```
import os
import json
import numpy as np
import time
import pickle
from fastapi import FastAPI, Request, WebSocket, WebSocketDisconnect
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
from fastapi.responses import HTMLResponse
from tensorflow.keras.models import load_model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import BinaryCrossentropy,
MeanSquaredError
from sklearn.preprocessing import MinMaxScaler
import uvicorn
import asyncio
from typing import Dict, Any

# Create static directory if it doesn't exist
os.makedirs("static", exist_ok=True)

app = FastAPI()
templates = Jinja2Templates(directory="templates")

class ConnectionManager:
    def __init__(self):
        self.active_connections: list[WebSocket] = []

    async def connect(self, websocket: WebSocket):
        await websocket.accept()
        self.active_connections.append(websocket)

    def disconnect(self, websocket: WebSocket):
        self.active_connections.remove(websocket)

    async def send_personal_message(self, message: str, websocket:
WebSocket):
        await websocket.send_text(message)

    async def broadcast(self, message: dict):
        for connection in self.active_connections:
            try:
                await connection.send_json(message)
            except:
                self.disconnect(connection)

manager = ConnectionManager()

class IoTCommunication:
    def __init__(self, config):
```

```
        self.config = config
        self.vrms = 230  # Initial values
        self.irms = 100
        self.temp = 25
        self.time_counter = 0
        self.control_active = False
        self.adjustments = [0, 0]
        self.anomaly_detected = False
        print(f"Connected to IoT gateway at {config['iot_gateway']}")

    def generate_sinusoidal_values(self):
        """Generate clean sinusoidal signals with occasional
anomalies"""
        self.time_counter += 0.1

        # Base sinusoidal values
        vrms = 230 + 10 * np.sin(self.time_counter)
        irms = 100 + 5 * np.sin(self.time_counter * 1.2)
        temp = 25 + 3 * np.sin(self.time_counter * 0.5)

        # Apply control adjustments if active
        if self.control_active:
            vrms += self.adjustments[0]
            irms += self.adjustments[1]
            # Gradually reduce adjustments to show control effect
            self.adjustments[0] *= 0.9
            self.adjustments[1] *= 0.9
            if abs(self.adjustments[0]) < 0.1 and
abs(self.adjustments[1]) < 0.1:
                self.control_active = False

        # 10% chance of anomaly when not in control mode
        if not self.control_active and np.random.random() < 0.1:
            self.anomaly_detected = True
            anomaly_type = np.random.choice(['overvoltage',
'overcurrent', 'both'])
            if anomaly_type in ['overvoltage', 'both']:
                vrms += 30 * np.sin(self.time_counter * 5)  # High
frequency spike
            if anomaly_type in ['overcurrent', 'both']:
                irms += 20 * np.sin(self.time_counter * 5)
        else:
            self.anomaly_detected = False

        # Update values
        self.vrms = max(0, vrms)
        self.irms = max(0, irms)
        self.temp = max(0, temp)
```

```
        return {
            'Vrms': self.vrms,
            'Irms': self.irms,
            'Temp': self.temp,
            'time': time.time(),
            'anomaly': self.anomaly_detected
        }

    def adjust_controls(self, adjustments):
        """Apply control adjustments"""
        self.adjustments = adjustments
        self.control_active = True
        print(f"Applying control adjustments: ΔVrms =
{adjustments[0]:.2f}V, ΔIrms = {adjustments[1]:.2f}A")

class MPCAutoController:
    def __init__(self):
        # Create default config if not exists
        if not os.path.exists('config/system_config.json'):
            os.makedirs('config', exist_ok=True)
            with open('config/system_config.json', 'w') as f:
                json.dump({
                    'iot_gateway': 'localhost:8000',
                    'time_steps': 10,
                    'ts_features': ['Temp', 'Solar', 'Wind', 'Dew',
'Vrms'],
                    'threshold': 0.7
                }, f)

        # Load configuration
        with open('config/system_config.json') as f:
            self.config = json.load(f)

        # Create dummy models if not exists
        if not os.path.exists('models/anomaly_detector.keras'):
            os.makedirs('models', exist_ok=True)
            from tensorflow.keras.models import Sequential
            from tensorflow.keras.layers import Dense
            model = Sequential([Dense(10, input_shape=(10,5)),
Dense(1)])
            model.compile(optimizer='adam',
