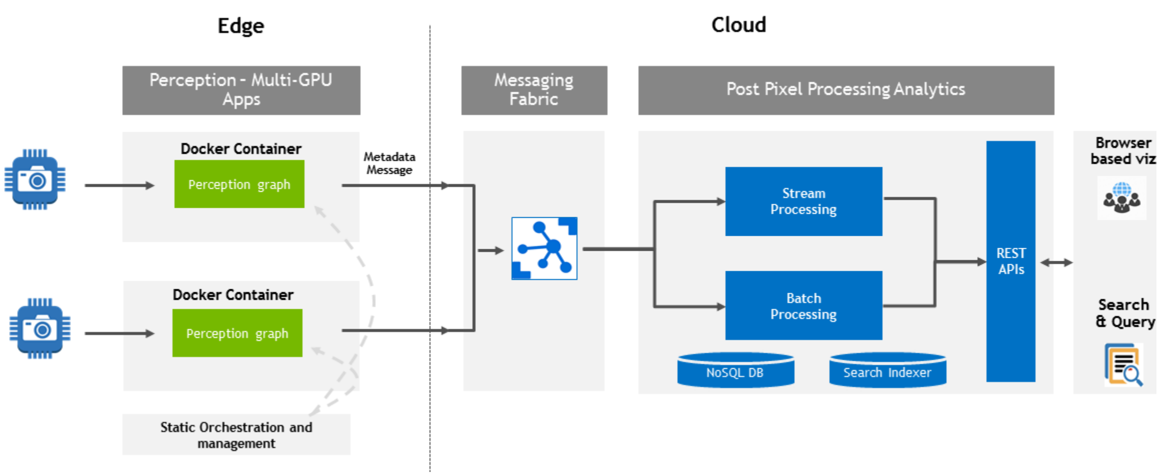

Figure 1. DeepStream 3.0 architecture

The perception capabilities of a DeepStream application can now seamlessly be augmented with data analytics capabilities to build complete solutions, offering rich data dashboards for actionable insights. This bridging of DeepStream's perception capabilities with data analytics frameworks is particularly useful for applications requiring long term trend analytics, global situational awareness, and forensic analysis. This also allows leveraging major Internet of Things (IOT) services as the infrastructure backbone.

1.1 EDGE TO CLOUD INTEGRATION

The data analytics backbone is connected to DeepStream applications through a distributed messaging fabric. DeepStream 3.0 offers two new plugins, `gstnvmsgconv` and `gstnvmsgbroker`, to transform and connect to various messaging protocols. The protocol supported in this release is Kafka.

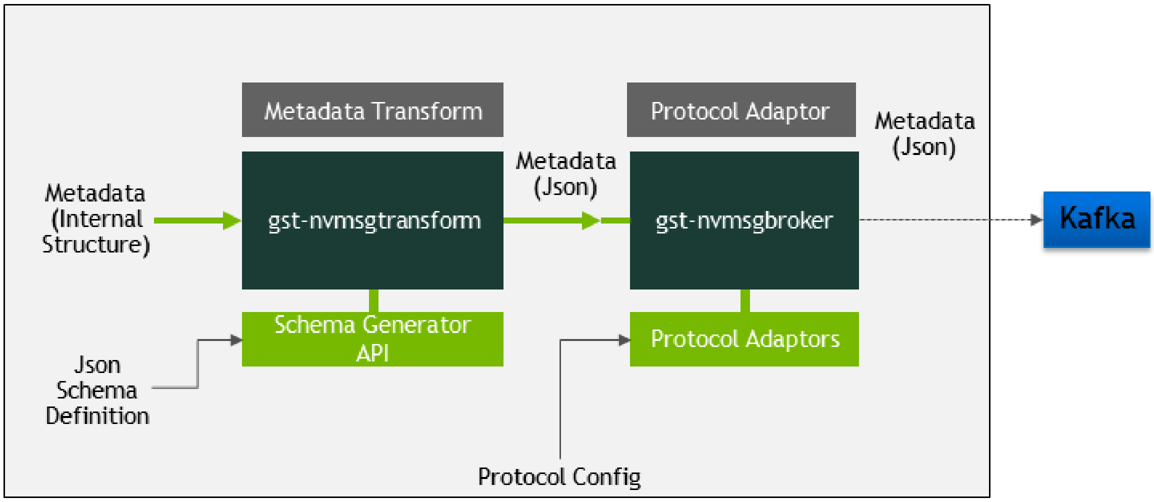

Figure 2. Use of protocols

1.2 STREAMING PIPELINE

To build an end to end implementation of the Analytics layer, DeepStream 3.0 uses open source tools and frameworks that can easily be reproduced for deployment on an on-premise server or in the Cloud.

The framework comprises stream and batch processing capabilities. Every component of the Analytics layer, Message Broker, Streaming, NoSQL, and Search Indexer can be horizontally scaled. The streaming analytics pipeline can be used for processes like anomaly detection, alerting, and computation of statistics like traffic flow rate. Batch processing can be used to extract patterns in the data, look for anomalies over a period of time, and build machine learning models. The data is kept in a NoSQL database for state management, e.g. the occupancy of a building, activity in a store, or people movement in a train station. This also provides the capability for forensic analytics, if needed. The data can be indexed for search and time series analytics. Information generated by streaming and batch processing is exposed through a standard API for visualization. The API can be accessed through REST, WebSocket, or messaging, based on the use case. The user interface allows the user to consume all the relevant information.

Deployment is based on an open source technology stack. The modules and technology stack are shown with respect to the Streaming Data pipeline.

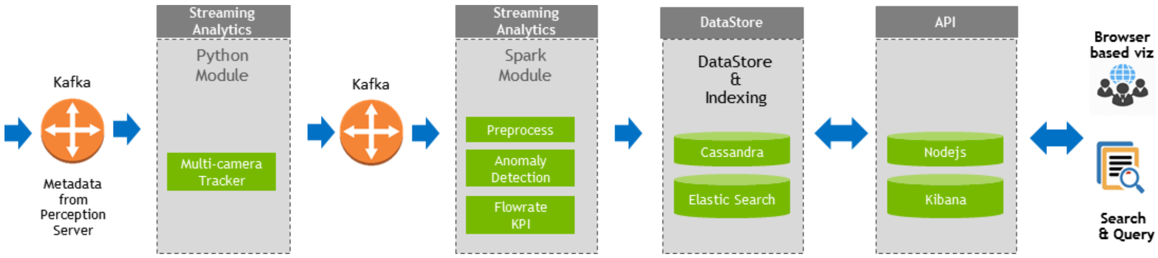

Figure 3. Streaming data pipeline

The reference implementation uses Kafka as the message broker. Processing between modules is decoupled using Kafka when needed.

The reference implementation uses Apache Spark for stream processing. A custom Python module performs multi-camera tracking. The key streaming modules are:

- Multi-camera tracking is explained below. Primarily it is responsible for deduplicating detection of the same object seen by multiple cameras and tracking the object across cameras.
- The preprocessing module validates every JSON message sent from the perception layer. It maps each JSON message to a strongly typed domain object.
- The anomaly detection module maintains the state of each vehicle or other object. The trajectory of each vehicle is maintained over time, so it is very easy to compute information like speed of vehicle, how long a vehicle has stayed in a particular location, and whether a vehicle is stalled in an unexpected location.
- The flowrate module is used to understand traffic patterns and flow rate. This involves micro-batching data over a sliding window.

Data is persisted in Cassandra and Elasticsearch. Cassandra maintains the state of buildings, intersections, parking garages, etc. at a given point in time. For example, the state of a parking garage comprises parking spot occupancy, car movement in the aisle, and car entry and exit. The application can show the current state of the garage, and enables playback of events from a given point in time. All data related to events and anomalies are indexed in Elasticsearch for search, time series analytics, and dashboard.

1.3 BATCH PIPELINE

DeepStream 3.0 performs batch processing based on the accumulated data over a period of time. The data ideally is stored in a distributed file system like HDFS or S3, but since the reference application is deployed in a Docker container and not in the Cloud, it reads all of the data from a given Kafka topic to do batch processing on it. The output of batch processing is stored in the persistent store and is consumed by the API.

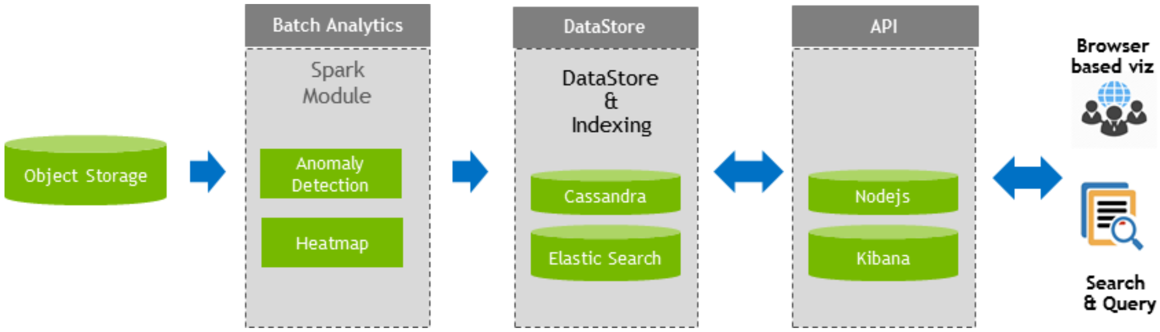

Figure 4. Batch data pipeline

The API layer is built using Node.js. It provides a REST API to query events, alerts, stats information. It also provides WebSocket, which is used for live streaming. The user interface uses React.js.

2.0 DOCKER DEPLOYMENT

To demonstrate the full end to end capabilities of DeepStream 3.0, and to help developers jump-start development, DeepStream 3.0 comes with a complete reference application which provides a smart parking solution for a garage or parking lot. You can deploy this reference application on edge servers or in the Cloud. You can leverage this application and adapt it to specific use cases. The reference application provides Docker containers to further simplify deployment, adaptability, and manageability.

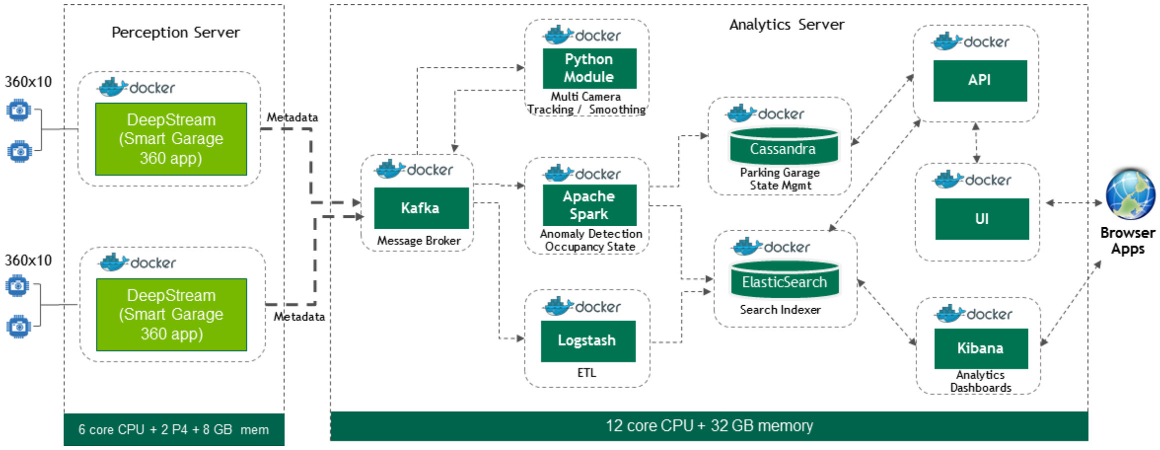

Figure 5. Perception server deployment

2.1 CLONE THE SMART PARKING APPLICATION REPOSITORY

The 360-D smart parking application is available from GitHub at:

```
https://github.com/NVIDIA-AI-IOT/deepstream_360_d_smart_parking_application.git
```

Download the application, either by clicking the Download button on the GitHub page, or by entering the command:

```
git clone https://github.com/NVIDIA-AI-IOT/deepstream_360_d_smart_parking_application.git
```

2.2 DEPLOYING THE ANALYTICS SERVER

This section explains how to deploy the analytics server.

Dependencies

Make sure you have a recent version of Docker installed on your local machine.

Environment Variables

Export the following environment variables:

- `IP_ADDRESS`: The IP address of host machine
- `GOOGLE_MAP_API_KEY`: The API key for Google Maps

Follow the instructions in "Get API Key" to get an API key for Google Maps:

```
https://developers.google.com/maps/documentation/javascript/get-api-key
```

Configurations

The application runs in two modes:

- Playback: Used to play back events from a specified point in time.
- Live: Used to see events and scene, as and when they are detected.

The application's default mode is Playback.

If you use live mode:

- Go to `node-apis/config/config.json` and change the configuration setting `garage.isLive` to `true`.
- Send the data generated by DeepStream 3.0 to the Kafka topic `metromind-raw`.

To install the Analytics Server

1. Install Docker.
2. Go to the analytics_`server_docker directory` (the top level directory in the repository downloaded or cloned in section 2.1).

   ```
   cd analytics_server_docker
   ```

3. Change configurations (if required).
4. Export the environment variables that specify the IP address of the host machine and the Google Map API Key.

   ```
   export IP_ADDRESS=xxx.xxx.xx.xx
   export GOOGLE_MAP_API_KEY=<YOUR GOOGLE_API_KEY>
   ```

5. Enter this command:

   ```
   sudo -E docker-compose up -d
   ```

   The command starts the following containers:

   - cassandra
   - kafka
   - zookeeper
   - spark-master
   - spark-worker
   - elasticsearch
   - kibana
   - logstash
   - api
   - ui
   - Python-module

6. When all the containers are up and running, start the `Spark Streaming` job:

   ```
   sudo docker exec -it spark-master /bin/bash
   ./bin/spark-submit  --class com.nvidia.ds.stream.StreamProcessor --
   master spark://master:7077 --executor-memory 8G --total-executor-
   cores 4 /tmp/data/stream-360-1.0-jar-with-dependencies.jar
   ```

7. If the application is running in playback mode, run the `spark` job to generate playback data.

```
sudo apt-get update
sudo apt-get install default-jdk
sudo apt-get install maven
cd ../stream
sudo mvn clean install exec:java -
Dexec.mainClass=com.nvidia.ds.util.Playback -
Dexec.args="<KAFKA_BROKER_IP_ADDRESS>:<PORT> --input-file <path to
input file>"
```

Note:	**Replace <KAFKA_BROKER_IP_ADDRESS> and <PORT> respectively with host IP address and port used by Kafka.** **Set the path to the input file as `data/playbackData.json` for viewing the demo data.** **You may add the option `--topic-name` to the command line to specify the Kafka topic to which data is sent. Set it to `metromind-raw` if input data is not tracked, or to `metromind-start` if input data has already gone through the tracking module. The default value used in step 7 is `metromind-start`.** **With the additional option, the command looks like this:** **`sudo mvn clean install exec:java -`** **`Dexec.mainClass=com.nvidia.ds.util.Playback -`** **`Dexec.args="<KAFKA_BROKER_IP_ADDRESS>:<PORT> --input-file`** **`<path to input file> --topic-name <kafka topic name>"`**

8. If the application is running in live mode, start the perception server (see section 2.3).
9. (Optional) Start the `spark` batch job, which detects the overstay anomaly. (The overstay anomaly occurs when a vehicle overstays by more than a configurable length of time.)

   Perform this step when enough data (at least 30 minutes) has been sent by the perception server.

