NETRAX AI - System Architecture

Table of Contents

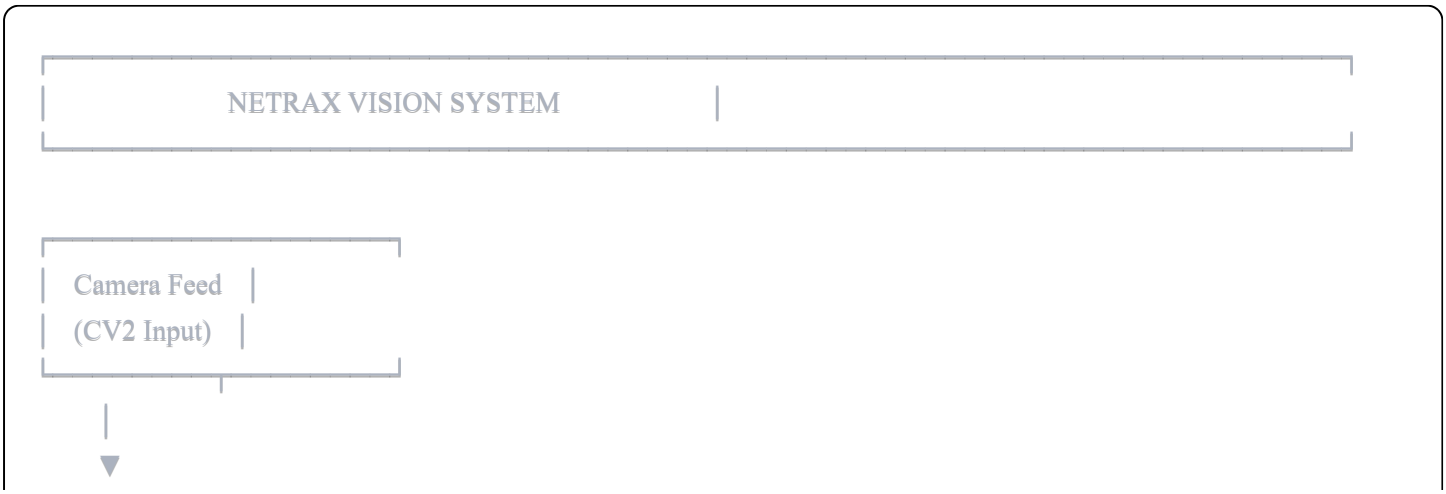
System Overview

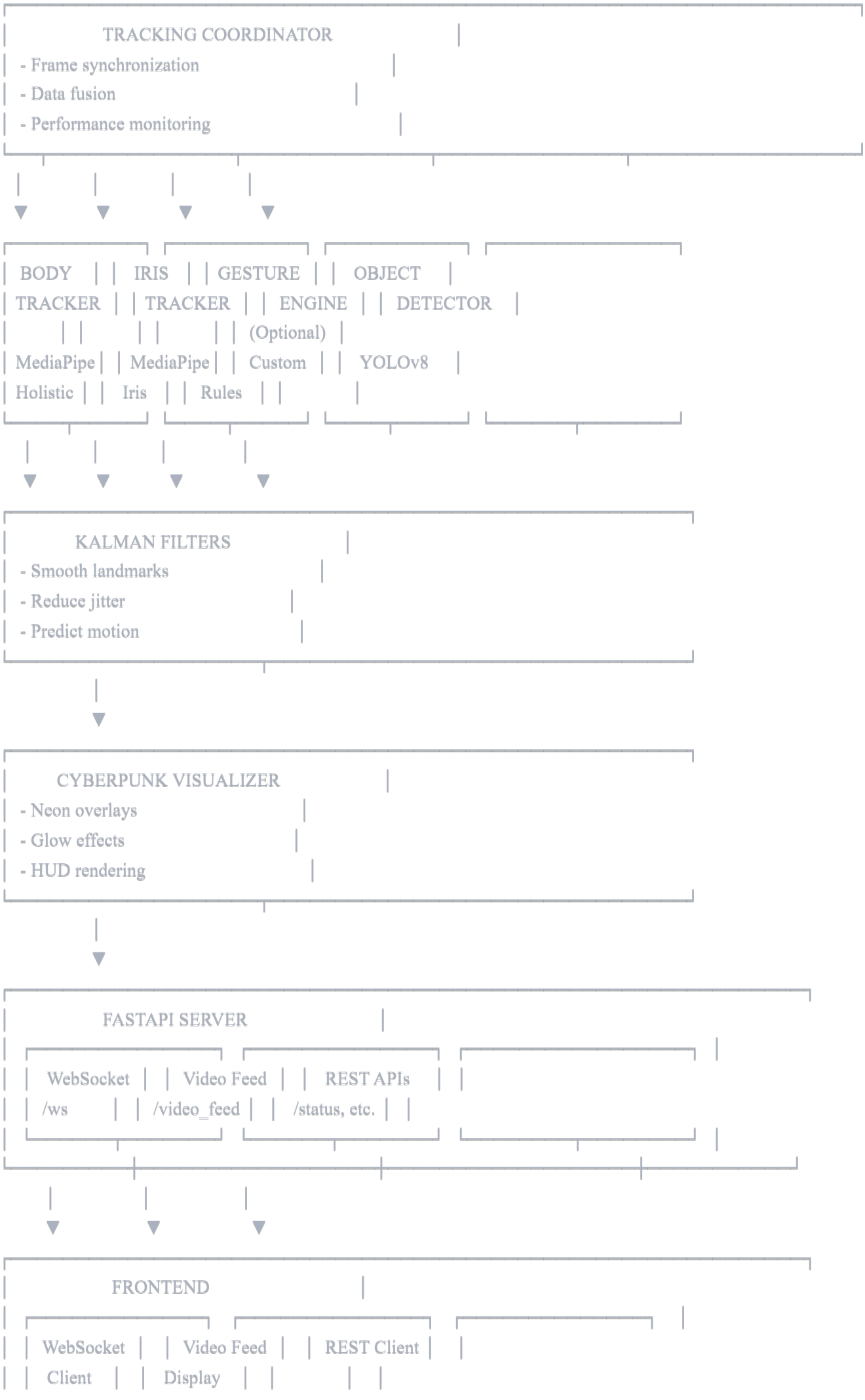
NETRAX is a **modular, production-grade computer vision system** designed for real-time human tracking and interaction. The system follows a **coordinator pattern** where a master orchestrator manages specialized vision engines.

