# LA Noire Police Department Web System
# Project Report

Kiarash Joolaei (400100949)
Mohammad Mahdi Farhadi (99105634)

February 27, 2026

# Contents

# Chapter 1

# Project Overview

## 1.1 Goal

Build a web-based system that digitizes police department workflows inspired by *L.A. Noire*, using:

- Front-end: React (Vite) + TypeScript
- Back-end: Django REST Framework
- Database: PostgreSQL
- Full Docker Compose packaging (db + backend + frontend)

The system implements role-based workflows for case formation, evidence registration, detective reasoning (board), suspect handling, interrogations, trial verdict entry, tips/rewards, notifications, and an optional payment flow for bail/fines.

## 1.2 Repository Structure (High-Level)

- `backend/`: Django project (`police_portal`) and domain apps in `backend/apps/*`
- `frontend/`: Vite React TypeScript app with pages, API clients, state management, and tests
- `docker-compose.yml`: PostgreSQL + Django + frontend containers

## 1.3 Implementation Notes That Matter For Consistency Checks

- Default backend permission is authenticated (`IsAuthenticated`), with explicit `AllowAny` only where required (e.g., login and public pages). See `backend/police_portal/settings.py`.
- Error responses are normalized into an envelope: `{ "error": { "code", "message", "details" } }` via `backend/police_portal/api_exceptions.py`.

- Swagger/OpenAPI documentation is produced by `drf-spectacular` and is enforced via regression tests in `backend/apps/stats/tests/test_stats_docs.py`.

# Chapter 2

# Evaluation Criteria Mapping (What Was Required vs What We Built)

This page summarizes the evaluation expectations from the project description and points to the concrete implementation in this repository.

## 2.1 Global Evaluation Expectations

### 2.1.1 1) Meet the documented requirements

We implemented the required workflows as domain apps and corresponding UI modules:

- Backend domain apps: `backend/apps/*` (cases, evidence, board, suspects, interrogations, trials, rewards, payments, notifications, stats, rbac, accounts)
- Frontend workflow pages: `frontend/src/pages/*Page.tsx` (cases, evidence, detective board, most wanted, reporting, tips/rewards, payments, notifications, admin panel)

### 2.1.2 2) Clean and maintainable code (software engineering principles)

Concrete practices in this repo:

- Layered DRF structure per app (models/serializers/views/urls/tests).
- Centralized error envelope handler (`backend/police_portal/api_exceptions.py`) to keep API behavior predictable for the UI.
- Explicit workflow policies/helpers to avoid scattered role logic (`backend/apps/cases/policies.py`, `backend/apps/suspects/utils.py`).

### 2.1.3 3) Commit frequency requirement (minimum 15 commits per checkpoint)

As of **February 27, 2026**, the main delivery branch (`front`) contains **52 commits** (`git rev-list -count HEAD`), and contributions are split across both team members (`git shortlog -sne HEAD`).

## 2.2 Checkpoint 1 (Backend-Focused) Requirements

The description emphasizes:

- Proper error handling
- Proper access-level management
- REST principles
- Complete Swagger documentation (with meaningful examples and explanations)
- Tests (at least 5 tests in two different apps)
- Dockerization
- Aggregated statistics endpoint
- Correct most-wanted ranking and reward logic

Repository evidence:

- Error envelope: `backend/police_portal/api_exceptions.py`
- Access control defaults: `backend/police_portal/settings.py` (`IsAuthenticated` by default) and per-endpoint overrides (`AllowAny` where needed)
- RBAC and role flexibility: `backend/apps/rbac/*`
- Swagger: `drf-spectacular` wiring (`backend/police_portal/urls.py`) and schema customizations (`backend/police_portal/schema.py`)
- Swagger regression tests: `backend/apps/stats/tests/test_stats_docs.py`
- Aggregated stats endpoint: `backend/apps/stats/views.py`
- Most-wanted formula: `backend/apps/suspects/utils.py`
- Docker and Compose: `backend/Dockerfile`, `docker-compose.yml`
- Backend tests: `pytest.ini` plus multiple tests under `backend/apps/*/tests/`

## 2.3 Checkpoint 2 (Frontend-Focused) Requirements

The description emphasizes:

