Data Center Security Monitoring System

Table of Contents

Introduction

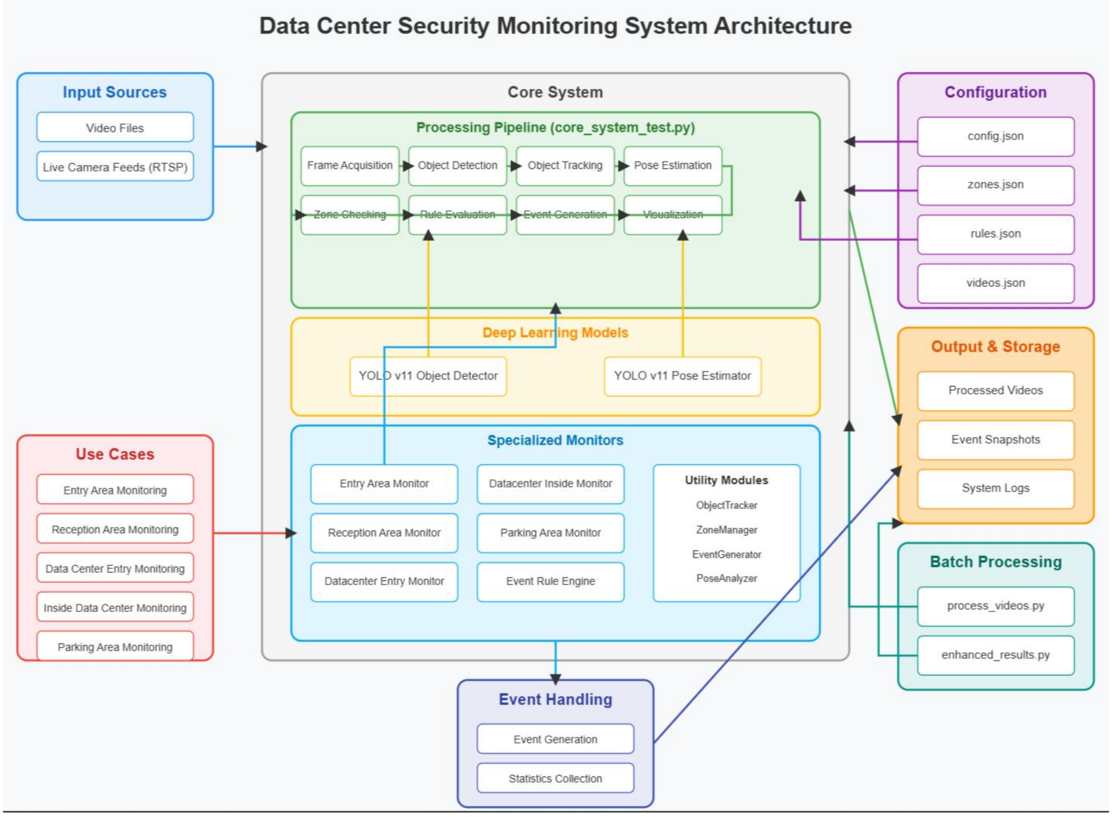
The Data Center Security Monitoring System is an advanced computer vision-based security solution designed to enhance surveillance capabilities in data center environments. The system uses state-of-the-art deep learning models for object detection and pose estimation to identify and respond to various security events across multiple areas.

Key Features

- **Real-time monitoring** of multiple security zones
- **AI-powered detection** of unauthorized activities
- **Pose analysis** for identifying suspicious behaviors
- **Video processing** with visualization and annotation
- **Customizable rules** for different security requirements
- **Modular architecture** for easy extension

