🌐 Modern IoT Stack
M1 Computer Science

Docker, MQTT, InfluxDB, Grafana & Node-RED

Complete Architecture for Professional IoT Applications

📡 ➡️ 💾 ➡️ 📊

Joseph Azar

Associate Professor - Université de Franche-Comté

Researcher FEMTO-ST

🤔 The Problem: Traditional IoT Setup

❌ Naive Approach (System Installation)

Scenario: Direct installation on the system

1. Manually install Mosquitto
2. Manually install InfluxDB
3. Manually install Grafana
4. Configure each service individually
5. Manage dependencies
6. Resolve port conflicts

Problems:

- 😱 "It works on my machine" syndrome
- 🔧 Different configurations per OS
- 🐛 Incompatible dependencies
- ⏰ Setup takes hours
- 🗑️ Difficult to clean up

✅ Modern Solution: Docker

Scenario: Docker Compose

```
# One single command
docker-compose up

# Everything starts:
# ✅ Mosquitto
# ✅ InfluxDB
# ✅ Grafana
# ✅ Node-RED
```

Benefits:

- ⚡ Setup in 30 seconds
- 🔄 Reproducible everywhere
- 🧹 Easy cleanup
- 🔒 Service isolation
- 📦 Precise versioning
- 🌐 Automatic networking

🐳 What is Docker?

> **Simple Definition:** Docker = "Magic boxes" for your applications

Analogy: Shipping Containers

- 📦 **Standardized**: Same format everywhere
- 🚢 **Portable**: Ship, train, truck
- 🔒 **Isolated**: Protected content
- 📋 **Inventoried**: Know what's inside

```dockerfile
# Image = "recipe" of container
FROM node:18

# What's inside the container
WORKDIR /app
COPY . .
RUN npm install

# How to start it
CMD ["node", "app.js"]
```

- 📦 **Image**: The "blueprint" of the container
- 🏃 **Container**: Running instance
- 🔒 **Isolated**: Own filesystem
- ⚡ **Lightweight**: Shares OS kernel

⚖️ Docker vs Virtual Machines

Feature	Virtual Machine (VM)	Docker Container
OS	Full OS (Windows, Linux...)	Uses host OS kernel (no full OS)
Size	🐘 Several GB (2-10 GB)	🐁 Few MB (10-500 MB)
Startup	🐌 Minutes (1-5 min)	⚡ Seconds (< 5s)
Isolation	🔒 Complete (virtual hardware)	🔓 Process-level (namespaces)
Performance	🐢 Significant overhead	🚀 Near native
Use Case	Different OS, max security	Microservices, CI/CD, dev

> 💡 **For IoT:** Docker is perfect - lightweight, fast to deploy, and easily orchestrates multiple services

🏗️ VM vs Docker: Architecture Explained

> 🔑 **Key Difference:** VM = Entire computer simulated | Docker = Just the application + libraries

```
Virtual Machine:                    Docker Container:

┌─────────────────────┐             ┌─────────────────────┐
│   App A   │  App B  │             │   App A   │  App B  │
├───────────┴─────────┤             ├───────────┴─────────┤
│    Guest OS (Linux) │             │   Libraries only    │
│    (2-10 GB)        │             │   (10-500 MB)       │
├─────────────────────┤             ├─────────────────────┤
│    Hypervisor       │             │   Docker Engine     │
├─────────────────────┤             ├─────────────────────┤
│    Host OS          │             │   Host OS (kernel)  │ ← Shared!
└─────────────────────┘             └─────────────────────┘
```

❌ Virtual Machine:
- Includes entire Guest OS
- Duplicates OS for each VM
- Heavy (GBs per VM)
- Slow boot (minutes)

✅ Docker Container:
- Shares host OS kernel
- Only app + dependencies
- Lightweight (MBs per container)
- Fast boot (seconds)

📦 Images vs Containers

> **Class Analogy:** Image = Class, Container = Instance

🖼️ Docker Image

- 📋 **Template** - read-only
- 🎂 **Layers** - stacked
- 📦 **Immutable** - never changes
- 🏭 **Build** once, run anywhere

```
# List images
docker images

# Download an image
docker pull nginx:latest

# Build an image
docker build -t myapp:1.0 .
```

🏃 Docker Container

- ▶️ **Instance** - running
- ✏️ **Writable** - can be modified
- 🔄 **Ephemeral** - can be destroyed
- 📊 **State** - running, stopped, paused

```
# List active containers
docker ps

# Create and run container
docker run -d -p 80:80 nginx:latest

# Stop a container
docker stop my_container
```

📦 Image vs Container Example

```
Image "node:18"          Container "app1"         Container "app2"

┌──────────────┐         ┌──────────────┐         ┌──────────────┐
│   Node.js    │  ===>   │  app.js      │         │  api.js      │
│   npm        │  run    │  PORT 3000   │         │  PORT 4000   │
│   base OS    │         │  RUNNING     │         │  RUNNING     │
└──────────────┘         └──────────────┘         └──────────────┘

   (immutable)             (instance #1)            (instance #2)
```

Key Points:

- One **image** can spawn multiple **containers**
- Containers are **isolated** from each other
- Each container has its own **writable layer**
- Image remains **unchanged** when containers run

💾 Docker Volumes: Data Persistence

> ⚠️ **Problem:** When you delete a container, ALL its data is lost!

