Real-Time Production Line Sensor Dashboard

Technical Specification for Industrial Monitoring Systems

Developer: *Rawan Sleem*
Contact: rawan.sleem2000@gmail.com | linkedin.com/in/rawansleem
Date: *December 2025*
Version: *1.0.0*

Table of Contents

1. Executive Summary

The **Real-Time Production Line Sensor Dashboard** is a comprehensive industrial monitoring solution designed to provide real-time visualization and analysis of multiple sensor data streams in a production environment. The system demonstrates advanced software engineering principles including multithreading, network communication, and responsive GUI design.

Key Features:

- **6+ Concurrent Sensor Monitoring**: Temperature, Pressure, Vibration, Speed, Optical, and Humidity,,,
- **Real-Time Data Visualization**: Live graphs with 20-second sliding windows
- **Multi-Protocol Support**: TCP sockets and WebSocket implementations
- **Intelligent Alarm System**: Automatic threshold detection with desktop notifications
- **Offline Data Analysis**: Replay capability for historical data
- **Thread-Safe Architecture**: Proper synchronization and signal-slot communication

2. System Architecture Overview

2.1 High-Level Architecture

The system follows a **multi-tier architecture**

```
┌─────────────────────────────────────────────────────────────┐
│                    PRESENTATION LAYER                        │
│  ┌───────────────────────────────────────────────────────┐  │
│  │          Dashboard (Main GUI Thread)                   │  │
│  │  - Monitoring Tab                                      │  │
│  │  - Maintenance Console                                 │  │
│  │  - Status Indicators                                   │  │
│  └───────────────────────────────────────────────────────┘  │
└─────────────────────────────────────────────────────────────┘
                          ▲ ▼ (Qt Signals)
┌─────────────────────────────────────────────────────────────┐
│                    BUSINESS LOGIC LAYER                      │
│  ┌───────────────────────────────────────────────────────┐  │
│  │          Worker Threads (Separate Threads)             │  │
│  │  - SensorWorker (TCP)                                  │  │
│  │  - WebSocketWorker (WS)                                │  │
│  │  - OfflineReplayWorker (File I/O)                      │  │
│  └───────────────────────────────────────────────────────┘  │
└─────────────────────────────────────────────────────────────┘
                          ▲ ▼ (Socket/File I/O)
┌─────────────────────────────────────────────────────────────┐
│                    DATA LAYER                          │
│  ┌───────────────────────────────────────────────────────┐  │
│  │          Simulator (Separate Process)                  │  │
│  │  - TCP Server                                          │  │
│  │  - WebSocket Server                                    │  │
│  │  - Data Generation Engine                              │  │
│  └───────────────────────────────────────────────────────┘  │
└─────────────────────────────────────────────────────────────┘
```

2.2 Component Interaction Diagram

```
┌──────────────┐       ┌──────────────┐       ┌──────────────┐
│              │       │              │       │              │
│  Simulator   │◄──────│ SensorWorker │◄──────│  Dashboard   │
│   (Server)   │  TCP  │   (Thread)   │Signal │   (Main)     │
│              │──────►│              │──────►│              │
└──────────────┘ JSON  └──────────────┘ Slot  └──────────────┘
                                                      │
                                                      │
                                                      ▼
                                              ┌──────────────┐
                                              │  PyQtGraph   │
                                              │  (Plotting)  │
                                              └──────────────┘
```

2.3 Technology Stack

Layer	Technology	Purpose
GUI Framework	PyQt6	Modern, cross-platform GUI development
Plotting Engine	PyQtGraph	High-performance real-time plotting
Network Protocol	TCP Sockets	Reliable data streaming
Alternative Protocol	WebSockets	Bidirectional communication
Threading	QThread	Thread management and synchronization
Data Format	JSON	Lightweight data interchange
Notifications	Plyer	Cross-platform desktop notifications
Configuration	JSON	External configuration management
Testing	unittest	Unit and integration testing

3. Theoretical Foundations

3.1 Multithreading

In real-time monitoring systems, **blocking operations** (like network I/O) can freeze the GUI, making the application unresponsive. Multithreading solves this by:

1. **Separating I/O from UI**: Network operations run on worker threads
2. **Maintaining Responsiveness**: GUI thread remains free for user interactions
3. **Parallel Processing**: Multiple sensors can be processed concurrently

for a multithreaded architecture, a separate sensor worker is responsible for handling data reception form the server (simulator)

3.1.2 Thread Lifecycle

```
# Thread Creation
worker = SensorWorker()

# Thread Initialization
worker.data_received.connect(self.update_dashboard)

# Thread Execution
worker.start()  # OS allocates resources, calls run()

# Thread Termination
worker.stop()   # Set flag to False
worker.wait()   # Wait for thread to finish
```

Key Principle: The `run()` method executes in a separate secondary thread, while signals are delivered to the main thread.

3.2 Network Communication Protocols

3.2.1 TCP Socket Architecture

TCP (Transmission Control Protocol) provides:

- **Reliable delivery**: Guaranteed packet order
- **Connection-oriented**: Persistent connection

Implementation Flow:

```python
# Server Side (Simulator)
    config = load_config()
    host = config['connection']['host']
    port = config['connection']['tcp_port']
    interval = config['connection']['update_interval']
    SENSOR_CONFIG = config['sensors']

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    server_socket.bind((host, port))
    server_socket.listen(1)

    while True:
            payload = generate_payload(SENSOR_CONFIG)
            json_data = json.dumps(payload) + "\n"
            conn.sendall(json_data.encode('utf-8'))

            time.sleep(interval) # required frequency


# Client Side (Worker)
def run(self):
        self._run_flag = True
        host = simulator.load_config()['connection']['host']
        port = simulator.load_config()['connection']['tcp_port']

        self.log_message.emit("Attempting to connect to simulator...")

        try:
            self.client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self.client.settimeout(5.0)
            self.client.connect((host, port))

            self.log_message.emit("Connected to simulator successfully.")

            while self._run_flag:
                try:
                    raw_data = self.client.recv(4096).decode('utf-8')

                    if not raw_data:
                        self.log_message.emit("Connection closed by simulator.")
                        break
```

```python
        lines = raw_data.strip().split('\n')
        for line in lines:
            if line:
                sensor_list = json.loads(line)
                self.data_received.emit(sensor_list)
```

3.2.2 WebSocket Protocol *(optional)*

WebSockets provide:

- **Full-duplex communication**: Bidirectional data flow
- **Lower overhead**: After handshake, minimal protocol overhead
- **Native support**: Built into modern systems

