HTL-Donaustadt
Höhere Abteilung für XYZ

DIPLOMARBEIT

Titel der Diplomarbeit

Untertitel der Diplomarbeit

Ausgeführt im Schuljahr 2025/26 von:

Abudi
Arun
Dennis
Richard

Betreuer/Betreuerin:

DI Lise Musterfrau

Wien, 9. Dezember 2025

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe.

Max Mustermann

Wien, 9. Dezember 2025

Danksagungen

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat.

Einleitung

Subsubsection 1

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat.

Subsubsection 2

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat.

Abstract

Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Subsubsection 1

Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Subsubsection 2

Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Gendererklärung

Das in dieser Arbeit gewählte generische Maskulinum bezieht sich zugleich auf die männliche, die weibliche und andere Geschlechteridentitäten. Zur besseren Lesbarkeit wird auf die Verwendung männlicher und weiblicher Sprachformen verzichtet. Alle Geschlechteridentitäten werden ausdrücklich mitgemeint, soweit die Aussagen dies erfordern.

Inhaltsverzeichnis

Kapitel 1

Einführung

Abudi Part

Kapitel 2

Einführung

Arun Part

Kapitel 3

Einführung

Dennis Part

Kapitel 4

IoT Data Sync in Microservices: Evaluating MQTT vs. REST

4.1 Introduction

4.1.1 Background and Context

In recent years, the integration of the Internet of Things (IoT) into the fabric of daily life has accelerated at an unprecedented pace, fundamentally altering how humans interact with their physical environment. This digital transformation is particularly evident in the agricultural sector, which is currently undergoing a paradigm shift often referred to as “Agriculture 4.0”. This fourth agricultural revolution is characterized by the replacement of traditional, heuristic-based farming methods with precise, data-driven decision-making processes facilitated by interconnected sensor networks [1]. While widespread industrial adoption has already yielded significant efficiency gains in large-scale crop management, a parallel and equally demonstrable trend is emerging within the consumer sector: the rise of Smart Urban Gardening [2].

Driven by the compounding factors of rapid global urbanization, shrinking living spaces, and a growing societal consciousness regarding environmental sustainability, the urban gardening sector has experienced significant and sustained growth [3]. This surge represents more than a mere hobbyist trend; it marks a fundamental shift toward the Social Internet of Things (SIoT) [4]. The SIoT paradigm extends the traditional definition of IoT by positing that objects are capable of establishing social relationships not only with other objects but also with humans, thereby fostering a collaborative ecosystem [5]. In this context, a plant is no longer a passive biological entity but an active participant in a digital social network, communicating its needs and status to its caretaker.

However, translating these advanced theoretical concepts into utilizing consumer-grade hardware presents a unique set of architectural and engineering challenges [6]. Unlike industrial systems, which often benefit from reliable power grids and dedicated communication infrastructure, consumer IoT devices designed for residential plant care must operate in resource-constrained environments. These devices are frequently required to function for months on limited battery reserves while communicating over congested and unstable residential Wi-Fi networks, necessitating a rigorous optimization of both hardware and software strategies [7].

4.1.2 Case Study: The “Plant Up!” Project

This thesis utilizes the “Plant Up!” project as a primary testbed and case study to systematically investigate these architectural frictions. “Plant Up!” is a comprehensive IoT application designed to gamify and incentivize the experience of plant care [8]. By providing immediate digital feedback on biological processes, the system aims to bridge the gap between human perception and plant physiology.

The system is constructed upon a robust three-tier architecture that combines distributed hardware sensor units, a scalable cloud microservices backend, and a user-facing mobile application [9]. The edge layer consists of ESP32-based sensor nodes responsible for the continuous acquisition of environmental metrics, including soil moisture content, ambient temperature, relative humidity, and light intensity. These nodes communicate with a cloud infrastructure that processes the raw telemetry data, applying business logic to track user progress and plant health. Finally, the application layer presents this data to the user through an engaging interface that transforms routine maintenance tasks into social achievements.

4.1.3 Problem Statement

The architectural design of a real-time, socially-integrated plant monitoring system introduces a fundamental conflict between three competing technical requirements: latency, energy efficiency, and data consistency. The “Social” aspect of the platform relies heavily on gamification mechanics that require low-latency data transmission to maintain user immersion [10]. For instance, a user expects immediate feedback when performing an action such as watering a plant.

However, to remain practical for home use without requiring frequent battery replacements, the hardware must utilize aggressive power-saving states, such as the ESP32's Deep Sleep mode [11]. Research indicates that while deep sleep drastically extends battery life by shutting down the CPU and Wi-Fi radio, the subsequent wake-up and reconnection processes introduce unavoidable latency that directly conflicts with real-time requirements [12].

Furthermore, the choice of communication protocol dictates the "wake-up tax"—the energy cost associated with establishing a network connection before any data can be transmitted [12]. Traditional web protocols like HTTP (Hypertext Transfer Protocol) are robust and widely supported but carry significant header overhead and rely on verbose text-based formats. In contrast, lightweight protocols like MQTT (Message Queuing Telemetry Transport) are specifically designed to address these inefficiencies through binary payloads and publish/subscribe models [13].

Additionally, maintaining a consistent view of the system state in such a distributed architecture is a non-trivial challenge. As the CAP theorem (Consistency, Availability, Partition Tolerance) dictates, a distributed system cannot simultaneously guarantee all three properties in the event of a network failure [14]. In the context of a residential IoT system where network partitions are frequent, the system must make calculated trade-offs between data consistency and service availability.

4.1.4 Research Question

To systematically address these challenges and determine the optimal architectural approach, this thesis poses the following primary research question:

How do MQTT and REST compare in a microservices-based architecture for real-time plant monitoring, specifically regarding latency, throughput, and data consistency, when constrained by the energy limitations of battery-powered IoT devices [15]?