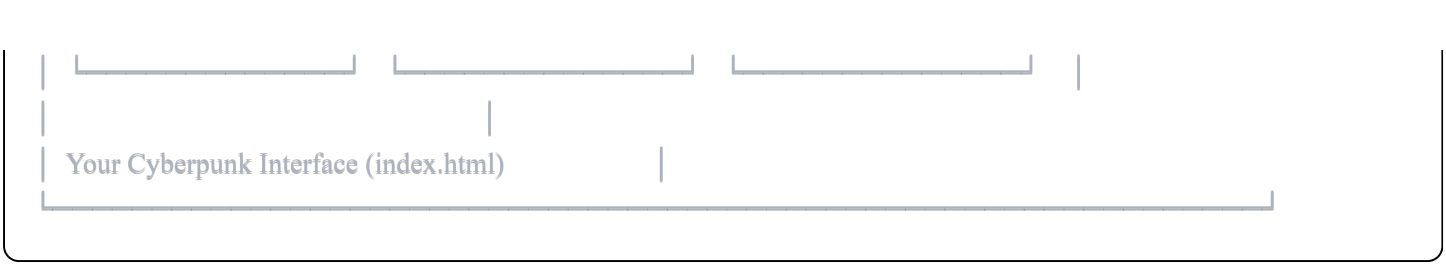
Design Principles

- **Modularity:** Each vision engine is independent and can be enabled/disabled
- **Performance:** Optimized for 60 FPS with GPU acceleration
- **Scalability:** WebSocket architecture supports multiple clients
- **Robustness:** Kalman filtering and error handling throughout
- **Aesthetics:** Cyberpunk visual style with neon overlays