❌ Without Volume

```
# Launch InfluxDB
docker run influxdb:1.8

# Create data
# ... write to DB

# Delete the container
docker rm -f container_id

# 😱 All data is gone!
```

💾 Docker Volumes: Solution

✅ With Volume

```
# Launch with volume
docker run -v influxdb_data:/var/lib/influxdb influxdb:1.8

# Create data
# ... write to DB

# Delete the container
docker rm -f container_id

# ✅ Data still there!
# Relaunch with same volume
docker run -v influxdb_data:/var/lib/influxdb influxdb:1.8
```

> **Result:** Persistent data survives container restarts!

📁 Types of Docker Volumes

Type	Syntax	Location	Use Case
Named Volume	`volumeName:/path`	Managed by Docker	🗄 Persistent data (DB)
Bind Mount	`./host/path:/container/path`	Absolute path on host	📝 Config, source code
tmpfs	`--tmpfs /path`	RAM memory	⚡ Temporary data

📖 Concrete Examples

```yaml
# docker-compose.yml
services:
  influxdb:
    volumes:
      - influxdb_data:/var/lib/influxdb  # Named volume

  mosquitto:
    volumes:
      - ./mosquitto.conf:/mosquitto/config/mosquitto.conf  # Bind mount

volumes:
  influxdb_data:  # Volume declaration
```

🗑️ What Happens When You Delete?

When you delete a container:

- 🗑️ **Container**: Deleted with its internal data
- ✅ **Volume**: Remains intact and can be reused
- 📦 **Image**: Still there, unchanged

```
# Example workflow
docker run -v mydata:/data myapp      # Create container with volume
# ... work with data ...
docker rm -f myapp_container          # Delete container

# Volume "mydata" still exists!
docker volume ls                       # See all volumes
docker run -v mydata:/data myapp      # Reattach to same data
```

💡 **Best Practice:** Always use volumes for important data (databases, configs, logs)

🎼 Docker Compose

Orchestrating Multiple Containers Made Easy

🐳 + 🐳 + 🐳 = 🎼

🎼 What is Docker Compose?

Definition: Tool to define and run multi-container applications with a YAML file

❌ Without Docker Compose

```
docker network create iot-network
docker run -d --name mosquitto --network iot-network -p 1883:1883 \
  -v ./mosquitto.conf:/mosquitto/config/mosquitto.conf eclipse-mosquitto
docker run -d --name influxdb --network iot-network -p 8086:8086 \
  -v influxdb_data:/var/lib/influxdb influxdb:1.8
docker run -d --name grafana --network iot-network -p 3000:3000 \
  -v grafana_data:/var/lib/grafana grafana/grafana
docker run -d --name nodered --network iot-network -p 1880:1880 \
  -v nodered_data:/data nodered/node-red
```

Problems: Complex, repetitive, error-prone

✅ With Docker Compose

```yaml
# docker-compose.yml
services:
  mosquitto:
    image: eclipse-mosquitto
    ports:
      - "1883:1883"
    volumes:
      - ./mosquitto.conf:/mosquitto/config/mosquitto.conf

  influxdb:
    image: influxdb:1.8
    ports:
      - "8086:8086"
    volumes:
      - influxdb_data:/var/lib/influxdb

  grafana:
    image: grafana/grafana
    ports:
      - "3000:3000"
    volumes:
      - grafana_data:/var/lib/grafana

  nodered:
    image: nodered/node-red
    ports:
      - "1880:1880"
    volumes:
      - nodered_data:/data

volumes:
  influxdb_data:
```

```
  grafana_data:
  nodered_data:
```

```
# One command!
docker-compose up -d
```

🛠️ Essential Docker Compose Commands

Command	Description	Use Case
`docker-compose up`	Create and start all services	🚀 First time or after changes
`docker-compose up -d`	Start in background (detached)	👻 Free up terminal
`docker-compose down`	Stop and remove containers	🛑 Cleanup (keeps volumes)
`docker-compose down -v`	Stop and remove volumes too	🗑️ Complete reset
`docker-compose ps`	List active containers	👀 Monitoring
`docker-compose logs`	View logs of all services	🐛 Debugging
`docker-compose build`	Rebuild images	🔨 After Dockerfile change
`docker-compose restart`	Restart services	🔄 After config change

