Real – Time Vehicle and Pedestrian Detection

1. Introduction

1.1 Purpose

The purpose of this project is to design and implement a **real-time vehicular detection of pedestrians' system** using deep learning techniques to improve road safety and reduce the risk of collisions. Pedestrian-related accidents remain a major cause of fatalities and injuries worldwide. The proposed system addresses this issue by leveraging the YOLOv9 object detection model, distance estimation techniques, and alert mechanisms to assist drivers in identifying pedestrians and vehicles in their surroundings. By providing timely and accurate information, the system enhances driver awareness and helps prevent potential accidents. This document is intended to serve as a reference for all stakeholders involved in the development and deployment of the system. The primary audience includes:

- **System Developers** – to understand the system architecture, design goals, and implementation details required for development.
- **Quality Assurance and Test Engineers** – to derive test cases, validation strategies, and performance benchmarks ensuring the system meets functional and non-functional requirements.
- **Project Managers** – to oversee project progress, ensure alignment with business objectives, and manage timelines, risks, and resources.
- **End Clients and Stakeholders** – to review the project scope, intended functionalities, and benefits in order to validate that the system aligns with safety objectives and user needs.
- **Future Maintainers and Researchers** – to use this document as a reference for enhancements, upgrades, or academic study in related domains.

1.2 Scope

The software product to be developed is the Vehicular Detection of Pedestrians System, a real-time object detection and alert solution that integrates deep learning techniques to enhance road safety.

The system will provide the following functionalities:

- **Real-Time Object Detection** – Identification of vehicles and pedestrians in video frames using the YOLOv9 deep learning model.
- **Distance Estimation** – Calculation of approximate distance between the vehicle and detected objects based on bounding box parameters and known dimensions.
- **Alert Mechanism** – Generation of visual and auditory alerts when pedestrians or vehicles are within a predefined critical distance threshold.

- Frontend Interface – A responsive web-based interface that allows users to upload videos, view live camera feeds, and monitor detection statistics.
- Backend Processing – A Flask-based backend responsible for video frame processing, object detection, distance estimation, and alert generation.

The system will not include:

- Autonomous Vehicle Control – The software is limited to detection and alerting; it will not take direct control of the vehicle (e.g., braking or steering).
- Comprehensive Traffic Analysis – The system will not perform broader traffic management or vehicle tracking beyond immediate detection of pedestrians and nearby vehicles.
- Integration with External Sensors – The system relies primarily on camera input and does not incorporate additional sensors such as LiDAR or RADAR in this version.

The application of this software lies in driver assistance systems for vehicles, serving as a preventive mechanism against pedestrian-related collisions. It is intended to provide drivers with real-time situational awareness and timely alerts, thereby reducing the risk of accidents in urban and highway driving scenarios.

The primary objectives and goals of the system are as follows:

1. To enhance driver awareness by detecting pedestrians and nearby vehicles in real-time.
2. To reduce accident risks by providing distance-based warnings and alerts.
3. To deliver a lightweight and deployable solution using deep learning techniques optimized for real-time performance.
4. To serve as a foundation for future research and development in intelligent transportation systems and autonomous driving assistance.

1.3 Definitions, acronyms and abbreviations

This subsection defines the key terms, acronyms, and abbreviations used throughout this Software Requirements Specification (SRS) to ensure clarity and consistency.

- YOLO (You Only Look Once): A family of real-time object detection models. YOLOv9 is the latest version used in this project to detect vehicles and pedestrians.
- YOLOv9: The deep learning model applied for object detection in this system, capable of real-time performance with high accuracy.
- Flask: A lightweight Python-based web framework used for backend integration, including video processing and alert generation.
- OpenCV (Open Source Computer Vision Library): A computer vision library used for image and video frame capture, preprocessing, and distance estimation.

- **GUI (Graphical User Interface)**: The web-based user interface developed using HTML, CSS, and JavaScript for user interaction.
- **FPS (Frames Per Second)**: A measure of how many video frames the system processes per second, reflecting real-time performance.
- **Bounding Box**: A rectangular outline generated by the detection model around identified objects (vehicles or pedestrians).
- **Distance Estimation**: A technique to approximate the distance between the vehicle and detected objects based on bounding box dimensions and known object sizes.
- **Alert Mechanism**: A visual and/or auditory notification system triggered when an object is within a critical safety threshold.
- **Driver Assistance System**: Software designed to support drivers by providing warnings and real-time information without taking autonomous control of the vehicle.

1.4 References

The following documents, research articles, and online resources have been referred to in preparing this Software Requirements Specification (SRS):

1. IEEE Standard for Software Requirements Specifications (IEEE Std 830-1998).
2. Redmon, J., & Farhadi, A. (2016). *You Only Look Once: Unified, Real-Time Object Detection.* Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).
3. YOLOv9: A Leap Forward in Object Detection Technology (Ultralytics YoloV9 Documentation)
4. OpenCV Documentation. *Open Source Computer Vision Library.*
5. Flask Documentation. *Flask Web Framework.*
6. National Highway Traffic Safety Administration (NHTSA). *Traffic Safety Facts: Pedestrians.*

1.5 **Overview**

This Software Requirements Specification (SRS) document is organized to provide a comprehensive and structured description of the Vehicular Detection of Pedestrians System. The subsequent sections of this document are arranged in alignment with IEEE standards for software requirements specification to ensure clarity, completeness, and ease of reference.

- Section2: Overall Description: Provides a high-level understanding of the product, including its perspective, core functions, user characteristics, operating constraints, assumptions, and dependencies. This section establishes the system context and sets the foundation for more detailed requirements.

- Section3: Specific Requirements: Details the functional and non-functional requirements of the system to a level sufficient for design and testing. Requirements are organized according to system functionality, user classes, and feature sets, ensuring unambiguous specification for developers and testers.

- Appendices: Provide supplementary details including definitions and diagrams that support the core requirements but are not essential to the main body of the SRS.

This organization ensures a logical progression from general description (system context) to detailed specification (functional and non-functional requirements). Stakeholders—including developers, testers, project managers, and end users—can navigate the document based on their specific needs while maintaining a shared understanding of the system requirements.

2. Overall Description

2.1 Product Perspective

The Pedestrian Detection System is largely self-contained. It interfaces primarily with vehicle camera sensors and the driver interface. The system may optionally integrate with existing vehicle electronics (e.g. sharing vehicle speed via CAN bus or providing output to an infotainment screen), but these are not required for core functionality. A conceptual block diagram of the system (shown below) would include: (1) Input Interface – one or more cameras providing live video; (2) Processing Unit – an onboard computer running the deep learning model (YOLOv9) and associated algorithms; (3) Output Interface – a user interface (visual display and speaker) for alerts and information. The system does not assume control over vehicle actuators and does not perform functions like steering or braking.

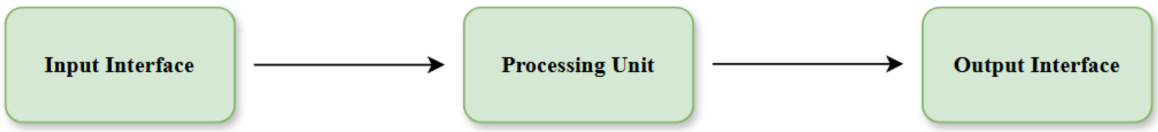

Fig. 1.1 Conceptual Block Diagram

The software operates within various constraints and interfaces:

2.1.1 System Interfaces

The system connects to video sources (e.g. vehicle-mounted cameras or external cameras). It receives video frames via standard interfaces (USB 3.0, Ethernet, or wireless streaming) at typical resolutions (e.g. 720p–1080p) and frame rates (≥15–30 FPS). Optionally, it can interface with a vehicle's data bus to obtain ancillary information (such as vehicle speed or GPS) for enhanced situational context, but this is not mandatory. It also provides output to external displays or infotainment units by sending annotated video or alert signals. All system I/O follows standard protocols (e.g. camera protocols and display output APIs).

