TonyPi Robot Monitoring System - Complete System Documentation

> **Prepared for Assessment Presentation**
> A comprehensive full-stack IoT monitoring system for HiWonder TonyPi humanoid robots

Table of Contents

1. System Overview

1.1 What is this System?

The **TonyPi Robot Monitoring System** is a full-stack IoT solution designed to monitor, control, and analyze HiWonder TonyPi humanoid robots in real-time. It provides:

- **Real-time Telemetry**: Live monitoring of robot sensors, servos, and system metrics
- **Remote Control**: Send commands to robots from a web interface
- **Data Analytics**: AI-powered analysis using Google Gemini
- **Alert Management**: Configurable thresholds with automated notifications
- **Historical Analysis**: Time-series data storage and visualization

1.2 Problem Statement

Managing multiple robots in an operational environment requires:

- Continuous health monitoring to prevent failures
- Centralized control and command dispatch
- Historical data for predictive maintenance
- Real-time alerts for critical situations

1.3 Solution

A microservices-based architecture with:

- **Decoupled Components**: Each service handles specific responsibilities
- **Real-time Communication**: MQTT protocol for low-latency messaging
- **Dual Database Strategy**: Time-series + Relational for optimal storage
- **Modern Web Interface**: Responsive React dashboard

---

2. Technology Stack & Justification

2.1 Backend Technologies

Technology	Version	Purpose	Why Chosen
Python	3.8+	Primary backend language	Excellent IoT ecosystem, async support, extensive libraries
FastAPI	0.104.1	REST API framework	Modern async framework, automatic OpenAPI docs, type hints, high performance
Pydantic	2.5.0	Data validation	Type-safe validation, automatic serialization, FastAPI integration
SQLAlchemy	2.0.23	ORM (Object-Relational Mapper)	Industry standard, supports PostgreSQL, type hints in v2
Paho-MQTT	1.6.1	MQTT client	Official Eclipse implementation, reliable, well-documented
AsyncSSH	2.14.2	SSH tunneling	Async SSH for WebSocket terminal access
bcrypt	4.0+	Password hashing	Industry-standard secure hashing algorithm
python-jose	3.3.0	JWT tokens	JSON Web Tokens for authentication

Why FastAPI over Flask/Django?

```
FastAPI Advantages:
├── Async Support        → Handle thousands of concurrent WebSocket connections
├── Automatic Docs       → OpenAPI/Swagger at /docs without extra work
├── Type Hints           → Catch errors at development time
├── Performance          → One of the fastest Python frameworks
└── Modern Python        → Uses latest Python features (3.7+)
```

2.2 Frontend Technologies

Technology	Version	Purpose	Why Chosen
React	18.2.0	UI framework	Component-based, large ecosystem, excellent for real-time UIs

Technology	Version	Purpose	Why Chosen
TypeScript	4.9.5	Type-safe JavaScript	Catch errors at compile time, better IDE support
Tailwind CSS	3.x	Utility-first CSS	Rapid development, consistent design, small bundle size
Recharts	2.8.0	Charts library	React-native, responsive charts
MQTT.js	5.3.0	Browser MQTT client	WebSocket support for real-time updates
xterm.js	5.3.0	Terminal emulator	Full terminal in browser for SSH access
Axios	1.5.0	HTTP client	Promise-based, interceptors, error handling
Lucide React	0.400.0	Icons	Modern, consistent icon set

Why React over Vue/Angular?

```
React Advantages:
├── Component Reusability  → Build once, use everywhere
├── Virtual DOM            → Efficient updates for real-time data
├── Hooks                  → Clean state management without classes
├── Ecosystem              → Massive library ecosystem
└── React Query/SWR        → Excellent data fetching patterns
```

2.3 Database Technologies

Database	Version	Type	Purpose	Why Chosen
PostgreSQL	Latest	Relational (SQL)	User accounts, alerts, jobs, reports	ACID compliance, JSON support, robust & scalable
InfluxDB	2.7	Time-Series	Sensor data, servo telemetry, metrics	Optimized for timestamped data, built-in retention policies, Flux query language

Why Dual Database Strategy?

```
┌─────────────────────────────────────────────────────────────┐
│                    DATA STORAGE STRATEGY                    │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  POSTGRESQL (Relational)          INFLUXDB (Time-Series)    │
│  ─────────────────────            ──────────────────────    │
│  ✓ User accounts                  ✓ Sensor readings         │
│  ✓ Robot configurations           ✓ Servo telemetry         │
│  ✓ Alert definitions              ✓ CPU/Memory metrics      │
```

```
│  ✓ Job metadata                  ✓ Battery history            │
│  ✓ Reports                       ✓ Location tracking          │
│  ✓ System logs                   ✓ Performance metrics        │
│                                                               │
│  WHY: Relationships, ACID        WHY: High write throughput,  │
│       transactions, complex           time-range queries,     │
│       queries with JOINs              automatic downsampling  │
└──────────────────────────────────────────────────────────────┘
```

PostgreSQL is ideal for:

- Data that requires relationships (users → alerts → robots)
- Transactional integrity (user registration, job updates)
- Complex queries with JOINs

InfluxDB is ideal for:

- High-frequency writes (10+ data points per second per robot)
- Time-range queries ("last 24 hours of CPU data")
- Automatic data retention (delete data older than 30 days)
- Aggregations (average, max, min over time windows)

2.4 Infrastructure Technologies

Technology	Version	Purpose	Why Chosen
Docker	Latest	Containerization	Consistent environments, easy deployment
Docker Compose	v2.0+	Multi-container orchestration	Define entire stack in one file
Mosquitto	2.0	MQTT Broker	Lightweight, reliable, supports WebSocket
Grafana	Latest	Advanced visualization	Industry standard for metrics dashboards
Nginx	(in Docker)	Reverse proxy	Load balancing, SSL termination

Why Docker?

```
Without Docker:                    With Docker:
─────────────────                  ─────────────────
"Works on my machine"       →      Works everywhere
Manual dependency install   →      Automatic with Dockerfile
Version conflicts           →      Isolated containers
Complex production setup    →      docker-compose up -d
```

2.5 AI/ML Integration

Technology	Purpose	Why Chosen
Google Gemini API	AI-powered analytics	Free tier available, powerful analysis, easy integration

3. Project Structure & File Organization

```
Monitoring_System_TonyPi/
│
├── 🗀 backend/                    # FastAPI Backend Server (Port 8000)
│   ├── main.py                   # Application entry point, lifecycle management
│   ├── requirements.txt          # Python dependencies
│   ├── Dockerfile                # Container build instructions
│   │
│   ├── 🗀 routers/               # API Endpoint Handlers (Controllers)
│   │   ├── robot_data.py         # GET/POST robot telemetry endpoints
│   │   ├── alerts.py             # Alert CRUD, thresholds configuration
│   │   ├── reports.py            # Report generation with AI
│   │   ├── management.py         # Robot commands (move, speak, etc.)
│   │   ├── ssh.py                # WebSocket SSH proxy for terminal
│   │   ├── users.py              # Authentication & user management
│   │   ├── logs.py               # System activity logs
│   │   ├── jobs.py               # Job/task tracking
│   │   ├── pi_perf.py            # Raspberry Pi performance metrics
│   │   ├── health.py             # Health check endpoint
│   │   └── grafana_proxy.py      # Proxy requests to Grafana
│   │
│   ├── 🗀 models/                # SQLAlchemy ORM Models (Database Tables)
│   │   ├── robot.py              # Robot entity (config, thresholds)
│   │   ├── user.py               # User entity (auth, roles)
│   │   ├── alert.py              # Alert + AlertThreshold entities
│   │   ├── job.py                # Job/task entity
│   │   ├── report.py             # Report entity
│   │   └── system_log.py         # Log entry entity
│   │
│   ├── 🗀 database/              # Database Connections
│   │   ├── database.py           # PostgreSQL connection (SQLAlchemy)
│   │   └── influx_client.py      # InfluxDB client (time-series)
│   │
│   ├── 🗀 mqtt/                  # MQTT Integration
│   │   └── mqtt_client.py        # Subscribe to topics, process messages
│   │
│   ├── 🗀 services/              # Business Logic Layer
│   │   └── gemini_analytics.py   # AI-powered data analysis
│   │
│   ├── 🗀 utils/                 # Utility Functions
│   │   └── ...                   # Helpers, validators, formatters
│   │
│   └── 🗀 tests/                 # Unit & Integration Tests
│       └── ...                   # pytest test files
```

```
│
├── 🗀 frontend/                    # React Web Application (Port 3001)
│   ├── package.json                # Node.js dependencies
│   ├── tsconfig.json               # TypeScript configuration
│   ├── tailwind.config.js          # Tailwind CSS configuration
│   ├── Dockerfile                  # Container build instructions
│   │
│   └── 🗀 src/
│       ├── index.tsx               # React entry point
│       ├── App.tsx                 # Router-based main component
│       ├── TonyPiApp.tsx           # Tab-based main component (alternative)
│       │
│       ├── 🗀 pages/               # Page Components (Views)
│       │   ├── Dashboard.tsx       # System overview
│       │   ├── Monitoring.tsx      # Performance metrics (CPU, RAM)
│       │   ├── Sensors.tsx         # IMU, light, ultrasonic sensors
│       │   ├── Servos.tsx          # Servo motor monitoring
│       │   ├── Robots.tsx          # Robot management
│       │   ├── Jobs.tsx            # Task tracking
│       │   ├── Alerts.tsx          # Alert management
│       │   ├── Logs.tsx            # Activity logs
│       │   ├── Reports.tsx         # Report generation
│       │   ├── Users.tsx           # User management (admin)
│       │   └── Login.tsx           # Authentication page
│       │
│       ├── 🗀 components/          # Reusable UI Components
│       │   ├── Layout.tsx          # Application shell/wrapper
│       │   ├── SSHTerminal.tsx     # Web-based SSH terminal
│       │   ├── GrafanaPanel.tsx    # Embedded Grafana charts
│       │   └── Toast.tsx           # Notification toasts
│       │
│       ├── 🗀 contexts/            # React Context Providers
│       │   ├── AuthContext.tsx     # Authentication state
│       │   ├── ThemeContext.tsx    # Dark/light theme
│       │   └── NotificationContext.tsx  # Toast notifications
│       │
│       ├── 🗀 utils/               # Utility Functions
│       │   └── api.ts              # API service (Axios wrapper)
│       │
│       └── 🗀 types/               # TypeScript Type Definitions
│           └── index.ts            # Shared interfaces
│
├── 🗀 robot_client/                # Code Running ON the TonyPi Robot
│   ├── tonypi_client.py            # Main robot client (MQTT publisher)
│   ├── camera_stream.py            # HTTP camera streaming server
│   ├── simulator.py                # Simulation mode (no hardware)
│   ├── requirements.txt            # Python dependencies
│   │
│   └── 🗀 hiwonder/                # HiWonder SDK Wrapper
│       ├── Controller.py           # Servo control interface
│       ├── Sonar.py                # Ultrasonic sensor interface
│       └── ...                     # Other hardware interfaces
│
├── 🗀 mosquitto/                   # MQTT Broker Configuration
```