🌐 Docker Compose: Automatic Networking

Magic: Docker Compose automatically creates a private network for your services!

```yaml
services:
  mosquitto:
    image: eclipse-mosquitto
    # Accessible via "mosquitto:1883"

  nodered:
    image: nodered/node-red
    depends_on:
      - mosquitto
    # Node-RED can connect via: mqtt://mosquitto:1883
```

✅ **Benefits:**

- 🏷️ Service name = hostname
- 🔒 Isolated from outside world
- ⚡ Fast internal communication
- 📡 Built-in DNS

🌐 Network Diagram

```
┌─────────────────────────────────────────┐
│  Docker Network: iot_default            │
│                                         │
│  ┌───────────┐        ┌───────────┐    │
│  │mosquitto  │<──────>│ nodered   │    │
│  │:1883      │        │:1880      │    │
│  └───────────┘        └───────────┘    │
│       │                    │            │
│       │                    │            │
│       └─>┌───────────┐<────┘            │
│          │ influxdb  │                  │
│          │:8086      │                  │
│          └───────────┘                  │
│                │                        │
│          ┌───────────┐                  │
│          │ grafana   │                  │
│          │:3000      │                  │
│          └───────────┘                  │
└─────────────────────────────────────────┘
        │ Port mapping
        ▼
    Host Machine
    localhost:1883 -> mosquitto:1883
    localhost:3000 -> grafana:3000
```

```javascript
// In Node.js code running inside Docker Compose
const mqtt = require('mqtt');

// ✅ CORRECT - use service name
const client = mqtt.connect('mqtt://mosquitto:1883');
```

🧹 Docker Cleanup: Prune Commands

> ⚠️ **Problem:** Docker accumulates data (images, containers, volumes) that takes up space

Command	What it cleans	⚠️ Warning
`docker container prune`	Stopped containers	✅ Safe, only stopped ones
`docker image prune`	Unused images (dangling)	✅ Safe, keeps tagged images
`docker image prune -a`	ALL unused images	⚠️ Removes tagged images too
`docker volume prune`	Unattached volumes	🚨 DATA LOSS possible!
`docker system prune`	Containers, networks, images	⚠️ Removes many things
`docker system prune -a --volumes`	EVERYTHING unused	🚨 COMPLETE RESET - Danger!

```
# Check disk space used by Docker
docker system df
```

📡 MQTT: IoT Protocol

Messaging for the Internet of Things

📱 📡 💡

📡 What is MQTT?

MQTT = Message Queuing Telemetry Transport

- 📨 Lightweight **messaging** protocol
- 🌐 Designed for **IoT** and unstable networks
- 📡 **Publish/Subscribe** pattern
- ⚡ Very low power consumption (battery friendly)

🏛️ MQTT Architecture

```
┌──────────────┐                    ┌──────────────┐
│  Publisher   │                    │ Subscriber 1 │
│  (ESP32-A)   │                    │ (Node-RED)   │
└──────┬───────┘                    └──────▲───────┘
       │ publish                           │ messages
       │ topic: sensor/temp                │
       │ message: {temp: 25}               │
       ▼                                   │
┌─────────────────────────────────────────┴─┐
│           MQTT Broker (Mosquitto)          │
│  - Receives messages                       │
│  - Routes to subscribers                   │
│  - Manages topics                          │
└──────────────────────────────────────────▲─┘
                                           │ subscribe
                                ┌──────────┴─────┐
                                │  Subscriber 2  │
                                └────────────────┘
```

📡 MQTT Roles

💡 **Three Main Roles:**
- 📤 **Publisher**: Sends messages to a topic
- 📥 **Subscriber**: Receives messages from a topic
- 🏢 **Broker**: Central server that routes messages

📁 Topic Structure

```
home/livingroom/temperature
home/livingroom/humidity
home/bedroom/temperature
sensor/esp32/a
sensor/esp32/b
sensor/esp32/led

# Wildcards:
home/+/temperature      # + = one level
sensor/#                # # = all levels
```

🔄 Publish/Subscribe Pattern

Key Principles:

- 🔌 **Decoupling**: Publisher and Subscriber don't know each other
- 📡 **Topics**: Communication channels
- 🎯 **Many-to-Many**: 1 publisher → N subscribers
- ⚡ **Asynchronous**: Non-blocking

Example: Temperature Sensor

```
// Publisher (ESP32 - sensor)
const mqtt = require('mqtt');
const client = mqtt.connect('mqtt://mosquitto:1883');

client.on('connect', () => {
    setInterval(() => {
        const temp = Math.random() * 30 + 20;
        const payload = JSON.stringify({
            temperature: temp.toFixed(2),
            humidity: 65,
            timestamp: Date.now()
        });
        client.publish('sensor/esp32/a', payload);
        console.log('Published:', payload);
    }, 10000); // Every 10s
});
```

