Developer Documentation

Introduction: EyePop.ai - A Self-Service AI Platform for Computer Vision Apps

Welcome to EyePop.ai, a revolutionary self-service AI platform designed to empower individuals of all technical backgrounds to effortlessly create applications powered by Computer Vision AI. Our platform is your gateway to the exciting world of computer vision technology, enabling you to harness the power of visual data for various applications. In this documentation, we'll explore the capabilities and features of EyePop.ai, giving you the knowledge and tools you need to dive into the realm of computer vision and build innovative applications.

What is Computer Vision?

Computer vision is a groundbreaking technology that allows machines to interpret and take action based on visual data, such as images or videos. This capability equips machines with the ability to perform tasks that require visual recognition, opening up a world of possibilities. Imagine products that adapt to user-generated content, instant demographic insights derived from visual media, or real-time analysis of live streams to measure audience engagement. EyePop.ai makes these possibilities a reality, and this documentation will guide you through the process of utilizing this technology for your own projects.

Stay with us as we delve deeper into the features, functionalities, and potential applications of EyePop.ai. Whether you're a seasoned developer or someone with little technical experience, EyePop.ai is here to support your journey in creating innovative applications driven by Computer Vision AI.

How does EyePop.ai work?

After signing up, you'll be equipped with an AI worker server tailored to streamline your detection tasks. We call that a “Pop”. Whether you're looking to post images, videos, or even connect through live webrtc, EyePop.ai offers the flexibility and efficiency to suit your requirements.

Start the process by heading to https://app.eyepop.ai/sign-up
Select Create a Pop
Choose API Pop

How do I get the configuration information to my Pop ?

After configuring an API Type Pop you’ll be given an endpoint and a token. You can use these to retrieve information for your AI Worker Server.

```
async function FetchPopConfig(pop_endpoint, token)
{
	return fetch(pop_endpoint, {
		method: 'GET',
		headers: {
			'Accept': 'application/json',
			'Authorization': 'Bearer '+token
		}
	})
	.then(response => response.json())
}
```

Pushing Media to your Pop

The Setsource operation allows developers to modify the source of a live Pop. There are various ways to set the source, each addressing different use cases.

1. **Set Source (image/video upload)**
    - HTTP Method: POST
    - Endpoint: '**{Your server}/pipelines/{id}/source**'
    - Description: Changes the live pipeline to use the uploaded image or video file as a new source.
    - Parameters:
        - '**file**': The new source as an image or video file. This needs to be provided in the form data.
        - '**id**': The ID of the pipeline to change. This is a required path parameter.

        - '**mode**': The mode to handle concurrent sources. Available values include: **'reject'**, **'preempt'**, and **'queue'**. This is a required query parameter.
        - '**processing**': The processing mode, either to wait for the result or to fire and forget. Available values are: **'sync'** (synchronous) and **'async'** (asynchronous). This is a required query parameter.
    - Responses:
        - '**200 OK**': The operation was successful.
        - '**204 No Content**': The operation was successful but there is no content to send in the response.
2. **Set Source (image/video url)**
    - HTTP Method: PATCH
    - Endpoint: '**{Your server}/pipelines/{id}/source**'
    - Description: Changes the live pipeline to use a new source specified by a URL.
    - Parameters:
        - **'sourceDef'**: Definition of the new source, including '**sourceType**' set to "URL" and '**url**' of the image.
        - **'id'**: The ID of the pipeline to change. This is a required path parameter.
        - **'mode'**: The mode to handle concurrent sources. Available values include: **'reject'**, **'preempt'**, and **'queue'**. This is a required query parameter.
        - **'processing'**: The processing mode, either to wait for the result or to fire and forget. Available values are: **'sync'** (synchronous) and **'async'** (asynchronous). This is a required query parameter.
    - Responses:
        - **'200 OK'**: The operation was successful.

        - **'204 No Content'**: The operation was successful but there is no content to send in the response.

