Galaxy ALPR Documentation

GalaxyALPR

Automatic License Plate Recognition

ALPR System Documentation

OPENTHEGATE HACK
BSS PARKING
06-25

Galaxy ALPR Core Documentation

📋 Table of Contents

🚀 Quick Start

Galaxy ALPR Core is a comprehensive AI-powered Automatic License Plate Recognition system that combines advanced computer vision models with generative AI for accurate vehicle and license plate detection and recognition.

Installation

1. **Clone the repository**

```
git clone <repository-url>
cd galaxy_alpr_core
```

2. **Install dependencies**

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

3. **Set up environment variables** Create a `.env` file in the project root:

```
GEMINI_API_KEY=your_gemini_api_key_here
```

4. **Download AI models** Place the following models in the `models/` directory:

- `model_vehicle_detector_yolo11n_v2.pt`
- `model_plate_detector_yolo11n_v3.pt`

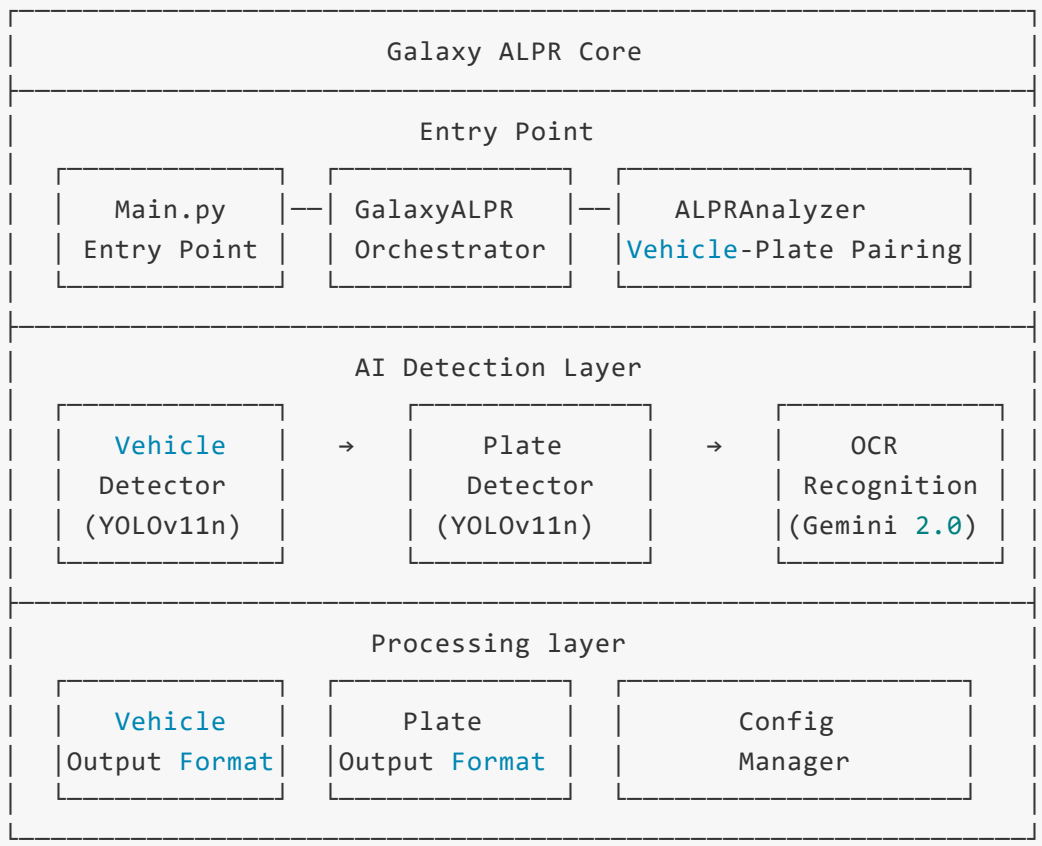
Basic Usage

```
from galaxy_alpr_core.main import run_alpr_image

# Process a single image
```

```
result = run_alpr_image("path/to/your/image.jpg")
print(result)
```

🏗️ System Architecture

Core Components



AI Models Used

1. Vehicle Detection Model

- **Model**: YOLOv11n.pt (v2)
- **Purpose**: Detects cars and motorcycles
- **Training Data**: 10,000 images
  - Training set: 7,000 images
  - Validation set: 2,000 images
  - Test set: 1,000 images
- **Classes**: Car, Motorcycle

2. Plate Detection Model

- **Model**: YOLOv11n.pt (v3)
- **Purpose**: Detects license plates
- **Training Data**: 10,000 images
  - Training set: 7,000 images
  - Validation set: 2,000 images
  - Test set: 1,000 images
- **Classes**: Plate

3. Plate OCR Model

- **Model**: Gemini 2.0 Flash-Lite
- **Purpose**: OCR for plate text extraction and attribute recognition
- **Provider**: Google AI Studio
- **Description**: Smallest and most cost-effective model, built for at-scale usage
- **Token Usage per Request**: ~1,834 tokens (1,715 prompt + 119 response)

Gemini 2.0 Flash-Lite Pricing & Limits:

Tier	Input Price	Output Price
Free Tier	Free of charge	Free of charge

Rate Limits (Free Tier):

- **Requests per minute (RPM)**: 30
- **Tokens per minute (TPM)**: 1,000,000
- **Requests per day (RPD)**: 1,500

Note: Rate limits are not guaranteed and actual capacity may vary

🔄 AI Workflow Pipeline

System Workflow Overview

```
Input Image → Vehicle Detection → Plate Detection → Plate Recognition → Pair Analysis
     ↓              ↓                  ↓                  ↓                    ↓
 [Original]    [YOLO 11n v2]      [YOLO 11n v3]    [Gemini 2.0 Flash]  [Smart Pairing]
                                                                            ↓
                                                                  ┌─ vehicle type
                                                                  ├─ plate number
                                                                  ├─ plate date
                                                                  ├─ plate type
                                                                  ├─ plate text color
                                                                  ├─ plate bg color
                                                                  └─ plate region
```

Visual Workflow Process

The system processes images through 5 main stages: Vehicle Detection using YOLO, Plate Detection using YOLO, OCR Recognition using Gemini AI, intelligent Vehicle-Plate pairing analysis, and structured output generation with comprehensive metadata.

Detailed Step-by-Step Process

Step 1: Vehicle Detection

- **Component**: `VehicleDetector`
- **AI Model**: YOLO 11n v2
- **Function**: Detects and crops vehicles (cars, motorcycles)
- **Input**: Original image
- **Output**: Vehicle bounding boxes, confidence scores, cropped vehicle images

```
# Configuration
vehicle_confidence_threshold: 0.25
vehicle_padding: 25 pixels
supported_classes: ["car", "motorcycle"]
```

Step 2: Vehicle Output Formatting

- **Component**: `VehicleOutputFormatter`
- **Function**: Structures vehicle detection results
- **Processing**: Assigns indices, formats paths, organizes metadata

Step 3: Plate Detection

- **Component**: `PlateDetector`

- **AI Model**: YOLO 11n v2
- **Function**: Detects license plates in vehicle images
- **Input**: Cropped vehicle images from Step 1
- **Output**: Plate bounding boxes, confidence scores, cropped plate images

```
# Configuration
plate_confidence_threshold: 0.25
plate_padding: 25 pixels
supported_classes: ["plate"]
```

Step 4: Plate Recognition (OCR)

- **Component**: `PlateRecognizer` + `PlateOCRGemini`
- **AI Model**: Google Gemini 2.0 Flash Lite
- **Function**: Extracts text and attributes from license plates
- **Advanced Features**:
  - Indonesian plate format recognition
  - Region code mapping (120+ regions)
  - Plate type classification (private, public, government, etc.)
  - Visual attribute detection (colors, icons, blue strips)

```
# OCR Output Structure
{
    "plate_number": "B 1234 CD",
    "plate_date": "08/29",
    "plate_text_color": "black",
    "plate_background_color": "white",
    "plate_icon": "government_seal",
    "plate_icon_color": "red",
    "plate_blue_strip": "no",
    "plate_type": "private",
```

```
    "confidence_score": 0.95,
    "plate_region_code": "B",
    "plate_region_name": "Jakarta/Metro Jaya"
}
```

Step 5: Results Combination

- **Function**: Merges detection and recognition data
- **Processing**: Combines plate detection metadata with OCR results

Step 6: Plate Output Formatting

- **Component**: `PlateOutputFormatter`
- **Function**: Structures final plate information with recognition data

Step 7: Vehicle-Plate Analysis & Pairing

- **Component**: `ALPRAnalyzer`
- **Function**: Intelligent pairing of vehicles with their license plates
- **Algorithms**:
  - **Containment Analysis**: Checks if plates are within vehicle bounding boxes
  - **Overlap Analysis**: Uses IoU (Intersection over Union) for partial overlaps
  - **Confidence Scoring**: Combines detection confidence with geometric relationship

```
# Pairing Methods
1. "containment" - Plate fully within vehicle box (preferred)
2. "overlap" - Plate partially overlaps with vehicle box
3. None - Unmatched vehicles or plates

# Detection Types
- "vehicle_with_plate" - Successfully paired
```

```
- "vehicle_without_plate" - Vehicle detected, no matching plate
- "plate_without_vehicle" - Plate detected, no matching vehicle
```

📚 API Reference

Core Classes

GalaxyALPR

Main orchestrator class for the complete ALPR pipeline.

```python
class GalaxyALPR:
    @staticmethod
    def run_alpr_image(input_image_path: str, timestamp: str = None) -> Dict:
        """
        Run complete ALPR pipeline on a single image.

        Args:
            input_image_path: Path to input image
            timestamp: Optional timestamp (YYYY-MM-DD_HH-MM-SS format)

        Returns:
            Complete ALPR results with vehicle-plate pairings
        """

    @staticmethod
    def detect_vehicle_from_image(input_image_path: str, timestamp: str = None) -> Dict:
        """Vehicle detection only"""

    @staticmethod
    def detect_plate_from_image(input_image_path: str, timestamp: str = None) -> Dict:
        """Plate detection only"""
```

```
    @staticmethod
    def ocr_plate_from_image(input_image_path: str) -> Dict:
        """OCR processing only"""

    @staticmethod
    def recognize_plate_from_image(input_image_path: str) -> Dict:
        """Complete plate recognition"""
```

VehicleDetector

YOLO-based vehicle detection component.

```
class VehicleDetector:
    def __init__(self, model_name: str = None):
        """Initialize with YOLO model"""

    def detect_vehicle_from_single_image(
        self,
        image: Union[str, np.ndarray],
        conf_threshold: float = None,
        padding: int = None,
        timestamp: str = ""
    ) -> Dict:
        """
        Detect vehicles in image.

        Returns:
            {
                "uploaded_image_path": str,
                "detected_vehicle_image_path": str,
                "list_cropped_vehicle_image_paths": List[str],
                "list_class_vehicle": List[str],
                "list_bounding_box_vehicle": List[List[int]],
                "list_confidence_score_vehicle": List[float],
                "vehicle_detection_processing_time": float
```

```
            }
        """
```

PlateDetector

YOLO-based license plate detection component.

```
class PlateDetector:
    def detect_plate_from_single_image(
        self,
        image: Union[str, np.ndarray],
        conf_threshold: float = None,
        padding: int = None,
        timestamp: str = ""
    ) -> Dict:
        """
        Detect license plates in image.

        Returns similar structure to VehicleDetector but for plates.
        """
```

PlateOCRGemini

Gemini AI-powered OCR for license plates.

```
class PlateOCRGemini:
    def __init__(self, model_name: str = None):
        """Initialize Gemini model"""

    def ocr_plate_from_single_image(
        self,
        image: Union[str, np.ndarray, Image.Image]
    ) -> List[dict]:
```

```
        """
        Extract text and attributes from single plate image.

        Returns:
            List of OCR results with comprehensive plate information
        """

    def ocr_plate_from_list_images(
        self,
        images: List[Union[str, np.ndarray, Image.Image]]
    ) -> List[dict]:
        """Batch processing of multiple plate images"""
```

ALPRAnalyzer

Intelligent vehicle-plate pairing system.

```
class ALPRAnalyzer:
    @staticmethod
    def analyze_vehicle_and_plate_pairing(
        vehicle_data: Dict,
        plate_data: Dict,
        timestamp: str
    ) -> Dict:
        """
        Analyze and pair vehicles with corresponding license plates.

        Uses containment analysis and overlap scoring for accurate pairing.

        Returns:
            {
                "timestamp": str,
                "uploaded_image_path": str,
                "detected_vehicle_image_path": str,
                "detected_plate_image_path": str,
```

```
                "processing_time": {
                    "vehicle_detection_ms": int,
                    "plate_detection_ms": int,
                    "plate_recognition_ms": int,
                    "total_ms": int
                },
                "summary": {
                    "total_detections": int,
                    "vehicles_with_plates": int,
                    "vehicles_without_plates": int,
                    "plates_without_vehicles": int
                },
                "detections": List[Detection]
            }
        """
```

Configuration Options

The system uses YAML configuration (`config/config.yaml`):

```yaml
models:
  vehicle_detector: models/model_vehicle_detector_yolo11n_v2.pt
  plate_detector: models/model_plate_detector_yolo11n_v3.pt

ocr:
  provider: gemini
  model_name: gemini-2.0-flash-lite-001

detection:
  vehicle_confidence_threshold: 0.25
  plate_confidence_threshold: 0.25
  vehicle_padding: 25
  plate_padding: 25

output:
```

```
  save_detected_images: true
  uploaded_vehicle_image_dir: images_processed/uploaded_vehicle_images
  uploaded_plate_image_dir: images_processed/uploaded_plate_images
  detected_vehicle_image_dir: images_processed/detected_vehicle_images
  cropped_vehicle_image_dir: images_processed/cropped_vehicle_images
  detected_plate_image_dir: images_processed/detected_plate_images
  cropped_plate_image_dir: images_processed/cropped_plate_images
  results_dir: images_processed/results

image_formats:
  - .jpg
  - .jpeg
  - .png

timezone: Asia/Makassar
```

🛠️ Advanced Usage

Custom Model Integration

```
# Use custom YOLO models
detector = VehicleDetector("path/to/custom/model.pt")

# Adjust detection parameters
result = detector.detect_vehicle_from_single_image(
    image="input.jpg",
    conf_threshold=0.5,  # Higher confidence
    padding=50           # More padding around detections
)
```

Batch Processing

```python
import os
from galaxy_alpr_core.main import run_alpr_image

def process_directory(input_dir: str, output_dir: str):
    """Process all images in a directory"""
    for filename in os.listdir(input_dir):
        if filename.lower().endswith(('.jpg', '.jpeg', '.png')):
            image_path = os.path.join(input_dir, filename)
            result = run_alpr_image(image_path)

            # Save results
            output_file = os.path.join(output_dir, f"{filename}_result.json")
            with open(output_file, 'w') as f:
                json.dump(result, f, indent=2)
```

Performance Tuning

```python
# Optimize for speed vs accuracy
config_fast = {
    'detection': {
        'vehicle_confidence_threshold': 0.5,  # Higher threshold = faster
        'plate_confidence_threshold': 0.5,
        'vehicle_padding': 10,                # Less padding = faster
        'plate_padding': 10
    }
}

# Optimize for accuracy
config_accurate = {
    'detection': {
        'vehicle_confidence_threshold': 0.1,  # Lower threshold = more detections
        'plate_confidence_threshold': 0.1,
        'vehicle_padding': 50,                # More padding = better context
        'plate_padding': 50
```

```
    }
}
```

🧪 Examples & Use Cases

Example 1: Basic ALPR Processing

```python
from galaxy_alpr_core.main import run_alpr_image
import json

# Process image
result = run_alpr_image("sample_image.jpg")

# Print summary
print(f"Total detections: {result['summary']['total_detections']}")
print(f"Vehicles with plates: {result['summary']['vehicles_with_plates']}")
print(f"Processing time: {result['processing_time']['total_ms']}ms")

# Access individual detections
for detection in result['detections']:
    if detection['detection_type'] == 'vehicle_with_plate':
        vehicle = detection['vehicle']
        plate = detection['plate']
        print(f"Vehicle {vehicle['vehicle_class']}: {plate['plate_number']}")
        print(f"Confidence: {detection['pairing_confidence']}")
```

Example 2: Component-Level Usage

```python
from galaxy_alpr_core.VehicleDetector import VehicleDetector
from galaxy_alpr_core.PlateDetector import PlateDetector
from galaxy_alpr_core.PlateOCRGemini import PlateOCRGemini
```

```python
# Step-by-step processing
vehicle_detector = VehicleDetector()
plate_detector = PlateDetector()
ocr = PlateOCRGemini()

# 1. Detect vehicles
vehicles = vehicle_detector.detect_vehicle_from_single_image("input.jpg")

# 2. Detect plates in vehicle images
plates = plate_detector.detect_plate_from_single_image(
    vehicles['detected_vehicle_image_path']
)

# 3. OCR on plate images
for plate_path in plates['list_cropped_plate_image_paths']:
    ocr_result = ocr.ocr_plate_from_single_image(plate_path)
    print(f"Plate text: {ocr_result[0]['plate_number']}")
```

