

NVIDIA Jetson Nano Developer Kit Setup Guide for InvEye

Important

This guide provides comprehensive instructions for setting up the NVIDIA Jetson Nano Developer Kit and deploying the InvEye Edge AI Video Analytics solution for real-time compliance, audit, and intelligence monitoring.

Table of Contents

1. [Overview](#)
 2. [Hardware Requirements](#)
 3. [Initial Setup & Configuration](#)
 4. [Software Installation](#)
 5. [InvEye Edge Deployment](#)
 6. [CCTV Integration](#)
 7. [Cloud Connectivity](#)
 8. [Performance Optimization](#)
 9. [Troubleshooting](#)
 10. [Maintenance & Best Practices](#)
-

Overview

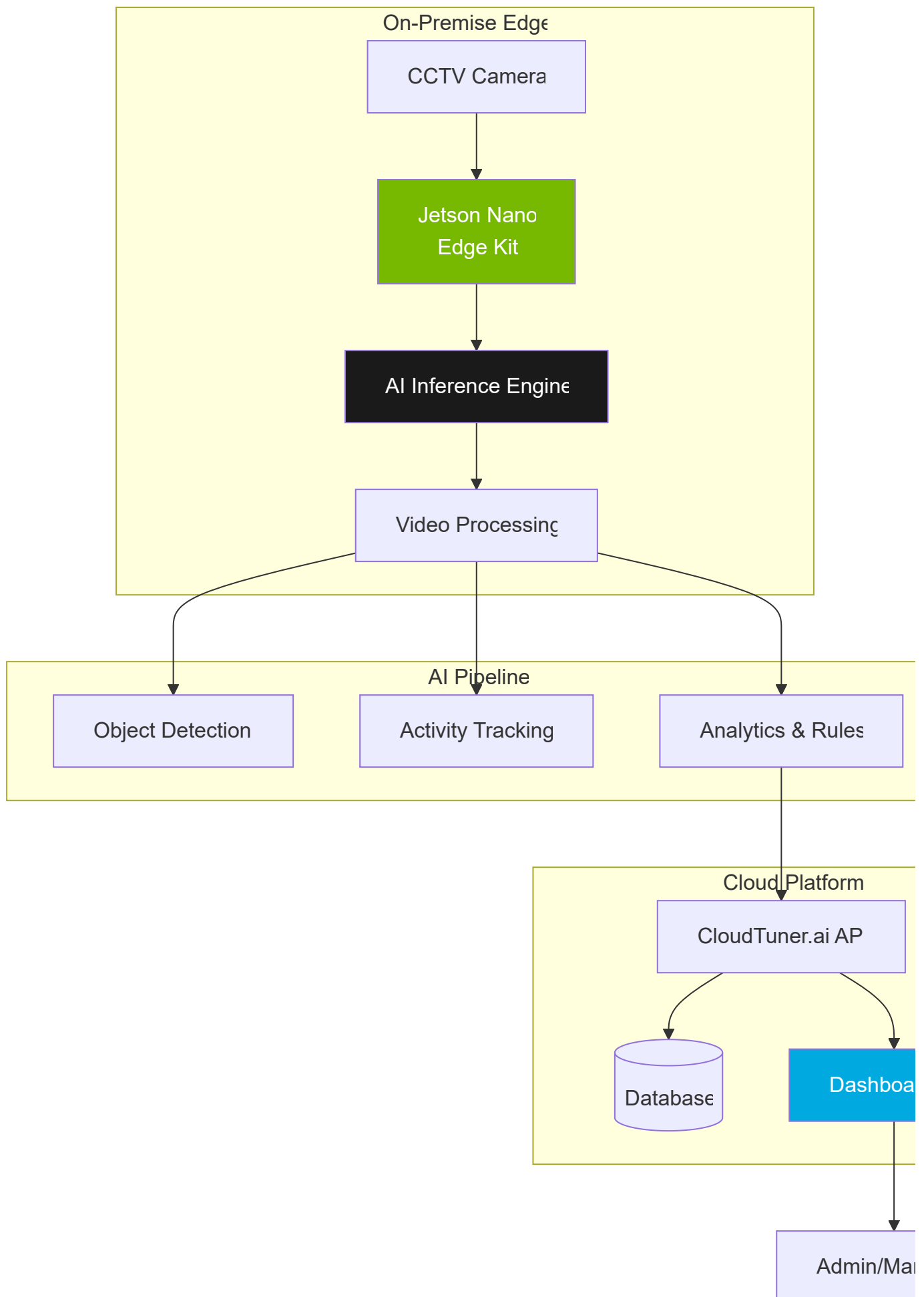
What is NVIDIA Jetson Nano?

The **NVIDIA Jetson Nano Developer Kit** is a small, powerful computer designed for edge AI applications. It delivers 472 GFLOPS of compute performance for running modern AI workloads and can process multiple high-resolution video streams simultaneously.

