



Fleet Management System

Project Engineering

Year 4

Alan Hynes (G00400498)

Bachelor of Engineering (Honours) in Software and
Electronic Engineering

Atlantic Technological University

2024/2025

Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Atlantic Technological University.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.



Alan Hynes

Acknowledgements

I would like to thank my supervisor, David Newell, for his guidance throughout this project.

I also wish to thank Ben Kinsella, Paul Lennon, Michelle Lynch, Niall O’Keeffe, and Brian O’Shea for their tutelage across the course.

Table of Contents

1	Summary	5
2	Poster	6
3	Introduction	7
4	Background	8
4.1	GPS and location sampling.....	8
4.2	Geofencing methods.....	8
4.3	Real-time delivery on the web.....	8
4.4	Data model and storage.....	9
4.5	Cloud deployment basics	9
4.6	Map rendering and UI points.....	9
4.7	Security and data protection	9
5	Project Architecture.....	10
6	Project Plan	11
7	Project Code.....	12
7.1	Frontend.....	12
7.2	Backend.....	16
8	Ethics	20
9	Conclusion.....	22
10	Appendix	23
11	References	28

1 Summary


This project builds a cloud-hosted fleet management system that shows live vehicle position on a Google Map, draws polygon geofences, and records entry and exit events. The aim is to provide a simple dashboard that a dispatcher can use to watch a small fleet, review recent movements, and react to alerts.

The system runs on a Node.js and Express API on AWS EC2. Location data comes from the Overland iOS app or a simulator. The server stores a current snapshot and a history of points in MongoDB Atlas, checks whether a point is inside or outside each fence, and streams updates to the React dashboard using Socket.io. The dashboard presents a vehicle list, colour-coded markers, a geofence manager, a violations panel with a resolve action, and a basic history view. PM2 manages the services on EC2 and a /health endpoint supports uptime checks.

The project moved from an early Raspberry Pi plan to a phone-based feed, which proved faster to deploy and easier to test. Two geofences were created for evaluation, ATU Galway Campus and Galway City Centre. Both simulated and live runs were recorded. Freshness on the dashboard during tests was typically a few seconds. Logged events confirmed automatic entry and exit detection and the UI allowed items to be marked as resolved.

The work shows that a lightweight phone feed, a small cloud API, and a web dashboard are enough to give real-time tracking and geofencing for a small fleet. Next steps are HTTPS, a simple API key for ingest, and email or SMS notifications.

2 Poster



Ollscoil
Teicneolaíochta
an Atlantaigh

Atlantic
Technological
University

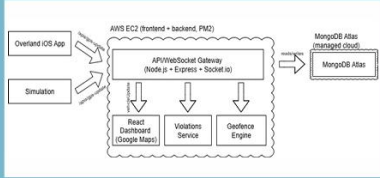
Fleet Management System

Alan Hynes
Beng (Hons) Software & Electronic Engineering

Summary

This project delivers a cloud-hosted fleet system that shows live vehicle position on a Google Map, draws polygon geofences, and detects **entry** and **exit** events in real time. An Overland iOS app (or a simulator) posts GPS points to a Node.js/Express API on AWS EC2. The server stores snapshots and history in MongoDB Atlas, checks geofences, and pushes updates to the React dashboard with Socket.io. The dashboard shows a vehicle list, colour-coded markers, a geofence manager, a violations panel with resolve actions, and a simple history view. The build is deployed on EC2 with PM2 and includes a /health check for uptime during tests.


Architecture Diagram



Tools Used

- **Frontend:** React, Google Maps JavaScript API
- **Backend:** Node.js Express, Socket.io
- **Database:** MongoDB Atlas
- **Cloud:** AWS EC2, PM2
- **Dev/Test:** VS Code, TalendAPI, curl
- **GPS Source:** Overland iOS app (simulator as backup)


Web Interface



Topics Covered

- Real-time data with Socket.io
- Geofencing: point-in-polygon and circle checks
- Cloud deployment on AWS EC2 with PM2
- Rest API design for ingest and read paths
- MongoDB Atlas: snapshot, history, and violations
- Google Maps integrations in React
- Monitoring: /health, logs, and basic metrics
- Testing with live phone data and a simulator

QR



SCAN ME

Figure 2.1 - Poster

3 Introduction

The goal of this project is to build a small, cloud-hosted fleet system that shows live vehicle position on a map, lets a user draw geofences, and raises entry and exit events. The motivation is simple. Many small operators in Ireland need basic visibility rather than a full telematics platform. A low-cost web dashboard that runs from a phone data feed can meet that need and is also a good fit for a final year build where time is tight.