2.1.2 User Interfaces

End users interact with the system through a graphical web-based GUI accessible on a browser or in-car display. The logical characteristics of this interface include live video display with overlay annotations (bounding boxes and distance labels), menus or buttons for settings (e.g. alert thresholds), and status dashboards (such as detection statistics and system health). The UI is designed to be responsive and intuitive: for example, a first-time user should immediately see the live camera feed and any active detections. Configuration options (long/short messages, alert volumes) are provided. Consistency in layout, clear labelling (e.g. “Pedestrian Detected”), and the use of color-coded warnings ensure users can quickly interpret information. Accessibility considerations (e.g. high-contrast mode, large fonts, language selection) are included so that even users with minimal technical expertise can operate the system effectively.

2.1.3 Hardware Interface

The software interfaces with the following hardware components:

1. **Cameras**: Standard digital cameras compatible with OpenCV (e.g. USB webcams, IP cameras). The system supports common resolutions (640×480 up to 1920×1080) and requires access to the raw video stream. Full-screen colour video capture is used.
2. **Processing Unit**: A computing platform (laptop, embedded PC, or specialized edge device) with a modern CPU and GPU (NVIDIA GPU with CUDA support) is required for real-time deep learning inference. For adequate performance (e.g. ≥15 FPS inference), hardware with at least 8 GB of GPU memory is recommended. The system supports running on Windows 10/11 or Linux OS, on either x86 or ARM architectures that meet these requirements.
3. **Audio Output**: A speaker or buzzer for audible alerts, connected via standard audio ports. The system may also use the host device’s audio output.
4. **Input Controls**: Any standard input device (keyboard, touchscreen, or physical buttons) connected to the processing unit to allow the user to adjust settings or acknowledge alerts.
5. **Power Supply**: If integrated into a vehicle, the system runs off vehicle power (12 V DC). Appropriate power regulators convert to required voltages for the CPU/GPU.

2.1.4 Software Interfaces

The system relies on the following software products and libraries:

1. **Operating System**: The host OS (e.g. Ubuntu 20.04+, or Windows 10+) must provide support for required drivers and runtimes. It is assumed that the OS and essential drivers (e.g. NVIDIA CUDA drivers, USB drivers) are already installed and available.
2. **Deep Learning Framework**: Ultralytics YOLOv9 (Python-based implementation) serves as the object detection model. This requires Python 3.8+ with PyTorch (GPU-enabled) and dependencies.
3. **OpenCV**: The OpenCV library (Python/C++ version) for video capture, image preprocessing, and basic computer vision utilities (e.g. reading frames, resizing, drawing bounding boxes).
4. **Web Framework**: Flask (a lightweight Python web framework) will implement the backend server. It interfaces with front-end HTML/CSS/JavaScript code to serve pages and stream results.
5. **Other Libraries**: Standard libraries for numerical processing (e.g. NumPy), audio playback, and video encoding/decoding as needed.

For each of the above, the specification includes the required version and source (e.g. "Python 3.9; PyTorch 2.0 with CUDA support; OpenCV 4.x; Flask 2.x; YOLOv9 repository). Interfacing with these components is handled via their APIs (e.g. OpenCV's VideoCapture, PyTorch model inference calls).

2.1.5 Communication Interfaces

The system can operate fully offline, but if network connectivity is available, it uses standard TCP/IP protocols. The web UI communicates with the backend via HTTP/HTTPS (for web pages and AJAX calls) or WebSocket for live updates. (If operating within a local vehicle network, it will use whatever LAN protocols are present.) There is no requirement for proprietary communication interfaces. The system does not rely on cellular or V2X communication, though it supports connecting to Wi-Fi or Ethernet if needed.

2.1.6 Memory Constraints

The software requires sufficient memory to load the YOLOv9 model and process video frames. At a minimum, the system should have ~16 GB of system RAM and ≥8 GB GPU VRAM. Primary memory (RAM) usage must be managed so that frame buffers and model inference do not exceed available memory. Secondary storage (disk) should have adequate space (tens of GB) for saving video logs and system data.

2.1.7 Operations

The system operates primarily in two modes:

1. **Live Detection Mode (Normal Operation)**: On vehicle startup or when activated by the user, the system continuously captures video frames, runs object detection on each frame, computes distances, and updates the display. This is a user-initiated operation and is primarily interactive (live). The user may pause the detection, switch cameras, or adjust thresholds on-the-fly via the UI. Interactive period is continuous while the vehicle is in use.
2. **Batch Analysis Mode**: (Optional) The user can upload a previously recorded video and have the system process it (without live capturing). This is a user-initiated operation and may run in an unattended way until the end of the video.
3. **Exception Handling**: If a camera feed is lost or an error occurs, the system should alert the user and attempt recovery or shut down safely.

Note: The system assumes continuous operation while the vehicle is on; periods of inactivity (vehicle off) are not part of normal function. Backup and recovery beyond rebooting the application is outside the current scope.

2.1.8 Site Adaptation Requirements

For each specific vehicle installation, certain parameters may need calibration:

1. **Camera Calibration**: Lens distortion and camera mounting height must be known to accurately estimate distances. The system requires initial calibration inputs (e.g. camera matrix or a known reference object at a known distance) that are specific to the installation site.
2. **Threshold Configuration**: The critical distance at which alerts are triggered may be adjusted per usage context (e.g. urban vs. highway driving). These safety limits should be configurable at installation or per driving scenario.
3. **Platform Features**: The software must be adaptable to variations in computing platforms (different GPUs, camera models). Feature toggles (e.g. disable GPU processing if no GPU is present) should be provided.

Overall, the product perspective is that of a self-contained intelligent camera system running on an onboard computer. It leverages advanced object detection (YOLOv9) and does not depend on external sensors (beyond cameras) or remote servers for its core function. It interfaces with standard hardware and software components and is primarily constrained by the computing platform available in the vehicle.

2.2 Product Functions

The system performs the following major high-level functions:

1. **Real-Time Object Detection**: Continuously process video frames to identify objects. The system uses a YOLOv9-based deep neural network to detect and classify vehicles and

pedestrians in each frame. Detection is done at real time to ensure timely recognition. Detected objects are marked by bounding boxes on the video output.

2. Distance Estimation: For each detected object, estimate the distance to the vehicle. This uses the size/position of the bounding box. In practice, the system computes distances by analysing bounding box centroids and object dimensions. The distance estimate is used to determine threat levels.
3. Warning/Alert Generation: If a detected pedestrian or vehicle is within a critical safety distance (configurable threshold), the system generates an alert to the driver. Alerts include both visual indicators (e.g. flashing icons on the display, a highlighted bounding box or distance number) and audible warnings (beeps or voice prompts). For example, many similar forward collision warning systems provide audible and visual alerts to prevent crashes. Our system mirrors that behavior for pedestrian/vehicle proximity, helping the driver take action.
4. User Interface Presentation: The system provides a graphical interface showing the camera feed with overlay of detection results (bounding boxes and distances). It also shows statistics (e.g. current frame rate, total pedestrians detected) and system status (e.g. CPU/GPU load). The interface allows the user to upload recorded videos for analysis, view past alert logs, and adjust settings such as detection sensitivity and alert thresholds. All information is presented in clear graphical or textual form to be quickly understood by the user.
5. Backend Processing: A Flask-based server runs the core logic. It handles video capture from the camera, runs the YOLOv9 model on each frame, performs distance calculations, and decides when to trigger alerts. The backend also logs events (detection times, object positions, alert triggers) to storage for later review or debugging. Non-functional goals include low latency (detect and alert within tens of milliseconds) and minimal resource usage consistent with embedded deployment.
6. Data Logging and Analytics: The system records detection events (e.g., "pedestrian detected at T=12.34s, distance=8.5m") along with timestamps. These logs can be viewed or exported by users (e.g. as CSV or in a database) for post-run analysis, such as counting how many pedestrians were detected on a route. A simple analytics function may compute averages (e.g. mean detection time) to assess performance.
7. Configuration and Maintenance Functions: Administrative functions allow calibrating the distance estimation (e.g. enter camera height or run a calibration routine with a marker), updating the object detection model (e.g. new YOLO weights), and upgrading software.

