Crowdsourced Map-Based Public Issue Tracker
Project Documentation

Armaan Shah, Aryan Nath, Urvashi Balasubramaniam

December 2025

1 Introduction

The Public Issue Tracker is a crowdsourced, map-based platform designed to bridge the gap between citizens and local authorities regarding civic infrastructure maintenance. Traditional methods of complaint lodging—such as written letters, generic grievance portals, or automated WhatsApp bots—often suffer from a lack of transparency, poor data utilization, and insufficient accountability.

This project introduces a transparent, visual, and data-driven approach to civic engagement. By leveraging geospatial data, gamification, and AI-driven moderation, the system empowers residents, commuters, and government officials (categorized in the system as the `PIGS` role) to report issues like potholes, garbage accumulation, and street light failures. Simultaneously, it provides a custom routing engine that helps commuters avoid hazardous zones based on real-time community reports.

2 Requirements

- User Expectations:
  1. **Ease of Reporting:** Users must be able to report issues quickly. The system must support guest submissions to preserve anonymity and encourage reporting from "Good Samaritans" who fear administrative pushback.
  2. **Visual Feedback:** The platform must provide an interactive map interface where issues are visualized as pins, allowing users to see the state of their locality at a glance.
  3. **Transparency:** Users expect to see the status of their complaints (Open, In Progress, Resolved) and track community engagement through upvotes and comments.
  4. **Safety via Routing:** Commuters expect the system to utilize the reported data to suggest safer travel routes, avoiding open hazards like open manholes or tree falls.
- Functionality:
  1. **User Accounts & Authentication:** The system must support secure login via Email/Password (using Bcrypt) and Google OAuth. It must support distinct roles: `USER`, `ADMIN`, `GUEST`, and `PIGS` (Government Officials detected via `.gov.in` domains).
  2. **Issue Reporting:** Users can report issues with title, description, category (e.g., `POTHOLE`, `STREETLIGHT_FAULT`), and precise geolocation (Lat/Lng). The system must allow image uploads which are then queued for AI moderation.
  3. **Gamification:** To incentivize participation, the system must award "Credibility Points" and badges (e.g., *Observer, Activist, Guardian*) based on user activity thresholds.
  4. **Resolution Voting:** To ensure accountability, a democratic process is required where the community votes to verify if a "Resolved" issue has actually been fixed. If the net verification score drops below a threshold (defined as -5), the issue must automatically revert to "In Progress".

5. **Custom Routing Engine:** The backend must calculate optimal paths between two points, applying dynamic penalties to road segments that are near reported issues.

- Non-Functional Requirements
  - **Performance:** The map must efficiently render thousands of markers using server-side clustering and grid-based caching (Redis) to ensure low latency.
  - **Scalability:** The backend must be stateless to allow horizontal scaling via containerization.
  - **Reliability:** The system must handle file system errors and external API failures (e.g., Google Gemini) gracefully without crashing the main application loop.
  - **Security:** All sensitive data must be hashed. API access must be secured via JWT. Content must be moderated to prevent abuse.

3 Analysis

The problem space addresses the inefficiency in current civic grievance redressal systems.

3.1 Problem Statement

1. **Data Silos:** Existing data is often lost in bureaucratic files. Our solution utilizes a persistent relational database to maintain queryable records.

2. **Lack of Accountability:** Officials often mark issues as resolved without work. Our "Resolution Vote" logic enforces community verification.

3. **Route Safety:** Standard navigation apps do not account for hyper-local, temporary civic hazards. Our custom pathfinding algorithm integrates live issue data to penalize dangerous routes.

