CrisisEcho: Final Project 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 final report documents the complete implementation, post-checkpoint refinements, and production deployment of CrisisEcho, a real-time crisis detection and emergency response platform. CrisisEcho ingests eight social and official data sources via Apache Kafka with a non-blocking asynchronous producer, processes them through a fully concurrent Celery-driven preprocessing queue (text cleaning, NLP-based event-location extraction, MinHash deduplication on shared Redis state, DistilBERT relevance filtering, SigLIP unified text-image embeddings at 768 dimensions, S3 image upload, and per-source 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. All confidence and severity thresholds are externalised to a tunable configuration file, and every gate in the pipeline writes an instrumentation record to a dedicated `gate_metrics` collection so drop rates can be measured per source, per gate, and per reason without redeploying. The system is deployed on Google Kubernetes Engine (GKE Autopilot, project `aurago`, region `us-east4`) as four Deployments (Go API, Python sidecar, Celery ingestion worker, Celery agent worker) behind a LoadBalancer Service, an internal ClusterIP Service, a ConfigMap, and three Secrets. A Cloud Build pipeline rebuilds and rolls out both images on every push to main. 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. The Go API exposes both REST/WebSocket interfaces and a new gRPC interface to the Python sidecar for low-latency pipeline triggering and natural-language query handling.**

***Index Terms*—crisis detection, RAG, LLM agents, MongoDB, Go, Python, Flutter, SOS, WebSocket, GKE, Celery, gRPC, 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.

Scope of this Final Report. Compared to the project checkpoint report, this document additionally covers: (a) the full migration from a Cloud Run target deployment to a Google Kubernetes Engine (GKE) Autopilot production deployment with four Deployments, two Services, one ConfigMap, and three Secrets; (b) the restructuring of the Python sidecar into a thin orchestrator plus dedicated `ingestion/`, `agent/`, and `tasks/` packages, with a gRPC server running alongside FastAPI; (c) the unification of the text and image embedding spaces under a single SigLIP model; (d) the move of preprocessing from a single daemon thread into a dedicated Celery queue with dynamic worker allocation; (e) the conversion of the Kafka producer to non-blocking asynchronous sends; (f) NLP-driven event-location extraction that no longer relies on poster geotags; (g) a configuration-driven threshold layer; (h) end-to-end gate-rejection instrumentation; and (i) a Cloud Build CI/CD pipeline that performs rolling updates of all four cluster Deployments on every push to main.

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 event-location-first integrity rule.

III. System Architecture Overview

CrisisEcho is implemented as a two-image, four-pod production architecture: a **Go Fiber HTTP API** (port 8080) serving the Flutter mobile frontend, a **Python AI sidecar** (FastAPI on port 8081 plus a gRPC server on port 8082) that hosts shared agent code and dispatches Celery tasks, a **Celery ingestion worker** pod that runs the eight ingestion workers and the eight-step preprocessing pipeline against the `ingestion` queue (concurrency 4), and a **Celery agent worker** pod that runs the retrieval and LLM agent pipeline against the `agent` queue (concurrency 2). Fig. 1 shows the full implemented architecture, and Fig. 2 shows the simplified data-flow view.

Architectural Progression Since Checkpoint. The following changes were made between the checkpoint report and this final submission:

- **Deployment target**: Cloud Run (two services) → GKE Autopilot (four Deployments + two Services + one ConfigMap + three Secrets).
- **Sidecar layout**: monolithic `pipeline.py` → `orchestrator.py` (FastAPI host, APScheduler, Kafka consumer) plus `grpc_server.py` (gRPC) plus an `ingestion/` package (`processor.py`, `dedup.py`), an `agent/` package (`agent.py`, `retrieval.py`, `crisis_verifier.py`), and a `tasks/` package (Celery task entry points).
- **Embeddings**: separate Vertex AI 1408-dim text embeddings and SigLIP 512-dim image embeddings → a single SigLIP `google/siglip-base-patch16-224` model used for both text and image embeddings at 768 dimensions, placing both modalities in one cosine space.
- **Preprocessing**: single daemon thread inside the sidecar pod → Celery `preprocess_post` tasks consumed by the `worker-ingestion` pod with concurrency 4. The orchestrator's Kafka consumer is now a thin dispatcher that fans envelopes out to Celery.
- **Worker assignment**: static one-source-per-thread → dynamic Celery worker pool that pulls from any source based on load. A flooding source is automatically absorbed by idle workers.
- **Kafka producer**: blocking synchronous sends → asynchronous, non-blocking sends with internal callbacks for delivery acknowledgement.
- **Location extraction**: GPS-only with `location_source="unresolved"` fallback → NLP-based event-location extraction (spaCy NER + place gazetteer) that runs before geotag fallback. Geotags are now treated as a low-priority hint, not as authoritative data.
- **Thresholds**: hard-coded constants in source → a single `config.yaml` loaded at boot and re-readable via SIGHUP, covering Jaccard threshold, relevance threshold, cluster confidence floor, severity floor, and image-alignment cutoff.
- **Instrumentation**: `logger.info` only → structured `gate_metrics` collection with one document per dropped post per gate, queryable for current and future dashboards.
- **Communication**: HTTP only → HTTP plus gRPC. Go API calls the sidecar's gRPC `TriggerPipeline`, `RunQuery`, and `Health` methods on port 8082 for lower-latency internal traffic.
- **CI/CD**: manual `docker compose up` → Google Cloud Build trigger on push to main, eight-step pipeline (parallel image builds, parallel pushes, four `kubectl set image` rolling updates) on an E2_HIGHCPU_8 machine.

