Proposal for CrisisEcho: 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*—CrisisEcho is a consumer-facing mobile platform that applies Retrieval-Augmented Generation (RAG) and multi-step LLM agents to detect, classify, and summarize hyperlocal emergencies from social media and official data sources in near real-time. Official emergency channels routinely lag events by minutes or hours; CrisisEcho closes this gap by ingesting eight social and official feeds via Apache Kafka, processing them through an eight-step ML preprocessing pipeline, and running a three-step LangChain agent chain that clusters posts, scores severity, verifies events through multi-source corroboration, and generates plain-language alerts. The system stores data across three MongoDB Atlas databases (document, vector, location), Aiven Valkey (Redis-compatible) for real-time pub/sub, and AWS S3 for media. The backend is split into a Go Fiber REST API serving a Flutter mobile application and a Python AI sidecar handling all machine learning workloads. A peer-to-peer SOS emergency system with proximity-based wave broadcasting, real-time WebSocket location tracking, AES-256-GCM encrypted chat, and Apple VoIP push notifications via CallKit extends the platform beyond passive crisis monitoring into active emergency response. The system is fully implemented and containerized for deployment to Google Cloud Run.**

***Index Terms*—emergency detection, social media mining, RAG, LLM agents, MongoDB Atlas Vector Search, real-time systems, crisis informatics, SOS, Flutter, Go, WebSocket**

I. Introduction

Official channels—911 dispatch, government sensors, police scanners—frequently lag fast-moving crises by minutes or hours. Ordinary people post hyperlocal signals on social media well before any official report: "Water rising fast on my street!" or "Heard gunshots near the mall." Existing tools are keyword-based and fail on indirect language, sarcasm, and regional idioms. CrisisEcho replaces rigid keyword matching with semantic RAG-powered reasoning.

Motivation. In the critical first minutes of an emergency, the gap between on-the-ground reality and official response can be fatal. Social media posts constitute the largest real-time sensor network on Earth, yet extracting actionable crisis intelligence from this firehose of unstructured, multilingual, often ambiguous text remains an unsolved problem at scale. According to the HumAID dataset [3], crisis-related posts span dozens of event types—from natural disasters to armed conflicts—and require contextual understanding that simple keyword filters cannot provide. A post saying "the ground won't stop shaking" is clearly about an earthquake, but contains none of the keywords a traditional system would match.

Problem Statement. Build a system that (1) continuously ingests social and official data sources, (2) identifies genuine crisis events using semantic understanding rather than keyword matching, (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 LLM-driven paradigm shifts: text to semantics (vector embeddings replace keyword filters), retrieval to reasoning (hybrid retrieval feeds chain-of-thought LLM agents), vertical to multi-domain (51 crisis categories from wildfires to epidemics), and closed-world to open-world generalization (the LLM recognizes novel crisis types without retraining). The addition of a real-time SOS system transforms the platform from passive monitoring into active emergency response infrastructure. Unlike existing crisis informatics tools that target researchers or government analysts, CrisisEcho is designed as a consumer-facing mobile application where any user can open a map, see verified crisis events nearby, and—in an emergency—broadcast an SOS to nearby helpers.

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. 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. Data Sources

The final integrated set of eight data sources is organized into two Kafka topics based on authority level:

Social (topic: `social_raw`): Reddit (PRAW streaming from crisis-related subreddits), Twitter/X (twscrape polling with crisis search queries), Bluesky (AT Protocol firehose with keyword filtering), and RSS feeds (configurable news sources).

Official (topic: `official_alerts`): USGS Earthquake (magnitude ≥ 2.5), GDACS—Global Disaster Alert and Coordination System (UN-backed), ReliefWeb (UN OCHA humanitarian crisis API), and NASA FIRMS (Fire Information for Resource Management System for satellite wildfire detection).