📥 MQTT Subscriber Example

```javascript
// Subscriber (Dashboard)
const mqtt = require('mqtt');
const client = mqtt.connect('mqtt://mosquitto:1883');

client.on('connect', () => {
    // Subscribe to topic
    client.subscribe('sensor/esp32/a', (err) => {
        if (!err) {
            console.log('Subscribed to sensor/esp32/a');
        }
    });
});

client.on('message', (topic, message) => {
    const data = JSON.parse(message.toString());
    console.log('Received:', data);
    // Display on dashboard, save to DB...
});
```

✅ **Benefits:**

- ESP32 doesn't need to know who's listening
- Multiple subscribers can receive the same message
- New subscribers can join anytime

⚖️ MQTT vs HTTP: Fundamental Differences

Feature	HTTP (Request/Response)	MQTT (Pub/Sub)
Pattern	Client-Server (1:1)	Publish-Subscribe (M:N)
Connection	Short, per request	Persistent (WebSocket)
Overhead	🐘 Large headers (~200 bytes)	🐁 Very light (~2 bytes)
Real-time	❌ Polling needed	✅ Instant push
Bandwidth	🔴 High	🟢 Minimal
Battery	🔴 Drains quickly	🟢 Very efficient
Unstable Network	❌ Frequent timeouts	✅ QoS, auto-reconnect

🎯 Scenario 1: Temperature Monitoring - HTTP

❌ HTTP Approach (Polling)

```
// Dashboard must poll every 5 seconds
setInterval(async () => {
    const response = await fetch('http://esp32/temp');
    const data = await response.json();
    updateUI(data);
}, 5000);
```

Problems with HTTP Polling:

- 📈 **17,280 requests / day**
  - 12 requests/minute × 60 min × 24h
  - Constant network traffic
- ⏰ **Latency (up to 5s before update)**
  - Temperature changes at 10:00:01
  - Dashboard updates at 10:00:05 (next poll)
  - Not real-time!
- 🔋 **ESP32 battery drained**
  - HTTP server always listening
  - Processing 17K+ requests daily
  - No sleep mode possible
- 📡 **Bandwidth wasted**
  - Most polls return same data
  - HTTP headers overhead (~200 bytes each)
  - ~3.5 MB/day just for headers

🎯 Scenario 1: Temperature Monitoring - MQTT

✅ MQTT Approach (Push)

```
// Publisher (ESP32) - sends only when temperature changes
client.on('connect', () => {
    const currentTemp = readSensor();

    // Only publish if change > 0.5°C
    if (Math.abs(currentTemp - lastTemp) > 0.5) {
        client.publish('sensor/temp',
          JSON.stringify({temp: currentTemp}));
        lastTemp = currentTemp;
    }
});

// Subscriber (Dashboard) - receives instantly
client.on('message', (topic, message) => {
    const data = JSON.parse(message);
    updateUI(data);  // Updates in < 100ms
});
```

Benefits with MQTT Push:

- 📉 **Few messages/day:** ~10-50 messages (vs 17,280 HTTP) = 99.7% reduction
- ⚡ **Real-time:** Updates in < 100ms (vs up to 5s latency with polling)
- 🔋 **Battery preserved:** ESP32 sleeps between readings, months vs days
- 📡 **Optimal bandwidth:** ~100 bytes/day (vs 3.5 MB) = 35,000x efficient

🎯 Scenario 2: LED Control - HTTP Approach

❌ HTTP Approach

```
// ESP32 must be an HTTP server
app.post('/led/on', (req, res) => {
    digitalWrite(LED_PIN, HIGH);
    res.json({status: 'on'});
});
```

Problems with HTTP:

- 🌐 **ESP32 needs fixed IP address**
  - Dashboard must know exact IP
  - Difficult with DHCP networks
- 🔥 **Firewall / NAT complications**
  - ESP32 behind router not accessible
  - Need port forwarding configuration
- ⚠️ **No auto-reconnect**
  - Lost connection = manual restart
  - No built-in recovery mechanism
- 💾 **Heavy server code on ESP32**
  - HTTP server library = large memory footprint
  - Complex request parsing

🎯 Scenario 2: LED Control - MQTT Approach

✅ MQTT Approach

```
// ESP32 is just a subscriber
const mqtt = require('mqtt');
const client = mqtt.connect('mqtt://mosquitto:1883');

client.subscribe('sensor/esp32/led');

client.on('message', (topic, message) => {
    const state = message.toString() === 'true';
    digitalWrite(LED_PIN, state ? HIGH : LOW);
    console.log(`LED turned ${state ? 'ON' : 'OFF'}`);
});

// Dashboard publishes (from anywhere)
client.publish('sensor/esp32/led', 'true');
```

Benefits with MQTT:

- 🚫 **No server on ESP32**
  - ESP32 is a client (subscriber)
  - Minimal memory footprint
- 🌐 **Works behind NAT**
  - ESP32 connects OUT to broker
  - No firewall/port issues
- 🔄 **Auto-reconnect**
  - Built-in reconnection logic
  - Resilient to network issues
- ⚡ **Minimal code**
  - Simple subscribe + callback
  - Lightweight MQTT library

🤔 Why MQTT for IoT?

