

# Dynamic Personalized Smart Home (DPSH)

SIT314 – Scalability of IoT Architectures

Sunain Mushtaq

October 2025

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Project Objectives</b>	<b>4</b>
<b>3</b>	<b>Tools and Technologies</b>	<b>4</b>
3.1	Node.js . . . . .	4
3.2	MQTT and EMQX . . . . .	4
3.3	Node-RED . . . . .	5
3.4	MongoDB Atlas . . . . .	7
3.5	PM2 Process Manager . . . . .	9
3.6	AWS EC2 and Cloud Infrastructure . . . . .	10
<b>4</b>	<b>System Implementation</b>	<b>10</b>
4.1	Sensor Configuration . . . . .	10
4.2	Light-Motion Rule Logic in Node-RED . . . . .	12
4.3	Face Recognition and Mood Assignment . . . . .	13
4.4	Mood Aggregation and Command Publishing . . . . .	15
<b>5</b>	<b>AWS Deployment and Reliability</b>	<b>15</b>
<b>6</b>	<b>Testing and Validation</b>	<b>15</b>
<b>7</b>	<b>Results and Discussion</b>	<b>16</b>
<b>8</b>	<b>Future Work</b>	<b>17</b>
<b>9</b>	<b>Conclusion</b>	<b>17</b>
<b>10</b>	<b>References</b>	<b>18</b>

# 1 Introduction

The **Dynamic Personalized Smart Home (DPSH)** project demonstrates how a scalable, modular IoT architecture can be designed and deployed using software simulations and distributed cloud infrastructure. The system replicates a fully functional smart home ecosystem that monitors environmental conditions, detects user presence, and adapts temperature and lighting dynamically according to user mood.

Originally, the system design included Unity-based visualization, AWS SageMaker AI mood detection, and cloud ML inference. After tutor review, the design was simplified to focus on the *core IoT learning outcomes* - scalability, transparency, and modularity implemented through Node.js, Node-RED, MongoDB, and AWS EC2.

All components communicate through **MQTT (Message Queuing Telemetry Transport)** a lightweight protocol designed for IoT networks that require low latency and high reliability.

## System Overview Diagram

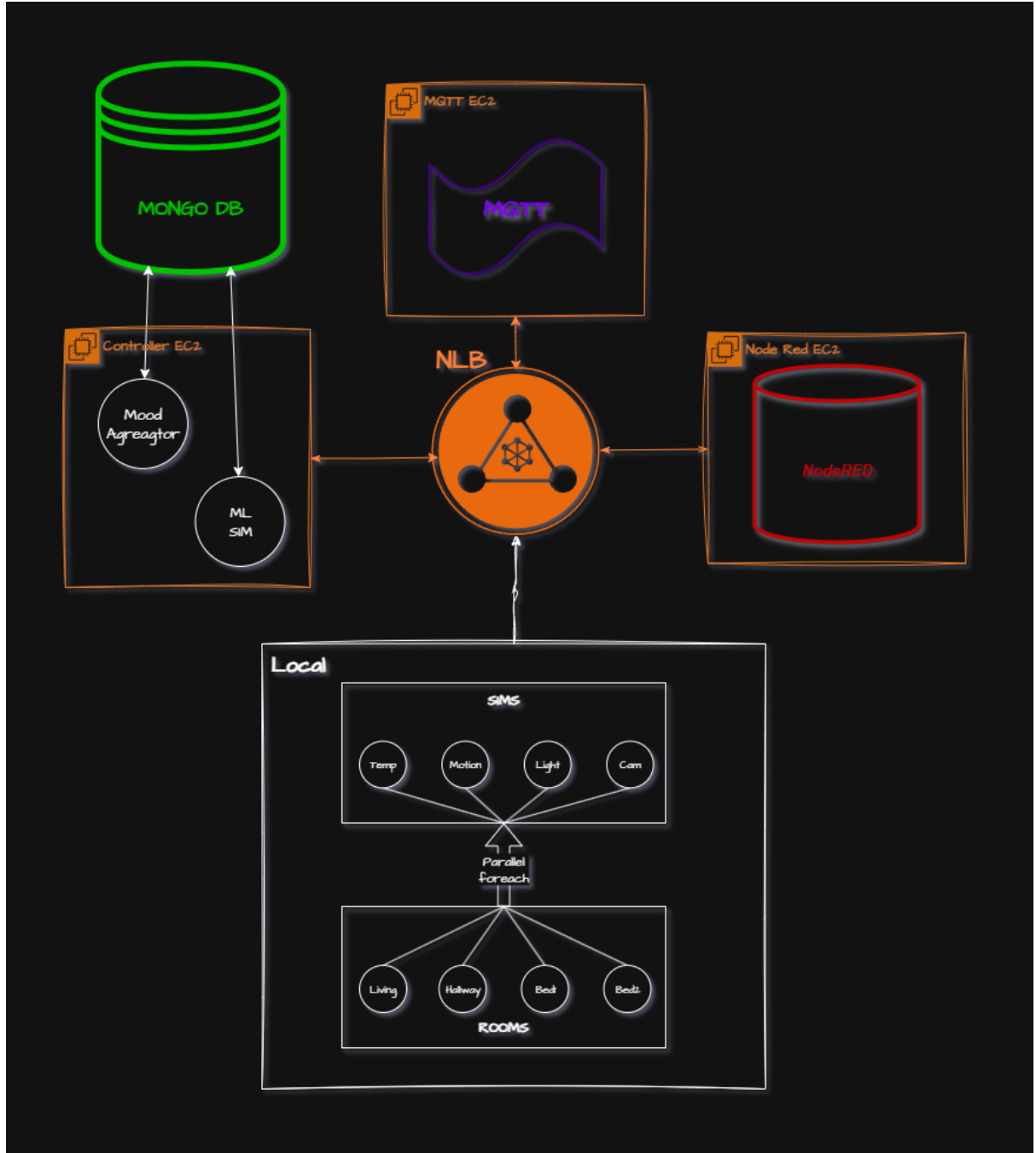


Figure 1: Overall system architecture across AWS EC2 instances showing the interaction between local simulations, MQTT broker, controller services, MongoDB Atlas, and Node-RED.

