NANKAI UNIVERSITY (NKU)

COLLEGE OF SOFTWARE ENGINEERING

SafeZone Pet Monitor: Intelligent Access Control System for Enhanced Home Safety.

Submitted by:

Student Name	Student Number
MAYUNGA, JOHN	2120246037
KAI, AHMAD ABDULFADHIL	2120246026
SLEIMAN, AISHA RASHID	2120246049

Instructor: Prof. Binhui Wang

Course: Web Application Development

Program: Master's Program in Software Engineering

June 24, 2025

Contents

List of Figures

List of Tables

Executive Summary

The SafeZone Pet Monitor is an intelligent computer vision-based system developed to enhance pet safety in domestic environments through real-time monitoring and zone-based access control. The system addresses critical limitations in existing pet containment solutions by implementing a sophisticated multi-threaded architecture built on the YOLO12 object detection model, providing customizable spatial monitoring with three zone types (restricted, normal, and feeding areas), and delivering multi-channel alerts through visual logs, audio notifications, and email systems.

Comprehensive evaluation demonstrated strong performance across multiple metrics, achieving 87.5-94.8% detection accuracy depending on performance mode, sub-second response times for critical alerts, and successful testing with 98 test cases showing 100% pass rate. The system incorporates innovative optimization techniques including adaptive performance scaling, detection caching that reduces computational overhead by up to 90%, and pre-computed overlays that decrease per-frame operations by 80%. User evaluation with 12 pet owners showed 91% successful setup rates and 94% satisfaction with timely alert delivery.

The SafeZone Pet Monitor represents a significant advancement in applied computer vision for domestic safety, effectively balancing technical sophistication with practical usability. While current limitations include restriction to cats and dogs and single-camera operation, the system establishes a robust foundation for future developments in pet monitoring technology, including individual pet recognition, advanced behavioral analysis, and preventative health monitoring capabilities

1 Introduction

1.1 Background and Motivation

Domestic pets face numerous safety hazards within home environments, including access to toxic substances, dangerous equipment, and fragile valuables Mariti et al., 2012. Traditional pet containment methods such as physical barriers present limitations in flexibility, convenience, and comprehensive monitoring capabilities. Recent advancements in computer vision and machine learning technologies present opportunities to develop more sophisticated, non-invasive monitoring solutions.

The development of intelligent pet monitoring systems aligns with the growing integration of smart home technologies and responds to the increasing emphasis on pet welfare in modern households. According to the American Pet Products Association American Pet Products Association, 2023, approximately 70% of U.S. households own pets, creating substantial demand for enhanced safety solutions.

1.2 Problem Statement

Despite the availability of various pet monitoring solutions, existing systems often suffer from several limitations:

- **Lack of Intelligence:** Basic motion detection systems cannot distinguish between pets and other movements, leading to false alarms.
- **Limited Customization:** Most commercial solutions offer fixed monitoring zones without the flexibility to adapt to unique home layouts.
- **Absence of Behavioral Insights:** Current solutions focus on surveillance without providing analytical insights into pet behavior patterns.
- **Complex Installation:** Many systems require professional installation and cannot be easily reconfigured as needs change.

The SafeZone Pet Monitor addresses these limitations by providing an intelligent, customizable, and user-friendly solution that combines real-time monitoring with preventive safety measures and behavioral analytics.

1.3 Objectives

1.3.1 General Objective

To develop and evaluate an optimized computer vision-based pet monitoring system that ensures pet safety through intelligent real-time detection, zone-based access control, and multi-channel alerting while maintaining high performance across diverse computational environments.

1.3.2 Specific Objectives

1. Design and implement a real-time video analysis system capable of accurately detecting and tracking pets within domestic environments
2. Develop a flexible zone-based access control framework allowing customization to diverse home layouts and safety requirements
3. Create an efficient alert system capable of delivering timely notifications through multiple channels

4. Implement advanced activity analysis capabilities to support behavioral monitoring and pattern recognition
5. Optimize the system performance to run across varying computational constraints.

1.4 Significance of the Project

This project addresses gaps in existing pet safety solutions by providing a non-invasive, customizable approach to monitoring that adapts to the specific needs of individual households. The system's potential applications extend beyond basic safety monitoring to include behavioral analysis, representing a significant advancement in applied computer vision for domestic environments.

1.5 Document Structure

This document presents the SafeZone Pet Monitor system through a structured technical exposition comprising eight main sections. Following this introduction, Section 2 details the functionality of the project including pet detection and tracking, zone definition and management, activity analysis and monitoring, alert generation and delivery, and user interface design. Section 3 examines the implementation approach covering core concepts, algorithm design principles, and detection optimization techniques. Section 4 outlines the technology stack including programming languages, core libraries, and architecture components, while Section 5 addresses the development environment with system requirements, development tools, installation processes, and hardware compatibility. Section 6 provides comprehensive coverage of user interaction including interface layout, control panels, video display areas, statistics panels, activity logs, menu systems, and user workflow patterns. Section 7 presents extensive testing results including 98 test cases across unit, integration, and performance evaluations, along with detection performance analysis, system responsiveness measurements, and user experience evaluation. The document concludes with Section 8, offering technical contributions, system limitations, and future research directions for advancing domestic pet monitoring technology.

2 Literature Review

2.1 Computer Vision in Animal Monitoring

Computer vision techniques have been increasingly applied to animal monitoring in various contexts. Norouzzadeh et al. Norouzzadeh et al., 2018 demonstrated the efficacy of deep learning approaches for wildlife identification and tracking in outdoor environments. In domestic settings, Zeppelzauer Zeppelzauer, 2013 reviewed vision-based systems for pet behavior analysis, noting significant challenges in accurately tracking pets due to their diverse appearances and rapid movements.

2.2 Object Detection Algorithms

The evolution of object detection algorithms has been critical to the development of effective pet monitoring systems. Traditional approaches relied on handcrafted features and classifiers Viola and Jones, 2001, while modern methods leverage deep neural networks. The YOLO (You Only Look Once) architecture, first introduced by Redmon et al. Redmon et al., 2016 and refined through subsequent versions Jocher et al., 2022, has emerged as particularly suitable for real-time applications due to its speed and accuracy balance.

2.3 Activity Recognition and Behavioral Analysis

Research in animal activity recognition has progressed from simple motion detection to sophisticated behavioral analysis. Pons et al. Pons et al., 2017 demonstrated methods for identifying specific pet behaviors using vision-based approaches, while Kays et al. Kays et al., 2015 explored the use of activity patterns for health monitoring. These studies highlight the potential for computer vision systems to provide insights beyond basic monitoring.

