UCLouvain

École polytechnique de Louvain

Secure, Trustworthy, and Interoperable API Gateways for Renewable Energy Assets

Authors: **Arthur De Neyer, Elie Khoury**
Supervisors: **Etienne Riviere, Annanda Rath**
Readers: **Rémi Van Boxem, Lionel Metongnon**
Academic year 2025–2026
Master [120] in Computer Science

Abstract

Renewable energy plays a crucial role in fighting against climate change by providing viable, low-carbon alternatives that are essential for an environmentally responsible future. The integration of information technology (IT) into renewable energy systems has enabled remote monitoring and management of renewable energy devices. However, it has also made these systems vulnerable to many threats and potential attacks, making the communication insecure. In addition, the communication with these different devices may also require various protocols and methods of interaction, making it challenging and complex to manage simultaneous communication with all these devices. In this work, we propose an API Gateway architecture and implementation based on Apache APISIX, an open-source API Gateway, to enable secure communication between cloud backends and renewable energy devices. The proposed architecture addresses four key security aspects: (1) Trustworthy and secure access control, (2) Security of API request/call, (3) Traffic overload protection and (4) Rate limitation. Additionally, it abstracts protocol-specific requirements through an integrated protocol translation mechanism, simplifying device interaction and management. The architecture is implemented and finally evaluated in a representative deployment environment simulating renewable energy devices. Performance testing shows that the security and translation layers add limited overhead to the communication, and that the gateway scales predictably with CPU allocation, capable of handling thousands of requests per second while maintaining low latency. The contribution is publicly available under an Apache 2.0 license at `https://github.com/GaecKo/MasterThesis`.

Acknowledgements

This work would not have been possible without the guidance of our two supervisors, Pr. Rivière and Dr. Rath. Their continuous support and feedback truly helped shape this contribution and pushed us to give our best throughout the process. From the early phases where we felt completely lost, to the gradual help, guidance, and direction we received, we were able to bring this work to completion.

We would also like to thank the external partners involved in this project, in particular the Sirris Research Centre, 3E, and the other members of the ITEA4 "Home-Pot" project. The opportunity to present our progress and receive their insightful feedback was both valuable and motivating. They often brought up additional use cases and perspectives we had not considered, which truly pushed us to think more broadly about our solution.

AI Usage

AI tools were used at different moments during this project, mainly for tasks that would otherwise be tedious or error-prone. This includes iterating on repetitive scripts (load testing infrastructure, plotting), cleaning up code, generating documentation for the codebase, and reviewing security-sensitive parts of the codebase to spot potential vulnerabilities. AI was also used to proofread and improve the writing in this document, though all technical content, analysis, and conclusions are authors' own.

Contents

1 Introduction **2**
1.1 Thesis Structure 4

2 Problem statement **5**
2.1 Background 5
2.1.1 Transport Layer Security (TLS and DTLS) 5
2.1.2 IoT Application Layer Protocols 6
2.1.3 API Gateways 7
2.1.4 Edge and Fog Computing 8
2.2 Context 9
2.2.1 Communication aspect 11
2.2.2 Security aspect 12
2.3 Problem 14
2.3.1 Problems from a communication point of view 14
2.3.2 Problem from a security point of view 14
2.4 Objective and Motivation 15
2.4.1 Trustworthy and secure access control 18
2.4.2 Security of API request/call 19
2.4.3 Traffic Overload Protection 20
2.4.4 Rate Limitation 20
2.4.5 Protocol Translation 21

3 State of the Art **23**
3.1 Protocol Interoperability and Translation 23
3.2 Multi-Protocol IoT Platform Architectures 25
3.3 The Web of Things Paradigm as an Interoperability Strategy 25
3.4 API Gateway Security in Edge and Microservice Architectures 26
3.5 Discussion and Research Gap 27

4 Evaluation and Selection of an Open-Source API Gateway **29**
4.1 Evaluation Methodology 29

4.2 Comparative Analysis . . . 31
4.2.1 Trustworthy and secure access control . . . 31
4.2.2 Key Storage . . . 33
4.2.3 Security of API request/call . . . 33
4.2.4 Traffic management . . . 34
4.2.5 Protocol support . . . 36
4.2.6 Deployment capabilities . . . 36
4.2.7 Configurability and runtime management . . . 37
4.2.8 Sustainability and community support . . . 38
4.3 Discussion and Final Selection . . . 39
4.4 APISIX . . . 40
4.4.1 Java Plugin Runner . . . 41
4.4.2 Filter-based routes . . . 43
4.4.3 Usage in this Contribution . . . 44

5 API Gateway Architecture and Implementation 45
5.1 General Architecture . . . 45
5.2 General Components and Principles . . . 47
5.2.1 Request Handler . . . 48
5.2.2 Logger . . . 48
5.3 Trustworthy and Secure Access Control . . . 48
5.3.1 Built-in Plugins and Their Usage . . . 49
5.3.2 Filters Architecture and Implementation Components . . . 50
5.3.3 Onboarding Filter . . . 52
5.3.4 Auth Filter . . . 53
5.4 Outbound Authentication . . . 55
5.5 Protocol Translation and Device Communication . . . 56
5.5.1 Translation Configuration Data Model . . . 57
5.5.2 Device Adapters . . . 59
5.5.3 Translation and Cleanup Engine . . . 61
5.5.4 Queuing Mechanism . . . 63
5.5.5 Translation Onboarding Filter . . . 65
5.5.6 Device Translation Filter . . . 65
5.6 Device-to-Backend Communication . . . 66
5.7 Security of API request/call . . . 68
5.7.1 General Approach . . . 68
5.7.2 Certificate Infrastructure . . . 68
5.7.3 Scope of TLS Coverage . . . 69
5.8 Traffic overload protection . . . 70
5.8.1 Network-Level Protection: `nftables` Firewall . . . 70
5.8.2 Application-Level Protection: Payload Enforcement . . . 71

5.9 Rate limitation . . . 72
5.9.1 `limit-count` Built-in Plugin . . . 72
5.9.2 `client-control` Built-in Plugin . . . 73

6 Deployment, Demonstration, and Evaluation **75**
6.1 Deployment Setup . . . 75
6.2 Integration / Utilization testing . . . 76
6.3 Security Testing . . . 78
6.3.1 Rate limitation (`limit-count`) . . . 78
6.3.2 IP Blacklisting firewall (`nftables`) . . . 79
6.3.3 Request body size limitation (`client-control`) . . . 79
6.4 Performance Testing . . . 80
6.4.1 System parameters . . . 81
6.4.2 Factors . . . 82
6.4.3 Metrics . . . 82
6.4.4 Context 1: Latency characterization per architecture . . . 83
6.4.5 Context 2: Overload testing . . . 86
6.4.6 Conclusion . . . 88

7 Conclusion and Perspectives **90**
7.1 Conclusion . . . 90
7.2 Future Work . . . 91

A API Reference **99**
A.1 Communication Configuration Routes . . . 99
A.2 Authorization Configuration Routes . . . 102
A.3 The `/commands` Route . . . 105
A.4 The `/onboarding/translation` Route . . . 106
A.5 The `/command` Route . . . 109
A.6 The `/backendForward` Route . . . 110

Chapter 1

Introduction

The increasing integration of information technology into industrial and infrastructure systems has produced highly heterogeneous environments, where devices from different manufacturers, generations, and domains must coexist and communicate. These systems rely on diverse protocols, proprietary interfaces, and incompatible data models, making unified management and secure communication a persistent challenge. As the number of connected devices grows, the need for scalable and interoperable solutions to bridge these heterogeneous ecosystems becomes increasingly critical.

Renewable energy systems constitute a representative and timely example of this challenge. The transition toward renewable energy has become a central objective of the European Union's energy strategy. In 2023, renewable sources accounted for 44.7% of the EU's total electricity production, making them the leading source of electricity generation (Figure 1.1). This shift reflects both environmental commitments and the increasing maturity of renewable energy technologies.

This domain alone encompasses a wide variety of systems, each introducing its own technical constraints and communication requirements. Renewable energy production encompasses a wide range of domains, including wind, solar, hydroelectric, and geothermal systems [2]. Each domain relies on distinct technologies and operational principles, resulting in a diverse set of energy assets deployed across different environments and scales.

The current expansion of renewable energy is the result of decades of research, development, and industrial deployment. Technologies such as wind turbines and photovoltaic systems have progressively evolved, with large-scale installations becoming increasingly visible over the past decades. As with many long-term technological developments, this evolution has produced a heterogeneous ecosystem composed of both legacy systems and more modern, optimized solutions.

These energy systems are built upon numerous electrical and mechanical components, such as generators, inverters, motors, and control units. To ensure safe, efficient,

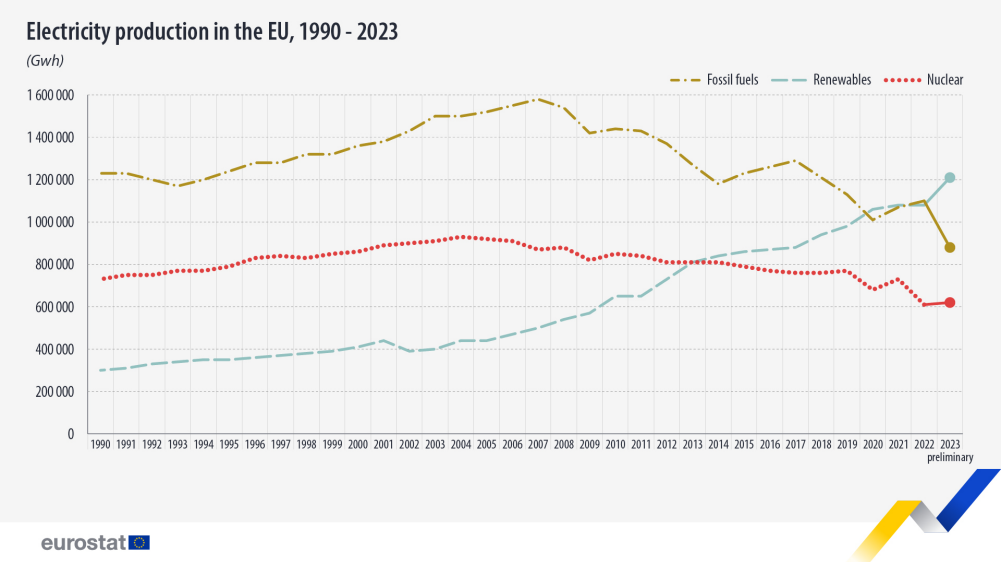

Figure 1.1: Electricity production in the EU between 1990 and 2023 [1].

and reliable operation, these components must be monitored, configured, and controlled throughout their lifecycle. As renewable installations scale up in number and capacity, effective system management becomes a critical concern.

In parallel with the growth of renewable energy production, the sector has undergone a significant digital transformation. Advances in communication technologies and the widespread adoption of Cloud Computing have enabled renewable energy assets to become increasingly connected. Today, devices such as inverters, solar panels, and wind turbines can exchange large amounts of data with cloud platforms, allowing remote monitoring, configuration, diagnostics, and performance optimization. These capabilities are often accessed through client applications designed to simplify asset management for operators and service providers.

However, this digitalization has also introduced new challenges. Renewable energy assets are frequently delivered by different manufacturers, rely on proprietary protocols, and expose heterogeneous interfaces. As a result, integrating these systems into unified cloud-based environments remains complex. The lack of standardization complicates interoperability, limits scalability, and increases the effort required to manage diverse fleets of renewable assets.

In this context, the growing heterogeneity of renewable digital assets and cloud infrastructures calls for solutions that can facilitate connectivity, integration, and interoperability between them.

In this master's thesis, we investigate how an **API Gateway** would contribute to such a system and address the challenges of secure and interoperable connectivity between heterogeneous renewable digital assets and cloud infrastructures.

1.1 Thesis Structure

The remainder of this thesis is structured as follows:

- **Chapter 2** further develops the context, problem statement, and motivation of this work, and introduces the key objectives and architectural aspects covered by the proposed API Gateway.
- **Chapter 3** reviews existing approaches and systems that aim to address similar integration and security challenges. This analysis serves as a foundation for identifying relevant design principles and positioning our contribution within the current state of the art.
- **Chapter 4** presents the selection of the base tool and practices used for the implementation, together with the rationale behind these choices.
- **Chapter 5** details the architecture and implementation of the proposed system, describing its components, design decisions, and the mechanisms developed to address communication and security requirements.
- **Chapter 6** demonstrates and evaluates the system through practical scenarios and performance analysis. Deployment considerations and possible integration strategies are also discussed.
- Finally, **Chapter 7** concludes this thesis by summarizing the contributions and outlining potential directions for future work.

The contribution is publicly available under an Apache 2.0 license at `https://github.com/GaecKo/MasterThesis` [3].

Chapter 2

Problem statement

This chapter establishes the context in which the contribution of this work is situated. It begins by providing additional background information, followed by a more precise and technical overview of the domain. The chapter then formally introduces the communication and security aspects that arise within architectures incorporating an API Gateway, along with the associated challenges. Finally, it concludes by defining the objectives and motivations addressed in this thesis.

2.1 Background

This section introduces the key concepts and technologies needed to understand the remainder of this work. First, the Transport Layer Security protocol is presented, as it provides the security foundation for all communication protocols discussed in this work. The application-layer protocols used by renewable energy devices to communicate with cloud backends are then defined. Next, an API gateway and its role in distributed architectures are explained. Finally, the edge and fog computing paradigms that contextualize the gateway's position in the system are covered.

2.1.1 Transport Layer Security (TLS and DTLS)

Transport Layer Security (TLS) is a cryptographic protocol designed to provide confidentiality, integrity, and authentication over a communication channel. It operates between the transport and application layers, wrapping the exchanged data in an encrypted envelope that prevents third-party eavesdropping and tampering. TLS achieves this through a handshake mechanism during which the communicating parties negotiate cipher suites, authenticate each other using certificates, and establish a shared session key used to encrypt subsequent exchanges. In the context of IoT application protocols, TLS is used to secure TCP-based communications, resulting in

the secure variants HTTPS and MQTTS.

Since CoAP runs over UDP rather than TCP, standard TLS cannot be applied directly. DTLS (Datagram Transport Layer Security) was introduced by the IETF as the UDP-compatible counterpart of TLS. It preserves the same security guarantees while adapting the handshake and record layer to tolerate the packet loss and reordering inherent to UDP. DTLS is therefore the standard security mechanism for CoAP, producing the secure variant CoAPS.

2.1.2 IoT Application Layer Protocols

Renewable energy devices communicate with cloud backends using application-layer protocols designed for the specific constraints of embedded and networked systems. Several such protocols coexist in practice, each reflecting different trade-offs between resource consumption, reliability, and communication patterns [4, 5].

HTTP (HyperText Transfer Protocol) is the fundamental client-server protocol of the Web and the one most compatible with existing network infrastructure. Communication between a client and a server occurs via request/response messaging: the client sends an HTTP request and the server returns a response containing the requested resource. HTTP is commonly used together with REST (Representational State Transfer), an architectural style that maps the create, read, update, and delete operations to the HTTP methods POST, GET, PUT, and DELETE, respectively. Data is typically exchanged in JSON or XML format [5]. HTTP uses TCP as its transport protocol, which guarantees reliable delivery but introduces overhead that can be problematic for resource-constrained devices sending small amounts of data occasionally. For security, HTTP relies on TLS, which enables a secure, encrypted communication channel, resulting in the secure version known as HTTPS [5]. (For an example, see Figure 2.1)

Figure 2.1: REST HTTP request/response interaction model (Figure from Dizdarević et al [5]).

MQTT (Message Queuing Telemetry Transport) is a lightweight publish/subscribe protocol released by IBM and designed for constrained devices operating over low-bandwidth or unreliable networks [4]. In MQTT, a central broker maintains named topics. Clients act as publishers by sending messages to the broker on a specific

topic, and as subscribers by registering interest in a topic and receiving automatic updates whenever a new message is published [4]. MQTT runs over TCP, which ensures reliable delivery. It defines three quality-of-service levels: fire-and-forget (no acknowledgement required), delivered at least once (acknowledgement required), and delivered exactly once (four-step handshake). Security is handled through TLS, and brokers can require username and password authentication via the CONNECT message [5]. Its lightweight header and publish/subscribe architecture make MQTT well suited for high-frequency telemetry from energy assets where devices do not need to poll for updates. (For an example, see Figure 2.2)

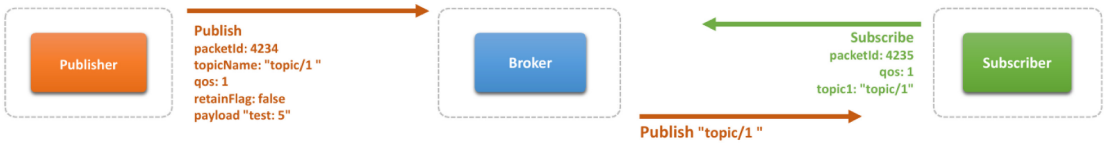

Figure 2.2: MQTT publish/subscribe interaction model (Figure from Dizdarević et al [5]).

CoAP (Constrained Application Protocol) was designed by the IETF for use on constrained devices with limited processing capabilities. Like HTTP, it uses a REST architecture and supports the GET, PUT, POST, and DELETE methods, but it runs over UDP instead of TCP to reduce overhead and remove the three-way handshake cost [4, 5]. To compensate for the unreliability of UDP, CoAP defines four message types: Confirmable messages that require an acknowledgement, Non-Confirmable messages that do not, Acknowledgement messages, and Reset messages. A simple stop-and-wait retransmission mechanism handles lost Confirmable messages [4]. CoAP also supports an observe option that allows a client to register interest in a resource and receive automatic notifications when its state changes, bringing the protocol closer to a publish/subscribe paradigm without abandoning its RESTful foundation [5]. Security is provided by DTLS, the UDP equivalent of TLS [4].

2.1.3 API Gateways

An API gateway is a component that acts as a single entry point for all requests coming from external clients to a set of services. Instead of allowing clients to communicate directly with each service, all traffic is routed through the gateway, which centralizes responsibilities that would otherwise be implemented independently by each service. These include request routing, authentication, authorization, rate limiting, payload validation, protocol translation, and observability through logging and monitoring.

The key benefit of this design is the centralization of cross-cutting concerns. Without a gateway, every service would need to implement its own authentication logic, access control checks, and input validation, leading to duplicated effort and inconsistent enforcement. The gateway resolves these concerns by handling them in a single, controlled layer before any request reaches the services. Services can therefore remain focused on their core logic and evolve independently of how clients interact with the system.

Figure 2.3 illustrates the general architecture of an API gateway. All client requests arrive at the gateway through a single interface. The gateway processes each request through its internal pipeline and routes it to the appropriate service. Responses travel back through the gateway to the client.

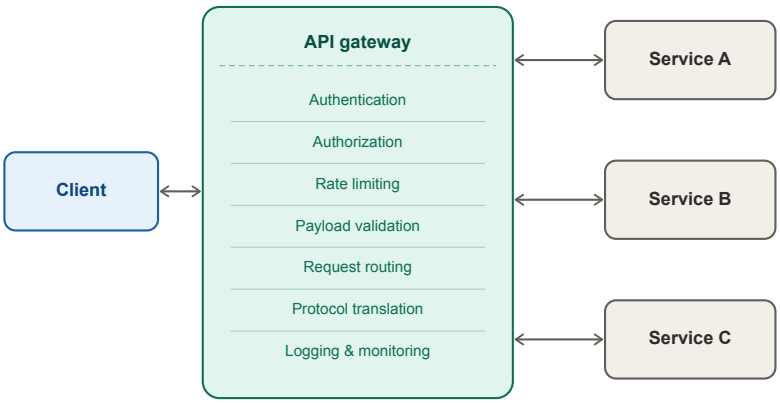

Figure 2.3: General architecture of an API gateway.

2.1.4 Edge and Fog Computing

Cloud computing centralizes storage, computation, and service logic in remote data centers. While this model offers scalability and ease of management, it is not efficient for applications that require processing large volumes of data in a very short time. The latency introduced by transmitting data to a distant cloud server negatively affects the quality of service of time-sensitive applications [6].

To address these limitations, edge computing was introduced as an architecture that decentralizes data processing from the cloud to the network edge. Rather than sending all data to a centralized server, computation is performed closer to the devices that generate the data, reducing latency and alleviating network congestion [6].

Fog computing is a closely related paradigm, introduced by Cisco, that acts as an intermediate layer between cloud infrastructure and end devices. It extends the cloud model by placing computation, storage, and connectivity capabilities at nodes located

between the cloud and the devices, rather than at the device boundary itself. Unlike cloud-only approaches, fog computing relies on a distributed set of heterogeneous systems that collectively provide flexible computation, storage, and connectivity without depending exclusively on centralized infrastructure [7]. This intermediate placement allows fog nodes to handle data locally, reducing the volume of traffic sent to the cloud and enabling faster responses for latency-sensitive operations.

While edge and fog computing are sometimes used interchangeably, fog computing typically refers to this structured intermediate layer managed in coordination with the cloud infrastructure, whereas edge computing emphasizes processing at or very close to the device itself [6, 7].

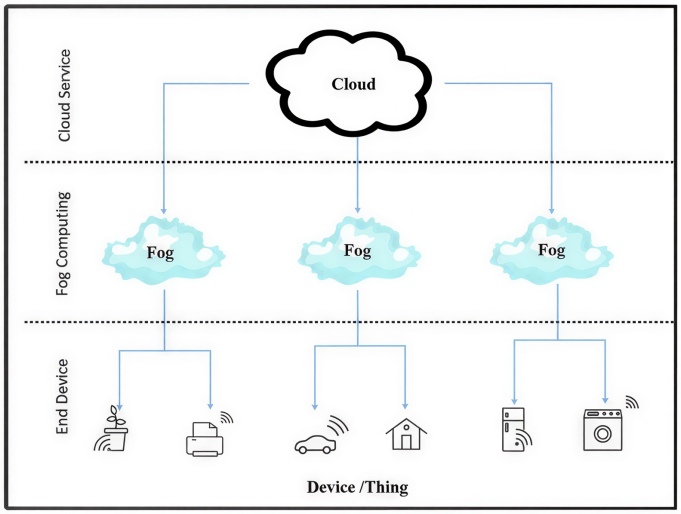

Figure 2.4: Cloud, fog/edge, and IoT device three-layer architecture (Figure from Abdali et al [7]).

2.2 Context

Renewable energy devices can be considered as *renewable digital assets* that communicate with a cloud-based backend infrastructure. This communication serves multiple purposes, including monitoring, configuration, diagnostics, etc. The general simplified interaction model between devices and the backend is illustrated by Figure 2.5.

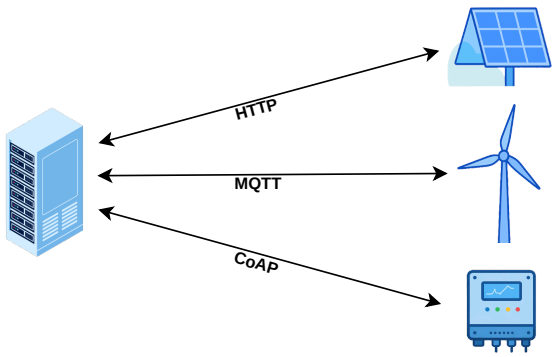

Figure 2.5: Backend ↔ devices architecture

Backend (West). On the **West** part, lies the **cloud backend**. In this work, the backend is assumed to operate in a controlled environment, typically within secured cloud infrastructure. It is responsible for device fleet management, configuration, monitoring, optimization, data storage, analytics, and integration with external services or client applications.

Although deployed in a relatively protected environment, the backend remains exposed to external communications originating from potentially untrusted networks. Its role is therefore both functional and critical: it must reliably communicate with a large number of devices, often highly heterogeneous in nature.

Devices (East). On the **East** part are the **renewable digital assets**, which correspond to field devices involved in energy production and monitoring. These include, for example, inverters, photovoltaic systems, wind turbines, and smart meters.

Such devices are inherently heterogeneous. They may differ in hardware architecture (e.g., embedded boards vs. industrial controllers), firmware versions, operating systems, communication interfaces, and supported protocols. For instance, some devices expose simple register-based data, while others provide structured information using formats such as JSON or XML.

These assets are typically geographically distributed and often deployed in remote or industrial environments, such as solar farms or offshore wind parks. They are commonly connected through public or semi-public networks, which places them in a less trusted and potentially hostile environment from a security perspective.

2.2.1 Communication aspect

Communication between the West and East parts is diverse and heterogeneous. Commonly used protocols include HTTP(S), MQTT(S), CoAP, and other IoT-oriented communication standards [8, 9]. While these protocols define transport and messaging mechanisms, they do not standardize how application data is structured.

In practice, message payloads (for example, HTTP request bodies or MQTT messages) often rely on flexible formats such as JSON or binary encodings. These structures typically use key-value representations, allowing a virtually unlimited number of ways to encode device information. As a result, two devices using the same transport protocol may still expose completely different data models and command structures.

Furthermore, communication patterns vary significantly:

Request-Response model Some devices operate in a request-response model, where a client explicitly queries the device to retrieve data or trigger an action, as introduced in Section 2.1.2 (HTTP). This pattern typically implies synchronous interactions and direct control over when data is exchanged.

Publish-Subscribe model Others rely on publish-subscribe mechanisms, as described in Section 2.1.2 (MQTT), where devices asynchronously send data to a broker or intermediary without being explicitly requested. This approach enables more scalable and decoupled communication, especially in distributed environments.

Connection type Some devices maintain persistent connections, while others communicate intermittently. In practice, this means certain devices remain continuously connected to the network, enabling real-time interactions, whereas others only establish communication periodically, for example to reduce power consumption or bandwidth usage.

Intermittent availability Some devices may not be continuously reachable due to network conditions, power constraints, or operational patterns. As a result, communication must be resilient, supporting mechanisms such as retries or delayed exchanges to ensure data can still be transmitted once the device becomes available again.

Authentication flow Firmware update mechanisms and authentication flows also differ across vendors. This reflects the diversity in how devices are managed and secured, with varying approaches to software updates, identity verification, and access control.