Source authority weights are used in retrieval ranking: official sources score 1.0, Reddit 0.7, Twitter/Bluesky 0.6, and RSS 0.5. Sources removed from the original proposal (Mastodon, Telegram, NWS, PulsePoint, GDELT, Nextdoor) were eliminated due to restrictive APIs, low signal-to-noise ratios, or geographic limitations superseded by the final source selection.

IV. System Architecture

CrisisEcho runs 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 MongoDB and Redis. Fig. 1 shows the complete system architecture.

Ingestion Layer. Eight Python workers (one per source) poll or stream their respective APIs and produce messages to Kafka. Each worker inherits a `KafkaWorker` base class with retry logic (exponential backoff, 5 attempts) and user privacy hashing (SHA-256 before serialization).

Preprocessing. The Python sidecar consumes Kafka messages and applies an eight-step pipeline per post: (1) text cleaning (URL/emoji removal), (2) geocoding (GPS coordinates only—no guessing), (3) MinHash LSH deduplication (85% Jaccard threshold, 5-minute window), (4) DistilBERT relevance filtering (drops $\approx$70% of non-crisis content), (5) text embedding via Google Vertex multimodal (1408-dim), (6) image embedding via SigLIP (512-dim), (7) S3 image upload, and (8) MongoDB persistence to per-source collections and vector indexes.

Retrieval & Agent. Every 60 seconds (or on volume spike detection: $>$10 posts in 30s from same grid cell), a Celery task dispatches the pipeline. The hybrid retriever runs Atlas Vector Search (semantic, 50km/2h window) and geo \$near queries across all source collections, merging results with composite scoring: $0.5 \times \text{vector} + 0.3 \times \text{recency} + 0.2 \times \text{authority}$. A three-step LangChain agent (Google Gemini 2.0 Flash) then clusters posts into distinct events, scores severity 1–5, and generates public alert text.

Verification. A multi-gate verification system prevents false positives: confidence ≥ 0.6, severity ≥ 3, plus crisis verification through three additive evidence paths—social corroboration ($\geq$2 sources AND $\geq$3 users, +0.5 confidence), official corroboration (any nearby USGS/GDACS/ReliefWeb/NASA FIRMS post, +0.4), and image corroboration (SigLIP cosine similarity ≥ 0.85, +0.2). Only verified events create Crisis documents visible on the map.

Entity Hierarchy. SourcePost (normalized raw post) $\rightarrow$ Cluster (internal LLM grouping, never exposed) $\rightarrow$ UnifiedPost (LLM synthesis) $\rightarrow$ Crisis (verified, map dot) $\rightarrow$ Alert (push notification via Redis pub/sub $\rightarrow$ Firebase/APNs).

V. Database Architecture

Three MongoDB Atlas databases serve distinct access patterns:

DB 1—Main (`crisisecho`). Per-source collections (`reddit_posts`, `usgs_alerts`, etc.), plus `clusters`, `unified_posts`, `crises`, `alerts`, `users`, `sos_sessions`, `sos_responses`, `sos_messages`, `community_reports`, `categories`, `subscriptions`, and `billing`. All location fields carry 2dsphere indexes.

DB 2—Vector (`crisisecho_vector`). Collection `source_post_embeddings` stores text vectors (1408-dim) and image vectors (512-dim) with Atlas Vector Search indexes. Each document carries `post_id`, `source`, `vector_type`, `location`, `timestamp`, and `crisis_type` as filterable metadata.

DB 3—Location (`crisisecho_location`). Collections `location_cache` (text hash $\rightarrow$ coordinates, TTL-indexed), `geo_priors` (known reference points), and `place_index` (place name lookups).

Redis (Aiven Valkey). Pub/sub channels: `alerts:live` (crisis alerts), `sos:{sessionId}` (location relay), `chat:{sessionId}:{helperId}` (encrypted chat). Also serves as Celery task broker/backend.

AWS S3. Image storage (bucket `auragouploader`); only S3 URLs stored in MongoDB.

