A. Implementation Details

We present additional implementation details to reproduce our baseline experiments below. We primarily consider two-stage baselines that first track all objects and then classify them into `referred objects`, `related objects`, or `other objects`. We repurpose 3D trackers from winning teams of the Argoverse 2 End-to-End Forecasting Challenge. All models are trained to output tracks for 26 annotated categories within Argoverse 2. We include an example prediction signature below:

```
{
      "timestamp_ns": int,
      "track_id": int,
      "confidence": float,
      "class_name": str,
      "translation_m": np.array,
      "size": np.array,
      "yaw": np.array,
}
```

We append a track for the ego vehicle with a consistent `track_id`, confidence of 1.00, class name of `EGO_VEHICLE`, yaw of 0 radians, size (<length, width height> in meters) of <4.877, 2, 1.473>, and translation (<x, y, z> in meters) of <1.422, 0, 0.25>. This translation represents the offset from the centroid of the ego-vehicle to the ego-vehicle reference coordinate near the rear axle.

All Tracks Oracle. Our naive baseline labels all ground truth tracks as `referred objects`. This means that all logs and timestamps are labeled as "positive", giving a balanced accuracy of 50%.

Filter by Referred Class. This baseline filters all tracks except those corresponding to the referred class. To get the referred class, we prompt Claude 3.7 Sonnet as follows:

> Categories: {AV2 Categories}
> Please select the one category that most corresponds to the object of focus in the description:{Natural Language Description}. As an example, for the description "vehicle turning at intersection with nearby pedestrians" your output would be VEHICLE. For the description "ego vehicle near construction barrel" your output would be EGO_VEHICLE. Your only output should be the category name.

We define two super-categories for `VEHICLE` and `ANY_OBJECT`. The `VEHICLE` category is manually defined to contain the classes `ARTICULATED_BUS`, `BOX_TRUCK`, `BUS`, `EGO_VEHICLE`, `LARGE_VEHICLE`, `MOTORCYCLE`, `RAILED_VEHICLE`, `REGULAR_VEHICLE`, `SCHOOL BUS`, `TRUCK`, and `TRUCK_CAB`.

Image Embedding Similarity. This baseline filters tracks by computing prompt-track CLIP similarity over time. At each timestep, we project the eight vertices of each object's bounding box onto the image plane of all seven cameras. We then cull any objects that do not partially fall within the frustum of each camera. For each projected bounding box, if three or more vertices fall within the image plane, we take a crop using the minimum and maximum uv image coordinates. We pad this crop by 30 pixels on each side and feed the resulting image into a CLIP ViT-L/14 model to obtain CLIP image embeddings. We also use CLIP ViT-L/14 to compute text embeddings for all prompts in the dataset. Next, we calculate cosine similarity scores between each track's image crops and the text prompt. Since individual similarity scores tend to be noisy, we follow ReferGPT [6] in applying a dynamic threshold and majority voting strategy to identify the `referred object` tracks. Since the dataset includes both positive and negative examples for each prompt, we group similarity scores by prompt and threshold the top 10% of scores. If more than half of the scores of a track fall within the top 10% threshold, we label the entire track as a `referred object`. To reduce the number tracks of length one (i.e., tracks spanning only one timestamp), we apply a similarity score modifier: $s_{\text{modified}} = s_{\text{cosine}} - \frac{0.05}{len(\text{track})}$.

LLM APIs as a Black Box We use the OpenAI Responses API to determine whether particular scenarios occur within a data log. We prompt GPT-5 with three files: an HD map data, 6-DoF ego-to-city transformations, and object tracker predictions. We use gpt-5-2025-08-07 with medium reasoning effort and access to a code interpreter tool. We give instructions to save a CSV containing the `referred object` id's and timestamps for each scenario. All 150 test data logs are run in parallel, with one request and one response for each data log. The saved CSVs are parsed into the RefAV format for evaluation as each response is delivered. We include the instructions given to GPT-5 below.

> You are an expert data analyst. Your job is to identify if a given scenarios occurs within an autonomous driving log. A log contains files with an HD map, 6-DoF poses of the ego vehicle, and bounding box annotations from an object tracker.
> For each of the given scenario descriptions, output a CSV file containing the track_uuids and timestamps of the referred objects. Some logs may not contain any referred objects. Your csv file should have "track_uuid" and "timestamp_ns" columns. Your output csv files should have the naming convention scenario{i}.csv, depending on the order the descriptions are given. Make sure to save all of your CSVs!
> Identify if each of these scenarios occurs within the log.
> {scenario_descriptions}

ReferGPT. This baseline uses a VLM to caption the cropped images of each track at each timestep. Tracks are filtered by computing a text similarity score between the prompt and track captions. We take several steps to adapt

ReferGPT to the RefAV dataset. At each timestep, we project the eight vertices of each predicted track's bounding box onto the image plane of all seven cameras. We then cull any objects that do not partially fall within the frustum of each camera. For each projected bounding box, if three or more vertices fall within the image plane, we take a crop using the minimum and maximum uv image coordinates. If the predicted track is visible in multiple cameras, we use the 2D bounding box with the highest area. We pad this crop by 10% of the crop's length on each side. For the ego-vehicle, the crop is the entire image from the front camera. Each of the per object-timestamp crops is also paired with the motion statistics. We compute a track i's motion statistics at timestep t as $\mathbf{C}_t^i = [x, y, \theta, v_x, v_y, \alpha, d]$, where x and y are the position of the track in meters, θ is the yaw of the track in degrees, v_x and v_y are the velocities of the track in meters per second, α is the yaw rate of the track in degrees per second. d is the distance from the centroids of the ego-vehicle and track in meters. All values are with respect to a static ego-centric coordinate frame. We prompt gpt-5-mini to caption each of the images as follows:

> Caption this image taken from an autonomous vehicle (referred to as the ego-vehicle). Our detection model classifies this object as {object_category}.
> Here are the coordinates of the object {object_motion}. This is in the format [x, y, yaw, x_velocity, y_velocity, angular_velocity, distance].
> This is in a coordinate system where positive x is to the front of the ego vehicle, positive y is to the left of the ego-vehicle, and positive z is above the ego-vehicle. Position and velocity are measured in meters and meters per second. Yaw is measured in degrees counter-clockwise (colloquially "left") from the x-axis. Angular velocity is in degrees per second. Distance is distance from the ego-vehicle.
> Please describe the appearance of the object and scene, if they are located relative to the road/ego-vehicle, how they are moving, and how they are interacting with any nearby objects (if any are visible). Treat values of x, y, x_velocity, and y_velocity near zero as zero. Feel free to add more relevant information in the captions.

We present an example caption below:

> A motorcyclist slightly to the front-right of the ego-vehicle is stopped or creeping forward beside a compact SUV, straddling a scooter with feet down and hands on the handlebars. The rider is close to the SUV's rear quarter, appearing stationary or moving very slowly while sharing the lane alongside other stopped traffic. The scene looks like an urban intersection or congested street with vehicles side-by-side and the rider positioned between them.

To determine if an object is `referred`, we compute a score $s = s_{cosine} + 0.1 s_{fuzzy}$. s_{cosine} is the cosine similarity between siglip2-base-patch16-naflex text embeddings of the caption and prompt. We follow the original ReferGPT implementation and compute s_{fuzzy} using Gestalt pattern matching to find the similarity between two strings. Finally, we determine that an entire track is `referred` if the majority the track's caption scores are above 0.63.

