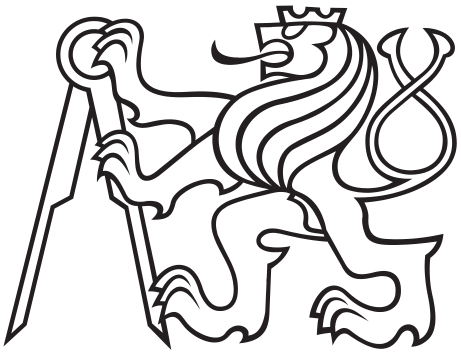
Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Microelectronics



Application of Servers and Unix-like Systems for Sensor Control in Smart Homes

Bachelor thesis

Weize Yuan

Study program: Electrical Engineering and Computer Science

Supervisor: prof. Ing. Miroslav Husák, CSc.

Prague 2026

Thesis Supervisor:
prof. Ing. Miroslav Husák, CSc.
Department of Microelectronics
Faculty of Electrical Engineering
Czech Technical University in Prague
Technická 2
160 00 Prague 6
Czech Republic

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Yuan Weize** Personal ID number: **485402**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Electrical Power Engineering**

Study program: **Electrical Engineering and Computer Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Application of Servers and Unix-like Systems for Sensor Control in Smart Homes

Bachelor’s thesis title in Czech:

Využití serverů a unixových systémů pro řízení senzorů v chytrých domácnostech

Name and workplace of bachelor’s thesis supervisor:

prof. Ing. Miroslav Husák, CSc. Department of Microelectronics FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **01.10.2025** Deadline for bachelor thesis submission: **06.01.2026**

Assignment valid until: **19.09.2027**

Head of department’s signature

Vice-dean´s signature on behalf of the Dean

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work.
The student must produce his thesis without the assistance of others, with the exception of provided consultations.
Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Yuan Weize

Student’s signature

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Yuan Weize** Personal ID number: **485402**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Electrical Power Engineering**

Study program: **Electrical Engineering and Computer Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Application of Servers and Unix-like Systems for Sensor Control in Smart Homes

Bachelor’s thesis title in Czech:

Využití serverů a unixových systémů pro řízení senzorů v chytrých domácnostech

Guidelines:

1. Analyze the role and advantages of using servers and Unix-like systems in a smart home. Focus the analysis on applications for monitoring and controlling sensor systems, e.g. for Home Assistant needs.
2. Design a specific model using a server to manage basic smart home functions and explore possibilities for intergrating IoT devices solutions. Use a very simple sensor system setup to demonstrate the model's function.
3. Evaluate the parameters of the proposed system and compare it with typical commercial solutions.

Bibliography / sources:

1. Bhatnagar, H. et al, Implementation model of Wi-Fi based Smart Home System. 2018 ICACCE.
2. Khan, A. et al, Design of an IoT smart home system. 2018 L&T.
3. Xiao, Z. et al, Design of Home Appliance Control System in Smart Home based on WiFi IoT. 2018 IAEAC.
4. Zeus Integrated. Smart Home Automation. 2021, https: //zeusintegrated.com/solutions/smart-home-automation.

FAKULTA ELEKTROTECHNICKÁ
FACULTY OF ELECTRICAL ENGINEERING
Technická 2
166 27 Praha 6

DECLARATION

I, the undersigned

Student's surname, given name(s):	Yuan Weize
Personal number:	485402
Programme name:	Electrical Engineering and Computer Science

declare that I have elaborated the bachelor's thesis entitled

Application of Servers and Unix-like Systems for Sensor Control in Smart Homes

independently, and have cited all information sources used in accordance with the Methodological Instruction on the Observance of Ethical Principles in the Preparation of University Theses and with the Framework Rules for the Use of Artificial Intelligence at CTU for Academic and Pedagogical Purposes in Bachelor's and Continuing Master's Programmes.

I declare that I used artificial intelligence tools during the preparation and writing of this thesis. I verified the generated content. I hereby confirm that I am aware of the fact that I am fully responsible for the contents of the thesis.

In Prague on 06.01.2026

Weize Yuan

.....

student's signature

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Ing. Miroslav Husák, CSc., for the initial discussions that helped shape the topic and scope of this thesis, as well as for his valuable high-level guidance.

I would also like to acknowledge the Faculty of Electrical Engineering and the Czech Technical University as a whole for providing an academic environment in which I was able to develop not only foundational theoretical knowledge and practical skills, but also the ability to think independently, adapt, and grow—competencies essential for future professional development and for functioning in society.

Finally, I am deeply grateful to my family for their patience, encouragement, and rock-solid support throughout my studies and my life. Their support has been a constant source of strength and motivation, without which this work would not have been possible.

Abstract

This thesis investigates the use of server-based architectures and Unix-like operating systems for sensor monitoring and control in smart home environments. The objective is to demonstrate how an open-source, self-hosted system can provide secure, scalable, and transparent management of distributed sensor devices without reliance on proprietary cloud services.

The proposed system integrates physical sensor nodes based on ESP32 and ESP32-S3 microcontrollers—the latter implementing edge computing algorithms for real-time fall detection—with a Python-based sensor simulator used to emulate a large number of devices for scalability testing. Communication between IoT components is primarily realized using the MQTT protocol with mutual Transport Layer Security (mTLS), complemented by native API integrations within the Home Assistant platform for device management and automation. An EMQX broker deployed on a remote server acts as the central messaging component. Home Assistant serves as the control and visualization layer and is deployed using the official Home Assistant Operating System (HAOS), a Linux-based environment, running as a virtual machine on an ESXi hypervisor.

Security is implemented using elliptic curve cryptography (ECC) certificates, with emphasis on practical deployment on resource-constrained IoT devices and on leveraging hardware features available on the ESP32-S3 platform. Basic experimental measurements are performed to observe connection establishment characteristics and message delivery behavior under load.

Scalability experiments demonstrate that the proposed architecture is capable of handling over 1,000 concurrent device connections under realistic conditions, with the design supporting larger deployments. A qualitative comparison with selected commercial smart home platforms highlights trade-offs between ease of deployment, flexibility, and data sovereignty. The results indicate that Unix-based, open-source solutions represent a viable alternative for smart home sensor control, particularly in scenarios requiring customization and full control over data processing.

Keywords: smart home, IoT, MQTT, mTLS, ECC, ECDSA, X25519, ESPHome, Home Assistant, ESP32-S3, edge computing, fall detection

Abstrakt

Tato bakalářská práce se zabývá využitím serverově orientované architektury a operačních systémů typu Unix pro monitoring a řízení senzorů v prostředí chytré domácnosti. Cílem je ukázat, že open-source řešení s vlastním hostingem (self-hosted) může zajistit bezpečnou, škálovatelnou a transparentní správu distribuovaných senzorových zařízení bez závislosti na proprietárních cloudových službách.

Navržený systém kombinuje fyzické uzly postavené na mikrokontrolérech ESP32 a ESP32-S3—přičemž druhý jmenovaný implementuje algoritmy edge computingu pro detekci pádu v reálném čase—s Python simulátorem senzorů, který umožňuje emulovat velké množství zařízení pro účely ověření škálovatelnosti. Komunikace mezi IoT komponentami je primárně realizována pomocí protokolu MQTT s oboustranným zabezpečením Transport Layer Security (mTLS), doplněná o nativní API integrace v rámci platformy Home Assistant určené pro správu zařízení a automatizaci. Centrálním prvkem messagingu je broker EMQX nasazený na vzdáleném serveru. Home Assistant plní roli řídicí a vizualizační vrstvy a je provozován jako oficiální Home Assistant Operating System (HAOS), založený na Linuxu, ve virtuálním stroji na hypervizoru ESXi.

Bezpečnost je zajištěna certifikáty využívajícími eliptickou kryptografii (ECC) s důrazem na praktickou implementaci v IoT zařízeních s omezenými zdroji a na využití hardwarových vlastností platformy ESP32-S3. Jsou provedeny základní experimenty zaměřené na charakteristiky navazování spojení a chování doručování zpráv při zátěži.

Experimenty škálovatelnosti ukazují, že navržená architektura je za realistických podmínek schopna obsloužit více než 1000 současně připojených zařízení, přičemž návrh podporuje i větší nasazení. Kvalitativní srovnání s vybranými komerčními platformami chytré domácnosti zdůrazňuje kompromisy mezi jednoduchostí nasazení, flexibilitou a suverenitou dat. Výsledky naznačují, že unixová open-source řešení představují životaschopnou alternativu pro řízení senzorů v chytré domácnosti, zejména v případech vyžadujících přizpůsobení a plnou kontrolu nad zpracováním dat.

Klíčová slova: chytrá domácnost, IoT, MQTT, mTLS, ECC, ECDSA, X25519, ESPHome, Home Assistant, ESP32-S3, edge computing, detekce pádu

Contents

List of Acronyms

ACL Access Control List.

ADC Analog-to-Digital Converter.

BBR Bottleneck Bandwidth and Round-trip propagation time.

BLE Bluetooth Low Energy.

CA Certificate Authority.

CNN Convolutional Neural Network.

CRL Certificate Revocation List.

CSR Certificate Signing Request.

DRAM Dynamic Random Access Memory.

ECC Elliptic Curve Cryptography.

ECDH Elliptic Curve Diffie–Hellman.

ECDHE Elliptic Curve Diffie–Hellman Ephemeral.

ECDSA Elliptic Curve Digital Signature Algorithm.

EdDSA Edwards-curve Digital Signature Algorithm.

EMQX Erlang MQTT Broker.

ESP-IDF Espressif IoT Development Framework.

ESP-NN Espressif Neural Network Library.

ESXi Elastic Sky X integrated.

FQ Fair Queueing.

GPIO General Purpose Input/Output.

HA Home Assistant.

HACS Home Assistant Community Store.

HAOS Home Assistant Operating System.

I2C Inter-Integrated Circuit.

IoT Internet of Things.

IRAM Instruction Random Access Memory.

LWT Last Will and Testament.

MCU Microcontroller Unit.

MEMS Micro-Electro-Mechanical Systems.

MQTT Message Queuing Telemetry Transport.

mTLS mutual Transport Layer Security.

NVS Non-Volatile Storage.

OTA Over-The-Air.

PIR Passive Infrared.

PWM Pulse Width Modulation.

QoS Quality of Service.

RSA Rivest–Shamir–Adleman.

RTT Round-Trip Time.

SMV Signal Magnitude Vector.

TFLite TensorFlow Lite.

TLS Transport Layer Security.

vCPU virtual Central Processing Unit.

VPS Virtual Private Server.

ZTNA Zero Trust Network Access.

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

The concept of smart homes has evolved significantly over the past decade, transforming from a futuristic vision into a widely deployed class of residential systems. Smart home systems integrate various sensors, actuators, and control platforms to provide automated environmental monitoring, security surveillance, and energy management capabilities [1].

At the core of modern smart home architectures lie servers and Unix-like operating systems that provide the computational infrastructure for data processing, storage, and decision-making. These systems handle the aggregation of sensor data from potentially hundreds of devices, execute automation rules, and provide user interfaces for monitoring and control.

The proliferation of Internet of Things (IoT) devices has created new challenges in terms of scalability, security, and interoperability [2]. Commercial smart home platforms such as Amazon Alexa, Google Home, and Xiaomi Mi Home offer convenient solutions but often rely on cloud connectivity and raise concerns about data privacy and ecosystem dependency [3], [4].

1.2 Problem Statement

Despite the availability of numerous commercial smart home solutions, several challenges remain inadequately addressed:

- **Data Sovereignty:** Commercial platforms typically transmit sensor data to cloud servers, limiting user control over personal information and raising privacy concerns.
- **Vendor Lock-in:** Proprietary ecosystems restrict device interoperability and create dependency on specific manufacturers, as evidenced by platform discontinuations that render devices unusable [4].
- **Security Vulnerabilities:** Many IoT devices lack adequate security measures, with unencrypted communications and weak authentication mechanisms being common issues [5].
- **Scalability Limitations:** Consumer-grade solutions often struggle to handle large numbers of devices or high-frequency sensor data streams.
- **Latency and Availability:** Reliance on remote cloud processing and wide-area networks can introduce variable end-to-end latency and reduce system availability during internet outages, which motivates local-first and edge-oriented designs [6].
- **Limited Customization:** Commercial ecosystems are typically designed for common household scenarios, making it difficult to implement specialized requirements (e.g., custom sensors, non-standard automation logic, or constrained network environments) without relying on vendor-specific extensions.

This thesis addresses these challenges by designing and implementing an open-source smart home system that prioritizes local data processing, secure communications, and scalable architecture.

1.3 Research Objectives

The primary objectives of this thesis are threefold:

1. **Analytical Objective:** Analyze the role of servers and Unix-like systems in smart home sensor monitoring and control, examining the architectural patterns and communication protocols employed in modern IoT deployments.

2. **Design Objective:** Design and implement a model smart home system using the MQTT protocol and Home Assistant platform, demonstrating integration of both real hardware sensors (ESP32/ESP32-S3) and Python-based simulated devices that appear as Home Assistant entities, with the simulator additionally enabling scalable load generation for benchmarking.

3. **Evaluation Objective:** Compare the implemented solution with commercial alternatives in terms of scalability, performance, privacy, and reliability, supported by representative measurements and qualitative system characterization.

1.4 Thesis Organization

The remainder of this thesis is organized as follows:

Chapter 2 introduces the theoretical background, covering smart home systems, the MQTT protocol, transport layer security, Home Assistant and ESPHome, edge computing, and the role of Unix-like systems in IoT infrastructure.

Chapter 3 presents the system architecture and overall design choices, with emphasis on the communication flow, deployment topology, and security considerations.

Chapter 4 describes the implementation of hardware sensor nodes, the Python-based simulator, Home Assistant integration, and supporting data pipeline and automation components.

Chapter 5 presents empirical evaluation covering performance characteristics, scalability, security considerations, system reliability, and a comparative analysis with commercial smart home platforms.

Chapter 6 summarizes the contributions, outlines limitations, and suggests directions for future work.

1.5 Thesis Contributions Overview

Figure 1.1 illustrates the system architecture developed in this thesis. The implementation builds upon a rich ecosystem of open-source technologies spanning messaging infrastructure (EMQX), home automation (Home Assistant, Node-RED), embedded firmware frameworks (ESPHome, mbedTLS), time-series data management (InfluxDB, Grafana), containerization and deployment (Docker, Nginx), and secure networking (Tailscale). Green-shaded components indicate areas where the author performed design, development, or integration work.

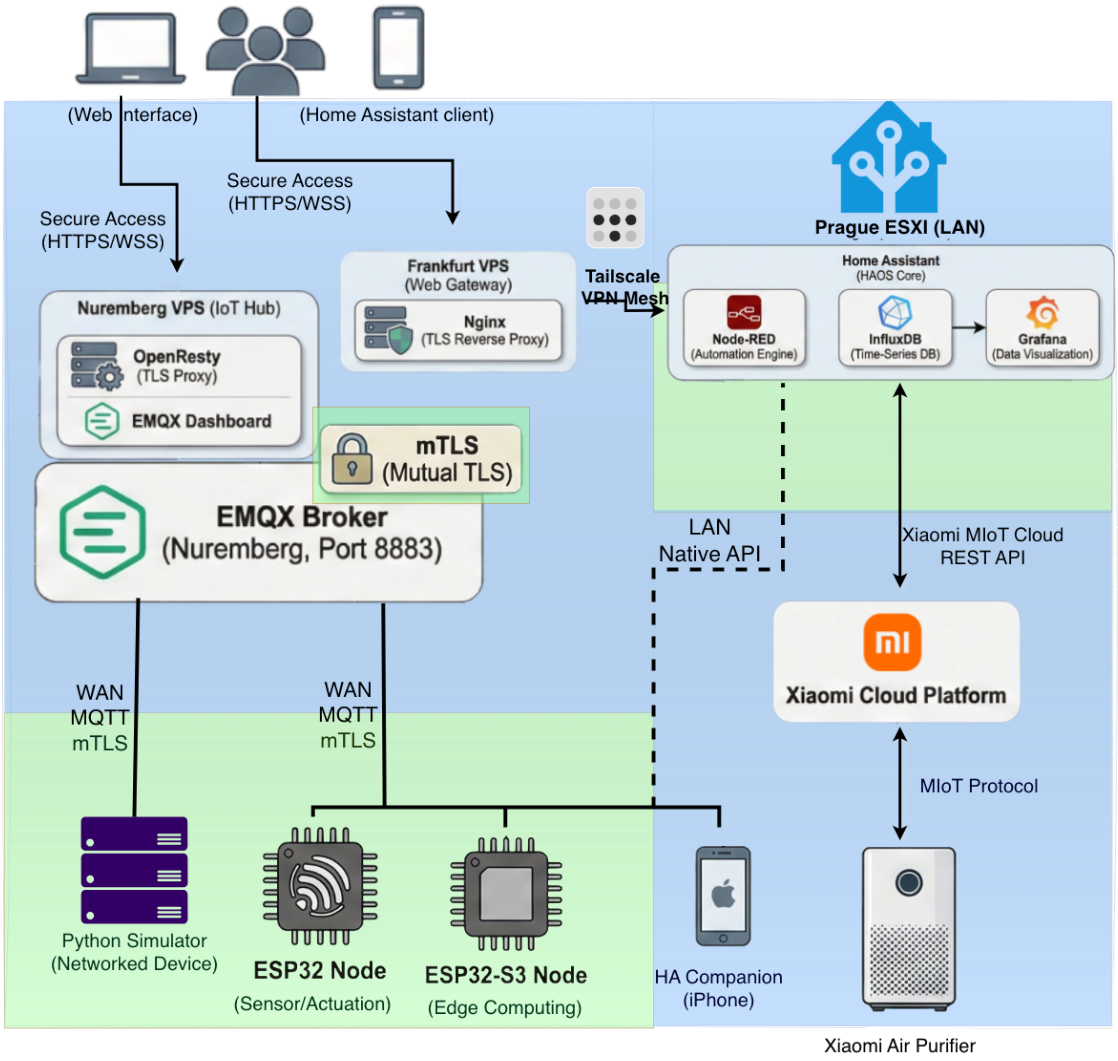


Figure 1.1: System architecture overview. Green: work performed in this thesis; Blue: open-source platforms utilized.

Chapter 2

Theoretical Background

This chapter provides the theoretical foundation for understanding smart home systems, communication protocols, and security mechanisms employed in this thesis.

2.1 Smart Home Systems Overview

A smart home system consists of interconnected devices that monitor and control various aspects of a residential environment. These systems are commonly described using a three-layer IoT reference architecture [2], [7]:

- **Application Layer:** Software platforms that process sensor data, execute automation rules, and provide user interfaces for monitoring and control.
- **Network Layer:** Communication infrastructure that enables data transmission between devices. Common wireless technologies include Wi-Fi (IEEE 802.11), Zigbee (IEEE 802.15.4), and Bluetooth Low Energy (BLE) [1]. Prior work has explored Wi-Fi-based smart home implementations using microcontrollers and mobile applications [8].
- **Perception Layer:** Sensors and actuators that interact with the physical environment, including temperature sensors, motion detectors, smart switches, and lighting controls.

The evolution of smart home technology has been driven by advances in microcontroller capabilities, wireless communication standards, and cloud computing infrastructure [1]. Modern systems increasingly emphasize edge computing, where data processing occurs locally rather than in centralized cloud servers [6].

2.2 MQTT Protocol

Message Queuing Telemetry Transport (MQTT) is a lightweight publish-subscribe messaging protocol designed for constrained devices and low-bandwidth networks [9]. Developed by IBM in 1999, MQTT has become an established standard for IoT communications due to its low protocol overhead and simple implementation [10].

Comparative studies have demonstrated MQTT's advantages over HTTP for IoT applications, including reduced network overhead and lower energy consumption [11].

Unlike traditional request-response protocols, MQTT employs a publish-subscribe model with a central broker:

- **Publishers** send messages to specific topics without knowledge of subscribers.
- **Subscribers** register interest in topics and receive all messages published to those topics.

- **Broker** acts as an intermediary, routing messages from publishers to subscribers based on topic matching.

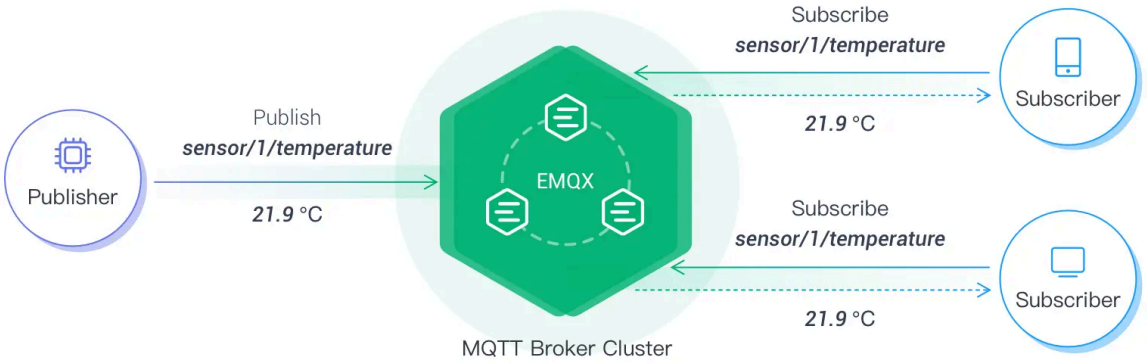


Figure 2.1: MQTT Publish-subscribe Architecture [12]

This decoupled architecture enables scalable many-to-many communication patterns suitable for IoT deployments.

2.2.1 MQTT Protocol Characteristics

MQTT is designed for reliable messaging in constrained and bandwidth-limited IoT environments, emphasizing low protocol overhead and a broker-mediated publish–subscribe model [9], [10]. Key characteristics include:

- **Lightweight control packets:** A compact fixed header and small message framing reduce bandwidth and parsing overhead compared to many request–response protocols.
- **Decoupled communication:** Publishers and subscribers do not need direct knowledge of each other, improving scalability and simplifying multi-producer/multi-consumer deployments.
- **Topic-based addressing and filtering:** Hierarchical topics and wildcard subscriptions enable flexible routing and selective consumption of device telemetry and control streams.
- **Session semantics:** Keep-alive, retained messages, and Last Will and Testament (LWT) support robustness in the presence of intermittent connectivity [9].

MQTT topics use a hierarchical structure with forward slash separators, enabling flexible subscription patterns through single-level (`+`) and multi-level (`#`) wildcards [9]:

Table 2.1: MQTT topic hierarchy and wildcard subscription patterns.

Concept	Example	Notes
State topic	`smarthome/bathroom/temperature/state`	leaf topic
Single-level wildcard	`smarthome/bedroom/+/state`	one level
Multi-level wildcard	`smarthome/#`	any depth

This hierarchical organization facilitates logical grouping of devices and supports pattern-based subscriptions for efficient message filtering.

Table 2.2: Comparison of MQTT Quality of Service (QoS) Levels [9]

QoS Level	Description	Overhead
0 (At most once)	Fire and forget. No guarantee of delivery.	Lowest
1 (At least once)	Guaranteed delivery, duplicates possible. Requires PUBACK.	Medium
2 (Exactly once)	Guaranteed delivery, no duplicates. Four-step handshake.	Highest

2.2.2 Quality of Service Levels

MQTT defines three Quality of Service (QoS) levels to balance reliability against overhead:

QoS 0 provides the lowest overhead but no delivery guarantees. QoS 1 ensures delivery through acknowledgment but may result in duplicates. QoS 2 guarantees exactly-once delivery at the cost of additional message exchanges.

