QuakeGuard

Distributed Earthquake Early Warning System

Technical Architecture & Protocol Specification

Release: **v1.2.0** (AI Emergency Reports)

Core Maintainers: @GiZano, @riccardo0731

Table of Contents

1. System Architecture & Overview

QuakeGuard is a distributed, high-throughput backend system designed for the real-time ingestion, cryptographic validation, and processing of seismic data from IoT devices[cite: 1]. It serves as the core infrastructure for an Earthquake Early Warning (EEW) system[cite: 1]. The architecture is explicitly designed to handle high-concurrency event firehosing during seismic swarms while maintaining strict security boundaries.

1.1. High-Level Topology

The infrastructure is decoupled into three primary tiers:

- **Edge Layer (IoT):** Composed of ESP32-C3 SuperMini microcontrollers interfaced with ADXL345 digital accelerometers[cite: 1]. These nodes execute on-device Digital Signal Processing (DSP) using the STA/LTA (Short Term Average / Long Term Average) algorithm[cite: 1].
- **Core Backend & Processing:** A polyglot backend architecture utilizing FastAPI (Python) as the API gateway[cite: 1]. Validated data is asynchronously offloaded to a Redis message queue (`seismic_events`) and consumed by a background worker[cite: 1]. The worker persists time-series data into a PostgreSQL/PostGIS database and triggers system-wide alerts via Redis Pub/Sub[cite: 1].
- **Client Presentation Layer:** A React Native (Expo) mobile application providing users with real-time seismograph telemetry and instantaneous critical event notifications delivered through WebSockets and native push notifications[cite: 1].

1.2. Data Plane and Control Plane

Following the v1.1.0 cloud migration, the architecture strictly separates the data and control pipelines:

- **Data Plane (Telemetry):** Flows exclusively through a HiveMQ Cloud Serverless broker on port 8883[cite: 1]. Communication is fully authenticated and TLS-encrypted[cite: 1]. A Python-based MQTT bridge (`mqtt_subscriber.py`) subscribes to the `quakeguard/telemetry` topic and forwards payloads to the internal FastAPI ingestion pipeline via HTTP POST[cite: 1].
- **Control Plane (Provisioning & Management):** Device onboarding, cryptographic handshakes, and REST retrieval operations are routed through an ngrok HTTPS tunnel, directly exposing the FastAPI endpoints (e.g., `/devices/register`)[cite: 1].

1.3. Key Design Principles

- **Zero-Trust Security:** Every telemetry payload must be cryptographically signed using an ECDSA (NIST256p) private key stored securely in the ESP32′s Non-Volatile Storage (NVS)[cite: 1]. The backend validates these signatures (SHA-256) and device timestamps to prevent spoofing and replay attacks[cite: 1].
- **Asynchronous Decoupling:** The ingestion API is strictly non-blocking. Validated payloads are immediately pushed to Redis, allowing the gateway to acknowledge the IoT device in milliseconds while background workers handle heavy database transactions and magnitude estimations[cite: 1].
- **Spatial Awareness:** Leveraging PostGIS, the system automatically assigns newly provisioned sensors to geographic macro-regions using polygon containment queries (`ST_Contains`)[cite: 1]. This enables targeted, geographically bounded alert broadcasting[cite: 1].

2. Hardware & Edge Computing (ESP32-C3)

The edge layer of QuakeGuard is designed to operate on resource-constrained microcontrollers, specifically targeting the ESP32-C3 SuperMini platform (RISC-V architecture)[cite: 1]. The node interfaces

with an ADXL345 digital accelerometer to acquire, filter, and analyze seismic data locally, transmitting only cryptographically signed anomaly reports to conserve bandwidth[cite: 1].

2.1. Hardware Configuration & Interfacing

Due to the specific physical layout of the ESP32-C3 SuperMini, the I2C bus is software-mapped to non-standard GPIO pins[cite: 1]:

- **SDA (Data):** Mapped to GPIO 7, requiring internal pull-up resistors[cite: 1].
- **SCL (Clock):** Mapped to GPIO 8, requiring internal pull-up resistors[cite: 1].
- **Power Constraints:** The ADXL345 is powered strictly via the 3.3V rail; applying 5V will result in immediate hardware damage[cite: 1].

The sensor operates at a 100Hz sampling rate (`ADXL345_DATARATE_100_HZ`) with a measurement range of $pm16$G (`ADXL345_RANGE_16_G`)[cite: 1]. To prevent I2C bus race conditions during startup, the sensor object is instantiated dynamically in memory only after the hardware bus is fully stabilized[cite: 1].