- Role-based, responsive UI
- Modular dashboard
- Detective board, most-wanted page, case/complaint status, evidence registration/review, admin panel (non-Django admin)
- Loading indicators and error display
- Proper frontend state management
- Frontend tests (at least 5)
- Full Docker Compose packaging

Repository evidence:

- Role-based dashboard modules: `frontend/src/features/dashboard/modules.ts`
- Implemented pages: `frontend/src/pages/*Page.tsx` (see `frontend/README.md` for the module list)
- Frontend tests: `frontend/src/pages/*.test.tsx` and utility tests (run via `npm test`)
- State management:

- Server state: React Query (`@tanstack/react-query`) - Client auth persistence: Zustand (`zustand`)

- Dockerization:

- Frontend container: `frontend/Dockerfile`, `frontend/nginx.conf` - Compose wiring: `docker-compose.yml`

# Chapter 3

# Team Responsibilities

This section is intentionally aligned with the git history and the actual codebase.

## 3.1 Mohammad Mahdi Farhadi (99105634)

Primary responsibilities:

1. **Docker packaging and runtime orchestration** The initial setup for Docker and backend dependencies exists as early commits such as `889cb45` ("add docker setup and backend dependencies") and the Compose-based structure that includes PostgreSQL. 2. **Core backend scaffolding** Established the Django project scaffold and the initial domain split into apps (accounts/rbac/cases/evidence/board/suspects/interrogations/trials/rewards/payments/notifications/stats). 3. **README and operational documentation (initial baseline)** Wrote the initial run instructions and baseline repository hygiene (`fe58b00` and related changes).

Evidence in repository: commit author "Mohammad Mahdi" appears on the early backend foundation commits (scaffold, dependencies, and feature modules), visible in `git log`.

## 3.2 Kiarash Joolaei (400100949)

Primary responsibilities:

1. **Backend hardening to match requirements and to unblock frontend** Fixed logical and implementation issues in the backend so that frontend development could rely on correct workflow rules and access control. Examples include: - Making the homepage stats endpoint explicitly public (`backend/apps/stats/views.py`, commit `01ec0d2`). - Enforcing case-scoped ownership/assignment access checks in workflows (e.g., `backend/apps/cases/policies.py`, `backend/apps/suspects/views.py`). 2. **Swagger/OpenAPI completeness improvements** Implemented domain-specific request/response examples and per-endpoint descriptions across the API surface (`backend/police_portal/schema.py`) and added regression tests to keep documentation quality from drifting (`backend/apps/stats/tests/test_stats_docs.py`), mainly in commits `01ec0d2` and `3bb278c`. 3. **Frontend development and integration** Implemented the React UI, role-based dashboard modules, workflow pages, and frontend testing (see `frontend/src/pages/*`

and commits prefixed `feat(frontend)` and `test(frontend)`).

Evidence in repository: commit author "KiaJJ" appears on the integration and polish commits (frontend, docs, tests, and backend fixes), visible in `git log`.

## 3.3 Shared Responsibilities

- Both team members followed a consistent commit-message convention and worked in a checkpoint-driven delivery model.
- Both participated in requirement interpretation and iterative correction, as allowed by the project rubric (backend modifications during the frontend checkpoint were permitted and expected when requirement analysis gaps are found).

# Chapter 4

# Project Management Approach

## 4.1 Task Creation

We decomposed the project into tasks that directly mirror the rubric and workflow chapters from the project description:

- Authentication and registration
- RBAC (dynamic roles + assignment)
- Case formation (complaint flow + crime scene flow)
- Evidence registration (typed evidence with validations)
- Case resolution (detective reasoning and handoff)
- Suspect identification + interrogation approvals
- Trial verdict entry
- Notifications
- Rewards/tips workflow
- Optional payment workflow (bail/fine)
- Swagger completeness and tests
- Dockerization and operational runbook

This decomposition is visible in the repository through the Django app structure (`backend/apps/*`) and through commit scopes/types (examples in Development Conventions).

## 4.2 Task Distribution

- Mohammad owned the initial platform work (Docker, backend scaffolding, and baseline documentation), enabling the rest of the system to be built incrementally.
- Kiarash owned the integration work: fixing backend workflow mismatches discovered during UI work, completing frontend modules, and raising documentation/testing quality to satisfy evaluation criteria.

