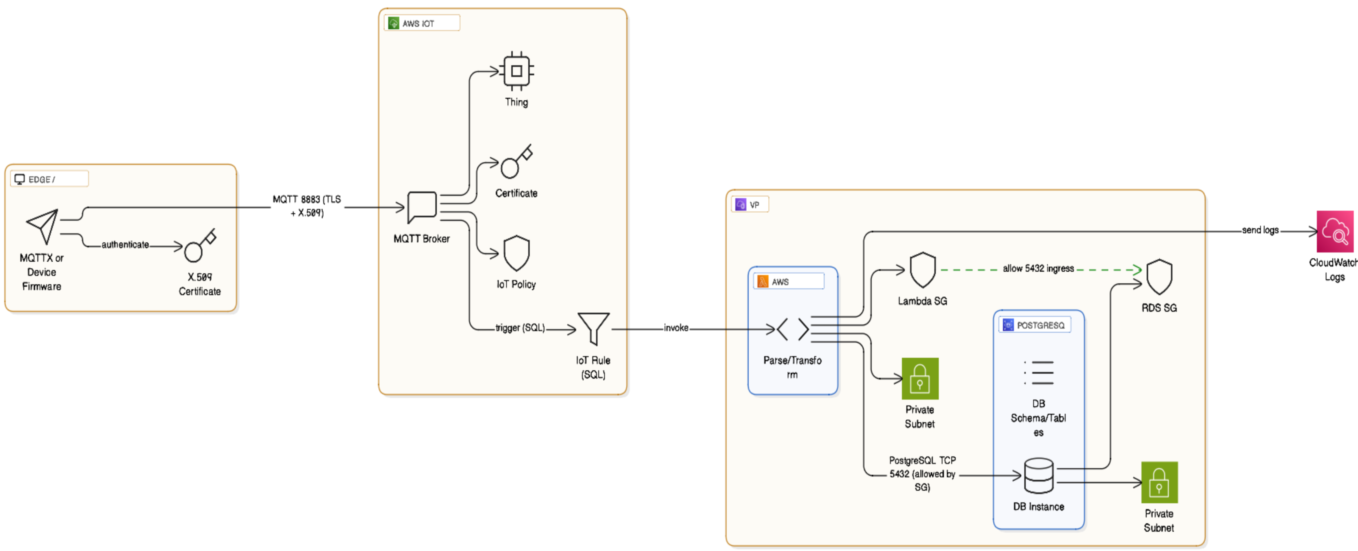
Direct Edge MQTT to AWS RDS - Storing Industrial Telemetry in PostgreSQL

Direct Edge MQTT → AWS IoT → Lambda → RDS (PostgreSQL)

Hands-On Step-by-Step Guide (Practice Version)

Final Architecture



PHASE 1 — Lock Contracts & Database Schema

Goal

Freeze interfaces so AWS setup becomes mechanical.

Step 1.1 — Lock MQTT Topic

`devices/{device_id}/telemetry`

Example:

`devices/esp32_001/telemetry`

Step 1.2 — Lock MQTT Payload

```
{
  "device_id": "esp32_001",
  "timestamp": "2026-01-03T10:15:30Z",
  "metrics": {
    "temperature": 26.4,
    "pressure": 101.2,
    "humidity": 58.1
  }
}
```

PHASE 2 — Amazon RDS (PostgreSQL)

Goal

Create a low-cost DB and prove it works **before MQTT**.

Step 2.1 — Create RDS Instance

Region

`us-east-1`

Settings

- Engine: PostgreSQL 15.x
- Template: Free tier
- DB identifier: `iot-telemetry-db`
- DB name: `iotdb`
- Username: `postgres`
- Instance: `db.t3.micro`
- Storage: 20 GB
- **Additional storage configuration: No**
- Public access: **Yes (temporary)**

Security group: **Create new**
`rds-postgres-sg`

-
- Monitoring: **Database Insights – Standard**
- Performance Insights: **Enabled (7 days)**
- Deletion protection: ❌ Off

Create database → wait until **Status = Available**

Step 2.2 — Attach RDS to SG Open Port 5432 (Local Access)

Go to:

EC2 → Security Groups → rds-postgres-sg → Inbound rules

Add rule:

