Cover Page

Document Title: technical_documentation.pdf

Document Purpose: Provides a technical breakdown of our project. Highlights and shows the system architecture, data model and database design, back and front end logics, testing applications, how to deploy the website, security concerns and what bugs are present in current submission.

Project Title: Crowdsourced Disaster Relief Platform

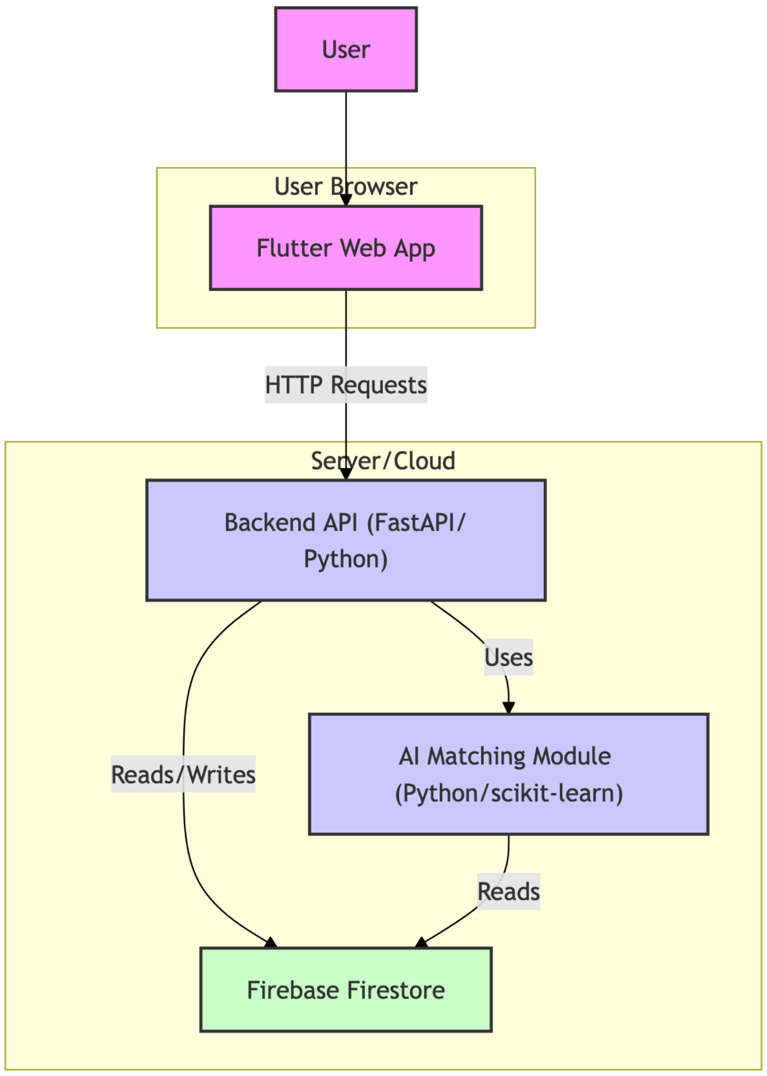
Group Number: 2

Group Members: Casey Nguyen, Kevin Pulikkottil, Andy Jih, Sawyer Anderson

Table of Contents

Overview of the System Architecture:

High Level Diagram:



Technologies Used:

- Backend: Python, FastAPI, Firebase Firestore, scikit-learn, NumPy, Geopy, Uvicorn, Joblib, Requests. Docker support is optionally available.
- Frontend: Flutter, Dart.
- Database: Firebase Firestore.
- Development Environment: VSCode.
- Testing: Pytest.

Data Models & Database Design:

Explanation of Key Models:

We use Firebase Firestore as our cloud-hosted NoSQL database. Data is stored in collections, each containing documents that represent individual records. Each document has fields and values of varying data types.

- Users: Stores information about the user with their login info, currently has the email and password in the model but with integration in the next deliverable, a UID will be assigned for primary identification.
- Requests: Stores information about the person making a request like the name, type of assistance, description of the situation, the longitude and the latitude of the exact request.
- Donations: Stores information about the person making the donation, what type of donation (money or a resource) and then a description of the donation type ($ or what kind of resource)
- Alerts: Stores information about what kind of alert is given out on the website, contains the type/name of the emergency, a description of what is going on and the threat level and then the date the alert was sent out on the website.
- Resources: Stores information about what kind of resource is in the database. Has the name of the resource, the quantity of the resource and then the area where the resource is at.

