Testing Document

Team 19

Project 4

Mohammed Suleman Riaz	22050135	m.riaz18@bradford.ac.uk
Sara Ali	22010736	s.ali232@bradford.ac.uk
Noreen Fatima	23009863	n.fatima10@bradford.ac.uk
Sakib Iqbal	21006625	s.iqbal132@bradford.ac.uk
Uzair Ali	22009327	u.ali33@bradford.ac.uk
Shad Mortaza	21001976	s.mortaza@bradford.ac.uk
Sahil Farooq	23014923	s.farooq16@bradford.ac.uk
Sana Ullah	23040219	s.ullah20@bradford.ac.uk

Contents

Acceptance Testing 1
The scope of this testing includes verifying that: 1
Test Methodology: 1
1. Handling Historical and Real-Time Data & API Development 2
2. Secure Login System 3
3. Interactive Sensor Data Visualisation with Graphs and Charts 4
4. Traffic Light System for Anomaly Detection 4
5. Responsive Dashboard for PC and Mobile Devices 6
6. Synthetic Data Generation using a Random Number Generator 6
Conclusion: 7
Unit Testing 8
Code Inspection 17
GitHub Links 21
Peer Review 21

Acceptance Testing

Acceptance testing ensures that the software meets the specified functional and non-functional requirements.

The scope of this testing includes verifying that:

- The system functions as expected according to the requirements document.
- Input validation works correctly for expected and unexpected values.
- The system handles edge cases properly.

Test Methodology:

- Each function has a set of test sequences covering correct inputs and incorrect inputs.
- The test execution follows the incremental testing approach, covering features progressively.

1. Handling Historical and Real-Time Data & API Development

Description:

The system retrieves historical and real-time sensor data from a database. It provides an API that allows users and other systems to access, process, and analyse sensor readings efficiently.

Who uses it?

- The dashboard retrieves and displays real-time data.
- The dashboard access historical data for trend analysis.
- The API allows external systems to fetch and process sensor data.

How it works?

- Sensor data is continuously stored in a MySQL database.
- A Flask API retrieves the latest sensor values and serves them upon request.
- Users or applications send API requests to fetch specific data (real-time or historical).

Input Sequence	Description	Input	Expected Output	Actual Output	Comments
API Request	Fetch real-time data	API call: /api/realtime?sensor_id=01	Returns JSON with latest sensor data	Returned JSON with latest sensor data	Pass
API Request	Fetch historical data	API call: /api/historical?sensor_id=01&date=2025-03-28	Returns JSON with latest sensor data	Returned JSON with latest sensor data	Pass
Database Connection	Ensure connection to MySQL	None	Show message: “connected”	Show message: “connected”	Pass
Flask route	Serve results	Open the website	HTML page shows	Page rendered	Pass

	via web interface		updated sensor status		
Database Connection	Retrieve latest data	Database query execution	Latest sensor reading is returned	Latest sensor reading is returned	Pass

2. Secure Login System

Description:

The system provides a secure login mechanism where users authenticate to gain access. Passwords are encrypted.

Who uses it?

- Admins log in to manage system access and users.
- Users log in to monitor real-time data.
- Any authorised user uses the login system to access the dashboard.

How it works?

- Users enter their details.
- The system hashes passwords before storing or verifying them.
- If credentials are incorrect, the system prompts users to try again.

Input Sequence	Description	Input	Expected Output	Actual Output	Comments
Login	Correct details	Username :"admin"and password:"admin"	User is logged in	User is logged in	Pass
Login	Incorrect Username	Wrong username "kftgoerpg"	Login fails with message	Login fails with message	Pass
Login	Incorrect password	Wrong password: "amfoapijf"	Login fails with message	Login fails with message	Pass

Login	No input at all	Nothing Just press log in without input	Login fails with message no input	Login fails with message no input	Pass
Logout	Check logout function	Press logout	User will be logged out	User is logged out	Pass

3. Interactive Sensor Data Visualisation with Graphs and Charts

Description:

The dashboard presents real-time sensor statistics in interactive charts (e.g., pie charts) for better data analysis and monitoring.

Who uses it?

- Users to monitor current conditions.
- Users to review sensor trends.
- Users to identify patterns in historical data.

How it works?

- The system fetches data from the database via the API.
- Data is dynamically displayed using tableau.
- Users can filter and interact with the graphs for better insights.

