Backend & Python Scripts Documentation

Backend Architecture & Analysis Engine

Date: October 9, 2025

Structure: Reorganized Frontend/Backend

□ Backend Architecture

Technology Stack

- **Runtime:** Node.js v18+
- **Framework:** Express.js
- **Process Management:** Child Process (spawn)
- **Data Storage:** JSON files
- **AI/ML:** PyTorch, YOLOv8, OpenCV

□ New Backend Structure

```
backend/                        # All backend code
├── server.js                   # Node.js Express API server
├── backend.log                 # Server logs
│
├── analysis/                   # Python AI Analysis Modules
│   ├── main_v2.py             # Main orchestrator (entry point)
│   ├── enhanced_speed_detection.py         # Speed detection (consolidated)
│   ├── enhanced_proximity_detection.py     # Close encounter detection
│   ├── enhanced_traffic_detection.py       # Traffic violation detection
│   └── driving_score_calculator.py         # Scoring algorithm
│
├── utils/                      # Utility Scripts
│   ├── speed_graph.py         # Speed visualization
│   ├── check_gpu_status.py    # GPU diagnostics
│   └── model_manager.py       # Model management
│
├── models/                     # AI Models
│   ├── yolov8n.pt            # YOLOv8 nano (lightweight)
│   └── yolov8s.pt            # YOLOv8 small (recommended)
│
├── config/                     # Configuration
│   ├── analysis_config.json   # Analysis parameters
│   ├── improved_video_calibrations.json
│   ├── video_calibrations.json
│   └── requirements.txt       # Python dependencies
│
├── data/                       # Application Data
│   └── users.json             # User accounts & analysis history
│
├── videos/                     # Input Videos
│   ├── Dashcam001.mp4
│   ├── Dashcam002.mp4
│   └── ...
│
└── outputs/                    # Analysis Results
    ├── analysis/               # JSON analysis results
    ├── enhanced_analysis/      # Enhanced results
    └── logs/                   # Processing logs
```

Key Changes:

- □ All backend code in `/backend/` directory
- □ Core analysis modules in `/backend/analysis/`
- □ Utility scripts in `/backend/utils/`
- □ Clean separation from frontend
- □ Consolidated speed detection (5 files → 1 file)

□ Data Flow Summary

```
┌──────────────┐
│ User uploads │
│ video file   │
└──────┬───────┘
       ↓
┌──────────────┐     ┌─────────────┐
│   Backend    │────▶│   Python    │
│  (Node.js)   │     │  Analysis   │
└──────┬───────┘     └──────┬──────┘
       ↓                    ↓
┌──────────────┐     ┌─────────────┐
│              │     │  AI Models. │
│ Save video   │     │ YOLO + CV   │
└──────────────┘     └──────┬──────┘
                            ↓
                     ┌─────────────┐
                     │   Results   │
                     │ JSON files  │
                     └──────┬──────┘
                            ↓
                     ┌─────────────┐
                     │ Merged File │
                     │ (single src)│
                     └──────┬──────┘
                            ↓
┌──────────────┐     ┌─────────────┐
│   Frontend   │◀────│  API Call   │
│  Dashboard   │     │ GET /merged │
└──────────────┘     └─────────────┘
```

Python Analysis Scripts

1. main_v2.py - Main Analysis Orchestrator

Location: `backend/analysis/main_v2.py`

Purpose: Core analysis engine that processes videos and extracts all metrics.