2.2. Digital Signal Processing (DSP) Pipeline

Raw acceleration data undergoes multiple stages of local processing before being evaluated:

- **High-Pass Filter (HPF):** A digital filter (`HPF_ALPHA = 0.9f`) isolates the dynamic vibration data by subtracting the static DC component (Earth's gravity, approx. 9.81 $\frac{m}{s^2}$)[cite: 1].
- **Noise Gate:** Micro-vibrations below the empirical threshold of 0.04G are clamped to zero to prevent false positives generated by electrical noise[cite: 1].
- **Dropout Protection:** The firmware automatically drops frames reporting near-zero absolute acceleration (less than 2.0 $\frac{m}{s^2}$ prior to filtering), effectively mitigating corrupted readings caused by temporary I2C wire disconnects[cite: 1].

2.3. STA/LTA Seismic Detection Algorithm

QuakeGuard implements the industry-standard Short-Term Average / Long-Term Average (STA/LTA) algorithm[cite: 1]. The implementation utilizes a custom, memory-efficient `RingBuffer` template class in C++ to maintain rolling sums, allowing $O(1)$ average calculations[cite: 1].

- **Short-Term Window (STA):** 100 samples (1 second of telemetry), representing the immediate seismic transient[cite: 1].
- **Long-Term Window (LTA):** 1000 samples (10 seconds of telemetry), representing the background noise floor[cite: 1].
- **Trigger Condition:** An earthquake is registered when the STA/LTA ratio exceeds `1.8f`, provided the STA absolute value is also above the noise floor[cite: 1].

2.4. FreeRTOS Task Architecture

The firmware utilizes FreeRTOS to decouple time-critical sensor acquisition from network latency[cite: 1].

- **`sensorTask`:** Pinned to a high priority, it wakes exactly every 10ms to poll the ADXL345 and execute the DSP pipeline[cite: 1]. Upon detecting a seismic event, it pushes a `SeismicEvent` struct to the `eventQueue`[cite: 1].
- **`networkTask`:** Operates asynchronously at a lower priority[cite: 1]. It consumes the `eventQueue`, synchronizes the event timestamp via NTP (`pool.ntp.org`), signs the payload, and dispatches the MQTT message[cite: 1]. This design ensures that network blocking or Wi-Fi reconnects never halt the continuous monitoring of the accelerometer[cite: 1].

3. Cryptographic Security & Provisioning

QuakeGuard implements a Zero-Trust security model for its IoT edge nodes[cite: 1]. To prevent rogue devices from injecting false seismic data (spoofing) or re-transmitting old events (replay attacks), the system relies on asymmetric cryptography and strict temporal validation[cite: 1].

3.1. Cryptographic Identity (ECDSA)

Upon its first boot, the ESP32-C3 utilizes the `mbedtls` library to generate a unique Elliptic Curve Digital Signature Algorithm (ECDSA) key pair using the NIST P-256 curve (`secp256r1`)[cite: 1].

- **Private Key:** Stored securely and permanently in the device's Non-Volatile Storage (NVS)[cite: 1]. It never leaves the device and is used exclusively to sign outgoing telemetry[cite: 1].
- **Public Key:** Extracted in DER format, converted to a hexadecimal string, and acts as the unforgeable cryptographic identity of the sensor within the backend database[cite: 1].

3.2. Automated Provisioning Handshake

Before transmitting any seismic data, an unregistered sensor must complete an automated handshake with the Control Plane[cite: 1]:

1. The device sends a POST request to the `/devices/register` endpoint, providing its generated `public_key_hex`, MAC address, GPS coordinates, and a hardcoded `ENROLLMENT_TOKEN`[cite: 1].
2. The backend validates the factory enrollment token to ensure the device is authorized to join the network[cite: 1].
3. Using PostGIS (`ST_Contains`), the backend spatially evaluates the provided GPS coordinates against predefined macro-regions (Zones) and assigns the sensor to the smallest containing geographic polygon[cite: 1].
4. A unique `sensor_id` is returned to the device, which saves it to NVS for all future communications[cite: 1].

3.3. Payload Authentication & Integrity

When a seismic event triggers the DSP pipeline, the firmware constructs a string containing the measured magnitude and the current NTP-synchronized Unix timestamp (format: `value:timestamp`)[cite: 1]. This string is hashed via SHA-256 and signed with the device's private key[cite: 1].