The solution uses a phone-based GPS source (the Overland iOS app) or a simulator to post locations to a Node.js and Express API on AWS EC2. The server keeps a current snapshot and a history in MongoDB Atlas, checks points against stored geofences, and streams updates to a React dashboard with Socket.io. The dashboard shows colour-coded markers, a vehicle list, a geofence manager, a violations panel with a resolve action, and a simple history view. PM2 runs the services on EC2 and a /health endpoint supports basic monitoring.

Scope and terms of reference:

The project covers: live ingest of GPS points, storage of snapshots and history, polygon and circle geofences, real-time entry and exit detection, a web dashboard, cloud deployment on one EC2 instance, and testing with both simulated and live data around Galway. The intended scale is a small fleet, roughly one to ten vehicles, with location samples at tens of seconds. Out of scope are user accounts, role-based access, HTTPS and token-based security, advanced analytics or routing, and third-party notifications. Email or SMS can be added later without changing the UI. An early Raspberry Pi approach was explored and then replaced by the phone feed, which proved more reliable and quicker to set up.

The remainder of the report describes the design and architecture, the implementation of the API and dashboard, testing and results, ethical considerations, a short discussion of limits and future work, and the references and appendices.

4 Background

This section covers the key concepts behind the system and the design choices made. It covers GPS sampling, geofencing methods, real-time delivery in the browser, storage models, cloud deployment, and data protection.

4.1 GPS and location sampling

Smartphones report latitude and longitude in WGS-84 with varying accuracy [1]. Outdoor fixes are often within a few metres, but accuracy drops near tall buildings or indoors [2]. Phones also smooth movement and may batch points to save battery. Two practical points follow from this:

- The dashboard should show last updated time and accept gaps between samples.
- The geofence checker should tolerate small jitter near edges.

The project uses two sources: the Overland iOS app for live phone data and a simulator for repeatable tests. Typical intervals are 30 to 60 seconds for phone runs and 5 seconds for the simulator [12].

4.2 Geofencing methods

Two fence types are used. Polygon fences apply a point-in-polygon test based on the even-odd rule (ray casting) to decide inside or outside [9]. Circle fences compute Haversine distance from the centre and compare with the radius [10]. A small merge window around boundaries reduces double firing when points jitter across an edge. The system records entry when a point moves from outside to inside and exit for the opposite case.

4.3 Real-time delivery on the web

For live updates the server keeps an open WebSocket so it can push changes without short polling [3], [11]. Socket.IO provides a stable connection with fallbacks where needed and carries a vehicleUpdate event whenever locations change [4]. The dashboard still performs initial reads, so a user sees data on first load.

4.4 Data model and storage

Two patterns are used in MongoDB Atlas. A snapshot collection holds the latest state per vehicle for fast map rendering. A history collection stores time-stamped points for the route view and audit. A violations collection stores entry and exit rows with fence id, vehicle id, time, latitude, and longitude. Atlas uses an IP access list; the current EC2 public address must be present for connections to succeed [6].

4.5 Cloud deployment basics

The API runs on AWS EC2 and is supervised by PM2. PM2 restarts the process if it exits and provides simple log viewing and status checks [7], [8]. A /health endpoint gives a quick signal for monitoring and for the demonstration.

4.6 Map rendering and UI points

The dashboard uses the Google Maps JavaScript API to render markers, polylines, and drawn polygons [5]. Markers are colour-coded by status. A geofence manager lets a user draw and delete polygons on the map. A violations panel lists recent entry and exit rows and allows an item to be marked as resolved. A basic history view draws a polyline for the selected vehicle.

4.7 Security and data protection

During testing only my own phone location was captured with consent. No third-party data were collected. The database is private to the project. Records were kept for development and assessment and will be deleted at the end of the project. Secrets are stored in environment variables, and access to MongoDB Atlas is restricted by IP. For a production build, I would enable HTTPS on EC2, restrict the Google Maps key by referrer, and add a token on ingest endpoints.

5 Project Architecture

This is the architecture diagram for the project.

