JusticeWatch: Texas Police Accountability Platform - Technical Report

Purpose and Motivation:
JusticeWatch is a civic engagement platform designed to visualize and track police brutality, related legislation, and department accountability across Texas. The project aims to increase transparency and public awareness of policing issues, ultimately contributing to improved accountability and policy reform. The platform serves as a comprehensive resource for citizens, activists, policymakers, and researchers to access and analyze data on police misconduct, legislative efforts, and departmental performance.

Architecture:
JusticeWatch employs a modern web architecture with a PostgreSQL database, Python backend with REST Framework for API development, and a React.js frontend. The system will use Google Maps for interactive mapping.

Phase 1 and 2 Backend API:

JusticeWatch employs a robust backend system leveraging Python with Flask and SQLAlchemy to manage data operations and API endpoints. The backend interfaces directly with a PostgreSQL database hosted remotely, defined by SQLAlchemy models representing the core data entities: Police Incidents, Legislation, and Department Scorecards. Each model corresponds to a dedicated database table, facilitating structured data storage and retrieval. The provided Python scripts handle database population through custom functions (populate_db(), populate_scorecard(), and populate_legi()), which ingest data extracted from external scraping modules (scrape.py). These functions iterate over dataframes returned by scraping utilities (extractInfo, extractScorecard, extractLegislation), instantiate model objects, and commit them to the database within transaction-safe sessions. Error handling mechanisms are implemented to ensure database integrity, rolling back transactions upon encountering exceptions.

The backend API is built using Flask, with endpoints designed for querying the database dynamically based on user-supplied parameters. The API supports pagination, filtering, and detailed retrieval of individual records by ID. It leverages SQLAlchemy's ORM capabilities to execute parameterized SQL queries securely and efficiently.
To enhance scalability and deployment consistency, the backend system is containerized using Docker. Docker encapsulates the Flask application along with its dependencies into a portable container image, enabling straightforward deployment across environments and ensuring consistency between development and production setups. This containerization simplifies deployment processes, enhances scalability, and streamlines continuous integration and deployment workflows facilitated by GitLab CI/CD pipelines

Phase 3 Backend API:
The provided API implementation supports pagination, filtering, searching, and sorting for three main entities: Legislation, Incident, and Agency. Pagination is handled through two main functions, get_pagination_metadata and paginate_request. These functions calculate the total count of records, determine the total number of pages, and apply SQL LIMIT and OFFSET clauses to fetch a specific subset of records. The pagination parameters, such as page (0-indexed) and per_page (default 10), are passed as query parameters, ensuring efficient data retrieval by limiting the result set.

Filtering is implemented by dynamically appending conditions to SQLAlchemy queries based on query parameters. For instance, in the /legislation endpoint, filters can be applied on fields like id, bill_number, state, title, and description. Similarly, the /incidents and /agencies endpoints allow filtering based on relevant fields such as state, city, or agency-specific identifiers. This flexibility enables users to retrieve data tailored to their needs.

The API also supports full-text searching using a search query parameter. This is achieved by iterating through all text-compatible columns (e.g., columns with type String) of the respective entity's table. A case-insensitive search (ILIKE) is performed to match the search term against any column, combining multiple conditions with an OR operator. This approach allows users to perform broad searches across multiple fields.

Sorting is implemented using the sort_by and sort_order query parameters. Users can specify a column to sort by and choose between ascending (asc) or descending (desc) order. The API validates whether the specified column exists in the model before applying sorting to avoid errors. Together, these features make the API robust and user-friendly, providing efficient data retrieval with customizable options for pagination, filtering, searching, and sorting.

Hosting:

The website is hosted on an AWS EC2 instance, with SSH access provided to team members via a shared SSH key. The server's IP address is currently configured to accept connections from any IP and port, which poses a security risk that should be addressed in the future. The website's static HTML files were generated using the npm run build command, and are served using PM2, a Node.js process manager, listening on port 3000. An Nginx reverse proxy is implemented to handle HTTPS traffic, utilizing Certbot for SSL certificate management. This proxy forwards all requests received on port 443 (HTTPS) to the PM2 server running on

localhost:3000. DNS configuration for justicewatch.me is managed through Cloudflare, with an A record pointing to the EC2 instance's IP address.

Toolchain:
Development utilizes Git for version control, with Gitlab CI/CD. The application will be containerized using Docker and deployed on AWS for scalability. We will be using Tailwind for implementation of style.

User Stories:

- **Phase 1:** Customers wanted to have more navigability and searchability to the different models, especially with Police Department scorecards and police violence incidents. We have decided to implement a map feature for both the models of Incidents and Department Scorecard. The map will allow users to search around for specific police departments or incidents where they want to see.
- **Phase 2:** Customers wanted a way to sort police departments scores and sort legislation data to have a better. We will implement this next phase as it is a requirement for phase 3. We have also begun adding extra information on tips when being approached by police. We also have a list of supporters and sponsors of legislation.
- **Phase 3;** Users wanted a better search, filtering out instances for all 3 models which we were able to accomplish through my new revamped api to handle searching, Users also wanted better sorting so they can see report cards based on how good the performance was.

API Documentation:
https://documenter.getpostman.com/view/42447157/2sAYdZtYvV

Models and Instances:

1. Police Misconduct Incidents (~13,000 instances): Captures details of incidents including date, location, victim information, and outcome.

2. Legislation Tracker (~8,000 instances): Tracks police reform bills, including status, sponsors, and topics covered.

3. Police Department Accountability Scorecards (1,000+ instances): Stores department performance metrics, funding information, and demographic data.

Challenges and Solutions:

1. AWS Hosting of the new API was challenging in that we couldn’t get the API up and running in parallel with the frontend. Database was returning empty when we had the API running but thankfully we were able to fix these issues.
2. Data Integration: Combining data from disparate sources with varying formats posed a significant challenge. We developed key apis and databases to store all the information of legislation, violence and department information

3. Geospatial Visualization: Rendering thousands of incidents on a map caused performance issues. We will implement clustering and lazy loading techniques to enhance the map's responsiveness and user experience.