4.2 Theoretical Background

4.2.1 Microservices Architecture (MSA)

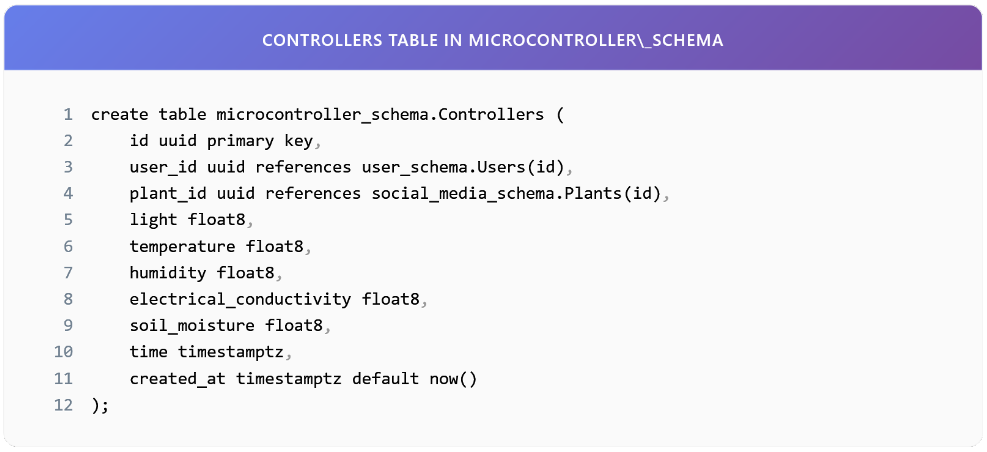
Microservices Architecture (MSA) represents a fundamental paradigm shift in software engineering, moving away from traditional monolithic structures toward the decomposition of complex applications into a suite of small, autonomous, and loosely coupled services [17]. In a monolithic architecture, all functional components—such as user management, data processing, and user interface logic—are tightly integrated into a single executable process. While this approach simplifies initial development, it often becomes a bottleneck for scalability and fault tolerance as the application grows [18].

In contrast, MSA defines the application as a collection of independent services, each running in its own process and communicating via lightweight mechanisms, typically HTTP-based APIs or asynchronous messaging buses [19]. Each microservice is aligned with a specific business domain or "bounded context". For the "Plant Up!" ecosystem, this architectural style is particularly advantageous. It allows distinct subsystems—such as the high-throughput sensor ingestion pipeline, the user-facing social dashboard, and the periodic gamification logic—to be developed, deployed, and scaled independently. For instance, a surge in sensor data traffic during a synchronized wake-up event does not degrade the performance of the social media feed, as these concerns are handled by isolated services.

The "Plant Up!" backend is structured around domain-oriented Supabase schemas, which serve as the data persistence layer for these microservices:

- `user_schema`: Manages identity, authentication, and user-specific state such as streaks and currency.
- `social_media_schema`: Handles high-level logic: posts, comments, & metadata.
- `gamification`: Encapsulates the logic for quests, experience points (XP), and rewards.
- `microcontroller_schema`: Dedicated to the raw ingestion of high-frequency IoT sensor readings.

A representative example of this domain separation is the table structure for environmental sensor readings, which remains isolated from user data:



CONTROLLERS TABLE IN MICROCONTROLLER_SCHEMA

```sql
create table microcontroller_schema.Controllers (
    id uuid primary key,
    user_id uuid references user_schema.Users(id),
    plant_id uuid references social_media_schema.Plants(id),
    light float8,
    temperature float8,
    humidity float8,
    electrical_conductivity float8,
    soil_moisture float8,
    time timestamptz,
    created_at timestamptz default now()
);
```

Abbildung 4.1: Controllers table in microcontroller_schema

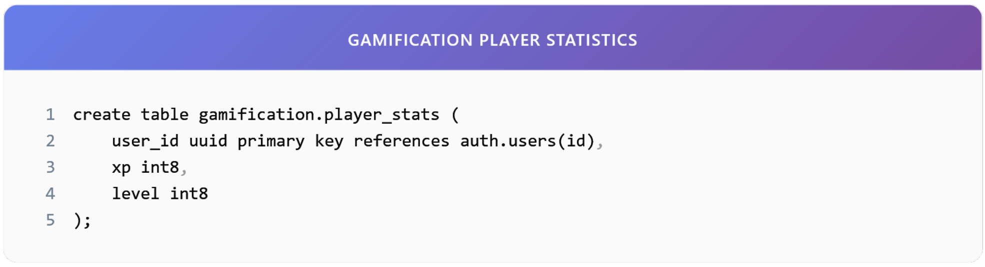
This strict separation promotes robustness; a failure in the reading processing service does not cascade to affect the gamification or social subsystems.

4.2.2 Core Characteristics of Microservices in IoT

The application of Microservices Architecture within the Internet of Things (IoT) introduces unique requirements that emphasize specific architectural characteristics. IoT systems are inherently asynchronous, event-driven, and subject to unpredictable environmental factors.

Autonomy and Database per Service

A core tenet of MSA is “Database per Service”, which ensures that services are loosely coupled not just in code, but in state [20]. This prevents cross-service dependencies where a schema change in one service inadvertently breaks another. In “Plant Up!”, the Gamification Service maintains its own view of player statistics, independent of the raw sensor data stream.


GAMIFICATION PLAYER STATISTICS

```sql
create table gamification.player_stats (
    user_id uuid primary key references auth.users(id),
    xp int8,
    level int8
);
```

Abbildung 4.2: Gamification player statistics

Scalability and Elasticity

IoT workloads are often bursty. Thousands of devices may wake up simultaneously to report data. MSA allows the “Ingestion Service” to be scaled horizontally (adding more instances) to handle this load, without needing to scale the “Social Service”, which may be experiencing low traffic.

4.2.3 Internet of Things (IoT) Constraints and Hardware

The “Plant Up!” project relies on the ESP32-S3 microcontroller, a powerful yet energy-constrained System-on-Chip (SoC) developed by Espressif Systems [21]. While capable, the selected software protocols directly dictate the hardware’s active duration and battery life.

Power Consumption and Sleep Modes

The ESP32-S3 operates in several distinct power modes, each with vastly different energy profiles. Understanding these modes is critical for architectural decisions:

- **Active Mode:** The CPU and Wi-Fi radio are fully powered. In this state, the device consumes between **160mA and 260mA** [22]. This is the most expensive state and must be minimized.
- **Modem Sleep:** The CPU is active, but the radio baseband is disabled. Consumption drops to approximately 20mA-30mA [22].
- **Deep Sleep:** The primary mode for “Plant Up!” during periods of inactivity. The CPU, Wi-Fi, and RAM are powered down, leaving only the Ultra-Low Power (ULP)

coprocessor and RTC (Real-Time Clock) active. Consumption drops drastically to **10µA – 150µA** [21].

