MEC Solution Accelerator Architecture

Contents

[Introduction and goals 1](#_Toc2068591721)

[Global Architecture 1](#_Toc2009745203)

[Architecture at the Edge (Short Term) 1](#_Toc1415469232)

[Microservice boundaries 3](#_Toc1644733946)

[Pub/Sub Event Bus and asynchronous communication between microservices 4](#_Toc338099561)

[Event bus building block in DAPR 4](#_Toc1311827279)

[Integration events 5](#_Toc1184149401)

[.NET Libraries 5](#_Toc308069128)

[Events hierarchy 5](#_Toc1606348702)

[The DetectionEvent class is derived from the base event class and represents events related to the detection of objects in a video stream. It contains information about the detected objects and their properties, such as location and type. 5](#_Toc846794000)

[Microservices Analysis 6](#_Toc817050365)

[Microservice 1: Frame Splitter WIP 7](#_Toc939202508)

[Microservice 2: AI Inferencer 8](#_Toc805346809)

[Microservice 3: Rules-Engine 9](#_Toc203635274)

[Microservice 4: Alerts-API 11](#_Toc98914312)

[Microservice 5: Alerts-UI 12](#_Toc1392727661)

[Alerts-UI Functionality and use cases: 12](#_Toc748903371)

[Adding a new rule to the Rule Engine 13](#_Toc1056885966)

[In-Depth Implementation Guide, for Adding a New Rule to the Rule Engine 14](#_Toc1900368954)

[How to provision a new feed in Frame Splitter 16](#_Toc2123869813)

[New Feed Provision 16](#_Toc104925036)

[How to replace the default AI model (Yolo) by a new custom DeepLearning mode 18](#_Toc772188355)

[Pytorch model 18](#_Toc58053574)

[Adding a model with no pytorch implementation 19](#_Toc1085352000)

MEC Solution Accelerator Architecture and Design

# Introduction and goals

The goals of this system are to process data from IoT and edge devices efficiently, using AI models to detect issues in real-time. The system uses a Publish/Subscription approach for event processing and generates alerts for immediate attention. The microservice-oriented architecture is autonomous, highly scalable, and easy to maintain. The system aims to improve overall operational efficiency and enable real-time decision-making.

# Global Architecture

This reference application provides an implementation of an event-driven and microservice-oriented architecture with multiple autonomous microservices. The primary goal of the system is to ingest data from IoT and edge devices, analyze it using AI models, and detect issues that require immediate attention. To achieve this goal, the system is designed to ingest video streams from wireless cameras and use AI models to analyze the video in real-time.

To ensure that the system can respond quickly to events, a Publish/Subscription approach is used to submit events to a messaging broker. Additional microservices then evaluate these events and determine if they require further action. Alerts are generated for events that require immediate attention and are published to various event handlers, such as an "Alerts dashboard" app or other integrated processes that need to react in real-time.

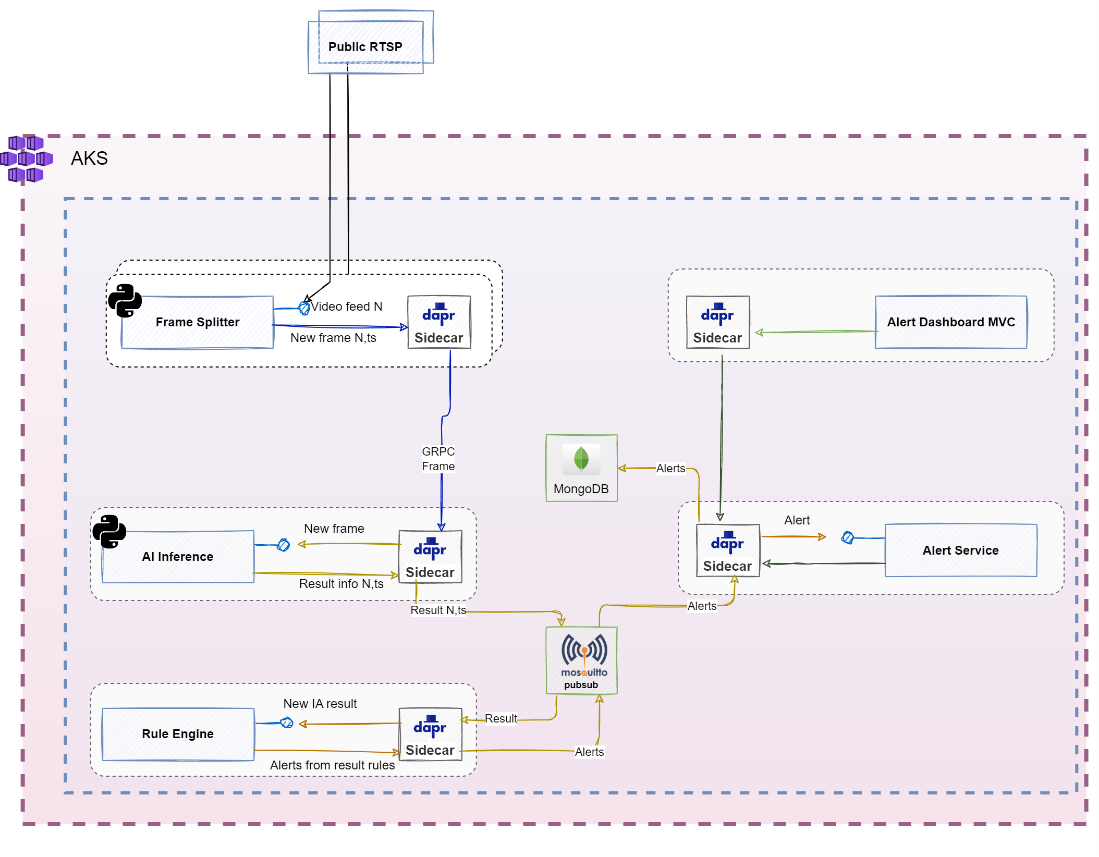
The microservices in the system are designed to be autonomous, allowing for easy scaling and maintenance. Overall, this architecture is designed to provide a scalable, fault-tolerant, and highly responsive system for processing data from IoT and edge devices.

## Architecture at the Edge (Short Term)

The data flow in a system designed to process video from thousands of cameras in real-time can be complex and challenging to manage. This text provides a detailed explanation of the data flow in such a system, covering the different components and their interactions. It also describes the role of different microservices-based pods and how they communicate.

