TRAFFIC INTEL DASHBOARD — OPENWEBNINJA (WAZE) CONTRACT (v3)

Target platform: Vercel Serverless Functions (web app).
Purpose: Implement a Traffic Intel dashboard/mode that scans the current map viewport (BBox) via OpenWebNinja Waze API, renders results on MapLibre, provides a product-grade UI layout, and supports cinematic fly-to + orbit on selection. **Security:** Never hardcode secrets or endpoints; environment variables only.

1) Non-Negotiable Requirements

1. Traffic Intel is community-sourced; do not present it as official emergency data.
2. Activated via dedicated dashboard/mode button (same bottom-sheet framework as Emergency Alerts).
3. User-initiated 'Scan View' uses current map viewport bbox.
4. Persist scan snapshots + individual events to database for analysis.
5. Use Vercel Serverless Functions for all API calls; client never calls OpenWebNinja directly.
6. Never hardcode secrets/endpoints; env vars only; fail fast if missing.

2) Environment Variables (Required)

```
# OpenWebNinja
OPENWEBNINJA_API_KEY=...                  # REQUIRED
OPENWEBNINJA_BASE_URL=https://www.openwebninja.com  # REQUIRED
OPENWEBNINJA_WAZE_ENDPOINT=/api/waze/alerts         # OPTIONAL (override path)

# Database
DATABASE_URL=postgres://...               # REQUIRED
DATABASE_SSL=1                            # OPTIONAL (managed Postgres)
```

3) UI Layout Specification (Mobile-First)

Traffic Intel must mirror the usability of Emergency Alerts panel, with clear scanning, filtering, and selection flows.

3.1 Panel Structure (Bottom Sheet)

1. **Header row:** Title “Traffic Intel (Community)”, small subtitle “Source: OpenWebNinja (Waze)”.
2. **Primary CTA:** “Scan View” button (disabled while scanning) + optional Cancel button.
3. **Status:** last scan time, in-view count, total count, and a small health indicator.
4. **Filters strip:** type chips (Accident/Hazard/Police/Closure/Jam), recency (15m/1h/6h/24h), confidence slider (optional).
5. **View toggle:** “Only show items in view” default ON; when OFF show last scan results for entire bbox.
6. **Results list:** cards sorted by recency then confidence; each card has icon, type/subtype, age badge, street/city.
7. **Detail view:** cleaned text, metadata (time, location fields, confidence), and link fields; always show source label.

3.2 Floating Map Controls

1. A small floating button to open/close Traffic Intel panel when in the mode.
2. Optional legend icon showing category color/icon mapping.
3. Do not overlap Recon joysticks; if Recon mode exists simultaneously, Traffic panel must not obstruct flight controls.

4) Scan-in-View Workflow (Serverless)

1. UI reads current map bbox (w,s,e,n) and zoom.
2. Client calls POST /api/traffic/scan with bbox + filters.
3. Serverless calls OpenWebNinja Waze API using env vars.
4. Serverless persists scan + upserts events, then returns normalized payload (scan id, counts, geojson).

5) Cinematic Selection: Click → Fly-To → Orbit on Approach

Selecting a traffic item (from list or map) must trigger a cinematic camera sequence and then orbit the target. This must reuse the shared flight/orbit module (no duplicate orbit logic).

5.1 Sequence (Exact)

1. **Step 1 (select):** highlight marker (pulse ring) and open detail panel.
2. **Step 2 (fly-to):** start a cinematic fly-to with easing; adjust bearing and pitch for an oblique approach.
3. **Step 3 (approach detection):** when camera is within a threshold distance OR animation completes, automatically start orbit.
4. **Step 4 (orbit):** orbit radius based on zoom/altitude proxy; orbit speed moderate; keep target centered.

5.2 Manual Override (Pilot Authority)

1. Any manual map gesture instantly cancels fly-to/orbit/random and returns to free movement.
2. Re-engage requires explicit action (Orbit button or reselect item).

5.3 Parameters (Recommended)

1. Fly-to duration: 800–1600ms (distance-scaled).
2. Approach pitch: 45–65° (tunable).
3. Approach bearing: align with street direction if available; otherwise maintain current bearing.
4. Orbit radius: 150–400m equivalent (tunable), larger at higher zoom-out.

