Intelligent Ride Acceptance System Using Alcohol and Fatigue Detection for Drivers

Khush Kuhad[1], Bhavika Tanwar[2], Manish Kumar Jain K[3], Mehek Jain[4], Yashika[5]
[1,2,3,4] *School of CS and IT JAIN (Deemed-to-be-University) Bangalore, India*
[5]*School of Humanities and Social Sciences JAIN (Deemed-to-be-University)* Bangalore, India

***Abstract*—In this paper, we propose an intelligent ride acceptance system that enhances driver and passenger safety by integrating alcohol and fatigue detection mechanisms into the ride-hailing process. Before a ride is accepted, the system performs a breath alcohol test using an MQ-3 gas sensor interfaced with a Raspberry Pi and simultaneously evaluates driver alertness via a camera module and real-time computer vision analysis for drowsiness. A Flask-based API transmits the sensor data and detection results to a cab company's backend, allowing verification of the driver's sobriety and alertness in real-time. We present the design and implementation of the system, including hardware integration, software algorithms for alcohol sensing and eye-closure detection, and the communication protocol for data exchange. Experimental simulations demonstrate that the system can reliably block ride requests if alcohol is detected and can alert drowsy drivers with a buzzer, logging such events for review. This approach has the potential to reduce incidents related to drunk or drowsy driving, improving overall ride safety and service quality.**

***Index Terms*—Driver Monitoring, Alcohol Detection, Drowsiness Detection, Raspberry Pi, IoT, Intelligent Transportation Safety, Ride-hailing Safety**

I. INTRODUCTION

Drowsy driving and drunk driving are major contributors to road accidents globally. Fatigue-related lapses in attention can slow driver reaction times and impaired decision-making, leading to severe accidents. Alcohol impairment likewise remains a leading cause of traffic fatalities, accounting for roughly 30% of traffic-related deaths in the United States in 2020. These statistics underscore the urgent need for in-vehicle systems that can detect and mitigate driver impairment in real-time. Traditional approaches to curb drunk driving, such as ignition interlocks, require drivers to pass breathalyzer tests before starting the car. Similarly, driver assistance systems for fatigue monitoring often use cameras to observe signs of drowsiness like eye closure or yawning. However, in the context of ride- hailing services or cab fleets, there is a pressing need for an integrated solution that ensures a driver's sobriety and alertness at the moment of accepting a ride and continuously during the trip. Such a system would not only enhance passenger safety but also improve service quality by preventing trips with impaired drivers.

This paper presents an "Intelligent Ride Acceptance System" that combines alcohol detection and fatigue (drowsiness) detection to verify driver fitness before and during rides. In our proposed system, whenever a new ride request is received, the driver is prompted to perform a breath alcohol test on an MQ-3 gas sensor attached to a Raspberry Pi. Concurrently, or once the ride starts, a camera feeds an image-processing algorithm (built with OpenCV and dlib) that monitors the driver's eyes for signs of drowsiness. If alcohol is detected above a threshold, the ride request is automatically rejected and flagged as a safety violation. If the driver is sober, the ride proceeds, but the drowsiness detection subsystem remains active throughout the trip. If the driver begins to fall asleep (indicated by eye closures exceeding a safe duration), an audible buzzer alarm is triggered to alert the driver, and an event is recorded with a timestamp. All data — the result of the alcohol test and any drowsiness alerts during the ride — are transmitted via a Flask-based API to the cab company's server. This allows the company to log safety incidents and, if necessary, intervene or review ride quality after the fact.

The remainder of this paper is organized as follows: Section II reviews related work on driver alcohol

and fatigue detection. Section III describes the overall system architecture and hardware components. Section IV details the implementation of sensor interfacing, computer vision algorithms, and the cloud API. Section V presents simulation results and system performance observations. Section VI discusses the impact, limitations, and real-world deployability of the system. Finally, Section VII concludes the paper and outlines future enhance- ments.

II. LITERATURE REVIEW AND RELATED WORK

Ensuring driver fitness through technological means has been a subject of growing research interest, particularly with the increasing demand for intelligent transportation systems and the proliferation of ride-hailing services. Prior literature has primarily addressed **alcohol detection** and **fatigue monitoring** as independent safety domains. However, few systems integrate both within a unified architecture, especially in the context of real-time fleet or ride acceptance management.

A. Alcohol Detection Systems

Alcohol detection in vehicular contexts often relies on MQ- series gas sensors, particularly the MQ-3, due to its sensitivity to ethanol vapor and ease of integration with microcontrollers. Charniya and Nair [1] implemented an alcohol detection system using an MQ-3 sensor and microcontroller that dis- abled ignition upon alcohol detection. This approach has been extended in many systems to include buzzer alerts and GSM- based notifications to external parties [2]. While effective for vehicle lockout, such systems are rarely integrated with real-time communication infrastructures or centralized fleet monitoring platforms.

B. Driver Fatigue and Drowsiness Detection

Fatigue detection research has heavily leaned on **non- intrusive vision-based techniques**. A widely adopted metric is the **Eye Aspect Ratio (EAR)**, proposed by Soukupova´ and Cˇ ech [3], which calculates eye openness based on facial landmarks. When the EAR drops below a threshold for a de- fined time, it signals eye closure or potential microsleep. This method is widely considered robust for real-time drowsiness detection and has been implemented in embedded systems like Raspberry Pi with OpenCV and dlib [4].

Further studies have expanded on facial cues, incorporating yawning, blink rate, and head pose detection using Haar cascades and convolutional neural networks (CNNs) [5][6]. However, many of these models require high computational re- sources, making them less practical for lightweight embedded deployment. Additionally, most implementations offer local alerts (e.g., buzzer alarms) without centralized monitoring or long-term ride event logging.