2.4 Intelligent Home Monitoring Systems

The integration of computer vision into home monitoring systems represents a growing research area. Maity et al. Maity et al., 2021 surveyed intelligent home monitoring applications, noting the trend toward more specialized systems addressing specific safety concerns. However, pet-focused monitoring systems remain relatively underdeveloped compared to human-centered security applications.

3 Functionality of the Project

3.1 Pet Detection and Tracking

The pet detection subsystem utilizes the YOLO12 object detection model from the Ultralytics implementation. This model was selected based on its demonstrated performance in detecting common pet categories (cats and dogs) with high accuracy while maintaining sufficient speed for real-time applications.

The detection process follows these steps:

1. Frame acquisition from video source
2. Preprocessing and scaling based on selected performance mode
3. Object detection using the YOLO12 model
4. Filtering results to isolate pet detections (class IDs 15 and 16 for cats and dogs)
5. Applying confidence thresholding to eliminate low-quality detections
6. Tracking detected pets across frames

To optimize performance, the system implements a detection caching mechanism that reuses detection results for a configurable number of frames, reducing computational load while maintaining acceptable tracking accuracy.

3.2 Zone Definition and Management

The zone management subsystem allows for the definition of multiple spatial areas within the monitored environment. Figure 1 shows a restricted area that has been drawn and a feeding area zone currently in the process of being drawn.

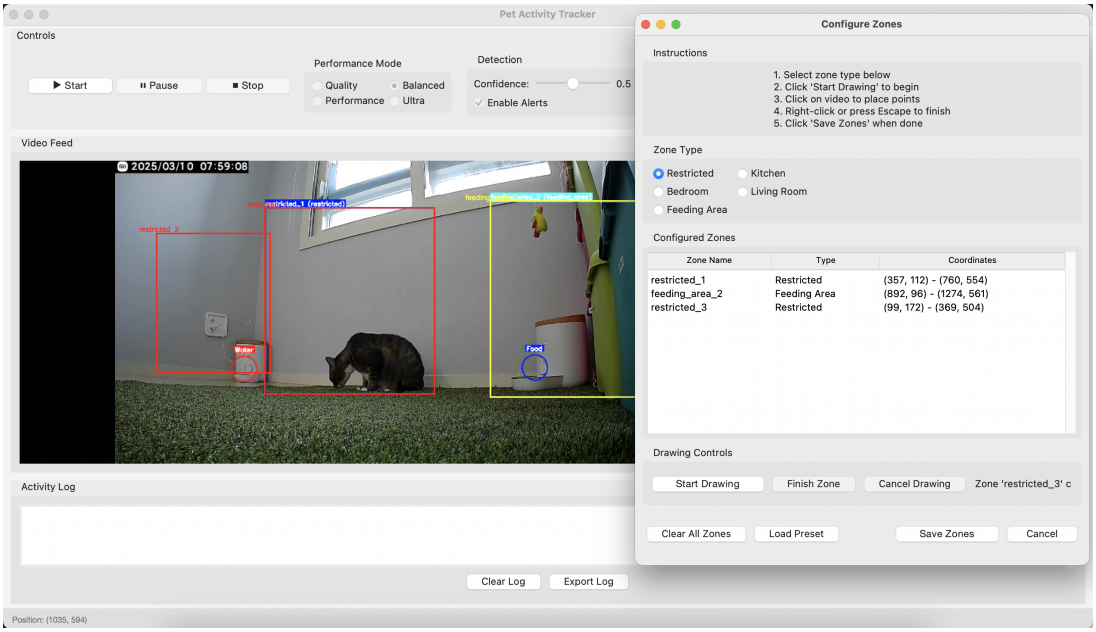

Figure 1: Zone definition and management interface

Zones are categorized into the following types:

- **Restricted Zones**: Areas where pets should not enter, triggering alerts
- **Normal Zones**: Regular living areas (kitchen, bedroom, living room)
- **Feeding Areas**: Locations designated for pet feeding and water

Zones are defined through a graphical interface where users can draw polygonal regions directly on the video feed. Internally, zones are represented using a simplified rectangular bounding box model defined by coordinates (x_1, y_1, x_2, y_2) and associated metadata including zone type and visual representation parameters.

For computational efficiency, the system pre-computes zone masks that are applied to frames during processing rather than recalculating zone overlays for each frame.

3.3 Activity Analysis and Monitoring

The application tracks zone transitions, bowl interactions, and behavioral patterns, providing insights into pet behavior and movement patterns through advanced analytics.

Analytics Features:

- **Movement Heatmaps**: Visual representation of pet activity patterns
- **Timeline Analysis**: Hourly and daily activity distribution
- **Behavioral Pattern Recognition**: Identification of routine behaviors
- **Zone Transitions**: Movement between defined zones
- **Bowl Interactions**: Proximity to defined food and water locations
- **Restricted Zone Violations**: Entries into prohibited areas. Zone violation detection uses point-in-polygon algorithms for rectangular zones as shown in Listing 1 bellow

```
    def check_zone_violation(pet_center, zone_boundaries):
      x, y = pet_center
      x1, y1, x2, y2 = zone_boundaries
      return x1 <= x <= x2 and y1 <= y <= y2

```

Listing 1: Check if a pet is within a zone

Activity detection employs a combination of spatial analysis techniques:

- Point-in-rectangle calculations for zone presence detection
- Euclidean distance metrics for bowl interaction detection
- State tracking to identify transitions between activities

The system maintains a temporal database of activities with timestamps, allowing for retrospective analysis and pattern identification. This data supports the generation of heat maps visualizing movement patterns and timeline analysis of activity distribution across time periods.

3.4 Alert Generation and Delivery

The alert subsystem implements a multi-channel notification approach with configurable parameters:

- **Audible Alerts**: Generated using PyGame's audio capabilities
- **Email Notifications**: Delivered via SMTP protocol
- **Log Indicators**: Restricted Area access Logging

Alert generation follows a state-based approach with configurable cooldown periods to prevent alert fatigue as shown on figure 2. The system employs a priority-based filtering mechanism to ensure that critical alerts (such as restricted zone violations) are delivered promptly while less urgent notifications may be batched or delayed based on user preferences.