3. **Set Source (image/video url list)**
    - HTTP Method: PATCH
    - Endpoint: '**{Your server}/pipelines/{id}/source**'
    - Description: Changes the live pipeline to use a new source specified by a list of image URLs synchronously.
    - Parameters:
        - **'sourceDef'**: Definition of the new source, including **'sourceType'** set to "LIST" and **'sources'**, which is an array containing the list of image URLs.
        - **'id'**: The ID of the pipeline to change. This is a required path parameter.
        - **'mode'**: The mode to handle concurrent sources. Available values include: **'reject'**, **'preempt'**, and **'queue'**. This is a required query parameter.
        - **'processing'**: The processing mode, specifically set to **'sync'** for synchronous processing. This is a required query parameter.
    - Responses:
        - '**200 OK**': The operation was successful.
        - '**204 No Content**': The operation was successful but there is no content to send in the response.
4. **Set Source (streaming video WebRTC)**
    - HTTP Method: PATCH
    - Endpoint: '**{Your server}/pipelines/{id}/source**'
    - Description: Changes the live pipeline to use a new streaming video source via WebRTC.
    - Parameters:

        - **'sourceDef'**: Definition of the new source, including **'sourceType'** set to "WEBRTC_DIRECT", **'webrtcSessionId'**, and **'streamId'**.
        - **'id'**: The ID of the pipeline to change. This is a required path parameter.
        - **'mode'**: The mode to handle concurrent sources. Available values include: **'reject'**, **'preempt'**, and **'queue'**. This is a required query parameter.
        - **'processing'**: The processing mode, either to wait for the result or to fire and forget. Available values are: **'sync'** (synchronous) and **'async'** (asynchronous). This is a required query parameter.
    - Responses:
        - '**200 OK**': The operation was successful.
        - '**204 No Content**': The operation was successful but there is no content to send in the response.

Examples:
https://github.com/eyepop-ai/Demos

```
let formData = new FormData()

formData.append('file', file)



fetch(config.url + '/pipelines/' + config.pipeline_id +
'/source?mode=preempt&processing=sync', {

      method: 'POST',

      headers: {

            'accept': 'application/json'

      },

      body: formData

})

.then(response => response.json())

.then(data => {

//YOUR CODE TO DO SOMETHING WITH THE RESULT

})

.catch(error => console.error('The party\'s over, there was an error:',
error));
```

EyePop.ai Computer Vision API Results

The EyePop.ai's computer vision API provides detailed results concerning the contents of the analyzed images. These results are structured in a JSON format.

1. **Source Image Dimensions:**
    - The results include the dimensions of the source image which was analyzed:
        - '**result."source_width"**': The width of the source image.

- ‘**result."source_height"**’: The height of the source image.

2. **Objects Detected in Image:**
   - The API identifies various objects within the image and provides details about each of these objects:
     - ‘**result."objects"**’: An array containing information about each object detected.
     - ‘**result.objects[0]."confidence"**’: The confidence score associated with the detected object. This value indicates the certainty of the detection.
     - ‘**result.objects[0]."classId"**’: An integer identifier for the class of the detected object.
     - ‘**result.objects[0]."classLabel"**’: The label of the class, e.g., "person".
     - ‘**result.objects[0]."x", result.objects[0]."y"**’: The top-left coordinates of the detected object within the image.
     - ‘**result.objects[0]."width", result.objects[0]."height"**’: The width and height dimensions of the detected object.
3. **Objects Detected Within Objects:**
   - For some primary detected objects, the API might identify secondary objects within them. For example, within a "person" object, the API could detect a "face" object.
     - ‘**result.objects[0]."objects"**’: An array containing secondary objects detected within a primary object.
     - ‘**result.objects[0]."objects"[0]."classLabel"**’: The label of the class of the secondary object, e.g., "face".
4. **Classifications on Objects within Objects:**
   - For some secondary objects, the API performs classifications to identify various attributes:
     - ‘**result.objects[0]."objects"[0]."classes"**’: Contains the classifications for the secondary object.
       - "inferId": 4: Refers to Emotion classification.