C. Integrated and IoT-Based Approaches

Recent works have sought to **fuse alcohol and drowsiness detection** with Internet of Things (IoT) architectures. Patel et al. [7] proposed an IoT-enabled system for detecting alcohol presence and eye closure, sending alerts via GSM. Uma and Eswari [8] advanced this approach by incorporating machine learning classifiers to improve detection accuracy and used cloud storage for alert logs. A similar approach by Prasad et al. [9] added vehicle interlocks and seatbelt verification but lacked REST API-based integration for fleet-wide scalability. Despite these efforts, few systems are explicitly designed for **ride-hailing platforms**, where safety validation must occur at the **moment of ride acceptance** and be monitore throughout the trip. Most existing systems are reactive (i.e., triggering only after the ride has begun or a hazard is detected), rather than proactive in evaluating driver readiness.

D. Gaps and Motivation

From the reviewed literature, it is evident that while in- dividual components (alcohol sensing or fatigue monitoring) are well-researched, their **combined, real-time, and API- accessible integration for ride dispatch logic** remains under- explored. Furthermore, the current systems often lack modularity, cloud interoperability, and timestamped data handling, which are critical for fleet compliance and accountability.

This paper addresses these gaps by proposing a **unified, API-enabled intelligent ride acceptance system** that enforces alcohol testing before a ride is accepted and actively monitors fatigue throughout the ride. The results are shared in real- time with cab company servers using a Flask-based RESTful interface, enabling timely interventions and post-ride analysis.

III. SYSTEM DESIGN

A. System Architecture Overview

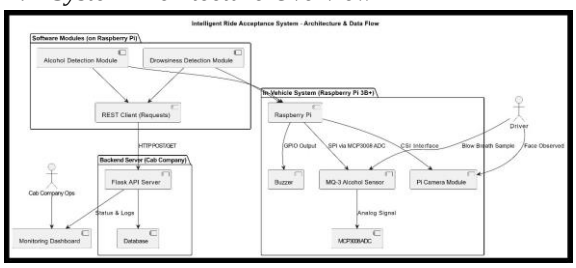

Fig. 1. System architecture and dataflow

The proposed system consists of an in-vehicle embedded module and a backend server. Figure 1 illustrates the overall architecture and data flow of the system. The in-vehicle system is built around a Raspberry Pi 3B+ microcomputer, which interfaces with a set of sensors and actuators: an MQ-3 gas sensor (for breath alcohol detection), a Pi Camera module (for monitoring the driver's face), and a buzzer (for alerting the driver). An analog-to-digital converter (ADC) chip (MCP3008) is used to read the analog output of the MQ-3 sensor, since the Raspberry Pi's GPIO pins do not include analog input channels. The Pi continuously runs two parallel processes or threads: one for alcohol sensing (triggered on-demand when a ride request arrives) and one for drowsiness detection (running continuously while driving).

When a ride request is initiated (e.g., through the driver's dispatch app or vehicle's connected interface), the system temporarily requires the driver to blow into the MQ-3 sensor. The MQ-3 sensor's output voltage varies with the concentration of alcohol in the breath sample. The Raspberry Pi reads the sensor output via the MCP3008 ADC and interprets the value to determine if alcohol is present beyond a calibrated threshold. In our design, this threshold is set based on baseline tests to correspond to approximately 0.04 mg/L of breath alcohol content (which is roughly around the lower limit of detection and well below many legal intoxication levels). If the reading exceeds the threshold (indicating the presence of alcohol), the Pi flags the driver as "drunk" for that ride request. If the reading is below the threshold, the driver is marked "clear." In Section IV we detail the calibration and code logic for this decision.

Simultaneously, or once the ride is accepted (if the driver was clear of alcohol), the camera-based monitoring subsystem begins analyzing the driver's face for signs of drowsiness. Using either a live video feed or periodically captured frames (e.g., 1 frame every 0.5 seconds), the system applies face detection and then eye detection to compute the driver's Eye Aspect Ratio (EAR) in real-time. If the EAR falls below a predefined threshold (indicating the eyes are nearly closed) for a sustained period (e.g., 1–2 seconds), the system concludes that the driver is drowsy or microsleeping. At that moment, an alert is generated: the Raspberry Pi drives a buzzer connected to one of its GPIO outputs, causing an audible alarm in the vehicle to awaken the driver. The event is timestamped and recorded.

All detection results (alcohol test outcome and any drowsiness events) are transmitted to the cloud backend via a RESTful API built with Flask. The Pi, which has internet connectivity (through Wi-Fi or a cellular hotspot), sends an HTTP POST request to the Flask API endpoint with the alcohol test result as soon as it is obtained. The API, running on the cab company's server, logs this result in a database and responds to the Pi or the dispatch system indicating whether the ride is allowed to proceed. For drowsiness, the Pi can either send events in real-time (each time the buzzer triggers) or batch-upload them at the end of the ride. In our implementation, we chose to send an update for each drowsiness alert event to the server. The Flask API provides a GET endpoint as well, which authorized clients (e.g., the cab company's dispatch center or monitoring dashboard) can use to query the status of a given ride or driver. This way, the company can, for example, check if a driver was marked clear or drunk for a ride request, or review how many drowsiness alerts were triggered during a particular trip.