- Type: PostgreSQL
- Port: 5432
- Source: **My IP**

Step 2.3 — Connect From Laptop

```
psql -h <RDS-ENDPOINT> -U postgres -d iotdb
```

Create PostgreSQL Schema

```
CREATE TABLE devices (

    id SERIAL PRIMARY KEY,

    device_id VARCHAR(64) UNIQUE NOT NULL,

    device_type VARCHAR(50),

    created_at TIMESTAMP DEFAULT NOW()

);

CREATE TABLE telemetry (

    id BIGSERIAL PRIMARY KEY,

    device_id VARCHAR(64) NOT NULL,

    ts TIMESTAMP NOT NULL,
```

```
    metrics JSONB NOT NULL,

    received_at TIMESTAMP DEFAULT NOW()

);

CREATE INDEX idx_telemetry_device_ts

ON telemetry (device_id, ts);
```

✅ Phase 1 Output

- Schema ready
- JSONB metrics supported

Run schema from Phase 1.

✅ Phase 2 Output

- DB reachable
- Tables created
- Manual inserts possible

Verify tables were created

Use curl command to ger the IP
curl https://checkip.amazonaws.com

Use command: psql -h iot-telemetry-db-practice.c8diekus4hre.us-east-1.rds.amazonaws.com -U postgres -d iotdbpractice

If it is giving timout error, update the IP in the EC2 → security Groups → rds-postgres-sg → Inbound rules

Run:

\dt

Expected output:

public | devices | table | postgres

public | telemetry | table | postgres

✅ Step 3: Verify table structure

\d devices

\d telemetry

You should see:

- `JSONB` for metrics
- `ts` timestamp
- indexes present

Check index specifically:

\di

✅ Step 4: Quick sanity insert test (optional but recommended)

```
INSERT INTO devices (device_id, device_type)
VALUES ('esp32_001', 'sensor-node');
INSERT INTO telemetry (device_id, ts, metrics)
VALUES (
  'esp32_001',
```

```
  NOW(),
  '{"temperature": 26.5, "humidity": 58}'
);
```

Verify:

```
SELECT * FROM telemetry;
```

PHASE 3 — AWS IoT Core (MQTT Ingestion)

Goal (what you are proving)

You want to confirm:

1. Your device/client (**MQTTX now, ESP32 later**) can connect to **AWS IoT Core** using **mutual TLS (mTLS)**
2. You can **publish** and **subscribe** to topics successfully
3. AWS IoT **receives messages** (MQTT Test Client shows them)

Step 3.1 — Create an IoT Thing

Where: AWS Console → **IoT Core** → **Manage** → **Things** → Create things

Do:

- Create a single thing
- **Thing name:** `esp32_001`

Why this matters:

- A “Thing” is AWS’s identity record for a device.
- It helps you manage certificates, policies, shadows, and fleet scaling later.

✅ Output: Thing `esp32_001` exists.

Step 3.2 — Create Certificate (Auto-generate) + Download files

Where: During thing creation (or later) → **Create certificate** → Auto-generate

Download these 3 files:

- `certificate.pem.crt` (device certificate / public cert)
- `private.pem.key` (device private key – keep secret)
- `AmazonRootCA1.pem` (AWS Root CA to verify AWS server)

Store them like:
`iot-certs/esp32_001/`

Why this matters (simple):

- AWS IoT uses **mTLS**, meaning:
  - **AWS proves it's AWS** to the client (using AmazonRootCA1)
  - **Client proves it's the device** to AWS (using cert + private key)

✅ Output: cert + key + CA saved locally.

Step 3.3 — Attach IoT Policy to the Certificate (and activate cert)

A) Activate the certificate

Where: IoT Core → **Security** → **Certificates**

- Select your certificate
- Set status to **ACTIVE**

Why:
If cert is INACTIVE, AWS IoT will reject the connection even if everything else is correct.

B) Create and attach a policy

Where: IoT Core → **Security** → **Policies** → Create

Policy name: `esp32_mqtt_policy`

Policy JSON:

```
{
  "Version": "2012-10-17",
  "Statement": [
      {
      "Effect": "Allow",
      "Action": [
      "iot:Connect",
      "iot:Publish",
      "iot:Subscribe",
      "iot:Receive"
      ],
      "Resource": "*"
      }
  ]
}
```

Attach policy to the certificate:

- Open certificate → **Attach policies** → select `esp32_mqtt_policy`

Why this matters:

- The certificate is like an **ID card**
- The policy is the **permissions**
- Without a policy attached, the device may connect but won't be allowed to publish/subscribe (or might be denied connection depending on setup)

✅ Output: Cert is ACTIVE + policy attached.