Example 3: Indonesian Plate Types

The system recognizes various Indonesian license plate types:

```python
# Private vehicle plates
"B 1234 CD"     # Jakarta private vehicle (white background, black text)

# Public transport
"B 7890 UP"     # Jakarta public transport (yellow background, black text)

# Government vehicles
"RI 1"          # Government vehicle (red background, white text)

# Police vehicles
"POLRI 1234"    # Police vehicle (black background, white text, police badge)
```

```
# Military vehicles
"TNI AU 1234"   # Air Force vehicle (black background, white text, military star)

# Diplomatic vehicles
"CD 1234 A"     # Diplomatic corps (white background, blue text, diplomatic emblem)
```

Example 4: Region Code Mapping

```python
# The system automatically maps region codes to province names
region_examples = {
    'B': 'Jakarta/Metro Jaya',
    'D': 'Jawa Barat (Bandung)',
    'L': 'Surabaya',
    'AA': 'Jawa Tengah (Magelang)',
    'DK': 'Bali',
    'PA': 'Papua'
}

# Access region information
for detection in result['detections']:
    if detection['plate']:
        plate = detection['plate']
        print(f"Plate: {plate['plate_number']}")
        print(f"Region: {plate['plate_region_name']}")
```

📊 Output Format

Complete Result Structure

```
{
  "timestamp": "2025-06-04T20:32:03Z",
```

```
"uploaded_image_path": "images_processed/uploaded_vehicle_images/2025-06-04_20-32-03_upload
"detected_vehicle_image_path": "images_processed/detected_vehicle_images/2025-06-04_20-32-0
"detected_plate_image_path": "images_processed/detected_plate_images/2025-06-04_20-32-03_de
"processing_time": {
  "vehicle_detection_ms": 150,
  "plate_detection_ms": 120,
  "plate_recognition_ms": 800,
  "total_ms": 1070
},
"summary": {
  "total_detections": 2,
  "vehicles_with_plates": 1,
  "vehicles_without_plates": 0,
  "plates_without_vehicles": 1
},
"detections": [
  {
    "detection_id": 1,
    "detection_type": "vehicle_with_plate",
    "pairing_method": "containment",
    "pairing_confidence": 0.92,
    "vehicle": {
      "vehicle_index": 1,
      "vehicle_class": "car",
      "vehicle_confidence_score": 0.95,
      "vehicle_bounding_box": [100, 150, 400, 300],
      "vehicle_image_path": "images_processed/cropped_vehicle_images/2025-06-04_20-32-03_de
    },
    "plate": {
      "plate_index": 1,
      "plate_class": "plate",
      "plate_confidence_score": 0.88,
      "plate_bounding_box": [180, 250, 280, 290],
      "plate_image_path": "images_processed/cropped_plate_images/2025-06-04_20-32-03_detect
      "plate_number": "B 1234 CD",
      "plate_date": "08/29",
      "plate_text_color": "black",
```

```
        "plate_background_color": "white",
        "plate_icon": "",
        "plate_icon_color": "",
        "plate_blue_strip": "no",
        "plate_type": "private",
        "confidence_score": 0.95,
        "plate_region_code": "B",
        "plate_region_name": "Jakarta/Metro Jaya"
      }
    }
  ]
}
```


🔧 Installation & Dependencies

Required Packages

```
# Python general packages
python-dotenv
python-multipart
pytz
PyYAML

# Web API
fastapi
uvicorn

# Image processing
opencv-python
numpy
pillow
```

```
# AI / Computer Vision
ultralytics
easyocr

# Generative AI
google.generativeai
```

Environment Setup

1. **Python Requirements**: Python 3.8+
2. **System Requirements**:
    - GPU recommended for faster YOLO inference
    - 8GB+ RAM recommended
    - Stable internet connection for Gemini API
3. **API Keys**:
    - Google Gemini API key required
    - Set in `.env` file as `GEMINI_API_KEY`

Directory Structure

```
galaxy_alpr_core/
├── config/
│   ├── config.yaml          # Main configuration
│   └── config.py            # Configuration loader
├── models/                  # AI model files
│   ├── model_vehicle_detector_yolo11n_v2.pt
│   └── model_plate_detector_yolo11n_v3.pt
├── images_processed/        # Auto-created output directories
```

```
│   ├── uploaded_vehicle_images/
│   ├── detected_vehicle_images/
│   ├── cropped_vehicle_images/
│   ├── uploaded_plate_images/
│   ├── detected_plate_images/
│   ├── cropped_plate_images/
│   └── results/
├── tools/
│   └── Timer.py            # Timestamp utilities
├── ALPRAnalyzer.py         # Vehicle-plate pairing logic
├── GalaxyALPR.py          # Main orchestrator
├── PlateDetector.py       # Plate detection
├── PlateOCRGemini.py      # Gemini OCR integration
├── PlateRecognizer.py     # Plate recognition pipeline
├── PlateOutputFormatter.py # Plate result formatting
├── VehicleDetector.py     # Vehicle detection
├── VehicleOutputFormatter.py # Vehicle result formatting
└── main.py                # Entry point and examples
```

🚨 Troubleshooting

Common Issues

1. **Model Loading Errors**

   ```
   FileNotFoundError: Cannot find model file
   ```

   Solution: Ensure model files are in the `models/` directory with correct names.

2. **Gemini API Errors**

```
EnvironmentError: "GEMINI_API_KEY" is missing
```

Solution: Set up `.env` file with valid Gemini API key.

3. **Low Detection Accuracy Solution**: Adjust confidence thresholds in config or use higher quality input images.

4. **Performance Issues Solution**:

    - Use GPU for YOLO inference
    - Reduce image resolution
    - Increase confidence thresholds
    - Reduce padding values

Performance Benchmarks

Processing Times by Component

Typical processing times on different hardware configurations:

Hardware	Vehicle Detection	Plate Detection	OCR Recognition	Total
CPU Only	800-1500ms	600-1200ms	800-1500ms	2.2-4.2s
GPU (RTX 3060)	50-150ms	40-120ms	800-1500ms	0.9-1.8s
GPU (RTX 4090)	20-80ms	15-60ms	800-1500ms	0.8-1.6s

Note: OCR time is primarily network-dependent due to Gemini API calls

Gemini API Usage Statistics

Typical Token Consumption per Image:

- Prompt tokens: ~1,715
- Response tokens: ~119
- Total tokens: ~1,834

Daily Processing Capacity (Free Tier):

- Maximum requests per day: 1,500 images
- Maximum tokens per day: 1,000,000 tokens
- Estimated processing capacity: ~545 images/day (based on token limits)
- Requests per minute limit: 30 images/minute

Free Tier Benefits:

- Input processing: Free of charge
- Output generation: Free of charge
- **Total cost per image: $0.00 USD**
- Cost per 1,000 images: $0.00 USD

Model Performance Metrics

Vehicle Detection (YOLOv11n v2):

- Training dataset: 10,000 images (7k train, 2k validation, 1k test)
- Classes: Car, Motorcycle
- Default confidence threshold: 0.25

Plate Detection (YOLOv11n v3):

- Training dataset: 10,000 images (7k train, 2k validation, 1k test)
- Classes: Plate

- Default confidence threshold: 0.25

Plate OCR (Gemini 2.0 Flash-Lite):

- Specialized for Indonesian license plates
- Supports 120+ regional codes
- Confidence scoring: 0.0-1.0 scale
- Advanced attribute recognition (colors, icons, plate types)

🤝 Contributing

Development Guidelines

1. **Code Style**: Follow PEP 8 conventions
2. **Documentation**: Add docstrings to all public methods
3. **Testing**: Test with various image types and quality levels
4. **Configuration**: Use config system for all parameters
5. **Error Handling**: Implement robust error handling and logging

Extending the System

To add new features:

1. **New Detection Models**: Extend `VehicleDetector` or `PlateDetector` classes
2. **Additional OCR Providers**: Implement new OCR classes following `PlateOCRGemini` pattern
3. **Custom Analysis**: Extend `ALPRAnalyzer` for specialized pairing logic
4. **Output Formats**: Create new formatter classes for different output requirements

📄 License

This project is developed and maintained by **@GalaxyDeveloper**.

🏷️ Citation

If you use Galaxy ALPR Core in your research or projects, please cite:

```
Galaxy ALPR Core - Advanced AI-Powered License Plate Recognition System
Developed by @GalaxyDeveloper (2025)
Built with YOLOv11n, Google Gemini AI, and intelligent pairing algorithms
```

Galaxy ALPR Core - Advanced AI-Powered License Plate Recognition System
Built with YOLOv11n, Google Gemini AI, and intelligent pairing algorithms

Developed by @GalaxyDeveloper - 2025

GalaxyALPR

Automatic License Plate Recognition

AI/ML Service Documentation

OPENTHEGATE HACK
BSS PARKING
06-25

Galaxy ALPR AI/ML Services Documentation

📋 Table of Contents

🚀 Quick Start

Installation

```
# Clone the repository
git clone <repository-url>
```

```
cd galaxy_alpr_core

# Install dependencies
pip install -r requirements.txt

# Install API-specific dependencies
pip install fastapi uvicorn python-multipart
```

Environment Setup

1. **Create `.env` file:**

   ```
   GEMINI_API_KEY=your_gemini_api_key_here
   ```

2. **Download AI models** and place in `models/` directory:
   - `model_vehicle_detector_yolo11n_v2.pt`
   - `model_plate_detector_yolo11n_v3.pt`

3. **Ensure directory structure:**

   ```
   galaxy_alpr_core/
   ├── API.py                    # Main API file
   ├── galaxy_alpr_core/         # Core ALPR modules
   ├── models/                   # AI model files
   ├── images_processed/         # Auto-created output directories
   └── .env                      # Environment variables
   ```

Running the API

```
# Start the API server
python API.py

# Or using uvicorn directly
uvicorn API:app --host 0.0.0.0 --port 8000 --reload
```

API Access:

- **Swagger Documentation**: http://localhost:8000/docs
- **ReDoc Documentation**: http://localhost:8000/redoc
- **Health Check**: http://localhost:8000/galaxy-alpr/v1/health

---

🏗️ API Overview

Base URL Structure

All API endpoints follow the versioned structure:

```
http://localhost:8000/galaxy-alpr/v1/{endpoint}
```

Authentication

Currently, the API does not require authentication. All endpoints are publicly accessible.

Response Format

All endpoints return JSON responses with consistent structure:

```
{
  "success": true,
  "message": "Operation completed successfully",
  "data": { /* Result data */ },
  "processing_time_ms": 1250,
  "timestamp": "2025-06-06T14:30:15.123456"
}
```

📚 API Endpoints

Information Endpoints

`GET /galaxy-alpr/v1`

Root endpoint with API information

Response:

```
{
  "message": "Galaxy ALPR Core API",
  "description": "Advanced AI-Powered Automatic License Plate Recognition System",
  "version": "1.0.0",
  "developer": "@GalaxyDeveloper",
  "year": "2025",
  "documentation": "/docs",
  "health": "/galaxy-alpr/v1/health",
  "pipeline_info": "/galaxy-alpr/v1/pipeline/info"
}
```

`GET /galaxy-alpr/v1/health`

Health check endpoint

Response:

```
{
  "status": "healthy",
  "version": "1.0.0",
  "uptime": "2:30:45.123456",
  "models_loaded": {
    "vehicle_detector": true,
    "plate_detector": true,
    "gemini_ocr": true
  },
  "timestamp": "2025-06-06T14:30:15.123456"
}
```

Status Values:

- `healthy` - All models loaded and API functioning normally
- `degraded` - Some models missing or issues detected

`GET /galaxy-alpr/v1/pipeline/info`

Get detailed information about the ALPR pipeline

Response:

```
{
  "success": true,
  "pipeline_info": {
    "pipeline_name": "ALPR Core Pipeline",
```

```
    "description": "Complete Automatic License Plate Recognition system",
    "steps": [
      {
        "step": 1,
        "name": "Vehicle Detection",
        "description": "Detect vehicles in the input image",
        "component": "VehicleDetector"
      }
      // ... additional steps
    ],
    "input": "Image file path and optional timestamp",
    "output": "Final paired vehicle and plate detection/recognition results"
  },
  "timestamp": "2025-06-06T14:30:15.123456"
}
```

ALPR Core Endpoints

`POST /galaxy-alpr/v1/alpr-core/process-single-image`

Process single image through complete ALPR pipeline

Parameters:

- `file` (required): Image file (JPG, JPEG, PNG)
- `custom_timestamp` (optional): Custom timestamp in format "YYYY-MM-DD_HH-MM-SS"

Example Request:

```
curl -X POST "http://localhost:8000/galaxy-alpr/v1/alpr-core/process-single-image" \
  -H "Content-Type: multipart/form-data" \
  -F "file=@sample_image.jpg" \
  -F "custom_timestamp=2025-06-06_14-30-15"
```

Response Structure:


```
{
  "success": true,
  "message": "ALPR processing completed successfully",
  "data": {
    "timestamp": "2025-06-06T14:30:15Z",
    "uploaded_image_path": "images_processed/uploaded_vehicle_images/2025-06-06_14-30-15_uplo
    "detected_vehicle_image_path": "images_processed/detected_vehicle_images/2025-06-06_14-30
    "detected_plate_image_path": "images_processed/detected_plate_images/2025-06-06_14-30-15_
    "processing_time": {
      "vehicle_detection_ms": 150,
      "plate_detection_ms": 120,
      "plate_recognition_ms": 800,
      "total_ms": 1070
    },
    "summary": {
      "total_detections": 2,
      "vehicles_with_plates": 1,
      "vehicles_without_plates": 0,
      "plates_without_vehicles": 1
    },
    "detections": [
      {
        "detection_id": 1,
        "detection_type": "vehicle_with_plate",
        "pairing_method": "containment",
        "pairing_confidence": 0.92,
        "vehicle": {
          "vehicle_index": 1,
          "vehicle_class": "car",
          "vehicle_confidence_score": 0.95,
          "vehicle_bounding_box": [100, 150, 400, 300],
          "vehicle_image_path": "images_processed/cropped_vehicle_images/2025-06-06_14-30-15_
        },
        "plate": {
```

```
          "plate_index": 1,
          "plate_class": "plate",
          "plate_confidence_score": 0.88,
          "plate_bounding_box": [180, 250, 280, 290],
          "plate_image_path": "images_processed/cropped_plate_images/2025-06-06_14-30-15_dete
          "plate_number": "B 1234 CD",
          "plate_date": "08/29",
          "plate_text_color": "black",
          "plate_background_color": "white",
          "plate_icon": "",
          "plate_icon_color": "",
          "plate_blue_strip": "no",
          "plate_type": "private",
          "confidence_score": 0.95,
          "plate_region_code": "B",
          "plate_region_name": "Jakarta/Metro Jaya"
        }
      }
    ]
  },
  "processing_time_ms": 1250,
  "timestamp": "2025-06-06T14:30:15.123456"
}
```


`POST /galaxy-alpr/v1/alpr-core/process-multiple-images`

Process multiple images through complete ALPR pipeline

Parameters:

- `files` (required): Array of image files (max 10 files)
- `custom_timestamp` (optional): Custom timestamp prefix

Example Request:

```
curl -X POST "http://localhost:8000/galaxy-alpr/v1/alpr-core/process-multiple-images" \
  -H "Content-Type: multipart/form-data" \
  -F "files=@image1.jpg" \
  -F "files=@image2.jpg" \
  -F "custom_timestamp=batch_2025-06-06"
```

Response Structure:

```
{
  "success": true,
  "message": "Batch processing completed for 2 images",
  "data": {
    "total_images": 2,
    "results": [
      {
        "batch_index": 1,
        "original_filename": "image1.jpg",
        // ... complete ALPR result for image1
      },
      {
        "batch_index": 2,
        "original_filename": "image2.jpg",
        // ... complete ALPR result for image2
      }
    ],
    "batch_summary": {
      "total_detections": 3,
      "vehicles_with_plates": 2,
      "vehicles_without_plates": 0,
      "plates_without_vehicles": 1
    }
  },
  "processing_time_ms": 2500,
```

```
  "timestamp": "2025-06-06T14:30:15.123456"
}
```

Component-Specific Endpoints

`POST /galaxy-alpr/v1/vehicle/detect-vehicle`

Vehicle detection only

Parameters:

- `file` (required): Image file
- `custom_timestamp` (optional): Custom timestamp

Response:

```
{
  "success": true,
  "message": "Vehicle detection completed successfully",
  "data": {
    "uploaded_image_path": "path/to/uploaded_image.jpg",
    "detected_vehicle_image_path": "path/to/detected_image.jpg",
    "list_cropped_vehicle_image_paths": ["path/to/vehicle1.jpg", "path/to/vehicle2.jpg"],
    "list_class_vehicle": ["car", "motorcycle"],
    "list_bounding_box_vehicle": [[100, 150, 400, 300], [450, 200, 650, 350]],
    "list_confidence_score_vehicle": [0.95, 0.87],
    "vehicle_detection_processing_time": 0.15
  },
  "processing_time_ms": 180,
  "timestamp": "2025-06-06T14:30:15.123456"
}
```