loss='binary_crossentropy')
            model.save('models/anomaly_detector.keras')

        if not os.path.exists('models/mpc_controller.keras'):
            model = Sequential([Dense(10, input_shape=(10,)),
Dense(2)])
            model.compile(optimizer='adam', loss='mse')
            model.save('models/mpc_controller.keras')
```

```
        # Load models with custom objects
        custom_objects = {
            'Adam': Adam,
            'BinaryCrossentropy': BinaryCrossentropy,
            'MeanSquaredError': MeanSquaredError
        }

        self.anomaly_model = load_model(
            'models/anomaly_detector.keras',
            custom_objects=custom_objects
        )
        self.mpc_model = load_model(
            'models/mpc_controller.keras',
            custom_objects=custom_objects
        )

        # Create dummy scalers if not exists
        if not os.path.exists('scalers/time_series_scaler.pkl'):
            os.makedirs('scalers', exist_ok=True)
            from sklearn.preprocessing import MinMaxScaler
            scaler = MinMaxScaler()
            with open('scalers/time_series_scaler.pkl', 'wb') as f:
                pickle.dump(scaler, f)
            with open('scalers/control_scaler.pkl', 'wb') as f:
                pickle.dump(scaler, f)
            with open('scalers/output_scaler.pkl', 'wb') as f:
                pickle.dump(scaler, f)

        # Load scalers
        with open('scalers/time_series_scaler.pkl', 'rb') as f:
            self.ts_scaler = pickle.load(f)
        with open('scalers/control_scaler.pkl', 'rb') as f:
            self.control_scaler = pickle.load(f)
        with open('scalers/output_scaler.pkl', 'rb') as f:
            self.output_scaler = pickle.load(f)

        # Initialize IoT
        self.iot = IoTCommunication(self.config)

        # Initialize time series buffer
        self.ts_buffer = np.zeros((self.config['time_steps'],
len(self.config['ts_features'])))

    def update_buffer(self, measurements):
        """Update time series buffer with new measurements"""
        new_point = np.array([
            measurements['Temp'],
            800 + 100 * np.sin(self.iot.time_counter * 0.3),  # Solar
```

```
            5 + 2 * np.sin(self.iot.time_counter * 0.7),       # Wind
            25,  # Placeholder for Dew
            measurements['Vrms']  # Using Vrms as Out temp proxy
        ])

        # Roll buffer and add new point
        self.ts_buffer = np.roll(self.ts_buffer, -1, axis=0)
        self.ts_buffer[-1] = new_point

    def detect_anomaly(self, control_params):
        """Detect anomalies using hybrid model"""
        try:
            # Prepare inputs
            ts_scaled =
self.ts_scaler.transform(self.ts_buffer).reshape(1,
*self.ts_buffer.shape)
            control_scaled =
self.control_scaler.transform([control_params])

            # Predict anomaly probability
            anomaly_prob = self.anomaly_model.predict(
                [ts_scaled, control_scaled], verbose=0)[0][0]

            return anomaly_prob > self.config['threshold'],
anomaly_prob
        except Exception as e:
            print(f"Anomaly detection error: {str(e)}")
            return False, 0.0

    def calculate_control(self):
        """Calculate MPC control adjustments"""
        try:
            # Prepare input features (current state)
            features = np.concatenate([
                self.ts_buffer.mean(axis=0),  # Mean of time series
                self.ts_buffer[-1]           # Latest values
            ]).reshape(1, -1)

            # Get control action
            control_scaled = self.mpc_model.predict(features,
verbose=0)
            return
self.output_scaler.inverse_transform(control_scaled)[0]
        except Exception as e:
            print(f"Control calculation error: {str(e)}")
            return 0.0, 0.0  # Return zero adjustments if error

    async def run_control_loop(self):
        """Main control loop that runs in background"""
```

```
        print("\nStarting MPC Anomaly Control System")
        print("Monitoring station parameters...\n")

        while True:
            # 1. Get measurements
            measurements = self.iot.generate_sinusoidal_values()
            self.update_buffer(measurements)

            # 2. Prepare control params
            control_params = [
                measurements['Temp'] + 5,  # MaxTemp
                measurements['Temp'],      # AmbientTemp
                5                          # TempDiff
            ]

            # 3. Check for anomalies
            is_anomaly, confidence =
self.detect_anomaly(control_params)

            if is_anomaly:
                print(f"\n! ANOMALY DETECTED (confidence:
{confidence:.1%}) !")