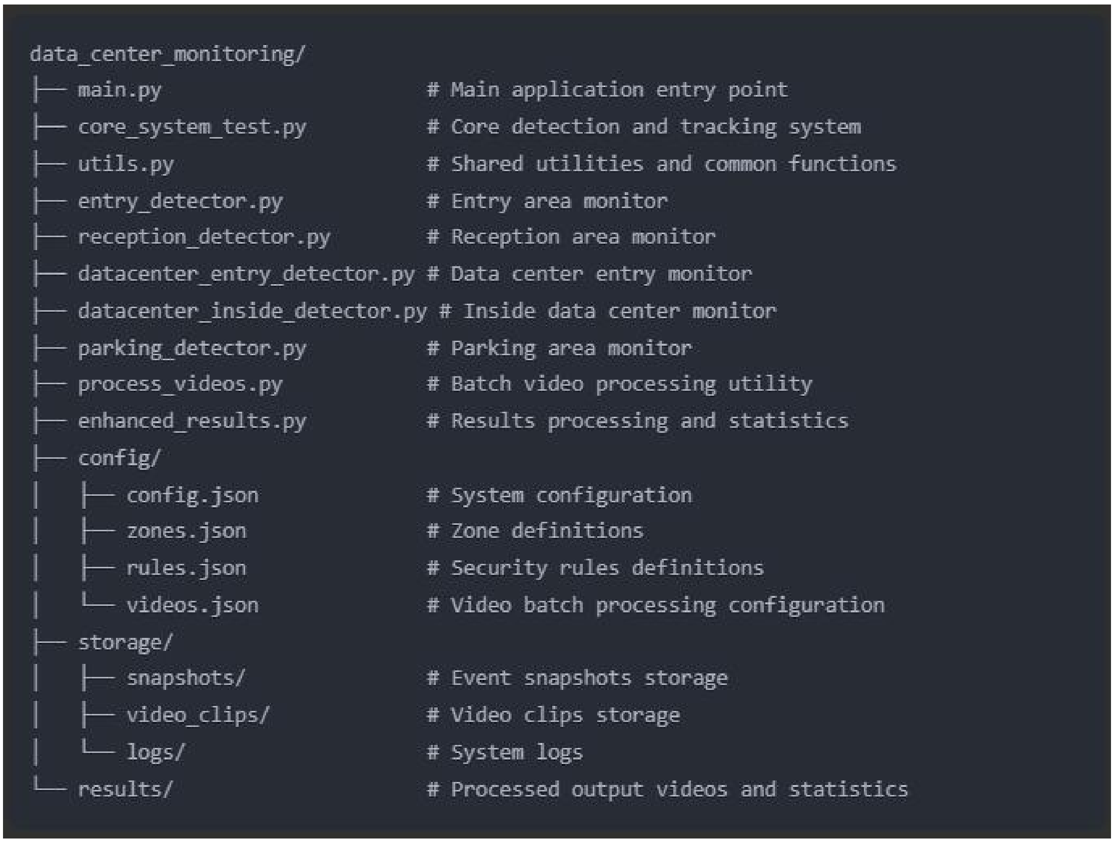
System Architecture

The system follows a modular architecture with specialized components for different monitoring tasks.



Component Overview

Files Structure



```
data_center_monitoring/
├── main.py                   # Main application entry point
├── core_system_test.py       # Core detection and tracking system
├── utils.py                  # Shared utilities and common functions
├── entry_detector.py         # Entry area monitor
├── reception_detector.py     # Reception area monitor
├── datacenter_entry_detector.py # Data center entry monitor
├── datacenter_inside_detector.py # Inside data center monitor
├── parking_detector.py       # Parking area monitor
├── process_videos.py         # Batch video processing utility
├── enhanced_results.py       # Results processing and statistics
├── config/
│   ├── config.json           # System configuration
│   ├── zones.json            # Zone definitions
│   ├── rules.json            # Security rules definitions
│   └── videos.json           # Video batch processing configuration
├── storage/
│   ├── snapshots/            # Event snapshots storage
│   ├── video_clips/          # Video clips storage
│   └── logs/                 # System logs
└── results/                  # Processed output videos and statistics
```

Core Components

Component	Description
Core System	Handles object detection, tracking, and event generation
Specialized Monitors	Area-specific monitoring for different use cases
Utility Modules	Shared functionality for tracking, zone management, and analysis
Configuration	JSON-based configuration for system settings

Use Cases

The system supports comprehensive monitoring across five critical areas of a data center facility:

1. Entry Area Monitoring

- **Sitting on stairs detection**: Identifies people sitting in stairwells (safety hazard)
- **Group gathering detection**: Monitors for unusual gatherings in entry areas
- **Vehicle entry monitoring**: Tracks vehicles that remain parked for too long