`POST /galaxy-alpr/v1/plate/detect-plate`

License plate detection only

Parameters:

- `file` (required): Image file
- `custom_timestamp` (optional): Custom timestamp

Response:

```
{
  "success": true,
  "message": "Plate detection completed successfully",
  "data": {
    "uploaded_image_path": "path/to/uploaded_image.jpg",
    "detected_plate_image_path": "path/to/detected_image.jpg",
    "list_cropped_plate_image_paths": ["path/to/plate1.jpg"],
    "list_class_plate": ["plate"],
    "list_bounding_box_plate": [[180, 250, 280, 290]],
    "list_confidence_score_plate": [0.88],
    "plate_detection_processing_time": 0.12
  },
  "processing_time_ms": 150,
  "timestamp": "2025-06-06T14:30:15.123456"
}
```

`POST /galaxy-alpr/v1/plate/ocr-plate`

OCR processing only

Parameters:

- `file` (required): Plate image file

Response:


```
{
  "success": true,
  "message": "Plate OCR completed successfully",
  "data": [
    {
      "plate_number": "B 1234 CD",
      "plate_date": "08/29",
      "plate_text_color": "black",
      "plate_background_color": "white",
      "plate_icon": "",
      "plate_icon_color": "",
      "plate_blue_strip": "no",
      "plate_type": "private",
      "confidence_score": 0.95
    }
  ],
  "processing_time_ms": 850,
  "timestamp": "2025-06-06T14:30:15.123456"
}
```


`POST /galaxy-alpr/v1/plate/recognize-plate`

Complete plate recognition (OCR + regional mapping)

Parameters:

- `file` (required): Plate image file

Response:


```
{
  "success": true,
```

```
  "message": "Plate recognition completed successfully",
  "data": {
    "list_result_plate_recognized": [
      {
        "plate_number": "B 1234 CD",
        "plate_date": "08/29",
        "plate_text_color": "black",
        "plate_background_color": "white",
        "plate_icon": "",
        "plate_icon_color": "",
        "plate_blue_strip": "no",
        "plate_type": "private",
        "confidence_score": 0.95,
        "plate_region_code": "B",
        "plate_region_name": "Jakarta/Metro Jaya"
      }
    ],
    "plate_recognition_processing_time": 0.85
  },
  "processing_time_ms": 900,
  "timestamp": "2025-06-06T14:30:15.123456"
}
```


System Management Endpoints

`GET /galaxy-alpr/v1/config`

Get current system configuration

Response:


```
{
  "success": true,
  "configuration": {
```

```
    "models": {
      "vehicle_detector": "model_vehicle_detector_yolo11n_v2.pt",
      "plate_detector": "model_plate_detector_yolo11n_v3.pt"
    },
    "ocr": {
      "provider": "gemini",
      "model_name": "gemini-2.0-flash-lite-001"
    },
    "detection": {
      "vehicle_confidence_threshold": 0.25,
      "plate_confidence_threshold": 0.25,
      "vehicle_padding": 25,
      "plate_padding": 25
    },
    "output": {
      "uploaded_vehicle_image_dir": "images_processed/uploaded_vehicle_images",
      "detected_vehicle_image_dir": "images_processed/detected_vehicle_images",
      "cropped_vehicle_image_dir": "images_processed/cropped_vehicle_images",
      "uploaded_plate_image_dir": "images_processed/uploaded_plate_images",
      "detected_plate_image_dir": "images_processed/detected_plate_images",
      "cropped_plate_image_dir": "images_processed/cropped_plate_images",
      "results_dir": "images_processed/results"
    },
    "image_formats": [".jpg", ".jpeg", ".png"],
    "timezone": "Asia/Makassar"
  },
  "timestamp": "2025-06-06T14:30:15.123456"
}
```


`GET /galaxy-alpr/v1/stats`

Get system statistics and status

Response:

```
{
  "success": true,
  "statistics": {
    "system": {
      "uptime": "2:30:45.123456",
      "uptime_seconds": 9045.123456,
      "start_time": "2025-06-06T12:00:00.000000",
      "current_time": "2025-06-06T14:30:15.123456"
    },
    "models": {
      "vehicle_detector_loaded": true,
      "plate_detector_loaded": true,
      "gemini_api_configured": true
    },
    "configuration": {
      "vehicle_confidence_threshold": 0.25,
      "plate_confidence_threshold": 0.25,
      "supported_formats": [".jpg", ".jpeg", ".png"],
      "timezone": "Asia/Makassar"
    }
  },
  "timestamp": "2025-06-06T14:30:15.123456"
}
```


Utility Endpoints

`GET /galaxy-alpr/v1/images/{image_path}`

Retrieve processed images

Parameters:

- `image_path` (path): Relative path to the image file

Example:

```
GET /galaxy-alpr/v1/images/uploaded_vehicle_images/2025-06-06_14-30-15_uploaded_image.jpg
GET /galaxy-alpr/v1/images/cropped_plate_images/2025-06-06_14-30-15_detected_plate1.jpg
```

Response: Returns the image file directly

Testing Endpoints

`GET /galaxy-alpr/v1/test/connection`

Test API connection

Response:

```
{
  "status": "connected",
  "message": "Galaxy ALPR API is running",
  "timestamp": "2025-06-06T14:30:15.123456"
}
```

`GET /galaxy-alpr/v1/test/models`

Test model loading and availability

Response:

```
{
  "success": true,
  "models": {
    "vehicle_detector": {
```

```
      "status": "loaded",
      "model_path": "/path/to/model_vehicle_detector_yolo11n_v2.pt",
      "classes": {
        "0.0": "car",
        "1.0": "motorcycle"
      }
    },
    "plate_detector": {
      "status": "loaded",
      "model_path": "/path/to/model_plate_detector_yolo11n_v3.pt",
      "classes": {
        "0.0": "plate"
      }
    },
    "gemini_ocr": {
      "status": "configured",
      "model_name": "gemini-2.0-flash-lite-001",
      "api_key_configured": true
    }
  },
  "timestamp": "2025-06-06T14:30:15.123456"
}
```

🔗 Request/Response Models

Standard Response Models

ALPRResponse

```
{
  "success": bool,                    # Request success status
  "message": str,                     # Response message
```

```
  "data": Optional[Dict],              # ALPR processing results
  "processing_time_ms": Optional[int], # Total processing time in milliseconds
  "timestamp": str                     # Processing timestamp (ISO format)
}
```

HealthResponse

```
{
  "status": str,                       # API health status ("healthy"/"degraded")
  "version": str,                      # API version
  "uptime": str,                       # API uptime
  "models_loaded": Dict[str, bool],    # Model loading status
  "timestamp": str                     # Health check timestamp
}
```

Error Handling

ErrorResponse

```
{
  "success": false,
  "error": str,                        # Error message
  "error_type": str,                   # Type of error
  "timestamp": str                     # Error timestamp
}
```

Common HTTP Status Codes

- **200 OK** - Request successful
- **400 Bad Request** - Invalid request parameters or unsupported file format

- **404 Not Found** - Image file not found
- **500 Internal Server Error** - Server processing error

🧪 Usage Examples

Complete ALPR Processing

Single Image:

```
curl -X POST "http://localhost:8000/galaxy-alpr/v1/alpr-core/process-single-image" \
  -H "Content-Type: multipart/form-data" \
  -F "file=@car_with_plate.jpg"
```

Multiple Images:

```
curl -X POST "http://localhost:8000/galaxy-alpr/v1/alpr-core/process-multiple-images" \
  -H "Content-Type: multipart/form-data" \
  -F "files=@image1.jpg" \
  -F "files=@image2.jpg" \
  -F "files=@image3.jpg"
```

Component-Level Processing

Vehicle Detection:

```
curl -X POST "http://localhost:8000/galaxy-alpr/v1/vehicle/detect-vehicle" \
  -H "Content-Type: multipart/form-data" \
```

```
  -F "file=@street_scene.jpg"
```

Plate Detection:

```
curl -X POST "http://localhost:8000/galaxy-alpr/v1/plate/detect-plate" \
  -H "Content-Type: multipart/form-data" \
  -F "file=@car_front.jpg"
```

Plate OCR:

```
curl -X POST "http://localhost:8000/galaxy-alpr/v1/plate/ocr-plate" \
  -H "Content-Type: multipart/form-data" \
  -F "file=@license_plate.jpg"
```

System Monitoring

Health Check:

```
curl -X GET "http://localhost:8000/galaxy-alpr/v1/health"
```

System Statistics:

```
curl -X GET "http://localhost:8000/galaxy-alpr/v1/stats"
```

Configuration:

```
curl -X GET "http://localhost:8000/galaxy-alpr/v1/config"
```

Python Client Example

```python
import requests
import json

# API base URL
BASE_URL = "http://localhost:8000/galaxy-alpr/v1"

def process_single_image(image_path):
    """Process a single image through ALPR pipeline"""
    url = f"{BASE_URL}/alpr-core/process-single-image"

    with open(image_path, 'rb') as f:
        files = {'file': f}
        response = requests.post(url, files=files)

    if response.status_code == 200:
        result = response.json()
        print(f"Success: {result['success']}")
        print(f"Processing time: {result['processing_time_ms']}ms")
        print(f"Detections: {len(result['data']['detections'])}")
        return result
    else:
        print(f"Error: {response.status_code}")
        print(response.json())
        return None

def check_health():
    """Check API health status"""
    url = f"{BASE_URL}/health"
    response = requests.get(url)

    if response.status_code == 200:
        health = response.json()
        print(f"Status: {health['status']}")
        print(f"Uptime: {health['uptime']}")
```

```
        print(f"Models loaded: {health['models_loaded']}")
        return health
    else:
        print(f"Health check failed: {response.status_code}")
        return None

# Usage
if __name__ == "__main__":
    # Check API health
    health_status = check_health()

    # Process an image
    if health_status and health_status['status'] == 'healthy':
        result = process_single_image("path/to/your/image.jpg")
        if result:
            # Process the results
            for detection in result['data']['detections']:
                if detection['detection_type'] == 'vehicle_with_plate':
                    vehicle = detection['vehicle']
                    plate = detection['plate']
                    print(f"Vehicle: {vehicle['vehicle_class']}")
                    print(f"Plate: {plate['plate_number']}")
                    print(f"Region: {plate['plate_region_name']}")
```

⚙️ Configuration

System Configuration

The API uses YAML configuration in `galaxy_alpr_core/config/config.yaml`:

```
models:
  vehicle_detector: models/model_vehicle_detector_yolo11n_v2.pt
```

```yaml
  plate_detector: models/model_plate_detector_yolo11n_v3.pt

ocr:
  provider: gemini
  model_name: gemini-2.0-flash-lite-001

detection:
  vehicle_confidence_threshold: 0.25
  plate_confidence_threshold: 0.25
  vehicle_padding: 25
  plate_padding: 25

output:
  save_detected_images: true
  uploaded_vehicle_image_dir: images_processed/uploaded_vehicle_images
  detected_vehicle_image_dir: images_processed/detected_vehicle_images
  cropped_vehicle_image_dir: images_processed/cropped_vehicle_images
  uploaded_plate_image_dir: images_processed/uploaded_plate_images
  detected_plate_image_dir: images_processed/detected_plate_images
  cropped_plate_image_dir: images_processed/cropped_plate_images
  results_dir: images_processed/results

image_formats:
  - .jpg
  - .jpeg
  - .png

timezone: Asia/Makassar
```

File Upload Limits

- **Maximum file size**: Determined by FastAPI/uvicorn settings
- **Supported formats**: JPG, JPEG, PNG
- **Batch processing limit**: Maximum 10 files per request

- **Temporary file cleanup**: Automatic cleanup via background tasks

---

🚨 Error Codes & Troubleshooting

Common Error Codes

400 Bad Request

```
{
  "success": false,
  "error": "Unsupported file format. Allowed formats: ['.jpg', '.jpeg', '.png']",
  "error_type": "HTTPException",
  "timestamp": "2025-06-06T14:30:15.123456"
}
```

Causes:

- Unsupported file format
- Invalid custom timestamp format
- Too many files in batch processing (>10)

404 Not Found

```
{
  "success": false,
  "error": "Image not found",
  "error_type": "HTTPException",
  "timestamp": "2025-06-06T14:30:15.123456"
}
```

Causes:

- Requested image file doesn't exist
- Invalid image path

500 Internal Server Error

```
{
  "success": false,
  "error": "ALPR processing failed: Model not found",
  "error_type": "HTTPException",
  "timestamp": "2025-06-06T14:30:15.123456"
}
```

Causes:

- Missing AI model files
- Invalid GEMINI_API_KEY
- Internal processing errors
- Insufficient system resources

Troubleshooting Guide

Model Loading Issues

Problem: Vehicle/Plate detector not loading

```
{
  "vehicle_detector": false,
```

```
  "plate_detector": false
}
```

Solution:

1. Verify model files exist in `models/` directory
2. Check file permissions
3. Ensure correct model file names in config

Gemini API Issues

Problem: OCR processing failures

```
{
  "gemini_ocr": false
}
```

Solution:

1. Verify `GEMINI_API_KEY` in `.env` file
2. Check API key validity
3. Verify internet connection
4. Check Gemini API rate limits

File Upload Issues

Problem: File upload failures

Solution:

1. Check file format (must be JPG, JPEG, or PNG)

2. Verify file size limits
3. Ensure proper multipart/form-data encoding
4. Check disk space for temporary files

Performance Issues

Problem: Slow processing times

Solution:

1. Use GPU acceleration for YOLO models
2. Reduce image resolution
3. Increase confidence thresholds
4. Monitor system resources

📊 Performance & Limitations

Processing Times

Component	CPU Only	GPU (RTX 3060)	GPU (RTX 4090)
Vehicle Detection	800-1500ms	50-150ms	20-80ms
Plate Detection	600-1200ms	40-120ms	15-60ms
OCR Recognition	800-1500ms	800-1500ms	800-1500ms
Total	**2.2-4.2s**	**0.9-1.8s**	**0.8-1.6s**

Note: OCR time is network-dependent due to Gemini API calls

Rate Limits

Gemini API Limits (Free Tier):

- **Requests per minute**: 30
- **Tokens per minute**: 1,000,000
- **Requests per day**: 1,500

API Server Limits:

- **Batch processing**: Maximum 10 files per request
- **Concurrent requests**: Limited by server resources
- **File upload timeout**: 60 seconds (configurable)

File Size Limits

- **Maximum image size**: 20MB (recommended: <5MB for faster processing)
- **Minimum image resolution**: 224x224 pixels
- **Recommended resolution**: 640x480 to 1920x1080 pixels

System Requirements

Minimum Requirements:

- **RAM**: 8GB
- **Storage**: 10GB free space
- **Python**: 3.8+
- **Internet**: Stable connection for Gemini API

Recommended Requirements:

- **RAM**: 16GB+
- **GPU**: NVIDIA GPU with 6GB+ VRAM
- **Storage**: 50GB+ free space
- **CPU**: 8+ cores

🏷️ Citation

If you use Galaxy ALPR API in your projects, please cite:

```
Galaxy ALPR Core API - Advanced AI-Powered License Plate Recognition System
Developed by @GalaxyDeveloper (2025)
Built with FastAPI, YOLOv11n, Google Gemini AI, and intelligent pairing algorithms
```

Galaxy ALPR Core API - Advanced AI-Powered License Plate Recognition System
Built with FastAPI, YOLOv11n, Google Gemini AI, and intelligent pairing algorithms

Developed by @GalaxyDeveloper - 2025

GalaxyALPR

Automatic License Plate Recognition

REST API Documentation

OPENTHEGATE HACK
BSS PARKING
06-25

Galaxy ALPR REST API

A robust REST API service for the Galaxy Automatic License Plate Recognition (ALPR) system. This backend powers real-time vehicle and license plate detection, integrates with AI/ML models for image analysis, manages detection data in a database, and serves results to a modern frontend dashboard.

🚀 Features

- **Image Upload & Detection**
  - Upload vehicle/plate images for AI-powered detection
  - Returns annotated images, detection metadata, and confidence scores
- **Detection Results & History**
  - Fetch latest detection results
  - Paginated history of all detection events
  - Detailed record retrieval by vehicle or plate ID
- **Plate & Vehicle Management**
  - CRUD operations for vehicle and plate records
  - Region and type classification for plates
  - Blacklist/whitelist management for access control
- **Statistics & Analytics**
  - Real-time statistics for dashboard KPIs
  - Vehicle/plate distribution, detection trends, and region analytics
- **Location & Session Management**
  - Manage entry/exit locations
  - Update and retrieve detection sessions
- **Static & Output File Serving**
  - Download annotated images and detection artifacts

- **Health & Status Endpoints**
  - API and database health checks

🛠️ Tech Stack

- **API Framework:** Python 3.10+, FastAPI, Uvicorn
- **Database:** SQLite, SQLAlchemy ORM
- **AI/ML Integration:** YOLOv8/YOLOv11 (PyTorch), OCR (Gemini)
- **HTTP Client:** httpx (for internal API calls)
- **Logging:** Python logging module
- **CORS:** FastAPI Middleware

⚡ Setup Guide

1. **Clone the Repository**

```
git clone https://github.com/your-org/galaxy-alpr-backend.git
cd galaxy-alpr-backend/backend-api-endpoint
```

2. **Create and Activate a Virtual Environment**

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

3. **Install Dependencies**

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

4. **Configure Environment Variables**
   - Copy `.env.example` to `.env` and update values as needed.