1. The video streams are received from thousands of cameras equipped with 5G technology. These streams are then processed and split into frames by the Frame Splitter deployment. Each camera has its own Frame Splitter deployment, which generates ten frames per second and send them using service invocation GRPC to the ai-inference pod. This frame limitation can be upgraded when creating more inference pods.
2. The ai-inference model deployment app receives the frames from its Dapr service invocation and performs object detection on them. When it detects an object, it generates a detection event that includes bounding boxes and detection information. These detection events are enqueued in the Pub/Sub of DAPR using the MQTT protocol. The messages are serialized using AVRO format to reduce their size.
3. The Rules Engine deployment consumes the detection events and applies configurable business rules to generate alerts. The rules engine is coded in .NET and uses JSON for rule configuration. The rules are designed to verify that the detection meets specific criteria, such as checking for a threshold of detections or verifying that the correct number and types of objects are present in the frame. When the rules engine generates an alert, it publishes it in a separate topic in the Pub/Sub of DAPR.
4. The Alerts API deployment exposes REST endpoints for viewing and managing alerts. It uses Cosmos DB to persist alerts for later retrieval. The API provides endpoints for listing alerts, sorting them, and retrieving details of each alert. The API subscribes to the Pub/Sub of DAPR to consume new alert messages as they are generated.
5. The Alerts UI deployment is a .NET MVC application that consumes alerts from the Pub/Sub using DAPR service invocation. The user interface is used to view and manage alerts generated by the system. It provides a list of alerts, and a detailed view of each alert, which includes the detection frame with bounding boxes and other information.

In summary, the data flow starts with video streams, which are processed and split into frames. The frames are published in the Pub/Sub using MQTT and AVRO. The YOLOv5 model reads the frames and generates detection events, which are consumed by the Rules Engine. The rules engine generates alerts, which are published in the Pub/Sub and consumed by the Alerts API. Finally, the Alerts UI provides a user interface for viewing and managing alerts.



# Microservice boundaries

* Frame Splitter - This microservice is responsible for processing video streams from thousands of cameras equipped with 5G technology. It splits the video streams into frames and generates ten frames per second for each camera. The purpose of this microservice is to create a boundary around video processing and to manage large volumes of data efficiently.
* AI Inferencer - This microservice is responsible for object detection on the frames generated by the Frame Splitter. It detects objects and generates detection events that include bounding boxes and detection information. The purpose of this microservice is to create a boundary around AI model deployment and to provide efficient object detection.
* Rules Engine - This microservice is responsible for applying configurable business rules to the detection events generated by the AI Inferencer. It verifies that the detection meets specific criteria, such as checking for a threshold of detections or verifying that the correct number and types of objects are present in the frame. The purpose of this microservice is to create a boundary around custom rules management, and to ensure efficient alert generation.
* Alerts API - This microservice is responsible for managing alerts generated by the Rules Engine. It exposes REST endpoints for viewing and managing alerts and uses Cosmos DB to persist alerts for later retrieval. The purpose of this microservice is to create a boundary around alert management and to enable efficient alert storage and retrieval.
* Alerts UI - This microservice is responsible for consuming alerts generated by the Rules Engine and displaying them in a user interface. It provides a list of alerts and a detailed view of each alert, which includes the detection frame with bounding boxes and other information. The purpose of this microservice is to create a boundary around alert presentation and to provide an efficient user interface for viewing and managing alerts.

# Pub/Sub Event Bus and asynchronous communication between microservices

In a system that processes video from thousands of cameras in real-time, it's crucial to process a large volume of data quickly. Asynchronous communication enables the system's different components to operate independently, without being blocked by another component's processing. This approach uses message passing and non-blocking I/O operations to achieve better system performance, scalability, and reduced latency.

Using an event bus based on the Pub/Sub pattern has several advantages in a real-time video processing system. It allows for better component decoupling, which makes the system more flexible and adaptable to changes. New subscribers can be added to the system without requiring changes to the publishers, improving scalability. Pub/Sub also allows the system to continue functioning even if some components fail, improving fault tolerance.

In summary, combining asynchronous communication with a Pub/Sub event bus can provide numerous benefits in a real-time video processing system. It enables better component decoupling, improves scalability, and improves fault tolerance, ensuring that the system can continue functioning smoothly even in the event of component failures.

## Event bus building block in DAPR

We used DAPR Pub/Sub with MQTT protocol in the real-time video processing system because it offers a decoupled messaging architecture that enables system components to operate independently and enhances system flexibility. The MQTT protocol is lightweight and ideal for use in real-time applications, ensuring low latency. Additionally, DAPR Pub/Sub provides built-in support for message serialization, routing, filtering, and dead-letter queues, ensuring reliable message delivery even in the event of a failure. By combining DAPR Pub/Sub with MQTT protocol, the system can efficiently process large volumes of data, improving system performance while simplifying the development process.

## Integration events

In the context of using DAPR Pub/Sub with MQTT for exchanging Integration Events, choosing the right serialization format is critical to ensuring efficient and reliable communication between microservices. AVRO serialization provides a compact and efficient format that reduces the size of message payloads, making them easier to transmit and process. By using AVRO serialization with DAPR Pub/Sub and MQTT, microservices can exchange Integration Events reliably and efficiently, while minimizing message overhead and simplifying the development process.

### .NET Libraries

On the .net side, there are two representative libraries for event serialization. On the one hand, for the generation of models and schemas:

[Avro Apache Tools](https://www.nuget.org/packages/Apache.Avro.Tools/)

In the other hand, for the serialization object to bytes, and the other way around:

[Avro Convert Serializer](https://github.com/AdrianStrugala/AvroConvert)

#### Python Libraries

On the python side, is used for serialization and deserialization of objects:

[Avro Json Serializer](http://avro-json-serializer)

### Events hierarchy

In the system, there are different types of events that are raised based on specific actions or occurrences within the system.

The base event class serves as the parent class for all events, while its derived classes provide additional functionality and specific data for different types of events.

### The DetectionEvent class is derived from the base event class and represents events related to the detection of objects in a video stream. It contains information about the detected objects and their properties, such as location and type.

The Alert class is derived from the DetectionEvent class and represents events that are generated when a certain condition is met, such as an object of interest being detected in the video stream. It contains additional information, such as the image frame where the object was detected, the time of the event, and other relevant data.

This hierarchy allows for a flexible and extensible design, where additional types of events can be easily added by creating new derived classes from the base event class.

Json Schemas

Using JSON Schema to define the structure of the system's events ensures consistency and well-defined data structures, making it easier to work with and share data between different components. This, in turn, helps to prevent errors and improve the overall reliability of the system.