Architecture Diagram



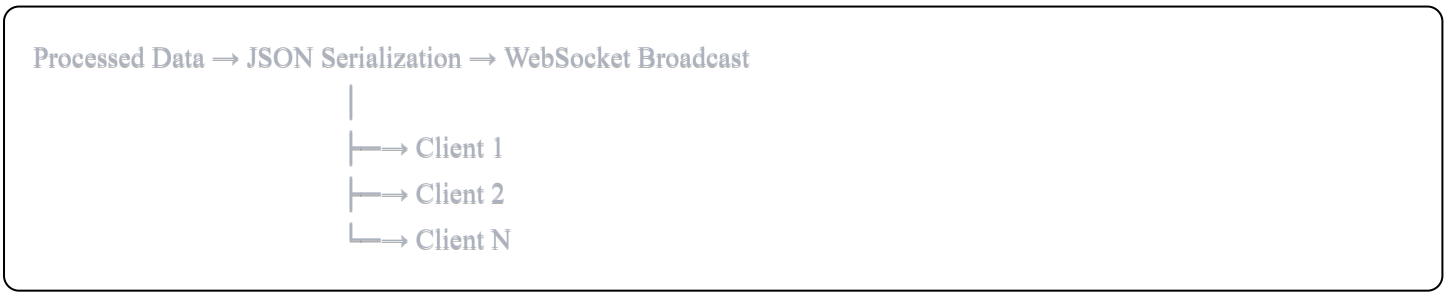

TRACKING COORDINATOR
- Frame synchronization
- Data fusion
- Performance monitoring
BODY TRACKER
MediaPipe Holistic
IRIS TRACKER
MediaPipe Iris
GESTURE ENGINE (Optional)
Custom Rules
OBJECT DETECTOR
YOLOv8
KALMAN FILTERS
- Smooth landmarks
- Reduce jitter
- Predict motion
CYBERPUNK VISUALIZER
- Neon overlays
- Glow effects
- HUD rendering
FASTAPI SERVER
WebSocket /ws
Video Feed /video_feed
REST APIs /status, etc.
FRONTEND
WebSocket Client
Video Feed Display
REST Client



Data Flow

1. Frame Capture → Processing

2. Data Streaming



3. Message Types

Stats Update (every frame):

```
{
 "type": "stats",
 "stats": {
  "fps": 30.5,
  "gesture_count": 15,
  "confidence": 0.95
 }
}
```

Gesture Command (on detection):

```
json

{
 "type": "gesture_command",
 "command": "peace",
 "confidence": 0.92
}
```

Detailed Tracking (optional):

```
json

{
 "type": "tracking_detail",
 "data": {
  "body": {...},
  "eyes": {...}
 }
}
```

Module Breakdown

🎯 Tracking Coordinator

File: `vision_engine/tracking_coordinator.py`

Responsibilities:

- Orchestrate all vision engines
- Synchronize frame processing
- Aggregate tracking data
- Calculate overall confidence

- Manage performance metrics

Key Methods:

```python
process_frame(frame) → (processed_frame, tracking_data)
calibrate() → None
reset() → None
get_stats() → Dict
```

🤸 Body Tracker

File: `vision_engine/body_tracker.py`

Technology: MediaPipe Holistic

Features:

- 33 pose landmarks
- Left/right hand tracking (21 landmarks each)
- Skeleton connections
- Body metrics (shoulder width, height)
- Visibility confidence

Output Structure:

```python
```

```
{
 "detected": True,
 "confidence": 0.95,
 "pose": {
  "landmarks": [...],  # 33 points
  "keypoints": {...},  # Key body parts
  "skeleton_connections": [...]
 },
 "hands": {
  "left": {...},
  "right": {...}
 },
 "metrics": {
  "shoulder_width": 0.3,
  "torso_height": 0.6
 }
}
```

👁️ Iris Tracker

File: `vision_engine/iris_tracker.py`

Technology: MediaPipe Face Mesh with Iris

Features:

- **Ultra-precision iris landmarks** (5 points per iris)
- **Pupil detection** with diameter estimation
- **Gaze direction** (3D vectors)
- **Blink detection** with confidence
- **Micro-saccades** (rapid eye movements)
- **Pupil dilation** tracking
- **Sub-pixel accuracy** with Kalman filtering

Output Structure:

python

```
{
 "detected": True,
 "confidence": 0.98,
 "left_eye": {
  "iris": {
   "center": {"x": 320, "y": 240, "z": 0.05},
   "radius": 15.2,
   "landmarks": [...]  # 5 points
  },
  "pupil": {
   "center": {...},
   "diameter": 5.3,
   "dilation_rate": 0.02
  },
  "openness": 0.85,
  "blink": False
 },
 "right_eye": {...},
 "gaze": {
  "x": 0.1, "y": -0.05, "z": 0.95,
  "magnitude": 1.2
 },
 "saccade": {
  "detected": True,
  "velocity": 0.08
 },
 "blink_rate": 15.3
}
```

👋 Gesture Engine

File: `vision_engine/gesture_engine.py`

Technology: Custom rule-based + temporal smoothing

Features:

- 9+ gestures (expandable)
- Temporal smoothing (buffer-based)
- Cooldown to prevent spam
- Confidence thresholding
- Gesture history tracking

Supported Gestures:

- Peace (✌️)
- Stop (🖐️)
- Thumbs Up/Down (👍 👎)
- Fist (✊)
- Point (👉)
- Swipe (➡️)
- Arms Crossed (🙅)

Output Structure:

```python
{
  "gesture": "peace",
  "confidence": 0.92
}
```

🎯 Object Detector

File: `vision_engine/object_detector.py`

Technology: YOLOv8

Features:

- Real-time object detection
- Configurable confidence threshold
- Multi-object tracking
- Class-based filtering
- GPU acceleration

Output Structure:

```python
```

```
{
 "detected": True,
 "detections": [
  {
   "bbox": {"x1": 100, "y1": 200, "x2": 300, "y2": 400},
   "label": "person",
   "confidence": 0.95,
   "class_id": 0
  }
 ],
 "count": 3,
 "confidence": 0.92
}
```

🎨 Cyberpunk Visualizer

File: `vision_engine/visualizer.py`

Features:

- Neon overlays (red/cyan)
- Glow effects
- HUD rendering
- Skeleton drawing
- Iris crosshairs
- Bounding boxes

Color Scheme:

- Primary: Red (0, 0, 255)
- Secondary: Cyan (255, 255, 0)
- Accent: Magenta (255, 0, 255)
- Success: Green (0, 255, 0)

🔧 Kalman Filters

File: `vision_engine/filters.py`

Purpose: Smooth noisy tracking data

Types:

- **1D Filter:** Single coordinate smoothing
- **2D Filter:** X,Y coordinate pairs

Benefits:

- Reduces jitter
- Predicts motion
- Handles occlusion
- Ultra-smooth tracking

Frontend Integration

WebSocket Integration

```javascript
```

```javascript
// Connection
const vision = new WebSocket('ws://localhost:8000/ws');

// Event handlers
vision.onopen = () => {
  console.log('NETRAX connected');
};

vision.onmessage = (event) => {
  const message = JSON.parse(event.data);

  switch(message.type) {
    case 'stats':
      updateFPS(message.stats.fps);
      updateConfidence(message.stats.confidence);
      break;

    case 'gesture_command':
      handleGesture(message.command);
      break;

    case 'tracking_detail':
      // Full tracking data for advanced visualizations
      renderTracking(message.data);
      break;
  }
};

vision.onerror = (error) => {
  console.error('WebSocket error:', error);
};

// Send commands
function calibrate() {
  vision.send(JSON.stringify({
    type: 'command',
    command: 'calibrate'
  }));
}
```

Video Feed Integration

```
html
```

```
<!-- Method 1: Direct embed -->
<img src="http://localhost:8000/video_feed"
     style="width: 100%; height: auto;">

<!-- Method 2: JavaScript -->
<canvas id="videoCanvas"></canvas>

<script>
const canvas = document.getElementById('videoCanvas');
const ctx = canvas.getContext('2d');
const video = new Image();

video.onload = () => {
  ctx.drawImage(video, 0, 0, canvas.width, canvas.height);
  video.src = 'http://localhost:8000/video_feed?' + Date.now();
};

video.src = 'http://localhost:8000/video_feed';
</script>
```

Performance Optimization

Frame Processing Pipeline

```
Frame Capture (1ms)
    ↓
RGB Conversion (0.5ms)
    ↓
Body Tracking (10ms)
    ↓
Iris Tracking (8ms)
    ↓
Gesture Recognition (2ms)
    ↓
Object Detection (15ms) [optional]
    ↓
Kalman Filtering (1ms)
    ↓
Visualization (3ms)
    ↓
Total: ~25-40ms (25-40 FPS)
```

Optimization Strategies

1. **Frame Skipping**

```python
ENABLE_FRAME_SKIP=true
FRAME_SKIP_INTERVAL=2  # Process every 2nd frame
```

2. **Resolution Scaling**

```python
# High FPS
FRAME_WIDTH=640
FRAME_HEIGHT=480

# High Quality
FRAME_WIDTH=1920
FRAME_HEIGHT=1080
```

3. **Module Selection**

```python
# Disable heavy modules
ENABLE_OBJECT_DETECTION=false
BODY_MODEL_COMPLEXITY=0  # Lite mode
```