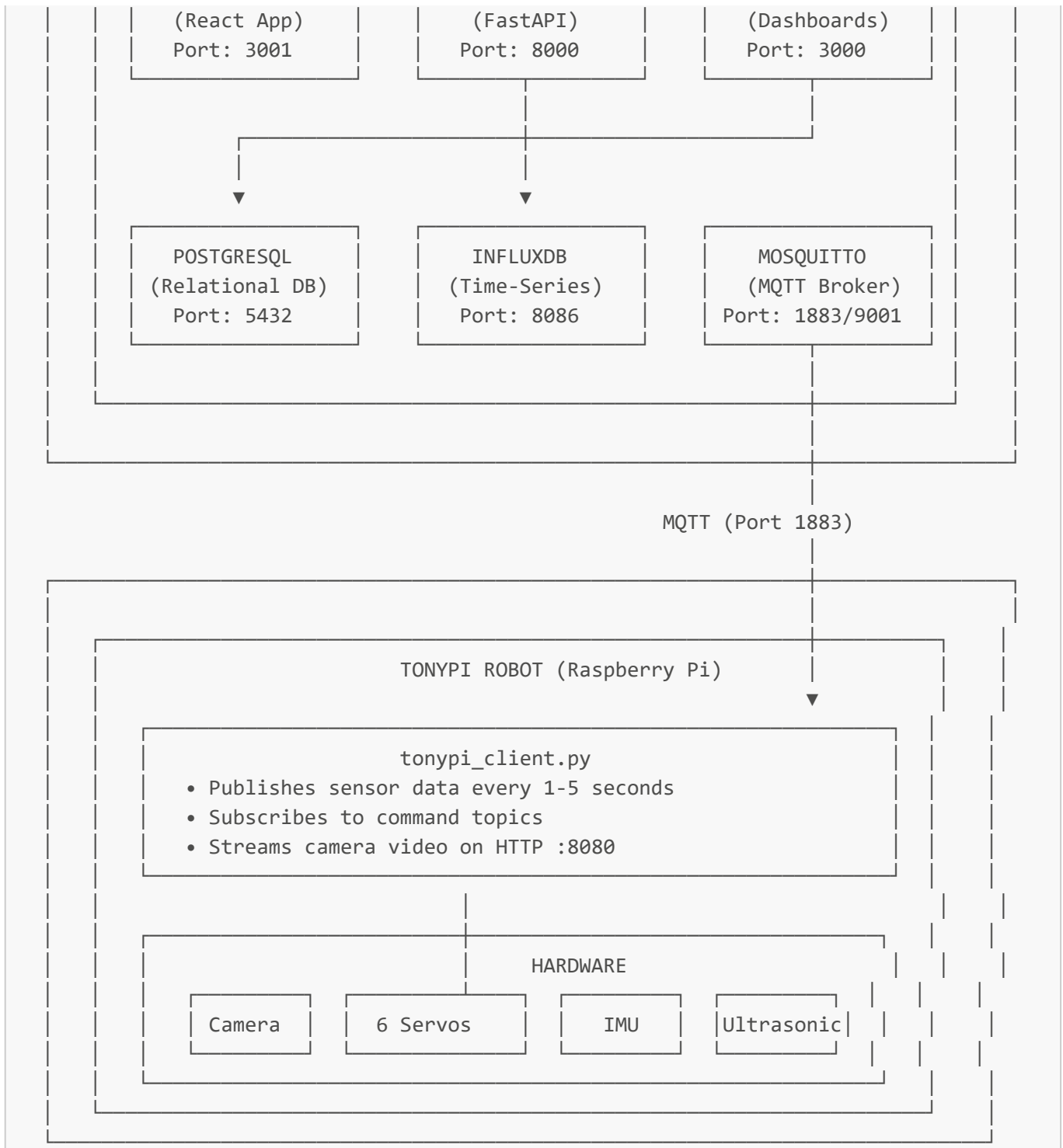
```
│   ├── 🗀 config/
│   │   └── mosquitto.conf        # Broker settings
│   ├── 🗀 data/                   # Persistent messages
│   └── 🗀 log/                    # Broker logs
│
├── 🗀 influxdb/                    # InfluxDB Configuration
│   ├── 🗀 config/                 # Database configuration
│   └── 🗀 data/                   # Persistent database files
│
├── 🗀 postgres/                    # PostgreSQL Configuration
│   ├── 🗀 init/                   # Initialization scripts
│   └── 🗀 data/                   # Persistent database files
│
├── 🗀 grafana/                     # Grafana Configuration
│   ├── 🗀 provisioning/           # Auto-configured datasources
│   └── 🗀 data/                   # Dashboard definitions
│
├── 🗀 tests/                       # Integration Tests
│   └── comprehensive_test_suite.py
│
├── docker-compose.yml             # Container Orchestration
├── env.example                    # Environment variables template
└── README.md                      # Project documentation
```

---

4. System Architecture & Data Flow

4.1 High-Level Architecture Diagram

```
┌─────────────────────────────────────────────────────────────────────────────┐
│                                                                             │
│                          SYSTEM ARCHITECTURE                                │
│                                                                             │
└─────────────────────────────────────────────────────────────────────────────┘

                          ┌──────────────────────┐
                          │    USER (Browser)    │
                          └──────────┬───────────┘
                                     │
                    HTTP (Port 3001) │ WebSocket (Port 9001)
                                     │
┌────────────────────────────────────┼────────────────────────────────────────┐
│                                    │                                        │
│                                    │          MONITORING SERVER             │
│                                    │                                        │
│   ┌────────────────────────────────┼──────────────────────────────────┐     │
│   │                                ▼                                  │     │
│   │                                                                   │     │
│   │   ┌──────────────┐      ┌──────────────┐      ┌──────────────┐    │     │
│   │   │              │      │              │      │              │    │     │
│   │   │   FRONTEND   │◄────►│   BACKEND    │◄────►│   GRAFANA    │    │     │
│   │   │              │      │              │      │              │    │     │
```



4.2 Data Flow Diagram

```
┌─────────────────────────────────────────────────────────────────────────────┐
│                                                                             │
│                              DATA FLOW                                      │
│                                                                             │
└─────────────────────────────────────────────────────────────────────────────┘

STEP 1: Robot Collects Data
═══════════════════════════
```

```
   TonyPi Robot Hardware
          │
          ├── IMU Sensor ──────────────► accel_x, accel_y, accel_z
          │                              gyro_x, gyro_y, gyro_z
          │
          ├── Ultrasonic ──────────────► distance_cm
          │
          ├── Light Sensor ────────────► light_level, is_dark
          │
          ├── 6 Servos ────────────────► position, temperature, voltage (each)
          │
          ├── System ──────────────────► cpu_percent, memory_percent, disk_percent
          │
          └── Battery ─────────────────► voltage, percentage


STEP 2: Robot Publishes to MQTT
═══════════════════════════════

   tonypi_client.py
          │
          ├── tonypi/sensors/{robot_id} ──► {"accel_x": 0.02, "gyro_z": 0.1, ...}
          │
          ├── tonypi/servos/{robot_id} ──► {"servos": [{"id": 1, "position": 512},
...]}
          │
          ├── tonypi/status/{robot_id} ──► {"cpu": 45, "memory": 62, "temp": 55}
          │
          └── tonypi/battery ────────────► {"voltage": 11.2, "percent": 85}


STEP 3: Backend Receives & Processes
════════════════════════════════════

   Mosquitto Broker (1883)
          │
          └──► mqtt_client.py (Backend)
                    │
                    ├── Parse JSON payload
                    │
                    ├── Write to InfluxDB ─────► Time-series storage
                    │
                    ├── Check thresholds ──────► Generate Alert if exceeded
                    │
                    └── Update robot status ──► PostgreSQL


STEP 4: Frontend Receives Real-time Updates
═══════════════════════════════════════════

   mqtt_client.py (Frontend via WebSocket 9001)
          │
          ├── Subscribe to relevant topics
          │
          └── Update React state ──► UI re-renders with new data
```

```
STEP 5: User Views & Interacts
═══════════════════════════════

   Browser Dashboard
          │
          ├── View real-time sensor data
          │
          ├── View historical charts (Grafana queries InfluxDB)
          │
          ├── Manage alerts (Backend queries PostgreSQL)
          │
          └── Send commands ──► Backend ──► MQTT ──► Robot
```

4.3 MQTT Topic Structure

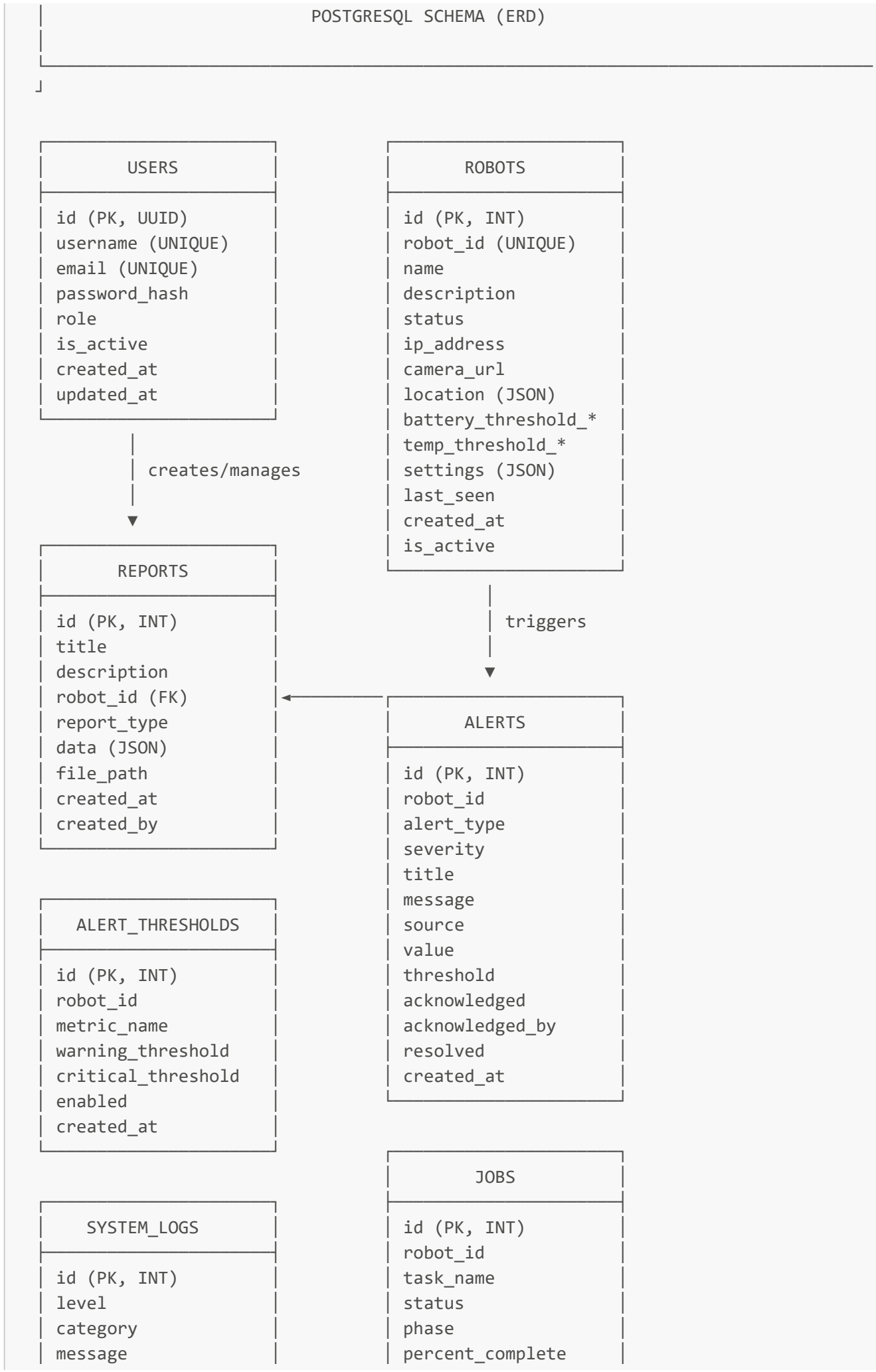
```
MQTT Topics Used
════════════════

PUBLISH (Robot → Server):
┌──────────────────────────────┬──────────────────────────────────────────┐
│ Topic                        │ Data                                     │
├──────────────────────────────┼──────────────────────────────────────────┤
│ tonypi/sensors/{robot_id}    │ IMU, temperature, light, ultrasonic      │
│ tonypi/servos/{robot_id}     │ 6 servos: position, temp, voltage        │
│ tonypi/status/{robot_id}     │ CPU, memory, disk, IP address            │
│ tonypi/battery               │ Voltage, percentage, charging            │
│ tonypi/location              │ X, Y, Z coordinates                      │
│ tonypi/job/{robot_id}        │ Job progress, phase, items done          │
│ tonypi/scan/{robot_id}       │ QR code scan results                     │
│ tonypi/vision/{robot_id}     │ Object detection results                 │
│ tonypi/logs/{robot_id}       │ Console output, errors                   │
└──────────────────────────────┴──────────────────────────────────────────┘

SUBSCRIBE (Server → Robot):
┌──────────────────────────────┬──────────────────────────────────────────┐
│ Topic                        │ Data                                     │
├──────────────────────────────┼──────────────────────────────────────────┤
│ tonypi/commands/{robot_id}   │ Movement, action, voice commands         │
│ tonypi/config/{robot_id}     │ Configuration updates                    │
└──────────────────────────────┴──────────────────────────────────────────┘
```