6) API Endpoints & Code (Vercel Serverless)

The following code is a reference implementation for Vercel/Next.js serverless route handlers.

6.1 POST /api/traffic/scan (Route Handler)

```
// app/api/traffic/scan/route.ts
import { NextResponse } from "next/server";
import { z } from "zod";
import { Pool } from "pg";

const Env = z.object({
  OPENWEBNINJA_API_KEY: z.string().min(10),
  OPENWEBNINJA_BASE_URL: z.string().url(),
  DATABASE_URL: z.string().min(10),
}).parse(process.env);

const pool = new Pool({
  connectionString: Env.DATABASE_URL,
  ssl: process.env.DATABASE_SSL ? { rejectUnauthorized: false } : undefined,
  max: 5,
});

const BodySchema = z.object({
  bbox: z.object({ w: z.number(), s: z.number(), e: z.number(), n: z.number() }),
  zoom: z.number().optional(),
  filters: z.object({
    types: z.array(z.string()).optional(),
    maxAgeMinutes: z.number().optional(),
    minConfidence: z.number().optional(),
    includeJams: z.boolean().optional(),
  }).optional(),
});

function cleanText(input: unknown): string {
  if (typeof input !== "string") return "";
  return input.replace(/<[^>]+>/g, " ").replace(/\s+/g, " ").trim();
}

export async function POST(req: Request) {
  const started = Date.now();
  const body = BodySchema.parse(await req.json());

  const endpoint = process.env.OPENWEBNINJA_WAZE_ENDPOINT || "/api/waze/alerts";
  const url = new URL(endpoint, Env.OPENWEBNINJA_BASE_URL);
  url.searchParams.set("w", String(body.bbox.w));
  url.searchParams.set("s", String(body.bbox.s));
  url.searchParams.set("e", String(body.bbox.e));
  url.searchParams.set("n", String(body.bbox.n));

  if (body.filters?.maxAgeMinutes) url.searchParams.set("max_age_min", String(body.filters.maxAgeMinutes));
  if (body.filters?.minConfidence != null) url.searchParams.set("min_conf", String(body.filters.minConfidence));
  if (body.filters?.types?.length) url.searchParams.set("types", body.filters.types.join(","));
  if (body.filters?.includeJams === false) url.searchParams.set("include_jams", "0");

  const owResp = await fetch(url.toString(), {
    headers: { "Authorization": `Bearer ${Env.OPENWEBNINJA_API_KEY}` },
    cache: "no-store",
  });
  if (!owResp.ok) {
    return NextResponse.json({ status: "error", code: "OPENWEBNINJA_UPSTREAM_ERROR", http: owResp.status }, { status: 50
  }

  const raw = await owResp.json();
  const alerts = Array.isArray(raw.alerts) ? raw.alerts : [];
  const jams = Array.isArray(raw.jams) ? raw.jams : [];

  const client = await pool.connect();
  try {
    await client.query("BEGIN");
    const scanRes = await client.query(
      `INSERT INTO traffic_scan (bbox_w,bbox_s,bbox_e,bbox_n,zoom,total_alerts,total_jams,request_meta,feed_health)
       VALUES ($1,$2,$3,$4,$5,$6,$7,$8,$9) RETURNING id, scanned_at`,
      [
        body.bbox.w, body.bbox.s, body.bbox.e, body.bbox.n,
        body.zoom ?? null,
        alerts.length,
        jams.length,
        JSON.stringify(body.filters ?? {}),
        JSON.stringify({ ms: Date.now() - started }),
      ]
    );
    const scanId = scanRes.rows[0].id;

    const upsert = async (item: any, kind: "alert"|"jam") => {
      const sourceEventId = String(item.alert_id ?? item.jam_id ?? item.id ?? "");
      if (!sourceEventId) return;

      const lat = Number(item.latitude ?? item.lat);
      const lon = Number(item.longitude ?? item.lon);
      if (!Number.isFinite(lat) || !Number.isFinite(lon)) return;

      const published = item.publish_datetime_utc ?? item.published_at ?? item.time ?? null;
      const description = cleanText(item.description ?? item.text ?? "");

      await client.query(
        `INSERT INTO traffic_event
         (source, source_event_id, event_kind, event_type, subtype, published_at_utc, lat, lon,
          country, city, street, confidence, reliability, thumbs_up, description_clean, official_link, raw, last_seen_at
         VALUES ('openwebninja_waze',$1,$2,$3,$4,$5,$6,$7,$8,$9,$10,$11,$12,$13,$14,$15,$16, now(), $17)
         ON CONFLICT (source, source_event_id) DO UPDATE SET
           last_seen_at = now(),
           last_scan_id = EXCLUDED.last_scan_id,
           confidence = COALESCE(EXCLUDED.confidence, traffic_event.confidence),
           reliability = COALESCE(EXCLUDED.reliability, traffic_event.reliability),
           thumbs_up = COALESCE(EXCLUDED.thumbs_up, traffic_event.thumbs_up),
           description_clean = COALESCE(NULLIF(EXCLUDED.description_clean,''), traffic_event.description_clean),
           raw = traffic_event.raw || EXCLUDED.raw`,
        [
          sourceEventId,
          kind,
          String(item.type ?? kind),
          String(item.subtype ?? ""),
          published ? new Date(published) : null,
          lat, lon,
          String(item.country ?? ""),
          String(item.city ?? ""),
          String(item.street ?? ""),
          item.alert_confidence ?? item.confidence ?? null,
          item.alert_reliability ?? item.reliability ?? null,
          item.num_thumbs_up ?? null,
          description,
          String(item.link ?? item.url ?? ""),
          JSON.stringify(item),
          scanId,
        ]
      );
    };

    for (const a of alerts) await upsert(a, "alert");
    for (const j of jams) await upsert(j, "jam");

    await client.query("COMMIT");

    return NextResponse.json({
      status: "ok",
      source: "openwebninja_waze",
      scan: { id: scanId, scanned_at: scanRes.rows[0].scanned_at, bbox: body.bbox, counts: { alerts: alerts.length, jams
      meta: { ms: Date.now() - started }
    });
  } catch (e) {
    await client.query("ROLLBACK");
    return NextResponse.json({ status: "error", code: "TRAFFIC_SCAN_FAILED" }, { status: 500 });
  } finally {
    client.release();
  }
}
```