## 4.3 Iteration and Acceptance

Acceptance criteria were taken from the rubric checklists (e.g., "complete Swagger documentation", "proper access-level verification", and specific pages/modules on frontend).

We treated tests as part of the definition-of-done:

- Backend: `pytest` + API-level flow tests and schema regression tests.
- Frontend: `vitest` + Testing Library tests for key pages and utilities.

# Chapter 5

# Development Conventions

## 5.1 Commit Message Format

Commit messages follow a "Conventional Commits"-style pattern:

`type(scope): short description`

Examples from `git log -oneline` include:

- `feat(frontend): implement case and complaint workflow operations`
- `feat(backend): enforce workflow ownership and role access across cases`
- `test: cover queues, access enforcement, rewards, and payment validation`
- `docs(frontend): add runbook and environment examples`
- `chore(docker): add frontend container and compose integration`

Scopes are used when a change is clearly bounded (e.g., `frontend`, `backend`, `docker`); otherwise `type:` is used.

## 5.2 Naming and Code Organization

Backend (Django):

- Apps are split by domain under `backend/apps/<domain>/` (e.g., `cases`, `evidence`, `suspects`).
- Python naming follows standard conventions:

- `PascalCase` for classes (`Case`, `CrimeSceneReport`) - `snake_case` for functions and variables (`can_user_access_case`, `compute_most_wanted`)

- Each app keeps the typical DRF layers co-located:

`models.py`, `serializers.py`, `views.py`, `urls.py`, and `tests/`.

Frontend (React + TypeScript):

- Pages live in `frontend/src/pages/*Page.tsx`.
- API clients are separated as `*Api.ts` next to the page modules (e.g., `casesApi.ts`, `boardApi.ts`).
- Component and type naming follows TS conventions:

- `PascalCase` for React components - `camelCase` for functions - explicit exported types (`RoleSlug`, `DashboardModule`)

## 5.3 Error Handling and API Envelope

Backend errors are normalized into a consistent response envelope via:

- `backend/police_portal/api_exceptions.py`

This keeps frontend error handling predictable across endpoints and matches the OpenAPI descriptions used throughout the API.

## 5.4 Documentation Convention (Swagger / OpenAPI)

We intentionally treated Swagger as a product requirement, not an afterthought:

- Per-endpoint narrative descriptions are included via view docstrings and `@extend_schema`.
- Domain-specific request/response examples are generated/attached through `backend/police_portal/schem`
- A regression test suite ensures documentation quality stays intact (`backend/apps/stats/tests/test_stats`

Chapter 6

# Development Flow Summary (What Changed and Why)

This is a condensed version of the development-flow notes included in the provided "Project Report" PDF, rewritten to stay consistent with the repository.

## 6.1 Fixing a Frontend Homepage Error (Public Stats)

Problem: the frontend homepage calls `GET /api/v1/stats/overview/` without authentication, but the backend default permission is authenticated (`IsAuthenticated`) via `backend/police_portal/settings.py`.

Root cause: the stats endpoint did not override the default, so anonymous users correctly received `401`, and the UI showed an error banner.

Final fix: `backend/apps/stats/views.py` sets `permission_classes = [AllowAny]` for `StatsOverviewView` and documents the endpoint as public.

Verification: `backend/apps/stats/tests/test_stats_docs.py` asserts the route returns `200` for anonymous requests and checks that Swagger examples are real (not placeholders).

## 6.2 Swagger / OpenAPI Documentation Improvements

Goal: meet the rubric requirement for "complete and reliable Swagger documentation (including appropriate request/response examples and full explanations)".

Key changes (repository evidence):

- Corrected authentication schema outputs by introducing structured response serializers (e.g., `backend/apps/accounts/serializers.py` and `backend/apps/accounts/views.py`).
- Wrapped refresh-token docs to produce higher quality OpenAPI (see `TokenRefreshDocsView` in `backend/apps/accounts/views.py`).
- Replaced generic placeholders with domain-specific examples via `backend/police_portal/schema.py`.

- Added a regression test suite so every operation has a description and JSON examples (`backend/apps/stats/tests/test_stats_docs.py`).

