DMS
Driver Monitoring System

Authors

Ola Mohamed Raafat Hassan
Nada Mohamed Metwaly
Nada Atia Eid Atia
Mariam Ahmed Fathy
Amr Ramadan Agamy
AbdElrhman Mamdouh Khalil
Ali Emad Abdellatif Abdo
Ahmed Mohamed Saeed Elgohary
Ahmed Mohamed Roshdi
Mostafa Sayed Ahmed Taha

Supervisor

Dr. Aida Ali El-Shafie

Sponsered by Ejad

Preface

The journey to creating this graduation book has been an enlightening and transformative experience for all involved. As we present this compilation, we celebrate not only the completion of our academic pursuits but also the progress we've made in the field of Driver Monitoring Systems (DMS) projects.

This book showcases our hard work, creativity, and teamwork. Throughout the project, we have learned a lot about DMS, which are systems designed to make driving safer by monitoring the driver's behavior. Our goal has been to help reduce accidents and save lives—lives we have seen firsthand how painful it is when they are lost. On December 10th, 2023, we lost our colleague, sister, and one of the kindest people one could ever meet, Ola Raafat. That day, we were all in the same car, witnessing with our own eyes how crucial a DMS is in our everyday life. After the accident, we decided to spare no effort to prevent such incidents from happening ever again and to dedicate our project to providing safer roads in memory of our lost sister.

The skills and knowledge we have gained will help us make important contributions to this field. We are dedicated to spreading the knowledge we have acquired through this journey and to paving the way for future engineers to build more robust systems for safer roads.

As we move forward into our careers, we feel a strong sense of responsibility. The work we do in DMS will directly impact the safety of drivers and passengers. We are committed to continuing our efforts with the same passion and dedication that have brought us this far.

Congratulations to the graduating engineers, and here's to a future full of innovation, progress, and safer roads for everyone.

Sincerely,
The Authors
Faculty of Engineering
Alexandria University

Acknowledgement

We are deeply grateful to Allah for His countless blessings upon us, and for granting us sufficient knowledge that enabled us to work on and complete this project, to benefit others thereafter, despite the challenges we faced.

Nothing happens without the presence of those who support you through it, witnessing every step and success; to them, we owe our deepest gratitude.

We extend all our respect and best wishes to our supervisor, Dr. Aida El-Shafie, for her continuous support, steadfast presence by our side, and responsiveness to our needs.

To everyone at Ejad Company, we are immensely grateful for their warm relationship and support, especially to our mentors Eng. Mahmoud Ghreeb and Eng. Ahmed Zain, for their guidance and support.

To our teammate, Ola Raafat, who started our first steps, and was the reason for us to continue and complete the project's journey, her spirit will remain an inspiration to us all.

To Nada Metwally, who started the journey with us and had an invaluable impact on defining the project's path.

To Mahmoud Marey, Michael Naeem, and Mostafa Sultan, who helped us with the AI and their kind intentions in working on the future work part, they have all our gratitude.

To all our team members, whose love, unity, and cooperation we sought and found.

To our technical teams Mind Cloud, Robot-Tech, and M.I.A, where our technical journey began, the communities that helped us achieve this level of knowledge.

Lastly, we would like to thank our families and friends for their unwavering support and encouragement throughout this journey. Their love and understanding have been a source of our strength.

Contents

List of Tables

List of Figures

Abstract

Driver fatigue and distraction pose significant risks to road safety, emphasizing the critical need for Driver Monitoring Systems (DMS). These systems are pivotal in detecting and mitigating driver impairment, thereby improving overall road safety standards. However, existing solutions often face challenges related to real-time responsiveness and accuracy, which are crucial for effectively preventing accidents caused by distracted or drowsy driving.

In response to these urgent challenges, this project aims to develop an innovative driver monitoring system. The primary objective is to create a robust system capable of classifying driver states into four categories: focused, sleepy, distracted, and drowsy. The system achieves this classification using a combination of computer vision, machine learning and deep learning techniques. Key features include eye closure detection, gaze tracking, yawn detection, and head pose estimation, all fed from an infrared camera to ensure functionality under various lighting conditions.

This system harnesses the capabilities of the Texas Instruments TDA4VM System on Chip (SoC), renowned for its high-performance computing tailored for automotive applications. By leveraging the TDA4VM SoC's dual-core 64-bit Arm Cortex-A72 processors, DSPs, and deep-learning accelerators, the system delivers high-performance and real-time driver monitoring. All processing is conducted on the edge, directly on the TDA4VM SoC, eliminating the need for distributed processing on external devices such as laptops or cloud servers. This design ensures the system is ready for seamless integration into smart car systems, providing a self-contained real-time monitoring solution.

Experimental results demonstrate the system's ability to accurately detect and classify driver states with a final accuracy of 90% and a processing rate of 17 frames per second (FPS) on average. The results indicate significant improvements in both detection accuracy and system responsiveness, confirming the efficacy of the TDA4VM SoC in automotive safety applications.

Moreover, we provide a mobile application and communication module to fit various use cases. This addition will allow for remote monitoring and alerts, facilitating scenarios such as fleet management or caregiver notifications. The mobile app will enable users to view real-time driver status updates, receive alerts for potential fatigue or distraction events, and access historical data for analysis and reporting purposes.

By combining advanced hardware capabilities with user-friendly mobile applications, this integrated solution not only aims to elevate safety standards within vehicles but also sets a precedent for future innovations in automotive safety technology.

Introduction

Car travel, despite its convenience, comes with significant risks. Each year, approximately 1.35 million people lose their lives in car crashes, and many more suffer serious injuries. This reality underscores the importance of implementing advanced safety technologies in vehicles. Among these, Driver Monitoring Systems (DMS) have emerged as crucial tools in enhancing road safety by monitoring driver behavior to prevent accidents caused by distraction and drowsiness.

The urgency for adopting DMS technology is reflected in recent regulatory and safety assessment developments. The European New Car Assessment Programme (Euro NCAP), renowned for its progressive safety standards, now requires DMS for vehicles to achieve a five-star safety rating. Furthermore, the European General Safety Regulation has mandated DMS technology for all new cars, vans, trucks, and buses starting in 2024. Across the Atlantic, the National Highway Traffic Safety Administration (NHTSA) recommends DMS as an effective means for driver engagement in Level 2 vehicles. Moreover, the US Congress is advocating for the inclusion of DMS in all new cars within six years. These regulatory pushes highlight the critical role of DMS in reducing the staggering number of fatalities caused by distracted driving—over 3,100 deaths per year in the US alone.

DMS technology has evolved significantly over the years. Early systems focused on indirect indicators of driver distraction and drowsiness, such as lane swerving and abnormal acceleration patterns. However, with the advent of Level 2 autonomous vehicles equipped with features like automated lane centering and adaptive cruise control, these secondary signals proved insufficient. Recognizing this gap, Euro NCAP now prioritizes advanced DMS that utilize direct monitoring techniques, such as eye tracking and infrared cameras, to provide real-time assessments of driver attentiveness.

The capabilities of modern DMS are impressive. These systems can detect whether a driver is inattentive or drowsy and generate alerts to refocus the driver's attention. If these alerts fail, some systems are even capable of taking control and safely stopping the vehicle. Beyond safety, DMS with eye tracking also offers convenience features, allowing drivers to control certain functions with their gaze and gestures, enhancing both security and comfort.

The market for DMS is rapidly expanding. In the first nine months of 2022 alone, installations of DMS and Occupant Monitoring Systems (OMS) surged by 130% year-on-year, with visual solutions becoming the mainstream choice. Leading models like the Trumpchi GS8, NIO ES6, and AION Y showcase the integration of DMS and OMS, underscoring the industry's commitment to these technologies.

As DMS and OMS technologies advance, they are increasingly integrated with intelligent driving systems and extended to enhance the overall driving experience. Features like personalized cockpit settings, multimodal interaction, and facial recognition are transforming how drivers interact with their vehicles. Models like the MIFA 9 and Changan CC PLUS exemplify these advancements, offering a glimpse into the future of intelligent and safe driving environments.

In this project, we present a comprehensive camera-based DMS system that achieves real-time performance. Reviewing the literature, we analyzed related work and spotted the most common challenge in this field, namely, hardware deployment. Hitting the real-time bar has been our main focus and throughout this

document, we present how we solved this challenge without relying on Graphics Processing Units (GPUs). The rest of the document is arranged as follows:

Section 1 delves into the intricate components and methodologies of our driver monitoring system, exploring the core AI functionalities such as face feature extraction, distraction analysis, and drowsiness detection. Section 2 breaks down our Software In the Loop (SIL) implementation, illustrating how we simulate and test our AI algorithms in a controlled software environment. Section 3 provides a detailed discussion on hardware selection, focusing on camera specifications and the processing platform. Section 4 explains the integration of hardware components, particularly the camera, with the BeagleBone AI-64, and the steps involved in deploying the application. Section 5 describes the deployment of AI models, including deep learning model selection and deployment, optimization techniques, and compilation. Section 6 is dedicated to benchmarking AI algorithms, highlighting execution time measurement and performance optimization strategies. Section 7 examines the communication module with a focus on communication interfaces, application performance monitoring, and designing a metrics monitoring and alerting system. Section 8 shifts to the development of a mobile application using Flutter, detailing its benefits, project structure, user interface design, and integration of data visualization with the DMS. Section 9 provides an in-depth comparison of build methods for Linux images, discussing the methodology of Buildroot and Yocto. Finally, Section 10 summarizes our findings, the overall impact of our driver monitoring system, and outlines potential future developments and improvements.

1 AI System Solution

The main application continuously analyzes the driver's state, focusing on drowsiness detection, distraction detection, and age and gender prediction using computer vision methods. The system captures and processes video data in real-time by utilizing infrared (IR) or RGB images.

Our approach simplifies the process by using a deep learning model for face detection and classical machine learning and image processing techniques for further analysis. After detecting faces, the system extracts facial landmarks. These landmarks are then used to estimate head pose, monitor eye and mouth aspect ratios, and track eye gaze direction to identify signs of drowsiness and distraction.

This method is computationally efficient, enabling real-time performance, and adaptable to different lighting conditions.

1.1 Face Feature Extraction [10]

In driver monitoring systems, the extraction of facial features plays a critical role in ensuring accurate and reliable detection of driver states such as drowsiness and distraction. This process involves several key steps, each contributing to the system's overall effectiveness. The system's design is depicted in Fig. 1. The DMS employs a camera (imager) as the input image source, and the captured image is used for face detection followed by facial landmarks estimation. These extracted facial landmarks serve as crucial features for estimating head pose and identifying instances of closed eyes and yawning, indicative of drowsy driving.

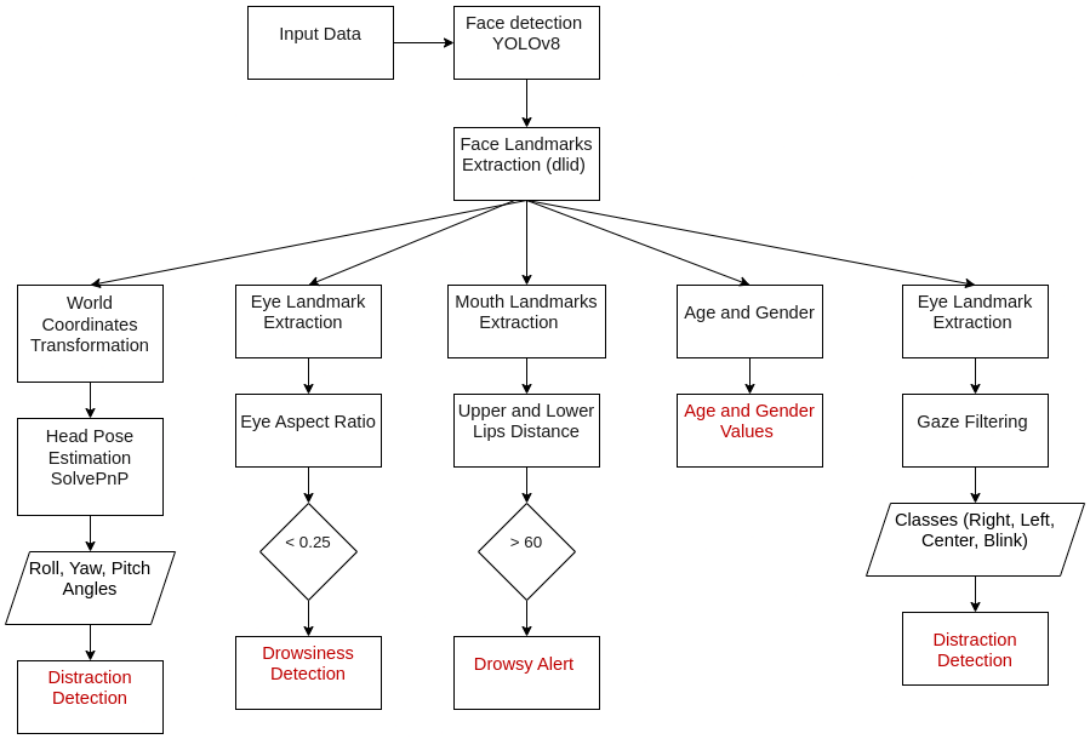


Figure 1: Main AI application system design

1.1.1 Face Detection Using YOLOv8n [31] [33]

The first step in facial feature extraction involves detecting the driver's face using YOLOv8, or You Only Look Once version 8 which will be replaced later by YOLOv5 due to some deployment problems. YOLOv8

is a 1-stage detection algorithm, similar to the SSD detector, offering both satisfactory performance and fast processing speed, specifically the nano version implemented by Ultralytics. As shown in Fig. 2, the nano version is chosen for its efficient use of computational resources without compromising on detection accuracy, making it ideal for real-time applications in resource-constrained environments.

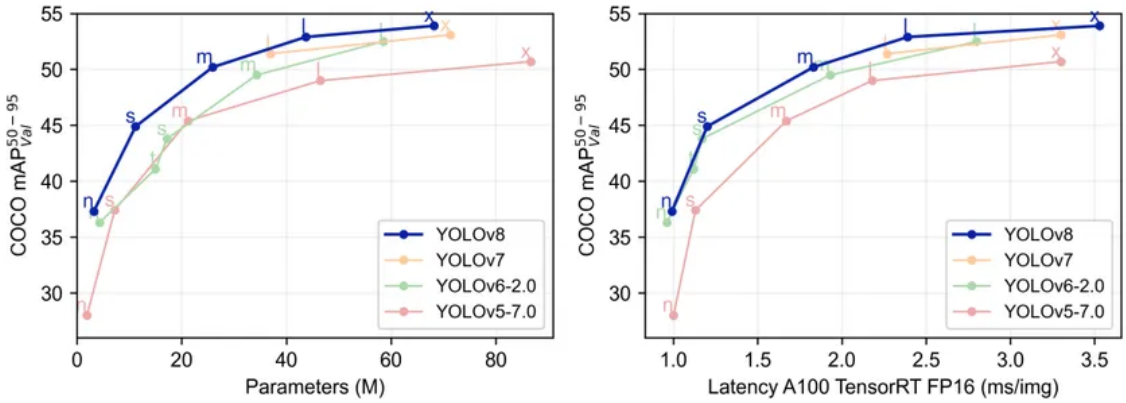


Figure 2: YOLO models comparison

Trained on the comprehensive WIDERFACE dataset. The WIDERFACE dataset is a large and diverse collection of over 32,000 images with nearly 400,000 labeled faces, designed to cover a wide range of challenging real-world scenarios for robust face detection.

The model generates bounding boxes that precisely outline each detected face, defined by coordinates for the top-left corner and dimensions. These bounding boxes are accompanied by confidence scores, quantifying the model's certainty that the box contains a face.

Figure 3: YOLOv8 real-time outputs

PyTorch was selected for training YOLO over TensorFlow for deployment in C++ environments and its capability to load models directly into memory, resulting in faster inference times crucial for real-time applications. PyTorch's support for seamless deployment across GPUs, CPUs, and custom accelerators ensures

scalability and performance optimization. Additionally, its superior support for multithreading enhances efficiency in handling concurrent tasks, making it suitable for high-throughput applications.

1.1.2 Facial Landmarks Extraction [9] [11] [24] [2]

Facial landmark extraction has made significant strides in recent years, with deep learning techniques like Openface and Retinaface at the forefront. However, in our DMS, the high computational costs associated with face detection using deep learning present a challenge. To mitigate this, we have chosen Kazemi's fast facial landmark extraction algorithm, which has been integrated into the Digital Library (Dlib) library. Kazemi's algorithm, based on random forests, provides an excellent balance of speed and performance, making it suitable for the DMS, which needs to function efficiently in an embedded environment rather than a desktop setup.

Dlib's first step is face detection, then applying facial landmark extraction built on Kazemi's algorithm's principles. Dlib includes two face detection methods:

1. **HOG + Linear SVM Face Detector**
   This method accessible via `dlib.get_frontal_face_detector()`, is known for its accuracy and computational efficiency.

2. **Max-Margin (MMOD) CNN Face Detector**
   This method is highly accurate and robust, capable of detecting faces under varying viewing angles, lighting conditions, and occlusion.

For our system, we replace the HOG + Linear SVM face detection method with the YOLOv8 face detection model. YOLOv8 provides more accurate and faster face detection, which is crucial for the real-time requirements of our DMS. By leveraging the precise bounding boxes generated by YOLOv8, we can then feed these into Dlib for accurate facial landmarks. This is crucial for analyzing the driver's state, and detailed explanations of their use will be provided in subsequent sections.

We use the `shape_predictor_68_face_landmarks_GTX.dat.bz2` model, a highly optimized and accurate facial landmark detector. This model is trained on the ibug 300-W dataset, a comprehensive benchmark dataset containing a variety of images with annotated facial landmarks. The GTX version of the model utilizes advanced training strategies to enhance performance and accuracy, including data augmentation, robust optimization techniques, and meticulous fine-tuning to ensure the model can accurately predict 68 key facial landmarks under diverse conditions.

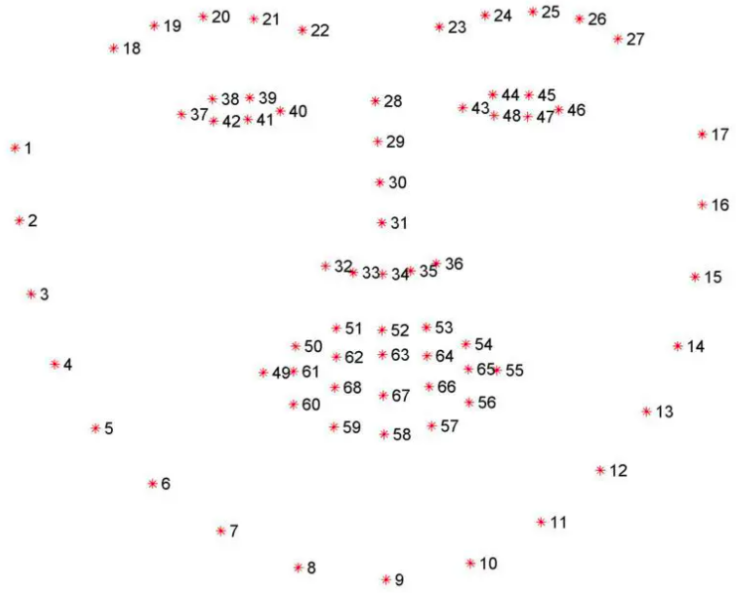


Figure 4: Facial 68 landmarks to localize various regions of the face, including eyes, eyebrows, nose, mouth, and jawline

The output of this stage is the key 68 landmarks, which are then fed into other algorithms for further analysis. This integration ensures that our DMS can reliably monitor the driver states.

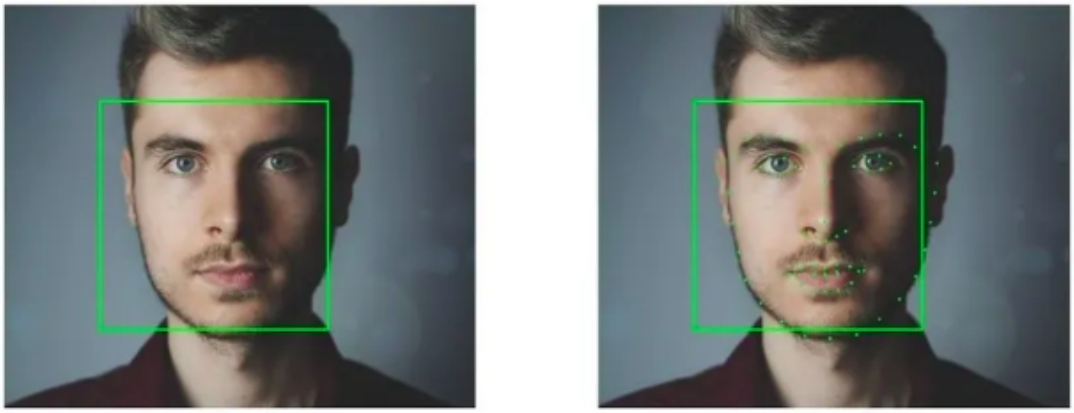


Figure 5: Face detection and landmarks extraction

1.1.3 Age and Gender Estimation [21] [3] [11] [15]

In the context of driver monitoring systems, estimating the age and gender of drivers is crucial for several reasons:

1. It enables personalized driving experiences, where vehicle settings such as seat position, temperature, and mirror angles can be automatically adjusted based on the identified driver's age and gender preferences.

2. Such estimations contribute to enhancing safety measures by tailoring alerts and warnings to suit the specific demographic characteristics of the driver.

3. Demographic data like age and gender aids in developing targeted marketing strategies for in-car services and products, thereby improving customer satisfaction and engagement.

Models Provided by Dlib and Cydral Technology

The pre-trained models for age and gender estimation used in this system are provided by Dlib and are available for free under the Creative Commons Zero v1.0 Universal license by Cydral Technology. These models offer robust performance due to their training on large, private datasets and their sophisticated architectures.

Gender Classifier: `dnn_gender_classifier_v1.dat.bz2`

This gender classifier model was trained on approximately 200,000 face images. The training followed the architecture and settings outlined in the paper *Minimalistic CNN-based ensemble model for gender prediction from face images*. Key features include:

- **CNN Ensemble Model:** The use of a minimalistic CNN ensemble approach helps improve classification accuracy by leveraging multiple CNNs.
- **Private Dataset:** The extensive dataset ensures that the model generalizes well to diverse real-world scenarios.

Age Predictor: `dnn_age_predictor_v1.dat.bz2`

This model leverages a ResNet-10 architecture and is trained using a private dataset of about 110k different labeled images. It ages by categorizing faces into one of 81 classes, representing ages 0 to 80 years. The age predictor model builds on the initial research documented by Z. Qawaqneh et al. in *Deep Convolutional Neural Network for Age Estimation based on VGG-Face Model*. Significant improvements have been made to enhance its performance. Notable features include:

- **ResNet-10 Architecture:** This backbone is known for its efficiency and effectiveness in various computer vision tasks. It enables the model to capture intricate details necessary for precise age estimation.
- **Extensive Training Dataset:** The model was trained on a dataset of about 110,000 labeled images, ensuring a high degree of accuracy.
- **Optimization and Data Augmentation:** During training, an optimized pipeline and data augmentation techniques were employed to enhance the model's robustness and accuracy.
- **Age Estimation:** Utilizes a softmax layer to generate a probability distribution over the 81 age classes. The predicted age is computed as a weighted sum of these probabilities.

In our DMS, we have integrated two pre-trained models into a single Python module called "age_gender". This module utilizes previously detected faces with YOLOv8 and their corresponding 68 facial landmarks to estimate the age and gender of individuals in images. The prediction method of this module returns a list of detected faces along with their estimated age and gender, including confidence levels for each prediction.

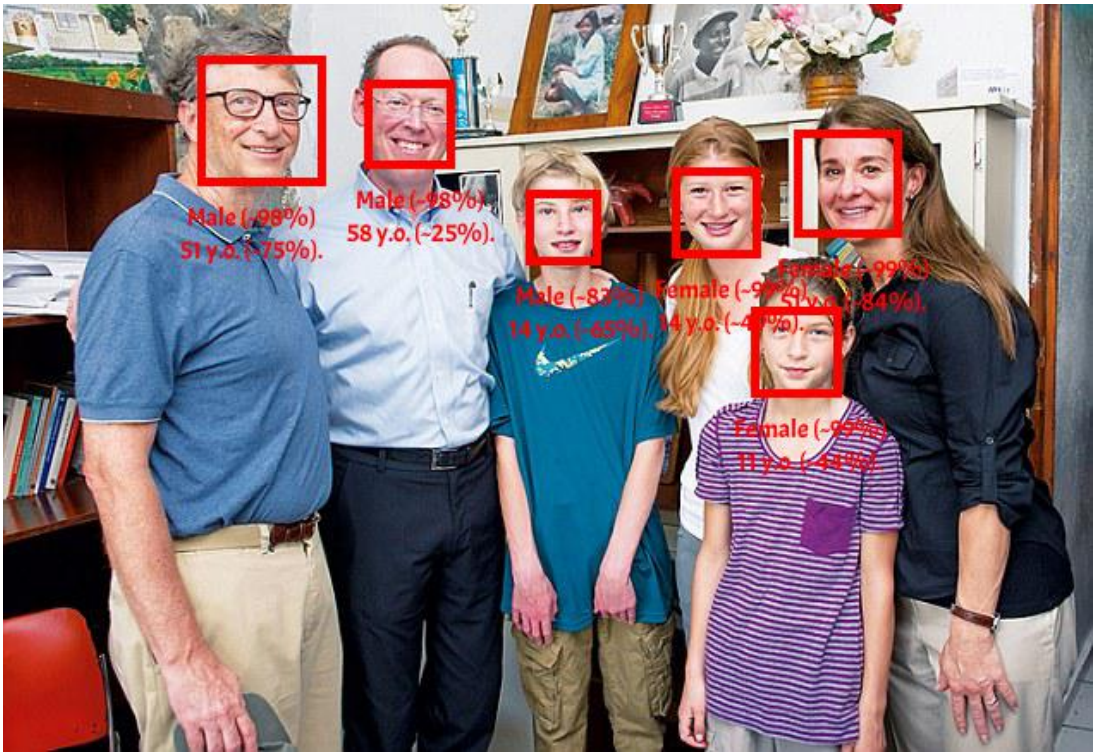


Figure 6: Age and gender estimation outputs and confidences

The module is implemented using "pybind11", a lightweight header-only library that exposes C++ types in Python and vice versa. The module integrates the age and gender prediction functionalities using the Dlib library.

1.2 Distraction Analysis

The primary aim of a DMS is to detect signs of driver distraction, inattention, and fatigue, which are significant contributors to road accidents. Distraction analysis within DMS involves the use of various techniques to monitor and interpret the driver's behavior and attentiveness. Two essential components of distraction analysis are Head Pose Estimation and Eye Gaze Estimation. Both of these techniques play a pivotal role in assessing whether the driver is focused on the road or is distracted by other activities.

1.2.1 Head Pose Estimation [14] [1]

The theory behind head pose estimation is rooted in computer vision and involves solving for the orientation and position of the head using known 3D points and their corresponding 2D projections which is known as the Perspective-n-Point (PnP) problem.
The goal is to find the rotation matrix and the translation vector such that the 3D points can be projected onto the image plane, matching the 2D points, thus determining the orientation of the driver's head.
Coordinate Systems
Before delving into calculating the rotation and translation information, it is important to understand the various coordinate systems involved:

- **World Coordinate System:** This is a global coordinate system in which the 3D points are defined. It represents the real-world locations of objects or features.
- **Camera Coordinate System:** This coordinate system is centered at the camera's optical center.

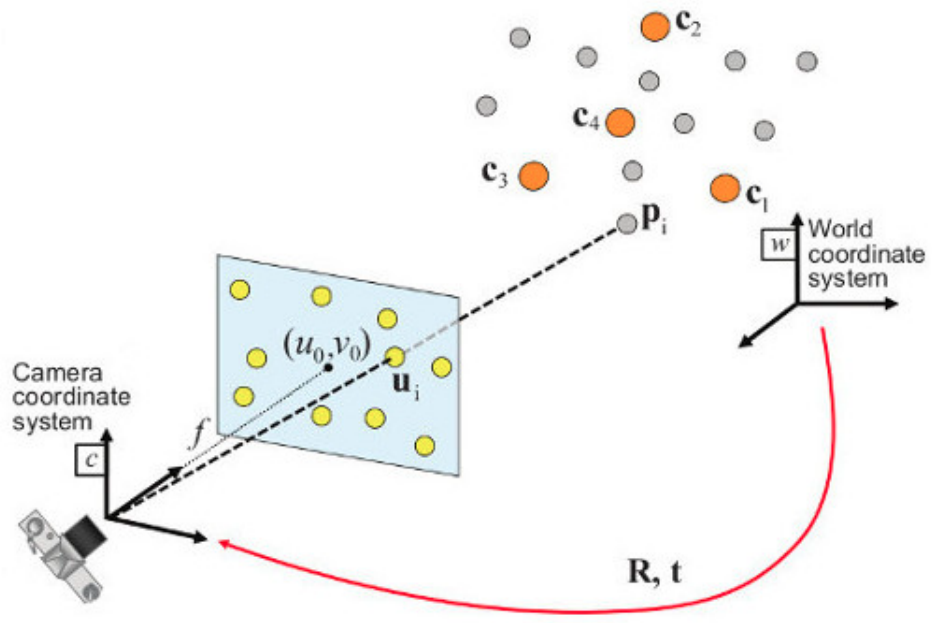


Figure 7: Coordinate systems

The camera's axis defines the orientation of this system: the z-axis points along the camera's optical axis, the x-axis is horizontal, and the y-axis is vertical.

- **Image Coordinate System:** This is a 2D coordinate system on the image plane (sensor) of the camera. Points in this system are expressed in pixel units.

So, if given:

- A set of 3D points $\mathbf{X}_i = (X_i, Y_i, Z_i)$ in the world coordinate system.
- Their corresponding 2D image points $\mathbf{x}_i = (x_i, y_i)$ in the image plane.

The goal is to find the rotation matrix $\mathbf{R}$ and the translation vector $\mathbf{t}$ such that the 3D points can be projected onto the image plane, matching the 2D points. The relationship is given by the projection equation:

$$s \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = \mathbf{K} \left(\mathbf{R} \begin{bmatrix} X_i \\ Y_i \\ Z_i \end{bmatrix} + \mathbf{t} \right)$$

where:

- s is a scale factor.
- $\mathbf{K}$ is the camera intrinsic matrix (containing focal length and principal point).
- $\mathbf{R}$ is the rotation matrix.
- $\mathbf{t}$ is the translation vector.

This equation can be solved to obtain the $\mathbf{R}$ matrix by using OpenCV's `solvePnP` method. This method requires known 3D points of the face and their corresponding 2D points, as well as the intrinsic parameters of the camera and the distortion coefficients. To prepare for this, we need to:

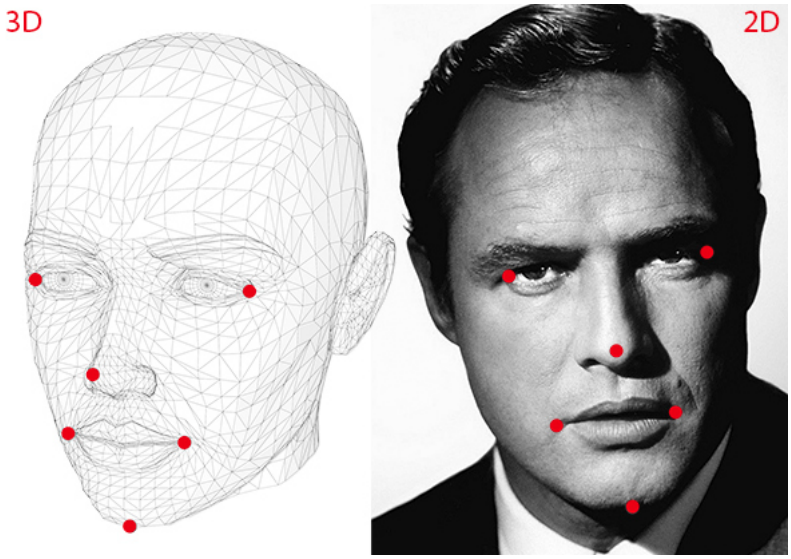


Figure 8: The needed points for the projection equation in 3D and 2D

1. Select a set of standard 3D points on a generic head model. These points are in a fixed coordinate system relative to the head. The number of points should be sufficient to form enough equations to solve the problem. In our implementation common points include the nose tip, the chin, eye corners, and mouth corners are used.

2. Obtain the corresponding 2D points using the dlib landmark extraction model mentioned earlier.

3. Define the camera's intrinsic parameters, including the focal length and the principal point. These parameters can be obtained from the camera's datasheet, If this data is not available, it can be obtained through a camera calibration process, which involves capturing several images from the camera and performing calculations to estimate the parameters.

4. Finally, Obtain the distortion coefficients, which account for lens distortion. These can be obtained through camera calibration or set to zero if calibration data is unavailable or the distortion can be neglected.

Once this data is provided to `solvePnP,` we can obtain the rotation matrix and determine the head orientation in roll, pitch, and yaw angles.

1.2.2 Eye Gaze Estimation [13]

The eye gaze tracking system leverages computer vision techniques to detect and track the driver's gaze direction based on pupil positions. This system is implemented using OpenCV for image processing and Dlib for facial landmark detection, with the core functionality spread across multiple classes, namely "Pupil", "Eye", and "Calibration".

1. **The Pupil class:** This class detects the iris within the eye frame and estimates the pupil position. The process begins with preprocessing the eye frame to isolate the iris. This involves several image-processing techniques:

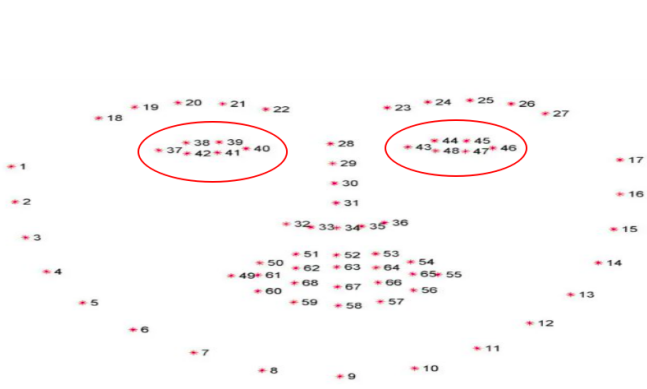
    - **Bilateral Filtering:** This technique smooths the image while preserving edges. It reduces noise and helps in better edge detection by applying a non-linear filter that averages pixel values but keeps significant edges sharp.

- **Erosion:** Erosion reduces the size of objects in an image. By applying a kernel (a small matrix), erosion removes pixels on object boundaries, which helps in eliminating small noise and separating closely connected objects.
- **Binarization:** This method converts the grayscale image into a binary image using a threshold value. Pixels above the threshold are set to the maximum value (white), and pixels below are set to the minimum value (black), which helps in distinguishing the iris from the rest of the eye.

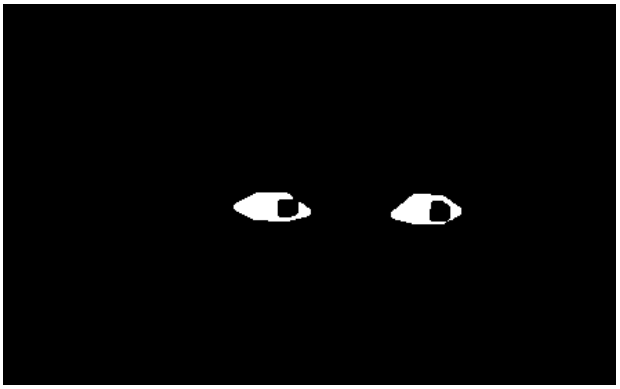
After preprocessing, the binarized image is used to detect contours. Contours are curves that join all continuous points along a boundary with the same color or intensity. The largest detected contour is presumed to be the iris. The centroid of this contour, calculated using image moments, provides the estimated pupil position.

2. **The Eye class:** This class isolates the eye region from the face frame using predefined landmark points and creates a new frame focusing solely on the eye. These landmark points correspond to specific facial features detected by Dlib's shape predictor. The class calculates a blinking ratio to determine if the eye is open or closed. This ratio is derived by measuring the distances between specific eye landmarks: the horizontal distance (eye width) and the vertical distance (eye height). The blinking ratio is the width divided by the height. A high ratio indicates the eye is open, while a low ratio suggests blinking. The 'Eye' class also integrates the 'Pupil' class to detect and position the pupil within the isolated eye frame.

3. **The Calibration class:** This class optimizes the pupil detection process by determining the best binarization threshold for each individual. It evaluates multiple frames to establish a threshold that results in the most accurate iris size estimation. The calibration process involves analyzing several frames to find a threshold that ensures the iris occupies an optimal percentage of the eye frame surface. This personalized threshold enhances the accuracy and reliability of pupil detection.



(a) Eye landmarks extraction



(b) Filtering to extract pupils

Figure 9: Eye gaze algorithm

Finally, the algorithm combines all these functionalities. Within the YOLO bounding boxes, Dlib's facial landmarks, enable the initialization of 'Eye' objects for both eyes. The gaze direction is calculated by comparing the pupil's position relative to the center of the eye frame. Then it analyzes gaze direction and determines if the user is blinking or where they are looking (left, right, center), which is used for distraction detection. This comprehensive approach ensures accurate and real-time monitoring of the driver's gaze direction, enhancing safety by detecting potential distractions.

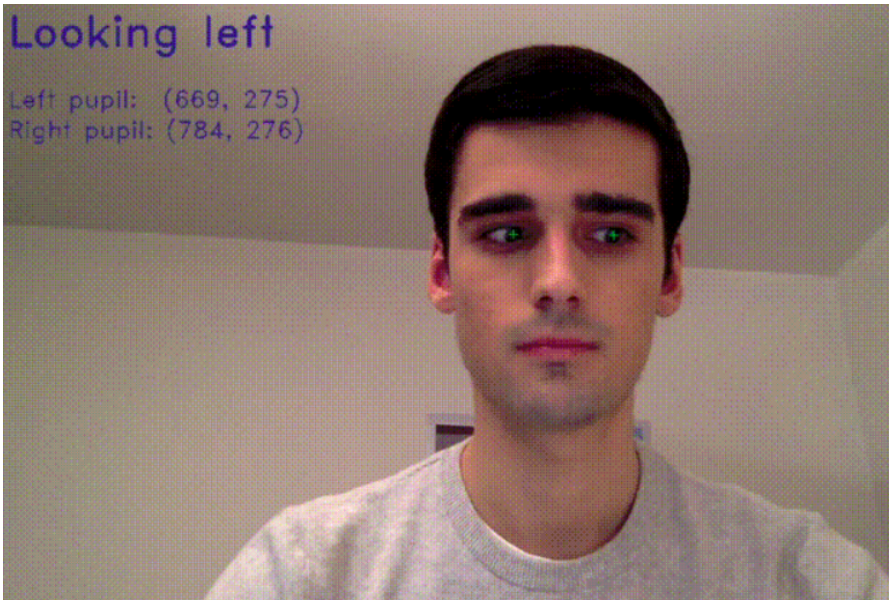


Figure 10: Eye gaze algorithm output

1.3 Drowsiness Analysis

1.3.1 Eye Closure Detection [26] [23]

In our previous discussions, we explored how to apply facial landmark detection to identify key regions of the face, such as the eyes, eyebrows, nose, ears, and mouth. This capability allows us to extract specific facial structures by referencing the indexes of particular face parts.