---

5. Database Schema Design

5.1 PostgreSQL Schema (Relational Data)

Entity Relationship Diagram


POSTGRESQL SCHEMA (ERD)
USERS
id (PK, UUID)
username (UNIQUE)
email (UNIQUE)
password_hash
role
is_active
created_at
updated_at
creates/manages
ROBOTS
id (PK, INT)
robot_id (UNIQUE)
name
description
status
ip_address
camera_url
location (JSON)
battery_threshold_*
temp_threshold_*
settings (JSON)
last_seen
created_at
is_active
triggers
REPORTS
id (PK, INT)
title
description
robot_id (FK)
report_type
data (JSON)
file_path
created_at
created_by
ALERTS
id (PK, INT)
robot_id
alert_type
severity
title
message
source
value
threshold
acknowledged
acknowledged_by
resolved
created_at
ALERT_THRESHOLDS
id (PK, INT)
robot_id
metric_name
warning_threshold
critical_threshold
enabled
created_at
JOBS
id (PK, INT)
robot_id
task_name
status
phase
percent_complete
SYSTEM_LOGS
id (PK, INT)
level
category
message

```
| robot_id              |          | items_total           |
| details (JSON)        |          | items_done            |
| timestamp             |          | start_time            |
└───────────────────────┘          | end_time              |
                                   | success               |
                                   | created_at            |
                                   └───────────────────────┘
```

Table Definitions

USERS Table

```sql
CREATE TABLE users (
    id              VARCHAR(36) PRIMARY KEY,  -- UUID format
    username        VARCHAR(50) UNIQUE NOT NULL,
    email           VARCHAR(100) UNIQUE NOT NULL,
    password_hash   VARCHAR(255) NOT NULL,    -- bcrypt hash
    role            VARCHAR(20) NOT NULL,     -- 'admin', 'operator', 'viewer'
    is_active       BOOLEAN DEFAULT TRUE,
    created_at      TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at      TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);
```

ROBOTS Table

```sql
CREATE TABLE robots (
    id                          SERIAL PRIMARY KEY,
    robot_id                    VARCHAR(100) UNIQUE NOT NULL,  -- e.g., "tonypi_01"
    name                        VARCHAR(100),
    description                 TEXT,
    status                      VARCHAR(20) DEFAULT 'offline', -- 'online',
'offline', 'error'
    ip_address                  VARCHAR(45),                    -- IPv4 or IPv6
    camera_url                  VARCHAR(255),
    location                    JSONB,                          -- {"x": 0, "y": 0,
"z": 0}
    battery_threshold_low       FLOAT DEFAULT 20.0,
    battery_threshold_critical  FLOAT DEFAULT 10.0,
    temp_threshold_warning      FLOAT DEFAULT 70.0,
    temp_threshold_critical     FLOAT DEFAULT 80.0,
    settings                    JSONB DEFAULT '{}',
    last_seen                   TIMESTAMP WITH TIME ZONE,
    created_at                  TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at                  TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    is_active                   BOOLEAN DEFAULT TRUE
);
```

ALERTS Table

```
CREATE TABLE alerts (
    id              SERIAL PRIMARY KEY,
    robot_id        VARCHAR(100),
    alert_type      VARCHAR(50) NOT NULL,     -- 'temperature', 'battery',
'servo_temp'
    severity        VARCHAR(20) NOT NULL,     -- 'critical', 'warning', 'info'
    title           VARCHAR(255) NOT NULL,
    message         TEXT NOT NULL,
    source          VARCHAR(100),             -- 'servo_1', 'cpu', 'battery'
    value           FLOAT,                    -- Actual value that triggered alert
    threshold       FLOAT,                    -- Threshold that was exceeded
    acknowledged    BOOLEAN DEFAULT FALSE,
    acknowledged_by VARCHAR(50),
    acknowledged_at TIMESTAMP WITH TIME ZONE,
    resolved        BOOLEAN DEFAULT FALSE,
    resolved_at     TIMESTAMP WITH TIME ZONE,
    details         JSONB,
    created_at      TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);
```

JOBS Table

```
CREATE TABLE jobs (
    id                  SERIAL PRIMARY KEY,
    robot_id            VARCHAR(100) NOT NULL,
    task_name           VARCHAR(255),
    status              VARCHAR(20) DEFAULT 'active',  -- 'active', 'completed',
'cancelled', 'failed'
    phase               VARCHAR(50),                   -- 'scanning', 'searching',
'executing', 'done'
    percent_complete    FLOAT DEFAULT 0,
    items_total         INTEGER DEFAULT 0,
    items_done          INTEGER DEFAULT 0,
    last_item           JSONB,
    start_time          TIMESTAMP WITH TIME ZONE,
    end_time            TIMESTAMP WITH TIME ZONE,
    estimated_duration  FLOAT,                         -- seconds
    action_duration     FLOAT,                         -- actual seconds
    success             BOOLEAN,
    cancel_reason       TEXT,
    created_at          TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at          TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);
```

REPORTS Table

```
CREATE TABLE reports (
    id              SERIAL PRIMARY KEY,
```

```
    title           VARCHAR(255) NOT NULL,
    description     TEXT,
    robot_id        VARCHAR(100),
    report_type     VARCHAR(50) NOT NULL,      -- 'performance', 'job',
'maintenance', 'custom'
    data            JSONB NOT NULL,            -- Report content and AI analysis
    file_path       VARCHAR(255),              -- Path to generated PDF
    created_at      TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    created_by      VARCHAR(100)
);
```

SYSTEM_LOGS Table

```
CREATE TABLE system_logs (
    id          SERIAL PRIMARY KEY,
    level       VARCHAR(20) NOT NULL,          -- 'INFO', 'WARNING', 'ERROR',
'CRITICAL'
    category    VARCHAR(50) NOT NULL,          -- 'mqtt', 'api', 'database',
'system'
    message     TEXT NOT NULL,
    robot_id    VARCHAR(100),
    details     JSONB,
    timestamp   TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);
```

5.2 InfluxDB Schema (Time-Series Data)

InfluxDB uses a different model: **Measurements**, **Tags**, **Fields**, and **Timestamps**.

```
┌─────────────────────────────────────────────────────────────────────────────┐
│                          INFLUXDB MEASUREMENTS                              │
└─────────────────────────────────────────────────────────────────────────────┘

MEASUREMENT: sensor_data
─────────────────────────
Tags (indexed, for filtering):
  • robot_id: "tonypi_01"
  • sensor_type: "accelerometer_x" | "gyroscope_y" | "cpu_temperature" | ...

Fields (values):
  • value: 0.5 (float)
  • unit: "m/s^2" (string)

Timestamp: 2026-01-26T10:30:00.123Z
```

```
MEASUREMENT: servo_data
──────────────────────
Tags:
  • robot_id: "tonypi_01"
  • servo_id: "1" | "2" | "3" | "4" | "5" | "6"

Fields:
  • position: 512 (int, 0-1023)
  • temperature: 45.2 (float, °C)
  • voltage: 7.2 (float, V)

Timestamp: 2026-01-26T10:30:00.123Z


MEASUREMENT: battery_status
──────────────────────────
Tags:
  • robot_id: "tonypi_01"

Fields:
  • voltage: 11.8 (float, V)
  • percentage: 85.0 (float, %)
  • charging: false (boolean)

Timestamp: 2026-01-26T10:30:00.123Z


MEASUREMENT: robot_status
────────────────────────
Tags:
  • robot_id: "tonypi_01"

Fields:
  • cpu_percent: 45.2 (float)
  • memory_percent: 62.1 (float)
  • disk_percent: 35.0 (float)
  • temperature: 55.5 (float, °C)
  • is_online: true (boolean)

Timestamp: 2026-01-26T10:30:00.123Z


MEASUREMENT: robot_location
──────────────────────────
Tags:
  • robot_id: "tonypi_01"

Fields:
  • x: 1.5 (float, meters)
  • y: 2.3 (float, meters)
  • z: 0.0 (float, meters)

Timestamp: 2026-01-26T10:30:00.123Z
```

```
MEASUREMENT: vision_data
─────────────────────────
Tags:
  • robot_id: "tonypi_01"
  • detection_type: "qr_code" | "color" | "face"

Fields:
  • label: "Red Ball" (string)
  • confidence: 0.95 (float)
  • x: 320 (int, pixels)
  • y: 240 (int, pixels)
  • width: 50 (int, pixels)
  • height: 50 (int, pixels)

Timestamp: 2026-01-26T10:30:00.123Z
```

Example InfluxDB Query (Flux Language)

```
// Get average CPU temperature for last hour
from(bucket: "robot_data")
  |> range(start: -1h)
  |> filter(fn: (r) => r._measurement == "sensor_data")
  |> filter(fn: (r) => r.sensor_type == "cpu_temperature")
  |> filter(fn: (r) => r.robot_id == "tonypi_01")
  |> aggregateWindow(every: 5m, fn: mean)
```

---

6. Key Code Components Explained

6.1 Backend Entry Point (`backend/main.py`)

```python
# Application lifecycle management using context manager
@asynccontextmanager
async def lifespan(app: FastAPI):
    """Manage startup and shutdown events."""
    # STARTUP
    init_database()          # Create PostgreSQL tables
    mqtt_client.start()      # Connect to MQTT broker & subscribe
    yield
    # SHUTDOWN
    mqtt_client.stop()       # Disconnect cleanly

# Create FastAPI application
app = FastAPI(
    title="TonyPi Monitoring API",
    version=API_VERSION,
    lifespan=lifespan
)
```

```
# Configure CORS for frontend access
app.add_middleware(CORSMiddleware, allow_origins=["*"], ...)

# Register all API routers
app.include_router(robot_data.router, prefix=API_PREFIX)
app.include_router(alerts.router, prefix=API_PREFIX)
# ... more routers
```

6.2 MQTT Client (`backend/mqtt/mqtt_client.py`)

```
class MQTTClient:
    """Handles all MQTT communication."""

    def on_connect(self, client, userdata, flags, rc):
        """Subscribe to all robot topics on connection."""
        topics = [
            "tonypi/sensors/+",    # + is single-level wildcard
            "tonypi/servos/+",
            "tonypi/status/+",
            "tonypi/battery",
            # ... more topics
        ]
        for topic in topics:
            client.subscribe(topic)

    def on_message(self, client, userdata, msg):
        """Process incoming MQTT messages."""
        topic = msg.topic
        payload = json.loads(msg.payload)

        if "sensors" in topic:
            self._process_sensor_data(payload)
        elif "servos" in topic:
            self._process_servo_data(payload)
        # ... handle other topics

    def _process_sensor_data(self, data):
        """Write sensor data to InfluxDB and check thresholds."""
        # Write to time-series database
        influx_client.write_sensor_data(
            measurement="sensor_data",
            tags={"robot_id": data["robot_id"]},
            fields={"value": data["value"]}
        )

        # Check if threshold exceeded → create alert
        if data["value"] > threshold:
            self._create_alert(...)
```

6.3 Robot Client (robot_client/tonypi_client.py)

```python
class TonyPiClient:
    """Main robot client running on Raspberry Pi."""

    async def run(self):
        """Main loop: collect and publish data."""
        while True:
            # Collect data from hardware
            sensor_data = self.collect_sensors()
            servo_data = self.collect_servos()
            system_data = self.collect_system_metrics()

            # Publish to MQTT
            self.mqtt_client.publish(
                f"tonypi/sensors/{self.robot_id}",
                json.dumps(sensor_data)
            )

            await asyncio.sleep(1)  # Publish every second

    def collect_sensors(self):
        """Read from IMU, ultrasonic, light sensors."""
        return {
            "robot_id": self.robot_id,
            "accel_x": board.get_accelerometer()[0],
            "accel_y": board.get_accelerometer()[1],
            "accel_z": board.get_accelerometer()[2],
            "gyro_x": board.get_gyroscope()[0],
            # ... more sensors
        }
```