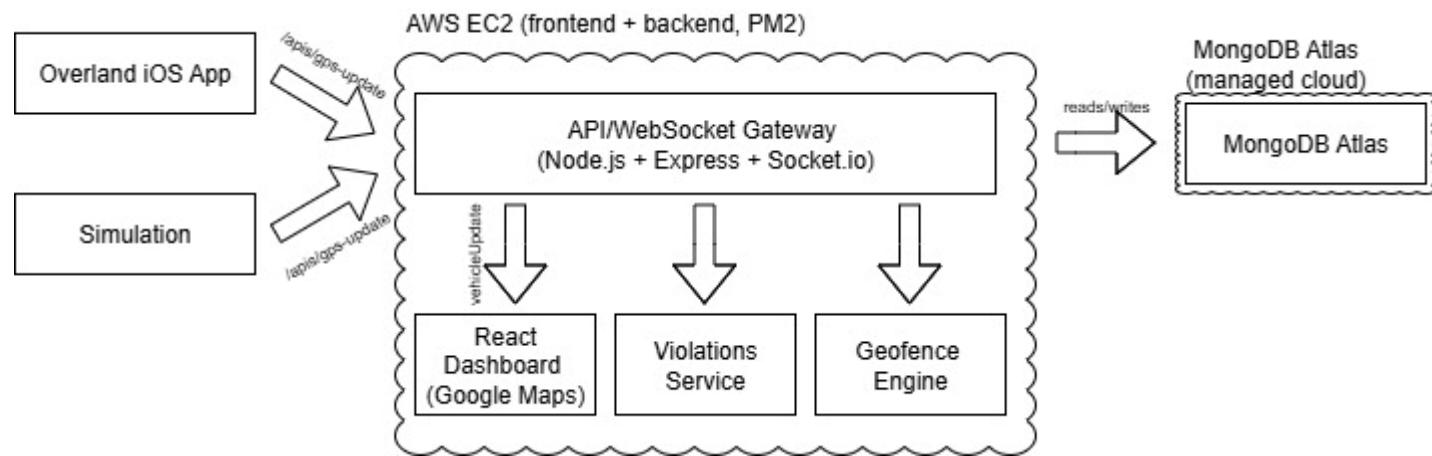


Figure 5-1 Architecture Diagram

6 Project Plan

Initial plan:

- Semester 1 (Sept-Dec 2024): live map, WebSockets, MongoDB, history view, status colours, simulator.
- Semester 2 (Jan-Apr 2025): cloud deployment, device feed, geofences and alerts, UI polish, testing, documentation.

What happened:

- Deployed API and dashboard to EC2 with PM2.
- Switched from Raspberry Pi to Overland on iPhone.
- Implemented polygon geofences for ATU Galway Campus and Galway City Centre.
- Built violations list with resolve button.
- Collected runs with both simulator and phone.

Changes to scope:

- Retired Raspberry Pi route and used phone data instead.
- Kept simulator for demos.

Challenges and fixes:

- Overland queued thousands of points during outages. Fix: restore API then reinstall app to clear the queue.
- Near-edge jitter caused double events. Fix: add short merge window in the checker.

7 Project Code

This section explains the main software elements of the project. It is split into the frontend and backend. Short code snippets are shown beside the text, with full listings in the appendix.

7.1 Frontend

Structure:

App.js mounts the map view. MapComponent.js handles Google Maps, sockets, markers, the geofence editor, the violations panel, and the history polyline. config.js defines the API base.

Initial loads and live updates:

Upon loading, the app fetches vehicles, geofences, and violations. A Socket.io client listens for vehicleUpdate so the map stays current.

Markers, history, and geofences:

Markers are colour-coded by status. A simple polyline draws the selected vehicle's recent route. The geofence editor lets a user draw a polygon and save or delete it. Violations can be marked as resolved.

```
frontend > fleet-dashboard > src > JS App.js > ...
1  import React from "react";
2  import MapComponent from "../MapComponent";
3
4  function App() {
5    return (
6      <div>
7        <h1>Fleet Management System</h1>
8        <MapComponent />
9      </div>
10   );
11 }
12
13 export default App;
```

Figure 7.1-1 App shell (App.js)

```
src > JS config.js > ...
1 // src/config.js
2 const API_BASE =
3   process.env.REACT_APP_API_BASE ||
4   `http://${window.location.hostname}:3001`;
5
6 export { API_BASE };
```