Key Specifications:

- **GPU:** 128-core NVIDIA Maxwell GPU
- **CPU:** Quad-core ARM Cortex-A57 @ 1.43 GHz
- **Memory:** 4GB 64-bit LPDDR4
- **Storage:** microSD card slot (minimum 32GB recommended)
- **Power:** 5V 4A via barrel jack or Micro-USB (5V 2A)
- **Connectivity:** Gigabit Ethernet, M.2 Key E for WiFi

InvEye Edge Architecture



Hardware Requirements

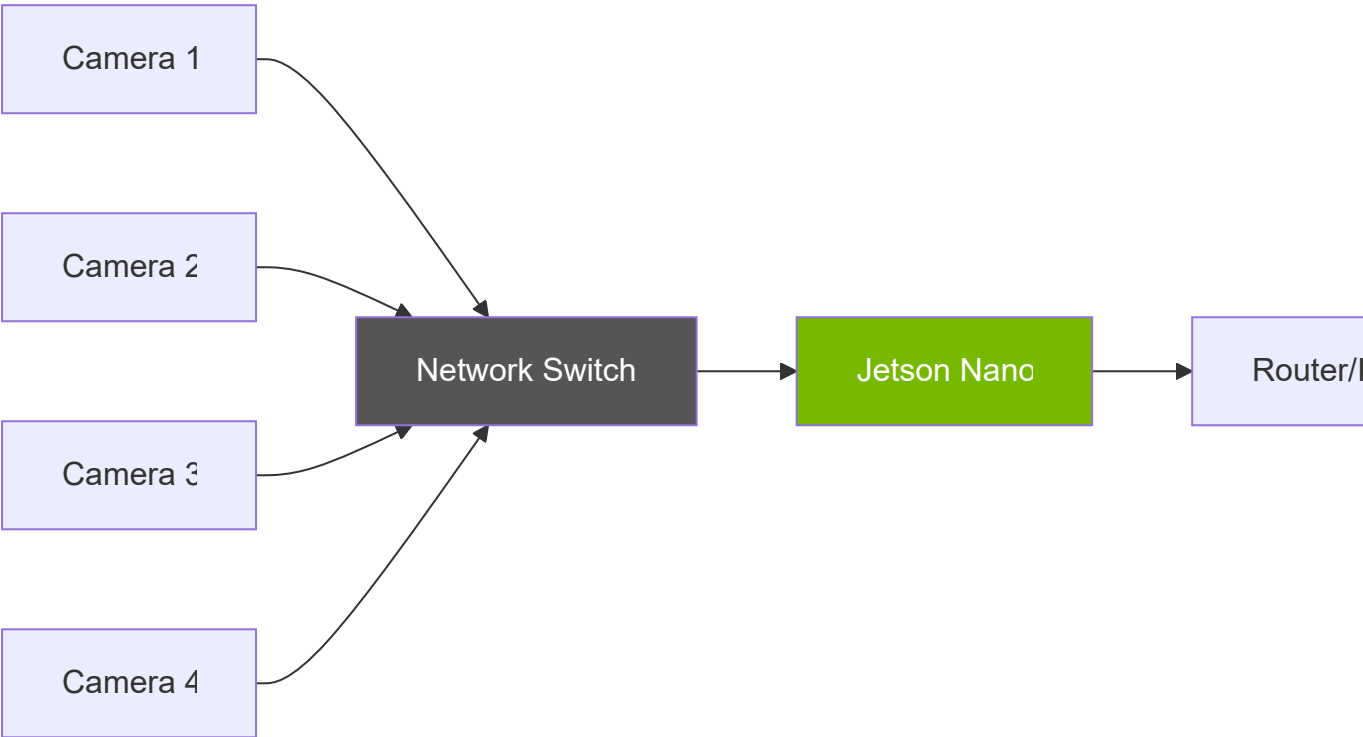
Essential Components

Component	Specification	Purpose
Jetson Nano Developer Kit	4GB RAM Model	Main compute unit
Power Supply	5V 4A barrel jack adapter	Recommended for AI workload
microSD Card	64GB UHS-1 or higher	Operating system and storage
CCTV Cameras	IP cameras with RTSP support	Video input sources
Network Switch	Gigabit Ethernet	Connect cameras and Jetson
Cooling	Active cooling fan (recommended)	Prevent thermal throttling

Optional Components

- WiFi Module:** M.2 Key E wireless adapter for wireless connectivity
- CSI Camera:** Official Jetson camera module for direct camera connection
- USB Hub:** For connecting multiple USB devices
- Case:** Protective enclosure with fan mounting
- UPS:** Uninterruptible power supply for critical deployments

Network Topology



Initial Setup & Configuration

Step 1: Prepare the microSD Card

Note

You'll need a computer with an SD card reader for this step.

Download JetPack SDK

1. Visit the [NVIDIA Jetson Download Center](#)
2. Download the latest **JetPack** image for Jetson Nano (approximately 6GB)
3. Current recommended version: **JetPack 4.6.1** or later

Flash the SD Card

Using Balena Etcher (Recommended):

1. Download and install [Balena Etcher](#)
2. Insert the microSD card into your computer
3. Open Balena Etcher
4. Select the downloaded JetPack image file
5. Select the microSD card as the target
6. Click "Flash" and wait for completion (10-15 minutes)

Using Command Line (Linux/macOS):

```
# Identify the SD card device

lsblk


# Flash the image (replace /dev/sdX with your SD card)

sudo dd if=jetson-nano-jp461-sd-card-image.img of=/dev/sdX bs=4M
status=progress

sync
```

Step 2: Hardware Assembly

1. **Insert microSD Card:** Place the flashed microSD card into the Jetson Nano's slot
2. **Attach Cooling Fan:** Connect 5V fan to J15 header (pins 1 and 3)
3. **Connect Peripherals:**
 - USB keyboard and mouse
 - HDMI monitor
 - Ethernet cable
4. **Power Configuration:**
 - For 5V 4A power supply: Place jumper on J48 pins
 - Remove jumper for Micro-USB power (2A mode)

Step 3: First Boot & System Configuration

1. **Power On:** Connect the power supply
2. **Initial Setup Wizard:**
 - Accept license agreements
 - Select language and keyboard layout
 - Create user account (e.g., `inveye-admin`)
 - Set timezone
 - Choose partition size (use entire SD card)
3. **System Updates:**

```
# Update package lists
```

```
sudo apt update
```

```
# Upgrade all packages
```

```
sudo apt upgrade -y
```

```
# Install essential tools
```

```
sudo apt install -y nano git curl wget htop
```