Input Sequence	Description	Input	Expected Output	Actual Output	Comments
Pie Chart	Display distribution	Sensor category data	Pie chart correctly reflects distribution	Pie chart correctly reflects distribution	Pass

4. Traffic Light System for Anomaly Detection

Description:

The system highlights unusual sensor readings using a traffic light system (Green, Amber, Red) and detects anomalies using a Machine Learning (ML) model.

Who uses it?

- Users to detect abnormal conditions.
- Users to monitor performance and respond to potential issues.

How it works?

- Sensor data is processed in real-time.
- The ML model predicts expected values and detects anomalies.
- The system assigns a traffic light status:

  Green: Normal readings

  Amber: Warning (close to anomaly thresholds given by lower and higher boundaries)

  Red: anomaly detected

Input Sequence	Description	Input	Expected Output	Actual Output	Comments
Normal Data	Check green status	Normal sensor value in range Sensor value = 250 Correct value = 250	Green status	Green status	Pass
Warning Data	Check amber status	Sensor value inside lower and higher boundaries and 15 out of range of the boundaries Sensor value = 240 Correct value = 250	Amber status	Amber status	Pass
Anomalous Data	Check red status	Out of range sensor value Sensor value = 50389 Correct value = 250	Red status	Red status	Pass

No Data	Check status	Nothing	Error	Error	Pass

5. Responsive Dashboard for PC and Mobile Devices

Description:

The dashboard is designed to be fully responsive, meaning it adapts to different screen sizes for desktop and Android users.

Who uses it?

- Users on mobile or tablets.
- Users on a desktop PC.

How it works?

- The frontend (HTML, CSS, JavaScript) is built using a responsive framework.
- UI elements adjust dynamically based on screen size and orientation.

Input Sequence	Description	Input	Expected Output	Actual Output	Comments
Desktop view	Open dashboard on a desktop	Access from PC	Dashboard displays correctly	Dashboard displays correctly	Pass
Mobile view	Open on a mobile	Access from an I-Phone	Dashboard displays correctly	Dashboard displays correctly	Pass
Desktop view	Open dashboard on a desktop	Access from a Laptop	Dashboard displays correctly	Dashboard displays correctly	Pass

6. Synthetic Data Generation using a Random Number Generator

Description:

A random number generator creates synthetic sensor data to simulate real-time conditions when live sensor data is unavailable.

Who uses it?

- Users who monitor sensor data on the dashboard.

How it works?

- The system generates random sensor values based on expected data ranges.
- Synthetic values are stored and processed just like real sensor readings.
- This allows for testing and development without requiring live sensor inputs.

Input Sequence	Description	Input	Expected Output	Actual Output	Comments
Data Simulation	Data Simulation in correct range	Run Simulation function	Data is generated within expected range	Data is generated within expected range	Pass
Data Simulation	Data Simulation in wrong range	Simulate extreme data out of range	System flags the data accordingly	System flags the data accordingly	Pass
Data Simulation	No Data simulation	Nothing	Error message	Error message	Pass

Conclusion:

The acceptance testing was successfully conducted. All functional and non-functional test cases have passed, verifying that the system meets the requirements. The project was ready for deployment after.

Unit Testing

test_sensor_anomaly_detection.py

```python
# test_sensor_anomaly_detection.py
import pytest
import pandas as pd
import numpy as np
import mysql.connector
import joblib
import datetime
from unittest.mock import patch, MagicMock, mock_open

# Import the module to test (adjust the import based on your module name)
# For this test, we'll mock the import and functions
# assuming your file is named sensor_anomaly_detection.py
# @patch('sensor_anomaly_detection.mysql.connector')
# @patch('sensor_anomaly_detection.joblib')
# @patch('sensor_anomaly_detection.pd')
```

This code is setting up a Python unit test file (using pytest) with mock imports for testing a sensor anomaly detection module, specifically mocking database connections, machine learning model loading, and pandas functionality.

```python

class TestSensorAnomalyDetection:

    @pytest.fixture
    def mock_cursor(self):
        """Create a mock cursor with fetchone method"""
        cursor_mock = MagicMock()
        return cursor_mock
```

This code defines a pytest fixture called mock_cursor that creates a mocked database cursor object (using MagicMock) for testing database interactions in the sensor anomaly detection module.

```python

    @pytest.fixture
    def mock_connection(self, mock_cursor):
        """Create a mock connection with is_connected and cursor methods"""
        conn_mock = MagicMock()
        conn_mock.is_connected.return_value = True
        conn_mock.cursor.return_value = mock_cursor
        return conn_mock
```