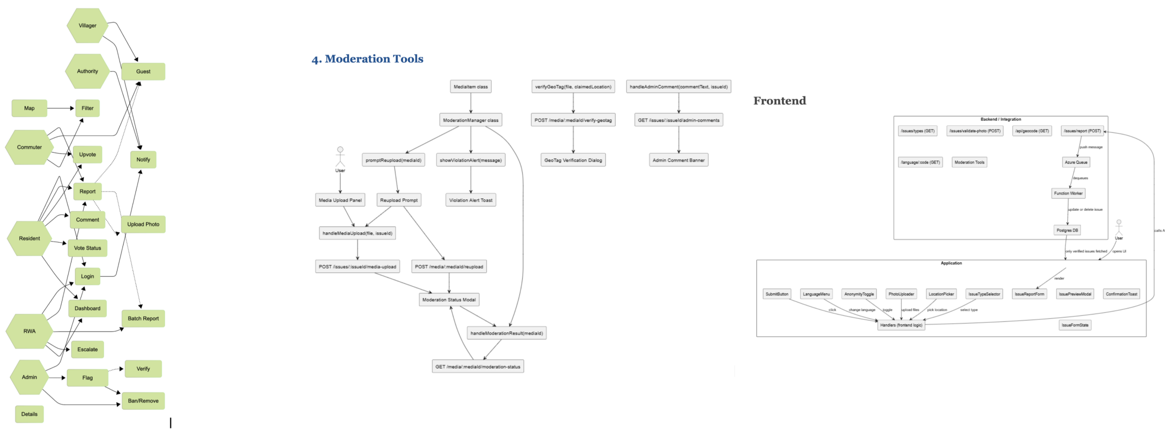

Figure 1: Use Case Diagrams for All Operations, Moderation Tools, and Frontend Reporting System

4 Specifications

This section details the technical blueprint of the Public Issue Tracker application, covering the architecture, technologies, backend specification, and specific low-level designs.

4.1 High-Level Architecture

1. **Backend:** A RESTful API built with **Node.js (Express)** and **TypeScript**. It handles business logic, routing algorithms, and data aggregation.
2. **Frontend:** A Single Page Application (SPA) built with **React.js (Vite)** and **TypeScript**. It utilizes Material UI for components and React-Leaflet for geospatial visualization.
3. **Microservices:** A Python-based Azure Function acts as a dedicated moderation service, processing images asynchronously using the Google Gemini API.
4. **Communication:** The frontend communicates with the backend via REST APIs secured by JWT tokens. The backend communicates with the moderation service via Azure Storage Queues.

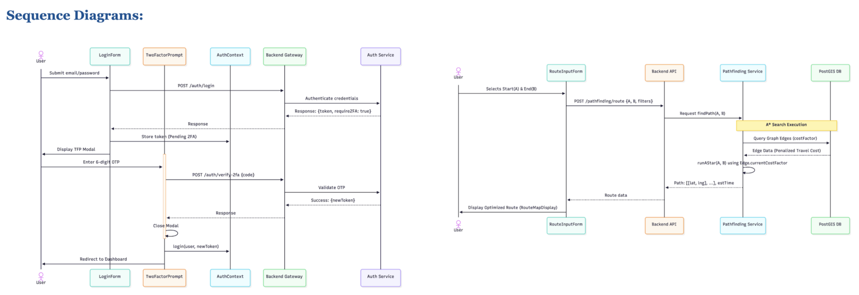

Figure 2: Sequence Diagrams

4.2 Technologies Used

1. **Application Framework:** React v19 with Vite for the frontend; Express.js for the backend.
2. **Database:** PostgreSQL accessed via Prisma ORM for relational data (Users, Issues) and graph data (Nodes, Edges).
3. **Caching:** Redis (simulated locally, Azure Redis in production) for session management and geospatial grid caching.
4. **AI & Processing:** Python, Azure Functions, and Google Gemini API (Generative AI) for content moderation.
5. **Mapping:** Leaflet, OpenStreetMap, Overpass API (for graph data import), and Turf.js for geospatial calculations.

4.3 Backend Specification

The backend provides a robust set of controllers and services.

4.3.1 Key Controllers and Functions

- **IssueController** (`issueController.ts`):
  - `createIssue`: Handles file uploads via Multer, creates database records, and triggers the `recalculatePenalties` service. It also pushes a task to the moderation queue.
  - `getIssuesInBounds`: Implements a specialized retrieval logic that calculates urgency scores based on time elapsed, upvotes, and comment counts.
- **ResolutionController** (`resolutionController.ts`):

  - `castResolutionVote`: Records a user's verification of a fix. It calculates a net score; if the score ≤ -5, it triggers `revertIssue`, automatically reopening the ticket and logging a system comment.

- **RouteController (`routeController.ts`):**
  - `calculateRoute`: Accepts start/end coordinates and invokes the `PathfindingService` to return an optimal path avoiding hazards.