   Using a second shell, run the following command to log in to spark master:

```
sudo docker exec -it spark-master /bin/bash
```

   Run the batch job:

```
./bin/spark-submit  --class com.nvidia.ds.batch.BatchAnomaly --
master local[8]  /tmp/data/stream-360-1.0-jar-with-dependencies.jar
```

10. Create `Elasticsearch start-Index` (optional):

Open the Kibana URL in your browser:

```
http://IP_ADDRESS:5601
```

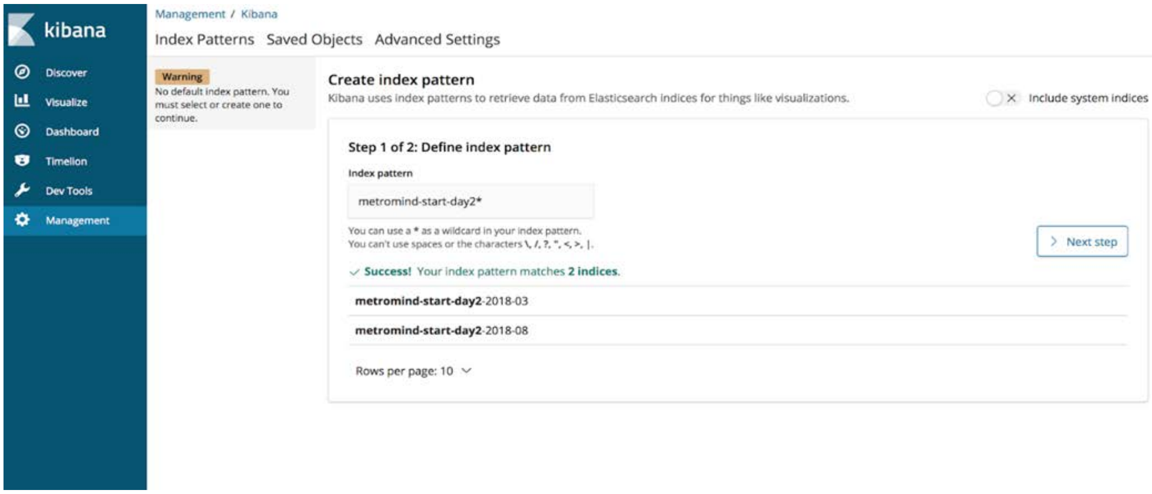

Figure 6. The page at the Kibana URL

11. Create `Elasticsearch anomaly-Index` (optional):

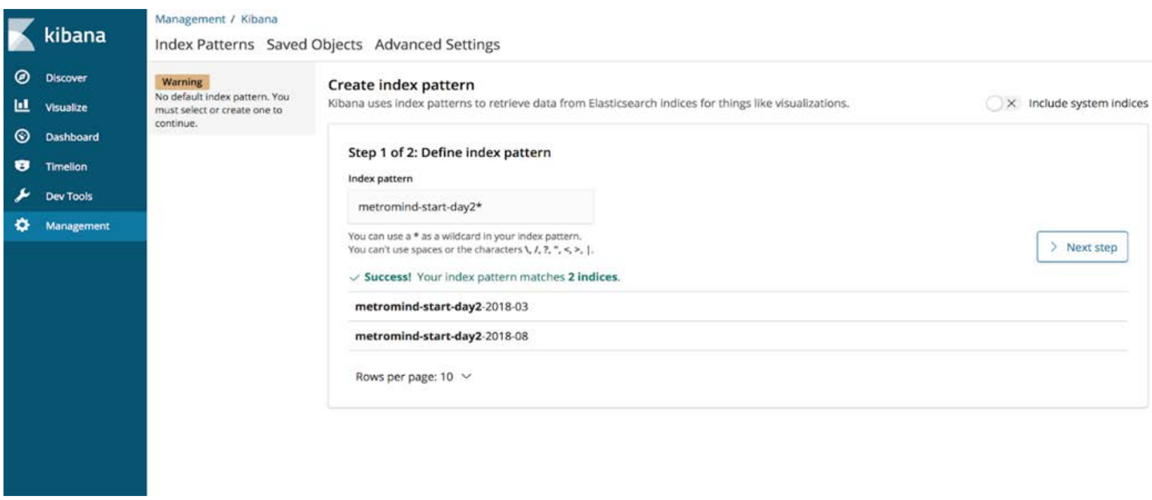

Figure 7. Create Elasticsearch anomaly-Index

12. To test your work, open `http://<IP_ADDRESS>` in your browser. If the page displays correctly, it looks like this:

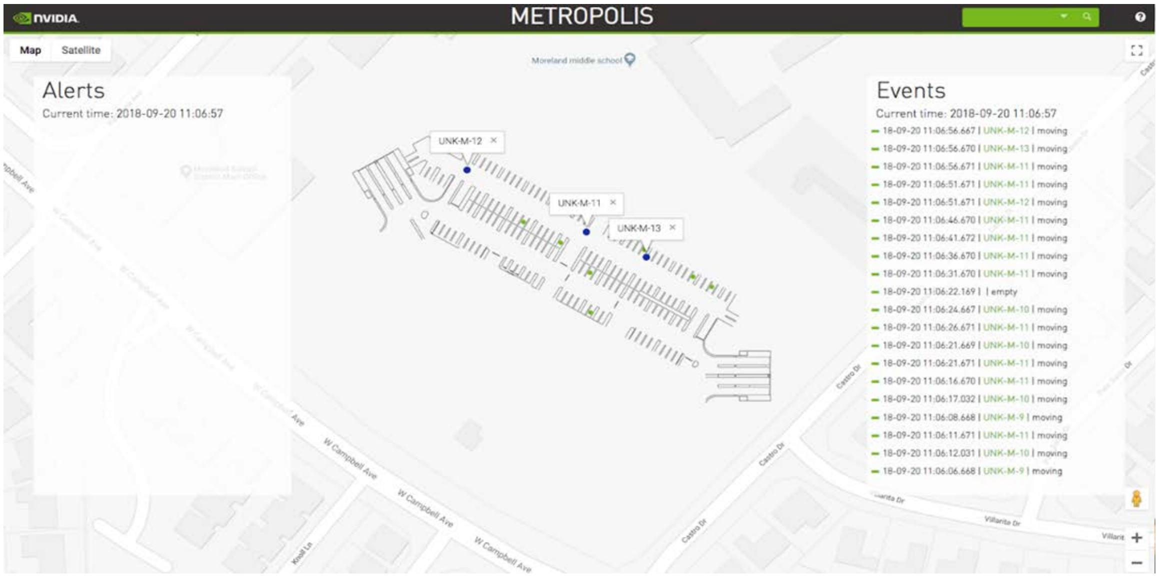

Figure 8. Test by opening http://<IP_ADDRESS>

Notes:

Docker deployment and the instructions above are sufficient to get you started, but each section has its own details, which you must understand if you want to build or modify each of the component separately.

You can use `start.sh` to start all the docker containers and run the `spark` streaming job. Before running the script you must replace `xxx.xxx.xx.xx` with the IP address of the host machine, and replace `<YOUR GOOGLE_API_KEY>` with the Google API key. If you choose to run the start script, then you may step steps 4–6 above. After the `spark` streaming job starts, you must run DeepStream Application Graph. You can run the script with the following command:

```
./start.sh
```

Start the DeepStream application only when the analytics server is up and running. Remember to shut down the docker-containers of the analytics server once the DeepStream application is shut down.

2.3 PERCEPTION SERVER DEPLOYMENT

The perception server is located in the `perception_docker` directory in the application package.

Follow the instructions below, or the similar instructions in the README file in the `perception_docker` directory, to obtain and run the `deepstream 360d` application:

1. Install any missing prerequisites for the perception server.
   - The host machine must have NVIDIA display driver version 410, as required for CUDA 10.

- `nvidia-docker` is installed.

2. Go to the `perception_docker` directory:

```
cd ./perception_docker
```

3. Create a directory named `videos` and make it the current directory:

```
mkdir videos && cd videos
```

4. Download the videos from this location and place them in the `videos` directory:

```
https://nvidia.app.box.com/s/ezzw0js1ti555vbn3swsggvepcsh3x7e
```

5. Log in to the NVIDIA container registry (nvcr.io):

```
docker login nvcr.io
```

Enter the username as `$oauthtoken` and copy your NGC APIKey as the password.

Note:	The login is sticky, and need not be repeated each time you deploy the server.

For more information, see:

```
https://docs.nvidia.com/ngc/ngc-getting-started-guide/index.html
```

6. Execute the `run.sh` script. (You may have to use `sudo`, depending on how docker is configured on your system.)

   This script pulls the 360-D docker image from NGC and runs the container.

7. After the container is started, install an editor such as nano in it. You need this editor to edit the configuration file.

8. Edit your chosen configuration file (located in the `DeepStream360d_Release/samples/configs/deepstream-360d-app/` directory) to set the URL of the Kafka broker instantiated in section 2.2.

9. Enable logging (optional):

```
DeepStream360d_Release/sources/tools/nvds_logger/setup_nvds_logger.s
h
```

10. Run the 360-D application:

```
deepstream-360d-app -c <path to config file>
```

The DeepStream360d_Release/samples/configs/deepstream-360d-app/ application directory contains two sample configuration files, each configured to be run with the deepstream 360d application to process a subset of the videos downloaded in step 3. The configuration files are:

- sample10_gpu0.txt: processes ten input videos on GPU 0
- sample9_gpu1.txt: processes nine (distinct) videos on GPU 1

On a system with two available GPUs that can each support the deepstream 360d application pipeline, you can run separate instances of the application concurrently to process a total of 19 videos on two GPUs. Command lines for executing the application with both configuration files are shown below:

```
deepstream-360d-app -c
DeepStream360d_Release/samples/configs/deepstream-360d-
app/source10_gpu0.txt
deepstream-360d-app -c
DeepStream360d_Release/samples/configs/deepstream-360d-
app/source9_gpu1.txt
```

Note:	**This is a “deployment” container, meaning that any compiler toolchains, header files, build libraries, etc. that are required to build samples and other software are not included in the container image.**

3.0 PERCEPTION LAYER: 360-D SMART PARKING APPLICATION

This application is a smart parking sample application using cameras with a 360° view.

This application detects the occupancy status of parking spots, i.e. whether they are empty or occupied. It also detects entry and exit of cars with cameras placed on the entrance of the garage, and tracks vehicle movement in the aisle areas of the garage.

Figure 9 shows the high level architecture of the 360-D application.

The generated metadata is sent to the Cloud for further processing.

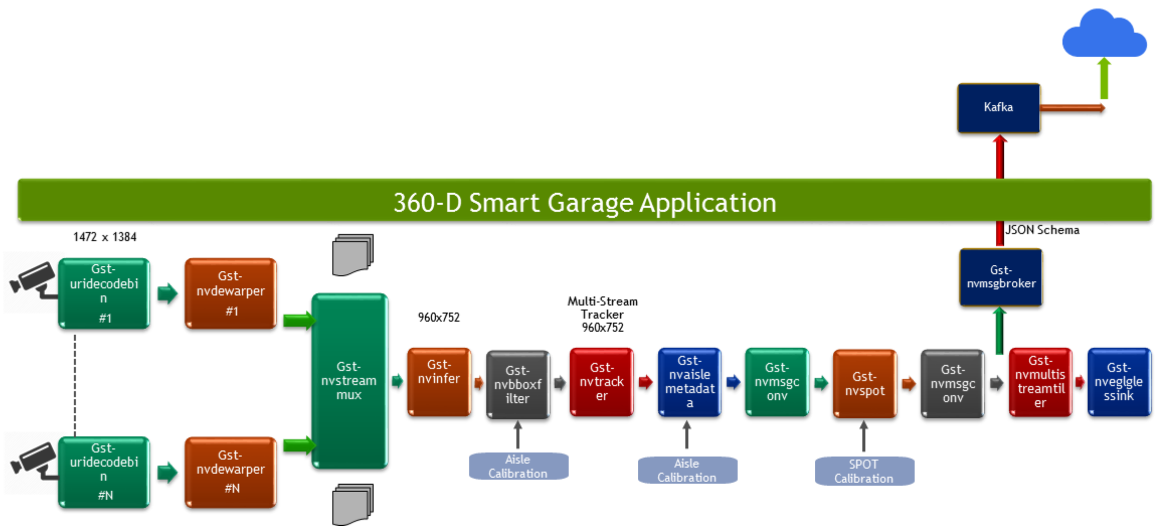

Figure 9. The 360-D application's architecture

3.1 THE GST-URIDECODEBIN PLUGIN

All of the 360° cameras transmit H.264 encoded frames over RTSP. This RTSP packetized data is received by the DeepStream pipeline. The `uridecodebin` plugin depacketizes the incoming stream and feeds the encoded data to the hardware accelerated decoder. The decoder decodes the frames into NV12 format.

Since the cameras give a 360° view, the decoded output (as shown below) must be dewarped for the areas of the interest. Two areas of interest in this example are Spot (`pushbroom` projection surface) and Aisle (`vertical cylinder` surface). Spot is a parking side view and Aisle is a passage area view where vehicles movement occurs.

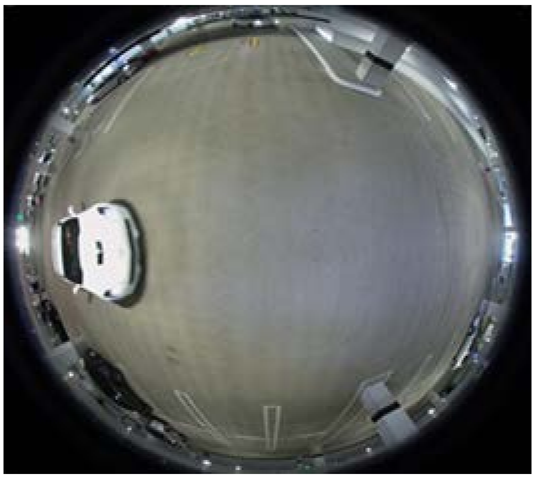

Figure 10. Typical output frame of decoder (1472×1384)

3.2 THE GST-NVDEWARPER PLUGIN

This plugin performs two functions: dewarping and scaling on dewarped surfaces.

3.2.1 Core Dewarping

The application sends decoded NV12 frames to the Dewarper binary. The Dewarper binary consists of the `nvidconv` plugin, which does format conversion, and `dewarper` plugin, which does dewarping.

The `dewarper` plugin uses an optimized GPU dewarping algorithm. The `nvvidconv` plugin converts decoded NV12 data to RGBA data and feeds it to the `dewarper` plugin. The `dewarper` plugin processes this data to produce two Aisle areas of interest, each of size 1902×1500, and two Spot areas of interest, each of size 3886×666 pixels.

The `dewarper` plugin takes dewarping parameters either from the `[sourceX]` group or from the `[property]` group in `config_dewarper.txt`. For the 360-D application, the aisle and spot calibration files provide the dewarping parameters. The dewarping parameters include the resolution for dewarping surfaces, yaw, roll, pitch, top angle, and bottom angle. Output of the dewarper algorithm is shown below.

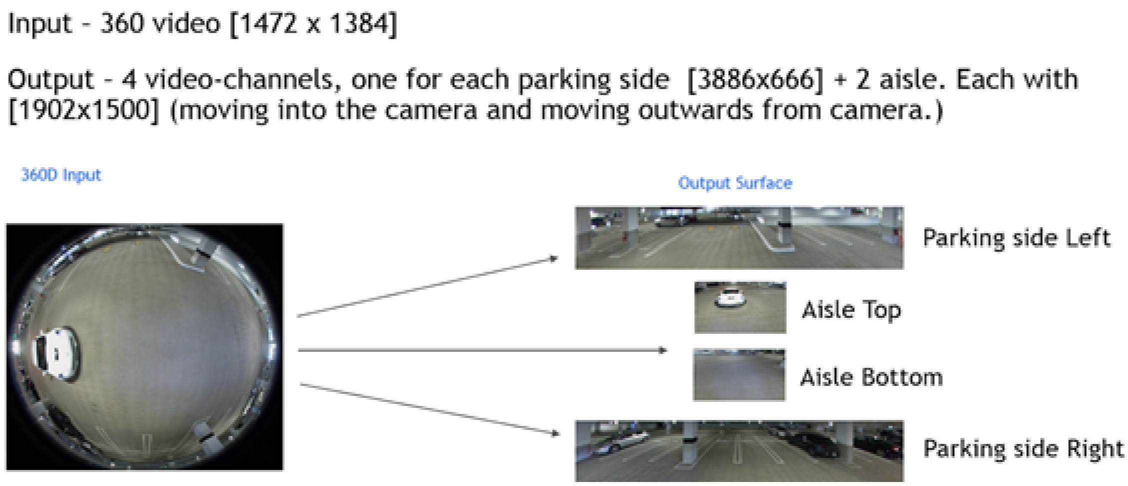

Figure 11. Sample output of the dewarper algorithm

3.2.2 Preprocessing

The `dewarper` plugin creates four surfaces. Each surface may be of type `NVDS_META_SURFACE_FISH_PUSHBROOM` (Spot view) or `NVDS_META_SURFACE_FISH_VERTCYL` (Aisle view). Each input stream from a 360° camera has two Spot surfaces and two Aisle surfaces.

You can set the dimensions of the dewarped output buffer with the `output-width` and `output-height` properties in `config-dewarper.txt`. The `dewarper` plugin scales the Spot and Aisle surfaces (3886×666 and 1902×1500 respectively) to `output-width` and `output-height` with appropriate padding. The output of scaled surfaces is shown below. In this example dewarped surfaces (from output of dewarper algorithm) are scaled to 960×752 because inference operates on that resolution.

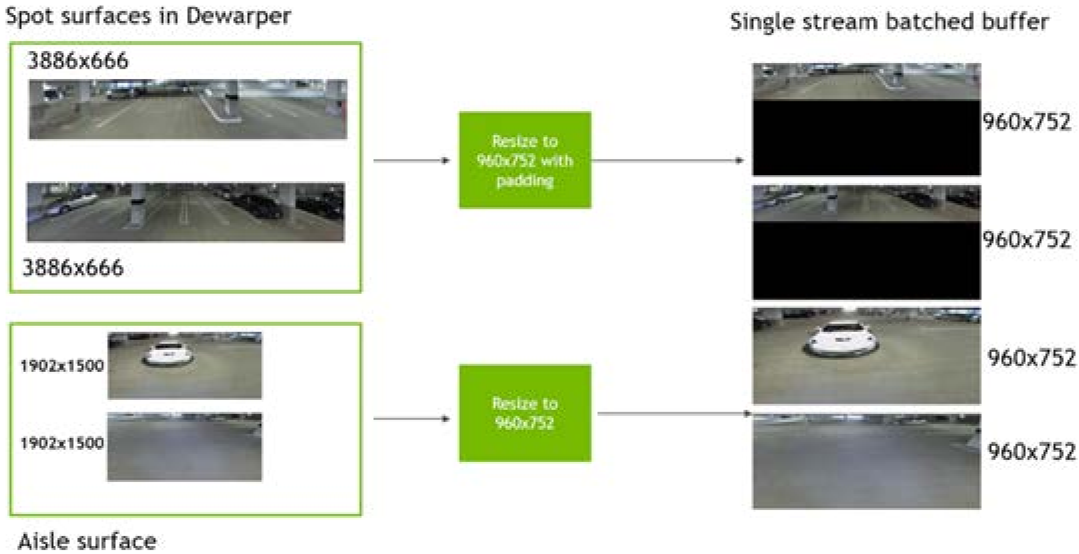

Figure 12. Output of scaled surfaces

3.3 THE GST-NVSTREAMMUX PLUGIN

The application feeds the output of each dewarper binary, consisting of four dewarped surfaces, to the muxer plugin, Gst-`nvstreammux`.

The muxer forms a batched buffer of all the dewarped surfaces from each stream and passes it to the inference plugin, Gst-`nvinfer`.

3.4 THE GST-NVINFER PLUGIN

`Gst-nvinfer` uses a Resnet-based model which detects two classes, `car` and `car front/back`). The `car` class is used for Aisle object tracking, and the `car front/back` class is used for spot detection.

3.5 THE GST-NVBBOXFILTER PLUGIN

The inference plugin's output is passed to the `nvbbox` filter, which filters detected objects based on class type, e.g. `car` and `car front/back`. It keeps only `car front/back` objects from the `pushbroom` projection surface (Spot), and `car` objects from `vertical cylinder` (Aisle). This filtering is required because spot availability analysis must be performed only on `car front/back` objects from `pushbroom`, and tracking of moving objects must be performed only on `car` objects from `vertical` cylinder, which is Aisle area. The car objects from vertical cylinder (Aisle area) are filtered further based on whether an object is in the region of interest (RoI). The RoI is specified in the aisle calibration file, which is selected by setting the aisle-calibration-file parameter in the [aisle] group in the application configuration file.

