Redback Racing – Weather Station Cloud Integration (2025 T3)

1. Introduction

Currently, Redback Racing captures and streams on-car telemetry through AWS services. To provide engineers with a better context, weather data must be integrated into the cloud platform. A trackside weather station will measure temperature, humidity, wind, and track surface conditions. The goal is to design a solution that is **secure, scalable, low-latency, and cost-effective**, while fitting cleanly into the existing AWS architecture.

---

2. Proposed Architecture

The proposed flow is:
Weather Station → AWS IoT Core (MQTT/TLS) → IoT Rule → Kinesis Firehose → S3 (raw) + Timestream (queries) → ECS Fargate weather-gateway-service → Redis → streaming-service → UI **(1)**

Explanation of each device:

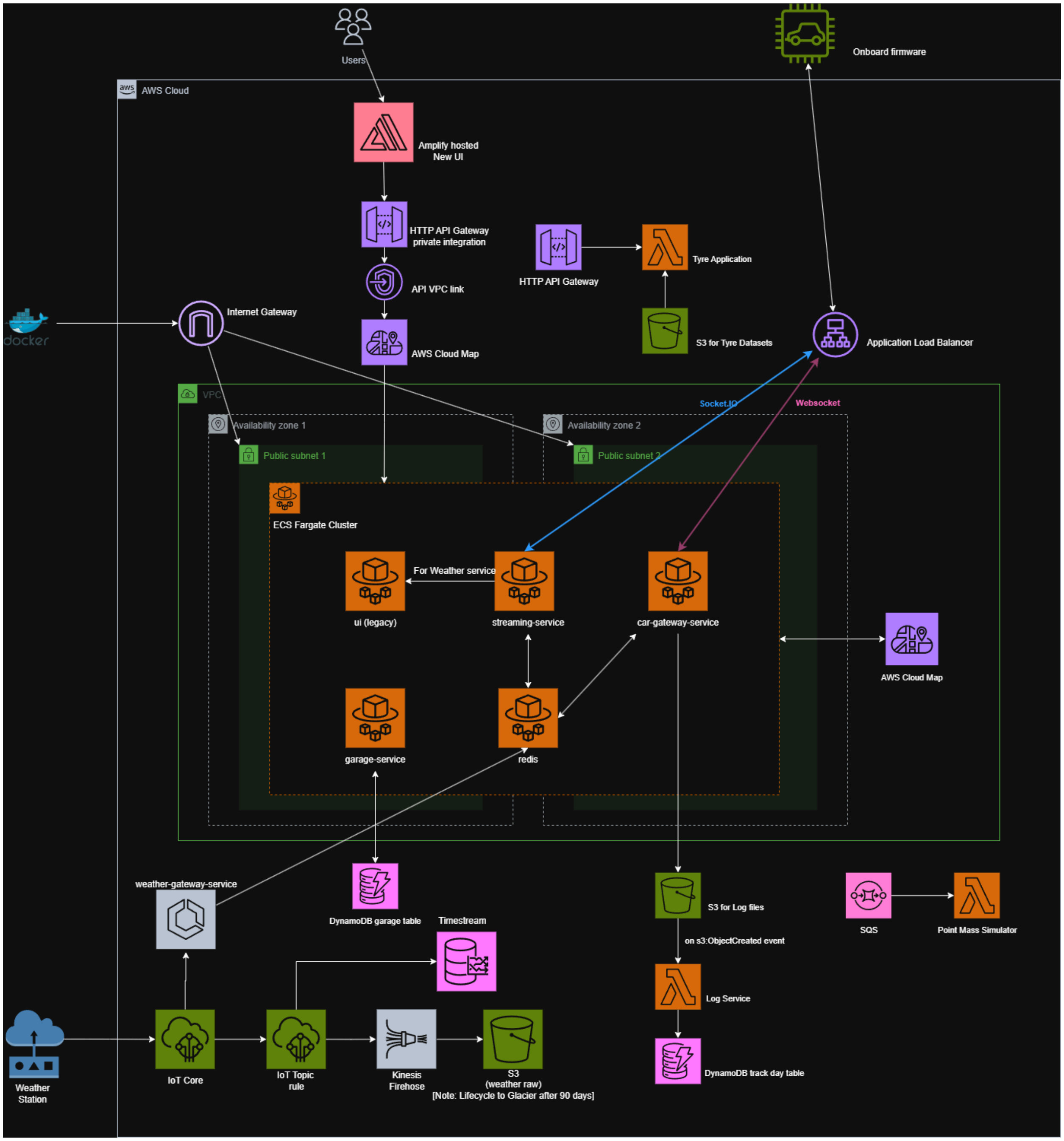
- Weather Station: publishes JSON sensor data over MQTT (topic: weather/{track_id}/{station_id}/metrics). (2)

- IoT Core: authenticates devices with X.509 certificates, terminates TLS (transport layer security), and routes messages.

- IoT Rules + Firehose: fan-out telemetry to both S3 for raw archival and Timestream for structured time-series queries.