We find that the SigLIPv2 text similarity does not effectively separate the `referred` and `related objects` in the prompt. For example, the query "car to the left of a pedestrian", has high similarity with both "the vehicle is to the left of the person" and "the person is to the right of the vehicle". To resolve this issue, we post-process predictions with *Filter by Referred Class* to obtain the final set of `referred objects`.

Referential Tracking by Program Synthesis. RefProg filters off-the-shelf tracks using Python programs synthesized by a large language model (LLM), without explicitly relying on any information from LiDAR or cameras. To reduce the time required to run these synthesized programs, we first filter the input tracks based on confidence. We compute each track's confidence score by summing its confidence values across all timestamps. For the classes `REGULAR_VEHICLE`, `PEDESTRIAN`, `BOLLARD`, `CONSTRUCTION_CONE`, and `CONSTRUCTION_BARREL`, we retain the top 200 tracks per class. For all other classes, we retain the top 100 tracks per category. We prompt the LLM as follows:

> Please use the following functions to find instances of a referred object in an autonomous driving dataset. Be precise to the description, try to avoid returning false positives.
> API Listing: {API Listing}
> Categories: {AV2 Categories}
> Define a single scenario for the description:{Natural Language Description}
> Here is a list of examples: {Prediction Examples}. Only output code and comments as part of a Python block. Do not define any additional functions, or filepaths. Do not include imports. Assume the log_dir and output_dir variables are given. Wrap all code in one python block and do not provide alternatives. Output code even if the given functions are not expressive enough to find the scenario.

We include the API listing in Appendix J. We found this prompt consistently produces reasonable code across all tested LLMs. After running the synthesized programs, we post-process the output tracks. We discard object relationships where the spatial distance exceeds 50 meters. For the remaining tracks, we apply timestamp dilation to reduce flickering of the `referred` object labels. We symmetrically dilate time segments to a minimum of 1.5 seconds.

For example, if the base tracker tracks an object with ID 1 from 3.0 to 15.0 seconds at 2 Hz, and the synthesized code marks it as referred at timestamps 4.5 and 5.0, we update the final output to label the track as referred from 4.0 to 5.5 seconds. Finally, we downsample all output tracks to 2 Hz for evaluation.

B. Token Statistics for LLM Baselines

ReferGPT, *LLM APIs as a Black Box*, and *RefProg* can be viewed as different ways of providing scaffolding around an LLM for RMOT and scenario mining tasks. Most commercially available LLM APIs bill according to four cost categories: uncached input tokens, cached input tokens, output tokens, and code interpreter sessions (CIS). Input token count scales approximately linearly with the length of the input text, the number of input images, and seconds of input video. *ReferGPT* requires orders of magnitude more input and output tokens because it calls a VLM call for every tracked object at every timestamp. It also requires processing images, which can use more than 1000 tokens (depending on resolution) using GPT-5-mini. *LLM APIs as a Black Box* only makes as many calls as are there logs of data. However, inspecting the reasoning traces reveals that GPT-5 automatically went through multiple rounds of generating data processing code and verifying the output. The LLM also often re-generates the same processing code between logs. Initializing code interpreter sessions represents approximately half of the total cost ($38.49) of inference. RefProg makes as many calls as there are unique scenario descriptions. Since the entire atomic function API must be given in context, it requires a relatively high number of input tokens. However, most of these tokens are cached between requests. The output of the RefProg is relatively compact because the LLM has access to high-level composable functions and does not need to write extensive Python code.

Table 5. **Token Efficiency by LLM Baseline** RefProg achieves the highest scenario mining performance for the lowest cost. Cost is in USD for the OpenAI Platform in November 2025. We report the total number of tokens used millions. CIS refers to code interpreter sessions.

Method	LLM	HOTA-Temporal ↑	Cost ↓	Uncached Input	Cached Input	Output	CIS
ReferGPT	GPT-5-mini	20.7	150.64	372.9M	4.0M	28.7M	0
LLM API as Black Box	GPT-5	37.2	77.59	4.5M	0	3.4M	1283
RefProg (Ours)	GPT-5	42.3	13.89	0.6M	2.6M	1.3M	0

C. Comparison of Recent Datasets

We provide a qualitative comparison of language expressions found in RefAV with other popular RMOT-for-driving and VQA-for-driving datasets. In particular, we examine nuPrompt, Refer-KITTI, and nuScenes-QA. To obtain a representative sample of each dataset, we compute text embeddings for each expression using Qwen-3-Embedding-8B. We display the first 30 samples using farthest point sampling initialized from the two samples furthest apart from each other. Many expressions in nuPrompt deal with object attributes that are minimally relevant to driving such as car color, a person's gender, or a person's clothing. Refer-KITTI similarly only refers to annotated vehicle and pedestrian classes. nuScenes-QA asks questions that are relevant to driving, but the dataset is limited by its lack of human annotations and unnatural sounding questions. We present examples from each dataset below.

RefAV (Ours)

```
- the vehicle two cars behind the ego
    vehicle
- pedestrian walking with dog on sidewalk
- stationary object
- motorcycle at stop sign
- group of people
- construction workers working next to the
    road
- fire truck in a non-emergency
- excavator
- large truck blocking view of vehicle that
     is about to turn left onto a main road
- bicyclist changing lanes to the right
- car with a stroller to its left
- wheelchair user at pedestrian crossing
- person directing traffic in a school zone
- skateboarder on the road
- unattended dog
- accelerating wheeled devices
- construction barrel with at least 2
    construction cones within 3 meters
- bus with multiple pedestrians waiting on
    the right side
- vehicle passing the ego vehicle going
    signifcantly over the speed limit on an
    urban road
- van facing the wrong way in the middle
    turn lane
- two of the same work truck
- pedestrian near message board trailer
- pedestrians over 30 meters away
- the vehicle behind another vehicle being
    crossed by a jaywalking pedestrian
- vehicle flashing their hazard lights
- ego vehicle driving while it is cloudy
- motorcycle or pedestrian within 5 meters
    to the right of the ego vehicle
- bicycle behind a bus
- scooter lying down on the side of the
    road
- vehicles being passed by a motorcycle on
    either side
```

nuPrompt [72]

- Back, the pedestrian
- a stationary orange parking is on the parking space
- at the parking space, the trailer
- Car waiting with the same direction
- A person on foot by the roadside
- The silver lightcolored parked car is stationary in the parking space
- The objects possess the color blue
- Males with a black bag are getting ready to cross the road
- The one riding the black lefthandturn bicycle is a person
- Men stand on the street wearing pants and hats
- Females walking across the street
- Men wear pants while walking across the street
- In a state of sitting
- The car is in the act of turning
- Pedestrian getting ready to cross over
- is in the process of making a turn
- The car in motion is black and moving in the opposite direction
- Males wear tshirts and pants while making a left turn on the opposite side of the road
- the pedestrian is situated at the back left
- The dark car is at thestandstill and awaiting with the opposite direction as the truck
- The truck is thepale silver color
- we see the car ahead
- Females walk while wearing pants in the same direction
- Security camera
- Men are wearing shorts while standing on the road
- There is the truck positioned in the parking spot
- Color is red
- The bus is in front
- On the pedestrian pathway
- The car that is moving in the same direction is blue

Refer-KITTI [71]

- cars which are faster than ours
- walking pedestrian in the left
- women back to the camera
- males in the right
- left cars which are parking
- people in the pants
- people
- women carrying a bag
- light-color vehicles
- vehicles in the counter direction
- walking males
- standing females
- men in the left
- right cars in silver
- red moving cars
- women
- walking people in the right
- vehicles
- pedestrian
- vehicles in front of ours
- vehicles which are in the left and turning
- vehicles in white
- cars in horizon direction
- black vehicles in right
- standing men
- vehicles which are braking
- walking women
- men
- right cars in the same direction of ours
- black cars

nuScenes-QA [51]

- How many without rider motorcycles are there?
- There is a thing that is both to the front of me and the front left of the traffic cone; what is it?
- Does the trailer have the same status as the truck that is to the back of the trailer?
- How many pedestrians are to the back of the moving bus?
- What is the with rider thing that is to the back right of the with rider bicycle ?
- Are there any things?
- Is there another barrier of the same status as the barrier?
- The car to the back right of the stopped construction vehicle is in what status?
- There is a parked truck; what number of barriers are to the front left of it?
- Are there any other bicycles that in the same status as the car?
- What status is the motorcycle that is to the front left of the standing pedestrian?
- The without rider thing that is to the front of the construction vehicle is what?
- What number of traffic cones are there?