This schema is used to generate Avro serialization classes that can be used to convert the events to and from binary data that can be transmitted over the network.

Here is the git link to the event schema folder:

[Shchema Folder](https://github.com/Azure/mec-app-solution-accelerator/tree/main/src/Services/Detections/ai_inferencer/events_schema)

# Microservices Analysis

## Microservice 1: Frame Splitter WIP

The Frame Splitter is a Python-based microservice that processes the input video to obtain the frames that will be sent to the inference microservice. With OpenCV, it receives the video stream and splits it in different frames. With Dapr, it performs service invocation of the inference microservice, sending frame with a real-time compromise, thus discarding frames when the inference microservice is busy.

Delving in the Frame splitter workflow:

1. **Selecting the input video:** By checking the microservice id and comparing it to the dynamically modified environment feeds dictionary, the microservice has the information to connect to the right feed. feed-configmap.yaml is used in a Kubernetes implementation while docker-compose-override.yaml is used in Docker Composte. In *How to provision a new feed in Frame Splitter*, all the information about how to change the input feed is presented.
2. **Retrieving the input video:** By using OpenCV, the microservice connects to the input video, and if it is a local video or a photo, it makes a loop to ensure infinite stream.
3. **Frame Splitting loop**:
   1. A frame is retrieved using OpenCV.
   2. The frame is transformed to base64.
   3. The message to send using service invocation with the base64 frame and some status aditional information in json format.
   4. Inference microservice is invoked to perform the inference sending the previous message.
   5. When there is an inference microservice free, the frame corresponding to that instant of time is sent.

The simplified flow diagram is depicted in figure 1.0.3

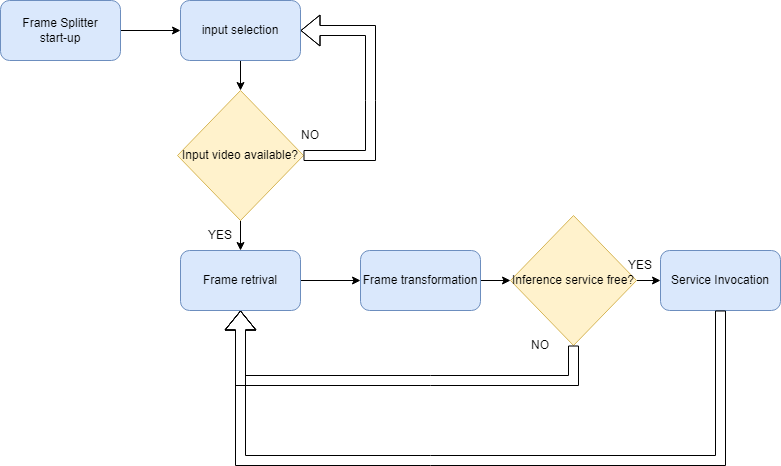


Figure 1.0.1: Frame splitter microservice flow.

## Microservice 2: AI Inferencer

The AI inferencer is a Python-based microservice that, given a frame, performs the inference using an ai model to find the desired artifacts and sends a detection event to be later consumed by the rules engine. With Dapr, creates an App which receives the json with the frame from the frame splitter through an App service invocation. With OpenCV, processes the image prior to the inference. With Pytorch, in this demo case, loads the model and generates the raw insights about the frame. With Avro, formats the event to be sent and finally with Dapr publishes the event.

Delving in the Frame splitter workflow:

1. App creation: when the microservice starts, using Dapr a Dapr App is executed which remains listening for future invocations.
2. Model load: In the process of App creation, the model is loaded and prepared to be executed.
3. Receiving frame information: Information about the video feed such as name, the frame in base64 and logging info is received through Dapr App when invoked by the Frame Splitter.
4. Inference Execution:
   1. Input data and Model are sent to the inference scrip (inference.py)
   2. Input image is loaded and preprocessed to be inferenced
   3. Inference outputs are processed to generate the event dictionary
   4. Event dictionary is serialized using AVRO
   5. Serialized event is published into Dapr Pub/Sub

The simplified flow diagram is depicted in figure 2.0.1

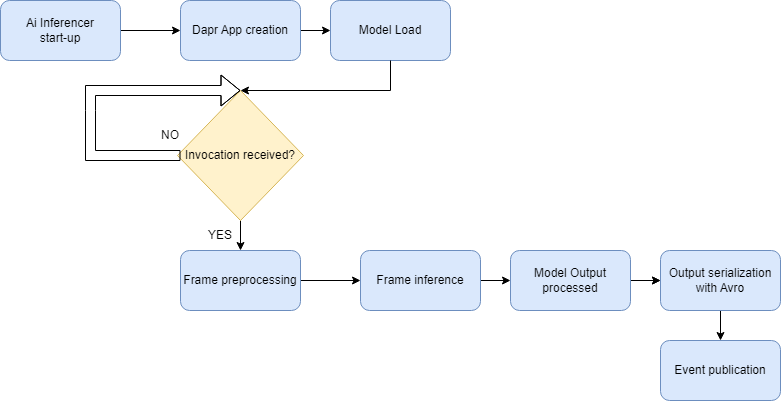


Figure 2.0.1: AI inferencer microservice flow.

## Microservice 3: Rules-Engine

The Rule Engine is a .NET-based microservice that scrutinizes and processes object detections stemming from the inference microservice. With MQTT and Dapr, it receives events, puts configurable logic rules to work, and generates alerts (or not) based on outcomes, these alerts are pushed to the next microservice, the “Alerts-API”.

Delving in the Rule engine’s workflow:

1. **Receiving events:** By subscribing an MQTT topic via Dapr, the Rule Engine acquires frames encompassing all detections made previously in the inference microservice.
2. **Event Handling**: Upon an event arrives, the Rule Engine springs into action, this event morphs into a Mediator command dubbed “Analyze Detection”, containing the detection information vital to launch an alert.
3. **Rules at play:** Dictated by a configuration like figure 3.0.1, dictated by a configuration outlining logic rule for each detection label. Factors like the number of objects detected or a precision threshold may come into play.



Figure 3.0.1: Rules engine configuration example

1. **Alerts arise**: Should the logic rules align with detection analysis and processing results, an alert is created and pushed to the Alerts-API Queue.