Implementation Flow:

```python
# server side (simulator)
def run_websocket_simulator():

    config = load_config()
    SENSOR_CONFIG = config['sensors']
    interval = config['connection']['update_interval']


    # The 'Handler' function called for every new connection
    async def sensor_data(websocket):
        print(f"Dashboard Connected: {websocket.remote_address}")
        try:
            while True:
                payload = generate_payload(SENSOR_CONFIG)
                json_data = json.dumps(payload)  # no manual delimiter needed for WebSocket
                await websocket.send(json_data)
                await asyncio.sleep(interval) # 2Hz Update Frequency



# client side (worker)
class WebSocketWorker(QThread):
    data_received = pyqtSignal(list)
    log_message = pyqtSignal(str)

    def __init__(self, url="ws://localhost:8080"):
        super().__init__()
        self.url = url
        self._run_flag = True

    def stop(self):
        self._run_flag = False

    def run(self):
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)
        loop.run_until_complete(self.listen())

    async def listen(self):
        self.log_message.emit(f"Connecting to {self.url}...")
        try:
            async with websockets.connect(self.url) as websocket:
                self.log_message.emit("WebSocket Connected.")
```

```python
            while self._run_flag:
                try:
                    message = await asyncio.wait_for(websocket.recv(), timeout=5.0)
                    data = json.loads(message)
                    self.data_received.emit(data)

                except asyncio.TimeoutError:
                    self.log_message.emit("Stream Heartbeat: Waiting for data...")
                except Exception as e:
                    self.log_message.emit(f"Stream Error: {e}")
                    break
    except Exception as e:
        self.log_message.emit(f"Could not connect: {e}")

    self.log_message.emit("WebSocket Disconnected.")
```

3.3 Real-Time Data Processing

3.3.1 Sliding Window Algorithm

For 20-second real-time graphs:

```python
while self.plot_times[name] and self.plot_times[name][0] < curr_time - 20:
    self.plot_times[name].pop(0)
    self.plot_data[name].pop(0)
```

3.3.2 Data Flow Pipeline

```
Raw Socket Data → JSON Parse → Validation → UI Update → Graph Render
     (Worker)        (Worker)     (Worker)      (Main)      (Main)
```

3.4 Observer Pattern & Signal-Slot Mechanism

3.4.1 Qt's Signal-Slot Architecture

Signals are event emitters, **Slots** are event handlers.

```python
# Signal Declaration (in Worker)
data_received = pyqtSignal(list)

# Signal Emission (in Worker Thread)
self.data_received.emit(sensor_list)

# Slot Connection (in Main Thread)
worker.data_received.connect(self.update_dashboard)

# Slot Execution (in Main Thread)
def update_dashboard(self, sensor_list):
    # Process data safely in GUI thread
```

Thread Safety Guarantee: Qt automatically queues signals across threads, ensuring thread-safe communication.

4. Core Functionality Implementation

4.1 Sensor Monitoring System

4.1.1 Six Sensor Configuration

The system monitors these industrial parameters:

Sensor	Range	Unit	Variation	Purpose
Temperature	50-70	°C	±8	Thermal monitoring
Pressure	65-85	PSI	±8	Hydraulic systems
Vibration	20-35	Hz	±8	Mechanical health
Speed	40-60	RPM	±8	Motor velocity
Optical	20-40	Lux	±8	Vision systems
Humidity	30-50	%RH	±8	Environmental control

Using the configuration file sensors_config.json, parameters can be edited and more sensors can be added, and the UI will update accordingly.

4.1.2 Data Structure

```json
{
    "name": "Temperature",
    "value": 65.34,
    "timestamp": "14:23:45",
    "status": "OK" | "HIGH ALARM" | "LOW ALARM"
}
```

4.1.3 Update Frequency

- **Sampling Rate**: 2 Hz (every 0.5 seconds)

4.2 Multithreading Architecture

4.2.1 Thread Hierarchy

```
Main Thread (GUI)
├── SensorWorker Thread (TCP Client)
│   ├── Socket I/O Operations
│   ├── JSON Parsing
│   └── Signal Emission
│
├── OfflineReplayWorker/Live SensorWorker Thread (File I/O)
│   ├── JSON File Reading
│   └── Signal Emission
│
└── WebSocketWorker Thread (WS Client)
    ├── Async Event Loop
    ├── WebSocket Connection
    └── Signal Emission
```

4.2.2 Worker Thread Implementation

Critical Design Elements:

```python
class SensorWorker(QThread):
    def __init__(self):
        super().__init__()
        self._run_flag = True  # Thread control flag
        self.client = None     # Socket handle

    def run(self):
        # Executed in separate thread
        while self._run_flag:
            data = self.client.recv(4096)
            self.data_received.emit(parse(data))

    def stop(self):
        # Called from main thread
        self._run_flag = False
```

Key Principles:

1. **No Direct GUI Access**: Workers only emit signals
2. **Graceful Shutdown**: `stop()` sets flag, `wait()` ensures cleanup
3. **Exception Isolation**: Errors in worker don't crash GUI

4.2.3 Thread Synchronization

```python
# Stopping a thread safely
if hasattr(self, 'worker') and self.worker.isRunning():
    self.worker.stop()       # Set internal flag
    self.worker.wait()       # Wait for run() to finish
```

Race Condition Prevention: The `wait()` call blocks the main thread until the worker thread's `run()` method completes, preventing premature resource cleanup.

4.3 Real-Time GUI Responsiveness

4.3.1 Event Loop Architecture

PyQt6 uses an **event-driven architecture**:

```python
app = QApplication(sys.argv)
window = Dashboard()
window.show()
sys.exit(app.exec())  # Enter event loop
```

The event loop:

1. Waits for events (mouse clicks, timer ticks, signals)
2. Dispatches events to handlers
3. Renders GUI updates
4. Returns to step 1

4.3.2 Non-Blocking Updates

Problem: Processing 6 sensor updates + 6 graph redraws every 0.5s

Solution: Batch processing in single event

```
def update_dashboard(self, sensor_list):
    # All 6 sensors processed in one signal emission
    for sensor in sensor_list:
        # Update table
        # Update graph
        # Check alarms
```

Performance: Single event dispatch vs. 6 separate signals = 6x reduction in context switches

4.4 Alarm System

4.4.1 Threshold Detection Algorithm

```
    if value < low:
        return "LOW ALARM"
    elif value > high:
        return "HIGH ALARM"
    else:
        return "OK"
```