This code defines a pytest fixture called mock_connection that creates a mocked database connection (using MagicMock) with simulated connectivity (is_connected) and cursor retrieval functionality, building on the mock_cursor fixture for testing database operations.

```
32
33      @pytest.fixture
34      def mock_prophet_model(self):
35          """Create a mock Prophet model"""
36          model_mock = MagicMock()
37          forecast = pd.DataFrame({
38              'ds': [pd.Timestamp('2025-04-08 12:00:00')],
39              'yhat': [120.0],
40              'yhat_lower': [110.0],
41              'yhat_upper': [130.0]
42          })
43          model_mock.predict.return_value = forecast
44          return model_mock
```

This code defines a pytest fixture called mock_prophet_model that creates a mocked Facebook Prophet model (using MagicMock) with a predefined forecast output, simulating prediction results for time-series anomaly detection testing.

```
45
46      def test_database_connection(self, monkeypatch):
47          """Test the database connection is established correctly"""
48          mock_connector = MagicMock()
49          mock_conn = MagicMock()
50          mock_conn.is_connected.return_value = True
51          mock_connector.connect.return_value = mock_conn
52
53          monkeypatch.setattr('mysql.connector.connect', mock_connector.connect)
54
```

This code tests the database connection functionality by mocking `mysql.connector.connect` using `monkeypatch` and verifying that a successful connection (with `is_connected=True`) can be established.

```
55          # Import only after monkeypatching
56          from importlib import reload
57          import sys
58
59          # This is a bit hacky for unit tests, but simulates what would happen
60          # when the script is run and the connection is established
61          with patch('sys.modules', sys.modules.copy()):
62              with patch('mysql.connector.connect', return_value=mock_conn):
63                  with patch('builtins.print') as mock_print:
64                      # We need to use exec to simulate the global code execution
65                      # as importing the module would execute the connection code
66                      connection_code = """
```

This code sets up a complex test environment using mocking and patching to simulate module imports and database connection behaviour, specifically isolating the test execution from real

database interactions while capturing print outputs.

```
import mysql.connector
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="",
    database="rakusens"
)
if conn.is_connected():
    print('Connected to database')
else:
    print('Connection failed')
"""
                    exec(connection_code)
                    mock_print.assert_called_with('Connected to database')

```

This code tests a database connection script by executing it in a mocked environment and verifying that the expected "Connected to database" message is printed when the connection succeeds.

```

    @patch('mysql.connector.connect')
    def test_get_latest_data_line4(self, mock_connect, mock_cursor):
        """Test get_latest_data function for line4"""
        # Setup mock return data
        mock_cursor.fetchone.return_value = (1, '2025-04-08 12:00:00', 100, 105, 110, 115, 120, 125, 130, 135)

        # Define the function separately to test it
        def get_latest_data(table_name):
            query = f"SELECT * FROM {table_name} ORDER BY timestamp DESC LIMIT 1;"
            mock_cursor.execute(query)
            row = mock_cursor.fetchone()

            if row:
                if table_name == 'line4_sensors':
                    columns = ["id", "timestamp"] + [f"r{str(i).zfill(2)}" for i in range(1, 9)]
                else:
                    columns = ["id", "timestamp"] + [f"r{str(i).zfill(2)}" for i in range(1, 18)]
                data = dict(zip(columns, row))
                return data
            return None
```

This code tests the `get_latest_data` function for a specific production line (line4) by mocking the database connection and cursor, simulating a database query response, and verifying the function correctly processes and returns sensor data in dictionary format.

```
        # Call the function
        result = get_latest_data('line4_sensors')

        # Assertions
        mock_cursor.execute.assert_called_with("SELECT * FROM line4_sensors ORDER BY timestamp DESC LIMIT 1;")
        assert result['id'] == 1
        assert result['timestamp'] == '2025-04-08 12:00:00'
        assert result['r01'] == 100
        assert result['r08'] == 135
```