These functions are organized logically to serve driver safety: the detection and distance modules feed into the alert module, while the interface and backend serve as the means for user interaction and system operation. A simplified function flow is:

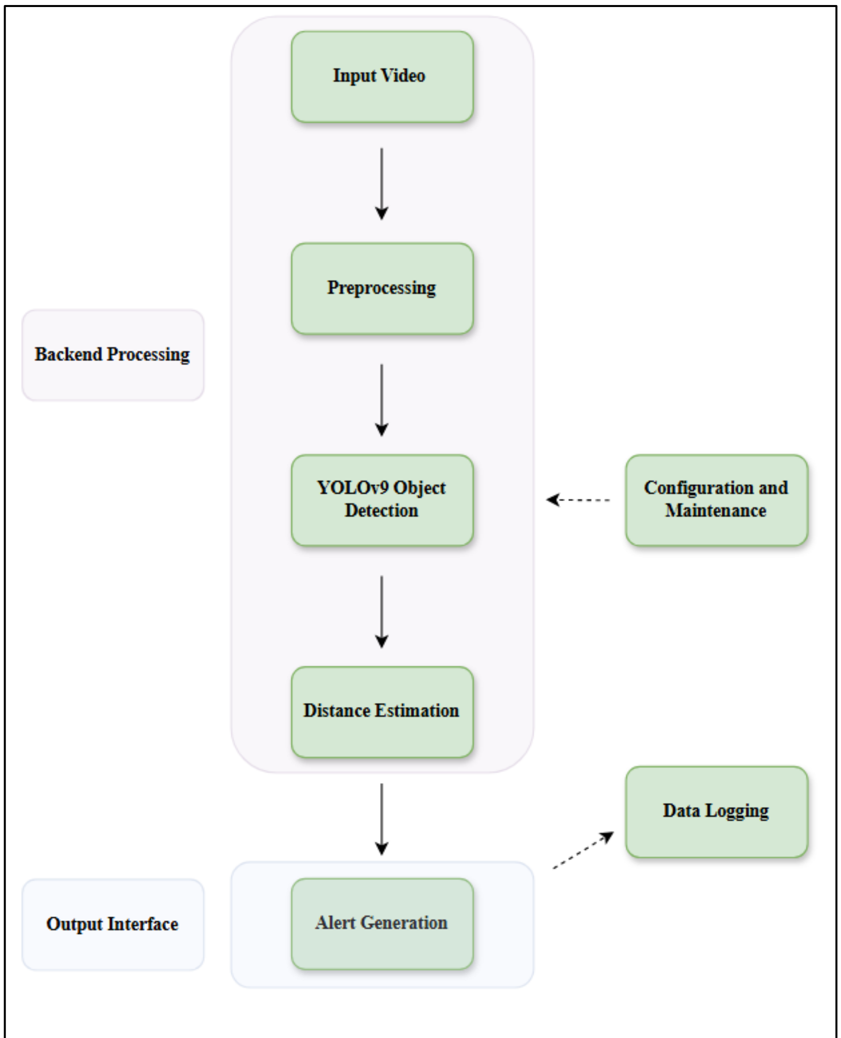

Fig. 1.2 Function Flow Diagram

By combining these functions, the system aims to enhance driver awareness and accident prevention: it automatically identifies pedestrians in the scene (leveraging modern AI) and provides timely warnings to the driver before a collision risk becomes critical. The output functions (GUI and alarms) ensure this information is conveyed clearly and promptly, integrating seamlessly into the driver's workflow.

2.3 User Characteristics

The end users of the system are primarily ordinary vehicle drivers and front-seat occupants who will rely on the system's alerts while driving. Their general characteristics are:

- Drivers (Primary Users): Adults of varying age (18 and above) who hold valid driver's licenses. They are skilled at basic vehicle operation but may have limited familiarity with computer systems. Most will have experience with standard in-car displays or smartphone apps (suggesting average digital literacy). The interface is thus designed to be simple and intuitive: large buttons, clear icons, and brief messages. Drivers may have different levels of attention and speed of reaction; the system assumes a driver can respond to an alert within a few seconds. Importantly, users are expected to remain fully engaged in driving; the system provides assistance but does not relieve them of responsibility. No specialized training beyond a short introduction (e.g. 10 minutes reading the manual) is assumed.

- Technical Staff (Secondary Users): While not system end-users, technical staff (such as installation technicians or system integrators) will interact with the system for setup and maintenance. These users have a higher technical background (familiar with software installation, hardware calibration, or vehicle electronics). They will use diagnostic tools or configuration menus to calibrate cameras, update software, and troubleshoot issues. The interface for technical users may present additional information (e.g. raw sensor data, performance metrics).
- Regulatory Reviewers or Researchers: In some contexts, transportation safety experts or researchers may review the system's documentation or logs. They expect clear explanations of features and access to data (e.g. accident reports or logs) to verify safety claims.

This SRS is written considering the above audiences: general descriptions are provided for drivers (lay users) as justification for design decisions, while the document also supports developer and tester understanding. For example, noting that drivers need simple controls justifies large button designs.

2.4 Constraints

Several constraints will limit the system's design choices and implementation:

1. Regulatory and Standards Constraints: The system must comply with relevant automotive safety standards (e.g. ISO 26262 for functional safety) and industry guidelines. It should follow NHTSA recommendations for ADAS features where applicable. Legal regulations on data privacy (e.g. GDPR/PDPA) constrain how camera data is stored or transmitted; the system should avoid saving personally identifiable information longer than needed. The alerts and interface must also meet human factors guidelines (e.g. alerts must not distract beyond defined durations).
2. Hardware Limitations: The performance of YOLOv9 (real-time object detection) is heavily dependent on computing hardware. Real-time operation (at least ~15–30 FPS) essentially requires a modern GPU; running on a CPU alone is typically too slow for live processing. For example, a CPU inference time per image (~500 ms) is much slower than on GPU (~15 ms). Therefore, the system assumes the host platform includes a CUDA-capable NVIDIA GPU with ~8 GB or more VRAM. Limited memory (RAM or GPU memory) will constrain the models that can be used. The camera hardware must provide sufficient resolution and frame rate under low-light conditions; if camera quality is poor, detection accuracy will suffer. Physical constraints (vibration,

temperature, dust) typical of vehicles mean the system hardware and mounting must be ruggedized.