✅ Key Advantages for IoT

1. 🔋 **Power Efficiency**
   - Persistent connection (no repeated handshake)
   - Ultra-light messages (2 bytes header vs 200+ HTTP)
   - Intelligent keep-alive
   - ➡️ **Batteries last months/years**
2. 📶 **Unstable Networks**
   - QoS (Quality of Service): 0, 1, 2
   - Messages persisted if client disconnected
   - Automatic reconnection
   - ➡️ **Works even with unstable 3G/4G**

🎯 MQTT Scalability

3. Scalability
- 1 broker can handle millions of clients
- Hierarchical topics with wildcards
- No direct coupling publisher-subscriber
- ➡️ **Add sensors without modifying code**

📊 Concrete Numbers

Metric	HTTP	MQTT
Minimum message size	~200 bytes	~2 bytes
Connection setup	Every request	Once at startup
Notification latency	Polling dependent (5-60s)	< 100ms (real-time)
Battery consumption	🔴 High	🟢 Minimal
Bandwidth (1000 msg/h)	~200 KB	~2 KB

🦟 Mosquitto: The MQTT Broker

> **Mosquitto** = Open-source MQTT broker, lightweight and performant (Eclipse Foundation)

🔧 Basic Configuration

```
# mosquitto.conf
listener 1883
allow_anonymous true

# For production:
# listener 1883
# allow_anonymous false
# password_file /mosquitto/config/passwd
```

📦 In Docker Compose

```yaml
services:
  mosquitto:
    image: eclipse-mosquitto
    container_name: mosquitto
    ports:
      - "1883:1883"   # MQTT
    volumes:
      - ./mosquitto.conf:/mosquitto/config/mosquitto.conf
```

🧪 Testing Mosquitto

```bash
# Start the stack
docker-compose up -d

# Subscriber in one terminal
docker exec -it mosquitto \
  mosquitto_sub -h localhost -t "test/#"

# Publisher in another terminal
docker exec -it mosquitto \
  mosquitto_pub -h localhost \
    -t "test/hello" \
    -m "Hello MQTT!"

# The subscriber receives:
# Hello MQTT!
```

📊 Monitoring

```bash
# View all messages (debug)
docker exec -it mosquitto \
  mosquitto_sub -h localhost -t "#" -v

# System topics (stats)
docker exec -it mosquitto \
  mosquitto_sub -h localhost -t "\$SYS/#"
```

🔴 Node-RED

Visual Programming for IoT

🧩 ➡️ 🔄 ➡️ 📊

🔴 What is Node-RED?

Node-RED = Flow-based visual programming tool for IoT

- 🧩 **Drag & Drop**: No coding needed (or almost!)
- 🔌 **Nodes**: Reusable blocks (MQTT, HTTP, DB...)
- 🌐 **Web Interface**: Browser accessible
- ⚡ **Node.js**: Built on Node.js (extensible)

🎯 Use Cases

- 📡 **MQTT Routing**: sensor → database
- 🔄 **Transformation**: JSON → InfluxDB format
- 🧠 **Logic**: If temp > 30, send alert
- 📊 **Dashboard**: Real-time visualization
- 🔗 **Integration**: APIs, webhooks, services

🧩 Node Types

Input	MQTT in, HTTP in, Inject
Output	MQTT out, HTTP, Debug
Function	Function, Switch, Change
Storage	InfluxDB, MySQL, File
Dashboard	Chart, Gauge, Text, Button

🐳 Node-RED in Docker Compose

```yaml
services:
  nodered:
    image: nodered/node-red
    container_name: nodered
    ports:
      - "1880:1880"
    volumes:
      - nodered_data:/data  # Persist flows
    depends_on:
      - mosquitto
    environment:
      - TZ=Europe/Paris

volumes:
  nodered_data:
```

🌐 **Interface:** http://localhost:1880 after `docker-compose up`

🔄 Node-RED Flow Example

> **Goal:** Receive MQTT data → Save to InfluxDB → Display on dashboard

Visual Flow

```
┌──────────────┐    ┌──────────────┐    ┌──────────────┐
│   MQTT In    │───>│   Function   │───>│   InfluxDB   │
│ sensor/esp32 │    │  Parse JSON  │    │ Write Point  │
└──────────────┘    └──────────────┘    └──────────────┘
                           │
                           ▼
                    ┌──────────────┐
                    │  Dashboard   │
                    │    Chart     │
                    └──────────────┘
```

Function Node Code

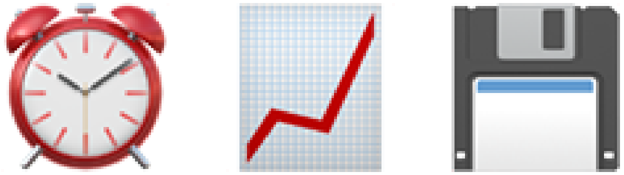
```
// Parse received JSON
const data = JSON.parse(msg.payload);

// Format for InfluxDB
msg.payload = {
    measurement: 'temperature',
    fields: {
        value: parseFloat(data.temperature),
        humidity: parseFloat(data.humidity)
    },
    tags: { sensor: 'esp32_a' },
    timestamp: new Date()
```

```
};
return msg;
```

InfluxDB

Time Series Database



📊 What is InfluxDB?