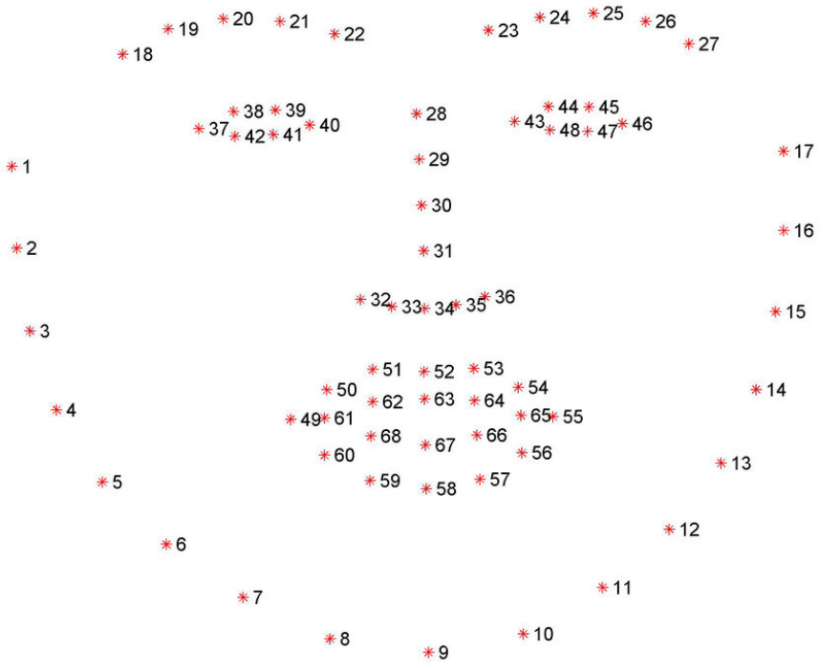


Figure 11: Facial 68 landmarks

For the purpose of blink detection, we focus on two sets of facial structures: the eyes. Each eye is represented by six (x, y)-coordinates, starting at the left corner of the eye (as if you were looking at the person) and moving clockwise around the rest of the region.

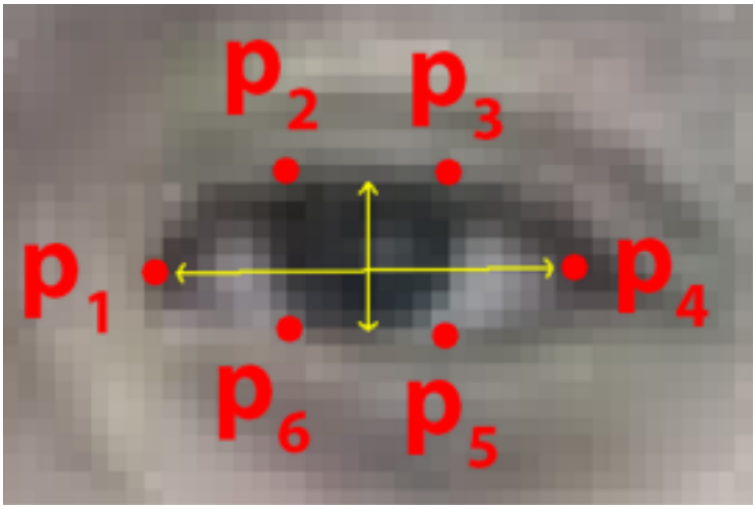


Figure 12: The six facial landmarks associated with the eye

Building on the work by Soukupová and Čech in their 2016 paper, *Real-Time Eye Blink Detection using Facial Landmark*, we derive an equation reflecting this relationship, known as the eye aspect ratio (EAR):

$$\text{EAR} = \frac{\|p_2 - p_6\| + \|p_3 - p_5\|}{2\|p_1 - p_4\|} \tag{1}$$

where $p_1, \ldots, p_6$ are 2D facial landmark locations. The numerator calculates the distance between the vertical eye landmarks, while the denominator computes the distance between horizontal eye landmarks, with appropriate weighting since there is only one set of horizontal points but two sets of vertical points. The eye aspect ratio remains approximately constant when the eye is open but rapidly decreases to zero when a blink occurs. This simple equation enables us to detect blinks by relying on the ratio of eye landmark distances, bypassing the need for complex image processing techniques.
Consider the following figures:

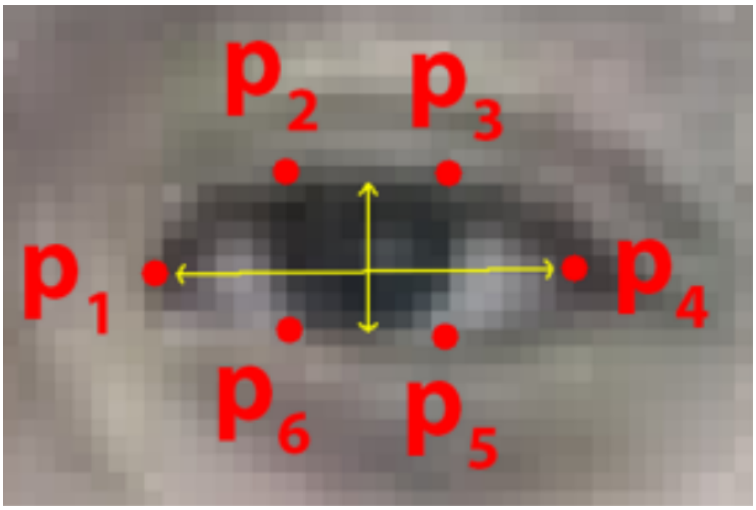


Figure 13: Eye landmarks when the eye is open

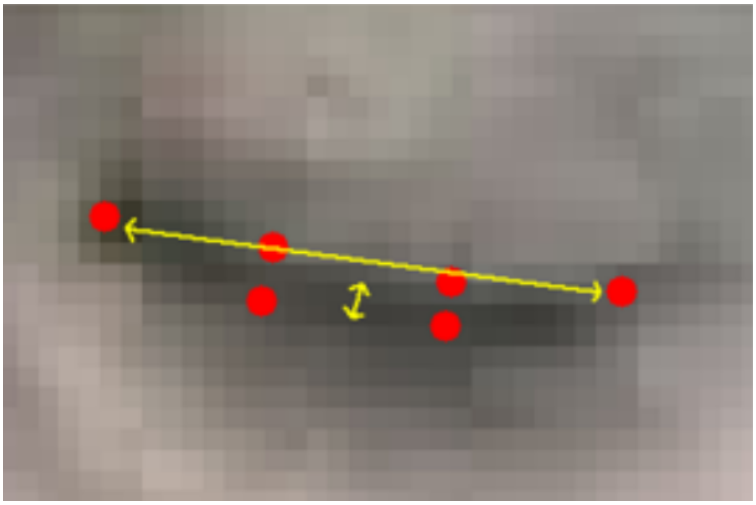


Figure 14: Eye landmarks when the eye is closed

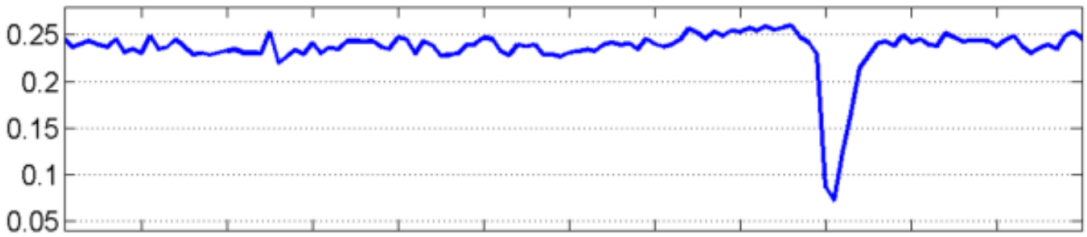


Figure 15: Plotting the eye aspect ratio over time. The dip in the eye aspect ratio indicates a blink

In Fig. 13, the eye is fully open, and the eye aspect ratio is relatively large and constant. When the person blinks in Fig. 14), the eye aspect ratio drops significantly, approaching zero.
To determine whether the eye is open or closed based on the Eye Aspect Ratio (EAR), a threshold value is crucial. Typically, the eye is classified as closed when the EAR drops below a specified threshold, such as the commonly recommended value of around 0.25. This value ensures accurate detection of blinks and prolonged eye closures, critical for evaluating driver alertness in driver monitoring systems. Adjusting the threshold to approximately 0.25 enhances performance under different lighting conditions and facial orientations.
Fig. 15 plots the eye aspect ratio over time for a video clip. As observed, the eye aspect ratio remains constant, then drops near zero, and rises again, indicating a blink.

1.3.2 Yawn Detection [23]

Yawn detection is another crucial indicator of driver fatigue. This detection algorithm analyzes the distance between the upper and lower lips using facial landmarks.
The algorithm calculates the mean positions of the upper and lower lips using the following steps:

1. Calculate the Mean Positions of the Outer Upper Lip Landmarks

$$\text{upper\_mean}_x = \frac{x_{u50} + x_{u51} + x_{u52} + x_{u61} + x_{u62} + x_{u63}}{6}$$

$$\text{upper\_mean}_y = \frac{y_{u50} + y_{u51} + y_{u52} + y_{u61} + y_{u62} + y_{u63}}{6}$$

2. Calculate the Mean Positions of the Outer Lower Lip Landmarks

$$\text{lower\_mean}_x = \frac{x_{l56} + x_{l57} + x_{l58} + x_{l65} + x_{l66} + x_{l67}}{6}$$

$$\text{lower\_mean}_y = \frac{y_{l56} + y_{l57} + y_{l58} + y_{l65} + y_{l66} + y_{l67}}{6}$$

3. Compute the Vertical Distance Between the Mean Positions

$$\text{distance} = \left|\text{upper\_mean}_y - \text{lower\_mean}_y\right|$$

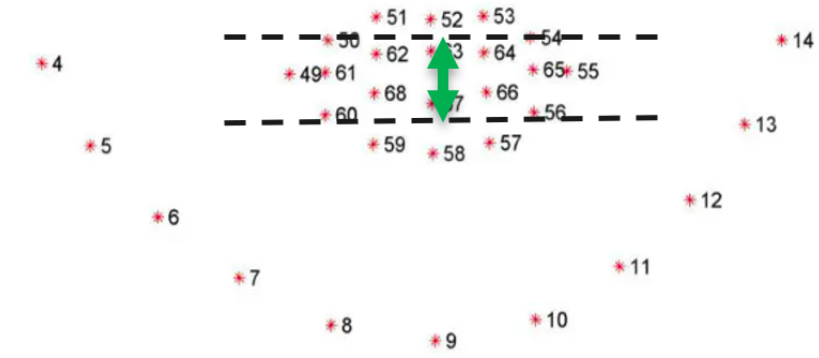


Figure 16: Facial landmarks for yawning detection

By measuring the distance between the upper and lower lip landmarks in Fig. 16, the algorithm can determine if the mouth is open, which is a key indicator of yawning. A significant increase in this distance indicates a yawn, suggesting that the driver may be fatigued.

Yawning detection also relies on a threshold, often determined by the vertical distance between upper and lower lip landmarks. While specific thresholds can vary, values around 0.5 to 1.0 have been used experimentally. Adjusting this threshold helps in accurately identifying yawning events, crucial for assessing driver fatigue in monitoring systems, despite varying facial expressions and conditions.
By integrating these advanced detection methods, the driver monitoring system can effectively assess driver fatigue and drowsiness, contributing to improved road safety.

1.4 Main Application Decision Layer

We present the decision layer of our DMS pipeline, which is essential for detecting and addressing potential issues related to driver attention and drowsiness. The decision layer processes input frames from the system and makes determinations based on several criteria. Below is a detailed explanation of the decision-making process, represented in the flowchart in Fig. 17.

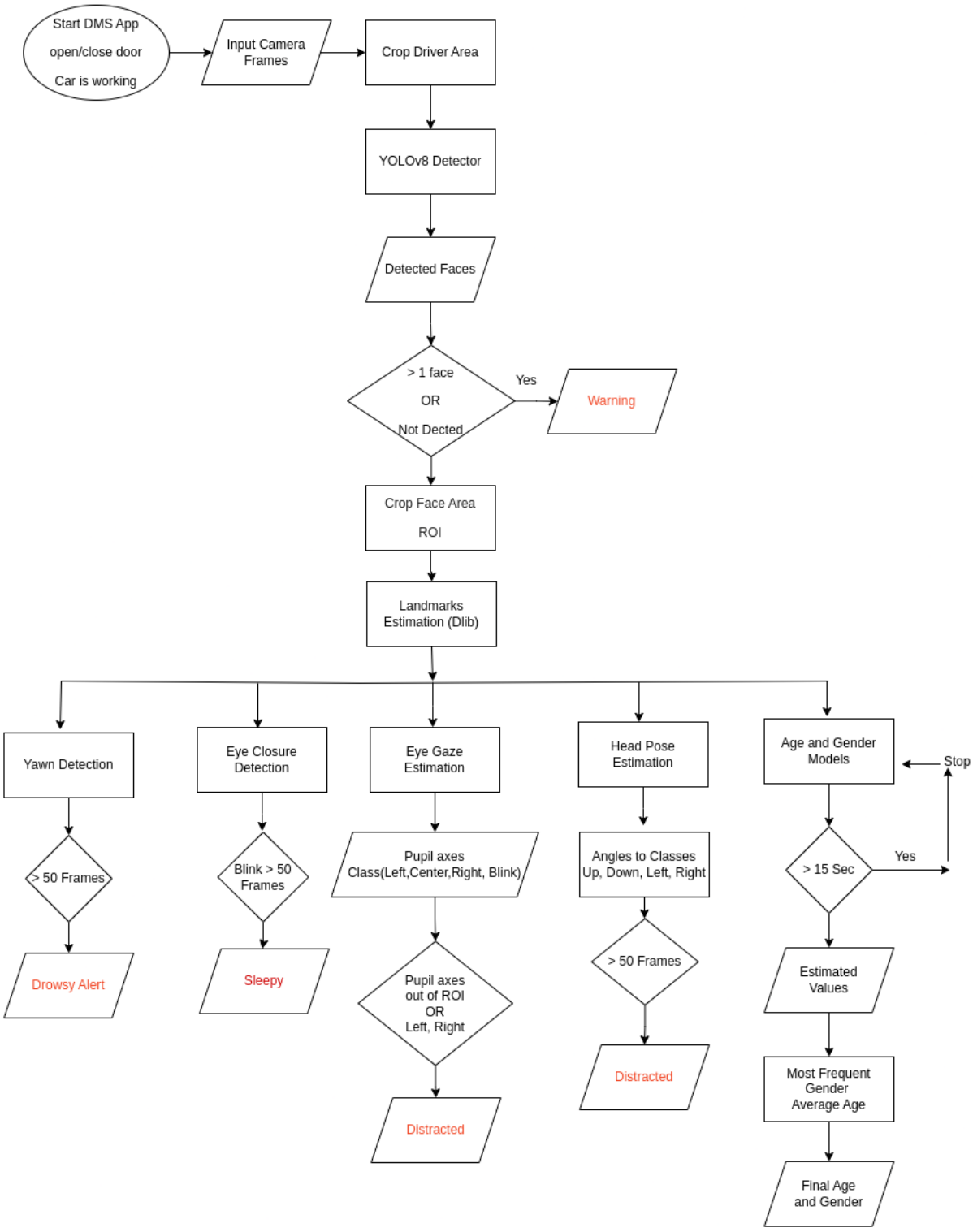


Figure 17: System decision layer

1. **Initialization**: The DMS application starts when the car is in operation and every time a car door is opened or closed. This information is obtained from the car's internal sensors, which can detect changes indicating a new driver. The initial input frame is captured through the imager.

2. **Driver Area Cropping**: The system crops the driver area from the input frame to avoid detecting other passengers.

3. **YOLO Detector**: The cropped driver area is passed to the YOLO detector, which identifies faces within the frame.

4. **Face Detection Check**: The system checks whether faces are detected:
    - If more than one face is detected or no face is detected, a `Warning` is triggered.
    - If exactly one face is detected, the system proceeds to crop the face area (Region of Interest, ROI).
5. **Facial Landmark Detection (Dlib)**: For further analysis, the Dlib library is used to detect facial landmarks, focusing on the mouth, eyes, head pose, and eye gaze.
6. **Age and Gender Estimation**: The cropped face area is analyzed to estimate the age and gender of the driver.
7. **Age and Gender Consistency Check**: If the system runs for more than 15 seconds, the most frequent gender and average age are calculated to get estimated values. This ensures the system is consistently identifying the driver. After this time, the age and gender module is stopped due to the high computational load it imposes, as it involves two additional deep learning models. The module is re-run if the car door is opened and then closed, indicating a driver change.
8. **Mouth Ratio Analysis**: If the mouth ratio exceeds a certain threshold for a certain time, a `Drowsy Alert` is triggered.
9. **Eye Ratio Analysis**: If the eye ratio falls below a certain threshold and the blink count exceeds 50 frames, the driver is classified as `Sleepy`.
10. **Head Pose Analysis**: The head pose is evaluated by converting angles into classes (left, right, up, down), and identifies the driver's `Distraction`.
11. **Eye Gaze Analysis**: If the pupil axes fall out of the ROI or indicate left/right movement, the driver is considered `Distracted` and `Sleepy` if it identifies Blink.

1.5 Evaluating Efficiency and Accuracy

In this section, we evaluate the efficiency and accuracy of the DMS. Our evaluation process is multifaceted, involving initial assessments of open-source datasets, subsequent dataset collection using our imager in various environments, and real-time performance evaluation. We employ various metrics to measure the confidence and accuracy of our system's output for each task while it is operational.

1.5.1 Real-Time Confidence Evaluation

For each task in our DMS, we choose specific metrics to evaluate real-time confidence and accuracy. These metrics are critical in understanding how well our system performs under actual operating conditions with the chosen imager.

1. **Head Pose Confidence [19]**
   To ensure the accuracy of our head pose estimation, we use the reprojection error evaluation metric that uses the camera calibration parameters.

   Reprojection Error
   It is a metric for evaluating the accuracy of head pose estimation. It measures the root mean square (RMS) error between the projected points (as predicted by the head pose estimation model) and the

actual points used during pose estimation. The lower the reprojection error, the more accurate the head pose estimation. This metric is crucial indicating how well the model translates 3D head positions and orientations into 2D image coordinates.

Camera Calibration Parameters

Camera calibration is an essential process that involves determining the camera's internal characteristics (intrinsic parameters) and external characteristics (extrinsic parameters). This calibration allows us to correct for distortions and accurately interpret the spatial relationships in the captured images, allowing us to use the reprojection error algorithm. The key calibration parameters we use include:

(a) **Intrinsic Matrix**

The intrinsic matrix is a 3x3 matrix that includes the focal lengths and the principal point of the camera. It is defined as follows:

$$\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

where f_x and f_y are the focal lengths in the x and y directions, respectively, and c_x and c_y are the coordinates of the principal point.

- **Focal Length**: Determines the magnification of the camera. This value affects how the 3D world is projected onto the 2D image plane.
- **Principal Point (Center)**: This is the point on the image sensor where the optical axis intersects.

(b) **Distortion Coefficients**

These coefficients account for the lens distortion, which includes radial and tangential distortions that can cause image warping.

Hopefully, all of these parameters are provided by the IMX supplier. For our IMX219 imager (which will be discussed in detail in the following sections), the focal length is 3.15 mm, and the principal point is $(0, 0)$.

2. **Eye Gaze Confidence**

The eye gaze confidence is a critical metric that quantifies the reliability and accuracy of gaze detection in real-time. This confidence is evaluated by tracking the success rate of the pupil and facial landmark detection over a sliding window of recent frames. Specifically, the algorithm maintains a record of successful pupil detections within the last 30 frames, computing an average success rate to yield a confidence score. Additionally, the stability and consistency of detected gaze points are monitored to ensure that the gaze estimation is not only accurate but also robust over time. By integrating these metrics, the algorithm can provide a dynamic confidence percentage that reflects real-time performance.

1.5.2 Overall System Accuracy

To comprehensively evaluate the accuracy of our system, we apply our algorithms to each task pipeline, starting from the YOLOv8 detector through to the Dlib landmark detector and ending with task-specific algorithm.

Dataset Used in the Evaluation [22]

To establish a baseline, we first test our system using open-source datasets. This initial step helps us fine-tune our algorithms and models before moving to real-world data. Once the preliminary validation is complete, we collect a new dataset using our imager in different environments, such as day and night settings, and varying weather conditions. We label this dataset using RoboFlow, a powerful tool that facilitates dataset management, annotation, and preprocessing.

1. **Mouth and Eye Closure Accuracy [16]**
   Firstly, we utilize the FL3D dataset, a sample image shown in Fig. 18, it is intended for the evaluation of drowsiness detection algorithms for mouth, and eye closure.

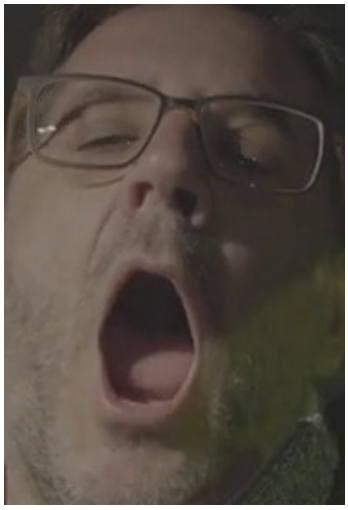


Figure 18: FL3D dataset for mouth and eye closure

   We apply our algorithms and compare the predictions with the actual annotations.

2. **Eye Closure Accuracy**
   Fig. 19shows the confusion matrix for a binary classification task distinguishing between `Asleep` and `Awake` states for eye aspect ratio. The metric indicates that with a threshold set at 0.25, the overall accuracy of the algorithm is 71%. The confusion matrix reveals that the model correctly predicts `Awake` instances 62% of the time but only correctly predicts `Asleep` instances 8.8% of the time. There is a notable misclassification where 18% of actual `Asleep` instances are predicted as `Awake` and 11% of `Awake` instances are predicted as `Asleep.` This discrepancy may be due to the difficulty in distinguishing the `Asleep` state, possibly because having one eye closed doesn't guarantee sleep. We are working to improve the model's ability to correctly classify `Asleep` instances.

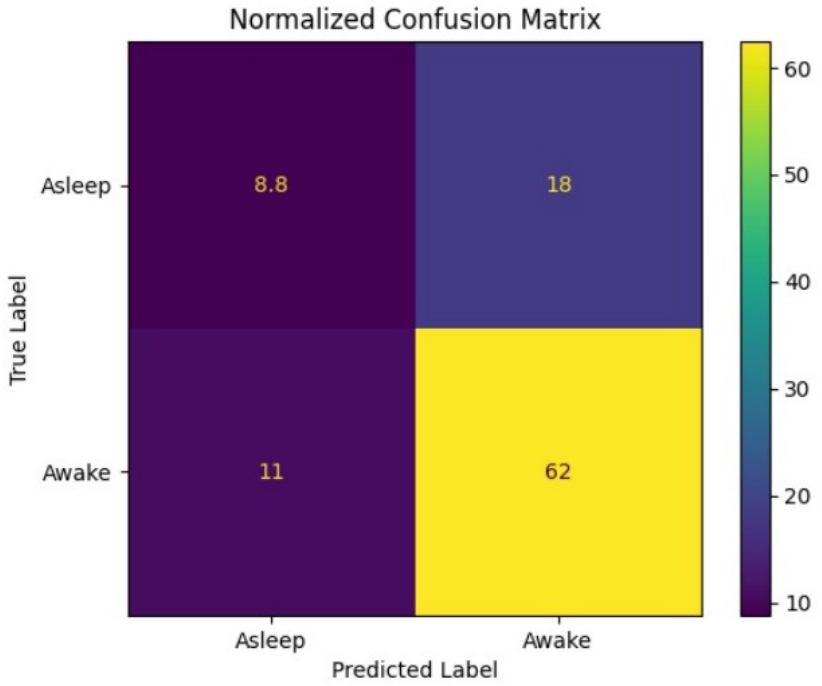


Figure 19: Eye closure confusion Matrix

3. **Mouth Closure Accuracy**

   The confusion matrix for mouth closure is shown in Fig. 20, with classes `Normal` and `Yawn`. The metric indicates that with a threshold set at 60, the algorithm has an overall accuracy of 97%. It correctly predicts `Normal` instances 88% of the time and `Yawn` instances 9.1% of the time. This indicates that the algorithm is highly accurate in yawn detection.

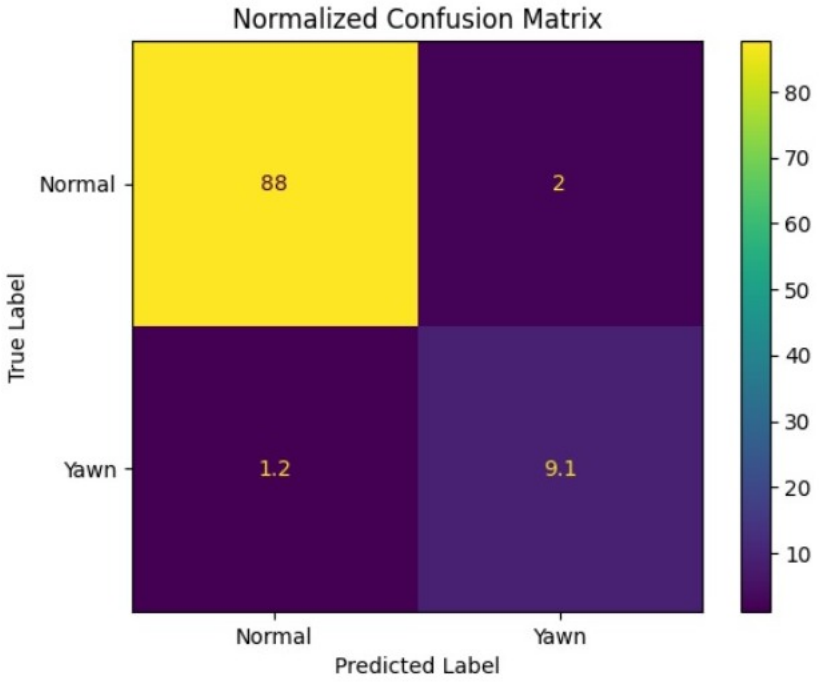


Figure 20: Mouth closure confusion Matrix

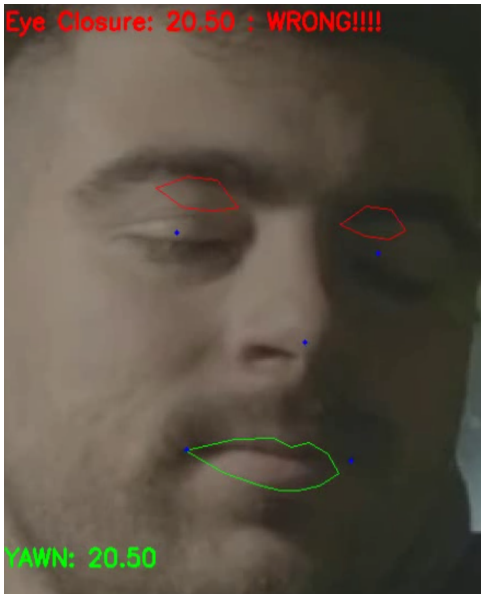


Figure 21: Apply eye and mouth closure algorithms on the dataset

4. **Dlib 68-point Landmark Model**
   This model has been reported to achieve good accuracy for general facial landmark detection tasks. On the 300-W dataset, the error rates are often measured using the normalized mean error (NME), which is typically in the range of 5-6% for the best models.

5. **Age Estimation Model Accuracy**
   Fig. 22 shows the age estimation model confusion matrix that compares predicted age ranges to actual age ranges. Each cell contains a percentage value that indicates how often instances of a particular actual age range were predicted to fall into a particular predicted age range.

Actual / Predict	0-2	4-6	8-13	15-20	25-32	38-43	48-53	60-
0-2	**93.17**	6.42	0.21	0.00	0.21	0.00	0.00	0.00
4-6	26.84	**62.11**	7.37	1.93	1.58	0.00	0.18	0.00
8-13	1.76	6.18	**42.06**	12.94	35.59	0.29	1.18	0.00
15-20	1.76	0.44	4.41	**24.23**	64.76	0.00	4.41	0.00
25-32	0.00	0.09	0.85	3.22	**86.17**	3.31	4.83	1.52
38-43	0.39	0.20	0.39	1.78	59.76	**8.88**	22.88	5.72
48-53	0.00	0.00	0.00	0.00	19.50	8.71	**38.17**	33.61
60-	0.00	0.00	0.00	1.17	1.95	1.17	35.02	**60.70**

Figure 22: Age estimation model confusion matrix

The 1-off accuracy is a metric that represents the accuracy within one year of the actual age. In this context, it means how often the predicted age is within one year of the actual age. This model achieved a 1-off accuracy of 90.57%.

6. **Gender Classification Model Accuracy**
   The model comes with a classification accuracy of 97.3%.

2 Software In the Loop (SIL)

During the early stages of the production cycle, the need for a simulation environment has become inevitable as we proceed further with the DMS app development. We followed a Continuous Integration / Continuous Development (CI/CD) pipeline to push releases, fixes, and updates seamlessly. To realize this CI/CD pipeline, we developed a Software In the Loop (SIL) component to virtualize the hardware platform and decouple the DMS app from the underlying hardware accelerators. This approach saved us invaluable time as the lead time for the TDA4VM kit would have delayed the workflow drastically (more on the TDA4VM in section 3.2).

The SIL is based on an x86 machine, our laptops, for instance, running a Linux distro. The core value of the SIL is that it simulates the processing cores provided by the TDA4VM SoC. This simulation is not meant to be a hardware emulation of the SoC. Instead, it serves as a good way to loosely test the performance of the AI algorithms. However, hardware emulation is a technique used in the design and development of hardware systems. It involves creating a model of the hardware in a specialized environment that, accurately, mimics the behavior of the actual hardware. This allows testing, verification, and debugging of the hardware design or software systems before having access to the real hardware.

NOTE: A detailed description of the hardware items will follow in section 3.

2.1 SIL Breakdown Structure

1. Virtual CPU Cores:
   The TDA4VM provides a dual-core ARM A7 CPU. These cores are simulated by two parallel processes communicating using Inter-Process Communication (IPC).

2. Virtual Imager:
   The imager used in this project has a Camera Serial Interface (CSI). However, to connect the imager to the SIL we need to use a CSI-to-USB converter. We had to bypass this step as the shipping time of the imager was too long (2 months) for us to wait for. Instead, we implemented a virtual imager driver using the OpenCV library. This driver mimics the act of fetching a frame from the physical image sensor. What is more important is that the driver is wrapped inside a wrapper to provide abstraction making modifications and updates easier. To automate the testing process, the virtual imager is capable of fetching frames from a video file or even from a set of still images in stored locally or in a remote directory.

3. Virtual Vision Processing Accelerator (VPAC):
   This dedicated hardware accelerator provided by the TDA4VM is responsible for the preprocessing of the fetched frame. Tasks such as rescaling, and color space manipulation are achieved by the VPAC. The VPAC comprises many submodules such as the Image Signal Processing (ISP) module, and all of them are encapsulated as a whole in the SIL using an OpenCV wrapper.

4. Virtual Digital Signal Processors (DSPs):
   The power of the TDA4VM lies in its vector DSP cores together with the Matrix Multiplication Accelerator (MMA). Inside the SIL, the MMA is not directly simulated as the SIL runs on x86 machines which have no DSPs nor MMA. Now the question is how about the VPAC, what is different about

it? The answer is that the virtual VPAC only has to execute simple vision tasks that can be easily accomplished by a GPU or even a general-purpose CPU. On the other hand, the tasks running on the MMA are executed in a totally different way manner and, most importantly, with a significant performance gain. In our case, the SIL is not used as an emulation platform. So, we are not interested in hardware profiling. With all that in mind, we simulated the DSPs using parallel threads called by parent CPU threads and communicating with each other using both shared memory and IPC as shown in fig.23.

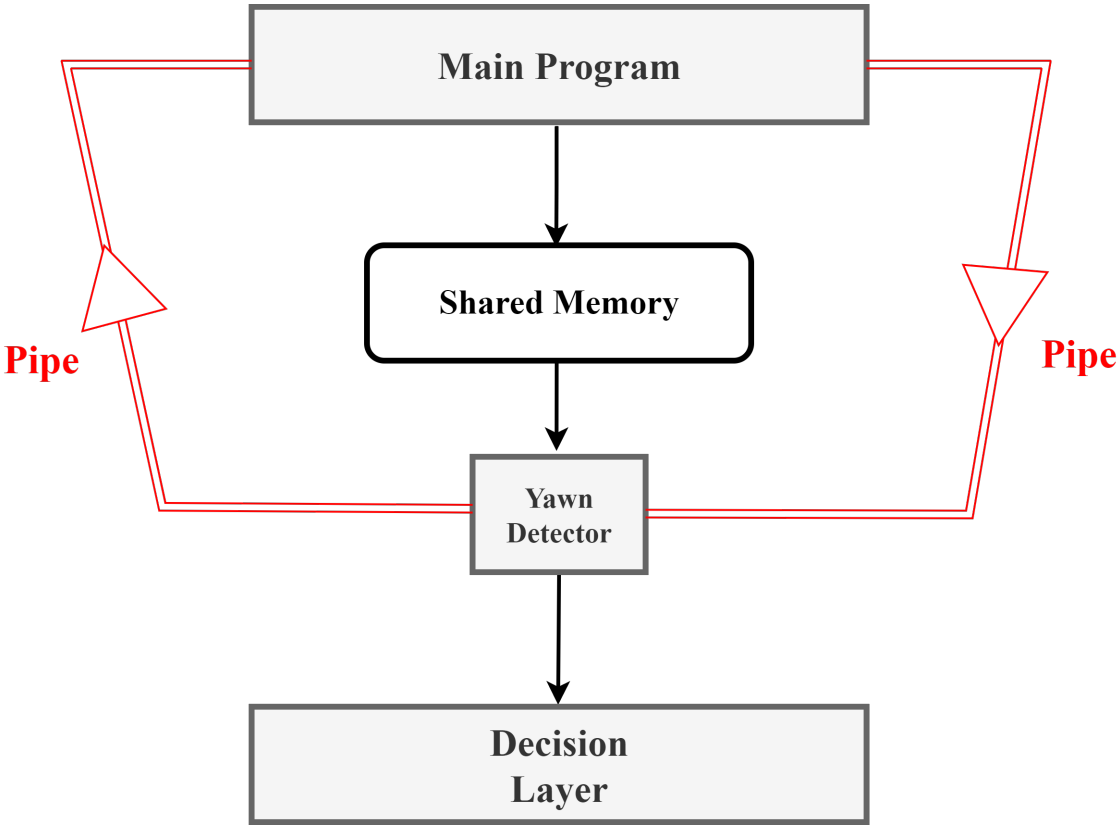


Figure 23: Inter-process communication in the SIL

2.2 The Main Application

The core application runs on the CPU, either in the SIL or on the HW platform. This application is the backbone of the DMS Application. It connects the imager to the communication module all the way through the processing nodes running our AI algorithms. However, there is something we have to admit when it comes to the system design of this application and the SIL. When we started to migrate the app to the actual HW platform, we noticed some sort of coupling to the SIL infrastructure. For instance, we had to adjust the tasks assigned to the DSPs as they were structured in such a way that they were suitable for the parallel processes of the SIL and not directly compatible with the actual DSPs. To solve this issue, we reimplemented the problematic modules and decoupled the app from the underlying platform. In the following subsections, we guide you on a journey with the frame starting at the capturing moment and ending at the display of the visualization module.

1. Fetching The Frame: The moment the imager shutter is fired (or the frame is fetched in the virtual imager), a frame is pulled from the imager buffer and directly supplied to the VPAC (or the virtual VPAC, anyway, you got the point) for preprocessing.

2. Preprocessing: Now, we have a raw frame to process. Multiple processing tasks are executed on the frame such as rescaling, cropping, and coloring, as described in **??**. Afterward, the frame is passed to the DPSs (or, again, virtual DSPs) for inferencing. In the SIL, the frame is written into the shared memory to be used for demonstration purposes where the final result is overlayed on the original frame. The state machine of the writer thread is shown in fig.24.

3. Deep Learning Inference: At this stage, the frame is in the right format required by the deep learning model. In the SIL we used to use the YOLO V8 model, whereas on the TDA4VM we had to fall back to YOLO V5 (more on that in the **??**). The output of this stage is a set of bounding boxes representing the detected faces in the frame. You may be asking why on earth would we have multiple faces in the driver's seat! But, that's why being an engineer is not an easy task. Imagine your DMS triggering a false alarm and maybe pulling your car to the shoulder just because your daughter's head is popping in the frame!

4. Machine Learning Inference: After receiving the bounding boxes, this module is responsible for extracting facial landmarks from the detected faces. Then, the extracted landmarks are forwarded to the postprocessing nodes. In the SIL the result is written into a shared memory accessed by the postprocessing nodes.

5. Postprocessing: In the SIL, the postprocessing nodes get the landmarks from the shared memory and execute the algorithms for eye closure, yawn detection, head-pose estimation, etc. 6. The Decision Layer: The results are forwarded to the decision layer to determine the state of the driver. The final decision (drowsiness, attentiveness, etc) is then passed to the communication module for publishing.

6. The Communication Module:

    - Local communication: In reality, the DMS as a module is connected to the car system through a Vehicular Interface Unit (VCU). This communication is achieved through a CAN bus. However, in our case, we decided to use UART instead as we don't have a VCU to communicate with.
    - Data Logging: On the server side, the DMS logs output data, DMS state, errors, faults, etc. This logging mechanism is illustrated in 7.2.

7. The Visualization Module: Finally, the DMS output is sent to the visualization module for display. This module can either be the infotainment system of the car or a remote display on the server side. Inside the SIL, the result if overlayed on the original frame read from the shared memory as shown in fig.25.

A holistic view of the system is illustrated in appendix.12.1

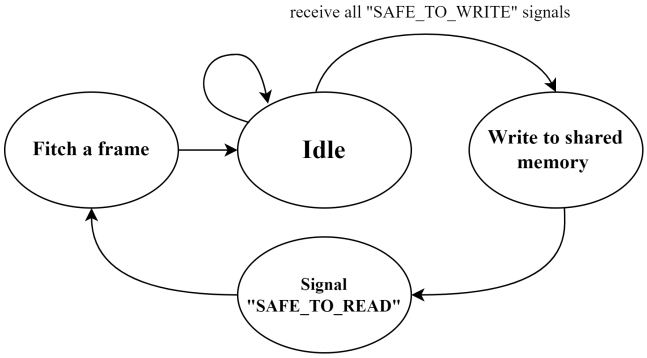


Figure 24: Writer thread state machine

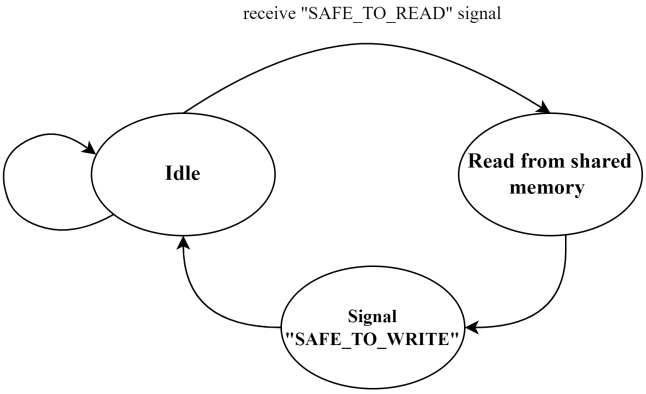


Figure 25: Reader thread state machine

3 Hardware Selection

3.1 Camera Selection for DMSs[20][12]

In embedded vision applications, particularly within DMSs, the camera serves as the primary sensor, capturing essential visual data for analysis. The selection of an appropriate camera is crucial to ensure accurate monitoring of the driver's face, eyes, and upper body movements. This section outlines key criteria for camera selection to optimize the effectiveness of DMS.
DMS cameras typically need to encompass the area where the driver's head and upper body movements occur. To estimate the dimensions of the required Field of View (FOV), the following dimensions were considered:

- **Width:** The camera should cover the full width of the driver's seat, typically between 18 to 22 inches (45 to 56 cm).
- **Height:** The height should extend from the top of the driver's head to at least chest level when seated, ranging from 24 to 36 inches (60 to 91 cm).
- **Depth:** The depth should cover the distance from the steering wheel to the back of the driver's head, typically around 20 to 28 inches (50 to 71 cm).

These ranges ensure that the DMS camera can capture all necessary details of the driver's face, eyes, and upper body for effective monitoring. Integrating these estimations into our camera selection criteria ensures that the chosen camera meets the spatial requirements necessary to monitor driver behavior accurately and comprehensively.