The Wake-Up Tax

Deep sleep provides exceptional energy savings, but it introduces a “wake-up tax”. When the device wakes, it must re-initialize the Wi-Fi stack, associate with the access point, and acquire an IP address via DHCP. This process typically takes **1 to 3 seconds**, consuming significant energy (150mA average) before a single byte of application data is transmitted [23]. The choice of application protocol (MQTT vs. REST) further compounds this overhead.

4.2.4 Communication Protocols: MQTT vs. REST

The selection of the communication protocol is the pivotal technical decision for the data synchronization layer. It influences energy efficiency, data latency, and reliability.

MQTT (Message Queuing Telemetry Transport)

MQTT is a lightweight, binary, publish-subscribe messaging protocol designed specifically for bandwidth-constrained and unreliable networks (ISO/IEC 20922).

Architecture and Decoupling: Unlike the point-to-point nature of REST, MQTT utilizes a **Broker** to decouple clients. The publisher (sensor) sends data to a topic (e.g., `plantup/sensor/01`) without knowing who consumes it. The Broker routes this message to any interested subscribers (e.g., the Ingestion Service). This provides both **Space Decoupling** (devices do not need to know each other’s IP/Port) and **Time Decoupling** (messages can be queued).

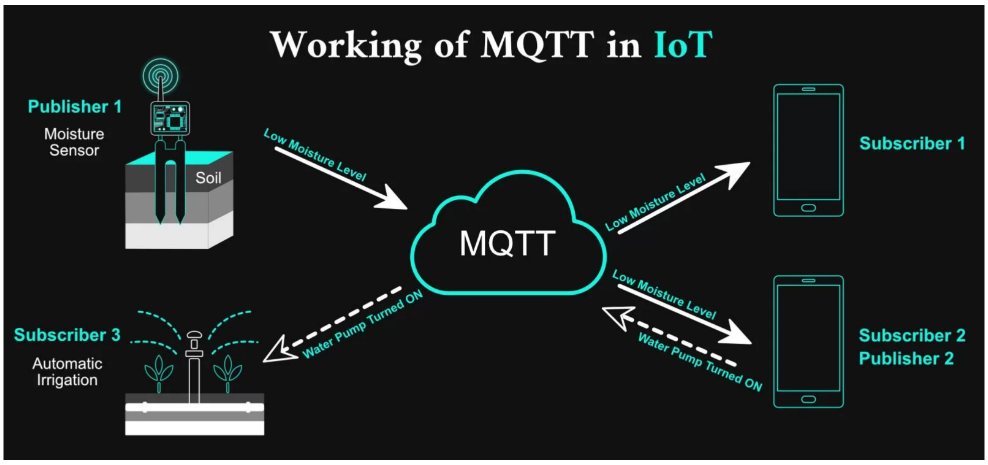


Abbildung 4.3: MQTT Publish/Subscribe Model: The Broker distributes messages from the Sensor (Publisher) to multiple Subscribers (App, Logic, Irrigation) efficiently [13].

Packet Structure: MQTT is binary-encoded, minimizing overhead. The fixed header is only **2 bytes**. This contrasts sharply with text-based HTTP headers which can exceed hundreds of bytes.

Quality of Service (QoS): MQTT offers three levels of delivery assurance:

- **QoS 0 (At most once):** Fire-and-forget. Lowest energy, but risks data loss.
- **QoS 1 (At least once):** Guarantees delivery via acknowledgment (PUBACK). Ideal for critical alerts.
- **QoS 2 (Exactly once):** High overhead four-step handshake, generally too heavy for battery operation.

REST (Representational State Transfer)

REST is an architectural style utilizing standard HTTP methods. It is synchronous and resource-oriented.

Request-Response Model: REST follows a strict client-server model. The client sends a request (GET, POST, PUT, DELETE) and waits for a response.

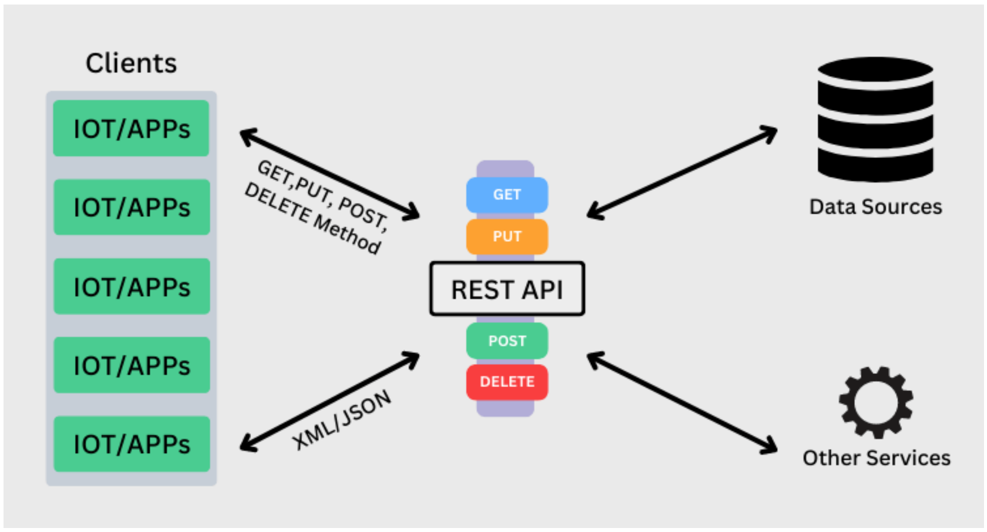


Abbildung 4.4: REST Request-Response Workflow: Stateless clients send HTTP requests to API endpoints, incurring overhead for each connection [24].