The resulting JSON payload includes the data, the timestamp, and the `signature_hex`[cite: 1]. Once received by the backend, the `validate_iot_payload` dependency pipeline enforces four security gates[cite: 1]:

- **API Key Verification:** Validates the `X-API-Key` header using a constant-time comparison (`hmac.compare_digest`) to prevent timing attacks[cite: 1].
- **Sensor Status:** Confirms the sensor ID exists and is marked as `active` in the PostgreSQL database[cite: 1].
- **Anti-Replay Protection:** Compares the device's timestamp against the server's UTC time. Payloads older than a 300-second threshold are outright rejected with a `403 Forbidden` error[cite: 1].
- **Signature Verification:** The backend utilizes the Python `cryptography` library to verify the ECDSA signature against the public key associated with that sensor[cite: 1]. The implementation robustly supports both DER-encoded signatures (native to MbedTLS) and raw concatenated $r \parallel s$ signatures[cite: 1].

4. Data Plane & Message Broker (MQTT)

With the release of v1.1.0, QuakeGuard migrated its Data Plane from a local, cleartext MQTT architecture to a robust, cloud-based infrastructure[cite: 1]. This decoupling ensures that high-frequency seismic

telemetry is handled by a dedicated message broker optimized for IoT, freeing the edge nodes from the overhead of HTTP round-trips[cite: 1].

4.1. HiveMQ Cloud Infrastructure

The core of the Data Plane is a HiveMQ Cloud Serverless broker[cite: 1].

- **Encrypted Transport:** All telemetry is transmitted over port 8883 using strict Transport Layer Security (TLS)[cite: 1].
- **Authentication:** Anonymous access (`allow_anonymous true`) has been deprecated[cite: 1]. The broker now requires explicit `MQTT_USERNAME` and `MQTT_PASSWORD` credentials for every connection[cite: 1].
- **Topic Topology:** All valid seismic anomalies are published to the unified `quakeguard/telemetry` topic[cite: 1].

4.2. Edge Node Implementation (Firmware)

On the ESP32-C3, the `networkTask` handles the MQTT transmission asynchronously[cite: 1].

- To support TLS on resource-constrained hardware, the firmware utilizes `WiFiClientSecure::setInsecure()` combined with the `PubSubClient` library[cite: 1].
- When a `SeismicEvent` is popped from the FreeRTOS queue, the firmware packages the JSON and fires the payload to the broker[cite: 1]. This "fire-and-forget" approach reduces transmission blocking time to milliseconds, ensuring the `sensorTask` is never starved of CPU cycles[cite: 1].

4.3. Internal MQTT Bridge Service

To securely ingest the MQTT data into the backend without exposing the FastAPI worker directly to the public internet, QuakeGuard employs a dedicated bridging microservice (`mqtt_subscriber.py`)[cite: 1].

- **Protocol Bridging:** Written in Python using the `paho.mqtt.client` (v2 API), the bridge acts as an authorized subscriber to the HiveMQ broker[cite: 1]. It initiates a secure connection using `client.tls_set(cert_reqs=ssl.CERT_REQUIRED)`[cite: 1].
- **Internal Routing:** Upon receiving a message on `quakeguard/telemetry`, the bridge wraps the payload and forwards it to the internal FastAPI ingestion endpoint (`/readings/`) via a standard HTTP POST request[cite: 1].
- **Security Injection:** The bridge injects the `X-API-Key` header into the HTTP request, acting as a trusted proxy between the external MQTT cloud and the internal Docker network[cite: 1]. If the API rejects the payload (e.g., due to an invalid cryptographic signature), the bridge logs the failure without crashing[cite: 1].

5. Backend Services & Event Processing

The QuakeGuard backend is engineered to handle massive telemetry spikes (firehosing) typical of seismic swarms[cite: 1]. To achieve this, the architecture strictly decouples data ingestion from data processing using a producer-consumer pattern mediated by Redis[cite: 1].

5.1. API Gateway & Asynchronous Ingestion

The primary entry point is a FastAPI application running behind an ASGI server (Uvicorn) within a Docker container[cite: 1].

- **Rate Limiting:** To protect against Denial of Service (DoS) and buggy edge nodes, the ingestion endpoint (`/readings/`) utilizes a sliding-window rate limiter backed by Redis sorted sets (`ZADD`, `ZREMRANGEBYSCORE`), restricting traffic to 50 requests per second per IP[cite: 1].
- **Queue Offloading:** Once a payload passes the strict cryptographic validation pipeline, the API does not block to perform database writes[cite: 1]. Instead, it immediately serializes the event and pushes

it to a Redis List (`seismic_events`) via an `lpush` command, returning an HTTP `202 Accepted` to the bridge in milliseconds[cite: 1].