Figure 7.1-2 API base (config.js)

```
const fetchVehicles = async () => {
  try {
    const res = await fetch(`${API_BASE}/api/locations`);
    const data = await res.json();
    setVehicles(data);
    setRoutes((prev) => {
      const next = { ...prev };
      data.forEach((v) => {
        if (!next[v.id]) next[v.id] = [];
        next[v.id].push({ lat: v.lat, lng: v.lng });
      });
      return next;
    });
  } catch (err) {
    console.error("Initial /api/locations fetch failed:", err);
  }
};

const fetchGeofences = async () => {
  try {
    const res = await fetch(`${API_BASE}/api/geofences`);
    const data = await res.json();
    setGeofences(data);
  } catch (err) {
    console.error("Error fetching geofences:", err);
  }
};
```

```
const fetchViolations = async () => {
  try {
    const res = await fetch(`${API_BASE}/api/violations`);
    const data = await res.json();
    setViolations(data);
  } catch (err) {
    console.error("Error fetching violations:", err);
  }
};
```

Figure 7.1-3 Initial loads (MapComponent.js)

```
// Socket.IO for real-time updates
useEffect(() => {
  const SOCKET_URL =
    process.env.REACT_APP_BACKEND_URL || API_BASE || `http://${window.location.hostname}:3001`;

  socketRef.current = io(SOCKET_URL, {
    transports: ["websocket", "polling"],
    withCredentials: false,
  });

  socketRef.current.on("vehicleUpdate", (updatedVehicles) => {
    setVehicles(updatedVehicles);
    setRoutes((prev) => {
      const next = { ...prev };
      updatedVehicles.forEach((v) => {
        if (!next[v.id]) next[v.id] = [];
        next[v.id].push({ lat: v.lat, lng: v.lng });
      });
      return next;
    });
  });
});
```

Figure 7.1-4 Socket wiring and live updates (MapComponent.js)

```
vehicles.forEach((v) => {
  const icon =
    v.alert === "breakdown"
      ? "http://maps.google.com/mapfiles/ms/icons/red-dot.png"
      : v.alert === "idle"
      ? "http://maps.google.com/mapfiles/ms/icons/yellow-dot.png"
      : "http://maps.google.com/mapfiles/ms/icons/green-dot.png";

  const marker = new window.google.maps.Marker({
    map: mapInstance,
    position: { lat: v.lat, lng: v.lng },
    title: `${v.address} - ${v.status}`,
    icon,
  });

  marker.addListener("click", () => setSelectedVehicleId(v.id));
  mapRef.current.markers.push(marker);
});
```

Figure 7.1-5 Marker colours and render (MapComponent.js)

```
/* Vehicle routes */
{selectedVehicleId && routes[selectedVehicleId] && (
  <Polyline
    path={routes[selectedVehicleId]}
    options={{ strokeColor: "■ #FF0000", strokeOpacity: 0.8, strokeWeight: 4 }}
  />
)}
```

Figure 7.1-6 Route polyline for selected vehicle (MapComponent.js)

```
// Finish drawing and save geofence
const finishDrawing = async () => {
  if (currentPath.length < 3 && drawingType === 'polygon') {
    alert('Polygon needs at least 3 points');
    return;
  }

  const name = prompt('Enter geofence name:');
  if (!name) return;

  try {
    const geofenceData = {
      name,
      type: drawingType,
      coordinates: drawingType === 'polygon'
        ? currentPath.map(p => [p.lng, p.lat]) // Convert to [lng, lat] format
        : [currentPath[0].lng, currentPath[0].lat], // Circle center
      radius: drawingType === 'circle' ? 1000 : undefined,
      alertOnEntry: true,
      alertOnExit: true
    };

    const response = await fetch(`${API_BASE}/api/geofences`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(geofenceData)
    });
  }
};
```

```
    if (response.ok) {
      fetchGeofences(); // Refresh geofences
    }
  } catch (error) {
    console.error('Error creating geofence:', error);
  }

  // Reset drawing state
  setIsDrawing(false);
  setCurrentPath([]);
};

// Cancel drawing
const cancelDrawing = () => {
  setIsDrawing(false);
  setCurrentPath([]);
};

// Delete geofence
const deleteGeofence = async (geofenceId) => {
  try {
    const response = await fetch(`${API_BASE}/api/geofences/${geofenceId}`, {
      method: 'DELETE'
    });
  }
};
```

```
    if (response.ok) {
      fetchGeofences();
      setSelectedGeofence(null);
      setShowGeofenceInfo(false);
    }
  } catch (error) {
    console.error('Error deleting geofence:', error);
  }
};
```