## 6.3 Docker and Runtime Verification

Runtime packaging:

- `docker-compose.yml` runs PostgreSQL (`db`), Django (`web`), and the frontend container (`frontend`).
- Backend image comes from `backend/Dockerfile`.
- Frontend image comes from `frontend/Dockerfile` and serves via `frontend/nginx.conf`.

## 6.4 Tests and Quality Gates

Backend: pytest-based API and schema tests live under `backend/apps/*/tests/` and are wired via `pytest.ini`.

Frontend: Vitest + Testing Library tests exist under `frontend/src/**` and are runnable via `npm test` (see `frontend/package.json` scripts).

# Chapter 7

# Key System Entities (And Why They Exist)

This section maps directly to the Django models under `backend/apps/*/models.py`.

## 7.1 Identity, Roles, and Access

**User** (`apps.accounts.models.User`) Exists because all workflows are role-gated and auditable by actor (who created, reviewed, approved).

**Role** and **UserRole** (`apps.rbac.models`) Exist to satisfy the requirement: "roles are dynamic and can be added/removed/modified without changing source code". `RoleRequiredPermission` uses role slugs to enforce access at runtime (`apps.rbac/permissions.py`).

## 7.2 Case Formation and Case Lifecycle

**Complaint** (`apps.cases.models.Complaint`) Represents complainant-initiated case formation with the required review loop (cadet review, officer review, strike count, return messages).

**Case** (`apps.cases.models.Case`) Represents the actual formed case, with a `source_type` of `complaint` or `crime_scene`, and a lifecycle status (`active`, `pending_superior`, `closed_*`, `voided`).

**CaseComplainant** Exists because a case/complaint may have multiple complainants and their identities must be approved/rejected by cadet workflow.

**CrimeSceneReport** and **CrimeSceneWitness** Exist to model officer-initiated case formation including witness identity capture (phone + national_id with validation against registered users) and superior approval.

**CaseAssignment** Exists because many workflows require case-scoped responsibility (e.g., a specific detective is assigned to a case, and only that detective can submit suspects or operate the board for that case).

## 7.3 Evidence

**Evidence** Uniform wrapper entity required by spec: title, description, registrant, registration date; linked to a case.

Typed evidence entities exist because the requirements specify different storage constraints:

- **WitnessStatementEvidence** and **EvidenceMedia**: transcription plus optional media (image/video/audio).
- **MedicalEvidence** and **MedicalEvidenceImage**: forensic/identity-db fields plus one or more images.
- **VehicleEvidence**: enforces the constraint "license plate and serial number cannot both exist" (validated in `apps.evidence.serializers.VehicleEvidenceSerializer`).
- **IdentityDocumentEvidence**: stores flexible key-value details via `JSONField`.

## 7.4 Detective Reasoning

**DetectiveBoard**, **BoardItem**, **BoardConnection** (`apps.board.models`) Exist to support the "detective board" workflow: free positioning (`x`, `y`), evidence references, notes, and red-line connections.

## 7.5 Suspects, Wanted Status, and Ranking

**Person**, **SuspectCandidate**, **WantedRecord** (`apps.suspects.models`) Separate "person identity" from "candidate within a specific case" and from "wanted status over time". Ranking and reward formula are implemented in `apps.suspects.utils.compute_most_wanted()` and matches the spec (days wanted *crime degree, reward = score* 20,000,000 Rials).

## 7.6 Interrogation and Trial

**Interrogation** (`apps.interrogations.models`) Exists to capture detective + sergeant guilt scoring (1-10) and multi-level approvals (captain, and chief for critical crimes).

**Trial** (`apps.trials.models`) Exists to store judge verdict and punishment, with the requirement that the judge can see the entire case dossier (served by reporting endpoints).

## 7.7 Tips / Rewards, Payments, Notifications

**Tip**, **TipAttachment**, **RewardCode** (`apps.rewards.models`) Exist to model the two-stage review queue (officer then detective), to issue a unique reward code, and to support reward lookup by code and national id.

**Payment** (`apps.payments.models`) Exists for optional bail/fine flows with gateway reference and verification timestamps, and to support a return page in the workflow.