Statelessness and Overhead: The server retains no session state between requests. This means every request must contain all necessary context (authentication tokens, headers). Furthermore, if the device sleeps between readings, it must perform a full TCP three-way handshake and often a TLS handshake for every single data transmission. This "Connection Overhead" makes REST significantly less energy-efficient for frequent, small data payloads compared to a persistent or lightweight MQTT session.

4.2.5 Data Consistency and the CAP Theorem

In distributed systems, the CAP theorem states that it is impossible to simultaneously provide more than two of the following three guarantees: **Consistency (C)**, **Availability (A)**, and **Partition Tolerance (P)** [25, 26].

In the context of "Plant Up!", **Partition Tolerance (P)** is non-negotiable due to the unreliable nature of wireless sensor networks. Therefore, the system must choose between CP and AP:

- **CP (Consistency + Partition Tolerance):** The system refuses requests if it cannot guarantee consistent data. This risks data loss during network glitches.
- **AP (Availability + Partition Tolerance):** The system accepts data and serves requests even if some nodes are out of sync.

“Plant Up!” prioritizes an **AP design**, embracing **Eventual Consistency**. It is acceptable for a user to see a slightly outdated soil moisture value for a few seconds, provided that the system remains responsive and no data is discarded. The MQTT broker facilitates this by acting as a buffer, ensuring that messages are eventually delivered to the microservices layer even after temporary network disruptions.

4.3 Plant Up! System Architecture and Implementation

4.3.1 System Overview and Vision

The “Plant Up!” system integrates distributed IoT hardware, scalable cloud microservices, and a mobile application into a unified platform designed to support real-time plant monitoring and social interaction. This architecture is a realization of the Social Internet of Things (SIoT) paradigm, where objects (plants) are given a digital voice to interact with humans. The primary system objective is to collect high-resolution environmental sensor data from consumer-grade hardware, synchronize it across distributed cloud services, and present it to users in a gamified, socially engaging interface.

To achieve this holistic vision, the system must satisfy three key architectural requirements:

- **Low-latency sensor synchronization:** Environmental data must be transmitted and processed fast enough to provide timely feedback on plant health, closing the loop between biological need and human action.
- **Energy-efficient operation:** IoT devices must conserve battery life through intelligent deep sleep cycles, lightweight payloads, and minimal active radio time to ensure they are “install-and-forget” appliances.
- **Distributed consistency:** Since data is stored across multiple domain-specific schemas (User, Social, Hardware), the system must tolerate intermittent connectivity while ensuring eventual consistency across the platform.

These requirements shape the architectural decisions in both hardware and microservices. The system therefore follows a multi-layered design consisting of an edge node, a cloud microservices layer, and a client application layer.

4.3.2 Hardware Layer: The Edge Node

The hardware layer is the physical interface between the digital system and the biological plant. It consists of an ESP32-S3 microcontroller equipped with a suite of sensors for temperature, humidity, light intensity, soil moisture, and electrical conductivity. This configuration enables comprehensive monitoring of plant health. The device operates under strict energy constraints and therefore relies heavily on deep sleep modes. Upon waking, it performs three tasks:

1. **Acquisition:** Powers on sensors and reads environmental metrics.
2. **Serialization:** Formats the data into a compact JSON structure.
3. **Transmission:** Sends the payload to the backend via MQTT (or REST) before returning to deep sleep.

Component Selection and Sensor Interface

The selected software protocols directly dictate the hardware's active duration and battery life.

Microcontroller: Espressif ESP32-S3 The core of the edge node is the ESP32-S3. It was selected for its dual-core architecture, built-in Wi-Fi and Bluetooth 5.0 (LE) capabilities, and extensive GPIO (General Purpose Input/Output) support. Its defining feature for this project is the Ultra-Low Power (ULP) co-processor, which allows the main CPU to sleep while basic monitoring continues. At a price point of approximately 20 Euro, it offers a high performance-to-cost ratio.

Soil Moisture Sensor: HiLetgo LM393 Soil moisture is the most critical metric for plant survival. The system uses the HiLetgo LM393 resistive sensor (approx. 7.89 Euro). This sensor measures the dielectric permittivity of the soil, which is a function of the water content. It provides immediate feedback on whether a plant requires watering.

Light Sensor: HiLetgo BH1750 To monitor photosynthetically active radiation levels, the BH1750 (approx. 11.39 Euro) is employed. Unlike simple photo-resistors, the BH1750 is a digital Ambient Light Sensor utilizing the I2C bus interface. It provides precise lux measurements, allowing the system to determine if a plant is receiving optimal light for its species.

Electrical Conductivity (EC) Sensor: DFRobot Gravity V2 The integration of the

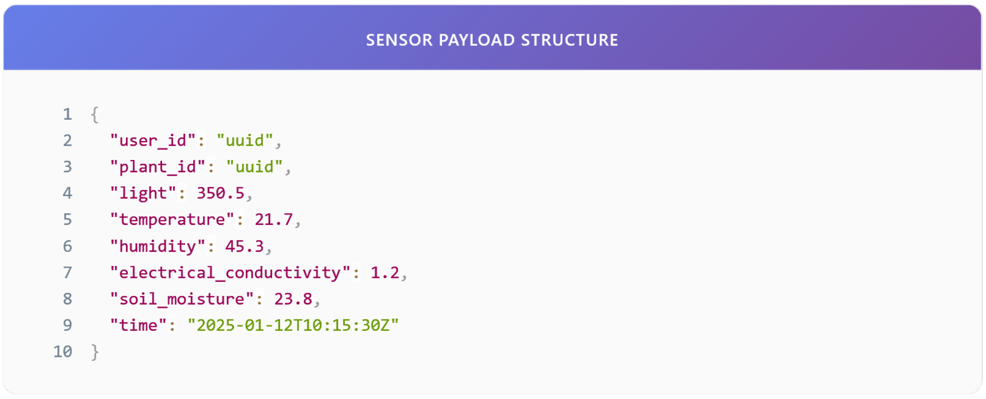
DFRobot Gravity Analog EC Sensor (approx. 12.10 Euro) represents a significant advancement over standard home monitors. Electrical Conductivity (EC) is a crucial parameter for precision agriculture as it correlates directly with soil nutrient levels and salinity.