3.6 THE GST-NVTRACKER PLUGIN

The Gst-nvbboxfilter plugin passes the filtered car objects to Gst-nvtracker, which tracks objects detected within a single camera view or a single video. The tracker generates a unique ID for each object. This is a multi-stream tracker.

The output of Gst-nvtracker is passed downstream to the Gst-nvaislemetadata plugin.

3.7 THE GST-NVAISLEMETADATA PLUGIN

The plugin operates on metadata which it receives with the buffer from the upstream Gst-nvtracker plugin. Based on the bounding box coordinates of the detected objects, the Gst-nvaislemetadata plugin determines the "moving/entry/exit" status of a car. The status of a car in the RoI region is then attached to the NvDsEventMsgMeta metadata and sent to the next downstream component, Gst-nvmsgconv.

This plugin also transforms local object coordinates to global coordinates and attaches them to NvDsEventMsgMeta. It reads the RoI and perspective transfer (H matrix) related parameters from the aisle calibration file.

For more details, refer to Aisle Calibration.

3.8 THE GST-NVSPOT PLUGIN

The Gst-nvspot plugin determines whether configured parking spots are occupied or empty.

The plugin accepts an input buffer along with object metadata from the upstream plugin Gst-nvbboxfilter. It also reads camera calibration parameters from the spot calibration file specified by the selected spot-calibration-file parameter in the application configuration file's [spot] group. Based on the bounding box coordinates of the detected objects and calibrated lines for the given spot, it determines the "occupied/empty" status of the parking spot. It attaches the status of the parking spots to the metadata in `NvDsEventMsgMeta` and sends it to the next downstream `Gst-nvmsgconv` plugin.

3.9 THE GST-NVMSGCONV PLUGIN

This plugin generates a schema payload from the metadata in `NvDsEventMsgMeta` to the buffer. It accepts a buffer from the `Gst-nvspot` and `Gst-nvaislemetadata` plugins.

The generated payload is attached to the input buffer as `NvDsPayload` metadata.

By default, it uses the DeepStream schema specification to generate a payload in JSON format.

3.10 THE GST-NVMSGBROKER PLUGIN

This plugin sends payload messages to the Analytics Server for further processing, using the Kafka communication protocol. It accepts any buffer that has `NvDsPayload` metadata attached, and sends messages to the server or Cloud for further analysis.

Note:	**For more details about these plugins, the *DeepStream 3.0 Plugin Manual*, which is shipped with DeepStream 3.0 SDK. You may also refer to the source code of the 360-D application.**

4.0 CALIBRATION AND REGIONS OF INTEREST

An important issue with extracting usable data from video streams is how to translate an object detected by the camera into a geolocation. Take a traffic camera as an example. When the camera sees a car, the raw image of the car isn't useful to a smart cities system. The car would ideally be placed in an information grid that also projects a live bird's eye view of the activities in the city, for an operator's use.

To do this, an application must translate the camera image into coordinates (e.g., latitude and longitude) representing the car's location. Technically, this is a transformation from the image plane of the camera (the image of the car) to a global geolocation (a latitude and longitude). Transformations like this are critical to a variety of use-cases beyond simple visualization. Solutions that require multi-camera object tracking, movement summarization, geofencing, and other geolocating operations for business intelligence and safety can leverage the same technique, called **calibration**.

Calibration enables an application to perform transformations between camera and global spaces. Broadly speaking, calibration is a tool to transform any pixel location in the camera space to a corresponding geolocation.

4.1 TYPES OF CALIBRATION

As discussed in section Perception Layer: 360-D Smart Parking Application, the reference application uses 360° cameras to monitor the garage. The original image from a 360° camera has objects appearing in shapes that look distorted.

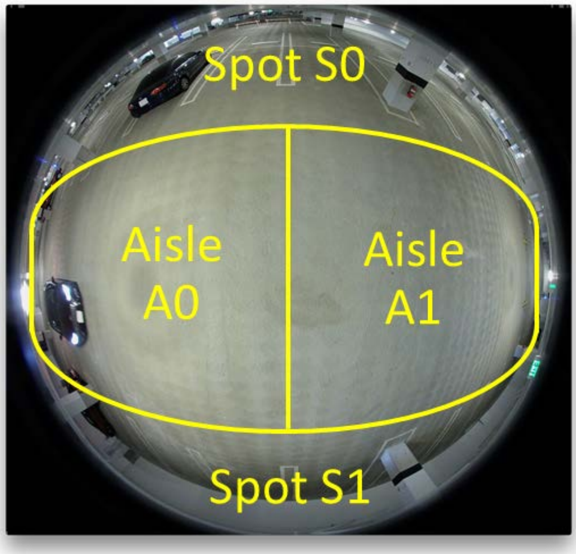

Figure 13. Distorted image from a 360° camera

The application dewarps this image into four surfaces: two aisle surfaces (A0 and A1) and two parking spot surfaces (S0 and S1).

The car is distorted in the original 360° camera view. Many deep learning based object detection systems cannot detect this distorted object since they are trained to detect cars in non-distorted images.

The application uses regions of interest (RoIs) in the four dewarped surfaces to perform calibration:

- Aisle surfaces (A0 and A1): Figure 14 and Figure 15 below show the dewarped image surfaces of the aisle, A0 and A1. Each aisle surface has an RoI marked as a yellow polygon. Only cars detected within the RoI are considered. (The blue box in Figure 14 is an example). Cars not lying within this the RoI are ignored.

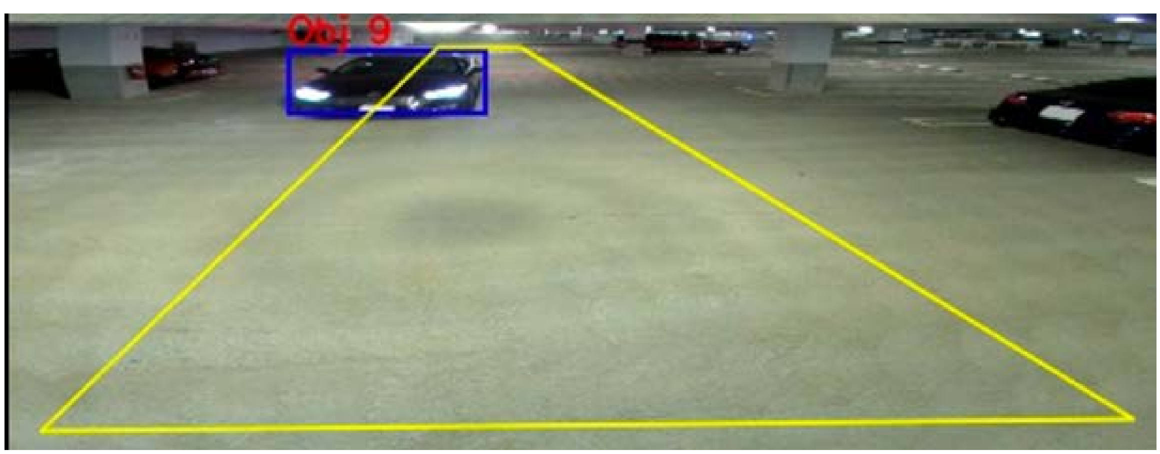

Figure 14. Aisle A0: Dewarped image with RoI (yellow box) and car (blue box)

Figure 15. Aisle A1: Dewarped image with RoI

- Parking spot surfaces (S0 and S1): Each parking spot within these surfaces has an imaginary **detection line** (marked in yellow below). If a car's bounding box cuts the line, that car is considered to be parked in that spot (e.g., the red boxes below). Otherwise the application considers that parking spot empty.

Figure 16. Spot S0: Dewarped image with detection lines (yellow) and car (red)

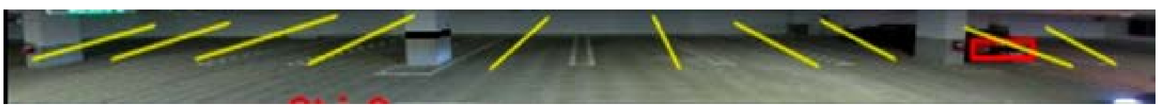

Figure 17. Spot S1: Dewarped image with detection lines and car

4.2 DESIGN

Both parking spots and aisles must be calibrated to get accurate locations for parked and moving cars.

4.2.1 Spot Calibration

Spot calibration marks each parking spot to a geolocation. To perform spot calibration we associate spot calibration lines (yellow lines in Figure 16 and Figure 17) with a global polygon. Figure 18 shows the overall process.

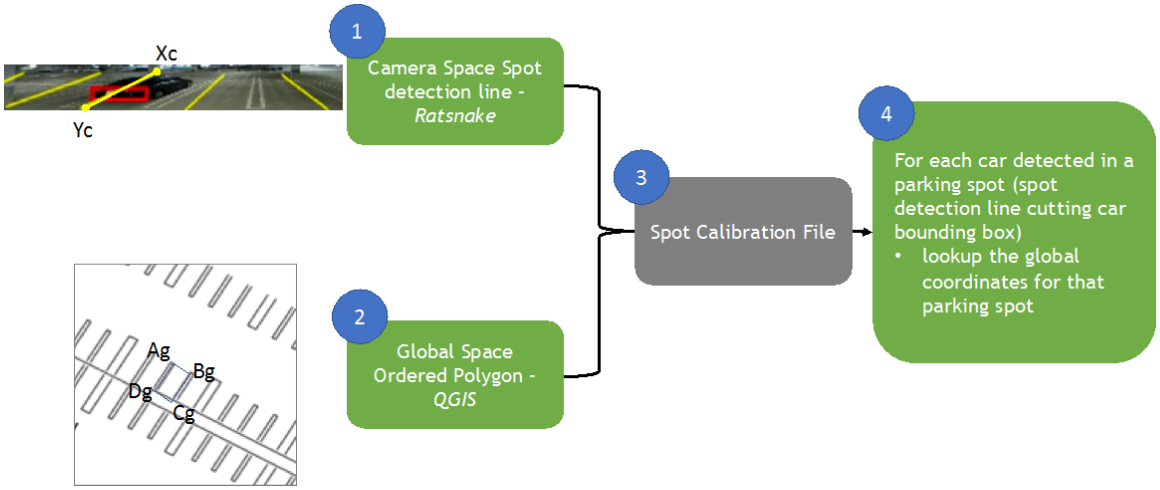

Figure 18. Overview of spot calibration process

The steps to annotate of the spot image and the global polygons are:

1. Use the image annotation tool Ratsnake to draw a line (the spot detection line) on every spot in the dewarped spot images. The line defines two points on the camera plane (e.g. points X_c, Y_c).
2. Use the GIS tool QGIS to draw a corresponding image on the global polygon, resulting in four corresponding points (e.g. points A_g, B_g, C_g, D_g).
3. Create a CSV file which contains the information required for calibration. For each camera, insert one row which contains the information <CameraID, X_c, Y_c, A_g, B_g, C_g, D_g>.
4. The application stores this information in its spot calibration CSV file. DeepStream then computes the mapping of parked car detections and empty parking spots.

4.2.2 Aisle Calibration

Aisle calibration is more challenging than parking spot calibration. This section describes an approach to calibration that is meant for complex, scalable environments, but does not require a physical presence at the site.

At a high level, this approach is based on constructing corresponding polygons in the camera image and global maps. It then uses a transformation matrix to map camera space to global space.

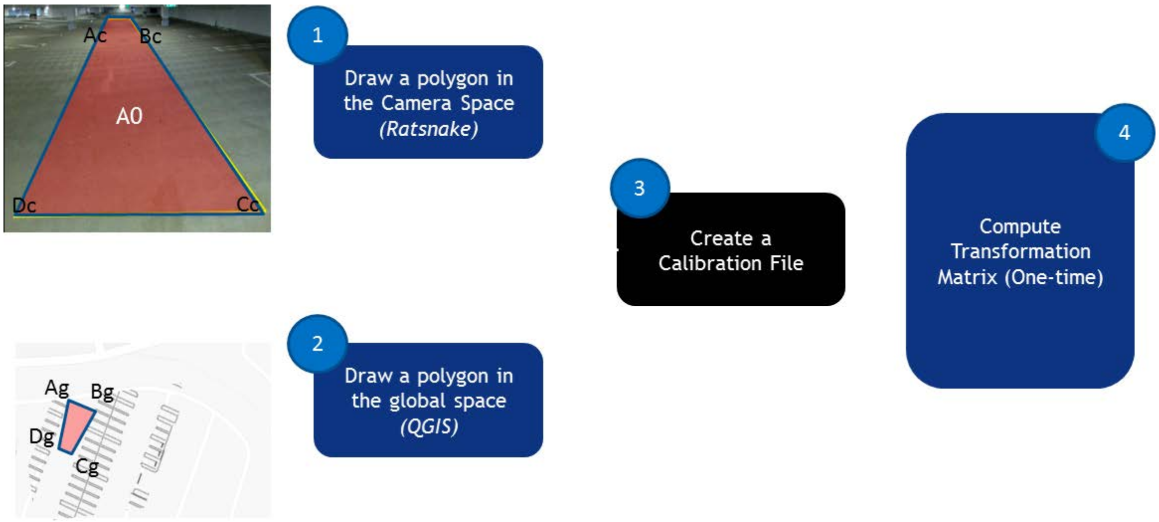

Figure 19. Overview of aisle calibration process

1. Use an image annotation tool to draw a polygon on one of the camera images. From this we get four points on the camera plane (e.g. points A_c, B_c, C_c, D_c).
2. Use a GIS tool to draw a corresponding image on the global polygon, resulting in four corresponding points (e.g., points A_g, B_g, C_g, D_g).
3. Create a CSV file which contains the information required for calibration. For each camera, insert one row which contains the information <CameraID, A_c, B_c, C_c, D_c, A_g, B_g, C_g, D_g>.
4. The application stores this information in the aisle calibration CSV. DeepStream then computes a transformation matrix (per-camera) that translates each pixel in the camera plane into global coordinates.

For each object detected by the camera, the transformation matrix is used to compute that object's global coordinates.

4.3 DETAILED METHODOLOGY

Calibration is a five-step process. The following sections describe each step.

4.3.1 Annotating maps

The process of annotating maps involves mapping coordinates in images and global maps. You do this with the open source tool QGIS. QGIS helps you draw polygons and lines with respect to a real map and export the resulting coordinates as a CSV file. You can use it to georeference the parking level image.

To learn more about installing and using QGIS, see:

- General information: http://www.qgistutorials.com/en/
- Georeferencer plugin: https://docs.qgis.org/2.8/en/docs/user_manual/plugins/plugins_georeferencer.html
- Georeferencing documentation: http://www.qgistutorials.com/en/docs/georeferencing_basics.html

4.3.2 Annotating images

Many image annotation tools are available. Ratsnake is a good, freely available tool; see https://is-innovation.eu/ratsnake/.

4.3.2.1 Step 1: Capture Image Snapshots from Cameras

The first step in calibration is to get snapshot images from all cameras. Snapshots must show clear, **salient feature points** in the region of interest. These salient feature points are to be mapped to the features seen on a global map. For example, if the cameras are installed inside a parking garage, the snapshots must clearly show pillars, parking spot lines painted on the ground, and other features of the building itself. Take the snapshots when the garage is empty or near-empty, so as few vehicles, pedestrians, and other large objects as possible are present to block the building features.

Store the snapshots in a directory and label it for easy reference. For example, store the snapshots in a directory called `CAM_IMG_DIR=/mnt/camdata/images/`. You may name individual snapshots with the IP address of the camera that took them and the name of surface that contains them. For example, you may save the aisle surfaces of camera at IP address 10.10.10.10 as `${CAM_IMG_DIR}/10_10_10_10_A0.png` and `${CAM_IMG_DIR}/10_10_10_10_A1.png`. You may save the same camera's spot surfaces as `${CAM_IMG_DIR}/10_10_10_10_S0.png` and `${CAM_IMG_DIR}/10_10_10_10_S1.png`.

4.3.2.2 Step 2: Blueprint/CAD Image

Download a blueprint or CAD image of the location being observed (the parking area, in this example). Save the map to a directory name `GIS_DIR` (e.g., `GIS_DIR==/mnt/camdata/gis/`). For example, you may save the `.png` file of the parking area as `${GIS_DIR}/parking.png`.

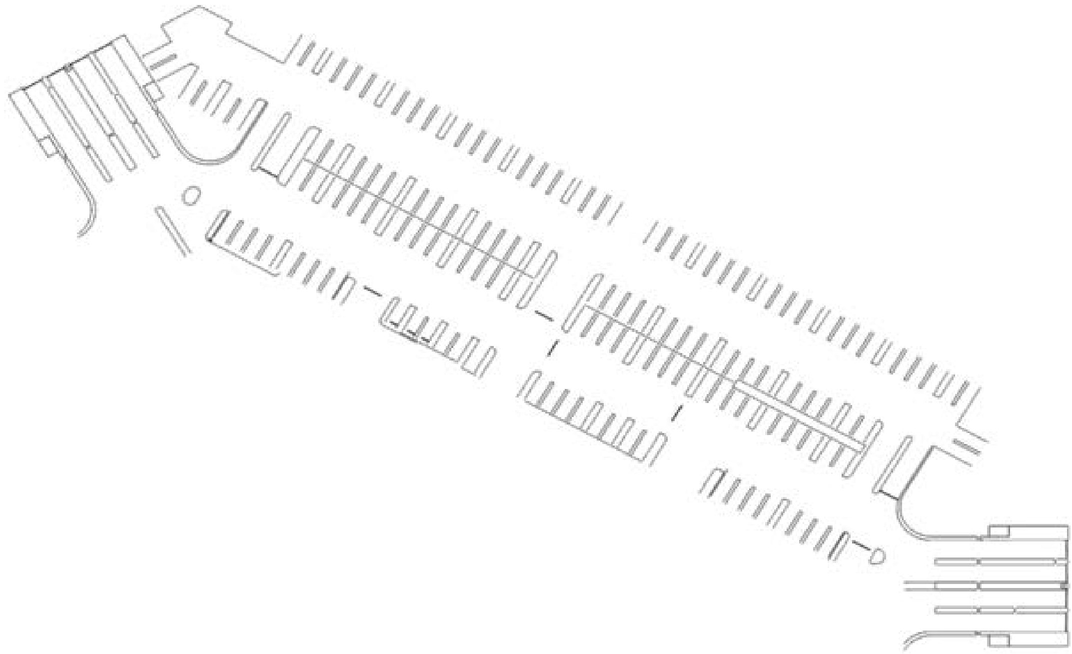

Figure 20. Example parking area image (${GIS_DIR}/parking.png)

4.3.2.3 Step 3: Georeferencing

Georeferencing is the process of mapping every point of the region being monitored into a global coordinate system, e.g., latitude and longitude. In other words, it maps the latitude and longitude for every point in the parking garage.