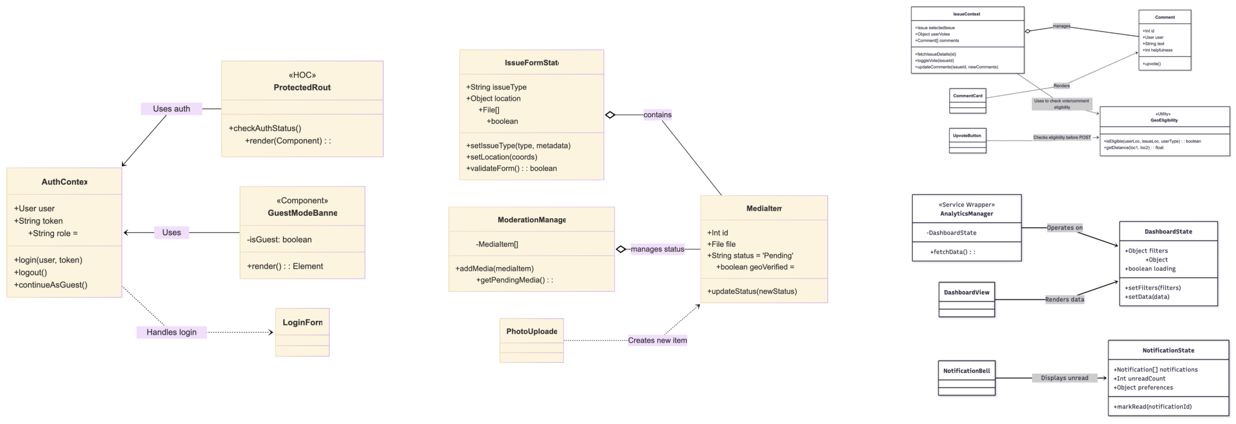

Figure 3: Authentication and Authorization — Form State and Content Management — Dashboard and Notifications

4.4 Low-Level Design: Data Structures & Algorithms

4.4.1 Data Structures

The application uses a hybrid approach, storing relational data and graph structures within PostgreSQL.

- **Issue Model:** Stores geolocation (`latitude`, `longitude`), metadata (`issueType`, `severity`, `status`), and relations to Users and Comments.
- **Graph Network:** To support custom routing, the road network is imported from OpenStreetMap into two specific tables:
  - `GraphNode`: Represents intersections/points (`id`, `osmId`, `lat`, `lng`).
  - `GraphEdge`: Represents roads connecting nodes. Crucially, it contains a `penalty` field (float, default 1.0). This multiplier is dynamically updated based on active issues.

4.4.2 Algorithms

1. Grid-Based Caching Strategy To solve the "slow startup" problem common in map apps with thousands of markers, the backend implements a geospatial caching mechanism in `IssueCacheService.ts`.

- The map is divided into grid cells of size $0.01° \times 0.01°$ (approx. 1km).
- Grid keys are generated as strings: `issues:grid:lat:lng`.
- When `getIssuesInBounds` is called, the service calculates relevant keys and fetches summaries from Redis in parallel.
- **Cache Invalidation:** When an issue is reported or updated, `invalidateIssueCache` clears the specific grid cell key to ensure data consistency.

2. Custom A* Pathfinding with Dynamic Penalties Unlike standard routing services (like OSRM), our routing engine (`PathfindingService.ts`) accounts for temporary civic hazards.

- **Penalty Calculation (`PenaltyService.ts`):** Active issues exert a "penalty field" on nearby edges.
  - `POTHOLE`: 1.5x penalty.
  - `ROAD_DAMAGE`: 5.0x penalty.
  - `TREE_FALL`: 10.0x penalty.
- **Graph Construction:** An adjacency list is built in-memory from `GraphEdge` data.
- **Heuristic:** The algorithm uses the Haversine distance (calculated via `turf.distance`) as the heuristic $h(n)$.
- **Execution:** A standard A* search is performed using a Binary Heap (via `tinyqueue`) to minimize $f(n) = g(n) + h(n)$, where $g(n)$ is the accumulated cost (distance $\times$ penalty).

3. Automated Moderation Pipeline To manage content at scale, an asynchronous pipeline is implemented:

1. The backend pushes a message to **Azure Storage Queue** (`QueueService.ts`).
2. A **Python Azure Function** triggers on the message.
3. It invokes the **Google Gemini API** with a specific prompt to evaluate:
   - **Safety:** Is the image NSFW?
   - **Relevance:** Does the image match the reported issue type?
   - **Severity:** A score from 1-5.
4. **GPS Verification:** The Python function extracts EXIF GPS data. It calculates the Haversine distance between the image location and the reported coordinates. If the distance > 100 meters, the issue is flagged as `INVALID_LOCATION`.

