Senior Design Capstone

CS 4485 - Professor Sarac - Team 2
Fall 2025 - “Risk Radar”

Medha Kotra, Susan Zhang, Oviya Selvakumar,
Jaszmine DeFranco, Sanjana Kotha, Fatima Carmona

Table of Contents

Table of Contents 2
Chapter 1: Business Context 4
1.1 Problem Statement 4
1.2 Project Scope and Objectives 4
1.3 Key Project Risks and Mitigation Strategies 4
1.4 Developer Information 6
Chapter 2: Solution Design and System Architecture 7
2.1 End-to-End System Design 7
2.2 Architecture Overview, Solution Design, and Tech Stack 7
2.3 Project Framework and Technical Assumptions 9
2.4 Team Responsibilities, Workflow, and Tools Used 10
2.5 Batch Processing Design and Rationale 11
2.6 Integration Strategy 12
2.7 Alternative Architectural Designs Considered 13
2.8 Summary of Project Outcome 14
Chapter 3: Data Strategy and Pipeline Development 15
3.1 Data Sources and Acquisition 15
3.2 Data Labeling and Taxonomy Development 16
3.3 Dataset Composition and Distribution 16
3.4 Data Preprocessing Pipeline 17
3.5 Evolving Classification System and Feature Pipeline 18
3.6 Production Data Flow 18
Chapter 4: Model Development and Benchmarking 20
4.1 Experimental Design 20
4.2 Comprehensive Model Evaluation 21
4.3 Benchmarking Results 22
4.4 Experimental Artifacts and Reproducibility 25
Chapter 5: System Architecture and Implementation 25
5.1 System Architecture Overview 25
5.2 Database Design and Schema 26
5.3 Real-time Processing Pipeline 30
5.4 Edge Functions and Cron Jobs 30
5.5 Front-end and Back-end Integration 31
5.6 Back-end Framework and APIs 32
5.7 Deployment Architecture 33
Chapter 6: Performance Analysis and Results 34
6.1 Production Pipeline Performance 34
6.2 Disaster Type Distribution Analysis 34
6.3 Model Confidence and Severity Analysis 35

6.4 Geographic and Temporal Patterns 35
6.5 System Limitations and Bottlenecks 36
6.6 Rate Limiting Analysis and Adaptive Optimization 36
Chapter 7: User Interface and Dashboard 38
7.1 Front-end Component Architecture 38
7.1.1 Front-end Tools 38
7.1.2 Front-end Components 38
Breaking Numbers 38
Heat Map, Markers, and Popup 39
Top Posts Widget 40
Help Requests Widget 41
Data Table 42
Analysis Charts 43
Resources 44
Appendix 45
A.1 UML Diagrams 45
A.2 Complete SQL Schema and Queries 45
A.3 Experimental Artifacts 45
A.4 API Configuration Details 45
A.5 Project Management Artifacts 45
A.6 Alternative Designs 46
A.7 Datasets 47

Chapter 1: Business Context

1.1 Problem Statement

Finding a singular source of information about an ongoing event can be difficult with the various content and forms of data available on the internet. This issue is only exacerbated during times of crises and disasters as accurate and up-to-date information is necessary. While there are articles and blurbs being pumped out constantly, these must all be found separately and compiled on one's own to gain the full idea of the current happenings. With the rise of social media, there is yet another stream of information. Unlike traditional media, social media is accessible by anyone and allows for instant updates, which increases the amount of information and misinformation available. Due to the high volume of posts on social media platforms, time-sensitive details are easily overlooked. There is no singular source that acts as a central hub to provide accurate information and updates to aid with fast relief efforts and awareness.

1.2 Project Scope and Objectives

Our main goal is to create a centralized dashboard that presents information about current disasters with information and statistics such as severity levels. We aim to:

- Using an LLM to Analyze Data: Use the Bluesky API to pull recent posts from Bluesky social media that will be processed through an LLM to identify disasters and extract their key details.
- Compile and Present Current Data: Utilize the LLM to label the extracted crisis details, such as disaster type, whether help is being requested, and location, allowing them to be properly presented with the necessary context and analysis.
- Create a Dashboard: Design an informational dashboard with a user-friendly interface that displays accurate updates, notifications, and information via various data visualizations such as graphs, tables, maps, and charts.
- Increase Accessibility and Efficiency: Provide a platform that acts as a centralized source of all necessary information, cutting down on research time and allowing for faster reaction time from action groups such as first responders.

1.3 Key Project Risks and Mitigation Strategies

Risk	**Mitigation Strategies**
Data Reliability and Accuracy (High): As the output shown on our system dashboard is directly dependent on Bluesky data, misinformation, rumors, or incomplete details	**Mitigation:** We implemented multi-layered validation including confidence scoring thresholds (posts below 0.7 confidence are flagged for review) and source credibility assessment. Our production system

could be displayed if users accidentally or intentionally spread false disaster information.	automatically filters low-confidence classifications and provides transparency about confidence levels to end-users.
LLM Model Bias and Misclassification (High): The LLM could misinterpret language, overlook emerging crises, generate biased outputs, or produce factual inaccuracies (hallucinations) regardless of the model used.	**Mitigation:** We conducted rigorous benchmarking across 16 models, selected Qwen-32B for its balanced performance, and implemented comprehensive prompt engineering with 25 iterative refinements. Continuous monitoring through our confidence scoring system and regular validation against ground truth data ensures ongoing accuracy.
Ethical and Privacy Concerns (High): Public Bluesky posts may contain sensitive personal information, names, or precise locations that could endanger individuals if improperly handled.	**Mitigation:** We maintain transparency by displaying Bluesky handles as they appear in public posts, recognizing this information is already publicly available on the Bluesky platform. For location data, we prioritize accuracy in disaster reporting by displaying exact coordinates when available, as these typically represent disaster locations rather than personal residences. Our system focuses on extracting and displaying disaster-relevant geographic information to ensure the general public receives precise location data for situational awareness and personal safety decisions, while all data handling complies with Bluesky's public data usage policies.
API Rate Limiting and Service Availability (High): Groq API's rate limits (60 RPM, 1,000 RPD) and potential service outages could disrupt real-time classification, causing backlog buildup and delayed disaster alerts.	**Mitigation:** We implemented multi-API key rotation, intelligent queue management with exponential backoff, and comprehensive monitoring to detect and respond to rate limiting issues. Our system includes fallback mechanisms and graceful degradation to maintain partial functionality during API disruptions.
Data Pipeline Scalability (Medium): Sudden surge in Bluesky post volume during major disasters could overwhelm our processing	**Mitigation:** We designed batch processing with configurable parameters, implemented queue depth monitoring, and built auto-scaling capabilities that can adjust

pipeline, causing significant delays in classification and alert delivery.	processing rates based on system load and API availability.
User Adoption and Usability (Medium): Relief workers and emergency responders might find the system unintuitive or visually cluttered during high-stress operational moments.	**Mitigation:** We employed user-centered design principles with iterative Figma prototyping and usability testing. The final dashboard features prioritized information hierarchy, clear visual distinctions between severity levels, and intuitive navigation patterns validated through stakeholder feedback sessions.
Model Adaptation to Emerging Crises (Medium): The LLM may struggle with classifying novel disaster types or emerging crisis patterns not well-represented in training data, leading to missed or misclassified events.	**Mitigation:** We established continuous model evaluation, implemented human-in-the-loop verification for low-confidence classifications, and built flexible prompt engineering workflows to quickly adapt to new disaster scenarios.