2.3 Transport Layer Security

Transport Layer Security (TLS) provides cryptographic security for communications over computer networks. In IoT contexts, TLS protects against eavesdropping, tampering, and impersonation attacks [5].

Standard TLS authenticates only the server to the client, verifying the server's identity through certificate validation. Mutual TLS (mTLS) extends this by requiring both parties to present certificates:

1. Client initiates connection with ClientHello message.
2. Server responds with certificate and requests client certificate.
3. Client presents its certificate for server verification.
4. Both parties complete key exchange and establish encrypted session.

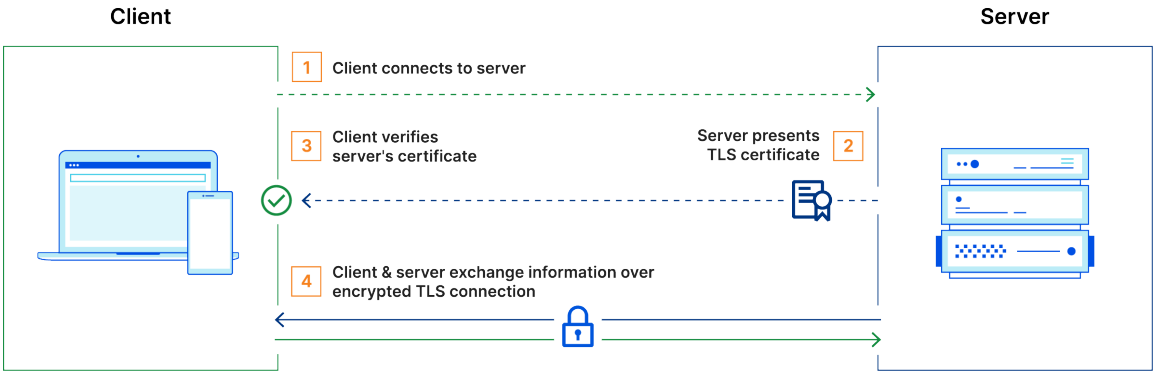


Figure 2.2: Mutual TLS (mTLS) authentication concept and handshake overview [13]

mTLS provides stronger security guarantees for IoT deployments by ensuring that only authorized devices can connect to the message broker.

2.3.1 ECC vs RSA Cryptography

Elliptic Curve Cryptography (ECC) offers equivalent security to RSA with significantly smaller key sizes, making it particularly suitable for resource-constrained IoT devices. According to NIST recommendations, a 128-bit security level corresponds to an RSA key size of 3072 bits, whereas an ECC key requires only 256 bits [14].

Table 2.3: Comparison of ECC and RSA Key Sizes [14]

Security Strength	**ECC Key Size**	**RSA Key Size**
80-bit	160 bits	1024 bits
112-bit	224 bits	2048 bits
128-bit	**256 bits**	3072 bits

The computational efficiency of ECC becomes evident when analyzing performance on specific microcontroller platforms. Benchmarks reported by Oryx Embedded[1] for ESP32 (Xtensa LX6) and ESP32-S3 (Xtensa LX7) at 240 MHz show substantial latency differences across public-key algorithms [15], [16].

Table 2.4: Digital Signature Performance on ESP32 platforms [15], [16]

Algorithm	**ESP32 (LX6)**			**ESP32-S3 (LX7)**		
	Sign	Verify	Total	Sign	Verify	Total
RSA-3072 (HW)	465 ms	314 ms	779 ms	315 ms	57 ms	372 ms
ECDSA P-256 (HW)	62 ms	57 ms	119 ms	67 ms	60 ms	127 ms
Ed25519 (SW)	**29 ms**	**28 ms**	**57 ms**	**26 ms**	**24 ms**	**50 ms**

Table 2.5: Key Exchange Performance on ESP32 Platforms [15], [16]

Algorithm	**ESP32 (LX6)**			**ESP32-S3 (LX7)**		
	Keygen	Shared	Total	Keygen	Shared	Total
ECDH P-256 (HW)	54 ms	54 ms	108 ms	63 ms	62 ms	125 ms
X25519 (SW)	**17 ms**	**16 ms**	**33 ms**	**14 ms**	**15 ms**	**29 ms**

Notably, Ed25519 and X25519 in pure software outperform NIST P-256 with hardware acceleration by 2–4×. This reflects the Curve25519 design goal of efficient arithmetic on general-purpose CPUs [17].

2.3.2 Selection of Cryptographic Algorithms for TLS

Based on the performance analysis above, this thesis initially targeted the **Curve25519** cryptographic suite: **Ed25519** (EdDSA) for digital signatures and **X25519** for Elliptic Curve Diffie–Hellman (ECDH) key exchange. The selection was motivated by three primary factors:

- **Efficiency on embedded CPUs:** The reported benchmarks show that Ed25519 and X25519 complete faster than the NIST P-256 alternatives on both ESP32 and ESP32-S3 (Tables 2.4 and 2.5).

[1] `https://www.oryx-embedded.com/benchmark/espressif/`

- **Implementation hardening:** X25519 is specified with a Montgomery-ladder construction that is commonly implemented in a constant-time style to reduce timing side channels [17], [18].
- **Parameter transparency and deterministic signing:** Curve25519 has been discussed as having more transparent parameter choices, and SafeCurves[2] highlights "rigidity" as a criterion where NIST P-256 is rated negatively due to opaque seeds [19]. For authentication, Ed25519 (EdDSA) uses deterministic nonce generation as standardized in RFC 8032, reducing reliance on runtime randomness during signing [20], [21]. This design addresses a fundamental weakness in ECDSA, where poor random number generation during signing can leak the private key—a critique extensively documented by Bernstein [22].

Library Limitations and Hybrid Approach

During implementation, a practical limitation was encountered: the ESP-IDF framework's underlying TLS library (Mbed TLS) supports X25519 for ECDH key exchange but does not currently support Ed25519 (EdDSA) for X.509 certificate signatures [23]. The ESP-TLS API documentation explicitly lists only `SECP256R1` and `SECP384R1` as supported ECDSA curves for certificate operations. This limitation is documented in the Mbed TLS project issue tracker[3], where Ed25519 certificate support has been requested as a feature enhancement [24]. According to the official Mbed TLS roadmap[4], EdDSA support for certificates is planned for future releases [25].

It is worth noting that alternative TLS libraries exist for embedded systems that provide full Curve25519 support. wolfSSL[5], a lightweight embedded TLS library, offers complete Ed25519 and X25519 support, including Ed25519-based X.509 certificate signatures [26]. The ESP-IDF framework officially supports wolfSSL as an alternative to Mbed TLS through its component system [23]. wolfSSL is designed for resource-constrained environments and supports TLS 1.3 with progressive cipher suites. However, for this thesis, ESPHome was chosen as the firmware framework due to its simplified YAML-based configuration and native Home Assistant integration. ESPHome currently uses Mbed TLS as its TLS backend and does not expose wolfSSL as a configuration option, making the hybrid approach (P-256 certificates with X25519 key exchange) the practical choice within this framework. For deployments requiring pure Curve25519-based mTLS, switching to native ESP-IDF with wolfSSL would be necessary.

Hybrid TLS Configuration. To address this limitation while maximizing performance, this thesis adopts a **hybrid cryptographic approach** that combines the strengths of both algorithm families:

- **Certificate signatures:** ECDSA with NIST P-256 (`prime256v1`), which is fully supported by ESP-TLS for X.509 certificate parsing and verification.
- **Key exchange:** X25519, automatically negotiated during TLS 1.3 handshake when the `supported_groups` extension includes `x25519` on both endpoints [27].

This separation is possible because TLS 1.3 decouples the authentication mechanism (certificate signatures) from the key exchange mechanism (ephemeral Diffie–Hellman). As specified in RFC 8446, the signature algorithm used in certificates is independent of the key exchange group negotiated in the handshake [27].

[2] `https://safecurves.cr.yp.to/`
[3] `https://github.com/Mbed-TLS/mbedtls/issues/2452`
[4] `https://mbed-tls.readthedocs.io/en/latest/roadmap/`
[5] `https://www.wolfssl.com/`

Performance Analysis of the Hybrid Approach. Table 2.6 presents the estimated cryptographic latency for a complete mTLS handshake under different configurations. The hybrid approach (P-256 signatures + X25519 key exchange) achieves a favorable balance: it avoids the slow RSA operations while leveraging X25519's efficiency for the latency-critical key exchange phase.

Table 2.6: Estimated TLS Handshake Cryptographic Latency on ESP32-S3 [16]

Configuration	Signature (Verify)	Key Exchange	Total
RSA-3072 + ECDH P-256	57 ms	125 ms	182 ms
ECDSA P-256 + ECDH P-256	60 ms	125 ms	185 ms
ECDSA P-256 + X25519 (Hybrid)	60 ms	**29 ms**	**89 ms**
Ed25519 + X25519 (Ideal)*	24 ms	29 ms	53 ms

*Theoretical; Ed25519 certificate signatures not supported by ESP-TLS.

The hybrid configuration reduces total cryptographic latency by approximately 52% compared to the pure P-256 configuration (89 ms vs 185 ms), primarily due to X25519's 4× faster key exchange. Certificate verification occurs only once per connection establishment, whereas the session key derived from ECDH protects all subsequent application data. For battery-powered IoT devices that employ deep-sleep modes and wake periodically to transmit sensor data, each wake cycle requires re-establishing the TLS connection [28]. Therefore, optimizing key exchange latency directly reduces the energy consumed during these frequent reconnection events.

2.4 Home Assistant and ESPHome

Home Assistant[6] is an open-source home automation platform designed to run on local hardware [29]. The platform is distributed as Docker containers built on Alpine Linux base images, providing a lightweight footprint suitable for embedded systems and single-board computers [30]. Key features include:

- **Integration Support:** Extensive integration ecosystem covering a wide range of commercial and DIY devices and services.
- **Local Control:** Local-first operation with optional cloud connectivity, enabling continued functionality even when external services are unavailable.
- **Automation Engine:** Flexible automation via YAML configuration and a visual editor, supporting complex rule-based behaviors.
- **Dashboard System:** Customizable dashboards for real-time monitoring, control, and status visualization across devices.

In addition, Home Assistant supports MQTT Discovery, which allows devices to be automatically registered by publishing configuration messages to predefined discovery topics.

For ESP32-based devices, firmware development and integration with Home Assistant are facilitated by the ESPHome framework.

ESPHome[7] is a firmware framework that generates custom firmware from YAML configuration files [31]. It supports a range of microcontroller platforms, including ESP8266, ESP32 variants, RP2040, and select LibreTiny-based chipsets. Its declarative configuration style is conceptually similar to earlier Wi-Fi-based appliance control systems that emphasized simplified device programming [32]. Key capabilities include:

[6] `https://www.home-assistant.io/`

[7] `https://esphome.io/`

- **Sensor Support:** Native drivers for hundreds of sensor types.
- **Native API:** Low-latency direct communication with Home Assistant.
- **MQTT Support:** Full MQTT client with TLS/mTLS capability.
- **OTA Updates:** Over-the-air firmware updates via WiFi.
- **Local Automation:** On-device automation without external dependencies.

The declarative YAML configuration approach reduces development complexity while maintaining flexibility through lambda expressions for custom logic.

Beyond built-in integrations, Home Assistant supports community-developed components distributed via the Home Assistant Community Store (HACS)[8]. This mechanism enables integration of devices from vendors who do not provide official Home Assistant support, including official manufacturer-provided integrations such as the Xiaomi Home Integration[9]. Such extensibility allows users to aggregate heterogeneous device ecosystems under a single local-first platform, contrasting with commercial solutions that typically rely on vendor-specific cloud infrastructure.

2.5 Edge Computing and Unix-like Systems

Edge computing moves parts of data processing closer to the data source, reducing latency and upstream bandwidth requirements and enabling continued operation when cloud connectivity is degraded [6].

Table 2.7: Comparison between Cloud Computing and Edge Computing in IoT Contexts [6]

Feature	Cloud Computing	Edge Computing
Latency	High (WAN-dependent)	Low (near real-time)
Bandwidth	High usage (raw data)	Low usage (processed data)
Privacy	Data leaves premises	Data stays local
Processing Power	Elastic (scalable)	Constrained (MCU/CPU)
Dependency	Internet connection	Local power

In smart home deployments, a central architectural choice is whether integration logic and data processing are executed in a vendor-operated cloud or on user-controlled infrastructure.

This thesis adopts a *server-based* design and implements the main software components—MQTT broker, Home Assistant, and the sensor simulator—on Unix-like systems within a controlled local environment. The motivation is not peak performance alone, but operational robustness and measurability, which are central for benchmarking and reproducible evaluation. Additionally, the open-source tooling and active community support lower the barrier for individual deployment and learning:

- **Operational maturity:** The process model, service supervision, and standard administration tooling support repeatable deployments and recovery workflows [33].
- **Security ecosystem:** Linux platforms provide widely deployed TLS stacks and cryptographic libraries, simplifying secure broker deployments and certificate management [34].
- **Observability for evaluation:** System-level tooling enables measurement of CPU scheduling, memory pressure, filesystem behavior, and network throughput, which is required to relate application metrics to underlying bottlenecks [35].

[8] `https://hacs.xyz/`

[9] `https://github.com/XiaoMi/ha_xiaomi_home`

In the experimental setup, the MQTT broker runs on a Debian-based server, and Home Assistant is deployed via the Home Assistant Operating System (HAOS), a Linux-based operating system built using Buildroot and optimized to host Home Assistant [36]. The sensor simulator is implemented in Python, enabling cross-platform execution, controllable entity emulation, and reproducible benchmarking across diverse host environments.

Chapter 3

System Architecture

This chapter presents the architecture of the implemented smart home system. The design emphasizes local-first control, secure cross-network communication, and integration of heterogeneous device types—including custom ESP32-based sensors, a Python-based simulator for virtual entity emulation also can use for load generation for benchmarking, and commercial IoT devices.

3.1 Architecture Overview

The system follows a distributed architecture with components spanning local and cloud-hosted infrastructure, connected through secure communication channels. Figure 3.1 illustrates the high-level topology.

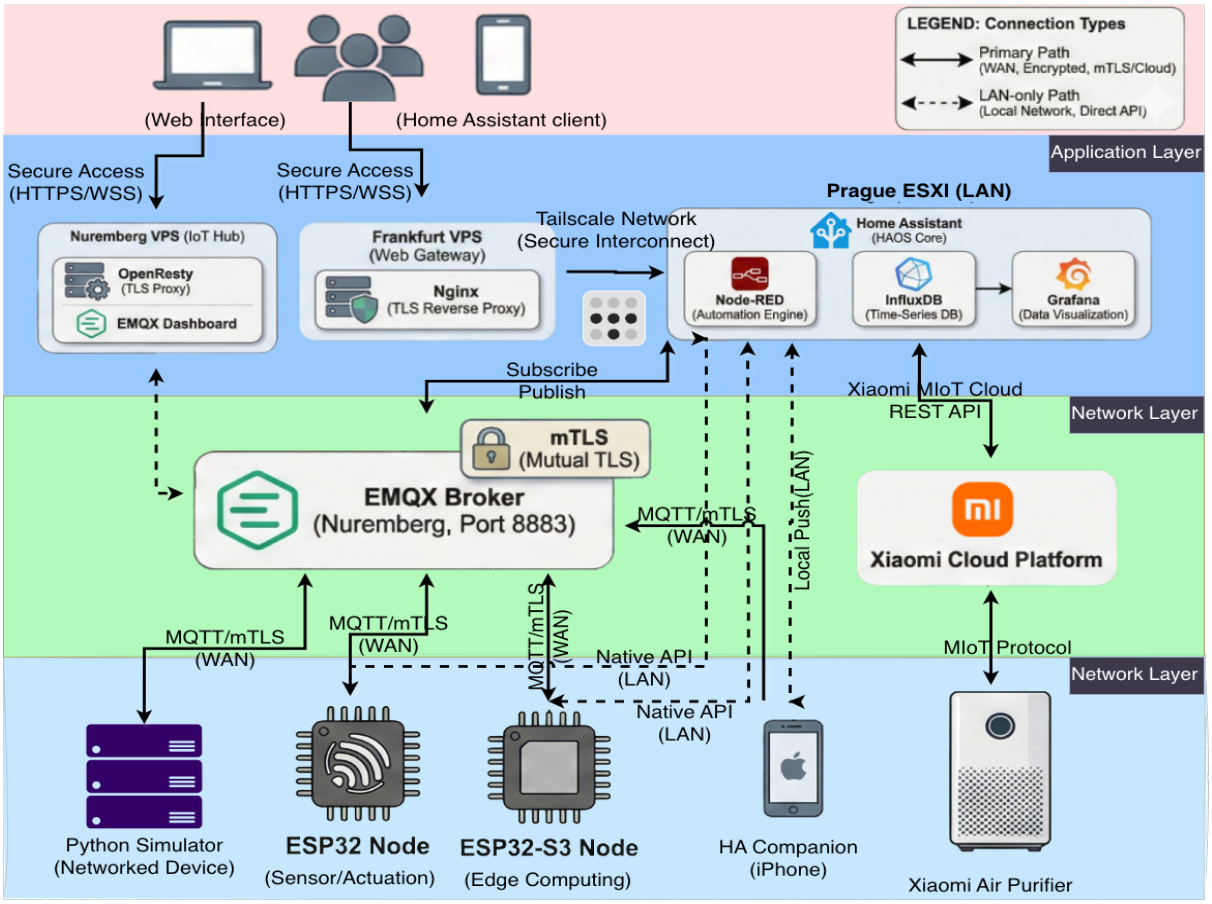


Figure 3.1: Overall system architecture and communication paths.

The architecture aligns with the three-layer IoT reference model introduced in Section 2.1:

- **Application Layer:** Home Assistant platform providing automation engine, device management, user dashboards, and optional data persistence.
- **Network Layer:** MQTT messaging over mTLS for wide-area communication; ESPHome Native API for low-latency local control [31]; cloud APIs for commercial device integration.
- **Perception Layer:** Physical sensor nodes (ESP32, ESP32-S3), commercial IoT devices (Xiaomi air purifier), and the Python simulator generating virtual device telemetry.

3.2 System Components

Table 3.1 summarizes the principal components of the implemented system, their deployment locations, and roles.

Table 3.1: System components and deployment topology.

Component	Location	Platform	Role
Web Gateway	VPS (Frankfurt)	Debian / Nginx	Bastion; HA TLS reverse proxy
EMQX Broker	VPS (Nuremberg)	Debian / OpenResty	MQTT routing
Home Assistant	Local (Prague)	HAOS on ESXi	Automation hub
ESP32 / ESP32-S3	Local (Prague)	ESPHome	Sensor & actuator
Xiaomi Air Purifier	Local (Prague)	Commercial	Air quality
Python Simulator	Network-connected host	Cross-platform	Virtual entities; load testing
HA Companion App[1]	Mobile	iOS / Android	Mobile client / Sensor

3.2.1 Custom Sensor Nodes

Two ESP32-based sensor nodes are deployed using ESPHome firmware. These nodes collect environmental data (temperature, humidity, ambient light) and expose actuators (LED indicators, relays). Communication with Home Assistant occurs via two paths:

- **ESPHome Native API [31]:** Used for local-network communication, providing low-latency bidirectional control without broker dependency.
- **MQTT over mTLS [9]:** Used for wide-area messaging when the controller or devices span different networks, ensuring encrypted and authenticated communication.

The dual-path design balances responsiveness for local automation with security for Internet-exposed scenarios.

3.2.2 Commercial Device Integration

Integrations connecting Home Assistant to external systems can be categorized by data flow direction: *inbound integrations* import devices into Home Assistant, while *outbound integrations* expose Home Assistant entities to external platforms.

Inbound integration (devices to Home Assistant). A Xiaomi smart air purifier is integrated via the Xiaomi Home Integration[2], installed through HACS[3]. According to the integration documentation, device state and events are delivered to Home Assistant via subscriptions to

[2] `https://github.com/XiaoMi/ha_xiaomi_home`
[3] `https://hacs.xyz/`

Xiaomi's MQTT-based cloud message bus, while control commands are issued via Xiaomi cloud HTTP interfaces. The integration also supports local-control modes under specific conditions (e.g., Xiaomi hub gateway or LAN control), otherwise operation is cloud-mediated. This contrasts with the fully local communication available for ESPHome-based nodes.

Outbound integration (Home Assistant to voice platforms). Home Assistant can expose entities to voice assistant ecosystems. The HomeKit Bridge[4] publishes entities to Apple Home for Siri control. The Google Assistant integration[5] exposes entities to Google Home. Home Assistant Cloud (Nabu Casa)[6] simplifies connectivity for Google Assistant and Amazon Alexa without manual cloud configuration.

Including these integrations serves two purposes:

- It validates Home Assistant's role as a unifying platform capable of aggregating heterogeneous device ecosystems under a single interface.
- It provides a concrete example of vendor cloud dependency, illustrating the privacy and availability trade-offs discussed in Section 1.2.

3.2.3 Python Simulator

The Python-based sensor simulator emulates large numbers of virtual devices as Home Assistant entities by publishing telemetry to the MQTT broker, subscribing to command topics, and supporting Home Assistant MQTT Discovery. This provides a controllable and repeatable way to validate message flows, automations, and dashboard behavior without requiring additional physical hardware.

Building on this entity-emulation capability, the simulator also serves as a scalable load generator for stress testing, enabling experiments with thousands of virtual entities under reproducible configurations.

The simulator supports:

- Configurable device count and entity types (sensors, binary sensors, switches, lights).
- Multiple data models (drift, uniform, sine wave, motion events).
- Home Assistant MQTT Discovery for automatic entity registration.
- mTLS client authentication matching the security requirements of the broker.

3.3 Communication Architecture

The communication design addresses three goals: (i) low latency for interactive local control, (ii) security for Internet-exposed messaging, and (iii) scalability for benchmarking with large device populations.

3.3.1 Topic Namespace Design

Building on the MQTT topic hierarchy introduced in Section 2.2.1, the system implements a consistent naming convention. Device telemetry and commands follow the pattern:

```
{base_topic}/device_{id}/{entity}/state
{base_topic}/device_{id}/{entity}/set
```

[4] https://www.home-assistant.io/integrations/homekit/
[5] https://www.home-assistant.io/integrations/google_assistant/
[6] https://www.nabucasa.com/

Home Assistant MQTT Discovery publishes configuration payloads under the `homeassistant/` prefix [29], enabling automatic device registration.

Table 3.2 shows representative MQTT subscriptions captured from the EMQX broker during system operation, illustrating the topic namespace in practice.

Table 3.2: MQTT subscriptions observed during system operation (EMQX dashboard).

Client	Subscription Topic	QoS	Purpose
ESP32	`smarthome-esp32/switch/led/command`	0	Actuator command
ESP32-S3	`smarthome-esp32s3/sensor/temperature/state`	0	Sensor telemetry
Simulator	`smarthome/sim/device_1/heater/set`	0	Virtual actuator
Home Assistant	`homeassistant/sensor/+/config`	0	MQTT Discovery

3.3.2 Communication Paths

Three distinct communication paths serve different deployment scenarios:

- **ESPHome Native API [31]:** ESP32 nodes on the same LAN as Home Assistant use the native API protocol, providing sub-second response times without Internet dependency.
- **MQTT over mTLS [9]:** The simulator and devices outside the local network connect to the EMQX broker on port 8883 with mutual TLS authentication. This path also serves as a fallback for ESPHome devices in distributed deployments.
- **Cloud APIs:** Commercial devices (e.g., Xiaomi air purifier) integrate via vendor infrastructure. In the Xiaomi Home Integration, state is delivered via Xiaomi's MQTT-based message bus and commands are sent through vendor HTTP interfaces; local-control modes are optional and deployment-dependent.