4.5 Codebase Organization & Implementation Structure

The project is organized as a monorepo containing three distinct directories, each representing a specific layer of the application architecture. This separation of concerns ensures maintainability and scalability.

4.5.1 Backend Application (`segfault-backend`)

The backend is a Node.js/Express application written in TypeScript. It follows a layered architecture, separating API definitions, business logic, and data access.

- **API Layer (`src/api/`)**
  - `routes/`: Defines the REST endpoints (e.g., `issueRoutes.ts`, `adminRoutes.ts`). It maps HTTP verbs to specific controller functions.
  - `controllers/`: Handles incoming HTTP requests, validates input, and orchestrates calls to services. Key controllers include:
    - `analyticsController.ts`: Aggregates data for dashboards and heatmaps.
    - `resolutionController.ts`: Manages the voting logic and automatic reversion of resolved status.
    - `routeController.ts`: The entry point for navigation requests.
  - `middleware/`: Contains `authMiddleware.ts` which intercepts requests to verify JWT tokens and checks against the Redis session store.
- **Service Layer (`src/services/`)** Contains the core complex business logic, isolated from the HTTP transport layer:

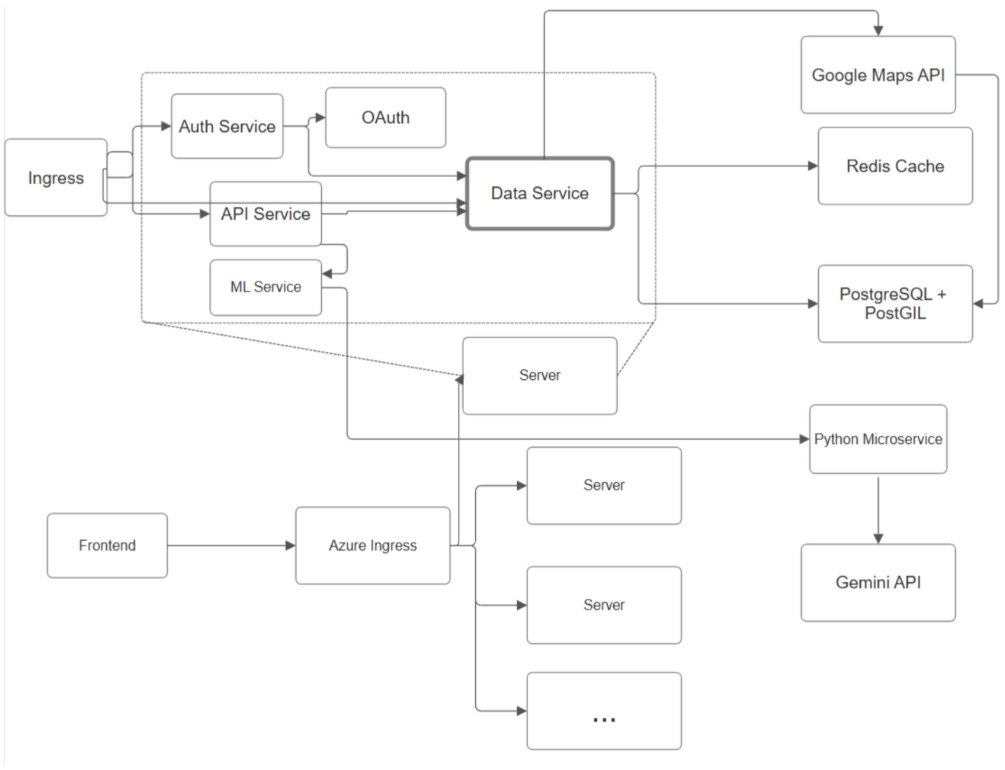

Figure 4: Data Flow Diagram

  - `PathfindingService.ts`: Implements the A* algorithm using `tinyqueue` and `turf.js` for distance calculations. It constructs the graph in-memory from the database.
  - `PenaltyService.ts`: Executes spatial queries to find road edges near reported issues and applies penalty multipliers (e.g., 5.0x for Road Damage).
  - `IssueCacheService.ts`: Manages the Redis caching strategy, dividing the map into geographical grids to serve issue summaries efficiently.
  - `GamificationService.ts`: handles logic for awarding credibility points and assigning badges like "Guardian" based on user activity thresholds.
  - `QueueService.ts`: Interfaces with Azure Storage Queues to offload image processing tasks.