3. **Software Interface Constraints**: The system depends on specific software components (Python 3.8+, PyTorch, OpenCV, Flask). These create dependencies: for example, the target computer must run an OS and architecture supported by these libraries. We assume the required OS (e.g. Linux kernel, Windows) and libraries are available. Any limitation in these (e.g. missing driver, incompatible library version) could prevent the system from working. There is also a constraint on language: implementation in Python (a higher-order language) means runtime performance is partly dependent on interpreter speed and library optimizations. The design assumes Python as specified.
4. **Communications Constraints**: The system primarily uses standard internet protocols. It assumes TCP/IP networking is available if needed, but it does not require high-bandwidth or low-latency networks for its core function (as all critical processing is local). If remote features (e.g. cloud logging) are added later, the system must use secure protocols (HTTPS, WSS) to protect data. Any wireless links used (Wi-Fi, Bluetooth) impose their own handshake/protocol constraints (e.g. XON/XOFF flow control is not applicable here; instead, we use TCP/IP stacks).
5. **Memory and Performance Constraints**: Real-time detection imposes strict performance requirements. The system must process each frame within a fixed time budget (dictated by the chosen frame rate). Exceeding this (e.g. due to large images or slow hardware) is not acceptable. Primary memory constraints include the available RAM for frame buffers and model loading. For instance, YOLOv9 variants can have tens of millions of parameters, requiring several gigabytes of memory for inference. Secondary memory (disk space) must accommodate video logs without degrading performance (e.g. using SSDs or high-speed SD cards).
6. **Reliability and Safety Constraints**: As a safety-related system, reliability is critical. The software must operate continuously without crashes during vehicle use, and must handle errors gracefully. Any failure mode (e.g. deadlock, memory leak) is unacceptable if it prevents alerts from reaching the driver. The system's criticality level is high: false negatives (missing a pedestrian) or system outages could lead to accidents. Therefore, the SRS will include accuracy targets (e.g. target 90% detection rate in test conditions) and uptime requirements (e.g. 99% availability). The system must also ensure that it does not produce excessive false alarms (which could reduce driver trust).
7. **Control Constraints**: Importantly, the software is not allowed to control vehicle actuators. All alerts are advisory. This constraint is a design decision (consistent with NHTSA "You Drive, You Monitor" guidance) and simplifies safety validation, since the system needs to only warn the driver rather than act.
8. **Security Constraints**: Although operating offline, the system must protect against unauthorized access. The web interface should require user authentication to change settings or view sensitive data (to prevent tampering with thresholds). Any external connections (e.g. USB) must be handled safely to avoid malware. The system should encrypt any stored sensitive logs if required by policy (e.g. encryption of video files).
9. **Parallel Operations**: The software should not monopolize the host system. It is assumed that the platform may run other applications (e.g. navigation). Therefore, the system

must run detection in parallel without interfering (e.g. by using only a subset of CPU cores if needed, or by using GPU with compute pre-emption).

These constraints (regulatory, hardware, interface, reliability, etc.) are derived from industry standards and the nature of the application. They define the boundaries within which the system must be designed.

2.5 Assumptions and dependencies

This section lists factors assumed to be true and external dependencies on which the requirements rely:

1. **Operating Environment**: We assume the vehicle has stable power and that the host computer boots and runs normally. The system presumes adequate illumination conditions for camera vision (very dark or glare-heavy scenes may degrade performance). We also assume normal weather conditions (extreme rain/fog might impair visibility; handling those is beyond current scope).
2. **Hardware Availability**: It is assumed that the target hardware (GPU, camera, speaker) meets the specifications. For example, an assumption is that an NVIDIA GPU with CUDA (compute capability ≥6.0) and a minimum of ~8 GB VRAM is available. If the required hardware is absent, the system's performance will not meet requirements.
3. **Software Availability**: We assume that all required software (OS, Python runtime, libraries) can be installed on the host. For instance, "Linux Ubuntu 20.04 or later is installed with Python 3.9, PyTorch 2.0 with CUDA, OpenCV 4.x, and Flask 2.x" is a baseline assumption. If these are not available, the software cannot run as specified. The required models (pre-trained YOLOv9 weights) are assumed to be provided and not part of this SRS.
4. **User Behavior**: The driver is assumed to remain attentive and ready to react. We rely on the fundamental ADAS assumption that "the user drives and monitors; all features are assistive" The system assumes users will not disable alerts or ignore them.
5. **Data Dependencies**: The performance of the detection model depends on the data it was trained on. It is assumed that the model (trained on datasets with vehicle and pedestrian images) generalizes well to the target environment. No additional sensor data (like LiDAR) are assumed – the software depends solely on video. We assume that the classes "pedestrian" and "vehicle" as used by YOLOv9 align with the objects in the environment; any objects outside these categories will not be detected.
6. **Calibration Data**: The distance estimation assumes certain camera calibration values (focal length, etc.). We assume that these values are either pre-calibrated for the camera used or will be input by the installer. Without calibration, distance estimates may be inaccurate, but the system still performs detection and warnings (albeit with reduced precision).
7. **External Interfaces**: If network features are enabled (e.g. uploading to cloud or remote monitoring), we assume network connectivity exists. However, core function does not depend on constant connectivity.

8. **Regulatory Environment**: We assume the regulatory landscape does not change such that in-vehicle warning systems become disallowed or require additional features.

These assumptions are not software requirements per se, but they affect the validity of requirements. For example, assuming a suitable GPU allows us to specify real-time performance targets. If later it turns out that the target installation does not have a GPU, the SRS would have to change accordingly.

Dependencies

The system depends on third-party components: the YOLOv9 model implementation (from Ultralytics) and associated deep learning libraries, OpenCV, and web frameworks. Updates to these components (e.g. new OS versions) could affect system compatibility. The SRS assumes these dependencies remain available and compatible with the system environment.

2.6 Apportioning of requirements

During the requirements analysis, certain features were identified as desirable but not essential for the initial release of the system. These requirements, while contributing significantly to the overall usability and scalability of the product, have been deferred to later versions due to considerations such as time-to-market, budgetary limitations, resource availability, or technical dependencies.

The following deferred requirements are categorized and described below:

2.6.1 Functional Enhancements

- **Advanced Analytics and Reporting**: The initial release will support core reporting capabilities (e.g., summary statistics, usage metrics). Predictive analytics, trend forecasting, and customizable dashboards are postponed to future iterations.
- **Multi-language Support**: Version 1.0 will operate in English only. Future versions will support internationalization (i18n) and localization (l10n) for additional languages, prioritized by market demand.

2.6.2 Platform and Accessibility Enhancements

- **Mobile Application Integration**: The first release will provide a responsive web interface. Native Android and iOS applications will be considered in subsequent releases to increase accessibility and mobility for end users.
- **Offline Functionality**: Continuous internet connectivity will be required in Version 1.0. Offline access with local caching and data synchronization will be included in later versions.

2.6.3 External System Integration

- **Third-party APIs**: Interfaces with external payment gateways, cloud storage providers, and third-party authentication services (e.g., OAuth, SAML) are planned for later releases. These integrations will be prioritized based on client needs and security assessments

2.6.4 Security and Compliance Extensions

- **Advanced Security Features**: The initial release will comply with baseline security standards (e.g., role-based access control, password encryption). Features such as multi-factor authentication (MFA), biometric login, and compliance with emerging regulations (e.g., GDPR expansions) will be deferred.

2.6.5 Future Research and Innovation

- **Artificial Intelligence (AI)-Driven Features**: AI-based recommendation engines, anomaly detection, and intelligent assistants are considered long-term enhancements. These require additional R&D and will be introduced in later stages of the product lifecycle.

Deferring these requirements ensures:

1. Timely delivery of the core system.
2. Reduced complexity in the initial development cycle.
3. Efficient allocation of resources toward business-critical features.
4. Scalability for accommodating future growth and user needs.

3. Specific Requirements

3.1 External interfaces

The following section contains all the input and output mappings of the system.

