CrisisEcho: Project Checkpoint Report LLM-Augmented Early Detection of Local Emergencies from Social Media

Samuel Enam Zih
Student ID: 906719049
Department of Computer Science, Virginia Tech
samuelez@vt.edu

***Abstract*—This report documents the complete implementation of CrisisEcho, a real-time crisis detection and emergency response platform. CrisisEcho ingests eight social media and official data sources via Apache Kafka, processes them through an eight-step ML preprocessing pipeline (text cleaning, GPS geocoding, MinHash deduplication, DistilBERT relevance filtering, Vertex AI text embeddings, SigLIP image embeddings, S3 image upload, and MongoDB persistence), and runs a three-step LangChain LLM agent that clusters posts into distinct crisis events, scores severity, verifies through multi-source corroboration, and generates public alerts. The system is built as a two-service architecture: a Go Fiber REST API with 20 domain modules and WebSocket support, and a Python AI sidecar with Celery task queues and gRPC. Data is stored across three MongoDB Atlas databases, Aiven Valkey (Redis-compatible) for real-time pub/sub, and AWS S3 for media. A peer-to-peer SOS emergency system with proximity-based wave broadcasting, Apple VoIP push notifications via CallKit, real-time WebSocket location tracking, and AES-256-GCM encrypted chat provides active emergency response capabilities. The Flutter mobile application comprises 24+ screens including an interactive crisis map, SOS interface, community reports, analytics dashboard, and Stripe billing. All components are fully implemented, containerized via Docker Compose, and pending deployment to Google Cloud Run.**

***Index Terms*—crisis detection, RAG, LLM agents, MongoDB, Go, Python, Flutter, SOS, WebSocket, real-time systems**

I. Introduction

Official emergency channels—911 dispatch, government sensors, police scanners—frequently lag fast-moving crises by minutes or hours. Social media posts constitute the largest real-time sensor network on Earth: ordinary people post hyperlocal signals well before any official report. CrisisEcho closes this gap by applying Retrieval-Augmented Generation (RAG) and multi-step LLM agents to detect, classify, verify, and summarize crisis events from social media and official data sources in near real-time.

Motivation. In the critical first minutes of an emergency, the gap between on-the-ground reality and official response can be fatal. Existing crisis informatics tools are either keyword-based (failing on indirect language and sarcasm), researcher-facing (not accessible to ordinary users), or limited to single event types. CrisisEcho addresses all three limitations with semantic RAG-powered reasoning, a consumer-facing mobile application, and coverage of 51 crisis categories.

Problem Statement. Build a system that (1) continuously ingests social and official data sources, (2) identifies genuine crisis events using semantic understanding, (3) verifies events through multi-source corroboration to eliminate false positives, (4) delivers verified, geolocated alerts to mobile users in near real-time, and (5) provides a peer-to-peer SOS mechanism for users in immediate danger.

Significance. CrisisEcho demonstrates four paradigm shifts over traditional crisis detection: text to semantics (vector embeddings replace keyword filters), retrieval to reasoning (hybrid retrieval feeds chain-of-thought LLM agents), vertical to multi-domain (51 categories from wildfires to epidemics), and closed-world to open-world generalization (the LLM recognizes novel crisis types without retraining). The SOS system extends the platform from passive monitoring to active emergency response.

II. Related Work

Imran et al. [1] and Olteanu et al. [2] established Twitter as a crisis sensor and built foundational lexicons, while Alam et al. [3] created HumAID, a benchmark of 77k annotated crisis tweets spanning multiple event types. Middleton et al. [4] showed geotagged posts correlate with ground-truth disaster events, motivating our geospatial retrieval layer. Lewis et al. [5] introduced RAG to ground generative models in retrieved evidence, reducing hallucination—the core technique behind our hybrid retrieval pipeline. Yao et al. [6] demonstrated chained LLM reasoning steps outperform single-prompt approaches, informing our three-step agent design. Karpukhin et al. [7] showed dense vector retrieval outperforms BM25, motivating our Atlas Vector Search hybrid retrieval. Sakaki et al. [8] demonstrated real-time earthquake detection from Twitter, validating social media as a first-responder data source. De Albuquerque et al. [9] analyzed geographic information from social media for flood assessment, supporting our GPS-only location integrity rule.

III. System Architecture Overview

CrisisEcho is implemented as a two-service architecture: a **Go Fiber HTTP API** (port 8080) serving the Flutter mobile frontend, and a **Python AI sidecar** (port 8081 HTTP, port 8082 gRPC) handling all machine learning workloads. The

services communicate via HTTP and gRPC, and share state through three MongoDB databases and Redis. Fig. 1 shows the architecture as originally proposed, and Fig. 2 shows the final implemented architecture.

Architectural Progression. Several significant pivots were made during development:

- **Frontend**: Next.js + Leaflet.js → Flutter + Google Maps (native mobile performance, single codebase for iOS/Android)
- **LLM**: Claude Haiku → Gemini 2.0 Flash (free tier for continuous 60-second pipeline runs during development)
- **Embeddings**: Voyage AI 1024-dim → Vertex AI multimodal 1408-dim (text) + SigLIP 512-dim (image)
- **Task Processing**: Direct execution → Celery with two queues (ingestion + agent) via Redis broker
- **Communication**: HTTP only → HTTP + gRPC (for pipeline triggering and queries)
- **New Subsystem**: SOS emergency system with VoIP push, WebSocket tracking, and encrypted chat—not in the original proposal

Data Flow. The canonical data flow is: Raw data → Kafka (two topics) → Preprocessor (8 steps) → SourcePost (per-source collections + vector embeddings) → Retrieval (hybrid: vector + geo + official) → LLM Agent (cluster → severity → verify → alert) → Crisis (map dot) → Alert (Redis pub/sub → FCM/APNs push → mobile).