> **InfluxDB** = Database specialized for **time series** (temporal data)

🤔 What is a Time Series?

```
Time Series = Data with timestamp

┌─────────────────────┬───────┐
│ Timestamp           │ Value │
├─────────────────────┼───────┤
│ 2024-01-15 10:00:00 │ 22.5  │
│ 2024-01-15 10:01:00 │ 22.7  │
│ 2024-01-15 10:02:00 │ 22.6  │
│ 2024-01-15 10:03:00 │ 23.1  │
└─────────────────────┴───────┘

Examples:
- 📈 Sensor temperature
- 💹 Stock prices
- 📡 Network traffic
- 🏃 Server metrics
```

🎯 Why InfluxDB and not SQL/MongoDB?

SQL (MySQL, Postgres)	❌ Not optimized for time series ❌ Slow on temporal aggregations ❌ Inefficient storage ❌ No native downsampling
NoSQL (MongoDB)	❌ No temporal compression ❌ Verbose temporal queries ❌ No retention policies ⚠️ Good but not specialized
InfluxDB	✅ 100x compression (time series) ✅ Ultra-fast aggregations ✅ Auto retention policies ✅ Continuous queries ✅ InfluxQL language (SQL-like)

🧠 InfluxDB Concepts

```
Database: iot_data
   │
   ├─ Measurement: temperature
   │     ├─ Tags (indexed)
   │     │   ├─ sensor = "esp32_a"
   │     │   └─ location = "livingroom"
   │     │
   │     ├─ Fields (values)
   │     │   ├─ value = 22.5
   │     │   └─ humidity = 65
   │     │
   │     └─ Timestamp: 2024-01-15T10:00:00Z

SQL Analogy:
- Database = Database
- Measurement = Table
- Tags = Indexed columns
- Fields = Columns
- Timestamp = Primary key
```

📝 Writing Data to InfluxDB

Line Protocol Format

```
measurement,tag1=value1,tag2=value2 field1=value1,field2=value2 timestamp

# Example
temperature,sensor=esp32_a,room=living value=22.5,humidity=65 1610712000000000000
```

Node.js Example

```
const Influx = require('influx');
const client = new Influx.InfluxDB({
    host: 'influxdb',
    database: 'iot_data'
});

client.writePoints([{
    measurement: 'temperature',
    tags: {
        sensor: 'esp32_a',
        location: 'livingroom'
    },
    fields: {
        value: 22.5,
        humidity: 65
    },
    timestamp: new Date()
}]);
```

🔍 InfluxQL Queries

```sql
-- Last 24 hours
SELECT * FROM temperature
WHERE time > now() - 24h

-- Average per hour
SELECT MEAN(value) FROM temperature
WHERE time > now() - 7d
GROUP BY time(1h), sensor

-- With tags
SELECT * FROM temperature
WHERE sensor = 'esp32_a'
AND time > now() - 1h
```

🌐 Why InfluxDB for IoT?

✅ IoT-Specific Advantages

1. 📦 **Temporal Compression**
    - Specialized algorithms (Gorilla, etc.)
    - 100x compression vs classic SQL
    - Example: 1 year of data/minute = 5 MB instead of 500 MB
2. ⚡ **Aggregation Performance**
    - Ultra-fast mean/min/max/percentile calculations
    - Continuous Queries: automatic pre-calculation
    - Example: "Average temp per hour over 1 year" in < 100ms

🗑️ Retention Policies (Auto-cleanup)

- Define data lifetime
- Automatic downsampling (1min → 1h → 1d)
- Example: Raw data 7d, aggregated 1 year

```sql
-- Create retention policy
CREATE RETENTION POLICY "one_week" ON "iot_data"
  DURATION 7d
  REPLICATION 1
  DEFAULT;

-- Continuous Query for downsampling
CREATE CONTINUOUS QUERY "cq_30m" ON "iot_data"
BEGIN
  SELECT MEAN(value) AS mean_value
  INTO "iot_data"."one_year"."temperature_30m"
  FROM "temperature"
  GROUP BY time(30m), sensor
END;

-- Result:
-- Raw data: 7 days (auto-deleted)
-- 30min averages: 1 year (for charts)
```

🐳 InfluxDB with Docker

```
services:
  influxdb:
    image: influxdb:1.8
    container_name: influxdb-server
    ports:
      - "8086:8086"
    volumes:
      - influxdb_data:/var/lib/influxdb
    environment:
      - INFLUXDB_DB=iot_data
      - INFLUXDB_ADMIN_USER=admin
      - INFLUXDB_ADMIN_PASSWORD=supersecret

volumes:
  influxdb_data:
```

🧪 Test via CLI

```
# Enter container
docker exec -it influxdb-server influx

# Inside InfluxDB CLI
> USE iot_data
> SELECT * FROM temperature ORDER BY time DESC LIMIT 5
```

📊 Grafana

Visualization and Monitoring

📈 📊 🎨

📊 What is Grafana?