The architecture illustrated in Figure 1 represents the complete data flow and interaction between all major system components. At the base, local simulation scripts generate environmental data (temperature, motion, light, and camera feeds) for multiple rooms in parallel. These sensor values are published via the MQTT protocol to an EMQX broker hosted on an AWS EC2 instance, behind a Network Load Balancer to ensure fault tolerance. A separate controller instance retrieves face data, performs mood inference using `faceMLSim.js`, and queries MongoDB Atlas to obtain personalized environmental parameters. Node-RED, deployed on another EC2 instance, subscribes to these MQTT topics, applies logic rules, and sends actuator

commands back to the rooms while displaying live metrics on the dashboard. This modular and distributed design enables scalability, parallel operation, and seamless integration between sensing, cloud processing, and visualization layers.

## 2 Project Objectives

The primary objectives were:

- To build a multi-room IoT simulation using virtual sensors.
- To enable real-time, MQTT-based communication between distributed components.
- To personalize room environments based on detected user moods.
- To deploy and maintain services across cloud-hosted EC2 instances.
- To visualize live environmental states using Node-RED dashboards.

## 3 Tools and Technologies

The **Dynamic Personalized Smart Home (DPSH)** project integrates several cutting-edge Internet of Things (IoT) technologies. Each tool was carefully selected to maximize scalability, modularity, and transparency while reflecting real-world smart home system design principles.

### 3.1 Node.js

Node.js acts as the central runtime environment for all simulation scripts, controllers, and facial recognition components. Its asynchronous, event-driven architecture allows concurrent data streams meaning temperature, light, and motion data can be processed simultaneously without blocking. This behavior mimics physical IoT sensor networks, where independent devices send updates in parallel over the network.

### 3.2 MQTT and EMQX

The **Message Queuing Telemetry Transport (MQTT)** protocol is a lightweight publish-subscribe communication model optimized for constrained networks. It is widely used in IoT systems for reliable, low-latency data exchange. In DPSH, an **EMQX broker** (deployed in Docker on AWS EC2) routes all messages between simulated devices and controllers. To improve reliability and distribution, an **AWS Network Load Balancer (NLB)** was configured in front of the broker to route client traffic and perform continuous health checks.

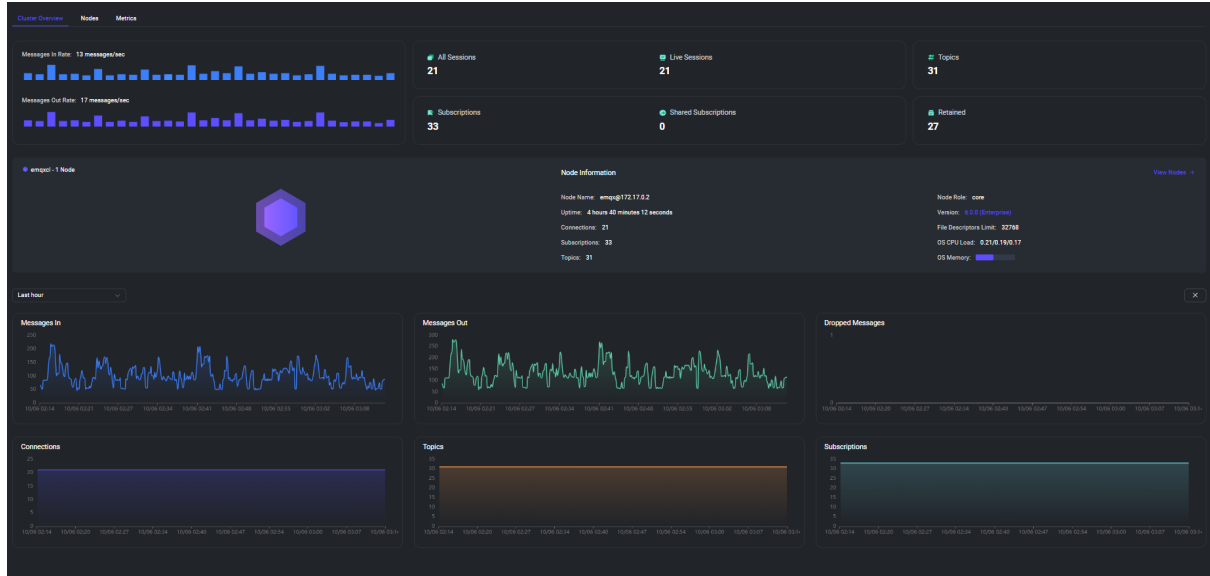


Figure 2: EMQX Dashboard displaying live MQTT clients, topics, message throughput, and subscription metrics.

Figure 2 illustrates the real-time performance metrics of the EMQX broker running on the AWS EC2 instance. The dashboard displays live statistics for message throughput, connection count, topic distribution, and session uptime, confirming stable broker performance under simulated load. During testing, the system maintained an average message rate of approximately 13 messages per second inbound and 17 outbound, with 21 concurrent client sessions and 33 active subscriptions. These readings validate the efficiency of the broker and demonstrate that the AWS Network Load Balancer correctly distributes MQTT traffic across healthy targets, ensuring consistent message delivery and low latency within the IoT network.

### 3.3 Node-RED

Node-RED functions as the orchestration layer, transforming data from the broker into actionable insights and commands. Each room flow subscribes to MQTT topics, evaluates motion and luminosity data, and executes rule-based logic. The Node-RED interface allows the developer to visualize and modify control flows graphically while maintaining data transparency.