Depending on the region you are monitoring, you may be able to use existing maps — particularly for outdoor regions. If you're using traffic cameras to monitor an intersection and traffic light, for example, there may already be a Google or QGIS map you can use to get the coordinates of the intersection and/or the traffic light itself.

However, in many use cases there are no pre-existing georeferenced maps suitable for use in calibration. This is especially true in indoor situations, like the parking garage example. That said, you can often find CAD images or blueprints of buildings, and other indoor map files (usually in PDF or a picture format) that provide coordinates for at least some key points in the region.

Once you have your CAD image, blueprint, or other indoor map, you're ready to georeference it. The process is straightforward: Place the blueprint accurately on the global map using QGIS.

Our methodology works for georeferencing if the area in question has at least a few key salient feature points that you can locate in both the blueprint and the map. Examples might include pillars or corners of staircases.

The process of georeferencing consists of:

1. Using a GPS receiver (such as a smartphone), log the latitude and longitude coordinates of various feature points.
2. Open the QGIS application. Launch the Georeferencer plugin.
3. Open the blueprint (a JPG or PNG image) in the Georeferencer plugin and follow the guide for georeferencing. Going back to the example from steps 1 and 2 above, map the parking area by using the corresponding PNG file `${GIS_DIR}/parking.png`. Map each feature point on both the QGIS map and the blueprint image.
4. The resultant output is a georeferenced TIFF file that provides accurate geocoordinates to any point on the map. Georeferencing yields one image for each blueprint. Save the images in the file `${GIS_DIR}/parking.tif`.

4.3.3 Polygon Drawing for Aisle Calibration

This section walks through the detailed steps for calibrating one aisle with one camera (e.g., camera A with IP 10.10.10.10, aisle surface A0). Assume that the snapshots for each camera are stored in `${CAM_IMG _DIR}`. You would repeat these steps for each camera the application uses.

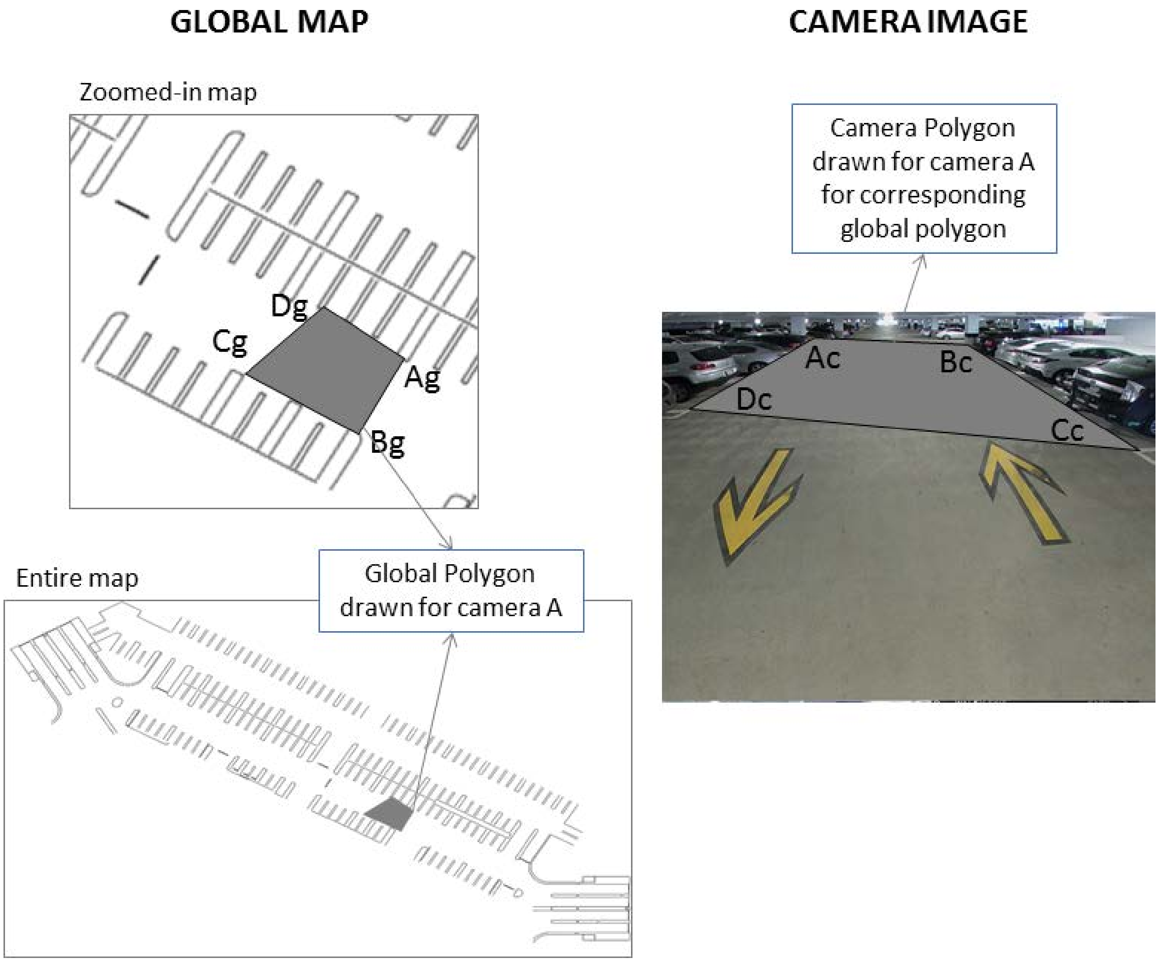

Figure 21. Global map and camera image for camera A

For each camera:

1. Open QGIS and load the global map. In this example, load the georeferenced image of the region covered by camera A. Since this camera covers the parking area, you load the file `${GIS_DIR}/parking.tif`.
2. Zoom in to the region covered by camera A on the global map. The left-hand side of Figure 21 shows the entire global map and a zoomed-in portion of the map.
3. Open the image snapshot `${CAM_IMG_DIR}/10_10_10_10_A0.png` using Ratsnake.
4. Identify the salient feature points that can be seen on both the global map and the snapshot. In this example, the salient feature points are the pillars and parking spot lines in the camera image.
5. Draw an identifying quadrilateral on the camera image using Ratsnake. Mark the points A_c, B_c, C_c and D_c (see the right-hand side of Figure 21).

6. Draw the same quadrilateral exactly on the global map. Call its corner points A_g, B_g, C_g, and D_g. (For details on drawing polygons in QGIS and Ratsnake, see Details of drawing polygons in QGIS and Ratsnake, below.)

Each point on the global map must map back to the corresponding point in the snapshot image, e.g. A_g must map to A_c, B_g to B_c, and so on. To map back the points in QGIS, you must draw the quadrilateral in the same direction, starting with the corresponding point, in both the camera image and QGIS.

This outline describes the details of drawing polygons in QGIS and Ratsnake:

1. To draw a polygon in QGIS, note the global coordinates of each corner point on the polygon (e.g. A_g, B_g, C_g, D_g). Each global coordinate (x, y) can be the number of meters from the origin ((x, y)=(0, 0)). The origin may be the center of the building, for example. In addition, the (longitude, latitude) for each point (x, y) is also given.

   To get the (x, y) from the QGIS tool:

   1. Drawing the polygons:

      1. Create a new "Vector Layer" in QGIS for drawing polygons.
      2. Add a feature named `CameraId` that corresponds to the ID of the camera's surface.
      3. Draw a polygon for each camera. Make sure that it has exactly four points, i.e. the polygon is a quadrilateral.
      4. Set the polygon's `CameraId` (surface ID) to the camera ID (e.g., `C_10_10_10_10_A0`).
      5. Make a note of the longitude and latitude of the origin (in this case, the center of the building).

      The below figure shows example polygons drawn for various cameras in an example parking area. The background image is the map (`${GIS_DIR}/parking.tif`). The gray boxes are the polygons that have been drawn.

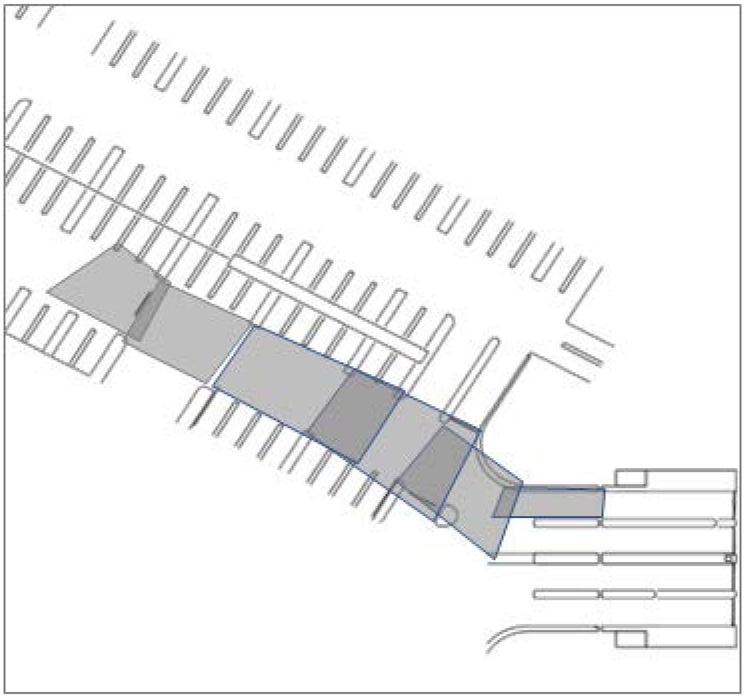

Figure 22. Polygons drawn in an example parking area

2. Get the Cartesian coordinates (x, y) of each vertex of the polygon from the geo-coordinates (latitude, longitude).

   You may get the geo-coordinates by reading the shapefile and getting the attributes of the shape. You may also use Python's `pyshp` package (https://pypi.org/project/pyshp/). The documentation shows how to read the latitude and longitude of the vertices and the polygon's camera ID.

   1. Export the vector layer created in step 1 as a CSV file. The data must have the following columns:

      - Camera ID
      - Coordinates (longitude and latitude) for each vertex of the polygon

   2. Fix a center point of the map and get this origin point's latitude and longitude.
   3. Convert each vertex's geo-coordinates (latitude, longitude) to Cartesian coordinates (x, y) based on the distance and angle between the origin and the vertex.

   You now have four global coordinate points relative to the origin: (gx0,gy0), (gx1,gy1), (gx2,gy2), (gx3,gy3).

3. Draw the polygon in Ratsnake.

   1. Export the camera coordinates of the polygon vertices A_c, B_c, C_c, D_c for each camera. Call them (cx0,cy0), (cx1,cy1), (cx2,cy2), (cx3,cy3) .

2. Export these points for each camera.
3. Update the aisle calibration CSV file (e.g., `nvaisle.csv`) with the below information. DeepStream uses this data to transform from camera coordinates to global coordinates.

Table 1. Aisle calibration CSV file

Column	Example	Comments
cameraId	10_110_127_164	ID of the camera
ipaddress	10.110.127.164	IP address of the camera
level	p1	Parking level
gx0	−105.8660603	Global coordinates
gy0	−12.57717718	
gx1	−105.9378082	
gy1	−4.760517508	
gx2	−96.0054864	
gy2	−4.86179862	
gx3	−95.99345216	
gy3	−11.80735727	
cx0	510	Camera coordinates
cy0	186	
cx1	1050	
cy1	126	
cx2	1443	
cy2	351	
cx3	21	
cy3	531	

4.3.4 Polygon Drawing for Spot Calibration

The process of drawing polygons for parking spots in QGIS is similar to the process of drawing polygons for aisles. You draw polygons around the parking spots instead of the aisles. This is easier in most cases, since parking spot maps have clearly demarcated parking spots as shown below.

In Ratsnake you draw lines for each parking spot on the spot surfaces (instead of drawing polygons), as shown below.

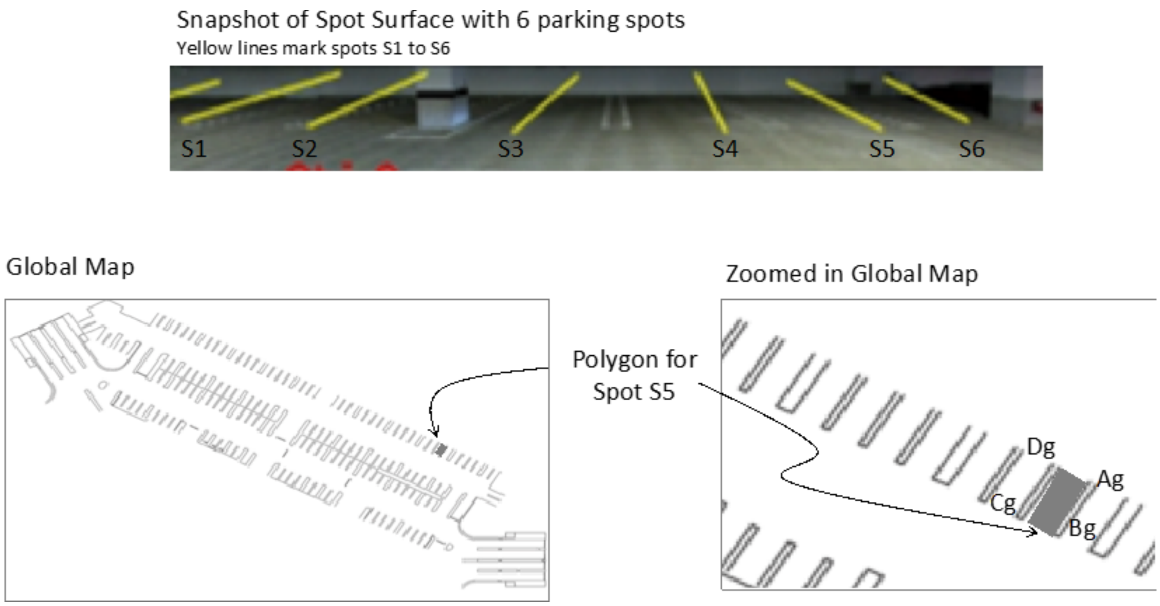

Figure 23. Spot calibration by drawing spot polygons and lines

4.3.5 Transferring CSV to the DeepStream Server

The CSVs created above for the aisle and spot file must be added to the DeepStream configuration directory, enabling DeepStream to infer the geolocations of detected cars and parking spots. See `nvaisle_2M.csv` and `nvspot_2M.csv` for the format of aisle and spot calibration files.

5.0 METADATA

Metadata is used to describe a scene. It acts as the glue between the perception layer and the analytics layer. The key elements of the metadata are

- **Sensor**: Represents the sensor (such as a camera) that recorded the scene. Attributes of this element contain details about the sensor, such as sensor ID and location.
- **VideoPath**: Pathname of video used for playback.
- **AnalyticsModule**: Identifies the module that analyzed the scene and generated the metadata.
- **Place**: Represents the type, location, and properties of the place where the event occurred, for example “building,” “intersection,” “airport,” etc.
- **Object**: Represents the type of an object in the scene, for example “person,” “vehicle,”, “bag,” etc.
- **Event**: Describes the event, e.g. whether a vehicle is “moving”, “stopped”, “entry” or “exit.”
- **Timestamp**: Represents the time when an event occurred.

5.1 ABOUT METADATA ELEMENTS

The following sections describe the types of metadata elements in more detail.

The metadata schema itself may be found at:

```
https://github.com/NVIDIA-AI-
IOT/deepstream_360_d_smart_parking_application/blob/master/analytics_se
rver_docker/nv-schema.json
```

Note:	The JSON examples in this section are meant to be illustrative. The properties defined in actual metadata, and their meanings, may differ. For specifics, see the metadata schema.

5.2 SENSOR

Every message contains one `sensor` element.

```
"sensor": {
    "id": "string",
    "type": "Camera/Puck",
    "location": {
        "lat": 45.99,
        "lon": 35.54,
        "alt": 79.03
    },
    "coordinate": {
        "x": 5.2,
        "y": 10.1,
        "z": 11.2
    },
    "description": "Entrance of Garage Right Lane"
}
```

5.3 VIDEO PATH

Every message for playback contains one `videoPath` element. This element specifies the URL of the playback video.

```
"videoPath": "<URL of playback Video>"
```

5.4 ANALYTICS MODULE

Every message contains one `analyticsModule` element.

```
"analyticsModule": {
    "id": "string",
    "description": "Vehicle Detection and global coordinate mapping",
    "source": "360-D",
```

```
    "version": "string"
}
```

5.5 PLACE

The properties of a `place` element depend on the type of object it represents. The examples below show `place` elements that represent an entrance, a parking spot, and an aisle.

Coordinates (properties `coordinate.x`, `.y`, and `.z`) are expressed in meters relative to the origin.

This `place` element represents an entrance to a parking structure.

```
"place": {
    "id": "string",
    "name": "garage-1”,
    "type": "building/garage",
    "location": {
        "lat": 37.37060687475246,
        "lon": -121.9672466762127,
        "alt": 0.00
    },
    "entrance": {
        "name": "entrance-1",
        "lane": "lane-1",
        "level": "P2",
        "coordinate": {
            "x": 1.0,
            "y": 2.0,
            "z": 3.0
        }
    }
}
```

This `place` element represents an entrance to a parking spot.

```
"place": {
    "id": "place-id",
    "name": "garage-1",
    "type": "building/garage",
    "location": {
        "lat": 37.37060687475246,
        "lon": -121.9672466762127,
        "alt": 0.00
    },
```

```
    "parkingSpot": {
        "id": "P2-PS-2",
        "type": " LEV/EV/CP/ADA",
        "level": "P2",
        "coordinate": {
            "x": 1.0,
            "y": 2.0,
            "z": 3.0
        }
    }
}
```

This `place` element represents an aisle/

```
"place": {
    "id": "place-id",
    "name": "garage-1",
    "type": "building/garage",
    "location": {
        "lat": 37.37060687475246,
        "lon": -121.9672466762127,
        "alt": 0.00
    },
    "aisle": {
        "id": "grid-id",
        "name": "Left Lane",
        "level": "P2",
        "coordinate": {
            "x": 1.0,
            "y": 2.0,
            "z": 3.0
        }
    }
}
```

5.6 OBJECT

An `object` element describes an object, such as a vehicle.