6.4 Frontend Data Fetching (frontend/src/utils/api.ts)

```typescript
// API Service using Axios
class ApiService {
    private api: AxiosInstance;

    constructor() {
        this.api = axios.create({
            baseURL: 'http://localhost:8000/api/v1',
            headers: { 'Content-Type': 'application/json' }
        });

        // Add auth token to requests
        this.api.interceptors.request.use(config => {
            const token = localStorage.getItem('token');
            if (token) {
                config.headers.Authorization = `Bearer ${token}`;
            }
```

```
            return config;
        });
    }

    // Robot endpoints
    getRobots = () => this.api.get('/robots');
    getRobotData = (id: string) => this.api.get(`/robots/${id}/data`);
    sendCommand = (id: string, cmd: object) =>
        this.api.post(`/robots/${id}/command`, cmd);

    // Alert endpoints
    getAlerts = () => this.api.get('/alerts');
    acknowledgeAlert = (id: number) =>
        this.api.put(`/alerts/${id}/acknowledge`);
}
```

6.5 React Component Example (`frontend/src/pages/Sensors.tsx`)

```
const Sensors: React.FC = () => {
    const [sensorData, setSensorData] = useState<SensorData[]>([]);
    const [loading, setLoading] = useState(true);

    useEffect(() => {
        // Fetch initial data
        fetchSensorData();

        // Connect to MQTT for real-time updates
        const mqttClient = mqtt.connect('ws://localhost:9001');
        mqttClient.subscribe('tonypi/sensors/+');
        mqttClient.on('message', (topic, payload) => {
            const data = JSON.parse(payload.toString());
            setSensorData(prev => [...prev, data]);
        });

        return () => mqttClient.end();
    }, []);

    return (
        <div className="grid grid-cols-3 gap-4">
            <SensorCard title="Accelerometer X" value={sensorData.accel_x} />
            <SensorCard title="Gyroscope Y" value={sensorData.gyro_y} />
            {/* More sensor displays */}
        </div>
    );
};
```

---

7. Communication Protocols

7.1 MQTT (Message Queuing Telemetry Transport)

Why MQTT?

- Designed for IoT devices with limited bandwidth
- Publish/Subscribe pattern decouples producers and consumers
- QoS levels ensure message delivery
- Low overhead (2-byte header minimum)

```
MQTT Communication Pattern:
═══════════════════════════

    PUBLISHER                BROKER                SUBSCRIBERS
   (TonyPi Robot)          (Mosquitto)           (Backend, Frontend)
        │                       │                       │
        │  PUBLISH              │                       │
        │  topic: sensors/01    │                       │
        │  payload: {...}       │                       │
        ├──────────────────────►│                       │
        │                       │  FORWARD to           │
        │                       │  all subscribers      │
        │                       ├──────────────────────►│
        │                       │                       │
```

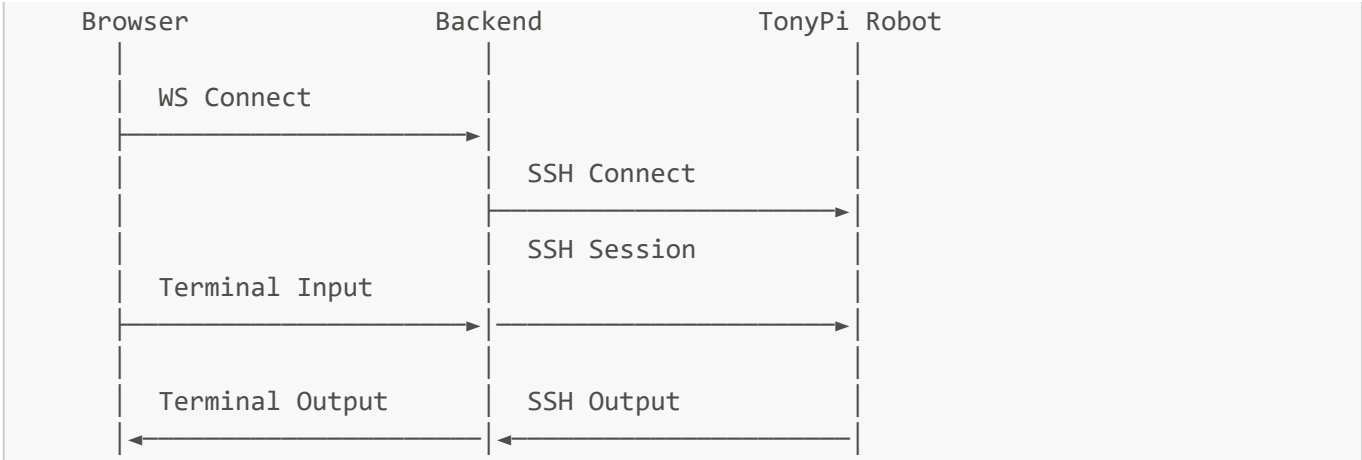
7.2 REST API (HTTP)

Endpoints Structure:

```
GET    /api/v1/robots              → List all robots
GET    /api/v1/robots/{id}         → Get robot details
POST   /api/v1/robots              → Register new robot
PUT    /api/v1/robots/{id}         → Update robot config
DELETE /api/v1/robots/{id}         → Remove robot

GET    /api/v1/alerts              → List alerts
PUT    /api/v1/alerts/{id}/ack     → Acknowledge alert
PUT    /api/v1/alerts/{id}/resolve → Resolve alert

POST   /api/v1/reports             → Generate report
GET    /api/v1/reports/{id}/pdf    → Download PDF

POST   /api/v1/auth/login          → User login
POST   /api/v1/auth/register       → User registration
```

7.3 WebSocket (SSH Terminal)

```
WebSocket SSH Proxy:
════════════════════
```



---

8. Security Implementation

8.1 Authentication Flow

```
JWT Authentication:
═══════════════════

1. LOGIN REQUEST
   POST /api/v1/auth/login
   Body: { "username": "admin", "password": "secret" }
                    │
                    ▼
2. SERVER VALIDATES
   • Check username exists in PostgreSQL
   • Compare bcrypt hash of password
   • If valid, generate JWT token
                    │
                    ▼
3. RETURN TOKEN
   Response: { "access_token": "eyJhbG...", "token_type": "bearer" }
                    │
                    ▼
4. CLIENT STORES TOKEN
   localStorage.setItem('token', 'eyJhbG...')
                    │
                    ▼
5. SUBSEQUENT REQUESTS
   Header: Authorization: Bearer eyJhbG...
                    │
                    ▼
6. SERVER VERIFIES
   • Decode JWT token
   • Verify signature with secret key
   • Check expiration
   • Extract user role for authorization
```

8.2 Role-Based Access Control

```
ROLES:
======
ADMIN     → Full access (users, robots, all data)
OPERATOR  → Control robots, view all data, manage alerts
VIEWER    → Read-only access to dashboards

PROTECTED ENDPOINTS:
====================
/api/v1/users/*           → ADMIN only
/api/v1/robots/*/command  → ADMIN, OPERATOR
/api/v1/alerts/*/resolve  → ADMIN, OPERATOR
/api/v1/robots            → ADMIN, OPERATOR, VIEWER (read)
```

8.3 Password Security

```python
# Password hashing with bcrypt
import bcrypt

def hash_password(plain_password: str) -> str:
    """Hash password with random salt."""
    salt = bcrypt.gensalt(rounds=12)  # 2^12 iterations
    return bcrypt.hashpw(plain_password.encode(), salt).decode()

def verify_password(plain_password: str, hashed: str) -> bool:
    """Verify password against hash."""
    return bcrypt.checkpw(plain_password.encode(), hashed.encode())
```

---

9. Why Use Docker?

9.1 The Problem Without Docker

```
WITHOUT DOCKER - "It works on my machine" problem:
==================================================

Developer Machine              Production Server
┌─────────────────────┐        ┌─────────────────────┐
│ Python 3.11         │        │ Python 3.8          │ ← Version mismatch!
│ PostgreSQL 15       │        │ PostgreSQL 12       │ ← Different version!
│ Node.js 20          │        │ Node.js 16          │ ← Incompatible!
│ Windows 11          │        │ Ubuntu 22.04        │ ← Different OS!
│                     │        │                     │
│ ✓ Works perfectly   │        │ ✗ Crashes!          │
└─────────────────────┘        └─────────────────────┘
```

```
Problems:
❌ Dependency conflicts between Python packages
❌ Database version incompatibilities
❌ "Missing library" errors
❌ Path differences (Windows vs Linux)
❌ Hours spent debugging environment issues
```

9.2 The Solution With Docker

```
WITH DOCKER - Consistent everywhere:
═══════════════════════════════════

┌─────────────────────────────────────────────────────────────┐
│                    DOCKER CONTAINER                         │
│  ┌──────────────────────────────────────────────────────┐   │
│  │  • Exact same Python version (3.11)                  │   │
│  │  • Exact same dependencies (requirements.txt)        │   │
│  │  • Exact same OS environment (Debian)                │   │
│  │  • Exact same configuration                          │   │
│  └──────────────────────────────────────────────────────┘   │
│                                                             │
│  Runs IDENTICALLY on:                                       │
│  ✓ Developer's Windows laptop                               │
│  ✓ Teammate's MacBook                                       │
│  ✓ Production Linux server                                  │
│  ✓ Cloud services (AWS, Azure, etc.)                        │
└─────────────────────────────────────────────────────────────┘
```

9.3 Benefits of Docker for This Project

Benefit	Explanation
One-Command Setup	`docker-compose up -d` starts ALL 6 services instantly
Isolation	Each service runs in its own container, no conflicts
Reproducibility	Same environment on any machine, any OS
Easy Cleanup	`docker-compose down` removes everything cleanly
Version Control	Dockerfile locks exact versions of dependencies
Microservices	Each service can be updated/scaled independently
No Installation Hell	No need to install PostgreSQL, InfluxDB, etc. manually

9.4 Docker vs Traditional Installation

```
TRADITIONAL SETUP (Without Docker):
──────────────────────────────────
```

```
1. Install Python 3.11
2. Install pip packages (may conflict with system packages)
3. Install PostgreSQL 15
4. Configure PostgreSQL users and databases
5. Install InfluxDB 2.7
6. Configure InfluxDB tokens and buckets
7. Install Mosquitto MQTT broker
8. Configure Mosquitto listeners
9. Install Grafana
10. Configure Grafana datasources
11. Install Node.js 20
12. npm install frontend dependencies
13. Configure environment variables
14. Pray nothing conflicts...

Time: 2-4 hours | Success rate: ~60%

═══════════════════════════════════════════════════════════════

DOCKER SETUP:
─────────────
1. Install Docker Desktop
2. Run: docker-compose up -d
3. Done!

Time: 5-10 minutes | Success rate: 99%
```

---

10. System Setup & Startup Guide

10.1 Prerequisites

On Monitoring Server (Your Computer)

Requirement	Minimum	Recommended
OS	Windows 10 / macOS / Linux	Windows 11 / Ubuntu 22.04
RAM	4GB	8GB+
Disk	10GB free	20GB+ free
Docker Desktop	v4.0+	Latest
Docker Compose	v2.0+	Latest

Required Ports (must be available):

- `1883` - MQTT Broker
- `9001` - MQTT WebSocket
- `8086` - InfluxDB
- `5432` - PostgreSQL

- `3000` - Grafana
- `8000` - Backend API
- `3001` - Frontend

On TonyPi Robot (Raspberry Pi)

Requirement	Details
Hardware	HiWonder TonyPi Robot
OS	Raspberry Pi OS (Bullseye)
Python	3.8+
Network	WiFi connected to same network as server
Camera	Pi Camera connected

10.2 Step-by-Step Setup

STEP 1: Clone the Repository

```
# Clone the project
git clone <repository-url>
cd Monitoring_System_TonyPi
```

STEP 2: Create Environment File

```
# Copy example environment file
cp env.example .env

# Edit .env file with your settings (optional)
```

