Prompts Log for Emergency Management Web Application

This document serves as a comprehensive log of my interactions with Grok 3 (xAI) during the development of an emergency management web application. The project is a Golang-based REST API with advanced GIS capabilities, designed to empower emergency coordinators through flood risk assessment, response planning, and geospatial visualization. It integrates public APIs such as Open-Meteo (for precipitation data), Overpass API (for water body proximity), OSRM (for evacuation routes), and Nominatim (for geocoding), delivering practical solutions for flood-prone regions. The prompts below detail how I leveraged AI to define functional requirements, integrate APIs, debug issues, and optimize the application. By focusing on GIS-driven features and robust backend design, this project reflects my readiness to contribute to Trimble's mission of advancing geospatial and engineering solutions.

Prompt 1:

Define the functional requirements and key features for an emergency management web application focused on flood risk assessment and response planning, including necessary API integrations, user roles, and expected outputs in JSON format for a Golang-based REST API.

Summary of Response:
Grok outlined functional requirements, including flood risk assessment (using APIs like OpenWeatherMap, Overpass API, Open Topo Data, ReliefWeb API), response plan management (with Nominatim, OSRM), user role management (field personnel, officials), geospatial visualization (GeoJSON outputs), and real-time flood event context. It provided a risk score formula (0.4*Precipitation + 0.3*(1-ElevationFactor) + 0.2*ProximityToWater + 0.1*FloodEventFactor) and API endpoints with JSON examples (e.g., GET /api/flood-risk, POST /api/response-plan). It suggested a modular architecture using Gin, with /services, /gis, and /api directories.

Application in Project Development:
I used this response to define the application's core features, implementing the /api/flood-risk endpoint with the provided risk score formula in /gis/risk.go, and integrating APIs in /services. I created POST /api/response-plan and GET /api/response-plans endpoints, using Nominatim and OSRM for geocoding and routing, and added role-based access for users. The modular architecture (/services, /gis, /api) improved modularity and maintainability.

Prompt 2:

Provide a detailed workflow for a Golang-based REST API integrating GIS data with public APIs (e.g., OpenWeatherMap, Overpass API) for flood risk assessment, including how handlers and controllers process inputs/outputs and the calculations involved in risk scoring.

Reason for Asking:
I needed a clear understanding of how to structure the backend to handle API integrations, perform GIS-based calculations, and process requests efficiently. This was critical to ensure the application met the practical utility and modularity criteria by organizing code into services, handlers, and GIS logic.

Summary of Response:
Grok provided a comprehensive workflow for each endpoint (e.g., GET /api/flood-risk). It detailed how handlers validate inputs (e.g., coordinates), call services to fetch data from APIs (Open-Meteo for precipitation, Overpass for water bodies), and perform calculations.

Prompt 3: Generate cURL commands for testing all REST API endpoints of my emergency management application in Postman, including both local endpoints (e.g., /api/flood-risk) and public APIs (e.g., OpenWeatherMap, Overpass API), with expected JSON responses and usage notes.

Reason for Asking:
I needed a systematic way to test my API endpoints and public API integrations to ensure functionality and debug issues before submission. This was crucial for validating practical utility and ensuring robust error handling in API responses.

Summary of Response:
Grok provided cURL commands for all local endpoints (e.g., curl -X GET "http://localhost:8080/api/flood-risk?lat=37.8&lon=-122.4") and public APIs (e.g., OpenWeatherMap: curl -X GET "https://api.openweathermap.org/data/2.5/forecast?lat=37.8&lon=-122.4&appid=YOUR_API_KEY&units=metric"). It included expected JSON responses (e.g., risk score with factors) and notes on API limits (e.g., OpenWeatherMap's 60 calls/minute).

Application in Project:
I used these cURL commands in Postman to test my application's endpoints, confirming that /api/flood-risk returned accurate risk scores and /api/response-plan generated correct GeoJSON routes. I also tested public APIs to verify data availability. These commands were added to /docs/README.md under "API Usage," enhancing documentation quality (30 marks). The testing process helped me identify and fix a validation bug in the /api/flood-zones endpoint, improving error handling.

Prompt 4:

OpenWeatherMap declined my card for free tier access, preventing precipitation data retrieval. Suggest alternative free weather APIs that provide hourly precipitation data globally, including cURL commands and integration guidance for a Golang backend

Reason for Asking:

I couldn't proceed with OpenWeatherMap due to payment issues, and I needed a free alternative to fetch precipitation data, a critical component for flood risk assessment, to ensure the project's functionality.

Summary of Response:
Grok recommended three free APIs: Open-Meteo, National Weather Service (NWS) API, and Weatherstack. It highlighted Open-Meteo for its no-key requirement and provided a cURL command: curl -X GET "https://api.open-meteo.com/v1/forecast?latitude=37.8&longitude=-122.4&hourly=precipitation". It also gave Go code for integrating Open-Meteo into my backend.

Application in Project:
I adopted Open-Meteo as my primary weather API, implementing the provided Go code in /services/weather/open_meteo.go. This allowed my /api/flood-risk endpoint to fetch precipitation data reliably. The switch to Open-Meteo ensured my project remained functional despite the OpenWeatherMap issue, demonstrating adaptability. I documented this change in /docs/README.md, contributing to the documentation score.

Prompt 5:

Provide an example Overpass API query to fetch water bodies within a 1km radius of a given location, along with a cURL command and Go code to calculate proximity to water bodies for flood risk assessment in a GIS-enabled application.

Reason for Asking:
I needed to integrate OpenStreetMap data via the Overpass API to identify water bodies near a location, a key factor in flood risk scoring, to enhance the GIS capabilities of my application, aligning with Trimble's geospatial focus and practical utility.

Summary of Response:
Grok provided an Overpass QL query: [out:json][timeout:25];way["waterway"](around:1000,37.8,-122.4);out geom;, with a cURL command: curl -X POST "https://overpass-api.de/api/interpreter" -d 'data=[out:json][timeout:25];way["waterway"](around:1000,37.8,-122.4);out geom;'. It also included Go code to parse the response and calculate proximity using the orb library.

Application in Project:
I implemented the query in /services/geospatial/overpass.go to fetch water bodies for the /api/flood-risk endpoint. The proximity calculation code was used in /gis/risk.go to compute the ProximityToWater factor, improving the accuracy of my risk scores. This GIS integration added significant value to the application, aligning with focus on geospatial solutions. I also cached Overpass responses to optimize performance.

Prompt 6:

Guide me through the steps to run my Golang-based emergency management application locally, considering it uses a PostgreSQL database and includes a start.sh script, with instructions to set up dependencies and test endpoints.

Reason for Asking:
I needed clear instructions to run the project locally after writing the complete codebase, ensuring all dependencies and database configurations were set up correctly to test the application.

Summary of Response:
Grok provided a step-by-step guide to set environment variables (if needed), and run the project using start.sh or go run main.go. It also suggested testing endpoints with cURL commands (e.g., curl -X GET "http://localhost:8080/api/flood-risk?lat=37.8&lon=-122.4").

Follow-up

I encountered an error when running my Golang application: 'failed to connect to user=postgres database=flood_risk_management: 127.0.0.1:5432: connection refused.' Provide steps to resolve this PostgreSQL connection issue, including database setup and schema initialization.

Summary of Response:
Grok diagnosed the issue as a PostgreSQL server not running or database not created. It provided steps to install PostgreSQL, start the server, create the flood_risk_management database, initialize the

schema, and update the connection string with the correct password. It also suggested a fallback to in-memory storage using sync.Map if PostgreSQL setup failed.

***