This code verifies that the `get_latest_data` function correctly executes the expected SQL query and properly formats the returned sensor data by asserting the database call was made with the right

```
112
113     @patch('mysql.connector.connect')
114     def test_get_latest_data_line5(self, mock_connect, mock_cursor):
115         """Test get_latest_data function for line5"""
116         # Setup mock return data - 17 sensors plus id and timestamp
117         mock_cursor.fetchone.return_value = tuple([2, '2025-04-08 12:05:00'] + list(range(100, 117)))
118
119         # Define the function separately to test it
120         def get_latest_data(table_name):
121             query = f"SELECT * FROM {table_name} ORDER BY timestamp DESC LIMIT 1;"
122             mock_cursor.execute(query)
123             row = mock_cursor.fetchone()
124
125             if row:
126                 if table_name == 'line4_sensors':
127                     columns = ["id", "timestamp"] + [f"r{str(i).zfill(2)}" for i in range(1, 9)]
128                 else:
129                     columns = ["id", "timestamp"] + [f"r{str(i).zfill(2)}" for i in range(1, 18)]
130                 data = dict(zip(columns, row))
131                 return data
132             return None
```

```
1   # test_simulation.py
2   import unittest
3   from unittest.mock import patch, MagicMock, call
4   import mysql.connector
5   import time
6   from datetime import datetime
7   import sys
8   import os
9   import importlib
10  import numpy as np
```

This code sets up necessary imports for unit testing, like mocking utilities (`unittest.mock`), database connections (`mysql.connector`), and other tools in preparation for simulation-related functionality to be tested.

```
11
12  # Set up path to import the simulation module
13  # Assuming the test file is in the same directory as the simulation script
14  sys.path.append(os.path.dirname(os.path.abspath(__file__)))
15
```

To enable importing the simulation module for unit testing, this code adds the directory containing the current test file to the system path.

```
# Mock the simulation module to avoid actual database connections during tests
class TestSensorSimulation(unittest.TestCase):
    @patch('mysql.connector.connect')
    def test_database_connection(self, mock_connect):
        # Setup mock
        mock_conn = MagicMock()
        mock_connect.return_value = mock_conn
        mock_conn.is_connected.return_value = True

        # Import the module
        with patch.dict('sys.modules', {'mysql.connector': mock_connect}):
            # Import the module under test
            sim_module = importlib.import_module('simulate_real_time_data')

        # Assert connection was attempted with correct parameters
        mock_connect.assert_called_once_with(
            host="localhost",
            user="root",
            password="",
            database="rakusens"
        )

        # Assert connection check was performed
        mock_conn.is_connected.assert_called_once()

    @patch('time.sleep', return_value=None)  # Skip the sleep to make tests run faster
    @patch('mysql.connector.connect')
    def test_simulate_and_insert(self, mock_connect, mock_sleep):
```

Instead of really calling databases or waiting during unit tests, this code simulates database connections and time delays to test the simulate_real_time_data module.

```
        # Setup mock connection and cursor
        mock_conn = MagicMock()
        mock_cursor = MagicMock()
        mock_connect.return_value = mock_conn
        mock_conn.cursor.return_value = mock_cursor
        mock_conn.is_connected.return_value = True

        # Set up a side effect to break the infinite loop after one iteration
        mock_sleep.side_effect = [None, Exception("Stop iteration")]

        # Import the module and patch random generation for predictable results
        with patch.dict('sys.modules', {'mysql.connector': mock_connect}):
            with patch('numpy.random.randint') as mock_randint:
                # Set up consistent test data
                mock_randint.side_effect = lambda min_val, max_val, *args: 100

                # Import the module under test
                try:
                    sim_module = importlib.import_module('simulate_real_time_data')
                    sim_module.simulate_and_insert()
                except Exception as e:
                    if str(e) != "Stop iteration":
                        raise
```

This code tests the behaviour of the simulate_real_time_data module in a controlled, predictable manner by setting up mock database connections, forcing an early exit from an infinite loop, and overriding random number generation for consistency.

```

        # Assert cursor execute was called exactly twice (once for each table)
        self.assertEqual(mock_cursor.execute.call_count, 2)

        # Assert commit was called to save the data
        mock_conn.commit.assert_called()

        # Check that time.sleep was called with the right value
        mock_sleep.assert_called_with(30)
```

This code is responsible for ensuring that the database operations- such as executing SQL queries and the commit function - along with the time waits (sleep) are functioning properly during the unit test. It assesses the expected behavior of the mock calls and their arguments. The test makes sure that the database interactions are correct and that the sleep intervals are used appropriately.

```

    @patch('mysql.connector.connect')
    def test_data_generation_ranges(self, mock_connect):
        # Setup mock
        mock_conn = MagicMock()
        mock_connect.return_value = mock_conn
        mock_conn.is_connected.return_value = True