- "inferId": 5: Refers to Age classification.
- "inferId": 6: Refers to Gender classification.
- "inferId": 7: Refers to Race classification.

5. **Body Keypoints**:
    - For some primary objects like "person", the API identifies body keypoints, indicating specific landmarks such as the eyes, nose, wrists, etc.
        - '**result.objects[0]."keyPoints"."points"**': An array containing information about each keypoint detected.
        - '**result.objects[0]."keyPoints"."points"[0]."confidence"**': Confidence score for the detected keypoint.
        - '**result.objects[0]."keyPoints"."points"[0]."classLabel"**': The label of the keypoint, e.g., "nose".
        - '**result.objects[0]."keyPoints"."points"[0]."x"**', '**result.objects[0]."keyPoints"."points"[0]."y"**': The coordinates of the detected keypoint within the image.

Labels (People & Common Objects)

The following labels are recognized by the EyePop.ai Computer Vision API for people and common objects:

eyepop logo, person, bicycle, car, motorcycle, airplane, bus, train, truck, boat, traffic light, fire hydrant, street sign, stop sign, parking meter, bench, bird, cat, dog, horse, sheep, cow, elephant, bear, zebra, giraffe, hat, backpack, umbrella, shoe, eye glasses, handbag, tie, suitcase, frisbee, skis, snowboard, sports ball, kite, baseball bat, baseball glove, skateboard, surfboard, tennis racket, bottle, plate, wine glass, cup, fork, knife, spoon, bowl, banana, apple, sandwich, orange, broccoli, carrot, hot dog, pizza, donut, cake, chair, couch, potted plant, bed, mirror, dining table, window, desk, toilet, door, tv, laptop, mouse, remote,

keyboard, cell phone, microwave, oven, toaster, sink, refrigerator, blender, book, clock, vase, scissors, teddy bear, hair drier, toothbrush

Labels (Expressions)

The following labels are recognized by the EyePop.ai Computer Vision API for a person's expressions:

Happy, Neutral, Sad, Surprise, Angry, Fear, Disgust

Labels (Body Parts)

The following labels are recognized by the EyePop.ai Computer Vision API for a person's expressions:

Left Eye, Left Ear, Left Shoulder, Left Hip, Left Elbow, Left Wrist, Left Knee, Left Ankle, Right Eye, Right Ear, Right Shoulder, Right Hip, Right Elbow, Right Wrist, Right Knee, Right Ankle, Nose

Javascript SDK

EyePop.ai has introduced a dedicated npm package for JavaScript developers, streamlining the process of integrating your Pop. This package ensures an efficient and user-friendly setup experience.

NPM Installation:

```
npm install @eyepop.ai/javascript-sdk
```

CDN Link:

```
<script src="https://cdn.jsdelivr.net/npm/@eyepop.ai/javascript-sdk"></script>
```

Example usage for Webcam setup:
https://github.com/eyepop-ai/Demos/tree/main/Fitness%20Rep%20Counter

Draw Types

The SDK supports an array of drawing methods, organized alphabetically by type. To use, you must change the Draw parameter on the config object used to initiate the EyePopSDK.

