Telemetry & Real-Time Analytics – Proof of Concept

---

1. Objective of the POC

The primary objective of this Proof of Concept (POC) was to design and demonstrate a **real-time telemetry system** capable of continuously collecting vehicle sensor data, applying **edge data filtering**, and transmitting only meaningful information to the cloud.

This POC specifically focuses on **Battery Temperature Telemetry** using an **ESP32-based edge device**, Firebase Realtime Database as the cloud backend, and an external display/OKT507-C system for visualization.

---

2. Problem Statement

Most basic IoT implementations treat telemetry as:

- Sending all sensor data continuously to the cloud
- No differentiation between normal and abnormal data
- High bandwidth usage
- Tight coupling between embedded logic and display logic

This results in:

- Unnecessary cloud traffic
- Increased latency
- Poor scalability

A proper telemetry system must:

- Continuously collect real-time data
- Apply **edge filtering logic**
- Decide **what data stays on the vehicle** and **what goes to the cloud**
- Enable downstream systems to react in real time

---

3. Initial Approach and Design Considerations

During early design, the following telemetry approaches were evaluated:

3.1 Raw Data Streaming

• Sending every temperature sample to the cloud
• Rejected because:
– High bandwidth consumption
– No intelligence at the edge
– Poor scalability

3.2 Cloud-Only Decision Making

• Pushing all data to cloud and evaluating thresholds there
• Rejected because:
– Increased latency
– Dependency on constant connectivity
– Violates edge-compute principles

3.3 Edge-Based Filtering (Final Choice)

• Perform threshold logic on ESP32
• Send only **relevant data and alerts** to the cloud
• Chosen for efficiency and real-world applicability

Key Insight:
A good telemetry system filters data at the edge and transmits only **actionable information**.

---

4. Finalized Approach: Edge Data Filtering Telemetry

4.1 Why Edge Data Filtering

Edge data filtering was finalized because it:

• Reduces cloud bandwidth usage
• Improves real-time responsiveness
• Mimics real automotive telematics systems
• Separates sensing from visualization logic

---

5. Telemetry Definition

Item	Description
Telemetry Type	Battery Temperature Telemetry
Data Source	DS18B20 Temperature Sensor
Edge Device	ESP32
Cloud Backend	Firebase Realtime Database
Update Mode	Event-driven + periodic

6. Telemetry Logic Design

6.1 Data Collection

- Temperature sensor measures battery temperature continuously
- ESP32 reads sensor data at fixed intervals

6.2 Edge Filtering Logic

- Threshold set at **28°C**
- If temperature < 28°C
  – Data remains local
  – No alert generated
- If temperature ≥ 28°C
  – Alert flag is raised
  – Temperature value is pushed to cloud

This ensures only **meaningful telemetry** reaches the cloud.

---

7. System Architecture

7.1 Data Flow

1. Temperature sensor measures battery temperature
2. ESP32 reads sensor value
3. Edge logic evaluates threshold condition
4. Filtered telemetry is pushed to Firebase

5. OKT507-C / Display system reads cloud data
6. Alert or status is displayed in real time

7.2 Role of Each Component

Component	Responsibility
Temperature Sensor	Raw temperature measurement
ESP32	Telemetry collection + edge filtering
Firebase	Central telemetry and alert store
OKT507-C / Display	Visualization and alert display

8. Why Firebase-Based Integration Was Chosen

- Quick cloud setup for POC
- Real-time data synchronization
- Decouples embedded logic from UI logic
- Enables parallel development for display team
- Simplifies debugging and validation

Firebase acts as a **telemetry broker**, not a processing unit.

---

9. Challenges Faced

9.1 Excessive Data Transmission

- Initial design pushed all readings
- Solved using edge filtering logic

9.2 Threshold Tuning

- Required tuning to avoid unnecessary alerts
- Stabilized at 28°C for demonstration

9.3 Telemetry vs Diagnostics Confusion

• Clarified that telemetry ≠ diagnostics
• Telemetry reports values, not faults

---

10. Key Learnings

• Telemetry must be efficient, not verbose
• Edge filtering is critical for scalability
• Cloud should receive **events**, not raw noise
• Clear separation between:
– Telemetry
– Alerts
– Diagnostics

---

11. Current Status of the POC

• Battery temperature telemetry implemented
• Edge filtering logic validated
• Firebase integration complete
• OKT507-C successfully reads telemetry
• Real-time alerts visible on display in QT Environment

12.Firebase Creation – Step-by-Step Implementation

This section explains how Firebase Realtime Database is created and configured for the Telemetry & Edge Data Filtering use case.