                print(f"Current readings:
Vrms={measurements['Vrms']:.1f}V, Irms={measurements['Irms']:.1f}A")

                # 4. Calculate and apply control
                adjustments = self.calculate_control()
                self.iot.adjust_controls(adjustments)

                # Broadcast control action
                await manager.broadcast({
                    "type": "control",
                    "vrms_adj": float(adjustments[0]),
                    "irms_adj": float(adjustments[1]),
                    "time": time.time()
                })

            # Broadcast sensor data
            await manager.broadcast({
                "type": "sensor",
                "Vrms": float(measurements['Vrms']),
                "Irms": float(measurements['Irms']),
                "Temp": float(measurements['Temp']),
                "anomaly": measurements['anomaly'],
                "time": measurements['time']
            })

            await asyncio.sleep(1)  # Control interval
```

```
@app.get("/", response_class=HTMLResponse)
async def get_root(request: Request):
    return templates.TemplateResponse("index.html", {"request":
request})

@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await manager.connect(websocket)
    try:
        while True:
            # Keep connection alive
            await websocket.receive_text()
    except WebSocketDisconnect:
        manager.disconnect(websocket)
    except Exception as e:
        print(f"WebSocket error: {e}")
        manager.disconnect(websocket)

async def main():
    # Start the controller
    controller = MPCAutoController()

    # Run the control loop in the background
    control_task = asyncio.create_task(controller.run_control_loop())

    # Start the web server
    config = uvicorn.Config(app, host="0.0.0.0", port=8000)
    server = uvicorn.Server(config)
    await server.serve()

    # Wait for the control task to complete (which it won't)
    await control_task

if __name__ == "__main__":
    # Create the template if it doesn't exist
    os.makedirs("templates", exist_ok=True)
    if not os.path.exists("templates/index.html"):
        with open("templates/index.html", "w") as f:
            f.write("""<!DOCTYPE html>
<html>
<head>
    <title>IoT MPC Control Dashboard</title>
    <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
    <script
src="https://cdn.jsdelivr.net/npm/luxon@3.0.1/build/global/luxon.min.j
s"></script>
    <script src="https://cdn.jsdelivr.net/npm/chartjs-adapter-
luxon@1.2.0/dist/chartjs-adapter-luxon.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/chartjs-plugin-
streaming@2.0.0/dist/chartjs-plugin-streaming.min.js"></script>
```

```
    <style>
        body { font-family: Arial, sans-serif; margin: 20px; }
        .chart-container { width: 90%; margin: 20px auto; height:
300px; }
        .control-panel { background: #f5f5f5; padding: 20px; border-
radius: 5px; margin: 20px auto; width: 90%; }
        .anomaly { background-color: #ffcccc; }
    </style>
</head>
<body>
    <h1>IoT MPC Control Dashboard</h1>

    <div class="chart-container">
        <h2>Voltage (Vrms)</h2>
        <canvas id="voltageChart"></canvas>
    </div>

    <div class="chart-container">
        <h2>Current (Irms)</h2>
        <canvas id="currentChart"></canvas>
    </div>

    <div class="chart-container">
        <h2>Temperature (°C)</h2>
        <canvas id="tempChart"></canvas>
    </div>

    <div class="control-panel" id="controlPanel">
        <h2>Control Actions</h2>
        <div id="controlLog"></div>
    </div>

    <script>
        // Initialize WebSocket connection
        const socket = new WebSocket(`ws://$
{window.location.host}/ws`);

        // Initialize charts with streaming plugin
        const voltageCtx =
document.getElementById('voltageChart').getContext('2d');
        const currentCtx =
document.getElementById('currentChart').getContext('2d');
        const tempCtx =
document.getElementById('tempChart').getContext('2d');

        const chartOptions = {