3.1.1 Camera Feed (Live video input)

1. **Purpose**: Provide the live video frames used for detection and distance estimation.
2. **Source / Destination**: Source: vehicle-mounted camera (USB / CSI / IP stream). Destination: backend video-capture module.
3. **Format / Fields**: Raw frames in BGR or RGB format (OpenCV compatible). Supported encodings: raw, MJPEG, H.264. Frame metadata: timestamp (ISO 8601), *frame_id* (integer).
4. **Valid range / Units**: Supported resolution range: 640×480 up to 1920×1080. Frame rate: 15–60 FPS. Colour depth: 8 bits/channel.

5. **Timing / Frequency**: Continuous stream; backend must accept and process frames at the camera's configured FPS.
6. **Relationship / Notes**: Frames must include accurate timestamps; frame coordinate system origin is top-left. Camera calibration parameters (focal length, principal point, distortion coefficients) may be supplied separately at installation.
7. **Error / End messages**: *ERR_CAMERA_DISCONNECTED, ERR_UNSUPPORTED_CODEC, ERR_FRAME_DROP.*

3.1.2 Video Upload (Offline video input)

1. **Purpose**: Allow users to upload recorded videos for batch analysis.
2. **Source / Destination**: Source: user browser or file transfer. Destination: backend upload handler.
3. **Format / Fields**: Supported containers: MP4, MKV, AVI. Codecs: H.264, H.265. Metadata: filename, size (bytes), duration (s).
4. **Valid range / Units**: Max file size: 4 GB (configurable). Duration: up to 2 hours per upload.
5. **Timing / Frequency**: User-initiated; processed asynchronously; progress updates returned via API/WebSocket.
6. **Error / End**: *ERR_UPLOAD_TOO_LARGE, ERR_UNSUPPORTED_FORMAT, UPLOAD_COMPLETE*.

3.1.3 Frontend Controls (User commands)

1. **Purpose**: Provide user control signals: start/stop detection, change alert thresholds, select camera, initiate calibration.
2. **Source / Destination**: Source: web UI (browser). Destination: backend REST/WebSocket endpoints.
3. **Format / Fields**: JSON commands, e.g. {*"cmd": start", "camera_id":"cam1"}, {"cmd": set_threshold", "type": "pedestrian", "distance_m":3.0*}.
4. **Valid range / Units**: threshold distance: 0.5–100.0 meters; volume: 0–100 (%).
5. **Timing / Frequency**: Immediate; backend must acknowledge within 1 second.
6. **Error / End**: *ERR_INVALID_PARAMETER, CMD_ACK, CMD_NACK.*

3.1.4 Detection API (Frame-level detection output)

1. **Purpose**: Deliver detection results for each processed frame to the UI and logging subsystem.
2. **Source / Destination**: Source: backend inference engine. Destination: front end, logger, optional external clients.
3. **Format / Fields**: JSON per frame: *{"timestamp": "...", "frame_id": 123, "detections": [{"id": 1, "class": "pedestrian", "bbox": [x, y, w, h], "confidence": 0.87, "distance_m": 4.2}]}.*

4. **Valid range / Units**: bbox coordinates: integers within [0, frame_width / height]; confidence: 0.0–1.0; distance_m >= 0.0.
5. **Timing / Frequency**: Per-processed frame (matching processed FPS). End-to-end latency target specified in Performance section.
6. **Error / End**: *ERR_NO_DETECTIONS (no objects), ERR_INFERENCE_FAILURE.*

3.1.5 Visual alert output (UI overlays)

- **Purpose**: Display annotated live feed and alerts to driver/user.
- **Source / Destination**: Source: Detection API / Frontend. Destination: browser/UI or in-vehicle display.
- **Format / Fields**: Overlay graphics: bounding boxes, class labels, distance labels, color-coded threat levels, alert banners.
- **Timing / Frequency**: Rendered per processed frame; urgent alerts displayed persistently per alert policy.
- **Error / End**: *ERR_RENDER_FAIL*.

3.1.6 Auditory alert output (Speaker)

1. **Purpose**: Provide audible warnings when objects breach configured critical distances.
2. **Source / Destination**: Source: backend audio module. Destination: speaker attached to host.
3. **Format / Fields**: WAV/PCM playback or synthesized TTS phrases. Fields: alert_id, priority, message.
4. **Timing / Frequency**: Triggered immediately when threshold condition is met, subject to debounce/suppression policy (see REQs).
5. **Error / End**: *ERR_AUDIO_DEVICE_UNAVAILABLE*.

3.1.7 Optional vehicle data bus (CAN/OBD II)

1. **Purpose**: Optional source for vehicle speed, gear, and timestamp synchronization to refine distance and threat logic.
2. **Source / Destination**: Source: vehicle CAN/OBD-II bus via adapter; Destination: backend CAN handler.
3. **Format / Fields**: Standard CAN frames; parsed fields: vehicle_speed_kph, gear, timestamp.
4. **Valid range / Units**: speed 0–300 kph.
5. **Timing / Frequency**: Preferably >= 10 Hz updates.
6. **Error / End**: *ERR_CAN_UNAVAILABLE*.

3.1.8 Model update inference (Admin)

1. **Purpose**: Upload new detection model weights and metadata.

2. **Source / Destination**: Source: authenticated admin via web UI. Destination: backend model store.
3. **Format / Fields**: Binary model file (.pt/. pth), manifest. Json with model_version, checksum.
4. **Valid range / Units**: Max file size 2 GB (configurable).
5. **Timing / Frequency**: Admin-initiated; deployment occurs after integrity checks and optional validation tests.
6. **Error / End**: *ERR_MODEL_CORRUPT, MODEL_DEPLOYED*.

3.1.9 Logging/Storage

1. **Purpose**: Store detection events, system health metrics, and optional recorded clips.
2. **Source / Destination**: Source: backend modules. Destination: persistent storage (local disk / SQLite / optional network storage).
3. **Format / Fields**: detection_events table, system_health table (see DB requirements).
4. **Retention / Units**: retention policy configurable (default detection logs 90 days; video clips 30 days).
5. **Timing / Frequency**: Events logged on each detection/alert.

3.2 Functions

Each functional requirement is written as a testable statement. IDs are REQ-XXX. Grouped by feature.

3.2.1 Feature A — Video Acquisition & Preprocessing

REQ-0A1 Camera Initialization

- *Requirement:* The system shall initialize and open the configured Camera Feed and obtain the first frame within 10 seconds of a start command.
- *Verification:* Issue start command and measure time to first valid frame.

REQ-0A2 Frame Capture Rate

- *Requirement:* When the Camera Feed reports N FPS, the system shall capture and queue frames at ≥ (0.9 × N) FPS under normal load (N between 15 and 30).
- *Verification:* Log captured FPS over a 5-minute run; median ≥ 0.9×N.

REQ-0A3 Frame Validity Checks

- *Requirement:* For every acquired frame, the system shall verify frame integrity (non-null, dimensions in supported range) and discard frames failing checks, logging *ERR_FRAME_DROP*.
- *Verification:* Inject corrupted frames and confirm they are discarded and logged.

REQ-0A4 Offline Upload Acceptance

- *Requirement:* The system shall accept Video Upload files that conform to supported container/codec list and respond with UPLOAD_COMPLETE upon successful storage. *Verification:* Upload supported/unsupported files and observe responses.

3.2.2 Feature B — Detection Engine (YOLOv9 integration)

REQ-0B1 Model Load

- *Requirement:* At startup, the system shall load the configured YOLOv9 model weights and verify model integrity (checksum and successful dry-run inference on test input) before accepting live frames.
- *Verification:* Deploy known-good/bad weights; observe acceptance/rejection.