Grafana = Open-source visualization and monitoring platform
- 📈 **Dashboards**: Charts, gauges, tables...
- 🔌 **Data Sources**: InfluxDB, Prometheus, MySQL, etc.
- 🚨 **Alerting**: Automatic notifications
- 🎨 **Customizable**: Themes, plugins, panels

🎯 Use Cases

- 📡 **IoT Dashboards**: Real-time sensors
- 🖥️ **Infrastructure**: CPU, RAM, network
- 📊 **Business Metrics**: KPIs, analytics
- 🏭 **Industrial**: Production, quality

📊 Visualization Types

- Time Series (line, area)
- Gauge, progress bar
- Stat (single value)
- Table
- Heatmap
- Pie Chart, donut
- Geomap

🐳 Grafana with Docker

```yaml
services:
  grafana:
    image: grafana/grafana
    container_name: grafana
    ports:
      - "3000:3000"
    volumes:
      - grafana_data:/var/lib/grafana
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin

volumes:
  grafana_data:
```

🌐 **Interface:**
http://localhost:3000
Default: admin / admin

🎨 Creating a Grafana Dashboard

> **Steps:** Data Source → Dashboard → Panel → Query → Visualization

1️⃣ Configure Data Source

```
Configuration → Data Sources → Add data source
Type: InfluxDB
URL: http://influxdb:8086
Database: iot_data
Min time interval: 10s
```

2️⃣ Create Panel with Query

```sql
-- Real-time temperature
SELECT mean("value")
FROM "temperature"
WHERE $timeFilter
  AND "sensor" = 'esp32_a'
GROUP BY time($__interval) fill(null)

-- Grafana variables:
-- $timeFilter : automatic time filter
-- $__interval : adaptive interval
```

🏗️ Complete IoT Stack Architecture

```
┌─────────────────────────── PHYSICAL LAYER ─────────────────────────┐
│                                                                     │
│  ┌──────────┐    ┌──────────┐    ┌──────────┐                       │
│  │ ESP32-A  │    │ ESP32-B  │    │ ESP32-C  │                       │
│  │ Temp     │    │ LED      │    │ Motion   │                       │
│  └────┬─────┘    └────▲─────┘    └────┬─────┘                       │
└───────┼───────────────┼───────────────┼─────────────────────────────┘
        │               │               │
        │ publish       │ subscribe     │ publish
        │               │               │
┌───────┼───────────────┼───────────────┼─── MQTT LAYER ─────────────┐
│       ▼               │               ▼                             │
│                                                                     │
│  ┌─────────────────────────────────────────────────┐               │
│  │ 🦟 Mosquitto Broker - Port 1883                  │               │
│  └────┬──────────────────────────┬─────────────────┘               │
└───────┼──────────────────────────┼─────────────────────────────────┘
        │ subscribe                │ publish
        ▼                          │
┌─────────────────────────────── PROCESSING LAYER ───────────────────┐
│  ┌─────────────────────────────────────────────────┐               │
│  │ 🔴 Node-RED - Port 1880                          │               │
│  │ MQTT In → Parse → Transform → InfluxDB Write    │               │
│  └────┬────────────────────────────────────────┬───┘               │
└───────┼────────────────────────────────────────┼───────────────────┘
        │ write                                  │
        ▼                                        │
┌─────────────────────────────── STORAGE LAYER ──────────────────────┐
│  ┌─────────────────────────────────────────────────┐               │
│  │ 📊 InfluxDB - Port 8086                          │               │
│  │ Database: iot_data                              │               │
│  │ Measurements: temperature, humidity...          │               │
│  └────┬────────────────────────────────────────┬───┘               │
└───────┼────────────────────────────────────────┼───────────────────┘
        │ query                                  │
        ▼                                        │
┌─────────────────────────────── VISUALIZATION LAYER ────────────────┐
│  ┌─────────────────────────────────────────────────┐               │
│  │ 📈 Grafana - Port 3000                           │               │
│  │ Dashboards, Charts, Alerts                      │               │
```