VI. RAG Pipeline and LLM Agent

A. Hybrid Retrieval

Each pipeline trigger issues four concurrent sub-queries: (1) Atlas Vector Search on `source_post_embeddings` (top 50 by cosine similarity, filtered by 2-hour time window and 50km bounding box); (2) MongoDB \$near across all per-source collections (geographic proximity); (3) official signals query (USGS, GDACS, ReliefWeb, NASA FIRMS only); (4) location enrichment for unresolved geocodes via the location database. Results are merged, deduplicated, and ranked by composite score.

B. Three-Step LLM Agent

Google Gemini 2.0 Flash (primary) or Ollama Llama3 (fallback) executes three LangChain LCEL chains: **Step 1** cluster posts into distinct real-world events (JSON output with contributing post IDs and confidence score); **Step 2** score severity 1–5 with rationale (1=unconfirmed minor, 5=confirmed mass casualty); **Step 3** generate a 2–3 sentence public alert—calm, factual, actionable, no usernames. The publish threshold requires severity ≥ 3, confidence ≥ 0.6, and successful crisis verification.

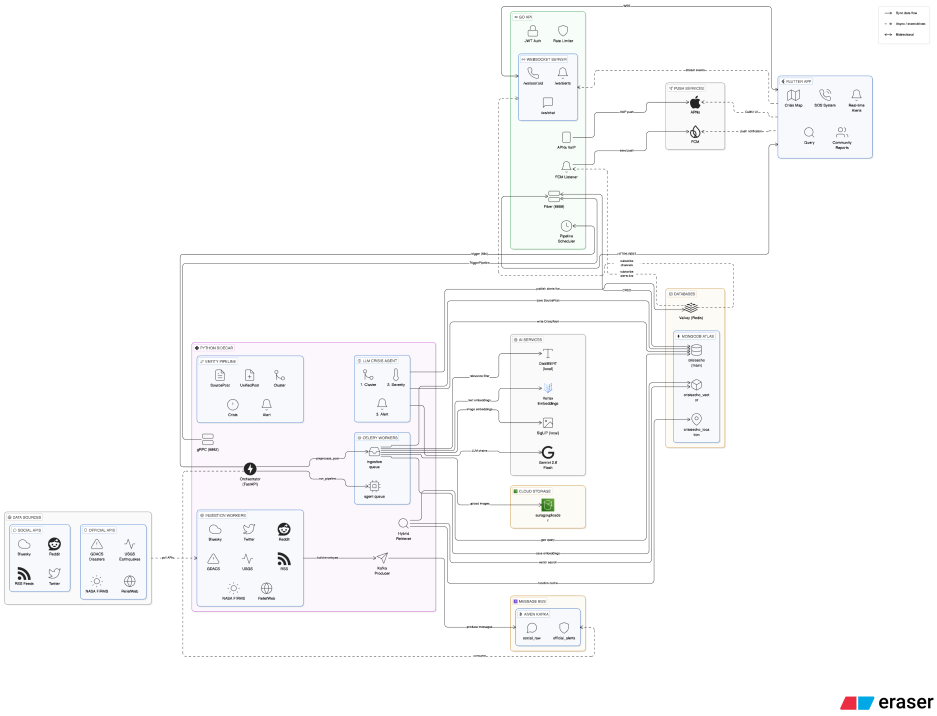

Fig. 1. CrisisEcho System Architecture (Final Implementation). Data flows left-to-right: eight ingestion workers publish to Kafka, the Python sidecar preprocesses and runs the LLM agent pipeline, MongoDB stores all entities, the Go API serves the Flutter app via REST and WebSocket, and push notifications reach users via Firebase Cloud Messaging and Apple APNs VoIP push.

C. Location Accuracy Rule

UnifiedPost centroids use *only* SourcePosts where `location_source=="gps"` and `location_confidence==1.0`. This prevents imprecise (city-level, IP-based) coordinates from distorting map pin placement, falling back to trigger coordinates when no GPS posts are available.

