

# Police Department Web System (*LA\_Noire*)

## Final Project Report

Kiarash Joolaei (Student ID: 400100949)

Mohammad Mahdi Farhadi (Student ID: 99105634)

February 27, 2026

### Abstract

This report documents the implementation of a web-based police department system inspired by *L.A. Noire*. It summarizes team responsibilities, conventions, project management, key system entities, selected frontend NPM packages, and the role of AI-assisted development. All claims are grounded in the repository state and code structure (branch `front`, snapshot commit `3001e5a` dated 2026-02-27).

## Contents

<b>1</b>	<b>Project Overview</b>	<b>2</b>
1.1	Goal . . . . .	2
1.2	Technology Stack . . . . .	3
<b>2</b>	<b>Evaluation Criteria and Evidence</b>	<b>3</b>
2.1	Repository Snapshot and Commit Frequency . . . . .	3
2.2	Checklist Alignment (High-Level) . . . . .	3
<b>3</b>	<b>Team Responsibilities and Contributions</b>	<b>4</b>
3.1	Mohammad Mahdi Farhadi . . . . .	4
3.2	Kiarash Joolaei . . . . .	4
<b>4</b>	<b>Project Management Approach</b>	<b>4</b>

<b>5 Development Conventions</b>	<b>4</b>
5.1 Commit Message Format . . . . .	4
5.2 Naming, Structure, and Error Handling . . . . .	5
5.3 Swagger/OpenAPI as a First-Class Deliverable . . . . .	5
<b>6 Key System Entities and Justification</b>	<b>5</b>
6.1 Identity and Access . . . . .	5
6.2 Case Formation and Lifecycle . . . . .	5
6.3 Evidence . . . . .	6
6.4 Detective Reasoning and Link Analysis . . . . .	6
6.5 Suspects, Interrogations, and Trials . . . . .	6
6.6 Cross-Cutting Workflow Entities . . . . .	6
<b>7 NPM Packages Used (Up to Six) and Justification</b>	<b>7</b>
7.1 react-router-dom . . . . .	7
7.2 @tanstack/react-query . . . . .	7
7.3 axios . . . . .	7
7.4 zustand . . . . .	7
7.5 react-hook-form + zod (@hookform/resolvers) . . . . .	7
7.6 reactflow . . . . .	7
<b>8 AI-Generated Code: Examples and Evaluation</b>	<b>8</b>
8.1 Examples of AI-Assisted Code in the Repository . . . . .	8
8.2 Strengths and Weaknesses of AI in Front-End Development . . . . .	8
8.3 Strengths and Weaknesses of AI in Back-End Development . . . . .	9
<b>9 Requirement Analysis: Initial vs Final</b>	<b>9</b>
9.1 Initial Analysis . . . . .	9
9.2 Final Analysis and Adjustments . . . . .	9
<b>10 Conclusion</b>	<b>9</b>

# 1 Project Overview

## 1.1 Goal

The project digitizes end-to-end police workflows: case formation (complaints and crime-scene reporting), evidence registration and review, detective reasoning via a “detective

board”, suspect handling and most-wanted ranking, interrogation approvals, trial verdict entry, tips/rewards, notifications, and an optional payment flow for bail/fines.

## 1.2 Technology Stack

The implementation is a containerized three-tier system: (1) Django REST Framework (backend) with PostgreSQL storage, and (2) a React (Vite) + TypeScript frontend. Docker Compose orchestrates the database, backend, and frontend containers (see `docker-compose.yml`, `backend/Dockerfile`, `frontend/Dockerfile`).

# 2 Evaluation Criteria and Evidence

The project rubric emphasizes meeting documented requirements, maintainable engineering practices, correct REST behavior, access control, and complete Swagger/OpenAPI documentation with meaningful examples. In addition, a minimum commit count per checkpoint is required.

## 2.1 Repository Snapshot and Commit Frequency

At the snapshot used for this report (2026-02-27), the branch contains 52 commits. The contributions are visible through git history and author shortlog:

Author (git)	Commits	Notes
KiaJJ	30	Integration, frontend, backend hardening, docs/tests
Mohammad Mahdi	21	Backend core structure and initial features
thisismmf	1	Initial commit identity

This comfortably exceeds the “15 commits per checkpoint” requirement while preserving small, focused changesets.

## 2.2 Checklist Alignment (High-Level)

The repository addresses key checklist items via: RBAC (`backend/apps/rbac/`), error normalization (`backend/police\_portal/api\_exceptions.py`), Swagger generation and customization (`backend/police\_portal/schema.py`), workflow-focused domain apps (`backend/apps/`), and automated tests across multiple apps (`backend/apps/*/tests/`).