- **Data Access Layer** (`src/data/`)
  - `prisma/`: Contains the `schema.prisma` file which serves as the single source of truth for the database structure, defining enums (`UserRole`, `IssueType`) and relationships.
  - `redisClient.ts`: A wrapper around the Redis connection, providing helper methods for caching strings and JSON objects with TTL (Time To Live).

- **Scripts** (`src/scripts/`)
  - `importGraph.ts`: A utility script that fetches OpenStreetMap data via the Overpass API for a defined bounding box (Delhi) and populates the `GraphNode` and `GraphEdge` tables.

4.5.2 Frontend Application (`segfault-frontend`)

The frontend is a React application built with Vite, structured to separate global state, UI components, and page views.

- **State Management** (`src/state/`)
  - `authContext.tsx`: Manages the user's session state. It handles JWT decoding to determine roles (e.g., detecting `isGov` for government officials) and persists authentication across reloads.
- **API Integration** (`src/api/`)
  - `axios.ts`: Configures the Axios instance with interceptors to automatically inject the `Authorization: Bearer` token into headers.
  - `routes.ts`: A centralized definition of all backend endpoints, typed with TypeScript interfaces (e.g., `Issue`, `MapIssue`) to ensure type safety across the network boundary.
- **Core Components** (`src/components/`)
  - **Dashboard/**: Contains high-level UI blocks like `MapInterface.tsx` (the main map view), `StatsGrid.tsx` (analytics cards), and `IssueReportForm.tsx` (the multi-step reporting wizard).
  - **Map/**: Leaflet-specific logic.
    - `IssueMarker.tsx`: Renders individual pins, dynamically changing icons based on urgency or status.
    - `LocationPicker.tsx`: Allows users to drag a pin to refine GPS coordinates.
    - `markerIcons.ts`: Generates SVG icons programmatically based on urgency scores (Green $\rightarrow$ Red gradient).
  - **Routing/**: Contains `RouteRenderer.tsx`, which draws the polyline path returned by the backend on the map.
- **Pages** (`src/pages/`) Represents full-screen views handled by React Router, such as `Landing`, `Login`, `Dashboard`, and the protected `AdminDashboard`.

4.5.3 Moderation Microservice (`segfault-moderation`)

This is a standalone Python-based Azure Function App designed for asynchronous processing.

- `function_app.py`**:** The core entry point.
  - **Trigger:** It listens to the `issue-moderation` queue.
  - **Logic:** It downloads the image blob, extracts EXIF GPS data to validate location, and sends the image to the Google Gemini API for content analysis.
  - **Database Interaction:** It connects directly to PostgreSQL using `psycopg2` to update the issue's `authorized`, `error`, and `severity` fields based on the AI's verdict.
- `requirements.txt`**:** Lists dependencies including `google-generativeai` for the LLM interaction and `Pillow` for image manipulation.

4.6 Pipeline for File Communication and State Management

1. **Authentication State:** The frontend uses a Context API provider (`authContext.tsx`) to manage user state. JWT tokens are stored in `localStorage`. Axios interceptors automatically attach the `Bearer` token to every outgoing request.
2. **Guest Session Handling:** For anonymous reporting, the backend generates a unique `GuestToken` linked to a UUID. This allows the system to track the guest's specific issues (via `getIssuesByGuestToken`) without requiring a full user profile.
3. **Map Data Flow:**
   (a) The `MapInterface` component listens for map move events.
   (b) It calculates the new bounding box.
   (c) A debounced API call fetches issue summaries.
   (d) `MarkerClusterGroup` aggregates pins on the client side to maintain rendering performance.