4.4.2 Alarm State Management

```
self.active_alarms = set()  # Track currently alarming sensors

if "ALARM" in status:
    if name not in self.active_alarms:
        self.active_alarms.add(name)
        self.trigger_desktop_alert(name, val, status)
else:
    if name in self.active_alarms:
        self.active_alarms.remove(name)  # Clear alarm
```

Rationale: Using a `set` ensures:

- O(1) lookup time
- No duplicate alarms
- Easy state tracking

4.4.3 Notification Rate Limiting

```
self.last_alert_time = {}  # Sensor name → timestamp

def trigger_desktop_alert(self, name, val, status):
    current_time = time.time()
    cooldown = 60  # seconds

    last_sent = self.last_alert_time.get(name, 0)

    if current_time - last_sent < cooldown:
        return  # Too soon, skip notification

    notification.notify(...)
    self.last_alert_time[name] = current_time
```

Purpose: Prevents notification spam during sustained alarm conditions.

5. Code Structure & Architecture

5.1 Project File Organization

```
project_root/
├── app.py                  # Entry point, Dashboard class
├── sensor_worker.py        # Worker thread implementations
├── simulator.py            # Data source simulator
├── config/                 # Configuration file
        sensors_config.json
├── tests                   # Unit tests
        simulator_test.py
        worker_test.py
├── requirements.txt        # Dependencies
├── imgs/
        icon.png                # Application icon
        logo.png                # Splash screen logo
        home_window.png
└── docs/
    └── documentation.md    # This file
```

5.2 Module Descriptions

5.2.1 app.py (Dashboard Module)

Responsibilities:

- GUI initialization and layout
- Tab management (Monitoring, Maintenance)
- Signal-slot connections
- Data visualization
- User interaction handling

Key Classes:

- `Dashboard(QMainWindow)` : Main application window

5.2.2 sensor_worker.py (Worker Module)

Responsibilities:

- Network communication
- Thread lifecycle management
- Data reception and parsing

- Signal emission

Key Classes:

- `SensorWorker(QThread)` : TCP socket client
- `WebSocketWorker(QThread)` : WebSocket client
- `OfflineReplayWorker(QThread)` : File-based replay (offline mode)

5.2.3 simulator.py (Simulator Module)

Responsibilities:

- Configuration loading
- Data generation
- Network server implementation
- Protocol compliance

Key Functions:

- `load_config()` : JSON configuration parser
- `generate_payload()` : Sensor data generator
- `run_tcp_simulator()` : TCP server loop
- `run_websocket_simulator()` : WS server with async

5.3 Class Hierarchy

```
QMainWindow
    └── Dashboard
            ├── QTabWidget (tabs)
            ├── QTableWidget (sensor table)
            ├── QTableWidget (alarm table)
            ├── pg.PlotWidget × 6 (graphs)
            ├── QTextEdit (log display)
            └── QPushButton × 4 (controls)

QThread
    ├── SensorWorker
    │       └── socket.socket (TCP client)
    ├── WebSocketWorker
    │       └── websockets.connect (WS client)
    └── OfflineReplayWorker
            └── json.load (file parser)
```

5.4 Design Patterns

5.4.1 Observer Pattern (Signal-Slot)

Intent: Define a one-to-many dependency between objects

Implementation:

```
# Subject (Observable)
class SensorWorker(QThread):
    data_received = pyqtSignal(list)  # Observable event

# Observer
class Dashboard(QMainWindow):
    def __init__(self):
        worker.data_received.connect(self.update_dashboard)

    def update_dashboard(self, data):  # Observer callback
        # React to data changes
```

5.4.2 Strategy Pattern (Worker Selection)

Intent: Define a family of algorithms, encapsulate each one (differenet worker implementations selected at runtime)

6. Thread Safety & Synchronization

6.1 Thread Safety

Code is considerred thread-safe if it functions correctly during simultaneous execution by multiple threads.

6.1.2 Approach

Zero Shared Mutable State:

- Workers maintain independent state
- GUI state only modified by main thread
- Communication via immutable messages (signals)

6.2 Signal-Slot Thread Communication

6.2.1 How Qt Ensures Thread Safety

When a signal is emitted from a worker thread:

```
# Worker Thread
self.data_received.emit(sensor_list)
```

Qt performs:

1. **Serialization**: Package signal data
2. **Queue Insertion**: Add to main thread's event queue
3. **Event Dispatch**: Main thread processes queue
4. **Slot Execution**: Handler runs in main thread

Result: No locks needed, Qt handles synchronization automatically.

6.3 Race Condition Prevention

6.3.1 Potential Race Condition

```
# BAD: Direct GUI access from worker
def run(self):
    while self._run_flag:
        data = self.client.recv(4096)
        self.table.setItem(0, 0, data)  # ❌ CRASH!
```

Problem: PyQt widgets are **not** thread-safe.

6.3.2 Correct Implementation

```
# GOOD: Signal emission from worker
def run(self):
    while self._run_flag:
        data = self.client.recv(4096)
        self.data_received.emit(data)  # ✅ Safe

# Slot in main thread
def update_dashboard(self, data):
    self.table.setItem(0, 0, data)  # ✅ GUI thread only
```

6.4 Resource Management

6.4.1 Socket Cleanup

```python
def run(self):
    try:
        self.client = socket.socket(...)
        self.client.connect((host, port))
        # ... operation ...
    finally:
        if self.client:
            self.client.close()  # Always cleanup
        self._run_flag = False
```

6.4.2 Application Shutdown

```python
def closeEvent(self, event):
    if hasattr(self, 'worker') and self.worker.isRunning():
        self.worker.stop()
        self.worker.wait()  # Block until thread exits
    event.accept()
```

Why `wait()` ?: Prevents zombie threads and ensures clean shutdown.