1.4 Developer Information

Medha - Team Lead and back-end
As the team lead, I am responsible for communicating with our faculty advisor, documenting team progress, and ensuring that we stay on track to meet our project objectives. I also worked on the back-end, where my main responsibilities included building and managing the database and working on Bluesky API integration.

Susan - Full stack and Front-end
As a software engineer, I was responsible for developing the user interface and collaborating with the data scientists and the back-end engineers to ensure that the LLM outputs would be meaningfully and informatively displayed. I was also responsible for creating wireframes and ideating features for our dashboard.

Oviya - Scrum Master and Front-end
As the SCRUM master, I was responsible for managing and maintaining an agile workflow through Jira story tracking and weekly reports. On the front-end team, I helped create interactive features and components to properly display information outputted by the API and LLM. I also integrated principles to make sure users could easily and efficiently navigate the website.

Jaszmine - Data Science
As a data scientist, I collaborated with Sanjana across the full model development lifecycle. My responsibilities included data set preparation, model benchmarking, and implementing the chosen LLM into our production pipeline. This involved developing and optimizing Supabase Edge Functions to handle LLM API calls, integrating rate limiting and error mitigation strategies

to ensure system reliability, and refining the model's classification logic for accuracy and performance. I also worked with the Groq API to leverage high-speed inference, ensuring our classification procedure was both fast and scalable within the application's architecture.

Sanjana - Data Science
As one of the data scientists, I was responsible for helping curate the benchmarking dataset, perform benchmarking to choose the optimal LLM and coordinate with front-end and back-end teams to integrate the LLM properly into our pipeline. This involved working within Supabase to create Edge Functions that properly interact with the LLM API endpoint as well as with the tables within Supabase to ensure the data was properly ready for the front-end team. I worked on this with Jaszmine.

Fatima - Full Stack and Backened
As one of the backend engineers for the team, I was responsible for building the ingestion and processing pipeline that connects Jetstream, Supabase, and the Groq LLM. This included developing the scheduled scripts that pull Bluesky data and ensure that structured results are reliably stored for the frontend. I also handled the deployment of backend components, environment configuration, and error-recovery mechanisms to maintain a stable pipeline.

Chapter 2: Solution Design and System Architecture

2.1 End-to-End System Design

Our crisis detection system follows a modular, serverless architecture designed for reliable, near-real-time processing. The system follows a streamlined, batch processing pipeline that transforms raw social media posts into actionable disaster intelligence through four integrated layers: **data ingestion**, **AI processing**, **data storage**, and **presentation**.

This end-to-end design, where ownership is split between the back-end (ingestion/orchestration), Data Science (model/classification), and front-end (presentation) teams, ensures minimal latency. The complete flow, from post creation to dashboard visualization, typically completes within minutes, providing the timely, reliable situational awareness required for geospatially-aware disaster analysis.

2.2 Architecture Overview, Solution Design, and Tech Stack

The system is built on a serverless, microservices-based architecture hosted on Supabase, chosen for its development velocity, cost-effectiveness at scale, and built-in real-time capabilities. As illustrated in Figure 2.1, the logical design separates concerns into four distinct layers, each with specific technologies and responsibilities.