Canonical Data Flow. Raw data → Kafka (two topics: `social_raw` and `official_alerts`) → orchestrator's Kafka consumer fans out `preprocess_post` tasks → Celery ingestion worker pod runs the eight-step preprocessor concurrently → SourcePost (per-source MongoDB collections plus SigLIP embeddings in `source_post_embeddings`) → APScheduler dispatches `run_pipeline` every 60 s, plus volume-spike triggers → Celery agent worker pod runs hybrid retrieval (vector + geo + official + location enrichment) → three-step LLM agent (cluster → severity → alert) → Crisis (only if verified) → Alert (Redis `alerts:live` pub/sub → FCM/APNs → Flutter mobile app).

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. All ingestion workers now run inside the `crisisecho-worker-ingestion` Celery pod.

Social Sources (topic: `social_raw`):

- **RedditWorker**: PRAW library streaming from crisis-related subreddits.
- **TwitterWorker**: `twscrape` polling with crisis-specific search queries.

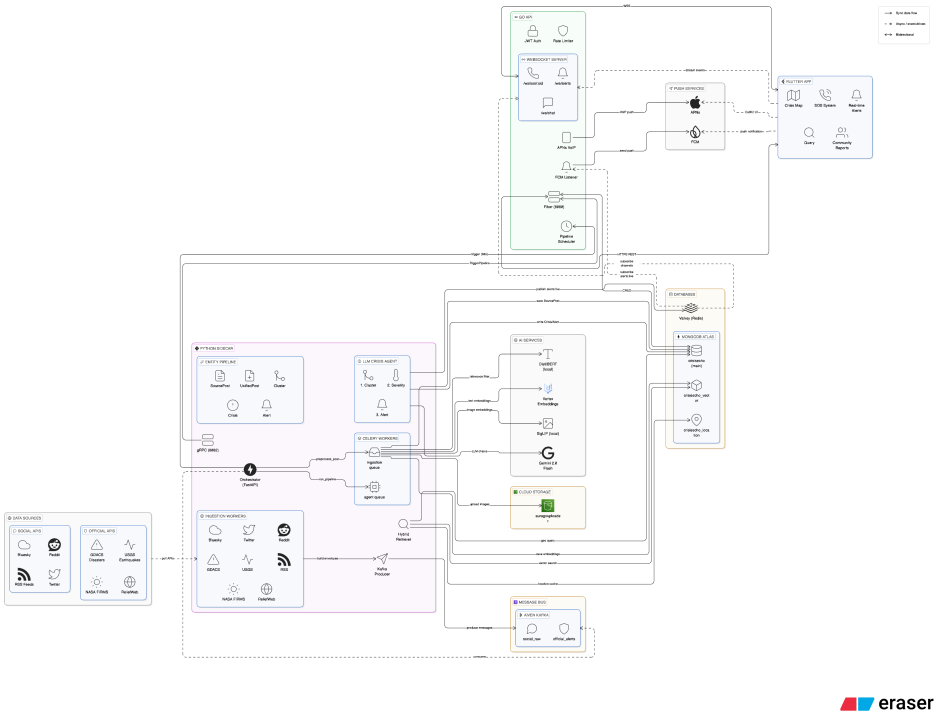


Fig. 1. Final Implemented System Architecture (GKE Autopilot, region us-east4). The system consists of four Kubernetes Deployments inside the `autopilot-cluster-1-crisisecho` cluster, three external MongoDB Atlas databases, Aiven Kafka with two topics, Aiven Valkey (Redis) for pub/sub plus Celery brokering, AWS S3 for image storage, FCM and APNs for push, and Google Cloud Build with Artifact Registry for CI/CD. The Flutter mobile app reaches the Go API through the LoadBalancer Service; the Go API reaches the Python sidecar via an internal ClusterIP Service.

- **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 $\geq$ 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 (1 s $\rightarrow$ 60 s, up to five attempts). Workers are registered in a `WORKER_REGISTRY` and toggleable via the `DISABLED_SOURCES` environment variable.

Asynchronous Kafka Producer. The Kafka producer in `kafka_worker_base.py` now uses non-blocking sends. Workers no longer block on broker acknowledgement before pulling the next post from their source iterator; delivery confirmation is handled by a future-based callback that logs failures and surfaces them to the retry layer. Throughput improves substantially during bursty source events (e.g., a high-magnitude USGS quake or a Bluesky firehose spike), where the previous synchronous producer became a per-message bottleneck.

Dynamic Worker Allocation. The original design assigned one Celery worker process to one source. The current design removes this static binding entirely. The four Celery workers in the `ingestion` queue pull tasks from whichever source is currently producing the most envelopes, so a quiet source no longer holds a worker idle while another source is flooded. This is implemented through a single `ingestion` Celery queue, fair dispatch (`worker_prefetch_multiplier=1`), and per-source rate-limit tokens that prevent any one source from monopolising the queue under sustained load.

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 is a `preprocess_post` Celery task implemented in