3.1.1 Resolution

The resolution parameter is crucial when selecting the sensor for a DMS as it directly affects the vision system's quality, accuracy, and performance. The resolution is defined as the ratio between the object size and the smallest detail that we want to detect in that object, represented in one pixel.

To determine the appropriate resolution for our DMS, we first considered the working plane within the vehicle, which is the area that the camera needs to monitor. For our application, we defined this working plane with approximate dimensions of 55 cm in width and 75 cm in height.

We aimed to detect small details accurately, such as the features of a driver's eyes, which we approximated to be about 1 mm. This smallest detail size is critical for accurate eye detection, as the clarity of these details directly impacts the performance of our eye closure detection algorithm.

Using the dimensions of our working plane and the smallest detail size, we calculated the required resolution as follows:

$$\text{Resolution} = \frac{\text{Object Size}}{\text{Smallest Detectable Detail Size}}$$

- **For the horizontal resolution (width):**

$$\text{Horizontal Resolution} = \frac{\text{Width of Working Plane}}{\text{Smallest Detail Size}} = \frac{55\,\text{cm}}{0.1\,\text{cm}} = 550\,\text{pixels}$$

- **For the vertical resolution (height):**

$$\text{Vertical Resolution} = \frac{\text{Height of Working Plane}}{\text{Smallest Detail Size}} = \frac{75\,\text{cm}}{0.1\,\text{cm}} = 750\,\text{pixels}$$

Thus, the resolution we require for our DMS is approximately 550 x 750 pixels, which equates to roughly 0.5 megapixels (MP).
This resolution can be covered by a wide range of sensors available on the market. However, it is important to choose a sensor with a suitable resolution to avoid over-computation and to maintain a balance between resolution, pixel size, and sensor size.

- **Resolution:** Higher resolution improves detail detection but increases computational load.
- **Pixel Size:** Larger pixels capture more light, improving image quality in low-light conditions.
- **Sensor Size:** A larger sensor with the same resolution as a smaller one will have larger pixels, which can be beneficial for low-light performance.

Maintaining this three-way balancing act ensures that the chosen camera sensor provides sufficient detail for accurate driver monitoring while not overburdening the processing capabilities of the DMS.

3.1.2 Pixel Size

Pixel size is a parameter that significantly impacts the performance of the vision system. Based on research and common practices in similar applications, we selected a pixel size in the range of 2.8 to 3.2 micrometers (µm). Opting for a relatively large pixel size is essential for our machine-vision applications for several reasons:

- **Lower Pixel Count for Real-Time Processing**
  Larger pixels result in a lower pixel count for a given sensor size, which reduces the amount of data that needs to be processed. This reduction in data volume is critical for achieving real-time processing speeds, which are essential for the timely detection of driver states and behaviors.
  The relationship between pixel size and pixel count for a given sensor resolution is as follows:

$$\text{Pixel Count} = \frac{\text{Sensor Area}}{\text{Pixel Area}}$$

  Thus, the pixel count with 3.2 µm pixels for example is lower than that with 2.8 µm pixels, which is beneficial for facilitating real-time processing.

- **Enhanced Low-Light Performance**
  Larger pixels can capture more light compared to smaller pixels. This increased light capture improves the sensor's performance in low-light conditions, which is vital for maintaining the accuracy of our DMS under various lighting environments, such as night driving or poorly lit conditions.
  Mathematically, the amount of light captured by a pixel, often referred to as the pixel's full well capacity, is proportional to the pixel area. The pixel area A can be calculated as:

$$A = \text{Pixel Size}^2$$

  For our selected pixel sizes:

  – For 3.2 µm pixels:

$$A_{3.2} = (3.2\,\mu m)^2 = 10.24\,\mu m^2$$

  – For 2.8 µm pixels:

$$A_{2.8} = (2.8\,\mu m)^2 = 7.84\,\mu m^2$$

  The difference in pixel area highlights the increased light-gathering capability of the larger pixel size:

$$\Delta A = A_{3.2} - A_{2.8} = 10.24\,\mu m^2 - 7.84\,\mu m^2 = 2.4\,\mu m^2$$

  This difference shows that 3.2 µm pixels can capture more light than 2.8 µm pixels, making them more suitable for low-light conditions.

3.1.3 Frame Rate

The frame rate of the camera sensor directly impacts the system's ability to capture and analyze driver behavior in real-time. For our DMS, a frame rate of 60 frames per second (fps) has been selected as it provides a suitable balance between data accuracy and processing efficiency. This frame rate ensures that the system can analyze fine details of the driver's movements, such as blinking and head turns, with minimal delay.

While higher frame rates could theoretically improve the accuracy of our DMS by providing more data points, they also increase the computational load. Higher frame rates require more processing power and memory, which can strain the system's resources and potentially hinder real-time processing. Conversely, a lower frame rate might reduce the computational demand but could compromise the system's ability to promptly detect and respond to changes in the driver's behavior.

Our choice of 60 fps strikes a balance by providing sufficient frame data for accurate analysis without overwhelming the processing capabilities of our system. This balance ensures that our AI models can operate efficiently, delivering reliable performance without unnecessary computational overhead.

Given that our sensor's resolution exceeds the minimum requirements for our application, we have the flexibility to adjust the frame rate by modifying the resolution. Decreasing the resolution allows us to increase the frame rate as the relation between resolution and frame rate is ruled by the data bandwidth as follows:

$$DataBandwidth = \text{Resolution} \times \text{Frame Rate}$$

For example we can convert From (1600 x 1300, 60 fps) to (1280 x 720, 90 fps) while keeping the same

bandwidth.
By managing this balance, we ensure that the DMS can be optimized for different operational conditions without sacrificing accuracy or performance.

3.1.4 Image Sensor (CCD vs. CMOS)

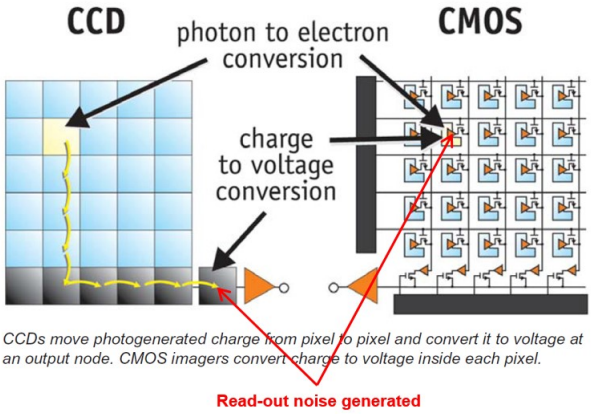


Figure 26: CCD vs. CMOS operation

The debate between (CCD) Charge-Coupled Device and (CMOS) Complementary Metal-Oxide-Semiconductor) sensors has shaped the evolution of digital cameras, particularly in still photography. CCD sensors dominated early digital cameras for their superior image quality and specific applications. They operate on a charge-transfer mechanism with a serial process, as shown in Fig. 26, requiring mechanical shutters to prevent smear artifacts during readout. In contrast, CMOS sensors gained prominence with their parallel readout capability, as shown in Fig. 26, enabling faster processing, lower power consumption, and integration with advanced functionalities like live-view and electronic shutters. Canon's introduction of the first full-frame CMOS sensor in 2002 marked a significant milestone, accompanied by sensor design and manufacturing advancements. Objectively, CCD sensors offer superior image quality with accurate color rendition and smooth tonal transitions but consume more power, generate heat, and have slower readout speeds. CMOS sensors, on the other hand, provide faster readout speeds, lower power consumption, and improved noise performance, particularly in low-light conditions, enhanced further by innovations like Backside Illumination (BSI). Subjectively, while CCD sensors excel in scenarios requiring precise color fidelity and minimal noise at base ISO, CMOS sensors are versatile, supporting high-speed continuous shooting, video recording, and real-time viewing, aligning well with evolving industry demands.

For DMS, CMOS technology is preferred due to its lower power consumption, superior noise performance, and compatibility with real-time video processing, making it ideal for capturing detailed driver behavior effectively.

3.1.5 Shutter Type (Global Vs. Rolling)

Exposure time is a period of the shutter from open to close. During the period, the light exposing on the chip's photosensitive array and Photoelectric effect occurs. After that, photoelectric charges are produced. By the A/D transformation, the value of each pixel is displayed. Under a certain light intensity, the longer the shutter is open, the longer the exposure time, the brighter the image. Long exposure time can show the

trajectory of slow moving objects on an image. Short exposure time can record things more accurately.

In digital cameras, there is usually no mechanical shutter which determines the exposure time. Instead, an electronic shutter is used. This process involves resetting the pixel just before the exposure period starts to return it to its initial state. At the end of the exposure period, the pixel is read out to capture the signal generated by the incident light. Two types of electronic shutters can be used: global shutter and rolling shutter, which differ in their timing mechanisms.

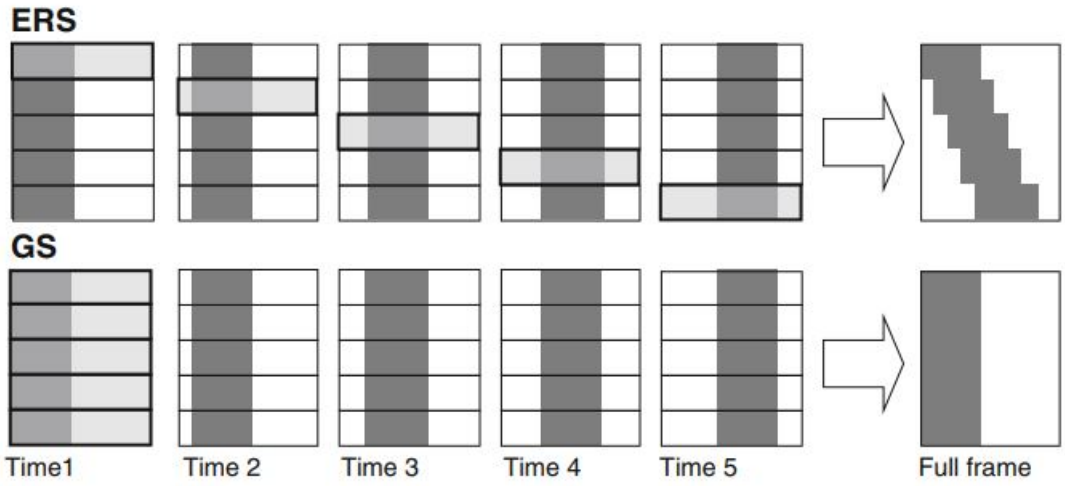


Figure 27: Difference of Electronic Rolling Shutter (ERS) and Global Shutter (GS)

Figure 27 shows, in global shutter mode, each pixel in the sensor begins and ends the exposure simultaneously. This mode requires a significant amount of memory, as the entire image must be stored in memory after the exposure ends and can then be read out gradually. The manufacturing process for global shutter sensors is relatively complex, making them more expensive. However, the advantage of a global shutter is its ability to capture high-speed moving objects without distortion, making it suitable for a wide range of applications.

In rolling shutter mode, different lines of the sensor array are exposed at different times as the readout sweeps through the sensor, as shown in Fig. 27. The first line is exposed first, followed by a readout period, then the second line starts exposure, and so on. This sequential exposure means each line reads out before the next line begins. Each pixel in a rolling shutter sensor requires only two transistors to transport electrons, resulting in lower heat production and reduced noise. Compared to global shutter sensors, rolling shutter sensors have a simpler structure and lower cost. However, because each line is not exposed simultaneously, rolling shutters can produce distortion when capturing high-speed moving objects, as shown in fig. 28.
For DMS applications, where accurately capturing the driver's movements and behaviors in real-time is crucial, the global shutter is the preferred choice.

3.1.6 Monochrome vs. Color Camera Modules

Monochrome Camera Module

Monochrome camera modules capture images exclusively in shades of gray.
One significant advantage of monochrome cameras is their higher sensitivity to light compared to color cameras. This increased sensitivity is due to the absence of a color filter array, which allows more light to

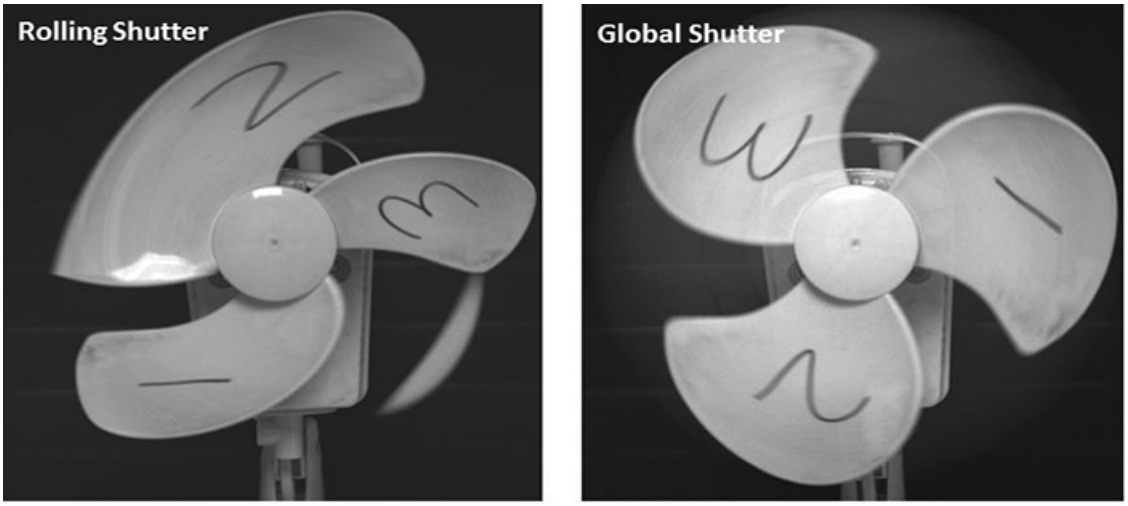


Figure 28: Comparison of rolling and global shutter in high-speed imaging of moving objects

reach the sensor. Consequently, monochrome cameras can capture clearer and sharper images in low-light conditions.

Additionally, monochrome cameras offer higher resolution. Each pixel in a monochrome camera captures all incoming light, whereas each pixel in a color camera captures only one color. This results in monochrome cameras being able to capture more details and produce higher-quality images.

However, a notable limitation of monochrome cameras is their inability to capture color images. If color information is essential for application, a color camera is necessary. Furthermore, monochrome cameras are typically more expensive than color cameras.

Color Camera Module
Color camera modules capture images in full color with many types of color filter arrays.
An advantage of color cameras is their ability to capture images in full color, which is crucial for applications requiring color information. Additionally, color cameras are generally more affordable than monochrome cameras, making them popular for consumer applications like smartphones and digital cameras.

Another benefit of color cameras is their ability to capture images at a faster rate. The color filter array in these cameras allows them to capture all three primary colors (red, green, and blue) simultaneously, enabling faster image capture.

However, color cameras have a lower sensitivity to light compared to monochrome cameras. The color filter array reduces the amount of light reaching the sensor, which can result in less sharp and detailed images in low-light conditions.

For applications like DMS, requiring high sensitivity and detailed resolution, especially in low-light conditions, monochrome cameras are preferable.

3.1.7 Protocol and Interface

When designing DMS, selecting the optimal interface for transmitting visual information is critical to its overall performance. This interface serves as the physical connection layer that links the camera to the

processing platform, facilitating image transmission and subsequent processing. Key considerations include throughput capabilities and transmission distance.

In response to increasing demands for high-speed connectivity, the market offers a range of flexible and robust interfaces. Among the most widely adopted are MIPI CSI-2, GMSL, and USB interfaces, each tailored to meet specific industry needs.

MIPI CSI-2

MIPI CSI-2, or Mobile Industry Processor Interface Camera Serial Interface Type-2, is a high-speed serial interface primarily developed for transmitting image and video data from mobile camera modules to embedded processors. Originally designed for mobile devices, MIPI CSI-2 has found widespread adoption in mobile phones, tablets, and handheld embedded systems due to its versatility and performance capabilities.

The interface supports a peak bandwidth of 6 Gbps, with a practical bandwidth typically around 5 Gbps. It utilizes four image data lanes, each capable of transmitting up to 1.5 Gbps, surpassing the speed of USB 3.0. MIPI CSI-2 is renowned for its efficiency and reliability, capable of handling video resolutions ranging from 1080p to 8K and beyond. It also minimizes CPU resource consumption, leveraging the capabilities of multi-core processors.

MIPI CSI-2 is designed with low power consumption in mind, making it well-suited for battery-powered embedded devices. Its scalability, enabled by multiple data lanes, allows for flexible adjustment of data transfer rates based on application requirements. The interface employs a differential signaling scheme, enhancing noise immunity and ensuring high reliability in data transmission.

However, MIPI CSI-2 has limitations, including a maximum supported cable length of 30 cm, which restricts its use over longer distances. Additionally, implementing MIPI CSI-2 requires specialized components, potentially increasing the cost of the embedded system.

USB

Universal Serial Bus (USB) is a widely adopted data transfer protocol in computing devices and embedded systems. In embedded vision systems, USB 2.0 and 3.0 protocols serve as common interfaces for transferring image and video data from camera sensors to host processors.

USB 2.0 supports a maximum data transfer rate of 480 Mbps, while USB 3.0 significantly increases this bandwidth to 5 Gbps. These interfaces are extensively utilized in embedded vision applications such as surveillance systems, industrial inspection, and machine vision, particularly favored for their low-cost and low-power attributes.

The plug-and-play capability is a notable advantage of USB camera interfaces, simplifying integration and reducing development costs. USB 3.0, with its higher bandwidth of up to 360 MB/s, aligns well with the USB3 Vision Standard in embedded vision systems, facilitating seamless device replacement and enhancing flexibility.

However, USB interfaces have limitations. Both USB 2.0 and 3.0 support a maximum cable length of 5 meters, which can restrict deployment options. Additionally, these interfaces lack dedicated video streaming

capabilities, potentially causing delays or loss of image data during transmission.

GMSL

Gigabit Multimedia Serial Link (GMSL) is a high-speed serial link protocol designed primarily for transmitting image and video data in embedded vision systems, particularly in automotive applications. GMSL enables the transmission of high-speed video, bidirectional control data, and power over a single coaxial cable, supporting distances of up to 15 meters with low latency and high frame rates.

GMSL utilizes differential pair transmission for enhanced noise immunity and data integrity. Its unique encoding scheme reduces electromagnetic interference (EMI), allowing for longer cable lengths and higher data rates compared to traditional serial link interfaces.

Commonly used in advanced driver assistance systems (ADAS), autonomous vehicles, and quality control systems, GMSL is favored for its high bandwidth, reliability, and suitability for long-distance transmission.

However, GMSL interfaces tend to be more expensive than alternatives like USB due to the specialized hardware and software required. Additionally, the protocol's complexity may pose challenges for users unfamiliar with its implementation.

In our project, we have chosen MIPI CSI-2 as the interface for transmitting visual information due to its compelling advantages in affordability and reliability. MIPI CSI-2 offers high-speed data transfer capabilities, making it well-suited for handling high-resolution video streams efficiently. Its low power consumption is crucial for our battery-powered application, ensuring extended operational lifetimes without compromising performance.

While MIPI CSI-2 excels in these areas, it's worth noting its limitations compared to USB and GMSL interfaces. Specifically, MIPI CSI-2 supports shorter cable distances compared to USB's 5-meter limit and GMSL's ability to extend up to 15 meters. This constraint may impact flexibility in our system setup but is outweighed by MIPI CSI-2's overall suitability for our application's needs.

3.1.8 Lens Selection

Even with the selection of the appropriate sensor format to cover our working plane, the choice of the lens will greatly affect the sensor performance as it can limit the sensor resolution or restrict the FOV. Hence, we were careful to ensure that the size of the lens is equal or greater than the size of the sensor (circle size) and provide a suitable FOV that is enough to cover the driver's upper body as the needed FOV is about (45 H) and (64 V). These calculations were done by a simple FOV calculator as shown in Fig. 29. The FOV shouldn't be too small, risking the loss of essential parts within the view, or excessively large, resulting in irrelevant data collection As illustrated in figure 30 .

3.1.9 Selected Camera (OV2311)

Based on the discussed parameters, we have selected the OV2311 camera from OmniVision for our application. Below are the detailed specifications of the OV2311:

- Resolution: 1600 x 1300 pixels (1.3 megapixels)

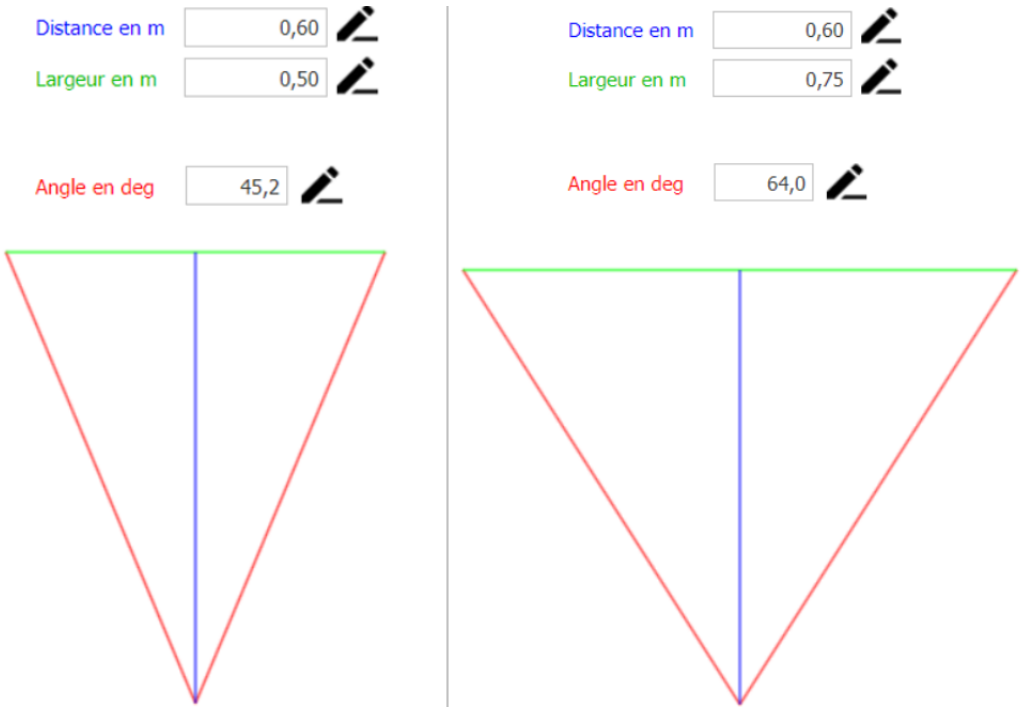


Figure 29: FOV calculations

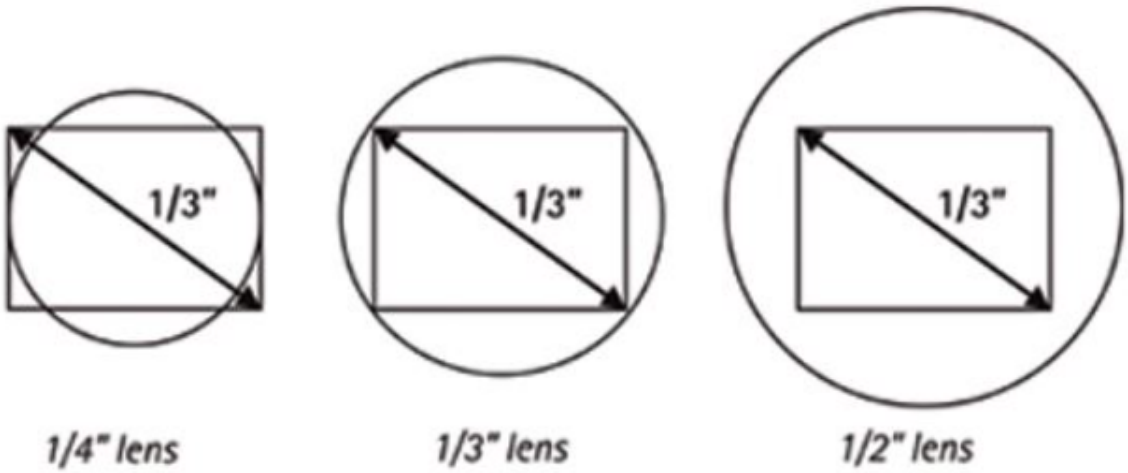


Figure 30: Image sensor circle size

- Sensor Type: CMOS
- Pixel Size: 3.0 µm
- Sensor Size: 1/4 inch
- Frame Rate: Up to 60 fps at full resolution
- Output Format: RAW Bayer format
- Sensitivity: 12 V/Lux-sec
- Dynamic Range: 105 dB
- Interface: MIPI CSI-2

However, due to purchasing constraints, we opted for the Sony IMX219 image sensor for our DMS. While the IMX219 may not fully meet our original criteria, its capabilities in image quality, low light sensitivity, and compact design make it a possible choice for us.

3.2 Processing Platform Selection

To achieve real-time performance, we have dedicated significant effort and time to conduct a thorough investigation in search of the ideal development board for our project. We recognize the criticality of selecting hardware that aligns with the objectives of our AI solution, including the required fps, resolution, and the ability to deploy and run AI models effectively.

This section presents a detailed justification for the selected hardware platform. It outlines the research and comparison process undertaken, providing insights into the evaluation criteria that guided our decision-making. Note that the terms Micro Control Unit (MCU) and hardware platform are used interchangeably throughout this section.

3.2.1 Project Requirements and Constraints

Performance Requirements:

- The DMS should be capable of processing and analyzing video data in real-time (soft real-time).
- Achieving a frame rate that ensures more smooth and accurate driver monitoring.
- It should be able to process and interpret driver behavior, including drowsiness, distraction, eye gaze tracking, and head pose estimation with high accuracy (>90%).
- It should support AI acceleration and deep learning capabilities to enable efficient execution of AI algorithms for facial recognition and behavior analysis.
- The MCU should have sufficient CPU processing power and memory capacity to handle the computational requirements of the DMS application.

Power Constraints:

- The MCU and image sensor should operate within specified power limits to prevent excessive power consumption.
- The system should be designed to optimize power usage and implement power-saving features when possible.

Communication Interfaces:

- The MCU should support communication interfaces necessary for the driver monitoring system, such as I2C, CAN, WI-FI, and Ethernet.
- The image sensor should have compatible interfaces with the MCU for data transfer and control.

Other Relevant Specifications:

- The MCU should have robust security features to ensure the protection of driver data and prevent unauthorized access.
- Both the MCU and image sensor should have a long product life cycle and reliable technical support from the manufacturers to ensure long-term availability and compatibility.

3.2.2 Development board Selection

Evaluation Criteria

Our selection of the development board was based on a comprehensive evaluation process that considered multiple factors. The following criteria and benchmarks played a significant role in our decision:

- **Architecture**
  We prioritized an ARM-based development board due to its widespread adoption, powerful performance, and industry-scale usage.

- **Processor Cores**
  To support multitasking and high performance, we need an MCU with more than one core. This allows for increased processing power, enabling the MCU to handle complex tasks and process data quickly.

- **Digital Signal Processor (DSP)**
  The presence of a DSP was crucial as it provides features like image enhancement and compression. This optimization is valuable for utilizing video streams from the camera in tasks such as facial recognition and gaze tracking. DSPs are optimized for real-time processing, ensuring time-critical tasks are handled within strict timing constraints while optimizing power consumption.

- **Accelerators**
  We need accelerators that support AI workload in our project, such as those designed for deep learning, matrix multiplications (MMA), and image signal processing (ISP). These accelerators enhance and optimize image quality while ensuring high computational performance.

- **Operations Per Second**
  Our benchmark for TOPS was a minimum of 3 trillion operations per second to support the high workload of running AI models, providing real-time responses, and optimizing power consumption.

- **Graphics processing unit (GPU)**
  We required a GPU with high performance, parallel processing capabilities, power efficiency, and robust graphics rendering.

- **Memory**
  Our chosen MCU needs memory to support AI models, the overall system, and endpoint connections to the server.

- **Communication Interfaces**

  - **CAN:** The MCU needs to support the CAN communication protocol for sensor interfacing, diagnostics, and validation.
  - **Wi-Fi:** The MCU requires Wi-Fi or ethernet capability to send updates to the server, and display driver status and logs, as it offers convenient development options.
  - **PCIe:** external storage and modules.
  - **SPI:** external modules.
  - **I2C:** camera sensor interface as most serializer-deserializers support I2C.
  - **UART:** serial interface for testing.

- **Camera Serial Interface (CSI)**
  The MCU needs CSI support to interface with cameras reliably, providing high-bandwidth and high-resolution video streams compared to other interfaces like USB.

- **HDMI or DSI**
  We require either HDMI or DSI support to stream the processed output.

- **Storage**
  The MCU needs high-speed, reliable, and secure storage to facilitate data logging, firmware and software updates, and real-time data processing. We also considered extended lifespan storage for longevity.

- **Linux Support**
  Linux support was a crucial factor in our selection process. We need an MCU with strong community support and compatibility with Linux, ensuring suitability for our embedded team.

- **Development Board** We prioritized a provider that offered a development board, simplifying the interfacing process for us beyond just a system-on-chip (SoC) or system-on-module (SoM).

- **Development Board Price**
  We considered the value-for-cost of the development board, seeking an affordable option without compromising essential features or performance.

Research and Analysis

Throughout our diligent research and analysis process, we have thoroughly explored multiple vendors, including Texas Instruments, NXP, Nvidia, Qualcomm, OmniVision, Xilinx, AMD, and Renesas, to identify the most suitable processor for our project. Each processor within their respective families was carefully evaluated against our specific requirements, and we have compiled the findings into a comprehensive comparison document.

To provide transparency and facilitate informed decision-making, we have prepared a detailed comparison table.1 summarizing the results of our extensive research as a shortlist of 6 out of 15 MCUs. The rest of MCUs were eliminated due to a lack of Linux support, not satisfying the minimal performance requirements, or very high pricing.

MCU	Processor Cores	DSP	Accelerator	TOPS	GPU	Memory	Peripherals	Storage	Dev Board Price
SK-TDA4VM (Texas Instruments)	Dual 64-bit Arm® Cortex®-A72 (2.0 GHz) + Six Arm® Cortex®-R5F MCUs (1.0 GHz)	C7x floating point, vector DSP, up to 1.0 GHz, 80 GFLOPS, 256 GOPS + Two C66x floating point DSP, up to 1.35 GHz, 40 GFLOPS, 160 GOPS	Deep-learning matrix multiply accelerator (MMA) + Vision Processing Accelerators (VPAC) with Image Signal Processor (ISP) and multiple vision assist accelerators + Depth and Motion Processing Accelerators (DMPAC)	8	3D GPU PowerVR® Rogue 8XE GE8430 (750 MHz), 96 GFLOPS, 6 Gpix/seca	4 GB LPDDR4-4266 + 8MB of on-chip L3 RAM + GPMC + 512KB on-chip SRAM	CAN-FD Wi-Fi slot 3x MIPI-CSI2.0 MIPI-DSI Tx HDMI port	One eMMC 5.1 interface + UFS 2.1 + others	$250
SK-AM69 (Texas Instruments)	Eight 64-bit Arm® Cortex®-A72 (2 GHz) + Dual-core Arm® Cortex®-R5F MCUs at up to 1.0 GHz + Dual-core Arm® Cortex®-R5F MCUs at up to 1.0 GHz	Four Deep Learning Accelerators	Deep-learning matrix multiply accelerator (MMA) + Two Vision Processing Accelerators (VPAC) with Image Signal Processor (ISP) and multiple vision assist accelerators + Depth and Motion Processing Accelerators (DMPAC)	32	3D Graphics Processing Unit, IMG BXS-4-64, up to 800 MHz, 50 GFLOPS, 4 GTexels/s, Supports up to 2048x1080 @60fps	4x 8GByte LPDDR4 DRAM (2133 MHz) + 512 Mb Non-Volatile Flash, Octal-SPI NOR	4x CAN-FD Wi-Fi slot 4x MIPI-CSI-4L MIPI-DSI Tx HDMI port	32GB eMMC	$400
BEAGL-BONE-AI-64 (Beagle Board)	Dual 64-bit Arm® Cortex®-A72 (2.0 GHz) + Six Arm® Cortex®-R5F MCUs (1.0 GHz)	C7x floating point, vector DSP, up to 1.0 GHz, 80 GFLOPS, 256 GOPS + Two C66x floating point DSP, up to 1.35 GHz, 40 GFLOPS, 160 GOPS	Vision Processing Accelerators (VPAC) with Image Signal Processor (ISP) and multiple vision assist accelerators + Depth and Motion Processing Accelerators (DMPAC)	8	3D GPU PowerVR® Rogue 8XE GE8430 (750 MHz), 96 GFLOPS, 6 Gpix/seca	4GB DDR4	2 x CAN Wi-Fi slot 2x MIPI-CSI2.0 4L MIPI-DSI Tx HDMI Display port (DP)	16GB embedded MultiMediaCard (eMMC) onboard flash storage	$190
SBC-S32V234 (NXP)	Arm® Cortex®-A53, 64-bit CPU -Up to 1000 MHz Quad Arm Cortex-A53 -32 KB/32 kB I-/D- L1 Cache Arm Cortex-M4, 32-bit CPU -Up to 133 MHz -16 KB/16 kB I-/D- L1 Cache	N/A	• Image Signal Processing • 2 x APEX2 – Image cognition Processing Open CL • h.264 Codec and MJPEG decoder	not provide	• 3D GPU GC3000 (4 Shader)	2 x 32b DDR3/LPDDR2 at 533 MHz	2x CAN 2x MIPI-CSI2 RGB to HDMI converter	Quad-SPI flash	$979.00 USD
The Coral Dev Board (NXP)	Arm Cortex-A53 MPCore platform Arm Cortex-M4 core platform	N/A	Edge TPU coprocessor	4 (2 TOPS per watt)	Vivante GC7000Lite 4 shaders 267 million triangles/sec 1.6 Gigapixel/sec 32 GFLOPs 32-bit or 64 GFLOPs 16-bit Supports OpenGL ES 1.1, 2.0, 3.0, 3.1, Open CL 1.2, and Vulkan	32/16-bit DRAM interface: LPDDR4-3200, DDR4-2400, DDR3L-1600	Wi-Fi 2x2 MIMO MIPI-CSI2.0 MIPI-DSI Tx HDMI port	eMMC 5.0 Flash QuadSPI Flash with support for XIP	$130
Jetson Orin NX 16GB (Nvidia)	8-core NVIDIA Arm® Cortex A78AE v8.2 64-bit CPU 2MB L2 + 4MB L3 (2 GHz)	N/A	2x NVDLA v2.0 (614 MHz) + 1 x PVA v2.0	100	1024-core NVIDIA Ampere architecture GPU with 32 Tensor Cores (918 MHz)	16GB 128-bit LPDDR5 102.4GB/s	1 CAN MIPI-CSI-2 + D-PHY 2.1 HDMI port	(Supports external NVMe)	$1,500

Table 1: Top 6 selected HW platforms

Selected SoC Overview

The thorough analysis of the available options gathered with the careful consideration of the system requirements led to the choice of the TDA4VM family from Texas Instruments (TI). However, selecting a proper development board was a bit tricky as the powerful TDA4VM is incorporated by many third-party organizations other than TI resulting in a great pool of options for us to choose from. Eventually, the BeagleBone AI-64 (BBAI-64) kit is selected as a development board providing a smooth interfacing scheme with the TDA4VM SoC. Provided hereby, in table.3.2.2 are the main specs of the BBAI-64 AI kit from BeagleBoard [4].

Chipset Features (TDA4VM)	
Feature	Description
Processor	Dual 64-bit Arm Cortex-A72 microprocessor subsystem at up to 2.0 GHz
Cache	1MB shared L2 cache per dual-core Cortex-A72 cluster, 32KB L1 DCache, 48KB L1 ICache per core
DSP	C7x floating point, vector DSP, up to 1.0 GHz, 80 GFLOPS, 256 GOPS
Deep-learning Accelerator	Matrix multiply accelerator (MMA), up to 8 TOPS (8b) at 1.0 GHz
Vision Processing	VPAC with ISP and multiple vision assist accelerators
Depth and Motion Processing	DMPAC
Microcontrollers	Six Arm Cortex-R5F MCUs at up to 1.0 GHz
Additional DSP	Two C66x floating point DSP, up to 1.35 GHz, 40 GFLOPS, 160 GOPS
GPU	3D GPU PowerVR Rogue 8XE GE8430, up to 750 MHz, 96 GFLOPS, 6 Gpix/sec
Memory Subsystem	Up to 8MB of on-chip L3 RAM with ECC and coherency
Audio	Twelve Multichannel Audio Serial Port (MCASP) modules

Table 2: TDA4VM features

Board Features (BBAI-64)	
Feature	Description
Expansion Headers	BeagleBone® Black header compatibility for expansion with existing add-on capes
MikroBus	MikroBus shuttle header giving access to hundreds of existing Click sensors and actuators
Memory	4GB LPDDR4, 16GB eMMC flash with high-speed interface, MicroSD card slot
High-Speed Interfaces	M.2 E-key PCIe connector for WiFi modules, USB 3.0 Type-C, 2* USB 3.0 Type-A, Gigabit Ethernet
Camera and Display	Mini DisplayPort, 2* 4-Lane CSI connectors, 4-Lane DSI connector
User Interfaces	Boot, Reset, Power buttons, Power LED, 5 User LEDs, 2* UART debug, JTAG connector

Table 3: BBAI-64 board features

Technical Justification

Given the wide range of available edge AI platforms, we conducted a careful assessment of the features, facilities, and even technical support provided by various vendors in the silicon market. Our choice of the BBAI-64 over the others is mainly based on the following:

Figure 31: BBAI-64 development board

- **Processing Power**

  Equipped with a Dual 64-bit Arm® Cortex®-A72 (2.0 GHz) and Six Arm® Cortex®-R5F SoCs (1.0 GHz), the TDA4VM stood out against all the competitors in the same category. Google Coral, Jetson boards by Nvidia (Nano and TX2), OmniVision's OAX8000, Renesas's V2M_EVK, and the

ZCU104 from Xilinx AMD are all examples of hardware platforms beaten by the TDA4VM in terms of processing power. A comprehensive comparison is provided in the Google sheet in section 3.2.

- **Cost Per TOPS**
  Cost analysis played a vital role in the selection process where we analyzed the value proposed by each company versus the cost of the product. The cost per TOPs factor is simply an indicator of the processing capacity for each unit price. The TDA4VM holds a $31.25/TOPs making it the best value for money choice. The only item we needed to consider is the BBAI-64 from BeagleBoard which employs the very TDA4VM having a $23.75/TOPs. The cost efficiency comes at the price of the accessibility of the CAN interface that is easily accessed on its SK-TDA4VM counterpart. It's worth noting that the BBAI-64 possesses a 16GB embedded MultiMediaCard (eMMC) onboard flash storage that is not provided by the SK-TDA4VM kit. The turning point is that the BBAI-64 is designed just for exploration and some capabilities of the TDA4VM are not accessible to the user.

Figure 32: SK-TDA4VM development board

- **AI Accelerators**
  The TDA4VM is equipped with an array of AI accelerators that, according to the publicly available benchmarks, set it apart from its competitors. Although it doesn't have either a Tensor Processing Unit (TPU) or a dedicated FPGA, the TDA4VM outperforms the boards provided by Google and Xilinx AMD that support these features. Google Coral, for instance, records a processing capacity of 4 TOPs, even though it incorporates a dedicated TPU. The TDA4VM manages to score high at this by employing a diverse set of independent accelerators including a dedicated on-board DSP unit.

- **Automotive Interfaces**
  Developed particularly for this purpose, the TDA4VM easily satisfied our system requirements regarding automotive connectivity. However, this factor was not of much impact as most of the available options supported the very CAN-FD, ethernet, and other automotive interfaces.

- **Camera Interface**
  The TDA4VM supports the MIPI CSI-2 and the high-speed FPD-link (through an adaptor) interfaces that are most commonly used for such edge-AI vision applications. This opened the door for the incorporation of imager data transmission for distances up to 15 meters through the FPD-Link interface.