Docker Compose. The system runs as four containers: (1) `crisisecho-api` (Go, port 8080), (2) `crisisecho-sidecar` (Python FastAPI + gRPC + Kafka consumer, ports 8081/8082), (3) `crisisecho-worker-ingestion` (Celery, concurrency=4), and (4) `crisisecho-worker-agent` (Celery, concurrency=2). The Go API depends on the sidecar's health check passing before starting.

IV. Ingestion Layer

Eight Python workers poll or stream their respective data source APIs and produce messages to two Apache Kafka topics on Aiven's managed Kafka service.

Social Sources (topic: `social_raw`):

- **RedditWorker**: PRAW library streaming from crisis-related subreddits
- **TwitterWorker**: twscrape polling with crisis-specific search queries
- **BlueskyWorker**: AT Protocol firehose with keyword filtering
- **RSSWorker**: Configurable RSS/Atom feed polling via feedparser

Official Sources (topic: `official_alerts`):

- **USGSWorker**: USGS earthquake feed (magnitude ≥ 2.5)
- **GDACSWorker**: Global Disaster Alert and Coordination System (UN-backed GeoRSS)
- **ReliefWebWorker**: UN OCHA humanitarian crisis API
- **NASAFirmsWorker**: NASA FIRMS satellite wildfire detection

Each worker inherits a `KafkaWorker` base class providing: a `stream()` generator interface, envelope wrapping with topic/source/timestamp metadata, user privacy hashing (SHA-256 before Kafka serialization), and retry logic with exponential backoff (1s → 60s max, 5 attempts). Workers are registered in a `WORKER_REGISTRY` and can be toggled via the `DISABLED_SOURCES` environment variable.

Source authority weights used in retrieval ranking: USGS, GDACS, ReliefWeb, NASA FIRMS = 1.0; Reddit = 0.7; Twitter, Bluesky = 0.6; RSS = 0.5.

A **SeedWorker** was also built to generate realistic test data: 53 crisis scenario templates across 30+ countries produce 636 synthetic posts (12 per scenario) using Groq LLaMA 3.1 8B, with 90-minute refresh cycles and direct preprocessor injection—enabling full pipeline testing without external API dependencies.

V. Preprocessing Pipeline

The preprocessing pipeline consumes Kafka messages via the orchestrator and dispatches Celery tasks to the `ingestion` queue (concurrency=4). Each post passes through eight sequential stages:

Step 1—Clean Text. Remove URLs, Unicode emojis, and normalize whitespace. Optional spaCy tokenization if the `en_core_web_sm` model is loaded.

Step 2—Geocode. If the payload contains `lat`/`lng` GPS coordinates, set `location_source="gps"` and `location_confidence=1.0`. Otherwise, mark as `"unresolved"` with confidence 0.0. No geocoding guessing is performed—location integrity is paramount.

Step 3—Deduplication. MinHash Locality-Sensitive Hashing with 128 permutations and a Jaccard similarity threshold of 0.85 over a 5-minute sliding window. Implemented via Redis-based shared state so all Celery workers see the same dedup window. Prevents viral reposts from inflating cluster contributor counts.

Step 4—Relevance Filter. A DistilBERT cross-encoder (`cross-encoder/nli-distilroberta-base`) classifies each post as crisis-related (threshold 0.6). Official sources (USGS, GDACS, ReliefWeb, NASA FIRMS) bypass this gate entirely. Drops approximately 70% of non-crisis social media content before it reaches the vector database.

Step 5—Text Embedding. Google Vertex AI multimodal embedding (`multimodalembedding@001`) generates a 1408-dimensional text vector. Used for semantic similarity search in the retrieval layer.

Step 6—Image Embedding. SigLIP (`google/siglip-base-patch16-224`) generates a 512-dimensional normalized vector per image (up to 4 images per post). Stored alongside text embeddings in the vector database for image corroboration during verification.

Step 7—S3 Image Upload. Images are downloaded, uploaded to AWS S3 (bucket `auragouploader`, key format `images/{source}/{post_id}_{i}.{ext}`), and the resulting S3 URLs are stored in the SourcePost document.