In most cases, these communication specifications are proprietary and documented by the device manufacturer. Consequently, the backend must explicitly un-

derstand and implement each device-specific protocol, data model, and behavioral pattern. This creates a growing integration burden as the number and diversity of devices increase.

Beyond interoperability, additional aspects must also be considered:

Scalability The scalability, as the number of connected assets grows. The system must be able to handle increasing volumes of devices and data without significant performance degradation, which requires efficient resource management and architecture design.

Devices evolution Versioning and backward compatibility when devices evolve. As devices and their interfaces change over time, it is important to ensure that newer and older versions can still interoperate without breaking existing integrations.

Visibility Observability and logging across heterogeneous communication channels. Given the diversity of protocols and devices, maintaining visibility over system behavior is essential for monitoring, debugging, and ensuring reliable operations.

Lifecycle management Device lifecycle management, including onboarding, decommissioning, and updating. Devices must be properly integrated into the system, maintained throughout their operational life, and securely removed or replaced when no longer in use.

2.2.2 Security aspect

From a security perspective, the architecture introduces asymmetrical trust zones.

Devices are deployed in environments that may be physically accessible or connected via public networks. Communications between devices and the backend may traverse untrusted infrastructure, exposing them to several risks [10, 11, 12]:

Eavesdropping and traffic interception Communications may be passively monitored by unauthorized parties when transmitted over untrusted networks. This can lead to the disclosure of sensitive information, such as operational data or credentials, especially if encryption is absent or improperly configured.

Man-in-the-Middle (MITM) attacks An attacker may intercept and alter communications between a device and the backend without either party being aware. This enables the manipulation of exchanged data or the injection of malicious commands.

Replay attacks Previously captured valid messages may be resent by an attacker to reproduce legitimate actions. Without proper safeguards, this can result in unintended or duplicated operations within the system.

Message tampering Data exchanged between devices and backend systems may be modified in transit. Such alterations can compromise the integrity of the system by introducing incorrect or misleading information.

Impersonation or spoofing of devices An attacker may attempt to masquerade as a legitimate device to gain unauthorized access to the system. This can allow malicious entities to send false data or receive sensitive information.

Even when encrypted protocols (e.g., HTTPS, MQTTS) are used, improper certificate validation, weak authentication mechanisms, or firmware vulnerabilities may still expose the system to compromise.

A compromised device represents a particularly critical threat. Since devices communicate directly with the backend, an attacker controlling such a device could have significant impact.

Injection of malicious data A compromised device may send malformed or intentionally crafted data to disrupt processing or mislead the backend system.

Exploitation of backend vulnerabilities By interacting directly with backend interfaces, an attacker may attempt to trigger software vulnerabilities, potentially leading to broader system compromise.

Unauthorized command triggering Malicious control of a device may allow the attacker to issue illegitimate commands, affecting system behavior or other connected components.

Denial-of-Service (DoS) conditions A compromised device may generate excessive traffic or repeated requests, potentially overwhelming backend services and degrading system availability.

Although the backend operates in a controlled cloud environment, it cannot be considered isolated. It remains logically exposed to every connected device. The direct communication channel between the East and West, therefore, constitutes a major attack surface.

Ensuring secure authentication, authorization, input validation, rate limiting, and isolation mechanisms becomes essential in such an architecture.

2.3 Problem

This thesis addresses some of the architectural, communication, and security challenges arising from the direct interaction between a heterogeneous fleet of renewable digital assets and a cloud backend, as well as the lack of a unified and efficient management approach for these heterogeneous systems.

2.3.1 Problems from a communication point of view

In the architecture presented in Figure 2.5, the backend carries a significant integration burden. Because it must communicate directly with many different devices, it needs to precisely implement and maintain knowledge of each device's protocol, message structure, authentication method, and behavioral model, requiring the backend to handle multiple communication mechanisms as well as heterogeneous data interfaces.

As the number of supported devices increases, the backend accumulates a growing set of communication adapters and device-specific logic. This leads to increased complexity, reduced maintainability, higher development and testing effort, and strong coupling between business logic and device integration logic. As a result, either the backend must be exhaustively configured for every device type, or client applications must be aware of device-specific communication details. Both approaches are suboptimal and contradict the principle of abstraction and separation of concerns.

This current architecture shifts the backend away from its primary role (data processing, orchestration, and service logic) and overloads it with low-level integration responsibilities.

2.3.2 Problem from a security point of view

From a security standpoint, the direct exposure of the backend to all field devices increases the overall attack surface. The backend effectively sits at the boundary between trusted cloud infrastructure and potentially compromised devices.

Renewable digital assets are prone to malicious activities (2.2.2). Yet, in the current architecture, a malicious or compromised device could communicate directly with core backend services. Without proper isolation and mediation mechanisms, this creates a high-impact attack vector. The backend must therefore simultaneously:

- Handle heterogeneous communication,
- Enforce strict authentication and authorization,
- Validate and sanitize incoming data,

- Protect against volumetric attacks and abuse.

Combining these responsibilities within the same architectural layer increases the risk of implementation errors and weakens the system's overall robustness.

2.4 Objective and Motivation

To address the communication and security challenges identified in the previous section (Section 2.3), this thesis proposes the design and implementation of a **Secure, Trustworthy, and Interoperable API Gateway**. This component constitutes the central contribution of this work.

The development of this gateway is conducted in the context of the **ITEA4 "HomePot"** project [13], part of the **EUREKA** RD&I cluster [14], in collaboration with the Sirris Research Center and 3E.

ITEA4 "HomePot" Project The ITEA4 "HomePot" project aims at developing a single, secure platform that allows the management of a wide range of devices [13].

RD&I cluster The RD&I is a cluster focused on driving innovation in software and systems. It aims at supporting a domain-agnostic digital transition through funded applied R&D projects aiming for impact [14].

Sirris Sirris is a non-profit scientific organisation that supports companies in innovation and digital transformation. It operates as an organization providing expertise, specialized labs, and tailored advice in areas like advanced manufacturing, AI, and sustainability [15].

3E 3E, on its side, is a consultancy and software firm focused on performance optimization for renewable energy projects solutions [16].

Within this framework, the API Gateway is introduced as an intermediate architectural layer positioned between cloud services and renewable digital assets. Revisiting the architecture presented in Figure 2.5, the introduction of the API Gateway modifies the system by inserting a mediation layer between the cloud backend and the devices. The resulting architecture is illustrated in Figure 2.6.

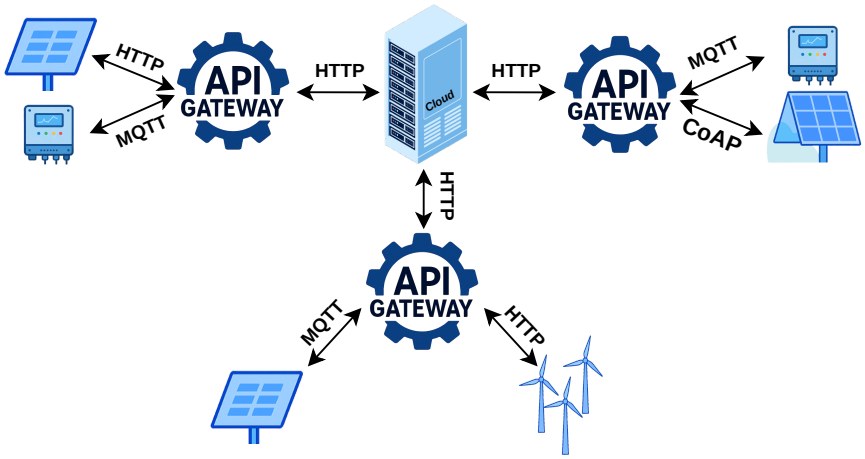

Figure 2.6: Cloud Backend ↔ Distributed API Gateways ↔ Devices: architecture

In this revised architecture, the API Gateway is positioned at the edge of the cloud environment. As such, it acts as a controlled boundary between a potentially untrusted environment and the protected backend infrastructure.

From a broader perspective, the system can be extended to scenarios where a backend cloud service is connected to multiple API Gateways, reflecting a geographically distributed architecture. Conversely, a single API Gateway may interface with multiple backend services, enabling different cloud instances to interact with diverse and heterogeneous devices.

Architectural Motivation

API Gateways are widely used in modern architectures, where they serve multiple purposes. In the context of this work, the main advantages of such an architectural component are outlined below.

First, it enables a clear separation of concerns. Device-specific communication logic, protocol handling, and cross-cutting functionalities are delegated to a dedicated component. This aligns with microservice principles, where the gateway acts as a centralized entry point that abstracts internal service complexity and reduces coupling between clients and backend services [17, 18].

Second, the gateway provides a central point for enforcing security and operational policies. Instead of distributing authentication, authorization, traffic control, and monitoring across multiple services, these responsibilities are consolidated within a single layer. This approach improves consistency, simplifies governance, and enhances protection against external threats [18].

Third, the gateway facilitates interoperability. By acting as a mediation layer, it normalizes communication between heterogeneous devices and backend services,

handling protocol translation and data transformation when needed. This allows the system to expose a unified interface while hiding internal diversity and complexity [17, 18].

These architectural motivations can also be interpreted within the broader paradigm of *Fog Computing*. Fog Computing extends cloud capabilities by introducing an intermediate layer closer to edge devices, aiming to reduce latency, improve scalability, and enhance reliability [19].

In this perspective, the proposed API Gateway can be viewed as a fog-layer component. Positioned between the cloud backend and renewable digital assets, it performs mediation, security enforcement, and protocol handling near the edge. While the system developed in this thesis does not implement a complete fog computing framework, it reflects several of its core principles by decentralizing processing and introducing computation at the boundary between cloud and devices.

Security-Oriented Objectives

Among multiple requirements defined by the Sirris Research Center and 3E, this thesis focuses primarily on four security-related objectives:

- **Trustworthy and secure access control**: ensuring that only authenticated and authorized devices and backends can interact with the system. This involves verifying the identity of each communicating entity and enforcing fine-grained access policies, preventing unauthorized access, data leakage, or malicious manipulation of energy assets.
- **Security of API requests and calls**: guaranteeing the integrity, confidentiality, and proper validation of exchanged messages. Even when parties are properly authenticated, the communication channel and payload remain potential targets for interception or tampering, making transport-level encryption and payload validation essential.
- **Traffic overload protection**: protecting backend services against denial-of-service scenarios and abnormal traffic patterns. Since the gateway sits at the boundary between untrusted device networks and the cloud backend, it must be capable of detecting and mitigating excessive or malicious traffic before it reaches core services.
- **Rate limitation**: enforcing usage constraints to prevent abuse and ensure fair resource allocation. Beyond global overload protection, per-device and per-consumer rate policies provide a finer-grained control layer that limits brute-force attempts, excessive polling, and other forms of resource exhaustion.

By positioning the API Gateway as a security enforcement point, the architecture reduces the direct exposure of the backend to compromised or malicious devices. It can also reduce the exposure of end devices to the public internet, thereby minimizing potential attack vectors, particularly for constrained or legacy devices with limited security capabilities. The gateway acts as an authorization, forwarding, and control layer before any request reaches core services.

Communication Objective

Beyond security considerations, a fundamental limitation of the initial architecture lies in the direct coupling between backend services and heterogeneous device communication protocols.

A core objective of this thesis is therefore to improve interoperability through protocol translation and mapping mechanism. The API Gateway is designed to handle device-specific communication and the backend to use commands towards these devices in a simple manner. This aspect is further detailed in the **Protocol Translation** section.

2.4.1 Trustworthy and secure access control

In an architecture where a cloud backend communicates with multiple heterogeneous renewable energy devices, controlling access to these resources becomes a critical security requirement. Trustworthy and secure access control ensures that only authenticated and authorized entities (e.g., cloud backend or devices) are allowed to interact with the system, preventing unauthorized access, malicious manipulation, or data leakage.

Given that renewable energy devices may expose configuration endpoints, monitoring data, and control interfaces, any unauthorized access could result in operational problems or safety risks. Therefore, a robust access control mechanism must guarantee strong authentication of communicating entities, enforce fine-grained authorization policies, and ensure the integrity of access decisions.

Several access control models and mechanisms can be used to achieve these objectives, each presenting different trade-offs in terms of flexibility, computational cost, and suitability for constrained environments.

Token-Based Access Control mechanisms such as JWT and OAuth2 are widely adopted in cloud and IoT architectures. However, as mentioned in [20], conventional token-based solutions designed for web and cloud applications cannot be directly applied to IoT systems, as they generally require high computational and bandwidth capabilities that constrained devices cannot provide, and offer poor interoperability with IoT-specific communication protocols.

Role-Based Access Control (RBAC) offers a simpler alternative by assigning permissions to predefined roles rather than individual entities, reducing administration overhead in environments with many devices sharing common access patterns. Its main limitation is its coarse granularity, which makes it difficult to express context-dependent or resource-specific policies.

Fine-Grained Access Control (FGAC) approaches, such as **Attribute-based access control** (ABAC) and **Policy-based access control** (PBAC) address this limitation by enabling richer and more expressive policies. Chen et al.[21] demonstrate that fine-grained access control is particularly valuable in IoT device-to-cloud monitoring architectures, where different entities require different levels of access to sensitive data and control interfaces. Ni et al.[22] further highlight that in fog computing environments, ABAC enables access decisions based on a combination of device attributes, user roles, and environmental conditions, making it particularly expressive for heterogeneous IoT deployments at the cost of increased policy complexity.

API Keys represent a lightweight alternative, particularly suited for resource-constrained devices that cannot participate in token-based flows. Issued during the onboarding of backends and devices, they provide a simple and vendor-agnostic mechanism to identify communicating parties. Their main drawbacks are the lack of contextual validation and the difficulty of revocation: a compromised key grants full access until it is manually invalidated.

In the proposed API Gateway architecture, a combination of some of these mechanisms will be used to establish a trustworthy access control layer that enforces authentication and authorization consistently across all communication flows.

2.4.2 Security of API request/call

Beyond access control, the security of API requests is essential to ensure the confidentiality, integrity, and authenticity of the exchanged data. In the context of renewable energy systems, this is particularly critical as API requests may carry sensitive configuration parameters or control commands that, if intercepted or tampered with, could directly affect the operation of physical assets [22, 10]. Ensuring the security of both the communication channel and the transmitted payload is therefore critical.

To achieve this objective, several mechanisms can be enforced:

- The use of protocols supporting Transport Layer Security (TLS) (e.g., HTTPS, MQTTS, CoAPS) to ensure encrypted communication channels between entities.
- Request validation and schema verification to prevent malformed or malicious payloads.

Within the proposed API Gateway architecture, these mechanisms are investigated and evaluated in terms of relevance, effectiveness, and integration feasibility. Not

all of them will necessarily be implemented. This work aims to select and integrate the most appropriate measures according to the system's security and performance requirements. The chosen mechanisms are further detailed in the implementation chapter.

2.4.3 Traffic Overload Protection

As previously discussed, the API Gateway is positioned at the edge of the cloud environment, directly interfacing with devices located in less trusted or potentially hostile environments. This positioning makes it a critical boundary component and, consequently, a potential target for overload or denial-of-service scenarios [12].

If a device becomes compromised or behaves abnormally, it may attempt to generate excessive traffic toward the gateway. Such behavior could aim to exhaust computational resources, saturate network bandwidth, or degrade the availability of the system. Even unintentional misconfigurations or software defects in devices may result in traffic bursts capable of impacting system stability.

Ensuring the availability and resilience of the API Gateway is therefore a fundamental objective. This work investigates mechanisms proposed in the literature [23] as well as those proposed by several API gateways on the market to detect and mitigate abnormal traffic patterns. Concretely, four categories of protection mechanisms are considered: **(1) concurrency limiting**, which caps the number of requests processed in parallel; **(2) dynamic IP blacklisting**, which blocks traffic from identified malicious sources; **(3) payload size enforcement**, which rejects oversized or malformed inputs; and **(4) response timeout enforcement**, which prevents the gateway from being blocked by unresponsive devices. The goal is to preserve service continuity for legitimate devices and backend services, even under adverse conditions.

2.4.4 Rate Limitation

In addition to global overload protection mechanisms, this thesis considers the implementation of rate limitation policies at the API Gateway level. Rate limiting provides a fine-grained control mechanism that constrains the number of requests a device or client can issue within a given time window.

Such mechanisms contribute to mitigating various attack vectors, including brute-force attempts, excessive polling, and certain forms of denial-of-service attacks. Beyond security, rate limitation also promotes fair resource usage among devices and helps maintain predictable system performance.

Several rate limitation strategies have been proposed in the literature [24] as well as those proposed by several API gateways on the market, each presenting different trade-offs in terms of precision, complexity, and performance. This work considers three approaches: **(1) a fixed-window hard quota**, which enforces a strict ceiling on

requests per time window, **(2) smooth throttling based on a leaky-bucket strategy**, which spreads bursts over time to ensure a more uniform load, **(3) per-consumer rate limiting**, which allows different quotas to be enforced per device or backend depending on its role or operational requirements. However, only the fixed-window hard quota approach will be implemented in this work, as discussed in the Chapter 5.

Configurability and adaptability are also important considerations. Rate policies may need to vary depending on device type, operational context, or service criticality. The objective is therefore to design a mechanism that balances protection, flexibility, and operational requirements.

2.4.5 Protocol Translation

Protocol translation constitutes a core functionality of the proposed API Gateway. It enables seamless communication between the cloud backend and heterogeneous renewable digital assets, despite differences in transport protocols, message formats, and device-specific data models.

From the backend perspective, communication is standardized through a unified interface, primarily based on HTTP. The backend issues generic and normalized requests, without embedding device-specific protocol details or formatting constraints. Upon receiving such a request, the API Gateway interprets it and performs the necessary protocol translation and data mapping operations. It then constructs a device-specific message adapted to the required transport protocol, payload structure, and command semantics before forwarding it to the target device.

This mechanism introduces an abstraction layer between the backend and the devices. The backend no longer needs to maintain detailed knowledge of individual device protocols, message structures, or communication patterns. As a result, integration complexity is reduced, coupling between business logic and device logic is minimized, and the system becomes more maintainable and extensible.

By centralizing protocol handling within the gateway, the architecture also facilitates the addition of new device types. Supporting a new device no longer requires modifying backend services, but instead extending the translation capabilities of the gateway.

The protocols that will be covered in this thesis are HTTP and MQTT. A more detailed description of the translation mechanisms and implementation choices is provided in the dedicated implementation chapter.

Summary of the Objectives and Motivations

In summary, this thesis aims to address the challenges of heterogeneous device integration and security in renewable energy systems by introducing a **Secure, Trustworthy, and Interoperable API Gateway**. The main objectives are to:

- Reduce the complexity on the backend side by centralizing device-specific communication and protocol handling.
- Enforce robust security policies, including authentication, authorization, and traffic protection.
- Facilitate interoperability between diverse devices and the cloud backend through a protocol translation and mapping mechanism.
- Provide resilience against overload and ensure fair resource usage via rate limitation mechanisms.

Together, these objectives guide the design and implementation of the proposed API Gateway, providing a practical solution for the secure and interoperable integration of heterogeneous renewable digital assets with a cloud environment.

Chapter 3

State of the Art

The communication and security challenges described in the previous chapter are not specific to renewable energy systems. Similar problems have been studied in the context of industrial IoT and microservice architectures, where researchers have proposed various solutions for multi-protocol communication and API security. Reviewing this literature is essential to understand what design principles and solutions already exist, what their limitations are, and where the contribution of this master's thesis fits within the current state of the art. This chapter organises existing work along four themes: multi-protocol interoperability and translation mechanisms, multi-protocol platform architectures, the Web of Things paradigm as a convergence strategy, and API gateway security in edge and microservice architectures.

3.1 Protocol Interoperability and Translation

Renewable energy devices, like many industrial systems, do not communicate over a single protocol. MQTT, CoAP, HTTP, AMQP, and WebSockets each have legitimate use cases depending on the device type, connectivity constraints, and communication pattern involved. Since protocol convergence on a single standard remains unlikely in the foreseeable future [25], significant research effort has gone into building translation mechanisms that allow these protocols to coexist within the same system.

Derhamy et al. [25] address this problem in the context of Industrial IoT by proposing a secure, on-demand, and transparent multiprotocol translator built within a Service-Oriented Architecture (SOA) framework. Their key observation is that direct protocol-to-protocol translation scales quadratically with the number of protocols, following $\frac{n(n-1)}{2}$, which quickly becomes unmanageable. To address this, the authors introduce an intermediate representation *format* (but not an intermediate protocol) that captures all protocol-specific information internally without being transmitted over the wire. This hub-and-spoke architecture reduces the number of required trans-

lators to $n-1$ while preserving the information fidelity of direct translation. The system operates on demand, meaning translation instances are provisioned only when a protocol mismatch is detected by the Arrowhead SOA orchestrator, and torn down after a configurable time-to-live. Security is handled through the Arrowhead authorisation system, which issues time-limited tokens and enables fine-grained access control decoupled from the translation logic itself. The translator is implemented in Java and is available as open-source software, actively maintained as part of the broader Eclipse Arrowhead project.

Performance. The translator was evaluated on a BeagleBone Black, benchmarked against the Californium HTTP-CoAP proxy running on the same hardware. Round-trip time was measured at the network interface, isolating processing overhead on the device from external network latency. The translator introduced average delays of 50-77 ms depending on payload size, compared to 146-177 ms for the Californium proxy under the same conditions.

Da Cruz et al. [26] take a more pragmatic, device-centric approach with *MiddleBridge*, an application-layer gateway that translates MQTT, CoAP, WebSocket, and DDS messages into HTTP. The motivation is that most IoT middleware platforms natively support only HTTP and MQTT, leaving devices using other protocols unable to communicate with the backend without code changes. Unlike earlier bridges such as Ponte and Gothings (both of which have since been deprecated) MiddleBridge additionally applies a payload reduction strategy: rather than forwarding full JSON or XML messages unchanged, it accepts a compact partial payload from the device, reconstructs the full HTTP request internally, and forwards it to the target middleware. The system is deployable on any Java-capable device and exposes a graphical user interface for runtime configuration. MiddleBridge is available as open-source under the GNU GPL v3 licence, though its repository has not seen active maintenance since its initial publication.

Performance. Experiments were conducted on a Raspberry Pi 3 hosting MiddleBridge, with an Orion context broker deployed on a separate workstation and load generated via Apache JMeter. Round-trip time and packet size were measured from the IoT device to MiddleBridge only, as MiddleBridge closes the device connection before forwarding to the middleware. Under 100 concurrent users, average round-trip times were 2 ms for MQTT and 8 ms for CoAP, compared to 343 ms for direct HTTP to the broker. In terms of packet size, the payload sent by the IoT device through MiddleBridge was 17 times smaller than an equivalent direct HTTP request.

3.2 Multi-Protocol IoT Platform Architectures

Beyond point-to-point protocol translation, a parallel line of research has focused on building integrated IoT platforms that natively support multiple application-layer protocols under a unified management and access-control framework.

Akasiadis et al. [27] present a multi-protocol IoT platform built on top of the SYNAISTHISI cloud platform, integrating four open-source components: RabbitMQ, Ponte, OM2M, and RDF4J, into a single Dockerised deployment that supports MQTT, AMQP, CoAP, REST/HTTP, and WebSockets simultaneously. RabbitMQ and Ponte serve as cross-translation engines, bridging messages across protocols transparently at the messaging layer. The oneM2M common service layer is provided through the OM2M framework, offering standardised device management and resource discovery. Semantic annotations via an RDF4J triplestore enable ontology-based service discoverability, which goes beyond protocol translation and addresses the semantic dimension of interoperability. Authentication and authorisation are implemented through a Flask-based portal, with access control extended to the message brokers so that publish and subscribe permissions are validated dynamically per user and per topic. The platform follows a marketplace architecture promoting service reusability and is delivered as a Dockerised container for local deployment. Of the four constituent components, RabbitMQ and RDF4J remain actively maintained open-source projects; Eclipse Ponte has been archived and is no longer actively developed, and Eclipse OM2M has similarly seen no active maintenance since the early 2020s.

Performance. Experiments were run on an Intel Core i5-4440 machine with a separate client machine connected over a 100 Mbps LAN. End-to-end message delivery was measured from publish to reception at a subscribing client. For publish/subscribe protocols, MQTT and AMQP maintained delivery times well below 25 ms under moderate load. AMQP consistently outperformed MQTT, as its authorisation checks are handled natively within RabbitMQ, whereas MQTT relies on external calls to the Flask portal, introducing additional overhead. For request/response protocols (CoAP and HTTP), average translation delays reached into the hundreds of milliseconds due to the polling-based message retrieval model, with CPU usage exceeding 80% under the same load conditions.

3.3 The Web of Things Paradigm as an Interoperability Strategy

The Web of Things (WoT) paradigm offers a higher-level approach to protocol interoperability by encouraging IoT device resources and functionalities to be exposed

as RESTful HTTP services, making them accessible using established web standards regardless of the underlying device protocol.

Benomar et al. [28] introduce a gateway-based WoT system that combines two platforms: Stack4Things (S4T), which handles dynamic DNS and remote accessibility of gateway-hosted services, and the Data eXchange Mediator Synthesizer (DeXMS), which automatically synthesises protocol mediators that adapt MQTT, CoAP, AMQP, and other protocols into RESTful HTTP interfaces. The system simultaneously addresses two interoperability challenges: (1) protocol heterogeneity, handled by DeXMS mediators, and (2) network reachability. This network reachability challenge, which arises when gateways are deployed behind NATs or firewalls, is addressed by S4T's NGINX reverse proxy and WebSocket tunnel infrastructure. The architecture follows the WoT indirect integration pattern, placing a smart gateway between resource-constrained IoT devices and cloud services, abstracting device protocols and exposing a unified HTTP interface to applications. Mediators are deployed as containerised Java JAR files on the gateway, generated on demand by the cloud-side DeXMS service and managed through IoTronic's remote provisioning system. The system supports bidirectional communication, allowing both data retrieval from devices and actuation towards them. Both Stack4Things and DeXMS are research-oriented open-source platforms whose active maintenance is limited to the academic groups that produced them.