REQ-0B2 Per-frame Inference

- *Requirement:* For each input frame, the system shall run inference and produce zero or more detections containing: class, bbox, confidence, and detection_id.
- *Verification:* Confirm JSON detection entries for test frames.

REQ-0B3 Detection Classes

- *Requirement:* The system shall only report detections for classes pedestrian and vehicle (other classes may be ignored or logged separately).
- *Verification:* Input frames with other classes; verify they're not present in detection output.

REQ-0B4 Confidence Threshold

- *Requirement:* The system shall only propagate detections with confidence >= detection_confidence_threshold (threshold default 0.5; configurable).
- *Verification:* Adjust threshold and verify detection output includes only compliant entries.

REQ-0B5 Duplicate Suppression / Tracking

- *Requirement:* The system shall maintain short-term association of detections across consecutive frames to assign consistent detection_id values for persisted objects for at least 1.0 second under normal conditions.
- *Verification:* Use moving target across frames; verify consistent IDs.

REQ-0B6 Inference Failure Handling

- *Requirement:* If inference fails for a frame, the system shall (a) log *ERR_INFERENCE_FAILURE*, (b) skip generating DetectionAPI output for that frame, and (c) continue processing subsequent frames.
- *Verification:* Simulate inference failure; confirm behavior.

3.2.3 Feature C — Distance Estimation

REQ-0C1 Calibration Requirement

- *Requirement:* The system shall require camera calibration parameters (focal length and mounting height or per-camera calibration file) before enabling distance estimation. If calibration is absent, the system shall continue detection but mark distance as unavailable.
- *Verification:* Start system with/without calibration and confirm.

REQ-0C2 Distance Computation

- *Requirement:* For each detection with sufficient bounding box data and available calibration, the system shall compute and attach distance_m using the calibrated pinhole camera approximation (or equivalent method) and report distance with accuracy ±10% under test conditions defined in section 6 (test plan).
- *Verification:* Use known-distance test objects and measure error.

REQ-0C3 Distance Bounds

- *Requirement:* Computed distance_m shall be non-negative and reported to two decimal places. Distances outside configured max range (default 100 m) shall be reported as > max_range.
- *Verification:* Test with long-range objects.

3.2.4 Feature D — Alerting and Notification

REQ-0D1 Critical Distance Alert

- *Requirement:* The system shall generate a visual and auditory alert when a pedestrian detection's distance_m is ≤ critical_distance (default 3.0 m; configurable per user).
- *Verification:* Simulate object crossing threshold and confirm alert.

REQ-0D2 Alert Debounce

- *Requirement:* After generating an alert for a given detection_id, the system shall suppress repeated audible alerts for that detection for alert_debounce_period (default 2.0 s). Visual alert remains until object exits the critical zone or for a configurable persistence interval.
- *Verification:* Rapid toggling across threshold should result in one audible alert per debounce period.

REQ-0D3 Multiple Threat Arbitration

- *Requirement:* If multiple detections concurrently meet critical criteria, the system shall raise the highest-priority alert (pedestrian > vehicle) and indicate the most imminent object (smallest distance).
- *Verification:* Present simultaneous threats at different distances; verify output.

REQ-0D4 User Acknowledgement

- *Requirement:* The system shall allow a user to acknowledge an alert via UI control; acknowledgement shall silence audible alert and record user_ack event in logs.
- *Verification:* Acknowledge and verify log entry.

REQ-0D5 Alert Escalation

- *Requirement:* If a detected pedestrian remains within critical distance for more than escalation_time (default 5 s) without user acknowledgement, the system shall repeat audible alerts and escalate visual prominence (blinking banner).
- *Verification:* Leave object in critical zone for escalation time and observe behavior.

3.2.5 Feature E — User Interface and Configuration

REQ-0E1 Live View with Overlays

- *Requirement:* The system shall present a live camera view with bounding boxes, class labels, and distance readouts superimposed for each detection; overlays shall be updated at the processed frame rate.
- *Verification:* Observe overlay updates during live detection.

REQ-0E2 Settings Persistence

- *Requirement:* The system shall persist user settings (thresholds, language, volume) across restarts and allow reset to defaults.
- *Verification:* Change settings, restart, confirm persistence.

REQ-0E3 Calibration UI

- *Requirement:* The system shall provide a calibration workflow (manual entry and guided calibration using reference object) and validate calibration parameters for plausibility before enabling distance reporting.
- *Verification:* Run calibration procedure; confirm validation.

REQ-0E4 Logs & Export

- *Requirement:* The system shall allow authorized users to view recent detection logs and export them as CSV for a user-selected time window. Export shall include timestamp, class, bbox, confidence, distance_m, and alert flag.
- *Verification:* Export and inspect CSV contents.

3.2.6 Feature F — Backend and Management

REQ-0F1 RESTful API

- *Requirement:* Backend shall expose RESTful endpoints secured by authentication for: start/stop, status, model upload, settings read/write, and log retrieval. Each endpoint shall return standard HTTP status codes.
- *Verification:* API conformance tests.

REQ-0F2 Model Deployment Safety

- *Requirement:* Model uploads shall be integrity-checked (checksum) and validated with a short inference test on a canned input before being activated; failures shall roll back to the previous model version.
- *Verification:* Upload corrupted model; confirm rollback.

REQ-0F3 Health Monitoring

- *Requirement:* The system shall sample and record system health metrics (CPU%, GPU%, memory%, temperature) at least once per 5 seconds and expose them to the UI.
- *Verification:* Verify metrics collection.

REQ-0F4 Graceful Degradation

- *Requirement:* If GPU resources are insufficient, the system shall (a) log WARN_NO_GPU, (b) attempt CPU fallback at reduced frame rate, and (c) notify the user of reduced performance.
- *Verification:* Disable GPU; observe fallback.

3.2.7 Feature G — Logging, Retention and Privacy

REQ-0G1 Event Logging

- *Requirement:* The system shall record all alert events and associated detection frames to detection_events (see DB spec) with timestamp and unique IDs.
- *Verification:* Trigger alerts and verify DB entries.

REQ-0G2 Retention Policy

- *Requirement:* By default, detection logs shall be retained 90 days and video clips 30 days; retention window shall be configurable by an administrator.
- *Verification:* Confirm retention settings and automatic purge after expiry.

REQ-0G3 Data Minimization / Privacy

- *Requirement:* The system shall provide an option to disable recording of raw video (store metadata only). When raw video is stored, it shall be encrypted at rest (see Security).
- *Verification:* Toggle option and validate storage behavior and encryption.

3.3 Performance requirements

All performance requirements are measurable and tied to verifiable tests.

3.3.1 PERF-001 Minimum Real-time Throughput

- *Requirement:* The system shall process live 1280×720 frames at minimum 15 FPS end-to-end (capture → inference → distance calc → overlay) on the recommended hardware (see Design Constraints).

- *Verification:* Measure sustained FPS over a continuous 10-minute run.

3.3.2 PERF-002 Preferred Throughput

- *Requirement:* On recommended hardware (NVIDIA GPU with ≥6 GB VRAM), the system shall achieve 30 FPS for 1280×720 input under typical operating conditions.
- *Verification:* Benchmark on recommended hardware.

3.3.3 PERF-003 Inference Latency

- *Requirement:* Median model inference time shall be ≤ 100 ms per frame; 95th percentile inference time shall be ≤ 250 ms. End-to-end latency (frame capture to alert) shall be ≤ 300 ms for critical events.
- *Verification:* Profiling of inference and full pipeline latency on benchmark inputs.