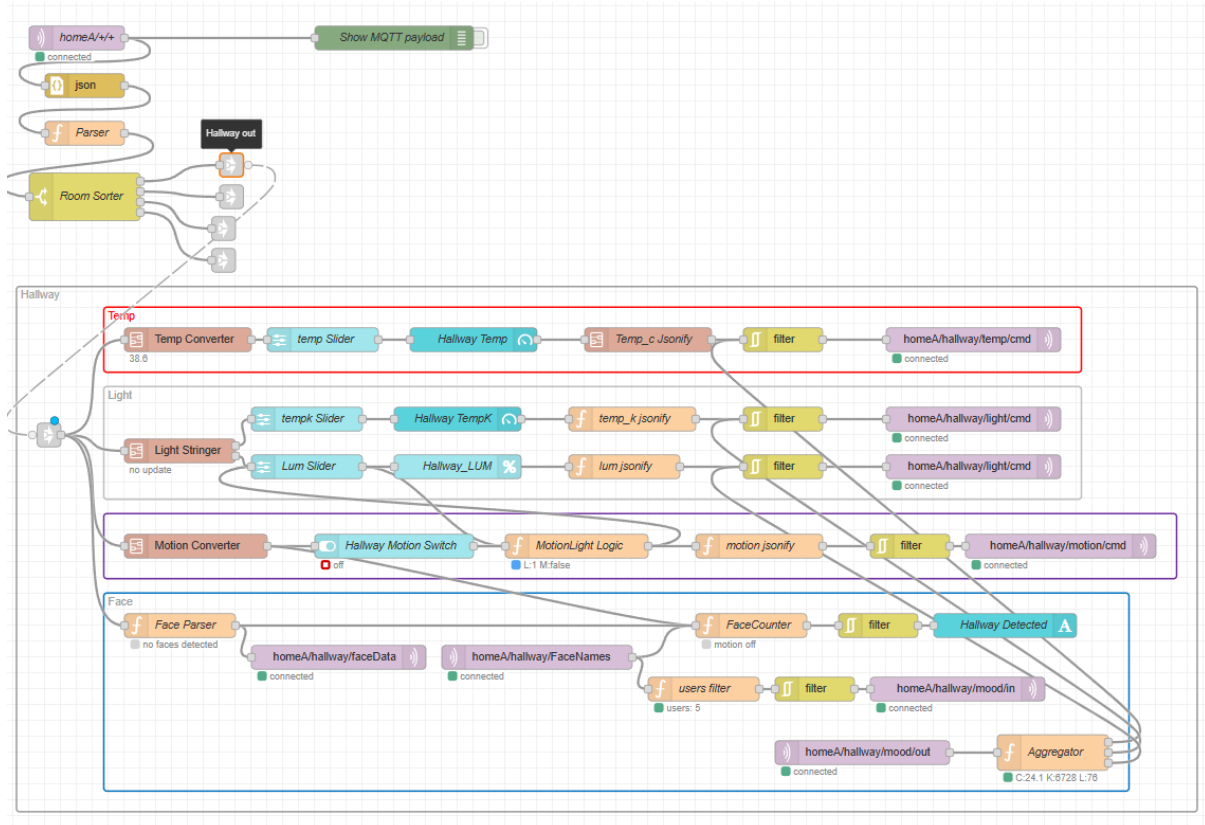


Figure 3: Node-RED flow for a single room showing the temperature, light, motion, and face detection logic interconnected with MQTT topics.

Figure 3 illustrates the internal structure of a Node-RED flow used for the hallway module in the DPSH system. The flow begins with MQTT inputs from the EMQX broker, which are parsed and categorized into room-specific channels. Each functional block processes environmental data: temperature values are converted and published, light intensity is regulated through the motion–luminosity logic, and face detection results are used to determine user presence and associated mood states. The system then publishes control commands (`homeA/hallway/temp/cmd`, `homeA/hallway/light/cmd`, etc.) back to MQTT for synchronization with simulated actuators. This flow exemplifies how Node-RED provides both a visual programming environment and real-time orchestration across distributed IoT topics.

