

Project Status Report

Scalable IoT-Based Forest Fire Risk Monitoring System
Student: Anjana Manoj (ID: 223483467)
Unit: SIT314: Software Architecture and Scalability for IoT
Date: September 2025

1. Introduction

Forest fires pose significant risks in Australia, where extreme temperatures, strong winds, and low rainfall can quickly escalate into devastating bushfires. Traditional fire monitoring systems are limited in scale and responsiveness.

This project aims to design a scalable IoT-based forest fire risk monitoring solution. The solution collects weather data (temperature, wind, rainfall) from distributed IoT sensor nodes, processes the data using Node-RED, and generates alerts when high-risk conditions are detected.

The final goal is to deploy this solution on Amazon Web Services (AWS) to demonstrate scalability and secure deployment. For now, progress has been made on the simulation, data collection, and flow-based processing pipeline.

2. Requirements and Scalability Concerns

- **Functional Requirements:**
 - Collect data from multiple distributed IoT sensors.
 - Process readings in near real-time.
 - Detect fire-prone conditions based on temperature, wind, and rainfall thresholds.
 - Notify stakeholders of potential risks.
- **Non-Functional Requirements:**
 - Scalability to 100+ simulated sensor nodes.
 - High availability and low latency for real-time alerts.
 - Secure data handling and communication.
- **Scalability Concerns:**
 - Large number of sensors transmitting data simultaneously.
 - Efficient filtering and aggregation of high-volume data streams.
 - Automatic scaling of microservices in AWS Lambda/ECS.
 - Avoiding bottlenecks in storage (DynamoDB final) or processing pipelines.

3. System Design and Current Implementation

The architecture is designed in stages:

1. Local Prototype: Sensor simulation, central Node.js server, Node-RED processing.
2. Cloud Deployment: DynamoDB storage, AWS Lambda microservices, scaling tests.

Current Prototype

- **Sensor Simulation (Node.js)**
Simulates temperature, wind, and rainfall values at different locations. Example (client.js):

```
const net = require('net');

const location = process.argv[2] || "Geelong";
const client = new net.Socket();

client.connect(5000, '127.0.0.1', () => {
  console.log(` Sensor at ${location} connected to server` );

  setInterval(() => {
    const reading = {
      location: location,
      timestamp: new Date().toISOString(),
      temperature: (20 + Math.random() * 15).toFixed(2), // 20–35 °C
      wind_speed: (5 + Math.random() * 20).toFixed(2), // 5–25 km/h
      rainfall: (Math.random() * 10).toFixed(2) // 0–10 mm
    };
    client.write(JSON.stringify(reading));
    console.log("Sent:", reading);
  }, 5000);
});
```

```
localhost:3000/data
pretty-print
{"location":"Geelong","timestamp":"2025-09-4T03:52:29.339Z","temperature":"29.95","wind_speed":"12.67","rainfall":"1.55"},
{"location":"Ballarat","timestamp":"2025-09-4T03:52:32.440Z","temperature":"25.65","wind_speed":"20.01","rainfall":"2.10"}]
```

- **Central Server (server.js)**

Runs a TCP server to receive sensor data and provides a REST API for Node-RED to fetch aggregated readings:

```
const net = require('net');
const express = require('express');
const app = express();

let sensorData = [];

// TCP server to receive data from sensors
const tcpServer = net.createServer(socket => {
  socket.on('data', data => {
    try {
      const reading = JSON.parse(data.toString());
      console.log("Received:", reading);
      sensorData.push(reading);
    } catch (err) {
      console.error("Invalid data:", data.toString());
    }
  });
});

tcpServer.listen(5000, () => console.log("TCP Server running on port 5000"));
```

```
// REST API for Node-RED
```

```
app.get('/data', (req, res) => {
```

```
  res.json(sensorData);
```

```
  sensorData = []; // clear after sending
```

```
});
```

```
app.listen(3000, () => console.log("REST API running on port 3000"));
```

- **Node-RED Flow**

- Polls the REST API every 5s.
- Applies filter logic for fire risk.
- Outputs alerts or confirmation of safe conditions.

```
let alerts = [];
```

```
msg.payload.forEach(reading => {
```

```
  const temp = parseFloat(reading.temperature);
```

```
  const wind = parseFloat(reading.wind_speed);
```

```
  const rain = parseFloat(reading.rainfall);
```

```
  if (temp > 30 && wind > 15 && rain < 2) {
```

```
    alerts.push(
```

```
      `ALERT | ${reading.location} | Temp: ${temp}°C | Wind: ${wind} km/h | Rain: ${rain} mm`
```

```
    );
```

```
  }
```

```
});
```

```
if (alerts.length === 0) {
```

```
  return { payload: "Conditions normal (no fire risk detected this cycle)" };
```