- **Development Environment**
  Regarding the hardware environment, SK-TDA4VM provides a fully functional platform exploiting all

the features of the onboard SoC. When it comes to software, TI provides a comprehensive SW support package facilitating the development process and cutting down the project execution time.

- **Ease of Deployment**
  TI provides various deployment options such as the Edge AI Studio which facilitates the hardware deployment process.

Development and Community Support The development forums and TI's proactive E2E technical support are some of the key advantages that supported our choice of the SK-TDA4VM kit. Needless to mention the active and continuous Linux support maintained by company engineers and third-party organizations. Items such as Board Support Packages (BSPs), device drivers, etc, are elegantly maintained making the development process as smooth as it could be.

Risk Assessment and Mitigation

To analyze the risk and how to mitigate it we need first to identify our requirements from the selected MCU (BBAI-64).

MCU Requirements analysis

- We need to detect the driver state accurately by running multiple AI models in parallel.
- We tend to deploy an optimized version to fit the processing power.
- We need a gateway interface to communicate over the network to the server (ex: Ethernet, USB modem, or Wi-Fi).
- We need automotive bus technologies (interfaces) to check the system diagnostics (ex: CAN bus).
- We need interfaces with the image sensor through (ex: a MIPI CSI-2).
- We need interfaces with external modules like SSD hard disks or external accelerators (DSPs) (ex: PCIe or SPI).
- We need some interfaces to communicate with the car units like the ADAS ECU (ex: automotive Ethernet, LIN, or CAN).
- We need this board to comply with the automotive standards (ISO 26262) like functional safety standards.

Consequences of faulty selection for the MCU:

- A SoC that does not support optimization (quantization) or has insufficient processing power to run all the DL models in parallel will affect the accuracy of the detection as higher resolution is inversely proportional to the frame rate for the same processing power so it will be required to reduce the frame rate for each model to operate together, so the resolution decreases, and subsequently, the accuracy decreases.
- The current situation in total is considered an optimistic case, the worst-case scenario will be that we cannot run all the models together, as one of them may fail which will break the system's reliability and give unpredictable results. Inconsiderable choices of the development board interfaces will limit our scalability options and inefficient utilization of the MCU peripherals. We may also face many bottleneck situations with no outlets like being limited in processing power if no interface for external accelerators is provided. Another problem will be the absence of a reliable interface with the camera,

other car ECUs, or the gateway which may lead to a drop in camera frames, a slower call of action with a higher probability for EMI (electromagnetic interference), and an inability to communicate with the server smoothly.

- The absence of software board support packages (BSPs), SDK, and official technical support (i.e. third-party support for the Yocto project in Nvidia kits) may lead to a hard software development cycle and will be a time-consuming process that may cause the project to fall behind schedule.
- A board that does not comply with automotive standards like ISO26262 will not be considered to go on production steps.

Risk Assessment:
Each system requirement is considered a task and each task has its risk, task risk assessment could be estimated by the risk matrix, which considers two main parameters: the consequences of requirement non-satisfaction and the likelihood or the probability such this sequence happens, considering both parameters give us more insights if we need to take more actions before going through the implementation steps, kindly you can check the risk assessment matrix through the risk analysis sheet 6

The Adopted Approach to Overcome Risky Situations and their Consequences
To make sure that our selection justifies the previous requirements we did the following:

- Investigated through different types of MCU and development kits for each of these service providers like Texas Instruments, NXP, Qualcomm, Nvidia, and Google.
- Read many research papers about DL model deployment on embedded targets for edge AI vision applications.
- Investigated research papers about DL model optimization in run time and size for embedded targets using some techniques like quantization, pruning, and clustering.
- Read many research papers about DL models benchmarking for embedded targets which compare different DL models for different embedded targets based on accuracy and run (inference) time.
- Tested some models from Edge AI Studio a TI platform for benchmarking and model testing done with pre-trained models from their model zoo.
- Checked TI documents for benchmarking on Edge AI processors.

Countermeasures and Final Result:

- MCUs with processing power lower than 3 TOPs were eliminated.
- MCUs with incompatible accelerators with the benchmarked DL models were eliminated.
- MCUs that do not have strong technical and community support or unofficial support for Linux kernel-based images were eliminated.
- MCUs that do not support optimization (quantization) or whose effect is neglectable were eliminated.
- Development boards with incomplete satisfaction for urgent interfaces were eliminated.
- Development boards that do not comply with the safety standards were eliminated.

After this evaluation process of various development boards, our narrowed selection comprises the SK-

TDA4VM, SK-AM69, and BBAI-64 from Texas Instruments, along with the SBC-S32V234 from NXP. Each of these options has demonstrated a low-risk profile, meeting the specified project requirements effectively.

Our top choice is the BBAI-64 from Texas Instruments. This decision is primarily driven (over the other 3 boards) by its competitive pricing and extensive track record in supporting DMS projects.

4 App Deployment

In the rapidly evolving landscape of automotive safety systems, the efficient deployment of applications on robust and capable hardware platforms is paramount. This section delves into the deployment of our DMS application and explore the process of transitioning our DMS application from a development environment to a production-ready state on the BeagleBone AI-64, leveraging the advanced capabilities of the Texas Instruments TDA4VM.

4.1 SDK

Before diving into the development process of our Driver Monitoring System (DMS) application, understanding the Software Development Kit (SDK) provided by Texas Instruments (TI) is crucial. The TI SDK plays a pivotal role in optimizing the full potential of our chosen hardware, the BeagleBone AI-64, by offering a comprehensive suite of tools and libraries.

To deploy our application on the TDA4VM platform effectively, we harness the power of dedicated hardware accelerators. These accelerators require communication to coordinate task assignments. OpenVX stands as an open, royalty-free standard by the Khronos Group, designed for cross-platform acceleration of computer vision applications. Texas Instruments' implementation, TI OpenVX (TIOVX), tailored for processors like those in the Jacinto and Sitara families, optimizes performance and power efficiency, unlocking the accelerators' capabilities.

Utilizing hardware accelerators ensures superior performance and efficiency, circumventing constraints seen with CPU and GPU-based execution. These specialized units efficiently handle image processing and intensive computer vision tasks, minimizing system latency and freeing up CPU and GPU resources for other operations.

Our strategy builds upon OpenVX, providing a highly optimized execution framework for computer vision tasks. TIOVX implementation on the BeagleBone AI-64 enables us to leverage OpenVX APIs seamlessly, offloading demanding vision processing tasks to accelerators. This approach enhances processing speed and power efficiency, crucial for our application's demands.

The TI SDK integrates essential components such as GStreamer, OpenCV, TensorFlow Lite (TFLite), ONNX, and Neo-AI Deep Learning Runtime (DLR). This integration adds an abstraction layer, accelerating development by leveraging existing libraries and frameworks. By combining these tools, we streamline our development process and optimize our application for performance and efficiency in our embedded vision system.

The Processor SDK Linux for Edge AI can be divided into 3 parts, Applications, Processor SDK Linux and Processor SDK RTOS, As shown in Fig. 33

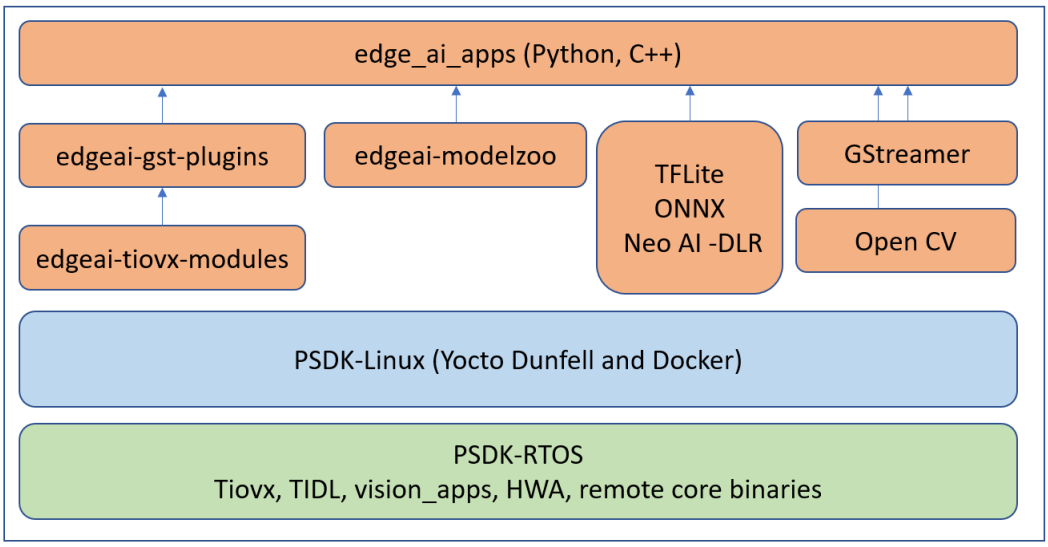


Figure 33: Processor SDK components

Fig.34, illustrates the assignment of the processing tasks to the accelerator cores of the SoC. In our implementation, the preprocessing node is assigned to both the VPAC and the DSP C661 core. The face detection task is assigned to the DSP C71 core with the MMA. Finally, the postprocessing task is assigned to the DSP C662 core. The secret sauce of this pipeline lies in the high computational power of the MMA reaching 8 Tera Operations Per Second (TOPS). This allows fast inferencing of the deep learning model used for face detection which, otherwise, would have represented a huge bottleneck.

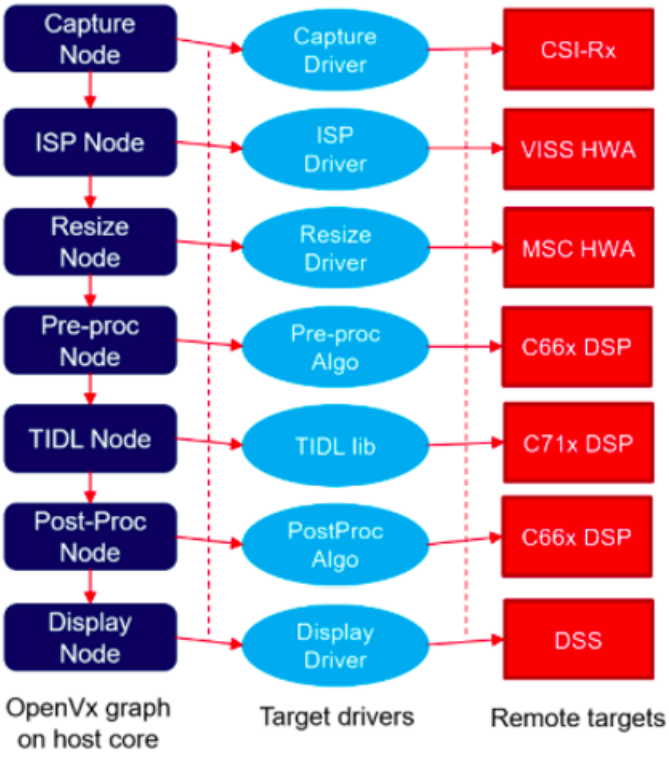


Figure 34: OpenVX Pipeline

4.2 Implementation

Our implementation can be divided into two major tasks: Integrating the camera into our app Modifying the app on SIL to fit the hardware

In this section, we will discuss the methodology for both tasks.

4.2.1 Camera Integration [29] [28] [25]

In our DMS, we used the 8-megapixel IMX219-160IR Camera from Waveshare. The camera comes with a CMOS sensor with a resolution of 3280 x 2464 pixels and a 1/4-inch sensor size. With an aperture of 2.35 and a focal length of 3.15mm, it offers a wide diagonal field of view (FOV) of 160 degrees, crucial for capturing a broad area within the vehicle cabin. The lens, measuring 6.5mm x 6.5mm, minimizes distortion to less than 14.3%, ensuring accurate image representation for effective driver monitoring applications.

4.2.2 Camera Interface

MIPI CSI-2 is a widely used camera interface standard that efficiently transmits image data from cameras to host devices. CSI-2 is divided into three main layers: PHY Layer, Protocol Layer, and Application Layer, as illustrated in the figure 35.

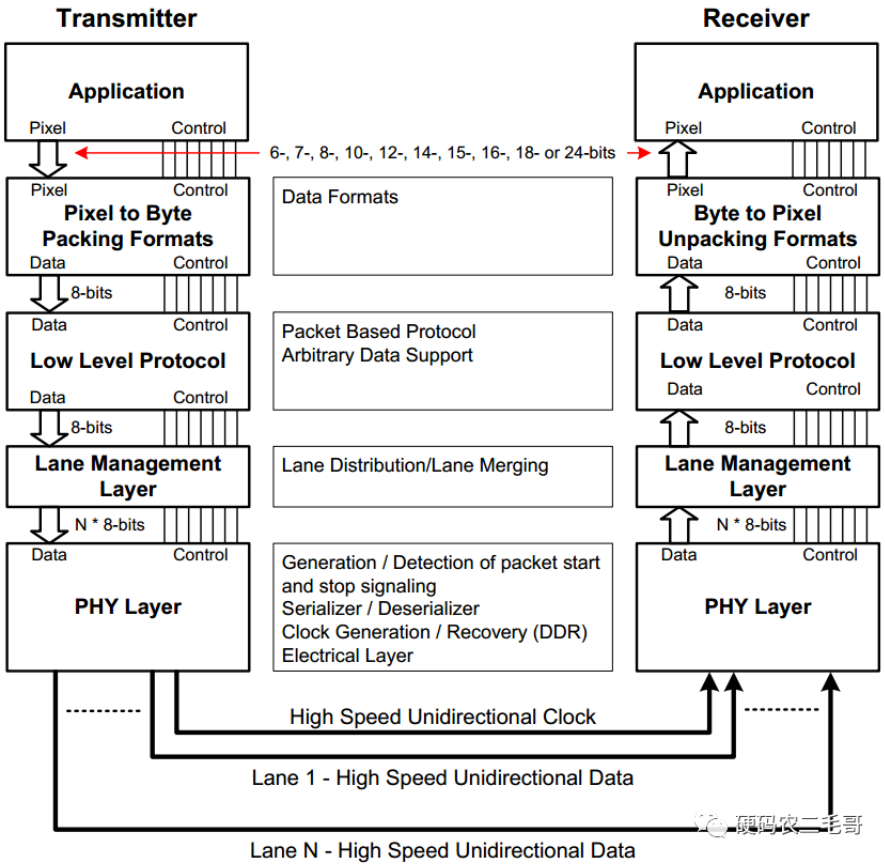


Figure 35: MIPI CSI Layered Architecture.

PHY Layer The PHY (Physical) Layer is responsible for the physical transmission of data and consists of:

- **Data Lanes**: 1 to 4 data lanes are used for serial data transmission. Each lane can operate at various speeds, contributing to the overall bandwidth.
- **Clock Lane**: A single clock lane synchronizes data transmission across the data lanes, ensuring accurate sampling and reconstruction.
- **D-PHY**: Data output from the MIPI camera is input into the D-PHY in the CSI-2 Receiver Subsystem, maintaining signal integrity through differential signaling to minimize electromagnetic interference and noise.

Protocol Layer The Protocol Layer defines the rules for data encapsulation, transmission, and error handling. It is further divided into three sub-layers:

- **Pixel/Byte Packing/Unpacking Layer**: This layer supports various pixel formats, ranging from 6 bits to 24 bits per pixel. On the receiving side, it disassembles the bytes from the Low-Level Protocol into pixel formats and transmits them to the Application Layer.
- **Low-Level Protocol (LLP)**: LLP handles the transmission of arbitrary data over short or long packets. Each packet transmission starts with Low-Power State (LPS) to Start of Transmission (SoT) and ends with End of Transmission (EoT) to LPS. This ensures structured data transfer, supporting both long and short packets.
- **Lane Management**: Depending on application requirements, 1, 2, 3, or 4 data lanes can be selected to match the required bandwidth, allowing for flexibility and scalability.

Application Layer The Application Layer is responsible for high-level encoding and parsing of the data stream. This layer ensures that the data is formatted correctly for the host processor and manages the following tasks:

- **Data Formatting**: Converts raw image data into specific formats (e.g., RGB or YUV) as required by the host processor.
- **Frame Management**: Synchronizes and sequences image frames, ensuring they are correctly ordered and timed.
- **Control Commands**: Handles control commands sent from the host processor to the camera sensor, such as exposure settings, focus adjustments, and other camera configurations.

Our system transmits data and clock signals using the CSI-2 interface utilizing 2 lanes. Outputs of data and clock come from CSI-2 output pins (DMO1P/DMO1N, DMO2P/DMO2N, DCKP/DCKN). A pair of DMO1P/DMO1N is called Lane 1 data and a pair of DMO2P/DMO2N is called Lane 2 data. Clock signals also come from CSI-2 output pins, DCKP/DCKN. The maximum output data rate is 912 Mbps/lane.
Additionally, the IMX219 camera module employs a 2-wire serial communication method for sensor control, adhering to the Camera Control Interface (CCI) standard. CCI utilizes an I2C fast-mode plus interface with a clock frequency (fSCK) ranging from 11.4 to 27 MHz, ensuring compatibility with standard I2C protocols. This communication circuit allows access to control and status registers of the IMX219, facilitating precise configuration and monitoring of the camera's operations.

4.2.3 Vision Pre-processing Accelerator (VPAC) Subsystem

To address the lack of an ISP (Image Signal Processor) unit for handling image processing tasks such as exposure control and white balance adjustment in our camera module, we utilize the Vision Pre-processing Accelerator (VPAC) on the TDA4 SoC illustrated in figure36.
The VPAC performs essential image pre-processing operations, including de-noising, scaling, and color space conversion, optimizing the raw image data from the IMX219 sensor for subsequent analysis and inference tasks in driver monitoring applications. This integration ensures high-quality image output and efficient processing within the system architecture. The VPAC also features an imaging pipeline that integrates with external camera sensors and performs memory-to-memory (M2M) processing on pixel data.
The VPAC subsystem includes a set of common vision primitive functions for pixel data processing. In our

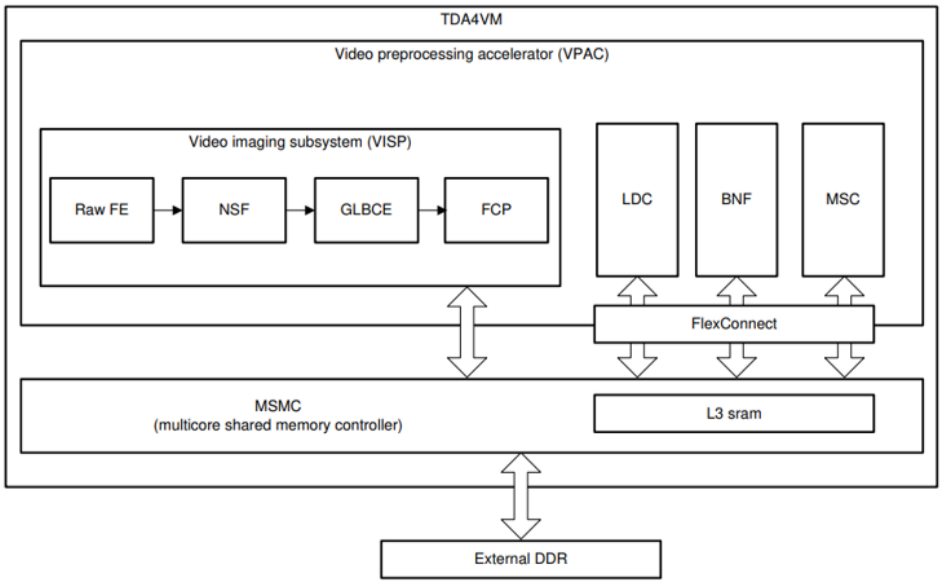


Figure 36: VPAC Overview

system, we utilize the Vision Imaging Sub-System (VISS), which processes raw data, including Wide Dynamic Range (WDR) merge, Defect Pixel Correction (DPC), Lens Shading Correction (LSC), Global/Local Brightness and Contrast Enhancement (GLBCE), demosaicing, color conversion, and Edge Enhancement (EE). It operates on sensor data either on-the-fly or in memory-to-memory mode.
also, the Lens Distortion Correction (LDC) block within the VPAC is a YUV domain processor designed to perform perspective and geometric transforms.
The output of the LDC block can be sent to external memory (DDR) or other VPAC sub-modules for further pre-processing via local shared memory.

4.2.4 Connecting IMX219 Camera with BeagleBone AI-64

To start using the IMX219 camera with your BeagleBone AI-64, follow these steps:
Enabling I2C Firstly, enable I2C communication with the board, as CSI cameras use I2C for communication. By default, I2C is not enabled. Follow these steps to enable it:

1. Install the overlays for I2C:

```
debian@BeagleBone:~$ cd /opt/source/dtb-5.10-ti
debian@BeagleBone:/opt/source/dtb-5.10-ti$ git pull
debian@BeagleBone:/opt/source/dtb-5.10-ti$ make
debian@BeagleBone:/opt/source/dtb-5.10-ti$ sudo make install
```

2. Add overlays to `extlinux.conf`:

```
debian@BeagleBone:~$ sudo vim /boot/firmware/extlinux/extlinux.conf
```

   Add the following line:

```
fdtoverlays /overlays/BONE-I2C1.dtbo /overlays/BONE-I2C2.dtbo
```

between:

```
fdtdir /
initrd /initrd.img
```

3. Reboot the board:

```
debian@BeagleBone:~$ sudo reboot
```

Verifying I2C Enablement To check if I2C is enabled, run the following command:

```
debian@BeagleBone:~$ sudo beagle-version | grep UBOOT
```

You should see output similar to:

```
UBOOT: Booted Device-Tree:[k3-j721e-beagleboneai64.dts]
UBOOT: Loaded Overlay:[BONE-I2C1.kernel]
UBOOT: Loaded Overlay:[BONE-I2C2.kernel]
```

Verify the I2C device trees:

```
debian@BeagleBone:~$ ls /dev/bone/i2c/
1  2
```

Enabling IMX219 Camera Identify and enable the IMX219 camera. If you have already installed the I2C overlays, the IMX219 overlays are included. If not, run the following commands:

```
debian@BeagleBone:~$ git clone -b v5.10.x-ti-unified https://git.beagleboard.org/beagleboard/Beagl
debian@BeagleBone:~$ cd ./BeagleBoard-DeviceTrees/
debian@BeagleBone:~$ make
debian@BeagleBone:~$ sudo make install_arm64
```

Add the following overlays to `extlinux.conf`:

```
fdtoverlays /overlays/BBAI64-CSI0-imx219.dtbo /overlays/BBAI64-CSI1-imx219.dtbo
```

After rebooting the board, check for the IMX219 bin files in the `/opt/imaging` folder. If not present, install them:

```
wget https://github.com/Hypnotriod/bbai64/raw/master/imaging.zip
sudo unzip imaging.zip -d /opt/
```

Verifying Camera Detection To verify camera detection, run the following command:

```
debian@BeagleBone:~$ sudo /opt/edge_ai_apps/init_script.sh
```

You should see output similar to:

```
CSI Camera 0 detected
    device = /dev/video2
```

```
name = imx219 7-0010
format = [fmt:SRGGB8_1X8/1920x1080]
subdev_id = 2
isp_required = yes
```

Testing the Camera To test the camera, run the RPI example in EdgeAI:

```
debian@BeagleBone:~$ cd /opt/edge_ai_apps/apps_python/
debian@BeagleBone:/opt/edge_ai_apps/apps_python$ ./app_edgeai.py ../configs/[rpi.............].ya
```

You can change configurations by editing:

```
edge_ai_apps/configs/[rpi.............].yaml
```

Modifying IMX219 to 10-bit Mode
To modify IMX219 from 8-bit to 10-bit mode, follow these steps:

- Edit `/opt/edge_ai_apps/scripts/setup_cameras.sh`:

```
CSI_CAM_0_FMT='[fmt:SRGGB10_1X10/1920x1080]'
CSI_CAM_1_FMT='[fmt:SRGGB10_1X10/1920x1080]'
```

- Change the imaging binaries to use 10-bit versions:

```
mv /opt/imaging/imx219/dcc_2a.bin /opt/imaging/imx219/dcc_2a_8b.bin
mv /opt/imaging/imx219/dcc_viss.bin /opt/imaging/imx219/dcc_viss_8b.bin
mv /opt/imaging/imx219/dcc_2a_10b.bin /opt/imaging/imx219/dcc_2a.bin
mv /opt/imaging/imx219/dcc_viss_10b.bin /opt/imaging/imx219/dcc_viss.bin
```

- Set the input format in the `/opt/edge_ai_apps/configs/rpiV2_cam_example.yaml` as `rggb10`.

4.2.5 Data Flow

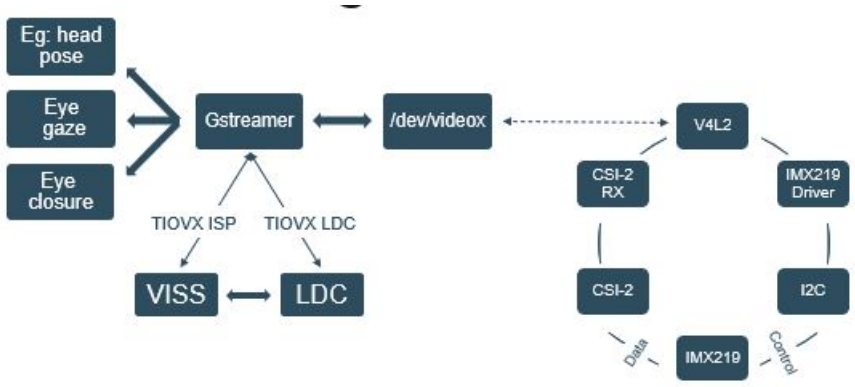


Figure 37: Data Flow Diagram

Our system's data flow, depicted in Figure 37, commences with the IMX219 sensor. The VPAC processes the captured data and sends feedback signals back to adjust the sensor parameters dynamically. The processed data is transmitted via 2 lanes to the CSI-2 RX interface on the board, managed by the V4L2 subsystem.

GStreamer as the Framework

GStreamer serves as our primary multimedia framework, facilitating the construction of complex media processing pipelines. While GStreamer handles basic multimedia tasks, leveraging the full capabilities of the VPAC necessitates integrating OpenVX.

OpenVX and TIOVX

OpenVX is an industry-standard API for accelerating computer vision applications, optimizing performance across heterogeneous platforms. Texas Instruments provides TIOVX, a tailored implementation of OpenVX for their SoCs like the TDA4, which includes optimized libraries and interfaces for VPAC functionalities. Figure 38 shows the GStreamer element after integrated with OpenVx.

Figure 38: Interplay of OpenVx and GStreamer elements

GStreamer Pipeline

Here's the GStreamer pipeline, tailored for our setup:

```
v4l2src device=/dev/video2 ! queue leaky=2 ! video/x-bayer, width=1920, height=1080,
    framerate=30/1 ,format=rggb ! tiovxisp sink_0::device=/dev/v4l-subdev2 sensor-name=
    SENSOR_SONY_IMX219_RPI dcc-isp-file=/opt/imaging/imx219/dcc_viss.bin sink_0::dcc-2a-file
    =/opt/imaging/imx219/dcc_2a.bin format-msb=7 ! video/x-raw,format=NV12 ! appsink
```

Listing 1: GStreamer Pipeline Example

Explanation of GStreamer Elements in the Pipeline

- **v4l2src**:
  - **Purpose**: Captures video frames from `/dev/video2`.
  - **Parameters**: No specific parameters are defined beyond the device name (`device=/dev/video2`).
- **queue**:
  - **Purpose**: Buffers incoming video frames to handle timing variations (`leaky=2` allows limited buffering).
- **video/x-bayer**:
  - **Purpose**: Specifies incoming video data in Bayer format.
  - **Parameters**:
    - `width=1920, height=1080`: Sets frame resolution.
    - `framerate=30/1`: Defines frame rate (30 frames per second).
    - `format=rggb`: Specifies Bayer pattern (Red-Green-Green-Blue).

- **tiovxisp**:
  - **Purpose**: Uses TIOVX ISP for processing Bayer-patterned frames.
  - **Parameters**:
    - `sink_0::device=/dev/v4l-subdev2`: Specifies sub-device for configuration.
    - `sensor-name=SENSOR_SONY_IMX219_RPI`: Identifies sensor type.
    - `dcc-isp-file=/opt/imaging/imx219/dcc_viss.bin`: DCC file for ISP configuration.
    - `sink_0::dcc-2a-file=/opt/imaging/imx219/dcc_2a.bin`: DCC file for 2A operations.
    - `format-msb=7`: Sets most significant bit format.
- **video/x-raw,format=NV12**:
  - **Purpose**: Converts processed output to NV12 format (YUV).
- **appsink**:
  - **Purpose**: Terminates pipeline, providing access to processed frames.

4.2.6 Main App Integration

In the development of our DMS, we explored two distinct approaches to implement and optimize our application: shaping our app as edge AI apps using their wrappers and running our app outside of the edge AI framework. Each approach has its own set of advantages and challenges, which we carefully evaluated and tested to determine the best fit for our system requirements.

- **Approach 1:** Shaping Our App as Edge AI Apps Using Wrappers
  In the section2 we have seen our application archectecture and modules. To make it fits the hardware we break it down into manageable pieces. We modified the inputs and outputs of each module to integrate seamlessly into the edge AI pipeline using its wrappers and APIs.
- **Approach 2:** Running Our App Outside of Edge AI In this approach, we developed our application using traditional software development techniques, without relying on the specialized edge AI wrappers. This involved writing custom code for model inference, data processing, and integration with the system hardware.
  This approach provided us with greater control over the processing pipeline and allowed us to experiment with different optimization strategies. However, it required deep engagement in AI model deployment, as discussed in Section 5.

5 AI Models Deployment

5.1 Introduction to Deep Learning Model Deployment

After the model training and fine-tuning phases, and once the model has been thoroughly tested and validated in the testing environment with satisfying results on host machines, the next step is to integrate and deploy it in the production environment. The process of integrating a machine learning model into a production environment where it can take in an input and return an output is known as model deployment. Production environments vary based on the product and the use-case, for example edge devices, cloud services, AI servers, data centers, etc. For our DMS product, we have more than one solution to set the models up and running:

- Solution 1: Deploy the DMS model on an embedded device.
- Solution 2: Cloud Deployment on AWS (Cloud Service)

5.1.1 Benefits and Constraints of Edge Deployment

Benefits:

- Speed: On-device inference can significantly reduce inferencing time for models optimized to work on less powerful hardware.
- Privacy: Data remains on the device, enhancing safety and privacy.
- Offline Functionality: The model can still perform inference even without an internet connection.
- Cost Efficiency: Offloading inference to the device reduces cloud serving costs.

Constraints:

- Model Size Limitation: The model must be optimized and small enough to run on the device.
- Hardware Processing Limitation: The model needs to be optimized to run on less powerful hardware.

These limitations can be addressed through optimization techniques like quantization, pruning, and clustering, which will be discussed in this chapter. Based on the DMS requirements, faster, safer, and cheaper solutions are mandatory for a real-time and efficient system, making the deployment on embedded device solutions more preferred.

5.1.2 Benefits and Constraints of Cloud Deployment

Benefits:

- Scalability: Cloud services can easily scale to handle large volumes of data and complex models.
- Ease of Maintenance: Updates and maintenance can be centrally managed, simplifying the process.
- Computing Power: Access to powerful cloud-based GPUs and TPUs can significantly enhance model performance.

Constraints:

- Latency: Data needs to be sent to and from the cloud, which can introduce latency and affect real-time performance.

- Privacy Concerns: Sending data to the cloud can raise privacy and security issues.
- Cost: Cloud services can incur ongoing costs, especially with large-scale data processing and storage needs.

By comparing the benefits and constraints of both edge and cloud deployments, it becomes evident that for a real-time and efficient DMS, edge deployment is often more advantageous for our case.

5.2 Texas Instruments (TI) Hardware Dependent Deployment

5.2.1 Texas Instruments Deep Learning (TIDL) [30]

TI provides within their SDK an optimized framework for deploying deep learning models on their own hardware, known as Texas Instruments Deep Learning (TIDL). It's a framework developed by Texas Instruments (TI) that facilitates deep learning inference on their embedded systems, particularly on their TDAx family of processors like the TDA4VM SoC. TIDL is optimized to run efficiently on these hardware platforms, allowing developers to deploy and execute deep learning models for various applications such as automotive driver assistance systems and industrial automation to solve problems like image classification, object detection, and semantic segmentation. It provides tools and libraries that enable developers to integrate and optimize neural networks for real-time processing on TI's embedded platforms.

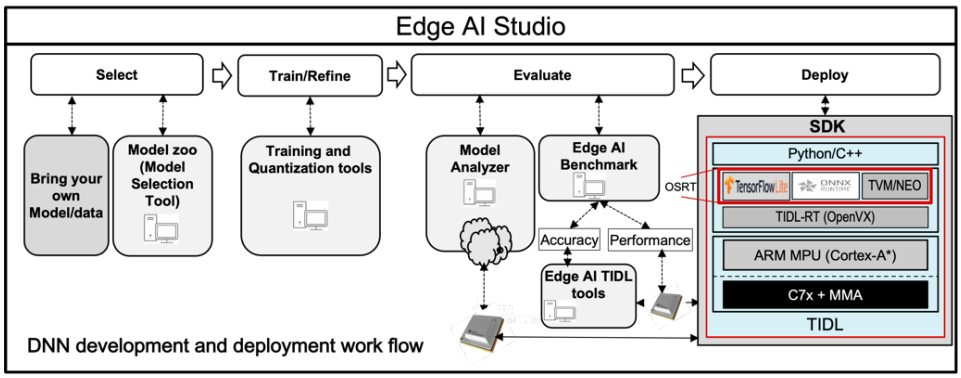


Figure 39: Development and deployment workflow overview

5.2.2 ONNX Conversion [17]

Open Neural Network Exchange (ONNX) is an open file format developed initially by Facebook and Microsoft, with ongoing support from organizations like IBM, Amazon (AWS), and Google. It is designed to enable machine learning models to be used across different AI frameworks and hardware, facilitating seamless transitions between frameworks. For example, a model trained in PyTorch can be exported to ONNX format and imported into TensorFlow.

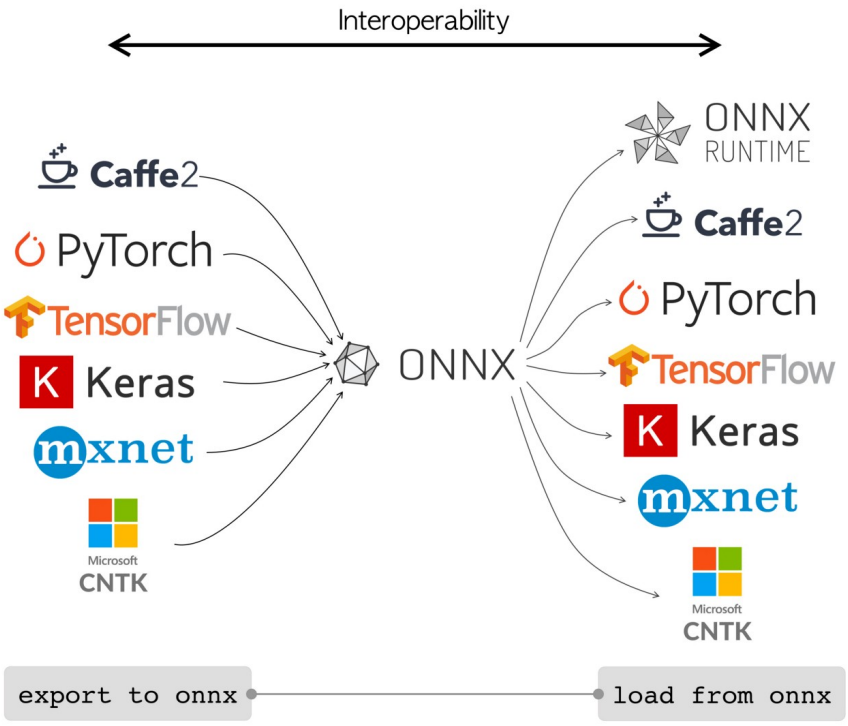


Figure 40: TIDL framework overview

5.2.3 ONNX Runtime [18]

ONNX Runtime is a versatile, cross-platform accelerator for ONNX models. It optimizes execution by leveraging hardware-specific capabilities, allowing models to run efficiently on various platforms, including CPUs, GPUs, and specialized accelerators. ONNX Runtime supports multiple frameworks such as PyTorch, TensorFlow, TFLite, and scikit-learn.

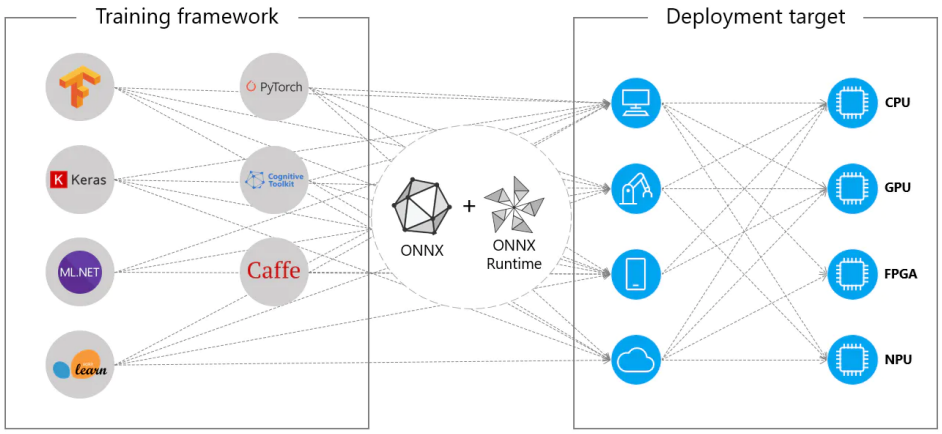


Figure 41: ONNX conversion process

Based on our selection for edge computing and our hardware selection for the TDA4VM SoC, we are now limited with some options for the development and deployment plan.
TIDL provides three options for model deployment related to the used runtime engine, these options are as follows:

1. Tensorflow Lite runtime based Heterogeneous Execution on A72+C7x-MMA

2. ONNX Runtime based Heterogeneous Execution on A72+C7x-MMA

3. TVM/Neo-AI-DLR based Heterogeneous Execution on A72+C7x-MMA

Due to a lack of support and documentation, the TVM/Neo-AI-DLR option was eliminated, leaving us with two great options. Both can operate well, and differences are not significant, but we have to choose only one. Based on recommendations from experienced automotive engineers, we have opted for ONNX Runtime, which we later found to be the more suitable choice for object detection cases.
ONNX Runtime for a DMS project makes sense given its prevalence within the BeagleBoard community [4] and the automotive industry. ONNX Runtime's flexibility, robust support, and optimizations for various hardware platforms make it a strong choice for deploying deep learning models. This alignment with community and industry standards provided better support and potentially more resources and tools for our deployment needs.

5.3 Model Selection [8]

After we have known the constaints of working with TI framework we have to recheck our model selection again to ensure if the it was compatible with the TDA4VM SoC or not, and unfortunately it was not.
When compiling YOLOv8 (the compilation processes will be explained in details in the next sub-section) with TIDL tools, we encountered an error related to the pooling layer configuration. Specifically, the error message in Fig. 42 states: `TIDL ALLOWLISTING LAYER CHECK - TIDL_PoolingLayer: kernel size 5x5 with stride 1x1 not supported.` This error indicates that the TIDL library does not support a pooling layer with a kernel size of 5x5 and a stride of 1x1. The pooling layer is common in convolutional neural networks (CNNs) used to reduce the spatial dimensions (width and height) of the input volume for the next layer, typically by downsampling the input. We found additional layers that are not supported by TI, such as the SiLU activation function. Fig. 43 shows the TI-supported layers from their official website.