Backend Logic:

How requests are handled and processed:

- The backend is a RESTful API built with FastAPI.
- It integrates with Firebase Firestore for data persistence.
- When a request for volunteer matching is received (via the `/match/{request_id}` or `/debug-match/{request_id}` endpoints), the system fetches the specific aid

request and all available volunteer data from Firestore [cite: Project Code/1_code/backend/main.py].

- The AI matching module (`matching_ai.py`) extracts features from the request (type, location, urgency) and volunteers (skills, location, availability) [cite: 22, Project Code/1_code/backend/matching_ai.py].
- Addresses are converted to latitude/longitude using Geopy [cite: 33, Project Code/1_code/backend/matching_ai.py].
- Categorical features like request type and volunteer skills are processed using one-hot encoding [cite: 32, Project Code/1_code/backend/matching_ai.py].
- A K-Nearest Neighbors (KNN) algorithm (from scikit-learn) is used to find the volunteers whose features are closest to the request's features, based on Euclidean distance after scaling the features [cite: 22, 33, Project Code/1_code/backend/matching_ai.py].
- The system returns the top `k` (default 3) matched volunteers [cite: Project Code/1_code/backend/matching_ai.py].

API endpoints & routes:

- `GET /`: Welcome message for the API [cite: Project Code/1_code/backend/main.py].
- `GET /match/{request_id}`: Production endpoint. Fetches the request by its ID, finds all volunteers, performs KNN matching, and returns a list of the top 3 matched volunteers [cite: 34, Project Code/1_code/backend/main.py]. Returns 404 if the request ID is not found or no volunteers are available [cite: Project Code/1_code/backend/main.py].
- `GET /debug-match/{request_id}`: Debug endpoint. Similar to `/match`, but returns detailed information about the matching process, including raw and scaled feature vectors, distances, and indices [cite: 16, 34, Project Code/1_code/backend/main.py, Project Code/1_code/backend/matching_ai.py].

Returns 404 if the request ID is not found or no volunteers are available [cite: Project Code/1_code/backend/main.py].

- *(Note: `Project Code/1_code/backend/app.py` contains Flask routes for `/donations` (GET and POST), but this seems to be an incomplete example and is separate from the main FastAPI application in `Project Code/1_code/backend/main.py`).*

Authentication/Authorization logic:

- The current backend code (`main.py`, `matching_ai.py`) does not show implemented authentication or authorization logic for accessing the API endpoints [cite: Project Code/1_code/backend/main.py, Project Code/1_code/backend/matching_ai.py].
- The frontend `auth_service.dart` includes functions for sign-up and sign-in by POSTing to placeholder `/signup` and `/signin` endpoints, but these are not implemented in the provided backend code and are noted as a future task [cite: 37, Project Code/1_code/lib/services/auth_service.dart, 161].

Frontend Logic:

Page/component breakdown:

- home_screen.dart: Default screen the user lands on. Has the website name and the following 5 functionalities under the name as buttons for redirections. The 5 functions are to see the resource inventory page, make a request page for assistance, donations page where users can donate, a page for emergency alerts (past and present) and then a Sign Up / Sign In page if the users wanted to make an account with their email and a password.
- donation_screen.dart : This page allows the user to input their name or type in anonymous as the first required line. The next line identifies what kind of donation

they would like to make, either money or a resource. Then the description on what they are donating as the third line. At the bottom under the donate button, is a list of recent donations. After the user submits, their name will be placed on the list in real time and then a thank you message will appear.

- emergency_alerts_screen.dart: The screen simply shows the past and current set of emergency alerts pushed out by the website. It displays each alert with the type of disaster, the description of it and then the date the alert was posted on.
- profile_screen.dart: The screen toggles between sign up and sign in depending on what the user clicked. It allows the user to sign in or up if they wish.
- request_posting_screen.dart: This screen allows the user to make requests for help. It takes in their name, type of assistance and the description. Their location is also pin pointed to ensure they get the location of help exactly. At the bottom of the submit button, there is a list of current requests for help. After submitting, the user's request will be updated onto the list.
- resource_inventory_screen.dart: This screen shows the available list of inventory in a specific area. It shows the name of the resource and how much of the resource is available.

User interface flow:

- The flow is simple and easy to use. The home screen with the name of the website and the 5 buttons/ tabs is the default landing page for the user. From there they can click on each button, go back to the homescreen if needed. After each submission they can make another one and navigate back and forth like they should as it is a website. The home screen allows the user to visualize the entire website and see what kind of info or help they need easily and fast.