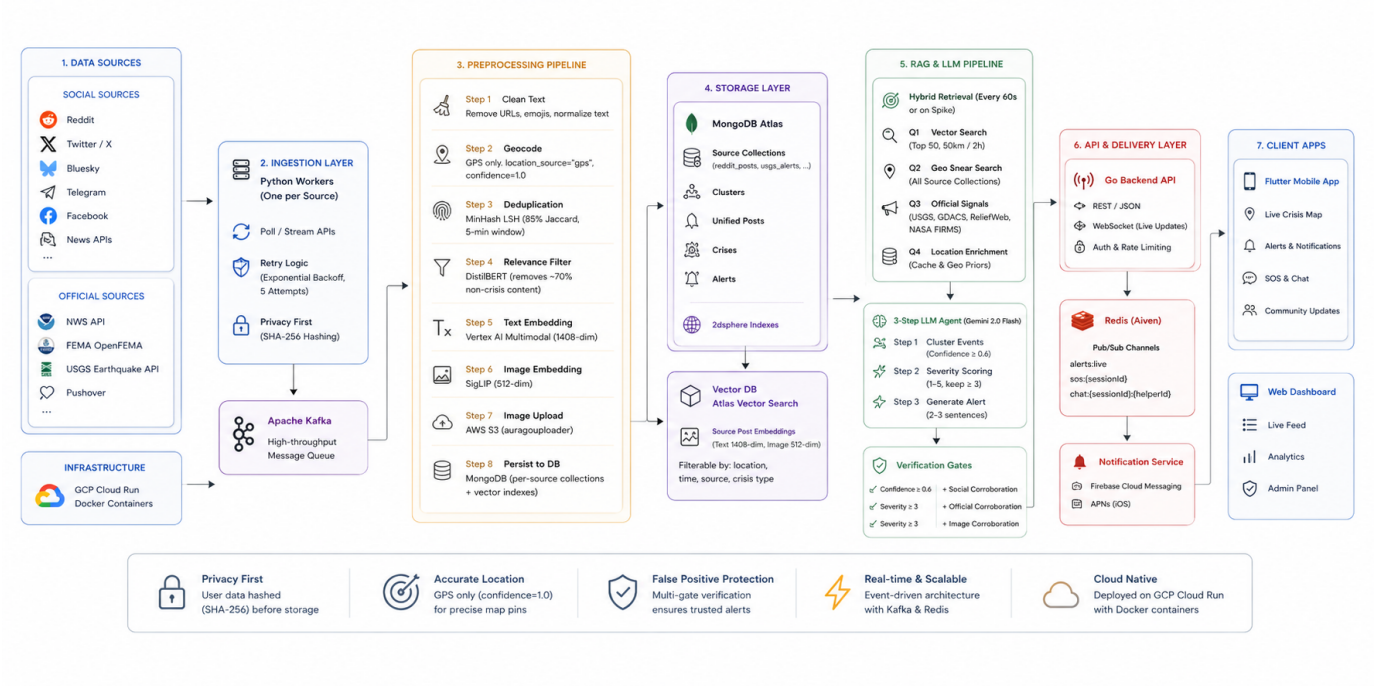


Fig. 2. Simplified Data-Flow View. Raw data flows left to right: Data Sources → Kafka → GKE cluster (preprocessing → retrieval → LLM agent) → MongoDB and Redis → Go API → Flutter mobile app. CI/CD, secrets, and external AI services attach around this spine.

`sidecar/ingestion/processor.py`. The orchestrator's Kafka consumer (`social_raw` and `official_alerts`) is now a thin dispatcher that wraps each envelope and calls `preprocess_post.delay(envelope)`; the actual processing happens in the `worker-ingestion` pod with concurrency 4. ML models (DistilBERT, SigLIP, spaCy `en_core_web_sm`) are loaded once per worker process at module import time, not per task invocation.

Each post passes through eight sequential stages:

Step 1—Clean Text. Remove URLs, Unicode emojis, normalise whitespace. Optional spaCy tokenisation via `en_core_web_sm`.

Step 2—NLP-Based Event-Location Extraction. The post text is run through spaCy NER. Place mentions (`GPE` and `LOC` entities) are resolved against a place-gazetteer cached in the `crisisecho_location` database. The location of the reported *event* (e.g., *"earthquake in Tokyo"* resolves to Tokyo regardless of the poster's geotag) is preferred over any device-attached coordinates. If NLP extraction yields a high-confidence place, `location_source="nlp"` and `location_confidence=0.9`. If NLP fails and the payload still carries a GPS pair, that is used with `location_source="gps"` and `location_confidence=0.6` (lower than before, because geotags reflect the poster, not the event). If both fail, the post is marked `location_source="unresolved"` and forwarded to the location-enrichment cascade in retrieval.

Step 3—Deduplication. MinHash Locality-Sensitive Hashing with 128 permutations. The Jaccard threshold and the sliding-window length are read from the configuration file (defaults: 0.85 Jaccard, 5-minute window). A new `RedisDedup` class shares the dedup window across all Celery workers using Redis-backed shared state, so identical viral reposts are detected even when they reach different worker processes.

Step 4—Relevance Filter. A DistilBERT cross-encoder (`cross-encoder/nli-distilroberta-base`) classifies each post as crisis-related. The threshold (default 0.6) is read from the configuration file. Official sources (USGS, GDACS, ReliefWeb, NASA FIRMS) bypass this gate.

Step 5—Text Embedding. SigLIP `google/siglip-base-patch16-224` produces a 768-dimensional text vector. Replacing the previous Vertex AI `multimodalembedding@001` text path with SigLIP places text and image vectors in the *same* 768-dim cosine space, so text-image alignment scores are now directly meaningful.

Step 6—Image Embedding. SigLIP image encoder produces a 768-dimensional normalised vector per image (up to four 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 `auragoupload er`, key format `images/{source}/{post_id}_{i}.{ext}`), and the resulting S3 URLs are stored in the SourcePost document. Original URLs are kept as a fallback if S3 upload fails.

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_type="text"); (C) image embedding documents (one per image) to source_post_embeddings (vector_type="image", image_index 0..n).

Gate Rejection Instrumentation. Every gate (dedup, relevance, location, malformed envelope) writes a single document to a new gate_metrics collection in the main database. Schema: {gate, source, post_id, drop_reason, occurred_at}. This makes drop rates queryable per source per gate per minute without redeploying. The collection is the data source for the upcoming operability dashboard described in Section XIV.

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), orchestrator.py's APScheduler dispatches a run_pipeline Celery task to the agent queue. The HybridRetriever class in sidecar/agent/retrieval.py 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 (768-dim, cosine). Returns the top 50 semantically similar posts within a 50 km bounding box and 2-hour lookback window.