Note: This policy is fine for practice. Later you'll restrict it to only the correct clientId and topics.

Step 3.4 — Get AWS IoT MQTT Endpoint

Where: IoT Core → **Connect** → **Connect one device** → **MQTT**

Copy endpoint like:
`xxxxxxxx-ats.iot.us-east-1.amazonaws.com`

Important notes:

- This is a **DNS endpoint**, not an ARN.
- "Ping" may not work and that's normal.
- You connect using **mqtts on port 8883**.

✅ Output: You have the endpoint.

Step 3.5 — Configure MQTTX (mTLS connection)

Open **MQTTX** → New Connection

Connection tab

- **Protocol:** `mqtts`
- **Host:** your IoT endpoint (the `…ats.iot…amazonaws.com`)
- **Port:** `8883`
- **Client ID:** `esp32_001`
- **Username/Password:** leave empty

TLS/SSL tab (most important)

- **CA File:** `AmazonRootCA1.pem`
- **Client Certificate:** `certificate.pem.crt`
- **Private Key:** `private.pem.key`

Click **Connect**.

What “Connected” proves:

- Your cert + key are valid
- Policy allows connect
- Endpoint + TLS settings are correct

✅ Output: MQTTX status shows **Connected**.

If it fails, these are the usual causes:

- Wrong endpoint region / copied wrong endpoint
- Cert not ACTIVE
- Policy not attached to cert
- Wrong files (mixing cert/key from another thing)

Step 3.6 — Verify Publish + Subscribe (end-to-end)

A) Subscribe in MQTTX

Subscribe topic:

- `devices/+/telemetry`

Why:

- `+` matches exactly one level.
- This subscription will receive:

    - `devices/esp32_001/telemetry`
    - `devices/esp32_999/telemetry` (future devices)
- It will NOT match deeper like `devices/esp32_001/sensors/telemetry`

B) Publish from MQTTX

Publish topic:

- `devices/esp32_001/telemetry`

Payload:

- Use your “locked JSON payload” (the schema you decided in Phase 1)

Click publish.

C) Verify in AWS IoT MQTT Test Client

Where: IoT Core → **Test** → **MQTT test client**

- Subscribe there too:
    - `devices/+/telemetry`
- Publish from MQTTX again and confirm AWS shows the message.

What this proves:

- Message truly reached AWS IoT Core broker
- Subscriptions work
- Topic format matches your rule design later

✅ Phase 3 Output:

- MQTT ingestion verified
- Certificates + policy confirmed
- MQTTX ↔ AWS IoT test client both see messages

PHASE 4 — AWS IoT Rule → Lambda → RDS (PostgreSQL)

Goal

Persist MQTT telemetry data into **PostgreSQL (RDS)** using:

```
MQTT → AWS IoT Core → IoT Rule → Lambda → RDS
```

Step 4.1 — Create IoT Rule (MQTT → Lambda)

Path
AWS Console → **IoT Core** → **Message routing** → **Rules** → **Create rule**

Rule name

```
telemetry_to_rds_rule
```

IoT SQL

```
SELECT topic() AS topic, encode(*, 'base64') AS payload FROM
'devices/+/telemetry'
```

Why this SQL

- Subscribes to all device topics

Filters only telemetry messages:
`devices/<device_id>/telemetry`

Action

- **Invoke Lambda function**
- Select your Lambda (created in step 2)

✅ Rule created.

Step 4.2 — Create Lambda Function

Path

AWS Console → **Lambda** → **Create function**

Settings

- Runtime: **Python 3.11** (or 3.10)
- Function name:

```
iot-telemetry-to-rds
```

Step 4.3 — Add Environment Variables (DB config)

Lambda → **Configuration** → **Environment variables**

Key	Value
DB_HOST	RDS endpoint (DNS, not IP)
DB_NAME	iotdb
DB_USER	postgres

DB_PASS	your_password
DB_PORT	5432

Why:

- Keeps secrets out of code
- Easier to migrate environments

Step 4.4 —IMPORTANT — Make `pg8000` available in Lambda

Lambda **does NOT include pg8000 by default**.