Table 3.3 summarizes the characteristics of each path.

Table 3.3: Communication path characteristics.

Path	Latency	Security	Dependency
ESPHome Native API [31]	Low (LAN)	API encryption	Local network
MQTT over mTLS [9]	Medium (WAN)	mTLS mutual auth	Self-hosted broker
Cloud APIs	Variable (WAN)	Vendor TLS	Vendor cloud

3.4 Security Architecture

The security architecture implements defense-in-depth across transport, network, and application layers.

3.4.1 Transport Layer Security

All MQTT communications between the simulator, ESPHome devices (when using MQTT), and the EMQX broker employ mutual TLS (mTLS) on port 8883. The certificate hierarchy consists of:

- A self-signed Certificate Authority (CA) issuing all certificates.
- Server certificate for the EMQX broker, verified by clients against the CA.

- Client certificates for each device and simulator instance, verified by the broker.

Certificates use ECDSA with NIST P-256 for signatures and X25519 for ephemeral key exchange, based on the hybrid TLS approach analyzed in Section 2.3.2. The complete certificate generation procedure is documented in Appendix A. This choice provides strong security with minimal computational overhead on resource-constrained ESP32 devices.

3.4.2 Network Layer Security

To avoid direct exposure of administrative interfaces to the public Internet, the system employs a layered access architecture:

- **MQTT (port 8883):** Clients connect directly to the EMQX broker in Nuremberg using mTLS; no reverse proxy is involved.
- **EMQX Dashboard:** A co-located OpenResty instance on the same host handles TLS termination (Let's Encrypt) and reverse proxying for the broker's web interface.
- **Home Assistant Dashboard:** A Nginx gateway in Frankfurt terminates TLS (Let's Encrypt) and proxies requests to the internal Home Assistant instance in Prague via Tailscale.

Tailscale[7], a mesh VPN built on WireGuard [37], interconnects external VPS nodes and the internal network. Access Control Lists (ACLs) enforce least-privilege principles, as shown in Table 3.4.

Table 3.4: Tailscale ACL rules for network segmentation.

Source Tag	Destination	Purpose
`tag:user`	`*:*`	Full access (owner devices)
`tag:bastion`	`tag:ha-gateway:8123`	Web gateway to HA dashboard only
`tag:node`	`tag:node:*`	Inter-node communication

This configuration ensures the web gateway can only reach Home Assistant on port 8123, while owner-controlled nodes (`tag:user`) retain full access for administration.

3.4.3 Application Layer Security

Home Assistant uses token-based authentication with configurable session timeouts [29]. For third-party integrations such as Xiaomi Home, the integration uses OAuth 2.0 login and stores user/device metadata (including tokens and certificates) in Home Assistant configuration files.

ESP32 nodes store credentials in encrypted Non-Volatile Storage (NVS) partitions provided by the ESP-IDF framework [38], protecting against casual firmware extraction.

3.5 Design Rationale

This section motivates the principal architectural decisions and relates each choice to the system goals outlined in this chapter (local-first operation, secure cross-network communication, and heterogeneous device integration).

[7] `https://tailscale.com/blog/how-tailscale-works/`

Hybrid local–cloud topology. Hosting the MQTT broker on a public VPS rather than locally reflects the experimental requirement for cross-network benchmarking. This enables the simulator to connect from arbitrary locations while maintaining mTLS-authenticated connectivity [9]. For production deployments, a local broker would reduce latency and eliminate external dependencies.

Dual communication paths for ESP32 nodes. Supporting both ESPHome Native API [31] and MQTT allows the system to optimize for latency locally while retaining flexibility for distributed deployments. The native API provides lowest latency on shared LANs; MQTT extends connectivity across network boundaries.

Inclusion of commercial devices. Integrating the Xiaomi air purifier demonstrates that an open-source platform can coexist with vendor ecosystems, aggregating heterogeneous sources into a unified interface. This also illustrates cloud dependency trade-offs analyzed in Chapter 5.

Simulator for scalability testing. Physical deployment of thousands of sensors is impractical for a thesis project. The Python simulator provides a controlled, repeatable method to stress-test the broker and controller, producing quantitative data presented in Chapter 5.

Chapter 4

Implementation

This chapter details the implementation of all system components, including hardware sensor nodes, software simulator, MQTT broker configuration, and integration with Home Assistant.

4.1 Hardware Implementation

Two ESP32-based sensor nodes were implemented to demonstrate different IoT paradigms: edge intelligence and traditional environment sensing. The ESP32 platform was selected for its combination of dual-core processing, WiFi connectivity, and extensive peripheral support [39].

4.1.1 ESP32-S3 Edge Intelligence Node

The ESP32-S3 node serves as an edge computing platform, performing local signal processing to reduce cloud dependency and enable real-time responses [40].

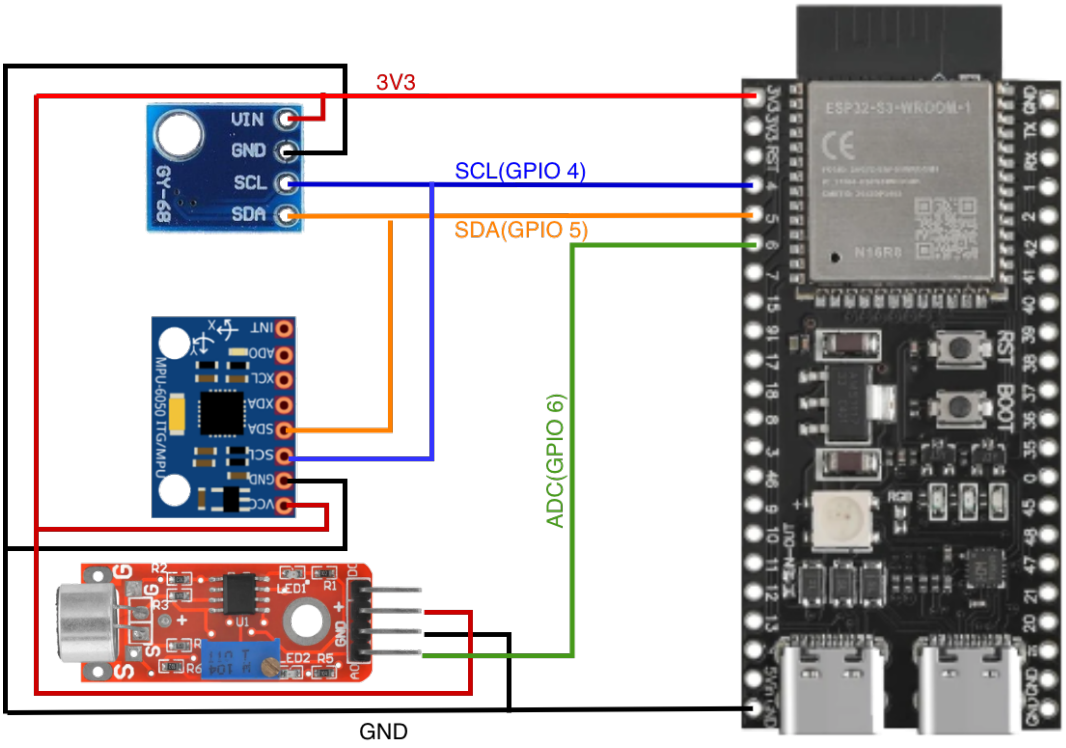


Figure 4.1: ESP32-S3 edge intelligence node with labeled sensor components.

Table 4.1: ESP32-S3 Node Sensor Configuration

Sensor	Model	Measurement	Interface
Barometric	BMP180	Temperature, Pressure	I2C
Motion	MPU6050 (GY-521)	6-axis Accelerometer/Gyro	I2C
Acoustic	KY-037	Sound Level	ADC

The BMP180 barometric sensor and MPU6050 inertial measurement unit share the same I2C bus (GPIO4 for SCL, GPIO5 for SDA). This is possible because the I2C protocol supports multiple devices on a single two-wire bus through 7-bit addressing—each device responds only to its unique address (BMP180: `0x77`, MPU6050: `0x68`) [41]. The shared-bus architecture reduces GPIO consumption from four pins to two while enabling bidirectional communication with both sensors.

Table 4.2 details the complete GPIO assignments and sensor module connections for the ESP32-S3 node. Each sensor module exposes multiple pins, but only a subset is required for basic operation; unused pins either provide redundant functionality or optional features not utilized in this implementation.

Table 4.2: ESP32-S3 Node Pin Connections and Sensor Module Details

Module	Pin	ESP32-S3 GPIO	Function	Notes
MPU6050[1]	VCC	3V3	Power (2.375–3.46 V)	—
	GND	GND	Ground	—
	SCL	GPIO4	I2C Clock	Shared with BMP180
	SDA	GPIO5	I2C Data	Shared with BMP180
	XDA/XCL	—	Aux I2C	—
	AD0	—	Address select	Unconnected = 0x68
	INT	—	Interrupt	—
BMP180[2]	VIN	3V3	Power (1.8–3.6 V)	—
	GND	GND	Ground	—
	SCL	GPIO4	I2C Clock	Shared with MPU6050
	SDA	GPIO5	I2C Data	Shared with MPU6050
KY-037[3]	VCC	3V3	Power (3.3–5 V)	—
	GND	GND	Ground	—
	AO	GPIO6	Analog output	Continuous sound level
	DO	—	Digital output	Threshold via potentiometer

The firmware execution flow is depicted in Figure 4.2. Upon boot, the device loads TLS configuration from `sdkconfig` and connects to WiFi. The subsequent MQTT connection triggers an mTLS handshake using ECDSA P-256 for certificate verification and X25519 for ephemeral key exchange (TLS 1.3). Sensor acquisition operates at differentiated rates: MPU6050 at 10 Hz for fall detection, BMP180 at 0.2 Hz for environmental monitoring, and KY-037 at 5 Hz for acoustic sampling. Calibrated inertial data feeds the edge computing pipeline, which computes SMV and angular velocity for threshold-based fall detection.

[1] `https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf`
[2] `https://cdn-shop.adafruit.com/datasheets/BST-BMP180-DS000-09.pdf`
[3] `https://www.espboards.dev/sensors/ky-037/`

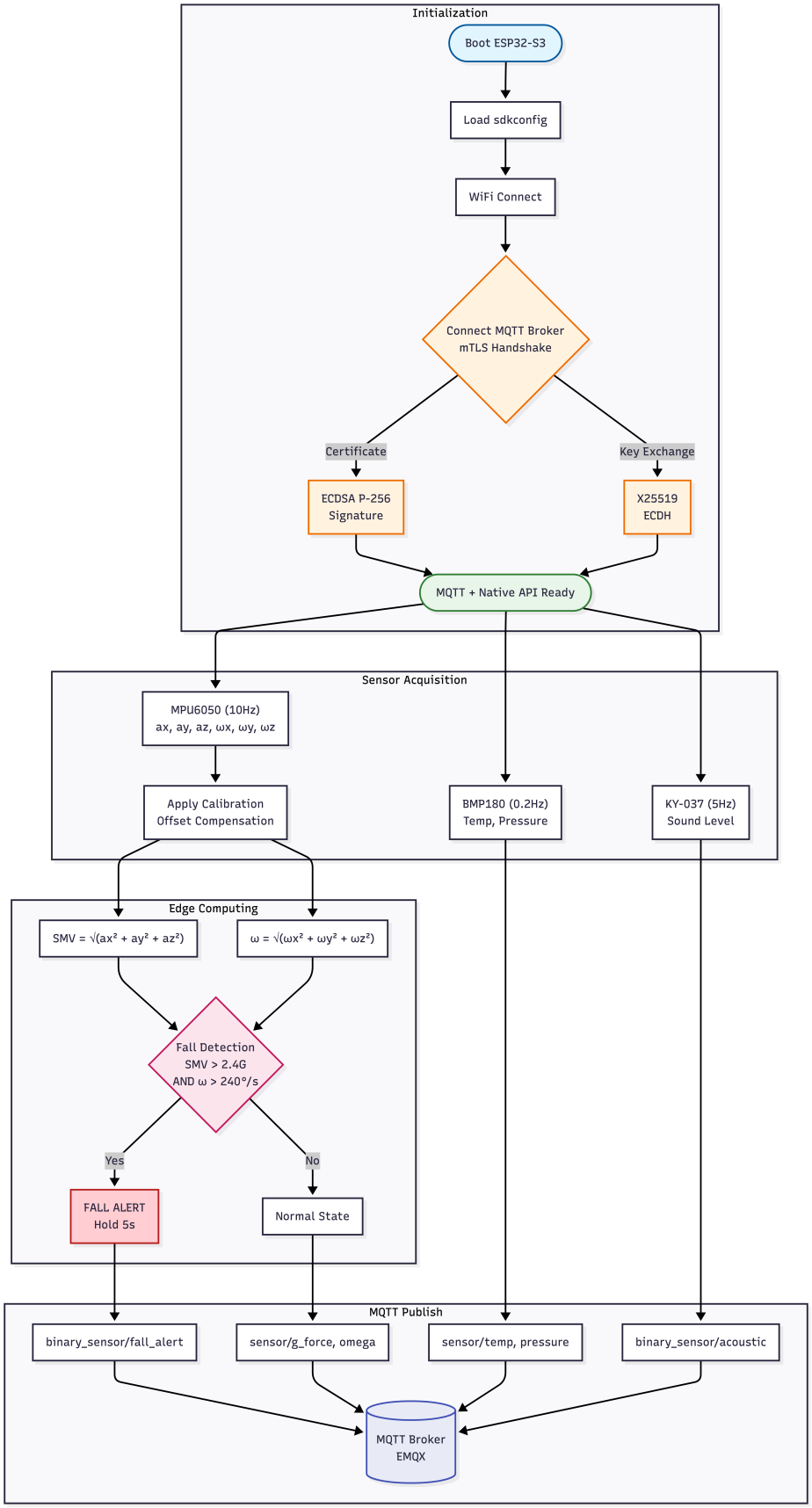


Figure 4.2: ESP32-S3 firmware execution flowchart: initialization with mTLS handshake (ECDSA P-256 + X25519), multi-rate sensor acquisition, edge computing for fall detection, and MQTT state publishing. The complete firmware listing is provided in Appendix C.2.

The ESP32-S3's vector instruction set enables efficient implementation of edge computing algorithms. Table 4.3 summarizes the implemented edge algorithms, including a threshold-based fall detection algorithm adapted from Huynh et al. [42].

Table 4.3: ESP32-S3 Edge Computing Algorithms

Algorithm	**Formula**	**Purpose**
Resultant G-Force	$\text{SMV} = \sqrt{a_x^2 + a_y^2 + a_z^2}$	Impact detection
Dynamic Vibration	$\lvert G - 1.0 \rvert$	Motion isolation
Angular Velocity Mag.	$\omega = \sqrt{\omega_x^2 + \omega_y^2 + \omega_z^2}$	Rotation detection
Acoustic Peak	ADC threshold	Sound event

The fall detection algorithm implements a dual-threshold approach adapted from Huynh et al. [42], which combines accelerometer and gyroscope data to distinguish falls from normal activities. A fall event is detected only when both the Signal Magnitude Vector (SMV) exceeds 2.4 G and the angular velocity magnitude exceeds 240 °/s. In the original study (36 subjects, 702 movements), this dual-sensor approach achieved 96.3% sensitivity and 96.2% specificity, compared to 82.72% specificity with accelerometer-only detection—the gyroscope data significantly reduces false positives from high-acceleration activities such as running.

It should be noted that gyroscope utility depends on the detection methodology. Casilari et al. [43] evaluated CNN-based classification on the SisFall dataset and found that accelerometer-only input outperformed combined accelerometer-gyroscope input. This suggests that threshold-based algorithms benefit from explicit rotational velocity checks, whereas deep learning architectures can implicitly extract discriminative features from acceleration signals alone.

For this implementation, the threshold-based approach was selected for several practical reasons. First, it provides computational efficiency suitable for the ESPHome framework, which does not natively support neural network inference. Second, the explicit dual-threshold logic offers interpretable detection decisions, facilitating debugging and parameter tuning. Third, the algorithm requires minimal memory footprint compared to CNN models.

Nevertheless, the ESP32-S3 platform offers significant potential for future enhancement through its vector instruction extensions and the ESP-NN optimized neural network kernels[1] [44]. According to Espressif's benchmarks[2], TensorFlow Lite Micro with ESP-NN achieves a 42× speedup on ESP32-S3 compared to unoptimized execution, reducing the inference time for a person detection model from 2300 ms to just 54 ms [45]. This performance demonstrates that the CNN-based fall detection approach recommended by Casilari et al. is feasible on ESP32-S3 hardware. Such an upgrade would require migrating from ESPHome to the native ESP-IDF framework with TFLite Micro integration, potentially achieving higher detection accuracy with accelerometer-only input while eliminating the gyroscope sensor cost. This represents a promising direction for future development.

Figure 4.3 illustrates the real-time fall detection algorithm executed on the ESP32-S3. Each sensor cycle computes SMV from tri-axial accelerometer readings and angular velocity magnitude from gyroscope data; both values must exceed their thresholds simultaneously to trigger a fall event, minimizing false positives from high-impact activities.

MPU6050 Sensor Calibration

MEMS inertial sensors such as the MPU6050 exhibit systematic bias errors—defined as the average output when no input is applied [46]—that must be compensated through calibration.

[1] `https://github.com/espressif/esp-nn`

[2] `https://github.com/espressif/esp-tflite-micro#performance-comparison`

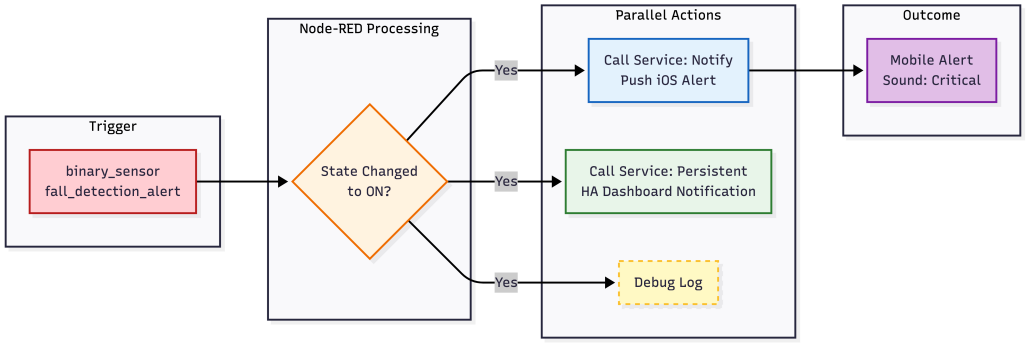


Figure 4.3: Fall detection algorithm flowchart: dual-threshold decision logic using SMV ($> 2.4\,\mathrm{G}$) and angular velocity ($> 240\,^\circ/\mathrm{s}$) for robust fall event classification.

The calibration procedure follows standard zero-offset compensation: the sensor is placed on a level surface in a stationary state, and the deviation from expected values is measured over multiple samples [47].

Table 4.4 presents the calibration parameters derived from stationary measurements. For the accelerometer, the X and Y axes should read zero when level, while the Z-axis should read $-g \approx -9.807\,\mathrm{m/s^2}$ (with the sensor chip facing upward). For the gyroscope, all three axes should read zero when stationary.

Table 4.4: MPU6050 Calibration Parameters from Stationary Measurements

Axis	**Type**	**Expected**	**Raw**	**Offset**	**Calibrated**
a_x	Accel.	$0\,\mathrm{m/s^2}$	$+0.853$	-0.877	-0.024
a_y	Accel.	$0\,\mathrm{m/s^2}$	-0.173	-0.145	-0.318
a_z	Accel.	$-9.807\,\mathrm{m/s^2}$	-9.243	-0.564	-9.807
ω_x	Gyro.	$0\,^\circ/\mathrm{s}$	-2.55	$+2.53$	-0.02
ω_y	Gyro.	$0\,^\circ/\mathrm{s}$	$+2.45$	-2.68	-0.23
ω_z	Gyro.	$0\,^\circ/\mathrm{s}$	-2.91	$+2.44$	-0.47

The gyroscope bias is particularly critical for the fall detection algorithm, as uncalibrated values would introduce a constant offset of approximately $4.6\,^\circ/\mathrm{s}$ to the angular velocity magnitude calculation ($\omega = \sqrt{\omega_x^2 + \omega_y^2 + \omega_z^2}$). After calibration, the residual error is reduced to approximately $0.5\,^\circ/\mathrm{s}$—negligible compared to the $240\,^\circ/\mathrm{s}$ detection threshold. The calibration offsets are applied as ESPHome filter configurations, adding the offset value to each raw reading before further processing.

4.1.2 ESP32 Environment Sensing Node

The standard ESP32 node focuses on environmental monitoring and safety detection with local automation capabilities.

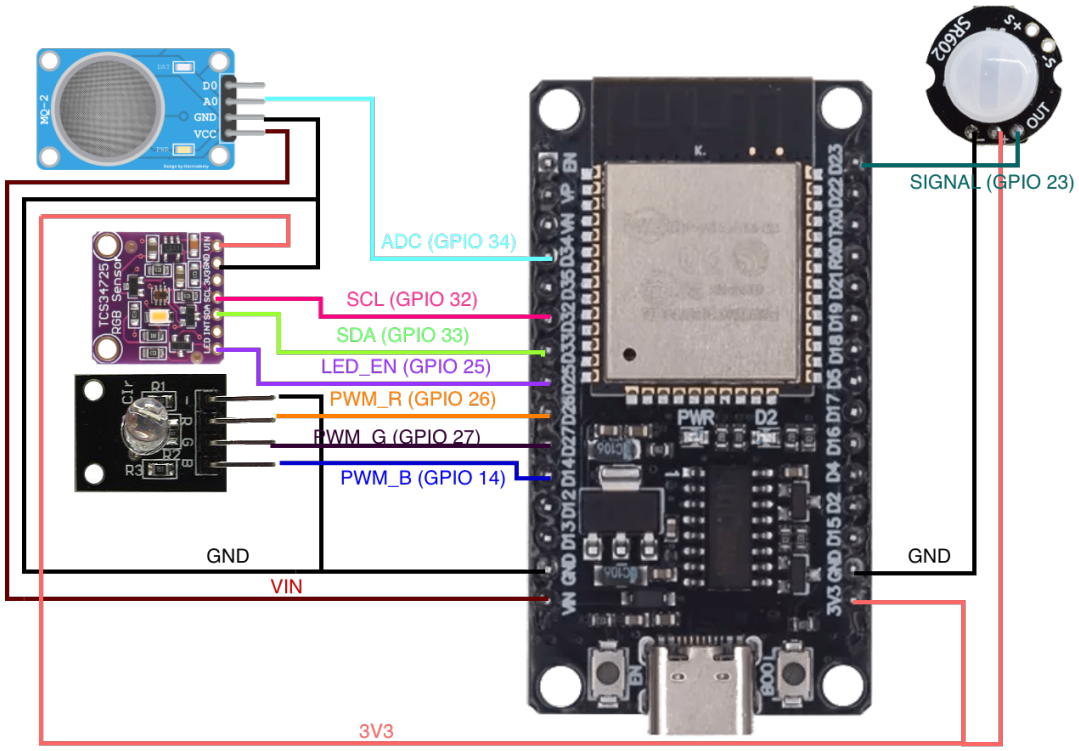


Figure 4.4: ESP32 environment sensing node with labeled sensor components.