Q2—Geo $near Search. Queries all eight per-source collections with MongoDB $near. Radius: 50 km. 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 → cached coordinates) and geo_priors ($near on known reference points).

Results are merged, deduplicated by post ID, and ranked by a composite score: $0.5 \times$ vector_similarity $+ 0.3 \times$ recency $+ 0.2 \times$ 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 plus 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:* cluster_confidence_floor (config default 0.6) → 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_floor (config default 3) → 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 → confidence += 0.5.

Path B—Official Corroboration: any USGS/GDACS/ReliefWeb/NASA FIRMS post nearby → confidence += 0.4.

Path C—Image Corroboration: image-text alignment ≥ image_alignment_floor (config default 0.75) AND ≥2 users → 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.

D. Configurable Confidence Thresholds

All previously hard-coded thresholds—Jaccard (0.85), relevance (0.6), cluster confidence (0.6), severity (3), image alignment (0.75)—are now resolved at boot from a single config.yaml mounted into every pod via the ConfigMap. The orchestrator handles SIGHUP to re-read the file at runtime, so an operator can retune thresholds (for example to tighten severity to 4 during a heat wave) without rolling a new image. The configuration loader exposes a singleton Thresholds object that every gate references, making operability changes a one-file edit.

E. Gate Rejection Instrumentation

Every gate in the pipeline—preprocessing dedup, preprocessing relevance, retrieval freshness, cluster confidence, severity, verification—emits a record into the gate_metrics MongoDB collection on every drop. Each record carries the gate name, source, post or cluster ID, drop reason (e.g., "jaccard_match", "below_relevance_threshold", "cluster_below_confidence", "severity_below_floor"), and an ISO timestamp. This enables backend queries such as "what fraction of Twitter posts are dropped by relevance per hour?" and is the data source for the operability dashboard described in Section XIV.

F. Entity Hierarchy

The entity hierarchy provides a complete audit trail: **SourcePost** (normalised raw post in per-source collection) → **Cluster** (internal LLM grouping, never exposed to the frontend) → **UnifiedPost** (LLM-synthesised 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.

TABLE I
Go API Domain Modules

Module	Purpose
crisis	Verified crisis events (map dots)
unifiedpost	LLM-synthesised 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 60 s)
ingest	Kafka consumer (logging only)
responder	Official responder profiles
location	Saved user locations

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).

A. gRPC Sidecar Interface

In addition to the legacy HTTP path, the Go API now talks to the Python sidecar over gRPC for low-latency internal calls. The contract lives in `sidecar/proto/crisisecho.proto` and exposes three RPCs:

- `TriggerPipeline(lat, lng) -> {status, lat, lng}`: dispatches a `run_pipeline` Celery task fire-and-forget. Used by the Go API's 60-second warm-up scheduler.
- `RunQuery(text, lat, lng) -> {digest, clusters_json[]}`: synchronously runs the LLM digest chain on active clusters near `(lat, lng)` and returns a natural-language answer. Used by `POST /api/query`.
- `Health() -> {ok, scheduler_running}`: liveness probe used as a deeper health check than the FastAPI `/health`.

The proto stubs are generated at Docker build time via `grpc_tools.protoc` and bundled into the sidecar image. The Go API uses generated Go bindings from the same `.proto` file.

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.

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 is assigned a random `_conn_id`, and messages with a 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. The wire format prepends a random 12-byte nonce to the ciphertext and base64-encodes the result for MongoDB storage. The encryption key is a 32-byte value from the `SOS_ENCRYPTION_KEY` environment variable (mounted from the `crisisecho-secrets` Kubernetes Secret). 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-coloured 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`), the new `gate_metrics` collection populated by gate-rejection instrumentation, 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` (768-dim, cosine) and `image_vector_index` (768-dim, cosine). Each document stores `post_id`, `source`, `vector`, `vector_type` (`"text"` or `"image"`), `location` (GeoJSON), `timestamp`, `crisis_type`, and optional `image_index`. Because both modalities live in the same SigLIP space, image-text alignment is computed directly via cosine similarity over the same index family.

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; populated and consulted by the NLP location-extraction step).

Redis (Aiven Valkey). Serves four roles: (1) pub/sub channels for real-time alerts, SOS location relay, and chat (`alerts:live`, `sos:{sessionId}`, `chat:{sessionId}:{helperId}`); (2) Celery task broker and result backend for the `ingestion` and `agent` queues; (3) shared MinHash-LSH dedup state across all ingestion workers; (4) durable termination key `sos_resolved:{sessionId}` (TTL 1 hour) used as the SOS WebSocket fallback signal.