2. Reception Area Monitoring

- **Crowd control**: Ensures not more than 10 people are in reception area
- **Gate jumping detection**: Identifies attempts to bypass security gates

3. Data Center Entry Monitoring

- **Unauthorized access detection**: Identifies unauthorized personnel attempting to enter
- **Emergency button monitoring**: Detects when emergency buttons are pressed

4. Inside Data Center Monitoring

- **Unauthorized zone transitions**: Detects people moving from restricted areas (e.g., fire exits)
- **Zone violation monitoring**: Ensures personnel only access authorized areas

5. Parking Area Monitoring

- **Parking violation detection**: Identifies vehicles parked outside designated zones
- **Exit blockage detection**: Ensures emergency exits remain clear
- **Suspicious activity monitoring**: Detects unusual behavior in parking areas

Installation Guide

System Requirements

- **Operating System**: Windows 10/11, Ubuntu 20.04+
- **CPU**: Intel Core i7/AMD Ryzen 7 or better
- **RAM**: 16GB minimum (32GB recommended)
- **GPU**: NVIDIA GPU with at least 8GB VRAM (for real-time processing)
- **Storage**: 20GB for software, 100GB+ for video storage
- **Python**: Version 3.8+

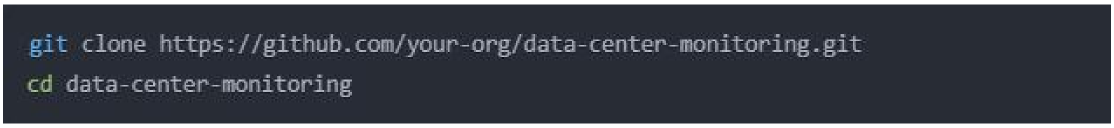
Prerequisites

- Python 3.8+
- CUDA Toolkit 11.7+ (for GPU acceleration)

- Compatible NVIDIA drivers

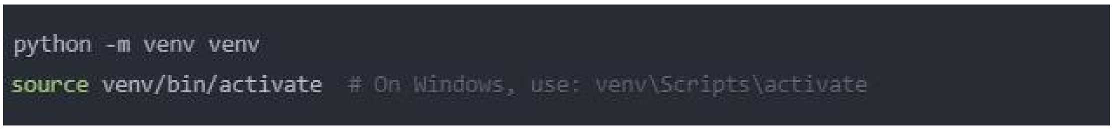
Installation Steps

1. **Clone the repository**



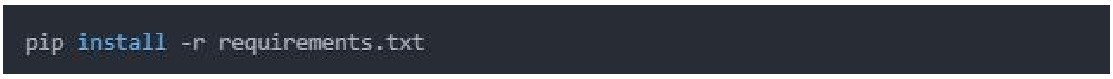
```
git clone https://github.com/your-org/data-center-monitoring.git
cd data-center-monitoring
```

2. **Set up a virtual environment (recommended)**



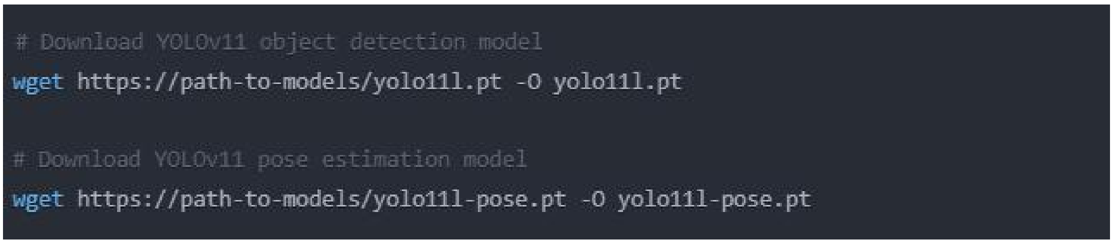
```
python -m venv venv
source venv/bin/activate  # On Windows, use: venv\Scripts\activate
```