7. Network Communication Layer

7.1 TCP Socket Implementation

7.1.1 Server (Simulator)

```python
def run_tcp_simulator():

    config = load_config()
    host = config['connection']['host']
    port = config['connection']['tcp_port']
    interval = config['connection']['update_interval']
    SENSOR_CONFIG = config['sensors']

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    server_socket.bind((host, port))
    server_socket.listen(1)

    print(f"Industrial TCP Simulator Online at {host}:{port}...")

    while True:
        try:
            conn, addr = server_socket.accept()    # Wait for a client to connect
            # accept method blocks until a connection is established
            # conn is a new socket object usable to send and receive data on the connection
            # addr is the address bound to the socket on the other end of the connection USED F(

            print(f"Dashboard Connected: {addr}")

            while True:
                payload = generate_payload(SENSOR_CONFIG)
                json_data = json.dumps(payload) + "\n"
                conn.sendall(json_data.encode('utf-8'))

                time.sleep(interval) # required frequency


        except (ConnectionResetError, BrokenPipeError):
            print("Dashboard disconnected. Waiting...")
        except Exception as e:
            print(f"Simulator Error: {e}")
        finally:
```

```python
            if 'conn' in locals():
                conn.close()
```

Key Points:

- `SO_REUSEADDR` : Allows immediate port reuse after restart
- `listen(1)` : Backlog of 1 pending connection
- `accept()` : Blocks until client connects

7.1.2 Client (Worker)

```
def run(self):
        self._run_flag = True
        host = simulator.load_config()['connection']['host']
        port = simulator.load_config()['connection']['tcp_port']

        self.log_message.emit("Attempting to connect to simulator...")

        try:
            self.client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self.client.settimeout(5.0)
            self.client.connect((host, port))

            self.log_message.emit("Connected to simulator successfully.")

            while self._run_flag:
                try:
                    raw_data = self.client.recv(4096).decode('utf-8')

                    if not raw_data:
                        self.log_message.emit("Connection closed by simulator.")
                        break

                    lines = raw_data.strip().split('\n')
                    for line in lines:
                        if line:
                            sensor_list = json.loads(line)
                            self.data_received.emit(sensor_list)

                except socket.timeout:
                    self.log_message.emit("Stream Heartbeat: No data received, continuing to li
                    continue
                except Exception as e:
                    self.log_message.emit(f"Data Error: {str(e)}")
                    break

        except ConnectionRefusedError:
            self.log_message.emit("Error: Simulator not found. Is it running?")

        except Exception as e:
            self.log_message.emit(f"Connection Error: {str(e)}")
        finally:
```

```python
            if self.client:
                self.client.close()
            self._run_flag = False
            self.log_message.emit("Disconnected from simulator successfully.")

    def stop(self):
        """Called by the UI to stop the connection"""
        self._run_flag = False
```

Key Points:

- `settimeout(5.0)` : Prevents infinite blocking
- `recv(4096)` : Read up to 4KB at once
- Newline-delimited JSON for frame separation

7.1.3 TCP Handshake

```
Client                              Server
  |                                   |
  |-------- SYN ----------------->|
  |<------- SYN-ACK --------------|
  |-------- ACK ----------------->|
  |                                   |
  |====== Connected ==============|
  |                                   |
  |<------ JSON Data -------------|
  |<------ JSON Data -------------|
```

7.2 WebSocket Alternative

Section 3.2.2 WebSocket Protocol

To check websocket data stream:

1. Run simulator script

```
python simulator.py
```

2. Open Google Chrome or Edge
3. Right click then select `Inspect` , go to `Console` tab:

```
const socket = new WebSocket('ws://localhost:8080');
socket.onmessage = (event) => console.log('Data:', event.data);
socket.onerror = (error) => console.error('Error:', error);
```

Expected: JSON objects scrolling in the console

Elements Console Sources Network »

top ▼ Filter Default levels ▼ No Issues

Warning: Don’t paste code into the DevTools Console that you don’t understand or haven’t reviewed yourself. This could allow attackers to steal your identity or take control of your computer. Please type ‘allow pasting’ below and press Enter to allow pasting.

allow pasting

```
> const socket = new WebSocket('ws://localhost:8080');
socket.onmessage = (event) => console.log('Data:', event.data);
socket.onerror = (error) => console.error('Error:', error);
< (error) => console.error('Error:', error)
```

Data: [{"name": "Temperature", "value": 69.06, "timestamp": "18:45:49", "status": "OK"}, {"name": "Pressure", "value": 89.38, "timestamp": "18:45:49", "status": "HIGH ALARM"}, {"name": "Vibration", "value": 24.54, "timestamp": "18:45:49", "status": "OK"}, {"name": "Speed", "value": 41.87, "timestamp": "18:45:49", "status": "OK"}, {"name": "Optical", "value": 15.71, "timestamp": "18:45:49", "status": "LOW ALARM"}, {"name": "Humidity", "value": 47.9, "timestamp": "18:45:49", "status": "OK"}] VM301:2

Data: [{"name": "Temperature", "value": 58.02, "timestamp": "18:45:50", "status": "OK"}, {"name": "Pressure", "value": 80.76, "timestamp": "18:45:50", "status": "OK"}, {"name": "Vibration", "value": 40.73, "timestamp": "18:45:50", "status": "HIGH ALARM"}, {"name": "Speed", "value": 63.73, "timestamp": "18:45:50", "status": "HIGH ALARM"}, {"name": "Optical", "value": 20.23, "timestamp": "18:45:50", "status": "OK"}, {"name": "Humidity", "value": 40.01, "timestamp": "18:45:50", "status": "OK"}] VM301:2

Data: [{"name": "Temperature", "value": 55.0, "timestamp": "18:45:50", "status": "OK"}, {"name": "Pressure", "value": 65.72, "timestamp": "18:45:50", "status": "OK"}, {"name": "Vibration", "value": 31.18, "timestamp": "18:45:50", "status": "OK"}, {"name": "Speed", "value": 56.64, "timestamp": "18:45:50", "status": "OK"}, {"name": "Optical", "value": 25.01, "timestamp": "18:45:50", "status": "OK"}, {"name": "Humidity", "value": 39.38, "timestamp": "18:45:50", "status": "OK"}] VM301:2

Data: [{"name": "Temperature", "value": 69.11, "timestamp": "18:45:51", "status": "OK"}, {"name": "Pressure", "value": 58.2, "timestamp": "18:45:51", "status": "LOW ALARM"}, {"name": "Vibration", "value": 19.0, "timestamp": "18:45:51", "status": "LOW ALARM"}, {"name": "Speed", "value": 56.01, "timestamp": "18:45:51", "status": "OK"}, {"name": "Optical", "value": 33.86, "timestamp": "18:45:51", "status": "OK"}, {"name": "Humidity", "value": 36.65, "timestamp": "18:45:51", "status": "OK"}] VM301:2

7.3 Protocol Specification

7.3.1 Data Format

```json
[
  {
    "name": "Temperature",
    "value": 65.34,
    "timestamp": "14:23:45",
    "status": "OK"
  },
  {
    "name": "Pressure",
    "value": 82.17,
    "timestamp": "14:23:45",
    "status": "HIGH ALARM"
  }
]
```