Performance. Tests were conducted on a Raspberry Pi 3 Model B+ gateway. CPU usage and message throughput were measured per mediator under increasing message rates. For the MQTT-to-HTTP mediator, CPU usage scaled approximately linearly with message rate, reaching around 43% at 200 messages per second, with message drops beginning near 175 messages per second. For the CoAP-to-HTTP mediator, saturation occurred earlier at approximately 125 messages per second, attributed to the absence of an intermediate message queue. RAM usage remained stable at approximately 9.4% for MQTT and 9.7% for CoAP across all tested rates, regardless of message rate.

3.4 API Gateway Security in Edge and Microservice Architectures

While the works reviewed in the preceding sections primarily address interoperability at the protocol and platform levels, a complementary body of research has focused on the security challenges that arise when IoT devices and microservices are exposed through API gateways in edge computing environments.

Xu et al. [29] propose a microservice security agent architecture that positions the API gateway as the central security enforcement point in an edge computing deploy-

ment. The motivation is a recognised limitation of conventional microservice security models: when security logic is distributed across individual services, each must independently implement and maintain authentication and access control mechanisms, introducing redundancy, inconsistency, and an enlarged attack surface. The proposed architecture addresses this by externalising security concerns to a dedicated agent co-located with the API gateway, so that all inbound requests are intercepted and validated before reaching any backend microservice. The security agent is built on top of the open-source Kong API gateway, deployed within the EdgeX Foundry framework. It integrates two enforcement layers: a JSON Web Token (JWT) authentication module implementing a Kong plugin that validates HS256- or RS256-signed tokens as specified in RFC 7519, and an Access Control List (ACL) authorisation module that associates each consumer with a named whitelist group, rejecting requests from consumers not belonging to an authorised group for the target resource. Both layers operate as a unified pipeline, ensuring that JWT authentication is evaluated before ACL authorisation. A companion client server provides a graphical interface for consumer registration, token generation, and ACL group assignment, while a security agent microservice exposes REST APIs to interact with Kong's admin interface programmatically. Kong remains an actively maintained open-source project, and EdgeX Foundry continues to be developed under the Linux Foundation.

Performance. Experiments were conducted on a desktop machine running Ubuntu 18.04 with an AMD 64 quad-core processor and 4 GB RAM. Round-trip time was measured from client to microservice for both secured and unsecured paths. Accessing a microservice through the full JWT and ACL enforcement pipeline yielded an average round-trip time of approximately 9 ms, compared to 3 ms for direct access to an unsecured microservice on the same hardware, giving an overhead of approximately 6 ms attributable to the two security enforcement layers.

3.5 Discussion and Research Gap

The reviewed works collectively address the two main technical dimensions this thesis tackles, each from a distinct angle and within its own application context. The following part situates the contribution of this thesis relative to what already exists and to clarify the particular combination of requirements that motivated a new design.

On the interoperability side, Derhamy et al. [25] and Da Cruz et al. [26] focus primarily on translation efficiency and device-side simplicity, with security considerations left to the surrounding system infrastructure. Akasiadis et al. [27] and Benomar et al. [28] take a broader platform view and include authentication mechanisms, though their security models are tailored to the general-purpose IoT ecosystems they target rather than to adversarial field environments. Across all four works, protocol trans-

lation is treated as a structural concern: the goal is to convert message framing and delivery semantics between heterogeneous protocols. Device-specific payload formatting, where the content of a command must conform to a vendor-specific structure, falls outside the scope of these systems, as it is more of an integration concern than a protocol concern. In the context of renewable energy assets, however, this distinction becomes important. Field devices from different vendors often expect commands in different payload structures, and a backend system benefits from being able to issue requests in a uniform format and have the gateway handle the adaptation per device.

On the security side, Xu et al. [29] demonstrate that centralising JWT authentication and ACL-based access control at the gateway layer is both architecturally sound and practically efficient within an HTTP microservice environment. Their work provides a strong reference point for our design choice. The context they address, however, is a homogeneous HTTP deployment with registered consumers, which differs from an environment where multiple device protocols coexist and where access policies need to be defined at the level of individual device commands rather than service endpoints.

The proposed architecture in this thesis brings these two dimensions together in a single gateway, combining multi-protocol translation for HTTP and MQTT devices with JWT-based authentication, role-based authorisation, and per-device command access policies. A particular focus is placed on payload mapping alongside protocol conversion, allowing backends to communicate in a uniform format while the gateway handles the device-specific translation.

The performance characteristics of this architecture are evaluated experimentally in Chapter 6.

Chapter 4

Evaluation and Selection of an Open-Source API Gateway

The design and implementation of an API Gateway architecture for renewable energy infrastructures requires the selection of a suitable existing open-source solution capable of meeting the functional and non-functional requirements mentioned in Table 4.1. In addition, it provides a solid and reliable foundation for the API Gateway architecture, where several required mechanisms may already be natively supported, reducing development complexity and implementation effort.

Given the security constraints, protocol heterogeneity, and operational demands of the targeted environment, the choice of the API Gateway represents a critical architectural decision. Making the wrong choice can limit our development and the possibility of achieving all the identified requirements.

This chapter presents a systematic evaluation of existing open-source API Gateway tools, followed by a comparative analysis and the justification of the selected tool.

4.1 Evaluation Methodology

In this section, the evaluation of five widely used open-source API gateways is explained based on a set of functional and non-functional requirements. Some of these API gateways are fully open source, while others offer both open-source and enterprise features. The focus is exclusively on the open-source features of each gateway to determine whether they meet the desired requirements and also the possibility of implementing the protocol translation mechanism using custom plugins.

The five API gateways are: (1) Kong [30], (2) Tyk [31], (3) Express Gateway [32], (4) KrakenD [33], and (5) Apache APISIX [34]. These gateways were selected based on their popularity in the industry, active community support, and rankings reported in online API gateway leaderboards [35, 36, 37, 38, 39].

For each API Gateway, each requirement is categorized according to the following scheme:

- ✓ : Fully supported natively or through built-in features and plugins
- ⚠: Partially supported, requiring additional configuration, plugins, or external components
- × : Not supported

The evaluation methodology is based on a set of functional and non-functional requirements:

Functional Requirements	
Category	**Criteria**
Trustworthy and Secure Access Control	Token-based authentication (JWT, OAuth2) Coarse-grained access control (RBAC) Fine-grained access control (ABAC, PBAC) Key authentication (API Keys)
Key Management	Internal storage External storage
Security of API Request/Call	TLS protocol support Payload integrity Request and schema validation
Traffic Management	Concurency Limiting Dynamic IP blacklisting Payload size enforcement Response timeout Fixed-window hard quota Leaky-bucket smooth throttling Per-consumer quota
Protocol Support	HTTP(S) MQTT(S) CoAP(S)
Non-Functional Requirements	
Deployment capabilities Configurability and runtime management Sustainability and community support	

Table 4.1: Functional and non-functional evaluation criteria for the comparative analysis of open-source API Gateways

4.2 Comparative Analysis

In this section, the evaluation of the selected API gateways across the functional and non-functional requirements mentioned in Section 4.1 is detailed.

4.2.1 Trustworthy and secure access control

This category covers all the mechanisms that an API gateway can use to restrict access of cloud backends and devices to each other's resources, as well as to the API Gateway itself.

Token-Based Authentication Mechanisms such as JWT and OAuth2 ensure that the party sending a request to the API Gateway is legitimate by verifying that the tokens included in the request headers are valid, with the correct structure, and issued by a trusted third party.

API Gateway	JWT	OAuth2
Kong	✓	✓
Tyk	✓	✓
Express Gateway	✓	⚠
KrakenD	✓	⚠
Apache APISIX	✓	✓

Table 4.2: Comparison of token-based authentication mechanisms

As shown in the Table 4.2, all five API Gateways provide native support for JWT authentication. Regarding OAuth2, Kong, Tyk, and Apache APISIX provide native OAuth2 capabilities, including both token issuance (either via an internal mechanism or by interacting with an external authorization server) and token validation. Express Gateway and KrakenD, on the other hand, partially support it, limited to validating the authenticity of received OAuth2 tokens.

Coarse- and Fine-Grained Access Control Mechanisms such as RBAC, ABAC, and PBAC can be used to specify who can access the routes exposed by the API Gateway and who can send requests to whom. For example, RBAC can assign roles to the admins of devices and cloud backends, allowing only authorized users to access specific configuration routes. PBAC can define policies that determine which type of backend can send commands to which type of device, based on information provided by the admins of the entities.

API Gateway	RBAC	ABAC	PBAC
Kong	⚠	⚠	⚠
Tyk	⚠	⚠	✓
Express Gateway	⚠	⚠	⚠
KrakenD	⚠	⚠	⚠
Apache APISIX	✓	⚠	✓

Table 4.3: Comparison of coarse- and fine-grained access control mechanisms

As shown in Table 4.3, for RBAC, only Apache APISIX supports it natively through an existing plugin that uses Wolf and a serverless pre-function plugin allowing to write functions to restrict access. The four other API Gateways only support it partially or not at all, which may require implementing a custom plugin. For ABAC, the five API Gateways support it partially, which may require implementing a custom plugin. Finally, PBAC is natively supported by Tyk, as well as by Apache APISIX through an existing plugin that can be integrated with the Open Policy Agent (OPA), but also through custom plugins.

Key Authentication Mechanisms such as API Keys provide an additional layer of security to specify whether a device or backend is authorized to send requests to the API Gateway. For example, when a device or cloud backend is registered with the API Gateway, a unique API Key can be generated specifically for it. The API Gateway then requires this key to be included in all future requests from that device or backend, ensuring that only authorized parties can access the exposed routes.

API Gateway	API Keys
Kong	✓
Tyk	✓
Express Gateway	✓
KrakenD	✓
Apache APISIX	✓

Table 4.4: Comparison of key authentication mechanisms

As shown in Table 4.4, all five API Gateways provide native support for API Key authentication, allowing the generation and validation of unique keys that can be assigned to devices or backends to control access to exposed routes.

4.2.2 Key Storage

This requirement focuses specifically on the storage of API keys and tokens generated and used by the API Gateway. Two main approaches can be identified: **Internal storage** and **External storage**. Internal storage refers to the ability of the API Gateway to store API keys or tokens within its own system, without requiring an external system. External storage refers to cases where the API Gateway relies on external systems, such as databases or key-value stores, to store and retrieve keys and tokens.

The goal is to assess whether each API Gateway can manage the storage of keys generated securely and practically, or whether there is a need for external systems to achieve that.

API Gateway	Internal Storage	External Storage
Kong	×	✓
Tyk	×	✓
Express Gateway	✓	⚠
KrakenD	×	✓
Apache APISIX	×	✓

Table 4.5: Comparison of key and token storage approaches

As shown in Table 4.5, only Express Gateway provides internal storage capabilities through in-memory or file-based storage. However, its support for external storage is limited and less commonly used, which justifies partial support for external integration. The four other API Gateways only depend on external systems and storage mechanisms for persistence. For example, Apache APISIX uses etcd as a distributed key-value store to persist configuration data, including credentials. In addition, databases like CouchDB and MongoDB can be used by these gateways to provide more storage features if what is used by default is not enough.

4.2.3 Security of API request/call

This requirement covers the security of the requests sent to and from the API Gateway. To ensure this security, several techniques can be used, such as protocols using TLS, payload integrity checks, and schema validation. For example, it can be required that anyone who wants to communicate with the API Gateway use protocols such as HTTPS. The payload can also be checked to ensure it has not been tampered with and to verify that the structure of the request is the one expected.

As shown in Table 4.6, all the API Gateways provide TLS support to secure communications between clients and the gateway. Payload integrity verification is also supported by most of the API Gateways, either natively or through built-in plugins.

API Gateway	TLS Support	Payload Integrity	Schema Validation
Kong	✓	⚠	⚠
Tyk	✓	✓	✓
Express Gateway	✓	⚠	⚠
KrakenD	✓	✓	✓
Apache APISIX	✓	✓	✓

Table 4.6: Comparison of security features for API requests and calls

Apache APISIX offers request-validation plugins that check request bodies and parameters. Kong also provides request-validation and schema-validation features, but these are only available in the Enterprise edition and are not part of the open-source gateway. KrakenD provides a JSON Schema Validation mechanism, while Tyk provides middleware for similar verification. However, Express Gateway requires a custom plugin or external packages to achieve payload integrity. Regarding schema validation, Tyk, KrakenD, and Apache APISIX can enforce request structure against defined JSON Schemas or OpenAPI specifications. Express Gateway does not provide built-in schema validation and relies on a custom implementation.

4.2.4 Traffic management

This requirement covers the mechanisms implemented in the API Gateway to manage and control the flow of incoming requests, protecting both the gateway and upstream services from abuse, overload, and abnormal traffic patterns. Two complementary categories are considered: traffic overload protection and rate limitation.

Traffic Overload Protection encompasses mechanisms that detect and mitigate abnormal or excessive traffic. **Concurrency limiting** caps the number of requests processed in parallel, preventing resource exhaustion under sudden load spikes. **IP blacklisting and dynamic IP banning** allow the gateway to block traffic from source IP addresses identified as malicious or abusive, for instance, in response to a DDoS attack originating from a compromised device or backend. **Request size limiting** rejects incoming requests whose payload exceeds a configurable threshold, protecting upstream services from oversized or malformed inputs. **Timeout enforcement** ensures that requests forwarded to upstream devices are automatically cancelled after a configurable delay, preventing the gateway from being blocked by unresponsive endpoints.

Rate Limitation mechanisms control how many requests a client can issue over time. A **global hard quota** enforces a fixed ceiling on the number of requests al-

lowed per route within a given time window, regardless of traffic distribution. **Global smooth throttling** applies a leaky-bucket or sliding-window strategy to spread request bursts over time, ensuring a more uniform and predictable load on upstream services. **Per-consumer rate limiting** extends this control to individual registered devices or backends, allowing different quotas to be enforced per entity based on their role, type, or operational requirements.

API Gateway	Concurrency Limiting	Dynamic IP Blacklisting	Payload Size Enforcement	Response Timeout
Kong	✓	✓	✓	✓
Tyk	✓	✓	✓	✓
Express Gateway	⚠	⚠	⚠	⚠
KrakenD	✓	⚠	✓	✓
Apache APISIX	✓	✓	✓	✓

Table 4.7: Comparison of traffic overload protection mechanisms

API Gateway	Fixed-Window Hard Quota	Leaky-Bucket Smooth Throttling	Per-Consumer Quota
Kong	✓	⚠	⚠
Tyk	✓	⚠	⚠
Express Gateway	⚠	⚠	⚠
KrakenD	✓	⚠	⚠
Apache APISIX	✓	✓	✓

Table 4.8: Comparison of rate limitation mechanisms

As shown in Tables 4.7 and 4.8, Kong, Tyk, KrakenD, and Apache APISIX natively support concurrency limiting, payload size enforcement, and upstream response timeout. For dynamic IP blacklisting, Kong, Tyk, and Apache APISIX provide native support through built-in plugins, while KrakenD only partially supports it through external tooling and Express Gateway requires custom middleware. Across all four overload protection mechanisms, Express Gateway consistently requires external components or custom development to achieve equivalent functionality.

Regarding rate limitation, all gateways except Express Gateway natively support a global fixed-window hard quota. For smooth throttling, only Apache APISIX provides full native support through its `limit-req` plugin, which implements a leaky-bucket algorithm. The four other gateways only partially support this and require additional configuration or external tooling. Finally, for per-consumer rate limiting, only Apache

APISIX provides full native support through the combination of its `limit-count`, `limit-req`, and `limit-conn` plugins applied at the consumer level, while all other gateways only partially support this and require additional custom development.

4.2.5 Protocol support

This requirement evaluates the ability of the selected open-source API Gateways to support and handle the communication protocols commonly used between cloud backends and devices. This requirement also reflects the interoperability of the API Gateway with heterogeneous communication protocols and external systems used in renewable energy systems. In this context, three types of protocols are considered: HTTP(S), MQTT(S), and CoAP(S).

API Gateway	**HTTP(S)**	**MQTT(S)**	**CoAP(S)**
Kong	✓	⚠	×
Tyk	✓	⚠	×
Express Gateway	✓	⚠	×
KrakenD	✓	⚠	×
Apache APISIX	✓	⚠	×

Table 4.9: Comparison of protocol support

As shown in Table 4.9, all the API Gateways natively support HTTP(S). For MQTT, it is partially supported or can be using custom plugins. Apache APISIX, for example, provides an MQTT-proxy plugin that only allows the transfer of an MQTT request that the gateway receives to a broker. However, if the gateway needs to translate an HTTP request to an MQTT request and send it, a custom plugin is needed to implement the process of sending an MQTT request. For the other API Gateways, MQTT support is only possible through custom plugins, as they do not provide any native or plugin-based MQTT functionality like APISIX. Regarding CoAP(S), there is no native support for any of the gateways, and implementing custom plugins to handle CoAP is considerably more complex compared to MQTT.

4.2.6 Deployment capabilities

The purpose of this non-functional requirement is to evaluate how easily an API Gateway can be deployed and made operational once it has been configured and is ready for release.

Kong is relatively easy to deploy. It supports multiple deployment methods, including Docker containers, Kubernetes (via Helm charts), traditional package installation on bare-metal systems, and cloud-based virtual machines [40]. Additionally, Kong supports various deployment topologies, including DB-less, hybrid mode, and traditional mode, which allow it to be fully pre-configured and ready to use after deployment, reducing the work required after startup [41].

Tyk API Gateway is also relatively easy to deploy. The open-source Tyk Gateway can be deployed using several methods, including Docker containers, Kubernetes deployments (via Helm charts), Ansible scripts, and native package installation on systems such as Ubuntu or CentOS [42]. APIs, policies, and configurations can be defined prior to deployment using configuration files or API definitions, allowing partial pre-configuration before startup.

Express Gateway can be deployed in Node.js environments and is typically installed using npm. It also provides official Docker images for container-based deployments. Its configuration is file-based, allowing the gateway to be pre-configured before startup and to be operational immediately after deployment [43].

KrakenD is distributed as a single binary and can be deployed without additional dependencies on multiple platforms. It also supports Docker and Kubernetes deployments. Its stateless architecture allows configuration to be fully defined in declarative configuration files (supporting formats such as JSON, YAML, and TOML) prior to deployment, enabling immediate operation after startup[44].

Apache APISIX supports multiple deployment methods, including Docker, Kubernetes (via Helm charts), and installation on Linux systems. It requires etcd for configuration storage, which must be deployed alongside the gateway. Configuration is loaded dynamically from etcd at runtime rather than being fully defined before startup, allowing the gateway to become operational once both APISIX and etcd are properly configured [45].

4.2.7 Configurability and runtime management

Kong provides configuration through declarative configuration files in DB-less mode or via an Admin API when using a database-backed deployment. It supports dynamic updates at runtime without requiring service restarts, and plugins can be configured or modified through the Admin API [46].

Tyk offers both file-based configuration and REST API-based management. The Tyk Gateway API allows runtime updates to APIs, policies, and keys without restarting the gateway. It also supports hitless reloads, allowing configurations to be altered dynamically without interrupting active requests [47].

Express Gateway relies primarily on declarative configuration using YAML files. While it provides limited administrative APIs, most configuration changes require updating configuration files and restarting the gateway, making runtime management relatively limited [48].

KrakenD uses a declarative configuration model. In the open-source version, configuration changes typically require a restart of the gateway, as runtime reconfiguration is limited. The gateway intentionally computes all routing and decision trees at startup for performance reasons [49].

Apache APISIX supports dynamic configuration through its Admin API and stores configuration in etcd. Changes to routes, services, and plugins are applied in real time without requiring a restart, enabling full runtime management [50].

4.2.8 Sustainability and community support

Kong is backed by a commercial company and maintains an active open-source community. It provides extensive documentation, regular releases, and a large ecosystem of plugins. Community support is available through forums, GitHub, and official documentation [51].

Tyk is maintained by Tyk Technologies and offers both open-source and commercial editions. It has active community support, documentation, and regular updates, with additional enterprise support available [52].

Express Gateway is an open-source project maintained on GitHub. It provides documentation and community-based support, although it has a smaller community and lower update frequency compared to other API gateways [53].

KrakenD has an active open-source community and is supported by a company offering enterprise features. It provides documentation, community forums, and regular updates, with additional support options available for enterprise users [54].

Apache APISIX is an Apache Software Foundation project, ensuring long-term sustainability and strong governance. It benefits from an active community, frequent contributions, and comprehensive documentation, with support available through mailing lists and GitHub [55].

4.3 Discussion and Final Selection

The comparative analysis highlights significant differences between the evaluated API Gateways in their ability to satisfy the functional and non-functional requirements defined earlier.

From Table 4.10, we can clearly observe that **Apache APISIX** achieves the highest number of fully supported criteria (✓ = 17), while also maintaining a low number of partial supports and unsupported features.

API Gateway	✓	⚠	×
Kong	11	8	2
Tyk	14	5	2
Express Gateway	5	15	1
KrakenD	11	8	2
Apache APISIX	17	2	2

Table 4.10: Summary of API Gateway feature support across all evaluated criteria

Apache APISIX stands out due to several key advantages:

- **Comprehensive traffic management:** Native support for advanced rate-limiting strategies, including leaky-bucket throttling and per-consumer quotas.
- **Extensibility through plugins:** A flexible plugin architecture that facilitates the implementation of custom logic, including protocol translation mechanisms required in this work.
- **Dynamic configuration:** Real-time updates via the Admin API without requiring service restarts, which is essential for highly dynamic environments.
- **Integration with external systems:** The use of etcd enables scalable and consistent configuration management.
- **Strong policy enforcement:** Integration with tools such as Open Policy Agent and built-in/custom plugins allows the implementation of fine-grained access control mechanisms.

Although **Apache APISIX** does not natively support all required protocols, such as CoAP and only partially supports MQTT, its plugin-based architecture makes it particularly well-suited for implementing custom protocol translation, which is a central requirement of this thesis.

Considering the specific constraints of renewable energy infrastructure, **Apache APISIX** provides the best balance between out-of-the-box functionality and extensibility, making it the best solution for this work, as it most effectively satisfies the identified requirements while minimizing the need for additional development effort. It is widely adopted in industry, with thousands of production deployments across various sectors, including large-scale enterprise systems. Although it is not specifically designed for renewable energy environments, its general-purpose architecture and extensibility make it well suited to such use cases.

4.4 APISIX

Apache APISIX is a dynamic, real-time, high-performance open-source API gateway developed under the Apache Software Foundation [56]. It is built on top of OpenResty, a combination of NGINX and LuaJIT, and uses `etcd` as its distributed configuration store [57]. This design choice gives APISIX two important properties: on one hand, it inherits the proven event-driven, non-blocking I/O model of NGINX, enabling it to sustain tens of thousands of requests per second with sub-millisecond average latency [56]. On the other hand, configuration changes are propagated to every running node in real time through etcd's watch mechanism, without requiring a gateway restart [58]. This aspect is particularly important, as the proposed solution requires to be configurable dynamically at runtime, without impacting the overall availability of the API Gateway.

Conceptually, APISIX separates its concerns into two main planes. The **data plane** is responsible for request matching, plugin execution, load balancing, and proxying traffic to upstream services. The **control plane** exposes a REST Admin API through which operators and automation tooling create and update routes, upstreams, consumers, and plugin configurations [57]. This separation makes it straightforward to deploy APISIX as a horizontally scalable cluster where every node independently processes requests while sharing a common configuration state.

Figure 4.1 illustrates the internal layered structure of the data plane. At its foundation lie **NGINX** and **OpenResty**, which provide the underlying HTTP server and LuaJIT scripting environment. Above them, the **APISIX Core** handles route matching, load balancing, and configuration management, while the **Plugin Runtime** orchestrates the execution of plugins across the request lifecycle. The top layer exposes built-in plugins grouped by concern: observability, security, and traffic management, as well as an extension point for custom plugins written in a broad range of languages, including

Java, Go, Python, and JavaScript [59].

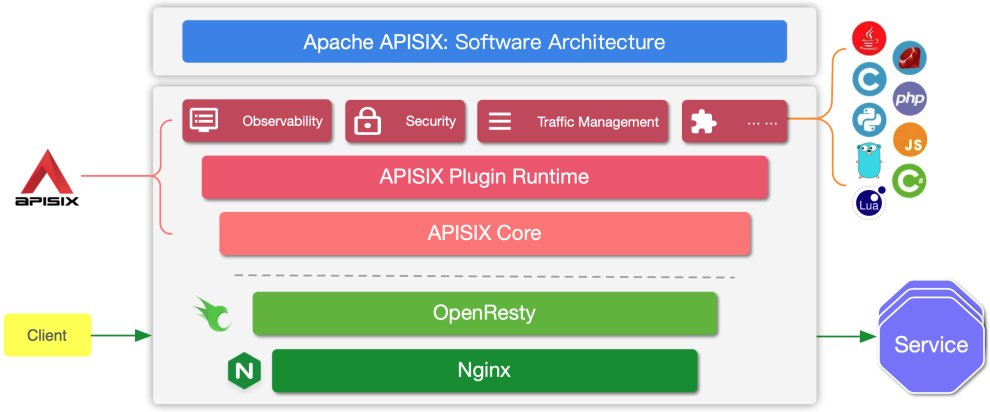

Figure 4.1: APISIX Architecture (Figure from APISIX documentation [57])

A central concept in APISIX is the *route*. A route binds a set of matching predicates (HTTP method, URI, host, headers, ...) to an upstream target and to an ordered list of plugins that must be executed for each matching request. Complementary abstractions such as *Services*, *Consumers*, and *Plugin Configs* allow configuration to be reused and composed across multiple routes [60]. Roughly one hundred built-in plugins are available out of the box, covering authentication (JWT, Key Auth, ...), traffic control (rate limition, ...), observability (Prometheus, Apache SkyWalking, ...), and request/response transformation, among others [56].

Because all built-in plugins are written in Lua and executed inside the NGINX worker process, APISIX also provides a multi-language extensibility layer that allows developers to implement custom plugins in Java, Go, Python, or JavaScript through a **plugin runner** mechanism [59]. This runner infrastructure is described in the next subsection.