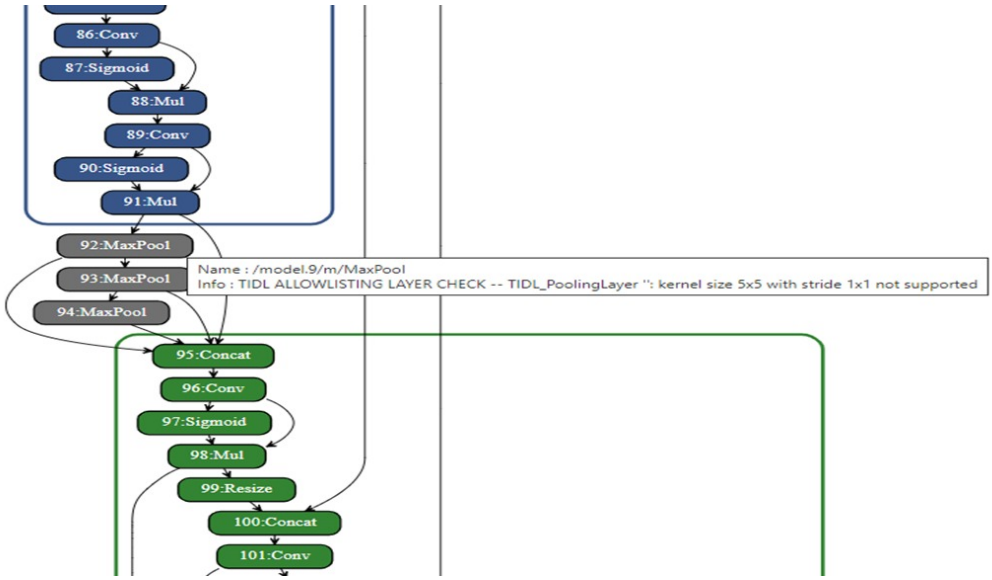


Figure 42: TI YOLOv8 compilation error

This makes YOLOv8 not supported. However, we found that YOLOv5 is officially supported by TI. They have a fork of the upstream repository with customizations for TIDL. This repository is probably most of the way toward compiling other kinds of models.

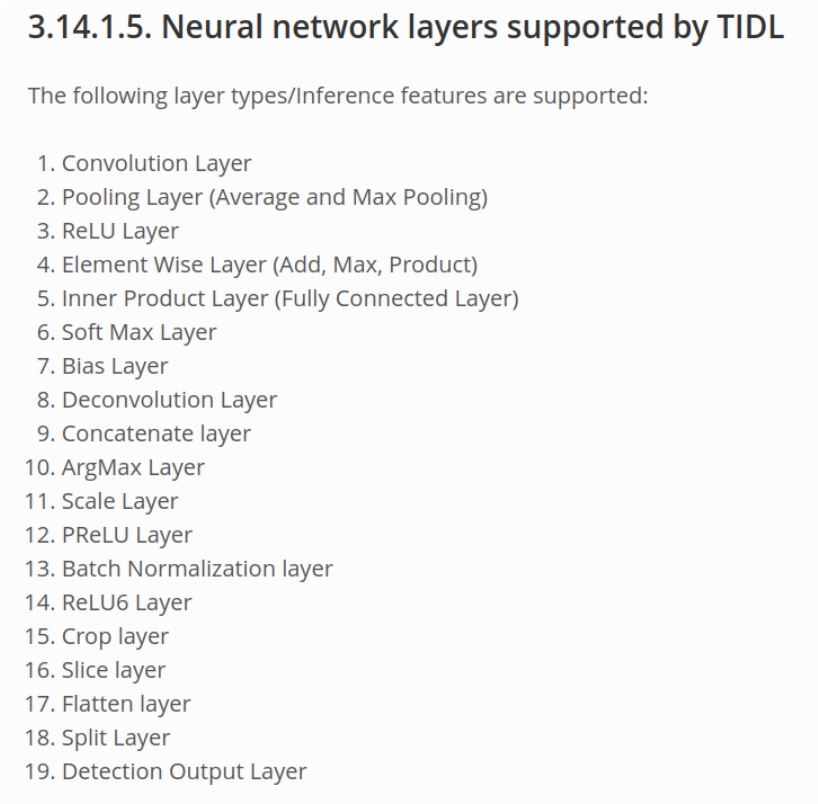


Figure 43: Supported layers by TIDL

5.3.1 Brief Description of Changes from YOLOv5 to YOLOv5-ti-lite [7] [32] [6]

- **Focus Layer Replacement:** In YOLOv5, the Focus layer is introduced as the initial layer, replacing several heavy convolution layers. This change reduces the network's complexity by 7% and decreases training time by 15%. However, the slice operations in the Focus layer are not suitable for embedded devices. To address this, the Focus layer has been replaced with a lightweight convolution layer.
- **Activation Function Adjustment:** The SiLU activation function used in YOLOv5 is not well-supported by embedded devices and is not quantization-friendly due to its unbounded nature. Similar issues were observed with the hSwish activation function when quantizing EfficientNet. Therefore, SiLU has been replaced with the ReLU activation function, which is more compatible with embedded applications. Fig. 46 shows a pictorial description of the changes from YOLOv5 to YOLOv5-ti-lite.

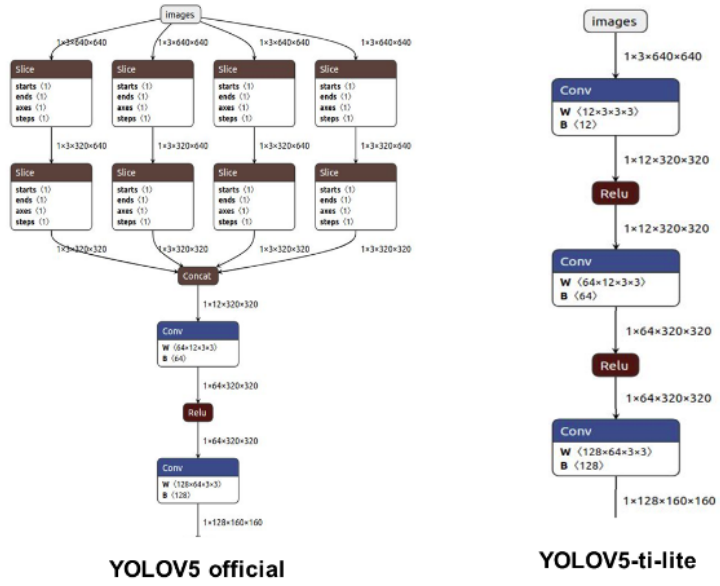


Figure 44: Changes from YOLOv5 to YOLOv5-ti-lite

- **SPP Module Modification:** YOLOv5's SPP (Spatial Pyramid Pooling) module originally uses max pooling layers with different kernel sizes: maxpool(k=13, s=1), maxpool(k=9, s=1), and maxpool(k=5, s=1). These have been replaced with combinations of maxpool(k=3, s=1) to retain the same receptive field and functionality while being more suitable for embedded devices. Specifically:
  - maxpool(k=5, s=1) is replaced with two maxpool(k=3, s=1)
  - maxpool(k=9, s=1) is replaced with four maxpool(k=3, s=1)
  - maxpool(k=13, s=1) is replaced with six maxpool(k=3, s=1) as shown below in Fig. 45 :

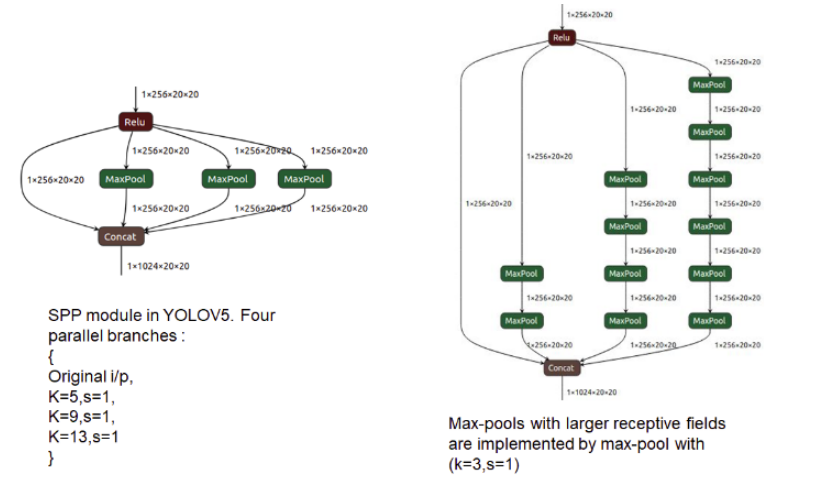


Figure 45: SPP module with maxpool changes

These modifications ensure that the model's performance remains unchanged in floating-point operations.

- **Inference Size Adjustment:** Variable size inference has been replaced with fixed size inference, which is preferred for edge devices. For instance, TensorFlow Lite (TFLite) models are exported with a fixed input size to better suit the constraints of edge devices.

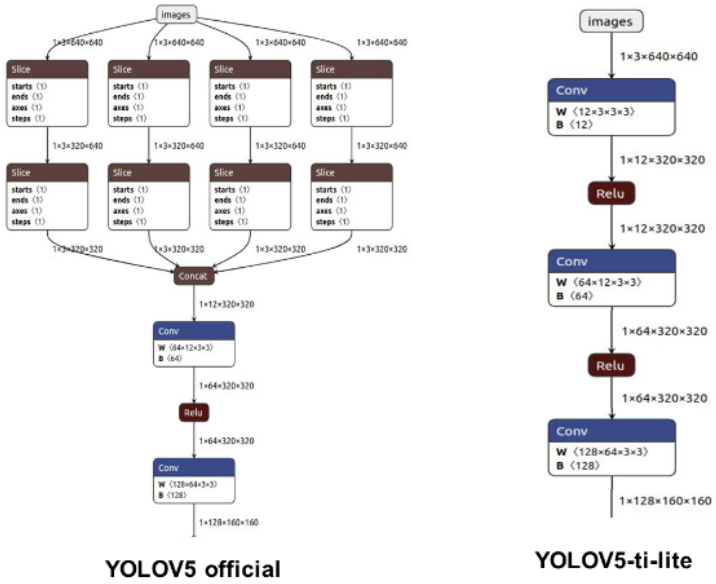


Figure 46: Changes from YOLOv5 to YOLOv5-ti-lite

Trained on the WIDERFACE dataset, we have our face detection YOLOv5-ti-lite model. Despite exploring other models such as YOLOX and various options provided by the TI model zoo, we found their accuracy insufficient for our system's requirements. YOLOv5-ti-lite outperformed these alternatives, meeting our accuracy and efficiency needs for embedded device applications.

5.4 Model Compilation

Having the model in the suitable runtime engine is not enough; we need another date to distribute the processing between cores. These generated data are the subgraph and node that TIDL uses to assign each core between DSPs and CPUs with a specific supported layer. All these data could be found within the model artifacts, and these artifacts are generated from the compilation process.

The compilation process takes two inputs: the .onnx model and the .prototxt (model definition file for each layer to validate if all layers are supported or not) and gives two outputs: the compiled model and the model deployment artifacts data we discussed.

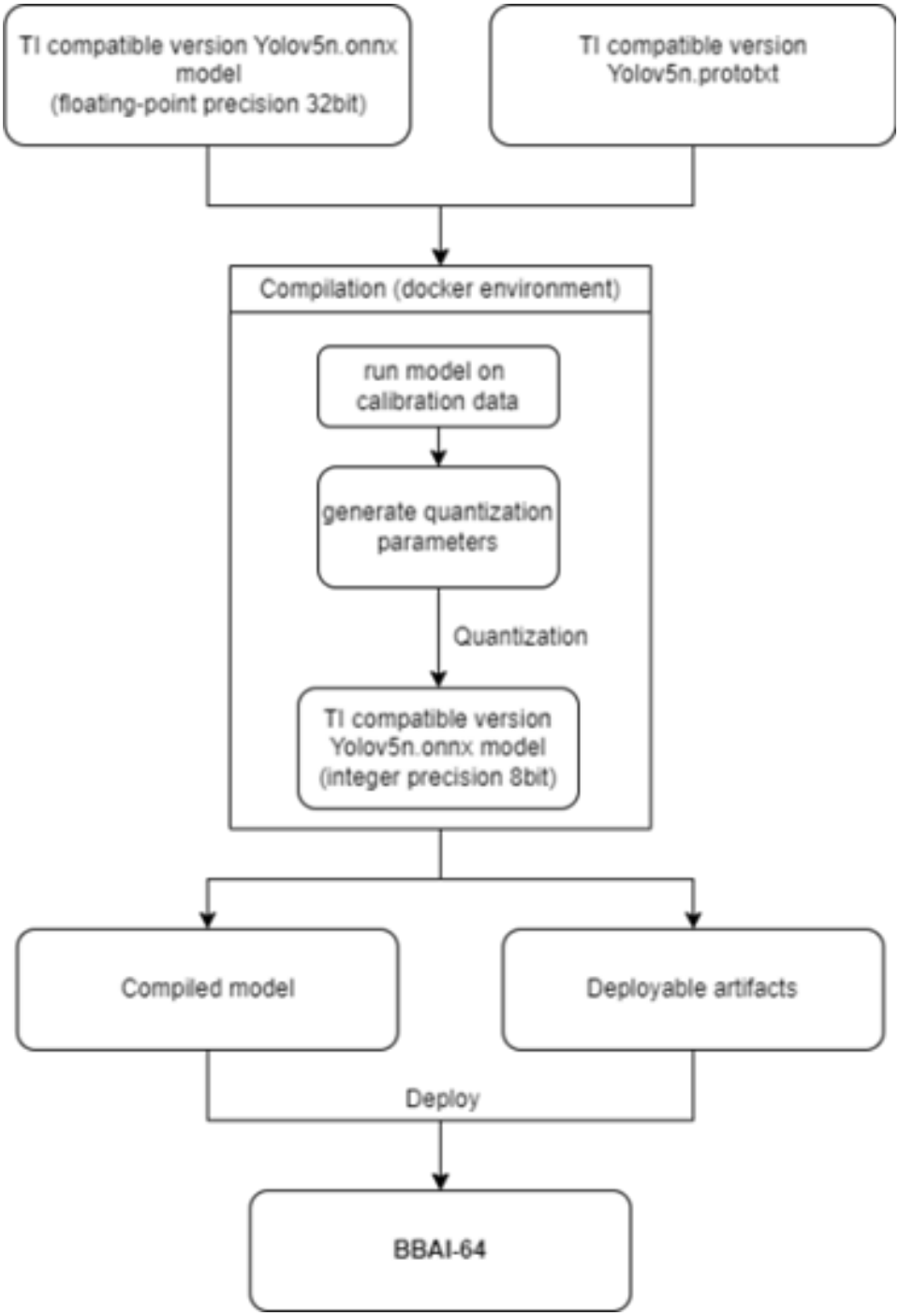


Figure 47: Model compilation process

5.4.1 Model Optimization (Quantization) [5]

Some challenges of the deployment on embedded target mentioned before were model size and hardware processing limitation. To overcome these challenges, we need some sort of model optimization for the embedded devices.

Quantization is a technique used to reduce the size and memory footprint of neural network models. It involves converting the weights and activations of a neural network from high-precision floating-point numbers to lower-precision formats, such as 16-bit or 8-bit integers. This can significantly reduce the model size and memory requirements, making it easier to deploy on edge devices with limited compute and memory resources.

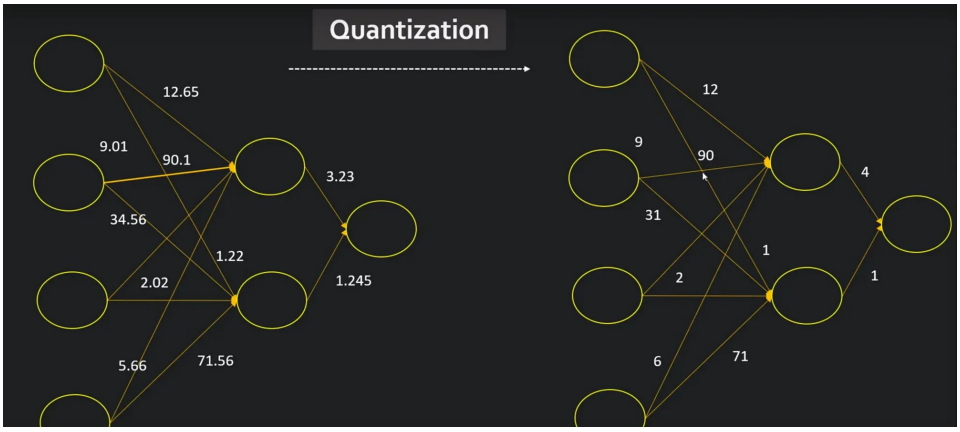


Figure 48: Quantization process

Post-Training Quantization (PTQ)

This approach involves quantizing weights and/or activations after the model has been trained using floating-point arithmetic. PTQ is typically applied as a separate step after model training, where the model parameters are converted into lower precision formats (e.g., 8-bit integers). It is effective for reducing model size and memory footprint, making models more suitable for deployment on resource-constrained edge devices. However, PTQ may lead to some loss of accuracy compared to the original floating-point model due to rounding errors and reduced precision.

Quantization-Aware Training (QAT)

QAT is a training technique where the model is trained with the awareness of eventual quantization to lower precision formats (e.g., 8-bit integers). During QAT, the training process includes simulated quantization effects to minimize the loss of accuracy that typically occurs when directly applying quantization post-training. By adjusting model parameters and optimization strategies, QAT aims to ensure that the quantized model maintains similar performance to its higher precision counterpart. QAT is more complex and computationally intensive than PTQ but often yields better accuracy retention for quantized models deployed on edge devices.

For choosing a more compatible version with the hardware we have gone with YoloV5 as mentioned before so we only apply the PTQ in our compilation process, but we have to be conscious with the quantization level we need as its size-accuracy trade off at some point.

Level	Input	Weights	Output	Description
0	float	float	float	No quantization (all data is FLOAT32)
1	float	int8	float	Quantization of model weights
3	float	int8	float	Quantization of weights and internal variables using a representative dataset. Input and output layers remain in FLOAT32
3	int8	int8	float	Quantization of input tensor uses the representative dataset
4	int8	int8	int8	Full integer conversion. All computation is intended to be done in embedded AI co-processor

Figure 49: Quantization Levels

5.4.2 Calibration

Calibration involves determining scaling factors or thresholds for quantized models to minimize the accuracy gap between the original floating-point model and the quantized version. This step is particularly important in scenarios where precision reduction (e.g., from 32-bit floating-point to 8-bit integer) may cause significant accuracy degradation.

Calibration typically involves running inference on a representative dataset, which may include images from the training data or a validation set. The goal is to gather statistics about the distribution of activations during inference.

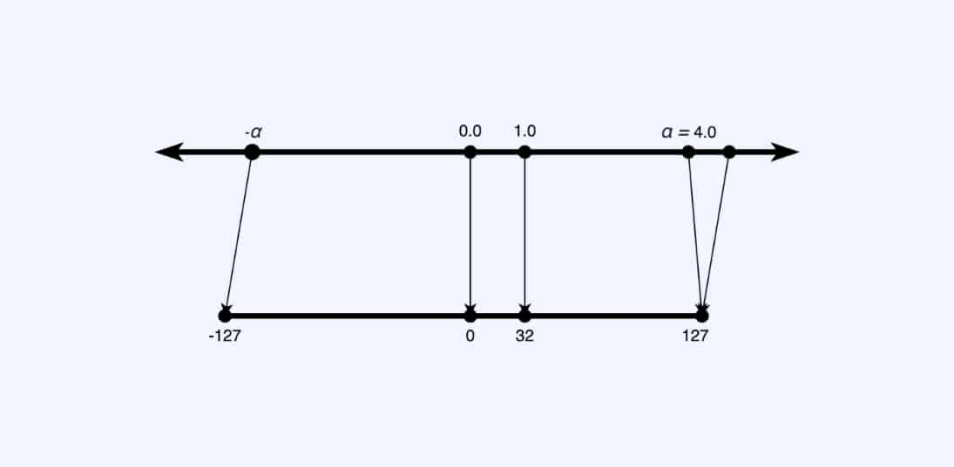


Figure 50: Calibration range

These statistics help determine the optimal range for input values (alpha), ensuring that the quantized model maintains accuracy. This process involves:

- Collecting Activation Ranges: During inference, the model processes the dataset, and the range of activation values (i.e., minimum and maximum values) is recorded for each layer.
- Determining Scale Factors: Once the activation ranges are collected, scale factors are calculated to map the floating-point values to the lower precision format (e.g., 8-bit integers).
- Optimizing Precision: The chosen α value and scale factors are fine-tuned to maximize precision, ensuring that the majority of activation values fall within the α-interval while minimizing the quantization error.

Using representative data during calibration is crucial because it ensures that the quantized model performs well on real-world data, maintaining a balance between reducing clipping errors and minimizing rounding errors.

5.4.3 Compilation Environment

Compiling deep learning models for the TDA4VM could be done in two ways:

1. Local on host machine with setting the suitable environment inside a docker container. We take this approach for compilation as the other option has limitations on the working period, which is limited to 3 hours a day, and we need to boost our productivity to iterate over the results. So this solution was more time-efficient.

2. Cloud compilation in EdgeAI studio using model analyzer and benchmark tools.

TI EdgeAI Studio [27] is a set of tools designed to streamline the development process for creating applications that use artificial intelligence (AI) on devices with limited processing power, also known as edge devices.

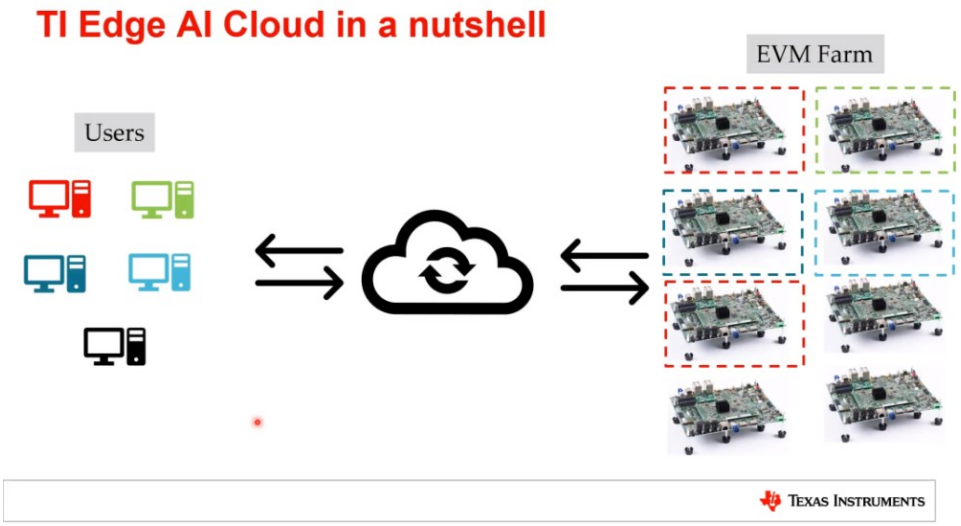


Figure 51: TI Edge AI cloud

Model Selection Tool The Model Selection Tool helps users find models that meet performance and accuracy goals from the TI Model Zoo, providing statistics such as FPS, latency, accuracy, and DDR bandwidth.

Model Analyzer: This is a free online service that lets you assess how well a deep learning model will perform on a particular TI development board. You can upload your model, and the service will run it on the board and provide you with performance metrics like frames per second (FPS), latency, and accuracy.

Model Maker Model Maker is a command-line tool for end-to-end model development, integrated with Model Composer and available for Linux® computers. It supports tasks like image classification, object detection, and semantic segmentation, and accepts annotated datasets for training and compilation.

Model Composer This is a graphical user interface (GUI) based tool that allows you to develop your entire AI application on your local machine. With Model Composer, you can collect and label data, train a model, and then optimize and compile it for deployment on a TI edge device. Model Composer even supports `Bring Your Own Data` (BYOD), which means you can re-train models from the TI Model Zoo with your own custom data to improve their performance for your specific use case.

We are more interested about mode analyzer as we can get model benchmarks and at the same time get the result of the compilation process (deployable artifacts).

First we select the SoC

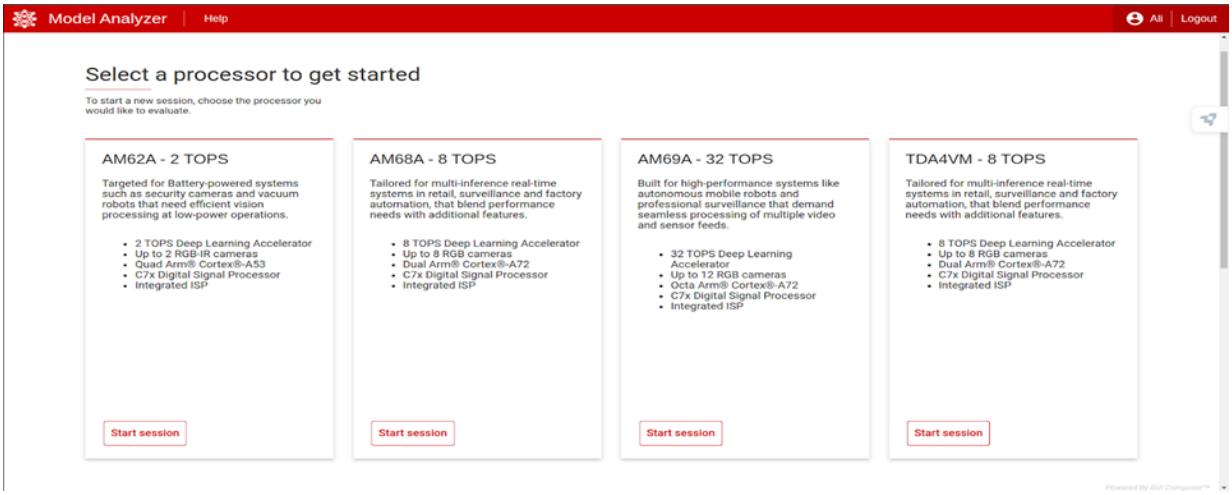


Figure 52: Model Analyzer interface

Second we chose the task to do benchmarks on a custom model.

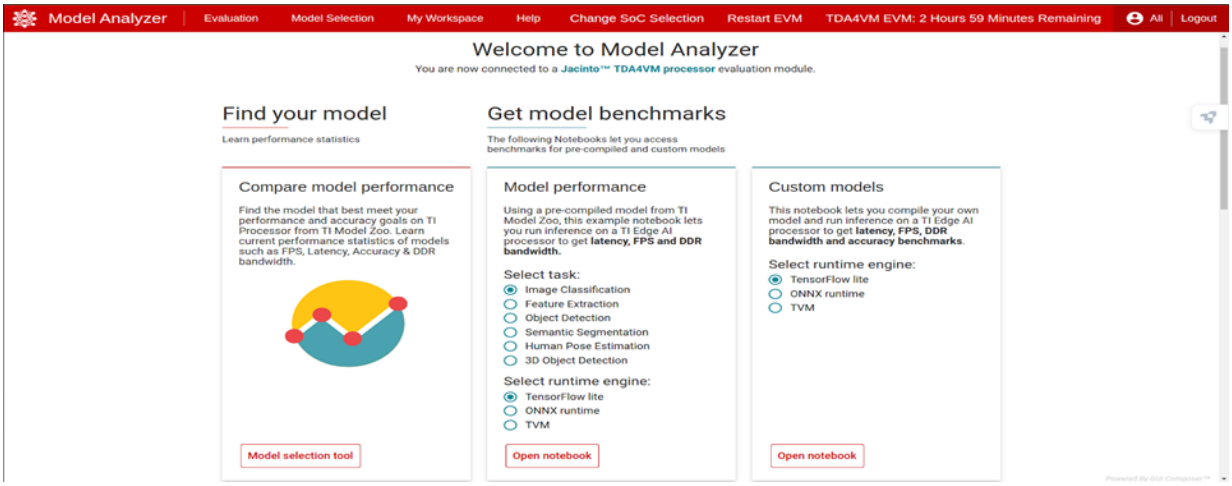


Figure 53: Model Analyzer interface

Third we start the time-limited jupyter note book session.

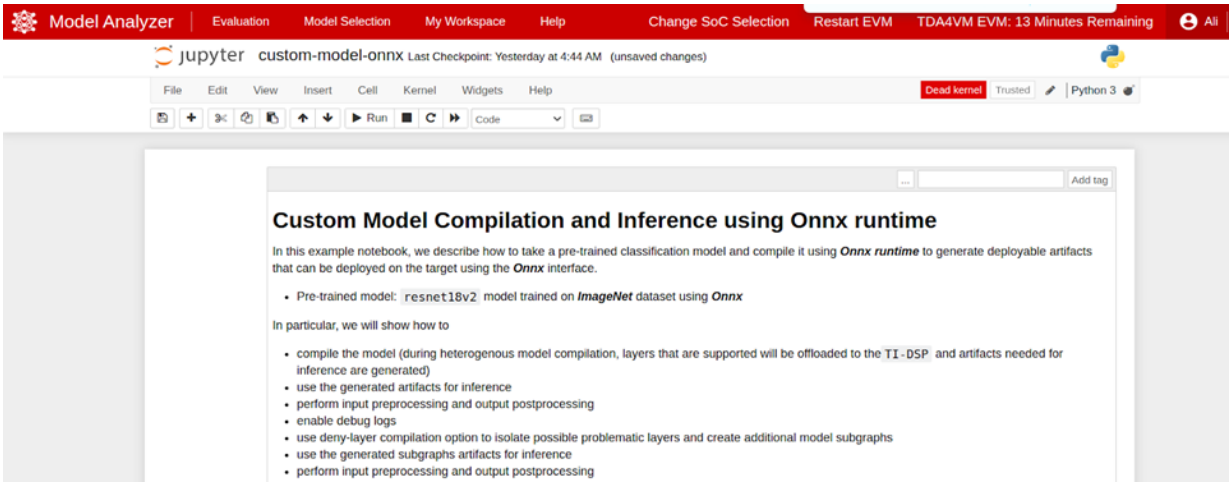


Figure 54: Model Analyzer interface

The output of this session are the deployable artifacts and performance analysis for the hardware usage.

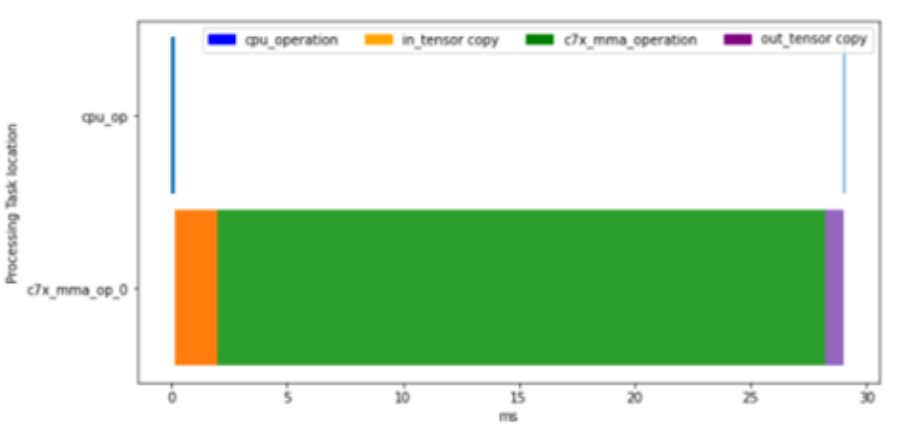


Figure 55: Model performance analysis for hardware usage

We used this approach also in model deployment for YoloV8n until we faced the unsupported layers challenge we mentioned before in 5.3, and in this case this approach is very useful as it can provide debugging information for the layers, the solution was known for us, its the model surgery APIs provided by TI but we were limited in time and we had to check for a substitution immidiatlly as the model surgry is a complicated process which may consume a full sprint, so we have gone with YoloV5 solution.

5.5 Deployment Process

Deploying AI models for real-world applications involves critical decisions influenced by the specific requirements of the deployment environment. For our DMS, the choice between edge and cloud deployment was carefully evaluated based on factors such as speed, privacy, offline functionality, and cost efficiency. Edge deployment emerged as the preferred solution due to its ability to meet real-time processing needs, ensure data privacy, operate offline, and reduce operational costs by offloading inference to embedded devices.

Specifically, leveraging Texas Instruments' TIDL framework and ONNX Runtime proved pivotal in optimizing our deployment strategy for the TDA4VM SoC. The decision to use ONNX Runtime was reinforced by its widespread adoption in the automotive industry and its robust support for heterogeneous execution across TI's hardware platforms. This alignment enabled efficient deployment of our YOLOv5-ti-lite model, tailored through techniques like quantization and layer customization to meet performance and accuracy requirements.

We can conclude the deployment journey with Fig 47 for YoloV5n.Onnx based work flow. We run the model compilation (Sub graphs creation and quantization) on PC and the generated artifacts can be deployed to running inference on the BBAI-64 with low latency and high accuracy results.

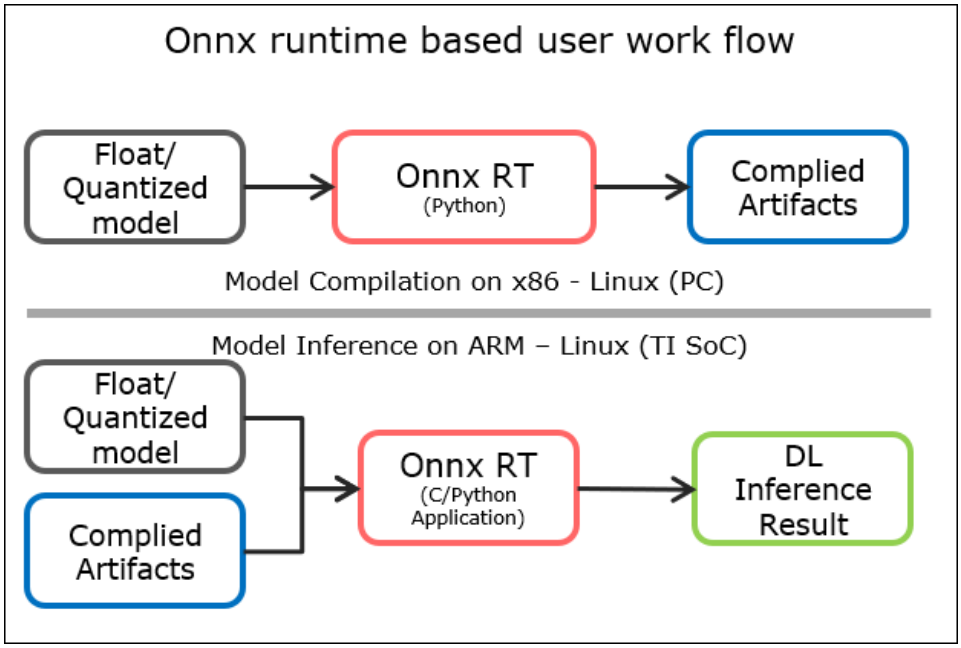


Figure 56: Model deployment workflow

Moving forward, continuous optimization efforts such as post-training quantization and calibration will be crucial for further enhancing model efficiency on embedded systems. By utilizing tools like TI EdgeAI Studio for model compilation and performance benchmarking, we ensured that our deployed models maintain high accuracy and low latency suitable for real-time driver monitoring applications on edge devices.

6 Benchmark

Benchmarking is a critical process in the development and evaluation of any system. It involves systematically measuring the performance of various application components under controlled conditions. This process helps to understand how efficiently the system operates, identify potential bottlenecks, and make informed decisions about optimizations and improvements.
In the DMS, benchmarking plays a pivotal role in ensuring that the system meets the necessary performance criteria for real-time monitoring and alerting. The primary objectives of benchmarking in a DMS include:

- Performance Evaluation: Assessing the speed and accuracy of different AI algorithms used for tasks such as driver behavior analysis, fatigue detection, and distraction recognition.
- Resource Management: Understanding the computational and memory requirements of the algorithms to ensure they can run efficiently on the target hardware, whether it's an in-car embedded system or a cloud-based server.
- Optimization: Identifying performance bottlenecks in the system and optimizing the code to improve overall efficiency and responsiveness.
- Comparison: Comparing the performance of various algorithms and configurations to determine the most suitable ones for the specific requirements of the DMS.
- Reliability: Ensuring that the DMS performs consistently and reliably under different conditions, such as varying lighting, driving environments, and driver behaviors.

6.1 Key Aspects of Benchmark

Before discussing the benchmarks conducted on our system, it's important to introduce some key aspects of benchmarking.The following are the primary aspects of benchmarking code and their significance When Benchmark our DMS:

Measurement and Metrics

This aspect involves defining what you want to measure and selecting the appropriate performance metrics. For our DMS, the key metrics include:

- Execution Time: The time taken for algorithms to process each frame or batch of frames. This is crucial for ensuring that the system can operate in real time.
- Memory Usage: The amount of memory consumed by the algorithms during execution. Efficient memory usage is vital for running the DMS on embedded platforms like the TDA4VM.
- Throughput: The number of frames processed per second. This metric helps assess the system's ability to handle high frame rates from the camera.
- Latency: The delay between capturing a frame and obtaining the analysis results. Low latency is essential for prompt driver alerts.
- Resource Utilization: The utilization of CPU, GPU, and DSP resources on the TDA4VM. Balanced utilization ensures that no single component becomes a bottleneck.

Selecting the relevant metrics is crucial for our specific use case. For instance, execution time and latency are critical for real-time performance, while memory usage and resource utilization are important for system stability and efficiency.

Controlled Testing Conditions

To obtain reliable and consistent benchmark results, we need to establish controlled testing conditions:

- Isolating Code: Ensuring that the code being benchmarked is isolated from other system processes that could affect performance measurements.
- Minimizing Interference: Reducing the influence of background processes and external factors. This can be achieved by running benchmarks on a dedicated hardware setup with minimal external interference.
- Stable System State: Ensuring that the system is in a stable state during testing. This includes having a consistent hardware environment and using a consistent software stack.
- Reproducibility: Ensuring that the same benchmark yields consistent results when run multiple times. This involves using fixed input data, controlled environmental conditions, and consistent benchmarking tools.

Interpretation and Analysis

Benchmarking is not just about collecting data; it also involves interpreting and analyzing the results:

- Comparing Results: Comparing benchmark results across different versions of the code or different configurations of the system. This helps identify which changes lead to performance improvements or regressions.
- Identifying Bottlenecks: Using benchmarking data to pinpoint performance bottlenecks. For example, if the execution time of an algorithm is high, analyzing which part of the code consumes the most time can guide optimization efforts.
- Informed Decisions: Making informed decisions about optimizations or resource allocation based on benchmark data. This could involve optimizing code, choosing more efficient algorithms, or adjusting hardware configurations.
- Effective Analysis: Drawing meaningful insights from the benchmark data. This involves using statistical methods to analyze results, visualizing data to identify trends, and correlating metrics to understand their impact on overall system performance.

Effective analysis is essential for optimizing our DMS. For instance, if benchmarking reveals that the eye closure detection algorithm has high latency, we might explore alternative methods or hardware acceleration options on the TDA4VM to improve performance.

6.2 Measuring Execution Time

As a metric, we decided to measure the execution time because it has a big effect on evaluating the performance of our system algorithms. Execution time directly affects the responsiveness of the system, which is crucial for timely detection of driver distractions and prompt alerts. This section delves into the importance of measuring execution time and the methodology we use.

Execution time refers to the duration it takes for an algorithm or a piece of code to complete its task. For our DMS, which includes algorithms like head pose estimation, eye closure detection, and yawn detection, execution time measurement is essential for several reasons:

- Real-time Performance: The DMS must process video frames in real time to detect driver distractions promptly. Long execution times can lead to delays, reducing the effectiveness of the system.
- Optimization: By measuring execution time, we can identify performance bottlenecks and optimize our algorithms to run more efficiently.
- Resource Allocation: Understanding the execution time helps in balancing the workload across the available hardware resources, such as the CPU, GPU, and DSPs on the TDA4VM.
- System Scalability: Measuring execution time allows us to assess how well the system can scale with increased load, such as higher frame rates or additional algorithms.