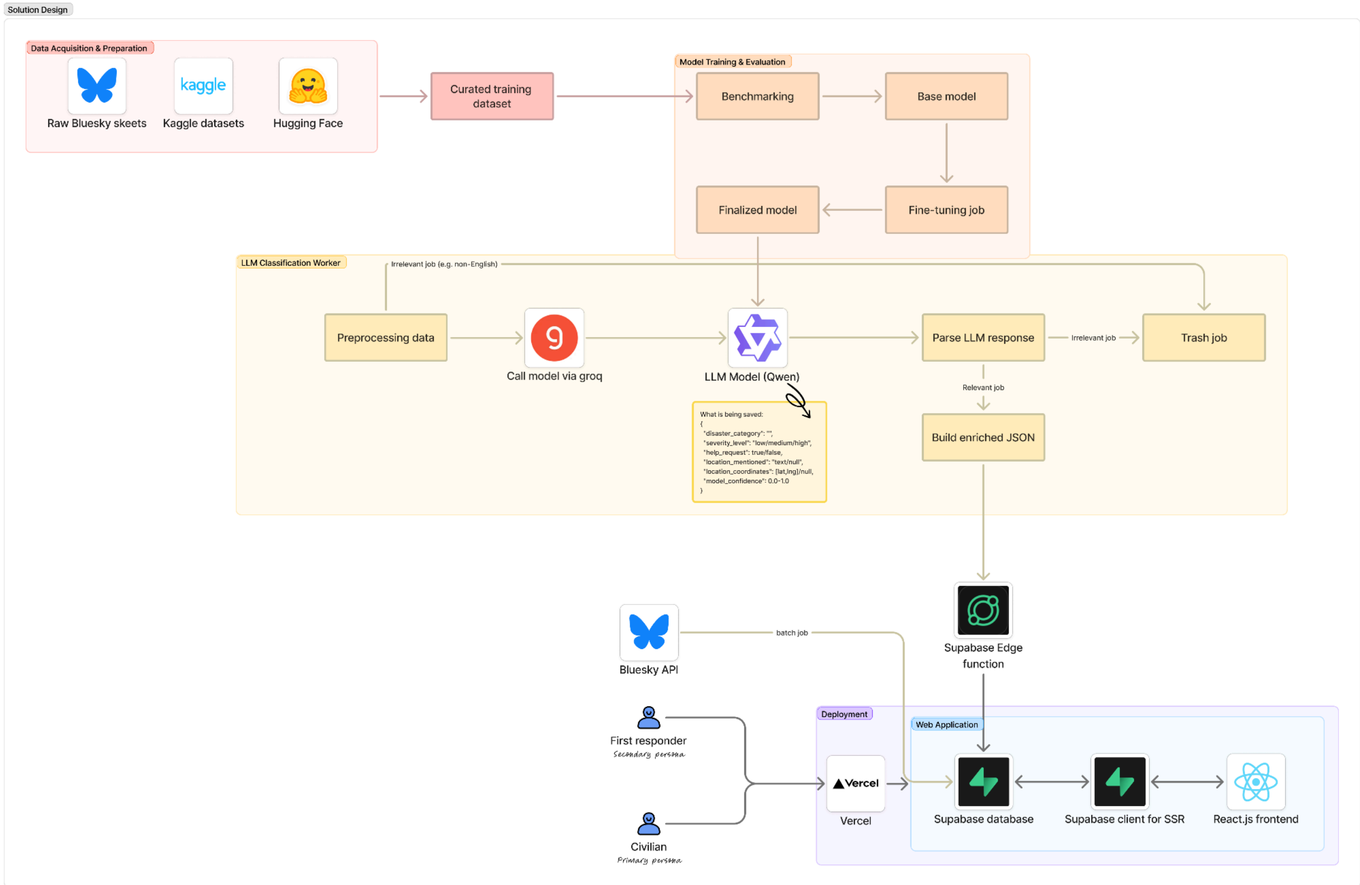


Fig. 2.1 Solution Design

Layer 1: Data Ingestion

- Technologies: Jetstream data service, AT Protocol (ATProto) API, Supabase.
- Responsibilities: This layer is responsible for connecting to the Bluesky firehose data stream, filtering posts based on configurable keywords and accounts, formatting the data into a consistent schema, and inserting it into the ingestion table in Supabase. It acts as the system's entry point.

Layer 2: Processing and Orchestration

- Technologies: Supabase Edge Functions (Node.js), PostgreSQL tables (as stateful queues) in Supabase, Groq Cloud Inference API.
- Responsibilities: This core layer manages the batch classification pipeline. Scheduled Edge Functions orchestrate the workflow: moving data between queue tables, calling the Groq API with the Qwen-3-32B model for classification, and handling errors, retries, and rate limiting. It transforms unstructured text into structured disaster intelligence.

Layer 3: Data Storage and Services

- Technologies: Supabase PostgreSQL database

- Responsibilities: This layer holds and maintains information from its initial ingestion and when it is displayed on the front-end dashboard. Row level security and an organized schema ensure that the data is easy to use.

Layer 4: Presentation and Visualization
- Technologies: React.js with Vite build tool, Shadcn/ui component library, Supabase JS client, Leaflet (maps), Recharts (graphs).
- Responsibilities: This layer consumes the processed data via the Supabase client and presents it through an interactive dashboard. Key components include a geographical heatmap, a filterable data table, and analytical charts.

Key Architectural Decisions:
- Serverless Edge Functions: Eliminates infrastructure management, as opposed to a back-end server, and scales automatically with the batch load, optimizing cost.
- Cron-based Scheduling: Provides predictable, controlled execution of the processing pipeline, avoiding constant polling and managing external API rate limits effectively.
- Loose Coupling via Database State: Using database tables as the interface between services (ingestion → processing → presentation) allows each layer to operate independently, enabling parallel team development and easier debugging.
- External AI Service Integration: Leveraging Groq's inference API for model execution, rather than self-hosting, provided access to high-performance LLMs without the overhead of GPU management.

This modular architecture successfully supported parallel development by the back-end, data science, and front-end teams, with clear integration contracts defined at the database level, ensuring both system stability and development velocity.

2.3 Project Framework and Technical Assumptions

Development Framework: Our team adopted an Agile/Scrum framework, facilitated by a dedicated Scrum Master. We operated on one-week sprint cycles, utilizing Jira for story tracking and sprint management and GitHub for version control and continuous integration. This iterative approach enabled rapid feature development, regular stakeholder feedback, and adaptive planning throughout the project lifecycle.

Technical Assumptions:
The successful operation of RiskRadar is predicated on the following technical assumptions:

- <u>Data Source Assumptions</u>:
    - The Jetstream data service provides reliable, real-time access to public Bluesky posts with consistent JSON formatting.
    - The Bluesky firehose contains a sufficient volume of posts related to disaster events for meaningful analysis.

    - A post's timestamp (indexedAt) and any embedded location metadata or text are accurate and relevant to the disaster event it describes, enabling correct temporal and geospatial analysis.
    - The AT Protocol API endpoints and data schemas remain stable for the duration of the system's operation.
- Infrastructure and Model Assumptions:
    - The Groq Cloud Inference API maintains consistent latency and availability, with classification request response times under 500ms.
    - The Qwen-3-32B model, as served by Groq, achieves a benchmarked accuracy of >85% for our specific multi-label disaster classification task.
    - Supabase infrastructure (Database, Edge Functions, Auth) provides 99.9% operational uptime and fulfills its service level agreements.
    - Client network connectivity is sufficient to support real-time data subscriptions and dashboard interactivity.

Theoretical Foundations: The system is built upon established computer science principles. It applies **Natural Language Processing (NLP)** and specifically the **transformer architecture** for contextual understanding and classification of disaster-related text. The pipeline implements **event-driven** and **batch processing** patterns for scalable data orchestration. Finally, the presentation layer employs principles of **geospatial information systems (GIS)** and **information visualization** to translate processed data into actionable situational awareness.

2.4 Team Responsibilities, Workflow, and Tools Used

Front-end Components:
- Heat map for severity-based prioritization, with filtering and search capabilities.
- Data table with filtering and search capabilities.
- Statistical charts and analytics displays.
- Visually present key performance indicators in a concise format.
- Provide resources for users to boost their understanding of natural disasters.

Back-end Services (BE Team):
- Integration with Bluesky through AT Protocol API.
- Utilizing Jetstream data services to access Bluesky data.
- Database schema design and optimization.
- Database maintenance.

Data Science Pipeline (DS Team):
- LLM evaluation and model selection.
- Disaster taxonomy development and refinement.
- Prompt engineering and optimization.
- Performance metrics analysis and validation.
- Classification accuracy monitoring.
- Edge function development for classification pipeline.

- Cron job scheduling and execution management.

Integration Points: The front-end React.js components connect to back-end services via Supabase client libraries, while the data science team provides the classification model that back-end functions call via Groq API. All teams collaborate on data schema design to ensure seamless data flow.

Team Workflow: The entire team utilized Jira to record and organize tasks for each sprint. Sub-teams (front-end, back-end, data science) set internal deadlines and split up work as needed. There were weekly check-ins with our faculty advisor, whole group meetings to review progress and plan for the next week, and sub-team meetings to discuss technical details.

Front-end Workflow and Tools:
The front-end engineers met weekly to discuss and work on each weekly sprint's tasks. Additionally, the front-end team regularly communicated with the data science and back-end teams, in addition to weekly check-ins with our faculty advisor and whole group meetings. Upon receiving the categorized Bluesky posts from the **Supabase client**, the front-end processed and adequately displayed the data through a user-friendly approach. **React.js** was used for its component-based architecture, state management capabilities, and client-side rendering. Client-side rendering was especially crucial to the front-end as our selected heatmap library **Leaflet** renders client-side; the Leaflet interactive map library was chosen for its customization in UI layers, controls, and data displayed. Additionally, **Vite** was used in addition to React.js for its faster development experience: instant server start, Hot Module Replacement, and optimized production builds. For styling, **Tailwind CSS, Lucide Icon Library**, and **Shadcn UI Library** were used for their ease of integration and modern and accessible designs.