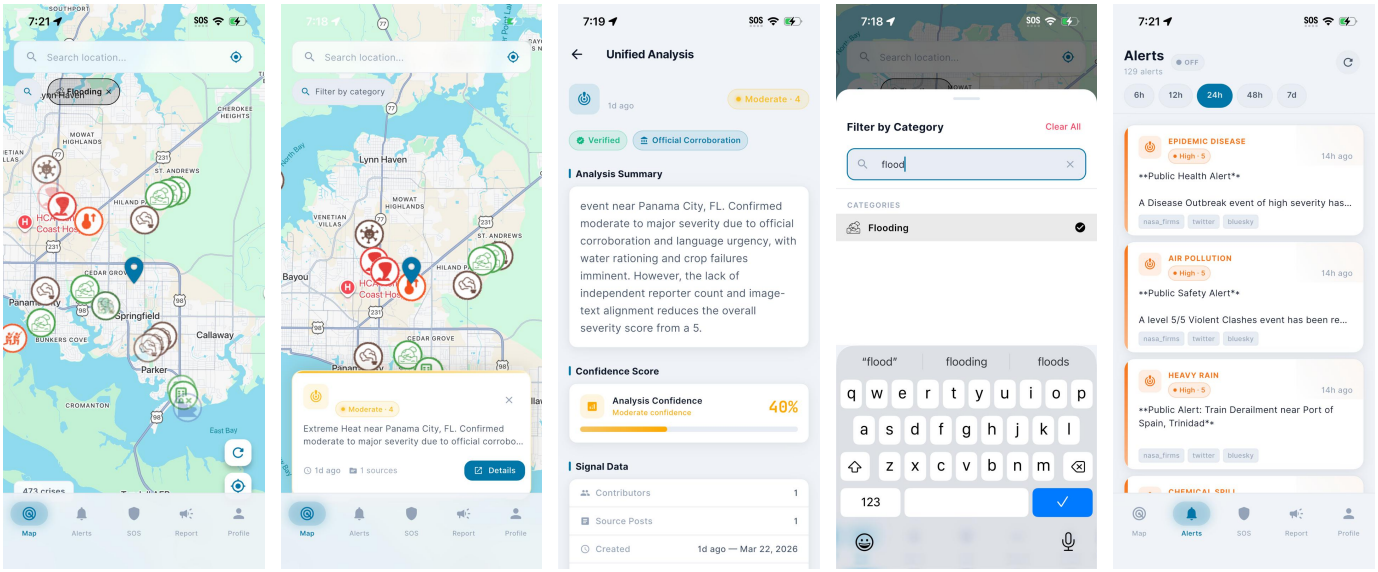
AWS S3. Stores source-post images in bucket `auragouploader`. The Go API mediates all S3 access, keeping AWS credentials in one place.

XI. Key Design Decisions

Two-Service Split. Go is optimised 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).

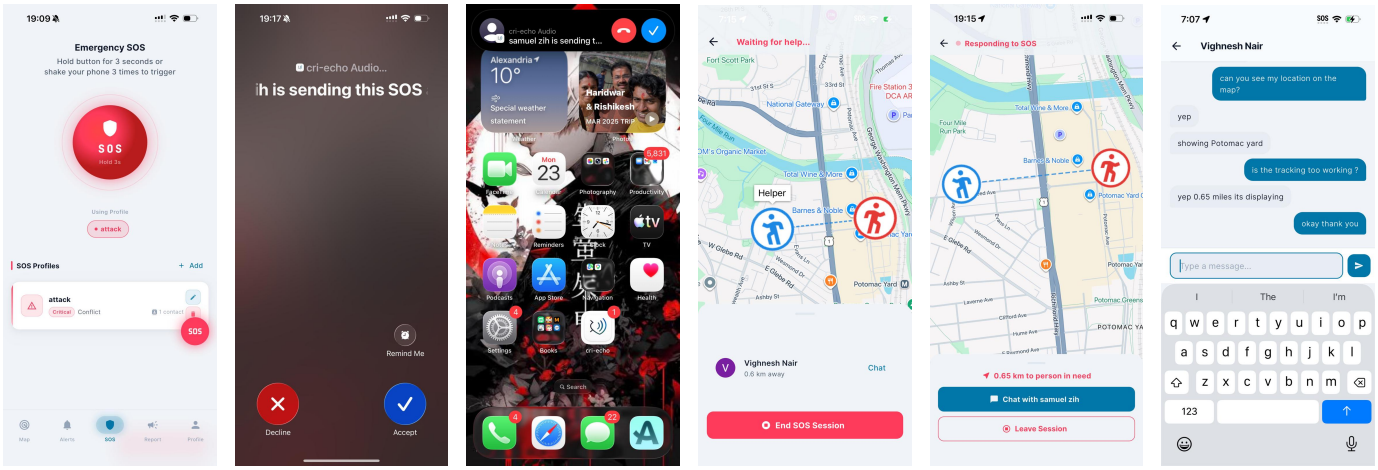
Three Databases. Vector search indexes and location lookups have different access patterns and scaling needs than operational CRUD. Separation prevents a large vector index from impacting user-facing query performance and makes it easier to swap the vector store (e.g., replace MongoDB with Pinecone) 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



(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-coloured 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.



(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) Minimised 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.

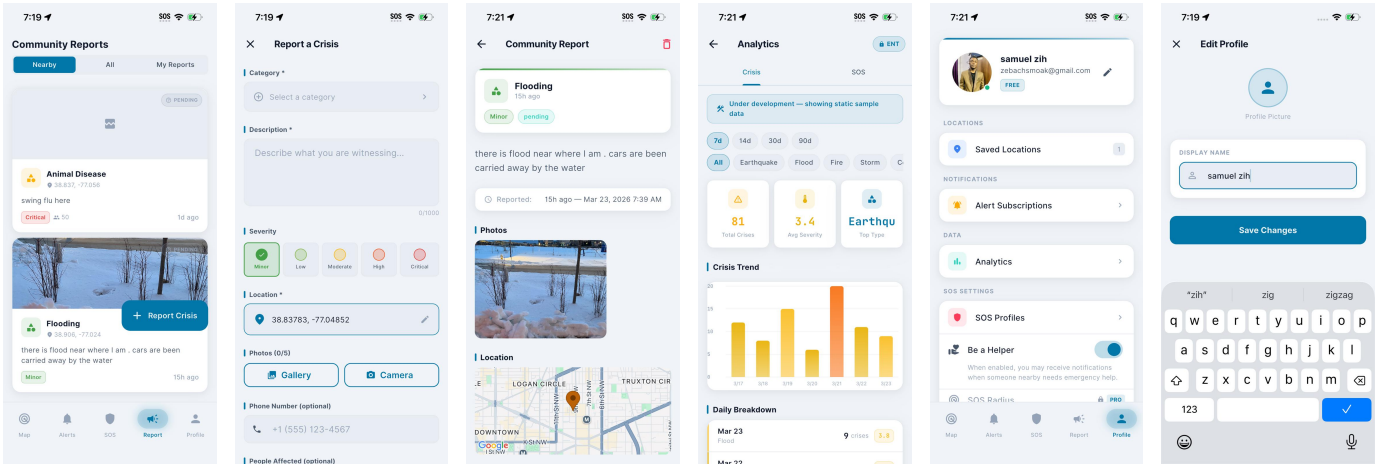
semantic understanding can separate events by meaning, not just proximity.