- **box:**
  - **Targets**: Specific objects or parts of objects you wish to encircle within a box. E.g., "person.face". A value of "*" means all labels.
- **box with Tracking:**
  - **Targets**: A list of objects to enclose within a box. E.g., "apple", "backpack", and more. A value of "*" means all labels.
  - **Tracking**:
    - *Tracking allows you to receive a unique ID for each person in a scene. This ID is held stable while a person remains on screen. Tracking has to be turned on for your account so please ask an EyePop.ai Team member to turn this on for you.*
    - **Labels**: An array of assigned labels for the tracking augmentation.
- **pose:**
  - **Targets**: Currently, in the present version, the pose type can only target "person", highlighting body key points.
- **posefollow:**

    - **Targets**: Objects like "person" that you aim to track. A value of "*" means all labels.
    - **Anchors**: Specific points within the target object where the augmentation is anchored, such as "right wrist".
    - **Image**: The image placed at the anchor point, like "./images/sunglasses3.png".
    - **Scale**: Augmentation image's scaling factor, e.g., 1.
- **posefollow with multiple Anchors:**
    - **Targets**: Objects to track, such as "person". A value of "*" means all labels.
    - **Anchors**: Multiple anchor points within the target object, like "right eye", "left eye".
    - **Image**: Image to be superimposed on the anchor points, such as "./images/scream.png".
    - **Scale**: The image's scaling factor, e.g., 3.5.

Example:
https://github.com/eyepop-ai/Demos/tree/main/Fitness%20Rep%20Counter

```
FetchPopConfig(pop_endpoint, token)
      .then(response => config = response)
      .then(() => {
            document.getElementById("title").innerHTML = config.name;
            config.Draw = [
                  {"Type": "box", "Targets":["*"]},
                  {"Type": "pose", "Targets": ["person"] },
            ];
            const ep = EyePopSDK.EyePopSDK.init(config)
      });
```

Class: Rules

The Rules class has been specifically crafted for processing the outputs of EyePop.ai's computer vision system. It provides functionalities to construct semantic rules, helping in the identification and extraction of specific features and attributes from photos and videos.

Methods:

1. **FindObjects(label, objects)**
    - Purpose: Filters the provided list of objects based on the specified class label.
    - Parameters:

        - **'label'**: String representing the class label of the desired object.
        - **'objects'**: Array of objects.
    - Returns: Array of objects that match the specified class label.
2. **Biggest(label, objects)**
    - Purpose: Identifies the object with the largest bounding box area for a specific class label.
    - Parameters:
        - **'label'**: String representing the class label of the object to compare.
        - **'objects'**: Array of objects.
    - Returns: Single object with the largest area.
3. **Area(object, source_width, source_height)**
    - Purpose: Computes the relative area of an object to the source's dimensions.
    - Parameters:
        - '**object**': Object whose area needs to be determined.
        - '**source_width**': Width of the source.
        - '**source_height**': Height of the source.
    - Returns: Relative area (fraction) of the object with respect to the source dimensions.
4. **Between(x, min, max)**
    - Purpose: Checks if a given value lies between a specified range.
    - Parameters:
        - '**x**': The value to be checked.
        - '**min**': Minimum value of the range.
        - '**max**': Maximum value of the range.
    - Returns: Boolean value indicating whether x lies between min and max.
5. **Amount(label, objects)**

- Purpose: Counts the number of objects that match a specific class label.
- Parameters:
  - '**label**': String representing the class label.
  - '**objects**': Array of objects.
- Returns: Integer count of objects that match the specified label.

6. **PosePoint(label, personObject)**
   - Purpose: Determines if a person object contains a specific pose point label.
   - Parameters:
     - '**label**': Pose point label.
     - '**personObject**': Object containing pose information.
   - Returns: Boolean indicating presence of the pose point label.
7. **Emotion(emotionLabel, personObject)**
   - Purpose: Checks the inferred emotion on a person's face.
   - Parameters:
     - '**emotionLabel**': The desired emotion label.
     - '**personObject**': Object containing facial information.
   - Returns: Boolean indicating the presence of the specified emotion.
8. **Gender(genderLabel, personObject)**
   - Purpose: Checks the inferred gender label of a person based on the identified facial features.
   - Parameters:
     - '**genderLabel**': Gender label to check.
     - '**personObject**': Person Object containing facial information.
   - Returns: Boolean indicating if the identified gender label matches the specified personObject.
9. **Position(object1, direction, object2)**
   - Purpose: Compares the relative positions of two objects based on the specified direction.