- ECS weather-gateway-service: subscribes to MQTT topics, forwards sanitised data into Redis, and integrates with the existing streaming-service and UI.

The diagram below shows the addition of the architecture:



---

3. Protocol Choice

The weather station uses MQTT over TLS 1.2+. MQTT is a lightweight publish/subscribe protocol optimised for IoT, offering delivery guarantees through Quality of Service (QoS) levels (3). The QoS level that we will choose is QoS 1, which will ensure a reliable message delivery even on unstable LTE/Wi-Fi links, with duplicates handled downstream. This is more efficient than HTTP polling and aligns with AWS IoT Core's native features (4).

---

4. Security

- Device identity: each station has an X.509 certificate registered in IoT Core with least-privilege policies (e.g., can only publish to weather/{track}/{station}/#).
- Encryption: TLS in transit; S3 SSE-KMS and Timestream encryption at rest.
- Networking: ECS tasks run in private VPC subnets with NAT; no direct internet access.
- Audit: CloudWatch for metrics/logs; CloudTrail for API activity.
- Secrets management: IAM task roles for ECS, no static credentials embedded in devices.

---

5. Scalability & Reliability

- Managed ingest: IoT Core and Firehose scale elastically to thousands of messages/sec.
- Firehose buffering: batches efficiently before writing to S3.
- Storage tiers: S3 lifecycle policies archive data to Glacier after 90 days.
- Retention: Timestream memory store holds hot data (72h), magnetic store retains 1 year.
- Reliability: devices use local ring buffers to store readings during outages, retry on reconnect, and publish with QoS 1. Firehose has an error S3 prefix for failed records.

---

6. Performance

- Live path: MQTT → weather-gateway-service → Redis → streaming-service → UI. This supports sub-second visibility in dashboards.

- Batch path: Firehose → S3 + Timestream for analytics. Optional Firehose transformations can convert to Parquet for Athena queries.
- Caching: Redis stores latest readings (weather:latest) with TTL, ensuring fast O(1) lookups.

---

7. Cost Analysis

- IoT Core: charged per million messages; low cost at weather station scale.
- Kinesis Firehose: pay per GB ingested and transformed; minimal overhead.
- S3: cost-efficient long-term storage; lifecycle to Glacier further reduces cost.
- Timestream: billed by ingest and query; cheaper than self-hosting a time-series DB.
- ECS Fargate: billed per vCPU and GB-hour; only pays when containers are running.
  This design avoids higher-cost services like MSK/Kafka, which are unnecessary at the current scale.

---

8. Integration with Existing System

The car telemetry path remains unchanged. The weather path adds one ECS service (weather-gateway-service) into the existing Fargate cluster, forwarding live data into Redis. This allows the streaming service and UI to display weather alongside car telemetry with minimal changes. Historical data in S3/Timestream can be joined with car logs for analysis.

---

9. Terraform (Infrastructure as Code) Examples

```
provider "aws" { region = "ap-southeast-2" }

resource "aws_s3_bucket" "weather_raw" {
  bucket = "rbr-weather-raw-${var.env}"
}

resource "aws_s3_bucket_lifecycle_configuration" "weather_raw_lc" {
  bucket = aws_s3_bucket.weather_raw.id
  rule {
```

```
    id     = "tiering"
    status = "Enabled"
    transition { days = 30 storage_class = "INTELLIGENT_TIERING" }
    transition { days = 90 storage_class = "GLACIER" }
  }
}

resource "aws_timestreamwrite_database" "weather" {
  database_name = "rbr_weather_${var.env}"
}

resource "aws_timestreamwrite_table" "measurements" {
  database_name = aws_timestreamwrite_database.weather.database_name
  table_name    = "measurements"
  retention_properties {
    memory_store_retention_period_in_hours  = 72
    magnetic_store_retention_period_in_days = 365
  }
}
```

---

10. Risks & Mitigations

- Connectivity loss → station caches locally, retries with QoS 1.
- Clock drift → devices sync with NTP; server-side timestamps as backup.
- Schema evolution → versioned MQTT topics (v1/metrics), JSON schema validation in gateway.
- Scaling edge cases → S3 lifecycle and Timestream retention prevent runaway costs.

---

11. Conclusion

This design securely integrates trackside weather telemetry into Redback's AWS system. It leverages managed services (IoT Core, Firehose, S3, Timestream, ECS Fargate) to minimise operational overhead, ensure low-latency live updates for engineers, and provide a durable, queryable history for analysis. The approach balances security, scalability, performance, and cost, while fitting neatly into the existing ECS + Redis + UI ecosystem.