Option A (Recommended) — Lambda Layer

1. Create a layer with `pg8000`
2. Attach it to the Lambda

Layer must include:

```
python/lib/python3.x/site-packages/pg8000
```

Option B — Zip deployment

- Zip your Lambda code + pg8000
- Upload as deployment package

✅ Lambda must import `pg8000.native` successfully.

Step 4.5 — Lambda Code (Final, tested)

```
import os

import json

import socket

import pg8000.native


# Prevent Lambda hanging forever on DB connection

socket.setdefaulttimeout(5)


def lambda_handler(event, context):

   print("Incoming event:", event)


   # Connect to PostgreSQL

   conn = pg8000.native.Connection(

       user=os.environ["DB_USER"],

       password=os.environ["DB_PASS"],

       host=os.environ["DB_HOST"],

       port=int(os.environ["DB_PORT"]),

       database=os.environ["DB_NAME"]

   )
```

```
# Handle IoT Rule wrapped payload OR direct JSON
payload = event
if "payload" in event:
    payload = json.loads(event["payload"])

# Insert telemetry
conn.run(
    """
    INSERT INTO telemetry (device_id, ts, metrics)
    VALUES (:d, :t::timestamptz, :m::jsonb)
    """,
    d=payload["device_id"],
    t=payload["timestamp"],
    m=json.dumps(payload["metrics"])
)

return {
    "status": "ok",
    "device_id": payload["device_id"]
}
```

Step 4.6 — Attach Lambda to VPC (CRITICAL STEP)

Why this is required

- RDS lives inside a **VPC**
- Lambda must be in the **same VPC** to connect

Step 4.6.1 — Create Lambda Security Group

Path

EC2 → **Security Groups** → Create security group

- Name:

```
lambda-to-rds-sg
```

- VPC: same VPC as RDS
- Inbound rules: ❌ none
- Outbound rules: ✅ allow all

Step 4.6.2 — Allow Lambda SG in RDS SG

Path

EC2 → Security Groups → `rds-postgres-sg` → **Inbound rules**

Add rule:

- Type: PostgreSQL
- Port: 5432
- Source: `lambda-to-rds-sg`

❌ Never use `0.0.0.0/0`

Step 4.6.3 — Attach VPC to Lambda

Lambda → **Configuration → VPC**

- VPC: **same VPC as RDS**
- Subnets: **2 subnets**, different AZs
- Security group: `lambda-to-rds-sg`

Step 4.6.4 — FIX IAM ROLE (mandatory)

Lambda → **Configuration → Permissions** → click **Execution role**

Attach this AWS-managed policy:

```
AWSLambdaVPCAccessExecutionRole
```

Without this:

- Lambda cannot create ENIs
- VPC attachment fails

Step 4.6.5 — Increase Lambda timeout

Lambda → **General configuration**

- Timeout: **15 seconds**

Step 4.7 — Final Networking Model (lock this in)

```
Lambda (lambda-to-rds-sg)

   |

   | TCP 5432

   v

RDS (rds-postgres-sg)
```

- Lambda SG → RDS SG
- Same VPC
- No NAT required

Step 4.8 — Test in Lambda Console (NO MQTT)

Lambda → **Test**

Test Event

```
{
  "device_id": "stm32_001",
  "timestamp": "2026-01-09T12:30:00Z",
  "metrics": {
    "temperature": 26.7,
    "humidity": 59.2,
    "pressure": 101.3
  }
}
```

Expected

- Lambda logs show DB connected
- Row inserted into DB

Verify:

```
SELECT * FROM telemetry ORDER BY ts DESC;
```

Step 4.9 — Test Using Terminal (DB verification)

From a machine that has DB access (or via bastion / public DB):

```
psql -h <RDS-ENDPOINT> -U postgres -d iotdb -p 5432
```

Then:

```
SELECT * FROM telemetry ORDER BY ts DESC;
```

✅ PHASE 4 COMPLETE OUTPUT

- IoT Rule triggers Lambda
- Lambda securely connects to RDS
- Telemetry stored as JSONB
- VPC + SG + IAM configured correctly