We can see a simplified flow class diagram for this microservice in the figure 3.0.2:

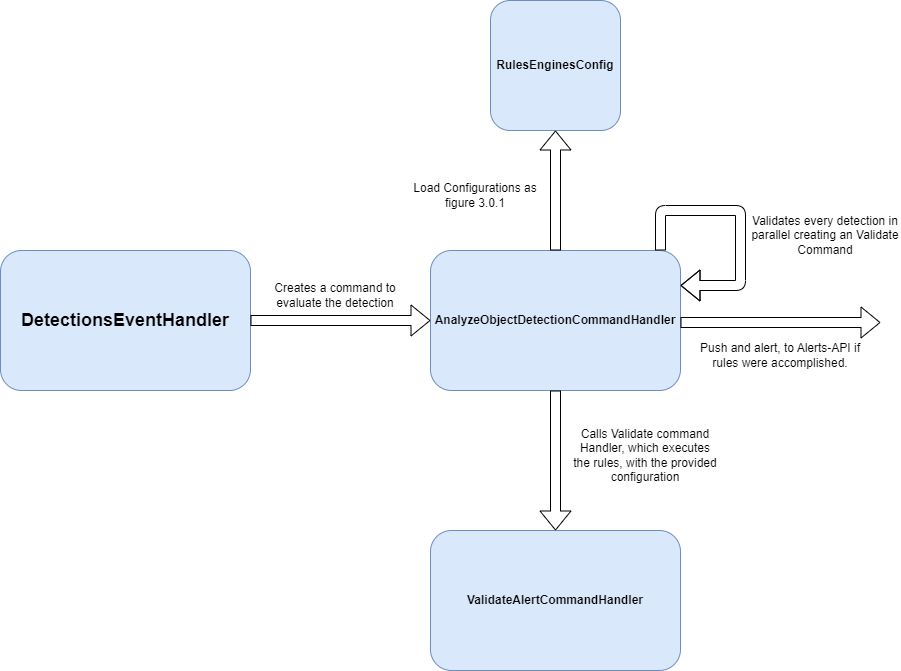


Figure 3.0.2: Rules-Engine flow class diagram simplified

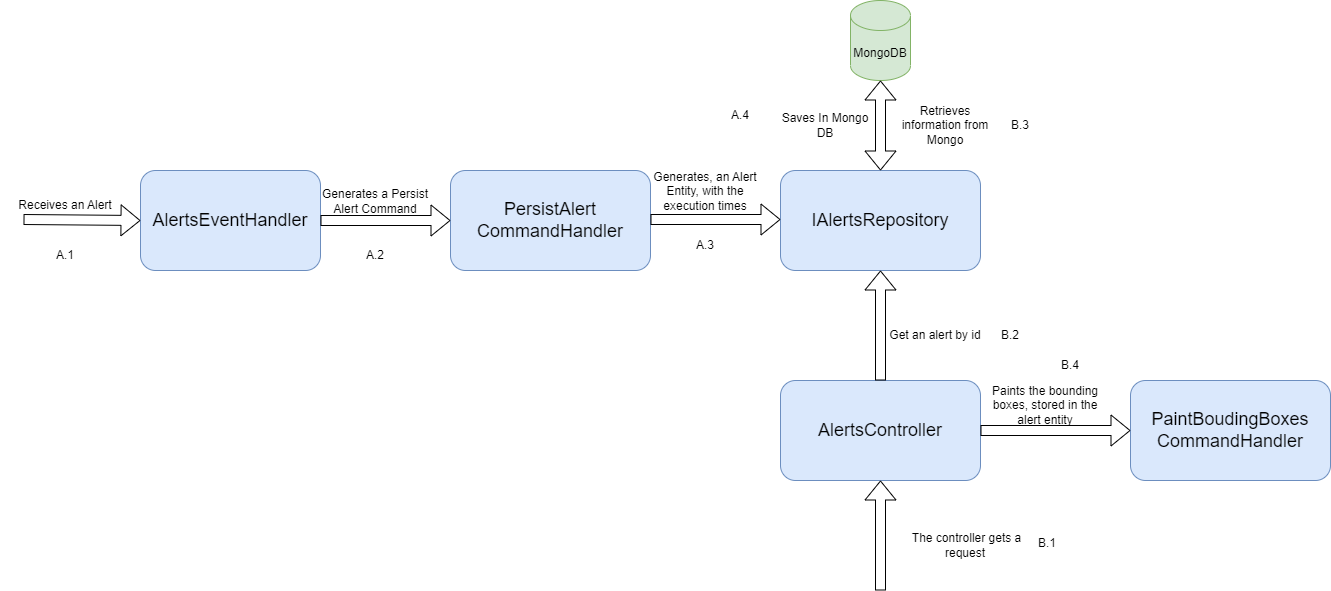
## Microservice 4: Alerts-API

The "Alerts-API" .NET based microservice is another crucial element within the system. Its main responsibilities are processing alerts generated by the Rule Engine, calculating execution times, storing relevant information in MongoDB, and providing access to stored data through an API.

Delving in the Alerts-API flow, and responsibilities:

1. **Receiving alerts**: The microservice receives alerts generated by the Rule Engine.
2. **Calculating execution times**: It calculates the time taken from video capture to alert generation.
3. **Storage command generation**: The microservice creates a storage command to persist the resulting information.
4. **Storing data in MongoDB**: It stores the data, including alerts and calculated execution times, in MongoDB.
5. **Exposing data through an API**: The microservice provides access to the stored information by exposing an API with two primary services:
6. a. **List alerts**: This service lists all alerts in chronological order based on their generation timestamps.
7. b. **Retrieve alert by ID**: This service retrieves a specific alert by its ID, including the detection image. It also draws bounding boxes around each detected object at the time of the alert.

A simplified flow class diagram for this alerts-api microservice, figure 3.0.3:

Figure 3.0.3: Alerts-API flow class diagram simplified

The Alert Management microservice seamlessly communicates with the Rule Engine microservice using the MQTT queues to receive alerts and related information. This efficient exchange of data enables the microservice to carry out its responsibilities of processing, calculation, and storage.

## Microservice 5: Alerts-UI

The "Alerts-UI" microservice is the final component in this system of multiple microservices. Developed as an MVC in .NET Core, it serves as the user interface for the alert management system. The microservice interacts with the previously mentioned services using service invocation, displaying an overview of recent alerts and providing detailed alert information upon user request.

### Alerts-UI Functionality and use cases:

1. **Overview of recent alerts**: The Alerts-UI microservice polls the "List alerts" service every two seconds, retrieving the latest 15 results. This overview displays the alert type, capture timestamp, alert generation timestamp, time taken for alert generation, the ID of the device that recorded the event, and an icon representing the "severity" based on the object detection precision.
2. **Detailed alert view**: When a user clicks on a specific alert, a side panel opens, providing more information:
3. a. **Detection image**: The image of the detection with bounding boxes around the detected object(s).
4. b. **Execution time information**: A detailed breakdown of the execution time for each step of the process.