4.4.1 Java Plugin Runner

Although Lua, the default language for plugin development in APISIX, is lightweight and highly performant, the complexity of the present contribution calls for a more object-oriented language suited to large-scale software projects. To accommodate this need, APISIX introduces the concept of an *external plugin runner* [59]. Concretely, *apisix-java-plugin-runner* is the official implementation that enables custom plugins written in Java to participate in the APISIX request-processing pipeline.

Deployment model. When the runner is configured in `config.yaml`, APISIX spawns the Java process as a *subprocess* that runs alongside the gateway under the same

system user. The runner is therefore treated as a sidecar: it shares the lifecycle of the gateway instance and is automatically restarted whenever APISIX is reloaded [61].

Communication protocol. APISIX and the Java runner communicate through a *Unix Domain Socket* (UDS). The socket path is transmitted to the runner at startup via the environment variable `APISIX_LISTEN_ADDRESS`. All messages exchanged over this channel are serialised with *FlatBuffers*, a zero-copy binary serialisation library, which keeps the inter-process overhead as small as possible [62].

For each request that must invoke the external runner, APISIX performs two types of *Remote Procedure Call* (RPC):

- `PrepareConf`: used to synchronise plugin configuration to the runner. When a route configuration carrying an external plugin is first seen, APISIX sends the plugin parameters to the runner, which stores them in a local cache and returns a short-lived *conf token*. Subsequent requests carry that token instead of the full configuration, avoiding redundant data transfer [62].
- `HTTPReqCall`: the per-request call. APISIX serialises the HTTP request (method, URI, headers, body) together with the conf token and sends it to the runner. The runner deserialises the payload, constructs a virtual request object, executes the registered Java filters in order, and returns to APISIX either a set of request mutations to apply before forwarding, or a complete response to send back directly to the client [62, 61].

Execution phases. Three dedicated plugin slots integrate the runner into the APISIX pipeline:

1. `ext-plugin-pre-req`: fires *before* the built-in Lua plugins are executed,
2. `ext-plugin-post-req`: fires *after* these plugins,
3. `ext-plugin-post-resp`: fires *after* the response from the upstream service.

This ordering gives developers fine-grained control over when Java logic runs relative to authentication, rate-limiting, or any other built-in plugin [61, 59]. Figure 4.2 illustrates these principles.

Filter interface. From the Java developer's perspective, each custom plugin is a class that implements the `PluginFilter` interface. The only mandatory method is `name()`, which returns the identifier used in the route configuration. Two processing methods can then be implemented depending on the desired phase: `filter(HttpRequest, HttpResponse, PluginFilterChain)` acts at request time, before the request is

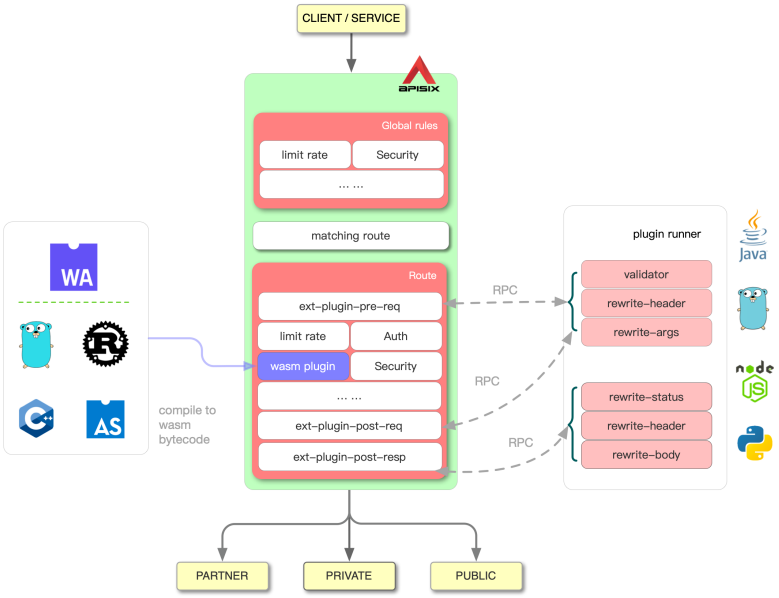

Figure 4.2: APISIX external plugin integration (Figure from APISIX documentation [59])

forwarded to the upstream, while `postFilter(PostRequest, PostResponse, PluginFilterChain)` acts at response time, after the upstream has replied. Finally, optional methods (`requiredVars()`, `requiredBody()`, and `requiredRespBody()`) allow the filter to specify in advance which NGINX variables or request/response body content it needs, so that APISIX can include them in the `HTTPReqCall` payload [63].

4.4.2 Filter-based routes

The request-processing model of APISIX follows a *filter-based pipeline* pattern [57]. Every route acts as the entry point for an ordered chain of plugins (filters), and each plugin may either mutate the request or response, terminate the exchange early, or pass control to the next plugin in the chain.

Plugin execution order. When a request arrives at the gateway, APISIX first performs route matching. Once a route is identified, the plugins attached to that route are executed sequentially, in a predefined priority order. The built-in Lua plugins run inside the NGINX worker process across several *OpenResty phases*: `rewrite`, `access`, `header_filter`, `body_filter`, and `log` [56]. External plugins (e.g. Java runner) are invoked via RPC at the `ext-plugin-pre-req` or `ext-plugin-post-req` slots, as described in Section 4.4.1.

Short-circuit semantics. A critical property of the pipeline is its *short-circuit* behaviour. If any plugin in the chain modifies the HTTP response object, for instance by setting a response body, status code, or headers, APISIX treats this as a terminal action and sends the response directly back to the client, bypassing all subsequent plugins and the upstream service entirely. This mechanism is used by authentication plugins (to reject unauthenticated requests with a `401`), by rate-limiting plugins (to reject excess traffic with a `429`), or by any custom plugin that needs to serve a response without consulting the upstream [62, 64]. Conversely, if no plugin terminates the exchange, the fully processed (and possibly rewritten) request is forwarded to the upstream, which produces the definitive response. That response may in turn be passed through `header_filter` and `body_filter` plugins before being relayed to the client.

4.4.3 Usage in this Contribution

In the context of this work, the execution phases of the pipeline prove particularly relevant: the built-in Lua plugins allow lightweight filtering to intercept and process requests before they reach the core logic, while the `ext-plugin-post-req` phase, executed after the built-in plugins, hosts the core logic of authentication, authorization, and translation, along with the possible direct construction of the HTTP response. Combined with the short-circuit semantics described above, these mechanisms allow complex logic to be encapsulated directly within the filters themselves, without delegating any processing to an upstream service. Concretely, this contribution relies on a selection of built-in plugins for network-level and authentication concerns, and extends the gateway through the Java plugin runner, which constitutes the primary development effort of this work.

Chapter 5

API Gateway Architecture and Implementation

This chapter presents the implementation of this contribution. It begins by describing the general architecture in Section 5.1, covering the defined routes and the overall operation of the system. The five main principles of the work are then detailed individually:

1. Trustworthy and Secure Access Control in Section 5.3
2. Outbound Authentication in Section 5.4
3. The Translation Mechanism in Section 5.5
4. The Security of API Requests in Section 5.7
5. Traffic Overload Protection in Section 5.8
6. Rate Limitation in Section 5.9

In addition, Section 5.6 covers how communication between devices and backends is handled.

5.1 General Architecture

This section presents the overall architecture of the system based on its route definitions. Following the APISIX route mechanism, each request passes through an ordered pipeline of plugins. This pipeline progresses from network-level concerns, such as with rate limitation (`limit-count`) and request size control (`client-control`), to consumer authorization and JWT verification (`serverless-pre-function`, `jwt-auth`), and finally to the Java plugin filters that implement the core business logic.

Figure 5.1 presents the routes related to onboarding. Each component appearing on these routes will be detailed throughout this chapter, organised by purpose, from security and access control to translation and bidirectional communication.

Although the Java filters appear as a single component in the route diagrams, they concentrate most of the system's logic: authentication, authorization, and the handling of device communication. Responses are also constructed directly within these filters.

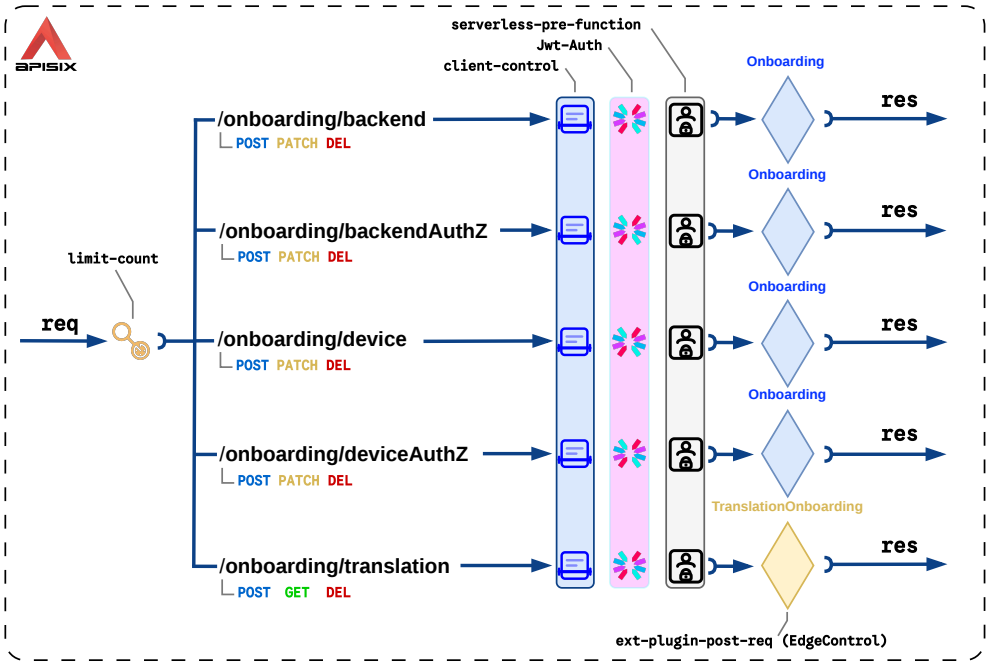

Figure 5.1: API Gateway: Onboarding routes

Figure 5.2 presents the routes used for backend ⟷ device communication. The `commands` route is a helper endpoint allowing backends to retrieve available commands. The `command` route handles command forwarding, translating backend requests into device-specific messages. Similarly, `BackendForward` enables devices to communicate with their authorised backends. All these components and their underlying principles are developed in detail throughout this chapter.

Edge Control. `EdgeControl` is the name given to the `SpringBootApplication` that loads and exposes the different filters, which are then composed onto the various routes. It contains five filters, serving distinct purposes: onboarding of devices and backends, onboarding of per-device translation configuration, authentication and command-specific communication authorization, translation for backend to device communication, and forwarding for device to backend communication.

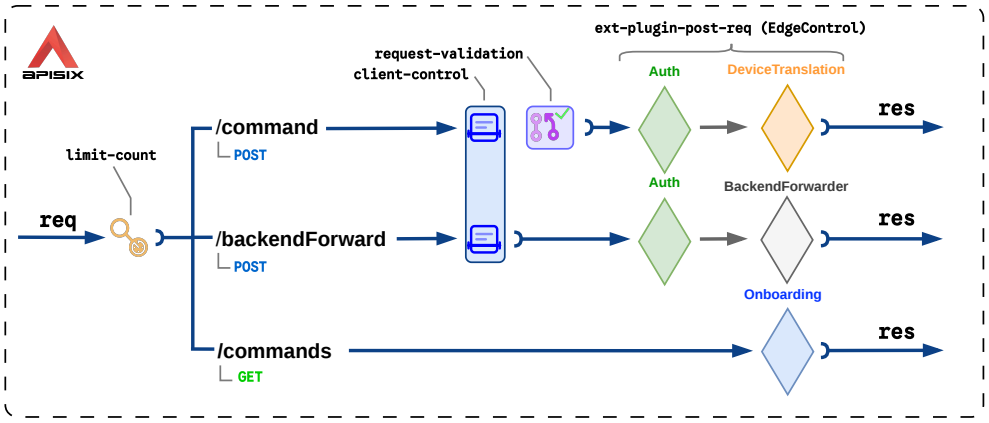

Figure 5.2: API Gateway: bi-directional communication and information routes

5.2 General Components and Principles

This section describes the internal components of `EdgeControl` that are shared across the entire system.

Spring Boot Application. `EdgeControl` runs as a Spring Boot server, making its filters available for APISIX to invoke via the Java plugin runner mechanism described in Section 4.4.1.

MongoDB. The system relies on a MongoDB database for the persistence of configuration and runtime data. MongoDB follows a database-and-collections model, where a single instance can host multiple databases, each containing collections of documents. The collections used by this system store, among other things, device configurations, translation configurations, authorization rules, and queued requests. The choice of a NoSQL document store is well suited to the heterogeneous and schema-flexible nature of device configuration.

Mosquitto. Mosquitto is an MQTT broker deployed alongside APISIX. It handles communication with external MQTT devices, acting as the message broker through which different MQTT clients can publish messages and subscribe to topics.

Handled Protocols. The system currently supports two protocols for device communication: HTTP and MQTT. Each protocol is encapsulated in a dedicated adapter, as described in Section 5.5.2.

5.2.1 Request Handler

When a request arrives at the Java plugin runner, it is processed by a chain of filters. For instance, the `/command` route shown in Figure 5.2 passes the request sequentially through the `AuthFilter` and then the `DeviceTranslationFilter`. As described in Section 4.4.2, APISIX supports short-circuit semantics between plugins, meaning that a plugin setting a response prevents the request from reaching subsequent plugins. However, this mechanism does not apply between individual filters within the Java plugin runner: all filters in the chain are always invoked. Furthermore, the filter chain is bounded by hidden APISIX-managed filters, one at the start responsible for constructing the request, and one at the end responsible for forwarding the response back to APISIX. Interrupting the chain would therefore break the plugin execution.

This is the role of the `RequestHandler`. When a filter receives a request, it registers its arrival: if no entry exists yet, the `RequestHandler` creates one in its local store and marks it as not to be skipped; otherwise it does nothing. Each filter must then call the method `shouldSkipRequest()` before executing any logic. While this will never return true for the first filter in the chain, a subsequent filter may receive a positive answer, upon which it immediately delegates to the chain without performing any processing. Such a scenario occurs when a previous filter, such as `AuthFilter`, has called `skipRequest()`, effectively halting the application-level processing while still preserving the integrity of the plugin chain.

5.2.2 Logger

Another component shared across the entire system is the custom `EdgeControl-Logger`. Although Spring Boot provides a default logging facility whose output is forwarded to the APISIX log system, these entries tend to be buried among the numerous internal APISIX log lines. To address this, a dedicated logger was introduced, writing directly to a separate log file that makes it straightforward to monitor filter behaviour in isolation. The logger is implemented as a singleton and can be easily disabled for production or performance testing environments.

5.3 Trustworthy and Secure Access Control

Direct exposure of backend services to heterogeneous field devices creates significant security risks, including device impersonation, unauthorized command execution, and data exfiltration. To address these risks, the architecture presented in Section 5.1 positions the API Gateway as the primary enforcement point for access control between cloud backends and renewable energy devices. This section presents the implementation of trustworthy and secure access control mechanisms that address these

threats through a layered approach, combining APISIX's native authentication plugins with custom authorization filters.

Subsection 5.3.1 describes the built-in APISIX plugins used for token-based authentication and coarse-grained role enforcement. Subsection 5.3.2 presents the overall filter architecture and its supporting components. Subsections 5.3.3 and 5.3.4 detail the implementation of the Onboarding and Auth filters respectively, including their integration within the route pipeline and their interaction with credential storage.

5.3.1 Built-in Plugins and Their Usage

Before any Java filter logic is invoked, two native APISIX plugins serve as the first enforcement layer for the onboarding routes: `jwt-auth` and `serverless-pre-function`.

JWT Authentication (`jwt-auth`) The `jwt-auth` plugin validates the JWT token supplied in the request's `Authorization` header against the set of registered APISIX consumers. Three consumers are defined in the system: `gateway-admin`, `device-admin`, and `backend-admin`, each provisioned with a dedicated HS256 secret during gateway initialisation, allowing them to generate their corresponding JWT token. If the token is absent, malformed, or signed with an unknown key, APISIX short-circuits the pipeline with an HTTP `401` response before the Java plugin runner is ever invoked. On success, APISIX populates the request context with the matched consumer identity (`ctx.consumer`), which is then available to the next plugin in the chain.

Role-Based Access control (`serverless-pre-function`) Immediately after `jwt-auth`, a `serverless-pre-function` Lua snippet reads `ctx.consumer.username` and checks it against a per-route allowlist. If the consumer's role is not permitted on that route, the plugin terminates the request with an HTTP `403` before any Java code runs. This constitutes a lightweight, coarse-grained role-based access control mechanism that rejects unauthorized roles at minimal computational cost.

Table 5.1 summarises which consumer roles are permitted on each onboarding route, reflecting the principle of least privilege: a device administrator cannot register new backends, and a backend administrator cannot register new devices, but both may configure cross-entity authorizations.
It is worth noting that the `jwt-auth` and `serverless-pre-function` plugins are only placed on the onboarding routes. The communication routes (`/command` and `/backendForward`) use a different authentication mechanism based on API keys, which is handled entirely within the `AuthFilter` Java plugin filter, as described in Subsection 5.3.4.

Route	gateway-admin	device-admin	backend-admin
/onboarding/backend	✓	×	✓
/onboarding/backendAuthZ	✓	✓	✓
/onboarding/device	✓	✓	×
/onboarding/deviceAuthZ	✓	✓	✓
/onboarding/translation	✓	✓	×

Table 5.1: Permitted consumer roles per onboarding route

5.3.2 Filters Architecture and Implementation Components

The access control logic of the system is implemented within `EdgeControl`, the Spring Boot application that hosts all Java plugin filters. This subsection presents the internal component architecture involved in authentication and authorization, before the individual filters are described in detail.

Figure 5.3 provides a component-level view of the classes involved in access control. The two filters, `OnboardingFilter` and `AuthFilter`, sit at the boundary between APISIX and the business logic. Each filter delegates to one or more managers, which in turn interact with the `AuthRegistry` cache and the MongoDB repositories. The `RequestHandler` and `EdgeControlLogger` components, introduced in Section 5.2, are used throughout.

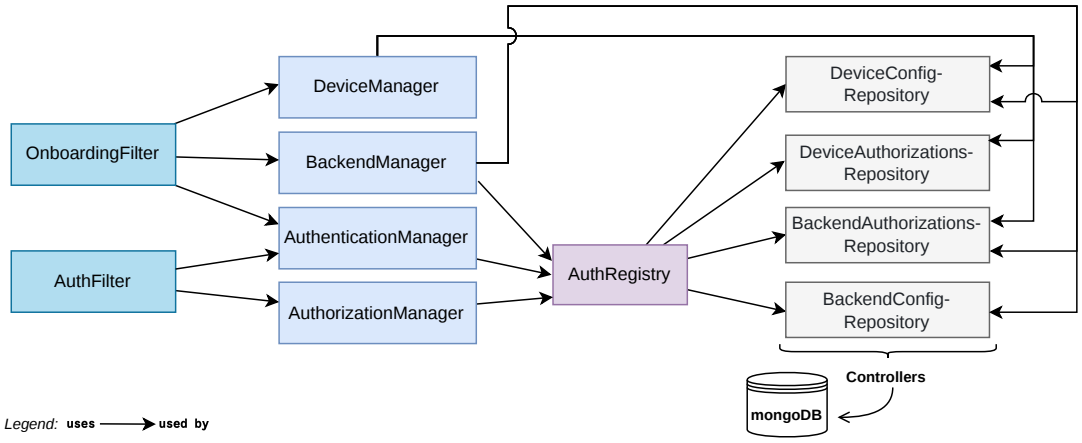

Figure 5.3: EdgeControl: access control component architecture

The key components are described hereafter.

OnboardingFilter The `OnboardingFilter` is the entry point for all onboarding routes except the `/onboarding/translation` route. It inspects the HTTP method and request path to dispatch the request to the appropriate handler method, which in turn calls either `BackendManager` or `DeviceManager`. Once a response is built, the filter sets the HTTP response body and status code directly and calls `requestHandler.skipChain()`, ensuring that no further filter in the chain processes the request.

AuthFilter The `AuthFilter` is the entry point for the `/command` and `/backendForward` routes. It delegates authentication to `AuthenticationManager`, then branches on whether the resolved identity belongs to a backend or a device, applying different authorization logic in each case. If the checks pass, the filter extends the request body with additional routing information before passing control to the next filter in the chain.

BackendManager The `BackendManager` orchestrates all backend lifecycle operations: creation, update, and deletion of communication configurations, as well as management of backend authorization entries. On creation, it coordinates with `BackendConfigRepository` to generate a unique `gatewayBackendId` and a cryptographically secure API key, storing only the key's hash in the database. It also exposes `getBackendAuthorizedCommands()`, which retrieves and enriches the authorization list with device metadata for the `/commands` helper route.

DeviceManager The `DeviceManager` is symmetric to `BackendManager` but operates on devices. A notable difference is that device creation accepts a `listOfDevices` array in the request body, enabling batch registration of multiple devices in a single call, returning a map of `deviceName` → `{gatewayDeviceId, apiKey}` for each registered device.

AuthenticationManager The `AuthenticationManager` is responsible for resolving an incoming API key to a gateway-scoped identifier. It hashes the provided key using SHA-256 and looks up the resulting hash in the `AuthRegistry` cache first. On a cache miss, it falls back to querying `BackendConfigRepository` or `DeviceConfigRepository` directly and caches the result. The resolved identifier always takes the form `backend_<UUID>` or `device_<UUID>`, allowing the caller to determine the entity type from the prefix alone.

AuthorizationManager The `AuthorizationManager` has two distinct responsibilities. `checkAuthorization()` verifies that a specific backend is permitted to issue a given command to a given device, by consulting the `backendAuthorizations` collection. The method `getDeviceEndpoints()` returns the map of `backendId` →

infoEndpoint for all backends that a given device is authorized to reach, which is used to populate the forwarding metadata for device-to-backend communication.

AuthRegistry The AuthRegistry is an in-memory cache implemented with ConcurrentHashMap, acting as a read-through layer between the managers and MongoDB. It caches the most frequently accessed data, API key resolution, authorization entries, and endpoint mappings, so that authentication and authorization checks can be served without a database round-trip on every incoming request. A ScheduledExecutorService triggers a full cache refresh every 30 seconds, ensuring eventual consistency with the database.

Repositories Four repository classes wrap the corresponding MongoDB collections: BackendConfigRepository, DeviceConfigRepository, BackendAuthorizationsRepository, and DeviceAuthorizationsRepository. Each encapsulates all CRUD operations for its collection and abstracts the database entirely from the managers above. Both configuration repositories also contain the API key generation and SHA-256 hashing logic, ensuring that plain-text keys are never persisted to the database.

5.3.3 Onboarding Filter

Objective and Design The OnboardingFilter handles the full lifecycle management of backends and devices within the gateway: registration, configuration updates, deletion, and authorization policy management. Its operation is split across five specialized routes, each exposed for a distinct purpose, all sharing the same underlying filter class: /onboarding/backend, /onboarding/backendAuthZ, /onboarding/device, /onboarding/deviceAuthZ, and the read-only /commands route. The /onboarding/translation route, although part of the onboarding mechanism, is handled by a dedicated filter and described separately in Section 5.5.

During registration, both backends and devices receive two pieces of identity information generated by the gateway: a unique identifier prefixed with backend_ or device_ followed by a UUID, and a cryptographically secure 256-bit API key generated using Java's SecureRandom. Only the SHA-256 hash of this key is stored in MongoDB while the plain-text key is returned to the caller exactly once in the registration response and is never persisted. This design ensures that a full compromise of the database does not expose credentials that could be used to authenticate against the gateway.

Authorization configuration is managed independently from devices and backends configuration through dedicated routes. This separation allows the communication

details of an entity to evolve, for example, a backend's endpoint being updated, without affecting its authorization policies, and vice versa.

A cross-collection cascade is implemented on deletion: removing a backend automatically removes that backend's identifier from the `listOfAuthorizations` of all devices in the `deviceAuthorizations` collection, and removing a device similarly cleans up all backend authorization entries that referenced it. This preserves referential integrity across collections without the use of a relational database.

The `/commands` Route In addition to the lifecycle routes, the `OnboardingFilter` also handles the `/commands` GET route, which allows an authenticated backend to retrieve a structured list of all devices it is authorized to send commands to, together with the full command definitions for each. The filter first authenticates the caller via `AuthenticationManager` using the `apikey` header, rejects the request if the API key is invalid or belongs to a device rather than a backend, and then delegates to `BackendManager.getBackendAuthorizedCommands()`. This method fetches the backend's authorization document, extends each entry with device metadata (device name, command parameter definitions) sourced from `DeviceConfigRepository`, and returns the combined response. This route serves as a discovery mechanism: a backend can use it to determine what commands it may use and what parameters each command requires before interacting with the `/command` route.

Route and API Overview The complete API exposed by the `OnboardingFilter` across its routes, including request bodies, response codes, and manager references, is documented in Appendix A.

5.3.4 Auth Filter

Objective and Design The `AuthFilter` serves as the runtime enforcement point for the two communication routes: `/command` and `/backendForward`. Unlike the onboarding routes, which rely on JWT tokens issued to administrative consumers, these routes are used at operation time by backends and devices, which authenticate using the API keys issued during their onboarding. The filter combines authentication and authorization in a single pass, and extends the request body with routing metadata before passing control to the next filter in the chain.

As shown in Figure 5.2, the `AuthFilter` is the first filter in the `ext-plugin-post-req` phase on both the `/command` and `/backendForward` routes, executed after the built-in Lua plugins have already enforced rate limits and payload validation.

Authentication For both routes, the filter begins by reading the `apikey` header and delegating to `AuthenticationManager.checkAuthentication()`. This method ha-

shes the provided key using SHA-256 and looks up the hash in the `AuthRegistry` cache. On a cache miss, it queries the database directly and caches the result for subsequent requests. The resolved identifier, either `backend_<UUID>` or `device_<UUID>`, determines the subsequent authorization path. If the key is absent or invalid, the filter responds immediately with an HTTP 403 and marks the request as to be skipped by all downstream filters via `requestHandler.skipChain()`.