3.3.4 PERF-004 Alert Latency

- *Requirement:* Audible/visual alert for a detection breaching critical distance shall be initiated within 250 ms of the detection being determined.
- *Verification:* Time measurement from frame timestamp to alert start.

3.3.5 PERF-005 Resource Utilization Limits

- *Requirement:* Under normal operation, CPU utilization across all cores shall stay below 85% and GPU memory utilization below 90%. If thresholds are exceeded, logging and graceful degradation (REQ-053) apply.
- *Verification:* System stress tests.

3.3.6 PERF-006 Startup Time

- *Requirement:* From process start, with a cached model, the system shall be ready to accept frames within 30 seconds.
- *Verification:* Measure cold start time.

3.3.7 PERF-007 Concurrent Connections

- *Requirement:* The backend shall support up to 5 concurrent frontend client connections (UI instances) without >10% degradation in FPS.
- *Verification:* Connect multiple UIs and measure FPS.

3.4 Logical Database requirements

3.4.1 Entities and Attributes

1. detection_events
    - event_id (PK, UUID), timestamp (ISO 8601), camera_id, frame_id, detections (JSON array of detection objects), alerted (boolean), user_ack (boolean), model_version, location (optional GPS).

    - Frequency: up to processed FPS × number_of_cameras; expected high write volume when running.
    - Access: append-only writes by backend; reads by admin/UI.
2. video_archive
    - video_id (PK), path, start_time, duration_s, size_bytes, encrypted (bool).
    - Retention default: 30 days.
3. system_health
    - sample_time, cpu_pct, gpu_pct, mem_pct, temp_c, status_code.
    - Sampling frequency: 5s default.
4. user_settings
    - user_id, settings_json (includes thresholds, UI preferences), last_updated.
5. model_registry
    - model_version, checksum, deployed_at, notes.

3.4.2 Integrity constraints and indices

- event_id unique; timestamp required. Index on timestamp and camera_id for efficient time-window queries.
- Detection JSON validated against defined schema (class ∈ {pedestrian, vehicle}, bbox coords in-range, confidence 0–1).
- Access controls: only authorized roles can modify user_settings, model_registry, or delete records.

3.4.3 Retention and backups

- Default retention: detection_events 90 days; video_archive 30 days. Admin may configure retention up to 365 days. Daily backup of metadata; weekly offsite backup recommended.

3.5 Design constraints

3.5.1 Platform and implementation

- Language and runtimes: Implementation shall use Python 3.8+ for backend services.
- Libraries: Must use PyTorch (GPU-enabled), OpenCV (4.x), Flask (2.x), and Ultralytics YOLOv9 repository. Exact versions and sources documented in deployment manifest.
- Containerization: The system shall provide Docker images for deployment to maximize reproducibility and portability.
- No vehicle actuation: The software shall not send control commands to vehicle actuators (brakes/steering). All outputs are advisory only (visual/audio).

3.5.2 Hardware

- Minimum hardware (for field use): 4-core CPU, 16 GB RAM, GPU with ≥6 GB VRAM (CUDA-capable), SSD storage (≥64 GB).
- Recommended hardware: 8-core CPU, 32 GB RAM, GPU with ≥8 GB VRAM (e.g., NVIDIA GTX/RTX class), NVMe SSD.

3.5.3 Third – party dependencies

- The product shall track and pin third-party dependency versions. All required drivers (GPU drivers) must be installed and compatible with the PyTorch/CUDA version.

3.5.4 Real time constraint

Real-time characteristics (PERF-001 ... PERF-004) constrain design: selection of model variant, image size, and batch size must enable these targets.

3.5.5 Standards Compliance

- **Functional safety / automotive guidance:** While this system does not control actuators, the design **shall** follow relevant guidance (e.g., ISO 26262 considerations and documented hazard analysis) when addressing safety-related features and human factors.
- **Data protection:** Data handling **shall** conform to applicable privacy regulations (e.g., GDPR where applicable): provide data minimization, access control, and user data deletion on request.
- **Transport & network:** All HTTP endpoints **shall** support TLS 1.2+; stored sensitive data **shall** be encrypted (AES-256 recommended).
- **Logging/Audit:** System **shall** retain an immutable audit trail for model deployments and administrative changes.

3.6 Software System attributes

3.6.1 Reliability

- **REQ-RLY-001:** The system shall achieve Mean Time Between Failures (MTBF) of at least 1000 hours under normal operating conditions.
- **REQ-RLY-002:** Data loss due to software faults shall be less than one event per 10,000 logged detection events.

3.6.2 Availability

- **REQ-AVL-001:** Availability during vehicle operation shall be at least 99%. Downtime for software restarts/updates shall be scheduled and limited; unplanned recovery time (MTTR) shall be ≤ 5 minutes where feasible.

3.6.3 Security

- REQ-SEC-001: All administrative API calls shall require authentication using secure credentials (e.g., token-based or certificate-based).
- REQ-SEC-002: All communications between frontend and backend shall be encrypted using TLS 1.2 or higher.
- REQ-SEC-003: Stored sensitive data (e.g., user settings, logs containing PII, video clips) shall be encrypted at rest using AES-256 or equivalent.
- REQ-SEC-004: User roles shall be defined (admin, operator, viewer), with least-privilege access enforced.
  REQ-SEC-005: Failed login attempts shall be rate-limited and logged.

3.6.4 Maintainability

- REQ-MNT-001: The system shall be modular, with separate components for acquisition, inference, alerting, logging, and UI.
- REQ-MNT-002: All configuration parameters (e.g., thresholds, model version, retention period) shall be externalized in configuration files or databases, avoiding hard-coded values.
- REQ-MNT-003: The system shall provide automated health and status monitoring endpoints to assist in troubleshooting.
- REQ-MNT-004: Codebase shall follow standard Python style guidelines (PEP8) and be documented using docstrings and inline comments.

3.6.5 Portability

- REQ-PRT-001: The system shall be deployable on Linux (Ubuntu 20.04+). Windows support is optional.
- REQ-PRT-002: The system shall provide Docker images to ensure portability across environments.
- REQ-PRT-003: The system shall support deployment both on standalone edge devices (e.g., Jetson, vehicle-mounted PC) and on cloud-based servers for offline analysis.

3.6.6 Usability

- REQ-USAB-001: The user interface shall provide an intuitive dashboard with clearly visible alerts, updated at the system frame rate.
- REQ-USAB-002: UI elements shall be operable with minimal training, requiring no more than 10 minutes of orientation for new users.
- REQ-USAB-003: Audible alerts shall be distinguishable (pedestrian vs vehicle threats) and maintain a maximum volume below 85 dB to comply with safety standards.

3.7 Organizing the specific requirements

3.7.1 System Mode

The system behaves differently based on operational mode:

- Normal Mode: Real-time detection of vehicles and pedestrians from live camera feed or uploaded videos, with distance estimation and alerts.
- Test Mode: Allows developers/testers to upload prerecorded videos or images to verify detection accuracy without generating real alerts.

3.7.2 User Class

Different users interact with the system in distinct ways:

- Driver/User: Uses the interface to receive alerts and view detection results (live feed, uploaded media, statistics).
- System Administrator/Developer: Configures YOLOv9 model parameters, manages backend services, and maintains threshold settings for alerts.

3.7.3 Objects

The primary real-world objects represented in the system include:

- Vehicle: Bounding box, class label, distance from camera.
- Pedestrian: Bounding box, class label, distance from camera.
- Camera Feed/Video Frame: Input object containing real-time or prerecorded data for analysis.
- Alert: Object generated when pedestrians/vehicles are detected within a critical distance threshold.