```

This code is designed to mimic a MySQL database connection, enabling us to test data generation ranges without the need for a real database when we're running unit tests.

```
        # Import the module and capture random generation without patching
        with patch.dict('sys.modules', {'mysql.connector': mock_connect}):
            sim_module = importlib.import_module('simulate_real_time_data')

        # Test line4 sensor data ranges
        for _ in range(100):  # Run multiple iterations to check range
            line4_data = {f"r{str(i).zfill(2)}": np.random.randint(0, 500) for i in range(1, 9)}
            for sensor_id, value in line4_data.items():
                self.assertGreaterEqual(value, 0)
                self.assertLess(value, 500)

        # Test line5 sensor data ranges
        for _ in range(100):
            line5_data = {f"r{str(i).zfill(2)}": np.random.randint(0, 390) for i in range(1, 18)}
            for sensor_id, value in line5_data.items():
                self.assertGreaterEqual(value, 0)
                self.assertLess(value, 390)
```

This code is designed to check if the randomly generated sensor data values for two production lines, line 4 and line 5, consistently stay within their defined valid ranges 0 to 500 for line4 and 0 to 390 for line 5 over several iterations.

```

    @patch('datetime.datetime')
    @patch('mysql.connector.connect')
    def test_timestamp_format(self, mock_connect, mock_datetime):
        # Setup mocks
        mock_conn = MagicMock()
        mock_cursor = MagicMock()
        mock_connect.return_value = mock_conn
        mock_conn.cursor.return_value = mock_cursor
        mock_conn.is_connected.return_value = True

        # Mock datetime.now() to return a fixed datetime
        mock_now = MagicMock()
        mock_now.strftime.return_value = '2023-10-25 14:30:00'
        mock_datetime.now.return_value = mock_now

        # Import the module and patch
        with patch.dict('sys.modules', {'mysql.connector': mock_connect}):
            with patch('time.sleep') as mock_sleep:
                # Set up to break after one iteration
                mock_sleep.side_effect = Exception("Stop iteration")

                with patch('numpy.random.randint', return_value=100):
                    try:
                        sim_module = importlib.import_module('simulate_real_time_data')
                        sim_module.simulate_and_insert()
                    except Exception as e:
                        if str(e) != "Stop iteration":
                            raise
```

This code is all about checking if the simulate_real_time_data module is doing its job right when it comes to formatting and using timestamps in database operations. It does this by simulating datetime functions, creating mock database connections, and ensuring the simulation loop can exit in a controlled manner. The test focuses on how timestamps are managed, all while steering clear of actual database calls and preventing infinite loops.

```
130
131         # Assert timestamp format is correct in the SQL queries
132         calls = mock_cursor.execute.call_args_list
133
134         # Check first call (Line 4)
135         line4_call = calls[0]
136         self.assertEqual(line4_call[0][1][0], '2023-10-25 14:30:00')
137
138         # Check second call (Line 5)
139         line5_call = calls[1]
140         self.assertEqual(line5_call[0][1][0], '2023-10-25 14:30:00')
141
```

This code is all about making sure that the timestamps being inserted into the database queries for both production lines (line4 and line5) are in the right format and match the expected mock datetime value ('2023-10-25 14:30:00'). The test looks at SQL execution calls to ensure that the timestamps are consistent during the simulated data insertion process.

```
142     @patch('mysql.connector.connect')
143     def test_database_failure(self, mock_connect):
144         # Simulate database connection failure
145         mock_conn = MagicMock()
146         mock_connect.return_value = mock_conn
147         mock_conn.is_connected.return_value = False
148
149         # Redirect stdout to capture print output
150         import io
151         from contextlib import redirect_stdout
152
153         # Import the module and check output
154         f = io.StringIO()
155         with redirect_stdout(f):
156             with patch.dict('sys.modules', {'mysql.connector': mock_connect}):
157                 sim_module = importlib.import_module('simulate_real_time_data')
158
159         # Check that the failure message was printed
160         output = f.getvalue()
161         self.assertIn('Failed to connect', output)
162
163
164 if __name__ == '__main__':
165     unittest.main()
```

Here, the code is testing how the simulate_real_time_data module handles a database connection failure. It simulates a failed connection and ensures that the expected error message ('Failed to connect') is printed to stdout. By using redirect_stdout, the test captures the output and checks that the error handling is working correctly.