Testing:

How to test:

- Ensure the FastAPI backend server is running and accessible (e.g., `make run` or `uvicorn main:app --reload`) [cite: Project Code/3_basic_function_testing/README3.txt].
- Ensure the Firestore database is populated with sample data (e.g., `make populate-db`) [cite: Project Code/3_basic_function_testing/README3.txt]. The population script (`populate_database.py`) connects to Firebase, clears existing data in 'volunteers' and 'requests' collections, and adds sample data [cite: Project Code/2_data_collection/README2.txt, 7, 8].
- Run the automated tests using Pytest (e.g., `make test`) [cite: Project Code/3_basic_function_testing/README3.txt].

What is being tested:

- The tests primarily focus on the AI volunteer matching endpoints (`/match/{request_id}` and `/debug-match/{request_id}`).
- Tests (`test_matching.py`) validate that:
  - Valid request IDs return a 200 OK status and the expected JSON structure (a list under `matched_volunteers`) [cite: Project Code/3_basic_function_testing/README3.txt, Project Code/3_basic_function_testing/test_matching.py].
  - The `matched_volunteers` list contains valid volunteer dictionaries with required keys (id, name, skills, location) [cite: Project Code/3_basic_function_testing/test_matching.py].
  - Invalid or non-existent request IDs (e.g., 999) return an appropriate error status code (like 404 or 500) [cite: 15, Project Code/3_basic_function_testing/test_matching.py].

   - The debug endpoint (`/debug-match`) returns the expected detailed keys (request_features, volunteer_features, X_scaled, req_scaled, distances, indices, matched_volunteers) [cite: 16, 17, 18, Project Code/3_basic_function_testing/test_matching.py].
   - There is consistency between the `matched_volunteers` returned by the production (`/match`) and debug (`/debug-match`) endpoints for the same valid request ID [cite: Project Code/3_basic_function_testing/test_matching.py].

Deployment Details:

Hosting Information:

- The primary backend database is Firebase Firestore, a cloud-hosted NoSQL database.
- The backend API is built with FastAPI and can be run using Uvicorn.
- Docker support is provided via `Dockerfile` and `docker-compose.yml` for containerized deployment [cite: 35, Project Code/1_code/backend/docker-compose.yml]. The Docker setup builds the FastAPI service and configures it to connect to Firebase using credentials mounted via a volume [cite: Project Code/1_code/backend/docker-compose.yml].
- The frontend is a Flutter web application, which can be built and run on a web server (like Chrome during development) [cite: Project Code/1_code/README1.txt].

Security Considerations:

Data protection:

- Firebase credentials (the `serviceAccountKey.json` file) are required to interact with the Firestore database [cite: Project Code/2_data_collection/README2.txt, Project Code/1_code/README1.txt].
- This key file should be kept secure and is excluded from Git via `.gitignore`. It needs to be placed in the correct directory (`1_code/` or `2_data_collection/`) or its path specified via the `GOOGLE_APPLICATION_CREDENTIALS` environment variable [cite: 40, Project Code/1_code/backend/main.py, Project Code/2_data_collection/populate_database.py].
- When using Docker, the service account key is mounted as a read-only volume into the container [cite: Project Code/1_code/backend/docker-compose.yml].

User validation:

- User authentication (sign-up/sign-in) is planned but not yet integrated between the frontend and backend. The frontend has UI and service placeholders [cite: Project Code/1_code/lib/screens/profile_screen.dart, Project Code/1_code/lib/services/auth_service.dart], but the backend API endpoints are not implemented in `main.py` [cite: Project Code/1_code/backend/main.py].

Bugs/Limitations:

What Doesn't Work:

- Currently the frontend and backend is not fully integrated, we worked on both of these ends separately. Just so we can have it done for the deliverable 1, we focused on the key functions like AI matchmaking and how the system will run. The backend is nearly complete. Now for deliverable 2 we will integrate both ends so the users data can be stored and uploaded into the database. This will allow for real time tracking and updates to the website.

- Since the backend is done but not integrated, the sign up / sign in page when you hit sign up or sign in creates an error on the website. This error is simply because when the frontend calls for the backend, there is nothing there. This is a small and easy fix for the next deliverable. This is also due to the professor saying not to worry about user authentication in this deliverable.
- The platform/website design is not done. It is just a stage 1 prototype as we focused on the main key functions instead of how everything looks. Our real design will be ready next deliverable as we are almost done with the key functions, just need to integrate both ends. So everything is just bland right now and basic/simple.