6.2 UI Selection → Cinematic Fly-To → Orbit (Pseudocode)

```
// Called when user taps a marker or list item
async function selectTrafficItem(item) {
  // 1) highlight & open detail
  ui.setSelected(item.id);
  mapLayer.pulse(item.id);

  // 2) cinematic approach (shared flight core)
  flight.cancelAutomation("new_selection");
  await flight.flyToTarget({
    lngLat: item.lngLat,
    durationMs: flight.computeDuration(item.lngLat),
    pitch: 55,                 // oblique approach
    bearing: flight.keepOrAlignBearing(item),
    padding: ui.currentPadding(),  // account for panel
    easing: "easeInOutCubic",
  });

  // 3) orbit on approach
  if (!flight.isCancelled()) {
    flight.startOrbit({
      target: item.lngLat,
      radiusMeters: flight.radiusFromZoom(map.getZoom()),
      speed: 1.0, // moderate
    });
  }
}

// Manual map gesture must cancel automation instantly
map.on("movestart", () => flight.cancelAutomation("manual_map_gesture"));
```

7) Visual Layer Rules & Icons

1. Map overlay is points only (Waze items are point events).
2. Cluster by default; cluster badge shows count and dominant icon.
3. Icons: Accident / Hazard / Police / Closure / Jam (canonical set).
4. Recency styling: <60 minutes = strong highlight; older = subdued.
5. Detail view always includes Source label and any provided link field.

8) Acceptance Tests (Pass/Fail)

1. UI: Traffic Intel mode opens panel; Scan View uses bbox and shows progress states.
2. Data: /api/traffic/scan uses env vars only; no hardcoded secrets or base URLs.
3. DB: scan record inserted and events upserted (dedupe works).
4. UX: click item triggers cinematic fly-to and then orbit on approach; manual map gesture cancels instantly.
5. Trust: visible labeling as community traffic intel; separated from official emergency alerts.