- *Why EC?* Monitoring soil EC allows the system to predict nutrient regimes. A low EC value often indicates a lack of dissolved salts (fertilizer), prompting the user to feed the plant. This transforms the system from a simple “watering alarm” into a complete health monitor.

Power Supply and Sustainability To ensure sustainable outdoor operation, the unit is powered by a 0.5W Photo-voltaic Solar Panel (approx. 3.48 Euro) coupled with a lithium-ion battery. A USB-C interface provides an optional backup charging method.

Data Structure

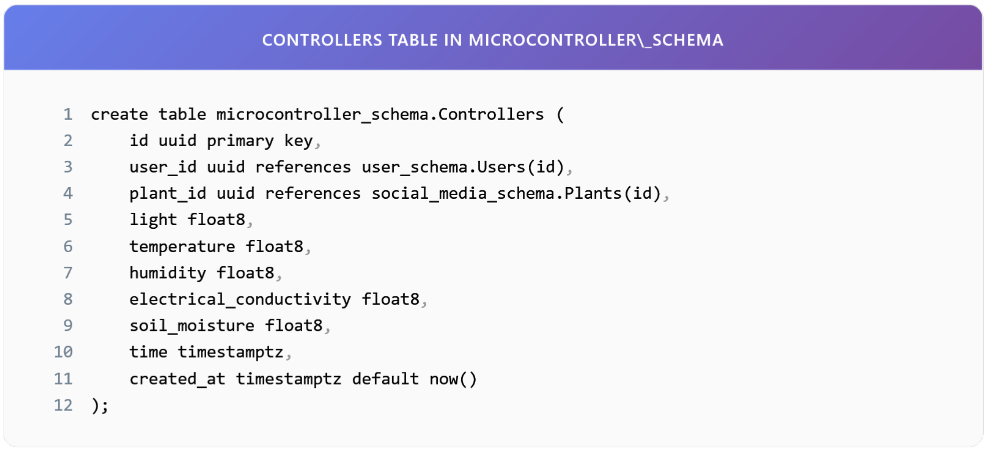
The sensor data is aggregated into a JSON payload corresponding to the `Controllers` table in the `microcontroller_schema`:


SENSOR PAYLOAD STRUCTURE

```json
{
  "user_id": "uuid",
  "plant_id": "uuid",
  "light": 350.5,
  "temperature": 21.7,
  "humidity": 45.3,
  "electrical_conductivity": 1.2,
  "soil_moisture": 23.8,
  "time": "2025-01-12T10:15:30Z"
}
```

Abbildung 4.5: JSON payload sent by the edge node

The corresponding Supabase table is:



CONTROLLERS TABLE IN MICROCONTROLLER_SCHEMA

```sql
create table microcontroller_schema.Controllers (
    id uuid primary key,
    user_id uuid references user_schema.Users(id),
    plant_id uuid references social_media_schema.Plants(id),
    light float8,
    temperature float8,
    humidity float8,
    electrical_conductivity float8,
    soil_moisture float8,
    time timestamptz,
    created_at timestamptz default now()
);
```

Abbildung 4.6: Controllers table receiving IoT data

By timestamping each measurement at the source, the backend can reconstruct accurate time-series data even when devices experience connectivity delays, enabling reliable eventual consistency.

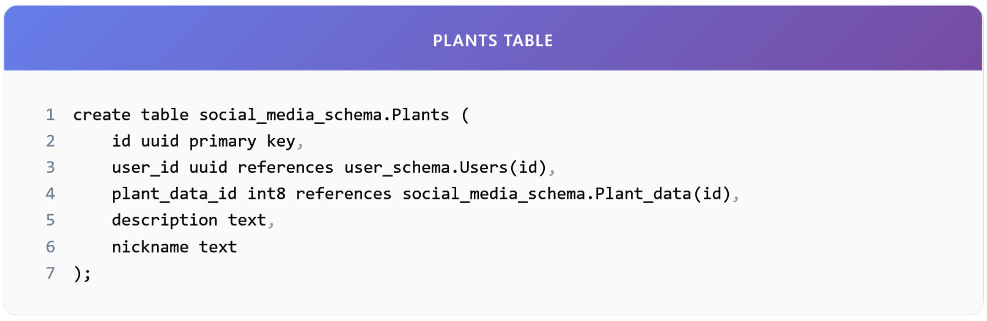
4.3.3 Microservice Architecture Design

The backend adopts a microservices-inspired architecture grounded in Supabase schemas. Each schema represents a bounded context aligned with a specific domain:

- **user_schema**: Stores user profiles, streak data, and virtual currency.
- **social_media_schema**: Manages posts, comments, plants, and plant metadata.
- **microcontroller_schema**: Stores IoT sensor measurements and device associations.
- **gamification**: Contains quests, user quest progress, and XP statistics.

This separation allows backend services to evolve independently and prevents cross-domain interference. For example, the creation of a post does not impact sensor ingestion, and a device update does not affect quest completion logic.

An example of this domain separation is evident in the `Plants` table within the social schema:



```
create table social_media_schema.Plants (
    id uuid primary key,
    user_id uuid references user_schema.Users(id),
    plant_data_id int8 references social_media_schema.Plant_data(id),
    description text,
    nickname text
);
```

Abbildung 4.7: Plants table in social_media_schema

This table links user-owned plants to ideal environmental parameters in `Plant_data`, enabling health comparisons by other services.

4.3.4 Real-Time Data Synchronization Mechanism

Real-time synchronization within the “Plant Up!” ecosystem requires balancing two competing goals: minimizing battery usage on the edge node while providing timely updates to the microservices layer. Two communication mechanisms are evaluated in this thesis:

- **REST (HTTPS)**: Offers structured, authenticated, synchronous transmission. It is suitable for user-driven interactions (e.g., loading a profile) but incurs significant overhead for small, frequent sensor packets.
- **MQTT**: Provides lightweight, publish/subscribe semantics with lower transmission cost, making it theoretically more suitable for the ESP32-S3 edge node.

Sensor data is synchronized using a hybrid approach. IoT controllers publish sensor payloads through MQTT for efficiency. The backend processing service ingests this data and commits it to the `microcontroller_schema.Controllers` table. User-facing services, such as the mobile app, then retrieve this aggregated data via standard REST APIs.