- Are there any trailers to the front of me?

- What is the thing to the back of the stopped thing?
- There is a without rider thing; are there any parked cars to the back of it?
- What number of barriers are to the back of the motorcycle?
- Are there any animals to the back right of me?
- The thing that is to the back right of the truck and the back right of the bus is what?
- There is a bus; is its status the same as the thing that is to the front of the bus?
- There is a with rider bicycle; what number of moving things are to the front of it?
- Are there any moving construction vehicles?
- The standing pedestrian to the back right of the traffic cone is what?
- How many buss are there?
- Is the status of the truck to the back of the pedestrian the same as the truck to the front of the bicycle?
- Is there another thing that has the same status as the animal?
- Are there any things to the back of the moving trailer?
- Are there any moving buss to the front of the without rider thing?
- The thing that is to the front of the moving truck and the front left of the stopped trailer is what?
- How many stopped trailers are there?

D. Scenario Mining Annotation Tool

In order to manually annotate interesting scenarios, we build a custom web app with Claude 3.7 (Figure 5). To construct a scenario, the user selects an object by clicking on a point within the image frame, writes a natural language description, and selects the start and end frames of the `referred objects` correspond to the prompt. We project a ray originating from the 2D point and find the ground truth 3D bounding box centroid closest to the ray. This tool allows us to quickly generate multi-object and multi-camera referential tracks.

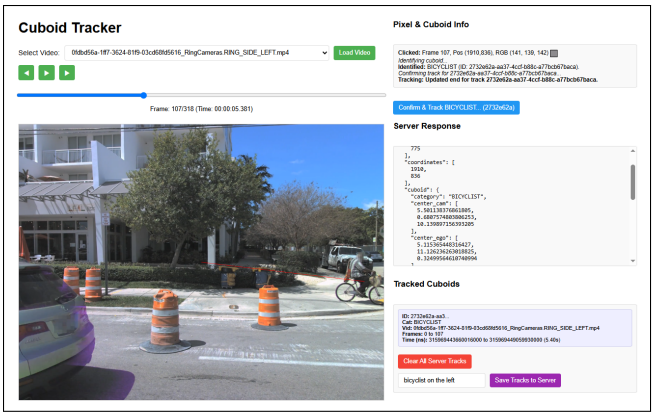

Figure 5. **Manual Annotation Tool**. We create an annotation tool to assist with labeling manually defined scenarios. Our tool allows us to quickly annotate multi-object referential tracks in AV2.

E. Referential Tracking Evaluation of `Related` and `Other` Objects

RefProg creates a scene graph for each natural language prompt that not only identifies `referred objects`, but also classifies tracks as `related objects` and `other objects`. For example, given the prompt `vehicle near a pedestrian`, the vehicle would be the `referred object`, the pedestrian would be the `related object`, and everything else in the scene would be considered `other objects`. We benchmark the base tracker with HOTA and evaluate HOTA-Temporal of `related objects` and `other objects` in Table 6. Since most objects are not related to the prompt, HOTA and HOTA-Temporal `Other` are similar.

Table 6. **Referential Tracking Accuracy for `Related` and `Other` Objects.** We present RefProg's referential tracking accuracy for `related objects` and `other objects`. We find that HOTA (for the base tracker) is similar to HOTA-Temporal (`Other`) because most objects are not relevant to the referential prompt.

Method	HOTA ↑	HOTA-Temporal (`Related`) ↑	HOTA-Temporal (`Other`) ↑
Referential Tracking by Program Synthesis			
Ground Truth	100.0	57.2	97.9
Le3DE2E [8]	78.9	35.6	74.2
ReVoxelDet [28]	69.4	28.2	63.4
TransFusion [2]	69.5	28.6	63.8
Valeo4Cast [74]	75.1	30.8	69.2

F. Impact of Sampling Rate on RefProg

We evaluate the impact of sampling rate on RefProg. First, we subsample three detectors (Le3DE2E [8], Valeo4Cast [74], and BEVFusion [37]) at 2Hz and 10 Hz. Next, we associate all detections with AB3DMOT [69] to generate 2Hz and 10Hz tracks respectively. Lastly, we evaluate RefProg on all trackers at 2Hz. We find that standard tracking

performance drops by 5% when subsampling due to the difficulty of associating sparse detections. We see a similar drop in HOTA-Track, Log Balanced Accuracy and Timestamp Balanced Accuracy. Interestingly, we do not see a similar drop in performance with HOTA-Temporal. This suggests that RefProg might be robust to temporally sparse inputs and ID switches.

Table 7. **Impact of Sampling Rate.** We evaluate RefProg's performance with subsampled inputs to measure the relative impact on referential tracking accuracy. We find that standard tracking performance drops by 5% when subsampling due to the difficulty of associating sparse detections.

Method	Hz	HOTA ↑	HOTA-Temporal ↑	HOTA-Track ↑	Log Bal. Acc. ↑	Timestamp Bal. Acc. ↑
Ground Truth	10	100.0	64.7	68.7	81.1	80.7
Ground Truth	2	100.0	64.2	68.8	80.6	80.1
Le3DE2E Det. [8] + AB3DMOT [69]	10	78.0	48.3	49.2	73.1	73.6
Le3DE2E Det. [8] + AB3DMOT [69]	2	73.0	45.2	44.1	69.3	69.5
Valeo4Cast Det. [74] + AB3DMOT [69]	10	74.8	42.8	45.5	71.6	71.8
Valeo4Cast Det. [37] + AB3DMOT [69]	2	69.9	45.2	44.1	69.3	69.5
BEVFusion Det. [37] + AB3DMOT [69]	10	76.0	46.4	47.8	73.7	72.3
BEVFusion Det. [37] + AB3DMOT [69]	2	70.9	46.4	45.2	72.1	72.9

G. More Dataset Statistics

Figure 6 quantifies important characteristics of RefAV such as the number of unique scenario descriptions, the number of positive and negative examples, the number of procedurally and manually defined scenarios, the number of scenarios that include object relationships, and the number of scenarios directly involving the ego-vehicle. We use the catch-all term "expression" for scenario description to better compare RefAV to other datasets. A positive match occurs when the data log contains at least one `referred object` corresponding to the scenario description. Figure 7 shows the number of positive and negative examples for representative scenario descriptions. In Figure 8, we evaluate the spatial distribution of `referred objects`.

H. Summary of Competition Top Performers

We summarize the contributions of top teams below.