Figure 7.1-7 Geofence save and delete (MapComponent.js)

```
// Resolve violation
const resolveViolation = async (violationId) => {
  try {
    const response = await fetch(`${API_BASE}/api/violations/${violationId}/resolve`, {
      method: 'PATCH'
    });

    if (response.ok) {
      fetchViolations();
    }
  } catch (error) {
    console.error('Error resolving violation:', error);
  }
};
```

Figure 7.1-8 Resolve a violation (MapComponent.js)

7.2 Backend

Process and sockets:

index.js creates the Express app and Socket.io server, connects to Atlas, and pushes vehicleUpdate to clients on a timer and after writes. A /health path supports simple checks.

Ingest and checks:

POST /api/gps-update accepts a phone or simulator payload, upserts a snapshot, appends a history row, runs geofence checks, writes a violation on transition, and emits vehicleUpdate.

REST for the dashboard:

The UI reads snapshots, fences, and violations, and can create or delete fences and resolve a violation. The notification hook logs alerts for the demo.

Open CORS:

For the demo the API and Socket.io server accept requests from any origin (origin: "*") so the dashboard could be opened from localhost and EC2 without blockers. For a live build restrict CORS to the dashboard host and use HTTPS and keep the MongoDB Atlas IP list and Google Maps key locked down.


```
const app = express(); const port = 3001;
app.use(cors({ origin: "*", methods: ["GET", "POST", "PATCH", "DELETE"] }));
app.use(express.json());
app.use(express.urlencoded({ extended: true })); // for form posts

const server = http.createServer(app);
const io = new Server(server, { cors: { origin: "*" } });
```

```
/* ----- Sockets ----- */
io.on("connection", async (socket)=>{
  console.log("Client connected:", socket.id);
  try { socket.emit("vehicleUpdate", await Vehicle.find({})); } catch(e){ console.error(e); }

  const interval=setInterval(async ()=>{
    const list=await Vehicle.find({});
    for(const v of list){
      if(v?.id && !v.id.startsWith('pi-') && !v.id.startsWith('iphone-')){
        if(v.status==='moving'){
          if(Math.random()<0.1) v.alert="breakdown";
          else if(Math.random()<0.1) v.alert="idle";
          else v.alert="";
          v.lat+=(Math.random()-0.5)*0.01;
          v.lng+=(Math.random()-0.5)*0.01;
        }
        v.lastUpdated=new Date(); await v.save();
        await VehicleHistory.create({ vehicleId:v.id, lat:v.lat, lng:v.lng, timestamp:v.lastUpdated });
      }
    }
    socket.emit("vehicleUpdate", await Vehicle.find({}));
  },2000);

  socket.on("disconnect", ()=>{ clearInterval(interval); console.log("Client disconnected:", socket.id); });
});
```

Figure 7.2-1 Server CORS, sockets (index.js)

```
app.get("/health", (_req,res)=>res.json({ok:true}));
```

Figure 7.2-2 Health endpoint (index.js)

```

app.post("/api/gps-update", async (req,res)=>{
  try{
    // accept both simple JSON and Overland shapes here too
    const b = req.body||{};
    const p = (b.current || b.locations) ? extractOverlandShape(b) : {
      id: b.vehicleId || b.id || b.device || 'iphone-vehicle-1',
      lat: b.latitude ?? b.lat, lng: b.longitude ?? b.lon ?? b.lng,
      status: b.status || b.activity || 'moving'
    };
    await upsertVehicle(p);
    res.json({ok:true});
  }catch(e){ console.error("error processing /api/gps-update:", e); res.status(500).json(
    {error:"internal server error"}); }
});

```

Figure 7.2-3 GPS ingest pipeline (index.js)

```

function calculateDistance(lat1,lng1,lat2,lng2){
  const R=6371, dLat=(lat2-lat1)*Math.PI/180, dLng=(lng2-lng1)*Math.PI/180;
  const a=Math.sin(dLat/2)**2+Math.cos(lat1*Math.PI/180)*Math.cos(lat2*Math.PI/180)*Math.sin(dLng/2)**2;
  return 2*R*Math.atan2(Math.sqrt(a),Math.sqrt(1-a))*1000;
}
function pointInPolygon(lat,lng,poly){let inside=false;
  for(let i=0,j=poly.length-1;i<poly.length;j=i++){
    const xi=poly[i][0], yi=poly[i][1], xj=poly[j][0], yj=poly[j][1];
    if(((yi>lat)!=(yj>lat))&&(lng<(xj-xi)*(lat-yi)/(yj-yi)+xi)) inside=!inside;
  } return inside;
}