Methodology for Measuring Execution Time

To accurately measure execution time, we considered the following aspects:

- Tool Selection: We use Python's `time.perf\_counter()` function for high-resolution timing. This function provides a non-adjustable and Monotonic clock. Also, it provides a precise measurement of time intervals, making it suitable for benchmarking.
- Consistent Input: To ensure reproducibility, we use the same input data for each benchmark run. This involves using pre-recorded video frames or synthetic data that simulate typical driving scenarios.
- Multiple Runs: We conduct multiple runs of the benchmark and calculate the average execution time to account for variability and ensure consistent results, while carefully considering the sample size. The number of benchmark runs impacts result reliability; a small sample size may not accurately reflect code performance, whereas a large sample size can enhance the accuracy of the results.
- Benchmarking Tools: We use additional tools, such as cProfile, to provide more detailed insights into the execution time of individual code segments.
- Isolated Environment: We run benchmarks on a dedicated hardware setup, such as a dedicated host machine or our hardware platform (BeagleBone AI-64 evaluation board). It's important to note that results from each machine are compared separately and cannot be directly compared with each other.

6.3 Execution Time Benchmark of AI Algorithms in SIL Environment

In the Software-in-the-Loop (SIL) environment, we established a sequential baseline of our application with all algorithms as a starting point for benchmarking. We applied execution time benchmarks to the AI algorithms to identify their processing loads, allowing us to compare them relative to each other and determine which algorithms are the most resource-intensive. This process provided insights into possible optimizations in the algorithmic code, and potential modifications to AI models, and helped identify bottlenecks to evaluate overall system performance.

For the benchmarking environment, all code, including the AI models, was executed on the same CPU of

the development machine. This setup was chosen to ensure that performance comparisons across different code segments were consistent and reliable.
We used `time.perf_counter()` for measuring execution time due to its high precision and its characteristics of being non-adjustable and monotonic, ensuring accurate and continuous time measurements.

The algorithms benchmarked include:

- Face Detection using a YOLO deep-learning model.
- Landmarks Extraction using the dlib machine learning model.
- Age and Gender Estimation using deep-learning models.
- Eye Closure Detection and Yawn Detection, which involve various computations using NumPy.
- Head Pose Estimation, also based on NumPy computations.

The benchmark results indicated the following order of processing load, from heaviest to lightest:

1. Face Detection (YOLO model)
2. Age Estimation model
3. Gender Estimation model
4. Landmarks Extraction (dlib model)
5. Head Pose Estimation
6. Eye Closure Detection
7. Yawn Detection

The Eye Closure Detection and Yawn Detection algorithms had nearly equal execution times, and their combined times approximately equaled the time taken by the Head Pose Estimation algorithm.
The results of these benchmarks are depicted in the figures 57,58,59 and60 below. These figures display the logs of the execution time for models or algorithms per frame, with multiple runs conducted and averaged to obtain more accurate results.
Based on these benchmarks, we derived several insights and optimization recommendations:

Age and Gender Models Optimization

Run the age and gender models only initially to estimate the driver's age and gender, then stop them. Since the driver remains the same throughout the journey, continuously running these models is unnecessary and burdensome for the system.

YOLO Model Optimization

Given that the YOLO model is the heaviest, it should be optimized. One approach is to adjust the quantization level of the model computations to approximate integer operations as closely as possible. Additionally, the YOLO model should be run on a powerful core like the C71 DSP. The C71 DSP is tightly coupled with the Matrix Multiplication Accelerator (MMA), which can significantly speed up computations for the model.

```
2024-03-24 06:38:16,837 - Yolo_logger - INFO - FRAME Number:2
------------------------------------------------------------
2024-03-24 06:38:16,941 - Yolo_logger - INFO - YOLO_SingleRun    |97.05673 msec
2024-03-24 06:38:17,010 - Yolo_logger - INFO - YOLO_SingleRun    |68.20645 msec
2024-03-24 06:38:17,092 - Yolo_logger - INFO - YOLO_SingleRun    |81.54894 msec
2024-03-24 06:38:17,092 - Yolo_logger - INFO - YOLO_AverageTime  |82.27071 msec

2024-03-24 06:38:17,200 - Yolo_logger - INFO - FRAME Number:3
------------------------------------------------------------
2024-03-24 06:38:17,439 - Yolo_logger - INFO - YOLO_SingleRun    |230.39742 msec
2024-03-24 06:38:17,528 - Yolo_logger - INFO - YOLO_SingleRun    |89.22814 msec
2024-03-24 06:38:17,637 - Yolo_logger - INFO - YOLO_SingleRun    |108.06193 msec
2024-03-24 06:38:17,637 - Yolo_logger - INFO - YOLO_AverageTime  |142.56250 msec

2024-03-24 06:38:17,738 - Yolo_logger - INFO - FRAME Number:4
------------------------------------------------------------
2024-03-24 06:38:17,837 - Yolo_logger - INFO - YOLO_SingleRun    |90.90465 msec
2024-03-24 06:38:17,914 - Yolo_logger - INFO - YOLO_SingleRun    |77.35590 msec
2024-03-24 06:38:17,997 - Yolo_logger - INFO - YOLO_SingleRun    |82.63441 msec
2024-03-24 06:38:17,997 - Yolo_logger - INFO - YOLO_AverageTime  |83.63165 msec

2024-03-24 06:38:18,096 - Yolo_logger - INFO - FRAME Number:5
------------------------------------------------------------
2024-03-24 06:38:18,247 - Yolo_logger - INFO - YOLO_SingleRun    |143.26445 msec
2024-03-24 06:38:18,330 - Yolo_logger - INFO - YOLO_SingleRun    |82.17698 msec
2024-03-24 06:38:18,409 - Yolo_logger - INFO - YOLO_SingleRun    |79.59295 msec
2024-03-24 06:38:18,410 - Yolo_logger - INFO - YOLO_AverageTime  |101.67812 msec
```

Figure 57: YOLO deep-learning model benchmark

```
2024-03-24 06:38:16,837 - dlib_logger - INFO - FRAME Number:2
------------------------------------------------------------
2024-03-24 06:38:17,094 - dlib_logger - INFO - MARKS_SingleRun   |2.07494 msec
2024-03-24 06:38:17,096 - dlib_logger - INFO - MARKS_SingleRun   |2.35439 msec
2024-03-24 06:38:17,099 - dlib_logger - INFO - MARKS_SingleRun   |2.18749 msec
2024-03-24 06:38:17,099 - dlib_logger - INFO - MARKS_AverageTime |2.20561 musec

2024-03-24 06:38:17,200 - dlib_logger - INFO - FRAME Number:3
------------------------------------------------------------
2024-03-24 06:38:17,640 - dlib_logger - INFO - MARKS_SingleRun   |2.44468 msec
2024-03-24 06:38:17,643 - dlib_logger - INFO - MARKS_SingleRun   |2.62999 msec
2024-03-24 06:38:17,645 - dlib_logger - INFO - MARKS_SingleRun   |2.01210 msec
2024-03-24 06:38:17,645 - dlib_logger - INFO - MARKS_AverageTime |2.36226 musec

2024-03-24 06:38:17,738 - dlib_logger - INFO - FRAME Number:4
------------------------------------------------------------
2024-03-24 06:38:18,000 - dlib_logger - INFO - MARKS_SingleRun   |2.30548 msec
2024-03-24 06:38:18,004 - dlib_logger - INFO - MARKS_SingleRun   |3.28291 msec
2024-03-24 06:38:18,006 - dlib_logger - INFO - MARKS_SingleRun   |2.17052 msec
2024-03-24 06:38:18,006 - dlib_logger - INFO - MARKS_AverageTime |2.58630 musec

2024-03-24 06:38:18,096 - dlib_logger - INFO - FRAME Number:5
------------------------------------------------------------
2024-03-24 06:38:18,412 - dlib_logger - INFO - MARKS_SingleRun   |2.12135 msec
2024-03-24 06:38:18,414 - dlib_logger - INFO - MARKS_SingleRun   |2.14670 msec
2024-03-24 06:38:18,417 - dlib_logger - INFO - MARKS_SingleRun   |2.56313 msec
2024-03-24 06:38:18,418 - dlib_logger - INFO - MARKS_AverageTime |2.27706 musec
```

Figure 58: DLib landmarks extraction model benchmark

Hints for Multiprocessing

For each frame, we need the values of eye closure, yawn detection, head pose estimation, and gaze estimation. Some of these features can be parallelized, while others can be executed sequentially. For example, since the combined times of eye closure and yawn detection almost equal the head pose estimation time, they can run sequentially on one process while running head pose estimation in parallel on another process.
These optimizations and strategies aim to enhance the efficiency and performance of our driver monitoring system, ensuring real-time responsiveness and accurate monitoring of the driver's state.

```
2024-03-24 06:38:16,837 - features_logger - INFO - FRAME Number:2
------------------------------------------------------------
2024-03-24 06:38:17,187 - features_logger - INFO -
EYE_TIME           |138.48600 usec
YAWN_TIME          |416.52400 usec
HEAD_TIME          |911.03900 usec

2024-03-24 06:38:17,200 - features_logger - INFO - FRAME Number:3
------------------------------------------------------------
2024-03-24 06:38:17,725 - features_logger - INFO -
EYE_TIME           |117.94800 usec
YAWN_TIME          |134.34900 usec
HEAD_TIME          |335.38800 usec

2024-03-24 06:38:17,738 - features_logger - INFO - FRAME Number:4
------------------------------------------------------------
2024-03-24 06:38:18,083 - features_logger - INFO -
EYE_TIME           |96.66300 usec
YAWN_TIME          |95.56800 usec
HEAD_TIME          |672.59300 usec

2024-03-24 06:38:18,096 - features_logger - INFO - FRAME Number:5
------------------------------------------------------------
2024-03-24 06:38:18,501 - features_logger - INFO -
EYE_TIME           |114.25300 usec
YAWN_TIME          |108.61700 usec
HEAD_TIME          |417.33300 usec
```

Figure 59: Eye closure, yawn detection, and head pose estimation benchmarks

```
2024-03-24 06:38:16,837 - Age_Gender_logger - INFO - FRAME Number:2
------------------------------------------------------------
2024-03-24 06:38:17,187 - Age_Gender_logger - INFO -
AGE_GENDER_TIME            |79.27076 msec
AGE_TIME                   |59.60900 msec
GENDER_TIME                |15.52900 msec

2024-03-24 06:38:17,200 - Age_Gender_logger - INFO - FRAME Number:3
------------------------------------------------------------
2024-03-24 06:38:17,726 - Age_Gender_logger - INFO -
AGE_GENDER_TIME            |71.63582 msec
AGE_TIME                   |55.10000 msec
GENDER_TIME                |13.35300 msec

2024-03-24 06:38:17,738 - Age_Gender_logger - INFO - FRAME Number:4
------------------------------------------------------------
2024-03-24 06:38:18,084 - Age_Gender_logger - INFO -
AGE_GENDER_TIME            |69.65363 msec
AGE_TIME                   |54.02100 msec
GENDER_TIME                |13.12900 msec

2024-03-24 06:38:18,096 - Age_Gender_logger - INFO - FRAME Number:5
------------------------------------------------------------
2024-03-24 06:38:18,502 - Age_Gender_logger - INFO -
AGE_GENDER_TIME            |74.68881 msec
AGE_TIME                   |57.08300 msec
GENDER_TIME                |15.57300 msec
```

Figure 60: Age and gender models benchmark

6.4 Execution Time Benchmark on Hardware

6.4.1 Time Analysis

After deploying the application on the BBAI-64, the benchmarks we performed before in the SiL environment will change as we have different processing capabilities, model architecture, and new stages for the hardware

assignment with the deployed application.
As the first run after deployment started with less than one fps, it becomes necessary to analyze the application from A to Z.
We have three main tasks to analyze divided based on the used hardware: pre-processing and post-processing on C66x DSP, and the inference on the C7x DSP.

6.4.2 FPS Optimization

YoloXs-Based Application Stages

1. Using TI Default object detection post-processing demos, which can run on 20 to 22 fps. These values will be the benchmark to know if we reached a high achievable rate on BBAI-64 or not. Our developed face detection post-processing represents the first trial to integrate our app within EdgeAi application.
2. FPS measurements problems, in this case the FPS varied a lot from 5 to 20 fps. After that, we found the problem was with the analysis approach as the used printing function was time-consuming due to changing the standard output to a log file, as the terminal was being occupied with printing the model results and hardware utilization thread.
3. Problem solved for our developed face detection post-processing, and we found that we have a satisfying fps (18 to 20), which becomes an on-the-shelf solution from the fps aspect.

YoloXs-Based Application Time Analysis	
Task	**Average Time (ms)**
Defualt Object Detection Postprocessing FPS: 20.5 ~ 22.5	
Capture and pre-processing	2.4
Infernece	13.1
Post-processing	29.1
Total	**45.4**
Custom Face Detection Postprocessing FPS: 5 ~ 20 (Unstability Problem)	
Capture and pre-processing	2.4
Infernece	14.1
Post-processing	33.5
Total	**51.8**
Custom Face Detection Postprocessing FPS: 18 ~ 20 (Sloved)	
Capture and pre-processing	2.4
Infernece	14.1
Post-processing	33.5
Total	**51.8**

Table 4: FPS Optimization for YoloXs-Based Application

Note: In case of providing a range of FPS (like 5 to 20 fps) the total time is calculated based on the upper bound (20 fps).

YoloV5-Based Application Stages

We can consider all of the first three stages (less overlays, not considering writing the frame as a video, and processing only one face) as some kind of code refactoring. We just reduced the unnecessary printed results and stopped writing the frame locally - only if it is required to review some results - which was found to be a very time-consuming task, taking around half of the time. After that, we applied some logical changes to the processed frame; we only process one detected box/face, neglecting the rest of the faces, as this simulates our case for processing the driver face only, neglecting the rest of the faces inside the car. Now we can reach a range of 17 to 18 fps max.

After that, we applied our processing on different frame sizes, which was found to change the fps a lot (from 5 to 22). As the frame size decreases, the fps increases, but this solution will not always be valid as we have a dedicated camera with a fixed frame size, so we have to look for another mechanism to optimize the fps more and make it controllable.

We reached an idea for this control. Thinking about every processed frame, we noticed that many frames do not differ a lot from the next window of upcoming frames until the driver performs some noticeable movement, which will be translated into a difference between two frames that exceeds a predefined threshold. This is the frame bypassing mechanism which is illustrated in the flowchart provided in Figure 61.

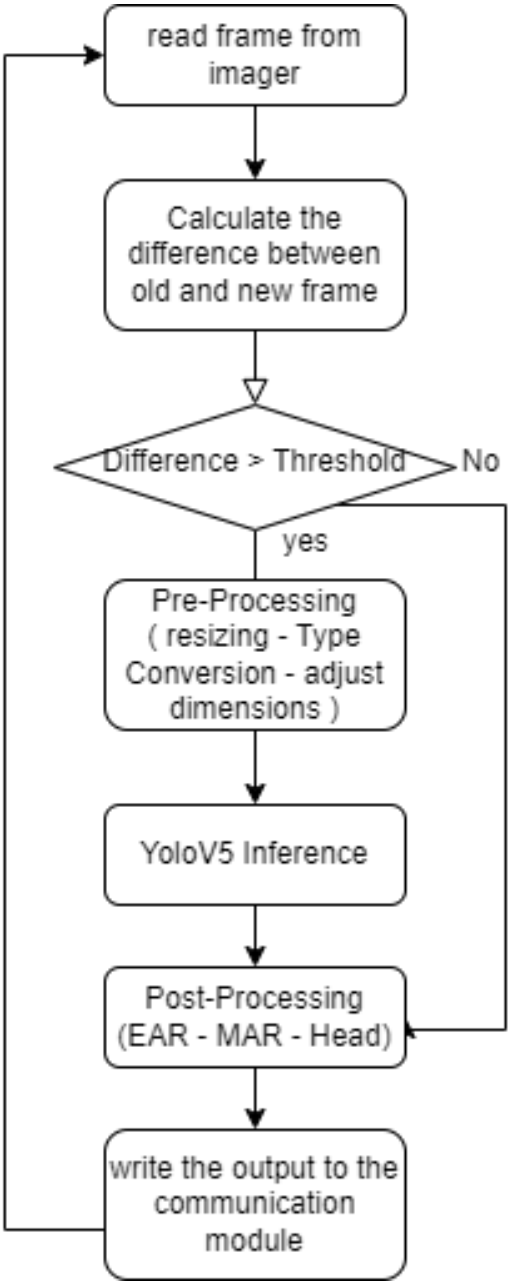


Figure 61: Frame Bypassing Mechanism

Now, we can change the fps as much as we need, reaching 23 fps, but what did it cost? For sure, we sacrificed the overall application accuracy. The following Figure 62 illustrates the percentage of bypassed frames with the fps level.

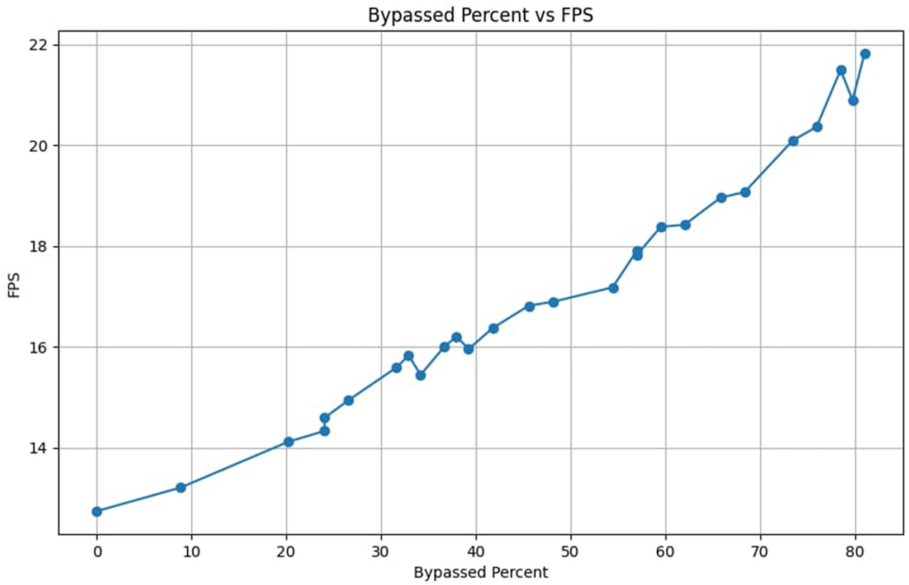


Figure 62: Percentage of Bypassed Frames with FPS Level

More improvement in the fps but the cost is high, so we have to optimize the bypassing mechanism itself more, and we came up with two ways for that. First, we changed the algorithm that calculates the difference between frames. At first, we relied on the sum of square differences (SSD), but we found that using the sum of absolute differences (SAD) is more computationally efficient for the CPU resources, as the squaring operation takes more execution time than the absolute value calculations.

$$\text{SAD} = \sum |da - d_o|$$

$$\text{SSD} = \sum (da - d_o)^2$$

Secondly, as we noticed that we do not need the color information of the frames in this comparison, we considered converting the frames in the comparison stage to grayscale images to have less data to process than the RGB images, which provided a faster bypassing algorithm leading to a satisfying fps that can reach 19 fps with 50 percent bypassing and satisfying accuracy for the eye closure, yawn detection, and head pose estimation tasks.

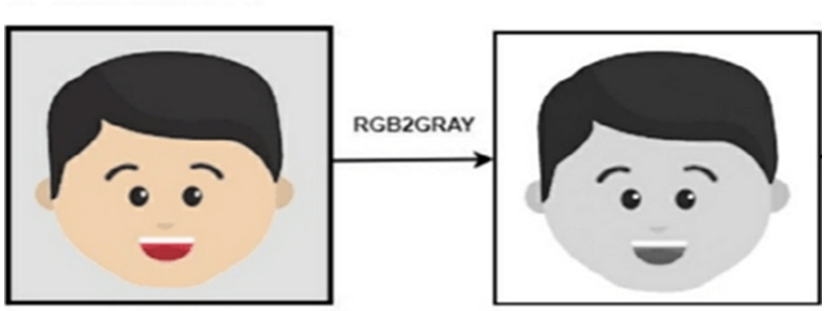


Figure 63: RGB to Grayscale

YoloV5-Based Application Time Analysis			
Task		**Average Time (ms)**	
Less Overlayes FPS: 9 ~ 10.5			
Capture and pre-processing	Reading next frame	5.4	30.13
	Pre-processing: Resizing	21.3	
	Pre-processing: Expand Dimensions	0.23	
	Pre-processing: Type Conversion	3.2	
Infernece	Infernece	14.3	14.3
Post-processing	Rendering box	0.5	53.8
	Post-processing	7.6	
	Writing frame	45.7	
Total		**98.23**	
Less overlayes + Not cosidring writing the frame as a video FPS: 15 ~ 17			
Capture and pre-processing	Reading next frame	5.4	30.13
	Pre-processing: Resizing	21.3	
	Pre-processing: Expand Dimensions	0.23	
	Pre-processing: Type Conversion	3.2	
Infernece	Infernece	14.3	14.3
Post-processing	Rendering box	0.5	9.1
	Post-processing	8.6	
Total		**53.53**	
Less overlayes + Not cosidring writing the frame as a video process only one face per frame FPS: 17 ~ 18			
Capture and pre-processing	Reading next frame	5.4	29.13
	Pre-processing: Resizing	21.3	
	Pre-processing: Expand Dimensions	0.23	
	Pre-processing: Type Conversion	2.2	
Infernece	Infernece	14.3	14.3
Post-processing	Rendering box	0.5	8.1
	Post-processing	7.6	
Total		**51.53**	
Vary in Image Resolution and Dimentions + Less Overlayes + No Write + One Face FPS: 5 ~ 22			
Capture and pre-processing	Reading next frame	27.4	53.73
	Pre-processing: Resizing	22.9	
	Pre-processing: Expand Dimensions	0.23	
	Pre-processing: Type Conversion	3.2	
Infernece	Infernece	14.3	14.3
Post-processing	Rendering box	0.5	8 ~ 110
	Post-processing	7.6	
	Writing frame	3 ~ 100	
Total		**45 ~ 185**	
Frame Bypassing + Less Overlayes + No Write + One Face FPS: 17 ~ 20 (Based on Threashold)			
Capture and pre-processing	Reading next frame	1.4	24.13
	Pre-processing: Resizing	19.3	
	Pre-processing: Expand Dimensions	0.23	
	Pre-processing: Type Conversion	3.2	
Infernece	Infernece	13.3	14.3
Post-processing	Rendering box	0.5	9.8
	Post-processing	5.6	
	Writing frame	3.7	
Total		**48.23**	

Table 5: FPS Optimization for YoloV5-Based Application

Note: In case of providing a range of FPS (like 17 to 20 fps) the total time is calculated based on the upper bound (20 fps).

7 Communication Module

7.1 Interface to the VCU

The DMS module is connected to the car system via a Vehicular Interface Unit (VCU), typically using a CAN bus for communication. However, since we don't have a VCU for communication, we opted to use UART instead for simplicity. UART (Universal Asynchronous Receiver/Transmitter) is simple to implement and requires minimal hardware and software resources, making it ideal for straightforward communication tasks. UART is well-suited for point-to-point communication between two devices, meaning that we can communicate with the DMS module just by using a serial cable.

The output of the DMS is hashed just before forwarding it over UART. The data is arranged in key-value pairs (dictionary, or hashmap) where each key corresponds to a Key Performance Indicator (KPI), drowsiness factor, for instance. Hash functions are algorithms that take an input (or message) and return a fixed-size string of bytes, typically a digest that appears random. Before sending the message over UART, the DMS module computes the hash of the message data. This hash (also known as a checksum or digest) is sent along with the original data over UART. The receiver (laptop), upon receiving the data and hash, recomputes the hash on the received data and compares it with the received hash. If they match, the data is considered intact; otherwise, it indicates possible data corruption. These techniques are also used for authentication between the DMS module and the VCU (the laptop in our case). This is achieved by blocking the DMS app until the start signal sent by the VCU is successfully received.

7.2 Application Performance Monitoring

Application performance monitoring (APM) involves a set of tools and processes aimed at helping IT professionals ensure that enterprise applications deliver the required performance, reliability, and user experience (UX) for employees, partners, and customers. APM is part of the broader concept known as application performance management. While APM specifically tracks the performance of an application, application performance management is concerned with overseeing and maintaining optimal performance levels of an application. Essentially, monitoring is a subset of management.
A robust APM platform should focus on several key areas:

- **Infrastructure Monitoring**: Observing the health and performance of the underlying infrastructure.
- **User Experience Monitoring**: Evaluating and analyzing the end-user experience.
- **Dependency Monitoring**: Checking the performance and reliability of dependent systems.
- **Business Transactions Monitoring**: Observing the performance of critical business transactions.

APM tools equip administrators with the necessary data to swiftly detect, isolate, and resolve issues that could negatively impact an application's performance. IT professionals utilize performance metrics collected by APM tools from one or multiple applications within the same network to pinpoint the root cause of any issues.
The data gathered by APM tools includes:

- Client CPU utilization
- Memory usage

- Data throughput
- Bandwidth consumption

This detailed data collection helps IT professionals maintain optimal application performance and ensure a positive user experience.

7.2.1 Purpose of Application Performance Monitoring (APM)

The primary purpose of APM is to ensure the continuous availability and optimal performance of applications, which is crucial for maintaining uninterrupted business processes. This helps prevent unnecessary disruptions and enhances customer satisfaction. An effective APM platform allows organizations to link application performance to business outcomes, isolate and resolve errors before they impact the end user, and reduce the mean time to repair (MTTR).

7.2.2 Drivers for Purchasing APM Tools

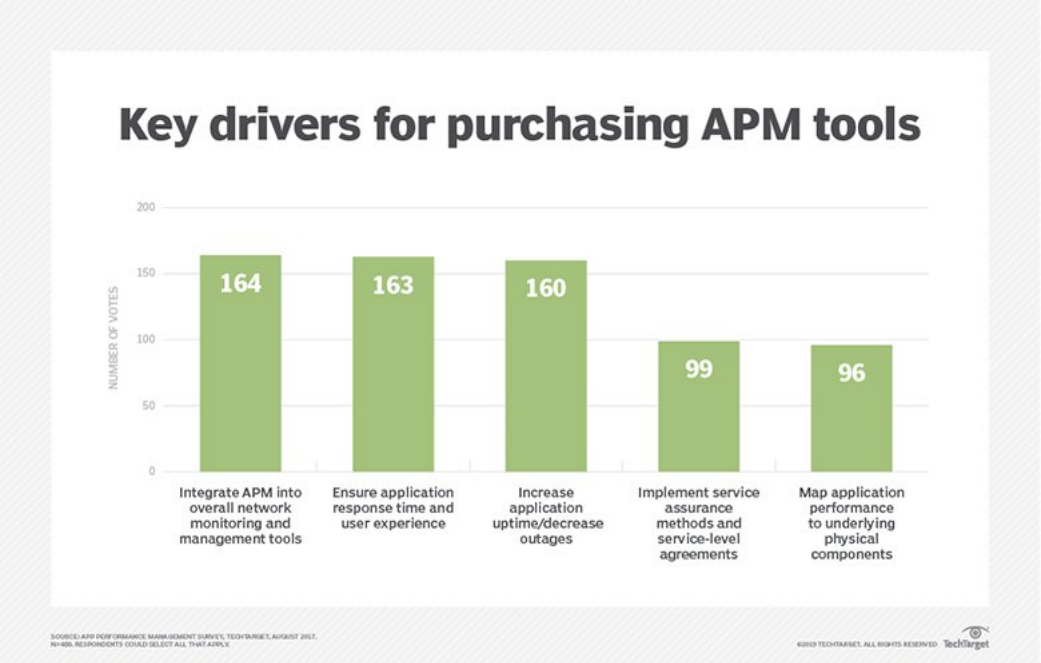


Figure 64: Key drivers for purchasing APM tools

Enterprises invest in APM tools for several key reasons:

- Comprehensive Data Collection: APM tools gather and quantify data from all components affecting an application's performance. This includes the application's hosting platform, process utilization, memory demands, and disk read/write speeds.
- Processor Utilization Tracking: APM tools monitor processor utilization by measuring the number of operations per second the CPU server performs.
- Memory Usage Monitoring: High memory usage can lead to performance issues. APM tools track short-term data storage by the CPU to prevent such problems.
- Error Rate Tracking: At the software level, APM tools monitor error rates to identify how often an application encounters issues or failures, such as insufficient memory access.

- Code Execution Monitoring: APM tools examine code execution to identify bottlenecks in memory-intensive processes, like database searches.
- Server Load Balancing Analysis: Automated load balancing can mask issues by making the combined performance of servers appear optimal. APM tools help IT professionals avoid this by tracking all servers simultaneously to detect uneven load distribution and potential problems.

7.2.3 Importance of Application Performance Monitoring (APM)

Application performance monitoring (APM) is crucial for several reasons, offering benefits that enhance organizational effectiveness, reputation, brand, and long-term cost efficiencies.

The 5 Benefits of APM for Businesses:
Application performance monitoring is essential for maintaining the health of an application. Running an application without APM software is like driving a car without any gauges – you can do it. Still, you never know when something unfortunate is happening or about to happen. This simple analogy highlights the core principle of APM: it's all about the data it collects.

1. Reduced Time to Recovery One of the most immediate benefits of APM is a faster recovery time from outages and other critical incidents. Mean time to recovery (MTTR) is a valuable metric as it identifies the risk each new release or significant infrastructure change introduces. A well-utilized APM tool helps identify problems early, reducing the likelihood that recovery is needed at all. Active measurement of critical metrics allows for proactive mitigation rather than reactive response.
2. Rapid Defect Diagnosis Not all issues result in downtime. Many defects and software bugs can harm UX without causing an outage. APM eliminates the `hunt and peck` method of diagnosing issues and helps zero in on the exact root cause. Easier defect diagnosis means developers can spend more time building new features, leading to a more stable and performant application and fostering innovation, product growth, and reduced customer churn.
3. Increased Engineering Productivity APM tools are beneficial in both production and pre-production environments. Integrating APM in development and staging environments helps identify potential issues that standard unit and integration tests might miss. This additional information prevents defects before they occur, reducing ticket rework. With more data at their disposal, development, and test teams can fine-tune the product before exposure to customers.
4. Lowered Cost of Goods A poorly architected application can become expensive quickly. APM provides insights into the application's performance, allowing for tuning of the underlying infrastructure to ensure the optimal use of resources. This can include different CPU and memory requirements or more efficient code design, ultimately reducing the cost of keeping the application running.
5. Customer Satisfaction Every benefit of APM leads to one outcome: customer satisfaction. Whether it's identifying and cutting unnecessary spending, improving product stability, or releasing features more efficiently, the result is a happier, more satisfied customer.

7.2.4 Components of Application Performance Monitoring (APM)

Application performance monitoring (APM) focuses on tracking five primary aspects of application performance:

- Runtime Application Architecture
- Real User Monitoring
- Business Transactions
- Component Monitoring
- Analytics and Reporting

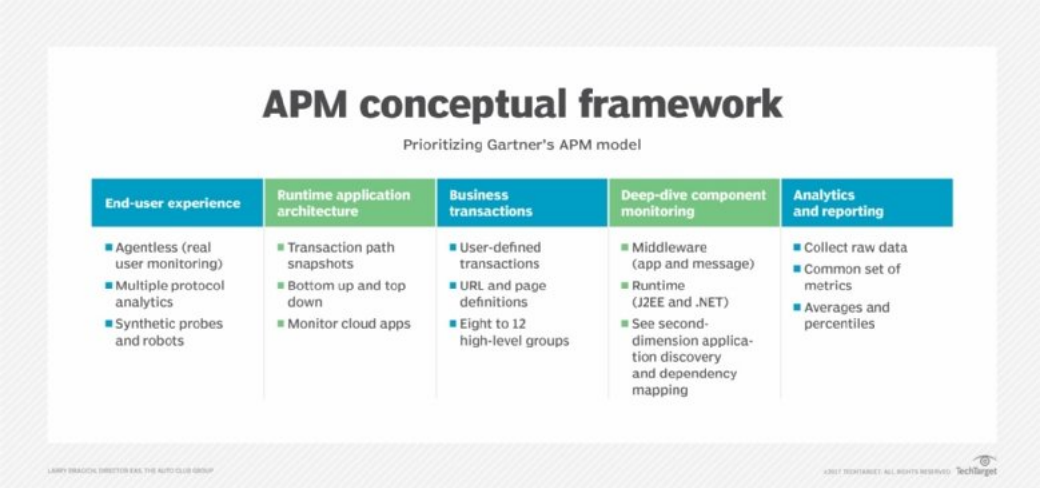


Figure 65: Networking APM conceptual framework

Runtime Application Architecture:
This component analyzes the hardware and software elements involved in the app's execution, as well as the communication paths they use. By recognizing patterns and identifying performance issues, IT professionals can anticipate future problems and plan for necessary upgrades, such as additional application storage, in a timely manner.

Component Monitoring:
Also known as application component deep dive, this involves monitoring all elements of the IT infrastructure extensively. It includes an in-depth analysis of all resources and events within the app performance infrastructure, covering servers, operating systems, middleware, application components, and network components. Component monitoring provides a comprehensive understanding of the various elements and pathways identified in previous processes.

Analytics and Reporting:
This component translates the data gathered from the above processes into actionable information. It includes:

- Defining a performance baseline using historical and current data to set expectations for normal app performance.
- Identifying potential areas for improvement by comparing infrastructure changes to performance changes.
- Efficiently identifying, locating, and resolving performance issues using historical and baseline data.

- Predicting and mitigating potential future issues using actionable insights.

Analytics and reporting are essential for ensuring the organization achieves a good return on investment (ROI) from both the application and the APM efforts.

7.3 How Application Performance Monitoring (APM) Works

APM tools or platforms function by monitoring the performance and behavior of applications. If an application deviates from expected behavior, these tools collect data on the source of the issue, analyze this data in terms of its impact on the business, and make adjustments to the application environment to prevent similar problems from occurring in the future. When setting up an APM platform, consider three categories of data:

Figure 66: Pillars of observability

- Logs: Textual records of events generated by infrastructure elements, providing historical context and real-time telemetry data.
- Metrics: Real-time operating data accessed through APIs or as generated events, driving fault management tasks.
- Traces: Records of information pathways or workflows that follow a transaction through various processes, indirectly assessing application logic.

Given that performance monitoring is a subset of the broader performance management field, it's crucial to understand that monitoring data and analytics alone might not guarantee an optimal user experience. Performance management systems often integrate monitoring data with automation and orchestration to enable autonomous problem remediation.

Observability is a strategy in management that emphasizes prioritizing the most relevant and critical issues within an operational workflow. This term also pertains to software processes that help distinguish essential information from routine data, as well as the extraction and processing of crucial information at the highest operational levels.

Observability is rooted in control theory, which posits that the internal states of IT systems can be inferred from their inputs and outputs. It's often considered a top-down assessment. The challenge lies more in collecting the right observations than in deducing the internal state from these observations.

7.3.1 Differences Between Monitoring and Observability

While monitoring and observability are related, they have distinct **differences:**

- Data Collection: Monitoring passively collects data, often resulting in overwhelming amounts of insignificant information. Observability, on the other hand, actively gathers data that is crucial for operational decisions and actions.
- Sources of Information: Monitoring uses existing data sources like management information bases, APIs, and logs. Observability may also introduce new data points to collect essential information.
- Focus Areas: Monitoring typically concentrates on infrastructure, whereas observability equally emphasizes applications, including workflows.
- Data Usage: In monitoring, the collected data is often the end goal. In observability, data is intended to feed into an analytic process that optimally represents the state of an application or system.

Key APM Metrics:

- Performance Metrics:
  - Response Time: Measures the time taken for an application to respond to requests.
  - Error Rates: Indicates the frequency of errors occurring within the application.
  - Throughput: Measures the rate at which the application processes requests or transactions.
- Resource Utilization Metrics:
  - CPU Usage: Percentage of processing capacity used by the application.
  - Memory Usage: Amount of memory consumed by the application, including page faults and disk access times.
  - Disk I/O: Input/output operations performed on disk storage by the application.
  - Network Latency: Delay in data communication over the network.
- Availability and Reliability Metrics:
  - Uptime: Measure of how long an application is available and operational.
  - Apdex Score: Standardized metric combining response time into a single score reflecting user satisfaction.
  - Database Performance: Metrics related to database query response time, transaction throughput, and utilization.

These metrics collectively provide insights into various aspects of application performance, resource usage, availability, and user satisfaction. Monitoring these metrics helps identify performance bottlenecks, optimize resources, and ensure a smooth user experience.

7.3.2 Designing a Metrics Monitoring and Alerting System

What are the major components of the system?

A metrics monitoring and alerting system consists of four components:

- Data collection: collects metric data from different resources.
- Data transmission: transfers data from sources to the metrics monitoring system.
- Data storage: organizes and stores incoming data.
- Alerting: analyzes the incoming data, detects anomalies and generates alerts. The system must be able to send alerts to different communication channels configured by the organization.

How to Design The Metrics for Monitoring and Alerting System

High-level Design

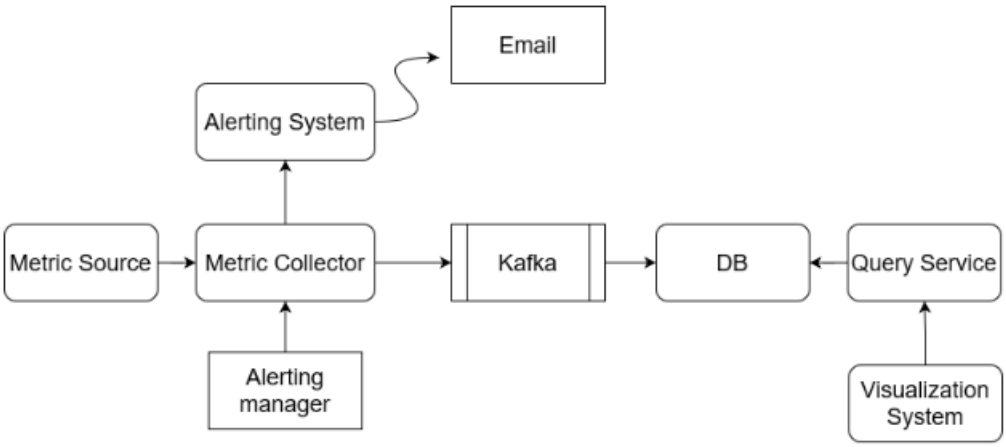


Figure 67: System Achitecture

- Metrics source: The Embedded kit in the vehicle
- Metrics collector: Gathers metrics data and writes data into the Kafka queue
- Query service: The query service makes it easy to query and retrieve data from the databases.
- Alerting system: This sends alert notifications to various alerting destinations.

Design Deep Dive

Let's investigate the designs in detail:

- Metrics collection.
- Scaling the metrics.
- Transmission pipeline.
- Query service.
- Alerting system.

Metrics Collection:

There are two ways metrics data can be collected – pull or push.

Pull model:

In a pull model, the metrics collector pulls the metrics from the sources. Consequently, the metrics collector needs to know the complete list of service ends to pull the data. We can use a reliable, scalable, and maintainable service like service discovery, provided by ETCD and Zookeeper. A service discovery contains configuration rules about when and where to collect the metrics.

- The metrics collector fetches the configuration metadata of the service endpoint from service discovery. Metadata includes pulling interval, IP addresses, timeout, and retry parameters.
- The metrics collector pulls the metric data using the HTTP endpoint (for example, web servers) or TCP (transmission control protocol) endpoint (for DB clusters).
- The metrics collector registers a change event notification with the service directory to get an update whenever the service endpoints change.

Figure 68: Pull Model

Push model:

In a push model, a collection agent is installed on every server that is being monitored. A collection agent is long-running software that collects the metrics from the service running on the server and pushes those metrics to the collector. To prevent the metrics collector from falling behind a push model, the collector should always be in an autoscaling position with a load balancer in front of it. The cluster should scale up or down based on the CPU (central processing unit) load of the metrics collector.
Pull or push?

So, what's best for a large organization? Knowing the advantages and disadvantages of each approach is important. A large organization needs to support both, especially serverless architecture.