3. **Install dependencies**



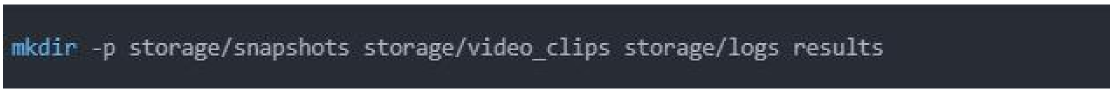
```
pip install -r requirements.txt
```

4. **Download the models**



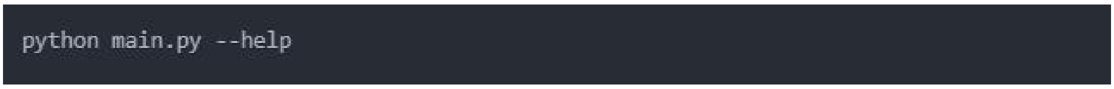
```
# Download YOLOv11 object detection model
wget https://path-to-models/yolo11l.pt -O yolo11l.pt

# Download YOLOv11 pose estimation model
wget https://path-to-models/yolo11l-pose.pt -O yolo11l-pose.pt
```

5. **Create required directories**



```
mkdir -p storage/snapshots storage/video_clips storage/logs results
```

6. **Verify installation**



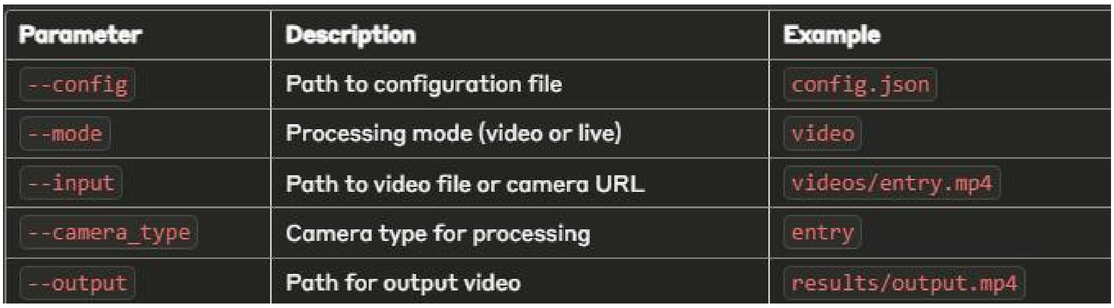
```
python main.py --help
```

Running the System

Processing a Single Video

python3 main.py --config config.json --mode video --input "videos/entry_video.mp4" --camera_type entry --output "results/output_entry.mp4"

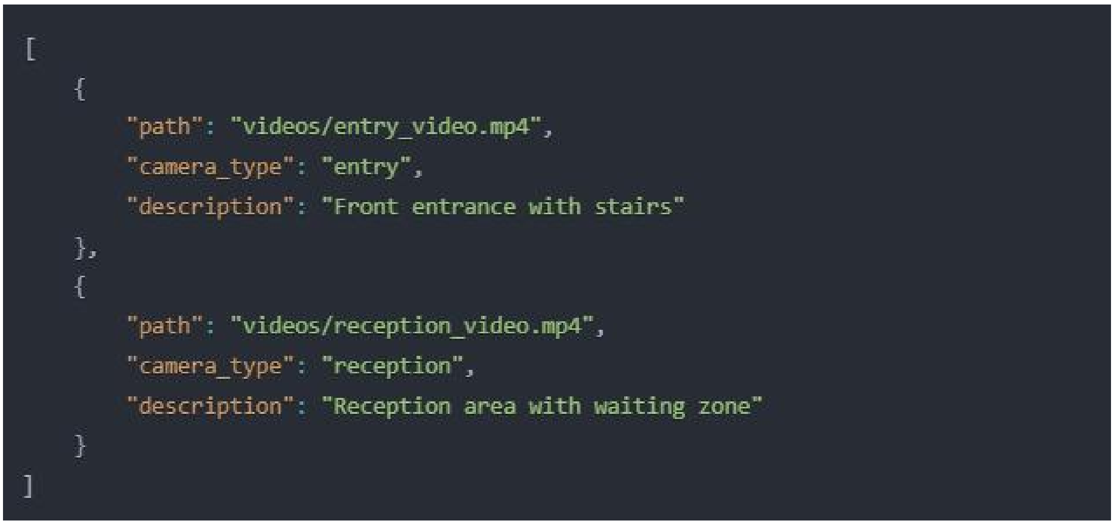
Parameters



Parameter	Description	Example
`--config`	Path to configuration file	`config.json`
`--mode`	Processing mode (video or live)	`video`
`--input`	Path to video file or camera URL	`videos/entry.mp4`
`--camera_type`	Camera type for processing	`entry`
`--output`	Path for output video	`results/output.mp4`