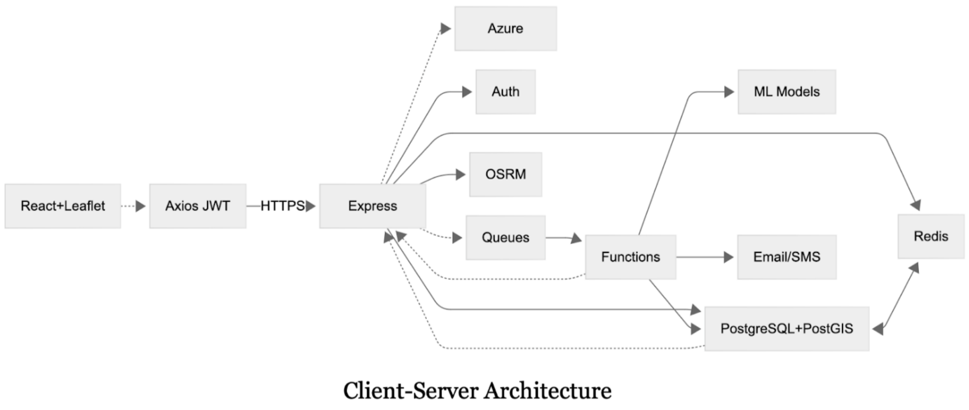

Figure 5: Client Server Architecture Diagram

4.7 Error Conditions and Handling

1. **Geofencing Violations:**
   - *Condition:* A user attempts to vote or comment on an issue while physically distant from the location.
   - *Handling:* The backend calculates the distance between the user's current location (sent via API) and the issue location using `turf.distance`. If distance $>$ 5km, a 403 Forbidden error is returned with a specific error message.
2. **AI Service Failures:**
   - *Condition:* Google Gemini API rate limits or times out.
   - *Handling:* The Python function catches exceptions. If analysis fails, it defaults to a "safe" state but logs the error to Azure Application Insights, ensuring the queue message doesn't block indefinitely.
3. **Graph Data Gaps:**
   - *Condition:* Users request a route in an area where OSM data hasn't been imported.
   - *Handling:* The `PathfindingService` checks for the nearest nodes. If snapping fails, it returns a 404 with a descriptive message ("No route found... area may not have road data loaded").

5 Theory

5.1 Computational Theory and Operating System Principles

The project leverages several core OS and computational principles:

- **Concurrency and Event Loop:** The Node.js backend relies on the single-threaded event loop. By offloading CPU-intensive tasks (image processing) to Azure Functions and I/O-intensive tasks (database queries) to the thread pool, the main application remains responsive.
- **Caching Mechanisms:** The implementation of the Redis cache for geospatial grids demonstrates the principle of locality. By storing spatially adjacent data together and retrieving it based on grid keys, we reduce the I/O cost of frequent database hits.

5.2 Algorithms: A* Pathfinding

The core innovation is the modification of the classic A* algorithm.

- **Standard A*:** Finds the shortest path based on physical distance.
- **Modified A*:** Finds the "lowest cost" path. The cost function $g(n)$ is modified:

  $$g(n) = g(n-1) + (distance(n-1,n) \times penalty(n-1,n))$$

  where *penalty* is dynamically updated by the `PenaltyService` based on active issue reports (e.g., $penalty = 5.0$ for road damage). This essentially "stretches" the mathematical length of damaged roads, forcing the algorithm to find alternative, safer routes.

5.3 Gamification Theory

The system implements a feedback loop to sustain user engagement:

- **Triggers:** External (seeing a pothole) and Internal (desire for better infrastructure).
- **Action:** Reporting the issue.
- **Variable Reward:** Earning "Credibility Points" and badges like "Guardian" (awarded at 500 pts). The variability comes from community upvotes and successful resolution.
- **Investment:** The user builds a reputation score stored in the `User` table, increasing their stake in the platform.

6 Testing

6.1 Testing Methodology

1. **Unit Testing:** Used to verify the logic of isolated components, particularly the gamification logic (point calculation) and pathfinding heuristics.
2. **Simulation Scripts:** Instead of manual data entry, custom TypeScript scripts were developed to simulate realistic load and behavior.

6.2 Test Scenarios & Scripts

The codebase includes specific scripts in the `scripts/` directory:

- **User Seeding (`createTestUsers.ts`):**
  - *Purpose:* To populate the leaderboard and provide actors for interaction.
  - *Logic:* It iteratively registers 100 users using a predefined list of Indian first names and surnames (e.g., "Aarav Sharma"), generating unique emails and linking them to the test database.
- **Issue Simulation (`createTestIssues.ts`):**
  - *Purpose:* To test map clustering and heatmap rendering.
  - *Logic:* It generates issues within a specific bounding box (Delhi: 28.60 - 28.68 N, 77.18 - 77.25 E). It uses a clustering algorithm where 30% of new issues are randomly offset from existing locations to simulate real-world problem hotspots.
- **Interaction Simulation (`createTestInteractions.ts`):**
  - *Purpose:* To test urgency score calculations and notification triggers.
  - *Logic:* Implements a "rich-get-richer" algorithm. It selects issues weighted by their current engagement (upvotes + comments), then applies random actions (80% chance to upvote, 15% comment, 5% comment upvote) to simulate organic viral growth of critical issues.