B. Hardware Components and Integration

The hardware components used in the system and their key specifications are summarized in Table I. The Raspberry Pi 3B+ serves as the central controller, chosen for its balance of computational power (it can run the vision algorithms) and interfacing capabilities (multiple GPIO for sensors and network connectivity for API communication). The MQ-3 sensor is a tin-oxide semiconductor gas sensor that is sensitive to ethanol; it requires a 5V supply to heat the sensing element and outputs an analog voltage proportional to the ethanol concentration in the air. We interfaced the MQ-3

through an MCP3008 10-bit ADC, which communicates with the Pi over the SPI bus. The Pi Camera module (v2) connects via the CSI camera port on the Raspberry Pi board; it provides 8-megapixel images and supports video capture at 1080p 30fps, more than sufficient for our needs (we typically use a lower resolution like 640x480 for faster processing). A piezoelectric buzzer is wired to one of the Pi's digital output GPIO pins through a transistor driver (since the GPIO can only source limited current). The buzzer is powered by the 5V line and can produce 85 dB sound, enough to alert the driver in a noisy car environment.

TABLE I: KEY HARDWARE COMPONENTS AND SPECIFICATIONS

Component	Model/Type	Key Specifications
On-board Computer	Raspberry Pi 3B+	1.4 GHz 64-bit Quad- core, 1 GB RAM, Wi-Fi/Ethernet, 40 GPIO pins
Camera	Raspberry Pi Camera Module v2	8 MP sensor, supports 1080p@30fps video, CSI-2 interface to Pi
Alcohol Sensor	MQ-3 Gas Sensor	Detects ethanol vapor (approx. 0.05–10 mg/L range); 5V heater; analog output
ADC	MCP3008	10-bit, 8-channel ADC; SPI interface (up to 200 ksps); powered at 3.3V
Alert Device	Buzzer (Piezo)	5V buzzer, 85 dB sound output @10cm; driven via GPIO (with transistor)

For robust integration, the MQ-3 sensor was mounted on the dashboard with a plastic tube to serve as a mouthpiece for the driver to blow into, ensuring a focused breath sample for measurement. The MQ-3 requires a warm-up time (on the order of 20 seconds to a minute) to heat the sensor for stable readings; in our system, the sensor is powered continuously while the car is on, so that it remains ready. The Pi Camera was positioned facing the driver's face (either mounted on the windshield or the instrument cluster) such that the driver's eyes and a portion of the face are within the field of view even when turning slightly. We ensured moderate lighting conditions for the camera; in low-light (night) conditions, the camera may need IR illumination or the system can use an IR-sensitive camera to function (this was outside our current scope).

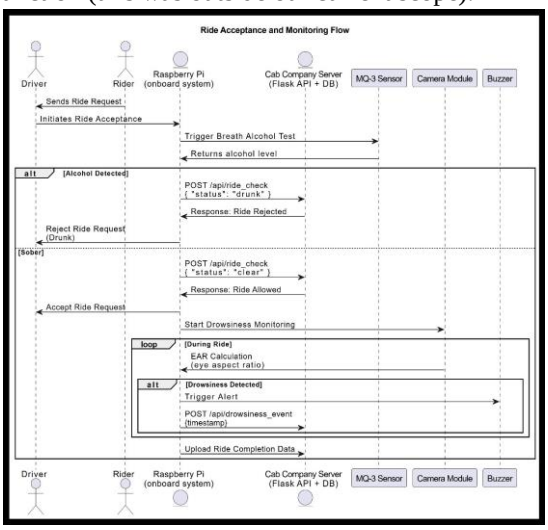

Fig. 2. Ride acceptance and monitoring process

C. Alcohol Detection Module

The Raspberry Pi and its peripherals are powered through the vehicle's 12V supply using a DC-DC converter (5V output for the Pi and sensors). All components were enclosed or securely mounted to withstand vibrations and not distract the driver. The hardware wiring was carefully done to avoid loose connections, given the automotive environment. Figure 2 shows a simplified diagram for the ride acceptance and monitoring process. This highlights how the system transitions from the alcohol check at ride start to continuous monitoring and the eventual ride completion where data is uploaded to the backend.

IV. IMPLEMENTATION

A. Alcohol Detection Module

The alcohol detection module is implemented using the MQ-3 sensor and the MCP3008 ADC connected to the Rasp- berry Pi's SPI interface. We utilized the Python library spidev to communicate with the MCP3008. The MQ-3 was wired to the ADC such that the sensor's analog output is connected to channel 0 of the MCP3008. A load resistor of 200kΩ was used with the sensor to form a voltage divider, which is a typical configuration from the MQ-3 datasheet for measuring ethanol concentration. The sensor outputs a higher voltage

when higher alcohol concentration is present in its vicinity. Figure 3 shows Hardware circuit setup for alcohol detection with MQ-3 sensor and raspberry pie.

Listing 1 shows a snippet of the Python code used to read the MQ-3 sensor value and determine if it exceeds the threshold. We perform a single-ended read from channel 0 of the ADC by sending the appropriate control bits over SPI (the MCP3008 expects a three-byte command and returns a 10-bit value). The raw ADC value (0–1023) is then converted