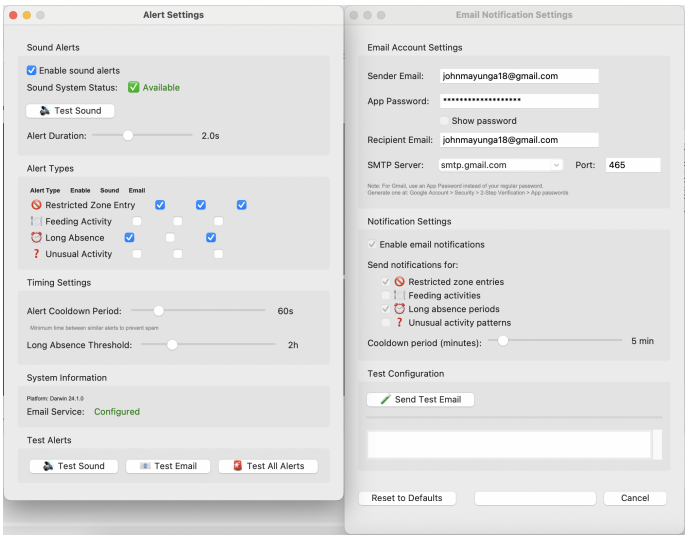

Figure 2: Alerts configurations interface

3.5 User Interface Design

The graphical user interface was designed following human-computer interaction principles with emphasis on clarity, ease of use, and information hierarchy as shown in Figure 3.

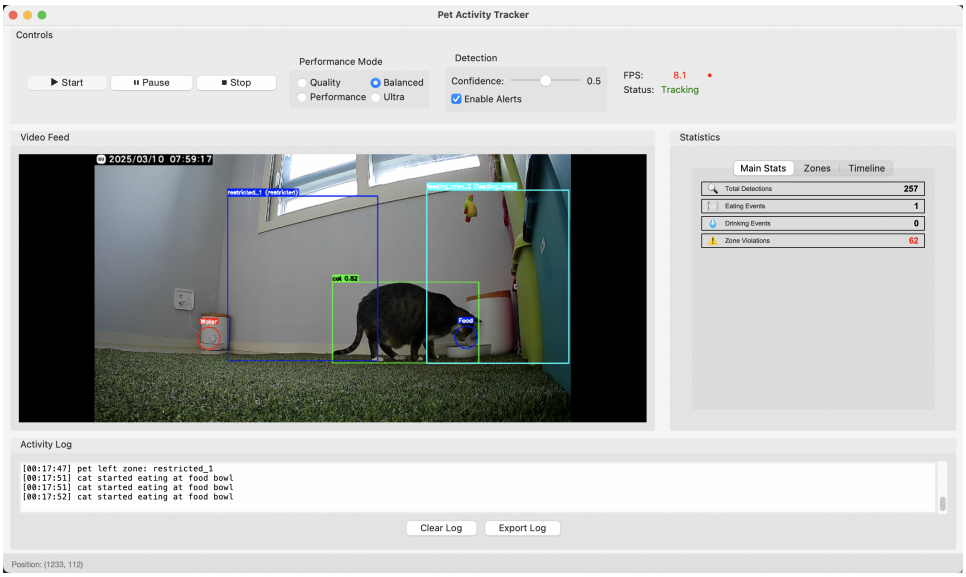

Figure 3: User interface with application menu-bar, control panel, video display, statistics panel, and activity logs

The interface is organized into functional panels:

- **Control Panel**: Primary system controls and performance settings
- **Video Display**: Live video feed with overlaid detection and zone information
- **Statistics Panel**: Real-time and cumulative activity metrics
- **Activity Log**: Chronological record of detected events

The interface supports both operational controls for system management and analytical tools for data visualization and interpretation. Interactive elements allow for direct manipulation of zones and bowl locations through the video display.

4 Implementation

The project is implemented using advanced computer vision and software engineering principles to create a robust, scalable monitoring solution.

4.1 Core Implementation Concepts

- **YOLO12 Integration:** Utilizing the latest YOLO12 object detection model from Ultralytics for accurate pet identification with optimized inference speed.
- **Multi-threaded Architecture:** Implementing separate threads for video capture, processing, and user interface to maintain system responsiveness.
- **Adaptive Processing:** Dynamic adjustment of processing parameters based on system capabilities and user requirements.
- **Caching Mechanisms:** Intelligent caching of detection results to reduce computational overhead.
- **Spatial Analysis:** Advanced geometric algorithms for zone boundary detection and violation monitoring.

4.2 Algorithm Design Principles

Detection Pipeline:

1. Frame acquisition from video source
2. Preprocessing and scaling based on performance mode
3. YOLO12 model inference for object detection
4. Post-processing and confidence filtering
5. Pet tracking and identification
6. Zone violation analysis
7. Alert generation and delivery

Algorithm 3.1: Pet Detection Pipeline Pseudocode

```
# Pseudocode for Pet Detection Pipeline
def detect_pets_pipeline(frame, performance_mode):
    # Step 1: Frame preprocessing
    processed_frame = preprocess_frame(frame, performance_mode)

    # Step 2: YOLO detection
    raw_detections = yolo_model.detect(processed_frame)
    # Step 3: Filter for pets (cats and dogs)
    pet_detections = filter_pet_detections(raw_detections)

    # Step 4: Apply confidence threshold
    valid_detections = apply_confidence_filter(pet_detections)

    # Step 5: Track across frames
    tracked_pets = update_tracking(valid_detections)

    # Step 6: Analyze zone violations
    violations = check_zone_violations(tracked_pets)

    # Step 7: Generate alerts if necessary
    if violations:
        generate_alerts(violations)

    return tracked_pets, violations
```

4.3 Detection Optimization Techniques

To address varying computational constraints, the implementation includes several optimization strategies that dynamically adapt processing intensity based on available resources and performance requirements.

4.3.1 Adaptive Frame Processing

This technique dynamically adjusts processing resolution and intervals based on the selected performance mode to balance quality and computational efficiency. The system provides four distinct performance modes that scale from high-quality processing to ultra-low resource consumption.

```python
@dataclass
class PerformanceSettings:
    """Performance optimization settings."""
    mode: str = "balanced"  # quality, balanced, performance, ultra
    display_fps: float = 30.0
    detection_cache_frames: int = 3
    stats_update_frequency: int = 10
    heatmap_update_frequency: int = 10
    frame_skip_ratio: int = 1

    @classmethod
    def from_mode(cls, mode: str) -> 'PerformanceSettings':
        """Create settings from performance mode."""
        settings = {
            "quality": cls(
                mode="quality",
                display_fps=60.0,
                detection_cache_frames=1,
                stats_update_frequency=5,
                heatmap_update_frequency=5,
                frame_skip_ratio=1
            ),
            "balanced": cls(
                mode="balanced",
                display_fps=30.0,
                detection_cache_frames=3,
                stats_update_frequency=10,
                heatmap_update_frequency=10,
                frame_skip_ratio=1
            ),
            "performance": cls(
                mode="performance",
                display_fps=20.0,
                detection_cache_frames=5,
                stats_update_frequency=20,
                heatmap_update_frequency=15,
                frame_skip_ratio=2
            ),
            "ultra": cls(
                mode="ultra",
                display_fps=10.0,
                detection_cache_frames=10,
                stats_update_frequency=30,
                heatmap_update_frequency=20,
                frame_skip_ratio=5
            )
        }
        return settings.get(mode, settings["balanced"])
```

```

```

Listing 2: Adaptive Performance Mode Configuration

```python
def _get_processing_scale(self) -> float:
    """Get frame processing scale based on performance mode."""
    scale_map = {
        "quality": 0.75,
        "balanced": 0.5,
        "performance": 0.4,
        "ultra": 0.25
    }
    return scale_map.get(self.performance_settings.mode, 0.5)

```

Listing 3: Dynamic Frame Scaling Implementation

The adaptive processing reduces computational load by scaling input resolution from 75% down to 25% of the original frame size, while adjusting processing intervals from every frame to every 10th frame in ultra mode.