5.2. Background Worker & Persistence

A decoupled Python worker (`worker.py`) runs in a continuous loop, consuming the `seismic_events` queue via a blocking `brpop` operation[cite: 1].

- **Database Pooling:** The worker and the API share a heavily optimized SQLAlchemy connection pool targeting a PostgreSQL/PostGIS database[cite: 1]. The engine is configured with aggressive pooling parameters (`pool_size`, `max_overflow`) to handle high-concurrency load testing without queue exhaustion[cite: 1].
- **Atomicity:** The worker saves the raw `Reading` and evaluates the alarm logic within a single, atomic database transaction (`db.commit()`), rolling back and routing failed events to a Dead Letter Queue (`seismic_events_dlq`) in case of DB errors[cite: 1].

5.3. Magnitude Estimation & Deduplication

For every processed event, the worker performs a physical magnitude estimation based on a MyShake-style MEMS calibration approach[cite: 1]. The raw Peak Ground Acceleration (PGA) is converted using the formula[cite: 1]:

$$M_{\text{IoT}} = \log_{10}(\text{PGA}_{\text{calib}}) + b$$

Where $\text{PGA}_{\text{calib}}$ accounts for the ADXL345 scale and hardware calibration constant (`K_CALIBRATION = 1.6`), and b is an empirical offset (`B_OFFSET = 3.0`)[cite: 1].

- **Thresholding:** If the estimated magnitude reaches or exceeds the critical threshold of 4.5, the worker triggers an `Alert` entity[cite: 1].
- **Redis Deduplication:** During a real earthquake, dozens of sensors in the same Zone will breach the threshold simultaneously. To prevent notification spam, the worker uses a Redis atomic check-and-set operation (`SET nx=True, ex=60`) keyed by `zone_id`[cite: 1]. This enforces a strict 60-second cooldown per geographic zone[cite: 1].
- **Outbox Pattern:** Only the first worker process that successfully acquires the Redis lock will persist the `Alert` to PostgreSQL and publish the JSON payload to the `quake_alerts` Redis Pub/Sub channel[cite: 1].

6. Mobile Client & Live Telemetry

The client-side presentation layer of QuakeGuard is a cross-platform mobile application built with React Native and the Expo framework[cite: 1]. It serves as the primary interface for users to monitor the seismic network and receive instantaneous Earthquake Early Warning (EEW) notifications[cite: 1].

6.1. Real-Time Alerting (WebSockets)

To guarantee sub-second delivery of critical alerts, the mobile app maintains a persistent, full-duplex connection to the backend via WebSockets[cite: 1].

- **Connection Management:** The `WebSocketContext` initializes a connection to the `/ws/alerts` endpoint, authenticating via a secure query parameter (`MOBILE_WS_TOKEN`)[cite: 1]. The client implements an exponential backoff strategy for automatic reconnections up to a maximum delay of 30 seconds[cite: 1].
- **Native Haptics & Notifications:** When the WebSocket receives a payload tagged as `"CRITICAL"`, the application bypasses standard UI updates and immediately triggers native hardware APIs[cite: 1]. It executes an SOS vibration pattern (`Vibration.vibrate`) and schedules a high-priority system push

notification using `expo-notifications` (`AndroidImportance.MAX`), ensuring the user is alerted even if the app is in the background[cite: 1].

6.2. Telemetry Visualization & State

The dashboard provides a live view of the network's health and recent seismic activity, relying on a robust state management architecture[cite: 1].

- **Data Fetching (TanStack Query):** Standard REST operations, such as fetching the active sensor list and the latest telemetry points (`/readings/`), are managed by React Query[cite: 1]. This provides automatic caching, background refetching (every 2 seconds for live readings), and loading state management[cite: 1].
- **Live Seismograph:** The raw telemetry data is fed into a `VictoryChart` component (`victory-native`), which renders a dynamic line chart representing the network's aggregated seismic activity in real-time[cite: 1].
- **Geospatial Map:** The `react-native-maps` library maps the network topology, displaying custom markers for each sensor based on their exact PostGIS coordinates and coloring them according to their active/offline status[cite: 1].

6.3. Resilience and Global State (Zustand)

Global application state, including alert history and user preferences, is managed using Zustand[cite: 1].