This `object` element represents a vehicle.

```
"object": {
    "id": "string",
    "vehicle": {
    "type": "sedan",
    "make": "Bugatti",
    "model": "M",
```

```
    "color": "blue",
    "confidence": 0.99,
    "license": "CGP93S",
    "licenseState": "CA"
    },
    "bbox": {
        "topleftx": 0.0,
        "toplefty": 0.0,
        "bottomrightx": 100.0,
        "bottomrighty": 200.0
    },
    "location": {
        "lat": 30.333,
        "lon": -40.555,
        "alt": 100.00,
        "orientation": 45.0,
        "direction": 225.0,
        "speed": 7.5
    },
    "coordinate": {
        "x": 5.2,
        "y": 10.1,
        "z": 11.2
    },
    "signature": [
        1.0,
        2.5,
        7.9
    ],
    "orientation": 45.0,
    "direction": 225.0,
    "speed": 7.5
}
```

5.7 EVENT

This `object` element represents an event.

```
"event": {
    "id": "event-id",
    "type": "entry"
}
```

The reference application generates these types of events:

- entry
- exit

- moving
- stopped
- parked
- empty
- reset

All types of events have the content shown above except for `reset`. A `reset` event has the following content:

```
"event": {
    "id": "event-id",
    "type": "reset",
    "source": "admin",
    "email": "admin@nvidia.com"
}
```

The `reset` event is not used in the reference application, but it can be used as a control message to reset the state of the application. In the case of the reference application, for example, it could be used to reset the parking spot count.

5.8 UNITS

The units used in metadata elements are:

- Distance: meters
- Speed: miles per hour
- Time: UTC

6.0 MULTI-CAMERA TRACKING

In a multi-camera system, objects move from the **field of view** (FoV) of one camera to another. For example, the reference application is intended to use about 150 cameras, each camera covering a few tens of meters of aisle. Cars invariably pass through multiple cameras' FoVs as they move through the garage. Such is the case in many other use cases involving multiple cameras, such as monitoring traffic in an intersection or freeway, or monitoring foot traffic in an airport.

Figure 24 shows a parking lot that could be a site for the reference application. Yellow dots represent cameras; the yellow circle around each dot represents the camera's field of view.

Figure 24. Example parking location with camera and fields of view

In a multi-camera system, it is vital to monitor and track objects across all cameras. Single-camera tracking of an object in a multi-camera system can lead to many ill effects, such as:

- Identity splits: A single object moving across multiple cameras is considered to be different cars.
- Multiple counting: A single object may be counted multiple times as it appears on different cameras.
- Broken paths: Travel paths may be limited to what one camera sees, making it impossible to obtain an object's entire path across an observation region.

Multi-camera tracking implementations may be divided into two broad categories based on real-time capabilities to track:

- **Batch trackers**: These accumulate data over a long interval and provides tracked identities at the end. The interval may vary based on the use case; usually it ranges from a few seconds to a few days.

- **Online trackers**: These trackers can track objects in real time. DeepStream, being a real-time inference platform, provides an online tracker.

  Online trackers' accuracy is typically low for two main reasons:

  - The data for the near-complete path of an object over long intervals is unavailable because the algorithms cannot go forward in time and peek at object's future location.
  - Most algorithms make instant decisions on tracked object identification, and cannot go back in time to correct identification errors.

This section describes DeepStream's multi-camera tracker, which enables users to track multiple objects that may be covered by multiple cameras in a streaming environment.

6.1 SYSTEM OVERVIEW

Figure 25 shows a high-level system diagram for the DeepStream multi-camera tracker. The multi-camera tracker interacts with two main components: the upstream DeepStream Perception Applications and the downstream Data Processing and Analytics engine.

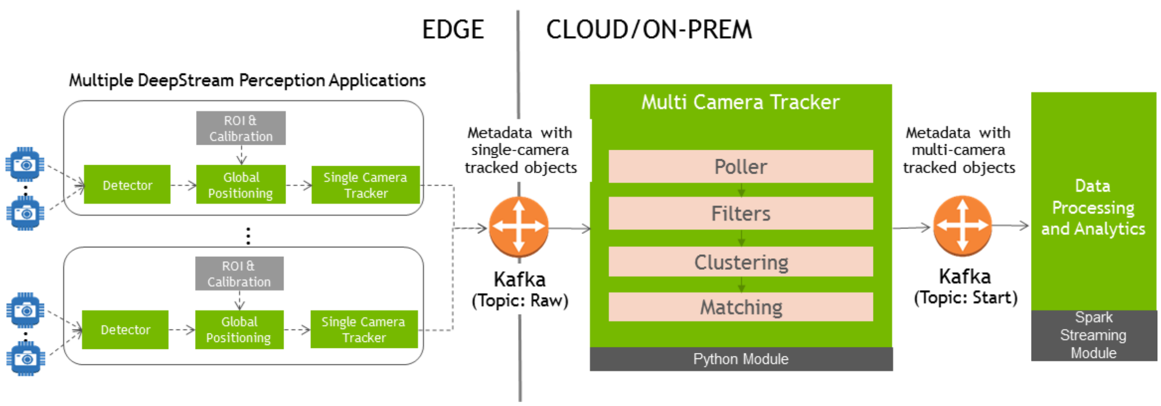

Figure 25. System overview of a multi-camera tracker

The **DeepStream Perception Applications** are the upstream components responsible for sending data to the multi-camera tracker. They are explained in detail in Perception Layer: 360-D Smart Parking Application.

In summary, these components process the camera streams and augments them with metadata containing the global coordinates of detected objects. They send the metadata to the tracker over Kafka on a given topic (say, topic "Raw"). They compute the global coordinates based on calibration data provided to the Perception Engine. Note that the tracker can track objects from multiple DeepStream perception applications that may run on different edge servers. All these applications may send the data to the same

Kafka topic, and the multi-camera tracker can perform tracking across all cameras that feed the different perception applications.

The multi-camera tracking application is a custom Python program which processes input from the Kafka stream, tracks multiple objects across multiple cameras, and emits the metadata by updating the unified ID that is assigned to each object by the tracker. It comprises several modules, described in the following sections.

6.1.1 Poller

The multi-camera system polls Kafka at specified intervals to collate all the data received in a given period. Usually the polling interval matches the frame rate or a small multiple of the frame rate. For example, reference application has a video feed at 2 frames per second (fps), and hence the polling interval is set to 0.5 seconds. All the data received within a given period is sorted by timestamp in the metadata. It is then passed to the next module.

6.1.2 Filters

In multi-camera systems, use cases often need to ignore certain regions of interest because objects in those regions are not, in fact, of interest. Use cases may also need to ignore certain regions because the number of false positives (e.g., detections of a car where no car exists) is very high there. This can happen, for example, because a region is subject to frequent lighting changes.

The DeepStream multi-camera tracker allows you to specify a list of polygon filters which excludes such regions. Each polygon filter specifies a set of convex polygons. The developers can then specify a list of such ignore polygons for each camera. The tracker ignores a detection from a given camera if it lies within one of the detection filters' polygons. The tracker passes only unignored detections to the next phase.

6.1.3 Clustering

The clustering module collates all the points detected in the given interval. A given object may be detected by multiple cameras at the same time. The clustering process groups detections that may represent the same object into one cluster.

Clustering is challenging in a multi-camera environment, especially in real time, because it must account for multiple scenarios. The DeepStream clustering module uses two main clustering schemes: **per-camera clustering** and **inter-camera clustering**

In a realistic system, it is very possible that the timestamps assigned to an object do not follow a strict periodicity. While video cameras can usually be configured to run at a constant frame rate, downstream applications may assign slightly different timestamps to frames that were captured simultaneously. For this reason the clustering module applies **per-camera clustering** first. It aggregates detections in multiple frames from the same camera that arrive in the same polling interval (0.5 second in the reference application). It **consolidates** detections for each camera and polling interval; that is, if multiple objects have the same single-camera tracker IDs and are sensed by the same camera, the clustering module retains the most recent detection and filters out the rest.

Inter-camera clustering identifies detections from multiple applications that may represent the same object. Clustering is achieved by first creating a distance matrix between all the points detected in the given frame, and then using hierarchical agglomerative clustering.

Measuring the distance is unfortunately not as simple as measuring a Euclidian distance (or any other sort of spatial distance) between consecutive points. Inter-camera clustering applies certain rules to specify the distance between two detections based on several observed constraints:

1. **Spatio-temporal distance**: The first criterion for identifying two detected objects as possibly the same is based on spatial distance. If two detections from different cameras are located very near each other, it is very possible that they represent the same object. Hence, by default we define the distance between the two points i and j (d[i,j]) as the spatial distance between i and j.
2. **Respect single-camera tracker IDs**: If the same camera has assigned different IDs to two objects in the same frame, then they cannot represent the same object. Hence, if two points i and j have different single-camera tracker IDs, the distance d[i,j] is set to a very large number (or to infinity).
3. **Maintaining multi-camera tracker ID over time**: Consider a scenario where an object is detected by cameras X and Y at time (t-1). Let X_1 and Y_1 be the IDs assigned to that object by the single-camera trackers of cameras X and Y at time (t-1), and let T_1 be the ID assigned to the same object by the multi-camera tracker at the same time. In the next frame *t*, if we get a point detected by camera X with ID X_1 and a point detected by camera Y with ID Y_1, then we assign both of these points the same multi-camera tracker ID, T_1. That is, both pf these points are assigned to the same cluster regardless of the spatial distance between them.

   Note that the single-camera tracker could have an error an accidentally assign the same ID to two different objects. However, this version of multi-camera tracking does not handle such cases.

   In order to assign previously chosen IDs, we maintain a list of single-camera tracker IDs (and corresponding camera IDs) that are assigned to each multi-camera tracker

ID. If two detections from different cameras (points i and j) have the same multi-camera tracking ID, the distance between them, d[i,j], is set to 0.

4. **Overlapping and conflicting cameras**: In many use cases it is necessary to combine only detections from certain pairs of cameras, called **overlapping** cameras. In other uses cases it is necessary to *not* combine detections from certain pairs of cameras, called **conflicting cameras**.

The multi-camera tracker generates a complete distance matrix based on the above rules. However, note that resulting distance function constructed from these rules is a heuristic, and may not be a formal mathematical metric. The tracker uses hierarchical agglomerative clustering with the "complete linkage method"; this ensures that as it clusters hierarchically, it takes the maximum distance between the branches of the hierarchy to create distances within the cluster. This ensures that two points whose distance is specified as infinite (because of rules 2 and 4 above) are not combined. It then cuts the resulting dendrogram at a specified distance threshold, specified in the configuration file by the key `CLUSTER_DIST_THRESH_IN_M`.

The tracker picks one element in each cluster as a **representative**, that is a representative element from the cluster. It also updates the list of IDs that are clustered together at time t.

6.1.4 Matching

This step assigns an ID to the possibly similar object found by the clustering module at frame *t*. It does so by comparing how much the representatives have moved from time (*t*-1) to time *t*.

Recall that in each frame the multi-camera tracker gathers objects into clusters and picks a cluster representative. Also, the matching step at frame (*t*-1) assigns a multi-camera tracker ID to the cluster representatives at time (t-1). Hence s multi-camera tracking ID has already been issued to objects detected at (t-1). The matching module compares the representatives at frame (*t*-1) to those at frame *t* and percolates the ID issued to the (t-1) objects to the best matching representatives at frame *t*.

This action reduces to the minimum cost bipartite matching problem, where one partition of points (representatives at frame (*t*-1)) are matched one-to-one with another partition of points (representatives at frame *t*). This problem can usually be solved in polynomial time by algorithms such as the Hungarian Assignment Algorithm.

Two borderline cases need to be handled:

- **Matching distance is too large**: If the spatial distance is too large, the matching module voids the match. In the reference application, for example, if the spatial distance between matched representatives A and B is 100 meters, they must not be matched; a vehicle would have to move 226 miles per hour to travel 100 meters in 0.5 second, which is unrealistic.

Note that the definition of "too large a distance" is use case dependent. It can be configured by the key `MATCH_MAX_DIST_IN_M`. Any matching that is greater than this threshold is considered to be unmatched.

- **Number of points is different**: If the partitions at time (*t*-1) and *t* contain different numbers of points, the matching module voids the match.

In either case, a point that is unmatched in the partition corresponding to frame *t* is considered as new detection, and is issued a new ID. In the reference application, this might be due to a vehicle entering the parking area for the first time.

This description ignores points that are unmatched in the partition corresponding to frame (*t*-1). In the reference application these points might represent vehicles that have exited the garage.

At the end of the matching process the matching module transmits all of the representative points for frame *t* to the downstream Kafka topic (e.g., topic `start`).

6.2 CONFIGURATION

The multi-camera tracker uses two configuration files.

The first file is the **stream configuration file**, which specifies the configuration needed for input/output of tracker. The stream configuration file looks like this:

```
{
    "profileTime": false,
    "msgBrokerConfig": {
        "inputKafkaServerUrl": "kafka:9092",
        "inputKafkaTopic": "metromind-raw",
        "outputKafkaServerUrl": "kafka:9092",
        "outputKafkaTopic": "metromind-start"
    }
}
```

The second file is the **tracker configuration file**, which specifies the configuration parameters for the tracker. This configuration file has the following elements

- `overlapping_camera_ids`: Specifies cameras that have overlapping coverage. If this dictionary contains one or more keys, the tracker merges only detections from overlapping cameras. It Detections from cameras that do not overlap are always kept separate.

- `conflict_cameras_adj_list`: Specifies cameras that have conflicting coverage. The tracker never merges detections from conflicting cameras.
- `MAP_INFO`: This key specifies road network information. It represents a road network graph as a set of lines. Each line has a set of points defined by (longitude, latitude) pairs. If `SNAP_POINTS_TO_GRAPH` is `true`, the multi-camera tracker uses the road network graph to snap detected points to the road network.
- `IGNORE_DETECTION_DICT_MOVING`: This dictionary value defines polygons inside which detections are to be ignored (not processed). For each camera, the configuration file can specify a list of polygons where detections have to be ignored.

6.3 RUNNING THE TRACKER

This section gives instructions for running the multi-camera tracker.

To run the tracker

- Enter the commands:

```
export PYTHONPATH="<tracker_dir>/code"
python3 usecasecode/360d/stream_track.py --sconfig=`<s_config>` --
config=`<t_config>`
```

Where:

  - `<tracker_dir>` is the directory cloned from Git (e.g. `/home/user/git/tracker`).
  - `<s_config>` is the pathname of the stream configuration file.
  - `<t_config>` is the pathname of the tracker configuration file.

The tracker displays the following messages as it starts:

```
Starting MC-Streaming app with following args:
consumer kafka server=<c_kafka_URL>
consumer kafka topic=<consumer topic>
producer kafka server=<producer kafka bootstrap url>
producer kafka topic=<producer topic>
Time profile=<True or False based on time profiling is enabled on
stream configuration file>
MC Tracker Configuration File=<stream config file>
...
```

Where:

- `<c_URL>` is the consumer Kafka bootstrap URL.
- `<c_topic>` is the consumer Kafka topic.

- <p_URL> is the producer Kafka bootstrap URL.
- <p_topic> is the producer Kafka topic.
- <b_t> is true if time profiling is enabled in the stream configuration file, or false otherwise.

For further details, refer to:

```
https://github.com/NVIDIA-AI-IOT/deepstream_360_d_smart_parking_application/tree/master/tracker
```

7.0 STATEFUL STREAM PROCESSING

The DeepStream implementation of stateful stream processing is based on Apache Spark structured streaming, a fault tolerant streaming engine. As maintaining trajectories requires advanced stateful operations, it uses the `mapGroupsWithState()` operation.

The API allows maintaining user-defined per-group state between triggers for a streaming dataframe. The timeout is set to clean up any state which has not seen activity for a configurable period of time.

Note:	The reference application uses processing time based on clock time, so it may be affected by changes in the system clock, such as clock skew.

DeepStream maintains a trajectory for each vehicle during the period when it is seen in the aisle area. It computes trajectories based on “moving” events. The perception layer determines the location of a vehicle in an aisle or parking spot by use of regions of interest (RoIs). Maintaining trajectories of vehicles gives DeepStream the ability to compute many vehicle properties like speed, time of stay, and whether a vehicle is stalled in a particular location.

DeepStream discards a vehicle’s trajectory if no events are seen with respect to the vehicle for a configurable length of time. For example, if a vehicle is parked after moving through the aisle, the perception graph coes not send any more moving events, and the trajectory is cleaned up after a configurable period of time.

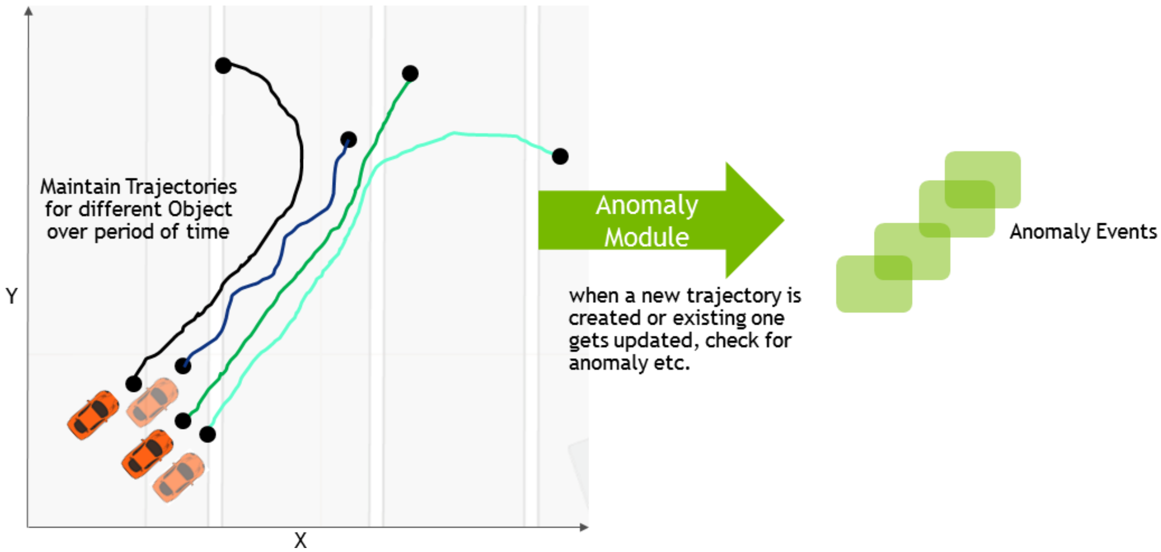

Figure 26. Stateful stream processing in the reference application

The reference application provides stateful stream processing for two kinds of anomaly:

- **Car understay**: A car stayed in the garage less than a configurable length of time.
- **Car stalled**: A car is stalled in the aisle, i.e. does not move for a configurable length of time.