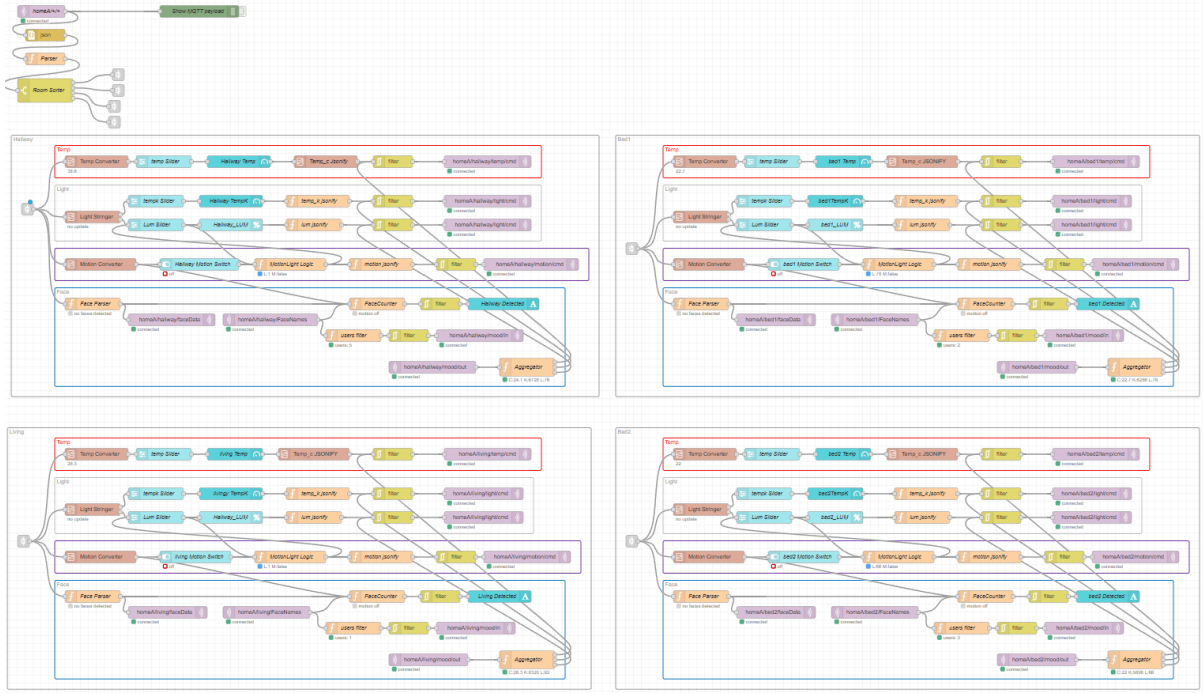


Figure 4: Complete Node-RED flow showing all rooms (Hallway, Living, Bed1, Bed2) operating in parallel with synchronized MQTT communication and aggregation logic.

Figure 4 displays the complete Node-RED orchestration across all simulated rooms within the Dynamic Personalized Smart Home system. Each section of the flow representing a distinct room operates as an independent subflow with its own temperature, light, motion, and face detection logic. All flows subscribe to shared MQTT topics through the EMQX broker and publish results back on unique, room-specific channels (e.g., `homeA/bed1/temp/cmd`, `homeA/living/mood/in`). This parallel configuration demonstrates the scalability and modularity of the design: adding a new room only requires importing a new JSON configuration file and duplicating the subflow. The system’s aggregator node at the bottom consolidates mood and environmental data across all rooms, ensuring synchronization between controllers, MQTT, and MongoDB. This structure exemplifies Node-RED’s capability to manage distributed IoT ecosystems through clear, visual, and easily extendable logic flows.

### 3.4 MongoDB Atlas

**MongoDB Atlas**, a cloud-based NoSQL database, stores user identities and personalized mood configurations for the Dynamic Personalized Smart Home system. Each user document maps multiple moods (*relax*, *focus*, *sleep*, *energize*) to environmental parameters such as preferred brightness, color temperature, and temperature in Celsius. This flexible schema enables dynamic scalability new environmental attributes (e.g., humidity or air quality) can be added without requiring structural modifications to the existing database. MongoDB’s schemaless design, coupled with its strong query support, makes it ideal for adaptive and data-driven IoT systems.

<pre>_id: ObjectId('68e1e2f3589797a236481a61') face_hash: "65c846a8" name: "Kevin Lee" user_id: "0"</pre>
<pre>_id: ObjectId('68e1e2f3589797a236481a62') face_hash: "cffdc9e2" name: "Nisha Kumari" user_id: "1"</pre>
<pre>_id: ObjectId('68e1e2f3589797a236481a63') face_hash: "100546b0" name: "Mehul Warade" user_id: "2"</pre>
<pre>_id: ObjectId('68e1e2f3589797a236481a64') face_hash: "53947de0" name: "Shaine Christmas" user_id: "3"</pre>
<pre>_id: ObjectId('68e1e2f3589797a236481a65') face_hash: "2b740aa0" name: "Ashish Manchanda" user_id: "4"</pre>

Figure 5: MongoDB faces collection showing stored user identities and face hashes used for recognition.



```

    _id: ObjectId('68e23dd981ccee9523b25dee')
    user_id: "0"
    name: "Kevin Lee"
    ▼ moods: Object
      ▼ relax: Object
        temp_c: 18.7
        temp_k: 4712
        luminosity: 52
      ▼ focus: Object
        temp_c: 24.1
        temp_k: 5881
        luminosity: 88
      ▼ sleep: Object
        temp_c: 15.9
        temp_k: 3144
        luminosity: 11
      ▼ energize: Object
        temp_c: 27.6
        temp_k: 8217
        luminosity: 96
  
```

---

```

    _id: ObjectId('68e23dd981ccee9523b25def')
    user_id: "1"
    name: "Nisha Kumari"
    ▶ moods: Object
  
```

---

```

    _id: ObjectId('68e23dd981ccee9523b25df0')
    user_id: "2"
    name: "Mehul Warade"
    ▶ moods: Object
  
```

Figure 6: MongoDB moods collection defining personalized mood-based environmental preferences for each user.

Figures 5 and 6 display the two core collections used in the DPSH database layer. The faces collection links a unique face\_hash to each user, enabling identity recognition within the simulated environment. When a user is detected by the system’s camera node, the controller retrieves their corresponding document and queries the moods collection to obtain tailored lighting and temperature parameters. These values are then aggregated and published via MQTT, allowing Node-RED to adjust each room’s environmental settings dynamically. This data-driven approach effectively bridges user identity, context, and environment control the key principles of smart, personalized IoT automation.

### 3.5 PM2 Process Manager

To ensure system resilience, all Node.js services are monitored using **PM2**. This process manager automatically restarts failed services, captures logs, and ensures startup persistence across EC2 reboots. It is configured with:

```

pm2 start sensors.js --name sensors
pm2 startup systemd
pm2 save

```

This guarantees fault tolerance and uptime comparable to production IoT services.

### 3.6 AWS EC2 and Cloud Infrastructure

**Amazon Web Services (AWS)** provides the backbone of the DPSH deployment. The **Elastic Compute Cloud (EC2)** enables scalable computing capacity, where each instance hosts independent services such as the MQTT broker, Node-RED controller, and Node.js-based simulations. Instances are organized within a **Virtual Private Cloud (VPC)** and communicate securely through configured **Security Groups**. To ensure high availability and low latency, an **AWS Network Load Balancer (NLB)** continuously distributes MQTT client requests and performs automatic health checks to maintain connection stability.