Step 4: Verify Installation

```
# Check NVIDIA GPU status
```

```
sudo nvidia-smi -q
```

```
# Verify CUDA installation
```

```
nvcc --version
```

```
# Check system information
```

```
cat /etc/nv_tegra_release
```

```
# Monitor system resources
```

```
tegrastats
```

Expected Output:

- CUDA Version: 10.2 or higher
- TensorRT Version: 8.x
- cuDNN Version: 8.x

Software Installation

Step 1: Install System Dependencies

```
# Update system
```

```
sudo apt update && sudo apt upgrade -y
```

```
# Install Python and development tools

sudo apt install -y python3-pip python3-dev python3-venv

sudo apt install -y libhdf5-serial-dev hdf5-tools libhdf5-dev zlib1g-dev

sudo apt install -y libjpeg8-dev liblapack-dev libblas-dev gfortran


# Install multimedia libraries

sudo apt install -y libavcodec-dev libavformat-dev libswscale-dev

sudo apt install -y libgstreamer1.0-dev gstreamer1.0-tools gstreamer1.0-
plugins-base

sudo apt install -y gstreamer1.0-plugins-good gstreamer1.0-plugins-bad
gstreamer1.0-plugins-ugly
```

Step 2: Install AI/ML Frameworks

Install PyTorch for Jetson

```
# Download PyTorch wheel for Jetson

wget
https://nvidia.box.com/shared/static/p57jwntv436lfrd78inwl7iml6p13fzh.whl -O
torch-1.12.0-cp36-cp36m-linux_aarch64.whl


# Install PyTorch

pip3 install torch-1.12.0-cp36-cp36m-linux_aarch64.whl


# Install torchvision

sudo apt install -y libjpeg-dev zlib1g-dev libpython3-dev libavcodec-dev
libavformat-dev libswscale-dev

git clone --branch v0.13.0 https://github.com/pytorch/vision torchvision

cd torchvision
```

```
python3 setup.py install --user
```

Install TensorFlow (Optional)

```
sudo pip3 install --pre --extra-index-url  
https://developer.download.nvidia.com/compute/redist/jp/v461 tensorflow
```

Install ONNX Runtime

```
# For optimized inference  
  
pip3 install onnxruntime-gpu
```

Step 3: Install Computer Vision Libraries

```
# Install OpenCV with CUDA support (pre-installed with JetPack)  
  
# Verify installation  
  
python3 -c "import cv2; print(cv2.__version__)"  
  
# Install additional CV tools  
  
pip3 install scikit-image pillow matplotlib
```

Step 4: Install InvEye Dependencies

```
# Create virtual environment for InvEye  
  
cd ~  
  
mkdir inveye-edge  
  
cd inveye-edge
```



```
python3 -m venv venv

source venv/bin/activate

# Install core dependencies

pip install --upgrade pip setuptools wheel

# Install video processing

pip install opencv-python-headless

pip install imutils

# Install AI inference tools

pip install onnx onnxruntime-gpu

pip install ultralytics # For YOLO models

# Install networking and cloud connectivity

pip install requests paho-mqtt

pip install websockets aiohttp

# Install monitoring and logging

pip install psutil prometheus-client
```

InvEye Edge Deployment

Architecture Overview

The InvEye edge module runs on Jetson Nano to perform:

1. **Video Stream Ingestion** from multiple RTSP cameras
2. **Real-time AI Inference** for object detection and activity recognition
3. **Event Processing & Filtering** based on business rules
4. **Cloud Synchronization** with CloudTuner.ai dashboard

Directory Structure

```
cd ~/inveye-edge

mkdir -p {config,models,logs,data,scripts}

# Create directory structure

tree -L 2
```

```
inveye-edge/
├── config/
│   ├── cameras.json
│   ├── inference.yaml
│   └── cloud.yml
├── models/
│   ├── yolov8n.onnx
│   ├── person_detection.onnx
│   └── activity_classifier.onnx
├── logs/
│   └── inveye.log
├── data/
│   ├── frames/
│   └── events/
```

```
├─ scripts/
|   ├─ stream_processor.py
|   ├─ inference_engine.py
|   ├─ cloud_sync.py
|   └─ main.py
└─ venv/
```

Configuration Files

1. Camera Configuration (`config/cameras.json`)

```
{
  "cameras": [
    {
      "id": "cam_001",
      "name": "Entrance Door",
      "rtsp_url": "rtsp://admin:password@192.168.1.101:554/stream1",
      "fps": 10,
      "resolution": [1920, 1080],
      "zones": [
        {
          "name": "Entry Zone",
          "coordinates": [[100, 100], [500, 100], [500, 400], [100, 400]],
          "analytics": ["person_count", "dwell_time"]
        }
      ]
    }
  ]
}
```

```

    },

    {

        "id": "cam_002",

        "name": "Checkout Counter",

        "rtsp_url": "rtsp://admin:password@192.168.1.102:554/stream1",

        "fps": 15,

        "resolution": [1920, 1080],

        "zones": [

            {

                "name": "Counter Zone",

                "coordinates": [[200, 200], [800, 200], [800, 600], [200, 600]],

                "analytics": ["queue_length", "transaction_time",
"staff_presence"]

            }

        ]

    }

]

}

```

2. Inference Configuration (config/inference.yaml)

```

inference:

    framework: "onnx"

    device: "cuda"  # Use GPU acceleration

    batch_size: 1

models:

```

person_detection:

path: "models/yolov8n.onnx"

confidence_threshold: 0.5

nms_threshold: 0.45

input_size: [640, 640]

activity_recognition:

path: "models/activity_classifier.onnx"

confidence_threshold: 0.7

classes: ["standing", "walking", "running", "sitting", "interacting"]

analytics:

person_tracking:

enabled: true

tracker: "deepsort"

max_age: 30

event_detection:

enabled: true

rules:

- name: "unauthorized_area"

type: "zone_intrusion"

severity: "high"

- name: "loitering"

type: "dwell_time"

threshold: 300 # seconds

severity: "medium"

```
- name: "crowd_detection"

  type: "person_count"

  threshold: 10

  severity: "low"
```

3. Cloud Configuration (config/cloud.yml)

```
cloud:

  endpoint: "https://api.cloudtuner.ai/v1/inveye"

  api_key: "${INVEYE_API_KEY}"

  device_id: "${JETSON_DEVICE_ID}"

  sync:

    interval: 30 # seconds

    retry_attempts: 3

    batch_size: 100

  events:

    upload_frames: true

    compression: "jpeg"

    quality: 85

  metrics:

    enabled: true

    interval: 60 # seconds
```

Core Application Code

Stream Processor (scripts/stream_processor.py)

```
import cv2

import threading

import queue

from datetime import datetime


class RTSPStreamProcessor:

    def __init__(self, camera_config):

        self.config = camera_config

        self.frame_queue = queue.Queue(maxsize=30)

        self.running = False

        self.cap = None

    def start(self):

        """Start video capture thread"""

        self.running = True

        self.thread = threading.Thread(target=self._capture_frames)

        self.thread.daemon = True

        self.thread.start()

    def _capture_frames(self):

        """Capture frames from RTSP stream"""

        rtsp_url = self.config['rtsp_url']

        # GStreamer pipeline for hardware decoding

        pipeline = (

            f"rtspsrc location={rtsp_url} latency=0 ! "

            "rtph264depay ! h264parse ! "
```

```

        "nvv4l2decoder ! nvvidconv ! "

        "video/x-raw,format=BGRx ! "

        "videoconvert ! video/x-raw,format=BGR ! "

        "appsink drop=1"

    )

    self.cap = cv2.VideoCapture(pipeline, cv2.CAP_GSTREAMER)

    if not self.cap.isOpened():

        print(f"[ERROR] Failed to open stream: {self.config['name']}")

        return

    fps = self.config.get('fps', 10)

    frame_interval = 1.0 / fps

    while self.running:

        ret, frame = self.cap.read()

        if ret:

            timestamp = datetime.now()

            # Add to queue (drop if full)

            try:

                self.frame_queue.put_nowait({

                    'camera_id': self.config['id'],

                    'frame': frame,

                    'timestamp': timestamp

                })

            except queue.Full:

                pass # Drop frame if queue is full

        else:

```



```

        print(f"[WARN] Frame read failed for {self.config['name']}")

        # Attempt reconnection

        self.cap.release()

        self.cap = cv2.VideoCapture(pipeline, cv2.CAP_GSTREAMER)

    def get_frame(self, timeout=1):

        """Get next frame from queue"""

        try:

            return self.frame_queue.get(timeout=timeout)

        except queue.Empty:

            return None

    def stop(self):

        """Stop capture thread"""

        self.running = False

        if self.cap:

            self.cap.release()

```

Inference Engine (scripts/inference_engine.py)

```

import onnxruntime as ort

import numpy as np

import cv2

class InvEyeInferenceEngine:

    def __init__(self, config):

        self.config = config

        # Initialize ONNX Runtime with CUDA

```

```

providers = ['CUDAExecutionProvider', 'CPUExecutionProvider']

# Load person detection model

detection_config = config['models']['person_detection']

self.detector = ort.InferenceSession(

    detection_config['path'],

    providers=providers

)

self.conf_threshold = detection_config['confidence_threshold']

self.nms_threshold = detection_config['nms_threshold']

self.input_size = tuple(detection_config['input_size'])

def preprocess(self, frame):

    """Preprocess frame for inference"""

    # Resize and normalize

    input_img = cv2.resize(frame, self.input_size)

    input_img = cv2.cvtColor(input_img, cv2.COLOR_BGR2RGB)

    input_img = input_img.astype(np.float32) / 255.0

    # Transpose to NCHW format

    input_img = np.transpose(input_img, (2, 0, 1))

    input_img = np.expand_dims(input_img, axis=0)

    return input_img

def detect_persons(self, frame):

    """Run person detection inference"""

    # Preprocess

    input_tensor = self.preprocess(frame)

    # Run inference

```

```

        outputs = self.detector.run(None, {self.detector.get_inputs()
[0].name: input_tensor})

    # Post-process detections

    detections = self._postprocess(outputs[0], frame.shape)

    return detections

def _postprocess(self, output, original_shape):

    """Convert raw output to bounding boxes"""

    detections = []

    h, w = original_shape[:2]

    scale_x = w / self.input_size[0]

    scale_y = h / self.input_size[1]

    # Parse output (format: [batch, num_detections, 6])

    # Each detection: [x_center, y_center, width, height, confidence,
class]

    for detection in output[0]:

        confidence = detection[4]

        if confidence > self.conf_threshold:

            x_center = detection[0] * scale_x

            y_center = detection[1] * scale_y

            width = detection[2] * scale_x

            height = detection[3] * scale_y

            x1 = int(x_center - width / 2)

            y1 = int(y_center - height / 2)

            x2 = int(x_center + width / 2)

            y2 = int(y_center + height / 2)

            detections.append({

```

```
        'box': [x1, y1, x2, y2],

        'confidence': float(confidence),

        'class': 'person'

    })

    return detections
```