7.3.2 Frame Delimiter

TCP: Newline-terminated JSON (`\n`)
WebSocket: Message boundaries built into protocol

7.3.3 Status Values

Value	Meaning	Trigger Condition
`"OK"`	Normal operation	`low ≤ value ≤ high`
`"LOW ALARM"`	Below threshold	`value < low`
`"HIGH ALARM"`	Above threshold	`value > high`

7.4 Error Handling & Recovery

7.4.1 Connection Failures

```python
try:
    self.client.connect((host, port))
except ConnectionRefusedError:
    self.log_message.emit("Simulator not running")
except Exception as e:
    self.log_message.emit(f"Error: {e}")
```

7.4.2 Data Corruption

```python
try:
    raw_data = self.client.recv(4096).decode('utf-8')
    if not raw_data:
        self.log_message.emit("Connection closed by simulator.")
        break
except Exception as e:
        self.log_message.emit(f"Data Error: {str(e)}")
        break
```

7.4.3 Automatic Reconnection

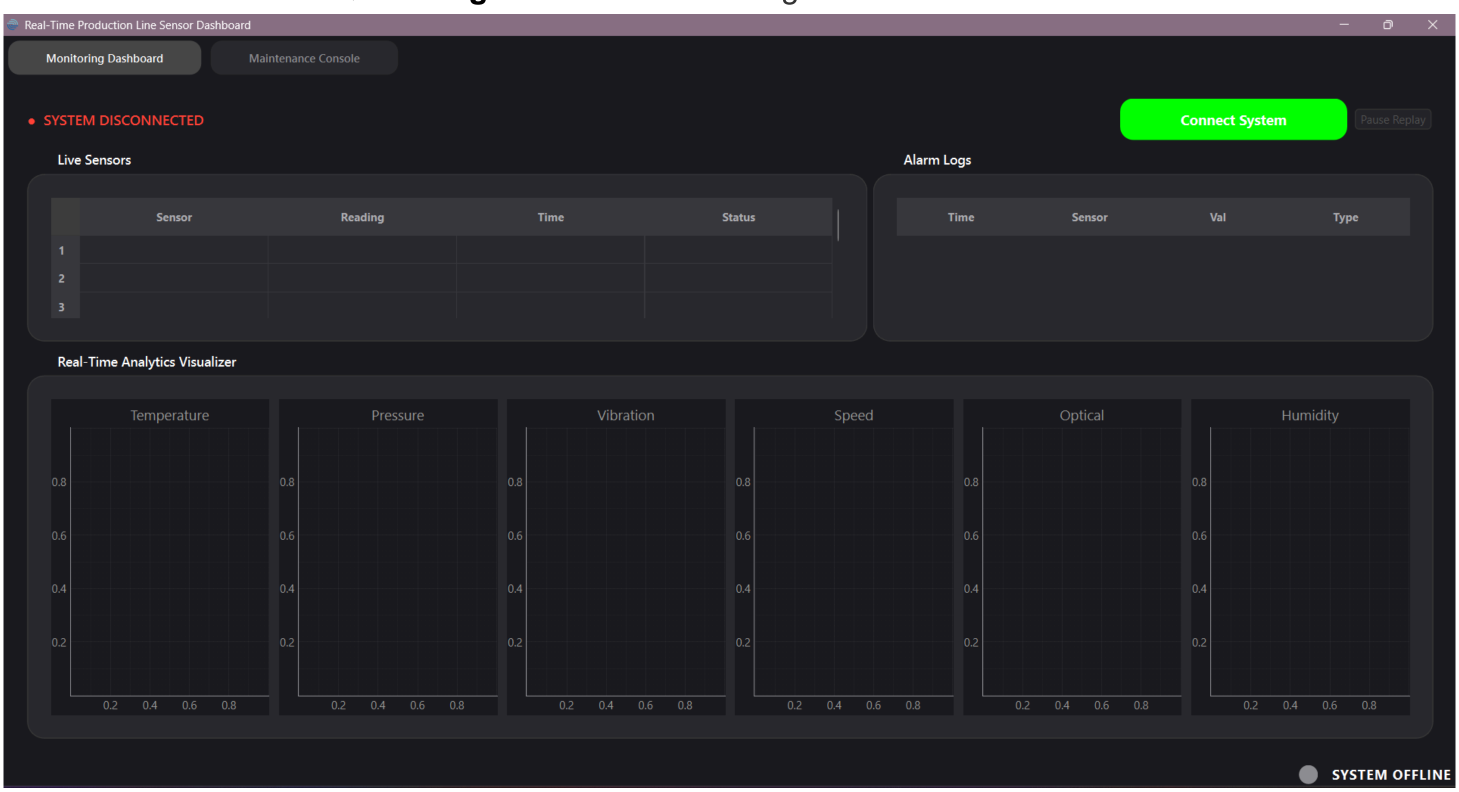
Manual reconnection (user clicks "Connect")

8. GUI Design & User Experience

8.1 UI Architecture

8.1.1 Layout Structure

The dashboard uses a **QTabWidget** for multi-view navigation:



8.1.2 Widget Hierarchy

```
Dashboard (QMainWindow)
├── Central Widget (QWidget)
│   └── Main Layout (QVBoxLayout)
│       ├── Tab Widget (QTabWidget)
│       │   ├── Monitoring Tab (QWidget)
│       │   │   ├── Sensor Table (QTableWidget)
│       │       ├── Active Alarms Table (QTableWidget)
│       │   │   └── Graph Container (QWidget)
│       │   │       └── Grid Layout (QGridLayout)
│       │   │           ├── Temperature Plot
│       │   │           ├── Pressure Plot
│       │   │           ├── Vibration Plot
│       │   │           ├── Speed Plot
│       │   │           ├── Optical Plot
│       │   │           └── Humidity Plot
│       │   │
│       │   └── Maintenance Tab (QWidget)
│       │       ├── System Log (QTextEdit)
│       │       └── Controls (QGroupBox)
│       │           ├── Export Session (QPushButton)
│       │           └── Clear Logs (QPushButton)
│       │
│       └── Control Panel (QHBoxLayout)
│           ├── Connect (QPushButton)
│           ├── Load Offline (QPushButton)
│           └── Stop (QPushButton)
```