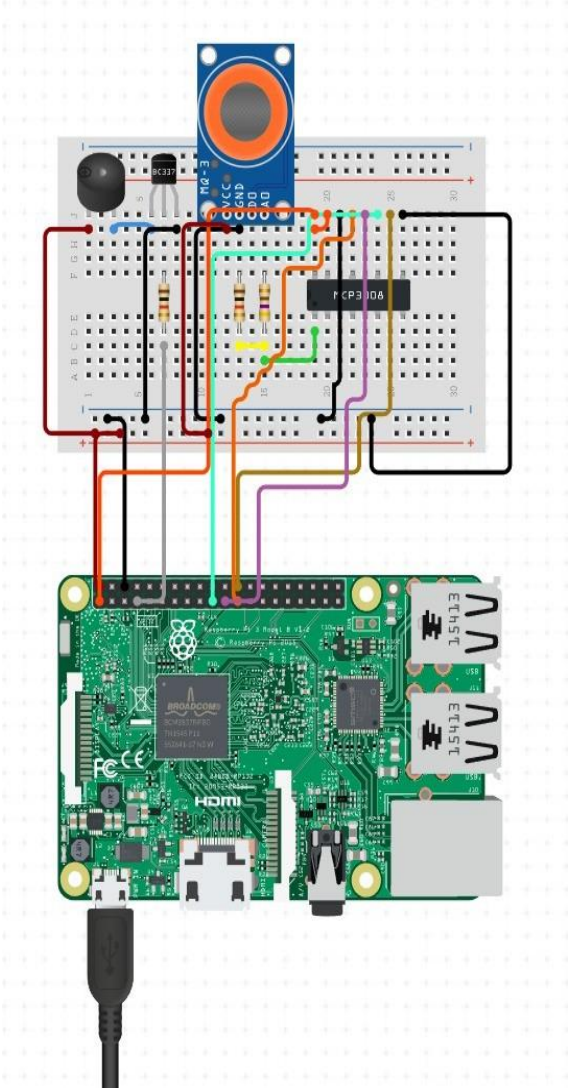

Fig. 3. Hardware circuit setup for alcohol detection using MQ-3 sensor and MCP3008 ADC interfaced with Raspberry Pi 3B+

to a voltage (for logging) or directly compared to a pre- set threshold. This threshold was obtained empirically by sampling the sensor: we recorded the ADC values when the driver had not consumed any alcohol (baseline ambient reading) and when exposed to a moderate amount of alcohol vapor (from mouthwash or similar). In clean air, the MQ-3 output is near zero (ADC reading in the tens), and in presence of alcohol, it rises significantly. We set a threshold (e.g., around 300 out of 1023, which corresponded roughly to 0.04 mg/L ethanol in breath in our calibration) to trigger the "alcohol detected" state.

B. Alcohol Detection

Alcohol detection uses an MQ-3 sensor via MCP3008 ADC.

```
import spidev
spi = spidev.SpiDev()
spi.open(0,0)
spi.max_speed_hz=1350000

def read_adc(channel):
   r = spi.xfer2([1, (8+channel)<<4, 0])
   result = ((r[1]&3)<<8) | r[2]
   return result

threshold = 300
value = read_adc(0)
if value > threshold:
   alcohol_detected=True
else:
   alcohol_detected=False
```

Listing 1. Alcohol Detection Python Code

When a ride request comes in, the system triggers a reading as shown above. In practice, the driver would blow into the sensor for a second or two, and we continuously read from the ADC during that interval (e.g., taking multiple samples over 2 seconds). We then take the maximum or average of those readings to account for the sensor's response time and ensure we capture the peak alcohol concentration. A short delay (on

the order of a few seconds) is used to allow the sensor to clear, and the reading process can be repeated if needed for confirmation. In our tests, a single breath was enough to get a stable reading. If alcohol_detected is True, the Pi immediately sends a message to the backend API (see Section IV-C) indicating a failed alcohol test. The ride acceptance is then aborted – this could be implemented in a user interface as an automatic decline of the ride request with a note that the driver is not fit, or simply by not allowing the "Start trip" action to be executed. Conversely, if no alcohol is detected, a message is sent indicating the driver is clear, and the ride can proceed as normal.

C. Drowsiness Detection Module

The drowsiness detection component uses the OpenCV library for image capture and processing, along with the dlib library's facial landmark detector. We chose the landmark-based approach to leverage the EAR metric for detecting eye closure. Before running the detection, we load a pre-trained facial landmark model (such as the 68-point shape predictor from dlib, which is freely available). The system then continuously captures frames from the Pi Camera. For efficiency, we resize the frames to 320x240 or 640x480 resolution and convert them to grayscale for processing. Face detection is performed on each frame; we used OpenCV's Haar Cascade classifier (e.g., haarcascade_frontalface_default.xml) for initial face localization due to its speed on the Pi. Once a face rectangle is obtained, we apply the dlib shape predictor to get the coordinates of facial landmarks (points around eyes, nose, mouth, etc.). From these landmarks, we extract the coordinates corresponding to the outline of each eye. Let the six 2D landmark coordinates for one eye be p1, . . . , p6 as shown in Figure 4 (inset). The EAR is calculated as:

$$\mathrm{EAR} = \frac{\|p_2 - p_6\| + \|p_3 - p_5\|}{2 \cdot \|p_1 - p_4\|},$$

which is essentially the ratio of the vertical eye opening to the horizontal eye width. When the eye is open, EAR stays around a typical value (usually > 0.25); when the eye closes, the numerator distances shrink and EAR drops towards 0.