Back-end Workflow and Tools:
Like the other subteams, the back-end team communicated regularly with each other and the rest of the team to ensure effective collaboration and integration.
The database and much of the back-end is managed using **Supabase**, through its database and PostgreSQL provisions, edge functions, REST API, and Supabase client. Additionally, the back-end scripts that pull data are scheduled using **GitHub Actions**.

2.5 Batch Processing Design and Rationale

The system employs a batch processing approach with 1-minute intervals and batch sizes of 6 posts per execution. This design was selected over true real-time processing due to several key considerations:

- **Rate Limit Management**: Groq API's 60 requests-per-minute limit necessitated controlled request pacing to avoid throttling and ensure consistent service availability.
- **Cost Optimization**: Batch processing reduces overall API calls by processing multiple posts in single classification requests, maximizing throughput within tier limits.

- **Error Handling**: Batched processing allows for comprehensive error tracking and retry mechanisms without blocking the entire pipeline.
- **Resource Efficiency**: The 1-minute interval provides near real-time performance while allowing sufficient time for API responses and database operations between batches.
- **Dynamic Scheduling Control**: Our architecture provides full operational flexibility through configurable cron job scheduling for all three core processing functions. The queue-loader edge function, which moves data from input to processing queue, operates on an independent schedule from the classification-worker that processes the queue, and the recover-failed function that handles retries. This separation allows us to dynamically rebalance table loads during testing and production scenarios. For instance, if the processing queue grows excessively, we can temporarily increase the classification worker frequency without affecting data ingestion. Conversely, during API rate limit issues, we can throttle the classification worker while maintaining queue population for later processing. This granular control proved essential during our testing phase, enabling us to optimize throughput while respecting external API constraints and maintaining system stability.

2.6 Integration Strategy

Our integration strategy employs a contract-first approach where teams define clear interfaces between components. The key integration points include:

Data Contract: A unified database schema with well-defined tables (posts_input, processing_queue, extracted_info_output) that serve as integration points between components.

- The **posts_input** table acts as an integration point between Bluesky, which we interact with through Jetstream and the AT Protocol API, and the database itself.
- The **processing_queue** connects the data to the LLM since the data is pulled from this table for processing.
- The **extracted_info_output** connects the classified posts to the front-end because this data is analyzed and displayed in various front-end charts, tables, and the heatmap.

API Integration: Utilizing Supabase's RESTful API, the **extracted_info_output** table is connected to our front-end components. Supabase automatically uses PostgREST to create a RESTful API based on the table's database schema; the front-end was able to directly import and communicate through the Supabase client.

Event-Driven Processing: Webhooks trigger classification workflows, while cron jobs, scheduled through Github Actions or Supabase, ensure consistent data ingestion and periodic processing of queued data. This process reflects effective integration between Bluesky, the database, the Groq API, and the LLM.

Error Propagation: Comprehensive logging and error tables (failed_classifications table) provide visibility into integration failures and enable automated recovery.

2.7 Alternative Architectural Designs Considered

During the design phase, the team evaluated several architectural and modeling alternatives. The following options were analyzed and ultimately rejected in favor of the chosen serverless, batch-processing pipeline, based on trade-offs between cost, complexity, latency, and maintainability.

Alternative 1: Full Real-Time Streaming via WebSockets

- **Approach**: Maintain persistent WebSocket connections to Jetstream and classify posts immediately upon arrival.
- **Rejected Reasons**:
  - Real time classification would require multiple requests per minute, quickly exceeding the Groq API limits and incurring unpredictable costs.
  - Would require additional overhead as backpressure handling and continuous streaming need error recovery.
- **Trade-off:** *Latency vs. Stability* - Achieves the lowest possible latency but sacrifices rate limit stability, cost predictability, and system maintainability.

Alternative 2: Dedicated back-end Server

- **Approach**: Deploy a long-running Node.js/Express server on a cloud VM (e.g., AWS EC2) that continuously listens to the Jetstream, hosting the entire ingestion, queuing, and classification orchestration logic.
- **Rejected Reason**: This model incurs higher and less predictable infrastructure costs compared to serverless compute. It also places the operational burden of server monitoring, scaling, and patching on the development team.
- **Trade-off**: *Control vs. Overhead* - Provides full control and deployment flexibility but requires significantly higher operational overhead and cost.

Alternative 3: Alternative Hosting Platform (e.g., Render.com)

- **Approach**: Host the ingestion script and back-end logic on Render.com using background workers and scheduled jobs.
- **Rejected Reason**: To achieve the necessary concurrency and scheduling reliability, this approach would require upgrading to multiple paid service tiers. The total cost was projected to be higher than the all-included Supabase platform for a similar feature set.
- **Trade-off**: *Simplicity vs. Cost* - Offers a simpler initial setup but results in higher platform lock-in and cost at scale.

Alternative 4: Multiple Model Ensemble

- **Approach**: Improve classification accuracy by employing a voting mechanism across multiple LLMs (e.g., Qwen, Llama, Mistral) via their respective APIs.
- **Rejection Reason**: This would multiply API costs by a factor of three or more and compound latency, making it prohibitive for a production system requiring timely results.
- **Trade-off**: *Accuracy vs. Cost* - Potential for marginal accuracy improvements at the expense of 3x operational cost and complexity.

Alternative 5: On-Premise Model Deployment

- **Approach**: Self-host an open-source LLM (like Llama 3) on a dedicated GPU instance to avoid external API costs and latency.
- **Rejection Reason**: This introduces massive infrastructure overhead, including GPU procurement/management, model serving infrastructure (e.g., vLLM), and ongoing maintenance. It would severely hamper development velocity.
- **Trade-off**: *Control vs. Velocity* - Provides theoretical cost control and data privacy but sacrifices development velocity, scalability, and team agility.

Alternative 6: React Front-end Framework

- **Approach**: We initially chose to create the front-end using Next.js but converted to React for the final project.
- **Rejection Reason**: This was done to accommodate the different versions of heatmaps we tested. We experimented with Google Maps API and Mapbox GL, both of which were Next.js compatible. However, we finalized Leaflet's React library for its strong visual heat layer and dynamic features, leading us to move frameworks.

2.8 Summary of Project Outcome

This solution design delivers RiskRadar, a comprehensive crisis detection system that addresses the critical need for timely, accurate disaster intelligence. By architecting a modular, serverless pipeline around modern cloud services and a high-performance LLM, the system successfully transforms the chaotic stream of social media into structured, actionable insights for emergency responders.