```

Figure 7.2-4 Geofence calculations (index.js)

```

app.post("/api/geofences", async (req,res)=>{ try{
  const { id,name,type,coordinates,radius,alertOnEntry,alertOnExit }=req.body;
  const g=new Geofence({ id:id||`geofence-${Date.now()}`, name, type:type||'circle', coordinates, radius:radius||
    null,
    alertOnEntry: alertOnEntry!==false, alertOnExit: alertOnExit!==false, createdAt:new Date() });
  await g.save(); res.json({success:true, geofence:g});
}catch(e){ res.status(500).json({error:"internal server error"}); } });

app.get("/api/geofences", async (_req,res)=>{
  try{ res.json(await Geofence.find({})); }catch(e){ res.status(500).json({error:"internal server error"}); } });

app.delete("/api/geofences/:id", async (req,res)=>{ try{ await Geofence.findByIdAndDelete(req.params.id);
  res.json({success:true}); }catch(e){ res.status(500).json({error:"internal server error"}); } });

```

Figure 7.2-5 Geofence CRUD (index.js)

```

app.get("/api/violations", async (_req,res)=>{
  try{ res.json(await Violation.find({}).sort({timestamp:-1}));
  }catch(e){ res.status(500).json({error:"internal server error"}); } });
app.get("/api/violations/:vehicleId", async (req,res)=>{
  try{ res.json(await Violation.find({vehicleId:req.params.vehicleId}).sort({timestamp:-1}));
  }catch(e){ res.status(500).json({error:"internal server error"}); } });
app.patch("/api/violations/:id/resolve", async (req,res)=>{
  try{ const violation = await Violation.findByIdAndUpdate(req.params.id, { resolved: true, resolvedAt: new Date()
  }, { new: true }); res.json(violation); }catch(e){ res.status(500).json({error:"internal server error"}); } });

```

Figure 7.2-6 Violations API (index.js)

```

const nodemailer = require('nodemailer');

class NotificationService {
  constructor() {
    this.enabled = process.env.EMAIL_USER && process.env.EMAIL_USER !== 'your-gmail@gmail.com';
  }

  async sendViolationAlert(violation) {
    const message = `FLEET ALERT: Vehicle ${violation.vehicleId} ${violation.violationType} ${violation.geofenceName}
    at ${new Date(violation.timestamp).toLocaleString()}`;

    if (this.enabled) {
      //email functionality would go here
      console.log('Would send email:', message);
    } else {
      console.log('Geofence Violation Alert:', message);
    }
  }
}

module.exports = NotificationService;

```

Figure 7.2-7 Notification hook (notification-service.js)

8 Ethics

Purpose and scope:

This project was built to show live tracking and geofencing for a small fleet. It is a demo, not a production monitoring tool.

Data collected:

The system stores latitude, longitude, time and a vehicle id. No names, phone numbers or home addresses are stored. During testing I used my own phone as a data source and a simulator.

Consent and transparency:

Live tests used my own data with my consent. No third-party location data were processed. Any future development must be opt-in, with a clear notice explaining what is collected, why it is collected, who can see it, and how to stop it (pause/opt-out).

Legal context (GDPR):

Location data is personal data in the EU. A real deployment would need lawful basis (e.g. informed consent), a privacy note, and a data processing agreement with any cloud provider.

Security:

For the demo, the API ran over HTTP with open CORS. A live build should use HTTPS, restrict CORS to the dashboard host, add an API key or token on ingest, lock the Google Maps key by referrer, and keep the MongoDB Atlas IP list tight. Secrets should be kept in environment variables, not in the repo.

Retention:

Test data were kept only for development and grading and will be deleted at the end of the project. A real system should define short retention periods and automatic deletion.

Risk and misuse:

Tracking can be misused for surveillance. Any rollout should set clear rules on when tracking is on, who can view the data, and how accuracy limits are handled so geofence alerts are not used as sole evidence.

9 Conclusion

This project delivers a working, cloud-hosted fleet system. A phone or the built-in simulator sends GPS points to a Node.js and Express API on AWS EC2. The server writes a snapshot and a history trail in MongoDB Atlas, checks polygon and circle geofences, and pushes live updates to the React dashboard with Socket.io. The dashboard shows a vehicle list with colour-coded markers, a geofence manager, a violations panel with resolve, and a simple history view. Services are kept running with PM2 and a /health check is available.