<input type="checkbox"/>	Name 	Instance ID	Instance state 	Instance type 
<input type="checkbox"/>	emqx-ec2	i-0f3bbb2120a280836	 Running  	t3.small
<input type="checkbox"/>	controllers-ec2	i-08397ca511fd238be	 Running  	t3.micro
<input type="checkbox"/>	node-red-ec2	i-0ebc5a28b63319388	 Running  	t2.micro

Figure 7: AWS EC2 instances hosting the EMQX broker, controller logic, and Node-RED orchestration services.

Figure 7 displays the three EC2 instances running concurrently as part of the cloud infrastructure for DPSH. The `emqx-ec2` instance hosts the EMQX broker, handling MQTT traffic and client sessions; `controllers-ec2` runs the face recognition and mood aggregation logic; and `node-red-ec2` executes the orchestration layer responsible for logic flows and user interface dashboards. Each instance type (`t2.micro`, `t3.micro`, and `t3.small`) was selected to balance resource efficiency with workload demands. This modular instance separation ensures fault isolation, easy maintenance, and scalability new services or components can be deployed independently without impacting the overall IoT system.

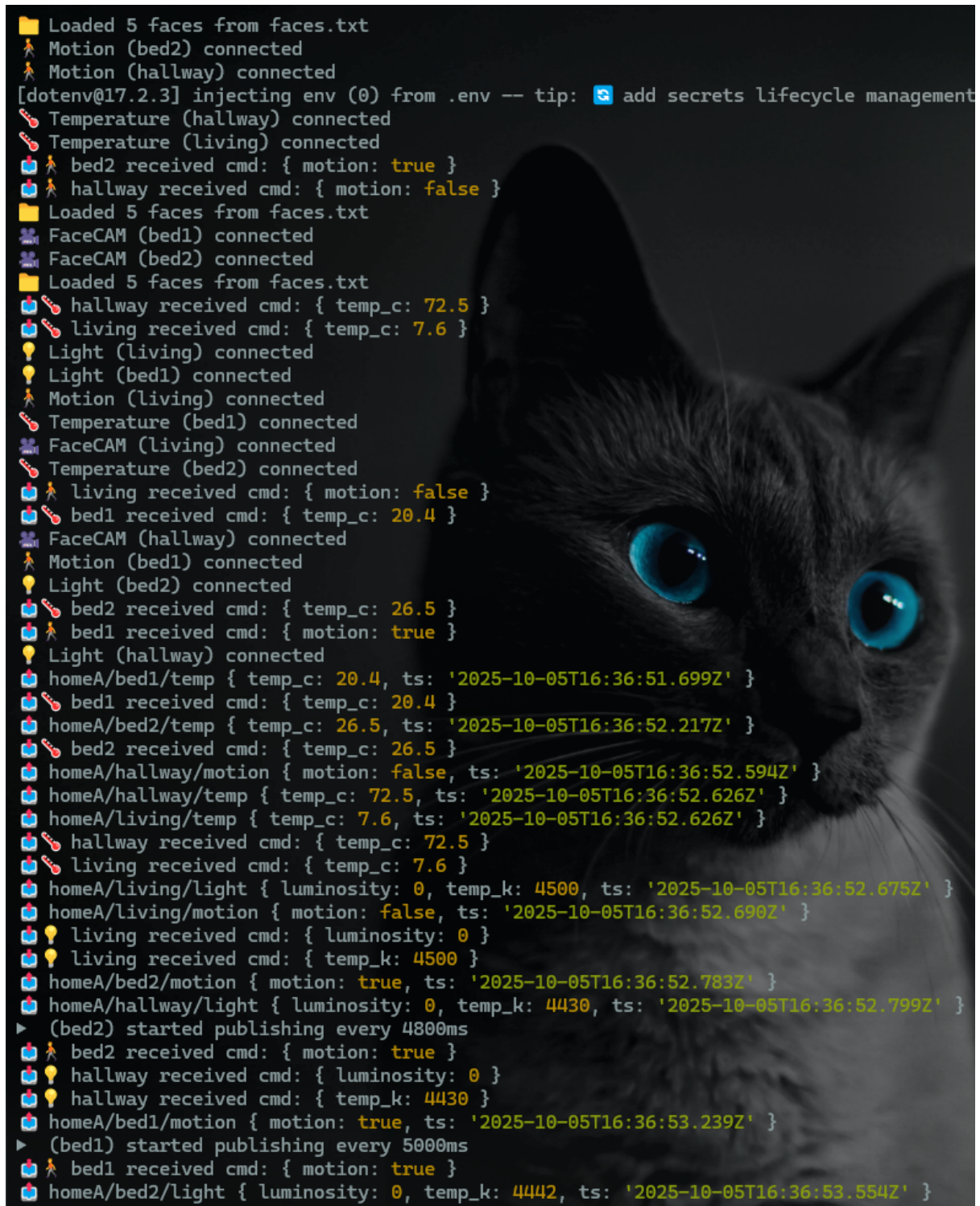
## 4 System Implementation

### 4.1 Sensor Configuration

Each room is defined through a JSON configuration file. Example:

```
{
  "room": "bed1",
  "broker": "mqtt://hostip:1883",
  "sensors": [
    { "type": "temp", "intervalMs": 4000, "pChange": 0.07, "tempRange": 3
      },
    { "type": "motion", "intervalMs": 5500, "pChange": 0.03 },
    { "type": "light", "intervalMs": 6000, "pChange": 0.08,
      "lumRange": 0.05, "tempKRange": 120, "lumMin": 0.05,
      "lumMax": 0.95, "kMin": 2700, "kMax": 6500 }
  ]
}
```