How to Run:

```
cd backend
python analysis/main_v2.py
```

Key Components:

A. Video Metadata Extraction

```
def get_video_metadata(video_path: str) -> Dict:
    cap = cv2.VideoCapture(video_path)
    return {
        'duration_seconds': frame_count / fps,
        'fps': cap.get(cv2.CAP_PROP_FPS),
        'frame_count': int(cap.get(cv2.CAP_PROP_FRAME_COUNT)),
        'resolution': {
            'width': int(cap.get(cv2.CAP_PROP_FRAME_WIDTH)),
            'height': int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
        }
    }
```

B. Speed Detection Integration

```
# Import enhanced speed detection (consolidated module)
from enhanced_speed_detection import estimate_speed_multimethod

# Calculate average speed with multi-method detection
speed_result = estimate_speed_multimethod(video_path)
if speed_result.get('successful'):
    avg_speed = speed_result['average_speed_kmh']  # Realistic speeds (44.5 km/h city, 80-100 highway)
    confidence = speed_result['confidence']
```

C. Close Encounter Detection

```
# Import enhanced proximity detection
from enhanced_proximity_detection import detect_close_encounters_enhanced

# Detect close encounters with distance estimation
encounters = detect_close_encounters_enhanced(
    video_path=video_path,
    model_path='models/yolov8s.pt',  # Using yolov8s for best accuracy
    device='mps'  # Apple Silicon GPU acceleration
)
```

D. Traffic Signal Detection

```
def detect_traffic_signal_violations(video_path: str, model: YOLO) -> Dict:
    """
    Detect traffic light violations using YOLO + color analysis
    """
    cap = cv2.VideoCapture(video_path)
    fps = cap.get(cv2.CAP_PROP_FPS)

    violations = []
    red_light_frames = []

    while True:
        ret, frame = cap.read()
        if not ret:
            break

        # YOLOv8 detection
        results = model(frame, device='mps')

        for box in results[0].boxes:
            if box.cls == 9:  # Traffic light class
                # Extract ROI and analyze color
                x1, y1, x2, y2 = map(int, box.xyxy[0])
                roi = frame[y1:y2, x1:x2]

                # Convert to HSV for color detection
                hsv = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)

                # Red color thresholds
                mask1 = cv2.inRange(hsv, (0, 100, 100), (10, 255, 255))
                mask2 = cv2.inRange(hsv, (170, 100, 100), (180, 255, 255))
                red_mask = mask1 | mask2

                red_ratio = np.sum(red_mask > 0) / red_mask.size

                if red_ratio > 0.15:  # 15% red threshold
                    red_light_frames.append(frame_num)

    # Merge consecutive frames into violations
    violations = merge_consecutive_frames(red_light_frames, fps)

    return {
        'violations': violations,
        'violation': len(violations) > 0
    }
```

E. Turn Detection (ORB Features)

```
def run_turn_count_orb(video_path: str) -> Dict[str, int]:
    """
    Detect turns using ORB feature matching
    """
    TARGET_W = 320
    MATCH_THRESH = 20
    MIN_TURN_FRAMES = 5

    cap = cv2.VideoCapture(video_path)
    orb = cv2.ORB_create(nfeatures=200)
    bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

    left_count = 0
    right_count = 0
    turning = False
    turn_direction = 0

    while True:
        ret, frame = cap.read()
        if not ret:
            break

        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        gray = cv2.resize(gray, (TARGET_W, int(h * TARGET_W / w)))

        kp, des = orb.detectAndCompute(gray, None)

        if prev_des is not None:
            matches = bf.match(prev_des, des)

            # Calculate rotation from matches
            rotation = calculate_rotation_from_matches(matches, prev_kp, kp)

            if abs(rotation) > TURN_THRESHOLD:
                if rotation > 0:
                    right_count += 1
                else:
                    left_count += 1

        prev_des = des
        prev_kp = kp

    return {
        'turn_count': left_count + right_count,
        'left': left_count,
        'right': right_count
    }
```