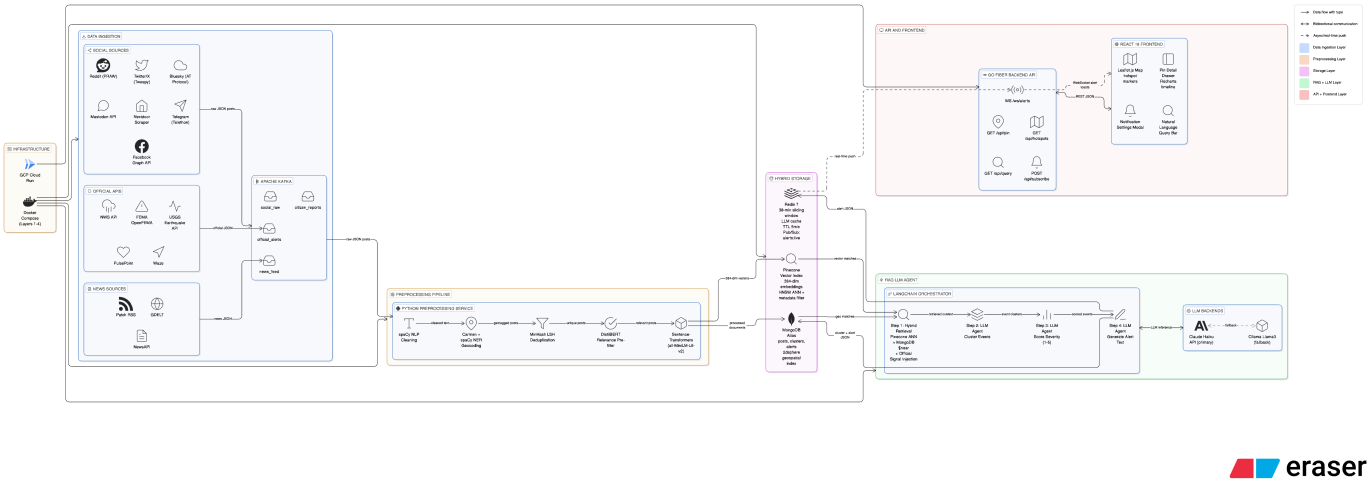


Fig. 1. Original Proposed System Architecture (from project proposal). The initial design envisioned a five-layer pipeline with a Next.js frontend, Voyage AI embeddings, and Claude Haiku as the primary LLM.

Step 8—MongoDB Persistence. Three writes per post: (A) SourcePost document to the per-source collection (e.g., `reddit_posts`, `usgs_alerts`); (B) text embedding document to `source_post_embeddings` (vector DB); (C) image embedding documents (one per image) to `source_post_embeddings`.

VI. Retrieval and LLM Agent

A. Hybrid Retrieval

Every 60 seconds (or immediately on volume spike detection: >10 posts in 30 seconds from the same 0.5° grid cell), the orchestrator dispatches a `run_pipeline` Celery task to the `agent` queue. The `HybridRetriever` class executes four parallel sub-queries:

Q1—Atlas Vector Search. Queries `source_post_embeddings` (vector_type="text") using Atlas Vector Search with the `text_vector_index`. Returns the top 50 semantically similar posts within a 50km bounding box and 2-hour lookback window.

Q2—Geo $near Search. Queries all per-source collections (8 collections) with MongoDB $near. Radius: 50km. Time window: 2 hours. Merges results across collections (limit 200 total).

Q3—Official Signals. Queries only official collections (usgs_alerts, gdacs_alerts, reliefweb_alerts, nasa_firms_alerts) as a boolean corroboration signal.

Q4—Location Enrichment. For posts with `location_source="unresolved"`: checks `location_cache` (text hash $\rightarrow$ cached coordinates) and `geo_priors` ($near on known locations).

Results are merged, deduplicated by post ID, and ranked by composite score: $0.5 \times \text{vector\_similarity} + 0.3 \times \text{recency} + 0.2 \times \text{source\_authority}$.

B. Three-Step LLM Agent

Google Gemini 2.0 Flash (primary) or Ollama Llama3 (offline fallback) executes three LangChain LCEL chains:

Step 1—Cluster Chain. Input: up to 50 retrieved posts + trigger location. The LLM identifies distinct real-world events and returns a JSON array of clusters with `event_type`, `location_description`, `contributing_post_ids[]`, and `confidence_score`. *Gate:* confidence $< 0.6 \rightarrow$ skip cluster.

Step 2—Severity Chain. Per cluster, the LLM rates severity 1–5 (1=unconfirmed minor, 2=possible minor, 3=confirmed moderate, 4=confirmed major, 5=confirmed mass casualty/critical infrastructure). *Gate:* severity $< 3 \rightarrow$ skip cluster.

Step 3—Alert Chain. The LLM writes a 2–3 sentence public alert: calm, factual, actionable, no usernames.

C. Verification System

Three additive evidence paths determine whether a cluster becomes a verified Crisis:

Path A—Social Corroboration: ≥ 2 distinct sources AND ≥ 3 distinct users $\rightarrow$ confidence += 0.5.

Path B—Official Corroboration: Any USGS/GDACS/ReliefWeb/NASA FIRMS post nearby $\rightarrow$ confidence += 0.4.

Path C—Image Corroboration: Image-text alignment ≥ 0.75 AND ≥ 2 users $\rightarrow$ confidence += 0.2.

Confidence is capped at 1.0. Unverified clusters still write a UnifiedPost (verified=false) for analytics, but no Crisis or Alert is created. This multi-gate design prevents false positives on the map while preserving all data for future analysis.