4. **GPU Acceleration**

```python
USE_GPU=true
YOLO_DEVICE=cuda
```

5. **Kalman Filtering**

```python
# Balance smoothness vs responsiveness
KALMAN_PROCESS_NOISE=0.01
KALMAN_MEASUREMENT_NOISE=0.1
```

Performance Profiling

```python
# Enable performance logging
LOG_PERFORMANCE=true

# Check logs
tail -f logs/performance.log
```

Deployment Options

1. Docker (Recommended)

Pros:

- Consistent environment
- Easy deployment
- Isolated dependencies

Cons:

- Larger image size
- Camera passthrough complexity

2. Systemd Service (Linux)

Pros:

- Native performance
- Auto-restart
- System integration

Cons:

- OS-specific
- Manual dependency management

3. PM2 (Node.js Process Manager)

Pros:

- Cross-platform
- Auto-restart
- Log management

Cons:

- Requires Node.js

4. Kubernetes

Pros:

- Scalable
- Load balancing
- High availability

Cons:

- Complex setup
- Overkill for single instance

Security Considerations

Authentication

```python
```

```python
# Add to main.py
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials

security = HTTPBearer()

@app.websocket("/ws")
async def secure_websocket(
    websocket: WebSocket,
    credentials: HTTPAuthorizationCredentials = Depends(security)
):
    # Verify token
    if not verify_token(credentials.credentials):
        await websocket.close(code=1008)
        return

    # ... normal WebSocket logic
```

Rate Limiting

python

```python
from slowapi import Limiter, _rate_limit_exceeded_handler
from slowapi.util import get_remote_address

limiter = Limiter(key_func=get_remote_address)
app.state.limiter = limiter

@app.get("/video_feed")
@limiter.limit("10/minute")
async def rate_limited_video_feed(request: Request):
    # ... video feed logic
```

CORS Configuration

python

```python
# Restrict origins in production
app.add_middleware(
    CORSMiddleware,
    allow_origins=["https://yourdomain.com"],  # Not "*"
    allow_credentials=True,
    allow_methods=["GET", "POST"],
    allow_headers=["*"],
)
```

Monitoring & Logging

Structured Logging

python

```python
import logging
from pythonjsonlogger import jsonlogger

logHandler = logging.StreamHandler()
formatter = jsonlogger.JsonFormatter()
logHandler.setFormatter(formatter)
logger.addHandler(logHandler)

logger.info("Gesture detected", extra={
    "gesture": "peace",
    "confidence": 0.92,
    "user_id": "user123"
})
```

Metrics Export

python

```python
from prometheus_client import Counter, Histogram

frames_processed = Counter('frames_processed', 'Total frames')
processing_time = Histogram('processing_time', 'Frame processing time')

# In processing loop
frames_processed.inc()
with processing_time.time():
    process_frame(frame)
```

Extending the System

Adding Custom Gestures

1. **Define detection function:**

python

```
# gesture_engine.py
def _detect_wave(self, body_data: Dict) -> float:
    # Your detection logic
    return confidence
```

2. **Register gesture:**

```
python

# In __init__
self.gesture_rules["wave"] = self._detect_wave
```

3. **Update frontend:**

```
javascript

// Handle new gesture
case 'wave':
  playWaveAnimation();
  break;
```

Adding Custom Vision Module

1. **Create module:**

```
python

# vision_engine/custom_module.py
class CustomModule:
    def process(self, frame):
        # Your processing
        return results
```

2. **Register in coordinator:**

```
python

# tracking_coordinator.py
self.custom_module = CustomModule()

# In process_frame
custom_results = self.custom_module.process(frame)
tracking_data["custom"] = custom_results
```

3. **Update config:**

```python
# config.py
ENABLE_CUSTOM_MODULE: bool = True
```

🎯 Summary

NETRAX is a **modular, high-performance vision system** designed for:

- Real-time human tracking
- Gesture-based interaction
- Production deployment
- Cyberpunk aesthetics

The architecture is **extensible, scalable, and optimized** for both development and production use.

Key Strengths:

- ✅ Modular design
- ✅ Real-time performance
- ✅ Production-ready
- ✅ Well-documented
- ✅ Extensible

Next Steps:

1. Review system requirements
2. Deploy using Docker
3. Integrate with your frontend
4. Customize for your use case
5. Deploy to production

NETRAX - Always watching. Always ready. 👁️