7.1 INSTALLATION

Installing stateful stream processing consists of three steps: compiling and installing the project, starting the Spark streaming job, and starting the Spark batch job.

To compile and install the project

- Enter the command:

```
sudo apt-get update
sudo apt-get install default-jdk
sudo apt-get install maven
mvn clean install -Pjar-with-dependencies
```

This command generates the required jars. Note the location of the new jars created:

```
 [INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ stream-360 ---
 [INFO] Building jar: /Users/home/git/stream-360/target/stream-360-
1.0.jar
 [INFO]
 [INFO] --- maven-assembly-plugin:2.5.5:single (make-assembly) @
stream-360 ---
```

```
 [INFO] Building jar: /Users/home/git/stream-360/target/stream-360-1.0-jar-with-dependencies.jar
```

To start the Spark streaming job

1. Install Apache Spark, or use an existing cluster if `spark-master` is running in a Docker container:

   ```
   docker exec -it spark-master /bin/bash
   ```

2. Enter the command:

   ```
   ./bin/spark-submit  --class com.nvidia.ds.stream.StreamProcessor --master spark://master:7077 --executor-memory 4G --total-executor-cores 4 /tmp/data/stream-360-1.0-jar-with-dependencies.jar
   ```

This job does following things:

- Manages the state of parking garage
- Detects car understay anomalies
- Detects car stalled anomalies
- Computes flowrate

To start the Spark batch job

- Enter the command:

  ```
  ./bin/spark-submit  --class com.nvidia.ds.batch.BatchAnomaly --master spark://master:7077 --executor-memory 4G --total-executor-cores 4 /tmp/data/stream-360-1.0-jar-with-dependencies.jar
  ```

This job detects car overstay anomalies.

8.0 DATABASE SCHEMA

The reference application's state is stored in Cassandra. All metadata is stored in Elasticsearch for search analytics. The following section shows how state is managed using Cassandra tables.

The following sections describe the reference application's key Cassandra tables.

aisle

- **Keys**:
  `PRIMARY KEY (messageid, timestamp)) WITH CLUSTERING ORDER BY (timestamp DESC)`
- **Description**: Stores events taking place in a garage aisle. The primary key is `messageid`, which is changed to `<garageid>-<garageLevel>` in the streaming pipeline. The combination of `messageid` with `timestamp` (a clustering key) enables the user to query for unique objects in the aisle.

parkingspotstate

- **Keys**:
  `PRIMARY KEY ((garageid, level), spotid)) WITH CLUSTERING ORDER BY (spotid ASC)`
- **Description**: Stores the state of the garage. This table is used when the application is running in `live` mode. Whenever a parking event occurs the record for the affected parking spot is updated.

parkingspotplayback

- **Keys**:
  `PRIMARY KEY ((garageid, level, sensortype, spotid), timestamp)) WITH CLUSTERING ORDER BY (timestamp DESC)`
- **Description**: Used to retrieve the parking spot state at a given time. This table is mainly used when the application is running in `playback` mode.

parkingspotdelta

- **Keys**:
  `PRIMARY KEY ((garageid, level, sensortype), timestamp, spotid)) WITH CLUSTERING ORDER BY (timestamp DESC, spotid ASC)`
- **Description**: Used to obtain all events related to a parking spot that occur in each time interval. The length of a time interval is configurable' the default is 500 msec.

9.0 API

The API provides endpoints to query the state of the garage and also the events and anomalies that are occurring in garage. The API endpoints that concern events and anomalies can also be used for searching. Events and anomalies are retrieved from Elasticsearch whereas the state and Key Performance Indicators (KPIs) of the garage are retrieved from Cassandra.

The API is not limited to statistics of the garage. It also supplies configuration details to the user interface. It has a Websocket which reads data from Cassandra to give live updates of the garage.

9.1 GETTING STARTED

Before you begin you must have installed on your local system:

- Node.js version 8.x or later
- The Elasticsearch database
- The Cassandra database

9.1.1 Environment Variables

You must export the following environment variables if you are running the API outside Docker:

- `IP_ADDRESS`: IP Address of the host system. If running locally, set it to `localhost`.
- `NODE_PORT`: The port where server is to listen for requests.

9.1.2 Installation

Install Node.js, Cassandra, Elasticsearch as they are prerequisites for this application.

Go to the `apis` directory:

```
cd apis
```

The file `package.json` provides a list of libraries which this application requires. Use npm install to install these libraries.

To start the server, enter the command:

```
npm start
```

9.2 ENDPOINTS EXPOSED

The API exposes the following endpoints.

9.2.1 Configuration-Related Endpoints

User Interface Configuration

Route: `/ui-config`

Params: N/A

Description: Sends configuration to the user interface. This helps the user interface render the markers for the garage and the details for endpoints and the Websocket URL.

Response: If the system is live:

```
{
    "home": {
        "name": "Home",
        "username_api": "",
        "googleMap": {
            "defaultCenter": {
                "lat": 37.2667081,
                "lng": -121.9852038
            },
            "defaultZoom": 14,
            "maxZoom": 21,
            "minZoom": 10,
```

```
            "mapTypeControl": true,
            "mapTypeId": "roadmap"
        },
        "locations": [
            {
                "name": "garage",
                "lat": 37.287535,
                "lng": -121.98473
            }
        ]
    },
    "garage": {
        "name": "Garage",
        "defaults": {
            "level": "P1"
        },
        "bounds": {
            "north": 37.2886489370708,
            "south": 37.2864695830171,
            "east": -121.983629765596,
            "west": -121.986218361030
        },
        "googleMap": {
            "defaultCenter": {
                "lat": 37.287535,
                "lng": -121.98473
            },
            "defaultZoom": 19,
            "maxZoom": 21,
            "minZoom": 10,
            "mapTypeControl": true,
            "mapTypeId": "roadmap"
        },
        "groundOverlay": {
            "p1GroundImage": "assets/X-StrpP1_simpleConverted.png",
            "p1Bounds": {
                "north": 37.2881998,
                "south": 37.2863798,
                "east": -121.9838699,
                "west": -121.9859025
            }
        },
        "isLive": true,
        "live": {
            "webSocket": {
                "url": "",
                "startTimestamp": "",
                "garageId": "endeavor",
                "garageLevel": "P1",
                "dialogAutoCloseSeconds": 5
            },
            "apis": {
```

```
                "baseurl": "",
                "alerts": "/es/alerts",
                "events": "/es/events",
                "kpi": "/stats/endeavor",
                "startTimestamp": "",
                "alertEventRefreshIntervalSeconds": 5,
                "uiDelaySeconds": 30,
                "alertEventListLength": 20
            }
        }
    }
}
```

If the application is a playback system then `isLive` is set to `false` and the `playback` attribute of the configuration is sent instead of the `live` attribute.

9.2.2 Garage-Related Endpoints

Stats

Route: `/stats/:garageId`

Params: q (contains a timestamp range for `stats` query)

Description: Gets the parking spot statistics of the `garageId` in the URL; also gets the flowrate of the garage.

Response:

```
{
    "id": <Id of garage>,
    "Free Spots": <Number of available spots>,
    "Occupied Spots": <Number of occupied spots>,
    "Entry Flow Rate": <Entry flowrate of garage>,
    "Exit Flow Rate": <Exit flowrate of garage>
}
```

Alerts

Route: `/es/alerts`

Params: q (contains a timestamp range for the `alerts` query; may also contain search tokens)

Description: Used to list all of the anomalies in the garage.

Response: Contains the result object from Elasticsearch. The result object contains the attributes of the car object exhibiting anomalous behavior.

Events

Route: `/es/events`

Params: q (contains a timestamp range for the `events` query; may also contain search tokens)

Description: Used to list all of the events in the garage. The endpoint compresses the results so that multiple types of events can be displayed during a time interval, rather than just viewing the moving event of a single car object.

Response: Contains the result object from Elasticsearch. The result object contains the attributes of the car object whose events are being detected.

Events-Deprecated

Route: `/es/events-deprecated`

Params: q (contains a timestamp range for the `events` query; may also contain search tokens)

Description: Used to list all the events in garage. This endpoint is deprecated, as it does not compress events.

Response: Contains the result object from Elasticsearch. The result object contains the attributes of the car object whose events are being detected.

9.3 WEBSOCKET EXPOSED

The WebSocket is used for live updates of Garage.

Route: /

Initial Message: The initial message sent by the client must be in JSON format, with the following attributes

```
{
    "garageId": <Id of Garage>,
    "garageLevel": <Level of Garage>,
    "startTimestamp": <The startTimestamp of the websocket. It should be in the following format: YYYY-MM-DDTHH:MM:SS.fffZ>
}
```

Description: The WebSocket provides parking spot and aisle related updates to the client.

Response: An array of car objects. A sample car object has the following attributes

```
{
    "timestamp": <Timestamp of Event>,
    "color": <Color of Object>,
    "garageLevel": <Level of garage>,
    "id": <Id of object>,
    "licensePlate": <License Plate>,
    "licenseState": <License State>,
    "orientation": <Orientation of object>,
    "parkingSpot": <Parking Spot Id. It will be null for moving cars.>,
    "sensorType": <Type of sensor>,
    "state": <State of the car. Possible values are
moving,parked,empty>,
    "eventType": <Type of event>,
    "removed": <A flag which indicates if the car needs to be
removed/retired from UI>,
    "type": <Type of vehicle>,
    "x": <X coordinate of object>,
    "y": <Y coordinate of object>,
    "lat": <Latitude of object>,
    "lon": <Longitude of object>
}
```

9.4 CONFIGURATION FILE

The configuration file `config.json` is stored in the `config` directory. It defines all of the settings used by the user interface and the back end.

The user interface part of the configuration file is explained in the user interface's README file at:

https://github.com/NVIDIA-AI-IOT/deepstream_360_d_smart_parking_application/blob/master/ui/README.md

The back end configuration settings are defined at:

https://github.com/NVIDIA-AI-IOT/deepstream_360_d_smart_parking_application/blob/master/apis/config/config.json

They like this:

```
{
    "cassandraHosts": <An array of Cassandra hosts>,
    "cassandraKeyspace": <Name of the Cassandra keyspace being used>,
```

```
    "esHost": <Elasticsearch host>,
    "esPort": <Elasticsearch port>,
    "esAnomalyIndex": <Anomaly index in Elasticsearch>,
    "esEventsIndex": <Events index in Elasticsearch>,
    "eventCompressionSize":<Compression factor for moving events of a car>,
    "anomalyEventQuerySize": <Size of anomaly, events that will be returned by Elasticsearch>,
    "eventApiQueryResultSize": <Size of result that will be returned by the elasticsearch for events api>,
    "parkingSpotConfigFile": <Parking Spot configuration file which lists all the spots available in a garage>,
    "sensorType": <Type of sensor being used>,
    "garageLevel": <Level of the garage>,
    "webSocketSendPeriodInMs": <Interval in msec after which Websocket should send messages>,
    "carRemovalPeriodInMs": <Interval in msec after which non-moving cars in aisle of garage should be retired>,
    "originLat": <Latitude of center of garage>,
    "originLon": <Longitude of center of garage>
}
```

10.0 USER INTERFACE

The user interface for the 360-D application is built using React.js. It is used to visualize the state of a garage along with events and anomalies occurring in the garage. The user interface can run in two modes: **live** and **playback**. Live mode monitors a garage in real time. Playback mode is used for demo purpose, i.e. to visualize pre-recorded data.

10.1 USER INTERFACE COMPONENTS

The component hierarchy of the user interface is shown in the following diagram:

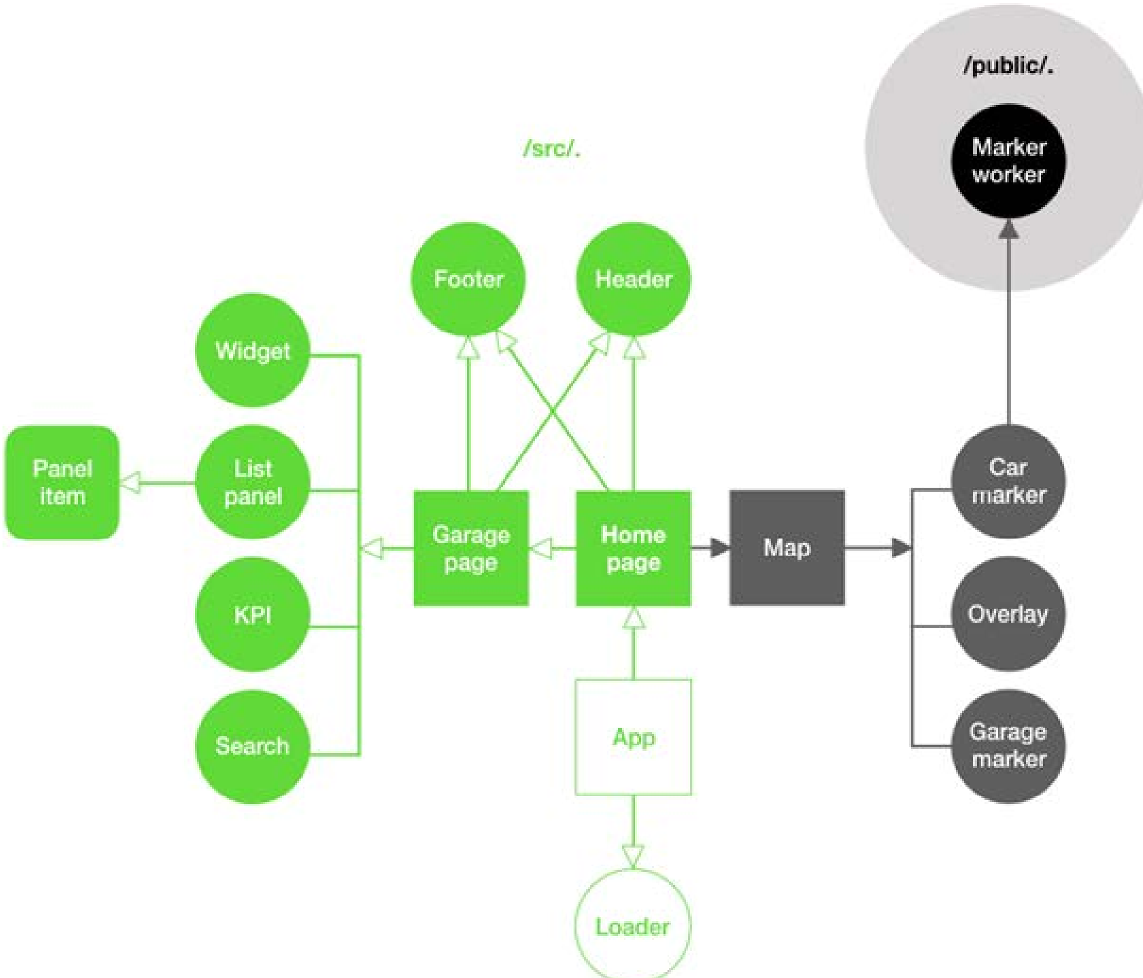

Figure 27. The React component hierarchy

The components of the application are:

1. **App**: `App.js` requests the configuration from the API and passes it down as properties. It also navigates to another route for the `home/garage` page.
2. **Home page**: `HomeMap.js` renders all garage markers, and Map.js, which contains a Google Maps component. The selected garage is rendered; its state is passed up to `HomeMap.js` and then down to `Map.js`. When the user clicks a location marker, `HomeMap.js` zooms the map in to the corresponding garage. It also handles actions and events executed by the user, i.e. playing videos, clicking on license plates to display an information dialog box, and switching the page between different garages or garage levels.

3. **Garage page**: `SmartGaragePage.js` displays the main page of the garage. It has five main components:
    - **List window**: Displays events and anomalies of detected cars.
    - **KPI information** of Garage: Displays the current state of parking spots and the entry/exit flowrate.
    - Ground Overlay of Garage
    - Header and Footer
    - Search bar
4. **Map**: `Map.js` contains three main components:
    - **GoogleMap**: A background for garage the overlay.
    - **GroundOverlay**: Overlay image of the garage.
    - **CarMarkers**: The dots that are used to visualize parked and moving cars.

Note:	**The center and bounds of the map change when the zoom level is changed.**

5. **Loader**: `Loader.css` determines the appearance of the loader for the browser.
6. **Footer**: `Footer.js` contains four elements:
    - Name of the application
    - Version of the application
    - User's information
    - Notes/Disclaimer
7. **Header**: `Header.js` contains five elements:
    - Company logo
    - Application name
    - Search bar with dropdown calendar
    - Question mark icon
    - Hidden exclamation icon which appears when the user selects invalid time bounds in search
8. **Widget**: `Widget.js` adjusts the size of a panel which displays video, or of the events/anomalies list, to fit the screen size.
9. **List panel**: `ListPanel.js` displays the latest events and anomalies of a car. You can configure the number of messages displayed by changing the `alertEventListLength` property. By default, the list panel refreshes its displays every `alertEventRefreshIntervalSeconds` seconds by sending an Ajax query to obtain the latest events and anomalies of cars from the back end. It updates the list panel with search results after the search query is triggered. If the search has no calendar, the list panel refreshes the messages every

`alertEventRefreshIntervalSeconds` seconds; if the user uses the calendar to search, only messages with tokens within the time bounds of search are shown.