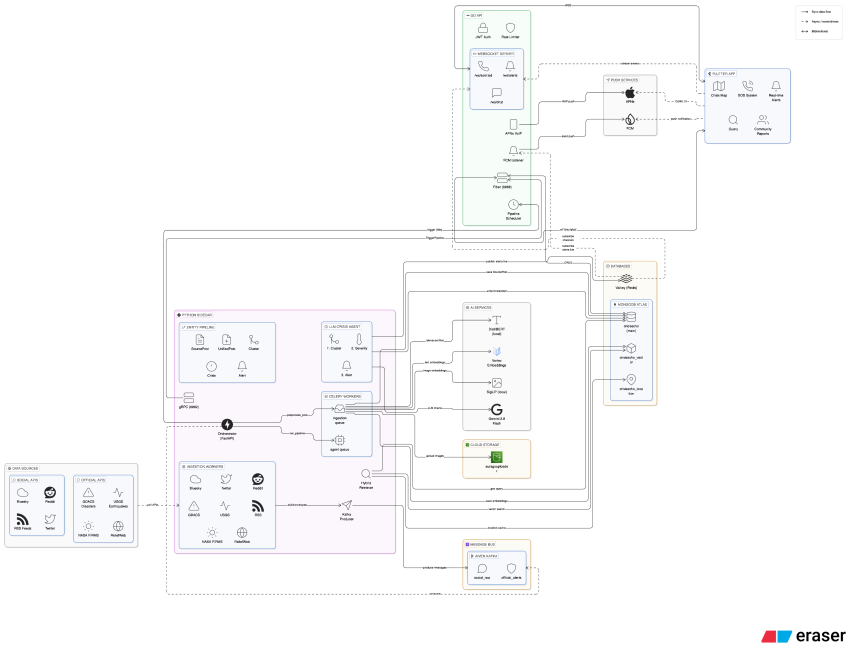


Fig. 2. Final Implemented System Architecture. Key changes from the proposal: Flutter replaced Next.js; Gemini 2.0 Flash replaced Claude Haiku; Vertex AI + SigLIP replaced Voyage AI + CLIP; Celery task queues and gRPC were added; the SOS emergency system with VoIP push was added as a major new subsystem.

D. Entity Hierarchy

The entity hierarchy provides a complete audit trail: **SourcePost** (normalized raw post in per-source collection) → **Cluster** (internal LLM grouping, never exposed to frontend) → **UnifiedPost** (LLM-synthesized summary) → **Crisis** (only if verified; the map dot) → **Alert** (push notification via Redis → FCM/APNs).

VII. Go API Layer

The Go API is built with Fiber v2 and follows a domain-driven structure: each entity has separate `model/`, `repository/`, `service/`, and `controller/` packages. Table I lists all 20 domain modules.

Middleware. Three middleware components: (1) `JWTAuth`—verifies Firebase Auth ID tokens and app-issued JWTs, storing user context in request locals; (2) `RateLimit`—sliding-window per-user rate limiting with auto-cleanup; (3) `RequirePlan`—gates features behind billing plans (Pro/Enterprise) by checking the user's Stripe subscription.

WebSocket. Three WebSocket endpoints backed by Redis pub/sub: `/ws/alerts` (live crisis alert stream), `/ws/sos/:sessionId` (SOS location relay room), and `/ws/chat/:sessionId/:helperId` (SOS private encrypted chat).

VIII. SOS Emergency System

The SOS system enables a user in distress to broadcast an emergency alert to nearby opted-in helpers using an Uber-style proximity broadcast model. This was a major feature addition not in the original proposal.

TABLE I
Go API Domain Modules

Module	Purpose
crisis	Verified crisis events (map dots)
unifiedpost	LLM-synthesized summaries
post	SourcePosts (per-source collections)
cluster	Internal LLM groupings
alert	Push notification records
user	User accounts + device tokens
auth	Firebase Auth (Google, Apple, Phone OTP)
sos	SOS profiles + alerts (legacy)
sos (session)	SOS sessions, responses, messages, contacts
community	Community crisis reports
analytics	Dashboard data + SOS analytics
billing	Stripe subscriptions + payment methods
category	51 parent categories + 78 subcategories
notify	Location-based subscriptions
upload	S3 presigned URLs + direct upload
query	Natural language queries (forwards to sidecar)
rag	Pipeline trigger (pings sidecar every 60s)
ingest	Kafka consumer (logging only)
responder	Official responder profiles
location	Saved user locations

A. Session Lifecycle

Trigger (`POST /api/sos/trigger`): Rate-limited to 3 per 10 minutes. Creates an `SOSSession` document (status=active), publishes to Redis `alerts:live`, notifies saved emergency contacts via FCM, and starts the wave broadcast goroutine.

Wave Broadcasting: A background goroutine runs in a loop: (1) check session still active; (2) query MongoDB 2dsphere index for nearest 20 opted-in users (excluding sender, previously notified, and other active SOS senders); (3) create pending `SOSResponse` records; (4) send push notifications (VoIP-first, FCM-fallback); (5) wait 60 seconds for responses; (6) expire unanswered responses; (7) repeat if accepted helpers < 4. Stops when ≥ 4 helpers accept or no candidates remain.

Accept/Decline/Leave/Resolve: Helpers accept (`POST /:sessionId/accept`), decline (`POST /:sessionId/decline`), or leave after accepting (`POST /:sessionId/leave`). Only the sender can resolve the session (`POST /:sessionId/resolve`), which sets a durable Redis key (TTL 1 hour), publishes termination events, sends FCM to all helpers, and schedules chat message purge after 24 hours.

B. Push Notifications

A VoIP-first, FCM-fallback strategy:

iOS with VoIP token: Apple PushKit VoIP push via HTTP/2 with certificate-based TLS (`.p12`). This triggers a full-screen CallKit incoming-call UI with native Accept/Decline buttons—works even when the app is force-quit or the phone is locked. If the VoIP token is stale (410 BadDeviceToken), it falls through to FCM.