websocketFunctions.php

```php
<?php

function isValidData($data) {
    return !empty($data) && is_string($data);
}

function broadcastToClients($clients, $data) {
    $count = 0;
    foreach ($clients as $c) {
        if (is_resource($c)) {
            fwrite($c, $data);
            $count++;
        }
    }
    return $count;
}
?>
```

Performs unit test on WebSocket functions to validate WebSocket data and successfully send messages to all connected clients.

websocketTest.php

```php
<?php
include('websocketFunctions.php');

// === Simple Unit Test Functions ===
function assertEquals($input, $expectedOutput) {
    if ($input == $expectedOutput) {
        echo "✅ SUCCESS<br>";
        return true;
    } else {
        echo "❌ FAILED - Expected " . var_export($expectedOutput, true) . ", got " . var_export($input, true) . "<br>";
        return false;
    }
}

function assertTrue($input) {
    return assertEquals($input, true);
}

function assertFalse($input) {
    return assertEquals($input, false);
}
```

This tests the logic of the websocket_server.php and defines custom test functions; assertEquals, assertTrue, assertFalse to check if the output expectations are met. It uses simple if-statements to manually check whether the function gives the expected result.

```php
// === Unit Tests ===
echo "<h2>WebSocket Utility Function Tests</h2>";

echo "<h3>Test isValidData()</h3>";
assertTrue(isValidData("sensor_001: 42"));
assertFalse(isValidData(""));
assertFalse(isValidData(null));

echo "<h3>Test broadcastToClients()</h3>";

// Create test clients (memory streams for simulation)
$client1 = fopen('php://memory', 'w+');
$client2 = fopen('php://memory', 'w+');
$clients = [$client1, $client2];

$data = "test_data";
$count = broadcastToClients($clients, $data);
assertEquals($count, 2);

// Close mock clients
fclose($client1);
fclose($client2);
?>
```

This runs unit tests on isValidData() with various inputs and the broadcastToClients() function using simulated clients. It uses memory streams and custom test functions to manually check if the output matches the expected result.

Code Inspection

Page: rakusens.html

Element	Criteria	Pass/Fail	Comments
Banner	Is the image aligned properly and visible?	Pass	The banner loads correctly and is well-aligned
Logo	Consistent with branding, positioned correctly?	Pass	Logo is centred and appropriately styled under the banner
Headings	Uses semantic tags with clear visual hierarchy?	Pass	<h1> and <h2> used with a clear hierarchy
Theme Switch	Proper placement and interactive styling?	Pass	Positioned top-right; clearly styled and functional toggle switch
Spacing	Even and visually clean spacing between elements?	Partial Pass	Padding below <h2> is slightly tight. Suggest increasing bottom margin

Responsiveness	Layout adapts well to various screen sizes?	Pass	Adapts the screen sizes
Colours	Follows consistent use of brand colours?	Pass	Colours align with Rakusen's theme
Fonts	Font family used consistently across all text elements?	Pass	Font family used uniformly
Text Alignment	Aligned for visual clarity and consistency?	Pass	Elements are visually clean and appropriately aligned

Page: styles.css

Component	Criteria	Pass/Fail	Comments
General Body styling	Base layout, font, and colour setting, transition animations	Pass	Proper base setup with smooth transition for colour changes
Dark Mode Colours	Provides good visual contrast in dark mode	Pass	Sufficient contrast between background and text colours in dark mode
Top Section	Visually distinguishes header in dark mode	Pass	Background and height settings work well to highlight
Logo & Banner	Proper alignment and sizing	Pass	Centred layout with consistent padding and image sizing
Typography	Consistent font style for headers	Pass	Fonts are consistent with good sizing and alignment
Toggle Switch Styling	Visually clear and interactive toggle design	Pass	Well-stayed with rounded slider and animated transition
Code Organisation	Logical grouping and consistent formatting	Pass	Organised by section with readable
Maintainability	Easy to update or extend	Pass	Modular approach makes the code easy to maintain and expand

Page: login.html

Component	Criteria	Pass/Fail	Comments
DOCTYPE & HTML	Correct <!DOCTYPE> declaration and root structure	Pass	Proper HTML5 declaration and structure used
Meta Tags	User viewport and charset declarations	Pass	Include UTF-8 charset and responsive viewport tag
Title	Descriptive and appropriate for the page	Pass	Title is set to "Login", which matches the page's function
Form Tag	Correct usage of form with action and method	Pass	Proper use of POST method and correct endpoint (authenticate.php)
Labels and Inputs	Inputs are labelled correctly for accessibility	Pass	Labels are correctly associated using for attributes