The Alerts-UI microservice integrates seamlessly with the Alert Management microservice, enabling efficient access to the services provided by the latter. By leveraging Dapr service invocation, Alerts-UI ensures smooth and consistent data retrieval, facilitating the display of recent alerts and detailed information.

Extending the system

## Adding a new rule to the Rule Engine

The Rule Engine microservice is a powerful and flexible component of the system, designed to apply configurable logic rules to object detections. To accommodate varying requirements and use cases, the Rule Engine allows the addition of new rules. The following section introduces creating and implementing a new rule in the Rule Engine.

When developing a new rule for the Rule Engine, the primary goal is to define the logic that governs whether an alert should be generated based on specific detection attributes or conditions. The steps to create and integrate a new rule into the Rule Engine include:

1. **Identify the rule requirements**: Clearly define the purpose of the rule and the conditions under which an alert should be generated. Consider aspects such as the type of objects detected, the number of objects, the precision threshold, or any other relevant detection attributes.
2. **Design the rule logic**: Develop the logic that dictates how the rule will analyze the detection data and determine if the conditions for generating an alert are met. This can involve using programming constructs such as conditional statements, loops, or other suitable techniques.
3. **Implement the rule**: Integrate the designed rule logic into the Rule Engine, ensuring compatibility with the existing rule application process. This may involve updating the configuration to include the new rule, adjusting the rule application mechanism, or modifying the event handler as needed.
4. **Test the rule**: Verify that the new rule works correctly and as expected by testing it with various scenarios and detection data. Ensure that the rule generates alerts when the conditions are met and doesn't produce false alerts when they aren't.

By following these steps, you can successfully create and integrate a new rule into the Rule Engine, extending its capabilities and allowing it to handle more diverse object detection scenarios and alert generation requirements.

## In-Depth Implementation Guide, for Adding a New Rule to the Rule Engine

The following guide provides a more in-depth explanation of how to implement a new rule within the Rule Engine. By following these steps, you can ensure that your rule is well-integrated and functions correctly within the existing system.

This example demonstrates the implementation of a new rule in the Rule Engine that checks for the simultaneous detection of two different classes in a single frame: And how to create a new alert called PersonDrivingScooter. This scenario could be useful for identifying situations where a person is riding a scooter, for example.

Step by Step Description:

1. **Rule requirements analysis**: The goal of this rule is to generate an alert when both a person and a scooter are detected in the same frame. The rule should consider the detected objects' classes and their presence in the same frame to determine whether to generate an alert.
2. **Designing the rule**: The rule's logic should first check if the frame contains at least one detection of each class (person and scooter). If this condition is met, the rule will generate an alert.
3. **Updating the configuration**: Update the Rule Engine's configuration to include the new rule, specifying the alert that it applies to both person and scooter detection labels. Ensure that the configuration reflects the intended rule logic. In the figure 4.2.1 you can see the resultant json.



Figure 4.2.1: Rules json configuration for PersonDrivingScooter rule.

4. **Implementing the rule logic**: Implement the rule logic by adding a new command as you can see in Figure 4.2.2. and implement he command Handler for this rule with the correspondent business logic as the figure 4.2.3

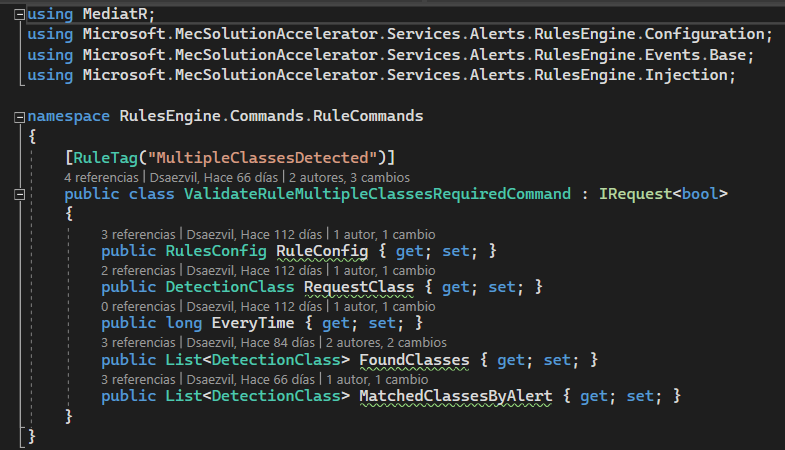


Figure 4.2.2: New Rule command, MultipleClassesDetected

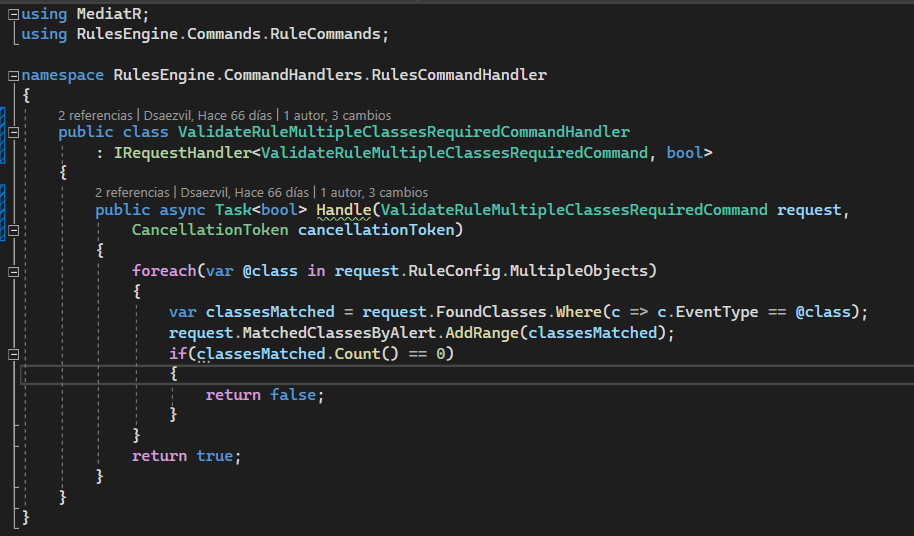


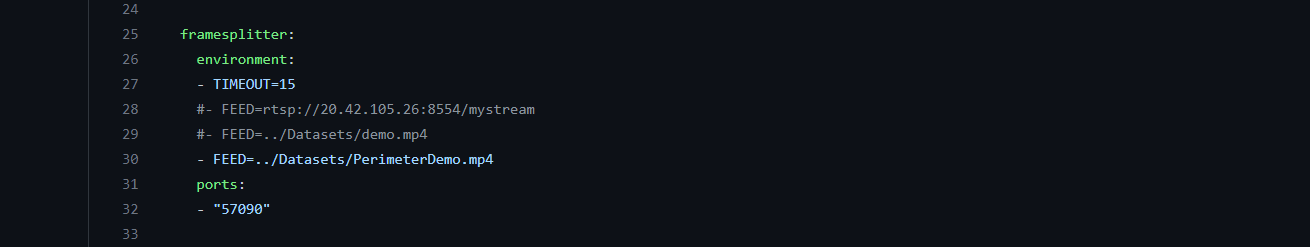
Figure 4.2.3: New Rule command Hander, ValidateRuleMultipleClassesRequiredCommandHandler

By following this example guide, you can successfully implement a new rule to detect situations where a person is riding a scooter, enhancing the Rule Engine's capabilities and providing valuable insights into the analyzed detections.