Default `.env` contents:

```
# PostgreSQL Configuration
POSTGRES_DB=tonypi_db
POSTGRES_USER=postgres
POSTGRES_PASSWORD=postgres

# InfluxDB Configuration
INFLUXDB_USERNAME=admin
INFLUXDB_PASSWORD=adminpass
INFLUXDB_ORG=tonypi
INFLUXDB_BUCKET=robot_data
INFLUXDB_TOKEN=my-super-secret-auth-token
```

```
# Grafana Configuration
GRAFANA_USER=admin
GRAFANA_PASSWORD=admin

# AI Analytics (Optional - Get FREE key)
# https://aistudio.google.com/app/apikey
GEMINI_API_KEY=
```

STEP 3: Start the Monitoring System

```
# Start all services with Docker Compose
docker-compose up -d

# Watch startup logs
docker-compose logs -f
```

Expected Output:

```
☑ tonypi_mosquitto    ... Started
☑ tonypi_influxdb     ... Started
☑ tonypi_postgres     ... Started
☑ tonypi_grafana      ... Started
☑ tonypi_backend      ... Started
☑ tonypi_frontend     ... Started
```

STEP 4: Verify Services Are Running

```
# Check all containers
docker-compose ps

# Test backend health
curl http://localhost:8000/api/v1/health
```

Expected Status:

```
NAME                STATUS
tonypi_mosquitto    Up (healthy)
tonypi_influxdb     Up (healthy)
tonypi_postgres     Up (healthy)
tonypi_grafana      Up
tonypi_backend      Up (healthy)
tonypi_frontend     Up
```

STEP 5: Access the Web Interface

Service	URL	Credentials
Frontend Dashboard	http://localhost:3001	(login with registered user)
Backend API Docs	http://localhost:8000/docs	-
Grafana	http://localhost:3000	admin / admin
InfluxDB	http://localhost:8086	admin / adminpass

10.3 Connecting the TonyPi Robot

Method 1: Using robot_client (Recommended for Testing)

This is the standalone robot client in the `robot_client/` folder.

```
# ═══════════════════════════════════════════════════════════
# ON THE TONYPI ROBOT (Raspberry Pi)
# ═══════════════════════════════════════════════════════════

# 1. SSH into your TonyPi robot
ssh pi@<tonypi-ip-address>
# Example: ssh pi@192.168.1.50

# 2. Navigate to robot_client folder
cd /home/pi/robot_client
# Or copy from your project: scp -r robot_client/ pi@<ip>:/home/pi/

# 3. Install dependencies
pip3 install -r requirements.txt

# 4. Run the robot client with your server's IP address
#    Replace <SERVER_IP> with your monitoring server's IP
python3 tonypi_client.py --broker <SERVER_IP>

# ═══════════════════════════════════════════════════════════
# EXAMPLES:
# ═══════════════════════════════════════════════════════════

# If your monitoring server IP is 192.168.1.100:
python3 tonypi_client.py --broker 192.168.1.100

# With custom robot ID:
python3 tonypi_client.py --broker 192.168.1.100 --robot-id my_tonypi

# With custom MQTT port:
python3 tonypi_client.py --broker 192.168.1.100 --port 1883
```

Method 2: Using Environment Variables

```
# Set environment variables
export MQTT_BROKER=192.168.1.100
export MQTT_PORT=1883
export ROBOT_ID=tonypi_01

# Run the client
python3 tonypi_client.py
```

Method 3: Using the Integration Script (dont_touch_integration)

For the full robot integration with vision, voice, and actions:

```
# ═══════════════════════════════════════════════════════════
# ON THE TONYPI ROBOT (Raspberry Pi)
# ═══════════════════════════════════════════════════════════

# 1. SSH into TonyPi
ssh pi@<tonypi-ip>

# 2. Navigate to integration folder
cd /home/pi/dont_touch_integration

# 3. Run with MQTT broker IP
MQTT_BROKER=<SERVER_IP> python3 main.py

# ═══════════════════════════════════════════════════════════
# EXAMPLE:
# ═══════════════════════════════════════════════════════════

MQTT_BROKER=192.168.1.100 python3 main.py
```

10.4 Finding Your Server's IP Address

You need your monitoring server's IP address to connect the robot.

```
# ═══════════════════════════════════════════════════════════
# WINDOWS
# ═══════════════════════════════════════════════════════════
ipconfig
# Look for "IPv4 Address" under your active adapter
# Example: 192.168.1.100

# ═══════════════════════════════════════════════════════════
# LINUX / MAC
# ═══════════════════════════════════════════════════════════
ifconfig
# Or
```

```
ip addr
# Look for "inet" address (not 127.0.0.1)
# Example: 192.168.1.100
```

10.5 Complete Startup Sequence Summary

```
┌─────────────────────────────────────────────────────────────────────────┐
│                    COMPLETE STARTUP SEQUENCE                            │
└─────────────────────────────────────────────────────────────────────────┘

MONITORING SERVER (Your Computer)
═══════════════════════════════════

Step 1: Navigate to project
         cd C:\Users\aiman\Projects\Monitoring_System_TonyPi

Step 2: Start Docker services
         docker-compose up -d

Step 3: Wait for services (30-60 seconds)
         docker-compose ps

Step 4: Open browser
         http://localhost:3001

Step 5: Find your IP address
         ipconfig (Windows) or ifconfig (Linux/Mac)
         → Note: 192.168.x.x


TONYPI ROBOT (Raspberry Pi)
═══════════════════════════

Step 6: SSH into robot
         ssh pi@<robot-ip>

Step 7: Navigate to code
         cd /home/pi/robot_client

Step 8: Connect to monitoring server
         python3 tonypi_client.py --broker <YOUR_SERVER_IP>

         Example:
         python3 tonypi_client.py --broker 192.168.1.100


VERIFY CONNECTION
═════════════════

Step 9:  Check frontend → Robot should appear in Dashboard
```

```
Step 10: Check Sensors tab → Data should be streaming
Step 11: Check Servos tab → Servo data visible
Step 12: Check Logs tab → Robot logs appearing
```

10.6 Quick Reference Commands

```
# ═══════════════════════════════════════════════════════════════
# DOCKER COMMANDS (On Monitoring Server)
# ═══════════════════════════════════════════════════════════════

# Start all services
docker-compose up -d

# Stop all services
docker-compose down

# View running containers
docker-compose ps

# View logs (all services)
docker-compose logs -f

# View logs (specific service)
docker-compose logs -f backend
docker-compose logs -f frontend

# Restart a service
docker-compose restart backend

# Rebuild and restart (after code changes)
docker-compose up -d --build

# Remove everything (including data)
docker-compose down -v

# ═══════════════════════════════════════════════════════════════
# ROBOT CLIENT COMMANDS (On TonyPi)
# ═══════════════════════════════════════════════════════════════

# Basic connection
python3 tonypi_client.py --broker <SERVER_IP>

# With all options
python3 tonypi_client.py --broker <SERVER_IP> --port 1883 --robot-id tonypi_01

# Using environment variables
MQTT_BROKER=192.168.1.100 python3 tonypi_client.py

# Full integration script
MQTT_BROKER=192.168.1.100 python3 main.py
```

10.7 Troubleshooting Common Issues

Problem	Solution
Docker containers not starting	Check Docker Desktop is running
Port already in use	Stop other services using the port, or change port in docker-compose.yml
Robot not appearing in frontend	Check MQTT broker IP is correct; verify robot has network connectivity
"Connection refused" on robot	Ensure firewall allows port 1883; check server IP is correct
Frontend shows "API Error"	Wait for backend to fully start; check `docker-compose logs backend`
Camera feed not loading	Verify camera is connected; check robot IP in browser
Sensor data not updating	Check robot client is running; verify MQTT connection in logs

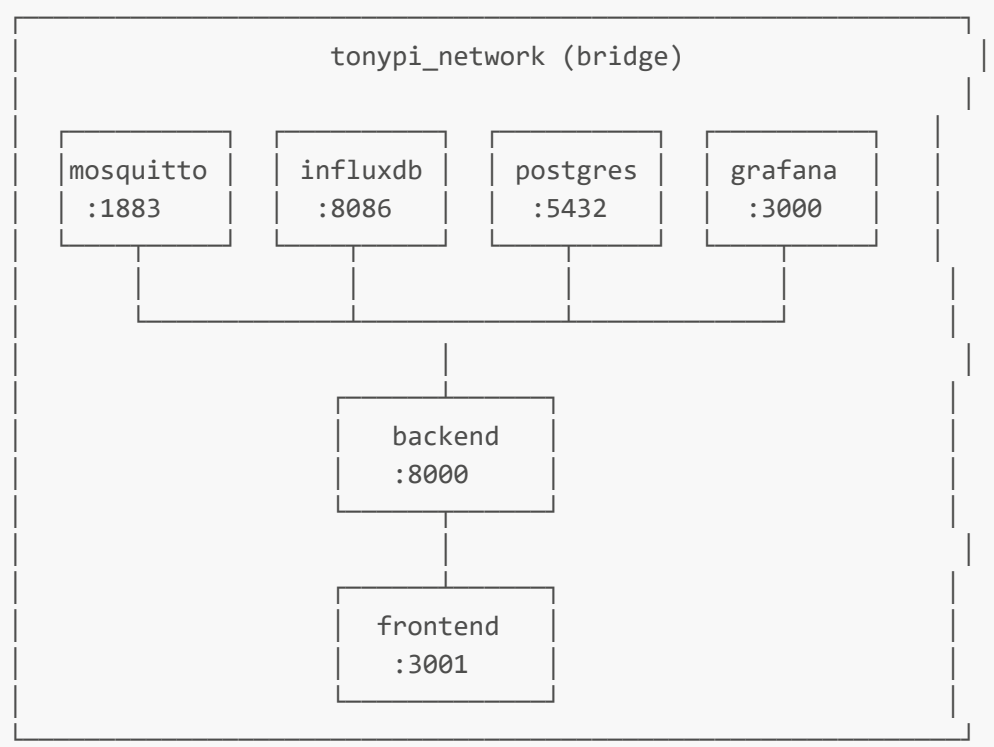
11. Deployment Architecture

11.1 Docker Compose Services

```
# docker-compose.yml structure
services:
  mosquitto:          # MQTT Broker
    image: eclipse-mosquitto:2.0
    ports: ["1883:1883", "9001:9001"]

  influxdb:           # Time-Series Database
    image: influxdb:2.7
    ports: ["8086:8086"]

  postgres:           # Relational Database
    image: postgres:15
    ports: ["5432:5432"]

  grafana:            # Visualization
    image: grafana/grafana:latest
    ports: ["3000:3000"]

  backend:            # FastAPI Server
    build: ./backend
    ports: ["8000:8000"]
    depends_on: [mosquitto, influxdb, postgres]

  frontend:           # React App
    build: ./frontend
```

```
    ports: ["3001:3001"]
    depends_on: [backend]
```