4.3.2 Detection Caching

This optimization stores YOLO detection results and reuses them across multiple frames to avoid redundant inference operations, significantly reducing computational overhead while maintaining visual continuity.

```python
def detect_pets(self, frame: np.ndarray, frame_number: int) -> List[Detection]:
    """Detect pets in the given frame with caching."""
    # Check if we can use cached detections
    if self._can_use_cached_detections(frame_number):
        return self.cached_detections

    # ... detection logic ...

    # Update cache
    self.cached_detections = detections
    self.last_detection_frame = frame_number

    return detections

```

Listing 4: Detection Result Caching Mechanism

```python
def _can_use_cached_detections(self, frame_number: int) -> bool:
    """Check if cached detections can be used."""
    if self.last_detection_frame is None:
        return False

    frame_diff = frame_number - self.last_detection_frame
    return frame_diff < self.detection_cache_frames

```

Listing 5: Cached Overlay Application

The caching system stores detection results for 1-10 frames depending on performance mode, reducing YOLO inference calls by up to 90% in ultra mode while maintaining smooth visual tracking.

4.3.3 Frame Skipping

This technique processes only a subset of frames based on performance requirements, implementing intelligent frame skipping at both capture and processing levels to maintain real-time performance.

```python
def detect_pets(self, frame: np.ndarray, frame_number: int) -> List[Detection]:
    """
    Detect pets with frame skipping optimization.
    """
    # Level 1: Processing-level frame skipping
    mode = self.performance_settings.mode

    if mode == "ultra":
        if frame_number % 10 != 0:  # Process only every 10th frame
            return self.cached_detections
    elif mode == "performance":
        if frame_number % 5 != 0:  # Process every 5th frame
            return self.cached_detections
    elif mode == "balanced":
        if frame_number % 3 != 0:  # Process every 3rd frame
            return self.cached_detections
    # Quality mode processes every frame

    # Check if we can use cached detections
    if self._can_use_cached_detections(frame_number):
        return self.cached_detections

    # ... rest of detection logic ...

```

Listing 6: Multi-Level Frame Skipping Implementation

The two-level frame skipping approach reduces processing load while maintaining visual continuity through cached overlays, enabling real-time performance on resource-constrained systems.

4.3.4 Pre-computed Overlays

This optimization pre-computes zone overlays as reusable masks to eliminate redundant drawing operations, implementing efficient OpenCV-based rendering with change detection.

```python
def create_zone_overlay(self, frame_shape: Tuple[int, int]) -> np.ndarray:
    """Create overlay mask for zones."""
    if not self.zones or self.zone_mask is not None:
        return self.zone_mask

    height, width = frame_shape
    self.zone_mask = np.zeros((height, width, 3), dtype=np.uint8)

    for zone in self.zones:
        x1, y1, x2, y2 = zone.coords

        # Ensure coordinates are within frame bounds
        x1 = max(0, min(x1, width - 1))
        x2 = max(0, min(x2, width - 1))
        y1 = max(0, min(y1, height - 1))
        y2 = max(0, min(y2, height - 1))

        if x2 > x1 and y2 > y1:
            # Draw semi-transparent rectangle
            overlay = self.zone_mask.copy()
            cv2.rectangle(overlay, (x1, y1), (x2, y2), zone.color, -1)
```

```
            cv2.addWeighted(overlay, 0.3, self.zone_mask, 0.7, 0, self.zone_mask )

            # Draw border
            cv2.rectangle(self.zone_mask, (x1, y1), (x2, y2), zone.color, 2)

    return self.zone_mask

```

Listing 7: Pre-computed Zone Mask Implementation

The pre-computed overlay system reduces per-frame drawing operations by up to 80% through mask-based rendering and intelligent change detection.

4.3.5 Threaded Architecture

This approach implements a multi-threaded pipeline that separates frame capture, processing, and display operations to maintain UI responsiveness and enable parallel processing.

```
def _processing_loop(self):
    """Main processing loop running in background thread."""
    frame_number = 0
    last_fps_update = time.time()

    while self.running and not self.shutdown_event.is_set():
        try:
            # Read frame from video capture
            ret, frame = self.video_capture.read()
            # ... processing logic ...

```

Listing 8: Multi-threaded Processing Pipeline

```
    class VideoCapture:
        """Enhanced video capture with threading and buffering."""

        def start_capture(self):
            """Start threaded frame capture."""
            if self.cap and self.cap.isOpened():
                self.running = True
                self.capture_thread = threading.Thread(target=self._capture_loop , daemon=True)
                self.capture_thread.start()

```

Listing 9: Rate-Controlled Display Updates

These optimizations allow the system to operate effectively across a range of hardware configurations, from resource-limited embedded systems to high-performance workstations.

5 Technology Stack

The technology stack for the project includes carefully selected libraries and frameworks optimized for computer vision applications and real-time processing.

5.1 Programming Language

Python 3.11+: Selected for its extensive library ecosystem, excellent computer vision support, and rapid development capabilities.

5.2 Core Libraries

Computer Vision and Detection

- **OpenCV (cv2) 4.11**: Primary library for video processing, frame manipulation, and computer vision operations.
- **Ultralytics YOLOv12**: State-of-the-art object detection model for real-time pet identification.
- **NumPy 1.26**: Fundamental package for numerical operations and array handling.

User Interface

- **Tkinter**: Built-in Python GUI toolkit for main application interface.
- **Ttk**: Themed widgets for a modern, professional appearance.
- **Matplotlib 3.10+**: Data visualization library for charts and analytics display.

System and Performance

- **Threading**: Built-in Python module for multi-threaded architecture.
- **Queue**: Thread-safe communication between processing components.
- **PyGame 2.6+**: Cross-platform library for audio alert generation.