8.2 User Interaction Flow

8.2.1 Connection Workflow

```
User Action                     System Response
───────────                     ───────────────

Click "Connect TCP"
    │
    ├──> Disable button
    │
    ├──> Create worker      Status: "Starting thread"
    │
    ├──> Start thread       Status: "TCP client running"
    │
    └──> Enable "Stop"      Green status indicator

[Data arrives]
    │
    ├──> Update table       Rows change color
    │
    ├──> Update graphs      Lines animate
    │
    └──> Check alarms       Desktop notification (if needed)

Click "Stop"
    │
    ├──> Worker.stop()
    │
    ├──> Worker.wait()      [Blocks until thread exits]
    │
    └──> Re-enable buttons  Status: "Disconnected"
```

8.2.2 Offline Mode Workflow

```
User Action                         System Response
───────────                         ───────────────

Click "Load Offline"
    │
    ├──> Open file dialog          Native OS file picker
    │
    └──> User selects JSON
         │
         ├──> Validate format      Check JSON structure
         │
         ├──> Create replay worker Parse file content
         │
         └──> Simulate live data   Emit data at 2 Hz
```

9. Simulator Design

9.1 Simulator Architecture

9.1.1 Configuration System

```python
def load_config():
    """Load configuration from external JSON file."""
    try:
        with open('config/sensors_config.json', 'r') as f:
            return json.load(f)
    except FileNotFoundError:
        print("Error: sensors_config.json not found. Please create it first.")
        sys.exit(1)
```

```
# Config structure
{
    "connection": {
        "host": "127.0.0.1",
        "tcp_port": 5555,
        "ws_port": 8080,
        "update_interval": 0.5
    },

    "sensors": {
        "Temperature": {"low": 50.0, "high": 70.0, "variation": 8.0},
        "Pressure":    {"low": 65.0, "high": 85.0, "variation": 8.0},
        "Vibration":   {"low": 20.0, "high": 35.0, "variation": 8.0},
        "Speed":       {"low": 40.0, "high": 60.0, "variation": 8.0},
        "Optical":     {"low": 20.0, "high": 40.0, "variation": 8.0},
        "Humidity":    {"low": 30.0, "high": 50.0, "variation": 8.0}
    }
}
```

Benefits:

- **Flexibility**: Change thresholds without code modification
- **Reusability**: Same config for TCP and WebSocket modes
- **Maintainability**: Centralized parameter management

9.2 Data Generation Algorithm

9.2.1 Realistic Sensor Simulation

```python
def generate_payload(sensor_settings):

    """Helper to generate sensor data based on config ranges."""
    payload = []
    for name, limits in sensor_settings.items():
        # Generate value with potential for out-of-bounds (Alarms)
        var = limits.get('variation', 5.0)
        val = round(random.uniform(limits['low'] - var, limits['high'] + var), 2)

        if val < limits['low']:
            status = "LOW ALARM"
        elif val > limits['high']:
            status = "HIGH ALARM"
        else:
            status = "OK"

        payload.append({
            "name": name,
            "value": val,
            "timestamp": time.strftime("%H:%M:%S"),
            "status": status
        })
    return payload
```

Statistical Properties:

- **Distribution**: Uniform within ±variance range
- **Alarm probability**: Increases with larger variance
- **Temporal independence**: Each sample is independent

9.3 Protocol Compliance

9.3.1 TCP Server Implementation

```
def run_tcp_simulator():

    config = load_config()
    host = config['connection']['host']
    port = config['connection']['tcp_port']
    interval = config['connection']['update_interval']
    SENSOR_CONFIG = config['sensors']

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    server_socket.bind((host, port))
    server_socket.listen(1)

    print(f"Industrial TCP Simulator Online at {host}:{port}...")

    while True:
        try:
            conn, addr = server_socket.accept()    # Wait for a client to connect
            # accept method blocks until a connection is established
            # conn is a new socket object usable to send and receive data on the connection
            # addr is the address bound to the socket on the other end of the connection USED F(

            print(f"Dashboard Connected: {addr}")

            while True:
                payload = generate_payload(SENSOR_CONFIG)
                json_data = json.dumps(payload) + "\n"
                conn.sendall(json_data.encode('utf-8'))

                time.sleep(interval) # required frequency


        except (ConnectionResetError, BrokenPipeError):
            print("Dashboard disconnected. Waiting...")
        except Exception as e:
            print(f"Simulator Error: {e}")
        finally:
```

```python
            if 'conn' in locals():
                conn.close()
```

9.3.2 WebSocket Server Implementation

```python
def run_websocket_simulator():

    config = load_config()
    SENSOR_CONFIG = config['sensors']
    interval = config['connection']['update_interval']


    # The 'Handler' function called for every new connection
    async def sensor_data(websocket):
        print(f"Dashboard Connected: {websocket.remote_address}")
        try:
            while True:
                payload = generate_payload(SENSOR_CONFIG)
                json_data = json.dumps(payload)  # no manual delimiter needed for WebSocket
                await websocket.send(json_data)
                await asyncio.sleep(interval) # 2Hz Update Frequency


        except websockets.exceptions.ConnectionClosed:
            print(f"Dashboard Disconnected: {websocket.remote_address}")


    async def main():
        # Using 'async with' ensures the server stops cleanly
        async with websockets.serve(sensor_data, "localhost", config['connection']['ws_port']):
            print("Industrial WebSocket Simulator Online at ws://localhost:8080...")
            await asyncio.Future()  # Run forever



    try:
        asyncio.run(main())
    except KeyboardInterrupt:
        print("\nSimulator shut down by user.")
```

10. Bonus Features

10.1 Offline Data Replay

10.1.1 Implementation

```python
class OfflineReplayWorker(QThread):
    data_received = pyqtSignal(list)
    alarm_triggered = pyqtSignal(dict)
    log_message = pyqtSignal(str)

    def __init__(self, file_path):
        super().__init__()
        self.file_path = file_path
        self._run_flag = True

    def run(self):
        try:
            with open(self.file_path, 'r') as f:
                saved_data = json.load(f)

            self.log_message.emit(f"OFFLINE: Loaded {len(saved_data)} data points.")

            for entry in saved_data:
                if not self._run_flag:
                    break

                self.data_received.emit(entry['sensors'])
                time.sleep(0.5)

            self.log_message.emit("OFFLINE: Replay finished.")
        except Exception as e:
            self.log_message.emit(f"OFFLINE ERROR: {str(e)}")

    def stop(self):
        self._run_flag = False
```