10. **KPI**: `Capacity.js` sends an Ajax query to obtain the latest KPIs of the garage every `alertEventRefreshIntervalSeconds` seconds. KPIs include available and occupied parking spots and entry/exit flow.
11. **Search**: `Search.js` creates the search query and passes it up to `SmartGaragePage.js`. Each query includes a search token, time bounds, and a hash location. If `isCalendar` is set to `true`, time bounds are set using the calendar. If `isTimeValid` is set to `true`, the selected time bounds are valid.
12. **Car marker**: `Map/CarMarkers.js` manages and updates each car marker, including its status and style.
13. **Overlay**: `Map/GroundOverlay.js` is a functional component which imports the garage's ground image and sets the bounds of the ground overlay.
14. **Garage marker**: `Map/LocationMarker.js` returns a clickable icon on the Google Map which represents the garage's location. The user can navigate to the garage page by clicking the marker. The marker's size changes as the zoom level changes.
15. **Panel item**: `PanelItem/Item.js` is a functional component which returns key information retrieved from each message of a detected car.
16. **Marker worker**: `public/MarkersWorker.js` uses web workers to manage and update the status of car objects, which are sent through WebSocket.

Figure 28 shows how the components are placed on the Garage Page.

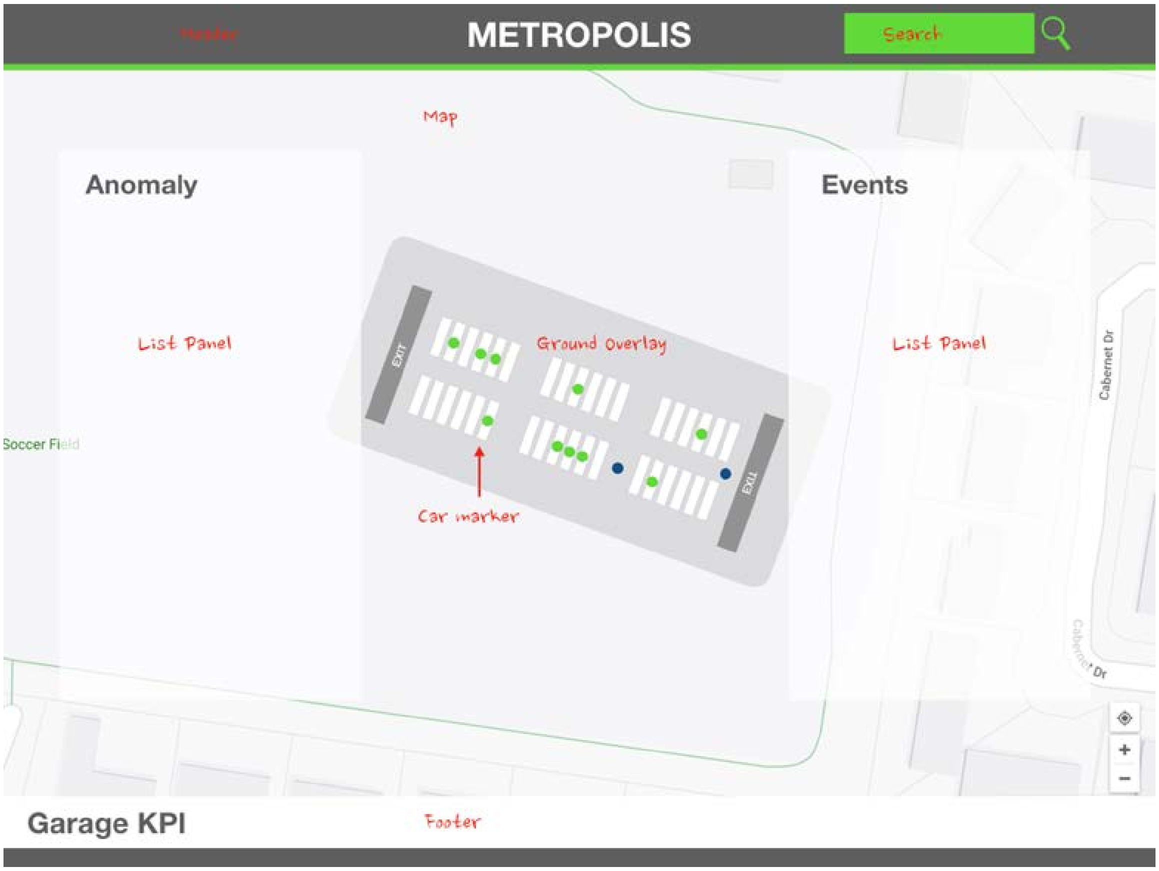

Figure 28. How components are placed on the Garage page

Figure 29 shows the home page with a garage marker. Although the image shows a single garage marker, the home page can contain multiple garage markers. The user can navigate to any of the garage by clicking its marker.

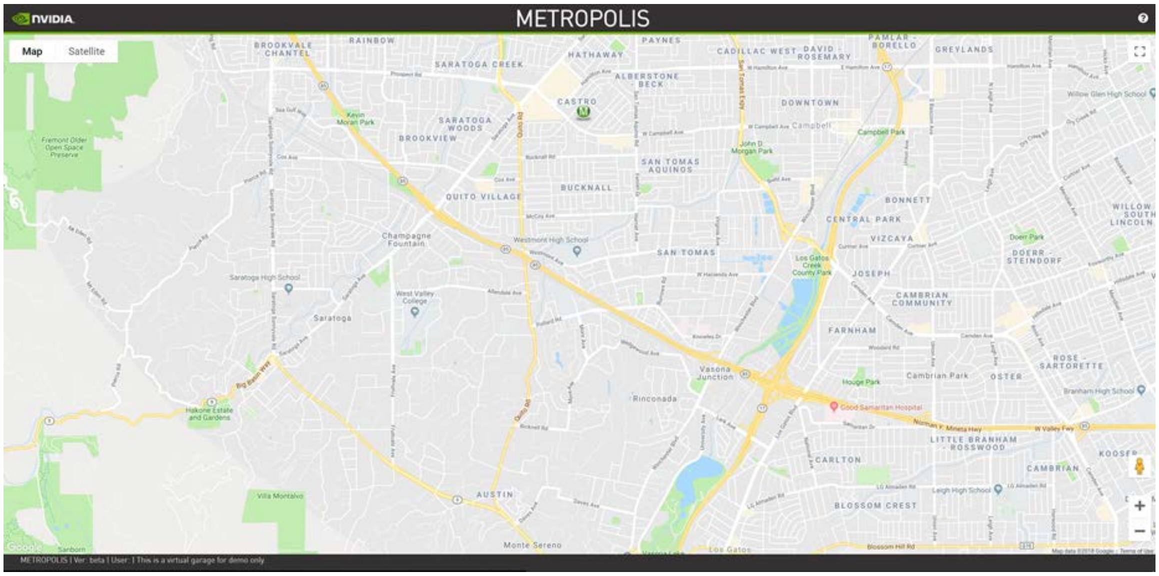

Figure 29. Home page with a garage marker

Once the user clicks a garage marker, the map zooms in to the garage location and renders the garage overlay along with other components.

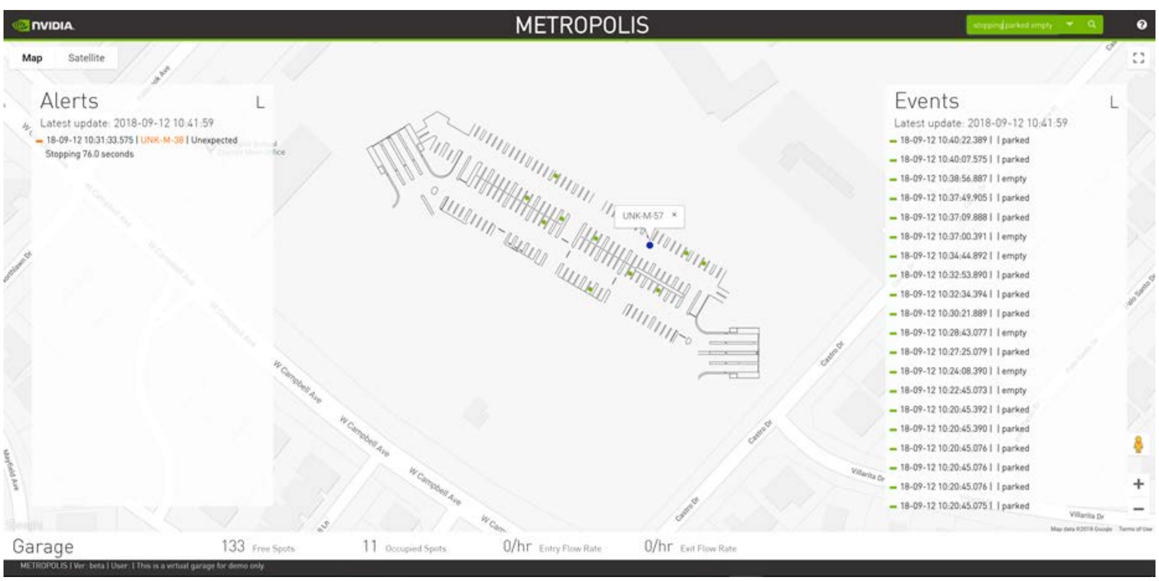

Figure 30. The garage overlay display

10.2 GETTING STARTED

This section explains how to get started using the reference application.

10.2.1 Dependencies

You must have installed on your local system:

- A recent version of Node.js
- The 360-D API

The API server must be running.

10.2.2 Environment Variables

Export the following environment variables if you are running the user interface outside Docker:

- `REACT_APP_BACKEND_IP_ADDRESS`: the IP address of the API server
- `REACT_APP_BACKEND_PORT`: the port where the server listens for requests
- `REACT_APP_GOOGLE_MAP_API_KEY`: The Google Maps API key

The Get API Key page gives instructions for getting a Google Maps API key:

```
https://developers.google.com/maps/documentation/javascript/get-api-key
```

Note:	**Custom environment variable names must begin with `REACT_APP_`, or they will be ignored (except for `NODE_ENV`).**

10.2.3 Installation

Install Node.js and the API described in section 9.0, "API," which are prerequisites for this application.

Go to the `ui` directory:

```
cd ui
```

`package.json` has a list of libraries which are required for this application. Enter this command to install the libraries:

```
npm install
```

To run this application in development mode, enter the command:

```
npm start
```

10.2.4 Deployment

Before deploying a DeepStream application to any web host, enter this command to create an optimized build for the production environment:

```
npm run-script build
```

The build contains minified JavaScript files, which can be found in the build directory. This build can be deployed to any web host using nginx by following the steps described at:

```
https://medium.com/@timmykko/deploying-create-react-app-with-nginx-and-ubuntu-e6fe83c5e9e7
```

10.2.5 Configuration

By maintaining a JSON configuration file in the back end, the application allows the users to customize their own garage. The configurable features are:

10.2.5.1 Home Page

To set the center of your map to your garage's location, edit the `lat` and `lng` properties:

```
"home": {
    "name": "Home",
    "username_api": "",
    "googleMap": {
        "defaultCenter": {
            "lat": 37.2667081,
            "lng": -121.9852038
        },
        "defaultZoom": 14,
        "maxZoom": 21,
        "minZoom": 10,
        "mapTypeControl": true,
        "mapTypeId": "roadmap"
    },
    "locations": [
```

```
        {
            "name": "garage",
            "lat": 37.287535,
            "lng": -121.98473
        }
    ]
},
. . .
```

10.2.5.2 Garage Page

Set the `bounds` properties to specify the bounds of the map shown on the entry page.

Set the `googleMap` properties to customize the Google Maps features on the background of the garage.

```
"garage": {
    "name": "Garage",
    "defaults": {
        "level": "P1",
    },
    "bounds": {
        "north": 37.2886489370708,
        "south": 37.2864695830171,
        "east": -121.983629765596,
        "west": -121.986218361030
    },
    "googleMap": {
        "defaultCenter": {
            "lat": 37.287535,
            "lng": -121.98473
        },
        "defaultZoom": 19,
        "maxZoom": 21,
        "minZoom": 10,
        "mapTypeControl": true,
        "mapTypeId": "roadmap"
    },
    . . .
```

`groundOverlay` defines a top-down perspective image of the garage. You can replace the image in the `/src/asset` directory. Set the bounds of the image to match the `lat` and `lng` of the garage on the Google Map.

```
    "groundOverlay": {
        "p1GroundImage": "assets/X-StrpP1_simpleConverted.png",
        "p1Bounds": {
            "north": 37.2881998,
            "south": 37.2863798,
```

```
            "east": -121.9838699,
            "west": -121.9859025
        }
    },
    . . .
```

`isLive` indicates the mode of the video source. If it is `true`, live video is used as the data source (live mode); otherwise, a pre-recorded video is used (playback mode).

```
    "isLive": false,
    . . .
```

10.2.5.3 Live versus Playback

For both live and playback modes, Websocket and the API are configurable. The following properties are used to configure the API:

- `alertEventRefreshIntervalSeconds`: Query interval. The API sends an AJAX query at intervals this number of seconds long to get events and anomalies data. The default value is 5 (a query is sent every 5 seconds).
- `alertEventListLength`: Maximum number of items shown on the list window. The default value is 20.
- `dialogAutoCloseSeconds:` Pop-up time interval of the dialog box which shows parking information of each car. The default value is 5 (5 seconds). The dialog box can also be triggered by clicking on a car marker.

The properties that differentiate live and playback modes are:

- `startTimestamp`: Required only in playback mode; the time at which playback starts.
- `uiDelaySeconds:` Introduces a delay to the user interface. This is required because the parking Spot data generated and sent by DeepStream is usually delayed by a few seconds. By delaying the user interface the application shows the accurate representation of the garage at that moment of time.
- `autoRefreshIntervalMinutes`: In playback mode, makes the application send queries in a timed loop., The property determines the interval of the loop. Set it based on the pre-recorded video’s length.

```
"live": {
    "webSocket": {
        "url": "",
        "startTimestamp": "",
        "garageId": "endeavor",
        "garageLevel": "P1",
        "dialogAutoCloseSeconds": 5
    },
```

```
    "apis": {
        "baseurl": "",
        "alerts": "/es/alerts",
        "events": "/es/events",
        "kpi": "/stats/endeavor",
        "startTimestamp": "",
        "alertEventRefreshIntervalSeconds": 5,
        "uiDelaySeconds": 30,
        "alertEventListLength": 20
    }
},
"playback": {
    "webSocket": {
        "url": "",
        "startTimestamp": "2018-08-30T21:49:48.500Z",
        "garageId": "endeavor",
        "garageLevel": "P1",
        "dialogAutoCloseSeconds": 5
    },
    "apis": {
        "baseurl": "",
        "alerts": "/es/alerts",
        "events": "/es/events",
        "kpi": "/stats/endeavor",
        "startTimestamp": "2018-08-30T21:49:48.500Z",
        "alertEventRefreshIntervalSeconds": 5,
        "autoRefreshIntervalMinutes": 30,
        "uiDelaySeconds": 20,
        "alertEventListLength": 20
    }
},
. . .
```

11.0 DASHBOARD

Apart from the custom user interface, you can easily build a Kibana dashboard, which is a collection of visualizations and searches. The dashboard comprises:

- Occupancy and available spots at a given time
- Entry/exit traffic pattern over the last 24 hours
- Anomaly chart over the last 24 hours
- Heatmap entry/exit showing the rush hour periods

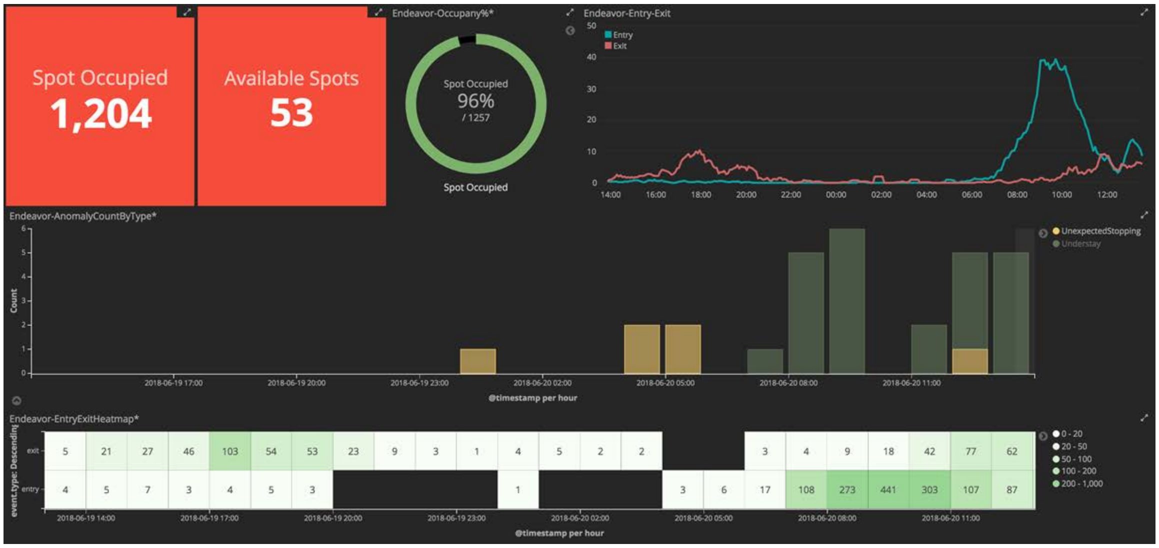

Figure 31. Analytics dashboard built with Kibana

12.0 APPENDIX A: 360-D SMART PARKING APPLICATION CONFIGURATION DETAILS

The NVIDIA® DeepStream 360-D smart parking application uses one of the sample configuration files from the `samples/configs/deepstream-360d-app` directory in the `DeepStream360d` package to:

- Enable or disable components
- Change the properties or behavior of components
- Customize other application configuration properties that are unrelated to the pipeline and its components

The configuration file uses a key file format based on the freedesktop specifications at:

```
https://specifications.freedesktop.org/desktop-entry-spec/latest
```

12.1 ABOUT CONFIGURATION GROUPS

The application configuration is divided into groups of properties for specific components. The following table describes the configuration groups.