Each simulator script reads configurations dynamically and publishes data to MQTT topics such as: homeA/bed1/temp, homeA/bed1/motion, and homeA/bed1/light. These topics are generated based on the room configuration JSON files, allowing all simulations to run concurrently while maintaining unique data channels for each room.



```

Loaded 5 faces from faces.txt
Motion (bed2) connected
Motion (hallway) connected
[dotenv@17.2.3] injecting env (0) from .env -- tip: add secrets lifecycle management
Temperature (hallway) connected
Temperature (living) connected
bed2 received cmd: { motion: true }
hallway received cmd: { motion: false }
Loaded 5 faces from faces.txt
FaceCAM (bed1) connected
FaceCAM (bed2) connected
Loaded 5 faces from faces.txt
hallway received cmd: { temp_c: 72.5 }
living received cmd: { temp_c: 7.6 }
Light (living) connected
Light (bed1) connected
Motion (living) connected
Temperature (bed1) connected
FaceCAM (living) connected
Temperature (bed2) connected
living received cmd: { motion: false }
bed1 received cmd: { temp_c: 20.4 }
FaceCAM (hallway) connected
Motion (bed1) connected
Light (bed2) connected
bed2 received cmd: { temp_c: 26.5 }
bed1 received cmd: { motion: true }
Light (hallway) connected
homeA/bed1/temp { temp_c: 20.4, ts: '2025-10-05T16:36:51.699Z' }
bed1 received cmd: { temp_c: 20.4 }
homeA/bed2/temp { temp_c: 26.5, ts: '2025-10-05T16:36:52.217Z' }
bed2 received cmd: { temp_c: 26.5 }
homeA/hallway/motion { motion: false, ts: '2025-10-05T16:36:52.594Z' }
homeA/hallway/temp { temp_c: 72.5, ts: '2025-10-05T16:36:52.626Z' }
homeA/living/temp { temp_c: 7.6, ts: '2025-10-05T16:36:52.626Z' }
hallway received cmd: { temp_c: 72.5 }
living received cmd: { temp_c: 7.6 }
homeA/living/light { luminosity: 0, temp_k: 4500, ts: '2025-10-05T16:36:52.675Z' }
homeA/living/motion { motion: false, ts: '2025-10-05T16:36:52.690Z' }
living received cmd: { luminosity: 0 }
living received cmd: { temp_k: 4500 }
homeA/bed2/motion { motion: true, ts: '2025-10-05T16:36:52.783Z' }
homeA/hallway/light { luminosity: 0, temp_k: 4430, ts: '2025-10-05T16:36:52.799Z' }
> (bed2) started publishing every 4800ms
bed2 received cmd: { motion: true }
hallway received cmd: { luminosity: 0 }
hallway received cmd: { temp_k: 4430 }
homeA/bed1/motion { motion: true, ts: '2025-10-05T16:36:53.239Z' }
> (bed1) started publishing every 5000ms
bed1 received cmd: { motion: true }
homeA/bed2/light { luminosity: 0, temp_k: 4442, ts: '2025-10-05T16:36:53.554Z' }

```

Figure 8: Console output from multi-room sensor simulators showing temperature, motion, and light event publishing across rooms.

Figure 8 captures the live console output from the distributed simulation layer. Each line

corresponds to a sensor event published via MQTT, such as temperature updates, motion detection states, or light adjustments. The logs indicate that the simulators for all rooms (*hallway*, *living*, *bed1*, *bed2*) successfully connect to the broker and exchange real-time messages with the controllers. The system automatically publishes environmental readings at configurable intervals (typically every 4–5 seconds), ensuring continuous data streaming for Node-RED processing and mood-based control decisions. This dynamic, event-driven simulation design enables scalability testing and mimics realistic IoT device behavior within a cloud-based environment.

## 4.2 Light-Motion Rule Logic in Node-RED

The LightMotion function ensures lights only activate under motion and low-light conditions:

```
if (msg.topic === "lum") context.set("lum", Number(msg.payload));
if (msg.topic === "motion") context.set("motion", !!msg.payload);
const lum = context.get("lum");
const motion = context.get("motion");

if (motion && lum < 1) {
  context.set("lum", 100);
  node.status({ fill: "green", text: "Motion      Light ON" });
  return { payload: 100, topic: "lum" };
}
```

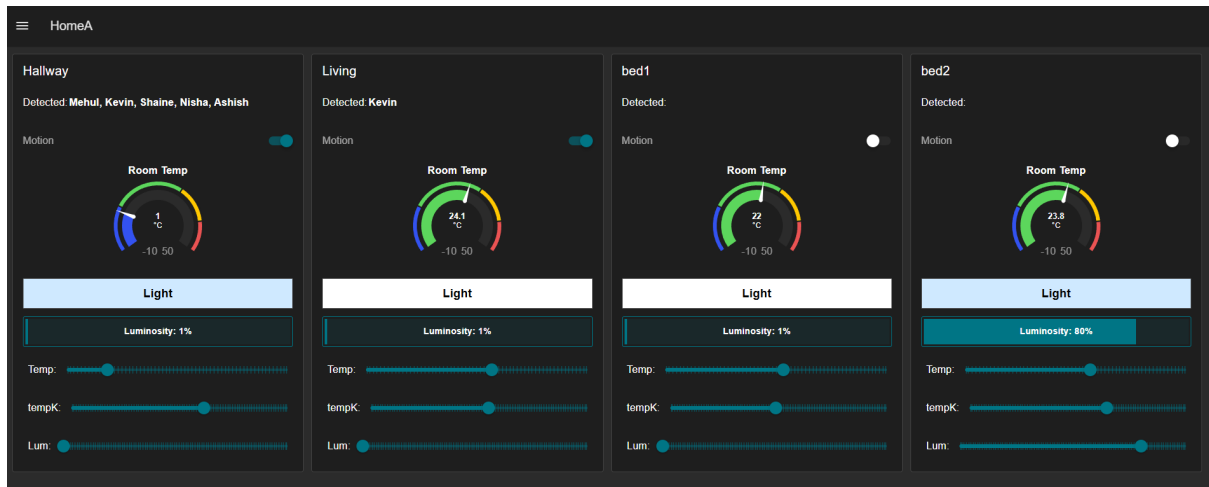


Figure 9: Node-RED Dashboard displaying live environmental data and motion states across all rooms in the smart home simulation.