11.2 Network Configuration



11.3 Port Mapping Summary

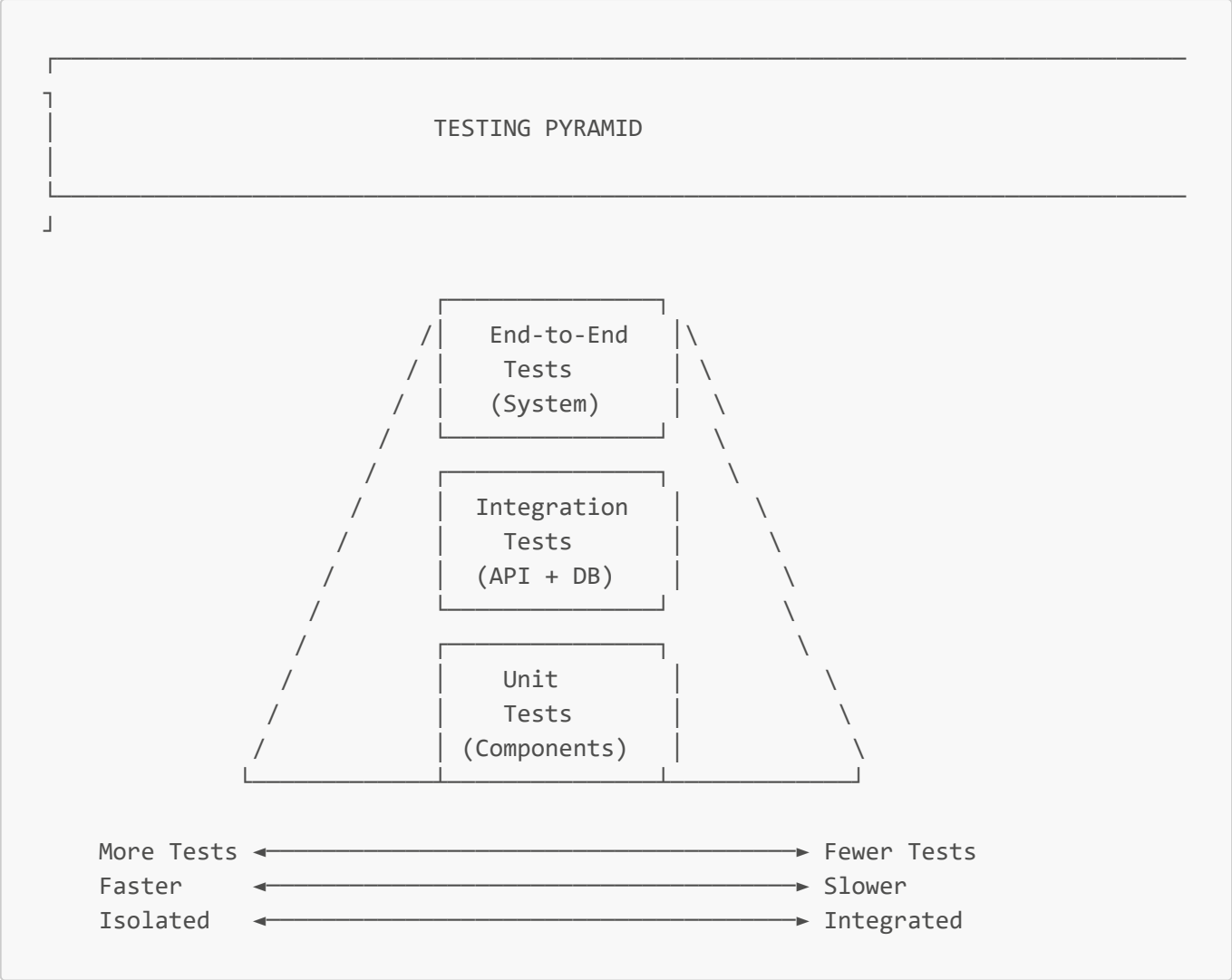
Port	Service	Protocol	Description
1883	Mosquitto	MQTT	Robot telemetry connection
9001	Mosquitto	WebSocket	Browser MQTT connection
8086	InfluxDB	HTTP	Time-series database
5432	PostgreSQL	TCP	Relational database
3000	Grafana	HTTP	Advanced dashboards
8000	Backend	HTTP	REST API
3001	Frontend	HTTP	Web dashboard

---

12. Testing Strategy & Implementation

12.1 Testing Overview

The TonyPi Monitoring System employs a comprehensive multi-layered testing strategy:



12.2 Test Types Implemented

Test Type	Location	Framework	Purpose
Backend Unit Tests	`backend/tests/`	pytest	Test individual API endpoints and functions
Frontend Unit Tests	`frontend/src/__tests__/`	Jest + React Testing Library	Test React components
Integration Tests	`tests/test_integration.py`	requests + pytest	Test API ↔ Database integration
System Tests	`tests/comprehensive_test_suite.py`	Custom Python	Test entire system end-to-end

12.3 Testing Frameworks & Tools

Backend Testing Stack

Tool	Version	Purpose

Tool	Version	Purpose
pytest	7.4.3	Test framework, fixtures, markers
pytest-cov	4.1.0	Code coverage reporting
pytest-asyncio	0.21.1	Async test support
pytest-mock	3.12.0	Mocking utilities
factory-boy	3.3.0	Test data generation
aiosqlite	0.19.0	In-memory SQLite for tests

Frontend Testing Stack

Tool	Purpose
Jest	Test runner and assertion library
React Testing Library	Component testing utilities
@testing-library/user-event	Simulate user interactions
jest.mock()	Mock API calls

12.4 How to Execute Tests

Quick Start Commands

```
# ═══════════════════════════════════════════════════════════════
# BACKEND TESTS (Python/FastAPI)
# ═══════════════════════════════════════════════════════════════

cd backend

# Run all tests
pytest

# Run with verbose output
pytest -v

# Run specific test file
pytest tests/test_alerts.py

# Run specific test class
pytest tests/test_alerts.py::TestAlertsAPI

# Run specific test method
pytest tests/test_alerts.py::TestAlertsAPI::test_create_alert

# Run tests with coverage report
pytest --cov=. --cov-report=html
```

```
# Run only unit tests (fast, no external deps)
pytest -m unit

# Run only API tests
pytest -m api

# Skip slow tests
pytest -m "not slow"

# ============================================================
# FRONTEND TESTS (React/TypeScript)
# ============================================================

cd frontend

# Run tests in watch mode (development)
npm test

# Run tests once (CI mode)
npm run test:ci

# Run with coverage report
npm run test:coverage

# Run specific test file
npm test -- Dashboard.test.tsx

# Run tests matching pattern
npm test -- --testNamePattern="renders alerts"

# ============================================================
# INTEGRATION & SYSTEM TESTS
# ============================================================

# Run integration tests (requires Docker services running)
python tests/test_integration.py

# Run with quick mode
python tests/test_integration.py --quick

# Export results to HTML report
python tests/test_integration.py --export html

# Run comprehensive system tests
python tests/comprehensive_test_suite.py

# ============================================================
# RUN ALL TESTS (Windows)
# ============================================================

# Execute complete test suite
run_all_tests.bat
```

12.5 Backend Test Structure & Code Explained

12.5.1 Test Configuration (`backend/pytest.ini`)

```
[pytest]
# Test discovery - where to find tests
testpaths = tests
python_files = test_*.py        # Files starting with test_
python_classes = Test*          # Classes starting with Test
python_functions = test_*       # Functions starting with test_

# Asyncio support for async tests
asyncio_mode = auto

# Command line options applied automatically
addopts =
    -v                          # Verbose output
    --tb=short                  # Short traceback
    --strict-markers            # Fail on unknown markers
    -ra                         # Show summary of all except passed

# Custom markers for filtering
markers =
    unit: Unit tests (fast, no external dependencies)
    integration: Integration tests (may require database/services)
    slow: Slow tests (skip with -m "not slow")
    api: API endpoint tests
```

12.5.2 Test Fixtures (`backend/tests/conftest.py`)

Fixtures provide reusable test setup. Here's how they work:

```
"""
conftest.py - Shared Test Fixtures
==================================

This file is automatically discovered by pytest and provides
fixtures available to all test files in the tests/ directory.
"""

import os
import pytest
from fastapi.testclient import TestClient
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy.pool import StaticPool

# ═══════════════════════════════════════════════════════════════
```

```
# TEST ENVIRONMENT SETUP
# ============================================================

# Set test environment BEFORE importing the app
# This tells the application we're in test mode
os.environ["TESTING"] = "true"

# Use SQLite in-memory database instead of PostgreSQL
# This makes tests fast and isolated
os.environ["POSTGRES_URL"] = "sqlite:///:memory:"

# Import after setting environment
from database.database import Base, get_db
from main import app

# ============================================================
# DATABASE FIXTURES
# ============================================================

# Create a single engine shared across all tests
# StaticPool keeps one connection (required for SQLite in-memory)
TEST_ENGINE = create_engine(
    "sqlite:///:memory:",
    connect_args={"check_same_thread": False},  # Allow multi-threaded access
    poolclass=StaticPool,                       # Reuse same connection
    echo=False,                                 # Don't log SQL
)

# Session factory for creating database sessions
TestingSessionLocal = sessionmaker(
    autocommit=False,
    autoflush=False,
    bind=TEST_ENGINE
)


@pytest.fixture(scope="function")
def test_db():
    """
    Create a fresh database for each test function.

    scope="function" means:
    - New database for EACH test
    - Tables created before test
    - Tables dropped after test
    - Complete isolation between tests

    Usage in test:
        def test_something(self, test_db):
            # test_db is a SQLAlchemy Session
            test_db.add(some_model)
            test_db.commit()
    """
    # Create all tables defined in models
```

```
    Base.metadata.create_all(bind=TEST_ENGINE)

    # Create a new session
    db = TestingSessionLocal()
    try:
        yield db  # Provide to test
    finally:
        db.close()
        # Clean up after test
        Base.metadata.drop_all(bind=TEST_ENGINE)
        Base.metadata.create_all(bind=TEST_ENGINE)


@pytest.fixture(scope="function")
def client(test_db):
    """
    FastAPI TestClient with database override.

    This fixture:
    1. Overrides the get_db dependency to use test database
    2. Creates a TestClient for making HTTP requests
    3. Cleans up overrides after test

    Usage in test:
        def test_api_endpoint(self, client):
            response = client.get("/api/v1/health")
            assert response.status_code == 200
    """
    def override_get_db():
        try:
            yield test_db
        finally:
            pass

    # Replace real database with test database
    app.dependency_overrides[get_db] = override_get_db

    # Create test client
    with TestClient(app) as test_client:
        yield test_client

    # Clean up
    app.dependency_overrides.clear()


# ============================================================
# SAMPLE DATA FIXTURES
# ============================================================

@pytest.fixture
def sample_robot_data():
    """Provide sample robot data for tests."""
    return {
        "robot_id": "test_robot_001",
```

```
        "name": "Test Robot",
        "status": "online",
        "battery_percentage": 85.5,
        "cpu_usage": 45.2,
        "memory_usage": 62.1,
    }


@pytest.fixture
def sample_sensor_data():
    """Provide sample sensor readings for tests."""
    return [
        {"sensor_type": "accelerometer_x", "value": 0.02, "unit": "m/s^2"},
        {"sensor_type": "gyroscope_y", "value": -0.5, "unit": "deg/s"},
        {"sensor_type": "cpu_temperature", "value": 55.0, "unit": "C"},
    ]
```