Team Zeekr UMCV extends RefProg with global context-aware generation to processes all queries collectively for improved reasoning and consistency. They also propose a multi-agent refinement system where a secondary refiner agent iteratively debugs and enhances the generated code.

Team Mi3 UCM proposes two key improvements to RefProg: (1) Fault-Tolerant Iterative Code Generation (FT-ICG), which refines faulty code by re-prompting the LLM with error feedback until successful execution, and (2) Enhanced Prompting for Spatial Relational Functions (EP-SRF), which clarifies function semantics to prevent misinterpretations of spatial relationships between objects.

Team ZXH refines RefProg by prompting LLMs with a batch of queries and in-context examples instead of prompting for each query one at a time. Importantly, this team's approach relies on in-context learning and batch prompting to improve consistency and efficiency.

Expression Type	# Expressions
Unique	407
Positive Match	7910
Negative Match	2090
Procedurally Defined	9880
Manually Defined	120
Includes `Related Objects`	5531
Ego is the only `Referred Object`	827
Ego is the only `Related Object`	1200

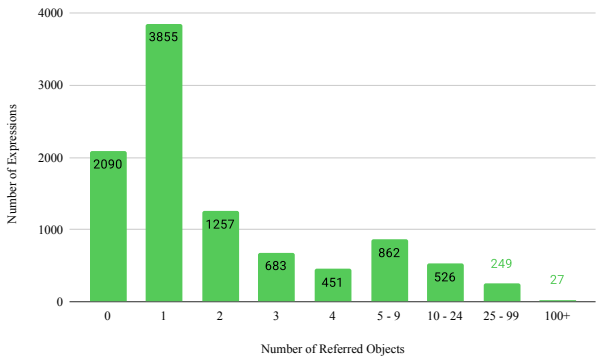

Figure 6. **RefAV Expression Statistics.** We compute the number of referring expressions categorized by expression type, count the number of the expressions that are positive or negative, and whether they are procedurally generated or manually constructed. We also highlight the number of positive expressions that include `related object` annotations and explicit references to the ego vehicle. The bar chart on the right shows the number of objects refered to by each expression. Most expressions in RefAV refer to zero (e.g. a negative match) or one object. On average, each expression refers to 3.92 objects.

I. Examples of RefProg's Synthesized Programs

We provide examples of RefProg's correctly synthesized programs below.

Prompt: Pedestrian crossing between stopped buses

```
# Description: pedestrian crossing between
    stopped buses

# First, let's get all pedestrians and buses from
     the dataset
pedestrians = get_objects_of_category(log_dir,
    category="PEDESTRIAN")
buses = get_objects_of_category(log_dir,
    category="BUS")

# We need to find stopped buses
stationary_buses = stationary(buses, log_dir)

# Alternatively, we could use has_velocity with
    a low threshold to find buses that are
    temporarily stopped
stopped_buses = has_velocity(buses, log_dir,
    min_velocity=0, max_velocity=0.5)

```

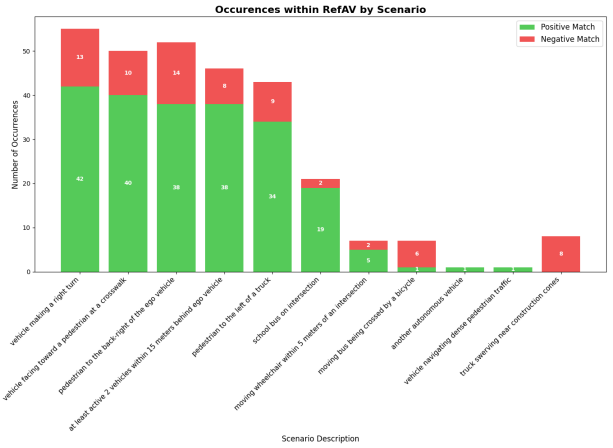

Figure 7. **Occurrences within RefAV by Scenario.** RefAV contains 407 unique scenario descriptions. We display 11 representative descriptions picked uniformly. After the diversity sampling process, RefAV includes about 40 positive matches for common scenarios such as `vehicle making a right turn`. Less common scenarios such as `moving bus being crossed by a bicycle` occur only in only a handful of logs across the Argoverse 2 dataset. RefAV includes negative examples for almost all procedurally defined scenarios. Manually annotated scenarios such as `another autonomous vehicle` do not have corresponding negative examples.

```
# Now we need to find pedestrians that are crossing between the stopped buses. First, identify pedestrians that are crossing in front of a stopped bus
peds_crossing_bus = being_crossed_by(stationary_buses, pedestrians, log_dir)

# Reverse the relationship to get the pedestrians that are doing the crossing
crossing_peds = reverse_relationship(being_crossed_by)(stationary_buses, pedestrians, log_dir)

# Now we need to find pedestrians that have buses on both sides (left and right). This means the pedestrian is between buses
peds_with_bus_left = has_objects_in_relative_direction(crossing_peds, stationary_buses, log_dir, direction="left", min_number=1, within_distance=10)
peds_with_bus_right = has_objects_in_relative_direction(crossing_peds, stationary_buses, log_dir, direction="right", min_number=1, within_distance=10)

# Combine to find pedestrians with buses on both sides
peds_between_buses = scenario_and([peds_with_bus_left, peds_with_bus_right])

# Output the scenario
output_scenario(peds_between_buses, "pedestrian crossing between stopped buses", log_dir, output_dir)
```

Prompt: Group of at least 3 moving bicyclists with 5m from each other

```
# Description: Group of at least 3 moving bicyclists within 5 meters of each other

# Get all bicyclists from the dataset
bicyclists = get_objects_of_category(log_dir, category="BICYCLIST")

# Filter out stationary bicyclists to get only moving ones
moving_bicyclists = scenario_not(stationary)(bicyclists, log_dir)

# Find bicyclists that have at least 2 other bicyclists within 5 meters (which makes a group of at least 3 bicyclists)
bicyclists_in_groups = near_objects(moving_bicyclists, moving_bicyclists, log_dir, distance_thresh=5, min_objects=2, include_self=False)

# Output the scenario
output_scenario(bicyclists_in_groups, "group of at least 3 moving bicyclists within 5 meters of each other", log_dir, output_dir)
```

Prompt: Accelerating vehicle changing lanes to the right

```
# Description: accelerating vehicle changing lanes to the right
# First, get all vehicles in the scene
vehicles = get_objects_of_category(log_dir, category="VEHICLE")

# Find vehicles that are accelerating using the accelerating function with a reliable minimum acceleration threshold of 0.65
accelerating_vehicles = accelerating(vehicles, log_dir, min_accel=0.65)

# Find vehicles that are changing lanes to the right. We use the changing_lanes function with direction='right'
right_lane_changes = changing_lanes(vehicles, log_dir, direction='right')

# Now find the intersection of accelerating vehicles and those changing lanes to the right. This gives us vehicles that are both accelerating and changing lanes to the right
accelerating_and_changing_right = scenario_and([accelerating_vehicles, right_lane_changes])

# Output the scenario
output_scenario(accelerating_and_changing_right, "accelerating vehicle changing lanes to the right", log_dir, output_dir)
```

We provide examples of RefProg's incorrectly synthesized programs below.