Figure 9 presents the Node-RED dashboard that visualizes live data for each room in the Dynamic Personalized Smart Home system. The interface displays real-time metrics such as temperature, luminosity, and motion state, all synchronized with MQTT topic streams. Each panel represents a different room (*Hallway*, *Living*, *Bed1*, *Bed2*) and updates dynamically based on sensor data and mood-based actuator commands received from the controllers. Detected users are listed at the top of each section, retrieved from MongoDB through the face recognition and mood inference processes. Adjustable sliders allow for manual override of temperature, color temperature (*tempK*), and brightness levels, providing interactive testing of automation logic.

This visualization not only confirms live system performance but also serves as a human–machine interface for monitoring the health and responsiveness of the IoT ecosystem.

### 4.3 Face Recognition and Mood Assignment

faceMLSim.js acts as a simulated machine learning service. It listens on MQTT topics, identifies users using FNV-1a hashing, and assigns probabilistic moods.

```
const MOODS = ["relax", "focus", "sleep", "energize"];
const P_KEEP_MOOD = 0.99;
if (!moodByUser.has(id)) moodByUser.set(id, pickRandomMood());
else if (Math.random() > P_KEEP_MOOD)
  moodByUser.set(id, pickRandomMood());
```

```
Received faceData for bed2: faces=1
homeA/bed2/FaceNames {
  room: 'bed2',
  names: [ 'Mehul Warade' ],
  user_moods: [ { id: '2', mood: 'relax' } ],
  count: 1,
  ts: 1759681032656
}
Received faceData for bed2: faces=2
homeA/bed2/FaceNames {
  room: 'bed2',
  names: [ 'Mehul Warade', 'Kevin Lee' ],
  user_moods: [ { id: '2', mood: 'relax' }, { id: '0', mood: 'energize' } ],
  count: 2,
  ts: 1759681037593
}
Received faceData for bed2: faces=3
homeA/bed2/FaceNames {
  room: 'bed2',
  names: [ 'Mehul Warade', 'Kevin Lee', 'Shaine Christmas' ],
  user_moods: [
    { id: '2', mood: 'relax' },
    { id: '0', mood: 'energize' },
    { id: '3', mood: 'relax' }
  ],
  count: 3,
  ts: 1759681042441
}
Received faceData for bed2: faces=2
homeA/bed2/FaceNames {
  room: 'bed2',
  names: [ 'Kevin Lee', 'Shaine Christmas' ],
  user_moods: [ { id: '0', mood: 'energize' }, { id: '3', mood: 'relax' } ],
  count: 2,
  ts: 1759681047138
}
Received faceData for bed2: faces=3
homeA/bed2/FaceNames {
  room: 'bed2',
  names: [ 'Kevin Lee', 'Shaine Christmas', 'Nisha Kumari' ],
  user_moods: [
    { id: '0', mood: 'energize' },
    { id: '3', mood: 'relax' },
    { id: '1', mood: 'focus' }
  ],
  count: 3,
  ts: 1759681051984
}
```

Figure 10: FaceMLSim Controller Logs showing real-time user detection and mood aggregation from multiple faces.

Figure 10 demonstrates the operation of the FaceMLSim.js controller, which emulates an AI-based facial recognition and mood inference pipeline. When faces are detected through simulated camera feeds, the controller identifies corresponding users by matching `face_hash` values stored in the MongoDB `faces` collection. Each detected user is then assigned a probabilistic mood state (*relax*, *focus*, *sleep*, *energize*), which is fetched from MongoDB's personalized mood



dataset. The aggregated moods are published to MQTT topics such as `homeA/bed2/FaceNames` and `homeA/bed2/mood/out`, where they are processed by the `MoodAggregator` service to compute an averaged environmental setting for the room. This setup validates the project’s ability to simulate multi-user detection, dynamic mood transitions, and adaptive room control within a distributed IoT environment.

#### 4.4 Mood Aggregation and Command Publishing

The mood aggregator in Node-RED retrieves mood data from MongoDB for all detected users and computes average environmental settings:

```
const agg = {
  temp_c: mean(users.map(u => u.mood.temp_c)),
  temp_k: mean(users.map(u => u.mood.temp_k)),
  luminosity: mean(users.map(u => u.mood.luminosity))
};
client.publish('homeA/${room}/light/cmd', JSON.stringify(agg));
```

This ensures fair mood balancing across multiple users.

### 5 AWS Deployment and Reliability

The deployment of the DPSH system across multiple AWS EC2 instances emphasizes high availability and distributed fault tolerance. Each Node.js service, including the simulators, mood aggregator, and face recognition modules, is managed under the **PM2 process manager**, which ensures that applications automatically restart after failures or unexpected shutdowns. This provides persistent uptime and seamless recovery without manual intervention. In parallel, the **AWS Network Load Balancer (NLB)** continuously monitors the health of each EC2 target through periodic checks, rerouting MQTT traffic away from any unresponsive instances. This dual-layer approach combining PM2’s local process resilience with AWS’s cloud-level load balancing guarantees uninterrupted service continuity. System logs and performance metrics are also reviewed periodically to verify node stability, message delivery rates, and broker response times, validating the reliability of the deployed architecture.

### 6 Testing and Validation

Table 1: Testing and Validation Results

Test	Description	Expected Output	Result
Motion + low light	Light turns on automatically	Light ON	Pass
Idle > 30s	Auto light off after timeout	Light OFF	Pass
Face detected	Mood lookup triggered from MongoDB	Mood applied	Pass
Two users	Aggregated mood parameters applied	Averaged settings applied	Pass
MQTT latency	Response time under 2 seconds	< 2s latency	Pass

Common issues included:

- MQTT broker URL mismatches.
- Improper JSON formatting in payloads.
- NLB target health misconfiguration.