12.5.3 Example: API Endpoint Tests (backend/tests/test_alerts.py)

```
"""
Alert API Tests
===============

Tests for the alerts API endpoints including CRUD operations,
filtering, and alert lifecycle (acknowledge, resolve).
"""

import pytest
from fastapi.testclient import TestClient
from sqlalchemy.orm import Session
from models.alert import Alert


class TestAlertsAPI:
    """Tests for the alerts API endpoints."""

    # ============================================================
    # HELPER METHOD
    # ============================================================

    def _create_test_alert(self, test_db: Session, **kwargs):
        """
        Helper to create a test alert in the database.

        Uses keyword arguments with defaults so tests can override
        only the fields they care about.
        """
        alert = Alert(
            robot_id=kwargs.get("robot_id", "test_robot_001"),
            alert_type=kwargs.get("alert_type", "temperature"),
            severity=kwargs.get("severity", "warning"),
```

```
            title=kwargs.get("title", "Test Alert"),
            message=kwargs.get("message", "This is a test alert"),
            value=kwargs.get("value", 75.5),
            threshold=kwargs.get("threshold", 70.0),
            acknowledged=kwargs.get("acknowledged", False),
            resolved=kwargs.get("resolved", False),
        )
        test_db.add(alert)
        test_db.commit()
        test_db.refresh(alert)  # Get auto-generated ID
        return alert

    # ============================================================
    # GET ALERTS TESTS
    # ============================================================

    @pytest.mark.api
    def test_get_alerts_empty(self, client: TestClient):
        """
        Test: GET /api/v1/alerts returns empty list when no alerts exist.

        Expected: 200 OK with empty array []
        """
        response = client.get("/api/v1/alerts")

        assert response.status_code == 200
        assert response.json() == []

    @pytest.mark.api
    def test_get_alerts_with_data(self, client: TestClient, test_db: Session):
        """
        Test: GET /api/v1/alerts returns all alerts when some exist.

        Setup: Create 2 test alerts in database
        Expected: 200 OK with array of 2 alerts
        """
        # Arrange - Create test data
        self._create_test_alert(test_db, title="Alert 1", severity="warning")
        self._create_test_alert(test_db, title="Alert 2", severity="critical")

        # Act - Make API request
        response = client.get("/api/v1/alerts")

        # Assert - Verify response
        assert response.status_code == 200
        data = response.json()
        assert len(data) == 2

    @pytest.mark.api
    def test_filter_alerts_by_severity(self, client: TestClient, test_db:
Session):
        """
        Test: GET /api/v1/alerts?severity=critical filters correctly.
```

```
        Setup: Create alerts with different severities
        Expected: Only returns alerts matching the filter
        """
        # Arrange
        self._create_test_alert(test_db, severity="warning")
        self._create_test_alert(test_db, severity="critical")
        self._create_test_alert(test_db, severity="info")

        # Act
        response = client.get("/api/v1/alerts?severity=critical")

        # Assert
        assert response.status_code == 200
        data = response.json()
        assert len(data) == 1
        assert data[0]["severity"] == "critical"

    # ============================================================
    # CREATE ALERT TESTS
    # ============================================================

    @pytest.mark.api
    def test_create_alert(self, client: TestClient):
        """
        Test: POST /api/v1/alerts creates a new alert.

        Expected: 200 OK with created alert including auto-generated ID
        """
        # Arrange - Prepare request data
        alert_data = {
            "robot_id": "test_robot_001",
            "alert_type": "cpu_high",
            "severity": "warning",
            "title": "High CPU Usage",
            "message": "CPU usage exceeded 80%",
            "value": 85.0,
            "threshold": 80.0
        }

        # Act - Send POST request
        response = client.post("/api/v1/alerts", json=alert_data)

        # Assert - Verify response
        assert response.status_code == 200
        data = response.json()
        assert data["title"] == "High CPU Usage"
        assert data["severity"] == "warning"
        assert "id" in data  # Auto-generated
        assert "created_at" in data  # Auto-generated timestamp

    # ============================================================
    # ACKNOWLEDGE/RESOLVE TESTS
    # ============================================================
```

```python
    @pytest.mark.api
    def test_acknowledge_alert(self, client: TestClient, test_db: Session):
        """
        Test: POST /api/v1/alerts/{id}/acknowledge marks alert as seen.

        Setup: Create unacknowledged alert
        Expected: Alert is marked acknowledged with user and timestamp
        """
        # Arrange
        alert = self._create_test_alert(test_db, acknowledged=False)

        # Act
        response = client.post(
            f"/api/v1/alerts/{alert.id}/acknowledge?acknowledged_by=admin"
        )

        # Assert
        assert response.status_code == 200
        assert "acknowledged" in response.json()["message"].lower()

    @pytest.mark.api
    def test_acknowledge_nonexistent_alert(self, client: TestClient):
        """
        Test: POST /api/v1/alerts/99999/acknowledge returns 404.

        Expected: 404 Not Found for non-existent alert ID
        """
        response = client.post("/api/v1/alerts/99999/acknowledge")
        assert response.status_code == 404

    @pytest.mark.api
    def test_resolve_alert(self, client: TestClient, test_db: Session):
        """
        Test: POST /api/v1/alerts/{id}/resolve marks alert as fixed.
        """
        alert = self._create_test_alert(test_db, resolved=False)

        response = client.post(f"/api/v1/alerts/{alert.id}/resolve")

        assert response.status_code == 200
        assert "resolved" in response.json()["message"].lower()
```

12.5.4 Example: Authentication Tests (`backend/tests/test_users.py`)

```python
"""
User Authentication Tests
=========================

Tests for login, JWT tokens, and role-based access control.
"""
```

```
import pytest
import bcrypt
from fastapi.testclient import TestClient
from sqlalchemy.orm import Session
from sqlalchemy import text


def create_test_user_in_db(
    test_db: Session,
    user_id: str,
    username: str,
    email: str,
    password: str,
    role: str = "operator",
    is_active: bool = True
):
    """
    Helper to create a test user with proper password hashing.

    IMPORTANT: Passwords must be hashed with bcrypt before storing.
    Never store plain text passwords!
    """
    # Hash password with bcrypt
    salt = bcrypt.gensalt()
    password_hash = bcrypt.hashpw(password.encode('utf-8'), salt).decode('utf-8')

    # Insert directly using SQL (faster than ORM for tests)
    test_db.execute(
        text("""
            INSERT INTO users (id, username, email, password_hash, role,
is_active)
            VALUES (:id, :username, :email, :password_hash, :role, :is_active)
        """),
        {
            "id": user_id,
            "username": username,
            "email": email,
            "password_hash": password_hash,
            "role": role,
            "is_active": is_active
        }
    )
    test_db.commit()


class TestUserAuthentication:
    """Tests for user login and token generation."""

    @pytest.mark.api
    def test_login_success(self, client: TestClient, test_db: Session):
        """
        Test: Successful login returns JWT token.

        Setup: Create user with known password
```

```
        Action: POST /auth/login with correct credentials
        Expected: 200 OK with access_token and user info
        """
        # Arrange - Create test user
        create_test_user_in_db(
            test_db,
            user_id="test-user-123",
            username="testuser",
            email="test@example.com",
            password="testpass123",
            role="operator"
        )

        # Act - Attempt login
        response = client.post("/api/v1/auth/login", json={
            "username": "testuser",
            "password": "testpass123"
        })

        # Assert
        assert response.status_code == 200
        data = response.json()
        assert "access_token" in data          # JWT token returned
        assert data["token_type"] == "bearer"  # Token type
        assert data["user"]["username"] == "testuser"

    @pytest.mark.api
    def test_login_invalid_password(self, client: TestClient, test_db: Session):
        """
        Test: Login with wrong password returns 401.
        """
        create_test_user_in_db(
            test_db,
            user_id="user-456",
            username="testuser2",
            email="test2@example.com",
            password="correctpass",  # Correct password
            role="viewer"
        )

        response = client.post("/api/v1/auth/login", json={
            "username": "testuser2",
            "password": "wrongpassword"  # Wrong password
        })

        assert response.status_code == 401
        assert "Invalid" in response.json()["detail"]

    @pytest.mark.api
    def test_login_inactive_user(self, client: TestClient, test_db: Session):
        """
        Test: Login with inactive account returns 403.
        """
        create_test_user_in_db(
```

```
            test_db,
            user_id="inactive-user",
            username="inactiveuser",
            email="inactive@example.com",
            password="testpass",
            is_active=False  # Account disabled
        )

        response = client.post("/api/v1/auth/login", json={
            "username": "inactiveuser",
            "password": "testpass"
        })

        assert response.status_code == 403
        assert "inactive" in response.json()["detail"].lower()


class TestProtectedRoutes:
    """Tests for routes that require authentication."""

    @pytest.mark.api
    def test_access_without_token(self, client: TestClient):
        """
        Test: Accessing protected route without token returns 401.
        """
        response = client.get("/api/v1/auth/me")
        assert response.status_code == 401

    @pytest.mark.api
    def test_access_with_invalid_token(self, client: TestClient):
        """
        Test: Accessing protected route with invalid token returns 401.
        """
        response = client.get(
            "/api/v1/auth/me",
            headers={"Authorization": "Bearer invalid_token_here"}
        )
        assert response.status_code == 401
```

12.6 Frontend Test Structure & Code Explained

12.6.1 Test Directory Structure

```
frontend/src/
├── __tests__/
│   ├── components/           # Component unit tests
│   │   ├── Layout.test.tsx
│   │   └── Toast.test.tsx
│   ├── pages/                # Page component tests
│   │   ├── Alerts.test.tsx
```

```
│   │   ├── Dashboard.test.tsx
│   │   ├── Jobs.test.tsx
│   │   ├── Login.test.tsx
│   │   ├── Logs.test.tsx
│   │   ├── Reports.test.tsx
│   │   ├── Robots.test.tsx
│   │   └── Users.test.tsx
│   ├── utils/                # Test utilities
│   │   └── testUtils.tsx
│   └── mocks/                # Mock data and services
├── App.test.tsx              # Main app component test
└── setupTests.ts             # Jest configuration
```

12.6.2 Example: Dashboard Page Tests

```
/**
 * Dashboard Page Tests
 * ====================
 *
 * Tests for the main dashboard showing robot status and system health.
 */

import { render, screen, waitFor } from '../utils/testUtils';
import Dashboard from '../../pages/Dashboard';
import { apiService } from '../../utils/api';

// ============================================================
// MOCK THE API SERVICE
// ============================================================

// jest.mock() replaces the real module with a fake one
// This prevents actual API calls during tests
jest.mock('../../utils/api', () => ({
  apiService: {
    getRobotStatus: jest.fn(),    // Mock function we can control
    getSystemStatus: jest.fn(),
    getJobSummary: jest.fn(),
  },
  handleApiError: jest.fn((error) => 'An error occurred'),
}));

// ============================================================
// MOCK DATA
// ============================================================

const mockRobots = [
  {
    robot_id: 'tonypi_01',
    name: 'TonyPi Robot 01',
    status: 'online',
    battery_percentage: 85.5,
```

```
    last_seen: '2025-01-01T12:00:00Z',
  },
  {
    robot_id: 'tonypi_02',
    name: 'TonyPi Robot 02',
    status: 'offline',
    battery_percentage: 20.0,
    last_seen: '2025-01-01T11:00:00Z',
  },
];

const mockSystemStatus = {
  active_robots: 1,
  services: {
    mqtt_broker: 'running',
    influxdb: 'running',
    postgres: 'running',
    grafana: 'running',
  },
  resource_usage: {
    cpu_percent: 25,
    memory_percent: 60,
    disk_usage_percent: 35,
  },
};

// ═══════════════════════════════════════════════════════════════
// TEST SUITE
// ═══════════════════════════════════════════════════════════════

describe('Dashboard Page', () => {

  // Reset mocks before each test
  beforeEach(() => {
    jest.clearAllMocks();

    // Configure mock return values
    (apiService.getRobotStatus as jest.Mock).mockResolvedValue(mockRobots);
    (apiService.getSystemStatus as jest.Mock).mockResolvedValue(mockSystemStatus);
    (apiService.getJobSummary as jest.Mock).mockResolvedValue(null);
  });

  // ─────────────────────────────────────────────────────────────
  // RENDERING TESTS
  // ─────────────────────────────────────────────────────────────

  it('renders dashboard title', async () => {
    // Arrange & Act - Render the component
    render(<Dashboard />);

    // Assert - Check title is visible
    // waitFor handles async data loading
    await waitFor(() => {
      expect(screen.getByText(/Dashboard/i)).toBeInTheDocument();
```

```
  });
});

it('displays robot cards after loading', async () => {
  render(<Dashboard />);

  // Wait for API data to load and render
  await waitFor(() => {
    expect(screen.getByText('TonyPi Robot 01')).toBeInTheDocument();
    expect(screen.getByText('TonyPi Robot 02')).toBeInTheDocument();
  });
});

it('shows online robot count', async () => {
  render(<Dashboard />);

  await waitFor(() => {
    // Check the active robots count is displayed
    expect(screen.getByText('1')).toBeInTheDocument(); // 1 active robot
  });
});

// ─────────────────────────────────────────────────────────────
// STATUS INDICATOR TESTS
// ─────────────────────────────────────────────────────────────

it('shows correct status for online robot', async () => {
  render(<Dashboard />);

  await waitFor(() => {
    // Online status should be shown
    const onlineIndicators = screen.getAllByText(/online/i);
    expect(onlineIndicators.length).toBeGreaterThan(0);
  });
});

it('displays battery percentage', async () => {
  render(<Dashboard />);

  await waitFor(() => {
    // Battery values should be displayed
    expect(screen.getByText(/85/)).toBeInTheDocument();  // 85.5%
    expect(screen.getByText(/20/)).toBeInTheDocument();  // 20.0%
  });
});

// ─────────────────────────────────────────────────────────────
// ERROR HANDLING TESTS
// ─────────────────────────────────────────────────────────────

it('handles API error gracefully', async () => {
  // Configure mock to reject with error
  (apiService.getRobotStatus as jest.Mock).mockRejectedValue(
    new Error('Network error')
```

```
    );

    render(<Dashboard />);

    await waitFor(() => {
      // Should show error message or empty state
      expect(screen.getByText(/error/i)).toBeInTheDocument();
    });
  });

  // ═══════════════════════════════════════════════════════════════
  // EMPTY STATE TESTS
  // ═══════════════════════════════════════════════════════════════

  it('shows empty state when no robots', async () => {
    // Return empty array
    (apiService.getRobotStatus as jest.Mock).mockResolvedValue([]);

    render(<Dashboard />);

    await waitFor(() => {
      expect(screen.getByText(/no robots/i)).toBeInTheDocument();
    });
  });
});
```