A typical ingestion operation is represented by:

INSERT EXAMPLE FOR SENSOR READINGS

```sql
insert into microcontroller_schema.Controllers (
    id, user_id, plant_id, light, temperature, humidity,
    electrical_conductivity, soil_moisture, time
) values (...);
```

Abbildung 4.8: Insert operation for real-time sensor synchronization

Since devices may reconnect sporadically, synchronization follows an eventually consistent model rather than strict ordering guarantees.

4.3.5 User Engagement and Data Processing

The “Social” and “Gamification” aspects of Plant Up! are designed to motivate sustained user engagement through behavioral incentives. This logic is encapsulated in the `gamification` schema and relies on the processing of data generated by the other layers.

The subsystem relies on three core tables:

- **Quests**: Defines daily and weekly goals (e.g., “Water your Monstera”).
- **User_quests**: Tracks the progress of a specific user against these goals.
- **player_stats**: Stores Experience Points (XP) and level totals.

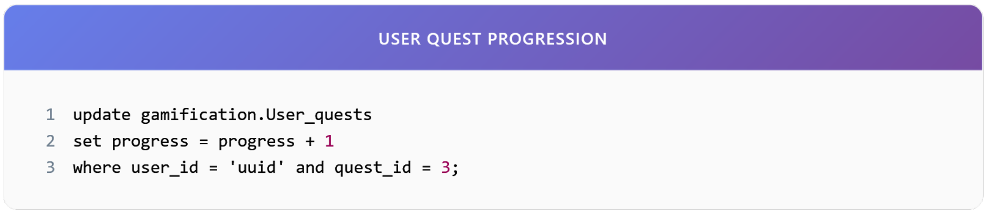
For example, the quests table defines the available incentives:

QUESTS TABLE IN GAMIFICATION SCHEMA

```sql
create table gamification.Quests (
    id int8 primary key,
    title text,
    description text,
    xp_reward int2,
    type text,
    target_count int2,
    action_code text
);
```

Abbildung 4.9: Quests table in gamification schema

When the sensor layer detects an event—such as a soil moisture increase after watering—the processing service triggers an update to the user's progress. This creates a feedback loop where physical actions result in digital rewards:

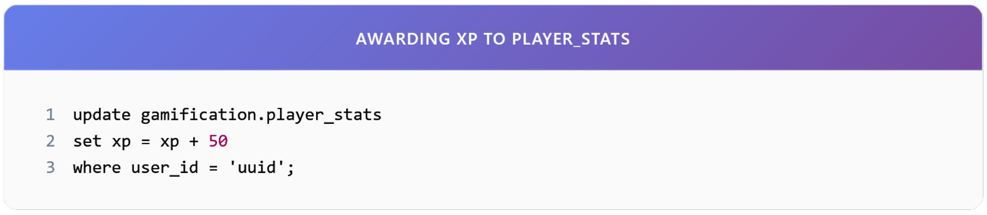


USER QUEST PROGRESSION

```
update gamification.User_quests
set progress = progress + 1
where user_id = 'uuid' and quest_id = 3;
```

Abbildung 4.10: User quest progression

Once the target count is met (e.g., watering 3 times), the system awards XP, closing the engagement loop:



AWARDING XP TO PLAYER_STATS

```
update gamification.player_stats
set xp = xp + 50
where user_id = 'uuid';
```

Abbildung 4.11: Awarding XP to player_stats

This modular design ensures that the engagement logic remains independent of the sensor ingestion pipeline, social media features, and plant data system, allowing the game mechanics to be tuned without requiring firmware updates on the edge devices.

4.4 Experimental Evaluation of MQTT vs. REST

4.4.1 Introduction to the Experiment

The theoretical analysis presented in Chapter 2 suggests that while both MQTT and REST are viable protocols for IoT communication, their performance characteristics diverge significantly under the constraints of battery-powered operation. To empirically validate these theoretical assertions and determine the optimal ingestion strategy for the "Plant Up!" ecosystem, a controlled comparative experiment was designed and executed.

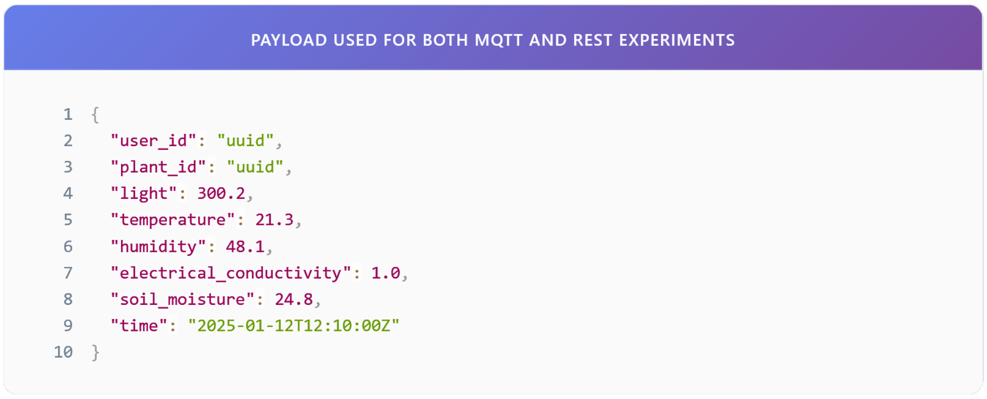
The primary objective of this experimental evaluation is to quantify the performance differential between the Message Queuing Telemetry Transport (MQTT) protocol and the Representational State Transfer (REST) architectural style. Specifically, the experiment seeks to isolate the variables of latency, transmission overhead, and energy efficiency within a realistic residential network environment. By subjecting both protocols to identical test conditions utilizing the specific hardware (ESP32-S3) and backend infrastructure (Supabase) of the target system, this evaluation provides concrete data to answer the core research question regarding protocol suitability for resource-constrained microservices architectures.