Authorization: Backend Path When the resolved identifier is a backend, the filter proceeds to an authorization check via `AuthorizationManager.checkAuthorization()`. This method consults the `backendAuthorizations` collection, verifying that the resolved `gatewayBackendId` has an entry that grants it permission to issue the specific `command` field value to the specific `gatewayDeviceId` field value present in the request body. Both fields are mandatory as their absence is treated as an authorization failure. If the check passes, the filter injects the resolved `gatewayBackendId` and the corresponding `callbackEndpoint` directly into the request body, so that downstream filters can identify the caller without re-authenticating. Under normal operation, the device response is returned directly through the original request-response cycle. The `callbackEndpoint` is only used by the queuing mechanism: if the target device is unreachable and the request is queued for later retry, a successful dequeued delivery will forward the device response asynchronously to the backend's `callbackEndpoint`. If it fails, the filter responds with HTTP 403 and skips the chain.

This per-command, per-device authorization model ensures that a backend credential grants only the precise access it was configured for: possession of a valid API key alone does not entitle a backend to issue arbitrary commands to arbitrary devices.

Authorization: Device Path When the resolved identifier is a device, the filter applies a different logic. Rather than checking per-command permissions, it calls `AuthorizationManager.getDeviceEndpoints()`, which returns the map of {`gatewayBackendId` → `infoEndpoint`} for all backends the device is authorized to reach. This map is injected into the request body under the key `listOfEndpoints`, alongside the resolved `gatewayDeviceId`.

The asymmetry between the backend and device authorization models is intentional. A backend issuing commands to a device requires fine-grained, per-command authorization, as commands may have operational or safety implications. A device forwarding its own telemetry or information to its backends requires only coarser-grained authorization. The gateway ensures it can only reach backends it was explicitly associated with, but does not restrict the content of what it forwards. The downstream `BackendForwarder` filter uses the injected `listOfEndpoints` map to dispatch the device's message to each authorized backend.

5.4 Outbound Authentication

When the API gateway communicates with backends and devices to transfer the request send by another backends or devices, it must itself authenticate to those entities. This outbound authentication requirement is distinct from inbound authentication, where backends and devices authenticate to the gateway. To address this, a dedicated subsystem was designed to securely store, retrieve, and inject access tokens into outbound HTTP requests. The current implementation focuses on the access token pattern, where a static bearer token is obtained and injected into the request header of HTTP request.

The subsystem is composed of four components, illustrated in Figure 5.4.

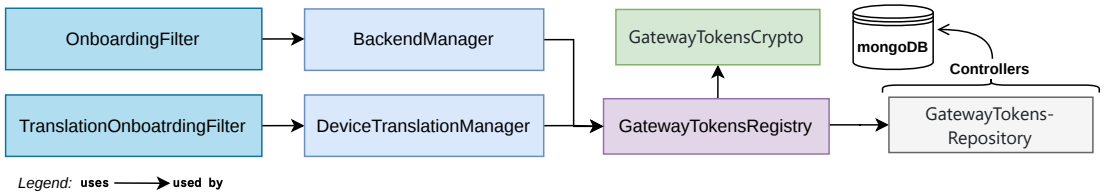

Figure 5.4: Outbound authetication architecture

GatewayTokensRepository is the persistence layer. It is responsible for storing and retrieving token documents in the `gatewayTokens` MongoDB collection. The repository operates exclusively on already-encrypted data. It stores and returns tokens as-is, with no knowledge of the encryption details. Each document stores the gateway identifier, the token type, the encrypted token value, and an expiration date.

GatewayTokensCrypto handles all cryptographic operations. It encrypts and decrypts tokens using AES-256-GCM, a symmetric authenticated encryption scheme that provides both confidentiality and integrity guarantees. A fresh random initialisation vector (IV) is generated for every encryption operation, ensuring that identical tokens produce different ciphertexts. The 128-bit GCM authentication tag further ensures that any tampering with a stored token is detected at decryption time. The AES-256 key is never embedded in the codebase or stored in the database. Instead, it is loaded at startup from a Docker secret mounted at `/run/secrets/token_encryption_key`, keeping it isolated from the application and the data it protects.

GatewayTokensRegistry acts as the central coordinator of the subsystem. It holds a reference to both `GatewayTokensCrypto` and `GatewayTokensRepository`, and orchestrates all token operations. On `upsertToken`, it validates the incoming metadata, encrypts the plaintext token via `GatewayTokensCrypto`, persists the encrypted token through the repository, and stores the encrypted form in a thread-safe `ConcurrentHashMap` cache. On `getDecryptedTokenByGatewayId`, it checks the

cache first. On a hit, it decrypts the cached encrypted token and returns it. On a miss, it fetches the encrypted token from the repository, populates the cache, and then decrypts before returning. This design ensures that plaintext tokens are never held in memory longer than necessary, and that decryption is always performed on demand rather than at storage time.

Filters and Managers serve as the entry point for token lifecycle management. Token creation is triggered during the backend configuration and device translation configuration onboarding process. When a backend is registered through the `OnboardingFilter`, a `security` field in the request body may optionally carry a token configuration. The filter delegates to `BackendManager`, which extracts this field and passes it to the `GatewayTokensRegistry`, which in turn encrypts and persists the token. The same mechanism applies to device translation configuration onboarding, where `TranslationOnboardingFilter` delegates to its associated manager. This design keeps token provisioning integrated into the existing onboarding flow without requiring a separate endpoint.

5.5 Protocol Translation and Device Communication

This section describes the translation mechanism through which the API gateway transforms backend requests into device-specific messages. The mechanism is first presented at the architectural level, followed by a description of the translation configuration data model that underpins the entire section. The individual components are then detailed: the device adapters in Section 5.5.2, the translation and cleanup engine in Section 5.5.3, and the queuing mechanism in Section 5.5.4. Finally, the two filters that drive the mechanism at runtime are covered: the `TranslationOnboardingFilter` in Section 5.5.5 and the `DeviceTranslationFilter` in Section 5.5.6. The full API specifications for routes on which these filters are involved are provided in Appendix A.4 and Appendix A.5.

Architectural overview. Figure 5.5 provides a high-level view of the components involved in the translation mechanism. Two distinct perspectives are useful for understanding how these components interact.

From a data perspective, the `DeviceRegistry` is the central element. It manages the hot-loading of adapters, the initialisation of the `QueueRegistry` at startup, the onboarding of new queuing configurations, and the lifecycle of device translation configurations through the `DevicesTranslationConfigRepository` database controller.

From a control perspective, all activity is initiated from the filters. The `TranslationOnboardingFilter`, handling the `/onboarding/translation` route, delegates creation, retrieval, and deletion of translation configurations to the `DeviceTranslation-`

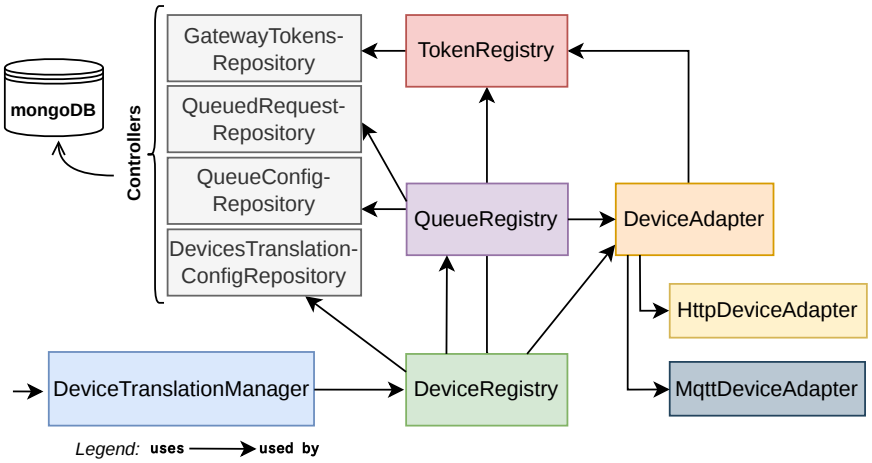

Figure 5.5: Translation mechanism: architectural overview

`Manager`. The `DeviceTranslationFilter`, on its side, requests the appropriate adapter from the `DeviceTranslationManager` and instructs it to handle the incoming request. Depending on the outcome, the filter either returns a response directly to the backend, or delegates to the `QueueRegistry` in the event that the target device is unreachable.

5.5.1 Translation Configuration Data Model

The translation configuration is the central data structure of the translation mechanism. It is registered per device during onboarding and governs how every subsequent command request for that device is processed. It consists of four main elements.

`adapter`. Specifies the protocol used to communicate with the device. The currently supported values are `http` and `mqtt`. This field determines which `DeviceAdapter` implementation is instantiated, as well as the expected structure of the rest of the configuration.

`queuing`. This optional field configures the queuing behaviour for the device, as described in detail in Section 5.5.4. It specifies the retry interval and the maximum time-to-live of a failed request.

Command definitions. A device exposes one or more named commands, each representing a distinct operation it can perform. A command carries adapter-specific routing information (an HTTP endpoint and method, or an MQTT topic and response topic), optional timeout settings, and the translation specification as described below.

HTTP and MQTT commands share the same translation specification format, differing only in their routing fields.

Translation specification. The translation specification is common to all adapter types and defines how a backend request is transformed into a device-specific payload. It consists of three components:

1. **Payload template.** The payload template defines the target JSON structure of the final device payload. It may contain fixed values, nested objects, and `null` fields that serve as placeholders for parameters provided at request time.

```
"payloadTemplate": {
    "<fixed_field>": <fixed_value>,
    "<parameter_field>": null,
    "<nested_field>": {
        "<sub_field>": null
    },
    ...
}
```

2. **Mappings.** The mappings associate each backend parameter name to a dot-notation path within the payload template, indicating where the corresponding value should be placed by the translation engine.

```
"mappings": {
    "<param_name>": "<path_in_payload_template>",
    ...
}
```

3. **Cleanup rules.** After the template has been filled, cleanup rules determine how unfilled or empty fields are handled before the payload is sent to the device. The available rules are `removeNulls`, `removeEmpty`, and `emptyObjectToNull`, and are described in detail in Section 5.5.3.

```
"cleanup": {
    "removeNulls": <true|false>,
    "removeEmpty": <true|false>,
    "emptyObjectToNull": <true|false>
}
```

Additionally, MQTT adapters require two extra top-level fields. The `connection` field configures the MQTT client options, including Quality of Service level, keep-alive interval, connection timeout, session persistence, and reconnect delay. The `subscriptions` field defines the topics the MQTT client subscribes to, and for each

topic, whether received messages should be forwarded to the authorised backends. A MQTT and HTTP configuration creation can be found in Appendix A.4

5.5.2 Device Adapters

The `DeviceAdapter` interface defines the contract that any device adapter must fulfil, and serves as the foundation for polymorphic adapter management. Three methods are defined in the interface:

1. `void init(DeviceConfig deviceConfig)`: called when a new adapter is instantiated during translation onboarding. It is used to initialise protocol-specific clients, such as an MQTT client, and to hot-load the command translation configuration for that device.

2. `void handleRequest(HttpRequest httpRequest, HttpResponse httpResponse, AdapterCallback adapterCallback)`: invoked by the `DeviceTranslationFilter` when a command request is received. The filter retrieves the adapter corresponding to the target device via the `DeviceTranslationManager` and the `DeviceRegistry`, then calls this method with the original request and response objects. The implementation is responsible for retrieving the relevant command configuration, invoking the `TranslationEngine` to produce the final device payload, and managing the protocol-level communication with the device. The result is communicated back to the filter via the `AdapterCallback`, described further below.

3. `void shutdown()`: called when an adapter is removed from the system. It allows the adapter to release its resources, such as closing protocol clients and clearing internal configuration state.

Figure 5.6 illustrates the component structure of the adapter layer. When the `TranslationOnboardingFilter` receives a new configuration, it requests the creation of the corresponding adapter from the `DeviceTranslationManager`. The `AdapterFactory`, accessed through the `DeviceRegistry`, instantiates the appropriate implementation, either `HttpDeviceAdapter` or `MqttDeviceAdapter`, and triggers its `init()` method.

Extensibility. The `DeviceAdapter` interface is designed for long-term extensibility. Any new protocol can be supported by implementing the interface and registering the adapter type with the `AdapterFactory`. One could envision a future extension allowing dynamically loaded `.jar` files to contribute new adapter implementations at runtime, without modifying the core system.

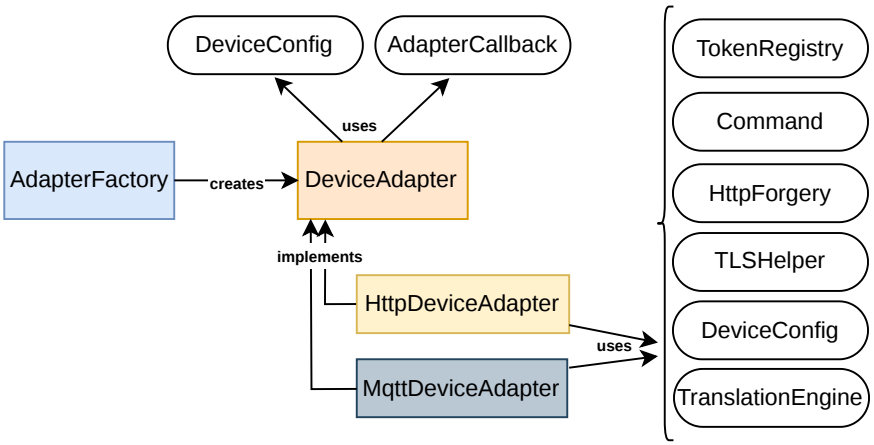

Figure 5.6: Device adapters: component architecture

`DeviceConfig`. `DeviceConfig` is the data class representing the full translation configuration of a specific device, as described in Section 5.5.1. It flows between the adapter, the database, and the registry, and serves as the input to `init()`.

`TLSHelper`. `TLSHelper` is a utility class used by both `HttpDeviceAdapter` and `MqttDeviceAdapter` to configure TLS for their respective protocol clients.

`AdapterFactory`. Given an adapter type identifier, `AdapterFactory` instantiates and returns the corresponding `DeviceAdapter` implementation. It is the single point of adapter creation in the system.

`HttpForgery`. `HttpForgery` is a utility class used throughout the project to construct `HttpRequest` objects given a method, endpoint, body, and headers. It makes use of shared `HttpClients` where possible for connection reuse.

`TranslationEngine` and `Command`. The `TranslationEngine` and `Command` instances are central to the operation of the adapters. `Command` instances are created during `init()`, and subsequently used by the engine at request time to transform the backend parameters into a device-specific payload. The `TranslationEngine` only depends on the `Command` interface, making it reusable by any adapter capable of expressing its commands through that interface. These components are described in depth in Section 5.5.3.

`TokenRegistry` The `TokenRegistry` is used as a store from which to retrieve the token information of a device for outbound communication. Currently, the `Http-`

`DeviceAdapter` is the only one to make use of it, by adding the retrieved access token in the header of the HTTP request.

`AdapterCallback`. `AdapterCallback` is a callback interface providing two methods:

1. `onSuccess()`: signals that the command was successfully delivered to the device, and that a response was provided.
2. `onDeviceUnreachable(String reason)`: signals that the device could not be reached, providing a reason string. Upon receiving this callback, the `DeviceTranslationFilter` either enqueues the request in the queuing mechanism (if a queuing configuration exists for the device) or returns a `502` response to the backend.

This callback mechanism serves two purposes. First, it avoids blocking the event loop thread: by deferring the continuation of logic to the callback implementation, the design follows the threading model recommended by the Java plugin runner [65]. Second, it provides the `DeviceTranslationFilter` with a clean entry point for triggering the queuing mechanism on failure.

5.5.3 Translation and Cleanup Engine

The `TranslationEngine` is responsible for producing the final payload that is sent to a device. It operates purely on the data level: given a payload template, a set of mappings, and the parameters provided by the backend, it assembles the final device-specific payload and applies cleanup rules. The translation specification it relies on is defined during onboarding, as described in Section 5.5.1.

Translation mechanism. Consider a device command whose payload template defines one fixed field and three nullable placeholders at different nesting levels:

```
"payloadTemplate": {
    "type": "commandA",
    "config": { "paramX": null, "paramY": [null] },
    "target": null
}
```

The associated mappings specify where each backend parameter should be placed:

```
"mappings": {
    "x": "config.paramX",
    "y": "config.paramY[0]",
    "t": "target"
```

```
}
```

The backend submits only the parameters it wishes to set, as some may be optional:

```
"params": { "y": 42,"t": "unit_1" }
```

The `TranslationEngine` fills the template by following each compiled mapping path and setting the corresponding value. It then applies the cleanup rules. Assuming `removeNulls` is enabled, the resulting payload is:

```
{
    "type": "commandA",
    "config": { "paramY": [42] },
    "target": "unit_1"
}
```

The unprovided parameter `x` remained `null` and was removed, following the configured cleanup rules. The fixed field `type` was left untouched.

Commands. A `Command` represents the translation specification for a single operation that a device can perform. It encapsulates all the information the `TranslationEngine` needs to translate a backend request into a device-specific payload: the payload template, the mappings, and the cleanup rules. Each device can expose multiple commands, and the adapter selects the appropriate one at request time based on the command name provided by the backend. Figure 5.7 illustrates the command component architecture. The `Command` interface defines the methods required by the `TranslationEngine`: `name()`, which returns the command identifier, `createPayloadInstance()` which returns a copy of the payload template, and `getCompiledMappings()`, which returns a `HashMap` mapping each parameter name to a `CompiledPath`. A command also carries cleanup rules, and adapter-specific metadata such as an HTTP endpoint or an MQTT topic. The `TranslationEngine` relies solely on the `Command` interface, making it agnostic to the underlying protocol. Two concrete implementations exist: `HttpCommandDefinition` and `MqttCommandDefinition`.

`PathCompiler`, `CompiledPath`, and `PathSegment`. For performance reasons, mapping paths are parsed once at adapter initialisation rather than at each request. A `PathSegment` represents either a named field access (e.g. `config`) or an indexed list access (e.g. `items[0]`). A `CompiledPath` is an ordered list of `PathSegment` instances representing a full mapping path such as `config.paramX`. It exposes an `apply()` method that, given a value and a payload object, traverses the structure and sets the value at the correct location. The `PathCompiler` transforms a mapping path string into a `CompiledPath` at command creation time, ensuring that subsequent request processing only requires fast object traversal rather than repeated string parsing.

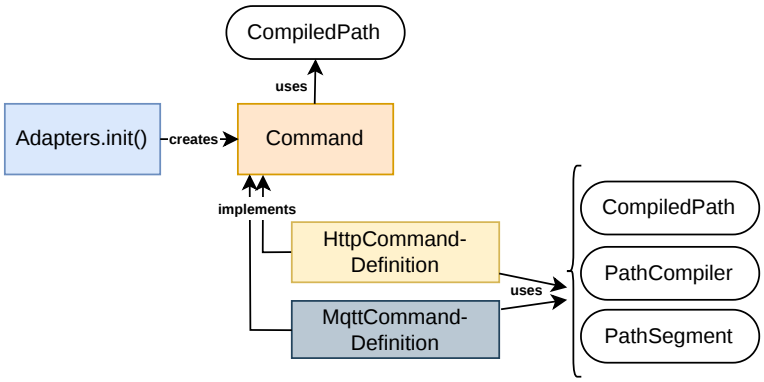

Figure 5.7: Commands: component architecture

Cleanup engine. The cleanup engine post-processes the filled payload according to three configurable rules: `removeNulls` removes fields whose value is `null`, `removeEmpty` removes empty objects and arrays, and `emptyObjectToNull` replaces empty objects with `null` before any null removal. These rules give full control over the shape of the final payload sent to the device.

5.5.4 Queuing Mechanism

Renewable energy assets are not always continuously reachable. A backend may need to send a command at a specific moment without knowing whether the target device is currently available. The queuing mechanism addresses this need by allowing failed command requests to be retained and retried until the device becomes reachable again, or until a maximum time-to-live is exceeded.

As illustrated in Figure 5.8, when a command request is received by the `Device-TranslationFilter`, the adapter attempts to deliver it to the device. If delivery succeeds, the device response is forwarded directly to the backend. If the device is unreachable and a queuing configuration exists for that device, the request is enqueued and the backend receives a 202 response indicating that the request has been accepted for retry. Once a queued request is eventually delivered successfully, the `QueueWorker` forwards the device response to the backend via its registered callback URL.

The queuing behaviour is configured per device through two parameters in the translation configuration:

- `retryIntervalSeconds`: the period between successive retry attempts.
- `maxTimeToLiveSeconds`: the maximum duration a request may remain in the queue. Once exceeded, the request is discarded and the backend is notified via

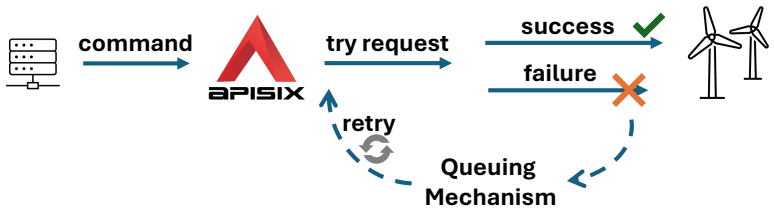

Figure 5.8: Queuing mechanism: request flow

its callback endpoint.

Figure 5.9 shows the component architecture of the queuing mechanism.

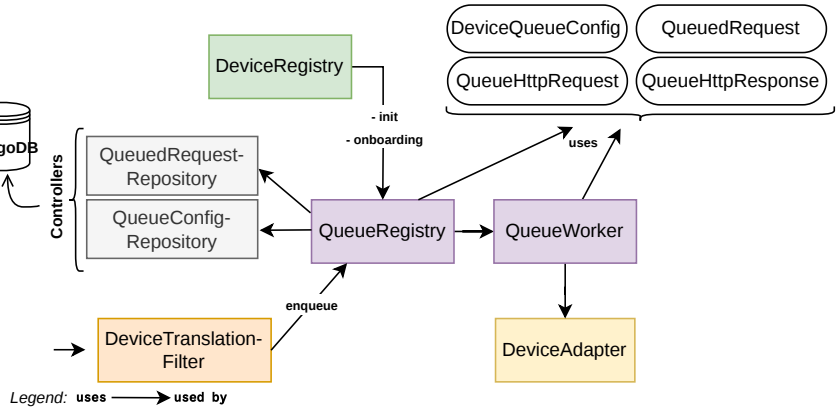

Figure 5.9: Queuing mechanism: component architecture

`DeviceRegistry`. The `DeviceRegistry` manages the lifecycle of translation configurations. As part of this role, it initialises the `QueueRegistry` at system startup and registers queuing configurations for newly onboarded devices.

`DeviceTranslationFilter`. When a device is unreachable, as signalled through the `AdapterCallback` (Section 5.5.2), the `DeviceTranslationFilter` submits the request to the `QueueRegistry` for deferred retry.

`QueuedRequest` and `DeviceQueueConfig`. `QueuedRequest` represents a request currently held in the queue. Each instance holds a reference to the synthetic request and response objects described below, and is synchronised with the database via the `QueuedRequestRepository` controller. Persisting queued requests to the database

ensures they survive a gateway restart, provided their time-to-live has not yet expired. `DeviceQueueConfig` holds the queuing parameters for a specific device and is similarly persisted via the `QueueConfigRepository`.

`QueueWorker`. The `QueueWorker` periodically iterates over the queued requests and attempts redelivery at the configured retry interval. For each request, it first verifies that the time-to-live has not been exceeded. If it has, the request is removed and the backend is notified via its callback endpoint. If delivery succeeds, the device response is forwarded to the same callback endpoint.

`QueueHttpRequest` and `QueueHttpResponse`. The `handleRequest()` method of the `DeviceAdapter` expects `HttpRequest` and `HttpResponse` objects, which are constructed by the APISIX-managed filter at the head of the plugin chain and cannot be instantiated directly. `QueueHttpRequest` and `QueueHttpResponse` are synthetic subclasses of these types that can be constructed by the `QueueWorker` from the persisted request data. They allow queued requests to be replayed through the adapter layer without modification to the adapter interface.

5.5.5 Translation Onboarding Filter

The `TranslationOnboardingFilter`, placed on the route `/onboarding/translation` (Figure 5.1), manages the lifecycle of device translation configurations. It handles the creation, retrieval, and deletion of configurations, and triggers the initialisation or teardown of the corresponding device adapters.

On creation (`POST`), the filter receives the complete translation configuration for a device, as described in Section 5.5.1, and delegates adapter instantiation to the `DeviceTranslationManager`. On retrieval (`GET`), the stored configuration for a given device is returned. On deletion (`DEL`), the configuration is removed and the corresponding adapter is shut down. As with all onboarding routes, requests are authorized via the consumer mechanism described in Section 5.3.1. The full API specification is provided in Appendix A.4.

5.5.6 Device Translation Filter

The `DeviceTranslationFilter` is placed on the `/command` route (Figure 5.2), immediately after the `AuthFilter` in the filter chain. All requests reaching it are therefore already authorised, following the `RequestHandler` mechanism described in Section 5.2.1.

Upon receiving a command request, the filter retrieves the adapter corresponding to the target device via the `DeviceTranslationManager`, invokes `handleRequest()`,

and handles the outcome through the AdapterCallback: a successful delivery results in the device response being forwarded to the backend, while an unreachable device triggers the queuing mechanism if a queuing configuration exists for that device.

Referring back to Figure 5.2, the /command route passes through client-control and request-validation before reaching the Java plugin filters. The request-validation plugin enforces the presence of required body fields upstream of EdgeControl, avoiding unnecessary load on the Java plugin runner. The AuthFilter then authenticate then verifies authorisation and attaches the callback endpoint of the requesting backend to the request context, as described in Section 5.3.4. The DeviceTranslationFilter subsequently retrieves the target adapter using the gatewayDeviceId, invokes handleRequest(), and processes the result. A successful delivery yields a 200 response carrying the device response. If the device is unreachable and a queuing configuration exists, the request is enqueued along with the backend callback endpoint, and a 202 is returned to the backend carrying the queuedRequestId for tracking. If no queuing configuration is defined, a 502 is returned immediately. The full API specification is provided in Appendix A.5.