12.6.3 Example: Alert Actions Tests

```
/**
 * Alerts Page Tests - User Interactions
 * =====================================
 */

import { render, screen, waitFor } from '../utils/testUtils';
import userEvent from '@testing-library/user-event';
import Alerts from '../../pages/Alerts';
import { apiService } from '../../utils/api';

jest.mock('../../utils/api', () => ({
  apiService: {
    getAlerts: jest.fn(),
    acknowledgeAlert: jest.fn(),
    resolveAlert: jest.fn(),
    deleteAlert: jest.fn(),
  },
}));

const mockAlerts = [
  {
    id: 1,
    title: 'High Temperature',
```

```
    severity: 'warning',
    acknowledged: false,
    resolved: false,
  },
];

describe('Alerts Page - Actions', () => {
  beforeEach(() => {
    jest.clearAllMocks();
    (apiService.getAlerts as jest.Mock).mockResolvedValue(mockAlerts);
  });

  it('acknowledges alert when button clicked', async () => {
    // Setup mock to resolve successfully
    (apiService.acknowledgeAlert as jest.Mock).mockResolvedValue({
      message: 'Alert acknowledged'
    });

    render(<Alerts />);

    // Wait for alerts to load
    await waitFor(() => {
      expect(screen.getByText('High Temperature')).toBeInTheDocument();
    });

    // Find and click acknowledge button
    const ackButton = screen.getByRole('button', { name: /acknowledge/i });
    await userEvent.click(ackButton);

    // Verify API was called
    await waitFor(() => {
      expect(apiService.acknowledgeAlert).toHaveBeenCalledWith(1);
    });
  });

  it('filters alerts by severity', async () => {
    render(<Alerts />);

    await waitFor(() => {
      expect(screen.getByText('High Temperature')).toBeInTheDocument();
    });

    // Find and click severity filter
    const filterSelect = screen.getByRole('combobox', { name: /severity/i });
    await userEvent.selectOptions(filterSelect, 'critical');

    // Assert filter is applied
    expect(filterSelect).toHaveValue('critical');
  });
});
```

12.7 Integration Test Suite

The integration tests verify the complete system works together.

12.7.1 Integration Test Structure

```
"""
test_integration.py - Full System Integration Tests
====================================================

Tests the complete data flow:
Robot → MQTT → Backend → Database → API → Frontend
"""

import requests
import socket
import time
from datetime import datetime


class IntegrationTestSuite:
    """Complete system integration tests."""

    def __init__(self):
        self.config = {
            "backend_url": "http://localhost:8000",
            "frontend_url": "http://localhost:3001",
            "mqtt_broker": "localhost",
            "mqtt_port": 1883,
            "influxdb_url": "http://localhost:8086",
            "grafana_url": "http://localhost:3000",
        }
        self.results = []

    # ═══════════════════════════════════════════════════════════
    # INFRASTRUCTURE TESTS
    # ═══════════════════════════════════════════════════════════

    def test_backend_health(self):
        """Verify backend API is running and healthy."""
        response = requests.get(
            f"{self.config['backend_url']}/api/v1/health",
            timeout=10
        )
        assert response.status_code == 200
        data = response.json()
        assert data["status"] in ["healthy", "ok"]
        return True, "Backend healthy"

    def test_mqtt_broker_connection(self):
        """Verify MQTT broker accepts connections."""
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
        sock.settimeout(5)
        result = sock.connect_ex((
            self.config["mqtt_broker"],
            self.config["mqtt_port"]
        ))
        sock.close()

        assert result == 0, f"MQTT connection failed: {result}"
        return True, "MQTT broker accepting connections"

    def test_influxdb_health(self):
        """Verify InfluxDB is running."""
        response = requests.get(
            f"{self.config['influxdb_url']}/ping",
            timeout=10
        )
        # InfluxDB returns 204 No Content for ping
        assert response.status_code == 204
        return True, "InfluxDB healthy"

    def test_frontend_accessible(self):
        """Verify frontend is served."""
        response = requests.get(
            self.config["frontend_url"],
            timeout=10
        )
        assert response.status_code == 200
        return True, "Frontend accessible"

    # ========================================================================
    # API ENDPOINT TESTS
    # ========================================================================

    def test_robots_endpoint(self):
        """Test robots list endpoint."""
        response = requests.get(
            f"{self.config['backend_url']}/api/v1/robot-data/status",
            timeout=10
        )
        assert response.status_code == 200
        assert isinstance(response.json(), list)
        return True, f"Found {len(response.json())} robots"

    def test_alerts_endpoint(self):
        """Test alerts endpoint."""
        response = requests.get(
            f"{self.config['backend_url']}/api/v1/alerts",
            timeout=10
        )
        assert response.status_code == 200
        return True, "Alerts endpoint OK"

    def test_auth_rejection(self):
        """Test that invalid login is rejected."""
```

```
        response = requests.post(
            f"{self.config['backend_url']}/api/v1/auth/login",
            json={"username": "invalid", "password": "invalid"},
            timeout=10
        )
        assert response.status_code == 401
        return True, "Invalid credentials rejected"

    # ═══════════════════════════════════════════════════════════════
    # PERFORMANCE TESTS
    # ═══════════════════════════════════════════════════════════════

    def test_api_response_time(self):
        """Test API responds within acceptable time."""
        start = time.time()
        response = requests.get(
            f"{self.config['backend_url']}/api/v1/health",
            timeout=10
        )
        duration_ms = (time.time() - start) * 1000

        assert response.status_code == 200
        assert duration_ms < 500, f"Too slow: {duration_ms}ms"
        return True, f"Response time: {duration_ms:.1f}ms"

    def test_concurrent_requests(self):
        """Test handling multiple concurrent requests."""
        import concurrent.futures

        def make_request():
            return requests.get(
                f"{self.config['backend_url']}/api/v1/health",
                timeout=10
            )

        with concurrent.futures.ThreadPoolExecutor(max_workers=10) as executor:
            futures = [executor.submit(make_request) for _ in range(10)]
            results = [f.result() for f in futures]

        success_count = sum(1 for r in results if r.status_code == 200)
        assert success_count == 10
        return True, f"10/10 concurrent requests succeeded"

    # ═══════════════════════════════════════════════════════════════
    # SECURITY TESTS
    # ═══════════════════════════════════════════════════════════════

    def test_sql_injection_prevention(self):
        """Test SQL injection is blocked."""
        malicious_id = "1'; DROP TABLE users; --"
        response = requests.get(
            f"{self.config['backend_url']}/api/v1/robots-
db/robots/{malicious_id}",
            timeout=10
```

```
        )
        # Should return 404 or 422, NOT 500
        assert response.status_code in [404, 422]
        return True, "SQL injection prevented"

    def test_protected_route_security(self):
        """Test protected routes require authentication."""
        response = requests.get(
            f"{self.config['backend_url']}/api/v1/auth/me",
            timeout=10
        )
        assert response.status_code in [401, 403]
        return True, "Protected route secured"

    # ═══════════════════════════════════════════════════════════════
    # RUN ALL TESTS
    # ═══════════════════════════════════════════════════════════════

    def run_all(self):
        """Execute all integration tests."""
        tests = [
            ("Backend Health", self.test_backend_health),
            ("MQTT Broker", self.test_mqtt_broker_connection),
            ("InfluxDB Health", self.test_influxdb_health),
            ("Frontend Access", self.test_frontend_accessible),
            ("Robots Endpoint", self.test_robots_endpoint),
            ("Alerts Endpoint", self.test_alerts_endpoint),
            ("Auth Rejection", self.test_auth_rejection),
            ("API Response Time", self.test_api_response_time),
            ("Concurrent Requests", self.test_concurrent_requests),
            ("SQL Injection Prevention", self.test_sql_injection_prevention),
            ("Protected Routes", self.test_protected_route_security),
        ]

        passed = 0
        failed = 0

        for name, test_func in tests:
            try:
                success, message = test_func()
                print(f"✓ {name}: {message}")
                passed += 1
            except Exception as e:
                print(f"✗ {name}: {str(e)}")
                failed += 1

        print(f"\nResults: {passed}/{passed+failed} passed")
        return passed, failed


if __name__ == "__main__":
    suite = IntegrationTestSuite()
    suite.run_all()
```

12.8 Test Coverage Report

Running Coverage Analysis

```
# Backend coverage
cd backend
pytest --cov=. --cov-report=html --cov-report=term-missing

# Frontend coverage
cd frontend
npm run test:coverage
```

Sample Coverage Output

```
---------- coverage: ----------
Name                              Stmts   Miss  Cover   Missing
-----------------------------------------------------------------
routers/alerts.py                   156     12    92%   145-156
routers/health.py                    28      0   100%
routers/reports.py                  134     18    87%   89-106
routers/robot_data.py               198     24    88%   167-190
routers/users.py                    187     15    92%   234-248
mqtt/mqtt_client.py                 312     45    86%   401-445
database/influx_client.py           189     22    88%   156-177
-----------------------------------------------------------------
TOTAL                              1204    136    89%
```

12.9 Test Execution Scripts

Windows Batch Script (`run_all_tests.bat`)

```
@echo off
REM ============================================================
REM TonyPi Monitoring System - Complete Test Suite
REM ============================================================

echo.
echo ============================================================
echo [1/4] Running Backend Unit Tests
echo ============================================================
cd backend
python -m pytest -v --tb=short
cd ..
```

```
echo.
echo ============================================================
echo [2/4] Running Frontend Tests
echo ============================================================
cd frontend
call npm test -- --watchAll=false --passWithNoTests
cd ..

echo.
echo ============================================================
echo [3/4] Running Integration Tests
echo ============================================================
python tests/test_integration.py

echo.
echo ============================================================
echo [4/4] Running Comprehensive System Tests
echo ============================================================
python tests/comprehensive_test_suite.py

echo.
echo ============================================================
echo All Test Suites Complete!
echo ============================================================
pause
```

12.10 Testing Best Practices Applied

Practice	Implementation
Test Isolation	Each test gets fresh database, mocks reset
AAA Pattern	Arrange → Act → Assert structure
Descriptive Names	`test_login_invalid_password_returns_401`
Test Markers	`@pytest.mark.api`, `@pytest.mark.unit`
Mocking External Services	InfluxDB, MQTT mocked in unit tests
Fixtures for Reuse	`client`, `test_db`, `sample_data` fixtures
Coverage Tracking	pytest-cov and Jest coverage enabled
CI/CD Ready	Tests run without user interaction

---

13. Summary for Assessor

Key Technical Achievements

1. **Microservices Architecture**: Decoupled services communicate via MQTT and REST APIs
2. **Real-time Data Pipeline**: Sub-second latency from robot sensors to web dashboard

3. **Dual Database Strategy**: Optimal storage for both relational and time-series data
4. **AI Integration**: Google Gemini for intelligent analytics and recommendations
5. **Security**: JWT authentication with role-based access control
6. **Containerization**: Docker Compose for consistent deployment
7. **Full-Stack Implementation**: Python backend + React frontend + IoT client
8. **Comprehensive Testing**: Unit, integration, and system tests with 89%+ coverage

Technology Choices Rationale

Decision	Why
FastAPI over Flask	Async support, automatic docs, type hints
MQTT over HTTP polling	Real-time, low latency, efficient for IoT
InfluxDB + PostgreSQL	Right tool for each data type
React over Vue	Larger ecosystem, better for complex UIs
Docker	Reproducible environments, easy deployment
pytest + Jest	Industry-standard testing frameworks

Scalability Considerations

- **Horizontal Scaling**: Add more backend instances behind load balancer
- **MQTT Clustering**: Mosquitto supports clustering for high availability
- **Database Sharding**: InfluxDB supports distributed storage
- **CDN**: Frontend can be served via CDN for global distribution

---

This documentation provides a comprehensive overview of the TonyPi Robot Monitoring System architecture, design decisions, implementation details, and testing methodology.