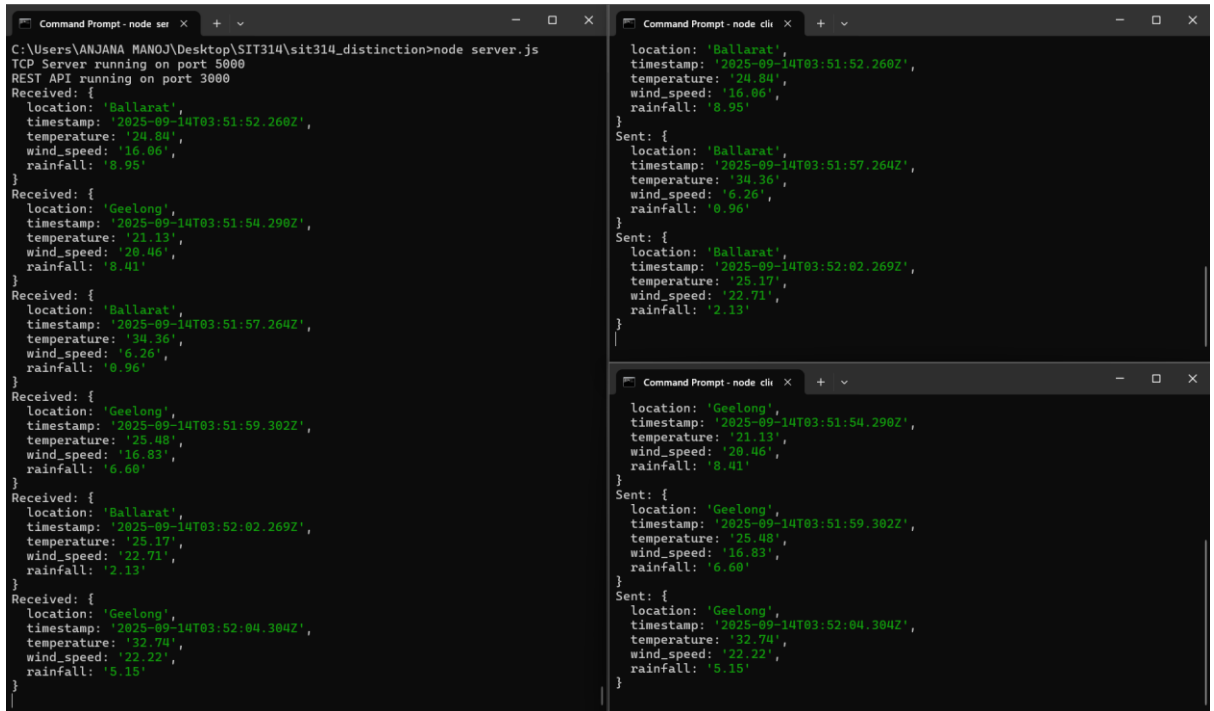
```
} else {
```

```
  return { payload: alerts };
```

```
}
```

4. Evidence of Progress

- Server Logs: receiving sensor readings continuously.