Table 4.5: ESP32 Node Sensor Configuration

Sensor	Model	Measurement	Interface
Color/Light	TCS34725	RGB, Lux	I2C
Gas	MQ-2	Smoke/Combustible gas	ADC
Motion	SR602	PIR presence	GPIO
Indicator	RGB LED	Status feedback	PWM

Power Supply Considerations. Unlike other sensors that operate on 3.3 V logic levels, the MQ-2 gas sensor requires a dedicated 5 V supply connected to the VIN pin. This requirement stems from the sensor's internal tin dioxide (SnO_2) heating element, which must maintain a specific operating temperature for accurate gas detection. According to the manufacturer datasheet, the heater voltage (V_H) must be $5.0\,\text{V} \pm 0.1\,\text{V}$, with a heater resistance of $29\,\Omega \pm 3\,\Omega$ and power consumption up to 950 mW [48]. Supplying only 3.3 V would result in insufficient heater temperature, causing unreliable gas concentration readings. The analog output (AO) pin, however, remains 3.3 V tolerant and connects directly to the ESP32's ADC input (GPIO34).

Table 4.6 details the complete GPIO assignments for the ESP32 environment sensing node.

Table 4.6: ESP32 Node Pin Connections and Sensor Module Details

Module	**Pin**	**ESP32 GPIO**	**Function**	**Notes**
TCS34725[4]	VIN	3V3	Power (2.7–3.6 V)	—
	GND	GND	Ground	—
	SCL	GPIO32	I2C Clock	Address 0x29 (fixed)
	SDA	GPIO33	I2C Data	—
	LED	GPIO25	White LED control	Illumination for color sensing
	INT	—	Interrupt	—
MQ-2[5]	VCC	VIN (5V)	Heater power	$V_H = 5.0\,\text{V} \pm 0.1\,\text{V}$
	GND	GND	Ground	—
	AO	GPIO34	Analog output	0–3.3 V proportional to gas
	DO	—	Digital output	Threshold via comparator
SR602 PIR	VCC	3V3	Power (3.3–5 V)	—
	GND	GND	Ground	—
	OUT	GPIO23	Digital output	High on motion detection
RGB LED	R	GPIO26	Red channel	LEDC PWM (8-bit)
	G	GPIO27	Green channel	LEDC PWM (8-bit)
	B	GPIO14	Blue channel	LEDC PWM (8-bit)
	GND	GND	Ground	—

The ESP32 node employs an identical mTLS configuration to the ESP32-S3 (Section 4.1.1). As shown in Figure 4.5, this node emphasizes on-device automation rather than edge analytics. The MQ-2 smoke sensor triggers threshold-based alerts at 5 Hz sampling, while the SR602 PIR sensor operates in interrupt-driven mode. Both sensors directly control the RGB LED indicator without cloud dependency.

[4] `https://cdn-shop.adafruit.com/datasheets/TCS34725.pdf`

[5] `https://www.winsen-sensor.com/d/files/semiconductor/mq-2.pdf`

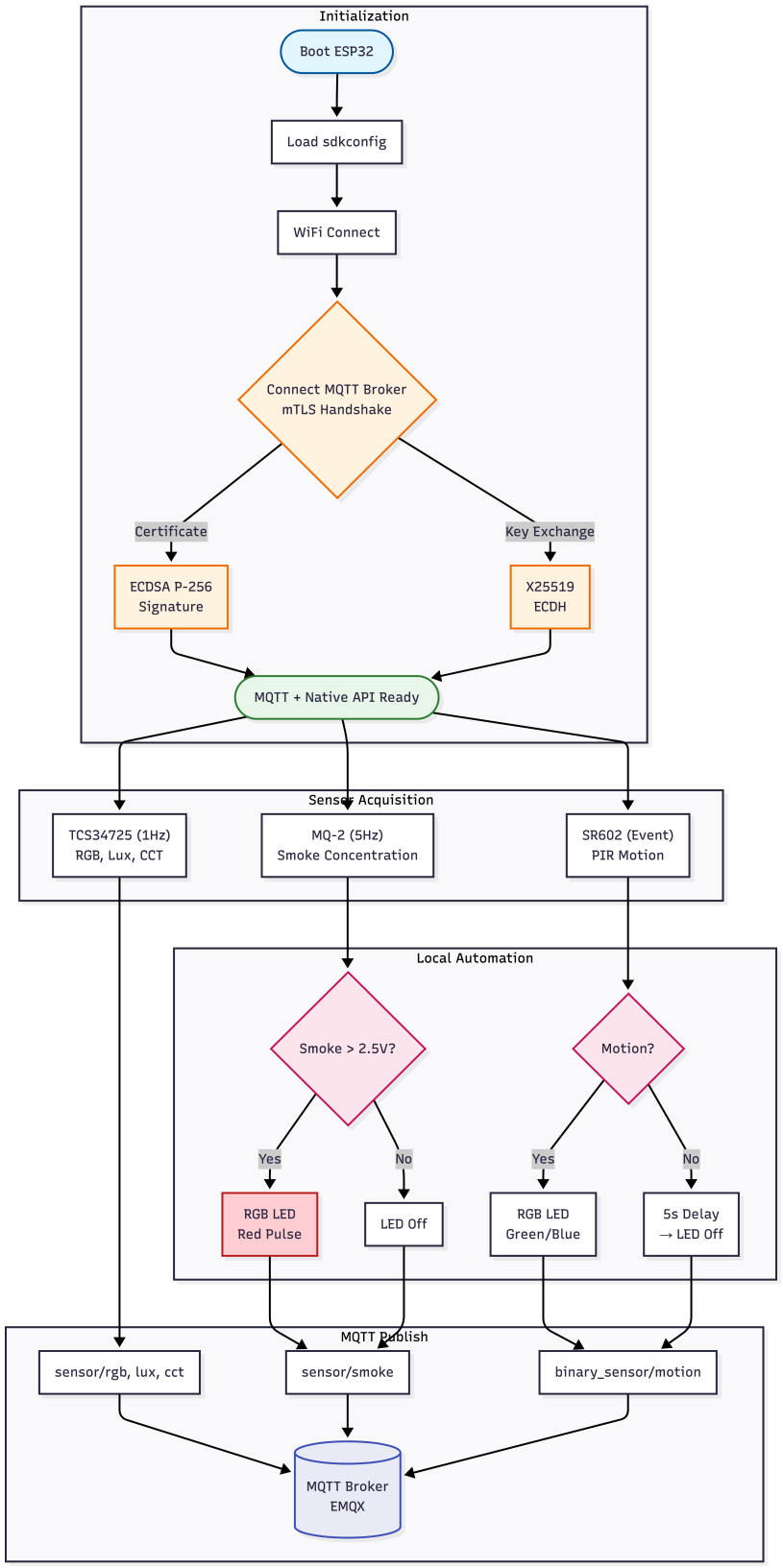


Figure 4.5: ESP32 firmware execution flowchart: initialization with mTLS handshake, sensor acquisition, local automation rules for smoke detection and motion response, and MQTT state publishing. The complete firmware listing is provided in Appendix C.3.

Local automation rules execute directly on the device: when the MQ-2 sensor voltage exceeds 2.5V, the RGB LED activates red pulsing effect and publishes an alert message to MQTT; SR602 motion detection triggers green LED illumination for visual feedback. Both nodes employ the dual-path communication design described in Section 3.3, with `discovery: true` enabling automatic entity registration in Home Assistant through the MQTT discovery protocol. The complete ESPHome configuration listings for both nodes, including TLS configuration and edge computing lambda expressions, are provided in Appendix C.

4.2 Software and Server Implementation

The deployed system is operated as a hybrid-cloud infrastructure comprising four distinct nodes that span public cloud locations and a private home network. The nodes are interconnected using a Tailscale mesh VPN that implements a Zero Trust Network Access (ZTNA) model, enabling service exposure without assigning a public IP address to the local controller.

4.2.1 Server Environment and Virtualization

The hybrid-cloud infrastructure comprises four nodes spanning public cloud locations and a private home network (Table 4.7). The deployment demonstrates portability across heterogeneous hardware, including both x86_64 and ARM64 architectures.

Table 4.7: Deployment Host Runtime Summary

Node	Location	Role	OS / Kernel	Arch / Resources
NUE	Nuremberg, DE	EMQX broker (Docker)	Debian 13 / 6.12.57	x86_64, 4 vCPU, 8 GB
FARM	Frankfurt, DE	SSL termination, bastion	Debian 12 / 6.1.0-41	arm64, 4 vCPU, 24 GB
HKG	Hong Kong, CN	Stress testing	Debian 12 / 6.1.0-39	x86_64, 2 vCPU, 2 GB
HAOS	Prague, CZ	Home Assistant (ESXi VM)	HAOS / 6.12.51-haos	x86_64, 4 vCPU, 4 GB

Internet-facing nodes (NUE, FARM, HKG) enable BBR congestion control with FQ queueing to improve throughput stability [49], while the local controller (HAOS) uses the default cubic/fq_codel configuration. For large-scale experiments, open-file limits were raised to 10^6 on public nodes, and TCP keepalive was set to 10 s on HKG for rapid detection of broken connections.

4.2.2 EMQX Broker Deployment

EMQX[3] was selected as the MQTT broker for its performance characteristics and native mTLS support [50]. The broker is hosted on the NUE node, exposing a single Internet-facing endpoint on port 8883 with mutual TLS authentication. The management dashboard is bound to loopback and published through OpenResty reverse proxy.

The broker was deployed using Docker (Docker Compose) to ensure reproducible configuration across environments. In high-concurrency scenarios, the Linux per-process file descriptor limit inside the container can become a bottleneck (a common default is 1024). Therefore, the deployment explicitly increases the `nofile` limit via Compose `ulimits` (Appendix D). This adjustment is required to support large numbers of simultaneous MQTT/TLS connections.

4.2.3 Python Sensor Simulator

The Python sensor simulator (`smarthome_sim` package) provides virtual entity emulation for testing Home Assistant integration without physical hardware. Key features include configurable

[3] `https://www.emqx.io/`

device counts, Home Assistant discovery payloads, mTLS connectivity, and multi-process execution. The simulator also supports load generation for scalability benchmarking as a secondary function, with results presented in Chapter 5. Key code excerpts demonstrating the mTLS connection handling and entity simulation logic are provided in Appendix E. A sample terminal output showing the simulator in operation is presented in Appendix F.

Figure 4.6 illustrates the high-level workflow of the Python simulator, from configuration loading through entity publishing.

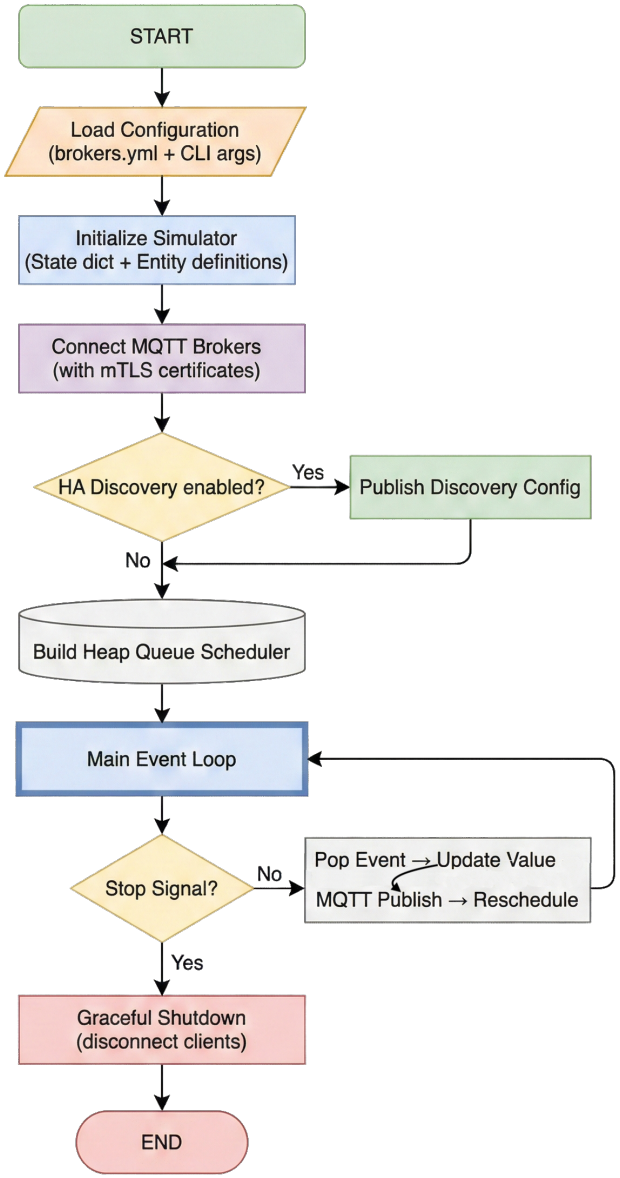


Figure 4.6: Python sensor simulator workflow: configuration loading, mTLS connection establishment, Home Assistant discovery publishing, and event-driven simulation loop with heap-based scheduling.

ESPHome configurations are maintained as YAML files supporting custom logic through lambda expressions for edge computing calculations. The complete ESPHome configuration files for both sensor nodes are provided in Appendix C.

4.3 Home Assistant Integration

Home Assistant's MQTT discovery feature enables automatic entity registration without manual configuration. Devices publish JSON payloads to designated discovery topics containing entity metadata, state topics, and device information. This mechanism reduces deployment friction and ensures consistent entity configuration across restarts. Home Assistant dashboards provide real-time visualization of sensor data and control interfaces, including sensor readings with historical graphs, device connectivity status, automation trigger history, and alert notifications.

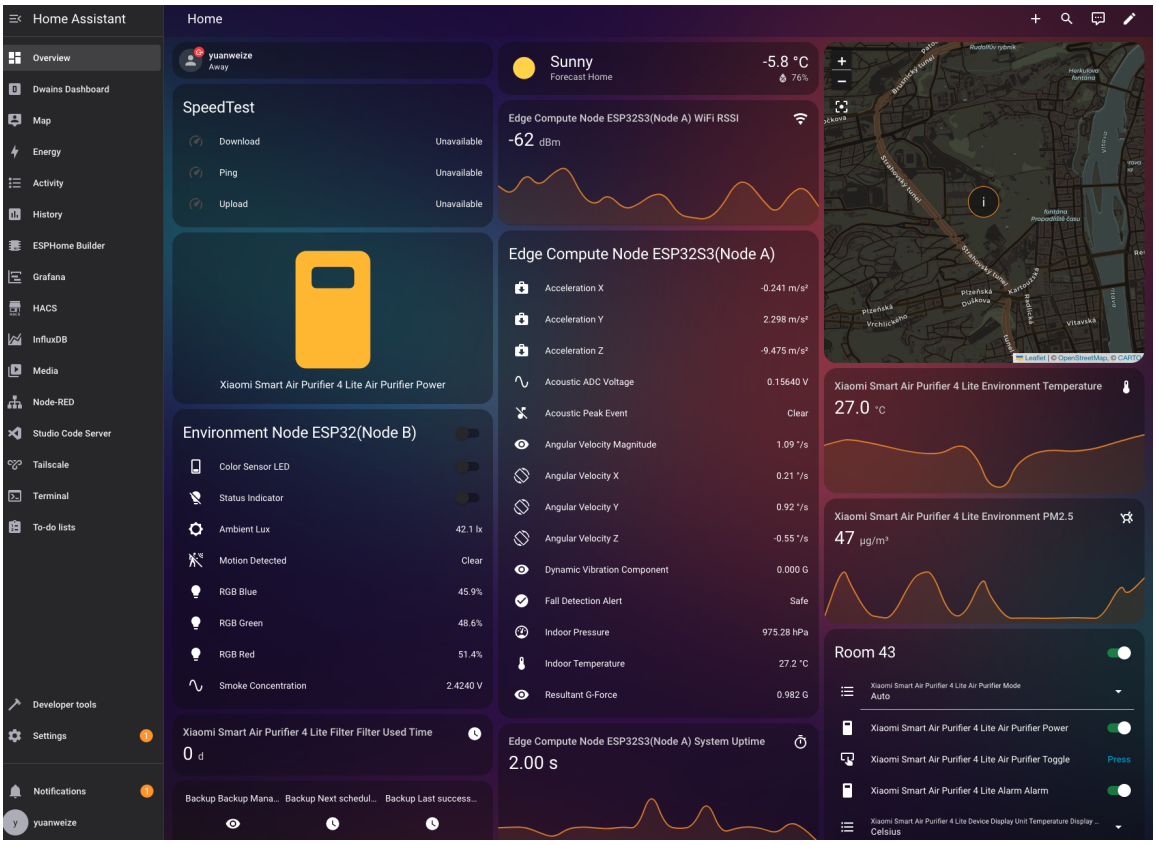


Figure 4.7: Home Assistant dashboard displaying real-time sensor data and device controls.

4.4 Data Pipeline and Automation

InfluxDB[4] serves as the time-series database for long-term sensor data retention [51]. Home Assistant's native integration automatically writes entity state changes, enabling historical analysis, trend detection, and anomaly identification. Figure 4.8 demonstrates a query result showing one hour of G-force measurements from the ESP32-S3 fall detection sensor, with data points aggregated at regular intervals. Grafana[5] connects to InfluxDB as a data source, providing advanced visualization capabilities including multi-sensor overlay comparisons, statistical aggregations, threshold-based alerting, and custom time range analysis (Figure 4.9).

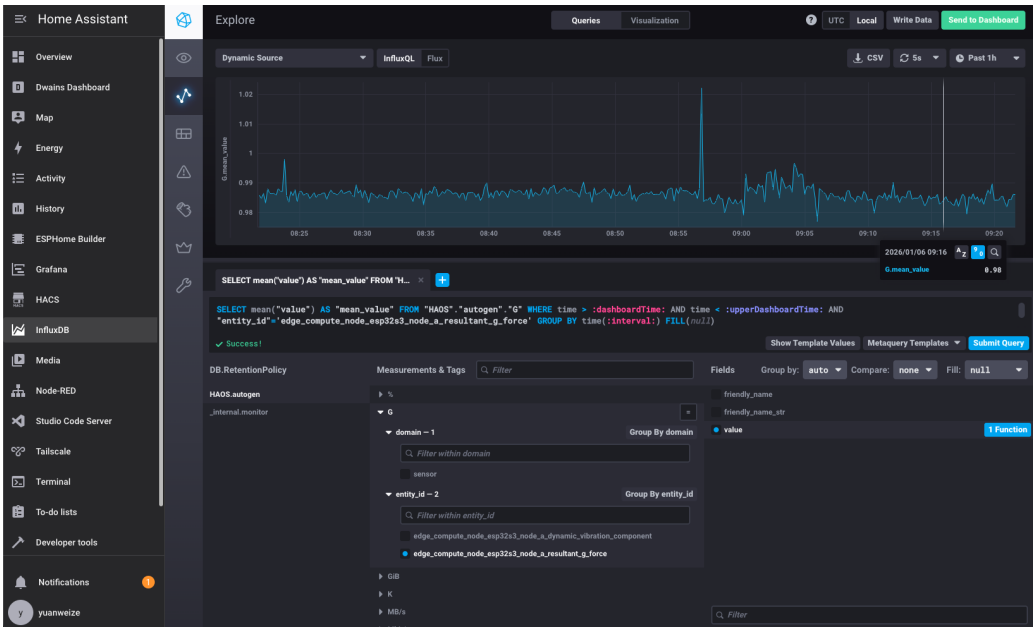


Figure 4.8: InfluxDB query result showing G-force sensor data over a one-hour period, demonstrating time-series data retention for historical analysis.

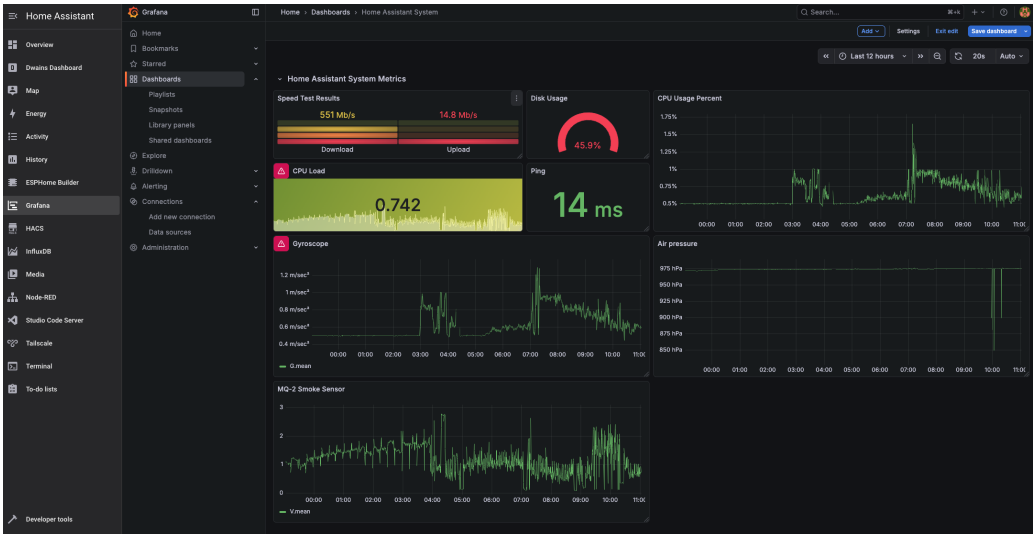


Figure 4.9: Grafana dashboard displaying real-time sensor data: barometric pressure (hPa), G-force magnitude, and MQ-2 smoke sensor analog voltage (V).

Native Home Assistant automations handle event-driven responses using YAML configuration,

[4] https://www.influxdata.com/
[5] https://grafana.com/

demonstrating integration between hardware sensors and mobile notifications. Node-RED[6] provides visual flow-based programming for complex automation scenarios requiring multi-branch conditional logic, data transformation, or external API integration. Figure 4.10 illustrates the fall detection notification flow implemented in Node-RED.

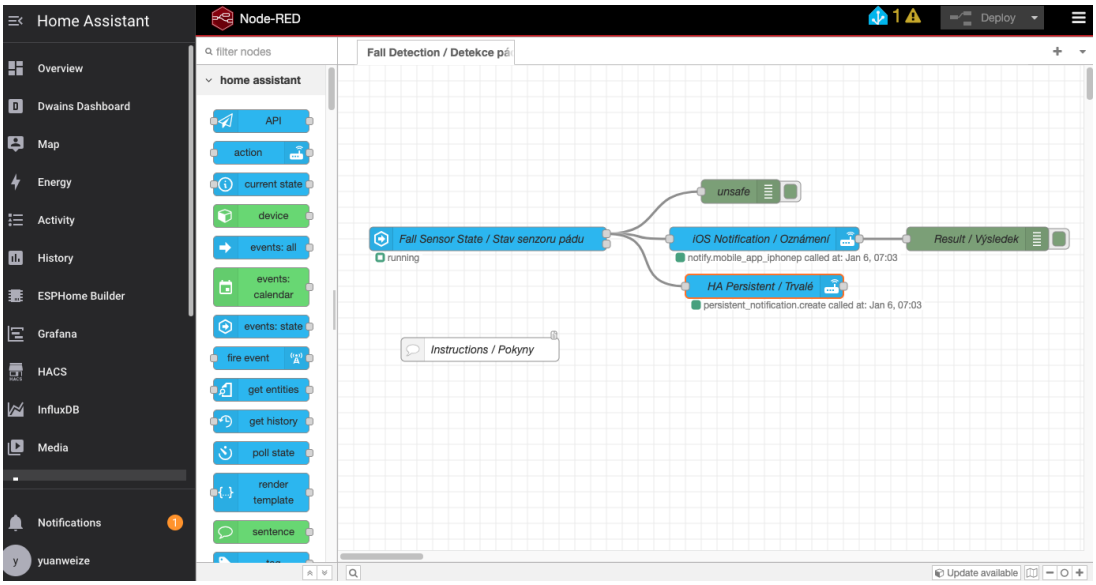


Figure 4.10: Node-RED flow for fall detection notification: sensor state triggers iOS push notification and persistent HA alert.