5. **Run the Development Server**

```
uvicorn app:app --reload
```

The API will be available at http://localhost:8000.

🔑 Environment Variables

Create a `.env` file in the project root with the following variables:

```
GEMINI_API_KEY=your-gemini-api-key
```

- `GEMINI_API_KEY` : API key for Gemini OCR

📚 API Documentation

Interactive Swagger UI is available at:
http://localhost:8000/docs

Example Endpoints

- **POST** `/detect_image`
  Upload an image for detection.
  Input: multipart/form-data (`file`)
  Output: JSON with detection results, image URLs

- **GET** `/latest_detection`
  Fetch the most recent detection result.

- **GET** `/detections?limit=10&offset=0`
  Paginated list of detection events.

- **GET** `/api/vehicles/{vehicle_id}`
  Get detailed info for a specific vehicle.

- **GET** `/api/plates/{plate_id}`
  Get detailed info for a specific plate.

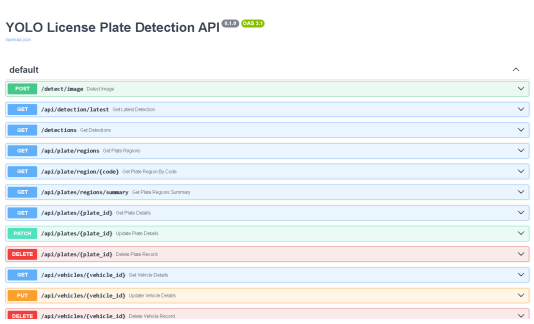
- **POST** `/api/plate-status`
  Add a plate to whitelist/blacklist.

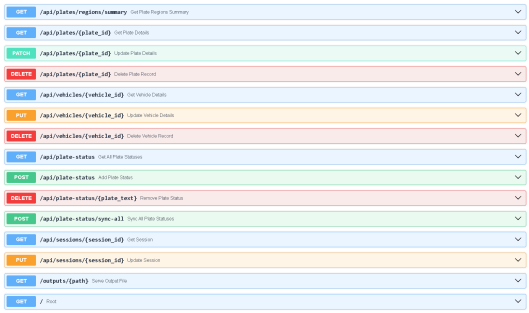
- **GET** `/api/statistics/dashboard`
  Get dashboard statistics for frontend.

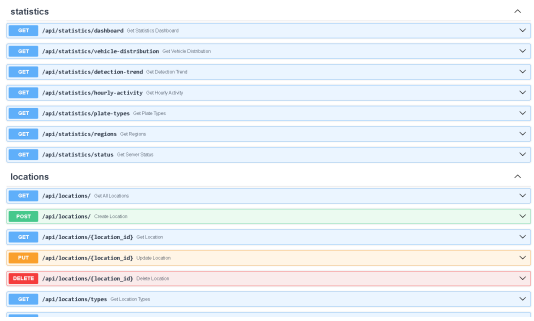
See `/docs` or `/redoc` for the full OpenAPI schema.

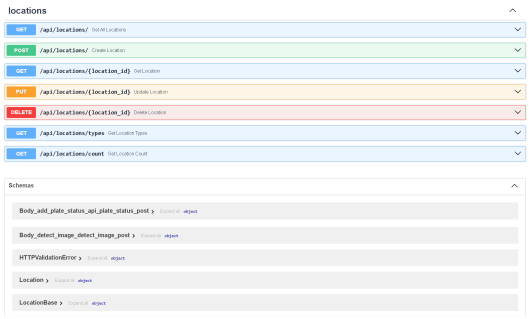
🖼️ API Documentation Preview

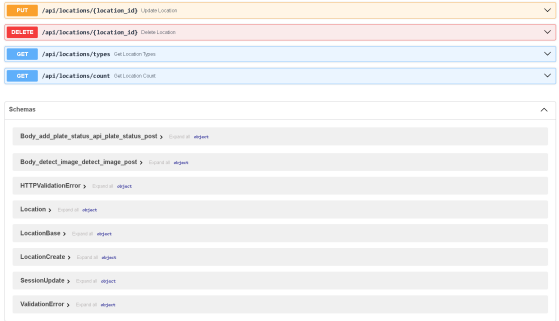
Below is a screenshot of the interactive Swagger UI, which provides a convenient way to explore and test all available API endpoints:











> The Swagger UI is accessible at http://localhost:8000/docs after starting the server.

For the latest endpoints and schemas, always refer to the live Swagger docs.

📂 Folder Structure

```
backend-api-endpoint/
│
├── app.py                  # Main FastAPI application and entry point
├── endpoints/              # Modular API endpoint definitions
│   ├── detect_image.py
│   ├── latest_detection.py
│   ├── detections.py
│   ├── plate_regions.py
│   ├── plate_queries.py
│   ├── vehicle_queries.py
│   ├── statistics.py
│   ├── plate_status.py
│   ├── session_queries.py
│   └── location_routes.py
├── PlateDetector.py        # AI/ML logic for plate and vehicle detection
├── database.py             # Database connection and ORM logic
├── requirements.txt        # Python dependencies
├── outputs/                # Output images, crops, and detection artifacts
├── uploads/                # Temporary storage for uploaded images
├── static/                 # Static files served by FastAPI
├── .env                    # Environment variables
└── README.md               # This documentation
```

📡 API Overview

🔑 Key Endpoints

Detection & Image Processing

Method	Endpoint	Description
POST	`/detect/image`	Upload an image for vehicle/plate detection
GET	`/api/detection/latest`	Fetch the most recent detection result
GET	`/detections`	List all detection events

Method	Endpoint	Description
GET	`/outputs/{path}`	Download detection result artifacts

Vehicle Management

Method	Endpoint	Description
GET	`/api/vehicles/{vehicle_id}`	Get vehicle details by ID
PUT	`/api/vehicles/{vehicle_id}`	Update vehicle details
DELETE	`/api/vehicles/{vehicle_id}`	Delete a vehicle and related plates

Plate Management

Method	Endpoint	Description
GET	`/api/plates/{plate_id}`	Get plate details by ID
PATCH	`/api/plates/{plate_id}`	Update plate details
DELETE	`/api/plates/{plate_id}`	Delete a plate and cascade if needed
GET	`/api/plate/regions`	Get all plate region mappings
GET	`/api/plate/region/{code}`	Get region name for a code

Plate Status (Whitelist/Blacklist)

Method	Endpoint	Description
GET	`/api/plate-status`	List all plate statuses
POST	`/api/plate-status`	Add plate to whitelist/blacklist
DELETE	`/api/plate-status/{plate_text}`	Remove plate from status list
POST	`/api/plate-status/sync-all`	Sync all plate statuses

Statistics & Analytics

Method	Endpoint	Description
GET	`/api/statistics/dashboard`	Get dashboard statistics
GET	`/api/statistics/vehicle-distribution`	Vehicle distribution data
GET	`/api/statistics/detection-trend`	Detection trend data
GET	`/api/statistics/hourly-activity`	Hourly activity data
GET	`/api/statistics/plate-types`	Plate type distribution
GET	`/api/statistics/regions`	Plate region distribution
GET	`/api/statistics/status`	API/database health

Location Management

Method	Endpoint	Description
GET	`/api/locations/`	List all locations
POST	`/api/locations/`	Create a new location
PUT	`/api/locations/{location_id}`	Update a location
DELETE	`/api/locations/{location_id}`	Delete a location

Static & Utility

Method	Endpoint	Description
GET	`/static/{path}`	Download static files
GET	`/`	API root/health check

Example Inputs & Outputs

POST `/detect/image`

Input:
Form-data:

- `file` : (image file, required)

- `location` : (string, optional)

Example Request (curl):

```
curl -F "file=@car.jpg" -F "location=Gate 1" http://localhost:8000/detect/image
```

Example Output:


```
{
  "timestamp": "2025-06-07_12-34-56",
  "processing_time": 1.234,
  "stored_original_path": "outputs/uploaded/2025-06-07_12-34-56_uploaded-image.j
  "detected_vehicle_image_path": "outputs/vehicles/2025-06-07_12-34-56_vehicle.j
  "detected_plate_image_path": "outputs/plates/2025-06-07_12-34-56_plate.jpg",
  "image_resolution": "1920x1080",
  "session_id": 42,
  "location": "Gate 1",
  "list_vehicle_and_plate_information": [
    {
      "vehicle": {
        "vehicle_class": "car",
        "vehicle_confidence_score": 0.98,
        "vehicle_image_path": "outputs/vehicles/crops/abc123.jpg"
      },
      "plate": {
        "plate_text": "B123ABC",
        "ocr_confidence": 0.97,
        "plate_image_path": "outputs/plates/crops/xyz456.jpg"
      }
    }
  ]
}
```


GET `/api/detection/latest`

Output:


```
[
  {
    "id": "42_v1_p1",
    "entity_type": "vehicle_with_plate",
    "vehicle_index": 1,
```

```
    "plate_index": 1,
    "plateNumber": "B123ABC",
    "confidence": 0.97,
    "plateType": "Regular",
    "plateStatus": "whitelist",
    "vehicleType": "car",
    "vehicleConfidence": 0.98,
    "detectedTime": "2025-06-07 12:34:56",
    "gateLocation": "Gate 1 (Entry)",
    "originalImage": "/outputs/uploaded/2025-06-07_12-34-56_uploaded-image.jpg",
    "plateImage": "/outputs/plates/crops/xyz456.jpg",
    "processingTime": "1.234s",
    "imageResolution": "1920x1080",
    "plateTextColor": "Black",
    "plateBackgroundColor": "White",
    "plateRegion": "B (Jakarta)",
    "algorithm": "YOLOv11"
  }
]
```


GET `/detections`

Query Parameters:

- `limit` (int, default 100)
- `offset` (int, default 0)
- `vehicleType` (string, optional)
- `plateType` (string, optional)
- `search` (string, optional)

Output:


```
{
  "detections": [
    {
      "id": 123,
      "detection_type": "plate",
      "timestamp": "2025-06-07 12:34:56",
      "original_path": "/outputs/uploaded/2025-06-07_12-34-56_uploaded-image.jpg
      "file_path": "/outputs/plates/crops/xyz456.jpg",
      "annotated_path": "/outputs/plates/2025-06-07_12-34-56_plate.jpg",
      "processing_time": "1.23ms",
      "image_resolution": "1920x1080",
      "detection_details": {
        "vehicle_type": "car",
```

```
        "vehicle_confidence": 0.98,
        "plate_text": "B123ABC",
        "ocr_confidence": 0.97,
        "confidence": 0.97,
        "region": "B (Jakarta)",
        "plate_status": "whitelist"
      }
    }
  ],
  "count": 1
}
```

GET `/api/vehicles/{vehicle_id}`

Output:


```
{
  "id": 1,
  "vehicle_class": "car",
  "vehicle_confidence_score": 0.98,
  "vehicle_image_path": "outputs/vehicles/crops/abc123.jpg",
  "vehicle_bounding_box": [100, 200, 300, 400],
  "timestamp": "2025-06-07 12:34:56",
  "plates": [
    {
      "id": 10,
      "plate_text": "B123ABC",
      "plate_bounding_box": [120, 220, 180, 260]
    }
  ]
}
```

PUT `/api/vehicles/{vehicle_id}`

Input:


```
{
  "vehicle_class": "truck",
  "vehicle_confidence_score": 0.95
}
```


Output:

```
{
  "message": "Successfully updated vehicle 1",
  "vehicle": {
    "id": 1,
    "vehicle_class": "truck",
    "vehicle_confidence_score": 0.95,
    ...
  }
}
```

DELETE `/api/vehicles/{vehicle_id}`

Output:


```
{
  "message": "Successfully deleted vehicle 1 with 1 plate records",
  "vehicle_deleted": true,
  "plates_deleted": 1,
  "session_deleted": false
}
```

GET `/api/plates/{plate_id}`

Output:


```
{
  "id": 10,
  "plate_text": "B123ABC",
  "plate_bounding_box": [120, 220, 180, 260],
  "plate_type": "Regular",
  "plate_status": "whitelist",
  "plate_region": "B (Jakarta)",
  "vehicle_class": "car",
  "timestamp": "2025-06-07 12:34:56"
}
```

PATCH `/api/plates/{plate_id}`

Input:

```
{
  "plate_text": "B456XYZ",
  "plate_status": "blacklist"
}
```

Output:

```
{
  "message": "Successfully updated plate 10",
  "plate": {
    "id": 10,
    "plate_text": "B456XYZ",
    "plate_status": "blacklist",
    ...
  }
}
```

DELETE `/api/plates/{plate_id}`

Output:

```
{
  "message": "Successfully deleted plate 10 and its associated vehicle",
  "plate_deleted": true,
  "vehicle_deleted": true,
  "session_deleted": false
}
```

GET `/api/plate/regions`

Output:

```
{
  "regions": [
    {"code": "B", "name": "Jakarta"},
    {"code": "D", "name": "Bandung"}
  ]
}
```

GET `/api/plate/region/{code}`

Output:


```
{
  "code": "B",
  "name": "Jakarta"
}
```

GET `/api/statistics/dashboard`

Output:


```
{
  "totalDetections": 1234,
  "vehicleDistribution": {"car": 900, "truck": 200, "motorcycle": 134},
  "detectionTrend": [
    {"date": "2025-06-01", "count": 100},
    {"date": "2025-06-02", "count": 120}
  ],
  "hourlyActivity": [
    {"hour": "08:00", "count": 50},
    {"hour": "09:00", "count": 70}
  ]
}
```

GET `/api/statistics/status`

Output:


```
{
  "status": "online",
  "timestamp": "2025-06-07T12:34:56.789123",
  "database": {
    "accessible": true,
    "path": "detections.db",
    "available_methods": ["get_statistics_dashboard_data", ...],
    "missing_methods": []
  },
  "router": {
    "prefix": "/api/statistics",
    "tags": ["statistics"],
```

```
    "routes": [
      {"path": "/api/statistics/dashboard", "methods": ["GET"]},
      ...
    ]
  }
}
```

GET `/api/plate-status`

Output:

```
{
  "status": "success",
  "plate_statuses": [
    {"plate_text": "B123ABC", "status": "whitelist"},
    {"plate_text": "D456XYZ", "status": "blacklist"}
  ]
}
```

POST `/api/plate-status`

Input:

```
{
  "plate_text": "B123ABC",
  "status_type": "whitelist"
}
```

Output:

```
{
  "status": "success",
  "message": "Plate B123ABC added to whitelist"
}
```

DELETE `/api/plate-status/{plate_text}`

Output:

```
{
  "status": "success",
  "message": "Plate B123ABC removed from status list"
}
```

POST `/api/plate-status/sync-all`

Output:

```
{
  "status": "success",
  "message": "Successfully synchronized 100 plate statuses",
  "details": {
    "whitelist": 60,
    "blacklist": 30,
    "unclassified": 10
  }
}
```

GET `/api/locations/`

Output:

```
[
  {"id": 1, "name": "Gate 1", "type": "Entry", "timestamp": "2025-06-07 12:00:00
  {"id": 2, "name": "Gate 2", "type": "Exit", "timestamp": "2025-06-07 12:10:00"
]
```

POST `/api/locations/`

Input:

```
{
  "name": "Gate 3",
  "type": "Entry"
}
```

Output:

```
{
  "id": 3,
  "name": "Gate 3",
  "type": "Entry",
  "timestamp": "2025-06-07 13:00:00"
}
```

PUT `/api/locations/{location_id}`

Input:

```
{
  "name": "Gate 1A",
  "type": "Entry"
}
```

Output:

```
{
  "id": 1,
  "name": "Gate 1A",
  "type": "Entry",
  "timestamp": "2025-06-07 12:00:00"
}
```

DELETE `/api/locations/{location_id}`

Output:

```
{
  "message": "Location 1 deleted successfully"
}
```

GET `/outputs/{path}`

Output:

Returns the requested file as a download (image or artifact).

If not found, returns:

```
{
  "detail": "File not found: {path}"
}
```

GET `/static/{path}`

Output:
Returns the requested static file.

For more endpoints and details, see the live Swagger UI at http://localhost:8000/docs.

📄 License

This backend system is developed and maintained by **@GalaxyDeveloper**.