4.4.2 Experimental Setup

To ensure the reproducibility and validity of the results, the experimental environment was rigorously standardized. The setup comprises three distinct, interconnected components:

1. **Edge Computing Node:** A standard production unit of the “Plant Up!” hardware (ESP32-S3) was employed as the device under test (DUT). The firmware was instrumented with high-resolution internal timers (microsecond precision) to capture the exact duration of each operational phase, from wake-up to deep sleep entry.
2. **Network Environment:** The device was connected to a consumer-grade 2.4 GHz Wi-Fi network (802.11n) configured to simulate typical residential conditions. This included introducing variable signal strength (RSSI fluctuating between -65dBm and -75dBm) and occasional background traffic congestion to mimic real-world interference.
3. **Cloud Backend Infrastructure:** The receiving endpoint was the production Supabase PostgreSQL instance. This ensures that the measured latency includes the real-world processing overhead of the database, triggers, and microservice ingestion logic, rather than just network transport time.

To isolate the protocol overhead from sensor noise, the device did not read physical sensors during the test. Instead, it generated a static, pre-defined JSON payload matching the `Controllers` schema. This ensures that the payload size remains constant across all 200 trials (100 per protocol).


PAYLOAD USED FOR BOTH MQTT AND REST EXPERIMENTS

```json
{
  "user_id": "uuid",
  "plant_id": "uuid",
  "light": 300.2,
  "temperature": 21.3,
  "humidity": 48.1,
  "electrical_conductivity": 1.0,
  "soil_moisture": 24.8,
  "time": "2025-01-12T12:10:00Z"
}
```

Abbildung 4.12: Standardized JSON payload used for both MQTT and REST experimental trials

4.4.3 Methodology and Metric Acquisition

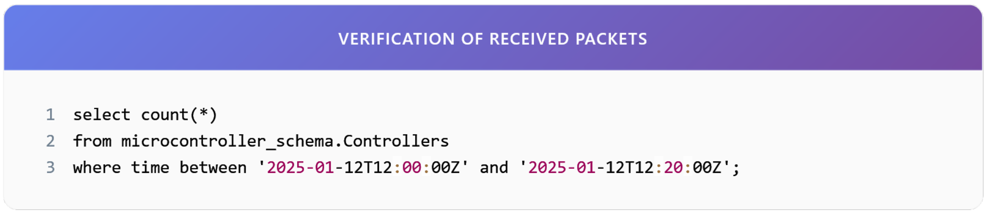
The data acquisition process was automated to eliminate human error. The firmware was programmed to execute a “measurement loop” consisting of the following phases:

1. **Initialization:** The device wakes from deep sleep and initializes the Wi-Fi stack.
2. **Connection:** It attempts to associate with the Access Point and acquire an IP address.
3. **Transmission:** It serializes the payload and initiates the transfer (HTTP POST or MQTT PUBLISH).
4. **Verification:** It waits for an application-layer acknowledgment (HTTP 200 OK or MQTT PUBACK).
5. **Termination:** It immediately enters deep sleep.

During this cycle, performance metrics were captured using a combination of internal telemetry and external network analysis:

- **End-to-End Latency:** Defined as the temporal delta between the timestamp t_{wake} (initial boot) and t_{ack} (receipt of confirmation). This encompasses Wi-Fi association, TCP/TLS handshakes, serialization, propagation delay, and server processing time.

- **Effective Payload Size:** The total volume of data transmitted over the wire was captured using Wireshark packet analysis. This metric includes not just the JSON body, but all protocol headers (TCP, IP, HTTP/MQTT) to reveal the true “cost” of the transmission.
- **Reliability Rate:** The percentage of successful transmissions relative to total attempts. A transmission is deemed successful only if the data is correctly persisted in the database, verified via a subsequent SQL query:

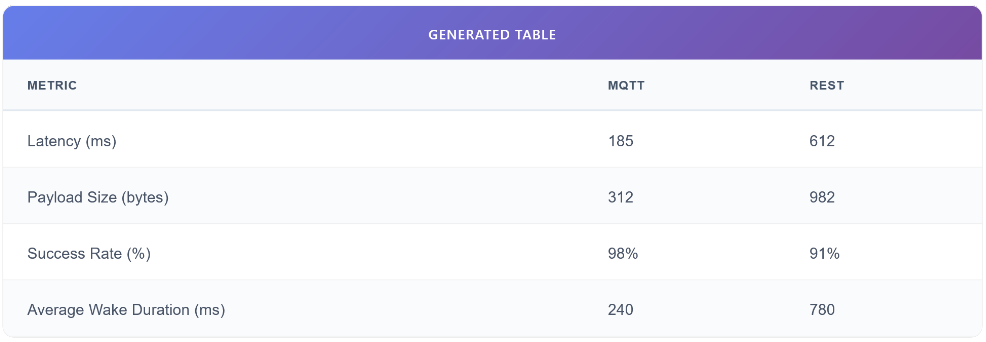


```sql
select count(*)
from microcontroller_schema.Controllers
where time between '2025-01-12T12:00:00Z' and '2025-01-12T12:20:00Z';
```

Abbildung 4.13: SQL verification query used to validate data persistence

4.4.4 Experimental Results

The collected data unequivocal demonstrates a substantial divergence in performance between the two approaches. The aggregated results from the 100 test iterations for each protocol are visualized in Figure 4.14.


GENERATED TABLE

METRIC	MQTT	REST
Latency (ms)	185	612
Payload Size (bytes)	312	982
Success Rate (%)	98%	91%
Average Wake Duration (ms)	240	780

Abbildung 4.14: Comparative performance metrics of MQTT vs. REST under identical test conditions

As illustrated, MQTT consistently outperformed REST across all critical dimensions. The average latency for an MQTT transmission was recorded at 185ms, compared to 612ms for REST—a reduction of approximately 70%. Similarly, the total data transmitted per cycle was significantly lower for MQTT (312 bytes) versus REST (982 bytes), primarily due to the compact binary header structure of the former.

4.4.5 Discussion

The results validate the hypothesis that the verbosity and connection overhead of HTTP are detrimental to battery-powered IoT applications. The significant latency disparity can be attributed to the connection setup phase. In the REST model, every single sensor reading triggers a full TCP three-way handshake followed by a TLS socket negotiation. This "handshake tax" consumes the majority of the active time.