Figure 4.11 demonstrates the complete fall detection pipeline in operation. The ESP32-S3 node continuously computes the Signal Magnitude Vector (SMV) from three-axis accelerometer data and the angular velocity magnitude from three-axis gyroscope data. When both computed values exceed their respective thresholds (SMV $> 2.4\,\mathrm{G}$ and $\omega > 240\,^\circ/\mathrm{s}$), the binary sensor state transitions to `unsafe`, triggering the Node-RED automation to deliver notifications via iOS push and Home Assistant persistent notification.

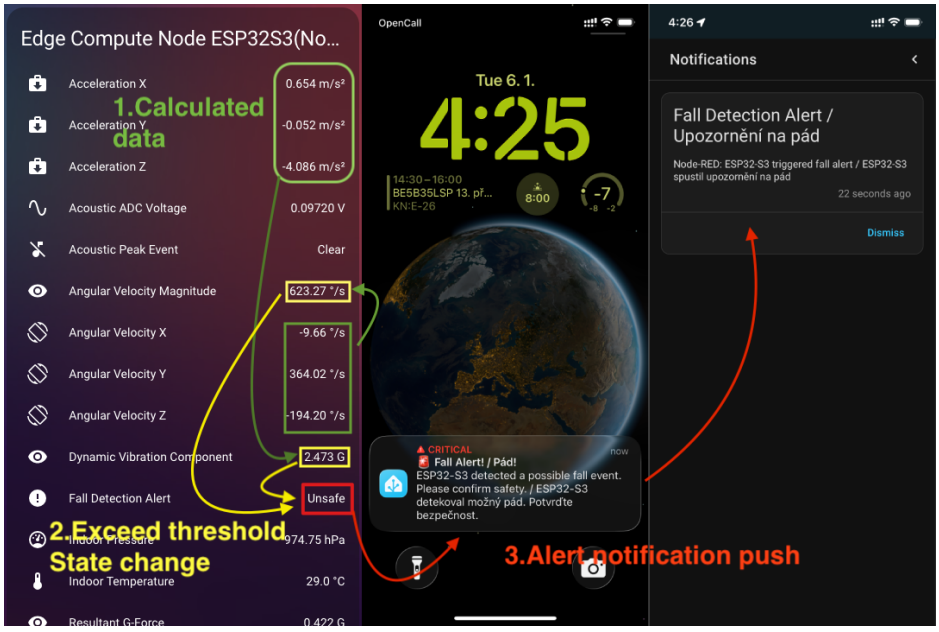


Figure 4.11: Fall detection demonstration: (P1) sensor data computation on ESP32-S3 dashboard, (P2) iOS critical notification, and (P3) Home Assistant persistent alert.

[6] `https://nodered.org/`

4.4.1 Backup Strategy

Home Assistant provides built-in backup functionality supporting both local and cloud storage destinations. The deployed system implements a two-tier backup strategy: local backups are created every three days with 14 retained copies (covering approximately six weeks of recovery points), while cloud backups to Google Drive are performed daily with 30-day retention. This configuration ensures rapid recovery from local storage for recent issues while maintaining off-site protection against hardware failures or local disasters.

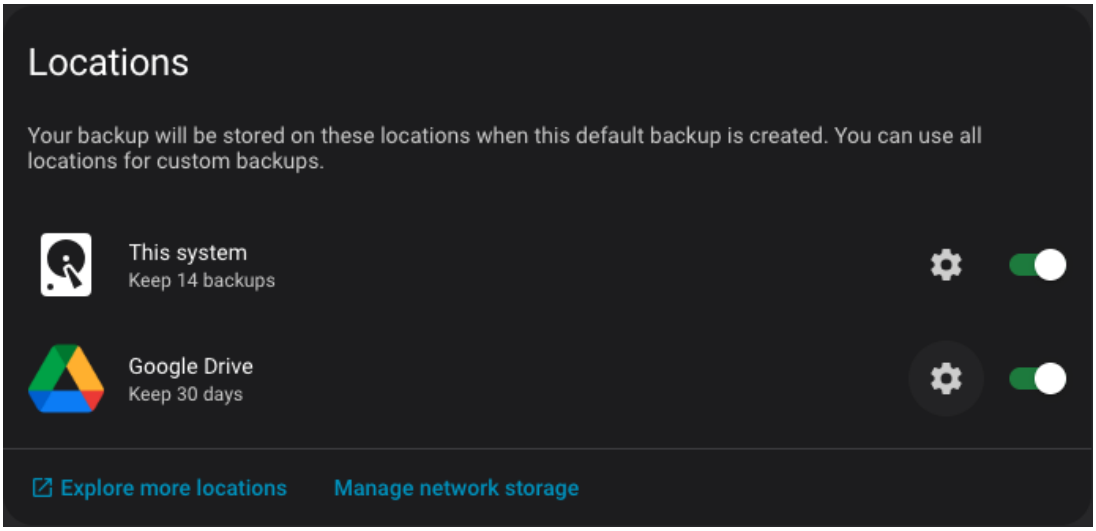


Figure 4.12: Home Assistant backup configuration interface showing local and cloud backup schedules.

While this approach does not constitute a full 3-2-1 backup strategy (three copies, two media types, one off-site), it provides practical data protection appropriate for a demonstration deployment. The combination of local and cloud storage destinations addresses the most common failure scenarios: accidental configuration changes can be recovered from local backups within seconds, while storage device failures are protected by cloud redundancy.

Chapter 5

Evaluation and Results

This chapter presents the evaluation methodology and experimental results, including performance metrics, security analysis, reliability assessment, and comparison with commercial solutions.

5.1 Test Environment

The evaluation was conducted using the production deployment described in Chapter 3, specifically the system components summarized in Table 3.1. To facilitate cross-network benchmarking, the Python simulator was deployed on a VPS in Hong Kong (2 vCPU x86_64, 2 GB RAM, Debian 12), providing geographic separation from the European infrastructure and enabling realistic wide-area network latency measurements.

5.2 Performance Metrics

Network path latencies were measured using `mtr` with approximately 200 samples per path. Table 5.1 presents the measured round-trip times for key communication paths in the deployed system.

Table 5.1: Network Path Latency Measurements (RTT)

Path	**Description**	**Avg (ms)**	**Best (ms)**	**Worst (ms)**
HKG → NUE	Simulator to EMQX broker	171.1	170.7	174.8
FARM → NUE	SSL termination to broker	6.4	6.3	6.6
NUE → HAOS	Broker to Home Assistant	8.4	8.0	9.4
FARM → HAOS	SSL termination to HA	7.7	7.6	8.2
HAOS → ESP32	Native API (LAN)	0.3	0.2	0.4

The end-to-end latency for MQTT messages from the Hong Kong simulator to Home Assistant in Prague comprises the HKG–NUE path (∼171 ms) plus the NUE–HAOS path (∼8 ms), totaling approximately 180 ms. In contrast, Native API connections within the local network achieve sub-millisecond latency, validating the dual-connection architecture design decision for local device responsiveness.

5.2.1 TLS Handshake Performance

The deployed system uses X25519 for key exchange. Due to ESP-IDF / ESP-TLS constraints on X.509 certificate signature algorithms, the deployed certificate chain uses ECDSA with NIST P-256 (`prime256v1`) for authentication, while retaining X25519 for ECDHE key exchange during

TLS 1.3 handshake. As established in Section 2.3.1, NIST SP 800-57 confirms that 256-bit ECC provides equivalent security to 3072-bit RSA with significantly smaller key sizes [14].

To contextualize the cost of cryptographic primitives during connection establishment, this thesis references the third-party microbenchmark published by Oryx Embedded for ESP32-S3 (Xtensa LX7 @ 240 MHz), generated with CycloneCRYPTO 2.5.0 and compiled with GCC at optimization level `-O3` [16]. The report lists ECDH (X25519) at approximately 14 ms for public key generation and 15 ms for shared secret computation. These figures are used as indicative reference points; absolute timings depend on the specific TLS stack, build configuration, and hardware acceleration availability.

5.2.2 Hybrid TLS Configuration Verification

To validate the hybrid cryptographic approach described in Section 2.3.2, the deployed TLS configuration was tested using OpenSSL's diagnostic client. Table 5.2 summarizes the negotiated parameters observed during mTLS handshake with the production EMQX broker.

Table 5.2: TLS Handshake Verification Results

Parameter	**Negotiated Value**
Protocol	TLS 1.3
Cipher Suite	`TLS_AES_256_GCM_SHA384`
Key Exchange (Peer Temp Key)	X25519, 253 bits
Signature Algorithm	`ecdsa_secp256r1_sha256`
Server Certificate	ECDSA P-256 (prime256v1)
Client Certificate	ECDSA P-256 (prime256v1)
Certificate Verification	OK

The verification confirms that the hybrid configuration operates as designed: certificates are signed using ECDSA P-256 (`ecdsa_secp256r1_sha256`), while key exchange utilizes X25519. This result demonstrates that TLS 1.3 successfully decouples the certificate signature algorithm from the key exchange mechanism, enabling optimization of each independently [27].

A comparative test was conducted by explicitly restricting the key exchange group to P-256:

```
openssl s_client -connect mqtt.eurun.top:8883 \
  -tls1_3 -groups P-256
```

This test confirmed fallback behavior: when X25519 is excluded from the `supported_groups` extension, the handshake successfully negotiates ECDH with P-256 instead (`Peer Temp Key: ECDH, prime256v1, 256 bits`). This demonstrates the system's backward compatibility while preferring X25519 when available.

```
(.venv) root@HKG /opt/SmartHome_Server/sensors # openssl s_client -connect mqtt.eurun.top:8883 -tls1_3 2>/de
v/null | tail -20
Acceptable client certificate CA names
CN = SmartHome Root CA, O = CTU FEL, C = CZ
Requested Signature Algorithms: RSA+SHA256:RSA+SHA384:RSA+SHA512:RSA-PSS+SHA256:RSA-PSS+SHA384:RSA-PSS+SHA51
2:rsa_pss_pss_sha256:rsa_pss_pss_sha384:rsa_pss_pss_sha512:ecdsa_brainpoolP256r1_sha256:ecdsa_brainpoolP384r
1_sha384:ecdsa_brainpoolP512r1_sha512:ECDSA+SHA256:ECDSA+SHA384:ECDSA+SHA512:ed448:ed25519
Shared Requested Signature Algorithms: RSA+SHA256:RSA+SHA384:RSA+SHA512:RSA-PSS+SHA256:RSA-PSS+SHA384:RSA-PS
S+SHA512:rsa_pss_pss_sha256:rsa_pss_pss_sha384:rsa_pss_pss_sha512:ECDSA+SHA256:ECDSA+SHA384:ECDSA+SHA512:ed4
48:ed25519
Peer signing digest: SHA256
Peer signature type: ECDSA
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 1466 bytes and written 358 bytes
Verification error: self-signed certificate in certificate chain
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 256 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 19 (self-signed certificate in certificate chain)
---
(.venv) root@HKG /opt/SmartHome_Server/sensors #
```

Figure 5.1: TLS handshake with X25519 key exchange (default configuration).

```
(.venv) root@HKG /opt/SmartHome_Server/sensors # openssl s_client -connect mqtt.eurun.top:8883 -tls1_3 -grou
ps P-256 2>/dev/null | tail -20
Acceptable client certificate CA names
CN = SmartHome Root CA, O = CTU FEL, C = CZ
Requested Signature Algorithms: RSA+SHA256:RSA+SHA384:RSA+SHA512:RSA-PSS+SHA256:RSA-PSS+SHA384:RSA-PSS+SHA51
2:rsa_pss_pss_sha256:rsa_pss_pss_sha384:rsa_pss_pss_sha512:ecdsa_brainpoolP256r1_sha256:ecdsa_brainpoolP384r
1_sha384:ecdsa_brainpoolP512r1_sha512:ECDSA+SHA256:ECDSA+SHA384:ECDSA+SHA512:ed448:ed25519
Shared Requested Signature Algorithms: RSA+SHA256:RSA+SHA384:RSA+SHA512:RSA-PSS+SHA256:RSA-PSS+SHA384:RSA-PS
S+SHA512:rsa_pss_pss_sha256:rsa_pss_pss_sha384:rsa_pss_pss_sha512:ECDSA+SHA256:ECDSA+SHA384:ECDSA+SHA512:ed4
48:ed25519
Peer signing digest: SHA256
Peer signature type: ECDSA
Server Temp Key: ECDH, prime256v1, 256 bits
---
SSL handshake has read 1501 bytes and written 373 bytes
Verification error: self-signed certificate in certificate chain
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 256 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 19 (self-signed certificate in certificate chain)
---
(.venv) root@HKG /opt/SmartHome_Server/sensors #
```

Figure 5.2: TLS handshake with P-256 key exchange (fallback test using `-groups P-256`).

5.2.3 mTLS Connection Latency Benchmark

To empirically measure the end-to-end mTLS connection establishment latency, the Python simulator's handshake benchmark mode (Python 3.11.2, paho-mqtt 1.6) was executed from the FARM host (Frankfurt, Germany) to an EMQX 6.0.0 broker (Nuremberg, Germany). This path exhibits a network RTT of approximately 6.4 ms (Table 5.1), minimizing the impact of network propagation delay on the measurement.

Table 5.3 summarizes the measured mTLS handshake latency over 800 connection samples. The complete benchmark methodology and raw data are provided in Appendix H.

Table 5.3: mTLS Handshake Latency Benchmark (FARM → NUE, $n = 800$)

Mean	P50	P90	P95	P99	Min	Max	Std	95% CI
80.50	80.55	91.61	92.82	96.79	67.84	104.07	7.49	[79.98, 81.02]

All values in milliseconds (ms). Pn denotes the nth percentile; Std = standard deviation; 95% CI = 95% confidence interval for the mean.

The measured mean latency of 80.50 ms represents the complete mTLS connection establishment time from an x86_64 host, including network round-trips and cryptographic operations. This benchmark primarily validates the network path stability and the EMQX broker's mTLS configuration—it does not reflect ESP32 client performance due to fundamental differences in hardware capability (x86 vs. Xtensa LX7) and TLS stack (OpenSSL vs. MbedTLS).

For ESP32-S3 performance estimation, Table 2.6 provides cryptographic primitive timings (approximately 89 ms for handshake-related operations). Combined with network RTT, total connection latency for ESP32 clients on similar paths would be approximately 100–120 ms—well within acceptable bounds for IoT reconnection scenarios.

The P99 latency of 96.79 ms confirms that even worst-case connection times remain under 100 ms, which is suitable for IoT applications requiring responsive reconnection behavior. The narrow 95% confidence interval (±0.52 ms) and zero connection failures across 800 samples demonstrate excellent measurement repeatability and connection reliability.

5.2.4 Scalability Testing

The Python simulator was used to evaluate system scalability by progressively increasing the number of simulated devices. Each test configuration was monitored for approximately 5 minutes to capture steady-state resource utilization. Monitoring screenshots from EMQX Dashboard, broker system metrics, and Home Assistant are provided in Appendix G.

Table 5.4: Scalability Test Results

Devices	Messages/sec	Broker CPU (%)	HA CPU (%)
100	∼229	3.72	1.1
1,000	∼1,618	4.03	1.4

The test results demonstrate that the EMQX broker handles 1,000 concurrent simulated devices with minimal resource impact—CPU utilization increased by only 0.31 percentage points (from 3.72% to 4.03%) when scaling from 100 to 1,000 devices, while message throughput increased proportionally from 229 to 1,618 messages per second. Home Assistant CPU utilization remained low at 1.1–1.4%, indicating substantial headroom for additional devices.

Testing beyond 1,000 devices was limited by the simulator host's computational capacity (2 vCPU VPS). At 5,000 devices, the simulator process itself reached 100% CPU utilization, making further scalability measurements unreflective of broker or Home Assistant limits. Nevertheless, the observed linear relationship between device count and message throughput, combined with near-constant broker CPU utilization, suggests the architecture can support significantly larger deployments. EMQX documentation reports theoretical limits of over 100 million concurrent connections in clustered configurations[1] [50].

[1] `https://www.emqx.com/en/blog/reaching-100m-mqtt-connections-with-emqx-5-0`

5.3 Security and Reliability

The implemented security architecture addresses common IoT threat vectors through transport layer protection. All MQTT communications are encrypted using TLS 1.2/1.3, preventing eavesdropping and tampering [27]. Mutual TLS (mTLS) authentication ensures that only devices with valid client certificates can connect to the broker, mitigating unauthorized access and device impersonation attacks [13].

The current implementation uses a single Certificate Authority for all certificates. While suitable for small deployments, production systems should consider intermediate CAs, automated certificate rotation, and CRL distribution.

The system demonstrated stable operation during the development and testing period. ESP32 nodes exhibited resilient behavior, automatically reconnecting after WiFi disconnections or broker restarts. The dual-connection architecture proved valuable during internet outages, maintaining local control capability through Native API while MQTT connectivity was disrupted.

5.4 Comparison with Commercial Solutions

This section compares the implemented system with commercial smart home platforms, analyzing trade-offs between different approaches.

Xiaomi's Mi Home platform employs a cloud-centric architecture where device states are synchronized via cloud-based MQTT subscription and commands are transmitted through HTTP APIs[2]. Strengths include competitive device pricing, intuitive mobile application, and wide device ecosystem. Limitations include cloud dependency for most features (though local control is available for some devices via gateway or LAN protocols), regional server restrictions, and limited customization options. The Xiaomi Air Purifier integrated in this thesis demonstrates interoperability through Home Assistant's Xiaomi Miio integration.

Voice-controlled platforms from Amazon and Google process commands through cloud-based speech recognition. Research has documented privacy implications of voice assistant data collection [3]. The Matter protocol initiative aims to improve cross-platform compatibility [52]. Due to device availability constraints, analysis relies on published documentation and academic studies.

5.4.1 Comparative Analysis

Table 5.5 presents a comparison across key evaluation criteria.

Table 5.5: Platform Comparison Matrix

Criterion	This System	Xiaomi	Alexa	Google
Data Storage	Local	Cloud	Cloud	Cloud
Privacy Control	Full	Limited	Limited	Limited
Customization	High	Low	Medium	Medium
Setup Complexity	Medium	Low	Low	Low
Offline Capability	Full	Partial	Minimal	Minimal
Vendor Independence	Full	None	None	None

Premium features may require subscription on commercial platforms

The fundamental architectural difference lies in data storage location. Commercial platforms transmit sensor data to cloud servers, where it may be analyzed for service improvement, used

[2] `https://www.home-assistant.io/integrations/xiaomi_miio/`

for targeted advertising, or subject to data breaches. The implemented system maintains all data locally.

Vendor dependency creates sustainability risks, as demonstrated by historical platform discontinuations. Open-source solutions eliminate this dependency by enabling community maintenance independent of any single vendor's business decisions.

The comparison reveals a fundamental trade-off between convenience and control. The self-hosted approach has higher upfront complexity but offers greater control and learning value. Open-source frameworks like ESPHome can significantly lower the entry barrier compared to writing custom firmware [29], [31]. Commercial ecosystems minimize initial setup complexity but introduce stronger vendor dependency [4].

The growing availability of Matter-compatible devices may reduce the technical barrier to local-first smart home deployments, potentially expanding adoption beyond technically sophisticated users.

Chapter 6

Conclusion

This chapter summarizes the thesis contributions, acknowledges limitations, and suggests directions for future work.

6.1 Summary of Contributions

This thesis presented the design and implementation of a smart home sensor control system utilizing servers and Unix-like operating systems. The primary contributions are:

1. **Distributed Architecture Design:** A geographically distributed system architecture was designed with components spanning multiple locations (Nuremberg, Frankfurt, Prague, Hong Kong), demonstrating practical deployment of cross-region IoT infrastructure with secure communications.

2. **Dual Hardware Node Implementation:** Two ESP32-based sensor nodes were implemented demonstrating different IoT paradigms:

   - ESP32-S3 edge intelligence node performing local signal processing (vibration analysis, acoustic event detection)
   - ESP32 environment sensing node with safety monitoring capabilities (smoke detection, occupancy sensing)

3. **Virtual Entity Simulation Framework:** A Python-based sensor simulator was developed for virtual entity emulation and load generation, enabling integration testing and scalability benchmarking beyond physical hardware limitations.

4. **Security Implementation:** Mutual TLS authentication was implemented using a hybrid cryptographic approach: ECDSA P-256 for certificate signatures (due to ESP-IDF library constraints) combined with X25519 for key exchange, balancing compatibility with performance [16], [23].

5. **Commercial Platform Comparison:** A qualitative comparison with commercial solutions (Xiaomi Mi Home) was conducted based on published documentation, highlighting trade-offs between convenience and data sovereignty.

The implemented system successfully demonstrates that open-source smart home solutions can provide comparable functionality to commercial alternatives while maintaining complete user control over data and system behavior.

6.2 Limitations

The following limitations should be acknowledged:

- **Setup Complexity:** The system requires significant technical expertise for initial deployment, including certificate generation, server configuration, and network setup. This barrier limits accessibility for non-technical users.
- **Hardware Scale:** Only two physical sensor nodes were implemented. Production deployments with hundreds of devices may encounter unforeseen challenges.
- **Voice Integration:** No voice control interface was implemented, limiting comparison with voice-centric commercial platforms.
- **Limited Commercial Testing:** Direct integration testing with Amazon Alexa and Google Home devices was not performed due to availability constraints; comparison relied on published documentation.
- **Scalability Validation:** While the simulator supports large device counts, detailed CPU utilization measurements under load were not completed within the thesis timeline.

6.3 Future Work

Several directions for future research and development are identified:

1. **Matter Protocol Integration:** Implementing Matter protocol support would enable interoperability with an expanding ecosystem of compatible devices while maintaining local control principles.
2. **Machine Learning at the Edge:** The ESP32-S3's vector instructions could be leveraged for more sophisticated edge AI applications, such as anomaly detection or predictive maintenance.
3. **Automated Certificate Management:** Implementing automated certificate lifecycle management (issuance, renewal, revocation) would reduce operational overhead for larger deployments.
4. **Energy Monitoring:** Extending the system to include power consumption monitoring and optimization would provide additional value for energy-conscious users.
5. **User Interface Improvements:** Developing simplified setup wizards and mobile applications could reduce the technical barrier for non-expert users.
6. **Custom Enclosure Fabrication:** The modular hardware design is well-suited for 3D-printed enclosures, enabling rapid customization for specific deployment environments. Recent work has demonstrated the feasibility of FDM-based enclosure prototyping for IoT sensor nodes, reducing fabrication time from weeks to hours while allowing iterative design refinement [53], [54].
7. **Redundancy and Failover:** Implementing MQTT broker clustering and Home Assistant high-availability would improve system resilience for critical applications.
8. **Cryptographic Library Contributions:** As noted in Section 2.3.2, the current ESP-IDF Mbed TLS implementation lacks Ed25519 support for X.509 certificate signatures. Should the opportunity arise, contributing to the upstream Mbed TLS project to address this limitation [24] would benefit the broader embedded systems community and enable pure Curve25519-based mTLS deployments on ESP32 platforms.