Android or no VoIP token: Firebase Cloud Messaging data-only push. The Flutter app builds a local notification with Accept/Decline action buttons.

Stale tokens (FCM: NOT_FOUND/UNREGISTERED; APNs: 410/BadDeviceToken) are automatically cleared from user records.

C. Real-Time Location Tracking

All SOS participants connect to `/ws/sos/{sessionId}`, a WebSocket room backed by Redis pub/sub channel `sos:{sessionId}`. Location updates are relayed in real-time with echo prevention (each connection gets a random `_conn_id`; messages with matching `_conn_id` are not relayed back to the sender). A fallback ticker polls the durable Redis key every 5 seconds, and a 15-second health ping prevents Aiven from dropping idle subscriptions.

D. Encrypted Chat

Each sender–helper pair communicates via `/ws/chat/{sessionId}/{helperId}`. Messages are encrypted at rest using AES-256-GCM: a random 12-byte nonce is prepended to the ciphertext, and the result is base64-encoded for MongoDB storage. The encryption key is a 32-byte value from the `SOS_ENCRYPTION_KEY` environment variable. Messages are auto-purged 24 hours after session resolution.

IX. Flutter Mobile Application

The frontend is a Flutter mobile application targeting iOS and Android from a single codebase. It uses Riverpod for state management, Go Router for navigation, Dio for HTTP with JWT interceptors, and Flutter Secure Storage for credential persistence.

Fig. 3 shows the crisis map and drill-down screens. The primary screen displays an interactive Google Maps view with severity-colored crisis dots and category-specific SVG icons spanning 51 parent categories. Tapping a dot reveals the LLM-generated analysis summary with confidence scores, contributor counts, and official corroboration badges.

Fig. 4 shows the SOS emergency system screens. The sender triggers an SOS, sees the wave broadcast progress, and once helpers accept, all participants appear on a shared live map. Private encrypted chat is available between the sender and each helper. On iOS, incoming SOS requests appear as full-screen CallKit alerts.

Fig. 5 shows community reports, analytics, and profile management screens. Users can submit crisis reports with images, view analytics dashboards (plan-gated), and manage their profiles.

Fig. 6 shows location management and billing screens. Users can save locations for custom alert radii, and subscribe to Pro or Enterprise plans via Stripe.

X. Database Design

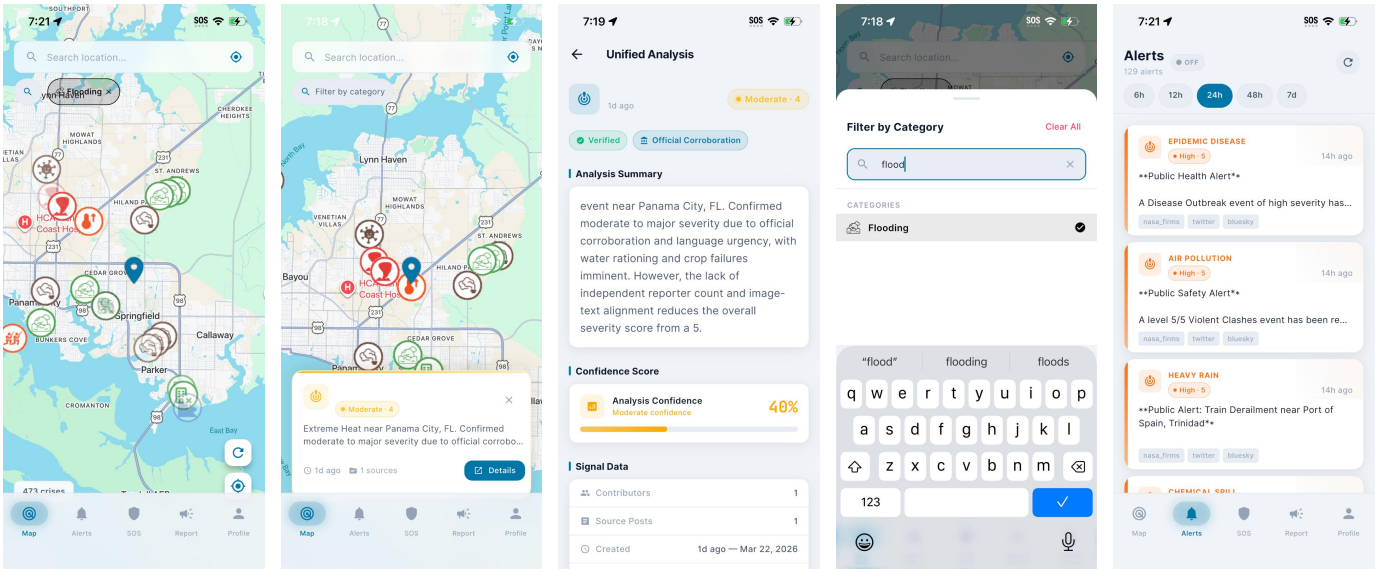
Three MongoDB Atlas databases serve distinct access patterns, avoiding contention between operational CRUD, vector similarity search, and location enrichment workloads.