12. Appendix.

(1) Some other alternatives that were considered but not picked:

1. API Gateway (WebSocket/HTTP) → ECS service → S3/DB (no IoT Core)
   a. Pros: Fewer AWS services; familiar REST/WebSocket model.
   b. Cons: No device identity/cert provisioning workflow; heavier protocol for embedded; you reinvent features IoT Core already solves (authN, topic routing, QoS).
   c. Why not: We need secure device onboarding + low-power telemetry. MQTT/IoT Core is purpose-built.
2. IoT Core → Lambda → S3/Timestream (no Firehose, no ECS gateway)
   a. Pros: Really simple serverless fan-out; fewer services; easy transforms in Lambda.
   b. Cons: Harder to feed live data into the existing Redis/UI without introducing a push channel; risk of Lambda cold starts for near-real-time UX.
   c. Why not: We want a guaranteed low-latency live path into Redis while still capturing a durable history.
3. IoT Core → Kinesis Data Streams → consumers (ECS/Lambda) → S3/Timestream
   a. Pros: Strong real-time streaming semantics, multiple consumer apps, replays.
   b. Cons: More Ops and cost than Firehose for a single weather feed; extra complexity (shards, scaling policies).
   c. Why not: Weather volume is modest; Firehose is simpler/cheaper for ingestion to storage.
4. Use DynamoDB (time-series table) instead of Timestream
   a. Pros: Familiar, versatile, predictable cost.
   b. Cons: You must design/tune TS schema, TTL, and queries; lacks native TS functions/retention tiers.
   c. Why not: Timestream is built for time-series (ingest/retention/queries) and pairs well with S3 for raw.
5. Run our own TSDB (InfluxDB/Timescale) on ECS/EKS

    a. Pros: Full control, rich TS features.
    b. Cons: You own backups, scaling, HA, patching → higher operational burden.
    c. Why not: Managed services reduce toil; scope is an assessment with limited time.
6. MSK/Kafka
    a. Pros: Great for large multi-topic, multi-consumer streaming ecosystems.
    b. Cons: Expensive and operationally heavy for one station's telemetry.
    c. Why not: Overkill here.
7. IoT Greengrass or local edge compute
    a. Pros: Local rules/aggregation if connectivity is poor; can pre-filter data.
    b. Cons: Extra device complexity; not needed unless bandwidth/offline operation is a hard requirement.
    c. Why not (for now): Keep edge simple; add later if requirements expand.
8. Skip weather-gateway-service and push UI directly from IoT Core (MQTT over WebSocket)
    a. Pros: Fewer components between device and UI.
    b. Cons: Auth/session management in browser, topic authorization complexity, bypasses existing Redis/streaming-service pattern.
    c. Why not: We want to reuse the same live data path (Redis + streaming-service) the team already operates.

Summary on why I picked what I picked:

- We chose IoT Core + Firehose + S3/Timestream for a secure, managed, low-ops backbone, and a small ECS gateway to slot the feed into the existing Redis/UI for real-time UX.
- The alternatives either add ops/cost (Kafka, DIY TSDB), weaken device identity/efficiency (pure API GW/HTTP), or lose low-latency UI fit (serverless-only fan-out).

(2) Example JSON code:

```
{
 "ts": 1725345600.123,
 "air_temp_c": 28.6,
```

"humidity_pct": 54.2,
"wind_speed_ms": 7.3,
"wind_dir_deg": 210,
"track_temp_c": 35.9,
"station_id": "ws-01",
"track_id": "smp"
}

(3) QoS → Quality of Service level:

- QoS 0 → “At most once”
  - The sender sends the message once, no acknowledgment.
  - If the network drops, the message is lost.
  - Lowest overhead, fastest.
  - Use when occasional loss is fine (e.g. live temperature updates every second).
- QoS 1 → “At least once”
  - Sender retries until it gets an acknowledgment from the receiver.
  - The message might arrive more than once (duplicates possible).
  - Most common in IoT — reliable but still lightweight.
  - Good for telemetry where you’d rather have duplicates than lose data.
- QoS 2 → “Exactly once”
  - Sender and receiver go through a 4-step handshake.
  - Guarantees delivery without duplicates.
  - Highest overhead and slowest.
  - Use only for critical transactions (e.g. turning a relay on/off).

(4) Reason why it is more efficient:

MQTT was designed for IoT and telemetry, whereas HTTP was designed for request–response applications. MQTT is more efficient than HTTP polling for several reasons:

1. Connection model
   - HTTP: Every request/response opens a new TCP/TLS connection unless you manage keep-alive.

- MQTT: Maintains a long-lived TCP/TLS connection with very low overhead.

2. Message size
   - HTTP: Includes verbose headers (hundreds of bytes per request).
   - MQTT: Minimal fixed header (2 bytes), so small sensor payloads aren't drowned out by metadata.
3. Communication style
   - HTTP: Client must poll ("ask repeatedly") for new data, wasting bandwidth if nothing has changed.
   - MQTT: Uses publish/subscribe, so devices only send when data changes, and subscribers get updates instantly.
4. Delivery guarantees
   - HTTP: Either you get the response or not, but no fine-grained control.
   - MQTT: Built-in QoS levels (0, 1, 2) let you balance reliability vs. performance.
5. Power consumption
   - For IoT devices on battery, HTTP polling burns power keeping radios active.
   - MQTT's lightweight, event-driven nature is more energy-efficient.

Summary:

MQTT over TLS reduces bandwidth, latency, and power consumption compared to HTTP polling, making it far better for IoT telemetry like weather stations.