The growing adoption of local-first smart home solutions and the emergence of interoperability standards like Matter suggest a promising trajectory for open-source home automation. Continued development in this area has the potential to democratize smart home technology while preserving user privacy and autonomy.

Bibliography

[1] B. K. Sovacool and D. D. F. Del Rio, "Smart home technologies in Europe: A critical review of concepts, benefits, risks and policies", *Renewable and Sustainable Energy Reviews*, vol. 120, p. 109 663, 2020. DOI: 10.1016/j.rser.2019.109663

[2] P. Sethi and S. R. Sarangi, "Internet of Things: Architectures, protocols, and applications", *Journal of Electrical and Computer Engineering*, vol. 2017, pp. 1–25, 2017. DOI: 10.1155/2017/9324035

[3] J. S. Edu, J. M. Such, and G. Suarez-Tangil, "Smart home personal assistants", *ACM Computing Surveys*, vol. 53, no. 6, pp. 1–36, 2021, Subtitle: A Security and Privacy Review. Published online 2020-12-06; published in print 2021-11-30. DOI: 10.1145/3412383

[4] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "SoK: Security evaluation of home-based IoT deployments", in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1362–1380. DOI: 10.1109/SP.2019.00013

[5] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani, "Demystifying IoT security: An exhaustive survey on IoT vulnerabilities and a first empirical look on internet-scale IoT exploitations", *IEEE Communications Surveys & Tutorials*, vol. 21, no. 3, pp. 2702–2733, 2019. DOI: 10.1109/COMST.2019.2910750

[6] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges", in *IEEE Internet of Things Journal*, vol. 3, 2016, pp. 637–646. DOI: 10.1109/JIOT.2016.2579198

[7] A. Khan, S. Al-Zahrani, M. Fahad, and A. Saad, "Design of an IoT smart home system", in *2018 15th Learning and Technology Conference (L&T)*, Presents IoT-based smart home design with sensor integration and remote monitoring., IEEE, 2018, pp. 1–5. DOI: 10.1109/LT.2018.8368484

[8] H. V. Bhatnagar, P. Kumar, S. Rawat, and T. Choudhury, "Implementation model of Wi-Fi based smart home system", in *2018 International Conference on Advanced Computation and Communication Engineering (ICACCE)*, Discusses Wi-Fi-based smart home architecture using microcontrollers and mobile applications., IEEE, 2018. DOI: 10.1109/ICACCE.2018.8441703

[9] OASIS, *MQTT version 5.0*, OASIS Standard, Accessed: 2025-11-03, 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[10] N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP", *IEEE International Systems Engineering Symposium (ISSE)*, pp. 1–7, 2017. DOI: 10.1109/SysEng.2017.8088251

[11] T. Yokotani and Y. Sasaki, "Comparison with HTTP and MQTT on required network resources for IoT", in *International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC)*, 2016, pp. 1–6. DOI: 10.1109/ICCEREC.2016.7814989

[12] EMQX Team, *Introduction to MQTT publish-subscribe pattern*, EMQX Blog, Accessed: 2025-11-08, 2023. [Online]. Available: https://www.emqx.com/en/blog/mqtt-5-introduction-to-publish-subscribe-model

[13] Cloudflare, *What is mutual TLS (mTLS)?*, Cloudflare Learning Center, Accessed: 2025-11-22, 2024. [Online]. Available: `https://www.cloudflare.com/learning/access-management/what-is-mutual-tls/`

[14] National Institute of Standards and Technology, "Recommendation for key management: Part 1 – general", NIST, Special Publication 800-57 Part 1 Rev. 5, 2020, Defines security strength equivalences for cryptographic algorithms; Table 2 (pp. 54–55) used for ECC vs RSA key-size equivalence. DOI: `10.6028/NIST.SP.800-57pt1r5`

[15] Oryx Embedded, *Crypto benchmark on ESP32 MCU (xtensa lx6 @ 240 MHz)*, Technical Benchmark Report (CycloneCRYPTO 2.5.0), Accessed: 2025-12-22. Ed25519: 29 ms sign / 28 ms verify (software); ECDSA P-256: 62 ms / 57 ms (hardware); RSA-3072: 465 ms / 314 ms (hardware)., 2024. [Online]. Available: `https://www.oryx-embedded.com/benchmark/espressif/crypto-esp32.html`

[16] Oryx Embedded, *Crypto benchmark on ESP32-S3 MCU (xtensa lx7 @ 240 MHz)*, Technical Benchmark Report (CycloneCRYPTO 2.5.0), Accessed: 2025-12-18. Benchmark generated with CycloneCRYPTO 2.5.0 and GCC -O3 on ESP32-S3 @ 240MHz. Selected figures: ECDH (X25519) 14 ms (pubkey) / 15 ms (shared secret) (software); RSA-2048 490 ms (sign, software) / 118 ms (sign, hardware); ECDSA (secp256r1) 67 ms (sign, hardware) / 60 ms (verify, hardware); Ed25519 26 ms (sign, software) / 24 ms (verify, software)., 2024. [Online]. Available: `https://www.oryx-embedded.com/benchmark/espressif/crypto-esp32-s3.html`

[17] D. J. Bernstein, *Curve25519: New diffie-hellman speed records*, Cryptology ePrint Archive / technical report, Accessed: 2025-12-12. Introduces Curve25519 and its efficient Montgomery-ladder-based scalar multiplication., 2006. [Online]. Available: `https://cr.yp.to/ecdh/curve25519-20060209.pdf`

[18] A. Langley, M. Hamburg, and S. Turner, *Elliptic curves for security*, RFC 7748, Accessed: 2025-11-10. Specifies X25519/X448 and discusses implementation considerations for secure, constant-time scalar multiplication., 2016. [Online]. Available: `https://www.rfc-editor.org/rfc/rfc7748`

[19] D. J. Bernstein and T. Lange, *SafeCurves: Choosing safe curves for elliptic-curve cryptography*, Cryptographic research website, Accessed: 2025-11-30. Evaluates ECC curves against security criteria including rigidity, twist safety, and completeness. NIST P-256 fails the rigidity test., 2017. [Online]. Available: `https://safecurves.cr.yp.to/`

[20] S. Josefsson and I. Liusvaara, *Edwards-curve digital signature algorithm (EdDSA)*, RFC 8032, Accessed: 2025-11-14. Specifies Ed25519/Ed448 and the deterministic nonce generation used by EdDSA., 2017. [Online]. Available: `https://www.rfc-editor.org/rfc/rfc8032`

[21] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures", in *Journal of Cryptographic Engineering*, The foundational paper for the Ed25519 signature algorithm., vol. 2, 2012, pp. 77–89. DOI: `10.1007/s13389-012-0027-1`

[22] D. J. Bernstein, *How to design an elliptic-curve signature system*, The cr.yp.to blog, Accessed: 2025-12-05. Critiques ECDSA design decisions; discusses DSA standardization by NSA/NIST and explains Ed25519's deterministic nonce derivation., Mar. 2014. [Online]. Available: `https://blog.cr.yp.to/20140323-ecdsa.html`

[23] Espressif Systems, *ESP-TLS: TLS library for ESP-IDF*, ESP-IDF Programming Guide, Accessed: 2025-12-11. Documents supported ECDSA curves (SECP256R1, SECP384R1) and TLS configuration options., 2024. [Online]. Available: `https://docs.espressif.com/projects/esp-idf/en/v5.5.2/esp32/api-reference/protocols/esp_tls.html`

[24] Mbed TLS Contributors, *Add support for Ed25519 in X.509 certificates*, GitHub Issue #2452, Accessed: 2025-12-21. Feature request for Ed25519 certificate signature support in Mbed TLS., 2019. [Online]. Available: `https://github.com/Mbed-TLS/mbedtls/issues/2452`

[25] Mbed TLS Project, *Mbed TLS roadmap*, Mbed TLS Documentation, Accessed: 2025-11-26. Official project roadmap indicates EdDSA (Ed25519) support planned for future releases., 2024. [Online]. Available: `https://mbed-tls.readthedocs.io/en/latest/project/roadmap/`

[26] wolfSSL Inc., *Wolfssl embedded SSL/TLS library*, Product Documentation, Accessed: 2025-12-14. Lightweight TLS library supporting TLS 1.3, Curve25519, Ed25519, and post-quantum cryptography; designed for embedded and resource-constrained environments., 2024. [Online]. Available: `https://www.wolfssl.com/products/wolfssl/`

[27] E. Rescorla, *The transport layer security (TLS) protocol version 1.3*, RFC 8446, Accessed: 2025-12-01. Defines TLS 1.3, including optional certificate-based client authentication used by mutual TLS deployments., 2018. [Online]. Available: `https://www.rfc-editor.org/rfc/rfc8446`

[28] Espressif Systems, *ESP-IDF sleep modes*, ESP-IDF Programming Guide, Accessed: 2025-12-27. Documents Light-sleep and Deep-sleep power saving modes; Wi-Fi/Bluetooth must be disabled before entering deep sleep, requiring connection re-establishment upon wake., 2024. [Online]. Available: `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/sleep_modes.html`

[29] Home Assistant, *Home assistant documentation*, `https://www.home-assistant.io/docs/`, Accessed: 2025-11-05, 2024.

[30] Home Assistant, *Home assistant base images (docker)*, GitHub repository documentation, Accessed: 2025-12-02, 2024. [Online]. Available: `https://github.com/home-assistant/docker-base`

[31] ESPHome, *ESPHome documentation*, `https://esphome.io/`, Accessed: 2025-11-18, 2024.

[32] Z. Xiao, D. Liu, D. Cao, and X. Wang, "Home appliance control system in smart home based on WiFi IoT", in *2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, Proposes a WiFi-based IoT architecture for home appliance control., IEEE, 2018. DOI: `10.1109/IAEAC.2018.8577217`

[33] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, 2010, ISBN: 9781593272203.

[34] OpenSSL Software Foundation, *OpenSSL: Cryptography and SSL/TLS toolkit*, `https://www.openssl.org/`, Accessed: 2025-11-15; widely deployed open-source TLS implementation, 2024.

[35] B. Gregg, *Systems Performance: Enterprise and the Cloud*, 2nd ed. Addison-Wesley, 2020.

[36] Home Assistant, *Home assistant operating system*, GitHub repository documentation, Accessed: 2025-11-28, 2024. [Online]. Available: `https://github.com/home-assistant/operating-system`

[37] J. A. Donenfeld, "WireGuard: Next generation kernel network tunnel", in *Network and Distributed System Security Symposium (NDSS)*, The underlying protocol used by Tailscale., Internet Society, 2017. DOI: `10.14722/ndss.2017.23160`

[38] Espressif Systems, *Non-volatile storage library (NVS)*, ESP-IDF Programming Guide, Accessed: 2025-11-19. Provides encrypted key-value storage for ESP32 credentials., 2024. [Online]. Available: `https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/storage/nvs_flash.html`

[39] Espressif Systems, *Esp32 datasheet*, Technical datasheet, Accessed: 2025-11-12, 2024. [Online]. Available: `https://documentation.espressif.com/esp32_datasheet_en.pdf`

[40] Espressif Systems, *Esp32-s3 datasheet*, Technical datasheet, Accessed: 2025-11-12, 2024. [Online]. Available: `https://documentation.espressif.com/esp32-s3_datasheet_en.pdf`

[41] NXP Semiconductors, *I2C-bus Specification and User Manual, Rev. 7.0*. NXP Semiconductors, 2021, Accessed: 2026-01-07. Defines I2C multi-master bus protocol with 7-bit addressing supporting up to 128 devices on shared SDA/SCL lines. [Online]. Available: https://www.nxp.com/docs/en/user-guide/UM10204.pdf

[42] Q. T. Huynh, U. D. Nguyen, L. B. Irazabal, N. Ghassemian, and B. Q. Tran, "Optimization of an accelerometer and gyroscope-based fall detection algorithm", *Journal of Sensors*, vol. 2015, pp. 1–8, 2015, Proposes optimized threshold values: UFT_acc = 2.4 G, UFT_gyro = 240°/s. Achieves 96.3% sensitivity and 96.2% specificity. DOI: 10.1155/2015/452078

[43] E. Casilari, M. Álvarez-Marco, and F. García-Lagos, "A study of the use of gyroscope measurements in wearable fall detection systems", *Symmetry*, vol. 12, no. 4, p. 649, 2020, CNN-based study using SisFall dataset. Concludes that gyroscope data does NOT improve deep learning fall detection; accelerometer-only achieves better results. DOI: 10.3390/sym12040649

[44] Espressif Systems, *ESP-NN: Optimised neural network functions for Espressif chipsets*, Accessed: 2025-12-09. Provides assembly-optimized kernels for ESP32-S3 vector instructions, achieving up to 10x speedup for convolution operations., 2024. [Online]. Available: https://github.com/espressif/esp-nn

[45] Espressif Systems, *ESP-TFLite-Micro: TensorFlow lite micro for Espressif chipsets*, Accessed: 2025-12-04. Person Detection benchmark: ESP32-S3 achieves 54ms inference with ESP-NN (vs 2300ms without)., 2024. [Online]. Available: https://github.com/espressif/esp-tflite-micro

[46] IEEE, "IEEE standard for inertial sensor terminology", Institute of Electrical and Electronics Engineers, Tech. Rep. IEEE Std 528-2019, 2019, Revision of IEEE Std 528-2001. Defines terminology for inertial sensors including bias, scale factor, and noise parameters. DOI: 10.1109/IEEESTD.2019.8863799

[47] InvenSense Inc., *MPU-6000 and MPU-6050 product specification revision 3.4*, Accessed: 2026-01-02. Specifies accelerometer zero-g output tolerance of ±80 mg and gyroscope zero-rate output of ±20 °/s, indicating the need for individual sensor calibration., InvenSense Inc., 2013. [Online]. Available: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf

[48] Zhengzhou Winsen Electronics, *MQ-2 semiconductor smoke sensor*, Manual Version 1.4 (2015-03-10). Heater voltage VH = 5.0 V ±0.1 V; heater resistance RH = 29 Ω ±3 Ω; heater power ≤950 mW; preheat time >48 hours., Zhengzhou Winsen Electronics Technology Co., Ltd., 2015. [Online]. Available: https://www.winsen-sensor.com/d/files/semiconductor/mq-2.pdf

[49] N. Cardwell, Y. Cheng, C. S. Gunn, S. Hassas Yeganeh, and V. Jacobson, "BBR: Congestion-based congestion control", *ACM Queue*, vol. 14, no. 5, 2016, Accessed: 2025-12-28. DOI: 10.1145/3012426.3022184 [Online]. Available: https://queue.acm.org/detail.cfm?id=3022184

[50] EMQ Technologies, *Reaching 100M MQTT connections with EMQX 5.0*, https://www.emqx.com/en/blog/reaching-100m-mqtt-connections-with-emqx-5-0, Accessed: 2025-11-25. Test achieved 100M concurrent connections on 23-node cluster (3 core + 20 replicant nodes) using c6g.metal instances., 2022.

[51] InfluxData, *InfluxDB: Open source time series database*, Official product documentation, Accessed: 2025-12-08; purpose-built time-series database for high-performance IoT data storage and querying, 2024. [Online]. Available: https://docs.influxdata.com/influxdb/v2/

[52] Connectivity Standards Alliance, *Matter specification version 1.0*, https://csa-iot.org/all-solutions/matter/, Accessed: 2025-12-15, 2022.

[53] N. M. Dhawale, D. V. Ghewade, S. S. Patil, R. S. Gangatirkar, and N. A. Inamdar, "An application of 3D printing technology for rapid prototyping of an IoT enabled sensor enclosure", *International Journal of Innovative Research in Science, Engineering and Technology (IJIRSET)*, vol. 11, no. 2, pp. 1178–1185, 2022, Presents methodology for designing and fabricating custom sensor enclosures using FDM 3D printing, comparing cost-effectiveness with injection moulding for IoT prototyping. DOI: 10.15680/IJIRSET.2022.1102038

[54] A. Osa-Sanchez and B. Garcia-Zapirain, "Real-time air quality monitoring: A smart IoT system using low-cost sensors and 3-D printing", *IEEE Journal of Radio Frequency Identification*, vol. 9, pp. 65–79, 2025, Demonstrates custom 3D-printed enclosures for portable IoT sensor nodes with Raspberry Pi; validated 95.30% pollutant detection reliability. DOI: 10.1109/JRFID.2025.3541816

Source Code Availability: The complete source code, configuration files, and documentation for this thesis project are available at:

`https://github.com/yuanweize/SmartHome_Server`

The GitHub repository may be updated with additional documentation and improvements; feedback and suggestions are welcome. A snapshot of the repository at the time of submission (06-01-2026) is also attached to this thesis in the CTU DSpace archive system[1].

[1] `https://dspace.cvut.cz/`

Appendix A

Certificate Generation

This appendix documents the certificate generation procedure used to establish the mutual TLS (mTLS) authentication implemented in this thesis. The system utilizes **ECDSA with NIST P-256** (`prime256v1`) for certificate signatures, as this algorithm is fully supported by the ESP-IDF's Mbed TLS implementation for X.509 certificate parsing and verification [23]. During TLS 1.3 handshake, **X25519** is automatically negotiated for ephemeral key exchange when supported by both endpoints.

All certificates are organized in subdirectories: `ca/`, `server/`, `client/`, and `ha/`.

A.1 Certificate Authority (Root CA)

The Root CA is self-signed and valid for 10 years (3650 days).

```
# Create directory structure
mkdir -p ca server client ha

# Generate CA private key (ECDSA P-256)
openssl ecparam -name prime256v1 -genkey -noout -out ca/ca.key

# Generate self-signed CA certificate
openssl req -new -x509 -sha256 -days 3650 \
  -key ca/ca.key -out ca/ca.pem \
  -subj "/CN=SmartHome␣Root␣CA/O=CTU␣FEL/C=CZ"
```

A.2 Server Certificate (EMQX Broker)

The server certificate is signed by the Root CA, valid for 5 years (1825 days). The Common Name (`CN`) must match the actual MQTT broker domain name.

```
# Generate server private key (ECDSA P-256)
openssl ecparam -name prime256v1 -genkey -noout -out server/server.key

# Generate Certificate Signing Request (CSR)
openssl req -new -key server/server.key -out server/server.csr \
  -subj "/CN=mqtt.example.com/O=SmartHome/OU=Broker"

# Sign server certificate with CA
openssl x509 -req -sha256 -days 1825 \
  -in server/server.csr \
  -CA ca/ca.pem -CAkey ca/ca.key -CAcreateserial \
  -out server/server.pem
```

A.3 Client Certificate (IoT Devices)

The client certificate for ESP32 devices is signed by the Root CA and valid for 1 year (365 days). A unique `CN` per device is recommended for device identification.

```
# Generate client private key (ECDSA P-256)
openssl ecparam -name prime256v1 -genkey -noout -out client/client.key

# Generate CSR
openssl req -new -key client/client.key -out client/client.csr \
  -subj "/CN=esp-client/O=SmartHome/OU=Sensors"

# Sign client certificate with CA
openssl x509 -req -sha256 -days 365 \
  -in client/client.csr \
  -CA ca/ca.pem -CAkey ca/ca.key -CAcreateserial \
  -out client/client.pem
```

A.4 Home Assistant Client Certificate

The Home Assistant controller requires a separate client certificate for mTLS authentication with the MQTT broker. This certificate is valid for 5 years (1825 days) to match the server certificate lifecycle.

```
# Generate HA client private key (ECDSA P-256)
openssl ecparam -name prime256v1 -genkey -noout -out ha/ha.key

# Generate CSR
openssl req -new -key ha/ha.key -out ha/ha.csr \
  -subj "/CN=homeassistant/O=SmartHome/OU=Controller"

# Sign HA client certificate with CA
openssl x509 -req -sha256 -days 1825 \
  -in ha/ha.csr \
  -CA ca/ca.pem -CAkey ca/ca.key -CAcreateserial \
  -out ha/ha.pem
```

Appendix B

Home Assistant Reference

Home Assistant is the open-source home automation platform used as the central controller in this thesis. This appendix provides essential reference information for reproducing the experimental setup.

B.1 Official Resources

- **Official Website:** `https://www.home-assistant.io/`
- **Installation Guide:** `https://www.home-assistant.io/installation/`
- **MQTT Integration:** `https://www.home-assistant.io/integrations/mqtt/`
- **MQTT Discovery:** `https://www.home-assistant.io/integrations/mqtt/#mqtt-discovery`
- **GitHub Repository:** `https://github.com/home-assistant/core`

B.2 Deployment Method

This thesis deploys **Home Assistant Operating System (HAOS) 16.3** as a virtual machine on VMware ESXi 8.0.0, using the official OVA appliance image[1]. Unlike Home Assistant Container (Docker), HAOS provides native support for Supervisor and Add-ons such as Node-RED for visual automation workflows, file editors, and integrated backup management.

The installation follows the official alternative installation guide[2], which recommends minimum resources of 2 GB RAM, 32 GB storage, and 2 vCPUs. The deployment in this thesis exceeds these requirements to accommodate additional Add-ons.

B.2.1 Host Server Specifications

The ESXi hypervisor runs on the following hardware:

- **CPU:** Intel Core i5-1135G7 @ 2.40 GHz (4 cores, 11th Gen Tiger Lake)
- **Memory:** 31.75 GB DDR4
- **Storage:** 913.75 GB VMFS6 datastore (689.67 GB used)
- **ESXi Version:** 8.0.0 (Build 20513097, standalone)
- **Install Date:** February 12, 2023

[1] `https://github.com/home-assistant/operating-system/releases/download/16.3/haos_ova-16.3.ova`
[2] `https://www.home-assistant.io/installation/alternative/`

B.2.2 HAOS Virtual Machine Configuration

The Home Assistant VM was deployed by importing the official `haos_ova-16.3.ova` appliance directly into ESXi. The VM is configured with resources exceeding the minimum requirements (2 vCPU, 2 GB RAM, 32 GB storage) to support additional Add-ons:

- **Guest OS:** Other (64-bit)
- **VM Compatibility:** ESXi 5.5 virtual machine
- **VMware Tools:** Installed (open-vm-tools)
- **vCPUs:** 4
- **Memory:** 4 GB
- **Hostname:** `homeassistant`
- **Network:** Bridged adapter (DHCP)

Figure B.1 shows the ESXi management interface with the HAOS virtual machine configuration.