Required Fields	Uses HTML5 required attribute for validation	Pass	Ensures users cannot submit the form without input
Semantic Structure	Uses of headers and form semantics	Pass	<h2> and <form> tags correctly used for semantic clarity
Accessibility	Basic form is accessible with keyboard navigation	Pass	Basic form layout works with screen readers and keyboards

Page: sensor-data.html

Component	Criteria	Pass/Fail	Comments
DOCTYPE & HTML	Proper document declaration and HTML structure	Pass	HTML5 is correctly declared and structured
Meta Tags	Responsive design and character encoding	Pass	Includes UTF-8 and viewport settings for mobile compatibility
Title	Accurate and relevant title	Pass	Title is appropriate and descriptive for the page's purpose
WebSocket Setup	Proper initialisation and connection handing	Pass	new WebSocket() is correctly implemented and assigned to ws
Message Handling	Receives and displays live data	Pass	WebSocket onmessage event is used to update the DOM with real-time data
DOM Manipulation	Safe and direct update to a known element	Pass	Uses getElementById() to safely modify known element's content
Readability	Clear and minimal JavaScripts usage	Pass	Code is concise and easy to follow
Accessibility	Minimal impact but content is visible to screen readers	Pass	Text output is accessible via basic HTML structure

Page: create_remove_users

Component	Criteria	Pass/Fail	Comment
DOCTYPE & Meta Tags	Proper <!DOCTYPE!>, charset and responsive meta tag.	Pass	Correct HTML5 structure and mobile-friendly viewport settings
Page Title	Descriptive and accurate	Pass	Title reflects admin context: "Rakusens Admin"
Banner & Logo Section	Aligned and consistently styled	Pass	Clean layout using .banner-container and .logo-container: consistent branding
Theme Toggle UI	Present and styled for visibility	Pass	Toggle switch is properly positioned with styling in CSS
Use of Headings	Semantic HTML with clear hierarchy (<h1>, <h2>, <h3>)	Pass	Headings follow a logical order, aiding readability and accessibility
Forms & Labels	Form fields correctly labelled and grouped	Pass	Each input uses a label with corresponding for and id attributes

Class Naming (CSS)	Follows readable and modular class naming conventions	Pass	Classes like .admin-form, .form-input, .notification are clear and organised
Button & Input Styling	Buttons and inputs styled consistently with layout	Pass	Use of .btn, .login-btn, .add-user-btn implies consistency
User Table Styling	Clearly structured and visually separated	Pass	Uses classes like .user-table, .table-header, .table-data for layout clarity
Accessibility	Structure supports screen readers and keyboard navigation.	Pass	Good use of <label>, headings, and semantic tags enhances accessibility

Page: sens_visual.html

Component	Criteria	Pass/Fail	Comments
DOCTYPE & Meta Tags	Correct HTML5 structure and responsive setup	Pass	<!DOCTYPE html> and viewport for mobile compatibility are correctly included
Title & Branding	Page is appropriately titled and branded	Pass	Title matches Rakusens branding; includes logo and banner with alt text
Google Charts Setup	Loads and initialises Google Charts API properly	Pass	Charts are correctly loaded using loader.js; callback registered with setOnLoadCallback()
Chart Configuration	Pie chart styled, 3D enabled, with proper labels	Pass	Pie chart uses 3D rendering with distinct slice offsets; labels are dynamically generated
Dynamic Data Fetch	Retrieves JSON data from a backend PHP endpoint	Pass	Uses fetch() with promise-based parsing and dynamic rendering
Auto-Refresh Chart	Updates every 30 seconds	Pass	setInterval(drawChart, 30000) used to refresh chart for real-time-like data view
Theme Toggle	Toggle present; applies light/dark switching (linked to external CSS)	Pass	Theme switch UI present with class theme-switch-wrapper; assumes dark-mode styling handled externally
Accessibility	Basic accessibility (alt text, semantic headings)	Pass	Alt attribute for banner image is present; semantic headings (<h1>, <h2>)
Responsiveness	Chart container is fixed size (900px); may not scale on smaller screens	Partial Pass	Suggest using % or max-width for chart container to support mobile responsiveness
Error Handling	No catch block for fetch errors	Partial Pass	Fails silently if db_connect.php is unreachable or returns bad data