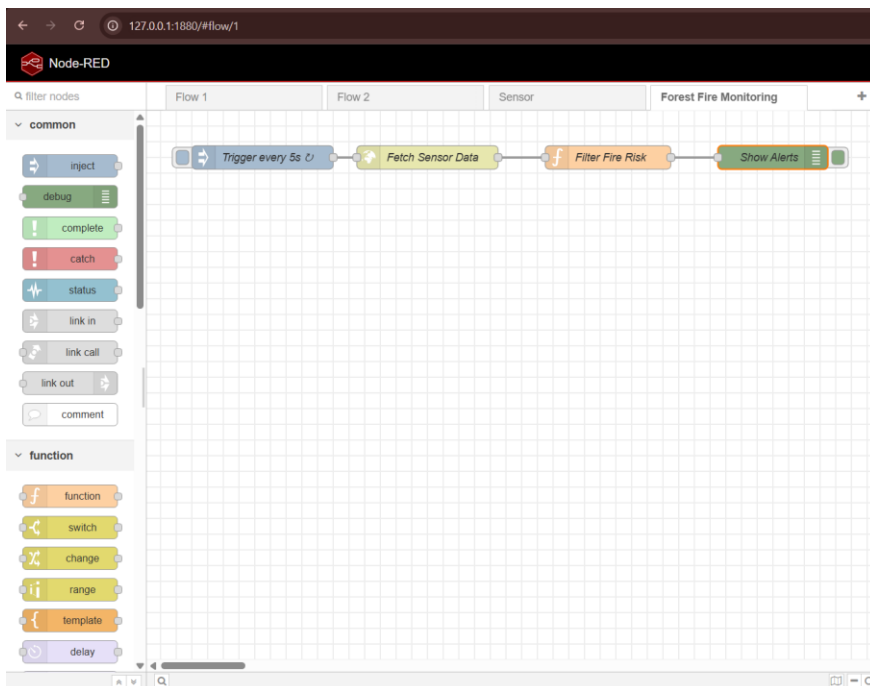
The image displays three terminal windows. The leftmost window shows the server logs for a Node.js application running on port 3000. It shows a continuous stream of received sensor data for two locations: Ballarat and Geelong. The data includes timestamp, temperature, wind speed, and rainfall. The middle and right windows show the corresponding sent data for each location, formatted as JSON objects.

```
Command Prompt - node server x + -
C:\Users\ANJANA MANOJ\Desktop\SIT314\sit314_distinction>node server.js
TCP Server running on port 3000
REST API running on port 3000
Received: {
  location: 'Ballarat',
  timestamp: '2025-09-14T03:51:52.260Z',
  temperature: '24.84',
  wind_speed: '16.06',
  rainfall: '8.95'
}
Received: {
  location: 'Geelong',
  timestamp: '2025-09-14T03:51:54.290Z',
  temperature: '21.13',
  wind_speed: '20.46',
  rainfall: '8.41'
}
Received: {
  location: 'Ballarat',
  timestamp: '2025-09-14T03:51:57.264Z',
  temperature: '34.36',
  wind_speed: '6.26',
  rainfall: '0.96'
}
Received: {
  location: 'Geelong',
  timestamp: '2025-09-14T03:51:59.302Z',
  temperature: '25.48',
  wind_speed: '16.83',
  rainfall: '6.60'
}
Received: {
  location: 'Ballarat',
  timestamp: '2025-09-14T03:52:02.269Z',
  temperature: '25.17',
  wind_speed: '22.71',
  rainfall: '2.13'
}
Received: {
  location: 'Geelong',
  timestamp: '2025-09-14T03:52:04.304Z',
  temperature: '32.74',
  wind_speed: '22.22',
  rainfall: '9.15'
}

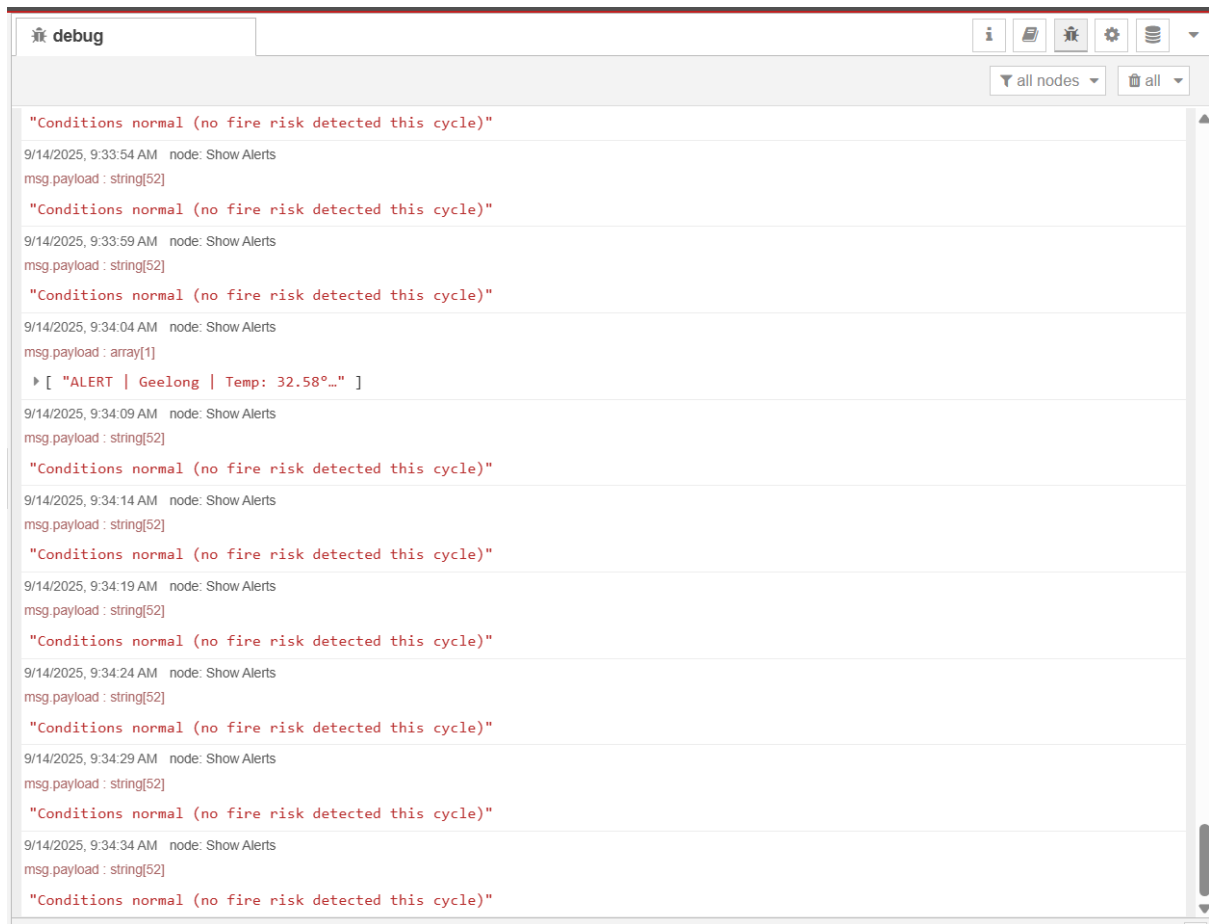
Command Prompt - node cli x + -
location: 'Ballarat',
timestamp: '2025-09-14T03:51:52.260Z',
temperature: '24.84',
wind_speed: '16.06',
rainfall: '8.95'
}
Sent: {
  location: 'Ballarat',
  timestamp: '2025-09-14T03:51:57.264Z',
  temperature: '34.36',
  wind_speed: '6.26',