VII. SOS Emergency System

Beyond passive crisis monitoring, CrisisEcho includes a peer-to-peer SOS system modeled after Uber-style proximity broadcasting. A user in distress triggers an SOS alert, which broadcasts to nearby opted-in helpers in expanding waves (20 users per wave, 60-second acceptance windows) until 4–7 helpers accept. The system is backed by six MongoDB collections (`sos_sessions`, `sos_responses`, `sos_messages`, `user_emergency_contacts`, `sos_profiles`, `sos_alerts`) and three Redis pub/sub channels per session.

Session Lifecycle. The sender calls `POST /api/sos/trigger` with their GPS coordinates. The server creates an `SOSSession` document (status=active), publishes an event to Redis `alerts:live`, asynchronously notifies saved emergency contacts via FCM, and starts a background goroutine that runs the wave broadcast loop. Each wave queries MongoDB's 2dsphere index for the nearest 20 opted-in users (excluding the sender, previously notified users, and other active SOS senders), creates pending `SOSResponse` records, and sends push notifications. Helpers who accept are added to the session's `AcceptedHelperIDs` array. Helpers may leave voluntarily without ending the session; only the sender can resolve (end) a session.

Push Notifications. A VoIP-first, FCM-fallback strategy ensures maximum delivery: iOS users receive Apple PushKit VoIP push notifications that trigger a full-screen CallKit UI with Accept/Decline buttons (works even when the app is force-quit or the phone is locked). Android users and iOS fallback receive Firebase Cloud Messaging data-only pushes. Stale device tokens (APNs 410 or FCM UNREGISTERED) are automatically cleared from user records.

Real-Time Tracking. All participants connect to a shared WebSocket room (`/ws/sos/{sessionId}`) backed by Redis pub/sub. Location updates are relayed in real-time with echo prevention via per-connection IDs. A fallback ticker polls a durable Redis key (`sos_resolved:{sessionId}`, TTL 1 hour) every 5 seconds to guarantee session termination delivery even if the pub/sub event is missed. A separate private chat WebSocket room (`/ws/chat/{sessionId}/{helperId}`) enables 1-on-1 communication between the sender and each individual helper.

Security. Chat messages are encrypted at rest using AES-256-GCM (random 12-byte nonce prepended to ciphertext,

base64-encoded for storage) and auto-purged 24 hours after session resolution. SOS triggering is rate-limited (3 per 10 minutes per user). Custom broadcast radius is gated behind Pro/Enterprise billing plans via plan middleware.

VIII. Mobile Application

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

Crisis Map. The primary screen is an interactive Google Maps view displaying severity-colored crisis dots with category-specific SVG icons spanning 51 parent categories and 78 subcategories. Users can filter crises by event type, severity, and date range. Tapping a crisis dot navigates to a drill-down page showing the LLM-generated analysis summary, confidence score, contributor count, official corroboration status, and links to contributing source posts.

SOS Interface. The SOS screen allows users to trigger emergency broadcasts, view the wave progress, and—once helpers accept—see all participants on a live map with real-time location updates. A built-in encrypted chat connects the sender with each helper individually. On iOS, incoming SOS requests appear as full-screen CallKit alerts via PushKit VoIP push.

Additional Features. The app includes: community report submission with multi-image uploads to S3; a real-time alerts feed via WebSocket; an analytics dashboard with crisis trend charts (plan-gated for Pro/Enterprise users); saved locations with custom notification radii; user profile management; and a Stripe-integrated billing system with Free, Pro ($15/month), and Enterprise ($499/month) tiers. The complete application comprises 24+ distinct screens.

IX. Technology Stack

Table I lists the finalized technology stack as implemented.