DB 1—Main (`crisisecho`). 30+ collections including per-source SourcePost collections (`reddit_posts`, `bluesky_posts`, `twitter_posts`, `rss_posts`, `usgs_alerts`, `gdacs_alerts`, `reliefweb_alerts`, `nasa_firms_alerts`), pipeline entities (`clusters`, `unified_posts`, `crises`, `alerts`), SOS collections (`sos_sessions`, `sos_responses`, `sos_messages`, `user_emergency_contacts`), and application collections (`users`, `community_reports`, `categories`, `subcategories`, `subscriptions`, `billing`, `saved_locations`). All location fields carry compound 2dsphere indexes.

DB 2—Vector (`crisisecho_vector`). Single collection `source_post_embeddings` with two Atlas Vector Search indexes: `text_vector_index` (1408-dim, cosine) and `image_vector_index` (512-dim, cosine). Each document stores `post_id`, `source`, `vector`, `vector_type` ("text" or "image"), `location` (GeoJSON), `timestamp`, `crisis_type`, and optional `image_index`.

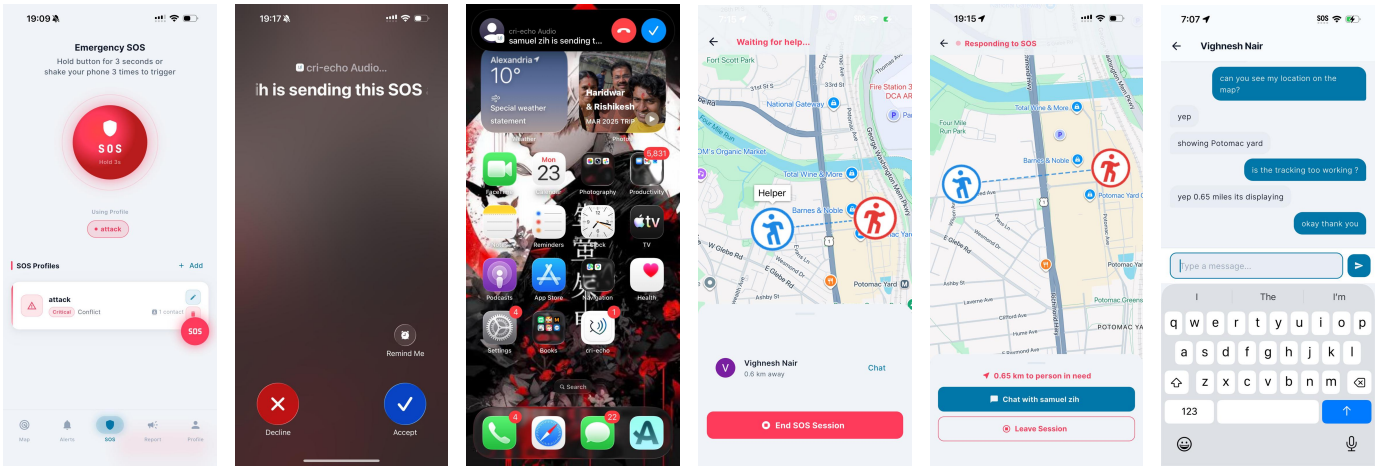
DB 3—Location (`crisisecho_location`). Three collections for geocoding support: `location_cache` (SHA-256 text hash $\rightarrow$ coordinates, TTL-indexed), `geo_priors` (known geographic reference points with 2dsphere index), and `place_index` (place name lookups).

Redis (Aiven Valkey). Serves three roles: (1) pub/sub channels for real-time alerts, SOS location relay, and chat; (2) Celery task broker and result backend; (3) durable keys for session termination fallback (`sos_resolved:{sessionId}`, TTL 1 hour).



(a) Crisis map (b) Selected crisis (c) Crisis details (d) Category filter (e) Alerts feed

Fig. 3. Crisis Map and Details. (a) Map displaying crisis events with severity-colored dots and category icons. (b) Map with a selected crisis showing a preview card. (c) Drill-down page with LLM-generated analysis, confidence score, and official corroboration status. (d) Category search and filter screen with 51 crisis categories. (e) Real-time alerts feed screen.



(a) SOS screen (b) CallKit alert (c) CallKit mini (d) Sender session (e) Helper session (f) Sender chat

Fig. 4. SOS Emergency System. (a) SOS trigger screen. (b) Full-screen CallKit alert on iOS via VoIP push with Accept/Decline buttons. (c) Minimized CallKit banner. (d) Sender's live map showing helper locations with "Waiting for help" status. (e) Helper's view of the sender's location. (f) Encrypted 1-on-1 chat between sender and helper.