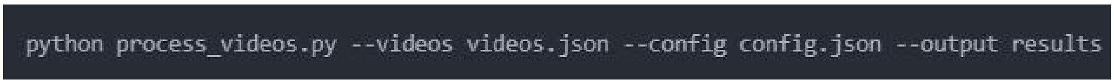
Batch Processing Multiple Videos

Create a videos.json file:



```
[
    {
        "path": "videos/entry_video.mp4",
        "camera_type": "entry",
        "description": "Front entrance with stairs"
    },
    {
        "path": "videos/reception_video.mp4",
        "camera_type": "reception",
        "description": "Reception area with waiting zone"
    }
]
```

Run the batch processing script:



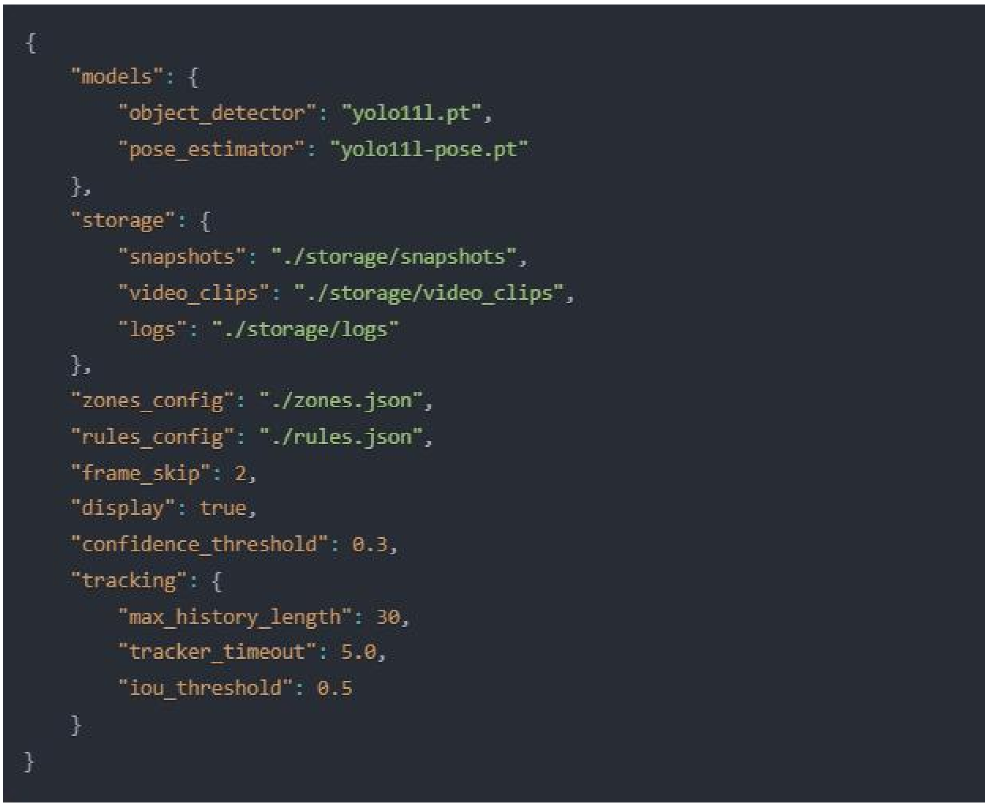
```
python process_videos.py --videos videos.json --config config.json --output results
```

Live Camera Monitoring & Processing Multiple Cameras in Parallel

Bash : python main.py --config config.json --mode live --input "rtsp://camera_ip:port/stream" --camera_type entry
Bash: ./process_parallel.sh