🏷️ Citation

If you use **Galaxy ALPR REST API** in your research, academic paper, or production system, please cite:

```
Galaxy ALPR Backend - Modular Backend for AI-Powered License Plate Recognition
Developed by @GalaxyDeveloper (2025)
Includes FastAPI, YOLOv11n, OCR, and SQLite Integration
```

Galaxy ALPR REST API – Modular, scalable backend for intelligent vehicle and plate detection. *Powered by FastAPI, YOLOv11n, OCR, and SQLite*

Developed by @GalaxyDeveloper — 2025

GalaxyALPR

Automatic License Plate Recognition

Front-End Documentation

OPENTHEGATE HACK
BSS PARKING
06-25

Galaxy ALPR System - Frontend Documentation

Table of Contents

Project Overview

The Galaxy ALPR System is a comprehensive **Automatic License Plate Recognition (ALPR)** application designed specifically for Indonesian vehicles. This modern web application provides real-time vehicle detection, license plate recognition, and parking management capabilities through an intuitive dashboard interface.

Purpose

- **Vehicle Detection**: Automatic recognition of cars and motorcycles entering/exiting parking facilities
- **License Plate Recognition**: High-accuracy OCR processing for Indonesian license plates with advanced entity detection
- **Traffic Management**: Real-time monitoring and analytics for parking operations
- **Data Analytics**: Comprehensive reporting and statistical analysis with interactive charts
- **Configuration Management**: Advanced detection area setup and camera integration
- **Access Control**: Blacklist/Whitelist management for enhanced security

Tech Stack

- **Frontend Framework**: React 19.1.0 with TypeScript 4.9.5
- **Styling**: Tailwind CSS 3.3.0 (core utility classes only)
- **State Management**: React Hooks (`useState` , `useEffect` , `useRef` , `useCallback`)
- **Routing**: React Router DOM 7.6.1
- **Testing**: React Testing Library & Jest DOM
- **Charts & Analytics**: Recharts 2.15.3
- **Icons**: Lucide React v0.511.0:
  - Lucide React 0.511.0
  - Phosphor React 1.4.1
- **Build Tool**: `react-scripts` (Create React App)
- **Package Manager**: npm or yarn

Core Architecture

Application Structure

The application follows a modular architecture with three main sections accessible via tabs:

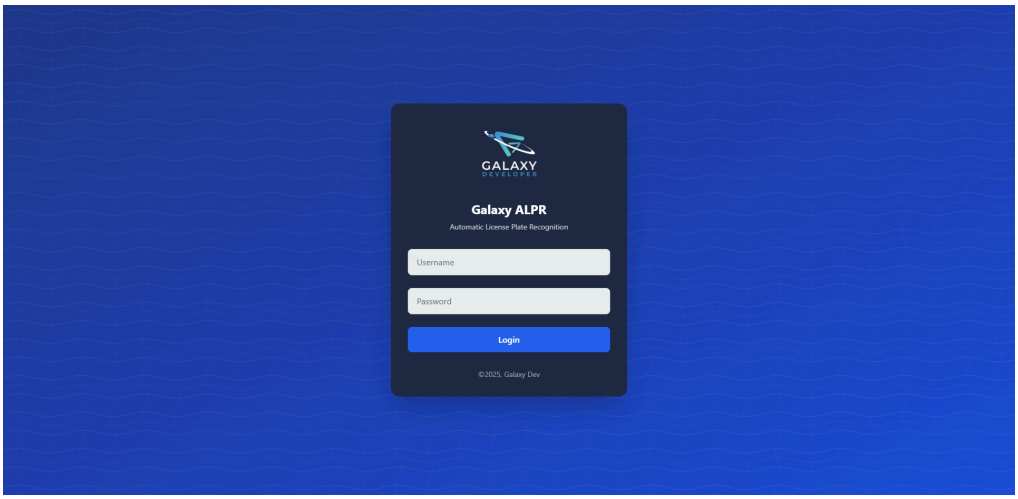
1. **Detection Area Setup** - Camera configuration and detection zone definition
2. **Access List Management** - Blacklist/Whitelist plate management
3. **Location Management** - Gate and location configuration

Component Hierarchy

```
App
├── Routes
│   ├── LoginPage (galaxy-alpr-login.tsx)
│   └── ProtectedLayout (wrapper for authenticated routes)
│       ├── Background (Global background component)
│       ├── Sidebar (Navigation)
│       ├── Header (Top navigation bar)
│       ├── Main Content Pages
│       │   ├── HomePage (galaxy-alpr-dashboard.tsx)
│       │   │   ├── ImageTab
│       │   │   └── WebcamTab
│       │   ├── ResultsPage (galaxy-alpr-results.tsx)
│       │   │   ├── ImageComparisonTab
│       │   │   └── DetectionDetailsTab
│       │   ├── StatisticsPage (galaxy-alpr-statistics.tsx)
│       │   ├── HistoryPage (galaxy-alpr-history.tsx)
│       │   ├── UserManagementPage (galaxy-alpr-users.tsx)
│       │   ├── DetectionConfigPage (galaxy-alpr-config.tsx)
│       │   │   ├── DetectionAreaTab
│       │   │   ├── AccessListTab
│       │   │   └── LocationManagementTab
│       │   └── DocumentationPage (galaxy-alpr-documentation.tsx)
│       └── Footer
```

Page Components

1. Login Page (`galaxy-alpr-login.tsx`)



The Login Page serves as the authentication gateway to the Galaxy ALPR System, featuring a modern design with advanced visual effects and secure form handling.

Form Management

State Structure:

```
const [formData, setFormData] = useState({
  username: "",
  password: "",
});
```

Input Handling:

```
const handleInputChange = (e: React.ChangeEvent<HTMLInputElement>) => {
  const { name, value } = e.target;
  setFormData((prev) => ({
    ...prev,
    [name]: value,
  }));
};
```

Authentication Flow

Current Implementation:

- Form validation is currently commented out for development ease
- Direct navigation to dashboard upon submission
- Console logging for debugging purposes

Prepared Validation Logic:

```
if (!username || !username.endsWith("@gmail.com")) {
  errors.push("Username must end with '@gmail.com'.");
}
if (!password || password.length < 6) {
  errors.push("Password must be at least 6 characters long.");
}
```

Navigation Integration:

```
const handleSubmit = () => {
  const { username, password } = formData;
  const errors: string[] = [];

  if (errors.length > 0) {
    alert(errors.join("\n"));
    return;
  }

  console.log("Login attempt:", formData);
  navigate("/dashboard");
};
```

Security Considerations

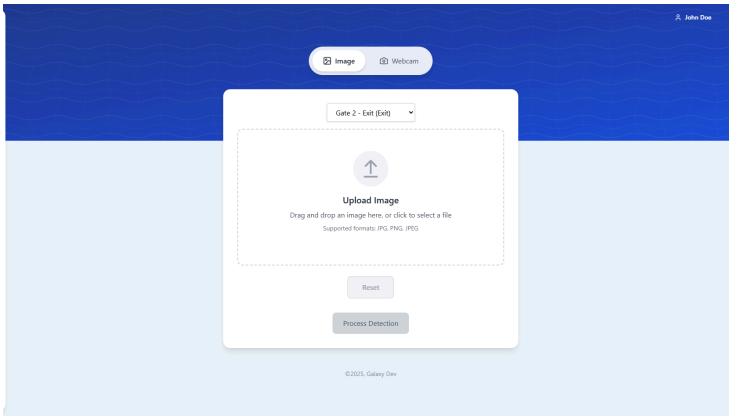
Development vs Production:

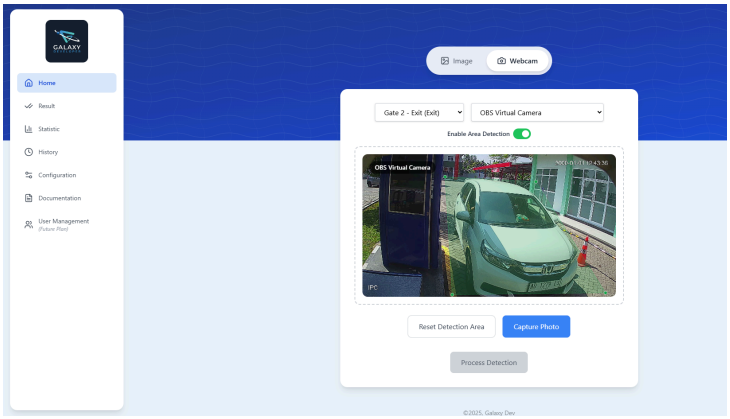
- Validation logic prepared but disabled for development
- Console logging for debugging (should be removed in production)
- Form data structure ready for secure transmission

Future Enhancements:

- JWT token integration
- Remember me functionality
- Password strength validation
- Account lockout protection
- Two-factor authentication support

2. Home Page (`galaxy-alpr-dashboard.tsx`)





The Home Page serves as the entry point for ALPR operations, allowing users to perform vehicle and plate recognition through image uploads or real-time webcam feeds.

Core Features

- **Tab Navigation:**
- **Image Upload Tab**: Users can upload images directly for detection.
- **Webcam Tab**: Enables live camera feed for real-time recognition.
- **Green Box Cropping**: Webcam captures are cropped to a defined polygon (green box) for focused detection.
- **Image Preprocessing**: Captured or uploaded images are converted to base64 and sent to the backend API.
- **Detection Workflow**:
  - Image sent via POST to `/detect/image`
  - Backend returns annotated vehicle and plate images
  - Success message with navigation to results or history
- **Camera Device Selection**: Detects all available cameras using `navigator.mediaDevices`.

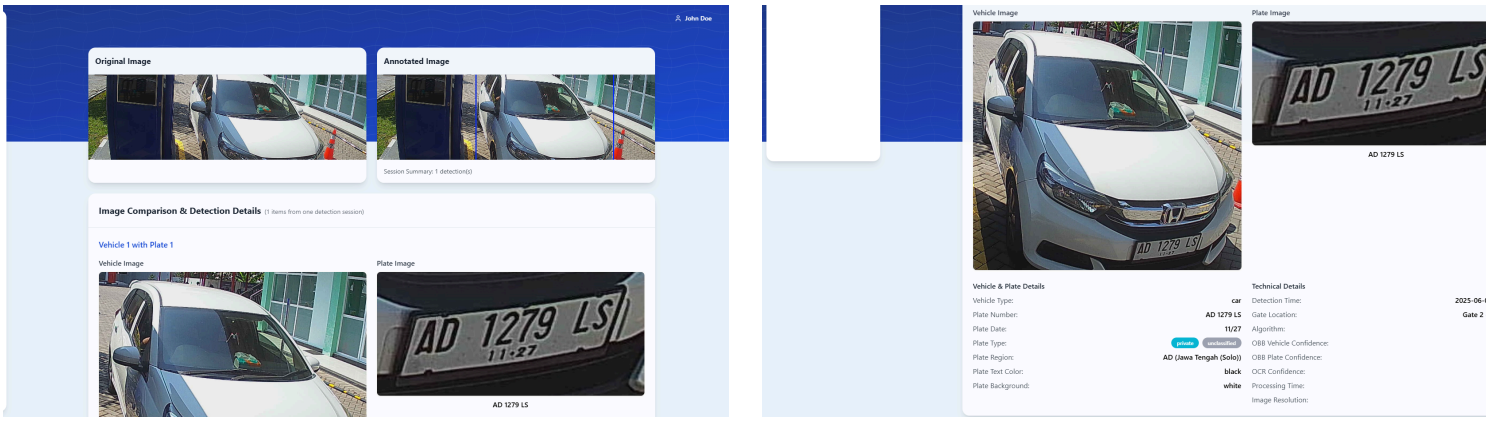
Webcam Cropping

```
function cropImageToGreenBox(video: HTMLVideoElement): Promise<string> {
  const width = video.videoWidth;
  const height = video.videoHeight;
  const points = GREEN_BOX_POINTS.map((p) => ({
    x: p.x * width,
    y: p.y * height,
  }));

  const minX = Math.min(...points.map((p) => p.x));
  const maxX = Math.max(...points.map((p) => p.x));
  const minY = Math.min(...points.map((p) => p.y));
  const maxY = Math.max(...points.map((p) => p.y));

  const canvas = document.createElement("canvas");
  canvas.width = maxX - minX;
  canvas.height = maxY - minY;
  const ctx = canvas.getContext("2d")!;

  ctx.clip();
  ctx.drawImage(
    video,
    minX,
    minY,
    canvas.width,
    canvas.height,
    0,
    0,
    canvas.width,
    canvas.height
  );
  return Promise.resolve(canvas.toDataURL("image/png"));
}
```

Detection API Integration

```
const handleProcessDetection = async () => {
  if (!uploadedImage) return;
  const blob = await (await fetch(uploadedImage)).blob();
  const file = new File([blob], "image.png", { type: "image/png" });

  const formData = new FormData();
  formData.append("file", file);

  const response = await fetch("http://localhost:8000/detect/image", {
    method: "POST",
    body: formData,
  });

  const result = await response.json();
  setProcessResult(result);
};
```

3. Results Page (`galaxy-alpr-results.tsx`)



Advanced detection results display with comprehensive analysis and image comparison.

Enhanced Detection Data Structure

```
interface DetectionResult {
  id: string;
  entity_type?:
    | "vehicle_with_plate"
    | "vehicle_without_plate"
    | "standalone_plate"
    | "no_detection";
  vehicle_index?: number;
  plate_index?: number;
  plateNumber: string;
  confidence: number;
  plateType: "Military" | "Regular";
  isBlacklist: "Blacklist" | "Whitelist";
  vehicleType: string;
  detectedTime: string;
  gateLocation: string;

  // Enhanced image paths
  originalImage: string;
  plateImage: string;
  storedOriginalPath: string;
  vehicleAnnotatedPath: string;
  vehicleImagePath: string;
  plateImagePath: string;

  // Technical metadata
  processingTime: string;
  imageResolution: string;
  plateTextColor: string;
  plateBackgroundColor: string;
  plateRegion: string;
  algorithm: string;
}
```

Advanced Features

Entity Type Detection:

- **Vehicle with Plate**: Complete detection with both vehicle and readable plate
- **Vehicle without Plate**: Vehicle detected but no readable plate
- **Standalone Plate**: Plate detected without associated vehicle
- **No Detection**: No entities found

Dynamic Image Loading:

```
// Smart image URL handling with backend integration
const getImageUrl = (path: string): string => {
  if (!path) return "/api/placeholder/300/200";
  if (path.startsWith("http")) return path;
  if (path.startsWith("/")) return `${backendUrl}${path}`;
  return `${backendUrl}/${path}`;
};
```

Error Handling and Loading States:

- Comprehensive error handling for API failures
- CORS-specific error detection and messaging
- Loading spinners and fallback states
- Graceful image loading with error fallbacks

Tabbed Analysis Interface

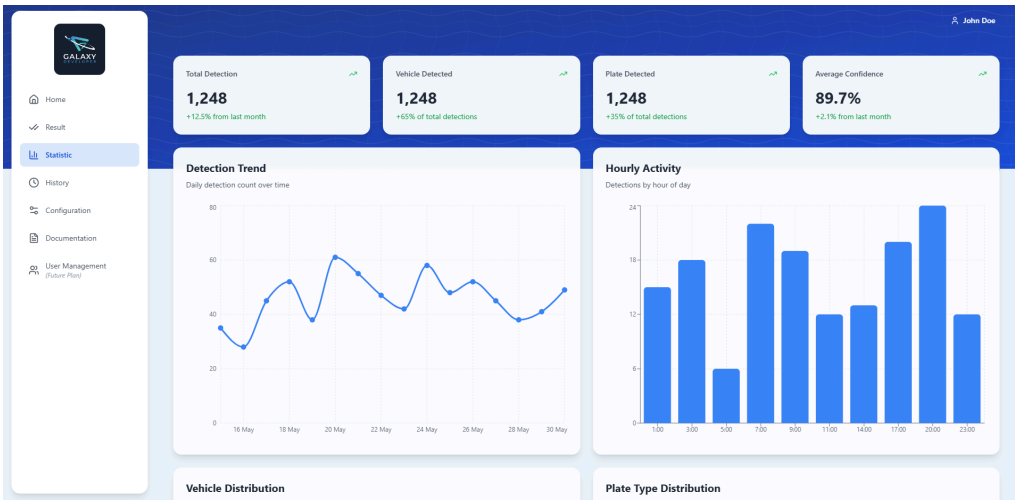
Image Comparison Tab:

- Side-by-side original and annotated image display
- Entity-specific image grids based on detection type
- Dynamic layout adaptation for different entity types

Detection Details Tab:

- Comprehensive metadata display
- Technical details including confidence scores
- Color-coded plate type indicators
- Processing time and resolution information

4. Statistics Page (`galaxy-alpr-statistics.tsx`)



Comprehensive analytics dashboard with interactive charts and KPI cards.