Step 1:

1. https://console.firebase.google.com

2. Click **“Get started by setting up a Firebase project”**.

3. Enter a **Project Name**
   Example:

edge-data-filtering

1. **Step 1: Create Firebase Project**
2. Open the Firebase Console:

3. Click **Continue**.

Step 2: Google Analytics Configuration

5. You will see the **Enable Google Analytics** screen.

6. Turn **OFF Google Analytics** (recommended for simplicity).

7. Click **Create Project**.

8. Wait for **10–20 seconds** until project creation completes.

9. Click **Continue** to enter the Firebase project dashboard.

The Firebase Project is now created successfully.

Step 3: Enable Realtime Database

10. In the left sidebar, click **Build**.

11. Select **Realtime Database**.

12. Click **Create Database**.

Step 4: Database Location Setup

13. Choose the database location:

asia-southeast1

(Recommended for India)

14. Click **Next**.

Step 5: Enable Test Mode

15. Select **Start in test mode**.

16. Click **Enable**.

->Test mode allows open read/write access and should be used **only for POC or testing**.

Step 6: Configure Database Rules

17. Once the database opens, click the **Rules** tab.

18. Replace existing rules with:

```
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```

19. Click **Publish**.

->Database rules are now configured.

Step 7: Copy Realtime Database URL

20. At the top of the Realtime Database page, copy the **Database URL**.

Example:

https://edge-data-filtering-default-rtdb.asia-southeast1.firebasedatabase.app/

–> **Save this URL** — it will be used in ESP32 and OKT507-C integration.

Step 8: Summary of Completed Steps

At this stage, the following setup is complete:

✔ Firebase project created
✔ Realtime Database enabled
✔ Database rules configured
✔ Database URL copied

Step 9: Get Database Secret (For OKT507-C HTTP Access)

The **Database Secret** allows devices like OKT507-C to write data to Firebase using HTTP requests.

Step 9.1: Open Project Settings

21. In Firebase Console, click the ⚙️ **Gear icon** (top-left).

22. Select **Project settings**.

Step 9.2: Navigate to Service Accounts

23. Click the **Service accounts** tab.
24. Scroll down to **Database secrets**.

Step 9.3: Generate Database Secret

25. Click **Show** under the Secret column.
26. Copy the generated secret string.

Example:

AIzaSyBxxxxxxxxxxxxxxxxxxxxxxxx

Keep this secret private. Do not share publicly.

Step 10: Firebase HTTP URL Format

Firebase HTTP requests use the following format:

https://DATABASE_URL/path.json?auth=DATABASE_SECRET

Example

https://edge-data-filtering-default-rtdb.firebaseio.com/test.json?auth=AIzaSyBxxxx

---

Step 11: Test Firebase Connection from PC

Before connecting ESP32 or OKT507-C, test Firebase access from the PC.

Open **Ubuntu Terminal** and run:

```
curl -X POST \
https://edge-data-filtering-default-rtdb.asia-southeast1.firebasedatabase.app/test.json \
-d '{"status":"firebase connected"}'
```

Open **Firebase Console → Realtime Database → Data**.

You should see:

```
{
  "test": {
    "status": "firebase connected"
  }
}
```

If this appears, Firebase is **working correctly**.

Firebase Realtime Database is now fully configured and verified.
The system is ready to:

• Receive telemetry data
• Apply edge data filtering
• Support ESP32 and OKT507-C integration

13.Sending Temperature Data from Arduino (ESP32) to Firebase

Required Credentials & Step-by-Step Setup

To send temperature data from **Arduino (ESP32)** to **Firebase Realtime Database**, the following credentials are required:

Required Firebase Credentials

1. API Key
2. Database URL
3. Email

4. Password

PART A: How to Get Firebase API Key (Required for ESP32)

Step A1: Open Firebase Console

1. Go to:
   https://console.firebase.google.com

2. Open your project:

–>edge-data-filtering

Step A2: Go to Project Settings

3. Click the ⚙️ **Gear icon** (top-left).

4. Select **Project settings**.

Step A3: Check for Web API Key

5. You will land on the **General** tab.

6. Scroll down to **Your apps** section.

If you see:

"There are no apps in your project"

This is **normal** and means:

- Firebase project exists
- Realtime Database exists
- Authentication exists
- No app (Web / Android / iOS) is registered yet

-> **ESP32 requires a Web API Key**, so we must add a **Web App**.

PART B: Add Web App (MANDATORY for ESP32)

Step B1: Open Your Firebase Project

1. Go to:
   https://console.firebase.google.com
2. Select **edge-data-filtering**

Step B2: Add a Web App

3. Click ⚙️ **Project settings**
4. Scroll to **Your apps**
5. Click the **</> Web icon**

Step B3: Register the Web App

6. Enter **App nickname**:

->ESP32_Web_App

7. Do NOT enable **Firebase Hosting**

8. Click **Register app**

Step B4: Copy API Key

After registration, Firebase will show:

```
const firebaseConfig = {
  apiKey: "AIzaSyXXXXXXXXXXX",
  authDomain: "...",
  databaseURL: "...",
}
```

COPY ONLY THIS VALUE:

apiKey: "AIzaSyXXXXXXXXXXX"

This is your **Firebase API Key**.

Step B5: Verify API Key

9. Click **Continue to console**

10. Go back to:

⚙️ Project settings → General

11. You will now see:

Web API Key: AIzaSyXXXXXXXXXXX

API Key is now ready for ESP32

PART C: Get Firebase Email & Password (ESP32 Authentication)

Firebase ESP32 library uses **Email/Password authentication**, so a user must be created.

Step C1: Enable Email/Password Sign-In

1. Open **Firebase Console**
2. Select **edge-data-filtering**
3. In left menu → click **Authentication**
4. Click **Get started** (if not enabled)
5. Go to **Sign-in method** tab
6. Click **Email/Password**
7. Enable the toggle
8. Click **Save**

—-> Email/Password authentication enabled

Step C2: Create Firebase User

9. Go to **Authentication → Users**
10. Click **Add user**

Fill details (example):

Field	Value

Email	tejaswini123@gmail.com
Password	esp32@7

11. Click **Add user**

Firebase user created successfully

PART D: Get Firebase Database URL

IMPORTANT RULE

Do NOT copy a full path
Do NOT include /test/...

Correct format = Base URL only

Correct Database URL

https://edge-data-filtering-default-rtdb.asia-southeast1.firebasedatabase.app/

PART E: Final Credentials for ESP32 Code

You should now have **ALL required values**:

Final Required Values

```
#define API_KEY "AIzaSyCR_FwvqqMGctW9i6MNn4ZUAGcuIjxPAqQ"
#define DATABASE_URL
"https://edge-data-filtering-default-rtdb.asia-southeast1.firebasedatabase.app/"
#define USER_EMAIL "tejaswini.k@gmail.com"
#define USER_PASSWORD "esp32@123"
```

“The Arduino (ESP32) is programmed with the temperature sensor detection logic along with the required Firebase credentials (API Key, Database URL, Email, and Password). Once powered on, the ESP32 connects to the Wi-Fi network and authenticates with Firebase. After successful authentication, the device continuously reads the

temperature sensor data and automatically uploads the real-time temperature values to the Firebase Realtime Database, where they are instantly available on the cloud.”

---

14. Conclusion

This POC successfully demonstrates a **real-time telemetry system** aligned with modern automotive and industrial IoT principles. By applying edge data filtering, the system efficiently transmits only relevant information to the cloud, reducing bandwidth usage while maintaining real-time visibility.

The architecture is clean, scalable, and serves as a strong foundation for future telematics and diagnostics integration.

---

15. Future Enhancements (Optional)

- Multiple telemetry signals (voltage, SOC, current)
- Adaptive thresholds
- Telemetry batching
- CAN-based data ingestion
- Integration with diagnostics layer