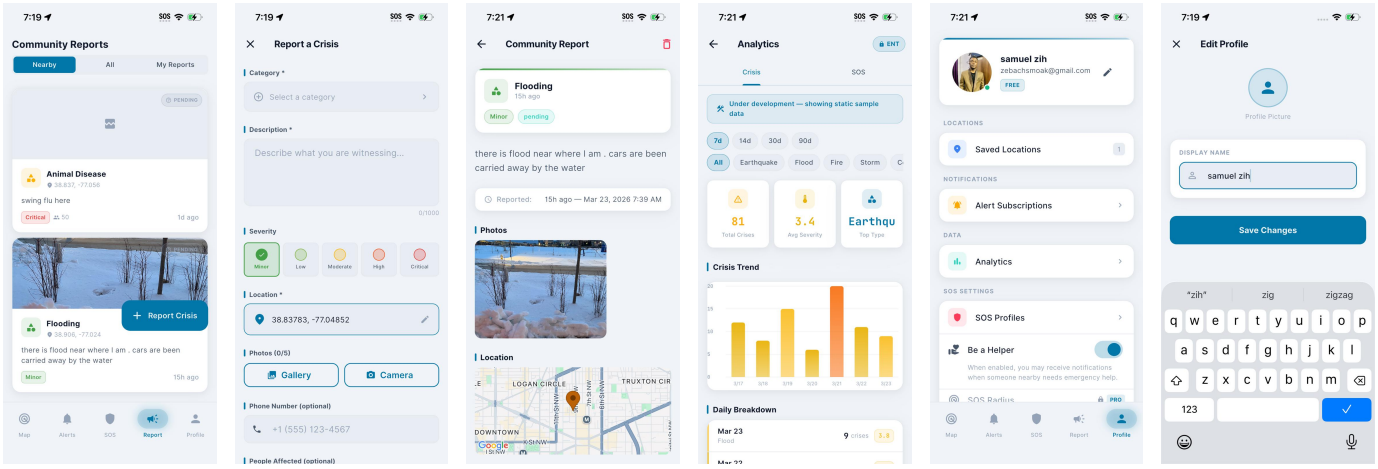
AWS S3. Stores source post images in bucket `auragouploader`. The Go API mediates all S3 access, keeping AWS credentials in one place.

XI. Key Design Decisions

Two-Service Split. Go is optimized for HTTP serving and goroutine concurrency but poor at AI/ML. Python has the ecosystem (LangChain, transformers, PyMongo, vector ops). Separate containers enable independent scaling, updates, and crash isolation. The tradeoff is deployment complexity (two Docker images, dependency ordering via health checks).

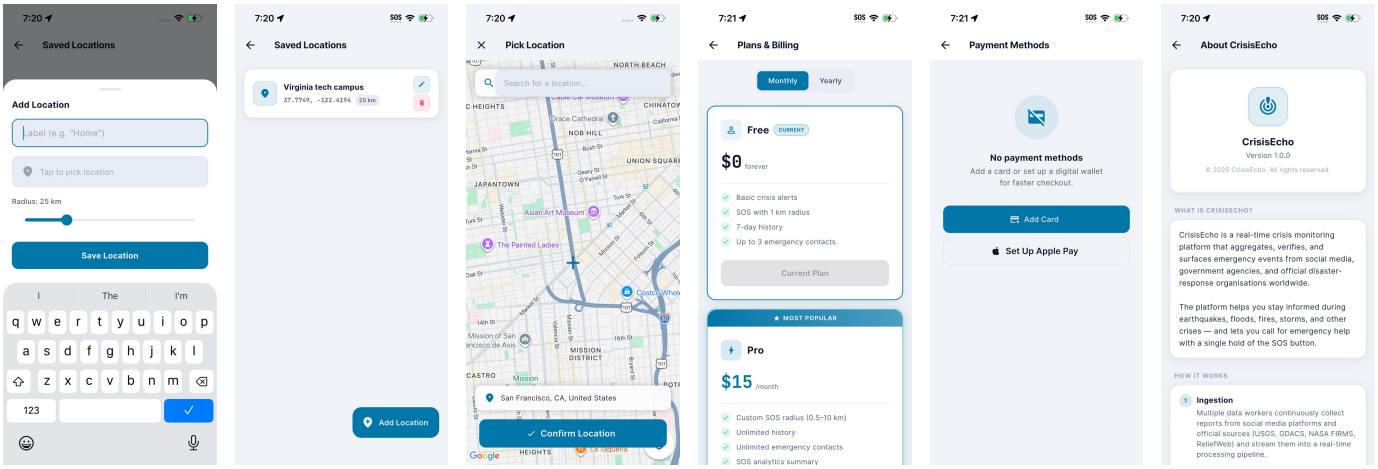
Three Databases. Vector search indexes and location lookups have different access patterns and scaling needs than operational CRUD. Separation prevents a large vector index from impacting user-facing query performance and makes it easier to swap the vector store (e.g., replace MongoDB with Pinecone) without touching the main database.

LLM for Clustering. Rule-based clustering (DBSCAN, k-means) cannot distinguish "earthquake" from "gas explosion" in the same city when both have nearby posts. The LLM's semantic understanding can separate events by meaning, not just proximity.



(a) Reports feed (b) Create report (c) Report details (d) Analytics (e) Profile (f) Edit profile

Fig. 5. Community, Analytics, and Profile. (a) Community crisis reports feed. (b) Create new report with severity, category, description, and image upload. (c) Report detail view with status tracking. (d) Analytics dashboard with crisis trend charts and category breakdowns (Enterprise plan). (e) User profile page. (f) Edit profile screen.



(a) Add location (b) Saved locations (c) Pick on map (d) Billing plans (e) Payment (f) About

Fig. 6. Locations, Billing, and Settings. (a) Add a saved location for custom alerts. (b) List of saved locations with notification radii. (c) Map-based location picker. (d) Billing plans: Free, Pro ($15/mo), Enterprise ($499/mo). (e) Stripe payment screen. (f) About CrisisEcho screen.

GPS-Only Location Rule. UnifiedPost centroids use only SourcePosts where `location_source=="gps"` and `location_confidence==1.0`. Social media posts often have city-level or IP-based locations that are tens of kilometers from the actual event. This rule prevents map pins from being placed incorrectly.