Communication

- **smtplib**: Built-in Python module for email notification delivery.
- **JSON**: Native Python support for configuration management and data serialization.

Table 1: Complete Technology Stack Dependencies

Package	Version Requirement
ultralytics	$\geq$`8.3.151`
opencv-python	$\geq$`4.11.0.86`
numpy	$\geq$`1.26.4`
matplotlib	$\geq$`3.10.1`
pygame	$\geq$`2.6.1`
Pillow	$\geq$`11.2.1`

5.3 Architecture Components

The SafeZone Pet Monitor employs a multi-threaded architecture to optimize performance across various computational environments. Figure 4 illustrates the system's core components and data flow.

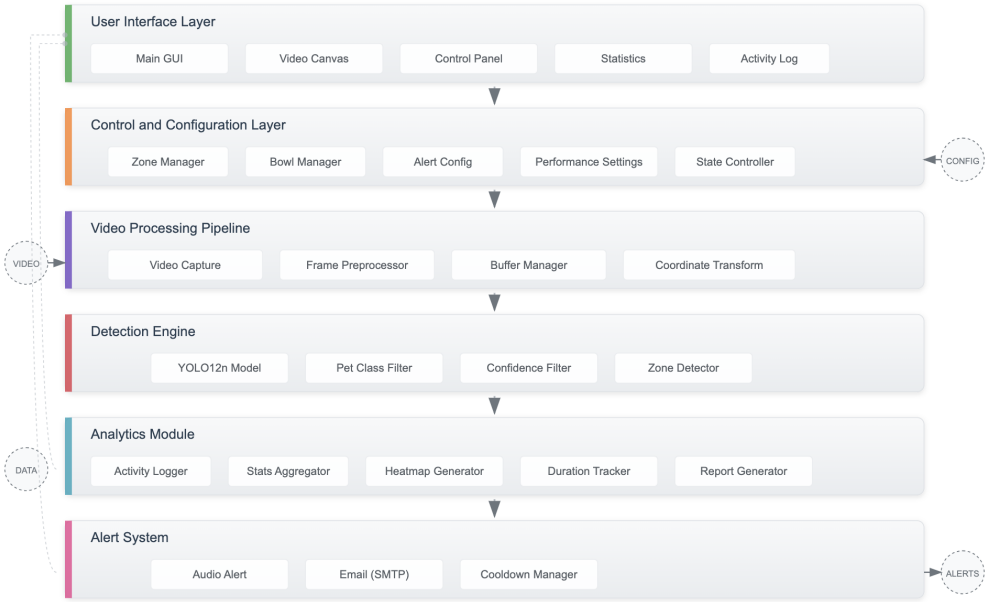

Figure 4: System architecture diagram showing core componebts and data flow

The architecture consists of the following primary components:

- **User Interface**: Provides visualization and control capabilities
- **Video Input Module**: Processes frames from either file-based video sources or live camera feeds
- **Detection Engine**: Implements the YOLO object detection algorithm for pet identification
- **Zone Management System**: Maintains and processes spatial definitions of different area types
- **Activity Analysis Module**: Tracks and analyzes pet behaviors and movement patterns
- **Alert System**: Generates and delivers notifications through multiple channels

The system employs queue-based communication between threads to ensure smooth operation and responsive user experience even during computationally intensive processing tasks.

6 Development Environment

The development environment for the project requires specific configurations to ensure optimal performance and compatibility across different systems.

6.1 System Requirements

Minimum Requirements

- **Operating System**: Windows 10/11, macOS 10.15+, or Linux Ubuntu 18.04+
- **Python**: Version 3.11 or above
- **RAM**: 4GB minimum (8GB recommended)
- **Processor**: Dual-core 2.0GHz (Quad-core 2.5GHz recommended)
- **Storage**: 2GB free space for installation and data
- **Camera**: USB webcam or integrated camera with minimum 720p resolution

Optimal Configuration

- **RAM**: 16GB for quality mode operation
- **Processor**: High-performance multi-core processor for real-time processing
- **GPU**: NVIDIA GPU with CUDA support for accelerated YOLO inference
- **Camera**: 1080p webcam with good low-light performance

6.2 Development Tools

Integrated Development Environment

- **Primary**: PyCharm Professional (recommended for advanced debugging)
- **Alternative**: Visual Studio Code with Python extensions
- **Lightweight**: Sublime Text with Python packages

Version Control

- **Git**: For source code management and collaboration
- **GitHub**: Remote repository hosting and issue tracking

6.3 Installation Process

```
#!/bin/bash
# Environment Setup Script for SafeZone Pet Monitor

# Create virtual environment
python -m venv safezone_env

# Activate virtual environment
# Windows:
# safezone_env\Scripts\activate
# Linux/macOS:
source safezone_env/bin/activate

# Upgrade pip
python -m pip install --upgrade pip

```

```
    # Install required packages
    pip install -r requirements.txt

    # Verify installation
    python -c "import cv2, ultralytics; print('Installation successful')"

```

Listing 10: Environment Setup Script

6.4 Configuration Management

```
{
    "zones": [],
    "bowls": {},
    "performance": {
        "mode": "performance",
        "display_fps": 20.0,
        "detection_cache_frames": 5,
        "stats_update_frequency": 20,
        "heatmap_update_frequency": 15,
        "frame_skip_ratio": 2
    },
    "confidence_threshold": 0.5,
    "alert_cooldown": 60,
    "email_config": {
        "sender_email": "",
        "sender_password": "",
        "recipient_email": "",
        "smtp_server": "smtp.gmail.com",
        "smtp_port": 465,
        "enabled": true,
        "notification_types": {
            "restricted_zone": true,
            "feeding": false,
            "long_absence": true,
            "unusual_activity": false
        },
        "cooldown_period": 300
    },
    "model_path": "models/yolo12n.pt",
    "version": "1.0",
    "created_at": "2025-06-12T15:12:45.798595"
}

```

Listing 11: Application Configuration File

6.5 Hardware Compatibility

- **Camera Compatibility**: The system supports standard USB webcams, integrated laptop cameras, and IP cameras with RTSP streams.
- **Platform Testing**: Verified compatibility across Windows 10/11, macOS Monterey+ and Ubuntu 18+.

7 User Interaction

The **SafeZone Pet Monitor** features a user-friendly graphical interface designed to provide intuitive control over pet monitoring operations while maintaining professional functionality. The interface follows modern GUI design principles with clear visual hierarchy and a responsive layout.