  rainfall: '0.96'
}
Sent: {
  location: 'Ballarat',
  timestamp: '2025-09-14T03:52:02.269Z',
  temperature: '25.17',
  wind_speed: '22.71',
  rainfall: '2.13'
}
}

Command Prompt - node cli x + -
location: 'Geelong',
timestamp: '2025-09-14T03:51:54.290Z',
temperature: '21.13',
wind_speed: '20.46',
rainfall: '8.41'
}
Sent: {
  location: 'Geelong',
  timestamp: '2025-09-14T03:51:59.302Z',
  temperature: '25.48',
  wind_speed: '16.83',
  rainfall: '6.60'
}
Sent: {
  location: 'Geelong',
  timestamp: '2025-09-14T03:52:04.304Z',
  temperature: '32.74',
  wind_speed: '22.22',
  rainfall: '9.15'
}
}
```

- Node-RED Flow



- Alerts in Debug Panel:



5. Challenges and Fixes

- Error: Cannot find module 'express' : Fixed by running npm install express.
- ECONNREFUSED (127.0.0.1:5000) : Occurred when starting sensor before server; fixed by running server.js first.
- Empty Debug Messages : Adjusted Node-RED function to return a clear message instead of [empty].
- Output Formatting : Enhanced alerts to show full details (Temp, Wind, Rain) for better readability.

6. Next Steps

- Replace local in-memory storage with AWS DynamoDB.
- Implement AWS Lambda functions for event-driven alerts.
- Deploy Node-RED to AWS EC2/ECS for cloud-based flow processing.
- Run scalability experiments with 50+ simulated sensors.
- Configure IAM security for safe deployment.
- Publish source code on GitHub and include evidence in the final submission.

7. Conclusion

Significant progress has been made toward building a scalable IoT fire risk monitoring system. The local simulation demonstrates data collection, flow-based processing, and alert generation. The next phase will focus on AWS integration, scalability testing, and secure deployment, aligning with the unit's distinction requirements.