## 3 Team Responsibilities and Contributions

### 3.1 Mohammad Mahdi Farhadi

Mohammad built the foundation that enabled incremental development: (a) Docker packaging and runtime orchestration (e.g., early Docker/Compose setup in commit `889cb45`), (b) core backend scaffolding (Django project structure and the initial domain decomposition into apps), and (c) baseline operational documentation and repository hygiene. His contributions are concentrated in the initial backend feature set on 2026-02-20, visible as “feat(...)” commits across multiple domain apps.

### 3.2 Kiarash Joolaei

Kiarash focused on closing gaps between requirements and implementation and on delivering the frontend: (a) backend hardening to unblock and correctly support the UI (e.g., making the homepage statistics endpoint public when required), (b) Swagger/OpenAPI completeness improvements and regression tests to prevent documentation drift, and (c) frontend pages, role-based dashboard modules, workflow integration, and frontend tests. These contributions appear in the later integration commits (2026-02-26 to 2026-02-27), including many `feat(frontend)` commits and backend access-control fixes.

## 4 Project Management Approach

We decomposed work into tasks that mirror the project description workflows and the evaluation checklist, then distributed ownership by area: Mohammad established the platform and backend baseline; Kiarash iterated on correctness, documentation quality, and frontend integration. Acceptance criteria were derived directly from the rubric (role-based flows, clean error handling, complete Swagger, and test coverage).

## 5 Development Conventions

### 5.1 Commit Message Format

Commits follow a Conventional Commits style: `type(scope): short description`. Scopes are used when changes are clearly bounded (e.g., `frontend`, `backend`, `docker`); otherwise a plain `type:` prefix is used.

## 5.2 Naming, Structure, and Error Handling

The backend follows Django/DRF conventions and keeps each domain app self-contained with `models.py`, `serializers.py`, `views.py`, `urls.py`, and `tests/`. Python uses standard naming (PascalCase classes, snake\_case functions). The frontend uses `frontend/src/pages/*Page.tsx` for pages and collocates API clients and utilities for each workflow module.

To keep frontend error handling predictable, backend errors are normalized into an envelope format via `backend/police\_portal/api\_exceptions.py`, and OpenAPI descriptions explicitly document that envelope.

## 5.3 Swagger/OpenAPI as a First-Class Deliverable

Swagger documentation is generated with `drf-spectacular` and intentionally made rubric-compliant with per-endpoint descriptions and domain-specific examples. Schema customization is centralized in `backend/police\_portal/schema.py`, and a regression test suite (`backend/apps/stats/tests/test\_stats\_docs.py`) validates that operations include descriptions and non-generic JSON examples.

# 6 Key System Entities and Justification

The entity model is the backbone of the system and was designed to reflect the workflow narrative from the project description. This section maps directly to Django models under `backend/apps/*/models.py`.

## 6.1 Identity and Access

**Role** and **UserRole** (`backend/apps/rbac/models.py`) exist to support dynamic roles (add/remove/modify roles without code changes) and to enforce role-based access control across endpoints. Actors are tracked throughout workflows via the custom **User** model (`backend/apps/accounts/models.py`), enabling auditability (who created, reviewed, approved, or rejected).

## 6.2 Case Formation and Lifecycle

**Complaint** and **Case** (`backend/apps/cases/models.py`) represent the two case-formation sources required by the spec (complaint-based and crime-scene-based). They store status, timestamps, and the actors who created or reviewed the data. **CaseComplainant** exists to support multiple complainants with cadet verification states, and **CaseAssignment** exists

to enforce case-scoped responsibility (e.g., only the assigned detective can perform certain operations).

### 6.3 Evidence

The abstract **Evidence** model provides a uniform wrapper (title, description, registrant, timestamps) required for consistent listing and reporting. Typed evidence models (witness statements, medical, vehicle, identity-document evidence) exist because each type carries unique constraints and validation requirements (implemented in evidence serializers under `backend/apps/evidence/serializers.py`).

### 6.4 Detective Reasoning and Link Analysis

**DetectiveBoard**, **BoardItem**, and **BoardConnection** (`backend/apps/board/models.py`) exist to implement the detective board: free positioning, evidence references, notes, and red-line connections between items. The frontend renders the board using graph UI primitives and supports export-to-image behavior.

### 6.5 Suspects, Interrogations, and Trials

**Person**, **SuspectCandidate**, and **WantedRecord** (`backend/apps/suspects/models.py`) separate identity from case-specific suspicion and from wanted status. The most-wanted ranking and reward formula is computed deterministically in `backend/apps/suspects/utils.py`. **Interrogation** (`backend/apps/interrogations/models.py`) captures detective and sergeant scoring and the captain/chief approval chain. **Trial** (`backend/apps/trials/models.py`) stores judge decisions and punishment details, backed by reporting endpoints that expose a case dossier.