Parameter validation. The TranslationEngine and the adapters do not perform type or value validation on the parameters provided by the backend. Parameter validation is assumed to be the responsibility of the backend or the calling client.

5.6 Device-to-Backend Communication

While backends are expected to send commands to devices, devices are equally able to push data towards backends. Such data may include telemetry readings, device status updates, error notifications, and similar asynchronous messages. Supporting this direction of communication is therefore an important aspect of this contribution.

To this end, the API gateway exposes a dedicated /backendForward route, whose sole purpose is to forward packets from devices to their authorised backends. Devices themselves have no knowledge of which backend their data is sent to: backends remain fully isolated from the device layer, improving their security posture. The API route definition is available in Appendix A.6

Following the filter pipeline shown in Figure 5.2, a request arriving on /backendForward first passes through the AuthFilter, which, based on the provided apikey header, automatically resolves the set of backend /info endpoints to which the device is authorised to communicate, and attaches that information to the request body (as described in Section 5.3.4). The request is then forwarded to the BackendForwarderFilter, which retrieves those endpoints and issues a POST request to each of them. A fire-and-forget strategy is applied: neither backend availability nor response content is considered, as the forwarded data is not expected to require guaranteed delivery

or special treatment. For each backend, the `BackendForwarder` filter also ensures to use the corresponding access token to the request header using the `TokenRegistry`.

Figure 5.10 illustrates the communication flow for an HTTP device. In this scenario, the device communicates directly with the gateway by posting to the `/backend-Forward` endpoint.

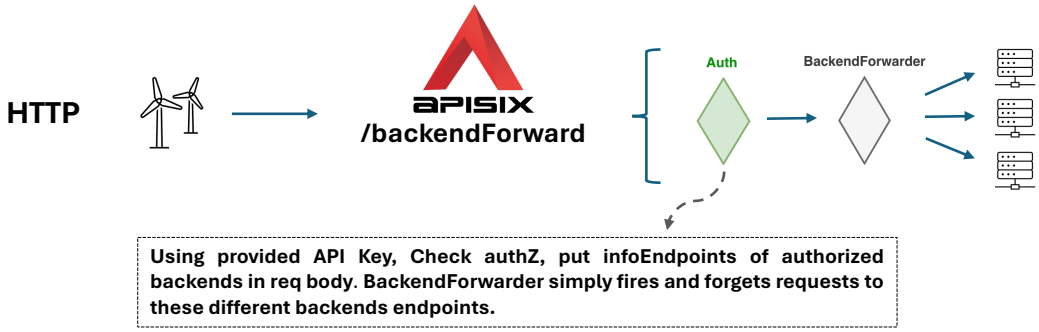

Figure 5.10: HTTP device to backends: communication flow

Figure 5.11 illustrates a more complex flow arising from the MQTT protocol. Messages received on subscribed topics are delivered directly to the `MqttDeviceAdapter`. However, adapters are not responsible for authorization: determining to which backends a device is authorised to forward data falls outside their scope. To preserve this separation of concerns, the adapter constructs a new HTTP request internally and submits it to the `/backendForward` route, delegating authorization and forwarding to the existing filter pipeline. Any future protocol adapter receiving messages directly through a client is expected to follow the same pattern, using the `HttpForgery` utility class for request construction.

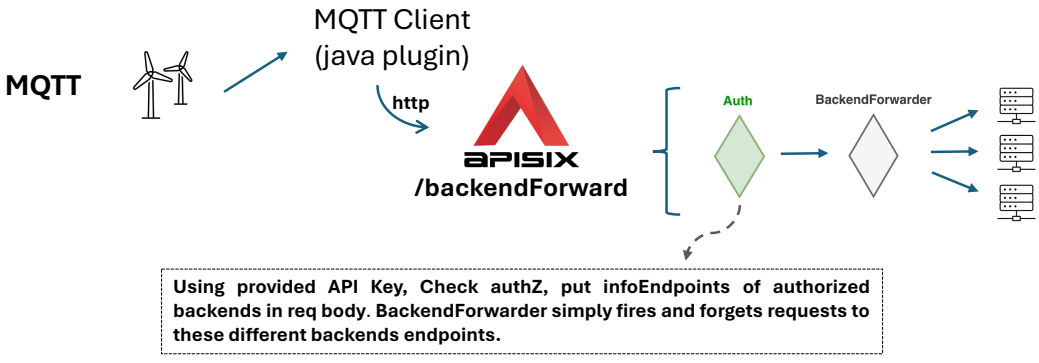

Figure 5.11: MQTT device to backends: communication flow

This mechanism enables fully bidirectional communication between devices and backends, while preserving strict isolation between components. Devices have no

knowledge of which backends receive their data, and backends have no knowledge of which device their commands are ultimately forwarded to. The gateway acts as the sole arbiter of these relationships, routing traffic exclusively according to the authorization configuration established by the administrator of the system.

5.7 Security of API request/call

Beyond access control, ensuring the confidentiality and integrity of data in transit is a fundamental requirement of the proposed architecture. Communications between backends, the API Gateway, and devices traverse networks that may be untrusted or publicly accessible, exposing them to eavesdropping, man-in-the-middle attacks, and message tampering. This section describes how Transport Layer Security (TLS) is enforced across all communication channels in the system.

5.7.1 General Approach

The API Gateway acts as the central TLS termination point for all inbound and outbound communications. Two protocols are secured: HTTPS for communications between backends and the gateway, and between the gateway and HTTP devices; and MQTTS for MQTT-based communication between devices and the Mosquitto broker.

For HTTPS communication, certificate-based authentication is enforced in both directions across two separate connections. When a backend or device initiates a connection to the gateway, the gateway acts as the TLS server and presents its certificate for the client to verify. When the gateway initiates a connection to a backend or HTTP device, the roles are reversed: the remote party acts as the TLS server and presents its certificate for the gateway to verify. In both connections, the server's identity is verified by the client, and since each party acts as a server in one of the two connections, both identities are verified across the full communication, achieving bidirectional authentication.

For MQTT communication from devices to the Mosquitto broker, one-way TLS is used: the broker acts as the TLS server and presents the gateway certificate to the MQTT device, which the device verifies before publishing.

5.7.2 Certificate Infrastructure

The system uses a self-signed certificate authority model suited to the controlled evaluation environment used in this work. The gateway generates a self-signed certificate with the `CA:TRUE` extension, making it act as its own certificate authority. This certificate is distributed to all communicating parties, backends and devices, which add it to their system trust store, enabling them to verify the gateway's identity when it acts

as a TLS server. Since backends and HTTP devices also act as TLS servers when the gateway initiates a connection to them, each generates its own self-signed certificate identifying itself by hostname and IP address, using a 2048-bit RSA key with a Subject Alternative Name extension covering both DNS name and IP address. The gateway, in turn, retrieves and trusts these certificates so it can verify their identity during the handshake.

The gateway certificate is registered in APISIX via its Admin API, bound to the gateway's hostname via Server Name Indication (SNI). APISIX then presents this certificate on port 9443 for HTTPS and the same certificate is used by the Mosquitto broker on port 8883 for MQTTS. Hostname resolution is configured on each machine (backends and devices) so that the gateway's domain name resolves to its IP address, enabling proper certificate validation against the SNI.

It is important to note that this self-signed setup introduces operational overhead that would not exist in a production deployment. In a real-world deployment, the gateway certificate would be issued by a recognised Certificate Authority or an internal enterprise CA for private network deployments. In that case, communicating parties would automatically be able to verify the certificate's validity as it was signed by an already known CA, eliminating the need to distribute the gateway certificate and configure per-host trust manually. The self-signed model is used here only because the evaluation environment does not have access to a trusted CA, and the steps described above automate the distribution and trust configuration that a real CA would handle transparently.

5.7.3 Scope of TLS Coverage

Table 5.2 summarises the TLS coverage across all communication channels.

Channel	**Protocol**	**Port**	**TLS mode**
Device/Backend → Gateway	HTTPS	9443	Certificate-based auth (both directions)
Gateway → Device/Backend	HTTPS	9443	Certificate-based auth (both directions)
Device → Broker	MQTTS	8883	One-way TLS (server-side only)

Table 5.2: TLS coverage per communication channel

5.8 Traffic overload protection

This section describes the two complementary protection layers implemented at the network and application levels. It is worth noting that the rate limitation mechanisms described in Section 5.9, namely the global request quota enforced by the `limit-count` plugin and the per-route payload size limits enforced by `client-control` also constitute techniques for traffic overload protection. They are presented separately in their own section, given their distinct configuration model.

5.8.1 Network-Level Protection: `nftables` Firewall

The first protection layer operates at the network level, independently of APISIX, using `nftables`, the Linux kernel packet filtering framework. A dedicated table named `apisix_guard` is created with a `FORWARD` chain at priority −10, which ensures it runs before Docker's own forwarding rules (priority 0) and therefore intercepts all traffic directed at the gateway before it reaches any container.

Dynamic IP Blacklist The core mechanism is a dynamic set named `blacklist`, declared with a 10-minute automatic expiry. When a source IP exceeds the allowed rate on any monitored port, its address is automatically added to this set. All subsequent packets from blacklisted sources are dropped at the top of the chain before any per-port rule is evaluated, eliminating processing overhead for known attackers. Entries expire automatically after 10 minutes, allowing legitimate sources that triggered the threshold transiently to recover without manual intervention.

Per-Protocol Rate Meters Traffic is metered independently per protocol and port using `nftables` stateful meters, which maintain per-source-IP rate counters in kernel space. A source IP is blacklisted as soon as its rate exceeds the configured threshold for that protocol. Table 5.3 summarises the monitored ports and their respective thresholds.

Protocol	Port	Threshold	Connection state
HTTP	9080	100 req/s	`new`
HTTPS	9443	100 req/s	`new`
MQTT	1883	35 req/s	`new, established`
MQTT WebSocket	9001	35 req/s	`new, established`
MQTTS	8883	35 req/s	`new, established`

Table 5.3: `nftables` flood protection thresholds per protocol

The MQTT threshold is intentionally lower than the HTTP threshold (35 versus 100 per second) to account for the persistent-connection nature of MQTT: a single device maintaining a legitimate connection should not approach this rate, so a lower threshold provides tighter protection against abusive publishers without affecting normal operation. HTTP and HTTPS meters only count `new` connections, whereas MQTT meters also apply to `established` connections to capture high-frequency publish flooding from already-connected clients.

Configurability The firewall rules are applied by running the `nftables-apisix.sh` script on the gateway VM. An administrator can modify the rate thresholds, blacklist expiry duration, or monitored ports by editing the script and re-running it. The table can be fully reset with a single `nft delete table ip apisix_guard` command, after which the script can be re-applied with updated parameters. The thresholds presented in Table 5.3 are defaults that should be adjusted by the administrator based on the protocols used, the number of devices, and the expected traffic patterns of the specific deployment.

5.8.2 Application-Level Protection: Payload Enforcement

The second protection layer operates within APISIX at the application level. Two plugins contribute here: `client-control` enforces a maximum request body size for all the routes, and `request-validation` enforces a JSON schema on the `/command` route. Both reject non-conforming requests before they reach the Java plugin runner, preventing malformed or intentionally oversized payloads from consuming processing resources. As noted above, `client-control` is also discussed in Section 5.9.2 from a rate limitation perspective. Its per-route body size limits are presented in full there.

The `request-validation` plugin on `/command` enforces that the request body is a JSON object containing three required fields: `gatewayDeviceId` (string), `command` (string), and `params` (object). Requests that do not conform to this schema are rejected with HTTP `400` before reaching the `AuthFilter`, preventing malformed requests from consuming authentication and authorization processing.

Configurability Both plugins are defined in the APISIX route configuration and applied via the Admin API at gateway startup. An administrator can update their parameters by modifying the corresponding route definitions in the setup script and re-running it, or by issuing a `PUT` request directly to the APISIX Admin API at runtime without restarting the gateway.

5.9 Rate limitation

Rate limitation enforces usage constraints to prevent abuse, excessive polling, and brute-force attempts while ensuring fair resource allocation across all connected entities. Beyond their role as rate limitation mechanisms, both plugins presented in this section also serve as techniques for traffic overload protection, complementing the network-level firewall described in Section 5.8. Two built-in APISIX plugins are used: `limit-count` for fixed-window global request quotas, and `client-control` for per-route body size constraints. The values presented throughout this section are defaults chosen to reflect the expected usage patterns of a small renewable energy deployment. They should be reviewed and adjusted by the API Gateway administrator based on the specific deployment context: the number of connected devices, the expected command frequency, the expected payload sizes, and the acceptable burst behaviour will all influence what constitutes an appropriate threshold. That is why the API Gateway administrator must be able to analyse the environment in which the gateway is deployed in order to define the most appropriate values for the different configurable parameters.

5.9.1 `limit-count` Built-in Plugin

The `limit-count` plugin implements a fixed-window hard quota: it counts requests within a sliding time window and rejects excess requests with HTTP `429` once the threshold is reached. In this implementation it is configured as a **global rule** applied before all route-level plugins, meaning it runs on every incoming request regardless of which route is targeted.

The global rule is configured as follows:

```
{
  "count": 100,
  "time_window": 1,
  "rejected_code": 429,
  "rejected_msg": "Too many requests - quota exceeded",
  "key_type": "constant",
  "key": "global"
}
```

The `key_type`: `constant` with `key`: `global` means that the counter is shared across all requests, not per source IP or per consumer. This establishes a global ceiling of 100 requests per second across the entire gateway, protecting it from being overwhelmed regardless of how many sources are contributing to the load. Requests exceeding this threshold receive a `429 Too Many Requests` response with a descriptive message before any route matching or plugin execution takes place.

This global quota complements the `nftables` firewall described in Section 5.8.1: while the firewall operates at the network level and blacklists individual source IPs that flood a specific port, the `limit-count` global rule operates at the application level and caps the total processing load on the gateway regardless of how traffic is distributed across sources.

It is worth noting that the firewall and the rate limitation are linked and should be considered together when configuring the system. For example, tightening the firewall thresholds reduces the volume of malicious traffic that reaches the application level, which in turn allows the global quota to be loosened without increasing the risk of overload from illegitimate sources.

Configurability The global rate limit is defined as a global rule via the APISIX Admin API and applied at gateway startup. An administrator can adjust the `count` and `time_window` values by modifying the `global_rules/1` entry in the setup script and re-running it, or by issuing a `PUT` request to the Admin API at runtime. Because APISIX propagates configuration changes dynamically via etcd without requiring a restart, the updated limit takes effect immediately.

5.9.2 `client-control` Built-in Plugin

The `client-control` plugin enforces a maximum request body size on each route individually. Requests whose body exceeds the configured limit are rejected before reaching any filter logic. Each route is assigned a limit proportional to the expected payload size of its operation, as summarised in Table 5.4.

Route	Methods	Max body size
`/command`	POST	1 000 B
`/backendForward`	POST	1 000 B
`/onboarding/backend`	POST, PATCH, DEL	1 000 B
`/onboarding/backendAuthZ`	POST, PATCH, DEL	2 000 B
`/onboarding/device`	POST, PATCH, DEL	4 000 B
`/onboarding/deviceAuthZ`	POST, PATCH, DEL	1 000 B
`/onboarding/translation`	POST, GET, DEL	5 000 B
`/commands`	GET	1 B

Table 5.4: `client-control` maximum body size per route

The `/onboarding/device` route has a large limit (4 000 bytes) to accommodate the `listOfDevices` batch registration payload, which may contain multiple device configurations, including command definitions and parameter schemas. The

`/onboarding/translation` route allows up to 5 000 bytes to support translation mapping configurations. Communication routes (`/command` and `/backendForward`) are kept at the lowest limit since their payloads carry only a device identifier, a command name, and a small parameter object. The `/commands` GET route is assigned a limit of 1 byte since it carries no request body by design, preventing any attempt to attach a payload to the request as a potential attack vector. A value of 1 is used rather than 0 because, as stated in the APISIX documentation [66], setting `max_body_size` to 0 disables the body size check entirely, making it equivalent to no protection. Setting the limit to 1 byte is therefore the smallest meaningful value that correctly enforces the constraint.

By constraining how much data a single request can carry, `client-control` also limits the bandwidth any single connection can consume, preventing resource exhaustion through large-payload flooding even when the request rate stays within the global quota enforced by `limit-count`.

Configurability The per-route body size limits are defined in the route configurations in the setup script and applied via the APISIX Admin API. Each route's limit can be updated independently at runtime without restarting the gateway, allowing an administrator to tighten or relax payload constraints for individual routes as operational requirements evolve.

Chapter 6

Deployment, Demonstration, and Evaluation

This chapter presents the deployment and evaluation of the proposed system, covering both a demonstration of its intended usage and an analysis of its key characteristics and their evaluation. The deployment setup is presented first, followed by a utilization flow in Section 6.2, which illustrates a typical usage scenario. Section 6.3 then examines the security mechanisms put in place and demonstrates their effectiveness against malicious behavior. Finally, Section 6.4 assesses the performance of the system under various scenarios.

6.1 Deployment Setup

This section presents the technical setup used to deploy the system. The deployment simulates a realistic usage environment composed of three main parts: a backend, the API gateway, and a set of devices. It is distributed across three dedicated machines:

- **NUC1**: hosts the backend and serves as the machine from which performance testing scripts and system metrics collection are launched.
- **NUC4**: hosts the API gateway and its associated components (database and MQTT broker).
- **NUC8**: hosts the simulated devices, both HTTP and MQTT.

NUCs. Intel NUCs are mini-computers well suited for constrained deployment environments. The models used in this work are Intel NUC UC7i7BNH units, each equipped with an Intel Core i7-7567U processor running at 3.5 GHz, quad-core, with 16 GB of RAM.

West part. The backend is simulated by a Node.js HTTP server running as a Docker container on NUC1. Throughout the different tests, the backend server acts primarily as a passive receiver, while the host machine is used to launch the test scripts and collect system metrics from NUC4.

East part. The devices are simulated on NUC8. HTTP devices are implemented as Node.js servers using the Express framework, and MQTT devices as Node.js clients using the `mqtt` package, all running in Docker containers. A total of six devices are simulated: three HTTP and three MQTT, each representing a distinct device identity in the system.

Central part. The API gateway runs on NUC4 as a Docker Compose stack comprising APISIX v3.0 [56], etcd, MongoDB, MongoExpress, and Mosquitto. APISIX and etcd form the core of the gateway: etcd handles configuration storage and propagation, while APISIX manages routing, matching, and plugin execution. MongoDB serves as the persistence layer for the Java plugin, and MongoExpress provides a lightweight web-based database viewer. Mosquitto acts as the MQTT broker, enabling communication with the MQTT devices.

Taken together, these components simulate a representative deployment environment. It is worth noting that the primary contribution of this work resides in the API gateway stack, which is fully containerised and therefore portable across different deployment contexts.

6.2 Integration / Utilization testing

This section demonstrates how the implemented API Gateway is used in practice, from the initial onboarding of entities to the execution of a command on a device. Figure 6.1 illustrates a representative example covering the full end-to-end flow.

The flow is organized into two phases. The first phase covers onboarding, during which backends and devices are registered to the API gateway and their respective authorization policies and translation configurations related to the registered devices. The second phase covers runtime communication, where an authenticated backend discovers the commands available to it and sends a command to a device through the gateway.

Onboarding Phase. The onboarding operations can be performed in any order, with two constraints: a backend or device must be registered in the configuration collection before its authorization entry can be created, and the translation configuration for a device must reference a device that has already been registered. Beyond these

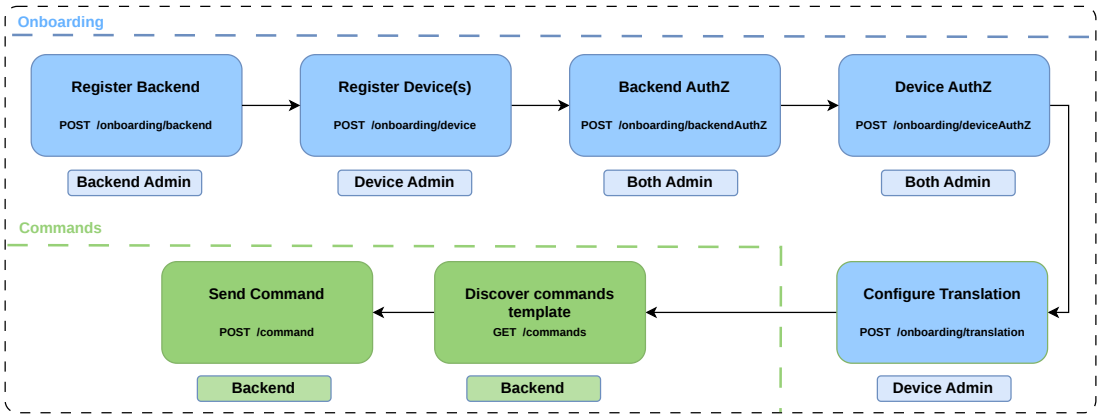

Figure 6.1: End-to-end utilization flow: from onboarding to command execution

dependencies, the order is flexible; for example, all backends could be registered before any device, or each backend and device could be onboarded at the same time. The flow shown in Figure 6.1 is therefore one possible case, not a strict sequence.

It is also worth noting that even if it is not represented in the Figure 6.1, the `gateway-admin` consumer has permission to perform all onboarding operations across all routes, meaning a single administrator can complete the entire onboarding without involving separate backend or device administrators.

Runtime Phase. Once onboarding is complete, the backend can interact with the gateway by first issuing a `GET /commands` request to retrieve the list of devices it is authorized to send a command to, along with their full command definitions and parameter schemas. This step is optional if the backend already knows the available commands. It then sends a `POST /command` request with the target device identifier, the command name, and the required parameters.

Testing. In addition, the implementation is supported by automated testing for which the code is available in the project repository. Integration tests reproduced the full end-to-end scenario (from onboarding to command execution), while unit tests validate individual components such as translation, parsing, and configuration handling classes.

It is important to mention that setting up integration tests for such a system is inherently complex, as it requires the orchestration of multiple dependent components, including HTTP and MQTT device mockups and a database. As a result, while the implemented tests cover the main end-to-end scenarios and critical components, exhaustive testing of all possible routes and configurations was not pursued.

6.3 Security Testing

In this section, the effectiveness of the implemented security mechanisms, including rate and request-size limitations and IP blacklisting, is evaluated. The goal is to verify that these mechanisms correctly enforce the defined policies and protect the system against abusive or malicious behaviour. All tests are conducted using HTTP. The results apply equally to HTTPS since the security mechanisms operate independently of the transport encryption layer. MQTT(S) is not tested for two reasons: the `limit-count` rate limitation plugin only applies to HTTP(S) traffic routed through APISIX, as all MQTT(S) traffic is translated into HTTP(S) and forwarded to the `/backendForward` route, and the `nftables` firewall behaviour for MQTT is identical to that of HTTP, with the only difference being that MQTT sources are blacklisted at a lower threshold (35 req/s versus 100 req/s).

6.3.1 Rate limitation (`limit-count`)

The security testing for rate limitation focuses on validating the behavior of the `limit-count` plugin when enforcing a maximum number of allowed requests per time window. The objective is to ensure that traffic exceeding the configured threshold is consistently rejected with an HTTP 429 response, while requests within the limit are processed normally.

To evaluate this mechanism, a spread test was conducted in which the number of requests sent per batch was progressively increased. Seven load levels were tested: 100, 300, 500, 800, 1000, 1400, and 2000 requests, each spread uniformly over a 5-second window.

Configuration. The test was performed by sending requests to the `/command` endpoint. The `limit-count` plugin was configured with a quota of **100 requests per 1-second window**, applied globally across all clients using a constant key.

Results. As illustrated in Figure 6.2, the results confirm the expected behavior across all load levels. For the first three batches (100, 300 and 500 req/5s), the total sent remains at or below the rate limit and no requests are dropped. Starting from the 800 req/5s level, the *Not Dropped* line (blue) stabilizes at 100 requests per second, while the *Dropped* line (red) grows proportionally with the excess load. The *Total Sent* line (grey) accurately tracks the sum of the blue and red lines. The consistent blue ceiling across all overload batches shows that the `limit-count` plugin enforces the quota reliably.

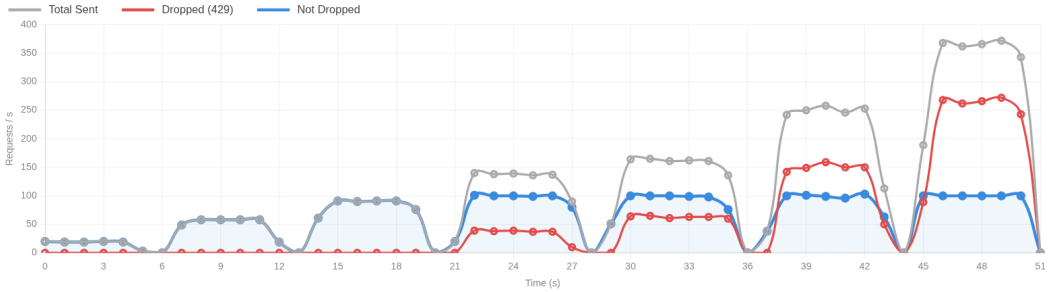

Figure 6.2: Request outcome per second across increasing load levels

6.3.2 IP Blacklisting firewall (`nftables`)

The security testing for network-level flood protection focuses on validating the behavior of the `nftables` `apisix_guard` ruleset when a source IP address exceeds the configured connection rate threshold. The objective is to confirm that the dynamic blacklist correctly identifies and blocks offending addresses while allowing legitimate traffic to continue unaffected.

To evaluate this mechanism, a spread test was conducted using the same progressive load methodology and the same seven request counts as in 6.3.1, each spread over a 5-second window.