**Notification** (`apps.notifications.models`) Exists because the spec requires notifying detectives when new evidence/documents are added or when decisions are made in workflows (implemented by creating notification rows during workflow transitions).

# Chapter 8

# NPM Packages Used (Up to 6)

The frontend uses multiple packages; this section highlights **six** that were intentionally chosen and are directly used in the shipped UI (`frontend/package.json`).

## 8.1 1) `react-router-dom`

Why it exists in this project: role-based navigation and multiple workflow pages require routing with nested layouts and guarded routes.

## 8.2 2) `@tanstack/react-query`

Why it exists in this project: workflow pages depend on server state (queues, case details, actions). React Query provides caching, request deduplication, loading/error states, and predictable mutation handling.

## 8.3 3) `axios`

Why it exists in this project: provides a clean HTTP client with interceptors (useful for JWT headers and consistent error handling), used across `frontend/src/pages/*Api.ts`.

## 8.4 4) `zustand`

Why it exists in this project: lightweight global state for auth/session persistence and cross-page shared state without introducing the overhead of larger state frameworks.

## 8.5 5) `react-hook-form` (plus `@hookform/resolvers` + `zod`)

Why it exists in this project: complex forms (complaint submission, evidence creation, admin role/user changes) benefit from performant form state handling and declarative validation. `zod` provides typed schemas and `@hookform/resolvers` integrates them.

## 8.6 6) `reactflow`

Why it exists in this project: the detective board requires node-like UI behavior and connections. React Flow provides interaction primitives that map cleanly onto the board workflow (drag/drop positioning and linking).

Note: additional UI and utility packages are present (e.g., `dayjs`, `clsx`, `html-to-image`, `lucide-react`), but the list above is limited to six per the report constraint.

# Chapter 9

# AI Usage (Examples and Evaluation)

This project used AI as an accelerator for boilerplate-heavy tasks, and then relied on manual review + tests to keep behavior correct and requirements-aligned.

## 9.1 Examples of AI-Generated (AI-Assisted) Code In This Repository

The following code areas were produced or heavily drafted with AI assistance and then refined to match the project rubric and the existing codebase:

Most of the AI-assisted backend work is concentrated in the documentation and verification layer (Swagger examples + schema tests), visible in commits like `01ec0d2` and `3bb278c` (see `git log`).

### 9.1.1 Example 1: Public homepage stats fix + documentation

File: `backend/apps/stats/views.py`

What it solves: the frontend homepage calls `GET /api/v1/stats/overview/` without authentication, so the backend must override the global `IsAuthenticated` default and allow anonymous access.

Implementation points:

- `permission_classes = [AllowAny]` on `StatsOverviewView`
- endpoint description explicitly states "Authentication: No JWT required."

Code excerpt:

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

### 9.1.2 Example 2: Domain-specific OpenAPI example system

File: `backend/police_portal/schema.py`

What it solves: generic placeholder examples (e.g., `{"example": "value"}`) do not satisfy the rubric requirement for complete Swagger docs with meaningful request/response examples.

Implementation points:

- A curated map keyed by serializer import path (`SERIALIZER_REQUEST_EXAMPLES`) provides realistic payloads for core workflows (complaints, evidence creation, suspect proposal, tips, payments, etc.).
- Schema generation is centralized through `REST_FRAMEWORK["DEFAULT_SCHEMA_CLASS"] = "police_portal.schema.PoliceAutoSchema"` in `backend/police_portal/settings.py`.

Code excerpt:

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

### 9.1.3 Example 3: Schema regression tests to prevent documentation drift

File: `backend/apps/stats/tests/test_stats_docs.py`

What it enforces:

- Every operation has a description.
- JSON request bodies include request examples.
- JSON success responses include non-generic examples.

This makes Swagger quality testable, which is valuable when AI is involved because it prevents "looks correct" documentation from silently regressing.

Code excerpt:

```
self.assertTrue(operation.get("description"))
self.assertTrue(request_content["application/json"].get("examples"))
```

## 9.2 Strengths and Weaknesses of AI in Front-End Development (Observed)

Strengths: AI is effective at accelerating repetitive UI scaffolding (pages/forms), generating initial API client wrappers, and suggesting test cases for page-level behavior. It also helps produce consistent error-state UI and loading-state patterns when the data model is stable.