Configuration

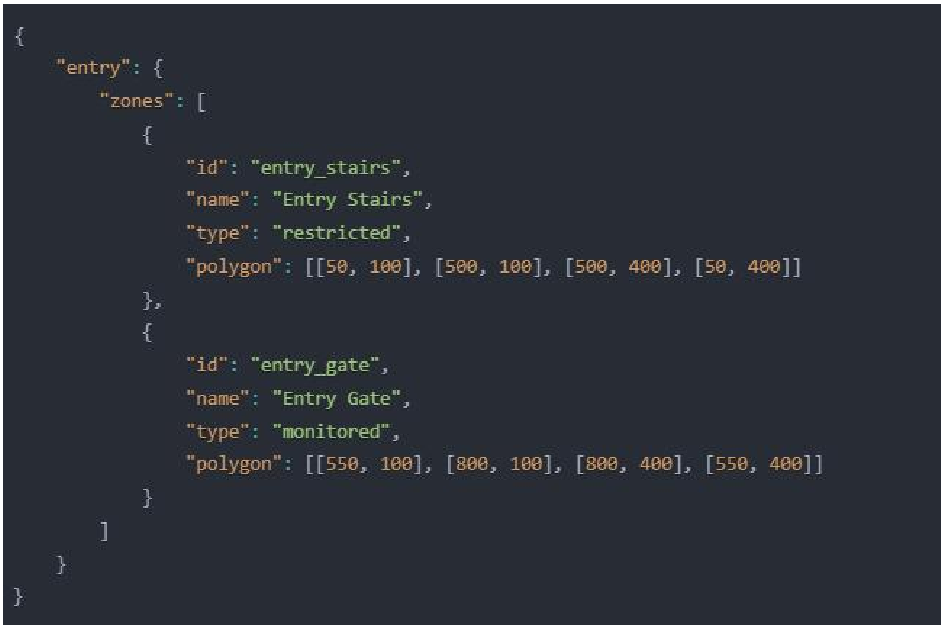
System Configuration (config.json)



```
{
    "models": {
        "object_detector": "yolo11l.pt",
        "pose_estimator": "yolo11l-pose.pt"
    },
    "storage": {
        "snapshots": "./storage/snapshots",
        "video_clips": "./storage/video_clips",
        "logs": "./storage/logs"
    },
    "zones_config": "./zones.json",
    "rules_config": "./rules.json",
    "frame_skip": 2,
    "display": true,
    "confidence_threshold": 0.3,
    "tracking": {
        "max_history_length": 30,
        "tracker_timeout": 5.0,
        "iou_threshold": 0.5
    }
}
```

Zone Definitions (zones.json)

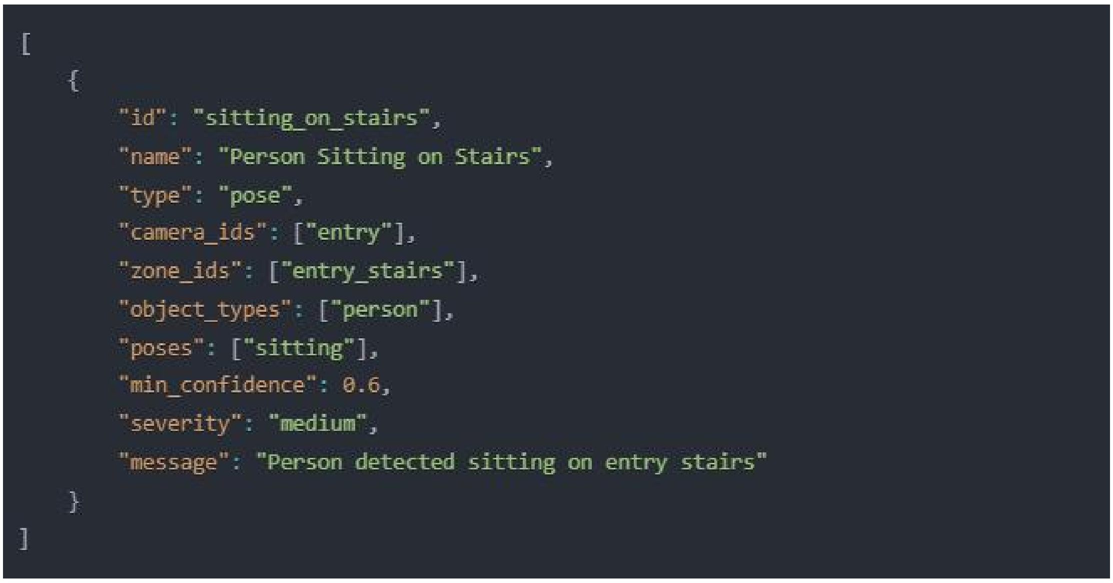
Zones define the monitored areas for each camera type:



```
{
    "entry": {
        "zones": [
            {
                "id": "entry_stairs",
                "name": "Entry Stairs",
                "type": "restricted",
                "polygon": [[50, 100], [500, 100], [500, 400], [50, 400]]
            },
            {
                "id": "entry_gate",
                "name": "Entry Gate",
                "type": "monitored",
                "polygon": [[550, 100], [800, 100], [800, 400], [550, 400]]
            }
        ]
    }
}
```