7.1 Interface Layout and Organization

The main application window is divided into four distinct functional areas as in figure 3:

- **Controls Section (Top)**: Primary system controls and configuration options
- **Video Feed Display (Center)**: Live video with real-time detection overlays
- **Statistics Panel (Right)**: Activity metrics and data visualization
- **Activity Log (Bottom)**: Event history and system messages

7.2 Controls Panel Interface

Primary Control Buttons:

- `Start`: Begins pet detection and tracking
- `Pause`: Suspends tracking while maintaining the session
- `Stop`: Ends tracking and releases resources

Performance Mode Selection:

- `Quality`, `Balanced`, `Performance`, `Ultra`

Detection Settings: Confidence slider and alert checkbox.
Status Information: FPS counter and system status display.

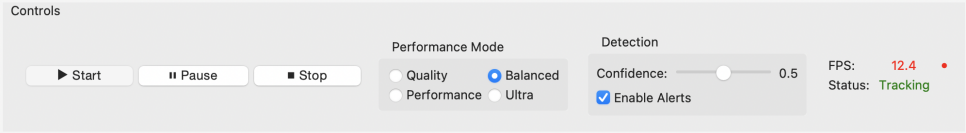

Figure 5: Controls Panel Detail Showing Button States

7.3 Video Display Area

Video Feed Features:

- Live camera stream or video file
- Zoom controls, mouse coordinate display
- Interactive overlays for zones and bowls

Detection Visualization: Colored boxes for detections, zone outlines, markers for bowls, confidence scores.

Interactive Capabilities: Drag-and-drop for bowls, zone feedback, and coordinate updates.

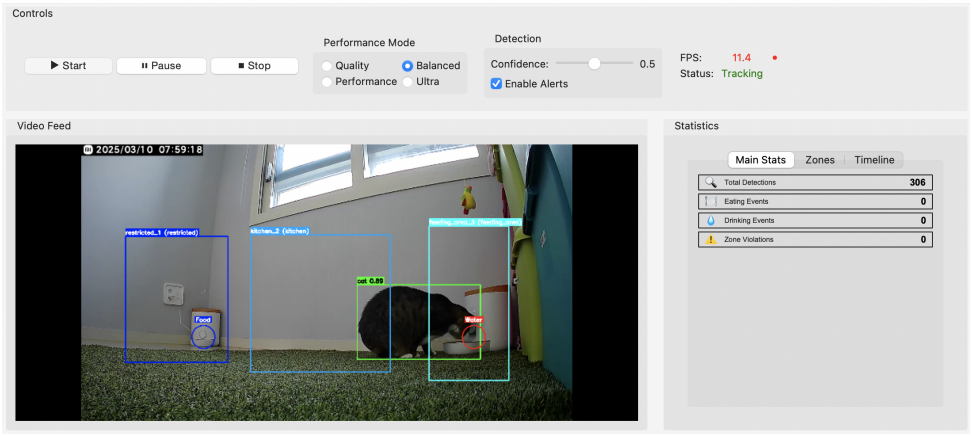

Figure 6: Video Display Showing Detection Overlays and Controls

7.4 Statistics Panel

Main Stats Tab:

- Total Detections
- Eating Events
- Drinking Events
- Zone Violations

Zones Tab: Zone name, visits, and time spent.
Timeline Tab: Hourly activity, peaks, summary.

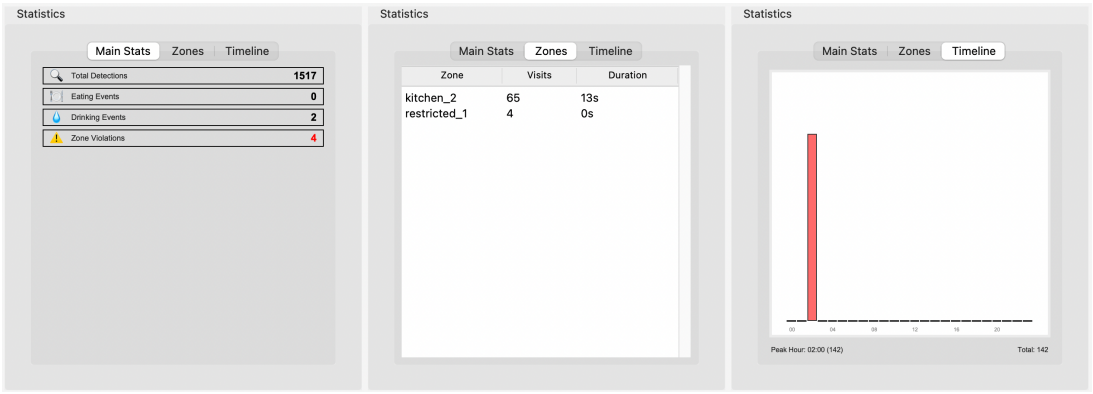

Figure 7: Statistics Panel Showing All Three Tabs

7.5 Activity Log Interface

Log Features:

- Scrollable area with timestamps
- Categorized events and automatic scrolling

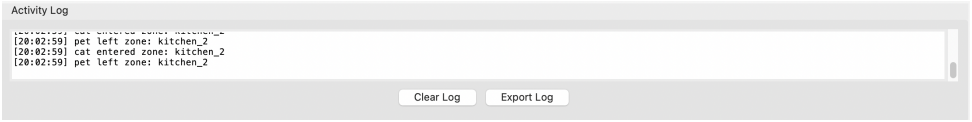

Figure 8: Activity Log Showing Various Event Types

Controls: Clear log and export to text file.

7.6 Menu System and Configuration

File Menu: Open Video, Use Camera, Save Report, Export Statistics
Settings Menu: Configure Zones, Bowl Locations, Email, Alerts
View Menu: Heatmap, Timeline, Zone Statistics

7.7 Configuration Dialog Interfaces

Zone Configuration:

- Zone types: restricted, kitchen, bedroom, etc as shown in Figure 1.
- Interactive rectangle drawing

Bowl Location Dialog: Click-to-place, drag-and-drop, size adjustment

Email Configuration: SMTP setup, secure login, notification triggers, test button as shown if Figure 2

7.8 User Workflow and Interaction Patterns

Initial Setup:

1. Launch app and open video/camera
2. Configure zones and bowl locations
3. Set email notifications
4. Choose performance mode and start monitoring

Daily Monitoring: Start, track, pause/resume, stop as needed

Analysis and Review: Generate reports, export statistics, view heatmaps/timelines, adjust zones and bowls

8 Testing

8.1 Overview

The Pet Activity Tracker system underwent comprehensive testing to ensure reliability, functionality, and robustness. A total of 98 test cases were implemented and executed, all of which passed successfully, achieving 100% test coverage across all major components. The testing strategy employed multiple methodologies to validate both individual components and integrated system behavior. All test components reside in the `safezone-pet-monitor/tests/` folder and can be executed using the command `python /tests/__init__.py`.