Group	Purpose
application	Properties that are not related to a specific application component.
tiled-display	Configure the application's tiled display.
source	Specify source properties. The application can have multiple sources. Groups for muliple sources are named `[source0]`, `[source1]`, etc.
streammux	Specify the properties and modify the behavior of `Gst-nvstreammux`.

Group	Purpose
primary-gie	Specify the properties and modify the behavior of the primary GPU Inteference Engine (GIE).
tracker	Specify the properties and modify the behavior of `Gst-nvtracker`.
osd	Specify the properties and modify the on-screen display (OSD) component which overlays text and rectangles on the frame.
sink	Specify the properties and modify the behavior of the sink components that represent outputs such as displays or files for rendering, encoding, and file saving. The pipeline can contain multiple sinks. Groups must be named `[sink0]`, `[sink1]`, etc.
tests	An experimental group for diagnostics and debugging.
spot	Specify properties for parking spot detection component.
aisle	Specify properties for aisle area tracking.
message-broker	Specify properties for the components which generate payload data and transmit it to the Cloud.
Dewarper	Specify the properties and modify the behavior of `Gst-nvdewarper`.

12.2 APPLICATION GROUP

This section describes the properties in the `application` group.

Property	Meaning	Type and Range	Example and Notes
enable-perf-measurement	Indicates whether application performance measurement is enabled.	Boolean (0 or 1)	enable-perf-measurement=1
perf-measurement-interval-sec	Interval at which performance metrics are sampled and printed, in seconds.	Integer >0	perf-measurement-interval-sec=10
gie-kitti-output-dir	Pathname of an existing directory where the application writes output in a modified KITTI metadata format.	String	gie-kitti-output-dir= /home/ubuntu/kitti_data/
enable_bboxfilter	Indicates whether the boundary box is enabled.	Boolean	enable_bboxfilter=1
select-rtp-protocol	Selects RTP protocol for streaming input data from camera. Possible values are: 7 = All (UDP/TCP) 4 = Only TCP	Integer 4 or 7	select-rtp-protocol=4

12.3 TILED-DISPLAY GROUP

This section describes the properties in the `tiled-display` group.

Property	Meaning	Type and Range	Example and Notes
enable	Indicates whether tiled display is enabled.	Boolean	enable=1
rows	Number of rows in the tiled 2D array Integer, > 0 rows=5	Integer >0	
columns	Number of columns in the tiled 2D array.	Integer >0	columns=6
width	Width of the tiled 2D array, in pixels.	Integer >0	width=1280
height	Height of the tiled 2D array, in pixels.	Integer >0	height=720
gpu-id	Device ID of the GPU to use if multiple GPUs are available.	Integer ≥0	gpu-id=0
cuda-memory-type	Type of CUDA memory to allocate for output buffers. Possible values are: 0 = cuda-pinned-mem: host/pinned memory allocated with `cudaMallocHost()`. 1 = cuda-device-mem: device memory allocated with `cudaMalloc()`. 2 = cuda-unified-mem: unified memory allocated with `cudaMallocManaged()`.	Integer 0, 1, or 2	cuda-memory-type=1

12.4 SPOT GROUP

This section describes the properties in the `spot` group.

Property	Meaning	Type and Range	Example and Notes
enable	Indicates whether spot is enabled.	Boolean	enable=1
result-threshold	Number of seconds a parking spot status must endure to be considered persistent.	Integer ≥0	result-threshold=10

Property	Meaning	Type and Range	Example and Notes
component-id	Unique ID of this component (plugin); attached to event metadata.	Integer 0–4,294,967,295	component-id=1
calibration-file	Pathname of file containing data related to parking spots.	String	calibration-file= ./csv_files/nvspot_2M.csv

12.5 AISLE GROUP

This section describes the properties in the `aisle` group.

Property	Meaning	Type and Range	Example and Notes
enable	Indicates whether aisle is enabled.	Boolean	enable=1
component-id	Unique ID of this component (plugin); attached to event metadata.	Integer 0–4,294,967,295	component-id=1
calibration-file	Pathname of file containing data related to `aisle` object tracking.	String	calibration-file= ./csv_files/nvaisle_2M.csv

12.6 MESSAGE-BROKER GROUP

This section describes the properties in the `message-broker` group.

Property	Meaning	Type and Range	Example and Notes
enable	Indicates whether message-broker is enabled.	Boolean	enable=1
broker-proto-lib	Pathname of a library containing the protocol adapter implementation, e.g. for Kafka.	String	broker-proto-lib=/usr/local/deepstream/libnvds_kafka_proto.so
broker-conn-str	Connection string to backend server.	String	broker-conn-str= foo.bar.com; 9092;dsapp1
proto-cfg-file	Pathname of the configuration file for broker-proto-lib (nvds_msg api as described in the *DeepStream 3.0 Plugin Manual*).	String	proto-cfg-file=config.txt

12.7 DEWARPER GROUP

This section describes the properties in the `dewarper` group.

Property	Meaning	Type and Range	Example and Notes
enable	Indicates whether dewarper is enabled.	Boolean	enable=1
gpu-id	Device ID of the GPU to use if multiple GPUs are available.	Integer ≥0	gpu-id=1
config-file	Pathname of dewarper configuration file. For more information, see “Gst-nvdewarper” in the *DeepStream 3.0 Plugin Manual*.	String	config-file= ./config_dewarper.txt

12.8 SOURCE GROUPS

This section describes the properties in the `source` group.

The DeepStream application can use multiple sources. The configuration file must define a `source` group for each source. The groups must be named `source0`, `source1`, etc.

Property	Meaning	Type and Range	Example and Notes
enable	Indicates whether this source is enabled.	Boolean	enable=1
type	Type of this source. Other properties of the source depend on its type. Possible values are: 1 = Camera - V4L2 2 = URI 3 = MultiURI	Integer 0, 1, or 2	type=1

Property	Meaning	Type and Range	Example and Notes
uri	URI of the encoded stream. May be a file, an HTTP URI, or an RTSP live source. Valid when `type` = 2 or 3. If `type=3`, `uri` may be specified several times to define several sources. Alternatively, `num-sources` may be used to generate several sources' URIs from a single value of the `uri` property.	String	uri= file:///home/ubuntu/source.mp4 uri=http://127.0.0.1/source.mp4 uri=rtsp://127.0.0.1/source1 uri= file:///home/ubuntu/source_%d.mp4
num-sources	Number of sources. May be used to define multiple source URIs if `type=3`. The source URIs are named `<uri>0`, `<uri>1`,… `<uri><num-sources-1>`, where `<uri>` is the value of the `uri` property.	Integer >0	num-sources=2
intra-decode-enable	Indicaties whether intra only decode is enabled.	Boolean	intra-decode-enable=1
num-decode-surfaces	Number of decode surfaces required for cuvid. Must be set to at least `num_ref_frames+1`.	Integer >0 and ≤20	num-decode-surfaces=6
gpu-id	Device ID of the GPU to use if multiple GPUs are available.	Integer ≥0	gpu-id=1
camera-id	A unique ID for the input source; the value is added to metadata. (Optional)	Integer ≥0	camera-id=2
camera-width	Width of frames to be requested from the camera, in pixels. Valid when `type=1`.	Integer >0	camera-width=1920
camera-height	Height of frames to be requested from the camera, in pixels. Valid when `type=1`.	Integer >0	camera-height=1080
camera-fps-n	Numerator part of a fraction defining the frame rate requested from the camera. Valid when the `type=1`.	Integer >0	camera-fps-n=30
camera-fps-d	Denominator part of a fraction specifying the frame rate requested from the camera. Valid when `type=1`.	Integer >0	camera-fps-d=1

Property	Meaning	Type and Range	Example and Notes
camera-v4l2-dev-node	Number of the V4L2 device node, e.g. `/dev/video<num>` for the open source V4L2 camera capture path. Valid when `type=1`.	Integer >0	camera-v4l2-dev-node=1
Latency	Jitterbuffer size in milliseconds; valid only for RTSP streams.	Integer ≥0	latency=200
cuda-memory-type	Type of CUDA memory to allocate for output buffers. Possible values are: 0 = cuda-pinned-mem: host/pinned memory allocated with `cudaMallocHost()`. 1 = cuda-device-mem: device memory allocated with `cudaMalloc()`. 2 = cuda-unified-mem: unified memory allocated with `cudaMallocManaged()`.	Integer 0, 1, or 2	cuda-memory-type=1

12.9 STREAMMUX GROUP

This section describes the properties in the `streammux` group.

Property	Meaning	Type and Range	Example and Notes
gpu-id	Device ID of the GPU to use if multiple GPUs are available.	Integer ≥0	gpu-id=1
live-source	Indicates whether sources are live.	Boolean	live-source=0
batch-size	Muxer batch size.	Integer >0	batch-size=4
batched-push-timeout	Time to wait after the first buffer is available to push the batch even if the complete batch is not formed, in microseconds.	Integer ≥−1	batched-push-timeout=40000 *Set to −1 for infinite timeout.*
width	Muxer output width.	Integer >0	width=1280
height	Muxer output height.	Integer >0	height=720

Property	Meaning	Type and Range	Example and Notes
enable-padding	Indicates whether to maintain source aspect ratio when scaling by adding black bands.	Boolean	enable-padding=0
cuda-memory-type	Type of CUDA memory to allocate for output buffers. Possible values are: 0 = cuda-pinned-mem: host/pinned memory allocated with `cudaMallocHost()`. 1 = cuda-device-mem: device memory allocated with `cudaMalloc()`. 2 = cuda-unified-mem: unified memory allocated with `cudaMallocManaged()`.	Integer 0, 1, or 2	cuda-memory-type=0

12.10 PRIMARY GROUP

This section describes the properties in the `primary` group.

Property	Meaning	Type and Range	Example and Notes
enable	Indicates whether the primary GIE must be enabled.	Boolean	enable=1
gie-unique-id	A unique ID assigned to the GIE so that its detected bounding boxes and labels can be identified by the application and other plugins.	Integer >0	gie-unique-id=2
gpu-id	Device ID of the GPU to use if multiple GPUs are available.	Integer ≥0	gpu-id=1
model-engine-file	Absolute pathname of the pre-generated serialized engine file for this mode.	String	model-engine-file=../../models/Primary_Detector/resnet10.caffemodel_b4_int8.engine

Property	Meaning	Type and Range	Example and Notes
cuda-memory-type	Type of CUDA memory to allocate for output buffers. Possible values are: 0 = cuda-pinned-mem: host/pinned memory allocated with `cudaMallocHost()`. 1 = cuda-device-mem: device memory allocated with `cudaMalloc()`. 2 = cuda-unified-mem: unified memory allocated with `cudaMallocManaged()`.	Integer 0, 1, or 2	cuda-memory-type=1
config-file	Pathname of a separate configuration file containing properties for the `Gst-nvinfer` plugin. For examples, see the sample file `samples/configs/deepstream-app/config_infer_resnet.txt` or the `deepstream-test2` sample application. For more details about the properties, see the "Gst-nvinfer" section of DeepStream 3.0 Plugin Manual.	String	config-file=/home/ubuntu/config_infer_resnet.txt
labelfile-path	Pathname of a text file containing labels for the model. The format of the labels file is defined in the *DeepStream 3.0 Development Guide*.	String	labelfile-path=/home/ubuntu/gie_labels.txt
batch-size	The number frames(P.GIE)/objects(S.GIE) to be inferred together in a batch.	Integer >0	batch-size=2
interval	Specifies the number of consecutive batches to be skipped for inference.	Integer >0	interval=2
bbox-border-color	Color of borders for objects of a specified class ID, in RGBA format. The property name is `bbox-border-color<class-id>`. May be used multiple times for multiple class IDs. If is not used for a given class ID, no borders are drawn for objects of that class-id.	R;G;B;A Each value Float >0.0 and ≤1.0	bbox-border-color2= 1;0;0;1 (red for class-id 2) *Four float values separated by semicolons.*
bbox-bg-color	Color of boxes drawn over objects of a specified class ID, in RGBA format. The property name is `bbox-bg-color<class-id>`. May be used multiple times for multiple class IDs. If not used for a given class ID, no boxes are drawn for objects of that class-id.	R;G;B;A Each value Float >0.0 and ≤1.0	bbox-bg-color3=0;1;0;0.3 (semi-transparent green for class-id 3)

Property	Meaning	Type and Range	Example and Notes
infer-raw-output-dir	The path to an existing directory where the application will dump the raw inference buffer contents in a file.	String	infer-raw-output-dir=/home/ubuntu/infer_raw_out

12.11 TRACKER GROUP

This section describes the properties in the `tracker` group.

Property	Meaning	Type and Range	Example and Notes
enable	Indicates whether the tracker is enabled or disabled.	Boolean	enable=1
tracker-width	Frame width at which the tracker operates, in pixels.	Integer >0	tracker-width=960
tracker-height	Frame height at which the tracker operates, in pixels.	Integer >0	tracker-height=752
gpu-id	Device ID of the GPU to use if multiple GPUs are available.	Integer ≥0	gpu-id=1
tracker-algorithm	Tracker algorithm to use. Possible values are: 0 = Default (KLT) 1 = KLT 2 = IOU	Integer 0, 1, or 2	tracker-algorithm=2
iou-threshold	Threshold for IOU tracker algorithm.	Float >0.0 and <1.0	iou-threshold=0.1
tracker-surface-type	Type of surfaces to track on. Possible values are: 0 = All surfaces 1 = Parking spot surfaces (360-D application) 2 = Aisle Surfaces (360-D application)	Integer 0, 1, or 2	tracker-surface-type=2

12.12 OSD GROUP

This section describes the properties in the `osd` group. This group specifies the properties and modifies the behavior of the `Gst-nvosd` plugin, which overlays text and rectangles on the video frame.

Property	Meaning	Type and Range	Example and Notes
enable	Enables or disables the on-screen display (OSD).	Boolean	enable=1
gpu-id	Device ID of the GPU to use if multiple GPUs are available.	Integer ≥0	gpu-id=1
border-width	The border width, in pixels, of the bounding boxes drawn for objects.	Integer ≥0	border-width=10 *0 disables bounding boxes.*
text-size	Size of the text that describes the objects, in points.	Integer ≥0	text-size=16
text-color	Color of the text that describes the objects, in RGBA format.	R;G;B;A Each value Float >0.0 and ≤1.0	text-color=0;0;0.7;1 (for dark blue)
text-bg-color	Background color of the text that describes the objects, in RGBA format.	R;G;B;A Each value Float >0.0 and ≤1.0	text-bg-color=0;0;0;0.5 (for semi-transparent black)
clock-text-size	Size of clock time text, in points.	Integer >0	clock-text-size=16
clock-x-offset	Horizontal offset of clock time text, in pixels.	Integer >0	clock-x-offset=100
clock-y-offset	Vertical offset of clock time text, in pixels.	Integer >0	clock-y-offset=100
font	Name of the font for text that describes objects.	String	font=Purisa *Use the* `fc-list` *command to list available fonts.*
show-clock	Enables or disables overlay of clock time on the frame.	Boolean	show-clock=1
clock-color	Color of clock time text, in RGBA format.	R;G;B;A Each value Float >0.0 and ≤1.0	clock-color=1;0;0;1 #Red
cuda-memory-type	Type of CUDA memory to allocate for output buffers. Possible values are: 0 = cuda-pinned-mem: host/pinned memory allocated with `cudaMallocHost()`. 1 = cuda-device-mem: device memory allocated with `cudaMalloc()`. 2 = cuda-unified-mem: unified memory allocated with `cudaMallocManaged()`.	Integer 0, 1, or 2	cuda-memory-type=1

12.13 SINK GROUP

This section describes the properties in the `sink` group. This group specifies the properties and modifies the behavior of the sink components for rendering, encoding, and file saving.

Property	Meaning	Type and Range	Example and Notes
enable	Enables or disables the sink.	Boolean	enable=1
type	Type of sink to use. Possible values are: 1 = Fake sink 2 = EGL bases windowed sink (nveglglessink) 3 = Encode + file save (encoder + muxer + filesink) 4 = Encode + RTSP streaming	Integer 1, 2, 3 or 4	type=2
sync	Indicates whether the stream must be rendered synchronously or as fast as possible. Valid only for renderers (`type` = 2, 3, or 4). Possible values are: 0 = As fast as possible 1 = Synchronously	Boolean	sync=0
source-id	ID of the source whose buffers this sink is to use. For example, for group [source1] source-id=1.	Integer ≥0	source-id=1 *Group* `source<n>` *defines buffers for source ID <n>. For example, group* `source2` *defines buffers for* `source-id` = *2.*
gpu-id	Device ID of theGPU to use if multiple GPUs are available.	Integer ≥0	gpu-id=1
container	Container to use to save the file. Valid only for `type` = 3. Possible values are: 1 = MP4 2 = MKV	Integer 1 or 2	container=1
codec	Encoder to use to save the file. Valid only for `type` = 3. Possible values are: 1 = H.264 2 = H.265 3 = MPEG4	Integer 1, 2, or 3	codec=1
bitrate	Bit rate to use for encoding, in bits/second. Valid only for `type` = 2.	Integer >0	bitrate=1000

Property	Meaning	Type and Range	Example and Notes
output-file	Pathname of the output encoded file. Valid only for `type` = 3.	String	output-file=/home/ubuntu/output.mp4
cuda-memory-type	Type of CUDA memory to allocate for output buffers. Possible values are: 0 = cuda-pinned-mem: host/pinned memory allocated with `cudaMallocHost()`. 1 = cuda-device-mem: device memory allocated with `cudaMalloc()`. 2 = cuda-unified-mem: unified memory allocated with `cudaMallocManaged()`.	Integer 0, 1, or 2	cuda-memory-type=1
rtsp-port	Port for the rtsp streaming server. Valid only for `type` = 4.	Integer Valid unused port number	rtsp-port=8554
udp-port	Port used internally by the streaming implementation. Valid only for `type` = 4.	Integer Valid unused port number	udp-port=5400

12.14 TESTS GROUP

This section describes the properties in the `tests` group. This group is used for diagnostics and debugging.

Property	Meaning	Type and Range	Example and Notes
file-loop	Indicates whether input files are looped infinitely.	Boolean	file-loop=1

www.nvidia.com