Main Application (scripts/main.py)

```
import json

import yaml

import logging

import signal

import sys

from stream_processor import RTSPStreamProcessor

from inference_engine import InvEyeInferenceEngine

from cloud_sync import CloudSyncManager


# Configure logging

logging.basicConfig(

    level=logging.INFO,

    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',

    handlers=[

        logging.FileHandler('logs/inveye.log'),

        logging.StreamHandler()

    ]

)
```

```
)
```

```
logger = logging.getLogger('InvEye')
```

```
class InvEyeEdge:
```

```
    def __init__(self):
```

```
        self.running = False
```

```
        self.streams = []
```

```
        # Load configurations
```

```
        with open('config/cameras.json') as f:
```

```
            self.camera_config = json.load(f)
```

```
        with open('config/inference.yaml') as f:
```

```
            self.inference_config = yaml.safe_load(f)
```

```
        with open('config/cloud.yaml') as f:
```

```
            self.cloud_config = yaml.safe_load(f)
```

```
        # Initialize components
```

```
        self.inference_engine = InvEyeInferenceEngine(self.inference_config)
```

```
        self.cloud_sync = CloudSyncManager(self.cloud_config)
```

```
    def start(self):
```

```
        """Start InvEye edge processing"""
```

```
        logger.info("Starting InvEye Edge System...")
```

```
        # Start stream processors
```

```
        for cam in self.camera_config['cameras']:
```

```
            processor = RTSPStreamProcessor(cam)
```

```
            processor.start()
```

```
            self.streams.append(processor)
```

```

        logger.info(f"Started stream: {cam['name']}")

    # Start cloud sync

    self.cloud_sync.start()

    self.running = True

    self.process_loop()

def process_loop(self):

    """Main processing loop"""

    logger.info("Processing started...")

    while self.running:

        for stream in self.streams:

            frame_data = stream.get_frame(timeout=0.1)

            if frame_data is None:

                continue

            # Run inference

            detections =
self.inference_engine.detect_persons(frame_data['frame'])

            # Process detections and generate events

            events = self._process_detections(

                frame_data['camera_id'],

                detections,

                frame_data['timestamp']

            )

            # Send to cloud

            if events:

                self.cloud_sync.queue_events(events)

def _process_detections(self, camera_id, detections, timestamp):

```

```

        """Process detections and generate events"""

        events = []

        # Example: Person count alert

        person_count = len(detections)

        if person_count > 10:  # Threshold

            events.append({

                'type': 'crowd_detected',

                'camera_id': camera_id,

                'timestamp': timestamp.isoformat(),

                'count': person_count,

                'severity': 'medium'

            })

        return events

    def stop(self):

        """Shutdown gracefully"""

        logger.info("Shutting down InvEye Edge...")

        self.running = False

        for stream in self.streams:

            stream.stop()

        self.cloud_sync.stop()

        logger.info("Shutdown complete")

def signal_handler(sig, frame):

    """Handle shutdown signals"""

    logger.info("Received shutdown signal")

```

```

app.stop()

sys.exit(0)

if __name__ == "__main__":

    app = InvEyeEdge()

    # Register signal handlers

    signal.signal(signal.SIGINT, signal_handler)

    signal.signal(signal.SIGTERM, signal_handler)

    try:

        app.start()

    except Exception as e:

        logger.error(f"Fatal error: {e}", exc_info=True)

        app.stop()

```

Deployment Steps

```
# 1. Clone/copy InvEye application
```

```
cd ~/inveye-edge
```

```
# 2. Download AI models
```

```
mkdir -p models
```

```
cd models
```

```
# Download YOLOv8 ONNX model
```

```
wget
```

```
https://github.com/ultralytics/assets/releases/download/v0.0.0/yolov8n.onnx
```



```
# 3. Set environment variables
```

```
echo "export INVEYE_API_KEY='your-api-key-here'" >> ~/.bashrc
```

```
echo "export JETSON_DEVICE_ID='jetson-nano-001'" >> ~/.bashrc
```

```
source ~/.bashrc
```

```
# 4. Update camera configuration
```

```
nano config/cameras.json
```

```
# Add your RTSP camera URLs
```

```
# 5. Test the setup
```

```
cd ~/inveye-edge
```

```
source venv/bin/activate
```

```
python scripts/main.py
```

CCTV Integration

Supported Camera Types

- **IP Cameras** with RTSP/RTMP streaming
- **USB Cameras** (via V4L2)
- **CSI Cameras** (Jetson-compatible camera modules)
- **NVR/DVR Systems** with streaming output

RTSP Stream Configuration

Finding RTSP URLs

Common formats for popular camera brands:

Brand	RTSP URL Format
-------	-----------------

|-----|-----|

| **Hikvision** | `rtsp://admin:password@IP:554/Streaming/Channels/101` |

| **Dahua** | `rtsp://admin:password@IP:554/cam/realmonitor?channel=1&subtype=0` |

| **Axis** | `rtsp://admin:password@IP:554/axis-media/media.amp` |

| **Generic** | `rtsp://username:password@IP:554/stream1` |

Testing RTSP Streams

```
# Install VLC for testing
```

```
sudo apt install vlc
```

```
# Test stream playback
```

```
vlc rtsp://admin:password@192.168.1.101:554/stream1
```

```
# Or use ffmpeg
```

```
ffmpeg -rtsp_transport tcp -i
```

```
rtsp://admin:password@192.168.1.101:554/stream1 -frames:v 1 test.jpg
```

Network Configuration

```
# Check network interfaces
```

```
ip addr show
```

```
# Set static IP for Jetson (optional)
```

```
sudo nano /etc/netplan/01-network-manager-all.yaml
```

Example configuration:

```
network:

  version: 2

  renderer: NetworkManager

  ethernets:

    eth0:

      dhcp4: no

      addresses: [192.168.1.100/24]

      gateway4: 192.168.1.1

      nameservers:

        addresses: [8.8.8.8, 8.8.4.4]
```