- **Alert Store:** The `useAlertStore` maintains a rolling history of the 10 most recent alerts, persisting them for the dashboard's activity log[cite: 1].
- **Offline Mode:** The `usePreferencesStore` controls a user-toggled "Offline Mode"[cite: 1]. When activated, the application cleanly and intentionally closes the active WebSocket connection and disables all React Query background polling (`enabled: !isOfflineMode`), effectively halting network traffic and conserving battery life during maintenance or low-power situations[cite: 1].

7. Deployment & Operations Guide

This section outlines the procedures for provisioning the QuakeGuard infrastructure in a local or development environment.

7.1. Prerequisites

To orchestrate the full stack, the host machine must have the following dependencies installed:

- **Backend:** Docker Engine and Docker Compose.
- **Edge (IoT):** Visual Studio Code with the PlatformIO IDE extension.
- **Mobile:** Node.js (v18+) and the Expo Go application installed on a physical smartphone.

7.2. Backend Provisioning

The backend services (PostgreSQL, Redis, FastAPI, Worker, and MQTT Bridge) are fully containerized.

1. Navigate to the backend directory:

   ```
   cd backend
   ```
2. Clone the environment template and configure the secrets:

   ```
   cp .env.example .env
   ```

   Ensure the `MQTT_BROKER`, `MQTT_USERNAME`, and `MQTT_PASSWORD` fields are correctly populated with your HiveMQ Cloud credentials.
3. Boot the infrastructure in detached mode:

   ```
   docker compose up --build -d
   ```

The API Gateway will be exposed at `http://localhost:8000`, with the OpenAPI Swagger documentation available at `/docs`.

7.3. Edge Node Flashing (ESP32-C3)

The firmware strictly requires compile-time secret injection to operate.

1. Navigate to the firmware directory:

   ```
   cd firmware
   ```
2. Clone the configuration template:

   ```
   cp esp32_config.env.example esp32_config.env
   ```
3. Edit `esp32_config.env` to include your local Wi-Fi credentials, the ngrok tunnel URL (or local IP) for the `SERVER_HOST`, and the `ENROLLMENT_TOKEN`.
4. Connect the ESP32-C3 via USB and trigger the PlatformIO upload sequence.
5. Open the Serial Monitor at `115200` baud. On its first boot, the device will generate its ECDSA keys, connect to the network, and automatically register with the backend.

7.4. Mobile Client Initialization

The React Native client must point to the backend's IP address.

1. Navigate to the mobile directory and install dependencies:

   ```
   cd mobile
   npm install
   ```
2. Create a `.env` file in the root of the mobile project matching the backend secrets:

   ```
   EXPO_PUBLIC_IOT_API_KEY=your_secret_key
   EXPO_PUBLIC_MOBILE_WS_TOKEN=your_ws_token
   EXPO_PUBLIC_API_BASE_URL=http://YOUR_LOCAL_IP:8000
   ```
3. Start the Expo development server:

   ```
   npx expo start
   ```
4. Scan the generated QR code using the Expo Go app. Both the smartphone and the backend host must be on the same Local Area Network (LAN).

7.5. Executing the Critical Stress Test

To validate the deployment, the system includes an End-to-End (E2E) stress test that simulates a massive seismic event.

From the `backend` directory, execute:

```
export API_URL="http://localhost:8000"
export NUM_SENSORS=150
export CONCURRENCY_LIMIT=50
python -m tests.stress_test
```

A successful test will conclude with the message 🏆 `SYSTEM CERTIFIED`, confirming that the rate limiter, security gates, and Redis deduplication locks are functioning nominally.

8. AI Emergency Report Service

With the release of v1.2.0, QuakeGuard introduces an on-premise AI layer that automatically generates human-readable emergency reports from confirmed seismic alerts[cite: 1]. The service is powered by a

locally hosted Large Language Model (LLM) via Ollama, guaranteeing that raw telemetry — including zone coordinates and magnitude estimates — never leaves the host machine[cite: 1].

8.1. Privacy-First Local Inference

The AI pipeline is strictly self-contained and requires no external API keys or cloud LLM endpoints[cite: 1].

- **Local Ollama Service:** A dedicated `ollama` Docker container exposes the native Ollama API (`/api/generate`) on an internal Docker network[cite: 1]. On first startup the entrypoint script (`init-scripts/ollama-entrypoint.sh`) automatically pulls the configured model (`OLLAMA_MODEL`, default `llama3.2:1b`) before the service becomes healthy[cite: 1].
- **On-Premise Isolation:** Because the model runs inside the Compose network, every telemetry attribute used to compose a report stays within the host's Docker bridge network[cite: 1]. No third-party receives the data[cite: 1].
- **Model Selection:** The default 1B-parameter model is intentionally small to keep CPU and RAM footprints low while remaining fast enough to generate a report in near real-time[cite: 1]. Operators may override `OLLAMA_MODEL` (e.g., `qwen2.5:1.5b`) for higher quality output[cite: 1].