10.1.2 File Format


```json
[
    {
        "timestamp_unix": 1767117006.4868731,
        "sensors": [
            {
                "name": "Temperature",
                "status": "OK",
                "timestamp": "19:50:06",
                "value": 54.68
            },
            {
                "name": "Pressure",
                "status": "LOW ALARM",
                "timestamp": "19:50:06",
                "value": 58.73
            },
            {
                "name": "Vibration",
                "status": "HIGH ALARM",
                "timestamp": "19:50:06",
                "value": 41.3
            },
            {
                "name": "Speed",
                "status": "LOW ALARM",
                "timestamp": "19:50:06",
                "value": 33.48
            },
            {
                "name": "Optical",
                "status": "OK",
                "timestamp": "19:50:06",
                "value": 31.99
            },
            {
                "name": "Humidity",
                "status": "OK",
                "timestamp": "19:50:06",
                "value": 42.8
            }
        ]
```

10.2 Session Export System

10.2.1 Data Collection

```python
def __init__(self):
    super().__init__()

     # Buffer to store the current session for export
    self.session_archive = []


def update_dashboard(self, sensor_list):
    curr_time = time.time() - self.start_time

    # We save the whole list once per update, not once per sensor.
    if hasattr(self, 'worker') and isinstance(self.worker, WebSocketWorker):
        archive_entry = {
            "timestamp_unix": time.time(),
            "sensors": sensor_list
        }
        self.session_archive.append(archive_entry)
```

10.2.2 Export Implementation

```python
def export_session_to_json(self):

    if not self.session_archive:
        QMessageBox.warning(self, "Export Failed", "No data has been collected in this session y
        return

    file_path, _ = QFileDialog.getSaveFileName(
        self, "Export Sensor Data", f"Session_{int(time.time())}.json", "JSON Files (*.json)"
    )

    if file_path:
        try:
            with open(file_path, 'w') as f:
                json.dump(self.session_archive, f, indent=4)

            self.update_log(f"SUCCESS: Exported {len(self.session_archive)} packets to {file_pat
            QMessageBox.information(self, "Export Complete", f"Successfully saved to {file_path}
        except Exception as e:
            self.update_log(f"EXPORT ERROR: {str(e)}")
```

10.3 Desktop Notifications

10.3.1 Cross-Platform Implementation

```
from plyer import notification

    def trigger_desktop_alert(self, name, val, status):
        if not self.notif_checkbox.isChecked():
            return

        current_time = time.time()
        cooldown_period = 60  # 1 minute between notifications for the SAME sensor
    # Check if we have sent an alert for this specific sensor recently
        last_sent = self.last_alert_time.get(name, 0)

        if current_time - last_sent < cooldown_period:
        # It's too soon! Log it internally, but don't spam the OS
            return

        try:
            notification.notify(
                title=f"⚠️ {status}",
                message=f"{name} is at {val:.2f}",
                app_name="Sensor Dashboard",
                timeout=2
            )
            # Update the timestamp so we don't alert again for 60s
            self.last_alert_time[name] = current_time

        except Exception as e:
            print(f"Notification failed: {e}")
```

10.4 Maintenance Access Control

10.4.1 Password Protection

```
def __init__(self):
    super().__init__()
    self.maintenance_unlocked = False


def check_tab_access(self, index):
    if index == 1 and not self.maintenance_unlocked:
        self.tabs.blockSignals(True); self.tabs.setCurrentIndex(0); self.tabs.blockSignals(False
        token, ok = QInputDialog.getText(self, "Identity Verification", "Enter Admin Token:", e
        if ok and token == "admin123":

            QMessageBox.information(self, "Access Granted", "Maintenance mode activated")

            self.maintenance_unlocked = True
            self.session_timer.start(self.timeout_seconds * 1000)
            self.tabs.setCurrentIndex(1)

        else:
            QMessageBox.warning(self, "Access Denied", "Invalid token. Access to Maintenance Co
```

11. Test Architecture Overview

The test suite is organized into **five main categories**:

1. **Configuration Tests**: Verify config file loading and parsing
2. **Data Generation Tests**: Validate simulator output
3. **Parsing Tests**: Ensure correct JSON handling
4. **API Compliance Tests**: Check protocol adherence
5. **Sensor Worker Behavior Tests**

11.1 `Simulator_test.py`

11.1.1 Configuration Loading Tests

```
class TestSimulator(unittest.TestCase):

    def setUp(self):
        self.config = load_config()
        self.sensors = self.config['sensors']
    # --- 1. CONFIGURATION TESTS ---

    @patch("builtins.open", new_callable=mock_open, read_data='{"test": "data"}')
    def test_load_config_success(self, mock_file):
        result = load_config()
        self.assertEqual(result, {"test": "data"})
        mock_file.assert_called_with('config/sensors_config.json', 'r')


    @patch("builtins.open", side_effect=FileNotFoundError)
    def test_load_config_file_not_found(self, mock_file):
        with self.assertRaises(SystemExit) as cm:
            load_config()
        self.assertEqual(cm.exception.code, 1)
        mock_file.assert_called_with('config/sensors_config.json', 'r')
```

Why `mock_open()` ?

- **Isolation**: Test doesn't depend on actual file system
- **Speed**: No disk I/O operations
- **Reliability**: Works regardless of file existence

11.1.2 Data Generation Tests

```python
# --- 2. PAYLOAD GENERATION TESTS ---
def test_payload_structure(self):
    """Verify generated payload format"""
    payload = generate_payload(self.sensors)

    self.assertIsInstance(payload, list)
    self.assertEqual(len(payload), 6)

    for sensor_data in payload:
        self.assertIn('name', sensor_data)
        self.assertIn('value', sensor_data)
        self.assertIn('timestamp', sensor_data)
        self.assertIn('status', sensor_data)
```

Why This Matters:

- **Contract Testing**: Dashboard expects specific JSON structure
- **API Stability**: Prevents breaking changes
- **Type Safety**: Ensures data types are consistent

11.1.3 Alarm Logic Test

```python
    # --- 3. ALARM LOGIC TESTS ---
    def test_alarm_logic(self):
        """Verify alarm status determination"""
        payload = generate_payload(self.sensors)

        for sensor_data in payload:
            value = sensor_data['value']
            status = sensor_data['status']

            # Find corresponding config
            sensor_config = self.sensors[sensor_data['name']]

            # Verify alarm logic
            if value < sensor_config['low']:
                self.assertEqual(status, "LOW ALARM")
            elif value > sensor_config['high']:
                self.assertEqual(status, "HIGH ALARM")
            else:
                self.assertEqual(status, "OK")

    # --- 4. VALUE RANGE TESTS ---
    def test_value_ranges(self):
        """Verify generated values are within expected ranges"""
        for _ in range(100):  # Test 100 samples
            payload = generate_payload(self.sensors)

            for sensor_data in payload:
                sensor_config = self.sensors[sensor_data['name']]

                # Value should be within low-variation to high+variation
                min_val = sensor_config['low'] - sensor_config.get('variation', 5.0)
                max_val = sensor_config['high'] + sensor_config.get('variation', 5.0)

                self.assertGreaterEqual(sensor_data['value'], min_val - 0.01)
                self.assertLessEqual(sensor_data['value'], max_val + 0.01)
```