📝 Our docker-compose.yml Explained

```
services:
  # 🦟 Mosquitto: MQTT Broker
  mosquitto:
    image: eclipse-mosquitto                  # Official image
    container_name: mosquitto                 # Fixed container name
    ports:
      - "1883:1883"                           # MQTT port (host:container)
    volumes:
      - ./mosquitto.conf:/mosquitto/config/mosquitto.conf  # Config (bind mount)

  # 🔴 Node-RED: Flow processing
  nodered:
    image: nodered/node-red
    container_name: nodered
    ports:
      - "1880:1880"                           # Web interface
    volumes:
      - nodered_data:/data                    # Flows persisted (named volume)
    depends_on:
      - mosquitto                             # Wait for mosquitto to be up

  # 📊 InfluxDB: Time series database
  influxdb:
    image: influxdb:1.8                       # Version 1.8 (InfluxQL)
    container_name: influxdb-server
    ports:
      - "8086:8086"                           # HTTP API
    volumes:
      - influxdb_data:/var/lib/influxdb       # Data persisted

  # 📈 Grafana: Visualization
  grafana:
    image: grafana/grafana
```

```yaml
    container_name: grafana
    ports:
      - "3000:3000"                      # Web interface
    volumes:
      - grafana_data:/var/lib/grafana     # Dashboards persisted

# 💾 Named volumes (managed by Docker)
volumes:
  nodered_data:
  influxdb_data:
  grafana_data:
```

🎯 Key Points

🌐 **Automatic Network:** All services can communicate via their names

💾 **Named Volumes:** Data survives `docker-compose down`

📁 **Bind Mount:** mosquitto.conf mounted from host

🔗 **depends_on:** Startup order (Node-RED after Mosquitto)

🧪 Hands-On: Launch Complete Stack

1️⃣ Start the stack

```
cd /path/to/week_1
docker-compose up -d

# Verify everything is running
docker-compose ps
```

2️⃣ Test MQTT with simulated ESP32

```
# Terminal 1 - Launch publisher (ESP32-A)
node esp32_a.js

# Terminal 2 - Launch subscriber (ESP32-B)
node esp32_b_subscriber.js

# Terminal 3 - Monitor all MQTT messages
docker exec -it mosquitto mosquitto_sub -h localhost -t "#" -v
```

🌐 Access Web Interfaces

Service	URL	Credentials
Node-RED	http://localhost:1880	None (default)
Grafana	http://localhost:3000	admin / admin
InfluxDB API	http://localhost:8086	-

🧹 Cleanup and Management

Basic commands

```
# Stop services
docker-compose stop

# Restart services
docker-compose start

# Restart specific service
docker-compose restart mosquitto

# View logs
docker-compose logs
docker-compose logs -f mosquitto
```

Cleanup

```
# Stop and remove containers
docker-compose down

# Remove containers + volumes
# ⚠️ LOSES ALL DATA
docker-compose down -v

# Rebuild images
docker-compose build
docker-compose up -d --build

# Check disk space
docker system df
```

⚠️ **Attention:**

- `docker-compose down`: Removes containers, KEEPS volumes
- `docker-compose down -v`: Removes EVERYTHING (containers + volumes)

📝 Summary

🐳 Docker

- ✅ Application containerization
- ✅ Images vs Containers
- ✅ Volumes for persistence
- ✅ Docker Compose multi-service
- ✅ Automatic networking
- ✅ Essential commands

📡 MQTT

- ✅ Pub/Sub pattern
- ✅ Mosquitto broker
- ✅ Topics and wildcards
- ✅ MQTT vs HTTP
- ✅ Why for IoT

🔴 Node-RED

- ✅ Visual programming
- ✅ Flows and nodes
- ✅ MQTT/InfluxDB integration
- ✅ Real-time dashboard

📊 InfluxDB & Grafana

- ✅ Time Series Database
- ✅ Why vs SQL/Mongo
- ✅ InfluxQL queries
- ✅ Visualization
- ✅ Dashboards and alerts

> ### 🎉 You now master a professional IoT stack!
>
> Docker + MQTT + Node-RED + InfluxDB + Grafana = 🚀

📚 Resources and Documentation

📖 Official Documentation

- 🐳 **Docker:** docs.docker.com
- 📡 **MQTT:** mqtt.org
- 🦟 **Mosquitto:** mosquitto.org
- 🔴 **Node-RED:** nodered.org
- 📊 **InfluxDB:** docs.influxdata.com
- 📈 **Grafana:** grafana.com/docs

💡 Training Projects

1. 🏠 **Smart Home:** Multi-sensors + dashboard
2. 🌱 **Plant Monitor:** Soil, light, temp
3. ⚡ **Energy Monitor:** Real-time consumption
4. 🚗 **GPS Tracker:** Location + history
5. 🏭 **Industrial:** Production monitoring

❓ Questions?

Feel free to ask your questions!

🤔 💭 💡

Joseph Azar

Associate Professor - Université de Franche-Comté

Researcher FEMTO-ST

📧 joseph.azar@univ-fcomte.fr