Statistics Architecture

```
// KPI Cards with trend indicators
interface StatCard {
  title: string;
  value: string;
  change: string;
  trend: "up" | "down";
}

// Chart data structures
const detectionTrendData = [
  { date: "15 May", detections: 35 },
  // ... time series data
];

const vehicleDistributionData = [
  { name: "Cars", value: 812, color: "#3B82F6" },
  { name: "Motorcycles", value: 436, color: "#10B981" },
];
```

Chart Components Integration

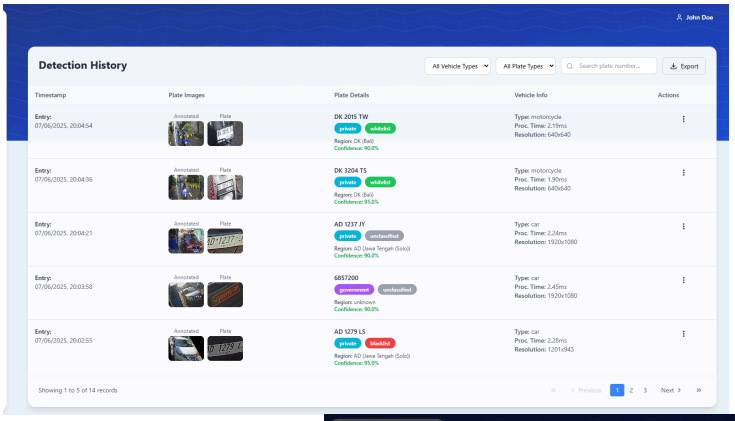
Custom Chart Components:

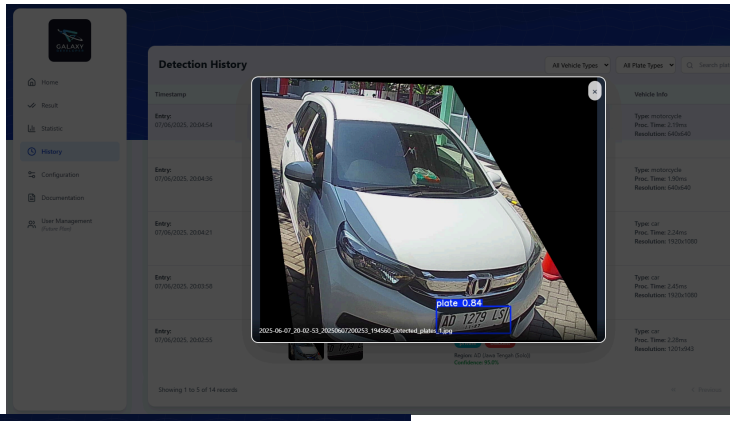
- `CustomPieChart` - For distribution analysis
- `CustomBarChart` - For categorical data
- `CustomLineChart` - For trend analysis
- `ChartCard` - Wrapper component with consistent styling

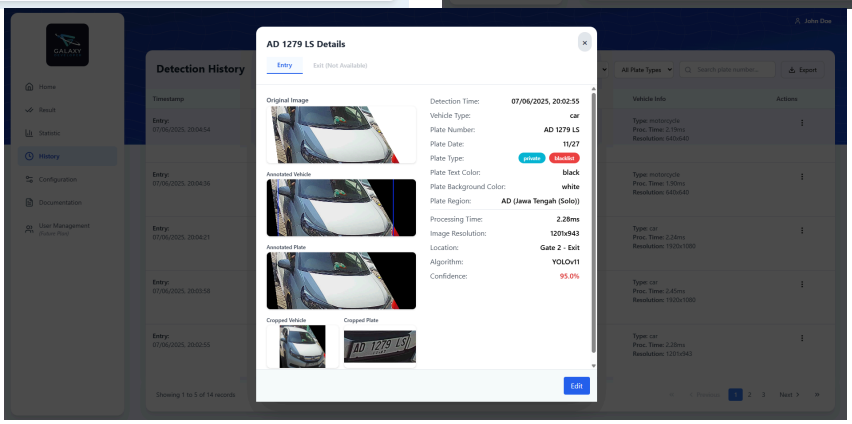
Chart Types:

- **Detection Trend** (Line Chart): Daily detection patterns over time
- **Hourly Activity** (Bar Chart): 24-hour traffic distribution
- **Vehicle Distribution** (Pie Chart): Vehicle type breakdown
- **Plate Type Distribution** (Pie Chart): Plate type classification
- **Blacklist/Whitelist Distribution** (Pie Chart): Access classification
- **Region Distribution** (Pie Chart): Plate region distribution

5. History Page (`galaxy-alpr-history.tsx`)







The History Page displays all previously processed detections in a paginated, filterable table.

Functional Highlights

- **Record Retrieval**:
  - Fetches paginated data from `/detections` API
  - Supports filters by vehicle type and plate type
  - Full-text search on plate number
- **Pagination**: Page control with navigation to first/last and numeric page buttons
- **Data Export**: Export current view to CSV format
- **Image Previews**:
  - Annotated entry/exit image
  - Cropped plate image
- **Metadata Display**:
  - Plate Number, Confidence %, Plate Type, Region, Vehicle Type
- **Modals**:
  - **Delete Confirmation**: Prompt before deleting record
  - **Edit Detail Modal**: Interface to edit a record
  - **Image Preview Modal**: Fullscreen annotated or plate image

Fetching Paginated Records

```
const fetchDetectionRecords = async (page = 1) => {
  const offset = (page - 1) * recordsPerPage;
  let url = `http://localhost:8000/detections?limit=${recordsPerPage}&offset=${offset}`;

  if (vehicleTypeFilter !== "all") url += `&vehicleType=${vehicleTypeFilter}`;
  if (plateTypeFilter !== "all") url += `&plateType=${plateTypeFilter}`;
  if (searchQuery.trim()) url += `&search=${encodeURIComponent(searchQuery)}`;

  const response = await fetch(url);
  const data = await response.json();
  setDetectionRecords(data.detections);
  setTotalEntries(data.count || 0);
};
```

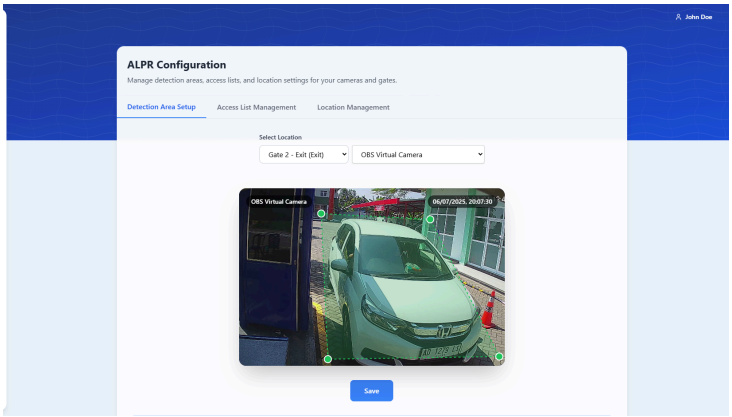
Image Fallback Logic

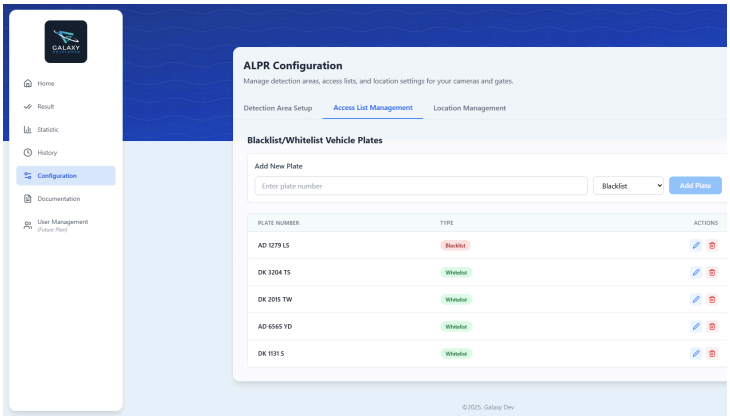
```
const handleImageError = (
  e: React.SyntheticEvent<HTMLImageElement, Event>,
  type: string = ""
) => {
  const img = e.currentTarget;
  console.error(`Image failed [${type}]:`, img.src);
  img.src = "https://via.placeholder.com/80x60?text=No+Image";
  img.alt = "Image not found";
  img.onerror = null;
};
```

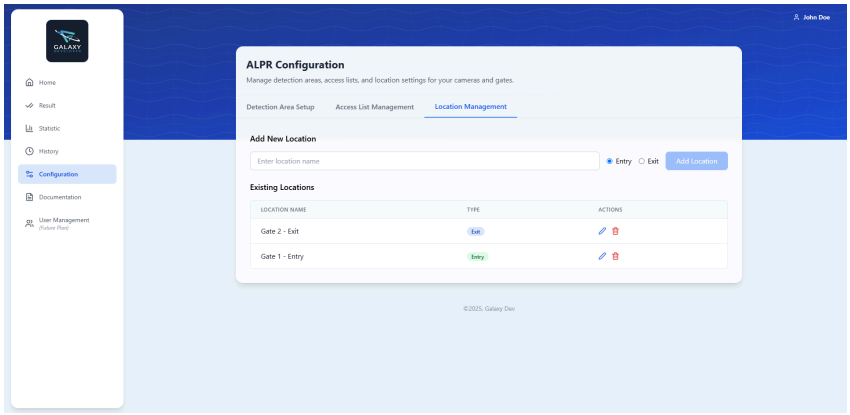
Export to CSV

```
const exportToCSV = () => {
  const headers = [
    "ID",
    "Entry Time",
    "Exit Time",
    "Plate",
    "Vehicle",
    "Region",
    "Type",
    "Confidence",
  ];
  const rows = detectionRecords.map((r) => [
    r.id,
    r.timestampEntry,
    r.timestampExit,
    r.plateNumber,
    r.vehicleType,
    r.region,
    r.plateType,
    r.confidence.toFixed(1),
  ]);

  const csv = [headers.join(","), ...rows.map((row) => row.join(","))].join(
    "\n"
  );
  const blob = new Blob([csv], { type: "text/csv" });
  const url = URL.createObjectURL(blob);
  const a = document.createElement("a");
  a.href = url;
  a.download = `detection-history-${new Date().toISOString().slice(0, 10)}.csv`;
  a.click();
};
```

6. Detection Configuration Page (`galaxy-alpr-config.tsx`)







The main configuration interface with tabbed navigation for system setup.

State Management

```
// Location and tab management
const [selectedParkingPark, setSelectedParkingPark] =
  useState("Parking Park 1");
const [activeTab, setActiveTab] = useState<"area" | "list" | "location">(
  "area"
);

// Access list management
const [plates, setPlates] = useState<
  { plate: string; type: "blacklist" | "whitelist" }[]
>([]);
const [newPlate, setNewPlate] = useState("");
const [newType, setNewType] = useState<"blacklist" | "whitelist">("blacklist");

// Camera and location setup
const [selectedLocation, setSelectedLocation] = useState<string>(
  LOCATION_OPTIONS[0]
);
const [cameraDevices, setCameraDevices] = useState<MediaDeviceInfo[]>([]);
const [selectedCameraId, setSelectedCameraId] = useState<string>("");

// Detection area polygon points
const [areaPoints, setAreaPoints] = useState<Point[]>([
  { x: 150, y: 300 },
  { x: 450, y: 280 },
  { x: 500, y: 350 },
  { x: 100, y: 370 },
]);
```

Key Features

Detection Area Setup:

- Interactive polygon drawing on camera feed
- Drag-and-drop point manipulation
- Real-time camera integration via `navigator.mediaDevices`

- Automatic coordinate scaling for different resolutions
- Save functionality with actual camera resolution mapping

Access List Management:

- Dynamic blacklist/whitelist plate management
- Add/remove plate numbers with type classification
- Real-time list updates

Location Management:

- Dynamic location creation and deletion
- Gate location selection (Gate 1-4 Entrance/Exit)
- Location-based configuration persistence

Advanced Camera Integration

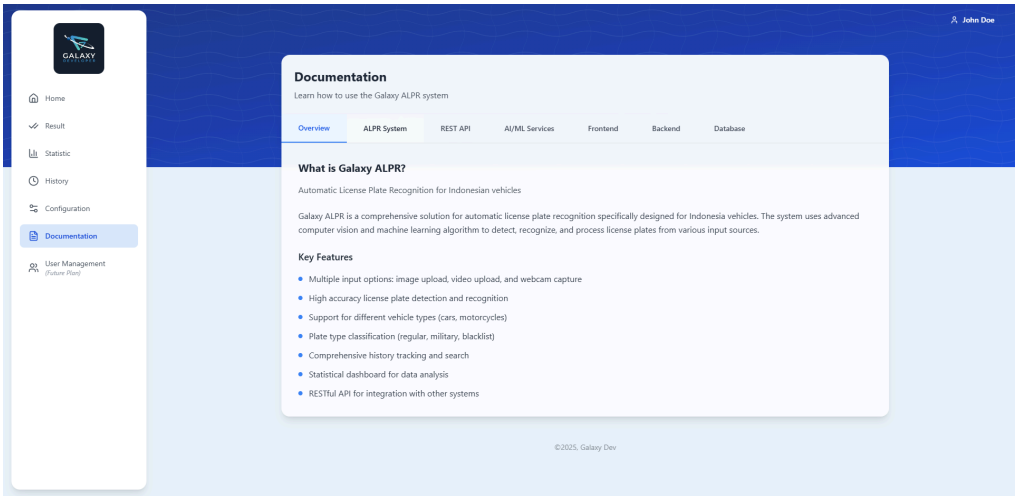
```
// Camera stream management with cleanup
useEffect(() => {
  let stream: MediaStream;

  if (activeTab === "area" && selectedCameraId) {
    navigator.mediaDevices
      .getUserMedia({
        video: { deviceId: { exact: selectedCameraId } },
      })
      .then((mediaStream) => {
        stream = mediaStream;
        streamRef.current = mediaStream;
        if (videoRef.current) {
          videoRef.current.srcObject = mediaStream;
        }
      })
      .catch((err) => {
        if (videoRef.current) {
          videoRef.current.srcObject = null;
        }
      });
  }

  // Cleanup on tab change or unmount
  return () => {
    if (stream) {
      stream.getTracks().forEach((track) => track.stop());
    }
  };
}, [selectedCameraId, activeTab]);
```

Interactive Detection Area

The detection area setup uses advanced pointer event handling for smooth polygon manipulation:

```
// Stable event handlers using useCallback
const handleWindowPointerMove = useCallback((e: PointerEvent) => {
  if (draggingIdxRef.current === null) return;
  const svg = document.getElementById("detection-area-svg");
  if (!svg) return;
  const rect = svg.getBoundingClientRect();
  const x = ((e.clientX - rect.left) / rect.width) * 600;
  const y = ((e.clientY - rect.top) / rect.height) * 400;

  setAreaPoints((pts) =>
    pts.map((pt, i) =>
      i === draggingIdxRef.current
        ? {
            x: Math.max(0, Math.min(600, x)),
            y: Math.max(0, Math.min(400, y)),
          }
        : pt
    )
  );
}, []);
```

7. Documentation Page (`galaxy-alpr-documentation.tsx`)



The Documentation page serves as an interactive guide for users and developers to understand and work with the Galaxy ALPR system. It includes a clean tabbed interface to switch between documentation categories, such as ALPR overview, API usage, AI/ML details, and integration steps.

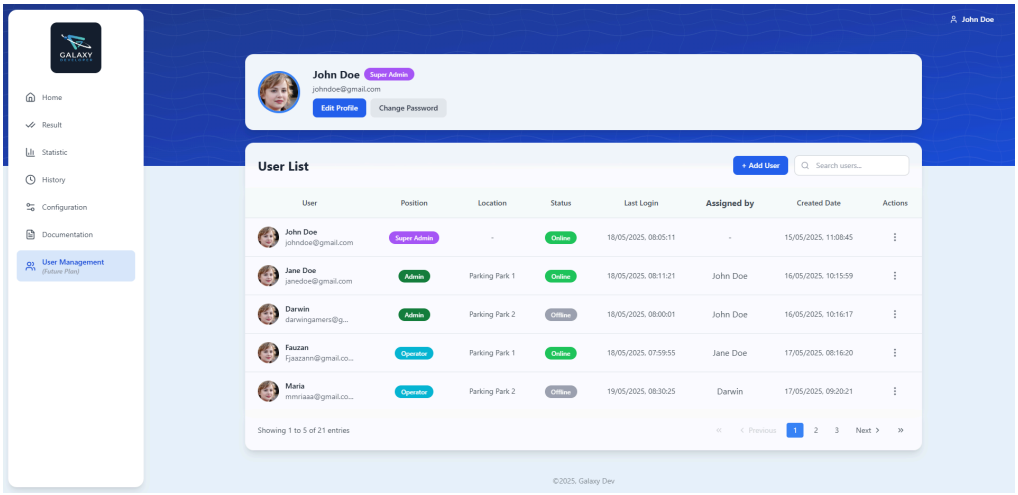
Documentation Content:

The documentation is organized into 7 main tabs that guide users through every aspect of the Galaxy ALPR system:

- **Overview**
  Introduction to Galaxy ALPR, its purpose, key features, and supported vehicle/plate types.
- **ALPR System**
  Explanation of how the detection pipeline works, including image preprocessing, YOLO-based detection, OCR recognition, and result classification.