# How to provision a new feed in Frame Splitter

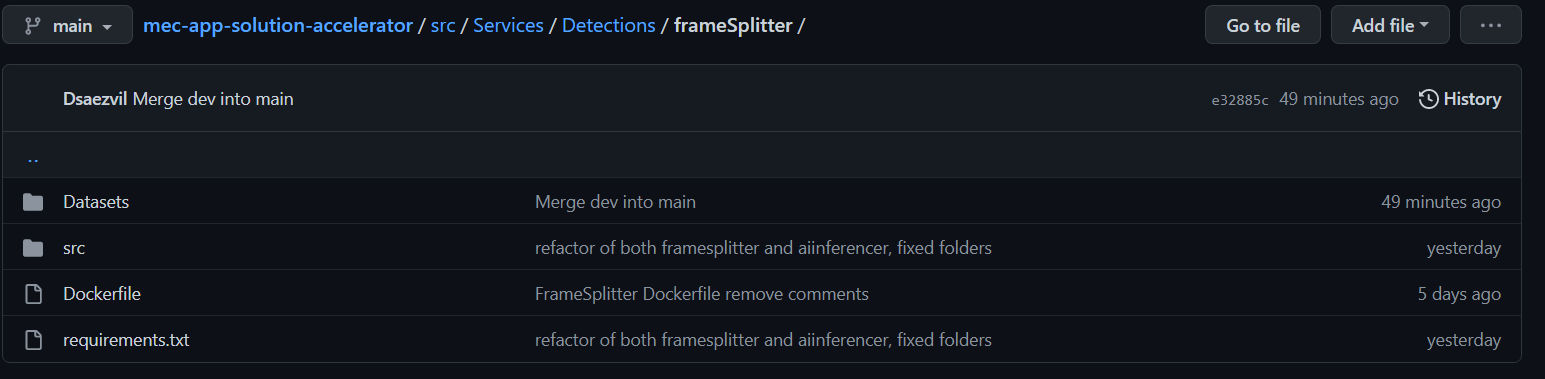
New Feed Provision   
  
The current solution supports the usage of custom images and videos for testing purposes.  
If you want to run your own video in Kubernetes, please refer to [Create RTSP Server](https://github.com/Azure/mec-app-solution-accelerator/blob/main/docs/HOW_TO_CREATE_RTSP_SERVER.MD)

Docker Compose  
  
In docker-compose.override.yml find the frame splitter service.

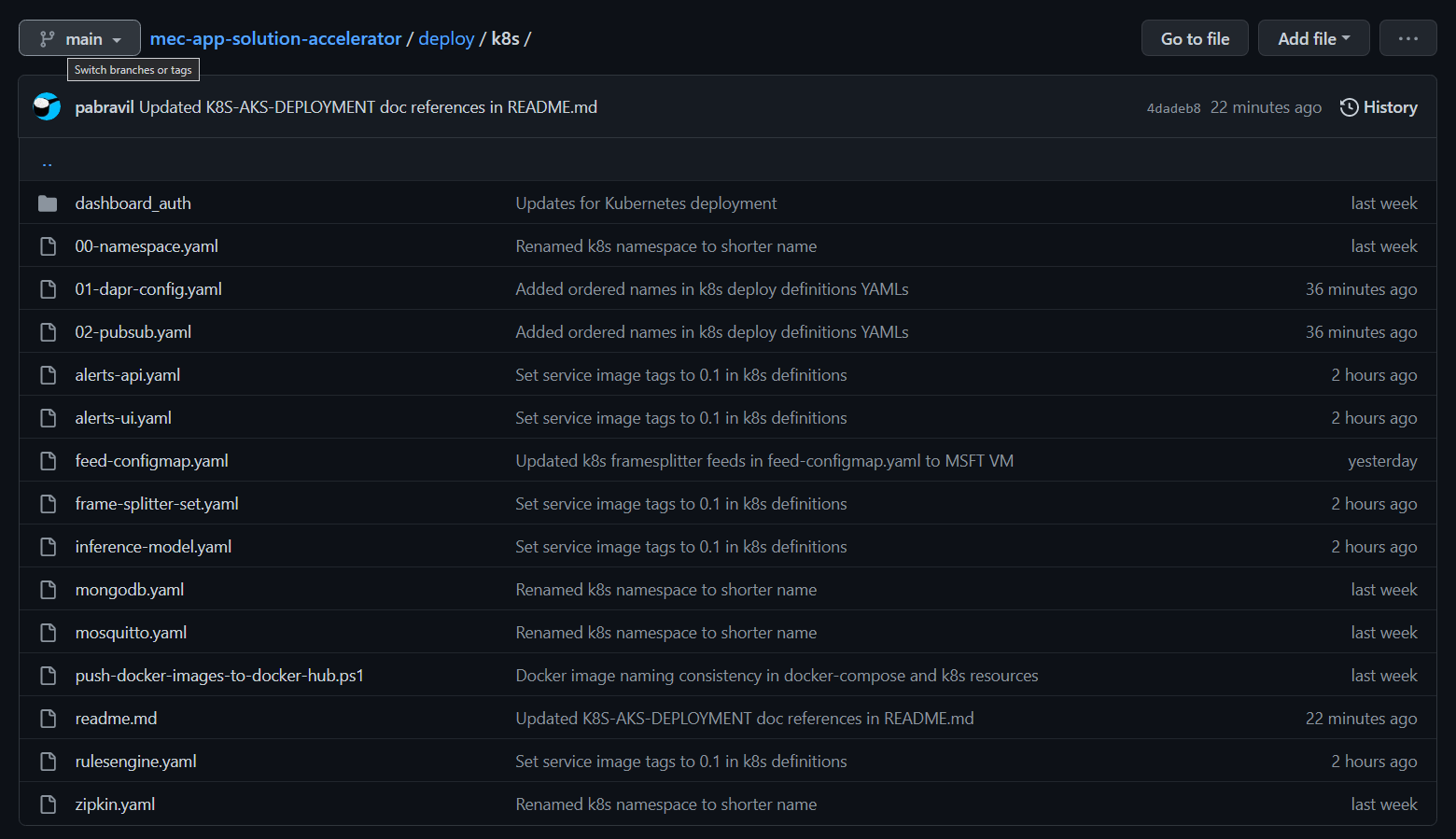
  
Figure 5.0.1: Feed configuration docker-compose-override

Comment the current feed line:

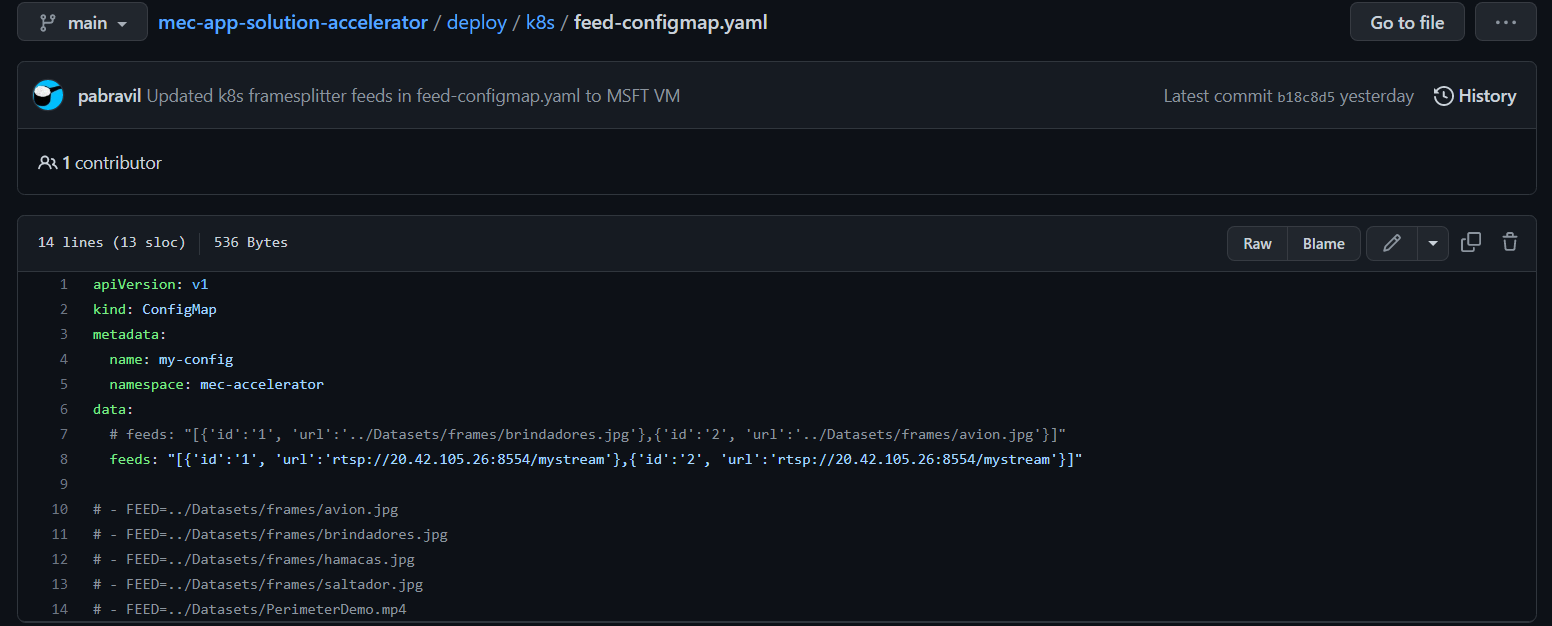
  
  
Uncomment and modify the rtsp url with your ip and port:  
  
  
  
It is also possible to run your own videos using compose, if you add your video to datasets folder, and reference it correctly.

  
Figure 5.0.2: Folder structure to add custom file.

  
  
Kubernetes  
  
In this case it is only possible to add a new feed using RTSP.  
  
Navigate to the k8s folder.

  
Figure 5.0.3: deploy/k8s folder.

Modify feed-configmap.yaml:

  
Figure 5.0.4: feed-configmap.yaml.

Add your new url to the tag url in the dictionary structure.  
  
IMPORTANT: you must take into account that these feeds are consumed by one unique frame splitter pod. If you want two rtsp to be consumed, you have the set the replicas properly, and the system will automatically assign one rtsp feed per frame splitter replica.  
  
If you want to modify an id or add a new one, follow the format proposed in the next line:  
  


# How to replace the default AI model (Yolo) by a new custom DeepLearning mode

## Pytorch model

AI Inferencer microservice is the one responsible of hosting and executing the model. The current AI inference is preparad for running pytorch models.

Even though the microservice is ready to execute pytorch models, modifications may be performed to run a custom model and generate the desired events.

First, we must navigate to invoke-sender-frames.py and replace the line marked in red and depicted in figure 6.0.1

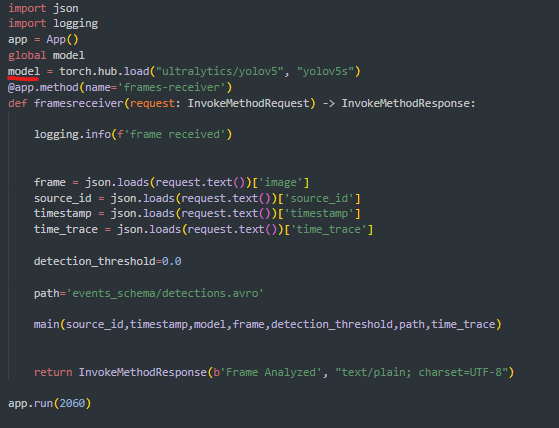


Figure 6.0.1: src/Services/Detections/ai\_inference/invoke-sender-frames.py

This line of code should be replaced with the code depicted in figure 6.0.2

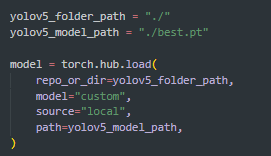


Figure 6.0.2: Custom model load.