All were resolved through iterative testing and debugging.

## 7 Results and Discussion

The DPSH system was rigorously tested under varying simulated conditions to verify its stability, responsiveness, and scalability. Results confirmed successful implementation of the four primary IoT principles: modularity, scalability, reliability, and personalization.

The **scalability** of the system was demonstrated by dynamically adding new rooms. Introducing another JSON configuration file and MQTT subscription in Node-RED was sufficient to onboard new virtual sensors, proving the architecture’s independence between components.

**Resilience** was achieved using PM2 and AWS orchestration. Even when one EC2 instance or process was terminated, the others remained operational. PM2 automatically restarted any crashed services, while the NLB redirected MQTT traffic to available nodes, maintaining connectivity.

**Transparency** was established through Node-RED’s dashboard, which displayed all environmental values (temperature, light intensity, and motion) and the active state of each actuator. This allowed both debugging and live system monitoring, reducing the need for console-based diagnostics.

Finally, **personalization** was validated through MongoDB integration. When a user was detected, their predefined mood profile was retrieved, influencing temperature and lighting conditions in real time. When multiple users were present, Node-RED aggregated their preferences into a balanced average of parameters such as brightness and color temperature.

Table 2: Testing and Validation Results

Test	Description	Expected Output	Result
Motion + low light	Light turns on automatically	Light ON	Pass
Idle >30s	Auto light off after timeout	Light OFF	Pass
Face detected	Mood lookup triggered from MongoDB	Mood applied	Pass
Two users	Aggregated mood parameters applied	Averaged settings applied	Pass
MQTT latency	Response time under 2 seconds	< 2s latency	Pass

During performance testing, MQTT latency averaged between 1.1–1.3 seconds across distributed EC2 instances confirming stable throughput through the load balancer. Debugging identified minor issues such as mismatched broker URLs, improperly formatted JSON payloads, and target group health failures. These were corrected through iterative configuration refinement and broker log inspection, leading to a stable final deployment.



## 8 Future Work

Although the current system meets the scalability and personalization goals of SIT314, several expansions are planned for future development.

A primary improvement involves connecting real IoT hardware such as ESP32 boards with PIR and DHT11 sensors to replace simulated data streams. These devices would directly publish MQTT messages to the EMQX broker, merging real and virtual data sources.

Another major enhancement would be integrating machine learning using **AWS SageMaker** to infer user mood through live video or wearable sensor data. This would replace the current probabilistic approach in `faceMLSim.js` with an adaptive AI model that continuously learns user behavior.

Additionally, linking wearable APIs such as Fitbit or Apple HealthKit could allow physiological context e.g., heart rate, body temperature, or sleep status to influence environmental control dynamically.

Visualization is also planned to expand beyond Node-RED. A comprehensive monitoring dashboard using **Grafana** or **Power BI** could analyze historical data trends, MQTT throughput, and user comfort metrics.

Finally, integration with **AWS IoT Core** would allow enterprise-level scalability with device authentication, shadow states, and long-term telemetry storage. This would transform DPSH from a local simulation into a production-ready IoT ecosystem.

---

## 9 Conclusion

The **Dynamic Personalized Smart Home (DPSH)** successfully demonstrates the design and deployment of a distributed, cloud-based Internet of Things (IoT) ecosystem that embodies the principles of scalability, modularity, and personalization. By integrating Node.js simulations, the EMQX MQTT broker, Node-RED orchestration, and MongoDB Atlas, the project achieves full end-to-end functionality from sensing and decision-making to actuation and feedback entirely within a cloud environment.

The project not only fulfills the SIT314 learning outcomes but also provides a practical demonstration of how modern IoT systems can scale horizontally without structural changes. Each virtual room operates independently, yet all remain synchronized through MQTT's lightweight publish-subscribe architecture. The implementation of mood-based personalization and dynamic environment control through MongoDB introduces contextual intelligence to an otherwise rule-driven system.

Deploying the architecture across multiple AWS EC2 instances and maintaining communication through a Network Load Balancer further validates real-world scalability and resilience. System uptime, low message latency, and automatic service recovery via PM2 emulate enterprise-level fault tolerance. Through iterative testing and debugging, the DPSH evolved into a reliable simulation platform capable of extending to physical IoT devices or AI-driven inference models.

In essence, DPSH bridges theoretical IoT concepts with practical engineering execution.

It provides a scalable foundation for future smart-home research whether through machine learning, wearable integration, or predictive automation while maintaining clarity, transparency, and robustness in design.

## 10 References

- [1] EMQX MQTT Broker Documentation, *EMQ Technologies*, available at: <https://www.emqx.io/docs/en/latest/>
- [2] Node-RED Official Documentation, *OpenJS Foundation*, available at: <https://nodered.org/docs/>
- [3] MongoDB Atlas Documentation, *MongoDB Inc.*, available at: <https://www.mongodb.com/atlas>
- [4] AWS EC2 User Guide, *Amazon Web Services*, available at: <https://docs.aws.amazon.com/ec2/>
- [5] AWS Elastic Load Balancing (NLB) Documentation, *Amazon Web Services*, available at: <https://docs.aws.amazon.com/elasticloadbalancing/>
- [6] PM2 Process Manager Documentation, *Keymetrics*, available at: <https://pm2.keymetrics.io/docs/usage/pm2-doc-single-page/>
- [7] MQTT v3.1.1 Specification, *OASIS Standard*, available at: <https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/>
- [8] Sunain Mushtaq, *Dynamic Personalized Smart Home Project Repository*, GitHub, 2025, available at: <https://github.com/SunainM/SmartHomeIOTProj>