NLP Event-Location over Geotags. Geotags reflect where the *poster* is, not where the *event* is. A user in San Francisco posting about a Tokyo earthquake should not place a map dot in San Francisco. NLP-based extraction reads the post text and resolves event location independent of the device location, making the map represent ground-truth events rather than user distribution.

Configurable Thresholds. Hard-coded thresholds make experimentation expensive: every retune is a redeploy. Externalising Jaccard, relevance, cluster confidence, severity, and image-alignment cutoffs to a single config file lets an operator tighten the map during low-information periods (e.g., "severity ≥ 4 only during the night-time low-volume window") without rolling a new image.

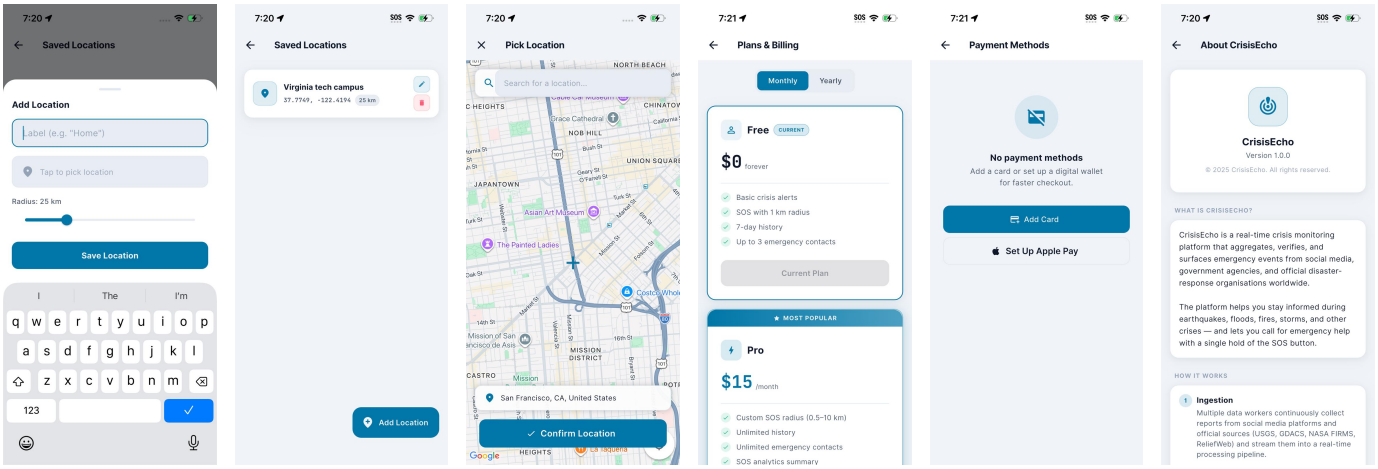
Gate Rejection Instrumentation. Every drop is a signal. Logging is ephemeral; the `gate_metrics` collection makes drop rates queryable per source per gate per minute. When a future operator notices the map is too sparse, they can query `gate_metrics` to find which gate is over-aggressive.

SigLIP Unified Embedding Space. Using SigLIP for both



(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.

text and image (768-dim) places both modalities in one cosine space. Image-text alignment becomes a direct cosine on the same index family rather than a cross-space similarity that is hard to interpret. This simplifies the Path-C verification gate and removes the previous Vertex AI text dependency.

Celery Preprocessing Queue. A daemon thread serialises preprocessing through a single worker, which becomes a hard bottleneck under bursty sources. A dedicated Celery `ingestion` queue with concurrency 4 removes that bottleneck. Worker prefetch is set to 1 (fair dispatch) so a heavy SigLIP-bound task does not starve a fast text-only task.

Dynamic Worker Allocation. Static one-source-one-worker assignment leaves capacity unused when sources are uneven. A single shared queue with rate-limit tokens lets the worker pool absorb whichever source is producing the most envelopes at any given moment.

GPS-Only Location Rule on UnifiedPost Centroid. Even with NLP extraction, the UnifiedPost centroid still uses only SourcePosts where `location_source="gps"` for legacy ground-truth fidelity, falling back to NLP-resolved locations and finally to trigger coordinates. This protects against degraded place-name extraction from noisy text.

Verification Gates. The triple gate (cluster 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.

GKE Autopilot over Cloud Run. Cloud Run's per-request scaling model is a poor fit for headless Celery workers that run continuously and pull from Redis. GKE Autopilot manages node provisioning and security patches automatically while letting us run four cooperating Deployments (api, sidecar, two Celery workers) in one cluster with shared Secrets and ConfigMap. The startup probe with a five-minute grace period gives the sidecar pod enough time to load DistilBERT and SigLIP weights.

XII. Deployment Architecture

Production—GKE Autopilot. The system runs in the `autopilot-cluster-1-crisisecho` cluster on Google Cloud Platform project `aurago`, region `us-east4`. GKE Autopilot manages node provisioning, scaling, and security patching automatically.

Deployments (4 pods). Table II summarises each Deployment.