Configuration. The `nftables` ruleset was configured to monitor new TCP connections on port 9080 for HTTP. A per-source-IP meter triggers when the connection rate exceeds **100 new connections per second**.

Results. As illustrated in Figure 6.3, the graph reveals a different pattern from the rate-limitation test. Traffic flows normally for the first three load levels (up to approximately 90 req/s), during which no drops are recorded and the *Not Dropped* line tracks the *Total Sent* line exactly. However, once the connection rate briefly exceeds the 100/s threshold during a higher-load batch, the source IP is added to the blacklist and *all subsequent traffic* from that address is dropped immediately. This is visible from second 24 onward, from where the *Not Dropped* and *Total Sent* lines collapse to zero, confirming the blacklist is active and working.

6.3.3 Request body size limitation (`client-control`)

The security testing for request size limitation focuses on validating the behavior of the `client-control` plugin when enforcing a maximum allowed request body size.

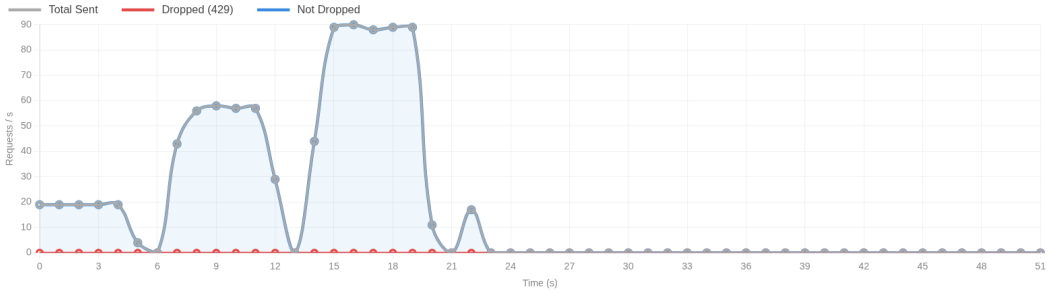

Figure 6.3: Request outcome per second during a progressive load test with `nftables` flood protection active.

The objective is to ensure that requests exceeding the configured limit are consistently rejected, while valid requests are processed normally.

To evaluate this mechanism, an experiment was conducted in which the size of the request body was progressively increased. Starting from a base payload, additional data was appended to each request, allowing us to precisely identify the rejection threshold enforced by the gateway.

Configuration. The test was performed by sending one request per second to the `/command` endpoint, with a payload size increasing from 218 bytes (the size of the initial request body) to 3000 bytes in steps of 50 bytes. The limit of size set on this route is 1000 bytes.

Results. As illustrated in Figure 6.4, all requests are accepted while the request body size remains below the configured limit. Once the payload exceeds approximately 1000 bytes, requests are systematically rejected by the `client-control` plugin with an HTTP 413 response. It is also worth noting that latency for accepted requests is between 7 ms and 14 ms. For the rejected requests, latency is slightly lower, between 5 ms and 9 ms. This difference indicates that the `client-control` plugin efficiently terminates invalid requests early in the processing pipeline, reducing the overall processing time.

6.4 Performance Testing

In distributed environments such as the context of renewable digital assets, the performance of an API gateway is critical. A misbehaving or overly slow system could compromise important commands or time-sensitive data flows. Similarly, a gateway

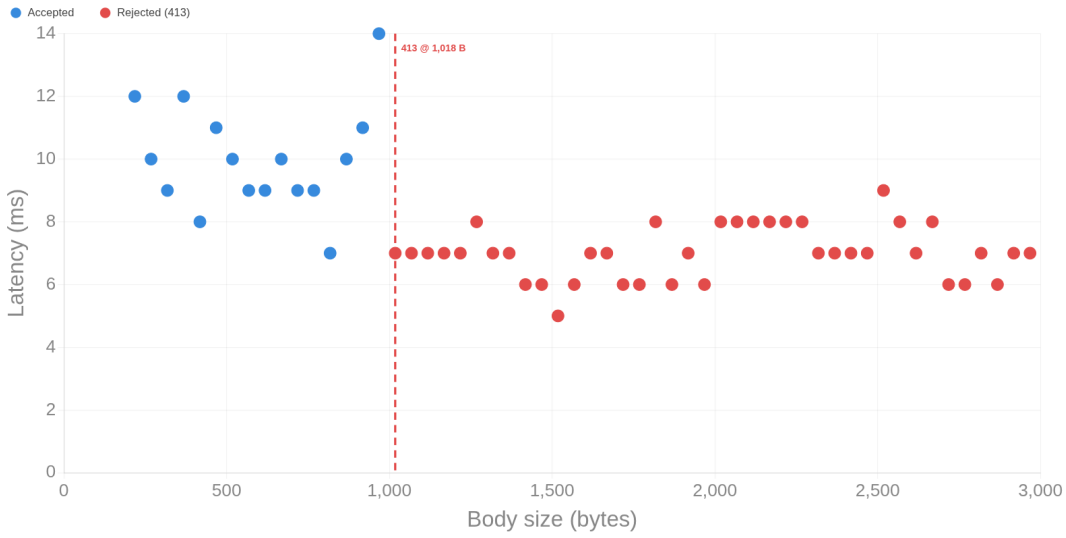

Figure 6.4: Request outcome and latency in function of body size, with a rejection threshold

capable of supporting a large number of devices while remaining efficient is desirable, as centralizing a vast device network behind fewer gateways simplifies management and reduces operational overhead.

This section demonstrates the performance of the proposed solution from two perspectives: an architectural comparative analysis and an overload characterization. All tests relied on `Locust` [67], an open-source load testing framework that spawns configurable numbers of virtual users, each firing requests at a defined rate, and collects per-request latency and throughput statistics.

6.4.1 System parameters

System parameters are inherent to the deployment setup and cannot be modified during testing, yet they can have a significant influence on results.

Bandwidth. All NUCs (nuc1, nuc4, nuc8) are interconnected via Ethernet links operating at over 100 MBytes/sec.

NUC devices. The machines used are Intel NUC mini-PCs, as presented in Section 6.1. The nuc1 acts as the load generator and backend, using the `Locust` client, while nuc4 hosts the API Gateway stack, and nuc8 hosts the simulated devices.

Devices and their server. Six devices run as Node.js servers on nuc8 to simulate real devices: three expose HTTP endpoints and three communicate over MQTT. They remain fixed throughout all performance tests.

6.4.2 Factors

The following factors define the operating conditions of each scenario. Some are fixed throughout a given context, while others vary.

Architecture related. The most influential factor: it determines whether the gateway is active, which plugin components are engaged, whether devices are mocked, and so on. Each scenario specifies its corresponding architectural configuration.

System related. The number of CPU cores allocated to APISIX, etcd, and the Java plugin sidecar. It takes values of 1, 2, or 4 cores, set directly in the docker-compose file of the gateway stack.

Network related. The request rate sent to the gateway, expressed in requests per second. It is either fixed (Context 1) or progressively increased via a ramp-up (Context 2).

Payload related. While exact field values vary between requests, each payload contains between 15 and 25 fields filled with realistic varying parameters. This represents an above-average use case, ensuring that normal usage is well covered.

Security related. The API Gateway network security (nftables rules and rate limitation) were set to non-constraining values, and were fixed for the entire experiment. Security processing therefore still occurs and its computational cost is reflected in the measurements, but it never artificially limits the achievable request rate.

6.4.3 Metrics

The following metrics are collected during each run to quantify the performance of the system.

Latency. The time elapsed between a request being issued by the client, processed through the full architectural stack, and the response being received. Latency is expressed in milliseconds and reported as median (p50) and tail percentiles (p90, p95, p99, p99.9).

Achieved request per second. The number of requests per second effectively handled by the gateway, used as the throughput metric.

RAM and CPU usage. The RAM and CPU usage of APISIX are measured throughout each run where the gateway is active, using `docker stats` polled every two seconds via SSH from nuc1.

6.4.4 Context 1: Latency characterization per architecture

The first context measures the impact of each architectural component on the latency of the system. This analysis creates comparison points that reveal how each layer of the proposed solution affects end-to-end performance.

Methodology. Each run consisted of a 120-second warmup phase, allowing the JVM, APISIX worker threads, and connection pools to reach steady state, followed by a 900-second capture window. The target rate was fixed at 100 req/s with 50 virtual users, yielding approximately 45 000 samples (requests) per protocol per scenario. Raw per-request latencies were captured via a Locust event listener and written to a dedicated CSV file during the capture phase only, excluding the warmup.

Fixed factors. No CPU limit was applied, so the gateway used all 4 available cores. The request rate was fixed at 100 req/s.

Varying factors. The only variation is architectural, expressed through the scenario definition below.

Scenarios. Five scenarios were tested, each adding one architectural layer to the previous:

- **Scenario A - Direct**: The baseline scenario. The Locust client, simulating a backend, communicates directly with the six devices on nuc8, bypassing the gateway entirely. HTTP requests are sent directly to each device's HTTPS endpoint, and MQTT commands are published directly to a broker running on the nuc1 (backend). This scenario characterizes the raw device round-trip latency and serves as the reference point for all subsequent scenarios.
- **Scenario B - GW Forward**: The gateway is introduced, but without any plugin on the request path. APISIX simply routes incoming HTTP requests to the appropriate device upstream using its native connection pooling. MQTT communication remains direct (as in Scenario A), as the gateway cannot handle MQTT without a plugin.

- **Scenario C - GW Plugin**: A passive Java plugin is attached to the route. The plugin is present in the request path and the communication between APISIX and the plugin sidecar occurs on every request, but the plugin does not perform any translation or authentication logic. This isolates the overhead of the plugin sidecar itself. As for Scenario A and B, MQTT communication is direct from nuc1 (backend) to nuc8 (devices).
- **Scenario D - GW Plugin Request**: The Java plugin now performs the full device communication itself. Upon receiving an HTTP request from the backend, it builds and sends a new HTTP or MQTT request directly from the Java plugin to the target device, then waits for the response before replying to the client. No authentication or translation is performed. The MQTT broker is now hosted on nuc4.
- **Scenario E - GW Full Stack**: The complete architecture. The plugin performs authentication, payload translation, and device communication, which represents the actual production behavior of the proposed solution.

Results. Table 6.1 reports the latency metrics for each scenario. Figure 6.5 shows the full latency distributions as box plots, separating HTTP and MQTT for each scenario. A symlog y-axis is used, with a linear scale below 80 ms where most values concentrate, and a logarithmic scale above to keep outliers visible without distorting the main distributions. The median CPU utilisation of APISIX (in cores) and its RAM usage are overlaid on a secondary axis.

	HTTP						**MQTT**					
Scenario	Avg	p50	p90	p95	p99	p99.9	Avg	p50	p90	p95	p99	p99.9
A	15.5	15	20	21	24	220	19.8	17	21	53	55	95
B	14.6	14	19	20	25	36	23.9	16	53	54	56	96
C	16.5	16	20	21	25	230	27.2	17	54	54	56	97
D	44.3	34	72	91	140	290	55.8	47	100	130	160	280
E	44.1	36	73	85	120	320	54.9	42	100	130	160	310

Table 6.1: HTTP and MQTT latency results - Context 1 (target: 100 req/s, 900 s capture). All values in milliseconds.

Interpretation. The results observed in the table and the plot are coherent and reveal several clear trends.

Scenarios A, B, and C show near-identical latency, confirming that the gateway and its plugin sidecar add negligible overhead when no device-level request is issued

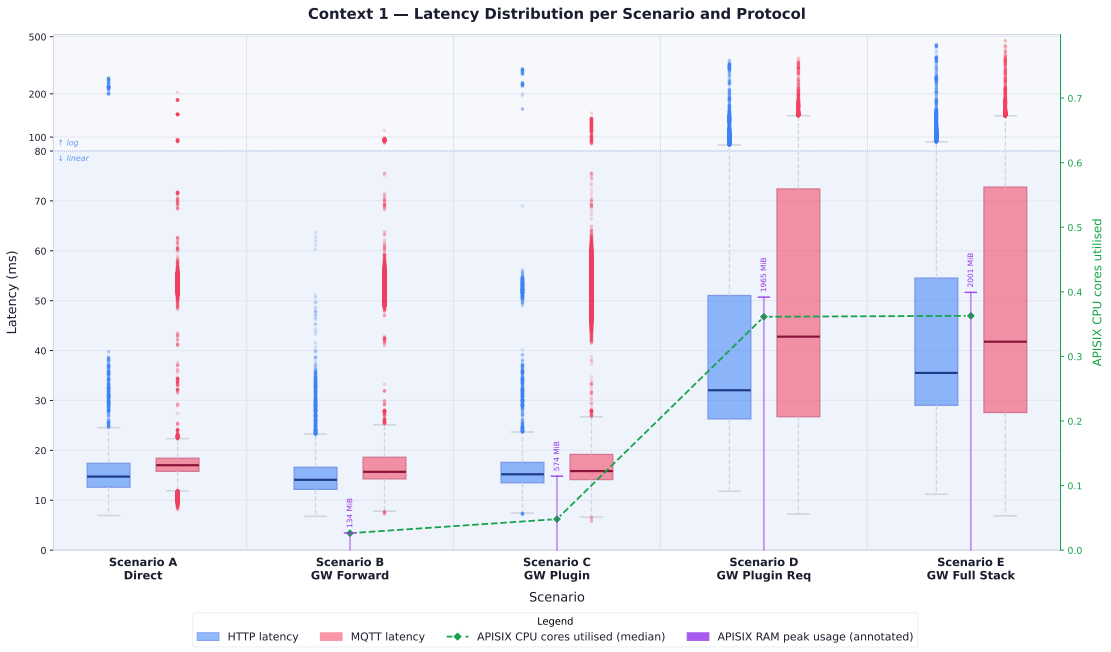

Figure 6.5: Context 1: latency distribution per scenario and protocol.

by the plugin. For HTTP, APISIX's connection pooling in Scenario B actually almost matches the direct latency of Scenario A, proving the efficiency of APISIX. In Scenario C, the presence of the Java plugin adds a small but measurable overhead due to the round-trip between APISIX and the sidecar on every request, visible as a slight increase in latency. MQTT is unaffected across these three scenarios since it remains direct in all cases. The small increase in CPU and RAM usage from Scenario B to C reflects the activation of the plugin sidecar.

Scenarios D and E show a clear jump in latency of approximately 2× compared to the previous group. This is caused by the plugin now creating and dispatching a new outbound request to the device on every inbound command, bypassing APISIX's native connection management. The resulting latency is dominated by the packet creation, network round-trip, and response handling within the Java runtime, which introduces both higher median latency and significantly higher variance. Reasons for this could be JVM scheduling and garbage collection. RAM and CPU usage increase substantially as a result.

The difference between D and E is marginal: authentication and translation add at most 2 ms to the HTTP median, and the MQTT median even decreases slightly, likely due to run-to-run noise. This confirms that the core value-adding operations of the plugin (authentication and payload translation) are well optimized and do not

constitute a meaningful overhead relative to the cost of device communication.

The actual bottleneck is relatively straightforward to identify. In a direct communication scenario (Scenario A), a single HTTP request is exchanged between the client and the device. In the proposed architecture, however, the initial request is first intercepted by the gateway, which then generates a second outbound request from the Java plugin towards the device. Once the device responds, the gateway forwards the response back to the original client request. As a result, a single communication flow effectively becomes two sequential HTTP exchanges (or 1 HTTP + 1 MQTT), which explains the observed near doubling of latency.

6.4.5 Context 2: Overload testing

The second context characterizes the maximum throughput capacity of the API Gateway under increasing load. A ramp-up mechanism is used to progressively stress the system until saturation is reached.

Methodology. A staircase ramp-up load test was conducted using Locust, increasing the target request rate by 10 req/s every 20 seconds. For each step, the first three seconds were discarded to eliminate transition noise, and the median achieved throughput and median p50 latency were computed from the remaining samples. Only every fifth step was retained for visualization, yielding one data point every 50 req/s of target load. Points are sorted by ascending latency and connected in that order, with a monotonicity filter applied to suppress noise-driven variance in the stable zone. Docker stats were collected on nuc4 throughout each run and matched to each step by timestamp to report CPU and RAM usage per throughput level.

Fixed factors. The gateway always runs the full authentication and translation stack. There is no direct backend → device communication. The ramp-up always starts at 10 req/s and proceeds in steps of 10 req/s until the median p50 latency exceeds at least 750 ms.

Varying factors. The number of allocated CPU cores (1, 2, or 4) is varied per run.

Scenarios. Two scenarios were tested:

- **Scenario A - No device mockup**: The full architecture is used end-to-end, including real device communication. The gateway must authenticate, translate, and dispatch each request to a physical device simulator (nuc8) and await its response.

- **Scenario B - Device mockup**: The plugin performs authentication and translation as usual, but the outbound device request is skipped and a 200 response is returned immediately. This isolates gateway processing capacity from device round-trip latency.

For the Scenario B / 4-CPU configuration, an external more powerful machine was used as the load generator to sustain the required throughput.[1]

Results. Table 6.2 reports the maximum achieved request rate for each configuration. Figure 6.6 shows the full latency versus throughput saturation curves for all six cases, with CPU core utilization overlaid on a secondary axis and with faint lines, and labels at peak throughput to indicate RAM usage.

Scenario	CPU cores	Max req/s
A	1	258
A	2	421
A	4	1085
B	1	1696
B	2	3026
B	4	5600

Table 6.2: Maximum achieved throughput per configuration - Context 2.

Interpretation. The results clearly expose the main bottleneck in the system.

In Scenario A, the device round-trip is the limiting factor. With 1 and 2 CPU cores, the gateway saturates at 258 and 421 req/s respectively, with a visible backward collapse where latency continues to rise while achieved throughput decreases, a sign that the gateway is so overwhelmed it can no longer sustain even its peak rate. With 4 cores, the gateway reaches 1 085 req/s, a better than linear gain relative to the 2-core case, suggesting that the workload parallelizes efficiently across cores at this scale. RAM usage in Scenario A is substantial (up to ~3.2 GiB), reflecting the cost of maintaining concurrent open device connections and in flight request state.

Scenario B reveals the gateway's intrinsic processing capacity in isolation. The difference compared to Scenario A is striking: without device round-trips, throughput increases by a factor of 5 to 6 across all CPU configurations, confirming that device communication is the dominant cost in Scenario A. RAM usage drops significantly as well, since no device connections need to be maintained. The scaling across CPU

[1]The gateway was capable of handling up to 5 600 req/s in this configuration; nuc1 could only generate up to approximately 3 600 req/s even with optimizations / multi-workers usage.

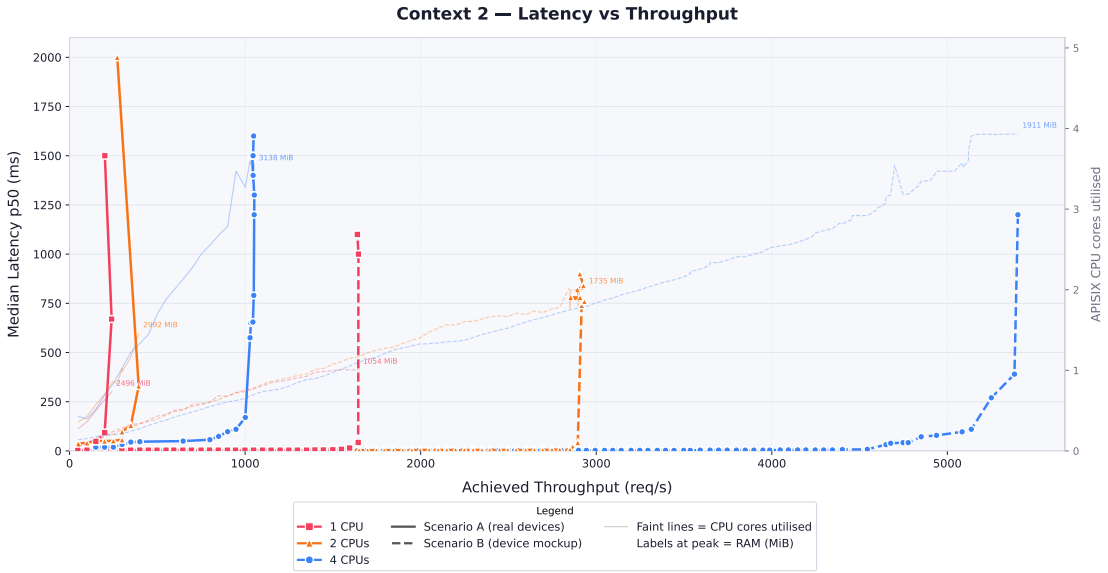

Figure 6.6: Context 2: median latency (p50) versus achieved throughput.

counts remains consistent: doubling from 1 to 2 cores roughly doubles throughput, and moving to 4 cores yields another near doubling. In both scenarios, the saturation point is less abrupt for 4 cores, with latency rising more gradually before the collapse, whereas 1 and 2 cores exhibit a sharper inflection point.

Taken together, these results demonstrate that the gateway's authentication and translation logic is well optimized and not a performance bottleneck. The bottleneck lies squarely in device communication latency (Scenario A) or, once devices are removed, in the raw packet processing capacity of the gateway machine itself (Scenario B).

6.4.6 Conclusion

The performance evaluation highlights two main findings.

First, the architectural overhead introduced by the complete plugin stack, including authentication and payload translation, remains limited. At 100 req/s, it increases the median latency by at most 2 ms compared to the plugin only handling outbound communication (Scenario D), indicating that the core processing operations are efficiently implemented. The primary performance cost originates from outbound device communication been handled by the Java sidecar, which approximately doubles the median latency once the plugin starts issuing external requests. This identifies device communication as the current performance bottleneck and a key area for future

optimization.

Second, the gateway demonstrates predictable scalability with increasing CPU resources and is capable of sustaining over 1 000 req/s with outbound device communication. Even on a modest 4-core machine, the gateway already delivers strong performance. In an edge-computing context, where more powerful hardware may be available, the architecture could therefore support communication with a large number of devices.

Chapter 7

Conclusion and Perspectives

7.1 Conclusion

This thesis addressed the challenge of enabling secure, trustworthy, and interoperable communication between cloud backends and heterogeneous renewable energy devices. The central contribution is the design and implementation of an API Gateway architecture based on Apache APISIX, developed in the context of the ITEA4 HomePot project in collaboration with the Sirris Research Center and 3E.

The gateway was designed to address three main challenges. First, from a communication standpoint, it abstracts the heterogeneity of renewable energy devices by providing a unified interface to the backend, regardless of the device's protocol, message format, or behavioral model. Through a dedicated protocol translation subsystem supporting HTTP(S) and MQTT(S), the gateway decouples backend services from low-level device integration logic, reducing complexity and improving maintainability as the fleet of devices grows.

Second, from a security standpoint, the gateway acts as a single enforcement point for authentication, authorization, input validation, and rate limitation. By positioning it between the cloud backend and the devices, which are often deployed in an untrusted environment, the overall attack surface is significantly reduced. The gateway prevents compromised or malicious devices from directly reaching core backend services. Access control is layered: JWT-based consumer authentication and role-based access control govern administrative operations, while API key validation and per-command, per-device authorization are enforced at runtime for all communication traffic. Credentials are generated using cryptographically secure randomness and stored only as SHA-256 hashes, ensuring that a database compromise does not expose usable credentials. TLS is enforced across all communication channels, with certificate-based authentication in both directions for HTTPS and one-way TLS for MQTTS. Traffic overload protection is implemented at two complementary levels: a

network-level `nftables` firewall with per-protocol rate meters and dynamic IP blacklisting, and application-level payload size enforcement combined with JSON schema validation on the command route.

Third, from a bidirectionality standpoint, the gateway supports not only the backend-to-device direction through the translation mechanism, but also the device-to-backend direction through the `/backendForward` route. Devices can push telemetry and status updates to their authorized backends without any knowledge of who receives the data, while the gateway enforces isolation between the two parties through its authorization configuration.

The architecture was evaluated on real hardware. The security mechanisms were verified to correctly enforce their respective policies: the `limit-count` plugin reliably caps the request rate at the configured threshold, the `nftables` firewall correctly blacklists sources exceeding the connection rate limit, and the `client-control` plugin rejects oversized payloads at the expected boundary. The performance evaluation revealed that the authentication and payload translation logic introduces minimal overhead, at most 2 ms at the median, and that the primary performance cost originates from outbound device communication, which approximately doubles the median latency. With 4 CPU cores, the gateway sustains over 1 000 requests per second with full device communication enabled, and over 5 000 requests per second when device round-trips are removed.

Taken together, these results demonstrate that the proposed architecture successfully addresses the identified challenges, providing a practical and extensible solution for the secure and interoperable integration of heterogeneous renewable energy assets with cloud infrastructure.

7.2 Future Work

While the proposed gateway addresses the core communication and security challenges identified in this thesis, several directions remain open for future work.

Outbound Communication Security. The current solution supports access tokens as an authentication mechanism for outbound backend and device communication. However, this remains limited to fixed header values and does not accommodate more complex flows such as OAuth 2.0 with refresh tokens.

Interoperability. The solution currently supports HTTP and MQTT, but the adapter-based design is intended to be extensible. Adding support for other IoT-relevant protocols, such as CoAP or AMQP, would broaden the range of devices the gateway can manage.

Real Environment Testing. The evaluation presented in this thesis relies on simulated devices. Deploying and testing the system in a real environment with physical renewable energy devices would help uncover practical issues and improve the overall usability of the tool.

Usability. The current solution is operated entirely through API calls. A graphical user interface could be built on top of the existing API to offer a more accessible experience for operators who are less familiar with REST interfaces.

Outbound Communication Performance. As shown by the performance evaluation, the main bottleneck lies in the outbound device communication handled by the Java plugin. Optimizing this path, towards what APISIX achieves natively, would significantly improve throughput, especially since the translation and authentication overhead is already minimal.

Resilience. On a longer horizon, the system should account for evolving threats, including the potential impact of quantum computing on current cryptographic primitives. Adopting post-quantum algorithms as they mature would help ensure the gateway remains secure over time.

Communication Types. Supporting additional communication patterns, such as data streaming or bidirectional channels, would make the system more versatile for devices that require continuous data exchange rather than request-response interactions.