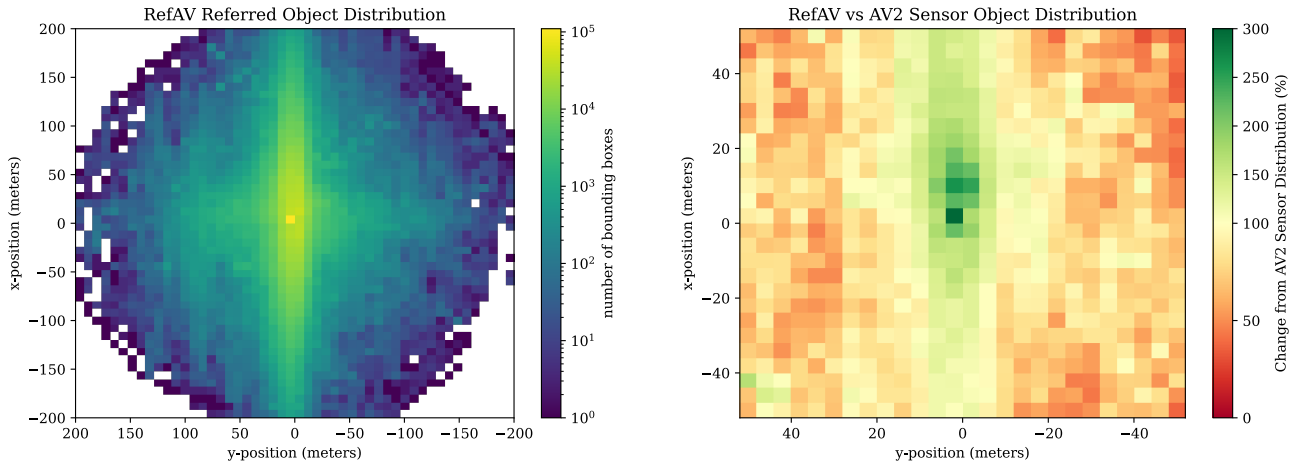

Figure 8. **RefAV Object Distribution.** RefAV includes `referred objects` in all directions up to 150m away from the ego vehicle. RefAV places a special focus on the ego vehicle and objects that interact with the ego vehicle. Therefore, `referred objects` are disproportionately located on the road in front of the ego vehicle. Both `referred object` heatmaps are in the ego vehicle coordinate frame.

Prompt: bus in front of ego vehicle obstructing view of red traffic light

The following program fails because the pre-defined atomic functions lack the expressivity to synthesize a correct program. Interestingly, the LLM realizes that the provided functions don't allow us to determine if a traffic light is being obstructed.

```
# Description: bus in front of ego vehicle obstructing view of red traffic light

# Get the ego vehicle and buses
ego_vehicle = get_objects_of_category(log_dir, category="EGO_VEHICLE")
buses = get_objects_of_category(log_dir, category="BUS")

# Find buses that are in front of the ego vehicle
# We need to find buses that are directly in front of the ego vehicle within a reasonable distance (20 meters) and with minimal lateral offset
buses_in_front_of_ego = get_objects_in_relative_direction(ego_vehicle, buses, log_dir, direction='forward', min_number=1, within_distance=20, lateral_thresh=3)

# We want buses that are moving in the same direction as the ego vehicle to ensure they're actually in the same lane and potentially blocking the view
buses_in_same_direction = heading_in_relative_direction_to(buses_in_front_of_ego, ego_vehicle, log_dir, direction='same')

# We need buses that are in the same lane as the ego vehicle
buses_in_same_lane = in_same_lane(buses_in_same_direction, ego_vehicle, log_dir)

# These buses are likely to be obstructing the view of traffic lights
# Note: The functions don't directly allow us to determine if a traffic light is being obstructed, but buses directly in front of the ego vehicle in the same lane are the most likely candidates for obstructing the view of traffic lights

output_scenario(buses_in_same_lane, "bus in front of ego vehicle obstructing view of red traffic light", log_dir, output_dir)
```

Prompt: bicycle following closely behind a vehicle

The following program fails due to imprecise semantics because the bicycle is not necessarily following the vehicle.

```
# Description: bicycle following closely behind a vehicle
# We need to find bicycles that are following vehicles closely

# Get all bicycles and vehicles from the dataset
bicycles = get_objects_of_category(log_dir, category="BICYCLE")
vehicles = get_objects_of_category(log_dir, category="VEHICLE")

# We need to identify bicycles that are behind vehicles using get_objects_in_relative_direction to find bicycles that are behind vehicles. The bicycles should be within a reasonable distance (e.g., 10 meters) to be considered "following closely"
bicycles_behind_vehicles =
```

```
    get_objects_in_relative_direction(vehicles,
    bicycles, log_dir, direction="backward",
    within_distance=10, lateral_thresh=2)

# We might also want to ensure the bicycle is
    moving (not stationary)
moving_bicycles = scenario_not(stationary)(
    bicycles, log_dir)

# Combine the conditions: bicycles that are both
    moving and following closely behind vehicles
bicycles_following_vehicles = scenario_and([
    bicycles_behind_vehicles, moving_bicycles])

# Output the scenario
output_scenario(bicycles_following_vehicles, "
    bicycle following closely behind a vehicle",
    log_dir, output_dir)
```

Prompt: vehicle starting to switch lanes but then going back

The following program uses incorrect logic. It fails to combine lane changes and acceleration and uses AND on exclusive sets.

```
# Strategy:
# 1. Get all vehicles
# 2. Find vehicles that are changing lanes
# 3. Find vehicles that are also changing lanes
    in the opposite direction shortly after
# 4. This indicates a vehicle that started to
    change lanes but then went back

# Get all vehicles
vehicles = get_objects_of_category(log_dir,
    category="VEHICLE")

# Find vehicles that are moving (not stationary)
moving_vehicles = scenario_not(stationary)(
    vehicles, log_dir)

# Find vehicles that are changing lanes in either
     direction
lane_changing_vehicles = changing_lanes(
    moving_vehicles, log_dir)

# Find vehicles that are changing lanes to the
    left
left_lane_changes = changing_lanes(
    moving_vehicles, log_dir, direction='left')

# Find vehicles that are changing lanes to the
    right
right_lane_changes = changing_lanes(
    moving_vehicles, log_dir, direction='right')

# A vehicle that starts to switch lanes but goes
    back would show lateral acceleration in one
    direction followed by lateral acceleration in
     the opposite direction. We can look for
    vehicles with significant lateral
    acceleration in both directions

# Find vehicles with lateral acceleration to the
    left
left_accel = has_lateral_acceleration(
    moving_vehicles, log_dir, min_accel=0.5)

# Find vehicles with lateral acceleration to the
    right
right_accel = has_lateral_acceleration(
    moving_vehicles, log_dir, min_accel=-0.5,
    max_accel=0)

# Vehicles that have both left and right
    acceleration patterns are likely the ones
    that started to change lanes but went back
lane_change_aborted = scenario_and([left_accel,
    right_accel])
# Output the scenario
output_scenario(lane_change_aborted, description,
     log_dir, output_dir)
```