8.2. Deterministic Report Generation

Emergency messaging must never fabricate data[cite: 1]. The AI client (`ollama_client.py`) therefore constrains the model with hard guarantees[cite: 1]:

- **Sampling:** Every request is issued with `temperature: 0.0` and `top_k: 1` (greedy decoding), eliminating stochastic drift between retries[cite: 1].
- **Streaming Disabled:** Reports are generated in a single non-streaming inference pass, simplifying parsing and validation on the backend[cite: 1].
- **Strict System Prompt:** The model is instructed: *"You are an emergency response AI. Only use the provided JSON telemetry. Do not invent data."*[cite: 1]
- **Telemetry Whitelist:** The prompt is built from a fixed allowlist of fields (`zone_id`, `zone_name`, `magnitude`, `timestamp`, `sensor_count`), so the model can only reason about authenticated, persisted data[cite: 1].
- **Failure Fallback:** If the inference request fails or returns an unparseable response, the pipeline stores an explicit `"AI report unavailable."` message rather than attempting to guess[cite: 1].

8.3. Asynchronous Worker & State Machine

Report generation is fully decoupled from the alert engine via a dedicated Redis queue (`ai_report_queue`) and a separate consumer process (`ai_report_worker.py`)[cite: 1].

- **Non-Blocking Enqueue:** When the main worker (`worker.py`) persists a confirmed `Alert`, it creates an `EmergencyReport` row in `PENDING` state and pushes the alert context to `ai_report_queue` via `lpush`[cite: 1]. The alert pipeline is never blocked by LLM latency[cite: 1].
- **State Machine:** Each report transitions through a tri-state lifecycle[cite: 1]:

$$\text{PENDING} \rightarrow \text{COMPLETED}$$

$$\text{PENDING} \rightarrow \text{FAILED}$$

- **Dedicated Consumer:** The `ai-worker` container loops on a blocking `brpop`, fetches the telemetry context, invokes Ollama, and atomically flips the report state[cite: 1]. On success the worker publishes

an `EMERGENCY_REPORT` payload to the `ai_reports` Redis Pub/Sub channel and commits the `COMPLETED` report with its `summary` and `recommendations`[cite: 1].
- **Dead Letter Queue:** Unrecoverable failures (missing report, persistent inference errors) push the event to `ai_report_queue_dlq` and mark the report `FAILED`[cite: 1]. The mobile app renders a "Report non disponibile" badge for `FAILED` reports instead of showing partial or fabricated content[cite: 1].
- **Graceful Shutdown:** The worker traps `SIGTERM`/`SIGINT` to drain in-flight inference calls before exiting[cite: 1].

8.4. WebSocket Delivery

The mobile client receives reports through the existing real-time channel[cite: 1].

- **Channel Subscription:** The backend WebSocket broadcaster (`main.py`) subscribes to both `quake_alerts` and `ai_reports`, delivering each `EMERGENCY_REPORT` payload over the authenticated `/ws/alerts` socket[cite: 1].
- **Correlation:** The payload carries the originating `alert_id`, allowing the app to attach the report to the matching alert in its history[cite: 1].
- **REST Fallback:** A new `GET /reports/{alert_id}` endpoint exposes the persisted report for clients that reconnect after the alert (e.g., after a WebSocket drop)[cite: 1].
- **Inline Presentation:** The mobile app renders `COMPLETED` reports as a highlighted banner for the latest alert and as a dedicated card inside the alert history feed, showing the generated summary and per-item recommendations[cite: 1].

8.5. Operational Notes

- **Compose Profile:** The `ollama` and `ai-worker` services are gated behind the `ai` profile so that the default `docker compose up` remains lightweight and CI stays hermetic[cite: 1]. Enable the pipeline with `docker compose --profile ai up -d`[cite: 1].
- **Feature Flag:** The main worker only enqueues reports when `AI_REPORT_ENABLED=true` (default `false`) to avoid orphaned `PENDING` rows when the AI profile is not running[cite: 1].
- **Timeouts:** Inference requests honor `OLLAMA_TIMEOUT`; the Ollama service uses `KEEP_ALIVE` to reduce cold-start latency between alerts[cite: 1].