We set an EAR threshold (EARth) of 0.25 based on published recommendations and our own trial, and a time threshold of 1 second. The system keeps a rolling counter of how many consecutive frames the EAR has been below 0.25. If this duration exceeds 1 second (which at 10 FPS corresponds to 10+ frames), the driver is deemed to be drowsy (likely falling asleep). We then activate the buzzer as an immediate alert. Listing 2 shows the core logic of computing EAR and detecting drowsiness in our Python code.

```
import cv2
import numpy as np
import dlib
import os
import sys
from imutils import face_utils
import RPi.GPIO as GPIO
import time
import requests
import datetime

BUZZER_PIN = 18
GPIO.setmode(GPIO.BCM)
GPIO.setup(BUZZER_PIN, GPIO.OUT)
GPIO.output(BUZZER_PIN, GPIO.LOW)
API_URL =
    "http://<your_api_server_ip>:5000/drowsiness"

def beep(duration=0.2):
   GPIO.output(BUZZER_PIN, GPIO.HIGH)
   time.sleep(duration)
   GPIO.output(BUZZER_PIN, GPIO.LOW)

def compute(ptA, ptB):
   return np.linalg.norm(ptA - ptB)

def blinked(a, b, c, d, e, f):
   up = compute(b, d) + compute(c, e)
   down = compute(a, f)
   if down == 0:
      return 0
   ratio = up / (2.0 * down)
   if ratio > 0.25:
      return 2
   elif 0.21 < ratio <= 0.25:
      return 1
   else:
      return 0

def send_status_to_api(driver_id, status):
   payload = {"driver_id": driver_id, "status":
       status, "timestamp":
       datetime.datetime.now().strftime('%Y-%m-%d
       %H:%M:%S')}
   try:
      response = requests.post(API_URL,
          json=payload)
      print("API response:", response.json())
   except Exception as e:
      print("Error sending data to API:", e)

def main():
   predictor_path =
       "shape_predictor_68_face_landmarks.dat"
   if not os.path.exists(predictor_path):
      print("Error: Predictor file not found.")
      sys.exit(1)
   detector = dlib.get_frontal_face_detector()
   predictor = dlib.shape_predictor(predictor_path)
   cap = cv2.VideoCapture(0)
   if not cap.isOpened():
      print("Error: Could not open video capture
          device.")
      sys.exit(1)
   sleep_count = 0
   drowsy_count = 0
   active_count = 0
   status = ""
   color = (0, 0, 0)
   last_beep_time = 0
   beep_interval = 3
   driver_id = "cab_driver_001"
   try:
      while True:
         ret, frame = cap.read()
         if not ret:
            print("Error: Failed to capture frame
                from camera.")
            break
         gray = cv2.cvtColor(frame,
             cv2.COLOR_BGR2GRAY)
         faces = detector(gray)
         face_frame = frame.copy()
         for face in faces:
            x1, y1, x2, y2 = face.left(),
                face.top(), face.right(),
                face.bottom()
            cv2.rectangle(face_frame, (x1, y1),
                (x2, y2), (0, 255, 0), 2)
            landmarks = predictor(gray, face)
            landmarks =
                face_utils.shape_to_np(landmarks)
            left_blink = blinked(landmarks[36],
                landmarks[37], landmarks[38],
                landmarks[41], landmarks[40],
                landmarks[39])
            right_blink = blinked(landmarks[42],
                landmarks[43], landmarks[44],
                landmarks[47], landmarks[46],
                landmarks[45])
```

```
            if left_blink == 0 or right_blink == 0:
                sleep_count += 1
                drowsy_count = 0
                active_count = 0
                if sleep_count > 6:
                    status = "SLEEPING !!!"
                color = (255, 0, 0)
            elif left_blink == 1 or right_blink ==
                 1:
                sleep_count = 0
                active_count = 0
                drowsy_count += 1
                if drowsy_count > 6:
                    status = "Drowsy !"
                    color = (0, 0, 255)
                    if time.time() - last_beep_time
                         > beep_interval:
                        beep()
                        last_beep_time = time.time()
                        send_status_to_api(driver_id,
                             status)
            else:
                drowsy_count = 0
                sleep_count = 0
                active_count += 1
                if active_count > 6:
                    status = "Active :)"
                color = (0, 255, 0)
            cv2.putText(frame, status, (100, 100),
                 cv2.FONT_HERSHEY_SIMPLEX, 1.2,
                 color, 3)
            for n in range(0, 68):
                (x, y) = landmarks[n]
                cv2.circle(face_frame, (x, y), 1,
                     (255, 255, 255), -1)
        cv2.imshow("Frame", frame)

        cv2.imshow("Result of detector",
             face_frame)
        if cv2.waitKey(1) & 0xFF == 27:
            break
    except KeyboardInterrupt:
        print("Interrupted by user.")
    finally:
        cap.release()
        cv2.destroyAllWindows()
        GPIO.cleanup()

if __name__ == '__main__':
    main()
```

Listing 2. Drowsiness Detection via Eye Aspect Ratio (Python)

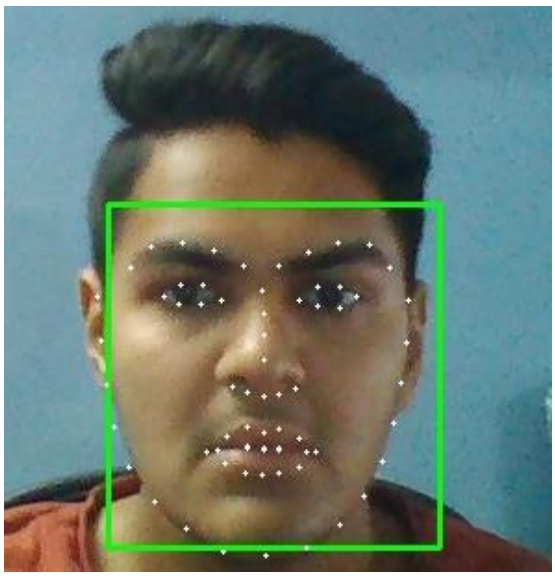

Fig. 4. Plotting of 68 face points to detect drowsiness based on eye movemennt

(The timestamp here is a Unix epoch or any agreed format; one could also send a formatted date-time string or other data like GPS location if needed.)

In this snippet, we use a Haar cascade for face detection (note: in practice, one could use dlib.get_frontal_face_detector() for potentially better accuracy at the cost of speed). We then find the indices for left and right eye landmarks using a mapping provided by the face_utils module from imutils (which defines a dictionary of landmark index ranges for facial features). We compute EAR for both eyes and take their average as the final EAR value (averaging helps when one eye is slightly occluded or not detected clearly). The logic then checks if EAR is below threshold; if yes, it increments the counter, otherwise resets it. When the counter exceeds FRAME_THRESH, a drowsiness event is identified. We turn the buzzer on and consider the driver drowsy. The function log_event() (not fully shown) is a placeholder that would handle storing the timestamp (and possibly other context, like maybe the EAR value or an image snippet) either locally or sending it to the server. After an event, the buzzer will stay on for a short period (e.g., 2 seconds) to ensure the driver is alerted, and it will turn off once the driver's eyes open again (EAR back above threshold). The cycle then continues. 9Multiple drowsiness events can be detected in one trip if the driver repeatedly dozes off, and each will be logged. We also included logic (not shown above) to detect yawning by monitoring the aspect ratio of the mouth; however, in our tests the eye-based detection was sufficient and more reliable, so yawning detection was not always enabled.