Security Rules (rules.json)

Rules define when security events are triggered:



```
[
  {
    "id": "sitting_on_stairs",
    "name": "Person Sitting on Stairs",
    "type": "pose",
    "camera_ids": ["entry"],
    "zone_ids": ["entry_stairs"],
    "object_types": ["person"],
    "poses": ["sitting"],
    "min_confidence": 0.6,
    "severity": "medium",
    "message": "Person detected sitting on entry stairs"
  }
]
```

Technical Details

Models Used

The system utilizes two main deep learning models:

Object Detector: YOLOv11-large

- Detects people, vehicles, and other objects
- Provides bounding boxes and confidence scores
- Classifies detected objects by type

Pose Estimator: YOLOv11-pose

- Provides skeleton keypoints for people
- Enables sophisticated pose classification
- Used for behavior and activity analysis

Processing Pipeline

1. **Frame Acquisition**: Get frame from video/camera
2. **Object Detection**: Detect and classify objects
3. **Object Tracking**: Track objects across frames
4. **Pose Estimation**: Determine poses for people
5. **Zone Checking**: Check objects against defined zones
6. **Rule Evaluation**: Evaluate security rules
7. **Event Generation**: Create events for violations
8. **Visualization**: Annotate frame with results

Algorithm Details

Object Tracking with Kalman Filters

The system uses Kalman filters for robust object tracking:

- **State Vector**: [x, y, width, height, dx, dy, dw, dh]
- **Observation Vector**: [x, y, width, height]
- **Tracking Association**: IoU-based matching between frames

Pose Classification

Poses are classified based on skeletal analysis:

Pose	Criteria
Sitting	Knee angles 70-110°, hip angles 70-110°
Standing	Knee angles >160°, hip angles >160°
Running	High knee lift with bent knees
Crouching	Low knee and hip angles with compressed posture

Troubleshooting

Common Issues and Solutions

Issue	Possible Cause	Solution
CUDA out of memory	GPU memory insufficient	Increase frame_skip, reduce resolution, or use smaller models
Pose estimation errors	Invalid keypoint data	Check for proper keypoint structure and valid confidence values
Zone detection failures	Misconfigured zones	Verify zone definitions and camera type names match
Slow processing	Hardware limitations	Use GPU acceleration, increase frame skip, or process at lower resolution
No events detected	Confidence threshold too high	Lower min_confidence in rules.json
False positive detections	Confidence threshold too low	Increase min_confidence in rules.json

Error Messages

Error Message	Explanation	Resolution
ValueError: not enough values to unpack (expected 3, got 1)	Pose keypoint format mismatch	Update _estimate_poses method with better error handling
IndexError: index 0 is out of bounds for axis 0 with size 0	Empty detection results	Add checks for empty detections arrays
No zones found for camera_id	Camera type missing in zones.json	Add camera type to zones configuration
CUDA out of memory	Insufficient GPU memory	Reduce batch size or model complexity

-------------------------------------End of report ---------------------------------------