Weaknesses: AI often guesses UX and workflow rules incorrectly when the requirements are nuanced (role-scoped queues, multi-stage approvals, visibility rules). It also tends to over-generate components or abstractions that look "clean" but do not match the project's existing patterns, which increases integration time.

## 9.3 Strengths and Weaknesses of AI in Back-End Development (Observed)

Strengths: AI is useful for quickly drafting DRF serializers/views, writing repetitive schema descriptions/examples, and producing test scaffolding. It can also help spot access-control gaps by reading workflows and suggesting enforcement points.

Weaknesses: AI can miss subtle authorization constraints (case-scoped assignment checks, role priority rules, and state-machine transitions). It may also produce Swagger docs that "compile" but do not reflect real payload shapes unless the team adds verification tests (which we did).

# Chapter 10

# Requirement Analysis (Initial vs Final)

This section summarizes requirement interpretation decisions and how they evolved based on implementation feedback, with an emphasis on consistency with the shipped code.

## 10.1 Initial Requirement Analysis (What We Assumed Early)

1. **Global authentication is the default** We configured DRF with `IsAuthenticated` as the default permission class (`backend/police_portal/settings.py`), expecting only a small set of endpoints to be public. 2. **Role system must be dynamic** We designed `Role` and `UserRole` models early to allow adding/removing/modifying roles without code changes (`backend/apps/rbac/models.py`). 3. **Workflows are stateful** We modeled complaint/case/evidence status as explicit enums (e.g., `ComplaintStatus`, `CaseStatus`) so that multi-stage approvals can be enforced and tested (`backend/apps/cases/co`

Strengths of these early decisions: they align with the rubric emphasis on access control, maintainable entity models, and workflow correctness.

Weaknesses (found later): some endpoints that are *functionally public* (homepage stats, most-wanted public view) must explicitly override the global auth default or the frontend will behave incorrectly.

## 10.2 Final Requirement Analysis (What The Code Now Enforces)

### 10.2.1 1) Public vs Private Endpoints

We kept a secure default and explicitly marked public endpoints with `AllowAny`, for example:

- Public homepage stats: `backend/apps/stats/views.py` (`StatsOverviewView.permission_classes = [AllowAny]`)
- Public most-wanted ranking: `backend/apps/suspects/views.py` (`MostWantedPublicView.permission_cl = [AllowAny]`)

This balances security with the project’s "public page" requirements.

### 10.2.2 2) Case-Scoped Access and Ownership

During integration we tightened access control so that role membership alone is not enough for sensitive actions.

Examples in code:

- `backend/apps/cases/policies.py` implements `can_user_access_case()` and assignment checks (`CaseAssignment`).
- `backend/apps/suspects/views.py` verifies the proposing detective is assigned to the case before allowing suspect proposals.

Strength: prevents cross-case data leaks and matches the real workflow ("responsible detective").

Tradeoff: adds complexity to API code and requires careful testing, which we covered with pytest flow tests.

### 10.2.3 3) Evidence Type Rules and Validations

We encoded evidence-type-specific requirements in serializers:

- `backend/apps/evidence/serializers.py` enforces the vehicle constraint (license plate XOR serial number).
- `EvidenceCreateSerializer` enforces presence of nested evidence payload by type.

Strength: pushes workflow correctness into validation (fail fast, predictable errors).

### 10.2.4 4) Most-Wanted Ranking and Reward Formula

We implemented the formula exactly as specified:

- `backend/apps/suspects/utils.py` computes:

- `ranking_score = days_wanted * crime_degree` - `reward_amount = ranking_score * 20000000`
- only includes suspects wanted for at least 30 days.

Strength: implementation is deterministic and testable; frontend can display ranking/reward confidently.

### 10.2.5 5) Swagger Documentation Quality As A Requirement

We treated Swagger examples and descriptions as a graded deliverable:

- Centralized schema customization in `backend/police_portal/schema.py`.
- Added regression tests in `backend/apps/stats/tests/test_stats_docs.py` to ensure examples and descriptions remain non-generic and complete.

Strength: reduces the risk of losing points due to documentation drift, especially when AI assistance is used to generate repetitive docs.