J. API Listing

We present our full Python API listing below.

```
def has_objects_in_relative_direction(
    track_candidates:dict,
    related_candidates:dict,
    log_dir:Path,
    direction:Literal["forward", "backward", "
    left", "right"],
    min_number:int=1,
    max_number:int=np.inf,
    within_distance:float=50,
    lateral_thresh:float=np.inf)-> dict:
    """
    Identifies tracked objects with at least the
    minimum number of related candidates in the
    specified direction. If the minimum number is
     met, will create relationships equal to the
    max_number of closest objects.

    Args:
        track_candidates: Tracks to analyze (
    scenario dictionary). related_candidates:
    Candidates to check for in direction (
    scenario dictionary).
        log_dir: Path to scenario logs.
        direction: Direction to analyze from the
    track's point of view ('forward', 'backward',
     'left', 'right').
        min_number: Minimum number of objects to
    identify in the direction per timestamp.
    Defaults to 1.
        max_number: Maximum number of objects to
    identify in the direction per timestamp.
    Defaults to infinity.
        within_distance: Maximum distance for
    considering an object in the direction.
    Defaults to infinity.
        lateral_thresh: Maximum lateral distance
    the related object can be from the sides of
    the tracked object. Defaults to infinity.

    Returns:
        dict: A scenario dictionary where keys
    are track UUIDs and values are dictionaries
    containing related candidate UUIDs and lists
    of timestamps when the condition is met for
    that relative direction.
```

```
    Example:
        vehicles_with_peds_in_front = has_objects_in_relative_direction(vehicles, pedestrians, log_dir, direction='forward', min_number=2)
    """


def get_objects_in_relative_direction(
    track_candidates:dict,
    related_candidates:dict,
    log_dir:Path,
    direction:Literal["forward", "backward", "left", "right"],
    min_number:int=0,
    max_number:int=np.inf,
    within_distance:float=50,
    lateral_thresh:float=np.inf)->dict:
    """
    Returns a scenario dictionary of the related candidates that are in the relative direction of the track candidates.


    Args:
        track_candidates: Tracks (scenario dictionary).
        related_candidates: Candidates to check for in direction (scenario dictionary).
        log_dir: Path to scenario logs.
        direction: Direction to analyze from the track's point of view ('forward', 'backward', 'left', 'right').
        min_number: Minimum number of objects to identify in the direction per timestamp. Defaults to 0.
        max_number: Maximum number of objects to identify in the direction per timestamp. Defaults to infinity.
        within_distance: Maximum distance for considering an object in the direction. Defaults to infinity.
        lateral_thresh: Maximum lateral distance the related object can be from the sides of the tracked object. Lateral distance is distance is the distance from the sides of the object that are parallel to the specified direction. Defaults to infinity.

    Returns:
        dict: A scenario dictionary where keys are track UUIDs and values are dictionaries containing related candidate UUIDs and lists of timestamps when the condition is met for that relative direction.

    Example:
        peds_in_front_of_vehicles = get_objects_in_relative_direction(vehicles, pedestrians, log_dir, direction='forward', min_number=2)
    """


def get_objects_of_category(
    log_dir,
    category)->dict:
    """
    Returns all objects from a given category from the log annotations. This method accepts the super-categories "ANY" and "VEHICLE".

    Args:
        log_dir: Path to the directory containing scenario logs and data.
        category: the category of objects to return

    Returns:
        dict: A scenario dict that where keys are the unique id (uuid) of the object and values are the list of timestamps the object is in view of the ego-vehicle.

    Example:
        trucks = get_objects_of_category(log_dir, category='TRUCK')
    """


def is_category(
    track_candidates:dict,
    log_dir:Path,
    category:str)->dict:
    """
    Returns all objects from a given category from track_candidates dict. This method accepts the super-categories "ANY" and "VEHICLE".

    Args:
        track_candidates: The scenario dict containing the objects to filter down
        log_dir: Path to the directory containing scenario logs and data.
        category: the category of objects to return

    Returns:
        dict: A scenario dict that where keys are the unique id of the object of the given category and values are the list of timestamps the object is in view of the ego-vehicle.

    Example:
        box_trucks = is_category(vehicles, log_dir, category='BOX_TRUCK')
    """


def turning(
    track_candidates: dict,
    log_dir:Path,
    direction:Literal["left", "right", None]=None)->dict:
    """
    Returns objects that are turning in the given direction.

    Args:
        track_candidates: The objects you want to filter from (scenario dictionary).
```

```
        log_dir: Path to scenario logs.
        direction: The direction of the turn, from the track's point of view ('left', 'right', None).

    Returns:
        dict: A filtered scenario dictionary where keys are track UUIDs that meet the turning criteria and values are nested dictionaries containing timestamps.

    Example:
        turning_left = turning(vehicles, log_dir, direction='left')
    """


def changing_lanes(
    track_candidates:dict,
    log_dir:Path,
    direction:Literal["left", "right", None]=None
    )-> dict:
    """
    Identifies lane change events for tracked objects in a scenario.

    Args:
        track_candidates: The tracks to analyze (scenario dictionary).
        log_dir: Path to scenario logs.
        direction: The direction of the lane change. None indicates tracking either left or right lane changes ('left', 'right', None).

    Returns:
        dict: A filtered scenario dictionary where keys are track UUIDs that meet the lane change criteria and values are nested dictionaries containing timestamps and related data.

    Example:
        left_lane_changes = changing_lanes(vehicles, log_dir, direction='left')
    """


def has_lateral_acceleration(
    track_candidates:dict,
    log_dir:Path,
    min_accel=-np.inf,
    max_accel=np.inf)-> dict:
    """
    Objects with a lateral acceleartion between the minimum and maximum thresholds. Most objects with a high lateral acceleration are turning. Postive values indicate accelaration to the left while negative values indicate acceleration to the right.

    Args:
        track_candidates: The tracks to analyze (scenario dictionary).
        log_dir: Path to scenario logs.
        direction: The direction of the lane change. None indicates tracking either left or right lane changes ('left', 'right', None).

    Returns:
        dict: A filtered scenario dictionary where keys are track UUIDs that meet the lane change criteria and values are nested dictionaries containing timestamps and related data.

    Example:
        jerking_left = has_lateral_acceleration(non_turning_vehicles, log_dir, min_accel=2)
    """


def facing_toward(
    track_candidates:dict,
    related_candidates:dict,
    log_dir:Path,
    within_angle:float=22.5,
    max_distance:float=50)->dict:
    """
    Identifies objects in track_candidates that are facing toward objects in related candidates. The related candidate must lie within a region lying within within_angle degrees on either side the track-candidate's forward axis.

    Args:
        track_candidates: The tracks that could be heading toward another tracks
        related_candidates: The objects to analyze to see if the track_candidates are heading toward
        log_dir:  Path to the directory containing scenario logs and data.
        fov: The field of view of the track_candidates. The related candidate must lie within a region lying within fov/2 degrees on either side the track-candidate's forward axis.
        max_distance: The maximum distance a related_candidate can be away to be considered by

    Returns:
        dict: A filtered scenario dict that contains the subset of track candidates heading toward at least one of the related candidates.

    Example:
        pedestrian_facing_away = scenario_not(facing_toward)(pedestrian, ego_vehicle, log_dir, within_angle=180)
    """


def heading_toward(
    track_candidates:dict,
    related_candidates:dict,
    log_dir:Path,
    angle_threshold:float=22.5,
    minimum_speed:float=.5,
    max_distance:float=np.inf)->dict:
```

```
    """