Key Outcomes and Connected Design Decisions:

1. **Timely Situational Awareness**: The **batch-processing** pipeline, orchestrated by Supabase Edge Functions, ensures new disaster posts are processed and displayed on the dashboard within minutes. This design directly supports the primary user benefit of **reduced information latency**, moving from hours of manual monitoring to near real-time alerts.
2. **Actionable, Structured Intelligence**: The integration of the **Groq API with the Qwen-3-32B model** converts unstructured text into standardized classifications (type, severity, location, confidence). This core design decision enables the **enhanced decision-making** outcome, providing responders with filtered, prioritized data instead of raw social media noise.
3. **Scalable and Reliable Operation**: The choice of a **serverless, cron-scheduled architecture** decouples system components. This grants **operational stability** under load and ensures scalability, as the pipeline can handle data volume spikes without service interruption, a critical requirement for emergency scenarios.
4. **Intuitive Geospatial Understanding**: The front-end's interactive **heatmap and filterable data** table are powered by real-time Supabase subscriptions. This direct link between the storage layer's real-time capabilities and the presentation layer delivers the

outcome of improved geospatial awareness, allowing users to quickly visualize disaster scope and location.

5. **Sustainable and Maintainable Foundation**: By leveraging managed services (Supabase, Groq API) over self-hosted alternatives, the design prioritizes **development velocity and lower operational overhead**. This outcome ensures the prototype is not only functional but also provides a **strong, cost-effective foundation for future expansion**, such as integrating additional data sources or model fine-tuning.

Overall, the project results in a functional, scalable prototype that validates how purpose-built AI classification integrated with modern cloud services can streamline disaster monitoring. The architectural decisions documented throughout this chapter collectively deliver a platform that improves the speed, clarity, and reliability of crisis-related information discovery.

Chapter 3: Data Strategy and Pipeline Development

This chapter details the comprehensive data strategy for our crisis detection system, encompassing the complete data lifecycle from initial acquisition and labeling through preprocessing and pipeline development. It covers our multi-source approach to dataset creation, the evolution of our disaster taxonomy, and the data processing pipeline that supports real-time classification through the Bluesky social network.

3.1 Data Sources and Acquisition

Our crisis detection system employed a phased approach to data acquisition, utilizing different sources throughout the development lifecycle. For the initial model benchmarking phase, the data science team leveraged **existing disaster datasets** from Kaggle and HuggingFace, which provided a robust foundation of historically labeled disaster-related social media posts. These established datasets offered the advantage of verified disaster annotations and comprehensive coverage across multiple disaster types, enabling initial model evaluation without the overhead of extensive data collection.

As the project progressed to validation and refinement, we expanded our data strategy to incorporate **real-world posts from Bluesky**. When constructing the 1,000-post validation dataset, we integrated a subset of Bluesky-sourced posts alongside the existing disaster dataset. This **hybrid approach allowed us to validate model performance against both historical disaster data and contemporary social media patterns**, ensuring our classification system would perform effectively in production environments with real-time Bluesky data streams.

The text input presented consistent characteristics including informal language, hashtags, URLs, and variable post lengths typically ranging from 50-500 characters. This consistency across data sources enabled seamless integration into our preprocessing pipeline while maintaining the authenticity needed for real-world deployment.

3.2 Data Labeling and Taxonomy Development

To establish a reliable ground truth for model evaluation, we implemented a rigorous manual labeling process. Our data science subteam members participated in labeling posts according to established guidelines that defined clear boundaries for each disaster category, with each post being labeled twice to ensure accuracy. This collaborative approach ensured consistency across our labeled datasets.

Our category taxonomy evolved significantly through iterative analysis of model performance and error patterns. We began with **9 primary categories**: accident, earthquake, fire, flood, hurricane, not_relevant, other_weather_disaster, shooting, and tornado. Through systematic evaluation of classification challenges, we expanded to **12 refined categories** to better capture real-world disaster scenarios and reduce ambiguity. Key additions included distinguishing between severe_storm and tornado, adding tropical_storm for specific weather phenomena, and creating auto_accident to separate vehicle incidents from other accident types.

3.3 Dataset Composition and Distribution

Our benchmarking approach utilized two primary datasets with distinct distribution strategies. The initial 540-post dataset employed a balanced design with exactly 60 samples per category, ensuring equitable representation across all disaster types for fair model comparison. This balanced approach eliminated category bias during our initial model evaluation phase (see chapter 4 for more).

Category	**Count**	**Percentage**
accident	60	11.1%
earthquake	60	11.1%
fire	60	11.1%
flood	60	11.1%
hurricane	60	11.1%
not_relevant	60	11.1%
other_weather_disaster	60	11.1%
shooting	60	11.1%
tornado	60	11.1%
Total	**540**	**100%**

Fig. 3.1 Initial 540-Post Benchmarking Dataset Distribution

For validation and stress testing, we developed a **1,000-post dataset** with a distribution reflecting real-world social media patterns. This dataset increased the proportion of not_relevant posts to 32%, mirroring the natural noise encountered in production environments, while maintaining sufficient samples across all disaster categories to ensure robust model evaluation.

Category	Count	Percentage
not_relevant	320	32%
auto_accident	110	11%
fire	100	10%
flood	100	10%
severe_storm	90	9%
earthquake	80	8%
shooting	70	7%
tornado	40	4%
hurricane	30	3%
extreme_heat	30	3%
tropical_storm	20	2%
other_disaster	10	1%
Total	**1000**	**100%**

Fig. 3.2 Revised 1000-Post Benchmarking Dataset Distribution

3.4 Data Preprocessing Pipeline

The Jetstream object has access to all of the public posts on Bluesky, but we included a variety of filters to ensure that data being classified is relevant.

```
// Filter settings
KEYWORDS: ['vehicle collision', 'freeway crash', 'highway crash', 'mult-car pileup', 'traffic fatality', 'drunk driving',
  'wildfire', 'smoke plume', 'fire containment',
  'flash flood warning', 'storm surge',
  'earthquake', 'epicenter', 'seismic activity', 'seismic tremor', 'richter magnitude',
  'hail damage', 'lightning', 'downed power lines', 'wind gusts', 'thunderstorm warning', 'thunderstorm watch', 'severe thunder',
  'active shooter', 'mass shooting', 'polic standoff',
  'tornado', 'funnel cloud', 'record winds',
  'hurricane', 'category 3', 'category 4', 'category 5', 'landfall', 'coastal evacuation',
  'extreme heat', 'record temperature', 'cooling center',
  'heavy rainfall', 'tropical depression',
  'chemical spill', 'train derailment', 'gas explosion', 'building collapse'
],
LANGUAGES: ['eng'],
TRACKED_ACCOUNTS: ['nws.noaa.gov', 'fema.govmirrors.com', 'actionnews5.com', 'npr.org', 'sacurrent.bsky.social', 'calfire.bsky.social', 'cbssundaymorning.bsky.social', 'ucanr.edu',
  'massdfs.bsky.social', 'denverpolice.bsky.social'],
```

Fig. 3.3 Example of Filters Used with Jetstream

By providing a language filter and keywords, the Jetstream pulls data that is more likely to be related to disasters. Additionally, the tracked accounts are reliable news sources.