F. Lane Change Detection (Optical Flow)

```python
def run_lane_change_count(video_path: str) -> Dict[str, int]:
    """
    Detect lane changes using Farneback optical flow

    Parameters configured for accuracy:
    - MAG_THRESH = 3.0 (strong motion required)
    - ENTER_THR = 0.75 (significant lateral movement)
    - MIN_TURN_SEC = 2.0 (sustained 2+ seconds)
    """
    cap = cv2.VideoCapture(video_path)
    fps = cap.get(cv2.CAP_PROP_FPS)

    # Configuration
    ROI_Y0 = int(height * 0.35)  # Focus on center road area
    ROI_Y1 = int(height * 0.65)

    left_count = 0
    right_count = 0
    ema = 0.0  # Exponential moving average

    while True:
        ret, frame = cap.read()
        if not ret:
            break

        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        roi = gray[ROI_Y0:ROI_Y1, :]

        # Calculate optical flow
        flow = cv2.calcOpticalFlowFarneback(
            prev_roi, roi, None,
            pyr_scale=0.5,
            levels=2,
            winsize=21,
            iterations=2,
            poly_n=5,
            poly_sigma=1.1,
            flags=0
        )

        u = flow[..., 0]  # Horizontal flow
        v = flow[..., 1]  # Vertical flow
        mag = np.hypot(u, v)

        # Filter strong motion
        mask = mag > MAG_THRESH

        if np.sum(mask) > 0:
            # Validate motion pattern
            motion_coverage = np.sum(mask) / mask.size
            mean_horizontal = np.mean(u[mask])
            mean_vertical = np.abs(np.mean(v[mask]))

            # Valid lane change criteria:
            # 1. Motion covers >25% of ROI
            # 2. Horizontal > 2x vertical
            # 3. Horizontal > 1.5 pixels
            if (motion_coverage > 0.25 and
                abs(mean_horizontal) > mean_vertical * 2.0 and
                abs(mean_horizontal) > 1.5):

                score = mean_horizontal / (mean_vertical + 1e-6)
            else:
                score = 0.0
        else:
            score = 0.0

        # Update exponential moving average
        ema = 0.12 * score + 0.88 * ema

        # Detect sustained lane change
        if abs(ema) >= ENTER_THR:
            if not turning:
                turning = True
                turn_direction = 1 if ema > 0 else -1
                frames_in_turn = 1
            else:
                frames_in_turn += 1
        else:
            if turning:
                duration_sec = frames_in_turn / fps
                if duration_sec >= MIN_TURN_SEC:
                    if turn_direction > 0:
                        right_count += 1
                    else:
                        left_count += 1
                turning = False

        prev_roi = roi

    return {
        'turn_count': left_count + right_count,
        'left': left_count,
        'right': right_count
    }
```

G. Bus Lane Violation Detection

```
def detect_bus_lane_violations(video_path: str) -> Dict:
    """
    Detect bus lane violations using red color detection
    """
    cap = cv2.VideoCapture(video_path)
    fps = cap.get(cv2.CAP_PROP_FPS)

    # ROI: Bottom-center of frame (bus lane area)
    ROI_WIDTH_FRAC = 0.22
    ROI_HEIGHT_FRAC = 0.18

    violation_frames = []

    while True:
        ret, frame = cap.read()
        if not ret:
            break

        h, w = frame.shape[:2]

        # Define ROI
        x1 = int(w * (0.5 - ROI_WIDTH_FRAC / 2))
        x2 = int(w * (0.5 + ROI_WIDTH_FRAC / 2))
        y1 = int(h * (1 - ROI_HEIGHT_FRAC - 0.04))
        y2 = int(h * (1 - 0.04))

        roi = frame[y1:y2, x1:x2]

        # Convert to HSV
        hsv = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)

        # Red color masks (0-10° and 170-180°)
        mask1 = cv2.inRange(hsv, (0, 80, 80), (10, 255, 255))
        mask2 = cv2.inRange(hsv, (170, 80, 80), (180, 255, 255))
        red_mask = mask1 | mask2

        # Calculate coverage
        coverage = np.sum(red_mask > 0) / red_mask.size

        if coverage > 0.12:  # 12% threshold
            violation_frames.append(frame_num)

    # Merge into violation ranges
    violations = merge_frames_to_ranges(violation_frames, fps)

    return {
        'violation_detected': len(violations) > 0,
        'violation_ranges': violations
    }
```