### 6.6 Cross-Cutting Workflow Entities

**Notification** provides required workflow notifications. **Tip**, **TipAttachment**, and **RewardCode** model the tips/rewards queue and redemption flow. **Payment** supports the optional bail/fine gateway flow and return URL behavior. These are implemented in `backend/apps/notifications/models.py`, `backend/apps/rewards/models.py`, and `backend/apps/payments/models.py`.

## 7 NPM Packages Used (Up to Six) and Justification

The frontend intentionally uses the following six packages (see `frontend/package.json`) because they directly support rubric-relevant behaviors and complex workflows:

### 7.1 `react-router-dom`

Provides role-aware routing and nested layouts for many workflow pages, including guarded routes for authenticated-only modules.

### 7.2 `@tanstack/react-query`

Manages server state (queues, details, mutations) with caching, deduplication, and consistent loading/error behavior, reducing accidental over-fetching during complex flows.

### 7.3 `axios`

Implements a consistent HTTP client and supports interceptors for JWT headers and centralized error handling. API clients are implemented around it (e.g., `frontend/src/api/client.ts`).

### 7.4 `zustand`

Provides lightweight global state for authentication/session persistence and cross-page state without heavyweight boilerplate.

### 7.5 `react-hook-form + zod (@hookform/resolvers)`

Supports complex, validated forms (complaint submission, evidence creation, and admin role/user changes) with typed schema validation and ergonomic form state.

### 7.6 `reactflow`

Enables node/edge interaction primitives that map cleanly onto the detective board workflow (dragging items and drawing connections).

## 8 AI-Generated Code: Examples and Evaluation

We used AI assistance primarily to accelerate boilerplate-heavy or highly repetitive tasks. We then relied on manual review and automated tests to keep behavior requirements-aligned.

### 8.1 Examples of AI-Assisted Code in the Repository

**Example 1: Public homepage statistics endpoint.** The frontend homepage requires anonymous access to aggregated counters. The backend’s secure default is authenticated access, so this endpoint must explicitly allow anonymous requests. In `backend/apps/stats/views.py`, `StatsOverviewView` overrides permissions:

```
class StatsOverviewView(APIView):
    permission_classes = [AllowAny]
```

**Example 2: Domain-specific OpenAPI examples.** Generic placeholder examples do not meet the “complete Swagger” rubric. We therefore curated domain-specific request/response examples in `backend/police\_portal/schema.py` and attached them systematically during schema generation.

```
SERIALIZER_REQUEST_EXAMPLES = {
    "apps.accounts.serializers.LoginSerializer": {
        "identifier": "officer@example.com",
        "password": "Pass1234!",
    },
}
```

**Example 3: Regression tests to prevent documentation drift.** To ensure AI-assisted docs remain accurate, `backend/apps/stats/tests/test\_stats\_docs.py` asserts that operations have descriptions and JSON examples, and that examples are non-generic.

### 8.2 Strengths and Weaknesses of AI in Front-End Development

In this project, AI was strongest at accelerating repetitive UI scaffolding (page shells and forms), suggesting mutation/query patterns, and generating initial testing ideas. Its main weakness was correctly inferring nuanced workflow rules (role-scoped visibility and multi-step approvals); those required careful manual alignment with the requirements and backend enforcement.

## 8.3 Strengths and Weaknesses of AI in Back-End Development

AI was effective for drafting DRF boilerplate, serializer/view wiring, and documentation/test scaffolding. However, it tended to miss subtle authorization and state-machine constraints unless explicitly guided by the workflow narrative. For this reason, we encoded access policies and added tests that enforce both behavior and documentation correctness.

# 9 Requirement Analysis: Initial vs Final

## 9.1 Initial Analysis

Early design decisions prioritized security and maintainability: default authenticated access (with explicit `AllowAny` only where required), a dynamic role system, and explicit workflow state enums. These aligned with the rubric's focus on access control and clean entity design.

## 9.2 Final Analysis and Adjustments

During frontend integration, we refined requirements interpretation at the boundaries where the UI expects public access or where role membership alone is insufficient. For example, public pages (homepage statistics and public most-wanted views) explicitly override the authenticated default, and sensitive operations require case-scoped assignment checks rather than broad role access. We also treated Swagger as a graded deliverable by attaching domain-specific examples and validating them through regression tests.

# 10 Conclusion

The final system provides a role-based implementation of the specified police workflows with an emphasis on clean domain separation, enforceable access policies, reliable documentation, and test-backed correctness. The division of responsibilities enabled rapid progress: a strong backend/Docker foundation followed by requirement-driven hardening and full frontend integration.