However, the language filter only guarantees that some of the text is in English characters, so posts that have some non-English characters are ingested as well. To prevent these posts from being analyzed and displayed, the following SQL query can remove these posts by checking if the text contains characters from foreign languages such as Chinese, Japanese, Korean, and Arabic using Unicode.

```sql
DELETE
FROM "be_posts_input"
WHERE text ~
```

```
'[\u4E00-\u9FFF\u3040-\u309F\u30A0-\u30FF\uAC00-\uD7AF\u0600-\u06FF\u0400-\u04FF\u0E00-\u0E7F\u0590-\u05FF]';
```

Fig 3.4 SQL Query Checking for Non-English Characters

For model input preparation, we implemented token limit management optimized for the 800-token capacity of our selected LLM, ensuring efficient processing without information loss.

The preprocessing strategy also handles the structured JSON output requirements of our production system, formatting model responses to include disaster category, severity assessment, location information, and confidence scores in a consistent schema that facilitates downstream processing and visualization.

3.5 Evolving Classification System and Feature Pipeline

Following the selection of our optimal model, we enhanced the classification system to improve real-world applicability and defined a comprehensive feature set for production deployment. The system now extracts multiple features beyond basic disaster categorization, enabling richer analysis and more informative display in our dashboard interface.

The feature extraction pipeline generates structured JSON output containing:

- **Disaster category** from our refined 12-category taxonomy
- **Severity level** assessment (low, medium, high) based on contextual analysis
- **Location information** including mentioned places and approximate latitude and longitude coordinates
- **Model confidence scores** providing transparency about classification certainty
- **Help request** indicator, in order to determine if the user requires help during the disaster

This multi-feature approach transforms raw social media posts into actionable intelligence, supporting informed decision-making for emergency responders and public safety officials.

3.6 Production Data Flow

Our architecture is designed for consistency, reliability, and smooth ingestion under rate limit constraints. The production data flow will go through the transfer of a Bluesky post to ingestion to front-end visualization.

1. **Ingestion** - Jetstream Firehose

The pipeline begins with a Jetstream instance that runs on a scheduled basis through GitHub Actions. Jetstream listens to the Bluesky firehose and streams posts that match our filtering criteria. To ensure relevance, each ingested post must satisfy at least one of the following conditions:

- Contains disaster-related keywords (“fire”, “flood”, “earthquake”, etc.).
- Originates from trusted news or emergency response accounts.
- Passes a language filter that ensures primarily English-language content.

Valid posts are immediately formatted into a uniform structure and inserted into the 'be_posts_input' table in Supabase. Each row contains the post text, author handle, URI, CID, and the Jetstream indexed timestamp.

2. **Queueing** - Structured Data Movement

To maintain stable ingestion under Groq's API limits, posts are not classified immediately. A scheduled Supabase edge function runs every minute and moves a small batch of new posts from 'be_posts_input' into the 'processing_queue' table. This separation between ingestion and classification provides:

- Isolation between data collection and classification workloads
- Ability to buffer posts during high volume periods

3. **Classification** - Groq via Edge Functions

A second scheduled edge function retrieves a batch of posts from 'processing_queue' and sends them through the Groq Cloud API. The Qwen3-32B model evaluates each post according to our 12 category disaster taxonomy and extracts multiple structured features:

- Disaster type
- Severity level
- Help request indicator
- Mentioned location and latitude/longitude
- Model confidence score

Successful classifications are written to the 'be_extracted_info_output' table. Failures (e.g. malformed content, timeout) are logged into 'failed_classifications', where a third edge function retries processing using the same structured workflow.

4. **Post Processing** - Popularity Metrics and Cleanup

After posts are successfully classified, the back-end retrieves like, report, and reply counts through the AT Protocol API. These values are appended to the output records since they enhance ranking and provide context for the front-end's "Top Posts" and help request indicators.

Non relevant or failed posts are periodically removed to keep the pipeline efficient and the database clean.

5. **Storage and Sync with Front-end**

All structured and enriched posts reside in 'be_extracted_info_output', which serves as the unified data contract for the dashboard. The front-end queries the database on page load or manual refresh, fetching the latest classified posts at the moment the user opens or reloads the dashboard. This pull based approach keeps the system simpler, avoids constant data streaming, and still provides a near real time awareness with minimal delay between ingestion and visualization.

6. **Visualization** - Display in the RiskRadar Dashboard

The front-end retrieves processed posts through the Supabase client. The data powers:

- Breaking numbers
- An interactive heatmap that uses latitude/longitude for location, severity, and model confidence
- A data table with multi-filed filtering (disaster type, severity, data range, help requests)
- Charts showing temporal and categorical patterns

This completes a full cycle from live Bluesky posts to structured, geospatially visualized disaster intelligence, typically within minutes.

Chapter 4: Model Development and Benchmarking

This chapter presents our systematic approach to identifying the optimal Large Language Model for real-time disaster classification. We detail a comprehensive benchmarking methodology that evaluated 16 models across multiple architectural families, employing rigorous experimental protocols to assess performance across accuracy, speed, and operational viability. The multi-stage evaluation process encompassed initial screening on a balanced 540-post dataset, iterative prompt refinement, and final validation on a 1,000-post real-world distribution, ultimately establishing Qwen/Qwen3-32B as the foundation for our production crisis detection system.

4.1 Experimental Design

Our model development process employed a rigorous, systematic approach to identify the optimal Large Language Model for real-time disaster classification.

API Platform Selection: After evaluating several API providers including OpenAI, Anthropic, and Google Gemini, we selected the **GroqCloud as our primary benchmarking platform**. This decision was driven by its superior value proposition for rapid, large-scale testing, characterized by exceptionally fast inference speeds (leveraging LPU hardware) and a generous free tier with manageable rate limits (Requests-Per-Minute and Requests-Per-Day). This combination allowed for the efficient execution of hundreds of test iterations without the prohibitive costs or strict throttling encountered with other providers. For comparative analysis, we also tested models available through the Mistral AI platform.

Postman Testing Framework: We developed an automated testing framework using Postman Collections with custom pre-request and test scripts. This infrastructure enables batch processing of prompts across multiple models with configurable rate limiting (2-9 second delays between requests), automated response validation against ground truth labels, and comprehensive metrics collection inducing latency, token usage, and accuracy. Our evaluation employed multiple metrics to assess model performance, with primary focus on classification accuracy, F1-score (macro), Precision, and Recall, while also tracking operational metrics like inference latency and token consumption for production viability assessment.