3.7.4 Features

The system's key features are organized as follows:

- Real-Time Detection: Continuous identification of vehicles and pedestrians in each frame.
- Distance Estimation: Approximation of distance to detected objects using bounding box dimensions.
- Alert System: Visual and auditory notifications triggered when a pedestrian or vehicle is within a danger zone.
- Frontend Dashboard: Uploading videos, monitoring live feeds, and viewing detection statistics.

3.7.5 Stimulus

System functions are triggered by the following stimuli:

- Visual Stimuli: Input from live video stream or uploaded video/image.
- Detection Stimuli: Identification of an object classified as “vehicle” or “pedestrian.”
- Proximity Stimuli: Object detected within the critical distance threshold, triggering alerts.

3.7.6 Response

For each stimulus, the system produces responses:

- Detection Response: Display bounding boxes and class labels on the frontend in real time.
- Alert Response: Trigger visual/auditory alerts for nearby pedestrians or vehicles.
- Logging Response: Record detection events and statistics for monitoring and analysis.

3.7.7 Functional Hierarchy

The system can also be organized into a functional hierarchy:

1. Input Layer: Capture video frames (camera feed or uploaded media).
2. Processing Layer: Run YOLOv9 object detection and distance estimation.
3. Decision Layer: Compare distance with threshold values to determine risk.
4. Output Layer: Display detection results, generate alerts, and update statistics on the frontend.

3.8 Additional Comments

In preparing the SRS for the Vehicular Detection of Pedestrians system, multiple organizational techniques have been applied to optimize clarity and traceability of requirements. Specifically, the requirements have been structured by:

- System Mode (normal vs. test operation),
- User Class (driver/end-user vs. administrator/developer),
- Objects (vehicles, pedestrians, video frames, alerts),
- Features (detection, distance estimation, alerting, dashboard), and
- Functional Hierarchy (input, processing, decision, output layers).

This multi-perspective organization ensures that requirements are easily understood by both technical and non-technical stakeholders. For example:

- Finite State Diagrams may be applied to represent the transitions between system modes (e.g., idle, detection, alert).
- Stimulus-Response Models may be used to specify the system's behavior when a pedestrian enters the danger zone.
- Object-Oriented Analysis provides a structured way to define attributes and interactions of detected entities (vehicles, pedestrians, alerts).
- Data Flow Diagrams (DFDs) and data dictionaries are suitable for modelling the flow of video input, processing through YOLOv9, and output of detection results.

4. Index

    3. Feature C — Distance Estimation (REQ-0C1 to REQ-0C3)
    4. Feature D — Alerting and Notification (REQ-0D1 to REQ-0D5)
    5. Feature E — User Interface & Configuration (REQ-0E1 to REQ-0E4)
    6. Feature F — Backend & Management (REQ-0F1 to REQ-0F4)
    7. Feature G — Logging, Retention, and Privacy (REQ-0G1 to REQ-0G3)

3. Performance requirements (PERF-001 to PERF-007)
4. Logical database requirements

    1. Entities and Attributes
    2. Integrity Constraints & Indices
    3. Retention & Backups

5. Design constraints

    1. Platform and Implementation
    2. Hardware
    3. Third-party Dependencies
    4. Real-time Constraints
    5. Standards Compliance

6. Software system attributes

    1. Reliability
    2. Availability
    3. Security
    4. Maintainability
    5. Portability

7. Organizing the specific requirements

    1. System Mode
    2. User Class
    3. Objects
    4. Feature
    5. Stimulus
    6. Response
    7. Functional Hierarchy

8. Additional comments

5. Appendix

The appendix in this SRS is not a part of the requirements

5.1 References

The following documents, research articles, and online resources have been referred to in preparing this Software Requirements Specification (SRS):

1. IEEE Standard for Software Requirements Specifications (IEEE Std 830-1998).
2. Redmon, J., & Farhadi, A. (2016). *You Only Look Once: Unified, Real-Time Object Detection.* Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).
3. YOLOv9: A Leap Forward in Object Detection Technology (Ultralytics YoloV9 Documentation)
4. OpenCV Documentation. *Open Source Computer Vision Library.*
5. Flask Documentation. *Flask Web Framework.*
6. National Highway Traffic Safety Administration (NHTSA). *Traffic Safety Facts: Pedestrians.*

5.2 Supplementary diagrams

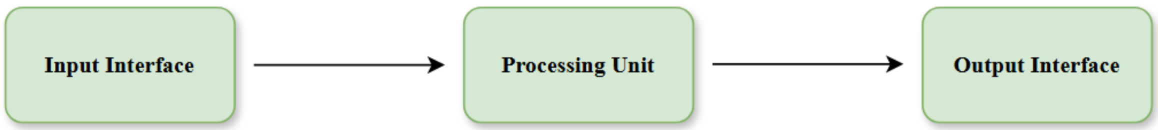

Fig. 1.1 Conceptual Block Diagram

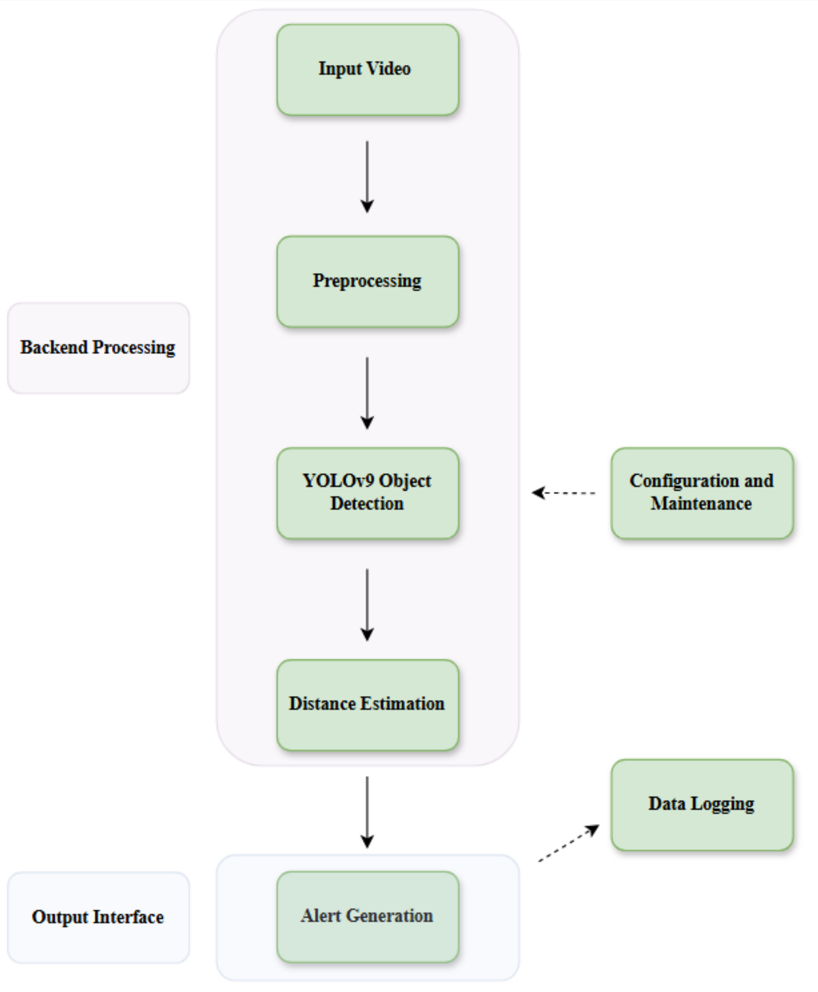

Fig. 1.2 Function Flow Diagram