GitHub Links

https://github.com/CSSSully/Team19/tree/0bdad8220ebe468241ca976dc62f25f0c05ad4b6/Meeting%20Minutes

https://github.com/CSSSully/Team19/tree/0bdad8220ebe468241ca976dc62f25f0c05ad4b6/Software

Peer Review

Sara Ali = 10

Noreen Fatima = 10

Mohammed Suleman Riaz = 10

Sakib Iqbal = 10

Uzair Ali = 10

Sahil Farooq = 10

Shad Mortaza = 10

Sana Ullah = 4

Sara demonstrated excellent leadership skills as she communicated regularly with the team by organising regular team meetings. She worked well with Noreen on the code inspection and testing documentation to test all the html and css code as it couldn't be tested using unit testing. She wasn't assigned unit testing but stepped in to handle testing for PHP and also took care of commenting on the code, due to Sana Ullah's delay, displaying quick thinking and handled the pressure well to get the task done on time. She ensured all coding and documentation tasks were split evenly and accordingly between all team members. She ensured everyone contributed towards a part in the code and documentation. She also ensured all weekly deadlines were met and supported team members on their tasks when needed.

Noreen was outstanding at communicating and listening to team members as she attended all team meetings, she took charge of writing the meeting minutes. She wrote very detailed minutes which included task assignments for each team member and their progress updates

every week. She contributed significantly towards the testing as she worked alongside Sara to complete the code inspection on all html and css code and participated in completing the testing documentation. Since her partner had to focus on parts of the unit testing, she took lead on code inspection and handled it really well. She managed most of the code inspection and stayed on top of things, spotted issues early making sure the code inspection was done to a high standard.

Suleman demonstrated strong teamwork ability, taking lead on authentication system while also stepping in to help teammates when needed. He participated in linking the website pages to the database in order for it to run. As the team speaker he effectively raised questions and concerns from the team to the client, ensuring we stayed aligned with their needs. Suleman consistently completed his tasks on time and to a high standard, and his regular attendance and positive attitude made him an important part of the team throughout the project. In addition to this, Suleman played a large part in visualising the data that was sent to the database from our ML model. Partnered with Uzair the two took it upon themselves to test out different ways of extracting the data and making the visualisation user-friendly. While Uzair created the traffic light system and created a system in the ML model where data was sent to the MySQL database, Suleman worked with Tableau and Google Charts to clean the data and turn into pie charts.

Sakib has evoked excellent communication and skills within a team- based environment, working effectively with his peers and successfully meet his objectives required. He contributed to the admin page and helped link the website pages to the database, playing a key role in making sure everything was working smoothly. Sakib further implemented his skills through assisting in the creation of the database, and the hashing of passwords in PHP.

Uzair played an integral role in developing our sensor monitoring system, contributing significantly to both user interface design and machine learning integration. He helped build the main HTML page, delivering a responsive and visually clean layout with features like a dark mode toggle and real-time sensor data display. He also designed and implemented a traffic light classification system, using machine learning models to detect anomalies and categorise sensor readings as green, amber, or red. In addition, he created the documentation interface, presenting technical information in a clear and user-friendly manner.

Sahil was great at communication, always showing up to meetings and regularly updating his progress. Sahil was assigned to do unit testing with Sana Ullah. While Sahil worked effectively and kept the process moving forward, his partner wasn't updating him on progress. As the result, other team members had to step in to ensure the task was completed on time. Despite, the challenges, Sahil managed the situation well by continuously contacting Sana Ullah and by

completing parts of the unit testing for python. Sahil did unit testing for the python code and also commented and justified his code in the testing document.

Shad has done an outstanding job in conducting acceptance testing and significantly enhancing the usage of the traffic light system powered by machine learning models. The testing process was thorough and methodically executed, ensuring that all components met the functional requirements and worked seamlessly within the overall system. His improvements to the traffic light system not only increased its accuracy but also made it more intuitive and reliable for real-time anomaly detection. Shad's collaborative approach remained evident throughout, as he worked closely with the team to share insights and offer assistance where needed. Overall, his contributions have been instrumental in improving both the robustness and usability of the system, further demonstrating his strong command of machine learning and system integration.

Sana Ullah was assigned to do unit testing with his partner, but his communication skills were lacking, and he failed to keep his partner updated on his progress. As a result, he only managed to complete parts of the unit testing an hour before the deadline. Due to this delay, other team members had to step in and take over the task to ensure it was completed on time.