These directions offer meaningful avenues for extending the contribution. The current system, however, already addresses the key challenges it set out to solve, and is ready to be deployed and evaluated in broader contexts.

Bibliography

[1] Eurostat. (2024) Renewables take the lead in power generation in 2023. https://ec.europa.eu/eurostat/web/products-eurostat-news/w/ddn-20240627-1. Accessed: 2026-02-07.

[2] Wikipedia contributors. (2026) Renewable energy - Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Renewable_energy&oldid=1338697469. Ccessed: 2026-02-06.

[3] A. De Neyer and E. Khoury, "Edge control – API gateway for secure renewable energy communication," https://github.com/GaecKo/MasterThesis, 2025-2026, master's thesis repository.

[4] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, and J. Alonso-Zarate, "A Survey on Application Layer Protocols for the Internet of Things."

[5] J. Dizdarević, F. Carpio, A. Jukan, and X. Masip-Bruin, "A Survey of Communication Protocols for Internet of Things and Related Challenges of Fog and Cloud Computing Integration," *ACM Comput. Surv.*, vol. 51, no. 6, pp. 116:1–116:29, Jan. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3292674

[6] M. Laroui, B. Nour, H. Moungla, M. A. Cherif, H. Afifi, and M. Guizani, "Edge and fog computing for IoT: A survey on current research activities & future directions," *Computer Communications*, vol. 180, pp. 210–231, Dec. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366421003327

[7] T.-A. N. Abdali, R. Hassan, A. H. M. Aman, and Q. N. Nguyen, "Fog Computing Advancement: Concept, Architecture, Applications, Advantages, and Open Issues," *IEEE Access*, vol. 9, pp. 75 961–75 980, 2021. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9435363

[8] E. Al-Masri, K. R. Kalyanam, J. Batts, J. Kim, S. Singh, T. Vo, and C. Yan, "Investigating messaging protocols for the internet of things (iot)," *IEEE Access*, vol. 8, pp. 94 880–94 911, 2020. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9090208

[9] S. Sinche, D. Raposo, N. Armando, A. Rodrigues, F. Boavida, V. Pereira, and J. S. Silva, "A survey of iot management protocols and frameworks," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 2, pp. 1168–1190, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8848791

[10] A. Rekeraho, D. T. Cotfas, P. A. Cotfas, E. Tuyishime, T. C. Balan, and R. Acheampong, "Enhancing Security for IoT-Based Smart Renewable Energy Remote Monitoring Systems," *Electronics*, vol. 13, no. 4, p. 756, Jan. 2024. [Online]. Available: https://www.mdpi.com/2079-9292/13/4/756

[11] J. Ye, A. Giani, A. Elasser, S. K. Mazumder, C. Farnell, H. A. Mantooth, T. Kim, J. Liu, B. Chen, G.-S. Seo, W. Song, M. D. R. Greidanus, S. Sahoo, F. Blaabjerg, J. Zhang, L. Guo, B. Ahn, M. B. Shadmand, N. R. Gajanur, and M. A. Abbaszada, "A Review of Cyber–Physical Security for Photovoltaic Systems," *IEEE Journal of Emerging and Selected Topics in Power Electronics*, vol. 10, no. 4, pp. 4879–4901, Aug. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9534741/

[12] I. Zografopoulos, N. D. Hatziargyriou, and C. Konstantinou, "Distributed Energy Resources Cybersecurity Outlook: Vulnerabilities, Attacks, Impacts, and Mitigations," *IEEE Systems Journal*, vol. 17, no. 4, pp. 6695–6709, Dec. 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10238347/

[13] ITEA 4. (2024) HOMEPOT - Homogenous Cyber Management of End-Points and OT. https://itea4.org/project/homepot.html. Accessed: 2026-02-07.

[14] EUREKA. (2026) Itea4 – vision and mission. Accessed: 2026-02-25. [Online]. Available: https://itea4.org/vision-and-mission.html

[15] Sirris. (2026) About sirris. https://www.sirris.be/en/about-us. Accessed: 20 March 2026.

[16] 3E. (2026) About 3e. https://www.3e.eu/about-us. Accessed: 20 March 2026.

[17] J. Zhao, S. Jing, and L. Jiang, "Management of api gateway based on micro-service architecture," in *Journal of Physics: Conference Series*, vol. 1087, no. 3. IOP Publishing, 2018, p. 032032.

[18] S. Gadge and V. Kotwani, "Microservice architecture: Api gateway considerations," 2017. [Online]. Available: https://mainlab.cs.ccu.edu.tw/presentation/pdf/(2017)Microservice-Architecture-API-Gateway-Considerations.pdf

[19] T.-A. N. Abdali, R. Hassan, A. H. M. Aman, and Q. N. Nguyen, "Fog Computing Advancement: Concept, Architecture, Applications, Advantages, and Open Issues," *IEEE Access*, vol. 9, pp. 75 961–75 980, 2021. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9435363

[20] S. Sciancalepore, G. Piro, D. Caldarola, G. Boggia, and G. Bianchi, "OAuth-IoT: An access control framework for the Internet of Things based on open standards," in *2017 IEEE Symposium on Computers and Communications (ISCC)*, Jul. 2017, pp. 676–681. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8024606

[21] Y. Chen, W. Sun, N. Zhang, Q. Zheng, W. Lou, and Y. T. Hou, "Towards Efficient Fine-Grained Access Control and Trustworthy Data Processing for Remote Monitoring Services in IoT," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 7, pp. 1830–1842, Jul. 2019. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8566004

[22] J. Ni, K. Zhang, X. Lin, and X. Shen, "Securing Fog Computing for Internet of Things Applications: Challenges and Solutions," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 601–628, 2018. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8066283

[23] B. Alhijawi, S. Almajali, H. Elgala, H. Bany Salameh, and M. Ayyash, "A survey on DoS/DDoS mitigation techniques in SDNs: Classification, comparison, solutions, testing tools and datasets," *Computers and Electrical Engineering*, vol. 99, p. 107706, Apr. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045790622000234

[24] S. Serbout, A. El Malki, C. Pautasso, and U. Zdun, "API Rate Limit Adoption – A pattern collection," in *Proceedings of the 28th European Conference on Pattern Languages of Programs*. Irsee Germany: ACM, Jul. 2023, pp. 1–20. [Online]. Available: https://dl.acm.org/doi/10.1145/3628034.3628039

[25] H. Derhamy, J. Eliasson, and J. Delsing, "IoT Interoperability—On-Demand and Low Latency Transparent Multiprotocol Translator," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1754–1763, Oct. 2017. [Online]. Available: https://ieeexplore.ieee.org/document/7908961

[26] M. A. Da Cruz, J. J. Rodrigues, P. Lorenz, P. Solic, J. Al-Muhtadi, and V. H. C. Albuquerque, "A proposal for bridging application layer protocols to HTTP on IoT solutions," *Future Generation Computer Systems*, vol. 97, pp. 145–152, Aug. 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167739X18322763

[27] C. Akasiadis, V. Pitsilis, and C. D. Spyropoulos, "A Multi-Protocol IoT Platform Based on Open-Source Frameworks," *Sensors*, vol. 19, no. 19, p. 4217, Jan. 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/19/4217

[28] Z. Benomar, M. Garofalo, N. Georgantas, F. Longo, G. Merlino, and A. Puliafito, "Bridging IoT Protocols with the Web of Things: A Path to Enhanced Interoperability," in *2024 IEEE International Conferences on Internet of Things (iThings) and IEEE Green Computing & Communications (GreenCom) and IEEE Cyber, Physical & Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics*, Aug. 2024, pp. 44–51. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10731787

[29] R. Xu, W. Jin, and D. Kim, "Microservice Security Agent Based On API Gateway in Edge Computing," *Sensors*, vol. 19, no. 22, Nov. 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/22/4905

[30] Kong Inc. (2026) Kong Gateway Documentation. https://developer.konghq.com/gateway/. Accessed: 2026-02-12.

[31] Tyk Technologies Ltd. (2026) Tyk API Gateway Documentation. https://tyk.io/docs/tyk-oss-gateway. Accessed: 2026-02-12.

[32] Express Gateway. (2026) Express Gateway Documentation. https://www.express-gateway.io/docs/. Accessed: 2026-02-12.

[33] KrakenD. (2026) KrakenD API Gateway Documentation. https://www.krakend.io/. Accessed: 2026-02-12.

[34] The Apache Software Foundation. (2026) Apache APISIX Documentation. https://apisix.apache.org/. Accessed: 2026-02-12.

[35] daily.dev. (2026) Top 6 open-source api gateway frameworks. https://daily.dev/blog/top-6-open-source-api-gateway-frameworks. Accessed: 12 February 2026.

[36] Nordic APIs. (2026) 6 open-source api gateways. https://nordicapis.com/6-open-source-api-gateways/. Accessed: 12 February 2026.

[37] API7.ai. (2026) Top 11 api gateways platforms compared. https://api7.ai/top-11-api-gateways-platforms-compared. Accessed: 12 February 2026.

[38] Cyberlands. (2026) Open source api gateways. https://www.cyberlands.io/opensourceapigateways. Accessed: 12 February 2026.

[39] APIdog. (2026) Best api gateways. https://apidog.com/blog/best-api-gateways/. Accessed: 26-02-12.

[40] Kong Inc. (2026) Kong Gateway Installation Guide. https://developer.konghq.com/gateway/install/. Accessed: 2026-02-12.

[41] Kong Inc. (2026) Kong Gateway deployment topologies. https://developer.konghq.com/gateway/deployment-topologies/. Accessed: 2026-02-12.

[42] Tyk Technologies. (2026) Tyk Open Source Gateway Installation Guide. https://tyk.io/docs/apim/open-source/installation. Accessed: 2026-02-12.

[43] Express Gateway Project. (2023) Express gateway installation guide. https://www.express-gateway.io/docs/installation/. Accessed: 2026-02-28.

[44] KrakenD. (2024) Krakend installation and deployment. https://www.krakend.io/docs/overview/installing/. Accessed: 2026-02-28.

[45] Apache Software Foundation. (2024) Apache apisix installation guide. https://apisix.apache.org/docs/apisix/installation-guide/. Accessed: 2026-02-28.

[46] Kong Inc. (2024) Kong admin api. https://docs.konghq.com/gateway/latest/admin-api/. Accessed: 2026-02-28.

[47] Tyk Technologies Ltd. (2024) Tyk gateway configuration. https://tyk.io/docs/tyk-oss-gateway/configuration/. Accessed: 2026-02-28.

[48] Express Gateway Project. (2023) Express gateway configuration. https://www.express-gateway.io/docs/configuration/. Accessed: 2026-02-28.

[49] KrakenD. (2024) Krakend configuration. https://www.krakend.io/docs/configuration/. Accessed: 2026-02-28.

[50] Apache Software Foundation. (2024) Apache apisix admin api. https://apisix.apache.org/docs/apisix/admin-api/. Accessed: 2026-02-28.

[51] Kong Inc. (2024) Kong gateway community and support. https://konghq.com/community. Accessed: 2026-02-28.

[52] Tyk Technologies Ltd. (2024) Tyk community and support. https://tyk.io/docs/developer-support/community. Accessed: 2026-02-28.

[53] Express Gateway Project. (2023) Express gateway github repository. https://github.com/ExpressGateway/express-gateway. Accessed: 2026-02-28.

[54] KrakenD. (2024) Krakend community. https://www.krakend.io/support/. Accessed: 2026-02-28.

[55] Apache Software Foundation. (2024) Apache apisix community. https://apisix.apache.org/docs/general/join. Accessed: 2026-02-28.

[56] A. S. Foundation. (2024) Apache apisix: The cloud-native api gateway. https://github.com/apache/apisix. Accessed: 2026-04-04.

[57] A. S. Foundation. (2024) Apache apisix: Architecture design. https://apisix.apache.org/docs/apisix/architecture-design/apisix/. Accessed: 2026-04-04.

[58] A. S. Foundation. (2024) Apache apisix: Overview. https://apisix.apache.org/. Accessed: 2026-04-04.

[59] A. S. Foundation. (2024) Apache apisix: External plugin (plugin runner). https://apisix.apache.org/docs/apisix/external-plugin/. Accessed: 2026-04-04.

[60] A. S. Foundation. (2024) Apache apisix: Getting started. https://apisix.apache.org/docs/apisix/getting-started/. Accessed: 2026-04-04.

[61] A. A. Community. (2021) How to write an apache apisix plugin in java. https://apisix.apache.org/blog/2021/06/21/use-java-to-write-apache-apisix-plugins/. Accessed: 2026-04-04.

[62] A. S. Foundation. (2024) The implementation of plugin runner: Apache apisix internal documentation. https://apisix.apache.org/docs/apisix/internal/plugin-runner/. Accessed: 2026-04-04.

[63] A. S. Foundation. (2024) Apache apisix java plugin runner: Development guide. https://apisix.apache.org/docs/java-plugin-runner/development/. Accessed: 2026-04-04.

[64] A. A. Community. (2022) xrpc details and multilingual support in apache apisix. https://apisix.apache.org/blog/2022/01/21/apisix-xrpc-details-and-miltilingual/. Accessed: 2026-04-04.

[65] A. S. Foundation. (2025) The internal of apisix java plugin runner. https://apisix.apache.org/docs/java-plugin-runner/the-internal-of-apisix-java-plugin-runner/. Accessed: 2026-04-11.

[66] Apache Software Foundation, “Apache APISIX: client-control plugin,” https://apisix.apache.org/docs/apisix/plugins/client-control/, 2024, accessed: 2026-04-12.

[67] J. Heyman, L. Holmberg, and A. Baldwin. What is locust? hhttps://docs.locust.io/en/stable/what-is-locust.html. Accessed: 2026-05-14.

Appendix A

API Reference

This appendix documents the full API exposed by APISIX, detailing the effect of each HTTP method, the expected request body, the possible responses and the EdgeControl manager method it invokes.

A.1 Communication Configuration Routes

Communication configuration
`/onboarding/backend` and `/onboarding/device`

`POST` **Register a new entity**

Registers the entity and returns a unique gateway identifier and a plain-text API key to the caller. For devices, the body may contain a `listOfDevices` array, enabling batch registration in a single call. Each registered device receives its own identifier and key. For backends, an optional `security` field may be provided to register an outbound access token, used by the gateway when authenticating to external HTTP services related to this specific backend.

Headers:

`Authorization`	`Bearer <token>`	*required*

Backend body:

```
{
  "backendName": "<name>",
  "type": "<type>",
  "infoEndpoint": "<info_endpoint>",
  "callbackEndpoint": "<callback_endpoint>",
  "security": {
    "type": "<token_type>",
    "token": "<bearer_token>",
    "expiracyDate": "<date>"
  }
}
```

Device body:

```
{
    "listOfDevices": [
        {
            "deviceName": "<name>",
            "type": "<type>",
            "commands": {
                "<command_name>": {
                    "name": "<command_name>",
                    "params": {
                        "<param_name_1": { "type": "<type>", "
                            description": "<description>", "mandatory
                            ": <true|false> },
                        "<param_name_2": { "type": "<type>", "
                            description": "<description>", "mandatory
                            ": <true|false> }
                    }
                }
            }
        }
    ]
}
```

200 {"gateway<Device|Backend>Id":"<device|backend>_<UUID>", "apiKey":"<plain_text>"}

400 Failed to create <Device|Backend>

Managers:

`BackendManager.createBackend()`

`DeviceManager.createDevice()`

PATCH **Update communication configuration**

Modifies the communication details of an existing entity (e.g., endpoint URL, protocol parameters). Supports adding and modifying fields via `$set`, and removing specific fields via a `fieldsToRemove` list, without replacing the full document.

Headers:

`Authorization`	`Bearer <token>`	*required*

Backend body:

```
{
    "gatewayBackendId": "backend_<UUID>",
    "infoEndpoint": "<info_endpoint>",
    "fieldsToRemove": ["callbackEndpoint"]
}
```

Device body:

```
{
    "gatewayDeviceId": "device_<UUID>",
    "commands": {
    "<command_name>": {
        "params": {
            "<param_name_1": { "type": "<type>", "description": "<
                new_description>", "mandatory": <true|false> }
        }
    }
    },
    "fieldsToRemove":["commands.<command_name>.parms.<param_name_2>
        "]
}
```

200 `{"status":"success", message:"<Device|Backend> configuration updated successfully."}`

400 Failed to update <Device|Backend> configuration. <Device|Backend> may not exist or invalid request format.

Managers:

`BackendManager.updateBackend()`
`DeviceManager.updateDevice()`

DELETE **Remove entity**

Deletes the entity's configuration document and cascades the deletion to the authorization collection of the other entity type, as described above.

Headers:

`Authorization`	`Bearer <token>`	*required*

Backend body:

```
{
  "gatewayBackendId": "backend_<UUID>"
}
```

Device body:

```
{
  "gatewayDeviceId": "device_<UUID>"
}
```

200 `{"status":"success","message":"Deleted the <Device|Backend> configuration and all related authorizations."}`

400 `{"status":"partial_failure","message":"Failed to delete related authorizations. <Device|Backend> configuration deleted successfully."}`

400 `{"status":"partial_failure","message":"Failed to delete <Device|Backend> configuration. Related authorizations deleted successfully."}`

400 `{"status":"failure","message":"Failed to delete <Device|Backend> configuration and related authorizations. <Device|Backend> may not exist or invalid request format."}`

Managers:

`BackendManager.deleteBackend()`
`DeviceManager.deleteDevice()`

A.2 Authorization Configuration Routes

Authorization configuration
`/onboarding/backendAuthZ` and `/onboarding/deviceAuthZ`

POST **Create authorization entry**

Creates a new authorization entry for the entity. The data model differs between entity types: backend entries map each `gatewayDeviceId` to a list of permitted command names, while device entries hold a flat list of authorized `gatewayBackendId` values.

Headers:

`Authorization`	`Bearer <token>`	*required*

Backend body:

```
{
    "gatewayBackendId": "backend_<UUID>",
    "listOfAuthorizations": {
        "device_<UUID>": ["<command_name_1>", "<command_name_2>"]
    }
}
```

Device body:

```
{
    "gatewayDeviceId": "device_<UUID>",
    "listOfAuthorizations": ["backend_<UUID1>", "backend_<UUID2>"]
}
```

200 `{"status":"success","message":"<Device|Backend> entry added successfully."}`

400 Failed to add <Device|Backend> authorization entry. <Device|Backend> does not exist in configuration collection.

400 Failed to add <Device|Backend> authorization. Authorization entry may already exist or invalid request format.

Manegers:

`BackendManager.addBackendAuthorizationConfig()`
`DeviceManager.addDeviceAuthorizationConfig()`

PATCH **Update authorization entry**

Incrementally modifies an existing authorization entry via `listOfAuthorizationsToAdd` and `listOfAuthorizationsToRemove` fields. MongoDB `$addToSet` and `$pull` are applied in two sequential operations to avoid conflicts when the same field appears in both.

Headers:

`Authorization`	`Bearer <token>`	*required*

Backend body:

```
{
    "gatewayBackendId": "backend_<UUID>",
    "listOfAuthorizationsToAdd": {
        "device_<UUID>": ["<command_name_1>"]
    },
    "listOfAuthorizationsToRemove": {
        "device_<UUID>": ["<command_name_2>"]
    }
}
```

Device body:

```
{
    "gatewayDeviceId": "device_<UUID>",
    "listOfAuthorizationsToAdd": ["backend_<UUID1>"],
    "listOfAuthorizationsToRemove": ["backend_<UUID2>"]
}
```

200 `{"status":"success","message":"<Device|Backend> updated successfully."}`

400 Failed to update <Device|Backend> authorization entry. <Device|Backend> does not exist in configuration collection.

Managers:

`BackendManager.updateBackendAuthorizationConfig()`
`DeviceManager.updateDeviceAuthorizationConfig()`

DELETE **Remove authorization entry**

Removes the entire authorization document for the entity. Does not cascade; cross-entity cleanup is handled by the `DELETE` on the configuration routes above.

Headers:

`Authorization`	`Bearer <token>`	*required*

Backend body:

```
{
    "gatewayBackendId": "backend_<UUID>"
}
```

Device body:

```
{
    "gatewayDeviceId": "device_<UUID>"
}
```

200 {"status":"success","message":"<Device|Backend> and related authorizations deleted successfully."}

400 Failed to delete <Device|Backend>. <Device|Backend> may not exist or invalid request format.

Managers:

`BackendManager.deleteBackendAuthorizationConfig()`
`DeviceManager.deleteDeviceAuthorizationConfig()`

A.3 The /commands Route

GET **Retrieve authorized commands**

Returns the list of all devices the calling backend is authorized to command, extended with each device's name and full command definitions, including parameter schemas. The caller is identified exclusively from the apikey header, no request body is required. The request is rejected if the key resolves to a device rather than a backend, for example.

Headers:

apikey	<plain-text API key>	*required*

200
```
{"status":"success",
"authorizedDevices":{
  "device_<UUID>":{
     "deviceName":"<name>",
    "commands":{
          "<command_name_1>":{params:  ...},
          "<command_name_2>":{params:  ...}
     }
  },
  ...}}
```

403 {"message":"Unauthorized access:API key belongs to a device"}

403 {"message":"Invalid API key"}

400 {"message":"Failed to retrieve backend authorizations. Backend may not exist or invalid request format."}

Managers:

`BackendManager.getBackendAuthorizedCommands()`

A.4 The `/onboarding/translation` Route

Translation Configuration

`/onboarding/translation`

POST **Create translation entry**

Creates a new translation configuration for a device and initialises the corresponding adapter.

Headers:

`Authorization`	`Bearer <token>`	*required*

Body – HTTP device configuration:

```
{
    "gatewayDeviceId": "device_<UUID>",
    "adapter": "http",
    "queuing": {
        "retryIntervalSeconds": <nb_sec>,
        "maxTimeToLiveSeconds": <nb_sec>
    },
    "commands": {
        "<command_name>": {
            "name": "<command_name>",
            "endpoint": "<http_endpoint>",
            "method": "<GET|POST|PUT|PATCH|DELETE>",
            "timeouts": {
                "connect": <nb_sec>,
                "request": <nb_sec>
            },
            "payloadTemplate": {
                "<fixed_field>": "<fixed_value>",
                "<param_field>": null,
                "<nested_field>": {
                    "<sub_field>": null
```

```
                }
            },
            "mappings": {
                "<param_name>": "<path_in_payload_template>",
                "<param_name>": "<nested_field>.<sub_field>"
            },
            "cleanup": {
                "removeNulls": <true|false>,
                "removeEmpty": <true|false>,
                "emptyObjectToNull": <true|false>
            }
        }
    }
}
```

Body – MQTT device configuration:

```
{
    "gatewayDeviceId": "device_<UUID>",
    "adapter": "mqtt",
    "queuing": {
        "retryIntervalSeconds": <nb_sec>,
        "maxTimeToLiveSeconds": <nb_sec>
    },
    "connection": {
        "qos": <1|2|3>,
        "keepAliveInterval": <nb_sec>,
        "connectionTimeout": <nb_sec>,
        "cleanSession": <true|false>,
        "reconnectDelay": <nb_sec>
    },
    "subscriptions": [
        {
            "topic": "<subscribe_topic>",
            "forwardToBackend": <true|false>
        }
    ],
    "commands": {
        "<command_name>": {
            "name": "<command_name>",
            "topic": "<publish_topic>",
            "responseTopic": "<response_topic>",
```

```
            "timeouts": {
                "response": <nb_sec>
            },
            "payloadTemplate": {
                "<fixed_field>": "<fixed_value>",
                "<param_field>": null,
                "<nested_field>": {
                    "<sub_field>": null
                }
            },
            "mappings": {
                "<param_name>": "<path_in_payload_template>",
                "<param_name>": "<nested_field>.<sub_field>"
            },
            "cleanup": {
                "removeNulls": <true|false>,
                "removeEmpty": <true|false>,
                "emptyObjectToNull": <true|false>
            }
        }
    }
}
```

200 Device Translation Created

400 CorruptedConfiguration: <custom_message>

401 Unauthorized - {"message":"failed to verify jwt"}

Managers:

`deviceTranslationManager.createAdapter()`

GET **Get device translation configuration**

Retrieves the stored translation configuration for a given device.

Headers:

`Authorization`	`Bearer <token>`	*required*

body:

```
{
    "gatewayDeviceId": "device_<UUID>"
}
```

200 {<device_translation_configuration>}

404 Not Found - X-Error: Unknown device - no config to retrieve

Managers:

`deviceTranslationManager.getConfig()`

DEL **Delete device translation configuration**

Deletes the translation configuration of a device and shuts down its adapter.

Headers:

`Authorization`	`Bearer <token>`	*required*

body:

```
{
    "gatewayDeviceId": "device_<UUID>"
}
```

200 Device Translation Config Deleted - success

404 Not Found - X-Error: Unknown device - no config to delete

Managers:

`deviceTranslationManager.deleteDeviceConfig()`

A.5 The `/command` Route

Command Request

`/command`

POST **Send a command to a device**

Dispatches a command to the specified device, translating the backend parameters into a device-specific payload, using the appropriate device protocol.

Headers:

`apikey`	`<backend_api_key>`	*required*

body:

```
{
    "gatewayDeviceId": "device_<UUID>",
    "command": "<command_name>",
    "params": {
        "<param_1>": <value_1>,
        ...
    }
}
```

200 <Device Response>

202 {"status":"queued","message":"Device unreachable - request queued for retry","deviceId":"device_<UUID>","queuedRequestId":"<request_UUID>"}

400 Bad Request: property "<property>" required

400 property "command" validation failed: wrong type: expected <type-1>, got <type-1>

403 Forbidden - IllegalOperation: Unauthorized access

502 Bad Gateway - {"error":"Device unreachable: <error_reason>, device has no queuing mechanism setup"}

A.6 The `/backendForward` Route

Device → backend communication
`/backendForward`

POST **Forwards a message to authorized backends**

Dispatches a device message to the authorized backends, given the device API key.

Headers:

`apikey`	`<device_api_key>`	*required*

body:

```
{
    <device_communication>
}
```

202 {"status":"Forwarded"}

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**