TABLE II
Kubernetes Deployments

Deployment	Configuration
crisisecho-api	Go binary, port 8080. Liveness + readiness probes on /health. Requests 250m CPU / 256Mi mem; limits 1000m / 512Mi. Mounts /app/certs (TLS) and /app/config (Firebase).
crisisecho-sidecar	Python FastAPI on 8081, gRPC on 8082. Startup probe with 5-minute grace (loads DistilBERT + SigLIP at boot). Requests 500m / 512Mi; limits 2000m / 2Gi. Mounts /app/certs.
crisisecho-worker-ingestion	Same image, command `celery -A celery_app worker -Q ingestion -concurrency=4`. Headless. Requests 500m / 512Mi; limits 2000m / 2Gi.
crisisecho-worker-agent	Same image, command `celery -A celery_app worker -Q agent -concurrency=2`. Headless. Requests 250m / 256Mi; limits 1000m / 1Gi.

Services (2). `crisisecho-api` is a *LoadBalancer* Service exposing port 80 → 8080 to external traffic from the Flutter app and the public internet. `crisisecho-sidecar` is a *ClusterIP* Service exposing port 8081 internally; the Go API reaches it via `PYTHON_SIDECAR_URL=http://crisisecho-sidecar:8081`. The two worker Deployments have no Service—they are headless Celery consumers that pull tasks from the Redis-backed queues.

Configuration. A single `crisisecho-config` ConfigMap holds non-sensitive environment variables (database names, broker endpoint, default coordinates, `DISABLED_SOURCES`, S3 bucket, APNs topic, SOS thresholds, threshold YAML path). Three Secrets hold sensitive data: `crisisecho-secrets` (MongoDB URIs, Redis URL, JWT, encryption keys, AI keys, AWS keys, ingestion API keys, Stripe keys, APNs password); `crisisecho-certs` (Kafka and Redis TLS `ca.pem`/`service.cert`/`service.key` plus APNs VoIP `Certificates.p12` mounted at `/app/certs`); and `crisisecho-firebase` (Firebase admin JSON mounted at `/app/config`). All four Deployments load env via `envFrom` on the ConfigMap and the secret.

Inter-Service Communication. The Flutter app reaches the LoadBalancer Service. The Go API reaches the sidecar via the ClusterIP Service. Both worker Deployments and the orchestrator pull from Aiven Redis as a Celery broker. All four pods reach MongoDB Atlas, Aiven Kafka, and Aiven Redis as external managed services over TLS.

Manifests. The `kubernetes/` directory contains `configmap.yaml`, `secrets-template.yaml`, `api-deployment.yaml`, `api-service.yaml`, `sidecar-deployment.yaml`, `sidecar-service.yaml`, `worker-ingestion-deployment.yaml`, and `worker-agent-deployment.yaml`.

XIII. CI/CD Pipeline

Trigger. Google Cloud Build runs on every push to `main`. Machine type: `E2_HIGHCPU_8`.

8-Step Pipeline.

1) Build the Go API image (`Dockerfile`) tagged `crisisecho-api:$SHORT_SHA` and `:latest`.
2) In parallel: build the Python sidecar image (`sidecar/Dockerfile`) tagged `crisisecho-sidecar:$SHORT_SHA` and `:latest`.
3) Push the Go API image to Artifact Registry `us-east4-docker.pkg.dev/aurago/crisisecho/`.
4) In parallel: push the Python sidecar image.
5) `kubectl set image deployment/crisisecho-api api=crisisecho-api:$SHORT_SHA`.
6) `kubectl set image deployment/crisisecho-sidecar sidecar=crisisecho-sidecar:$SHORT_SHA`.
7) `kubectl set image deployment/crisisecho-worker-ingestion worker-ingestion=crisisecho-sidecar:$SHORT_SHA`
8) `kubectl set image deployment/crisisecho-worker-agent worker-agent=crisisecho-sidecar:$SHORT_SHA`.

Deployment Strategy. Rolling update via `kubectl set image` (the default Kubernetes strategy). Each deployment is tagged with the commit SHA for traceability. The sidecar image is reused for both worker Deployments—only the container `command` differs.

Container Images.

- *Go API*: two-stage build. Stage 1 (`golang:1.25-alpine`) compiles a static binary with `CGO_ENABLED=0`, `-ldflags '-s -w'`,

`-tags netgo`. Stage 2 (`debian:bookworm-slim`) copies the binary, installs `ca-certificates` and `curl` (for probes), runs as non-root, exposes 8080.

- *Python sidecar*: single-stage `python:3.12-slim`. Installs LangChain, Gemini and Groq SDKs, sentence-transformers, kafka-python, pymongo + motor, transformers + torch, spaCy with the `en_core_web_sm` model, datasketch, boto3, APScheduler, FastAPI + uvicorn, PRAW, atproto, feedparser, redis, celery[redis], grpcio + grpcio-tools, geopy, and pillow. Pre-warms DistilBERT and SigLIP. Generates the gRPC stubs from `crisisecho.proto` during build. Runs as non-root, exposes 8081 (HTTP) and 8082 (gRPC). Default `CMD`: `python orchestrator.py`.

XIV. Observability and Operability

The system exposes three main observability hooks:

Health Probes. The Go API and the Python sidecar each expose `GET /health`. The sidecar's check verifies APScheduler is running and that MongoDB and Redis are reachable. The sidecar Deployment uses a startup probe with a five-minute grace period to accommodate ML model loading at boot.

Gate Metrics. The `gate_metrics` collection (introduced in this release) records every drop event with the gate name, the source, the post or cluster ID, the drop reason, and an ISO timestamp. Operators can query `gate_metrics` directly today; a future dashboard will visualise drop rates per source per gate per minute. Sample queries:

- "Which source contributes the most relevance-filter drops?"
- "What fraction of clusters are dropped at the severity gate per hour?"
- "Did the Jaccard threshold change cause a spike in dedup drops?"