    Identifies objects in track_candidates that are heading toward objects in related candidates. The track candidates acceleartion vector must be within the given angle threshold of the relative position vector. The track candidates must have a component of velocity toward the related candidate greater than the minimum_accel.

    Args:
        track_candidates: The tracks that could be heading toward another tracks
        related_candidates: The objects to analyze to see if the track_candidates are heading toward
        log_dir:  Path to the directory containing scenario logs and data.
        angle_threshold: The maximum angular difference between the velocity vector and relative position vector between the track candidate and related candidate.
        min_vel: The minimum magnitude of the component of velocity toward the related candidate
        max_distance: Distance in meters the related candidates can be away from the track candidate to be considered

    Returns:
        dict: A filted scenario dict that contains the subset of track candidates heading toward at least one of the related candidates.

    Example:
        heading_toward_traffic_cone = heading_toward(vehicles, traffic_cone, log_dir)
    """


def accelerating(
    track_candidates:dict,
    log_dir:Path,
    min_accel:float=.65,
    max_accel:float=np.inf)->dict:
    """
    Identifies objects in track_candidates that have a forward acceleration above a threshold. Values under -1 reliably indicates braking. Values over 1.0 reliably indiciates accelerating.

    Args:
        track_candidates: The tracks to analyze for acceleration (scenario dictionary)
        log_dir:  Path to the directory containing scenario logs and data.
        min_accel: The lower bound of acceleration considered
        max_accel: The upper bound of acceleration considered

    Returns:
        dict: A filtered scenario dictionary containing the objects with an acceleration between the lower and upper bounds.

    Example:
        accelerating_motorcycles = accelerating(motorcycles, log_dir)
    """


def has_velocity(
    track_candidates:dict,
    log_dir:Path,
    min_velocity:float=.5,
    max_velocity:float=np.inf)->dict:
    """
    Identifies objects with a velocity between the given maximum and minimum velocities in m/s. Stationary objects may have a velocity up to 0.5 m/s due to annotation jitter.

    Args:
        track_candidates: Tracks to analyze (scenario dictionary).
        log_dir: Path to scenario logs.
        min_velocity: Minimum velocity (m/s). Defaults to 0.5.
        max_velocity: Maximum velocity (m/s)

    Returns:
        dict: Filtered scenario dictionary of objects meeting the velocity criteria.

    Example:
        fast_vehicles = has_min_velocity(vehicles, log_dir, min_velocity=5)
    """


def at_pedestrian_crossing(
    track_candidates:dict,
    log_dir:Path,
    within_distance:float=1)->dict:
    """
    Identifies objects that within a certain distance from a pedestrian crossing. A distance of zero indicates that the object is within the boundaries of the pedestrian crossing.

    Args:
        track_candidates: Tracks to analyze (scenario dictionary).
        log_dir: Path to scenario logs.
        within_distance: Distance in meters the track candidate must be from the pedestrian crossing. A distance of zero means that the object must be within the boundaries of the pedestrian crossing.

    Returns:
        dict: Filtered scenario dictionary where keys are track UUIDs and values are lists of timestamps.

    Example:
        vehicles_at_ped_crossing = at_pedestrian_crossing(vehicles, log_dir)
    """
```

```
272
273
274 def on_lane_type(
275     track_uuid:dict,
276     log_dir,
277     lane_type:Literal["BUS", "VEHICLE", "BIKE"])
      ->dict:
278     """
279     Identifies objects on a specific lane type.
280
281     Args:
282         track_candidates: Tracks to analyze (
      scenario dictionary).
283         log_dir: Path to scenario logs.
284         lane_type: Type of lane to check ('BUS',
      'VEHICLE', or 'BIKE').
285
286     Returns:
287         dict: Filtered scenario dictionary where
      keys are track UUIDs and values are lists of
      timestamps.
288
289     Example:
290         vehicles_on_bus_lane = on_lane_type(
      vehicles, log_dir, lane_type="BUS")
291     """
292
293
294 def near_intersection(
295     track_uuid:dict,
296     log_dir:Path,
297     threshold:float=5)->dict:
298     """
299     Identifies objects within a specified
      threshold of an intersection in meters.
300
301     Args:
302         track_candidates: Tracks to analyze (
      scenario dictionary).
303         log_dir: Path to scenario logs.
304         threshold: Distance threshold (in meters)
       to define "near" an intersection.
305
306     Returns:
307         Filtered scenario dictionary where keys
      are track UUIDs and values are lists of
      timestamps.
308
309     Example:
310         bicycles_near_intersection =
      near_intersection(bicycles, log_dir,
      threshold=10.0)
311     """
312
313
314 def on_intersection(
315     track_candidates:dict,
316     log_dir:Path)->dict:
317     """
318     Identifies objects located on top of an road
      intersection.
319
320     Args:
321         track_candidates: Tracks to analyze (
      scenario dictionary).
322         log_dir: Path to scenario logs.
323
324     Returns:
325         dict: Filtered scenario dictionary where
      keys are track UUIDs and values are lists of
      timestamps.
326
327     Example:
328         strollers_on_intersection =
      on_intersection(strollers, log_dir)
329     """
330
331
332 def being_crossed_by(
333     track_candidates:dict,
334     related_candidates:dict,
335     log_dir:Path,
336     direction:Literal["forward", "backward", "
      left",
337     "right"]="forward",
338     in_direction:Literal['clockwise','
      counterclockwise',
339     'either']='either',
340     forward_thresh:float=10,
341     lateral_thresh:float=5)->dict:
342     """
343     Identifies objects that are being crossed by
      one of the related candidate objects. A
      crossing is defined as the related candidate'
      s centroid crossing the half-midplane of a
      tracked candidate. The direction of the half-
      midplane is specified with the direction.
344
345     Args:
346         track_candidates: Tracks to analyze .
347         related_candidates: Candidates (e.g.,
      pedestrians or vehicles) to check for
      crossings.
348         log_dir: Path to scenario logs.
349         direction: specifies the axis and
      direction the half midplane extends from
350         in_direction: which direction the related
       candidate has to cross the midplane for it
      to be considered a crossing
351         forward_thresh: how far the midplane
      extends from the edge of the tracked object
352         lateral_thresh: the two planes offset
      from the midplane. If an related candidate
      crosses the midplane, it will continue being
      considered crossing until it goes past the
      lateral_thresh.
353
354     Returns:
355         dict: A filtered scenario dictionary
      containing all of the track candidates that
      were crossed by the related candidates given
      the specified constraints.
356
357     Example:
358         overtaking_on_left = being_crossed_by(
      moving_cars, moving_cars, log_dir, direction=
      "left", in_direction="clockwise",
      forward_thresh=4)
359         vehicles_crossed_by_peds =
      being_crossed_by(vehicles, pedestrians,
      log_dir)
360     """
361
362
```

```python
def near_objects(
    track_uuid:dict,
    candidate_uuids:dict,
    log_dir:Path,
    distance_thresh:float=10,
    min_objects:int=1,
    include_self:bool=False)->dict:
    """
    Identifies timestamps when a tracked object is near a specified set of related objects.

    Args:
        track_candidates: Tracks to analyze (scenario dictionary).
        related_candidates: Candidates to check for proximity (scenario dictionary).
        log_dir: Path to scenario logs.
        distance_thresh: Maximum distance in meters a related candidate can be away to be considered "near".
        min_objects: Minimum number of related objects required to be near the tracked object.

    Returns:
        dict: A scenario dictionary where keys are timestamps when the tracked object is near the required number of related objects and values are lists of related candidate UUIDs present at those timestamps.

    Example:
        vehicles_near_ped_group = near_objects(vehicles, pedestrians, log_dir, min_objects=3)
    """


def following(
    track_uuid:dict,
    candidate_uuids:dict,
    log_dir:Path)-> dict:
    """
    Returns timestamps when the tracked object is following a lead object. Following is defined simultaneously moving in the same direction and lane.
    """


def heading_in_relative_direction_to(
    track_candidates,