            responsive: true,
            maintainAspectRatio: false,
            scales: {
```

```
            x: {
                type: 'realtime',
                realtime: {
                    duration: 30000,  // show 30 seconds of data
                    refresh: 1000,
                    delay: 1000,
                    onRefresh: chart => {}
                }
            },
            y: {
                title: {
                    display: true,
                    text: 'Value'
                }
            }
        },
        interaction: {
            intersect: false
        }
    };

    const voltageChart = new Chart(voltageCtx, {
        type: 'line',
        data: {
            datasets: [{
                label: 'Vrms (V)',
                borderColor: 'rgb(75, 192, 192)',
                backgroundColor: 'rgba(75, 192, 192, 0.1)',
                borderWidth: 2,
                pointRadius: 0,
                data: []
            }]
        },
        options: chartOptions
    });

    const currentChart = new Chart(currentCtx, {
        type: 'line',
        data: {
            datasets: [{
                label: 'Irms (A)',
                borderColor: 'rgb(255, 99, 132)',
                backgroundColor: 'rgba(255, 99, 132, 0.1)',
                borderWidth: 2,
                pointRadius: 0,
                data: []
            }]
        },
        options: chartOptions
```

```
        });

        const tempChart = new Chart(tempCtx, {
            type: 'line',
            data: {
                datasets: [{
                    label: 'Temperature (°C)',
                    borderColor: 'rgb(54, 162, 235)',
                    backgroundColor: 'rgba(54, 162, 235, 0.1)',
                    borderWidth: 2,
                    pointRadius: 0,
                    data: []
                }]
            },
            options: chartOptions
        });

        // Handle incoming WebSocket messages
        socket.onmessage = function(event) {
            const data = JSON.parse(event.data);

            if (data.type === 'sensor') {
                // Update charts with new data points
                const timestamp =
luxon.DateTime.fromSeconds(data.time).toJSDate();

                voltageChart.data.datasets[0].data.push({
                    x: timestamp,
                    y: data.Vrms
                });

                currentChart.data.datasets[0].data.push({
                    x: timestamp,
                    y: data.Irms
                });

                tempChart.data.datasets[0].data.push({
                    x: timestamp,
                    y: data.Temp
                });

                // Highlight during anomalies
                const controlPanel =
document.getElementById('controlPanel');
                if (data.anomaly) {
                    controlPanel.classList.add('anomaly');
                } else {
                    controlPanel.classList.remove('anomaly');
                }
```

```
                // Update charts
                voltageChart.update('none');
                currentChart.update('none');
                tempChart.update('none');

            } else if (data.type === 'control') {
                // Log control actions
                const timestamp = new Date(data.time *
1000).toLocaleTimeString();
                const logEntry = document.createElement('p');
                logEntry.innerHTML = `<strong>${timestamp}</strong> -
Control applied: ` +
                                      `ΔVrms = <span
style="color:blue">${data.vrms_adj.toFixed(2)}V</span>, ` +
                                      `ΔIrms = <span
style="color:red">${data.irms_adj.toFixed(2)}A</span>`;

document.getElementById('controlLog').prepend(logEntry);
            }
        };

        // Handle WebSocket errors
        socket.onerror = function(error) {
            console.error('WebSocket error:', error);
        };
    </script>
</body>
</html>""")

    asyncio.run(main())
```