Celery Task Metrics. Worker prefetch is fixed at 1 (fair dispatch) so heavy SigLIP-bound tasks do not starve fast text-only tasks. Task retries use exponential backoff (`2 ** retry` seconds) up to three attempts before being dead-lettered.

Sidecar Warm-Up. The Go API's `rag` domain pings `POST /internal/pipeline/trigger` every 60 seconds even when no map view is open, ensuring the sidecar's APScheduler stays warm and that ML models do not unload from CPU caches.

XV. Current Status and Remaining Work

Completed:

- Go API: 20 domain modules, 3 middleware, 3 WebSocket endpoints, Firebase Auth, FCM, APNs VoIP push, gRPC client to the sidecar.
- Python Sidecar: 8 ingestion workers, 8-step preprocessor (now Celery-backed with NLP location extraction), hybrid retriever, 3-step LLM agent, image and crisis verifiers, gRPC server, FastAPI, volume-spike detection, configuration-driven thresholds, gate-rejection metrics, dynamic worker allocation, asynchronous Kafka producer.
- Databases: 3 MongoDB Atlas databases with 30+ collections (including the new `gate_metrics`), 2dsphere and Atlas Vector Search indexes, Redis pub/sub, Celery broker, and durable session keys.
- Frontend: Flutter app with 24+ screens (crisis map, SOS, community reports, analytics, billing, profiles, locations).
- Infrastructure: GKE Autopilot in `us-east4` with four Deployments, two Services, ConfigMap, three Secrets; Cloud Build CI/CD on every push to main with rolling updates; Artifact Registry image hosting; Aiven Kafka, Aiven Valkey, MongoDB Atlas managed services; AWS S3 media storage; Firebase configuration.
- Mobile app distribution: the Flutter app has been submitted to both the Apple App Store and Google Play and is currently awaiting review on both stores.

Remaining:

1) Receive store-review approval and complete the public production rollout on Apple App Store and Google Play.
2) Build the operability dashboard on top of the `gate_metrics` collection.
3) Hand off API keys for the production-tier inference budget.

XVI. Task Assignment

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

TABLE III
Task Assignment

Member	Responsibilities
S. Zih	Backend: Go Fiber API (20 modules), Python sidecar (LLM pipeline, preprocessing, 8 ingestion workers, Celery, gRPC, configuration loader, gate-rejection metrics), MongoDB (3 databases, 30+ collections), Kafka (asynchronous producer), Redis (pub/sub + Celery broker + dedup state), WebSocket, APNs VoIP, FCM, Docker; Frontend: Flutter (24+ screens), Google Maps, SOS tracking, encrypted chat, Stripe billing; AI/ML: DistilBERT relevance, SigLIP unified text+image embeddings, NLP-based event-location extraction, LangChain agent, verification; Infrastructure: GKE Autopilot, Cloud Build CI/CD, Artifact Registry, Kubernetes manifests, Secrets and ConfigMap; Report: all sections.

XVII. Schedule

Table IV compares the originally planned schedule with actual progress.

XVIII. Future Work

While CrisisEcho is fully implemented, deployed on GKE Autopilot, and submitted to both mobile app stores, several extensions will move the platform from real-time *detection* to closed-loop *response*. The following four items form the post-launch roadmap.

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 and 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 and 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	Migrated deployment from Cloud Run to GKE Autopilot; built Cloud Build CI/CD; restructured sidecar; added NLP location extraction, async Kafka, dynamic worker allocation, configurable thresholds, gate-rejection metrics; upgraded embeddings to unified SigLIP.

A. Emergency Services Integration

A direct API connection to public-safety dispatch centres (e.g., 911 Public Safety Answering Points in the United States, equivalent regional dispatch endpoints elsewhere) will allow verified crises to auto-generate official emergency tickets the moment the verification gates close. This shifts CrisisEcho from a passive sensor that pushes alerts to end-users into an active upstream input to the official emergency response chain, shortening the time between social-media signal and dispatcher awareness. Integration will respect dispatch-side rate limits, require severity-floor agreements per jurisdiction, and include a human-review override for political or contested events.

B. Evacuation Route Planning

On the user-facing side, the next step is integration with mapping APIs (Google Maps Directions, Mapbox Directions) to compute and suggest *safe* routes away from active crises in real time. The crisis polygon (centroid plus severity-scaled radius) becomes a routing avoid-zone, and the Flutter app overlays the suggested route directly on the existing crisis map. Combined with the SOS system, an accepting helper can be routed around active crises rather than into them.

C. Historical Crisis Archive

Every verified Crisis already lives in the `crises` collection, every UnifiedPost in `unified_posts`, and every contributing SourcePost in its per-source collection, with full audit trails. A historical-archive layer will expose this data through a searchable, public, research-grade interface: filter by category, severity, region, date range, and corroboration path. The intended audiences are researchers, journalists, policy analysts, and accountability bodies. Privacy is preserved by the existing SHA-256 user hashing performed pre-Kafka and by the strict GPS/NLP-only location integrity rule already enforced upstream.

D. IoT and Emergency Robotics Integration

Looking further out, the SOS broadcast channel is well positioned to become the input layer for autonomous emergency systems as robotics infrastructure matures. The same proximity-based wave-broadcast that today notifies opted-in human helpers can, with a targeted policy check, also notify a fleet of nearby autonomous responders—rescue drones, robotic AEDs, autonomous medical-supply delivery, or building-level emergency robots. CrisisEcho's existing entity hierarchy (SourcePost $\rightarrow$ Cluster $\rightarrow$ UnifiedPost $\rightarrow$ Crisis $\rightarrow$ Alert) already provides the structured, verified, geolocated event needed by such systems. The extension is mostly a matter of an additional consumer of the `alerts:live` Redis channel and a per-device authorisation policy.

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.