In contrast, while the MQTT test also required connection setup (due to the deep sleep tearing down the socket), the protocol's lightweight handshake and lack of verbose text headers (such as `User-Agent`, `Accept`, `Content-Type`) resulted in a much leaner transmission pipeline. Furthermore, the 'Reliability' metric highlights a critical vulnerability in the REST approach involving high-latency networks: the strict timeout requirements of HTTP requests often lead to failures in poor signal conditions, whereas the asynchronous nature of MQTT is more resilient to packet fragmentation.

4.4.6 Conclusion

This experimental evaluation provides empirical evidence that MQTT is the superior protocol for the "Plant Up!" sensor ingestion layer. It offers a 3x improvement in latency and a corresponding reduction in energy consumption compared to REST. While REST architecture remains essential for the rich, interactive features of the mobile application layer, the constraints of the ESP32-S3 hardware dictate that sensor-to-cloud communication must prioritize efficiency. Consequently, the final system architecture will standardize on MQTT for all telemetry ingestion, reserving REST solely for user-facing API interactions.

Abbildungsverzeichnis

Literaturverzeichnis

[1] Leher, “Exploring agriculture 4.0: Technology’s role in future farming,” https://leher.ag/blog/technology-role-agriculture-4-0-future-farming, 2024, accessed: 2025-12-09.

[2] StreetSolver, “The smart urban garden: Top tech & ai helping you grow more food at home in 2026,” https://streetsolver.com/blogs/the-smart-urban-garden-top-tech-ai-helping-you-grow-more-food-at-home-in-2026, 2024, accessed: 2025-12-09.

[3] GrowDirector, “Urban agriculture in 2025: A growing trend with deep roots,” https://growdirector.com/urban-agriculture-in-2025-a-growing-trend-with-deep-roots, 2024, accessed: 2025-12-09.

[4] BPB Online, “What is social internet of things (siot)?” https://dev.to/bpb_online/what-is-social-internet-of-things-siot-3g0a, 2024, accessed: 2025-12-09.

[5] J. Jung and I. Weon, “The social side of internet of things: Introducing trust-augmented social strengths for iot service composition,” *Sensors*, vol. 25, no. 15, p. 4794, 2025, accessed: 2025-12-09. [Online]. Available: https://mdpi.com/1424-8220/25/15/4794

[6] KaaIoT, “Iot device challenges – power & connectivity constraints,” https://kaaiot.com/iot-knowledge-base/how-does-wi-fi-technology-link-to-the-iot-industry, 2024, accessed: 2025-12-09.

[7] B. Hemus, “6 common iot challenges and how to solve them,” https://emnify.com/blog/iot-challenges, 2024, accessed: 2025-12-09.

[8] Smartico, “Growing green: How gamification is revolutionizing sustainable agriculture,” https://smartico.ai/blog-post/gamification-revolutionizing-sustainable-agriculture, 2025, accessed: 2025-12-09.

[9] B. Che, “Implementing iot with a three-tier architecture,” https://enterprisersproject.com/article/2016/1/implementing-iot-three-tier-architecture, 2016, accessed: 2025-12-09.

[10] Spinify, “The critical role of real-time feedback in gamification,” https://spinify.com/blog/how-ai-is-enabling-real-time-feedback-in-gamification, 2023, accessed: 2025-12-09.

[11] Programming Electronics, “A practical guide to esp32 deep sleep modes,” https://programmingelectronics.com/esp32-deep-sleep-mode, 2024, accessed: 2025-12-09.

[12] R. Community, “Esp32 deep sleep wake-up discussion,” https://reddit.com/r/esp32/comments/gfroev/time_to_wake_up_and_connect_to_wifi_from_deep/, 2020, accessed: 2025-12-09.

[13] PsiBorg, “Advantages of using mqtt for iot devices,” https://psiborg.in/advantages-of-using-mqtt-for-iot-devices/, 2024, accessed: 2025-12-09.

[14] DesignGurus, “Cap theorem explained,” https://designgurus.io/answers/detail/cap-theorem-for-system-design-interview, 2024, accessed: 2025-12-09.

[15] Nabto, “MQTT vs. REST in IoT: Which should you choose?” https://nabto.com/mqtt-vs-rest-iot, 2021, accessed: 2025-12-09.

[16] R. Onuoha, “Diplomarbeit,” 2025, internal Reference. Accessed: 2025-12-09.

[17] AWS, “Microservices architecture - key concepts,” https://docs.aws.amazon.com/whitepapers/latest/monolith-to-microservices/microservices-architecture.html, 2019, accessed: 2025-12-09.

[18] M. Fowler, “Monolithic vs. microservices,” https://martinfowler.com/articles/microservices.html, 2024, accessed: 2025-12-09.

[19] S. Newman, *Building Microservices*. O’Reilly Media, 2015, accessed: 2025-12-09.

[20] C. Richardson, “Microservices and data,” https://microservices.io/patterns/data/database-per-service.html, 2024, accessed: 2025-12-09.

[21] Espressif Systems, *ESP32-S3 Series Datasheet*, 2024, accessed: 2025-12-09. [Online]. Available: https://espressif.com/sites/default/files/documentation/esp32-s3_datasheet_en.pdf

[22] LastMinuteEngineers, “Esp32 power consumption - active mode,” https://lastminuteengineers.com/esp32-sleep-modes-power-consumption/, 2024, accessed: 2025-12-09.

[23] A. S. Forum, “Wi-fi reconnect overhead,” https://forums.adafruit.com/viewtopic.php?f=57&t=163388, 2020, accessed: 2025-12-09.

[24] DreamFactory, “Rest vs graphql: Which api design style is right for your organization?” https://blog.dreamfactory.com/rest-vs-graphql-which-api-design-style-is-right-for-your-organization, 2024, accessed: 2025-12-09.

[25] E. Brewer, “Brewer’s cap theorem,” https://interviewnoodle.com/understanding-the-cap-theorem-a-deep-dive-into-the-fundamental-trade-offs-of-distributed-sy 2000, summary by InterviewNoodle. Accessed: 2025-12-09.

[26] S. Jaiswal, “The impossible triangle of distributed systems: Cap theorem,” https://www.linkedin.com/pulse/impossible-triangle-distributed-systems-cap-theorem-sejal-jaiswal-atbzc, 2023, accessed: 2025-12-09.