TABLE I
CrisisEcho Final Technology Stack

Component	Technology	Rationale
Backend API	Go (Fiber v2)	High throughput; goroutine WebSocket
AI Pipeline	Python, LangChain	LLM/NLP ecosystem; LCEL chains
Frontend	Flutter	Native iOS/Android; single codebase
Main DB	MongoDB Atlas	2dsphere geospatial; free tier
Vector DB	Atlas Vector Search	Multi-modal ANN; no extra service
Location DB	MongoDB Atlas	Geocoding cache + geo priors
Media	AWS S3	Image storage; lean MongoDB docs
Cache/PubSub	Aiven Valkey	Redis-compatible; TLS; Celery broker
Queue	Aiven Kafka	Managed; decouples ingestion
LLM	Gemini 2.0 Flash	Free tier; fast inference
Embeddings	Vertex AI + SigLIP	Text 1408-dim; image 512-dim
Push	FCM + APNs VoIP	Full-screen CallKit on iOS
Billing	Stripe	Checkout + webhooks
Deploy	GCP Cloud Run	Serverless; two services

TABLE II
Implementation Scope Summary

Area	Deliverables
Go API	20 domain modules (model/repo/service/controller), 3 middleware, Fiber v2 REST, 3 WebSocket endpoints
Python Sidecar	8 ingestion workers, 8-step preprocessor, hybrid retriever, 3-step LLM agent, crisis verifier, Celery (2 queues), gRPC server, FastAPI
Databases	3 MongoDB Atlas databases, 30+ collections, 2dsphere and vector indexes, Redis pub/sub
Frontend	Flutter app with 24+ screens, Google Maps, SOS tracking, encrypted chat, Stripe billing
Infrastructure	Docker Compose (4 containers), Kafka topics, S3 media storage, Firebase Auth/FCM, APNs VoIP

X. Progress Summary

All backend, AI pipeline, database, and frontend components have been implemented and tested end-to-end. Table II summarizes the implementation scope.

The system runs in Docker Compose with four containers: the Go API, the Python sidecar (FastAPI + gRPC + Kafka consumer), a Celery ingestion worker (concurrency=4), and a Celery agent worker (concurrency=2). Several key architectural pivots were made during development: the frontend was changed from Next.js to Flutter for native mobile performance; the LLM was switched from Claude Haiku to Gemini 2.0 Flash for its free tier; embeddings moved from Voyage AI (1024-dim) to Google Vertex multimodal (1408-dim text) and SigLIP (512-dim image); and the SOS emergency system—not in the original proposal—was added as a major new feature.

Remaining Work. (1) Deploy the Go API to GCP Cloud Run, (2) deploy the Python sidecar to a separate Cloud Run service (communicating via HTTP), and (3) publish the Flutter app to the Apple App Store. The project was completed by a single developer (Samuel Enam Zih) who was responsible for all architecture, backend, AI/ML, frontend, database design, Docker containerization, and documentation.

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.

APPENDIX

Task Assignment. All architectural decisions, backend development (Go API with 20 domain modules, Python AI sidecar with LLM pipeline, 8 ingestion workers, preprocessing, retrieval, verification), database design (3 MongoDB databases, Redis, S3), frontend development (Flutter mobile app with 24+ screens), SOS emergency system, Docker containerization, and documentation were completed by Samuel Enam Zih (sole team member).

TABLE III
RESPONSIBILITIES

Member	Responsibilities
S. Zih	Backend: Go Fiber API (20 modules), Python sidecar, MongoDB (3 DBs), Kafka, Redis, LangChain agent, SOS system, WebSocket, APNs VoIP, FCM, Docker; Frontend: Flutter (24+ screens), Google Maps, SOS tracking, encrypted chat; AI/ML: preprocessing pipeline, hybrid retrieval, LLM clustering, verification; Report: all sections

TABLE IV
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
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 Celery task queues, gRPC
9–10	Next.js frontend; evaluation	Pivoted to Flutter; built 24+ screens
11–12	Stretch features; system testing	Built full SOS system with VoIP push, billing, community reports
13	Documentation; demo	Cloud Run deployment (in progress)