Push Monitoring System:

Advantages:

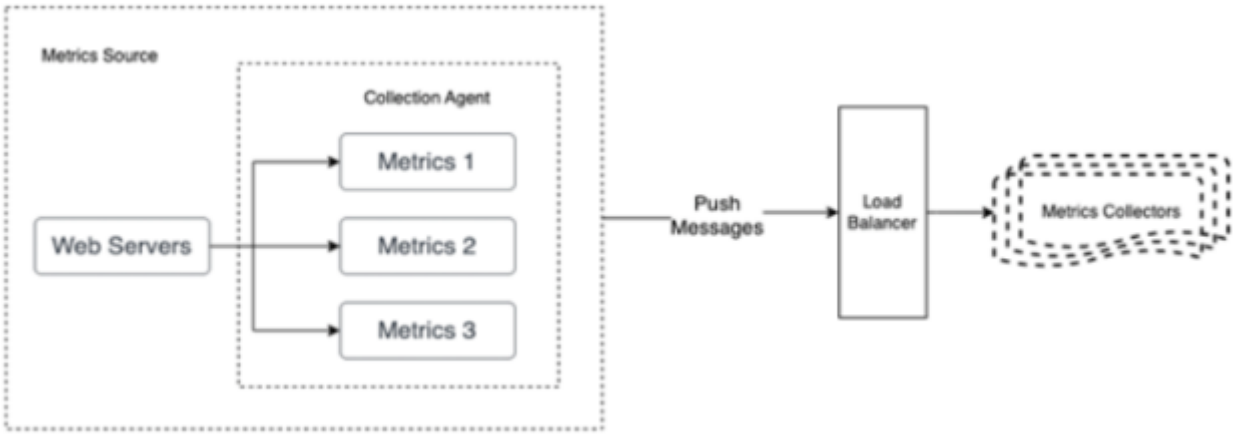


Figure 69: Push Model

- Real-time notifications of issues and alerts.
- Can alert multiple recipients at once.
- Can be customized to specific needs and requirements.
- Can be integrated with other systems and applications.

Disadvantages:

- Requires a constant and reliable internet connection to function properly.
- Can be overwhelming with too many notifications and alerts.
- Can be vulnerable to cyber-attacks and security breaches.

Pull Monitoring System:

Advantages:

- Can be accessed remotely and for multiple devices.
- Can be set up to check specific metrics and parameters at regular intervals.
- Can be easily configured and customized.
- Can provide detailed and historical data for analysis and reporting.

Disadvantages:

- Requires manual intervention to check and review the data.
- May not provide real-time alerts and notifications.
- Can be less efficient in identifying and responding to issues and anomalies.

Scaling the Metrics Transmission Pipeline:

Whether we use the push or pull model, the metrics collector of servers and the cluster receive enormous amounts of data. There's a risk of data loss if the time-series database is unavailable. To navigate through

the risk of losing data, we can use a queueing component. In this design, the metrics collector sends metric data to a queuing system like Kafka. Then consumers or streaming processing services such as Apache Spark process and push the data to the time-series database. This approach has several advantages:

- Kafka is used as a highly reliable and scalable distributed messaging platform.
- It decouples the data collection and processing services from one another.
- It can easily prevent data loss when the database is unavailable by retaining the data in Kafka.

Query service

The query service comprises a cluster of query servers that access the time-series database and handle the requests from the visualization or alerting systems. Once you have a dedicated set of query servers, you can decouple the time-series database from the visualization and alerting systems. This provides us with the flexibility to change the time-series database or the visualization and alerting systems, whenever needed.

Alerting system

A monitoring system is very useful for proactive interpretation and investigation, but one of the main advantages of a full monitoring system is that administrators can be disconnected from the system. Alerts allow you to define situations to be actively managed while relying on passive monitoring of software to watch for changing conditions.

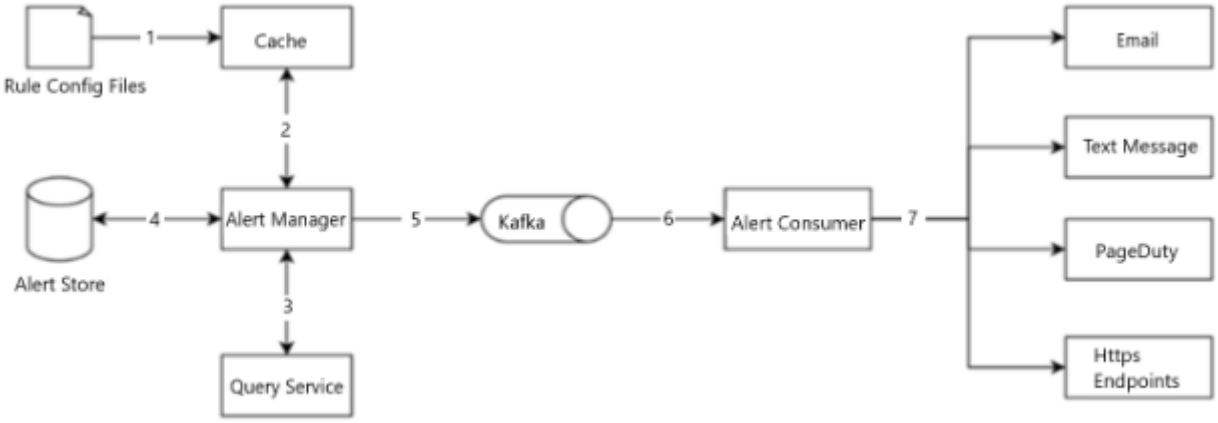


Figure 70: Alerting Architecture

The alert flow works as follows:

1. Load the config files to the cache servers. Rules are defined as config files on the disk. The alert manager fetches alert configs from the cache.
2. Based on the config rules, the alert manager calls the query service at a predefined interval. If the value violates the threshold, an alert event is created. The alert manager is responsible for the following:
    - Filter, merge, and dedupe alerts. Here's an example of merging alerts that are triggered within one instance in a short amount of time.
    - Access control—to avoid human error and keep the system secure, it is essential to restrict access to certain alert management operations to authorized individuals only.

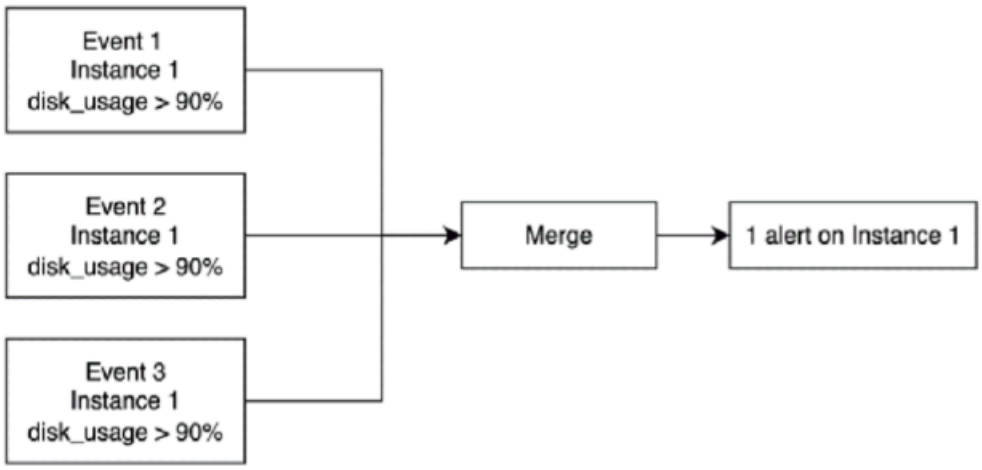


Figure 71: Alerting Rule

- Retry—the alert manager checks alert states and ensures a notification is sent at least once.

3. The alert store is a key-value database such as Cassandra, that keeps the state (in-active, pending, firing, resolved) of all alerts. It ensures a notification is sent at least once.
4. Eligible alerts are inserted into a messaging and queuing system such as Kafka.
5. Alert consumers pull alert events from the messaging and queuing system.
6. Alert consumers process alert events from the messaging and queuing system and send notifications to different channels such as email, text message, PageDuty, or HTTP endpoints.

7.3.3 Driver Monitoring System

Scenario: Real-time Monitoring and Analysis for a Driver status and Embedded kit health.

Objective:
To ensure the safety, efficiency, and performance of a fleet of delivery vehicles by using a driving monitoring system that collects and analyzes metrics and logs.

Components of the System:

- Metrics Collection:
  - CPU Usage: Percentage of processing capacity used by the application on BeagleBone AI-64.
  - Memory Usage: Amount of memory consumed by the application, including page faults and disk access times.
  - Disk I/O: Input/output operations performed on disk storage by the application.
- Logs Collection:
  - Event Logs: Logs of specific events such as Driver yawn/sleepy, Engine start/stop, and Driver changes.
  - Error Logs: Logs capturing any system malfunctions, Thrown Exception, or anomalies.

– Trip Logs: Detailed records of each trip, including start and end times, routes taken, and stops made. Communication Logs: Logs of communications between the vehicle and the central monitoring system.

Implementation Steps:

Data Collection:
A background job in the main Application has two tasks:

- Fired periodically with Kit health metrics (e.g. CPU usage)
- Fired once the application issued logs utilizing MQTT broker as Metric collector that the kit pushes its metric and logs to it.

Metric collector:
A Roud side unit works as an MQTT broker that has a number of connections with the vehicle.

Alerting System:
Embedded in each metric collector and calibrate streamed metrics against its configurable rules, which can be configured using.YAML file In that file, we can specify:

- Alerting policy (e.g Kit temperature below 50 Celsius)
- Alerting means (e.g. Email)

Queuing:
In case the number of vehicles increases, we must scale up the number of metric collectors as well. So we will add a queue like Kafka(partition-based queue system) to work as a buffer between the vehicle's logs and database.

Query service:
Wrapper on Database to feasible the way to query the data from.

Visualization System:
Provide a user interface (Mobile app) by which users can fetch logs and updates By utilizing the Query service.

8 Visualization

To effectively visualize data related to drivers and cars, utilizing mobile applications is the most straightforward and efficient approach. Mobile applications are widely used for several key reasons.

8.1 Mobile Application

Why Use Mobile Applications

Mobile applications have become an integral part of our daily lives, providing a seamless and efficient way for users to interact with various services and perform tasks on the go.

8.1.1 Benefits of Mobile Applications

- **Convenience and Accessibility:** Mobile applications provide users with access to services and information wherever they are, as long as they have their mobile device. Mobile devices are portable, making them easier to carry and use compared to traditional desktop computers.
- **User Engagement:** Mobile apps offer personalized experiences and direct communication channels, increasing user engagement.
- **Integration with Device Features:** Mobile apps can leverage device features such as GPS, camera, and sensors to enhance functionality and user experience.
- **Push Notifications:** Mobile apps can send push notifications to keep users engaged and informed about updates, promotions, or important information.
- **Interactive Features:** Apps often have more interactive features compared to mobile websites, providing a better user experience.
- **Faster Loading Times:** Mobile apps generally load faster than mobile websites because they store data locally on the device.
- **Enhanced Functionality:** Apps can offer more complex functionalities and a smoother user experience since they can utilize the device's hardware and software capabilities.
- **Offline Access:** Mobile apps can store data locally, allowing users to access certain features and content even without an internet connection.
- **Data Protection:** Mobile apps can offer better security features to protect user data compared to mobile websites.

8.1.2 Impact on Business for DMS as a Product

- **Increased Visibility:** A mobile app increases a brand's visibility and presence in the market.
- **Data Collection:** Mobile apps can collect valuable user data, helping businesses understand user behavior and preferences to improve services and marketing strategies.

8.2 Utilizing Flutter for Mobile Application Development

To visualize data in a way that is easily accessible to all customers, we considered multiple approaches, including mobile applications and web pages (desktop). We decided to use a mobile application for its convenience and widespread usage.

Why Use Flutter?
Mobile applications have become an integral part of our daily lives, providing a seamless and efficient way for users to interact with various services. However, the challenge arises when we need to develop applications for both Android and iOS platforms. This is where cross-platform mobile application development with a single codebase becomes advantageous.

It is an open-source mobile application development framework created by Google. It uses the Dart programming language to build high-performance, cross-platform mobile applications with a single codebase. Flutter allows developers to build beautiful, fast, and responsive apps for both Android and iOS platforms.

Flutter uses a unique approach to building user interfaces called `widgets`. Widgets are the building blocks for creating user interfaces, and they can be combined and customized to create complex layouts and designs. Flutter provides a rich collection of customizable widgets that can be easily integrated into an application's user interface.

Flutter also includes a `hot reload` feature, which allows developers to quickly see changes they make to their code in real-time. This feature significantly speeds up the development process, as developers can iterate quickly and see the results of their changes instantly.

8.2.1 Advantages of Using Flutter

- **Single Codebase for Multiple Platforms:** Write once, run anywhere. Flutter allows you to use a single codebase for both iOS and Android, saving time and effort.
- **Fast Development:** Flutter's hot reload feature allows you to see the changes you make in your code almost instantly, without losing the current application state.
- **High Performance:** Flutter apps are compiled directly to native ARM code, making them faster and more efficient than those using JavaScript bridges.
- **Expressive and Flexible User Interface (UI):** Flutter provides a rich set of customizable widgets that adhere to both Material Design and Cupertino (iOS) principles.

8.3 Flutter Development

8.3.1 Setting Up Flutter

1. **Install Flutter SDK:** Download and install the Flutter SDK from the official website (`https://flutter.dev`).
2. **Set Up the Editor:** Configure your preferred code editor (e.g., Visual Studio Code, Android Studio) with Flutter and Dart plugins.

3. **Create a New Flutter Project:** Use the Flutter CLI to create a new project with the command `flutter create my_app`.

8.3.2 Project Structure

- **lib/main.dart:** The entry point of your Flutter application.
- **pubspec.yaml:** Contains metadata about the app, including dependencies.
- **android/:** Contains Android-specific code and configuration.
- **ios/:** Contains iOS-specific code and configuration.

8.3.3 Building the User Interface

- **Widgets:** The building blocks of a Flutter app. Using widgets to create the UI.
- **Stateful and Stateless Widgets:** Stateful widgets maintain state that can change during the lifetime of the widget, while stateless widgets do not.
- **Layouts:** Use layout widgets like `Row`, `Column`, `Stack`, and `Container` to arrange other widgets on the screen.

8.3.4 State Management

- **Stateful Widgets:** Manage state directly within a widget.
- **Provider:** A recommended state management approach that makes it easy to manage and share state across your app.

8.3.5 Networking and Data Storage

- **HTTP Requests:** Use the `http` package to fetch data from the internet.
- **Local Storage:** Use packages like `shared_preferences` and `sqflite` for local storage.

8.3.6 Testing

- **Unit Testing:** Test individual units of code.
- **Widget Testing:** Test the UI and interactions of individual widgets.
- **Integration Testing:** Test the complete app or large parts of it.

8.3.7 Integration with Data Visualization

We demonstrate the integration of data visualization in our mobile app using Flutter. Here are some screenshots of our application:

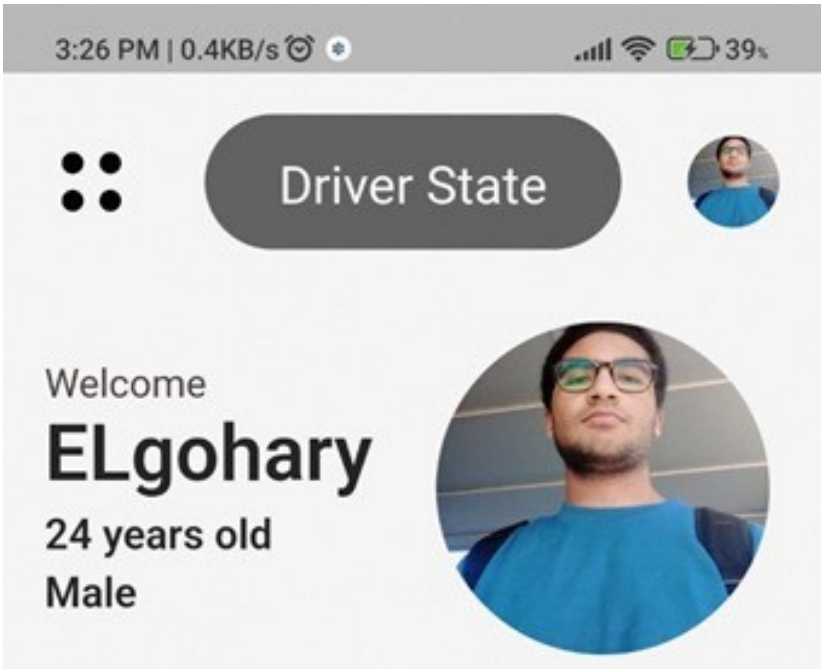


Figure 72: Screenshot showing a sample driver profile

Fig. 72 displays a sample driver profile page, including a picture of the driver, age, and gender.

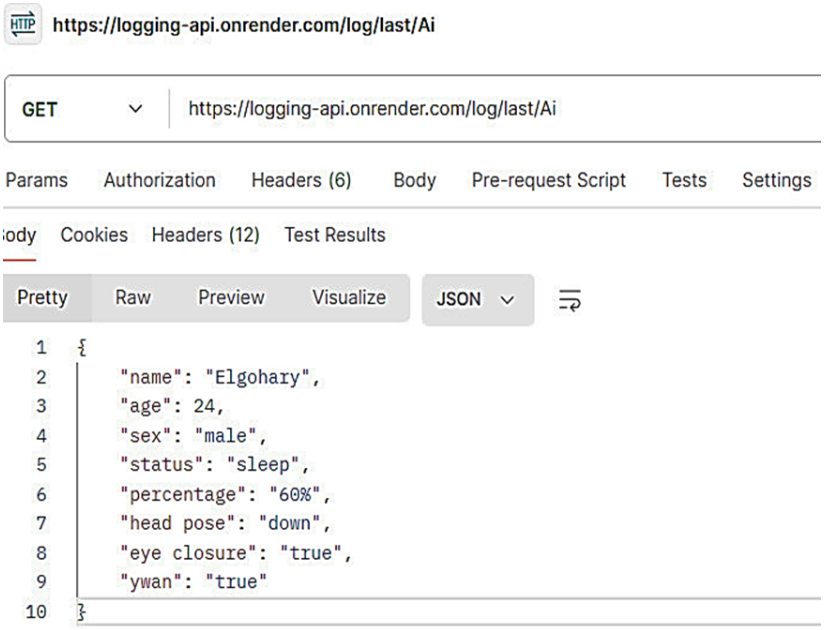


Figure 73: JSON file received from server

Fig. 73 illustrates the JSON file received from the server, which contains data used for visualizing driver information.

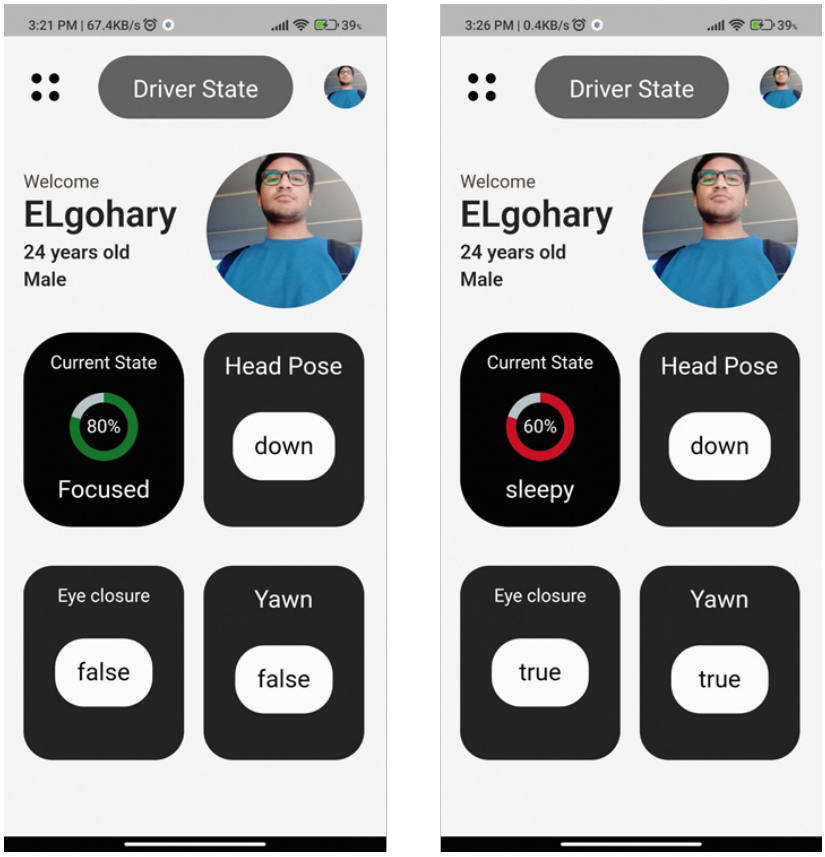


Figure 74: Visualization of driver data in the UI

Fig. 74 demonstrates how data from the JSON file is processed and visualized in the user interface of our Flutter application.

8.3.8 Deployment

- **Android:** Generate a signed APK or App Bundle for distribution on the Google Play Store.
- **iOS:** Create an iOS archive and upload it to the Apple App Store using Xcode.

8.3.9 Strategies for Optimization

- **Use Fast Devices:** Utilize fast and modern development devices with high CPU and RAM specifications.
- **Hot Reload and Hot Restart:** Use hot reload for small changes and hot restart for larger changes to

9 Linux Image

Throughout this project, many tasks of building a minimal Linux image for BeagleBone AI-64 using Yocto were under-looked. This was achieved through exploring various resources to gain a comprehensive understanding of Linux images and the tools available for creating them. first of all studying the Linux kernel and its role in Linux images,then an in-depth examination of Buildroot and Yocto, two prominent tools for building custom Linux distributions. Despite facing several challenges, including storage limitations and configuration issues, a successful build and testing the desired minimal image for BeagleBone AI-64 using Yocto was reached. This section provides a detailed account of the methodologies, challenges, and solutions encountered during the project, offering insights into the process of creating custom Linux images for embedded systems.

9.1 Introduction

This section focuses on the creation of a minimal Linux image for BeagleBone AI-64 using Yocto. Custom Linux images are essential for embedded systems, providing tailored operating systems for specific applications. The primary tool explored in this project is Yocto, known for its extensive customization capabilities and support for various hardware platforms. however another building tool like buildroot was used to provide the project with comparison process to select best tool suitable for the project.

9.2 Background

9.2.1 Comparison Between Bare Metal Programming, RTOS, and Embedded GPOS

Bare Metal Programming: Bare Metal Programming refers to running software on the hardware without an Operating System. It would involve the programming of registers of the hardware directly and taking care of every minor detail in hardware control, timing, and scheduling. This approach gives very great control and top performance but is hard to understand deeper into the hardware and complex to handle when the system grows.

Advantages: This provides maximum performance, full control of the HW, and very little overhead.
Disadvantages: If there is high complexity; it is complicated to manage large systems, and there is no portability.

Real-Time Operating System, RTOS: An RTOS is designed to support real-time applications and is projected to have by far the most time-deterministic activities with requisite responsiveness. The RTOS offers only basic scheduling between the tasks, resource management, and inter-task communications but has the ability to execute high-priority tasks within strict time bounds. FreeRTOS, VxWorks, and Micrium are examples of RTOS.

Advantages: Deterministic timing, improved complex task management, improved reliability.
Disadvantages: Fewer features than GPOS can offer; still complex to develop.

Embedded GPOS: An embedded GPOS, like Linux, provides a full operating environment with multitasking, networking, file systems and a large body of applications. These systems provide many more features than RTOS however they don't guarantee real time performance.

Advantages: It has a rich set of features; its support is just great for many applications; it has large community support.
Disadvantages: It involves higher overhead, reduced control over hardware, and possibly non-deterministic behavior.

9.2.2 Linux Kernel

The Linux kernel is the core component of the Linux operating system. It manages system resources and enables communication between hardware and software components. any operating system consists of three essential components which are :

1. **The hardware:** The physical machine—the bottom or base of the system, made up of memory (RAM) and the processor or central processing unit (CPU), as well as input/output (I/O) devices such as storage, networking, and graphics. The CPU performs computations and reads from, and writes to, memory.

2. **The Linux kernel:** The core of the OS. (See? It's right in the middle.) It's software residing in memory that tells the CPU what to do.

3. **User processes:** These are the running programs that the kernel manages. User processes are what collectively make up user space. User processes are also known as just processes. The kernel also allows these processes and servers to communicate with each other (known as inter-process communication, or IPC).

The kernel provides essential services such as process management, memory management, device drivers, and system calls. Its modular architecture allows for customization, making it suitable for a wide range of devices from desktop computers to embedded systems. Understanding the Linux kernel is crucial for developing custom Linux images, as it forms the foundation upon which the entire operating system is built.

9.2.3 Linux Image

A Linux image is a complete package that includes the Linux kernel, system libraries, and necessary applications to run a Linux-based operating system. Creating a Linux image involves configuring the kernel, selecting the appropriate packages, and compiling them into a single bootable image. Linux images can be tailored to meet the specific needs of different applications, which is particularly important for embedded systems that require minimal and efficient operating environments.

9.2.4 Comparison of target image build methods

Manual Compilation: The manual building of a Linux image consists of downloading the source code of a kernel, cross-compiling it in view of the target architecture, and finally assembling the root filesystem together with necessary libraries and applications. This method is highly time-consuming and full of errors, but on the other hand, it allows for full control.

Figure 75: Building Methods

Advantages: Full control of the build process, high customizability.
Disadvantages: Long, requires deep expertise, increased risk of error.

Build Systems Tools: Build systems tools like Buildroot, OpenWrt, and Yocto provide tools and scripts to configure, compile, and package Linux images. This makes the procedure much easier and reduces possible errors.
Advantages: Easy building procedure, reduced errors, re utilization of configurations is possible.
Disadvantages: there's a little learning curve; some limitations in flexibility of customization.

9.2.5 Comparison Between Buildroot, OpenWrt, and Yocto:

Buildroot:

Buildroot is the simplest, most effective, and user-friendly tool to create embedded Linux systems by cross-compilation. It provides a collection of Makefiles and patches and does all the download, configuration and compilation of all the needed components. It is an open-source tool that simplifies the process of generating embedded Linux systems through cross-compilation. It provides a set of makefiles and patches that automate the download, configuration, and compilation of necessary components. Buildroot is known for its simplicity and ease of use, making it a popular choice for quickly setting up embedded Linux environments.

Toolchain: The toolchain includes the compiler, linker, and other tools required to build software for the target architecture.

Configuration: Buildroot uses a menu-driven configuration tool that allows users to select the packages and features to include in the final image. This tool simplifies the customization process.

Build Process: Once configured, Buildroot automates the build process, generating a root filesystem, kernel, and bootloader for the target system. This streamlined approach makes it efficient for developing embedded Linux systems.
Advantages: simplicity, easy set-up and build, minimalistic.
Disadvantages: less flexible to customize when comparing to Yocto, fewer features.

OpenWrt

OpenWrt is a Linux distribution primarily aimed at the target of embedded devices, mostly routers. It provides a fully writable file system with package management and hence is pretty flexible and extendable.
Advantages: Package management, wide hardware support, strong community.
Disadvantages: The project is targeted mostly at networking devices; therefore, it might not work for

many other applications.

Yocto

The Yocto Project is an open-source collaboration project that provides templates, tools, and methods for creating custom Linux-based systems. It is highly customizable and is widely used for developing embedded systems. Yocto offers greater flexibility and customization compared to Buildroot but also comes with increased complexity.
BitBake: is the task execution engine used by Yocto. It processes recipes that define how software is built, ensuring dependencies are resolved and tasks are executed in the correct order.

Recipes: are files that describe how to build a particular piece of software. They include information about source code location, dependencies, and build instructions. Recipes are central to the customization capabilities of Yocto.

Layers: Yocto uses a layered architecture that allows developers to add or modify functionality without altering the core system. Layers can include board support packages (BSPs), middleware, and applications. This modularity enables extensive customization and reuse of components.

Poky: is the reference distribution of the Yocto Project. It includes the OpenEmbedded build system, metadata, and a set of default configurations to get started with Yocto. Poky serves as a starting point for developing custom distributions. it is a base specification of the functionality needed for a typical embedded system as well as the components from the Yocto Project that allow you to build a distribution into a usable binary image.

One big advantage of the Yocto Project is that it builds everything in stages and layers. If you make any changes (e.g. add a layer, change to a different image, tweak kernel settings), subsequent builds will take far less time.
This speeds up development process when you are trying to add low-level support in Linux. We build image using bitbake and we can make change to bitbake.
The first run take longer time than the following runs as advantage of yocto is that it tries to memorize where it was last time ($\tilde{1}$0 min)
Open Embedded (OE) is essentially a database of these recipe (.bb) and configuration (.conf) files that developers can draw on to cross-compile combinations of components for a variety of embedded platforms.
Advantages: Highly customizable, scalable, supports a wide array of hardware.
Disadvantages: Steeper learning curve, complex setup.

9.3 Methodology

9.3.1 Buildroot Methodology

Overview: This section outlines the steps taken to create a minimal Linux image for BeagleBone AI-64 and QEMU using Buildroot.

Materials and Tools

- **Hardware:** BeagleBone AI-64, development PC.

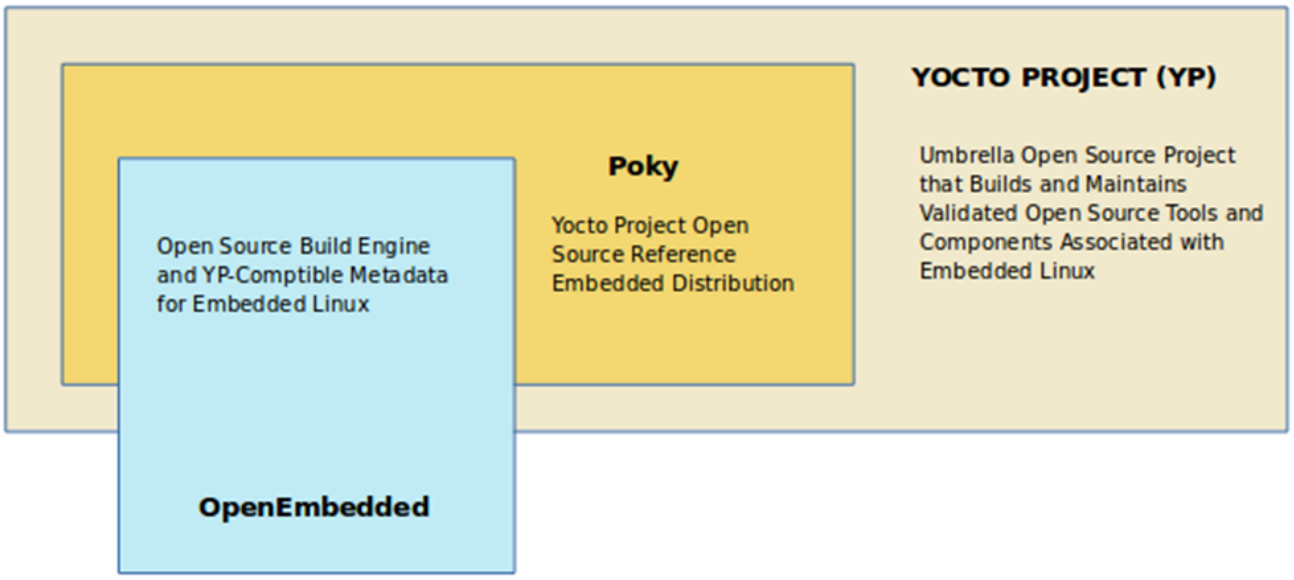


Figure 76: Hierarchical Architecture of Yocto Project

- **Software:** Buildroot, QEMU.
- **Tools:** Cross-compilation toolchain, terminal emulator, text editor.

Procedure

Initial Setup

- Download and install Buildroot from the official website:
- Make sure that you update your current version of ubuntu
- You may not have installed ncurse already in your device use :
  Note : ncurse is used for configuration of the image using make menuconfig which will be used later in this project.

Buildroot Configuration

- Get into configs file to see what is there.
- Select the appropriate target architecture (e.g., Aim for
- Run ‘make menuconfig‘ in the Buildroot directory to open the configuration menu.
- Choose the necessary packages and kernel options for a minimal image.
- Save the configuration.
  Note : error may occur if ncurse library isn’t installed

Building the Image

- Execute ‘make‘ to start the build process.

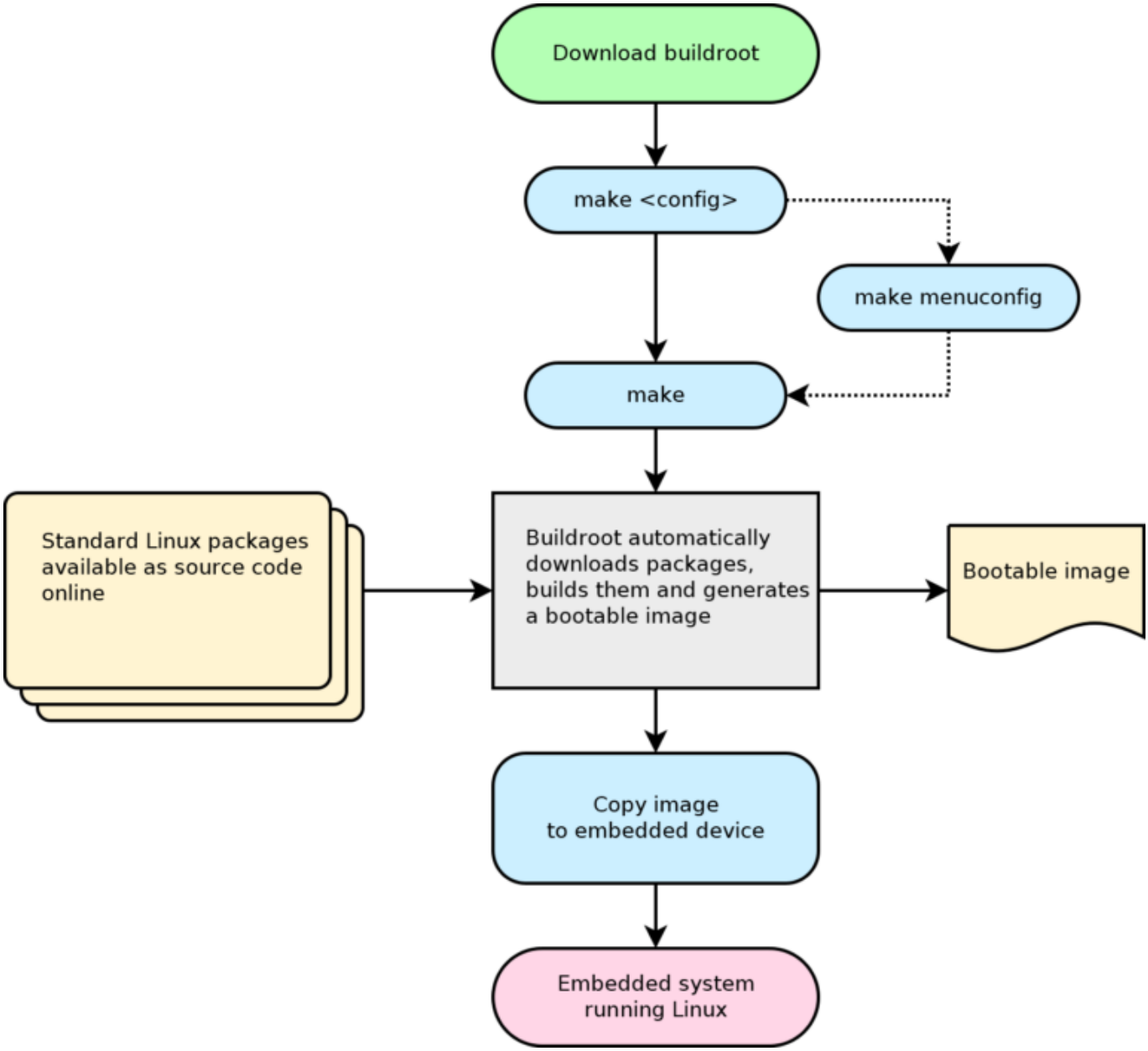


Figure 77: Buildroot Flowchart

- Monitor the build process for any errors or missing dependencies.
- Upon completion, locate the built image in the output directory.
  ls -la output/images/

Testing the Image on QEMU

- Install QEMU on the development PC if not already installed.
- Run the built image on QEMU using the appropriate command.
- Verify the boot process and basic functionality.
  try : echo , date ,...etc

Testing the Image on BeagleBone AI-64

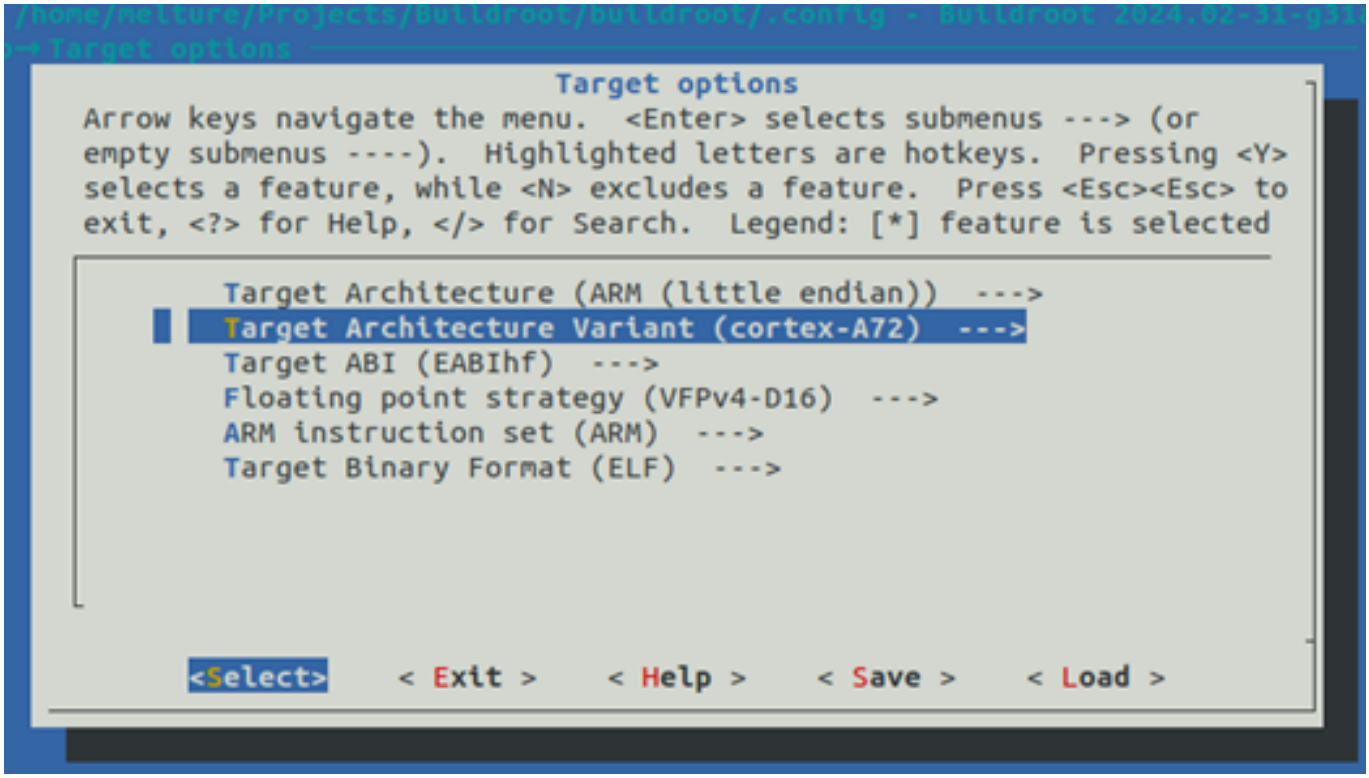


Figure 78: Configuration Menu

- Flash the built image onto an SD card using a tool like ‘dd‘ or Etcher.
  using dd :
  to upload image to SD-Card :
  First to list storage devices :
  lsblk
  Copy image bit by bit using sudo to have root privileges
  sudo dd if=”path of file to be copied” of=”destination to be copied to” bs=”set block size(1M)”
  Etcher will be displayed when using Yocto image later in this book

- Insert the SD card into the BeagleBone AI-64 and power it on.