- **REST API**
  Describes available endpoints for detection, history retrieval, access list management, and camera configuration — including request/response examples.
- **AI/ML Services**
  Outlines the computer vision and machine learning models used (e.g., YOLOv8, OCR), model input/output formats, and tuning parameters.
- **Frontend**
  Highlights the client-side structure, core React components, state management, routing, and user interaction flow.
- **Backend**
  Describes the server-side architecture, FastAPI endpoints, authentication strategy, and response structure.
- **Database**
  Provides schema overview, key relationships (e.g., detections ↔ vehicles ↔ plates), indexing strategy, and retention policy.

8. User Management Page (`galaxy-alpr-users.tsx`)



Admin-only interface for managing users and viewing operator statistics.

Features:

- **User Role Segmentation**:
  - Super Admin: Full access
  - Admin: Limited to managing Operators
  - Operator: View only
- **Live Status View**: Online/Offline with color-coded badges
- **Search**: Filter users by name or email
- **Pagination**: Per-page user management
- **Profile Card**:
  - Editable avatar and name
  - Read-only fields for email and role
- **User Actions** (Admin/Super Admin only):
  - Add, Edit, Delete users via dropdown menu
- **Color-Coding**:
  - Super Admin: Purple
  - Admin: Green

- Operator: Cyan

User Search, Sort, and Paginate

```
const filteredUsers = allUsers
  .filter(
    (user) =>
      user.name.toLowerCase().includes(search.toLowerCase()) ||
      user.email.toLowerCase().includes(search.toLowerCase())
  )
  .sort((a, b) => positionOrder[a.position] - positionOrder[b.position]);

const pagedUsers = filteredUsers.slice(
  (currentPage - 1) * recordsPerPage,
  currentPage * recordsPerPage
);
```

Dynamic Badge Styling

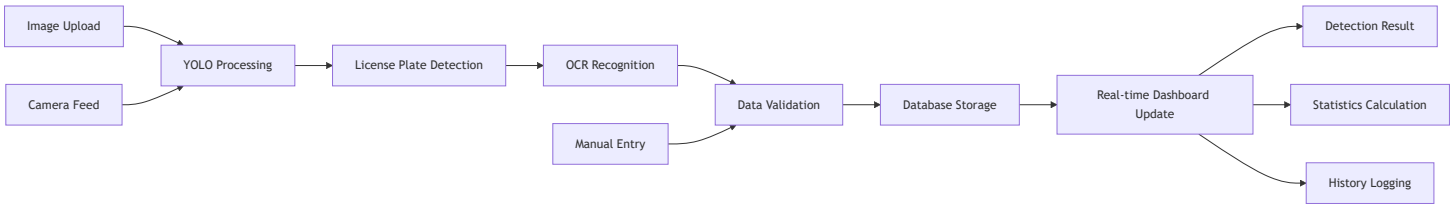
```
const getPositionColor = (position: string) => {
  switch (position) {
    case "Super Admin":
      return "bg-purple-500 text-white";
    case "Admin":
      return "bg-green-700 text-white";
    case "Operator":
      return "bg-cyan-500 text-white";
    default:
      return "bg-gray-500 text-white";
  }
};

const getStatusColor = (status: string) => {
  return status === "Online"
    ? "bg-green-500 text-white"
    : "bg-gray-400 text-white";
};
```

Profile Picture Update

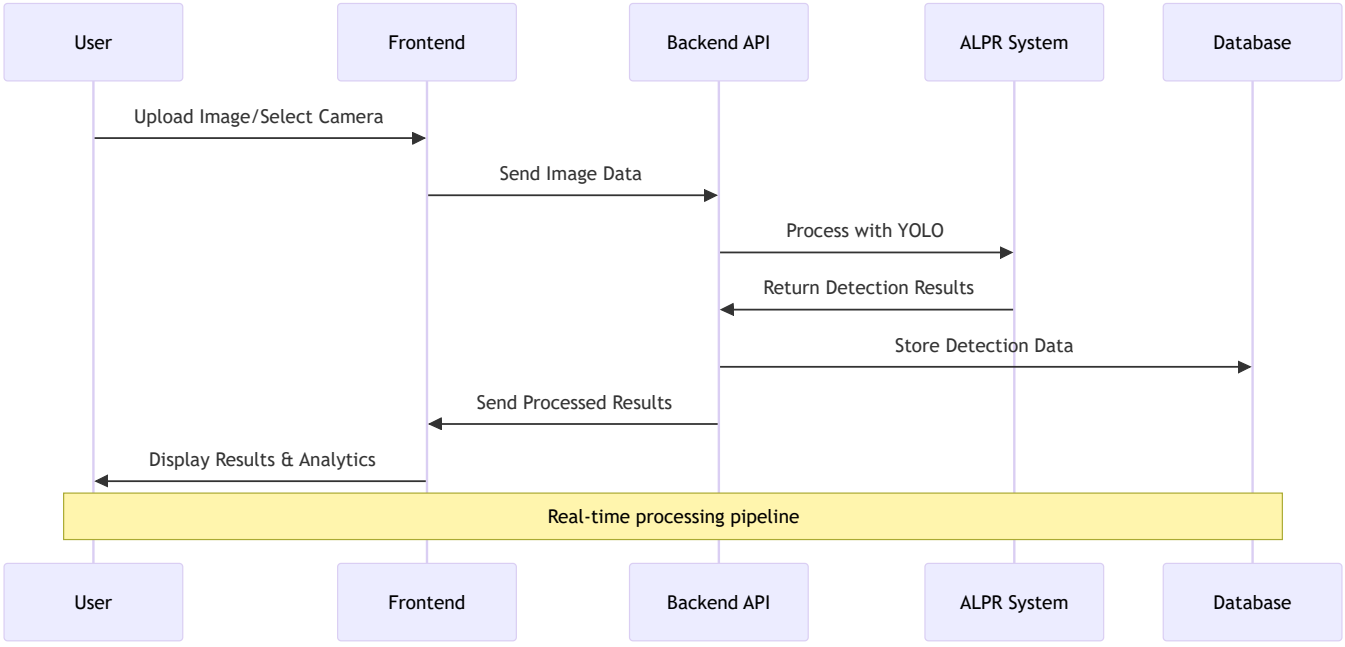
```
const handleProfilePicChange = (e: React.ChangeEvent<HTMLInputElement>) => {
  if (e.target.files?.[0]) {
    const reader = new FileReader();
    reader.onload = (ev) => {
      setProfilePic(ev.target?.result as string);
    };
    reader.readAsDataURL(e.target.files[0]);
  }
};
```

State Management Patterns

Data Flow Patterns



User Interaction Flow



API Integration

Backend Communication

```
// API base configuration
const backendUrl = "http://localhost:8000";

// Latest detection data fetching
const fetchLatestDetection = async () => {
  try {
    const response = await fetch(
      `${backendUrl}/api/detection/latest?_=${new Date().getTime()}`,
      {
        method: "GET",
        headers: {
          Accept: "application/json",
        },
      }
    );

    if (!response.ok) {
      throw new Error(
        `API returned status: ${response.status} ${response.statusText}`
      );
    }

    const data = await response.json();
    return data;
  } catch (error) {
    // Handle CORS and network errors
    console.error("API Error:", error);
    throw error;
  }
};
```

Error Handling Strategy

CORS Detection:

```
if (err instanceof TypeError && err.message.includes("fetch")) {
  setError(
    "CORS error: Cannot connect to the backend server. Make sure the backend is running and CORS is configured pro
  );
}
```

Graceful Degradation:

- Fallback images for failed loads
- Error states with retry mechanisms

- Loading states for better UX

Performance Optimizations

Component Optimization

Stable References:

```
// Use refs for values that don't need re-renders
const draggingIdxRef = useRef<number | null>(null);

// Memoized event handlers
const handlePointerDown = useCallback(
  (idx: number) => (e: React.PointerEvent) => {
    setDraggingIdx(idx);
    // ... handler logic
  },
  []
);
```

Resource Management:

```
// Camera stream cleanup
useEffect(() => {
  return () => {
    if (streamRef.current) {
      streamRef.current.getTracks().forEach((track) => track.stop());
      streamRef.current = null;
    }
  };
}, [selectedCameraId, activeTab]);
```

Data Processing

Efficient State Updates:

```
// Batch state updates for better performance
setAreaPoints((pts) =>
  pts.map((pt, i) =>
    i === draggingIdxRef.current
      ? { x: Math.max(0, Math.min(600, x)), y: Math.max(0, Math.min(400, y)) }
      : pt
  )
);
```

Development Setup

Prerequisites

- Node.js 18+ and npm/yarn
- Modern web browser with camera support
- Backend API server running on localhost:8000

Installation Steps

1. **Clone and Install**

```
git clone https://github.com/your-org/bss-parking-system.git
cd bss-parking-system
npm install
```

2. **Environment Configuration**

```
REACT_APP_API_BASE_URL=http://localhost:8000
REACT_APP_UPLOAD_MAX_SIZE=10MB
REACT_APP_SUPPORTED_FORMATS=jpg,jpeg,png
REACT_APP_DETECTION_CONFIDENCE_THRESHOLD=0.8
```

3. **Development Server**

```
npm run dev
```

Project Structure

```
frontend/
├── public/
│   ├── assets/
│   └── ...
├── src/
│   ├── components/
│   │   ├── config/
│   │   │   ├── access-list-tab.tsx
│   │   │   ├── detection-area-tab.tsx
│   │   │   └── location-management-tab.tsx
│   │   ├── dashboard/
│   │   │   └── dashboard-content.tsx
│   │   ├── history/
│   │   │   ├── delete-confirmation-modal.tsx
│   │   │   ├── edit-detail-modal.tsx
│   │   │   └── image-preview-modal.tsx
│   │   ├── statistics/
│   │   │   ├── chart-card.tsx
│   │   │   ├── custom-barchart.tsx
│   │   │   ├── custom-linechart.tsx
│   │   │   └── custom-piechart.tsx
│   │   ├── users/
│   │   │   ├── profile-card.tsx
│   │   │   └── user-list-card.tsx
│   │   ├── background.tsx
│   │   ├── delete-confirmation-modal.tsx
│   │   ├── header.tsx
│   │   └── sidebar.tsx
│   ├── pages/
│   │   ├── galaxy-alpr-config.tsx
│   │   ├── galaxy-alpr-dashboard.tsx
│   │   ├── galaxy-alpr-documentation.tsx
│   │   ├── galaxy-alpr-history.tsx
│   │   ├── galaxy-alpr-login.tsx
│   │   ├── galaxy-alpr-results.tsx
│   │   ├── galaxy-alpr-statistics.tsx
│   │   └── galaxy-alpr-users.tsx
│   ├── utils/
│   │   ├── image-utils.ts
│   ├── App.css
│   ├── App.js
│   └── ...
```

Configuration Options

Detection Area Setup

- **Interactive Polygon Drawing**: Click and drag to define detection zones
- **Coordinate Scaling**: Automatic scaling between preview and actual camera resolution
- **Multiple Area Support**: Define multiple detection zones per camera

Camera Integration

- **Device Enumeration**: Automatic detection of available video input devices
- **Stream Management**: Proper resource cleanup and error handling
- **Resolution Support**: Configurable video resolution settings

Access Control

- **Dynamic Lists**: Real-time blacklist/whitelist management
- **Plate Validation**: Input validation for Indonesian plate formats
- **Bulk Operations**: Support for batch plate management

Deployment Considerations

Production Build

```
npm run build
```

Environment Variables

```
REACT_APP_API_BASE_URL=https://your-api-domain.com
REACT_APP_ENABLE_ANALYTICS=true
REACT_APP_LOG_LEVEL=warn
```

Security Considerations

- HTTPS required for camera access
- CORS configuration for API communication
- Input validation for all user inputs
- Secure handling of detection data

Future Enhancements

Planned Features

- Real-time WebSocket integration

- Advanced analytics with AI insights
- Multi-language support
- Advanced reporting exports

Technical Improvements

- Service Worker for offline functionality
- Advanced caching strategies
- Progressive Web App features
- Enhanced accessibility compliance

License

System Version: v1.0.0
React Version: 19.1.0
Last Updated: June 2025
Documentation Version: 1.0

⬆️ Back to Top ⬆️

GalaxyALPR

Automatic License Plate Recognition

Back-End Documentation

OPENTHEGATE HACK
BSS PARKING
06-25

Galaxy ALPR Backend

A comprehensive backend platform for **Automatic License Plate Recognition (ALPR)**, enabling real-time vehicle and plate detection using AI/ML, robust database integration, and seamless communication with a modern frontend. This system is designed for smart parking management, security, and analytics in enterprise and public environments.

📖 Table of Contents

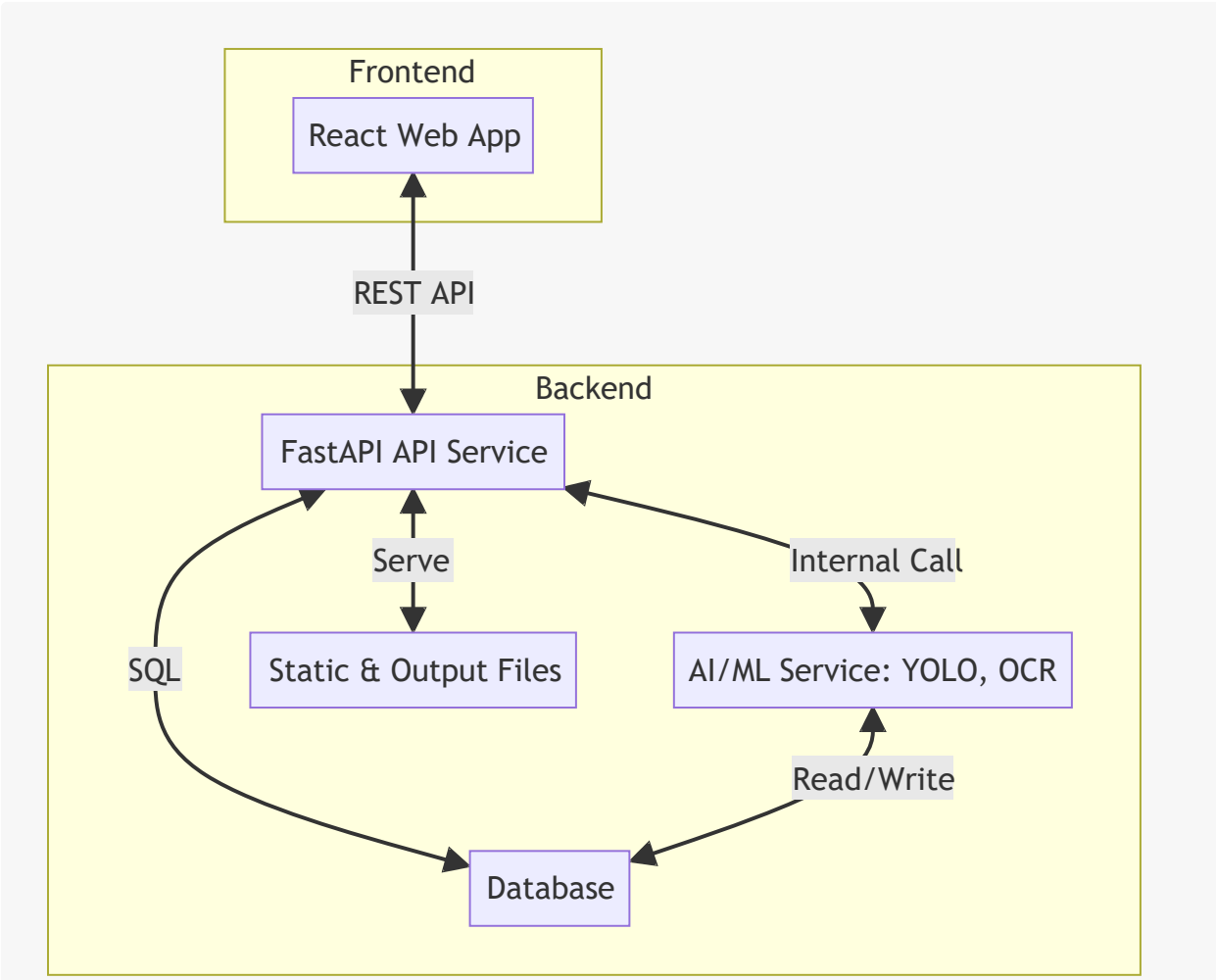
📝 Project Overview

Galaxy ALPR Backend provides:

- **RESTful API** for vehicle and license plate detection, querying, and statistics.
- **AI/ML Service** for high-accuracy detection using YOLO and OCR models.
- **Database Integration** for storing detection events, user sessions, and analytics.
- **Static & Output File Serving** for detected images and results.
- **Frontend Communication** for real-time dashboards and management.

This backend is designed to be modular, scalable, and easy to extend for new detection types, analytics, or integrations.

🏗️ Architecture



- **Frontend**: Sends HTTP requests to the API for detection, history, and analytics.
- **API Service**: Handles requests, triggers AI/ML inference, and manages data.
- **AI/ML Service**: Performs detection and recognition, returns structured results.
- **Database**: Stores all detection events, user data, and statistics.
- **Static/Output Files**: Served via FastAPI for detected images and artifacts.