- Parameters:
  - '**object1**': First object.
  - '**direction**': String representing the desired direction (above, below, left, right).
  - '**object2**': Second object.
- Returns: Boolean indicating the relative position of object1 with respect to object2 based on the given direction.

10. **Check(resultSet, rules, rulesState)**
    - Purpose: Evaluates a set of conditions on the provided resultSet and tracks the state of rule evaluations.
    - Parameters:
      - **'resultSet'**: Data to be evaluated.
      - **'rules'**: Array containing conditions to evaluate on the resultSet.
      - **'rulesState'**: Object to track the state of rule evaluations.
    - Returns: Array of results for each rule evaluation.

Examples:
https://github.com/eyepop-ai/Demos/tree/main/AI%20CDN%20-%20Computer%20Vision%20Endpoint%20%26%20UGC%20Ruleset
(See: 4_add_rules_to_results.html)

Low Code Rules (Coming Soon)

EyePop Command Parser

The EyePop Command Parser, built on the PEG.js framework, provides the ability to interpret and process a variety of commands related to computer

vision tasks. There are different types of commands available for developers, each serving a specific purpose:

- Count Command (countCommand)
  - **Format**: [Entity] Count [Operator] [Integer Value]
  - **Description**: Checks the count of a specified entity.
  - **Example**: '**Person Count > 2**'
- Count Rule Command (countRuleCommand)
  - **Format**: Rule Count [Rule Index] [Operator] [Integer Value]
  - **Description**: Checks the count of a specified rule index.
  - **Example**: '**Rule Count 3 > 5**'
- Between Command (betweenCommand)
  - **Format**: [Entity] Area between [Low Percentage]% and [High Percentage]%
  - **Description**: Verifies if the area of a specified entity falls between the given percentages.
  - **Example**: '**Car Area between 10% and 50%**'
- Position Command (positionCommand)
  - **Format**: [Entity with Pose Part] [Relation] [Entity with Pose Part]
  - **Description**: Checks the position relation between two entities with specific pose parts.
  - **Example**: '**Person left eye above Person right hand**'
- Emotion Command (emotionCommand)
  - **Format**: [Entity] Expression = [Emotion]
  - **Description**: Validates if a specified entity (typically a face) is exhibiting a given emotion.
  - **Example**: '**Person Expression = Happy**'
- Combined Position Command (combinedPositionCommand)
  - **Format**: A combination of multiple positionCommand entries.
  - **Description**: Allows checking multiple position relations in one command.

- **Example**: '**Person left eye above Person right hand Person left wrist below Person left elbow**'

Supported Definitions

- **Entities (entity)**: This includes a variety of objects such as 'Person,' 'Car,' 'Bicycle,' 'Teddy Bear,' and many more.
- **Emotions (emotion)**: Supported emotions include 'Happy,' 'Neutral,' 'Sad,' 'Surprise,' 'Angry,' 'Fear,' and 'Disgust.'
- **Size Parts (sizePart)**: Defines the relative size of an entity, with options like 'Biggest' and 'Smallest.'
- **Entity Part (entityPart)**: A combination of size part and an entity. For example, 'Biggest Car.'
- **Entity Pose Part (entityPosePart)**: Combines an entity part with a pose part. For example, 'Biggest Person left eye.'
- **Pose Part (posePart)**: Refers to specific body parts in a direction, like 'left eye' or 'right wrist.'
- **Operators (op)**: Determines the relationship for comparison, with supported operators including '>' (greater than) and '=' (equal to).
- **Relations (relation)**: Describes the positional relationship, such as 'above' or 'below.'
- **Percentage (percentage)**: An integer percentage, such as '10' or '50.'
- **Integer (integer)**: Regular integer values.
- **Word (word)**: General string values without spaces.