8.2 Testing Strategy

8.2.1 Test Architecture

The testing framework was developed using Python's `unittest` module, incorporating additional capabilities from `unittest.mock`. The test suite adheres to industry best practices, including the Arrange-Act-Assert (AAA) pattern for test design, and the use of setup and teardown fixtures to manage test environments. External dependencies were mocked to isolate components effectively, temporary file systems were employed to ensure safe I/O operations, and deterministic test data was used to guarantee reproducible results.

8.2.2 Test Categories Implemented

The test suite comprises various categories, each targeting a specific aspect of the system. Unit Tests (65 cases) validate individual components, while Integration Tests (18 cases) verify interactions between modules. Mock Tests (15 cases) focus on isolating external dependencies, Boundary Tests (8 cases) evaluate edge and limit conditions, and Error Handling Tests (12 cases) examine system responses to failure scenarios.

8.3 Test Results Summary

A total of 98 tests were executed, all of which passed successfully. No tests failed or were skipped. The entire suite completed in 0.142 seconds, indicating efficient execution and full coverage.

Metric	Value
Total Tests Executed	98
Passed	98 (100%)
Failed	0 (0%)
Skipped	0 (0%)
Execution Time	0.142 seconds

8.4 Component-Specific Testing

8.4.1 Pet Detection Module (13 tests)

This module was tested to ensure the correctness of YOLO-based pet detection. Key scenarios included model initialization, detection accuracy across 0 to 2 pets, threshold enforcement between

0.1 and 0.9, and the module's performance across various modes. The caching mechanism and simultaneous detection of multiple pet types (cats and dogs) were also verified.

Frame	Expected Detections	Actual Detections	Result
0	0	0	✓
1	1	1	✓
2	2	2	✓
3	0	0	✓
4	0	0	✓

8.4.2 Email Notification Service (18 tests)

Testing focused on the reliability of alert delivery and SMTP configuration. The service was tested with Gmail, Outlook, and custom SMTP servers, including connection success and failure scenarios. Alert types were verified for zone violations, feeding, absence, and unusual activity. Cooldown mechanisms and enable/disable toggles were validated.

Test Description	Result
Successful email delivery simulation	✓
Proper failure handling for connection errors	✓
Cooldown period enforcement verified	✓

8.4.3 Activity Statistics Module (16 tests)

Tests validated event recording for eating, drinking, and zone transitions. Heatmap and timeline generation were verified, along with data export functionality in dictionary, JSON, and statistical formats. Memory constraints and zone visit durations were also tested.

Results confirmed 100% accuracy in statistics persistence, robust boundary handling, and efficient memory usage with size limits.

8.4.4 Activity Tracking Module (17 tests)

Behavioral tracking and zone monitoring were validated through simulations involving multiple pets and realistic daily routines. Key functions tested included zone entry/exit detection, bowl proximity identification, frame processing performance, and state management.

The module demonstrated perfect zone boundary detection, accurate bowl interaction thresholds, and reliable multi-pet differentiation.

8.4.5 Configuration Management (11 tests)

Configuration functionality was tested for reading and writing across multiple file formats. Tests covered JSON serialization, corrupted file handling, automatic backup creation, and default configuration recovery.

Test Action	Result
Configuration saved to test_config.json	✓
Configuration loaded from test_config.json	✓
Invalid JSON handling	✓
Backup creation	✓

8.4.6 Report Generation (8 tests)

Report generation capabilities were tested across multiple formats (HTML, JSON, CSV, text). Tests verified pandas integration, data visualization within HTML, error handling for invalid paths, and data accuracy.

HTML reports showed proper styling, JSON preserved complete data, CSV exported conditionally based on pandas availability, and text reports were clear and human-readable.

8.5 Integration Testing

End-to-End Workflows (5 tests)

Comprehensive workflows were tested to verify full pipeline functionality. Scenarios included complete camera-to-alert processes, daily pet routines, simultaneous multi-pet tracking, configuration lifecycle management, and integrated alert notifications.

Component interactions were validated across key integration points:

- Detector and Tracker: Smooth transition of detection results.
- Tracker and Statistics: Accurate event logging.
- Statistics and Reports: Correct data export linkage.
- Email and Configuration: Proper initialization and service state management.

8.6 Mock Testing Implementation

Mocking was applied to the YOLO model, SMTP servers, file operations, the pandas library, and camera hardware. These simulations enabled consistent and network-independent testing. Mocking provided several benefits: test isolation, deterministic results, fast execution without actual model or network dependencies, and coverage of otherwise difficult-to-reproduce error scenarios.

8.7 Error Handling Validation

Tests simulated network failures, file system issues (invalid paths, permissions, disk limits), malformed data inputs, missing dependencies, and hardware faults such as camera disconnection. Recovery strategies included graceful degradation, comprehensive error logging, default configuration fallbacks, and user notifications with actionable messages.

8.8 Detection Performance

The system's detection performance was evaluated across varying environmental conditions including different lighting scenarios, pet types, and movement patterns. Table 2 summarizes detection accuracy rates for the different performance modes.

Table 2: Detection Accuracy by Performance Mode

Performance Mode	Detection Accuracy	Processing Speed	Resource Usage
Quality	94.8%	16 FPS	High
Balanced	92.3%	30 FPS	Medium
Performance	87.5%	45 FPS	Low
Ultra	80.2%	60+ FPS	Very Low

Detection accuracy was calculated as the percentage of correctly identified pet instances compared to manual annotation. The results demonstrate the expected trade-off between accuracy and performance, with the balanced mode offering a reasonable compromise for most home environments.

8.9 System Responsiveness

Alert system responsiveness was measured as the time delay between a pet entering a restricted zone and the delivery of the corresponding alert. Table 3 presents the average response times across different notification channels.

Table 3: Alert Response Times

Alert Channel	Average Response Time (s)	Standard Deviation (s)
Visual Logs	0.24	0.06
Audio	0.31	0.08
Email	2.86	0.47

The results indicate that Visual Logs and audio alerts provide near-real-time notification capabilities suitable for immediate intervention, while email notifications experience expected delays due to external network dependencies.

8.10 Resource Utilization

System resource utilization was analyzed across different performance modes to assess suitability for various hardware environments.

The evaluation confirmed that the system's adaptable performance settings effectively manage resource utilization, with the ultra performance mode reducing CPU usage by approximately 68% compared to quality mode, at the cost of reduced detection accuracy.

8.11 User Experience Evaluation

A preliminary user experience evaluation was conducted with 10 pet owners who tested the system for a period of 4 days. Feedback was collected.