4.2 Comprehensive Model Evaluation

4.2.1 Algorithm Selection and Diversity

We evaluated multiple distinct algorithmic approaches representing different architectural families to ensure comprehensive coverage. The OpenAI GPT family, represented by gpt-oss-20b and gpt-oss-120b, provided general-purpose transformer-based performance with strong instruction following capabilities. The Qwen series, specifically qwen3-32b, offered optimized classification performance with efficient reasoning. Finally, specialized instruc models including moonshotai/kimi-kw-instruct and deepseek-r1-distill-llama-70b provided task-optimized architecture for comparison.

Models Evaluated: We systematically tested 16 different LLMs accessible via the GroqCloud and Mistral AI API platforms to ensure comprehensive coverage across architectural approaches and parameter scales. The tested models, grouped by their provider and parameter count, are listed below.

API Provider	Model Name	Size (Params)
Groq	allama-2-7b	7B
	deepseek-r1-distill-llama-70b	70B
	gemma2-9b-it	9B
	compound-mini	-
	llama-3.1-8b-instant	8B
	llama-3.3-70b-versatile	70B
	meta-llama/llama-4-maverick-17b-128e-instruct	17B
	meta-llama/llama-4-scout-17b-16e-instruct	17B
	moonshotai/kimi-k2-instruct	-
	moonshotai/kimi-k2-instruct-0905	-
	openai/gpt-oss-120b	120B
	openai/gpt-oss-20b	20B
	qwen3-32b	32B
Mistral	open-mistral-7b	7B
	open-mistral-nemo	-
	open-mixtral-8x7b	56B

Fig. 4.1 Comprehensive List of Models Evaluated

Experimental Framework: Our testing methodology was built on a Postman collection runner, which automated the process of sending each of the 540 prompts to every model's API endpoint. To manage API rate limits without sacrificing throughput, we implemented a configurable delay (typically 2-5 seconds) between requests. Each model was invoked with a standardized prompt and identical parameters (temperature=0, top-p=1) to ensure a fair comparison.

Evaluation Metrics: The primary metric for model selection was classification accuracy. However, to fully assess viability for a real-time system, we also collected data on inference

latency (response time), and calculated secondary metrics including F1 score, Precision, and Recall to understand error patterns beyond simple accuracy. This multi-faceted evaluation approach ensured we selected a model that balanced predictive performance with operational requirements.

4.2.2 Parameter Optimization Strategy

Our experimental design included systematic parameter optimization across key hyperparameters. We conducted controlled experiments varying temperature (0, 0.1, 0.2), top-p values (0.9, 1.0), max_tokens (500, 800, 1000), and penalty parameters to identify optimal configurations for each model architecture. This systematic approach ensured fair comparisons and identified the specific parameter combinations that maximized performance for each algorithmic approach.

Parameter	Values Tested
Temperature	0, 0.1, 0.2
Top-p	0.9, 1.0
Max Tokens	500, 800, 1000
Frequency Penalty	0, 0.1, 0.15, 0.2
Presence Penalty	0, 0.1, 0.15, 0.2

Fig. 4.2 Hyperparameter Search Space

4.3 Benchmarking Results

4.3.1 Initial 540-Post Evaluation

The initial 540-post benchmark provided a clear performance hierarchy. The results revealed that GroqCloud models consistently outperformed those from Mistral AI for our specific task. The top-performing model was openai/gpt-oss-2b, which achieved an overall accuracy of 93.0%. It was closely followed by qwen/qwen3-32b (92.6%) and moonshotai/kimi-k2-instruct (92.2%). A key finding was that larger parameter count did not guarantee better performance, as the 20B parameter model surpassed the 120B version, suggesting a **"sweet spot" for this classification task**.

API Provider	Model	Core Metrics Accuracy	F1 Score	Precision	Recall	Worst Cat. Accuracy	Operational Metrics/Parameters FPR	RPM	Latency/Tokens		Temp.	Top-p	
Groq	allam-2-7b	76.80%	51.50%	53.10%	53.20%	0.00%	0.00%	30	7K	310.1 ms	10	0	1
Groq	deepseek-r1-distill-llama-70b	91.40%	91%	91.20%	91.30%	62.70%	1.64%	30	1K	1562.1	500	0.1	0.9
Groq	gemma-2-9b-it	87.60%	87.80%	89.10%	87.60%	60.30%	2.30%	30	14.4K	141.8 ms	10	0	1
Groq	compound-mini	89.80%	89.90%	90.70%	89.90%	76.70%	4.59%	30	250	1562.1 ms	10	0	1
Groq	llama-3.1-8b-instant	79.40%	65.70%	70.00%	65.00%	48.30%	6.69%	30	14.4K	421.5 ms	10	0	1
Groq	llama-3-70b-versatile	90.60%	90.30%	91.00%	90.60%	60.00%	1.04%	30	1K	395.3 ms	10	0	1
Groq	meta-llama/llama-4-mav-erick-17b-128k-instruct	90.90%	68.20%	68.60%	68.20%	65.00%	1.48%	30	1K	426.9 ms	10	0	1
Groq	meta-llama/llama-4-scout-17b-16k-instruct	90.60%	90.70%	91.50%	90.60%	80.00%	5.63%	30	1K	480.2 ms	10	0	1
Groq	moonshotai/kimi-k2-instruct	92.20%	92.10%	92.70%	92.20%	66.70%	3.54%	60	1K	322.2 ms	10	0	1
Groq	moonshotai/kimi-k2-instruct-0905	89.40%	89.30%	89.90%	89.70%	70.60%	5.00%	60	1K	1544.5 ms	15	0	1
Groq	openadapt/gpt-oss-120b	92.00%	92.00%	92.40%	92.00%	73.30%	3.75%	30	1K	507.2 ms	500	0	1
Groq	openadapt/gpt-oss-20b	93.00%	92.90%	93.30%	93.00%	76.70%	3.12%	30	1K	758.5 ms	500	0	1
Groq	qwen3-32b	92.60%	92.40%	92.00%	92.60%	75.00%	1.67%	60	1K	973.7 ms	1000	0	1
Mistral	open-mistral-7b	76.70%	54.20%	58.50%	53.10%	45.00%	3.79%	500K	33M	326.4 ms	10	0	1
Mistral	open-mistral-nemo	81.50%	62.10%	65.80%	61.10%	56.70%	1.68%	500K	33M	448.6 ms	15	0	1
Mistral	open-mistral-8x7b	85.20%	76.50%	78.30%	76.70%	55.00%	3.76%	500K	33M	416.2 ms	10	0	1
Target		> 85%				> 70%	< 5%			**i.e. -0.4162 sec/per request**			

CODE/KEY	
Good	(meets or exceeds target)
Acceptable	(close to target; w/in 10%)
Needs improvement	(below target)

KEY INSIGHTS							
Provider	**Model**	**Accuracy**	**F1 Score**	**Precision**	**Recall**	**Worst Cat. Accuracy**	**FPR**
Groq	qwen3-32b	92.60%	92.40%	92.00%	92.60%	75.00%	1.67%
Groq	moonshotai/kimi-k2-instruct	92.20%	92.10%	92.70%	92.20%	66.70%	3.54%

Fig. 4.3 Comprehensive Benchmarking Results for All Tested Models

4.3.2 Scaling Validation and Iterative Prompt Refinement

To validate our findings on a larger scale and enhance statistical significance, we created a second dataset of over 1,000 posts. The distribution was adjusted to be more representative of real-world social media patterns, with a higher proportion of not_relevant posts reflecting the natural noise encountered in production environments. As mentioned in chapter 3, we also refined our category taxonomy based on initial error analysis, expanding from the original 9 to 12 categories to resolve ambiguities identified during early testing, such as adding tropical_storm to better distinguish between different weather phenomena.