Apply configuration:

```
sudo netplan apply
```

Cloud Connectivity

CloudTuner.ai Integration

Authentication Setup

1. Obtain API Credentials:

- Log in to CloudTuner.ai dashboard
- Navigate to: Settings → InvEye → Edge Devices
- Generate new API key for your Jetson device

2. Configure Device:

```
# Set credentials

export INVEYE_API_KEY="ct-ai-xxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

export JETSON_DEVICE_ID="store-001-jetson-01"


# Save to environment

echo "export INVEYE_API_KEY='${INVEYE_API_KEY}'" >> ~/.profile

echo "export JETSON_DEVICE_ID='${JETSON_DEVICE_ID}'" >> ~/.profile
```

Cloud Sync Manager (scripts/cloud_sync.py)

```
import requests

import json

import time

import threading

from queue import Queue

import logging


logger = logging.getLogger('CloudSync')


class CloudSyncManager:

    def __init__(self, config):

        self.config = config['cloud']

        self.event_queue = Queue()

        self.running = False

    def start(self):
```

```

        """Start cloud sync thread"""

        self.running = True

        self.thread = threading.Thread(target=self._sync_loop)

        self.thread.daemon = True

        self.thread.start()

        logger.info("Cloud sync started")

    def queue_events(self, events):

        """Add events to sync queue"""

        for event in events:

            self.event_queue.put(event)

    def _sync_loop(self):

        """Sync events to cloud"""

        batch = []

        interval = self.config['sync']['interval']

        batch_size = self.config['sync']['batch_size']

        while self.running:

            # Collect events

            while not self.event_queue.empty() and len(batch) < batch_size:

                batch.append(self.event_queue.get())

            # Send batch if ready

            if batch:

                self._send_batch(batch)

                batch = []

            time.sleep(interval)

    def _send_batch(self, events):

```

```

"""Send event batch to cloud"""

endpoint = f"{self.config['endpoint']}/events"

headers = {

    'Authorization': f"Bearer {self.config['api_key']}",

    'Content-Type': 'application/json',

    'X-Device-ID': self.config['device_id']

}

payload = {

    'device_id': self.config['device_id'],

    'events': events,

    'timestamp': time.time()

}

try:

    response = requests.post(

        endpoint,

        headers=headers,

        json=payload,

        timeout=10

    )

    if response.status_code == 200:

        logger.info(f"Synced {len(events)} events to cloud")

    else:

        logger.error(f"Sync failed: {response.status_code}")

        # Re-queue events

        for event in events:

```

```

        self.event_queue.put(event)

    except Exception as e:

        logger.error(f"Sync error: {e}")

        # Re-queue events

        for event in events:

            self.event_queue.put(event)

def stop(self):

    """Stop sync thread"""

    self.running = False

    logger.info("Cloud sync stopped")

```

Monitoring & Metrics

Create a systemd service for automatic startup:

```
sudo nano /etc/systemd/system/inveye.service
```

```

[Unit]

Description=InvEye Edge AI Service

After=network.target


[Service]

Type=simple

User=inveye-admin

WorkingDirectory=/home/inveye-admin/inveye-edge

Environment="INVEYE_API_KEY=your-api-key"

```

```
Environment="JETSON_DEVICE_ID=your-device-id"

ExecStart=/home/inveye-admin/inveye-edge/venv/bin/python scripts/main.py

Restart=always

RestartSec=10


[Install]

WantedBy=multi-user.target
```

Enable and start service:

```
sudo systemctl enable inveye.service

sudo systemctl start inveye.service

sudo systemctl status inveye.service
```

Performance Optimization

Power Mode Configuration

Jetson Nano has two power modes:

```
# Check current power mode

sudo nvpmode1 -q


# Set to maximum performance (10W mode)

sudo nvpmode1 -m 0


# Set to power-saving mode (5W mode)
```



```
sudo nvpmode1 -m 1
```

Recommendation: Use 10W mode for production InvEye deployments.

GPU Clock Optimization

```
# Enable maximum clocks
```

```
sudo jetson_clocks
```

```
# Check current clocks
```

```
sudo jetson_clocks --show
```

Memory Management

```
# Create swap file for additional memory
```

```
sudo fallocate -l 4G /swapfile
```

```
sudo chmod 600 /swapfile
```

```
sudo mkswap /swapfile
```

```
sudo swapon /swapfile
```

```
# Make permanent
```

```
echo '/swapfile none swap sw 0 0' | sudo tee -a /etc/fstab
```

Inference Optimization Tips

1. **Use TensorRT** for maximum performance:

```
import tensorrt as trt
```

```
# Convert ONNX to TensorRT engine

def build_engine(onnx_path):

    logger = trt.Logger(trt.Logger.WARNING)

    builder = trt.Builder(logger)

    network = builder.create_network(1 <<
int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))

    parser = trt.OnnxParser(network, logger)

    with open(onnx_path, 'rb') as model:

        parser.parse(model.read())

    config = builder.create_builder_config()

    config.max_workspace_size = 1 << 30 # 1GB

    engine = builder.build_engine(network, config)

    return engine
```

2. **Reduce Resolution:** Process at 640x640 or 416x416 instead of full HD
3. **Frame Skipping:** Process every 2nd or 3rd frame for non-critical cameras
4. **Multi-threading:** Use separate threads for capture, inference, and upload

Monitoring System Performance

```
# Real-time monitoring

tegrastats

# GPU utilization

sudo -H pip3 install jetson-stats

jtop
```