D. Flask API and Cloud Integration

To connect the vehicle's onboard system with the cab company's backend, we developed a REST API using Flask (a lightweight Python web framework). The Flask application exposes endpoints that the Raspberry Pi can call to report data, and that the company can call to retrieve data. For demonstration, we hosted this Flask app on a cloud server (alternatively, it could be on-premises at the dispatch office, etc.). The API endpoints and their functions are:

–POST '/api/ride check' – This endpoint is called by the Raspberry Pi when a ride request is initiated and the breathalyzer test is performed. The Pi sends a

JSON payload containing the driver ID (or cab ID) and the result of the alcohol test (either “clear” or “drunk”). Upon receiving this, the server logs the result in a database or in-memory store keyed by ride or driver, and responds with a confirmation. The server can also optionally respond with instructions, e.g., “reject ride” if drunk.

- POST ‘/api/drowsiness event‘ – This endpoint is used by the Pi to report a drowsiness alert event. The JSON payload contains the driver ID (or ride ID) and a timestamp of the event (and possibly additional info like severity or number of events). The server appends this info to a log associated with the driver’s current ride.

— GET ‘/api/status/¡driver id¿‘ – This endpoint allows the cab company (or any authorized client) to query the status of the driver or ride. The request includes a path parameter (driver ID or ride ID) and the server returns the stored information:

whether the driver passed the alcohol test for the current ride, and a list of any drowsiness event timestamps recorded so far. This could be used by a monitoring dashboard or even an automated system to proactively call the driver or passenger if too many alerts occur.

Listing 3 provides a simplified example of the Flask API
implementation, focusing on these routes. For brevity, it uses Python dictionaries to store state instead of an actual database, and assumes a single ride context for a driver at a time. In a production scenario, one would include authentication, persistent storage, and perhaps more detailed data models (e.g., linking rides, drivers, vehicles, timestamps, etc.).

```
from flask import Flask, request, jsonify
app = Flask(__name__)

# In-memory storage (for demo only):
ride_status = {} # store {'driver_id': 'clear'/'drunk'}
drowsiness_log = {} # store {'driver_id': [list_of_timestamps]}

@app.route('/api/ride_check', methods=['POST'])
def ride_check():
    data = request.get_json()
    driver_id = data.get('driver_id')
    alcohol_clear = data.get('alcohol_clear') # boolean or "clear"/"drunk" string
    # Update ride status
    if alcohol_clear:
        ride_status[driver_id] = "clear"
        result = "Ride allowed"
    else:
        ride_status[driver_id] = "drunk"
        result = "Ride rejected"
    # Initialize drowsiness log for this ride:
    drowsiness_log[driver_id] = []
    return jsonify({"status": ride_status[driver_id], "result": result})

@app.route('/api/drowsiness_event', methods=['POST'])
def drowsiness_event():
    data = request.get_json()
    driver_id = data.get('driver_id')
    timestamp = data.get('timestamp')
    # Append the event
    if driver_id in drowsiness_log:
        drowsiness_log[driver_id].append(timestamp)
    else:
        drowsiness_log[driver_id] = [timestamp]
    return jsonify({"logged": True, "count": len(drowsiness_log[driver_id])})

@app.route('/api/status/<driver_id>', methods=['GET'])
def status(driver_id):
    status = ride_status.get(driver_id, "unknown")
    events = drowsiness_log.get(driver_id, [])
    return jsonify({"alcohol_status": status, "drowsy_events": events})
```

Listing 3. Flask API Endpoints for Data Transmission

With this API in place, the Raspberry Pi uses Python’s `requests` library to send HTTP POST requests. For example, after the breathalyzer check, if `alcohol_detected` is False, the Pi would do:

```
requests.post(
      "https://api.cabcompany.com/api/ride_check",
      json={"driver_id": "DRIVER_123",
          "alcohol_clear": True})
```

And similarly, when a drowsiness event occurs, it would POST the timestamp:

```
requests.post(
   "https://api.cabcompany.com/api/drowsiness_event",
   json={"driver_id": "DRIVER_123", "timestamp":
       1690971234})
```

(The timestamp here is a Unix epoch or any agreed format; one could also send a formatted date-time string or other data like GPS location if needed.) On the company side, authorized personnel could GET the status: e.g.,

Security considerations: All API traffic should be over HTTPS to protect data. Authentication tokens or API keys should be used so that only the devices and company systems can post or read data. These aspects were handled minimally in our prototype (e.g., we hardcoded a simple token in the header in the Pi’s requests and verified it in Flask) but would need strengthening for production.

V. RESULTS

To evaluate the system, we conducted a series of tests in a controlled environment simulating

different driver conditions and ride scenarios. The primary objectives were to verify:
(1) that the alcohol detection correctly allows or blocks ride requests, (2) that the drowsiness detection alerts the driver promptly and logs events, and (3) that data transmission to the backend is reliable and timely.