- Verify the boot process and basic functionality on the actual hardware.
  verification can be done using picocom
  To get serial console from board we can install serial terminal
  You may need to call out usermod in dialout group to avoid permission denied even using sudo
  We can now use picocom
  picocom -b “baud rate” “Path of device”
  Reset the board and if you get login prompt then every thing works fine
  • Type root and there is no password for login
  • To write script use
  type I to insert text
  after you finished press esc and type :wq
  • Exit picocom

Troubleshooting and Optimization:

- QEMU image run perfectly without graphic interface
- To optimize our images , only the necessary libraries are included while the rest are discarded however Any change repeat all this steps again (long time wasted)

9.3.2 Yocto Methodology

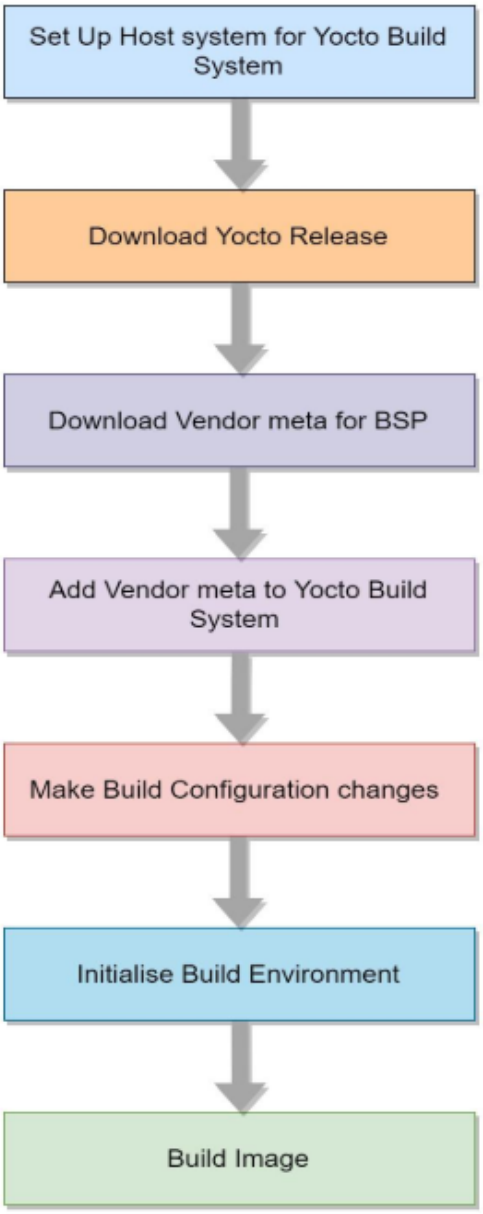


Figure 79: Yocto Flowchart

Overview: This section outlines the steps taken to create a minimal Linux image for BeagleBone AI-64 and QEMU using Yocto.

Materials and Tools:

- **Hardware:** BeagleBone AI-64, development PC.
- **Software:** Yocto Project tools (BitBake, OpenEmbedded , poky distribution), QEMU.

• At least 90 Gbytes of free disk space, although much more will help to run multiple builds and increase performance by reusing build artifacts.

* We use virtual machine of about 150 GB

• At least 4 Gbytes of RAM, though a modern modern build host with as much RAM and as many CPU cores as possible is strongly recommended to maximize build performance.
We allocated 8 GB RAM for virtual machine

• Runs a supported Linux distribution (i.e. recent releases of Fedora, openSUSE, CentOS, Debian, or Ubuntu). For a list of Linux distributions that support the Yocto Project, see the Supported Linux Distributions section in the Yocto Project Reference Manual. For detailed information on preparing your build host, see the Preparing the Build Host section in the Yocto Project Development Tasks Manual.

- Git 1.8.3.1 or greater
- tar 1.28 or greater
- Python 3.8.0 or greater.
- gcc 8.0 or greater.
- GNU make 4.0 or greater
- Python version

- **Tools:** Cross-compilation toolchain, terminal emulator, text editor.

Procedure:

Initial Setup:

- Because the Yocto Project tools rely on the "python" command, you will likely need to alias "python" to "python3." Edit your .bashrc file
  Scroll to the bottom and add the following to a new line (press 'a' to append new text):
  Save and exit ('esc' followed by entering ":wq"). Re-run the .bashrc script to update your shell
  Check your Python version:
  Note : It should say something like "Python 3.8.xxx."

- Download and set up the Yocto Project environment from the official website:`https://www.yoctoproject.org/`.

- Install the required dependencies and tools for Yocto.

- Clone the necessary Yocto layers, including 'poky'

Yocto Configuration:

- Source the Yocto environment setup script: 'source oe-init-build-env'.

- Configure the build environment by editing the 'conf/local.conf' file.

- Specify the target machine in 'conf/local.conf' (e.g., 'MACHINE = `beaglebone-yocto`') for BeagleBone AI-64.

- Add necessary layers using 'bitbake-layers add-layer' command.

- Add a Hardware Layer (arago)

```
10 BBLAYERS ?= " \
11   /home/melture/Projects/yocto/poky/meta \
12   /home/melture/Projects/yocto/poky/meta-poky \
13   /home/melture/Projects/yocto/poky/meta-yocto-bsp \
14   /home/melture/Projects/yocto/sources/meta-openembedded/meta-python \
15   /home/melture/Projects/yocto/sources/meta-openembedded/meta-oe \
16   /home/melture/Projects/yocto/sources/meta-openembedded/meta-networking \
17   /home/melture/Projects/yocto/sources/meta-openembedded/meta-gnome \
18   /home/melture/Projects/yocto/sources/meta-openembedded/meta-filesystems \
19   /home/melture/Projects/yocto/sources/meta-arago/meta-arago-distro \
20   /home/melture/Projects/yocto/sources/meta-arago/meta-arago-extras \
21   /home/melture/Projects/yocto/sources/meta-arago/meta-arago-test \
22   /home/melture/Projects/yocto/sources/meta-qt5 \
23   /home/melture/Projects/yocto/sources/meta-ti/meta-ti-bsp \
24   /home/melture/Projects/yocto/sources/meta-ti/meta-ti-extras \
25   /home/melture/Projects/yocto/sources/meta-arm/meta-arm \
26   /home/melture/Projects/yocto/sources/meta-arm/meta-arm-toolchain  "
27
```

Figure 80: Layers for Yocto Image

Building the Image:

- Execute ‘bitbake core-image-minimal‘ to start the build process.
- Monitor the build process for any errors or missing dependencies.
- Upon completion, locate the built image in the ‘tmp/deploy/images‘ directory.

Testing the Image on QEMU:

- Run the built image on QEMU using ‘runqemu‘ command.
- Verify the boot process and basic functionality.

Testing the Image on BeagleBone AI-64:

- Flash the built image onto an SD card using a tool like ‘dd‘ or Etcher.
  Note: if you are using Virtual Machine, you should mount SD Card or any disk connected to USB in main bar Input>USB and select disk you want to mount.
- Insert the SD card into the BeagleBone AI-64 and power it on.
  • Then use picocom to monitor board behaviour :
  o Connect FTDI cable to laptop and run the following commend to ensure it is read
  o Change directory to /dev
  o And look for ttyUSB0
  o Then prepare Picocom (make sure that baud-rate is 11500)
  o Finally press boot button down and plug power supply don’t leave button until there is booting observed in the laptop
  o Some tests can be made :
- Verify the boot process and basic functionality on the actual hardware.

Troubleshooting and Optimization:

Figure 81: Yocto QEMU Image

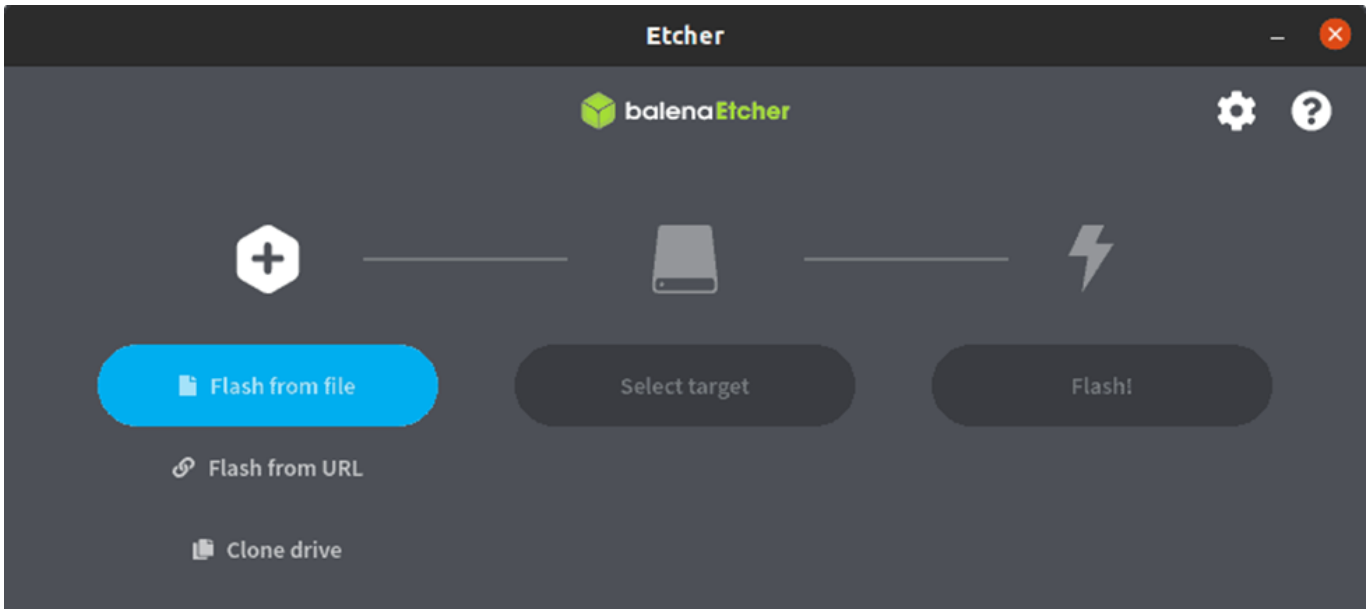


Figure 82: Etcher

- Storage issue : due to large space needed by yocto project there was two options :

  1- use poky tiny : however poky tiny only reduces the size of image produced not the total storage consumed in the host and here is a brief comparsion between poky and poky tiny :

term of Comparison	Poky	Poky Tiny
RootFS(MB)	4	1
Kernel(MB)	7	2.7
Boot time(Sec):	9.1	5.2
Main Components		
Base-Utils	BusyBox	BusyBox
C Library	GLIBC	Musl
Dev manager	: Udev/Eudev	: busybox-mdev
Other	Util-linux	busybox

2 - Resolve storage issues by freeing up disk space as necessary.
as mentioned early, building yocto image require minimum of 80 GB and it is recommended to have 150 GB to build multiple images.

- Handle any branch mismatch and BSP compatibility issues.
  More about the layers used in building Board Image:
  1. **meta**: This is the core layer of the Yocto Project, also known as Poky. It provides essential recipes, configurations, and tools for building Linux distributions using the Yocto build system. This layer includes basic system components and serves as the foundation for customizing Yocto-based distributions.
  2. **meta-poky**: This layer is part of the Poky reference system and provides configurations and recipes specific to the Poky distribution. It includes settings for the default system configuration, package management, and other Poky-specific features.
  3. **meta-yocto-bsp**: This layer contains Board Support Packages (BSPs) for various hardware platforms supported by the Yocto Project. It provides configurations and recipes tailored to specific boards, including kernel configurations, bootloader settings, and device tree files.
  4. **meta-arago/meta-arago-distro**: This layer is part of the meta-arago project, which focuses on supporting Texas Instruments (TI) platforms with the Yocto Project. The meta-arago-distro layer defines distributions specific to TI hardware, including configurations and recipes optimized for TI platforms like the BeagleBone series.
  5. **meta-arago/meta-arago-extras**: This layer provides additional components and configurations beyond the basic meta-arago-distro layer. It may include extra packages, optimizations, and features tailored for TI platforms.
  6. **meta-openembedded/meta-networking**: This layer, part of the OpenEmbedded project, provides recipes and configurations for networking-related software. It includes packages for network protocols, utilities, and services, enabling networking functionality in Yocto-based distributions.
  7. **meta-openembedded/meta-python**: This layer focuses on Python-related software and provides recipes for Python packages, libraries, and utilities. It includes tools for Python development, as well as Python bindings for various libraries and frameworks.
  8. **meta-openembedded/meta-oe**: This is a comprehensive layer within the OpenEmbedded ecosystem and contains a wide range of additional recipes and configurations beyond the core Poky layer. It includes packages for desktop environments, development tools, multimedia software, and much more.
  9. **meta-openembedded/meta-gnome**: This layer specializes in GNOME desktop environment-related packages and configurations. It includes recipes for GNOME desktop

components, applications, and libraries, enabling the creation of Yocto-based distributions with GNOME desktop support.
10. **meta-openembedded/meta-filesystems**: This layer focuses on filesystem-related software and provides recipes for various filesystem utilities, formats, and tools. It includes support for different filesystem types and features, such as encryption and compression.
11. **meta-qt5**: This layer focuses on the Qt framework and provides recipes for Qt libraries, tools, and applications. It enables the development of graphical user interfaces (GUIs) using Qt on Yocto-based distributions.
12. **meta-ti/meta-ti-bsp**: This layer is part of the meta-ti project, which focuses on supporting TI platforms with the Yocto Project. The meta-ti-bsp layer provides Board Support Packages (BSPs) for TI hardware, including configurations and recipes tailored to specific TI platforms.
13. **meta-ti/meta-ti-extras**: Similar to meta-arago/meta-arago-extras, this layer provides additional components and configurations specific to TI platforms. It may include extra packages, optimizations, and features beyond the basic meta-ti-bsp layer.
14. **meta-arm/meta-arm**: This layer provides additional support for ARM-based architectures within the Yocto Project. It includes configurations, recipes, and optimizations for building distributions targeting ARM-based hardware platforms.
15. **meta-arm/meta-arm-toolchain**: This layer focuses on toolchain support for ARM architectures. It provides recipes and configurations for cross-compilation toolchains targeting ARM-based systems, enabling development and deployment of software on ARM platforms.
16. **meta-virtualization**: This layer provides support for virtualization-related software and technologies within the Yocto Project. It includes recipes for virtualization tools, hypervisors, and related components, enabling the creation of Yocto-based distributions for virtualized environments.
image were built based on Krikstone branch and the reason for choosing this branch specifically was : Choosing criteria: the yocto release must line up with poky reference distribution the yocto release must have long term support (upto 2 years) the codename need to line up with yocto version (check second column) latest stable release and/or Long Term Support :

Yocto Releases Comparison

Codename	Yocto Version	Release Date	Current Version	Support Level
Kirkstone	4.0	May 2022	4.0.16	until Apr. 2026
Dunfell	3.1	April 2020	3.1.31	until Apr. 2024

In addition to selecting the release based on queries and Beagle Bone Community :
`https://forum.beagleboard.org/t/bbai-64-yocto-bsp-support/32813/6`
so each layer must be converted to the same layer so that this error is resolved.

- Address ‘do_fetch‘ task failures by ensuring local access to required resources.

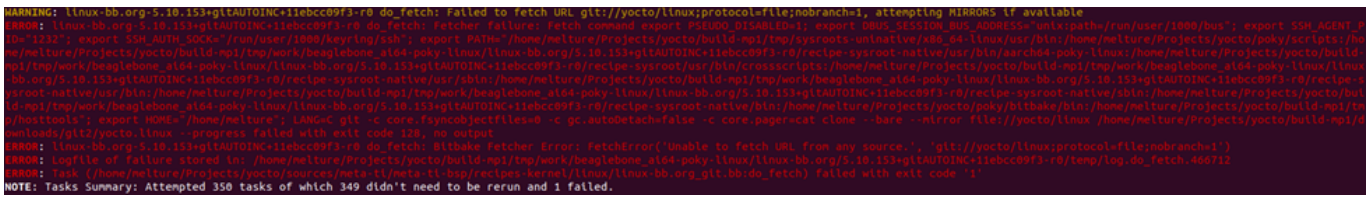


Figure 83: do_fetch error log

to solve it you must make sure that git is configured to allow access to local files using the following command :

git config –global core.longpaths true
and check that this command was done using :
git config –list
Note: Usually `do\_fetch` stuck error is caused by network issue.
Please check your network or change a new network and try in different time.
If you have used yocto to compile successfully before, you can directly copy the relevant files downloaded before to the current project. This way you can skip the `do\_fetch` step.

9.4 Results:

The produced image will have name of core-image-minimal-beaglebone-ai64.wic.xz xz is compressed version of image.

9.4.1 Buildroot Image Results:

The process of building the Linux image using Buildroot was completed successfully for the QEMU . The image was built without significant issues, and the following results were obtained:

- **QEMU:** The Buildroot image booted successfully in the QEMU emulator, demonstrating basic functionality.

9.4.2 Yocto Image Results:

Building the Linux image using Yocto presented several challenges, including disk space limitations and branch mismatches. However, these issues were resolved, and the following results were obtained:

- **QEMU:** The Yocto image booted successfully in the QEMU emulator after addressing initial configuration issues.
- **BeagleBone AI-64:** The Yocto image was built and flashed onto the BeagleBone AI-64. Despite initial do_fetch task failures, the final image booted successfully after resolving local access requirements.

9.4.3 Comparison of Buildroot and Yocto Images:

- **Build Time:** Buildroot images were quicker to build compared to Yocto images, which required more configuration and build time.
- **Customization:** Yocto provided more customization options and granular control over the build process.
- **Performance:** Both images performed comparably in basic tests on QEMU and BeagleBone AI-64, but further testing is required for a detailed performance comparison.

beaglebone.org also provide a pre-built image by looking to the size of pre-built image is about 537 MB while our built image is about 36 MB so the used storage size will reduce to about 6.7% making the proposed image more optimized than the one provided by beaglebone.org.

9.4.4 Analysis of Build Processes:

The build processes for both Buildroot and Yocto demonstrated unique advantages and challenges. Buildroot was straightforward and efficient, making it suitable for quick and simple builds. Yocto, while more complex, offered greater customization and control, which is beneficial for more advanced and specific requirements.

9.4.5 Performance Interpretation:

The performance metrics indicated that both Buildroot and Yocto images performed similarly in basic tests. However, Yocto's additional customization options could lead to better optimization in more complex scenarios. The slightly higher resource usage in Yocto images is a trade-off for its flexibility.

9.4.6 Advantages and Disadvantages:

- **Buildroot:** The main advantage of Buildroot is its simplicity and speed. It is ideal for projects that require a minimal and quick setup. The disadvantage is its limited customization compared to Yocto.
- **Yocto:** Yocto offers extensive customization and is well-suited for complex and specific project requirements. The downside is its complexity and longer build times.

9.4.7 Challenges and Solutions:

Several challenges were encountered during the build process, particularly with Yocto. Disk space limitations were a significant hurdle, requiring careful management of resources. Branch mismatches and do_fetch task failures were resolved through diligent troubleshooting and configuration adjustments.
This project successfully demonstrated the process of building minimal Linux images for the BeagleBone AI-64 and QEMU using both Buildroot and Yocto. The results highlighted the trade-offs between the simplicity and efficiency of Buildroot and the customization and flexibility of Yocto.
Key findings include:

- Buildroot is ideal for quick, simple builds with minimal configuration.
- Yocto offers extensive customization, making it suitable for more complex and specific requirements.
- Both methods produced functional images, with comparable basic performance metrics.

10 Conclusion

The development and implementation of the AI Solution for Driver Monitoring Systems (DMS) has been a comprehensive and multifaceted endeavor, spanning face feature extraction, distraction analysis, drowsiness detection, and the integration of these components into a cohesive application. Utilizing state-of-the-art techniques such as YOLOv8n for face detection, Dlib models for facial landmarks extraction, and robust algorithms for age, gender, and drowsiness analysis, this project has successfully addressed critical aspects of driver safety and attentiveness.

The Software In the Loop (SIL) methodology provided rigorous testing environments. It simulates the hardware, ensuring the reliability and efficiency of the deployed system. The selection of appropriate hardware, such as the IMX219 camera and the BeagleBone AI-64 processing platform, was crucial for achieving the desired performance metrics. The integration with Texas Instruments' TIDL and the optimization of models through techniques like quantization further enhanced the system's real-time capabilities.

Benchmarking and performance evaluation demonstrated the system's ability to operate within the required parameters, providing valuable insights into execution times and frame rate optimization. The deployment of the AI models, both on the edge and cloud, highlighted the benefits and constraints of each approach, guiding future deployment strategies.

Explored the interface to the VCU and the critical role of Application Performance Monitoring (APM) in optimizing system performance. The interface design to the VCU ensures seamless integration and efficient communication within embedded systems, offering real-time insights into application behavior and performance metrics.

The visualization of data through a mobile application developed using Flutter added an important dimension to the project, offering a user-friendly interface for monitoring and interacting with the DMS.

In conclusion, this project represents a significant step forward in the field of driver monitoring systems, combining advanced AI techniques, robust software engineering practices, and careful hardware selection to deliver a comprehensive solution. The knowledge and experience gained through this endeavor lay a strong foundation for future work in optimizing performance using multi-task learning approach and using multiprocessing techniques.

11 Future work

11.1 Optimizing Performance Using Multi-Task Learning Approach

In future work, we propose optimizing the DMS using a Multi-Task Learning (MTL) strategy. MTL is a machine learning approach where related tasks are learned simultaneously, sharing common representations at the encoder layers and then becoming task-specific in the decoder layers as in Fig. 84. This can improve performance compared to using image processing techniques for each task independently or using deep learning models for each task due to computational restrictions. This is particularly relevant for the DMS tasks of detecting drowsiness and distraction, which are interrelated in learning facial features.

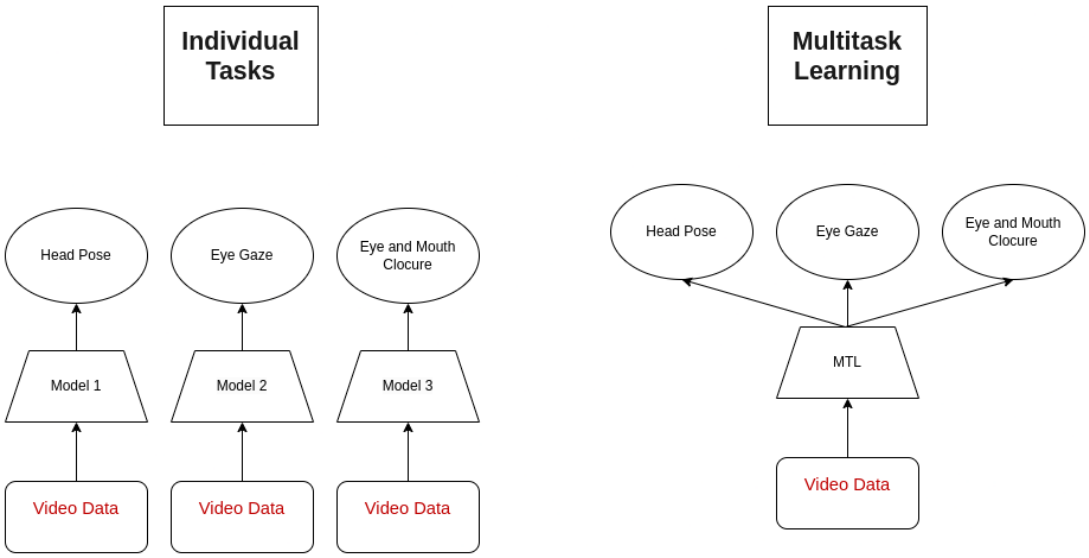


Figure 84: Individual tasks Vs. MTL network

Choosing MTL Backbone

The proposed model will use MobileNet or EfficientNet for feature extraction, given their efficiency and suitability for real-time, resource-constrained environments. Initially, each task will be trained individually using MobileNet to establish baseline performance metrics. These results will then be compared to the performance of the MTL model to evaluate the benefits of the MTL approach.
Furthermore, to handle regression tasks such as estimating age and head pose angles, modifications to the MobileNet architecture will be necessary. Specifically, the decoder layers will be redesigned to support regression outputs, ensuring compatibility with the DMS objectives.

11.2 Optimizing Performance Using Multiprocessing

We already utilized the benefits of multiprocessing within the development phase in the SiL and we intend to continue optimizing the application fps through applying multiprocessing in the deployed application. As we have clear tasks like pre-processing, inference, and post-processing up and running, and each delivers its output to the following stage, we can now provide a dedicated process for the time-consuming tasks such as pre-processing in YoloV5-based applications, especially the resizing stage. Then we can provide a stand-alone worker for the bottleneck while the rest of the application continues the main processing.

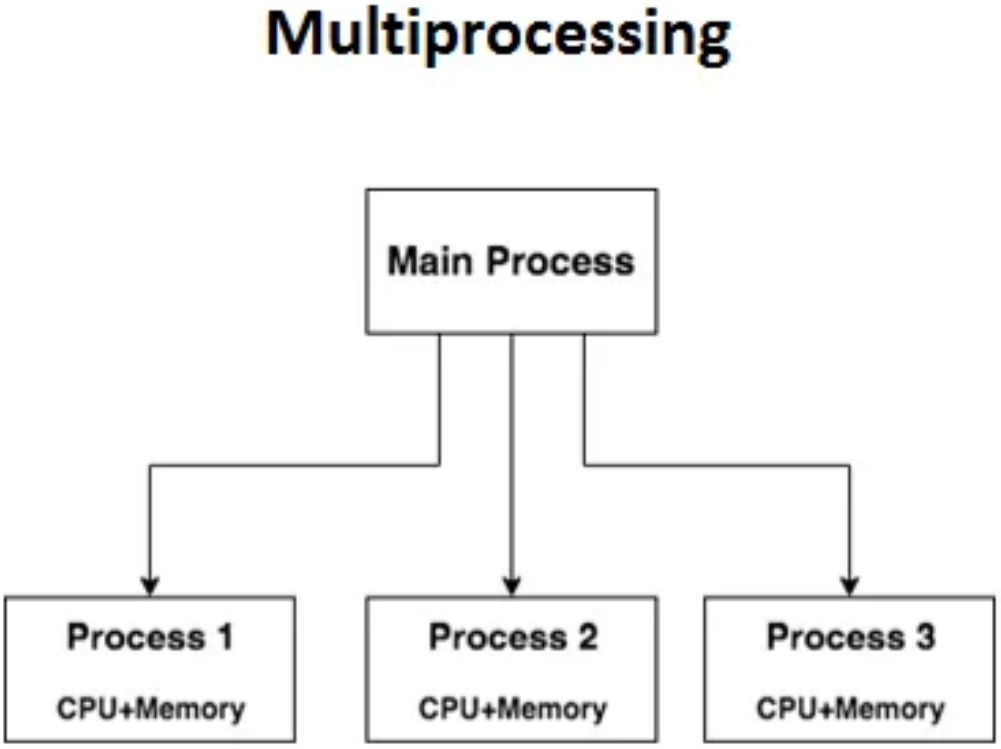


Figure 85: Multiprocessing Task parallelization

References

[1] Y. Abdel-Aziz. "Direct linear transformation from comparator coordinates into object space in close-range photogrammetry". In: *ASP Symposium Proceeding on Close-Range Photogrammetry*. American Society of Photogrammetry. Falls Church, 1971, pp. 1–18.

[2] Carlos Alvarez Casado and Manuel Bordallo Lopez. "Real-time face alignment: evaluation methods, training strategies and implementation optimization". In: *Springer Journal of Real-Time Image Processing* (2021). URL: https://gitlab.com/visualhealth/vhpapers/real-time-facealignment.

[3] Grigory Antipov, Sid Ahmed Berrani, and Jean-Luc Dugelay. "Minimalistic CNN-based ensemble model for gender prediction from face images". In: *Pattern Recognition Letters* 70 (2015), pp. 70–76. DOI: 10.1016/j.patrec.2015.11.011. URL: https://doi.org/10.1016/j.patrec.2015.11.011.

[4] BeagleBoard.org. *BeagleBone AI*. https://www.beagleboard.org/boards/beaglebone-ai-64. Accessed: 2024-06-30. 2024.

[5] Deci. *Quantization and Quantization-Aware Training*. https://deci.ai/quantization-and-quantization-aware-training/. Accessed: 2024-06-26. 2024.

[6] Texas Instruments. *Edge AI Webinar Dec 2021 - YOLOv5*. https://www.ti.com/content/dam/videos/external-videos/en-us/8/3816841626001/6286792047001.mp4/subassets/edgeaiwebinardec2021_yolov5_final.pdf. 2021.

[7] Texas Instruments. *edgeai-yolov5*. https://github.com/TexasInstruments/edgeai-yolov5. 2023.

[8] Texas Instruments. *Foundational Components - TIDL*. https://software-dl.ti.com/processor-sdk-linux/esd/docs/05_00_00_15/linux/Foundational_Components_TIDL.html. 2023.

[9] Vahid Kazemi and Josephine Sullivan. "One millisecond face alignment with an ensemble of regression trees". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2014, pp. 1867–1874. DOI: 10.1109/CVPR.2014.241. URL: https://doi.org/10.1109/CVPR.2014.241.

[10] Dohun Kim et al. "Real-time driver monitoring system with facial landmark-based eye closure detection and head pose recognition". In: *Scientific Reports* 13 (2023), Article 18264. DOI: 10.1038/s41598-023-44955-1. URL: https://doi.org/10.1038/s41598-023-44955-1.

[11] Davis King. *dlib-models repository*. GitHub. n.d. URL: https://github.com/davisking/dlib-models.

[12] Z. Lai. *Cameras in Advanced Driver-Assistance Systems and Autonomous Driving Vehicles*. Ed. by Y. Li and H. Shi. Singapore: Springer, 2022. DOI: 10.1007/978-981-19-5053-7_7. URL: https://doi.org/10.1007/978-981-19-5053-7_7.

[13] Antoine Lame. *GazeTracking repository*. GitHub. n.d. URL: https://github.com/antoinelame/GazeTracking.

[14] Learn OpenCV. *Head-pose-estimation-using-opencv*. 2023. URL: https://learnopencv.com/head-pose-estimation-using-opencv-and-dlib/.

[15] Mowshon. *Age and gender prediction repository*. GitHub. n.d. URL: https://github.com/mowshon/age-and-gender.

[16] Matjaz Muc. *Frame-Level Driver Drowsiness Detection (FL3D)*. Kaggle. n.d. URL: https://www.kaggle.com/datasets/matjazmuc/frame-level-driver-drowsiness-detection-fl3d.

[17] ONNX Community. *ONNX: Open Neural Network Exchange*. https://onnx.ai/onnx/. Accessed: 2024-06-30. 2024.

[18] ONNX Runtime. *ONNX Runtime Documentation*. https://onnxruntime.ai/docs/. Accessed: 2024-06-26. 2024.

[19] OpenCV. *How to verify the accuracy of solvePnP return values?* OpenCV Q&A Forum. n.d. URL: https://answers.opencv.org/question/149759/how-to-verify-the-accuracy-of-solvepnp-return-values/.

[20] M. Punke et al. *Automotive Camera (Hardware)*. Ed. by H. Winner et al. Cham: Springer, 2016. DOI: 10.1007/978-3-319-12352-3_20. URL: https://doi.org/10.1007/978-3-319-12352-3_20.

[21] Zakariya Qawaqneh, Arafat Abu Mallouh, and Buket D. Barkana. "Deep convolutional neural network for age estimation based on VGG-Face model". In: *arXiv preprint arXiv:1709.01664* (2017). URL: https://arxiv.org/abs/1709.01664.

[22] Roboflow. *Roboflow*. n.d. URL: https://roboflow.com/.

[23] Ananya Bhavana D. S. and N. Sivakumar. "Real-Time Driver Drowsiness Detection Using Eye Closure and Yawn Detection using Facial Landmarks". In: *International Journal of Creative Research Thoughts* (2021). URL: https://ijcrt.org/papers/IJCRT2106033.pdf.

[24] Christos Sagonas et al. "300 Faces In-the-Wild challenge: Database and results". In: *Image and Vision Computing* 47 (2016), pp. 3–18. URL: https://ibug.doc.ic.ac.uk/resources/facial-point-annotations/.

[25] Sony Corporation. *IMX219PQ CMOS Image Sensor Datasheet*. 2019. URL: https://www.opensourceinstruments.com/Electronics/Data/IMX219PQ.pdf.

[26] Tereza Soukupova and Jan Cech. "Real-Time Eye Blink Detection using Facial Landmarks". In: *21st Computer Vision Winter Workshop*. Ed. by Luka Cehovin, Rok Mandeljc, and Vitomir Struc. Rimske Toplice, Slovenia, 2016, pp. 1–10. URL: https://www.semanticscholar.org/paper/Real-Time-Eye-Blink-Detection-using-Facial-Soukupov%C3%A1-Cech/4fa1ba3531219ca8c39d8749160faf1a877f2ced.

[27] Texas Instruments. *Edge AI Studio*. https://dev.ti.com/edgeaistudio/. Accessed: 2024-06-26. 2024.

[28] Texas Instruments. *Repository to host GStreamer plugins for TI's EdgeAI class of devices*. 2024. URL: https://github.com/TexasInstruments/edgeai-gst-plugins.

[29] Texas Instruments. *TDA4VM Technical Reference Manual*. 2023. URL: https://www.ti.com/lit/zip/spruil1.

[30] Texas Instruments. *TIDL User Guide*. https://software-dl.ti.com/jacinto7/esd/processor-sdk-rtos-jacinto7/07_03_00_07/exports/docs/tidl_j7_02_00_00_07/ti_dl/docs/user_guide_html/index.html. Accessed: 2024-06-26. 2024.

[31] Ultralytics. *Ultralytics repository*. GitHub. n.d. URL: https://github.com/ultralytics/ultralytics.

[32] WasabiFan. *tidl-yolov5-custom-model-demo*. https://github.com/WasabiFan/tidl-yolov5-custom-model-demo. 2023.

[33] Shuo Yang et al. "WIDER FACE: A face detection benchmark". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016. URL: http://shuoyang1213.me/WIDERFACE/.

12 Appendix

12.1 System Block Diagram in The SIL

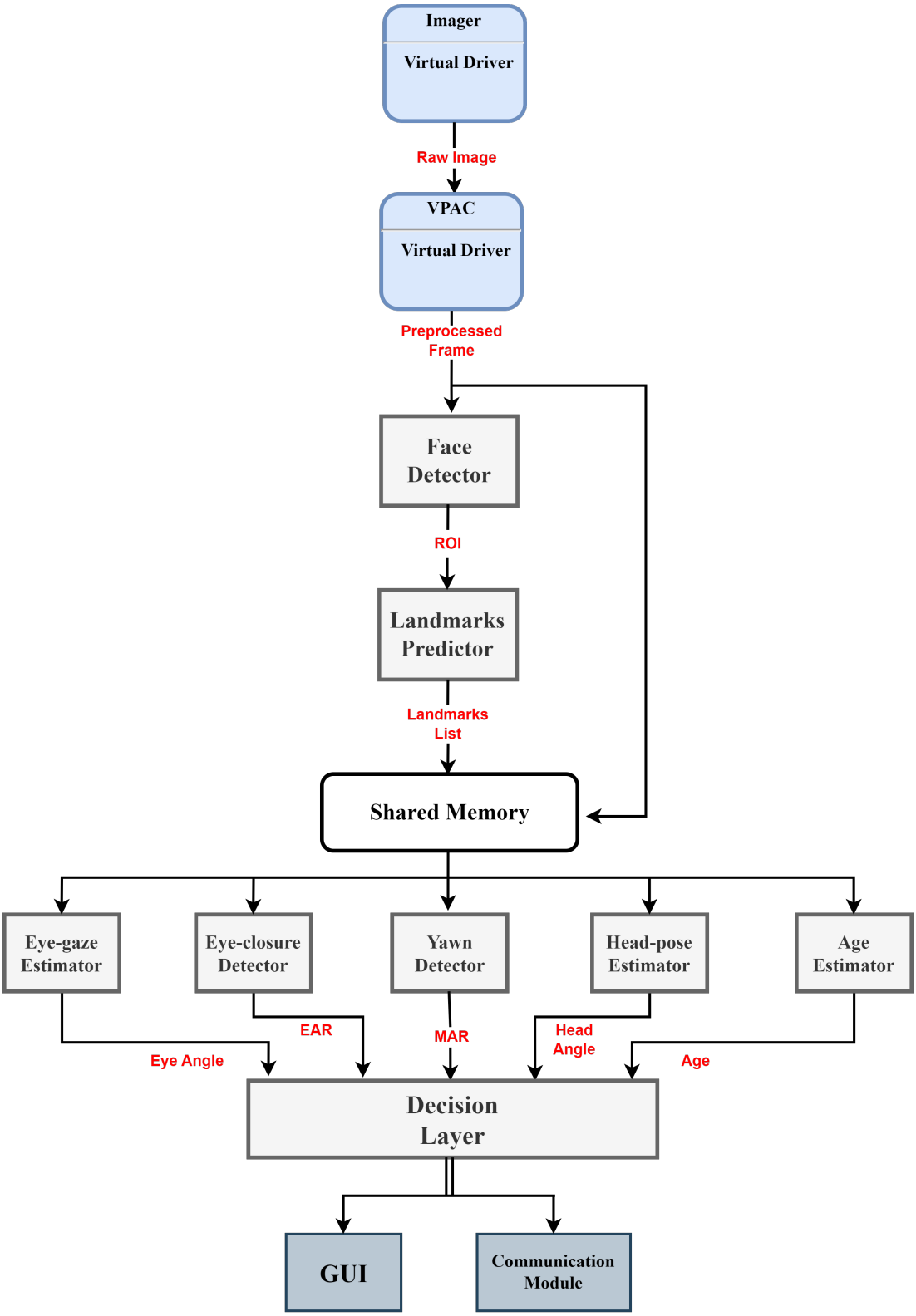


Figure 86: Full system block diagram in the SIL

12.2 Risk Analysis in Hardware Selection

Risk ID	Risk Description	Impact Descrption	Impact	Probability	Risk	Mitigating Actions
R1	The selected MCU (SK-TDA4VM) may not be able to run 4 DL models simultaneously or efficiently.	The system performance may degrade, affecting the accuracy and reliability of the driver state detection.	High	Medium	High	- Benchmark the selected MCU with different DL models and compare the results with other MCUs or development kits. - Optimize the DL models using techniques such as quantization, pruning, and clustering. - Use external accelerators or FPGA if needed.
R2	The selected MCU may not have enough interfaces or compatible interfaces to communicate with the network, the system diagnostics and the other car systems.	The system functionality may be limited, affecting the data acquisition, transmission, and integration.	Low	Low	Low	- Review the specifications and datasheets of the selected MCU and the other components. - Use adapters or converters if needed. - Use alternative interfaces or protocols if possible.
R3	The selected MCU may not comply with the automotive standards such as functional safety.	The system safety may be compromised, affecting the driver and the passengers.	Low	High	Medium	- Review the standards and regulations for the automotive industry. - Use certified components and tools.
R4	The selected MCU may not have compatible interfaces to communicate with the external modules.	The system functionality may be limited if any addational processing is needed	Medium	Medium	Medium	- Review the specifications and datasheets of the selected MCU and the other components.

Table 6: Risk analysis for hardware selection

Contacts

If you have any questions, feedback, or require further information about our Driver Monitoring System project, please feel free to reach out to us.
You can contact us through the following:

Nada Atia Eid Atia	nada.atia.23@gmail.com
Mariam Ahmed Fathy	mariam.zaid1907@gmail.com
Amr Ramadan Agamy	amrramadan322@gmail.com
AbdElrhman Mamdouh Khalil	abdelrhmanm2012@gmail.com
Ali Emad Abdellatif Abdo	aliemadabdellatif@gmail.com
Ahmed Mohamed Saeed Elgohary	ahmedm.saeedelgohary@gmail.com
Ahmed Mohamed Roshdi	ahmedroshdi79@gmail.com
Mostafa Sayed Ahmed Taha	mostafa.sayed01028@gmail.com

Additional Resources

For more resources and information about our project, please follow this link or scan the QR code provided.