The switch from a Raspberry Pi to a phone feed made data capture faster and more reliable. Two named geofences were created for testing and both simulated and live runs were recorded. The system logged real entry and exit events for Galway City Centre and ATU Galway Campus, and the map refreshed within a few seconds during drive tests. The build also coped with EC2 IP changes once the Atlas allow list was updated.

The outcome is a demonstrable end-to-end tracker with real-time geofencing, history, and a usable web interface. Small operators could use this pattern today for basic visibility at low cost.

Future work would be to add HTTPS, restrict CORS, use an API token on ingest, enable email or SMS alerts, fix the public IP with an Elastic IP or domain, and add automated tests and a richer history view.

10 Appendix

A. API reference

Base URL: `http://<EC2_PUBLIC_IP>:3001`

The EC2 public IP can change after a stop or a restart.

Update steps when the IP changes:

1. Find IP: EC2 console → Instance → Public IPv4, or `curl -s http://checkip.amazonaws.com` over SSH
2. Health check:


```
IP=<paste_current_ip>
curl http://$IP:3001/health # expect {"ok":true}
```
3. Frontend: open the dashboard at `http://<EC2_PUBLIC_IP>:3000`, or set `REACT_APP_API_BASE=http://<EC2_PUBLIC_IP>:3001`, rebuild, and `pm2 restart fleet-ui`
4. Overland app endpoint: `http://<EC2_PUBLIC_IP>:3001/api/overland`.
5. Atlas: add `<EC2_PUBLIC_IP>/32` to the Atlas Network Access list, then `pm2 restart fleet-api`

Health:

- `GET /health → {"ok": true}`

Vehicles:

- `GET /api/locations → array of current snapshots`
- `GET /api/vehicle/:id/history → [{ vehicleId, lat, lng, timestamp }]`

Ingest:

- `POST /api/gps-update`
 Body :

```
{"vehicleId":"iphone-vehicle-1","latitude":53.2737,"longitude":-9.0199,"status":"moving"}
```

- POST /api/overland

Body:

```
{
  "current": {
    "geometry": { "coordinates": [-9.0199, 53.2737] },
    "properties": { "device_id": "iphone-vehicle-1", "activity": "moving" }
  }
}
```

Both return {"ok": true}

Geofences:

- GET /api/geofences → list of fences
- POST /api/geofences

```
{
  "name": "Galway City Centre",
  "type": "polygon",
  "coordinates": [[-9.061, 53.274], [-9.039, 53.281], [-9.030, 53.269]],
  "alertOnEntry": true,
  "alertOnExit": true
}
```

Note: Polygon coordinate pairs are [lng, lat]

DELETE /api/geofences/:id → {"success": true}

Violations:

- GET /api/violations → newest first
- GET /api/violations/:vehicleId → filter by vehicle
- PATCH /api/violations/:id/resolve → updated row

Notes for live use:

Serve over HTTPS, restrict CORS to the dashboard host, use a simple token on ingest, and lock the Google Maps key by referrer.

B. Environment and deployment

Stack:

AWS EC2 Ubuntu, Node.js Express, Socket.io, MongoDB Atlas, PM2.

Environment (redacted):

MONGO_URI=mongodb+srv://<user>:<password>@<cluster>/<db>?retryWrites=true&
w=majority
PORT=3001

PM2:

Start:

```
pm2 start fleet-api    # backend (index.js)
pm2 start fleet-ui     # serves React build on :3000
pm2 status
pm2 logs fleet-api --lines 20
```

CORS:

Demo uses origin: “*”. For production set origin: “https://<your-dashboard-host>” on Express and Socket.io.

Atlas IP allow list:

Add the current EC2 public IP in Atlas after any restart.

C. Test procedures and sample results

C1. Simulator run:

1. Open the dashboard.
2. In DevTools:

```
sim.start({ durationMin: 10, intervalMs: 5000, vehicleId: 'sim-1' });
```
3. Watch markers move and check the violations panel.