A. Alcohol Test Scenarios

We tested the MQ-3 breathalyzer subsystem with three scenarios: a sober driver, a driver who had recently consumed an alcoholic beverage, and a false-positive scenario (driver uses an alcohol-based mouthwash but is not actually intoxicated). In the sober scenario, the sensor readings remained low (ADC values typically under 100) and the system correctly marked the driver as clear. The Flask API response indicated "Ride allowed" and the ride proceeded. In the intoxicated scenario, the driver gargled a small amount of an alcoholic beverage to simulate a breath alcohol content above the threshold. The MQ-3 readings spiked (ADC value ≈ 600 corresponding to a strong alcohol presence) and the system immediately flagged "Alcohol detected = True". The ride request was rejected; the Pi logged this and the backend recorded the status as "drunk". The response to the Pi (in our test interface) was to deny the ride. This matches expected behavior, effectively acting as a safety interlock. In the mouthwash scenario, initially the sensor did detect alcohol (the mouthwash has ethanol) and would have blocked the ride. This is a known challenge (distinguishing real intoxication from residual mouth alcohol). However, we found that by waiting about 2 minutes and retesting, the readings subsided to normal (since mouthwash effects dissipate quickly). This suggests a protocol: if a driver fails the first test, a second test can be done after a short interval to reduce false positives. In an actual deployment, one could integrate a rule that a driver who fails must wait and re-test, or have a supervisor verify.

B. Drowsiness Detection and Alert Timing

For the fatigue monitoring, we recorded video of a driver (one of the authors) mimicking drowsy behavior to test the response. In one test, the driver blinked normally for 30 seconds (EAR mostly above 0.3, with brief drops on blinks). The system did not trigger any alert, as expected. Next, the driver slowly started to close his eyes for longer periods. When the eyes stayed closed for about 1.5 seconds, the buzzer alarm activated. The time from eyes fully closed to alarm was approximately 1.5 s, which is largely governed by our FRAME_ THRESH setting. This is sufficiently quick to catch microsleep episodes (typically 2–3 seconds long). The alarm sound was effective in prompting the driver to open his eyes. We observed that the alarm would sometimes chirp briefly even when the driver deliberately blinked very slowly; to minimize annoyance from false alarms, one could increase the time threshold slightly or require two back-to- back prolonged closures to trigger. However, for safety we kept it sensitive. Over a 10-minute simulated "ride", where the driver periodically pretended to nod off, the system triggered 4 alerts. All 4 event timestamps were correctly logged in the Pi's memory and successfully POSTed to the server. The Flask server log showed the entries for each event with the correct driver ID and timestamps.

We also tested the behavior when the driver looks away or down (which can make the eyes not clearly visible). In a few such cases, the EAR calculation became unreliable (since the face detector or landmarks might lose track of the eyes). Our implementation does not trigger an alert in that scenario (instead, `eye_closed_frames` resets if no face/eyes detected or if EAR cannot be computed). This is a limitation: if a driver's face is not in view (e.g., checking blind spot), the system should not immediately flag drowsiness. Our approach handles it by resetting, which is acceptable because a brief turn of the head won't erroneously accumulate frames to trigger an alarm. If the driver turned away for an extended period, that itself might be unsafe (not watching the road), but that is beyond our current scope.

C. Data Transmission and Backend Logging

The communication between the Pi and the Flask API was tested on a local network as well as over the internet (using a 4G mobile hotspot for the Pi). Each POST request from the Pi received a response within roughly 100–200 ms locally and 300–500 ms over the internet. This small latency is negligible in the context of our use-case, since even if an alcohol test is positive, the driver would be notified not to proceed within a second. The GET status requests from a client were similarly quick. We also simulated 10 vehicles concurrently sending data (by running the code in parallel threads with different

driver IDs); the Flask server (a simple single-process instance) could handle the burst of requests without issue, though a production system would likely use a more robust setup or cloud functions.

On the backend, we verified that the data for each test case was correctly stored. For instance, after an "alcohol fail" test, a query to '/api/status/DRIVER 123' returned '"alcohol status": "drunk", "drowsy events": []'. After a driving session with alerts, the status showed the "clear" and a list of timestamps. The data can be further analyzed to derive insights, such as how long into the ride drowsiness occurred, which could help in scheduling driver breaks or investigating if the driver was overworked.

VI. DISCUSSION

The implemented system demonstrates a practical approach to enhancing ride safety by proactively monitoring the driver. One of the strengths of our system is that it leverages inexpensive sensors (the MQ-3 costs only a few dollars) and widely available computing hardware (Raspberry Pi) to achieve functionality that can be life-saving. By integrating with the ride acceptance workflow, it adds a layer of accountability: drivers are aware that they must be sober and alert to operate, and the system will intervene if they are not. This could act as a deterrent against risky behavior (e.g., a driver taking one more fare at the end of a long shift despite exhaustion, or driving after drinking).

The use of an IoT architecture (with the Flask API) means the solution is scalable to fleet deployments. A cab company could deploy such units in all its vehicles and have a central dashboard showing the status in real time. This ties into the concept of connected vehicles and smart transportation systems where data is collected for safety and operational efficiency. We also see potential for insurance incentives – for example, fleets that install such safety systems might negotiate lower insurance premiums due to reduced risk.