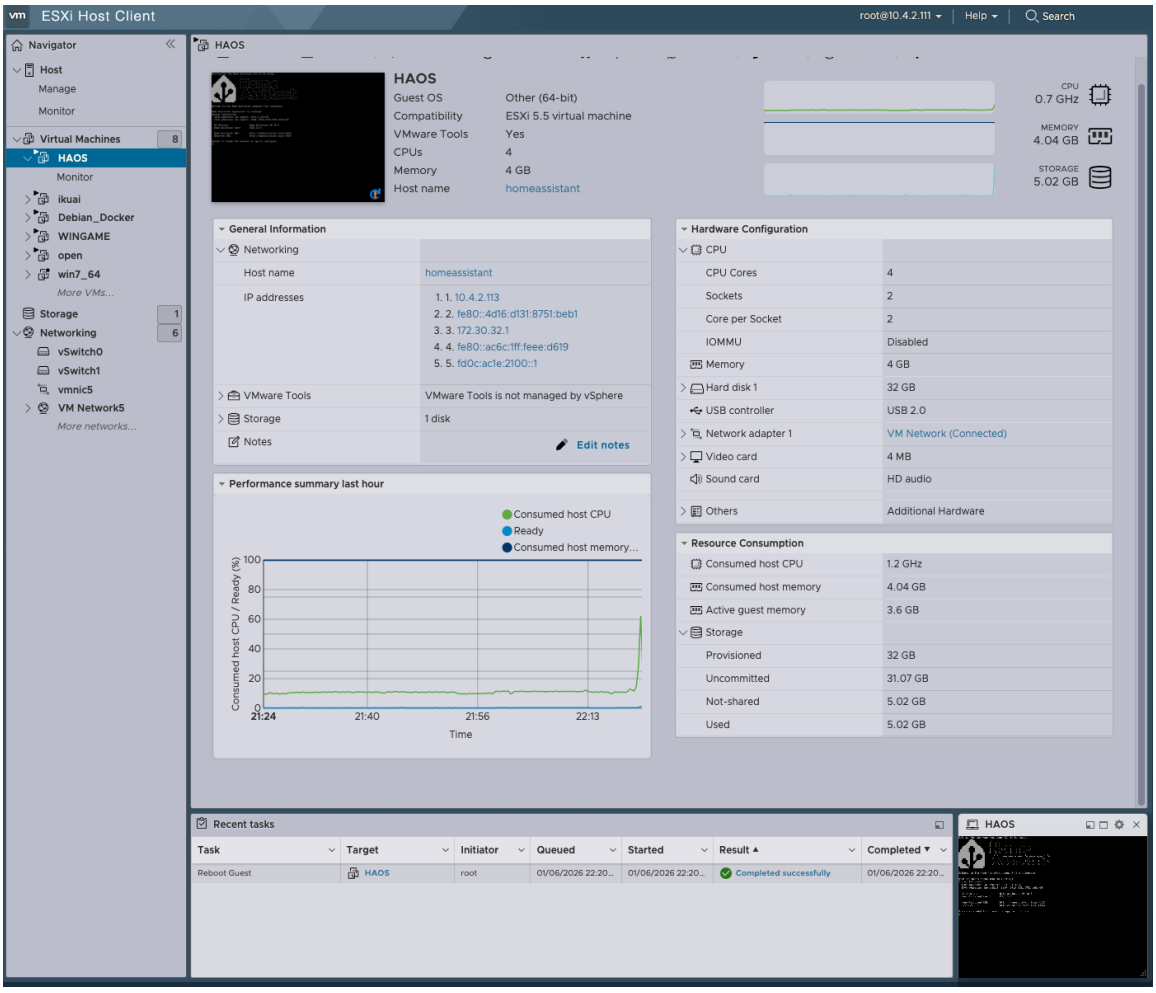


Figure B.1: HAOS virtual machine running on VMware ESXi 8.0.0

Appendix C

ESPHome Configuration Files

This appendix contains the ESPHome configuration files used for the ESP32 sensor nodes. ESPHome generates optimized C++ firmware from YAML configurations and supports direct Home Assistant integration.

C.1 Secrets Configuration Template

Sensitive credentials are stored in a separate `secrets.yaml` file (excluded from version control). The following template shows the required fields:

```yaml
# esphome/secrets.example.yaml
# Copy to secrets.yaml and fill in actual values

# Wi-Fi credentials
wifi_ssid: "YOUR_WIFI_SSID"
wifi_password: "YOUR_WIFI_PASSWORD"

# OTA update password
ota_password: "CHANGE_ME"

# Native API encryption keys (32-byte base64)
api_key_esp32s3: "CHANGE_ME_BASE64"
api_key_esp32: "CHANGE_ME_BASE64"

# MQTT broker (mTLS)
mqtt_broker: "mqtt.example.com"
mqtt_port: 8883

# TLS certificates (PEM format, use |- for multiline)
mqtt_ca_cert: |-
  -----BEGIN CERTIFICATE-----
  ... (CA certificate content)
  -----END CERTIFICATE-----
mqtt_client_cert: |-
  -----BEGIN CERTIFICATE-----
  ... (Client certificate content)
  -----END CERTIFICATE-----
mqtt_client_key: |-
  -----BEGIN PRIVATE KEY-----
  ... (Client private key content)
  -----END PRIVATE KEY-----
```

C.2 ESP32-S3 Edge Intelligence Node

The ESP32-S3 node (Node A) implements edge computing for fall detection using MPU6050 IMU data. Key features include TLS 1.3 support, 10Hz sensor sampling with 1Hz MQTT reporting, and dual-threshold fall detection algorithm.

```yaml
# esphome/esp32s3.yaml (excerpt)
esphome:
  name: smarthome-esp32s3
  friendly_name: "Edge Compute Node ESP32-S3 (Node A)"

esp32:
  board: esp32-s3-devkitc-1
  framework:
    type: esp-idf
    sdkconfig_options:
      # Certificate signatures: ECDSA P-256
      CONFIG_MBEDTLS_ECP_DP_SECP256R1_ENABLED: "y"
      CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_ECDSA_ENABLED: "y"
      CONFIG_MBEDTLS_ECDSA_DETERMINISTIC: "y"
      # Key exchange: X25519
      CONFIG_MBEDTLS_ECP_DP_CURVE25519_ENABLED: "y"
      CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_RSA_ENABLED: "y"
      # TLS 1.3 (prioritizes X25519 by default)
      CONFIG_MBEDTLS_SSL_PROTO_TLS1_3: "y"

ota:
  - platform: esphome
    password: !secret ota_password

api:
  encryption:
    key: !secret api_key_esp32s3

wifi:
  ssid: !secret wifi_ssid
  password: !secret wifi_password
  fast_connect: true

mqtt:
  broker: !secret mqtt_broker
  port: !secret mqtt_port
  certificate_authority: !secret mqtt_ca_cert
  client_certificate: !secret mqtt_client_cert
  client_certificate_key: !secret mqtt_client_key

logger:
  hardware_uart: USB_CDC

globals:
  - id: fall_detected_time
    type: unsigned long
    restore_value: no
    initial_value: '0'

i2c:
  sda: GPIO5
  scl: GPIO4
```

```
  scan: true
  id: bus_a

sensor:
  - platform: wifi_signal
    name: "WiFi RSSI"
    update_interval: 10s

  - platform: uptime
    name: "System Uptime"

  # MPU6050 IMU with calibration offsets
  - platform: mpu6050
    i2c_id: bus_a
    address: 0x68
    accel_x:
      name: "Acceleration X"
      id: acc_x
      filters:
        - offset: -0.877
        - throttle: 0.3s
    accel_y:
      name: "Acceleration Y"
      id: acc_y
      filters:
        - offset: -0.145
        - throttle: 0.3s
    accel_z:
      name: "Acceleration Z"
      id: acc_z
      filters:
        - offset: -0.564
        - throttle: 0.3s
    gyro_x:
      name: "Angular Velocity X"
      id: gyro_x
      filters:
        - offset: 2.53
        - throttle: 0.3s
    gyro_y:
      name: "Angular Velocity Y"
      id: gyro_y
      filters:
        - offset: -2.68
        - throttle: 0.3s
    gyro_z:
      name: "Angular Velocity Z"
      id: gyro_z
      filters:
        - offset: 2.44
        - throttle: 0.3s
    update_interval: 0.1s  # 10Hz internal sampling

  - platform: bmp085
    i2c_id: bus_a
    address: 0x77
    temperature: { name: "Indoor Temperature" }
    pressure: { name: "Indoor Pressure" }
```

```
    update_interval: 5s

  # Edge computing: G-force calculation
  - platform: template
    name: "Resultant G-Force"
    unit_of_measurement: "G"
    update_interval: 0.1s
    filters:
      - throttle: 0.3s
    lambda: |-
      float ax = id(acc_x).state;
      float ay = id(acc_y).state;
      float az = id(acc_z).state;
      if (isnan(ax) || (ax==0 && ay==0 && az==0)) return 0.0;
      return sqrt(ax*ax + ay*ay + az*az) / 9.80665;

  # Angular velocity magnitude for fall detection
  - platform: template
    name: "Angular Velocity Magnitude"
    unit_of_measurement: "deg/s"
    id: omega_mag
    update_interval: 0.1s
    filters:
      - throttle: 0.3s
    lambda: |-
      float wx = id(gyro_x).state;
      float wy = id(gyro_y).state;
      float wz = id(gyro_z).state;
      if (isnan(wx)) return 0.0;
      return sqrt(wx*wx + wy*wy + wz*wz);

binary_sensor:
  # Fall detection: dual-threshold algorithm (Huynh et al. 2015)
  - platform: template
    name: "Fall Detection Alert"
    device_class: safety
    id: fall_alert
    lambda: |-
      const float UFT_ACC = 2.4;    // G threshold
      const float UFT_GYRO = 240.0; // deg/s threshold
      const unsigned long HOLD_MS = 5000;
      float ax = id(acc_x).state;
      float ay = id(acc_y).state;
      float az = id(acc_z).state;
      if (isnan(ax)) return id(fall_alert).state;
      float smv = sqrt(ax*ax + ay*ay + az*az) / 9.80665;
      float omega = id(omega_mag).state;
      unsigned long now = millis();

      if ((smv > UFT_ACC) && (omega > UFT_GYRO)) {
        id(fall_detected_time) = now;
        return true;
      }
      if (id(fall_detected_time) > 0 &&
          (now - id(fall_detected_time)) < HOLD_MS) {
        return true;
      }
      return false;
```

C.2.1 ESP32-S3 Firmware Compilation Output

Figure C.1 shows the memory usage summary generated during firmware compilation for the ESP32-S3 node. The compiled firmware occupies approximately 1014 KB of flash storage, with RAM utilization at 11.78% (38.5 KB of 327 KB available). The firmware image is transferred to the device via OTA update over WiFi, completing in approximately 4 seconds.

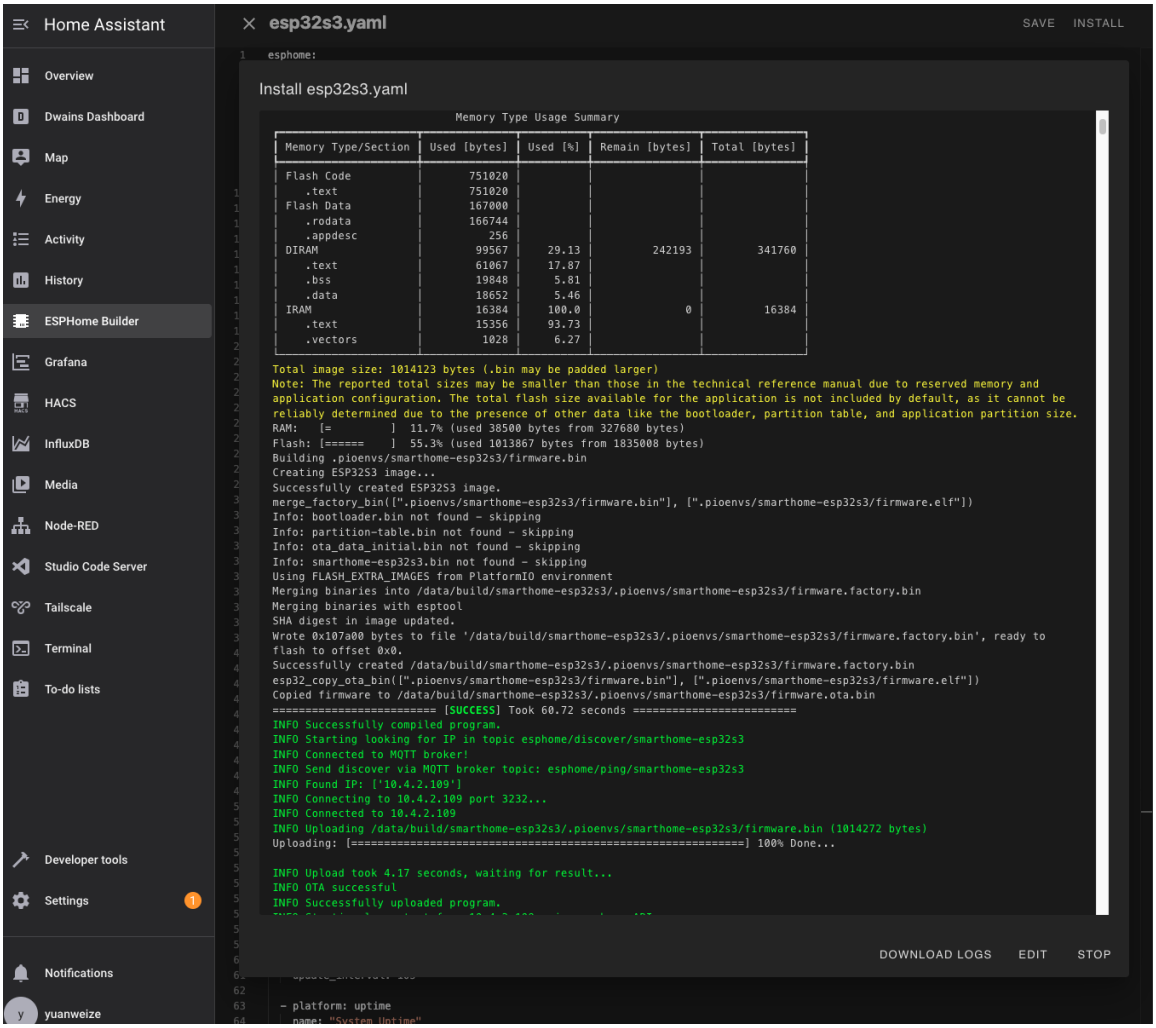


Figure C.1: ESPHome firmware compilation output for ESP32-S3 showing memory allocation summary: Flash code (751 KB), Flash data (167 KB), DRAM (99.5 KB), IRAM (16 KB), and OTA upload progress.

C.2.2 ESP32-S3 Runtime Log

Figure C.2 demonstrates the ESP32-S3 node operating in real-time, displaying sensor readings (accelerometer X/Y/Z axes, gyroscope data) and network connection details including WiFi SSID, BSSID, signal strength, and IP address assignment.

C.3 ESP32 Environment Sensing Node

The ESP32 node (Node B) focuses on environmental monitoring with TCS34725 color sensor, MQ-2 smoke detector, and SR602 motion sensor. Local automation rules trigger visual alerts via RGB LED.

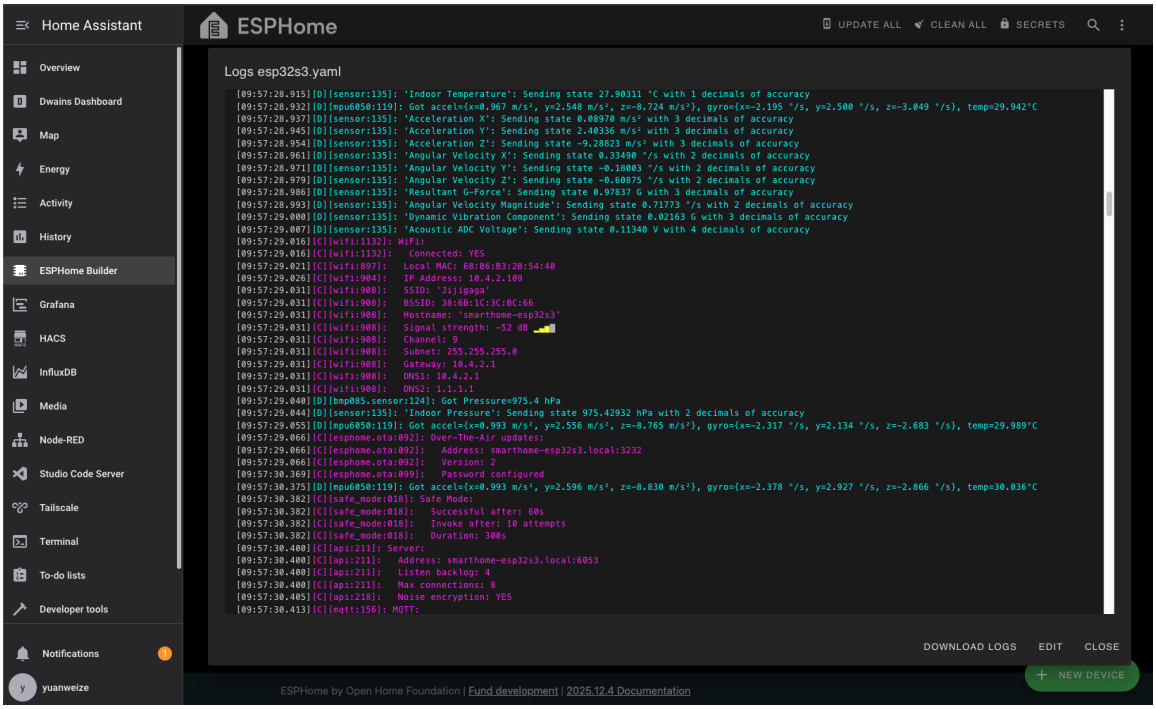


Figure C.2: ESP32-S3 runtime log showing sensor data reporting (MPU6050 accelerometer/gyroscope readings) and network connection status (WiFi SSID, MAC address, IP configuration).

```yaml
# esphome/esp32.yaml (excerpt)
esphome:
  name: smarthome-esp32
  friendly_name: "Environment Node ESP32 (Node B)"

esp32:
  board: esp32doit-devkit-v1
  framework:
    type: esp-idf
    sdkconfig_options:
      # Same TLS configuration as ESP32-S3
      CONFIG_MBEDTLS_ECP_DP_SECP256R1_ENABLED: "y"
      CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_ECDSA_ENABLED: "y"
      CONFIG_MBEDTLS_ECDSA_DETERMINISTIC: "y"
      CONFIG_MBEDTLS_ECP_DP_CURVE25519_ENABLED: "y"
      CONFIG_MBEDTLS_KEY_EXCHANGE_ECDHE_RSA_ENABLED: "y"
      CONFIG_MBEDTLS_SSL_PROTO_TLS1_3: "y"

ota:
  - platform: esphome
    password: !secret ota_password

api:
  encryption:
    key: !secret api_key_esp32

wifi:
  ssid: !secret wifi_ssid
  password: !secret wifi_password
  fast_connect: true

```

```
mqtt:
  broker: !secret mqtt_broker
  port: !secret mqtt_port
  certificate_authority: !secret mqtt_ca_cert
  client_certificate: !secret mqtt_client_cert
  client_certificate_key: !secret mqtt_client_key

i2c:
  sda: GPIO33
  scl: GPIO32
  scan: true
  id: bus_b

switch:
  - platform: gpio
    pin: GPIO25
    name: "Color␣Sensor␣LED"
    id: tcs_led

sensor:
  - platform: tcs34725
    i2c_id: bus_b
    red_channel: { name: "RGB␣Red" }
    green_channel: { name: "RGB␣Green" }
    blue_channel: { name: "RGB␣Blue" }
    illuminance: { name: "Ambient␣Lux" }
    color_temperature: { name: "Color␣Temperature" }
    gain: 4x
    integration_time: 24ms
    update_interval: 1s

  - platform: adc
    pin: GPIO34
    name: "Smoke␣Concentration"
    attenuation: 12db
    update_interval: 0.2s
    filters:
      - sliding_window_moving_average:
          window_size: 3
          send_every: 1
      - delta: 0.01
    # Local automation: smoke alert
    on_value_range:
      - above: 2.5
        then:
          - light.turn_on:
              id: status_light
              red: 100%
              green: 0%
              blue: 0%
              effect: "Pulse"
      - below: 2.4
        then:
          - light.turn_off: status_light

binary_sensor:
  - platform: gpio
    pin: GPIO23
```

```
    name: "Motion Detected"
    device_class: motion
    on_press:
      then:
        - light.turn_on:
            id: status_light
            red: 30%
            green: 50%
            blue: 70%
    on_release:
      then:
        - delay: 5s
        - light.turn_off: status_light

light:
  - platform: rgb
    name: "Status Indicator"
    id: status_light
    red: output_r
    green: output_g
    blue: output_b
    effects:
      - pulse:
          name: "Pulse"
          transition_length: 0.5s
          update_interval: 0.5s

output:
  - platform: ledc
    pin: GPIO26
    id: output_r
  - platform: ledc
    pin: GPIO27
    id: output_g
  - platform: ledc
    pin: GPIO14
    id: output_b
```

Appendix D

EMQX Docker Deployment

This appendix provides the Docker Compose configuration used to deploy the EMQX broker with mTLS enabled on port 8883.

When running high-scale experiments, the container may require an increased file descriptor limit (`ulimit -n`); a common default is 1024. The following Compose configuration raises `nofile` to support large numbers of concurrent connections.

```yaml
# broker/emqx/docker-compose.yml
services:
  emqx:
    image: emqx/emqx:latest
    container_name: emqx
    restart: always

    # Increase file descriptor limit for high concurrency
    ulimits:
      nofile:
        soft: 65535
        hard: 65535

    ports:
      # Dashboard (localhost only for security)
      - "127.0.0.1:18083:18083"
      # MQTTS (TLS)
      - "8883:8883"

    volumes:
      - ./data:/opt/emqx/data
      - ./log:/opt/emqx/log
      - ./certs:/opt/emqx/certs

    environment:
      # TLS Listener Configuration
      EMQX_LISTENERS__SSL__DEFAULT__BIND: "8883"
      EMQX_LISTENERS__SSL__DEFAULT__SSL_OPTIONS__CACERTFILE: >-
        /opt/emqx/certs/ca.pem
      EMQX_LISTENERS__SSL__DEFAULT__SSL_OPTIONS__CERTFILE: >-
        /opt/emqx/certs/server.pem
      EMQX_LISTENERS__SSL__DEFAULT__SSL_OPTIONS__KEYFILE: >-
        /opt/emqx/certs/server.key

      # mTLS Enforcement
      EMQX_LISTENERS__SSL__DEFAULT__SSL_OPTIONS__VERIFY: "verify_peer"
      EMQX_LISTENERS__SSL__DEFAULT__SSL_OPTIONS__FAIL_IF_NO_PEER_CERT: "
```

```
        true"

    # Disable anonymous access (recommended with mTLS)
    EMQX_ALLOW_ANONYMOUS: "false"
```

Appendix E

Python Simulator Architecture

This appendix presents key excerpts from the Python sensor simulator. The complete source code is available in the project repository. For flowchart reference, see Figure 4.6.

