Problems Faced and How We Overcame Them

The "Noisy" Steering Wheel Potentiometer

Problem: We needed a low-cost way to detect steering input (behaviour). The obvious choice, a simple rotary potentiometer, was extremely "noisy." Its analog readings would jitter and "ghost" (show small movements) even when the wheel was held perfectly still, making it impossible to tell if the driver was **actually** making small corrections or if it was just electrical noise. This would have led to false positives (thinking the driver is steering) or false negatives (missing real movement).

Solution: We implemented a sophisticated, **three-stage software filter** to create a rock-solid reading from the cheap hardware.

1. **Median Filter:** We didn't just take one reading. The `readMedian5` function takes five rapid samples and chooses the **middle** value, instantly throwing out any random high or low electrical spikes.

2. **Exponential Moving Average (EMA):** We then fed this median value into an EMA filter (`ema = EMA_ALPHA * raw + ...`). This heavily smooths the signal over time, filtering out any remaining high-frequency jitter and giving us a clean, stable value representing the **true** steering position.

3. **Deadband:** Finally, we used a `DEAD_BAND = 20`. This means the system won't register any "movement" at all unless the *smoothed* value changes by more than 20 units. This was the final step to ignore any tiny, residual drift and ensure that only clear, intentional steering from the driver would reset the emergency timer

HTTPS Emergency SMS Failing on the ESP32

Problem: A critical feature in both `vitals.' and `behaviour` was sending an emergency SMS via the Twilio API. However, `api.twilio.com` requires an **HTTPS** connection. When we first tried this, the ESP32 would have failed to connect every time. This is because SSL/TLS (the 'S' in HTTPS) requires the device's system clock to be accurate to validate the server's security certificate. The ESP32 has no real-time clock and boots up thinking it's 1970, causing an immediate SSL handshake failure.

Solution: We ingeniously used the **GPS module as a network time source.** In `vitals' we wrote the `syncClockToGPS()` function. This function runs at startup, polls the GPS module (`feedGPS()`), and waits until it gets a valid satellite fix (`gps.date.isValid()`). It then reads the precise UTC date and time from the GPS data, converts it to an epoch timestamp, and uses it to set the ESP32's internal system clock (`settimeofday`). With an accurate time, the SSL handshake to Twilio succeeded, allowing the emergency SMS to be sent reliably.

Vision System Freezing During Audio Alerts

Problem: In our `vision` script, when the system detected fatigue (like a microsleep), it needed to play an audio alert (`break.mp3`). A simple or naive implementation (like `os.system` or `subprocess.run`) would have **blocked the entire Python script**. This means while the 3-second alert was playing, the camera feed would freeze, and the script would be blind. It couldn't detect if the driver woke up, or if their head slumped further, making the system unresponsive.

Solution: We used `subprocess.Popen` to play the audio. Unlike `subprocess.run`, `Popen` launches the `mpg123` audio player as a **separate, non-blocking background process**. Our main Python script was freed **instantly** and could immediately loop back to processing the next camera frame from the PiCamera. This ensured zero lag in the vision analysis, allowing the system to stay fully responsive and monitor the driver's reaction **while** the alert was playing.

Real-time Face Mesh Performance on Raspberry Pi

Problem: The MediaPipe Face Mesh model (`vision`) was computationally expensive. Running it in real-time on a Raspberry Pi (implied by `picamera2`) was a major challenge. We initially experienced very low, laggy frame rates, which would cause the system to miss quick yawns or blinks and feel unresponsive.

Solution: We optimized the vision pipeline in two key ways.

1. **Hardware-Accelerated Camera:** We used the `picamera2` library. This is the modern, hardware-accelerated library for Raspberry Pi cameras, which captures and provides frames far more efficiently than older libraries or a standard `cv2.VideoCapture`.

2. **Confidence Tuning:** We explicitly set `min_detection_confidence=0.6` and `min_tracking_confidence=0.6`. This was a balancing act. Setting these values too high (e.g., 0.9) would have made the model slow as it struggled to find a "perfect" face. Setting them too low (e.g., 0.3) would have resulted in fast, but inaccurate, detections. Our values (0.6) were a

well-tuned compromise, providing the best balance of speed and reliability for our specific hardware.

PC Dashboard (Tkinter) Freezing on Serial Communication

Problem: Our `vitals` dashboard has a Tkinter GUI, which has its own main loop. It also needed to listen for potential responses from the ESP32 over the serial port. If we had used the standard `ser.readline()` command, it would have been a "blocking" call it would have **frozen the entire GUI** and stopped the clock from updating, waiting for a message that might never come.

Solution: We implemented a **non-blocking, buffered serial processing loop**.

1. We created a global `serial_buffer`.

2. We created a function `process_serial_data()` that **never** blocks. It uses `ser.read(ser.in_waiting)` to read **only** the bytes that were already waiting in the hardware buffer.

3. It adds these bytes to our software `serial_buffer`, and **then** processes any complete lines (ending in `\n`) from that buffer.

4. This function is polled continuously by our Tkinter loop (`update_time()` calls it), meaning the GUI **never** froze and could handle serial data as it arrived, all while remaining fully responsive to the user.