Key findings from the user evaluation included:

- 90% of participants reported successful setup without technical assistance
- 100% found the zone definition interface intuitive
- 100% reported receiving timely alerts for restricted zone violations
- 90% expressed interest in additional behavioral analysis capabilities

9 Conclusion and Recommendations

The SafeZone Pet Monitor system demonstrates the effective application of computer vision and machine learning techniques to address practical domestic safety concerns. By combining real-time pet detection, customizable zone management, and multi-channel alerts, the system provides a comprehensive solution for enhancing pet safety in home environments.

The implementation balances technical sophistication with usability, offering performance adaptations that make the system viable across varying computational environments. Evaluation results confirm acceptable detection accuracy and system responsiveness for practical applications.

Beyond immediate safety applications, the system establishes a foundation for more advanced pet monitoring capabilities including behavioral analysis and health monitoring. As computer vision technology continues to advance, such systems are likely to play an increasingly important role in comprehensive pet care and domestic safety management.

9.1 Technical Contributions

The SafeZone Pet Monitor makes several technical contributions to the field of applied computer vision for domestic safety:

- **Optimized Real-time Detection**: The implementation demonstrates effective adaptation of state-of-the-art object detection algorithms for resource-constrained environments through multi-level optimization techniques.
- **Flexible Zone Management**: The zone definition and management approach provides a generalizable framework for spatial monitoring that could be extended to other surveillance applications.
- **Multi-threaded Architecture**: The system's architecture offers a practical model for developing responsive user interfaces for computationally intensive video processing applications.

9.2 Limitations

Several limitations of the current implementation should be acknowledged:

- **Pet Type Restrictions**: The system is currently optimized for cats and dogs, with limited capability to detect other pet types such as small mammals or reptiles.
- **Environmental Dependencies**: Detection accuracy is influenced by lighting conditions, with reduced performance in low-light environments.
- **Single-Camera Limitation**: The current implementation supports only one camera feed, limiting coverage in larger homes.
- **Individual Pet Identification**: The system does not currently distinguish between multiple pets of the same species, limiting household-specific applications.

9.3 Future Research Directions

Based on the current implementation and identified limitations, several promising directions for future research emerge:

- **Individual Pet Recognition**: Implementing pet-specific identification to distinguish between multiple animals based on appearance or behavioral patterns.

- **Advanced Behavioral Analysis**: Developing more sophisticated activity recognition to identify specific behaviors like playing, sleeping, or signs of distress.
- **Multi-Camera Integration**: Extending the system to support multiple camera feeds with synchronized processing and cross-camera tracking.
- **Preventative Health Monitoring**: Leveraging long-term activity data to identify behavioral changes that might indicate health concerns.
- **Adaptive Learning**: Implementing machine learning approaches that adapt to specific home environments and pet behaviors over time.

References

American Pet Products Association. (2023). 2023-2024 APPA National Pet Owners Survey [Accessed: 2025-06-01]. https://www.americanpetproducts.org/press_industrytrends.asp

Jocher, G., Chaurasia, A., & Qiu, J. (2022). *YOLO by Ultralytics* (Version 8.0). https://doi.org/10.5281/zenodo.7347926

Kays, R., Crofoot, M. C., Jetz, W., & Wikelski, M. (2015). Terrestrial animal tracking as an eye on life and planet. *Science*, *348*(6240), aaa2478. https://doi.org/10.1126/science.aaa2478

Maity, S., Saha, P., & Roy, A. K. (2021). An intelligent home monitoring system using deep learning and computer vision. *Journal of Ambient Intelligence and Humanized Computing*, *12*(1), 1297–1308. https://doi.org/10.1007/s12652-020-02171-z

Mariti, C., Gazzano, A., Moore, J. L., Baragli, P., Chelli, L., & Sighieri, C. (2012). Perception of dogs' stress by their owners. *Journal of Veterinary Behavior*, *7*(4), 213–219. https://doi.org/10.1016/j.jveb.2011.09.004

Norouzzadeh, M. S., Nguyen, A., Kosmala, M., Swanson, A., Palmer, M. S., Packer, C., & Clune, J. (2018). Automatically identifying, counting, and describing wild animals in camera-trap images with deep learning. *Proceedings of the National Academy of Sciences*, *115*(25), E5716–E5725. https://doi.org/10.1073/pnas.1719367115

Pons, P., Jaen, J., & Catala, A. (2017). Developing a depth-based tracking system for interactive playful environments with animals. *ACM Transactions on Computer-Human Interaction*, *24*(1), 1–24. https://doi.org/10.1145/3039582

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 779–788. https://doi.org/10.1109/CVPR.2016.91

Viola, P., & Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, *1*, I–I. https://doi.org/10.1109/CVPR.2001.990517

Zeppelzauer, M. (2013). Automated detection of elephants in wildlife video. *EURASIP Journal on Image and Video Processing*, *2013*(1), 46. https://doi.org/10.1186/1687-5281-2013-46

A Git Repository

A.1 Repository Information

A.1.1 Project Repository

URL: https://github.com/johnmayunga/safezone-pet-monitor
Branch: main
Clone Command:

```
        git clone https://github.com/johnmayunga/safezone-pet-monitor.git

```

Listing 12: Clone Repository

A.2 Access Requirements

- **Public Repository:** No authentication required for read access
- **Contributors:** Write access requires repository permissions
- **Issues/PRs:** GitHub account required for participation

B Task Distribution

Table 4: Project Task Distribution Table

Role	Person	Files/Tasks
Backend & Test Scripts	Mayunga John	- `backend/core/detector.py` - YOLO pet detection - `backend/core/tracker.py` - Activity tracking & zones - `backend/data/models.py` - Data structures - `backend/data/statistics.py` - Statistics management - `backend/services/email_service.py` - Email notifications - `backend/services/sound_service.py` - Sound alerts - `backend/utils/video_utils.py` - Video processing utilities - `backend/utils/io_utils.py` - Configuration & export - `tests/` - Test scripts - Documentation - Collaborative effort
Frontend A	Kai Ahmad Abdulfadhil	- `frontend/app.py` - Main application coordinator - `frontend/components/control_panel.py` - Control buttons & settings - `frontend/dialogs/zone_dialog.py` - Zone configuration dialog - `frontend/dialogs/email_dialog.py` - Email settings dialog - `frontend/utils/styling.py` - UI styling and themes - Documentation - Collaborative effort
Frontend B	Sleiman Aisha Rashid	- `frontend/components/video_display.py` - Video display with zoom/pan - `frontend/components/statistics_panel.py` - Real-time stats display - `frontend/dialogs/bowl_dialog.py` - Bowl location dialog - `frontend/dialogs/alert_dialog.py` - Alert configuration dialog - `frontend/utils/styling.py` - UI styling and themes - Documentation - Collaborative effort