Verification Gates. The triple gate (confidence $\geq$ 0.6, severity $\geq$ 3, crisis verification) prevents false positives on the map. Unverified data is still stored (verified=false) for analytics and future ML training—no data is ever discarded.

VoIP Push for SOS. FCM cannot trigger a full-screen UI on iOS when the app is killed. Only Apple PushKit VoIP push can wake the app and show CallKit Accept/Decline buttons. This is critical for emergency scenarios where the helper's phone is locked or the app is not running.

XII. Deployment Architecture

Current (Local). Docker Compose with four containers sharing a `.env` file. The Go API depends on the Python sidecar's health check. Aiven Kafka, Aiven Valkey, and MongoDB Atlas are external managed services—the same URIs work in both local Docker and cloud deployment.

Target (Cloud). Two Google Cloud Run services:

- `crisisecho-api`: Go binary, port 8080, environment variable `PYTHON_SIDECAR_URL` pointing to the sidecar's Cloud Run URL.
- `crisisecho-sidecar`: Python, ports 8081 (HTTP) + 8082 (gRPC), environment variable `GO_API_BASE` pointing to the API's Cloud Run URL.

Both services are deployed as separate Cloud Run instances. The Flutter app will be published to the Apple App Store.

XIII. Current Status and Remaining Work

Completed:

- Go API: 20 domain modules, 3 middleware, 3 WebSocket endpoints, Firebase Auth, FCM, APNs VoIP push
- Python Sidecar: 8 ingestion workers, 8-step preprocessor, hybrid retriever, 3-step LLM agent, crisis verifier, Celery (2 queues), gRPC server, FastAPI, volume spike detection
- Databases: 3 MongoDB Atlas databases with 30+ collections, 2dsphere and Atlas Vector Search indexes, Redis pub/sub channels
- Frontend: Flutter app with 24+ screens (crisis map, SOS, community reports, analytics, billing, profiles, locations)
- Infrastructure: Docker Compose (4 containers), Kafka topics, S3 media storage, Firebase configuration

Remaining:

1) Deploy Go API to GCP Cloud Run
2) Deploy Python sidecar to GCP Cloud Run
3) Publish Flutter app to Apple App Store

XIV. Task Assignment

All work was completed by a single developer. Table II details responsibilities.

TABLE II
Task Assignment

Member	Responsibilities
S. Zih	Backend: Go Fiber API (20 modules), Python sidecar (LLM pipeline, preprocessing, 8 ingestion workers, Celery, gRPC), MongoDB (3 databases, 30+ collections), Kafka, Redis, WebSocket, APNs VoIP, FCM, Docker; Frontend: Flutter (24+ screens), Google Maps, SOS tracking, encrypted chat, Stripe billing; AI/ML: DistilBERT relevance, SigLIP embeddings, Vertex AI embeddings, LangChain agent, verification; Report: all sections

XV. Schedule

Table III compares the originally planned schedule with actual progress.

TABLE III
Project Schedule—Planned vs. Actual

Weeks	Planned	Actual
1–2	Literature review; provision Atlas, Kafka, Redis; define schemas	Completed as planned
3–4	Ingestion workers; spaCy + geocoding; DistilBERT; embeddings	Completed; added SigLIP embeddings, Celery
5–6	Hybrid retrieval; LangChain 3-step agent	Completed; switched LLM to Gemini 2.0 Flash
7–8	Cluster persistence; Redis Pub/Sub; Go Fiber API	Completed; added gRPC, 20 domain modules
9–10	Next.js frontend; evaluation	Pivoted to Flutter; built 24+ screens including SOS
11–12	Stretch features; system testing	Built full SOS system with VoIP push, billing, community reports, categories
13	Documentation; demo	Cloud Run deployment (in progress); App Store submission pending

References

[1] M. Imran, P. Mitra, and C. Castillo, "Twitter as a lifeline: Human-annotated Twitter corpora for NLP of crisis-related messages," in *Proc. LREC*, 2016.

[2] A. Olteanu, C. Castillo, F. Diaz, and S. Vieweg, "CrisisLex: A lexicon for collecting and filtering microblogged communications in crises," in *Proc. ICWSM*, 2014.

[3] F. Alam, F. Ofli, and M. Imran, "HumAID: Human-annotated disaster incidents data from Twitter," in *Proc. ICWSM*, 2021.

[4] S. Middleton, L. Middleton, and S. Modafferi, "Real-time crisis mapping of natural disasters using social media," *IEEE Intell. Syst.*, vol. 29, no. 2, pp. 9–17, 2014.

[5] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proc. NeurIPS*, 2020.

[6] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *Proc. ICLR*, 2023.

[7] V. Karpukhin, B. Oğuz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W. Yih, "Dense passage retrieval for open-domain question answering," in *Proc. EMNLP*, 2020.

[8] T. Sakaki, M. Okazaki, and Y. Matsuo, "Earthquake shakes Twitter users: Real-time event detection by social sensors," in *Proc. WWW*, 2010.

[9] J. P. de Albuquerque, B. Herfort, A. Brenning, and A. Zipf, "A geographic approach for combining social media and authoritative data towards identifying useful information for disaster management," *Int. J. Geogr. Inf. Sci.*, vol. 29, no. 4, pp. 667–689, 2015.