## Adding a model with no pytorch implementation

In case of need of loading a different model, not using pytorch, the Dockerfile should be replaced by the one ready for running the model. Line 5 of the original Dockerfile can be commented, which is the responsible of installing pytorch-cpu implementation, and add the new needed packages to the requirements.txt file, such as tensorflow.

Texto

Descripción generada automáticamente

Figure 6.1.1: ai inference dockerfile.

Modification of line 8 is mandatory to be adapted to load models that rely on frameworks different than pytorch.

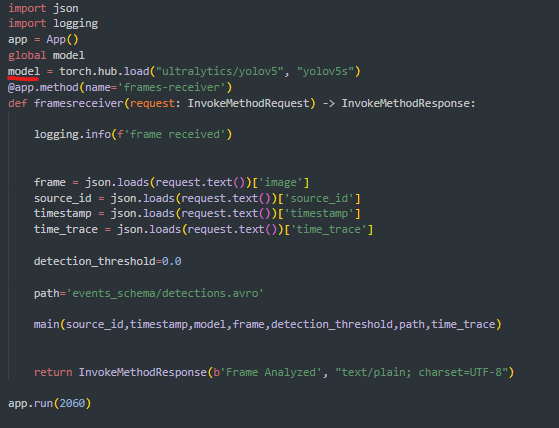


Figure 6.1.2: invoke-sender-frames.py

After this modification, user should go to inference.py and modify line 49 to perform inference as needed for the new model/framework.

Also, modification of line 53 is needed to load and process the output of the model.

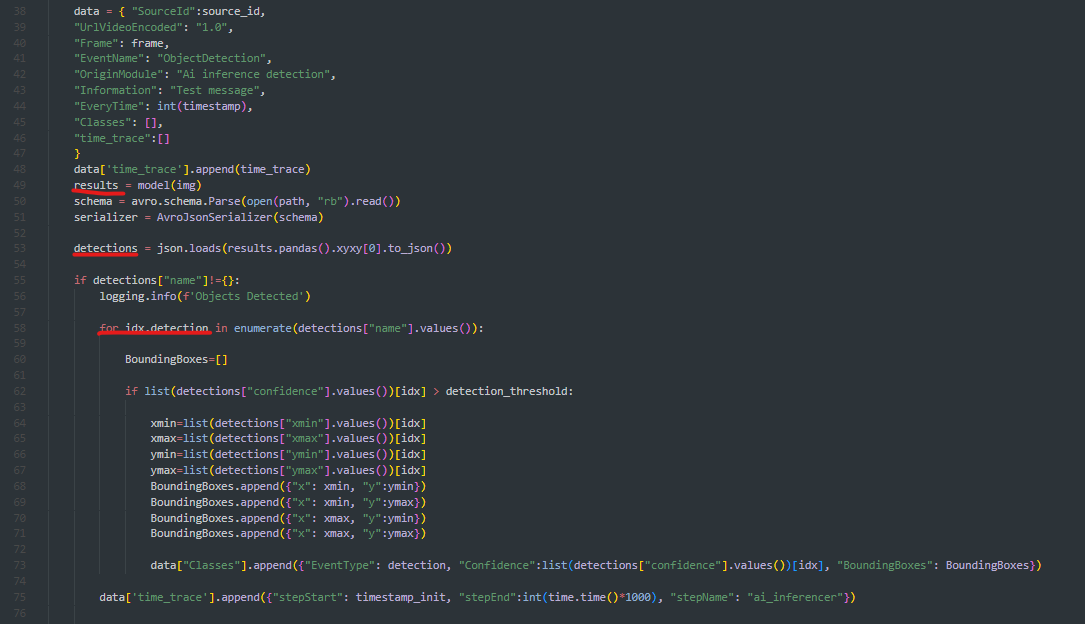


Figure 6.1.3: inference.py lines 44-56.

Finally, modification of lines 58 to 71 could be needed to iterate over the detections, get the confidence value, the name of the detection, the coordinates of the bounding box and populate the output json. Between lines 68 to 73, the json is populated with the name associated, the confidence, and set up de bounding box format to xmin, y min, xmax, xmax following the format presented in the event schema.

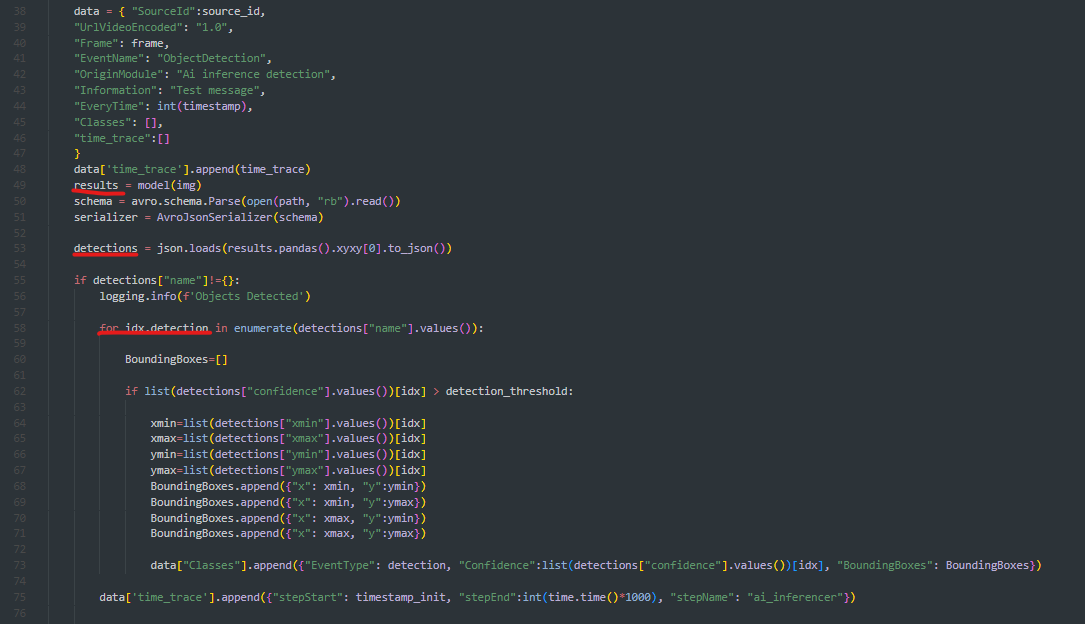


Figure 6.1.4: inference.py lines 58-75.

If a new event schema is needed, the event schema present in <https://github.com/Azure/mec-app-solution-accelerator/tree/main/src/Services/Detections/ai_inferencer/events_schema> have to be modified, as well as the code between lines 58 and 75.