Troubleshooting

Common Issues & Solutions

1. Camera Stream Not Connecting

Symptoms: RTSP stream fails to connect or times out

Solutions:

```
# Test network connectivity

ping <camera_ip>

# Check if RTSP port is accessible

telnet <camera_ip> 554

# Verify credentials and URL

ffmpeg -v trace -rtsp_transport tcp -i "rtsp://user:pass@ip:port/path" -c
copy -t 5 test.mp4

# Try different transport protocols

# In your code, add: cv2.CAP_PROP_BUFFERSIZE or use TCP transport
```

2. High CPU/GPU Temperature

Symptoms: System throttling, reduced performance

Solutions:

```
# Check temperature

tegrastats
```

```
# Ensure fan is running
```

```
sudo jetson_clocks --fan
```

```
# Improve airflow or add heatsink
```

```
# Monitor with:
```

```
watch -n 1 cat /sys/devices/virtual/thermal/thermal_zone*/temp
```

3. Out of Memory Errors

Symptoms: Application crashes, CUDA out of memory

Solutions:

```
# Check memory usage
```

```
free -h
```

```
# Reduce batch size in inference config
```

```
# Process fewer cameras simultaneously
```

```
# Add swap space (see Memory Management section)
```

```
# Clear cache
```

```
sudo sh -c 'echo 3 > /proc/sys/vm/drop_caches'
```

4. Low FPS / Inference Slow

Symptoms: Processing less than 10 FPS

Solutions:

```
# Enable maximum performance mode
```

```
sudo nvpmode1 -m 0

sudo jetson_clocks


# Optimize model:

# - Use smaller model (YOLOv8n instead of YOLOv8m)

# - Convert to TensorRT

# - Reduce input resolution


# Profile your application

python -m cProfile -o profile.stats scripts/main.py
```

5. Service Not Starting Automatically

Symptoms: InvEye service doesn't start on boot

Solutions:

```
# Check service status

sudo systemctl status inveye.service


# View logs

sudo journalctl -u inveye.service -f


# Ensure permissions

sudo chmod +x scripts/main.py


# Check paths in service file
```

```
sudo nano /etc/systemd/system/inveye.service
```

```
# Reload daemon
```

```
sudo systemctl daemon-reload
```

```
sudo systemctl enable inveye.service
```

Logging & Debugging

```
# Enable verbose logging
```

```
# In main.py, set logging level to DEBUG
```

```
logging.basicConfig(level=logging.DEBUG)
```

```
# Monitor logs in real-time
```

```
tail -f logs/inveye.log
```

```
# Check system logs
```

```
dmesg | tail -50
```

```
# GPU/CUDA errors
```

```
sudo dmesg | grep -i cuda
```

Maintenance & Best Practices

Regular Maintenance Tasks

Daily

- ☐ Monitor system health via CloudTuner.ai dashboard

- ☐ Check edge device status and connectivity
- ☐ Review critical alerts and events

Weekly

```
# Check disk space
```

```
df -h
```

```
# Review logs for errors
```

```
tail -n 100 logs/inveye.log | grep ERROR
```

```
# Monitor system temperature trends
```

```
jtop --log
```

Monthly

```
# System updates
```

```
sudo apt update && sudo apt upgrade -y
```

```
# Clean up old logs
```

```
find logs/ -name "*.log" -mtime +30 -delete
```

```
# Backup configuration
```

```
tar -czf config_backup_$(date +%Y%m%d).tar.gz config/
```

```
# Check SD card health
```

```
sudo smartctl -a /dev/mmcblk0
```

Security Best Practices

1. Change Default Passwords:

```
# Change user password

passwd

# Update camera credentials regularly
```

2. Firewall Configuration:

```
# Install UFW

sudo apt install ufw

# Allow only necessary ports

sudo ufw allow 22/tcp    # SSH

sudo ufw allow 8080/tcp  # InvEye API (if applicable)

sudo ufw enable
```

3. Secure API Keys:

```
# Use environment variables, never hardcode

# Store in secure location with restricted permissions

chmod 600 ~/.profile
```

4. Regular Security Updates:


```
# Enable automatic security updates

sudo apt install unattended-upgrades

sudo dpkg-reconfigure --priority=low unattended-upgrades
```

Backup & Recovery

Configuration Backup

```
#!/bin/bash

# backup_config.sh

BACKUP_DIR="/home/inveye-admin/backups"

DATE=$(date +%Y%m%d_%H%M%S)

mkdir -p $BACKUP_DIR

# Backup configurations

tar -czf $BACKUP_DIR/inveye_config_$DATE.tar.gz \

~/inveye-edge/config/ \

~/.profile \

/etc/systemd/system/inveye.service

echo "Backup created: inveye_config_$DATE.tar.gz"
```

Disaster Recovery

Create SD Card Image:

```
# From another Linux machine with SD card reader

sudo dd if=/dev/sdX of=jetson_backup.img bs=4M status=progress

gzip jetson_backup.img
```

Restore from Backup:

```
gunzip jetson_backup.img.gz

sudo dd if=jetson_backup.img of=/dev/sdX bs=4M status=progress
```

Performance Tuning

Camera Stream Optimization

```
# Use hardware-accelerated decoding

pipeline = (

    "rtspsrc location={rtsp_url} latency=0 ! "

    "rtph264depay ! h264parse ! "

    "nvv4l2decoder enable-max-performance=1 ! "

    "nvvidconv ! "

    "video/x-raw,format=BGRx ! "

    "videoconvert ! appsink"

)
```