    related_candidates,
    log_dir,
    direction:Literal['same', 'opposite', 'perpendicular'])->dict:
    """
    Returns the subset of track candidates that are traveling in the given direction compared to the related candidates.

    Arguments:
        track_candidates: The set of objects that could be traveling in the given direction
        related_candidates: The set of objects that the direction is relative to
        log_dir: The path to the log data
        direction: The direction that the positive tracks are
        traveling in relative to the related candidates:
            - "opposite" indicates the track candidates are traveling in a direction 135-180 degrees from the direction the related candidates are heading toward.
            - "same" indicates the track candidates that are traveling in a direction 0-45 degrees from the direction the related candiates are heading toward.
            - "perpendicular" indicates the track candidates that are traveling in a direction 45-135 degrees from the direction the related candiates are heading toward.

    Returns:
        dict: the subset of track candidates that are traveling in the given direction compared to the related candidates.

    Example:
        oncoming_traffic = heading_in_relative_direction_to(vehicles, ego_vehicle, log_dir, direction='opposite')
    """


def stationary(
    track_candidates:dict,
    log_dir:Path)->dict:
    """
    Returns objects that moved less than 2m over their length of observation in the scneario. This object is only intended to separate parked from active vehicles. Use has_velocity() with thresholding if you want to indicate vehicles that are temporarily stopped.

    Args:
        track_candidates: Tracks to analyze (scenario dictionary).
        log_dir: Path to scenario logs.

    Returns:
        dict: A filtered scenario dictionary where keys are track UUIDs and values are lists of timestamps when the object is stationary.

    Example:
        parked_vehicles = stationary(vehicles, log_dir)
    """


def at_stop_sign(
    track_candidates:dict,
    log_dir:Path,
    forward_thresh:float=10)->dict:
    """
    Identifies timestamps when a tracked object is in a lanee corresponding to a stop sign. The tracked object must be within 15m of the stop sign. This may highlight vehicles using street parking near a stopped sign.
```

```
447
448     Args:
449         track_candidates: Tracks to analyze (scenario dictionary).
450         log_dir: Path to scenario logs.
451         forward_thresh: Distance in meters the vehcile is from the stop sign in the stop sign's front direction
452
453     Returns:
454         dict: A filtered scenario dictionary where keys are track UUIDs and values are lists of timestamps when the object is at a stop sign.
455
456     Example:
457         vehicles_at_stop_sign = at_stop_sign(vehicles, log_dir)
458     """
459
460
461 def in_drivable_area(
462     track_candidates:dict,
463     log_dir:Path)->dict:
464     """
465     Identifies objects within track_candidates that are within a drivable area.
466
467     Args:
468         track_candidates: Tracks to analyze (scenario dictionary).
469         log_dir: Path to scenario logs.
470
471     Returns:
472         dict: A filtered scenario dictionary where keys are track UUIDs and values are lists of timestamps when the object is in a drivable area.
473
474     Example:
475         buses_in_drivable_area = in_drivable_area(buses, log_dir)
476     """
477
478
479 def on_road(
480     track_candidates:dict,
481     log_dir:Path)->dict:
482     """
483     Identifies objects that are on a road or bike lane. This function should be used in place of in_driveable_area() when referencing objects that are on a road. The road does not include parking lots or other driveable areas connecting the road to parking lots.
484
485     Args:
486         track_candidates: Tracks to filter (scenario dictionary).
487         log_dir: Path to scenario logs.
488
489     Returns:
490         dict: The subset of the track candidates that are currently on a road.
491
492     Example:
493         animals_on_road = on_road(animals, log_dir)
494     """
495
496
497 def in_same_lane(
498     track_candidates:dict,
499     related_candidates:dict,
500     log_dir:Path)->dict:
501     """
502     Identifies tracks that are in the same road lane as a related candidate.
503
504     Args:
505         track_candidates: Tracks to filter (scenario dictionary)
506         related_candidates: Potential objects that could be in the same lane as the track (scenario dictionary)
507         log_dir: Path to scenario logs.
508
509     Returns:
510         dict: A filtered scenario dictionary where keys are track UUIDs and values are lists of timestamps when the object is on a road lane.
511
512     Example:
513         bicycle_in_same_lane_as_vehicle = in_same_lane(bicycle, regular_vehicle, log_dir)
514     """
515
516 def on_relative_side_of_road(
517     track_candidates:dict,
518     related_candidates:dict,
519     log_dir:Path,
520     side=Literal['same', 'opposite'])->dict:
521     """
522     Identifies tracks that are in the same road lane as a related candidate.
523
524     Args:
525         track_candidates: Tracks to filter (scenario dictionary)
526         related_candidates: Potential objects that could be in the same lane as the track (scenario dictionary)
527         log_dir: Path to scenario logs.
528
529     Returns:
530         dict: A filtered scenario dictionary where keys are track UUIDs and values are lists of timestamps when the object is on a road lane.
531
532     Example:
533         bicycle_in_same_lane_as_vehicle = in_same_lane(bicycle, regular_vehicle, log_dir)
534     """
535
536 def is_color(
537     track_candidates: dict,
538     log_dir: Path,
539     color:Literal["white", "silver", "black", "red", "yellow", "blue"])->dict:
540     """
```

```
    Returns objects that are the given color, determined by SIGLIP2 feature similarity.

    Args:
        track_candidates: The objects you want to filter from (scenario dictionary).
        log_dir: Path to scenario logs.
        color: The color of the objects you want to return. Must be one of 'white', 'silver', 'black', 'red', 'yellow', or 'blue'. Inputting a different color defaults to returning all objects.

    Returns:
        dict: A filtered scenario dictionary where keys are track UUIDs that meet the turning criteria and values are nested dictionaries containing timestamps.

    Example:
        red_cars = is_color(cars, log_dir, color='red')
    """

def scenario_and(scenario_dicts)->dict:
    """
    Returns a composed scenario where the track objects are the intersection of all of the track objects with the same uuid and timestamps.

    Args:
        scenario_dicts: the scenarios to combine

    Returns:
        dict: a filtered scenario dictionary that contains tracked objects found in all given scenario dictionaries

    Example:
        jaywalking_peds = scenario_and([peds_on_road, peds_not_on_pedestrian_crossing])
    """


def scenario_or(scenario_dicts)->dict:
    """
    Returns a composed scenario where that tracks all objects and relationships in all of the input scenario dicts.

    Args:
        scenario_dicts: the scenarios to combine

    Returns:
        dict: an expanded scenario dictionary that contains every tracked object in the given scenario dictionaries

    Example:
        be_cautious_around = scenario_or([animal_on_road, stroller_on_road])
    """


def reverse_relationship(func)->dict:
    """
    Wraps relational functions to switch the top level tracked objects and relationships formed by the function.

    Args:
        relational_func: Any function that takes track_candidates and related_candidates as its first and second arguements

    Returns:
        dict: scenario dict with swapped top-level tracks and related candidates

    Example:
        group_of_peds_near_vehicle = reverse_relationship(near_objects)(vehicles, peds, log_dir, min_objects=3)
    """


def scenario_not(func)->dict:
    """
    Wraps composable functions to return the difference of the input track dict and output scenario dict. Using scenario_not with a composable relational function will not return any relationships.

    Args:
        composable_func: Any function that takes track_candidates as its first input

    Returns:
        dict: scenario dict with tracks that are not filtered by func

    Example:
        active_vehicles = scenario_not(stationary)(vehicles, log_dir)
    """


def output_scenario(
    scenario:dict,
    description:str,
    log_dir:Path,
    output_dir:Path,
    visualize:bool=False,
    **visualization_kwargs):
    """
    Outputs a file containing the predictions in an evaluation-ready format. Do not provide any visualization kwargs.
    """
```