H. Safety Violation Count

```
def compute_safety_violation(
    traffic_windows: List[Dict],
    bus_ranges: List[Dict],
    close_count: int
) -> int:
    """
    Calculate total violation count

    Note: Close encounters are tracked separately,
    not counted as violations
    """
    traffic_violations = 1 if len(traffic_windows) > 0 else 0
    bus_violations = 1 if len(bus_ranges) > 0 else 0

    return traffic_violations + bus_violations
```

4. driving_score_calculator.py

Purpose: Calculate driving scores based on analysis metrics

```
def calculate_driving_scores(analysis_data: Dict) -> Dict:
    """
    Calculate comprehensive driving scores
    """
    # Extract metrics
    close_encounters = analysis_data.get('close_encounters', {}).get('event_count', 0)
    traffic_violations = len(analysis_data.get('traffic_signal_summary', {}).get('violations', []))
    bus_violations = 1 if analysis_data.get('illegal_way_bus_lane', {}).get('violation_detected') else 0
    lane_changes = analysis_data.get('lane_change_count', {}).get('turn_count', 0)
    avg_speed = analysis_data.get('average_speed_kmph', 0)
    duration = analysis_data.get('video_metadata', {}).get('duration_seconds', 1)

    # ==================== SAFETY SCORE ====================
    safety_base = 100

    # Close encounters penalty (8 points each)
    safety_base -= close_encounters * 8

    # Traffic violations penalty (15 points each)
    safety_base -= traffic_violations * 15

    safety_score = max(0, min(100, safety_base))

    # ==================== COMPLIANCE SCORE ====================
    compliance_base = 100

    # Traffic violations (20 points each)
    compliance_base -= traffic_violations * 20

    # Bus lane violations (15 points each)
    compliance_base -= bus_violations * 15

    compliance_score = max(0, min(100, compliance_base))

    # ==================== EFFICIENCY SCORE ====================
    efficiency_base = 100

    # Excessive lane changes penalty
    expected_lane_changes = duration / 60  # 1 per minute is normal
    excessive_changes = max(0, lane_changes - expected_lane_changes)
    efficiency_base -= excessive_changes * 2

    # Speed penalty (if too slow or too fast)
    if avg_speed > 0:
        if avg_speed < 20:  # Too slow
            efficiency_base -= 10
        elif avg_speed > 100:  # Too fast
            efficiency_base -= 20

    efficiency_score = max(0, min(100, efficiency_base))
    # ==================== OVERALL SCORE ====================
    overall_score = (
        safety_score * 0.4 +
        compliance_score * 0.3 +
        efficiency_score * 0.3
    )

    # Determine category
    if overall_score >= 90:
        category = 'Excellent'
        description = 'Outstanding driving performance with minimal safety concerns'
        color = 'green'
    elif overall_score >= 70:
        category = 'Good'
        description = 'Solid performance with minor areas for improvement'
        color = 'blue'
    elif overall_score >= 50:
        category = 'Needs Improvement'
        description = 'Performance requires attention and safety improvements'
        color = 'yellow'
    else:
        category = 'Poor'
        description = 'Serious driving concerns requiring immediate attention'
        color = 'red'

    return {
        'overall_score': int(overall_score),
        'safety_score': int(safety_score),
        'compliance_score': int(compliance_score),
        'efficiency_score': int(efficiency_score),
        'category': category,
        'category_description': description,
        'category_color': color,
        'metrics_used': {
            'close_encounters': close_encounters,
            'traffic_violations': traffic_violations,
            'bus_lane_violations': bus_violations,
            'lane_changes': lane_changes
        }
    }
```

□ AI Models

YOLOv8 (You Only Look Once v8)

Model Files:

- `yolov8n.pt` - Nano model (lightweight, fast)
- `yolov8s.pt` - Small model (better accuracy)

Classes Used:

- Class 2: Car
- Class 3: Motorcycle
- Class 5: Bus
- Class 7: Truck
- Class 9: Traffic Light