C2. Live run with Overland

1. Configure Overland to `http://<EC2_PUBLIC_IP>:3001/api/overland`.
2. Drive across either the Galway City Centre or ATU Galway Campus fence.
3. Verify in DevTools:

```
fetch('http://<EC2_PUBLIC_IP>:3001/api/violations')
  .then(r => r.json())
  .then(v => console.table(v.slice(-5)));
```

D. Data model (from code)

```
/* ----- Schemas ----- */
const vehicleSchema = new mongoose.Schema({
  id: String, address: String, lat: Number, lng: Number,
  status: String, lastUpdated: Date, alert: String,
});
const Vehicle = mongoose.model("Vehicle", vehicleSchema);

const vehicleHistorySchema = new mongoose.Schema({
  vehicleId: String, lat: Number, lng: Number, timestamp: Date,
});
const VehicleHistory = mongoose.model("VehicleHistory", vehicleHistorySchema);

const geofenceSchema = new mongoose.Schema({
  id: String, name: String, type: String, coordinates: Array, radius: Number,
  alertOnEntry: Boolean, alertOnExit: Boolean, createdAt: Date
});
const Geofence = mongoose.model("Geofence", geofenceSchema);

const violationSchema = new mongoose.Schema({
  vehicleId: String, geofenceId: String, geofenceName: String,
  violationType: String, lat: Number, lng: Number, timestamp: Date,
  notificationSent: Boolean, resolved: { type: Boolean, default: false },
  resolvedAt: Date
});
const Violation = mongoose.model("Violation", violationSchema);
```

Figure 10-1 Data model (index.js)

E. Code listings included

- **Backend:** index.js, notification-service.js
- **Frontend:** src/MapComponent.js, src/config.js, src/App.js

Secrets are redacted.

F. Consent statement

I consent to the collection of my device's location while testing this project. The data are used only for demonstration and grading, will not be shared with third parties, and will be deleted after submission.

Signed: Alan Hynes

Date: 31.08.2025

G. Risk log snapshot

- Atlas blocked after EC2 IP change. Fix: add new IP to allow list and restart.
- Phone upload queue grew during outages. Fix: restore API, then reinstall Overland to clear the queue.
- Jitter near fence edges caused double events. Fix: add a short merge window in the checker.

H. Glossary

Snapshot latest row per vehicle for the map.

History time series of points for a vehicle.

Violation entry or exit event recorded on a fence crossing.

Fence polygon or circle zone.

11 References

- [1] National Geospatial-Intelligence Agency, “WGS 84,” [Online]. Available: <https://earth-info.nga.mil/index.php?dir=wgs84&action=wgs84>. [Accessed 27 08 2025].
- [2] U.S. GPS.gov, “How accurate is GPS?” [Online]. Available: <https://www.gps.gov/systems/gps/performance/accuracy/>. [Accessed 27 08 2025].
- [3] MDN Web Docs, “The WebSocket API (WebSockets)” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. [Accessed 29 08 2025].
- [4] Socket.IO Docs, “Documentation, v4 – Introduction.” [Online]. Available: <https://socket.io/docs/v4/>. [Accessed 28 08 2025].
- [5] Google Developers, “Maps JavaScript API – Overview,” [Online]. Available: <https://developers.google.com/maps/documentation/javascript/overview>. [Accessed 29 08 2025].
- [6] MongoDB Atlas Docs, “Configure IP Access List Entries,” [Online]. Available: <https://www.mongodb.com/docs/atlas/security/ip-access-list/>. [Accessed 27 08 2025].
- [7] Amazon Web Services, “What is Amazon EC2?” [Online]. Available: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html>. [Accessed 29 08 2025].
- [8] PM2, “PM2 – Node.js Production Process Manager,” [Online]. Available: <https://pm2.keymetrics.io/>. [Accessed 27 08 2025].
- [9] W. R. Franklin, “PNPOLY – Point Inclusion Polygon Test,” [Online]. Available: https://wrfranklin.org/Research/Short_Notes/pnpoly.html. [Accessed 28 08 2025].
- [10] Esri, “Distance on a sphere: The Haversine Formula,” [Online]. Available: <https://community.esri.com/t5/coordinate-reference-systems-blog/distance-on-a-sphere-the-haversine-formula/ba-p/902128>. [Accessed 28 08 2025].

[11] MDN Web Docs, “WebSocket,” [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/WebSocket>. [Accessed 27 08 2025].

[12] Overland, “Overland GPS Tracking App for iPhone,” [Online]. Available: <https://overland.p3k.app/>. [Accessed 27 08 2025].

[13] ChatGPT. (GPT-5). Open AI. [Online]. Available: <https://chat.openai.com/chat>. [Accessed 30 08 2025].