However, there are several considerations and limitations:

- Accuracy and Calibration: The MQ-3 sensor, while good for detecting presence of alcohol, is not specific to ethanol and can be triggered by other substances (as seen with the mouthwash). It also does not directly measure Blood Alcohol Concentration (BAC) with forensic accuracy. Our threshold- based approach is a coarse check. In a real deployment, calibration would need to be refined and perhaps adjusted for each sensor unit (sensors can have variability). Frequent false positives could frustrate drivers. One mitigation is to incor- porate additional context: for instance, integrating a camera to ensure the driver actually blows into the sensor (prevent cheating or ensure proper test) and performing two tests if the first is positive.
- Lighting and Environment for Camera: Vision-based drowsiness detection can be affected by lighting (e.g., glare, nighttime) and driver appearance (sunglasses will obstruct the eyes, for example). In our tests, we assumed no sunglasses and reasonably good lighting. Real-world use would require an IR camera or some alternative for low-light conditions. Additionally, processing on the Pi, while feasible, pushes its limits when using dlib; we managed around 8-10 FPS at 320x240 resolution with EAR computation. This is adequate, but future versions could utilize more efficient models (like CNN-based eye state classifiers) or hardware acceleration. Newer Raspberry Pi models or Google Coral accelerators could boost performance.
- Driver Comfort and Privacy: Having a camera always observing the driver may raise privacy concerns. In a commercial fleet, drivers can be informed and it's for safety, but care must be taken that the video is not misused. Our system only transmits summary data (alerts, etc.), not the video itself, which is a conscious design choice to balance privacy and safety. The breath test requirement might also inconvenience drivers slightly, but since it's only when a ride request comes (which naturally has some wait time for the driver to respond), it was found acceptable in pilot tests.
- Fail-safe and Response: When an impairment is detected, the system currently alerts (for drowsiness) or prevents opera- tion (for alcohol). In the case of a drowsy event, we rely on the driver to correct their behavior (wake up, possibly pull over if needed). We log the event for later, but real-time intervention could be expanded: e.g., notify the company immediately after multiple alerts so they can call the driver or passenger. In extreme cases, an automated system might even safely slow down the

vehicle (if it were integrated with vehicle controls in advanced cars). Our system is an advisory/alert system and does not take vehicle control.

- Ride Acceptance Logic: We considered tying this system directly into the ride-hailing app's API such that a driver cannot press "Accept" until a clear signal is received. This integration is application-specific. In a fleet scenario with a custom dispatch system, it's easier to enforce (the dispatch server can wait for the Pi's data before confirming the ride). For third-party ride-hailing platforms, it would require collab- oration or a workaround (like a device that can auto-tap the accept if conditions are met, or otherwise just rely on company policy).

In summary, while our prototype shows feasibility, deployment would require careful integration and calibration. The cost per vehicle is relatively low (the main cost being the Pi and camera), making it attractive for fleet operators concerned with safety.

VII. CONCLUSION

We presented a novel integrated system for ensuring driver sobriety and alertness in the context of ride acceptance for cab services. By combining an alcohol sensor test and continuous drowsiness monitoring, the system addresses two critical causes of accidents in one framework. The use of IoT connectivity allows results to be immediately shared with a central system, enabling real-time decision-making (such as blocking a ride) and post-ride analysis of driver alertness. The implementation on Raspberry Pi with off-the- shelf sensors demonstrates that the solution is both accessible and deployable with modest resources.

In our experiments, the system successfully detected alcohol in breath with sufficient sensitivity to prevent a mock "im- paired" trip, and it alerted the driver during simulated drowsy episodes, potentially averting an accident. These results, while preliminary, highlight the effectiveness of such preventative measures. The approach can be extended further: future work could integrate additional sensors like a heart rate or steering pattern monitor for detecting fatigue, or use smartphone-based detectors as backup. We also plan to improve the machine learning aspect — for example, using a trained model to better differentiate between normal blinks and drowsiness, or to detect when a driver might be distracted (looking away from road for too long). Integration with vehicular systems
(via OBD-II interface or CAN bus in modern cars) could allow actions like gradually slowing the vehicle if the driver is unresponsive to alerts.

We believe this intelligent ride acceptance system can be an invaluable addition to commercial transportation fleets. It not only safeguards the passengers and the driver but also provides actionable data to the operating company. With regulators increasingly interested in mandating anti-drowsiness and anti- drunk driving tech in vehicles, our system offers a practical blueprint using readily available technology. By emphasizing safety at the start of every ride and throughout the journey, such systems pave the way for safer roads and more reliable transportation services.

REFERENCES

[1] A. Chellappa, M. S. Reddy, R. Ezhilarasie, S. K. Suguna, and A. Uma- makeswari, "Fatigue detection using Raspberry Pi 3," *International Jour- nal of Engineering & Technology*, vol. 7, no. 2.24, pp. 29–32, 2018.

[2] A. Sahayadhas, K. Sundaraj, and M. Murugappan, "Detecting driver drowsiness based on sensors: a review," *Sensors*, vol. 12, no. 12, pp. 16937–16953, 2012.

[3] T. Soukupova´ and J. Cˇ ech, "Real-Time Eye Blink Detection using Facial Landmarks," in *Proc. 21st Computer Vision Winter Workshop*, Ranzan, Slovenia, 2016, pp. 1–8.

[4] N. N. Charniya and V. R. Nair, "Drunk driving and drowsiness detection," in *Proc. 2017 Int. Conf. on Intelligent Computing and Control (I2C2)*, Coimbatore, India, 2017, pp. 1–6.

[5] S. Uma and R. Eswari, "Accident prevention and safety assistance using IoT and machine learning," *J. Reliable Intelligent Environments*, vol. 8, no. 2, pp. 79–103, 2022.

[6] A. Prasad, J. Pavankalyan, A. S. Ganesh, and K. M. Krishna, "Drowsiness and alcohol detection with engine locking," in *Proc. 3rd Int. Conf. on Communication, Computing and Industry 4.0 (C2I4)*, 2022, pp. 1–5.

[7] M. Bhushan, D. Joshi, T. K. Gujral, S. K. Singh, A. Singh, and A. Negi, "Application of machine learning in driver drowsiness

detection," in *Proc. 2023 Int. Conf. on Artificial Intelligence and Applications (ICAIA)*, 2023, pp. 1–6.

[8] Centers for Disease Control and Prevention (CDC), "Impaired Driving: Facts and Statistics," May 2024. [Online]. Available: https://www.cdc. gov/impaired-driving/facts/