Batch Processing

```
# Process multiple frames in batches

def batch_inference(frames, batch_size=4):
```

```

results = []

for i in range(0, len(frames), batch_size):

    batch = frames[i:i+batch_size]

    batch_results = model.infer(batch)

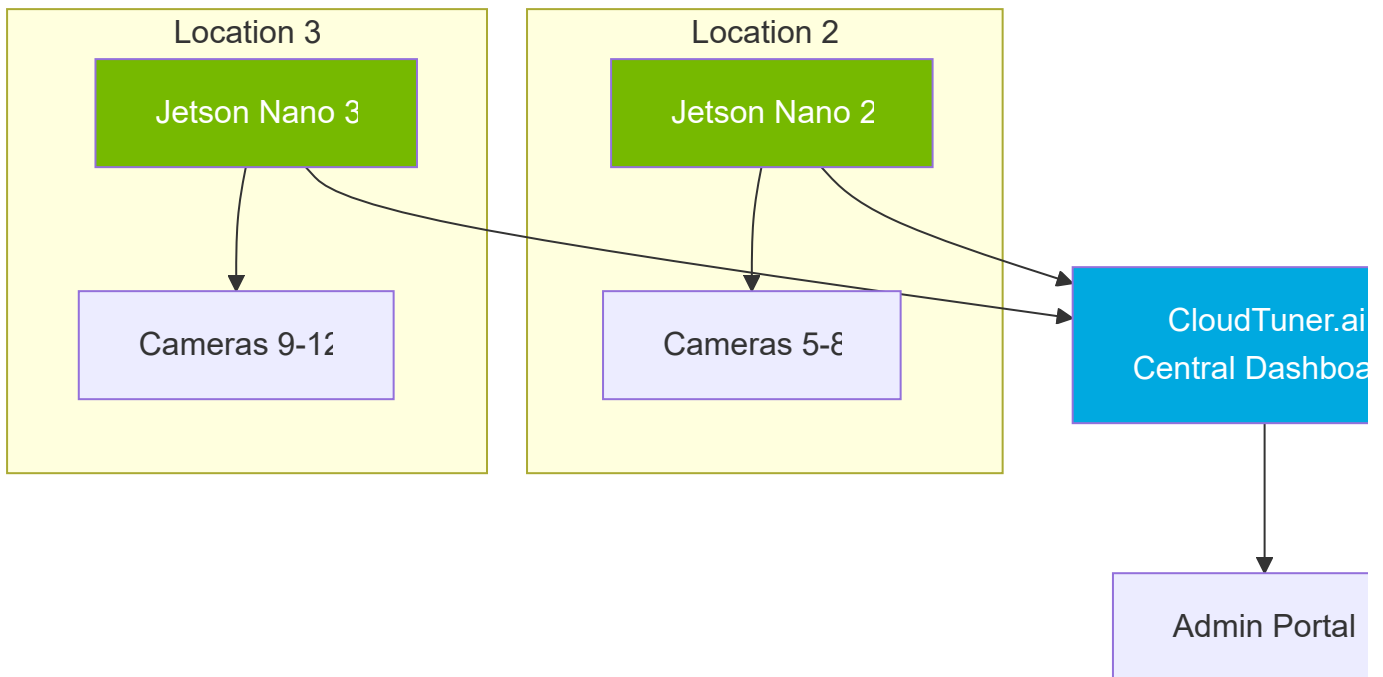
    results.extend(batch_results)

return results

```

Scaling to Multiple Devices

For multi-location deployments:



Device Management Best Practices:

- Use unique device IDs for each Jetson
- Centralize configuration via cloud dashboard
- Implement remote update mechanisms
- Monitor fleet health via aggregated metrics

Appendix

Useful Commands Reference

System Information

<code>cat /etc/nv_tegra_release</code>	# Jetson version
<code>uname -a</code>	# Kernel info
<code>nvidia-smi</code>	# GPU status
<code>tegrastats</code>	# Real-time stats

Performance

<code>sudo nvpmode1 -m 0</code>	# Max performance
<code>sudo jetson_clocks</code>	# Max clocks
<code>jtop</code>	# Interactive monitor

Networking

<code>ip addr show</code>	# Network interfaces
<code>sudo netstat -tulpn</code>	# Active connections
<code>ping -c 4 8.8.8.8</code>	# Test internet

Storage

<code>df -h</code>	# Disk usage
<code>du -sh ~/inveye-edge</code>	# Directory size
<code>lsblk</code>	# List block devices

Process Management

<code>ps aux grep python</code>	# Find Python processes
<code>top -u inveye-admin</code>	# Monitor user processes

```
kill -9 <PID> # Force kill process

# Logs

tail -f logs/inveye.log # Follow log file

journalctl -u inveye -n 50 # Service logs

dmesg | tail # Kernel messages
```

Resource Links

- **NVIDIA Jetson:**

- [Official Developer Site](#)
- [JetPack Documentation](#)
- [Jetson Projects Gallery](#)

- **InvEye Platform:**

- CloudTuner.ai Dashboard: <https://app.cloudtuner.ai>
- InvEye Documentation: <https://docs.cloudtuner.ai/inveye>
- Support: support@cloudtuner.ai

- **AI/ML Resources:**

- [NVIDIA DeepStream SDK](#)
- [TensorRT Documentation](#)
- [Ultralytics YOLOv8](#)

Glossary

Term	Definition
------	------------

-----	-----
-------	-------

RTSP	Real-Time Streaming Protocol for video streaming
-------------	--

ONNX	Open Neural Network Exchange format for AI models
-------------	---

TensorRT	NVIDIA's high-performance deep learning inference optimizer
-----------------	---

| **GStreamer** | Multimedia framework for video processing |

| **DeepStream** | NVIDIA's SDK for AI-based video analytics |

| **Edge Computing** | Processing data near the source rather than in the cloud |

| **Inference** | Running AI model predictions on new data |

| **KPI** | Key Performance Indicator |