11.1.4 JSON Parsing Test

```
    # --- 1. SENSOR PARSING ---
    def test_sensor_parsing(self):

        # Test that JSON sensor data is parsed correctly into expected structure --> list of di

        raw_json = '[{"name": "Temp", "value": 55.0, "status": "OK"}]'
        parsed_data = json.loads(raw_json)

        self.assertEqual(len(parsed_data), 1)
        self.assertEqual(parsed_data[0]['name'], "Temp")
        self.assertIsInstance(parsed_data[0]['value'], float)
```

Purpose: Verify JSON string correctly deserializes to Python objects.

11.1.5 API Compliance Test

```
  # --- 5. API OUTPUT TESTS ---
    def test_api_output_compliance(self):

        # 1. Setup sample config
        config = {"Temp": {"low": 20, "high": 30}, "Press": {"low": 50, "high": 100}}

        # 2. Generate the "API Output"
        payload = generate_payload(config)
        api_string = json.dumps(payload) + "\n"

        # 3. Assertions (The "Tests")
        self.assertTrue(api_string.endswith("\n"), "API Output must use newline termination.")

        decoded_payload = json.loads(api_string.strip())
        self.assertEqual(len(decoded_payload), 2, "API Output count must match config count.")
        self.assertEqual(decoded_payload[0]['name'], "Temp")
```

Purpose: Verify simulator output matches TCP protocol specification.

How It Works:

Part 1: Newline Termination

```python
self.assertTrue(api_string.endswith("\n"), "...")
```

- **Why**: TCP streams have no built-in message boundaries
- **Solution**: Use newline (`\n`) as frame delimiter
- **Verification**: Checks string ends with newline

Part 2: Sensor Count Validation

```python
decoded_payload = json.loads(api_string.strip())
self.assertEqual(len(decoded_payload), 2, "...")
```

- **Input**: 2 sensors in config
- **Expected**: 2 sensor objects in output
- **Verification**: Count matches

Part 3: Data Integrity

```python
self.assertEqual(decoded_payload[0]['name'], "Temp")
```

- **Verification**: First sensor has correct name
- **Ensures**: Order preservation and data accuracy

11.2 worker_test.py

```python
class TestSensorWorker(unittest.TestCase):

    def setUp(self):
        self.app = QCoreApplication([])
        self.config = load_config()


    # --- 1. WORKER INITIALIZATION TESTS ---
    def test_worker_initialization(self):
        """Verify worker thread initializes correctly"""
        worker = SensorWorker()

        self.assertIsNotNone(worker)
        self.assertFalse(worker.isRunning())
        self.assertTrue(worker._run_flag)


    # --- 2. SIGNAL EMISSION TESTS ---
    def test_signal_emission(self):
        """Verify signals are emitted correctly"""
        worker = SensorWorker()
        received_data = []
        log_messages = []

        def capture_data(data):
            received_data.append(data)

        def capture_log(msg):
            log_messages.append(msg)

        worker.data_received.connect(capture_data)
        worker.log_message.connect(capture_log)
        worker.start()

        # Wait for connection attempt
        QTest.qWait(1000)  # 1 second

        worker.stop()
        worker.wait()
```

```python
        # Should have attempted to connect
        self.assertIn("Attempting to connect to simulator...", log_messages)
        # If simulator is running, should receive data, but since it's not, just check attempt


    # --- 3. WORKER SHUTDOWN TESTS ---
    def test_shutdown(self):
        """Verify worker stops cleanly"""
        worker = SensorWorker()
        worker.start()

        self.assertTrue(worker.isRunning())

        worker.stop()
        worker.wait(5000)  # 5-second timeout

        self.assertFalse(worker.isRunning())
```

11.3 Test Execution and Coverage

11.3.1 Running the Tests

```bash
# Run all tests
python -m unittest simulator_test.py

# Run specific test class
python -m unittest simulator_test.TestSimulator

# Run single test
python -m unittest simulator_test.TestSimulator.test_alarm_logic
```

Expected Output:

```
----------------------------------------------------------------------
Ran 7 tests in 0.005s

OK
```

12. Conclusion

12.1 Project Summary

The **Real-Time Production Line Sensor Dashboard** successfully demonstrates the integration of multiple advanced software engineering concepts:

Key Achievements:

1. ✓ **Multithreaded Architecture**: Responsive GUI with concurrent I/O operations
2. ✓ **Network Communication**: Dual protocol support (TCP/WebSocket)
3. ✓ **Real-Time Visualization**: High-performance graphing at 2 Hz
4. ✓ **Robust Error Handling**: Graceful recovery from network failures
5. ✓ **Industrial Features**: Alarm system, data export, maintenance console

13.2 Technical Highlights

Threading Excellence:

- Zero race conditions through signal-slot architecture
- Proper resource cleanup and graceful shutdown
- Thread-safe communication without explicit locks

Protocol Design:

- Newline-delimited JSON for streaming
- Support for protocol switching without code changes

User Experience:

- Immediate visual feedback for all operations
- Desktop notifications for critical events
- Persistent configuration and session management

12.3 Learning Outcomes

This project provides hands-on experience with:

1. **PyQt6 Framework**: Modern GUI development patterns
2. **Asynchronous Programming**: Event loops and non-blocking I/O
3. **Network Protocols**: TCP sockets and WebSocket implementation
4. **Data Visualization**: Real-time plotting with PyQtGraph
5. **Software Testing**: Unit testing with mocks and patches

6. **Design Patterns**: MVC, Observer and Strategy
   uency queries

12.4 Acknowledgments

Technologies Used:

- **Qt Framework**: Cross-platform GUI toolkit
- **PyQtGraph**: Scientific graphics library
- **Python asyncio**: Asynchronous I/O
- **unittest**: Testing framework

Development Tools:

- **VS Code**: Primary IDE
- **Git**: Version control
- **pip**: Package management