📂 Folder Structure

```
backend/
│
├── app.py                # Main FastAPI application and entry point
├── endpoints/            # Modular API endpoint definitions
│   ├── detect_image.py
│   ├── latest_detection.py
```

```
│   ├── detections.py
│   ├── plate_regions.py
│   ├── plate_queries.py
│   ├── vehicle_queries.py
│   ├── statistics.py
│   ├── plate_status.py
│   ├── session_queries.py
│   └── location_routes.py
│
├── PlateDetector.py       # AI/ML logic for plate and vehicle detection
├── database.py            # Database connection and ORM logic
├── requirements.txt       # Python dependencies
│
├── outputs/               # Output images, crops, and detection artifacts
│   ├── vehicles/
│   ├── plates/
│   ├── uploaded/
│   └── ...
├── uploads/               # Temporary storage for uploaded images
├── static/                # Static assets served by the API
│
└── README.md              # This documentation
```

📂 Folder/Service Breakdown

File / Folder	Description
`app.py`	Main FastAPI app, server entry point, and configuration.
`endpoints/`	All API route definitions, organized by feature.
`PlateDetector.py`	AI/ML detection logic (YOLO, OCR, OpenCV, etc.).
`database.py`	Database models, schema, and connection logic.
`outputs/`	Stores detection results, vehicle/plate crops, and uploaded files.
`uploads/`	Temporary storage for user-uploaded images.
`static/`	Static files (docs, UI assets, etc.) served by FastAPI.
`requirements.txt`	Python dependencies for backend and AI/ML.

🛠️ Tech Stack

🔷 API Service

- Python 3.10+
- FastAPI (web framework)
- Uvicorn (ASGI server)
- CORS Middleware

🔷 AI/ML Service

- YOLOv8/YOLOv11 (PyTorch)
- OCR (Tesseract, Gemini, or custom)
- OpenCV, NumPy

🔷 Database

- SQLite (default, easy dev setup, offline)
- SQLAlchemy (ORM)

🔷 Frontend

- React + TypeScript (see `frontend/README.md`)

🔷 Utilities

- Logging via Python `logging` module
- File serving via FastAPI

⚡ Setup & Installation

1. Clone the Repository

```
git clone https://github.com/your-org/galaxy-alpr-backend.git
cd galaxy-alpr-backend
```

2. Python Environment

Create and activate a virtual environment:

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

Install dependencies:

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

3. Directory Preparation

The backend auto-creates `outputs/`, `uploads/`, and `static/` on startup. Create manually if needed:

```
mkdir -p outputs/ uploads/ static/
```

4. Run the API Service

```
uvicorn app:app --reload
```

API will be available at: http://localhost:8000

5. Configuration

- **CORS**: Default allows `localhost:5173`, `localhost:3000`, and all origins (`*`). Change in `app.py`.
- **Database**: Uses SQLite by default. For MySQL, edit connection string in `database.py`.

🔗 Service Communication

From	To	Protocol / Method
Frontend	API	RESTful HTTP (CORS)
API	AI/ML	Internal call or REST

From	To	Protocol / Method
API	Database	SQLAlchemy ORM
AI/ML	Database	Direct read/write
API	File Store	FastAPI static routes

📡 API Overview

🔑 Key Endpoints

Method	Endpoint	Description
POST	`/detect_image`	Upload an image for vehicle/plate detection
GET	`/latest_detection`	Fetch the most recent detection result
GET	`/detections`	List all detection events
GET	`/plate_regions`	Get plate regions in a specific image
GET	`/plate_queries`	Query plate detection history
GET	`/vehicle_queries`	Query vehicle detection history
GET	`/statistics`	Retrieve analytics and stats
GET	`/plate_status`	Check status of a specific plate
GET	`/api/locations`	Manage/query location data
GET	`/outputs/{path}`	Download detection result artifacts
GET	`/static/{path}`	Download static files

👨‍💻 Development & Contribution

1. Fork and branch from `main`
2. Follow PEP8 and internal code style
3. Add or modify routes in `endpoints/`

4. Include/update tests if available
5. Use clear commit messages and submit PRs

Code Quality

- Modular endpoint architecture
- Logging for key operations and errors
- Auto-creates required folders on first run

Subfolder Documentation

- `backend/README.md` — API and backend logic
- `backend/endpoints/README.md` — API endpoint documentation
- `backend/PlateDetector.py` — ML model logic and image processing
- `backend/database.py` — Database models and setup

License

This backend system is developed and maintained by **@GalaxyDeveloper**.

Citation

If you use **Galaxy ALPR Backend** in your research, academic paper, or production system, please cite:

```
Galaxy ALPR Backend - Modular Backend for AI-Powered License Plate Recognition
Developed by @GalaxyDeveloper (2025)
Includes FastAPI, YOLOv11n, OCR, and SQLite Integration
```

Galaxy ALPR Backend – Modular, scalable backend for intelligent vehicle and plate detection. *Powered by FastAPI, YOLOv11n, OCR, and SQLite*

Developed by @GalaxyDeveloper — 2025

GalaxyALPR

Automatic License Plate Recognition

Database Documentation

OPENTHEGATE HACK
BSS PARKING
06-25

Galaxy ALPR App – Database

This folder manages the relational database for the Galaxy ALPR (Automatic License Plate Recognition) App. It provides persistent storage for vehicle detections, license plates, session logs, and location metadata.

Project Purpose

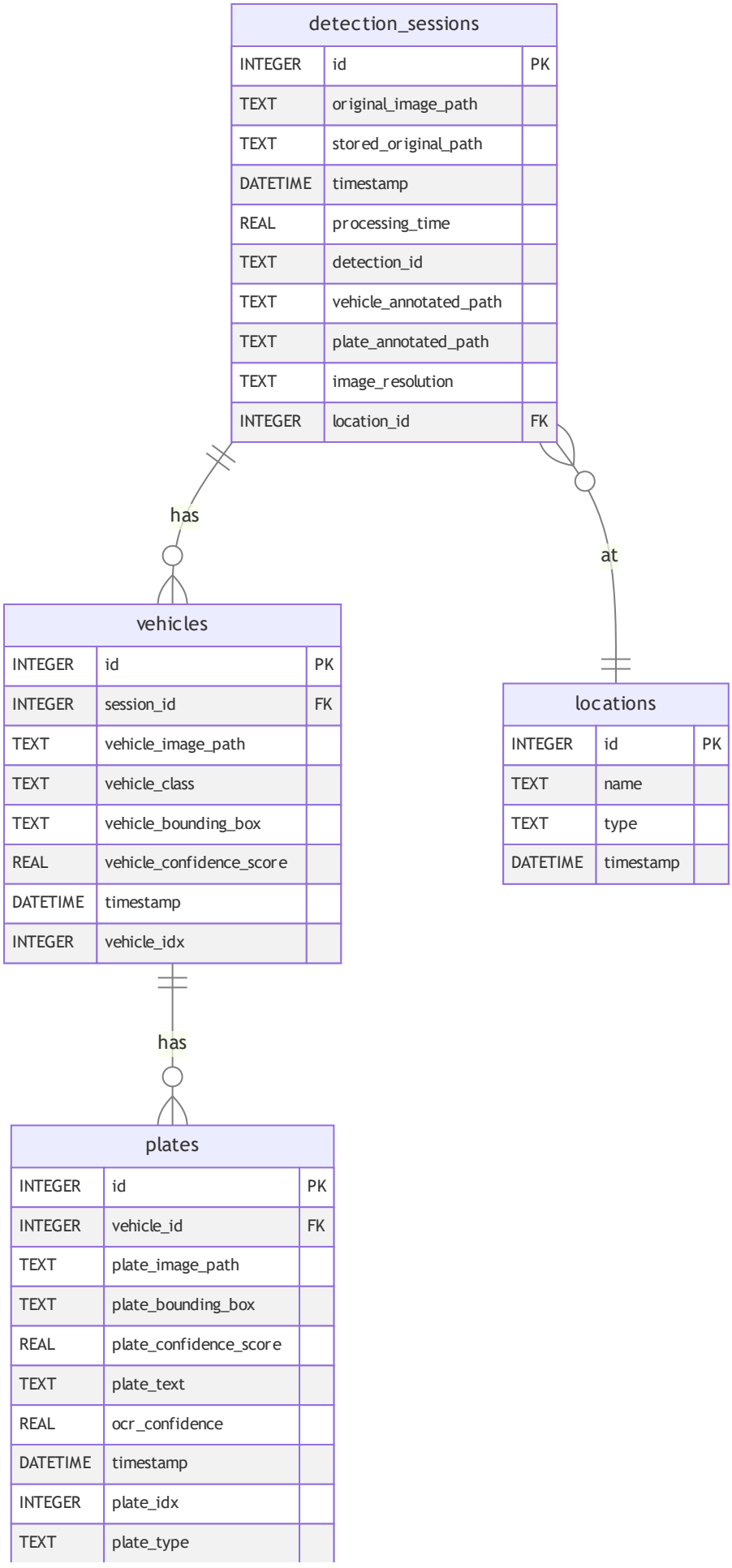
The database is designed to:

- Store all vehicle and license plate detections from the ALPR pipeline.
- Track detection sessions, including images, processing times, and locations.
- Maintain metadata for each vehicle and plate, including region, type, and status (whitelist/blacklist).
- Support analytics, dashboard statistics, and API queries for the backend.

Tech Stack

- **Database:** SQLite (default, file-based, easy for local/dev; can be swapped for PostgreSQL or others)
- **ORM/Access:** Python `sqlite3` module (see `database.py`)
- **Schema Management:** Programmatic migrations in `database.py` (no external migration tool required)
- **Directory:** All schema and DB logic is in the `database/` folder

---


detection_sessions
INTEGER id PK
TEXT original_image_path
TEXT stored_original_path
DATETIME timestamp
REAL processing_time
TEXT detection_id
TEXT vehicle_annotated_path
TEXT plate_annotated_path
TEXT image_resolution
INTEGER location_id FK
has
at
vehicles
INTEGER id PK
INTEGER session_id FK
TEXT vehicle_image_path
TEXT vehicle_class
TEXT vehicle_bounding_box
REAL vehicle_confidence_score
DATETIME timestamp
INTEGER vehicle_idx
locations
INTEGER id PK
TEXT name
TEXT type
DATETIME timestamp
has
plates
INTEGER id PK
INTEGER vehicle_id FK
TEXT plate_image_path
TEXT plate_bounding_box
REAL plate_confidence_score
TEXT plate_text
REAL ocr_confidence
DATETIME timestamp
INTEGER plate_idx
TEXT plate_type

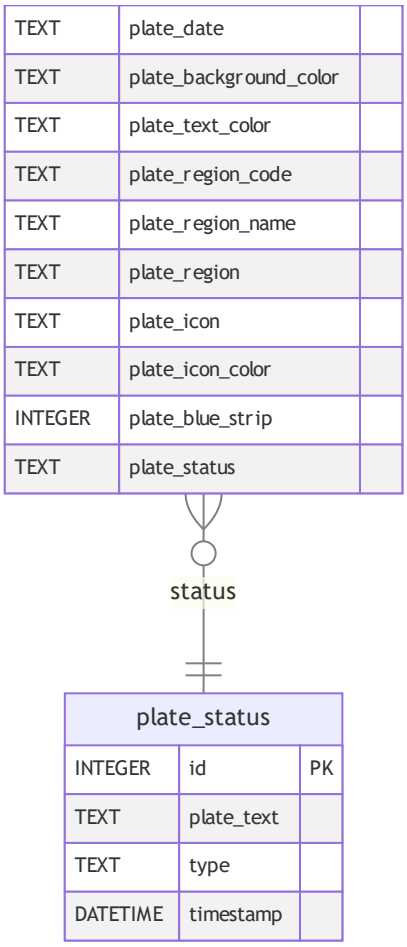


Figure: Entity-Relationship (ER) Diagram

Schema Outline

detection_sessions

- `id` (PK): Unique session ID
- `original_image_path`: Path to uploaded image
- `stored_original_path`: Path to stored copy in outputs
- `timestamp`: Detection time (WITA/GMT+8)
- `processing_time`: Time taken for detection (seconds)
- `detection_id`: Unique string for session
- `vehicle_annotated_path`: Path to annotated vehicle image
- `plate_annotated_path`: Path to annotated plate image
- `image_resolution`: e.g. "1920x1080"
- `location_id` (FK): Reference to `locations.id`

vehicles

- `id` (PK): Unique vehicle ID
- `session_id` (FK): Reference to `detection_sessions.id`
- `vehicle_image_path` : Path to cropped vehicle image
- `vehicle_class` : e.g. "car", "motorcycle"
- `vehicle_bounding_box` : JSON array [x1, y1, x2, y2]
- `vehicle_confidence_score` : Detection confidence (float)
- `timestamp` : Detection time
- `vehicle_idx` : Index within session

plates

- `id` (PK): Unique plate ID
- `vehicle_id` (FK): Reference to `vehicles.id`
- `plate_image_path` : Path to cropped plate image
- `plate_bounding_box` : JSON array [x1, y1, x2, y2]
- `plate_confidence_score` : Detection confidence (float)
- `plate_text` : OCR result
- `ocr_confidence` : OCR confidence (float)
- `timestamp` : Detection time
- `plate_idx` : Index within vehicle
- `plate_type` : e.g. "Regular", "Government"
- `plate_date` : Registration/expiry date (if available)
- `plate_background_color` : e.g. "White", "Yellow"
- `plate_text_color` : e.g. "Black"
- `plate_region_code` : e.g. "B"
- `plate_region_name` : e.g. "Jakarta"
- `plate_region` : Combined display (e.g. "B (Jakarta)")
- `plate_icon` : Icon on plate (if any)
- `plate_icon_color` : Color of icon
- `plate_blue_strip` : 1 if present, else 0
- `plate_status` : "whitelist", "blacklist", or "unclassified"

locations

- `id` (PK): Unique location ID
- `name` : e.g. "Gate 1"

- `type` : "Entry" or "Exit"
- `timestamp` : Creation time

plate_status

- `id` (PK): Unique status ID
- `plate_text` (UQ): Plate text (unique)
- `type` : "whitelist" or "blacklist"
- `timestamp` : Last update

Table Descriptions

- **detection_sessions**: Each detection event; links to original and processed images, processing time, and location.
- **vehicles**: Each detected vehicle in a session; stores class, image, and bounding box.
- **plates**: Each detected license plate; stores cropped image, OCR text, confidence, and rich metadata (type, region, color, etc.).
- **locations**: Named entry/exit points; referenced by sessions.
- **plate_status**: Tracks whitelist/blacklist status for plate texts.

Example Data Flow

1. **Image Upload:** User uploads an image via API.
2. **Detection:** Pipeline detects vehicles and plates, saves crops and metadata.
3. **Session Insert:** New row in `detection_sessions` with image paths, time, location.
4. **Vehicle Insert:** Each detected vehicle is added to `vehicles` (linked to session).
5. **Plate Insert:** Each detected plate is added to `plates` (linked to vehicle), with all OCR and region metadata.
6. **Status Update:** If a plate is whitelisted/blacklisted, `plate_status` is updated and all matching plates are marked.

Database Setup Instructions

1. **Install Python 3.8+** (if not already installed)

2. **No external DB install needed** (uses SQLite by default)
3. **First run:** The schema is auto-created by `database.py` when the backend starts.

Manual DB Management

- The SQLite file is created at `backend/detections.db` (or as configured).
- Use any SQLite browser (e.g., DB Browser for SQLite) for inspection.

SQL Scripts / Migration

- **No separate migration scripts**: All schema creation is handled in `Database.init_db()` in `database.py`.
- To reset the DB, delete the `.db` file and restart the backend.

Usage in API Backend

- All database access is via the `Database` class in `database.py`.
- The backend uses this class for all CRUD operations, statistics, and dashboard queries.
- See `database.py` for method documentation and usage.

How the DB is Used by the API

- **Detection Endpoints:** Insert sessions, vehicles, and plates on each detection.
- **Query Endpoints:** Fetch sessions, vehicles, plates, and statistics for dashboards.
- **Plate Status Endpoints:** Manage whitelist/blacklist and synchronize plate statuses.
- **Location Endpoints:** Manage entry/exit locations for detections.

Local Development

- **Local:** No setup required; DB file is created on first run.
- **Config:** Change the DB path by passing a different `db_path` to the `Database` class.

📄 License

This backend system is developed and maintained by **@GalaxyDeveloper**.

📌 Citation

If you use **Galaxy ALPR Database** in your research, academic paper, or production system, please cite:

```
Galaxy ALPR Database - Modular Backend for AI-Powered License Plate Recognition
Developed by @GalaxyDeveloper (2025)
Includes FastAPI, YOLOv11n, OCR, and SQLite Integration
```

Galaxy ALPR Database – Modular, scalable backend for intelligent vehicle and plate detection. *Powered by FastAPI, YOLOv11n, OCR, and SQLite*

Developed by @GalaxyDeveloper — 2025