E.1 Environment Setup and Execution

The following commands demonstrate how to set up and run the simulator for reproducibility. Tested on Python 3.10+ (macOS/Linux).

```
# Clone repository and navigate to sensors directory
git clone https://github.com/yuanweize/SmartHome_Server.git
cd SmartHome_Server/sensors

# Create and activate Python virtual environment
python3 -m venv .venv
source .venv/bin/activate    # Linux/macOS
# .venv\Scripts\activate     # Windows

# Install dependencies
pip install -r requirements.txt
# Required: paho-mqtt>=1.6,<2.0  PyYAML>=6.0
# Optional: matplotlib (for handshake benchmark plots)

# Copy and configure broker settings
cp brokers.example.yml brokers.yml
# Edit brokers.yml with actual broker host, TLS paths, etc.
```

Basic Usage:

```
# Dry-run test (no MQTT connection)
python sensor_simulator.py --config brokers.yml --dry-run

# Connect with Home Assistant discovery enabled
python sensor_simulator.py --config brokers.yml --ha-discovery

# Scalability test: 1000 devices, 4 parallel workers, QoS 0
python sensor_simulator.py --config brokers.yml \
    --devices 1000 --workers 4 --qos 0
```

Commands Used in Thesis Experiments:

```
# 100-device baseline test
python sensor_simulator.py --config brokers.yml \
    --devices 100 --workers 1 --qos 0

```

```
# 1000-device scalability test
python sensor_simulator.py --config brokers.yml \
    --devices 1000 --workers 2 --qos 0

# 5000-device stress test
python sensor_simulator.py --config brokers.yml \
    --devices 5000 --workers 3 --qos 0

# MQTT connection latency benchmark (includes TCP + TLS + MQTT CONNECT)
# Performs 200 connect/disconnect cycles with 3 warmup connections
python sensor_simulator.py --config brokers.yml \
    --handshake-samples 200 --handshake-out handshake_metrics \
    --handshake-only
```

E.2 Core Data Structures

```
# smarthome_sim/entities.py
@dataclass
class EntityDef:
    """Entity definition from configuration."""
    id: str
    kind: str       # sensor | binary_sensor | switch | light
    model: str      # drift | uniform | sine | motion
    interval: float = 5.0
    min: Optional[float] = None
    max: Optional[float] = None
    # ... additional fields omitted (name, unit, precision, etc.)

# smarthome_sim/broker.py
@dataclass
class Broker:
    """MQTT broker configuration with mTLS support."""
    host: str
    port: int = 1883
    tls: bool = False
    ca_file: Optional[str] = None      # CA certificate path
    cert_file: Optional[str] = None    # Client certificate path
    key_file: Optional[str] = None     # Client private key path
    # ... additional fields omitted (inline PEM, keepalive, etc.)
```

E.3 mTLS Connection (Excerpt)

```
# smarthome_sim/broker.py
def _configure_tls(self, client: mqtt.Client) -> None:
    """Configure TLS/mTLS on the MQTT client."""
    # ... PEM materialization logic omitted ...

    client.tls_set(
        ca_certs=ca_path,
        certfile=cert_path,       # Client cert for mTLS
        keyfile=key_path,         # Client key for mTLS
        cert_reqs=ssl.CERT_REQUIRED,
        tls_version=ssl.PROTOCOL_TLS_CLIENT,
    )
```

E.4 Simulation Loop (Excerpt)

```python
# smarthome_sim/simulator.py
def run(self) -> None:
    """Main simulation loop using heap-based scheduler."""
    self._publish_discovery()    # HA MQTT Discovery

    heap = []  # Priority queue: (next_time, device_id, entity_index)
    # ... initialization omitted ...

    while not self._stop:
        next_t, device_id, idx = heapq.heappop(heap)
        time.sleep(max(0, next_t - time.monotonic()))

        self._step_entity(device_id, ...)   # Update state
        self._publish_state(device_id, ...)  # Publish to MQTT

        heapq.heappush(heap, (time.monotonic() + interval, device_id, idx))
```

E.5 Sensor Value Models (Excerpt)

```python
# smarthome_sim/simulator.py - Supported generation models
if ent.model == "uniform":
    value = random.uniform(min_v, max_v)
elif ent.model == "sine":
    value = mid + amp * math.sin(2*math.pi*time.time()/period)
elif ent.model == "drift":  # Random walk with bounds
    value = current + random.gauss(0, 0.05)
elif ent.model == "motion":  # Binary sensor with hold time
    # ... motion event logic omitted ...
```

Appendix F

Simulator Terminal Output

This appendix provides a sample terminal output from the Python sensor simulator, demonstrating the connection establishment, discovery publishing, and message flow during operation.

```
(.venv) root@HKG /opt/SmartHome_Server/sensors # python -m smarthome_sim --config brokers.yml --devices 10 -
-log-level DEBUG
17:06:27 [INFO] Loaded 1 broker(s) from brokers.yml
17:06:27 [DEBUG] [mqtt.eurun.top] Configuring TLS...
17:06:27 [DEBUG] [mqtt.eurun.top] TLS source=primary ca=True cert=True key=True
17:06:27 [INFO] Started 10 device(s), entities=6, topic=smarthome/sim
17:06:29 [INFO] [mqtt.eurun.top] Connected via TLS
17:06:57 [INFO] Publishes=425, connected=1/1
17:07:27 [INFO] Publishes=892, connected=1/1
17:07:57 [INFO] Publishes=1360, connected=1/1
17:08:27 [INFO] Publishes=1816, connected=1/1
17:08:58 [INFO] Publishes=2286, connected=1/1
17:09:28 [INFO] Publishes=2756, connected=1/1
17:09:58 [INFO] Publishes=3214, connected=1/1
17:10:28 [INFO] Publishes=3690, connected=1/1
17:10:58 [INFO] Publishes=4144, connected=1/1
17:11:28 [INFO] Publishes=4609, connected=1/1
17:11:58 [INFO] Publishes=5079, connected=1/1
17:12:28 [INFO] Publishes=5543, connected=1/1
```

Figure F.1: Python sensor simulator terminal output showing configuration loading, mTLS connection establishment to the MQTT broker, Home Assistant discovery payload publishing, and periodic state updates for simulated sensor entities.

Appendix G

Scalability Test Screenshots

This appendix documents the scalability testing process described in Section 5.2.4. Each test configuration was monitored for approximately 5 minutes to capture steady-state resource utilization across the EMQX broker, broker host system, and Home Assistant.

G.1 Test Configuration: 100 Devices

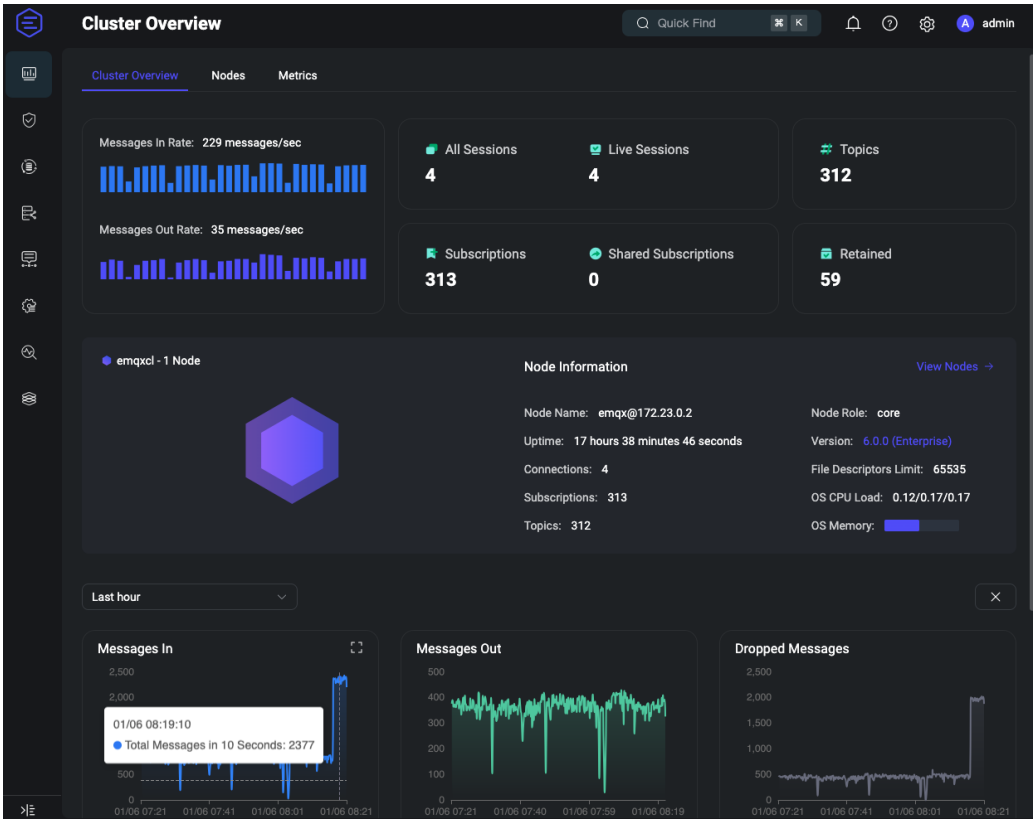


Figure G.1: EMQX Dashboard during 100-device test showing connection count and message throughput.

G.2 Test Configuration: 1,000 Devices

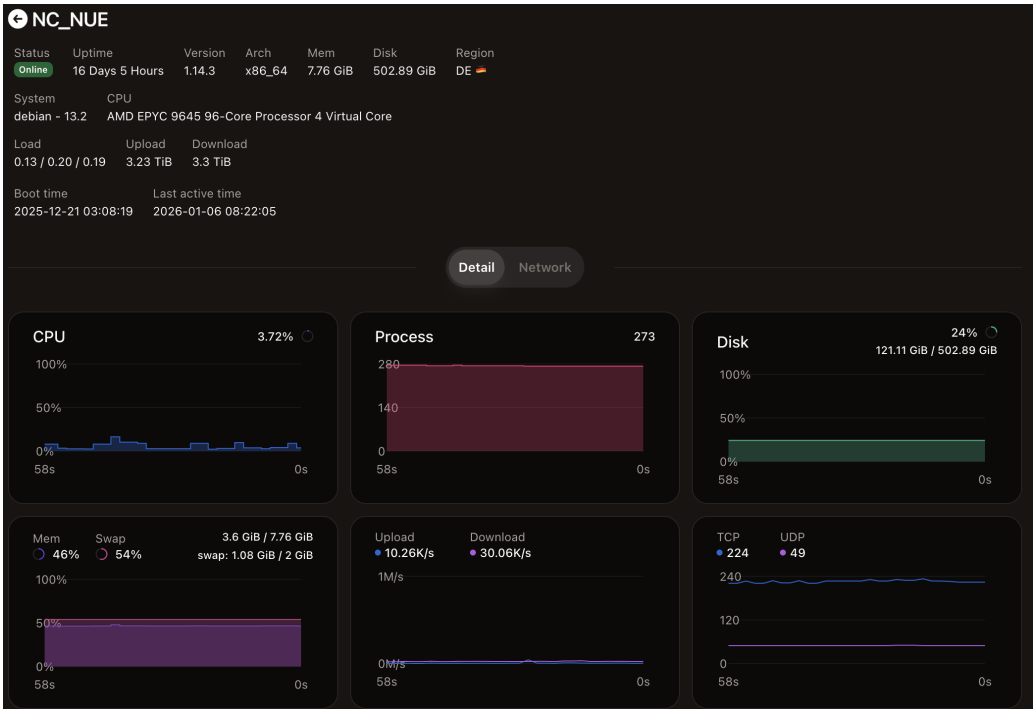


Figure G.2: Broker host system metrics during 100-device test showing CPU utilization at 3.72%.

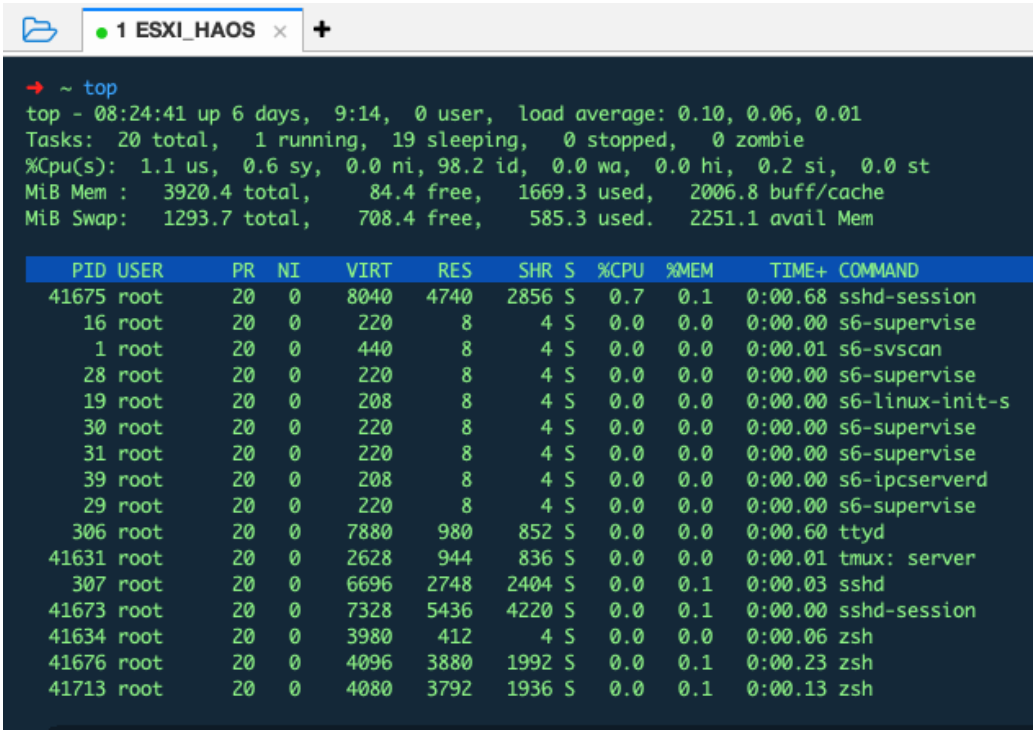


Figure G.3: Home Assistant OS `top` output during 100-device test showing CPU utilization at 1.1%.

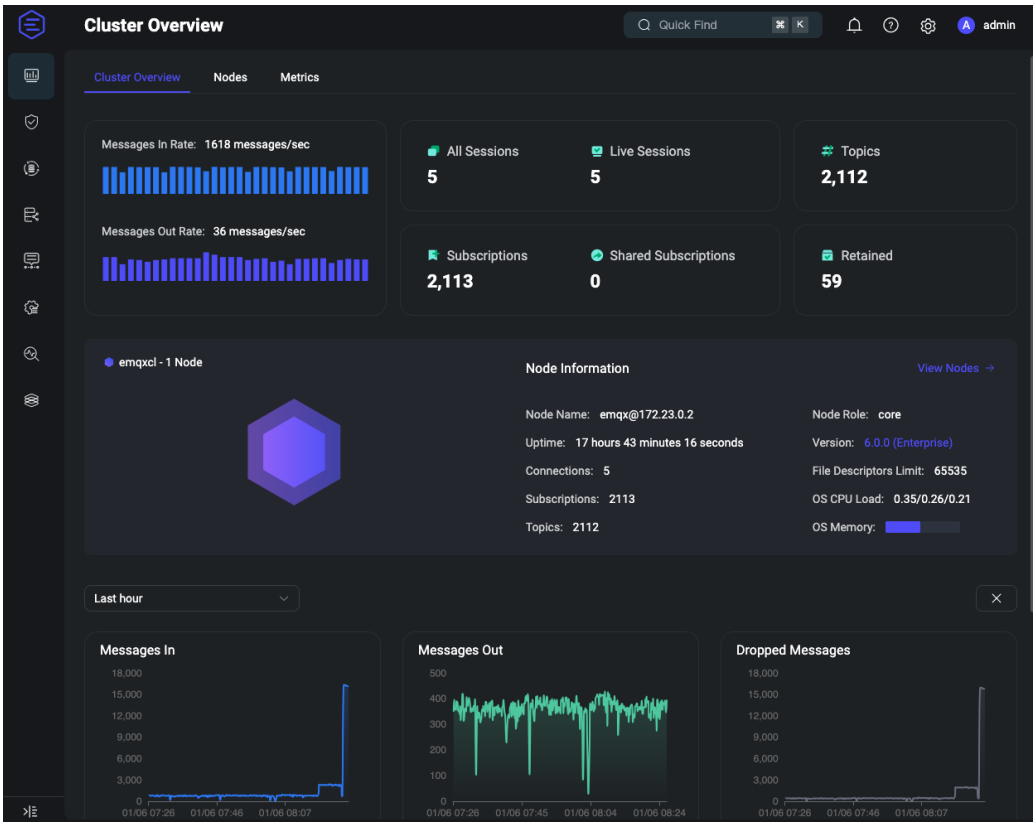


Figure G.4: EMQX Dashboard during 1,000-device test showing ∼1,618 messages/sec throughput.

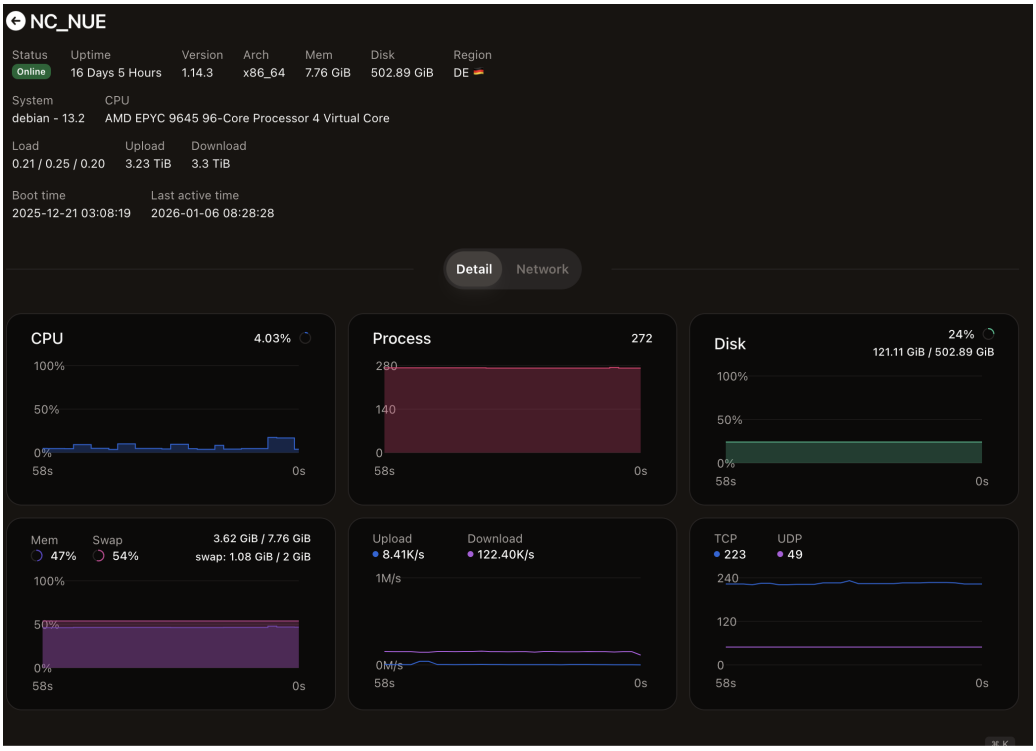


Figure G.5: Broker host system metrics during 1,000-device test showing CPU utilization at 4.03%.

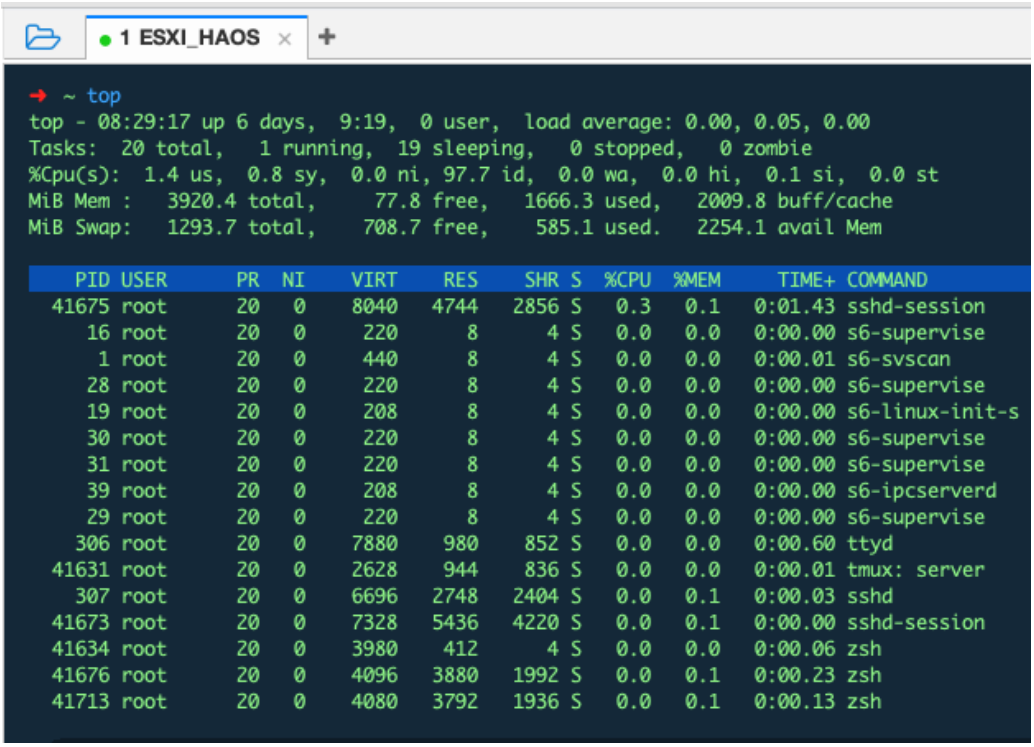


Figure G.6: Home Assistant OS `top` output during 1,000-device test showing CPU utilization at 1.4%.

Appendix H

mTLS Handshake Benchmark Data

This appendix provides the complete benchmark data for the mTLS handshake latency measurements discussed in Section 5.2.3. The benchmark was performed using the Python sensor simulator's handshake benchmark mode.

Test Configuration

- **Source**: FARM (Frankfurt, Germany) — Python 3.11.2 + paho-mqtt 1.6
- **Target**: NUE (Nuremberg, Germany) — EMQX 6.0.0 broker with mTLS
- **Network RTT**: 6.4 ms average (via `mtr`)
- **TLS Stack**: ECDSA P-256 (signature) + X25519 (key exchange)
- **Protocol**: TLS 1.3 with mutual authentication
- **Samples**: 800 connection attempts, 0 failures
- **Test Date**: 2026-01-07

Summary Statistics

```
{
  "n": 800,                    // Sample count (successful connections)
  "min": 67.84,                // Minimum latency observed (ms)
  "max": 104.07,               // Maximum latency observed (ms)
  "mean": 80.50,               // Arithmetic mean: sum(x_i) / n
  "stdev": 7.49,               // Sample standard deviation (statistics.stdev
      )
  "sem": 0.265,                // Standard error of mean: stdev / sqrt(n)
  "p50": 80.55,                // Median: 50th percentile
  "p90": 91.61,                // 90th percentile (linear interpolation)
  "p95": 92.82,                // 95th percentile
  "p99": 96.79,                // 99th percentile (tail latency)
  "ci95_mean_normal_approx": // 95% CI for true mean (z=1.96):
    [79.98, 81.02]             //   mean +/- 1.96 * sem
}
```

Percentile method: Linear interpolation between closest ranks, as implemented in `utils.percentile_sorted()`.

Command Used

```
python sensors/sensor_simulator.py --config sensors/brokers.yml \
  --handshake-samples 800 --handshake-out sensors/handshake_metrics \
  --handshake-only
```