On this larger and more realistic dataset, we conducted 25 detailed experimental runs focused on our top 5 candidate models. This intensive phase involved systematic prompt engineering to improve the model's ability to output structured JSON and correctly handle edge cases that had proven challenging in initial testing. This iterative refinement process was critical, with optimized prompts boosting the peak accuracy of qwen/qwen3-32b to 97% on a validation subsets, confirming its status as the most effective model for our pipeline.

gpt-oss-20b

Run	Accuracy	Time delay (in pre)	Prompt length	temp	max_tokens	top_p	freq_penalty	pres_penalty
1	N/A	3 seconds	short	0	500	1	N/A	N/A
2	76%	5 seconds	short	0	500	1	N/A	N/A
3	88%	5 seconds	short	0	800	1	N/A	N/A
4	87% / 94%	5 seconds	medium	0.1	1000	0.95	0.1	0.1
5	N/A	5 seconds	long ish	0.1	600	0.9	0.2	0.1
6	N/A	5 seconds	long	0.1	800	0.9	0.2	0.1
7	89%	9 seconds	long	0.1	800	0.9	0.2	0.1
8	91%	9 seconds	medium	0.1	800	0.9	0.15	0.15
9	88%	9 seconds	medium	0.1	800	0.9	0.15	0.15
10	92%	9 seconds	medium	0.1	800	0.9	0.15	0.15
11	87%	9 seconds	long ish	0.1	800	0.9	0.15	0.15

gpt-oss-120b

Run	Accuracy	Time delay (in pre)	Prompt length	temp	max_tokens	top_p	freq_penalty	pres_penalty
12	88%	9 seconds	short	0.1	800	0.9	0.1	0.1
13	86%	9 seconds	long ish	0.2	800	0.9	0.1	0.2
14	89%	9 seconds	short	0.1	800	0.9	0.15	0.15
15	86%	9 seconds	medium	0.1	800	0.9	0.15	0.15
16	91%	9 seconds	medium	0.1	800	0.9	0.15	0.15

qwen/qwen3-32b

Run	Accuracy	Time delay (in pre)	Prompt length	temp	max_tokens	top_p	freq_penalty	pres_penalty
17	88%	9 seconds	medium	0.1	800	0.9	0.15	0.15
18	93%	9 seconds	medium	0.1	800	0.9	0.15	0.15
19	94%	9 seconds	medium	0.1	800	0.9	0.2	0.2
20	97%	9 seconds	medium	0.1	800	0.9	0.2	0.2
21	95%	9 seconds	medium	0.1	800	0.9	0.2	0.2

moonshotai/kimi-k2-instruct

Run	Accuracy	Time delay (in pre)	Prompt length	temp	max_tokens	top_p	freq_penalty	pres_penalty
22	95%	9 seconds	medium	0.1	800	0.9	0.2	0.2
23	95%	9 seconds	medium	0.1	800	0.9	0.2	0.2

deepseek-r1-distill-llama-70b

Run	Accuracy	Time delay (in pre)	Prompt length	temp	max_tokens	top_p	freq_penalty	pres_penalty
24	94%	9 seconds	medium	0.1	800	0.9	0.2	0.2
25	95%	9 seconds	medium	0.1	800	0.9	0.2	0.2

Fig. 4.4 Validation Benchmarking Results for top 5 Models

4.3.3 Final Model Selection and Conclusion

The comprehensive, multi-stage benchmarking process conclusively identified Qwen/Qwen3-32B as the optimal model for our production pipeline. It delivered the best balance of high accuracy with consistently reproducible performance in the 94%-97%, reasonable latency suitable for real-time processing, and excellent cost-effectiveness for scalable deployment. A key factor in this selection was Qwen's reproducible performance ceiling of 97% accuracy, which demonstrated superior consistency and reliability compared to other top contenders.

While other models in the top five performed admirably, specific limitations informed our final decision. The OpenAI/gpt-oss-20b model, despite strong initial results, proved less consistent in later testing phases and was more sensitive to prompt variations. The Moonshotai/kimi-k2-instruct model was a strong competitor but did not match Qwen's peak accuracy across multiple validation sets. Furthermore, our testing of the deepseek-r1-distill-llama-70b model was unfortunately interrupted when the model was

deprecated by the provider mid-experiment, eliminating it from consideration for a stable production system.

This rigorous evaluation established **Qwen/Qwen3-32B** as the most reliable and high-performing foundation for our real-time crisis detection system, providing the **optimal balance of accuracy, operational efficiency, and stability required for emergency response scenarios** where both speed and reliability are critical.

4.4 Experimental Artifacts and Reproducibility

All experimental code, configuration files, and results from our comprehensive benchmarking process are archived in our project repository to ensure full reproducibility. For detailed access to the specific artifacts, including Postman collections, analysis scripts, result logs, and parameter configurations, please refer to **Appendix A: A.3 Experimental Artifacts**.

Chapter 5: System Architecture and Implementation

5.1 System Architecture Overview

Our crisis detection system employs a distributed, event-driven architecture designed for near real-time processing of social media data. The system follows a modular design pattern that separates concerns across data ingestion, processing, storage, and presentation layers. The architecture leverages serverless computing for scalable, cost-effective operation while maintaining low-latency response times critical for emergency scenarios.

The core data flow begins with real-time posts from the Bluesky social network, which are processed through a pipeline of serverless functions, classified by the Qwen model, stored in a structured database, and finally presented through an interactive dashboard. This end-to-end pipeline operates continuously with minimal human intervention, providing near real-time disaster detection and analysis.

The user interacts directly with front-end components that communicate with the back-end Supabase database to pull and present information. The data extraction and classification pipeline is not affected by the user. If the user wishes to specify the information displayed, they can do so through the filters and sorting features created in the heat map and data table. These narrow down the presented information by severity, disaster, posting date, and help request status.

To support this workflow, the system distributes responsibilities across independently scalable components. Jetstream handles ingestion, Supabase manages structured storage and scheduled back-end functions, Groq performs classification, and Vercel delivers the user-facing dashboard. This separation of concerns ensures that each layer can be updated or expanded such as adding new disaster categories, changing ingestion frequency, or improving visualization without disrupting the rest of the system.

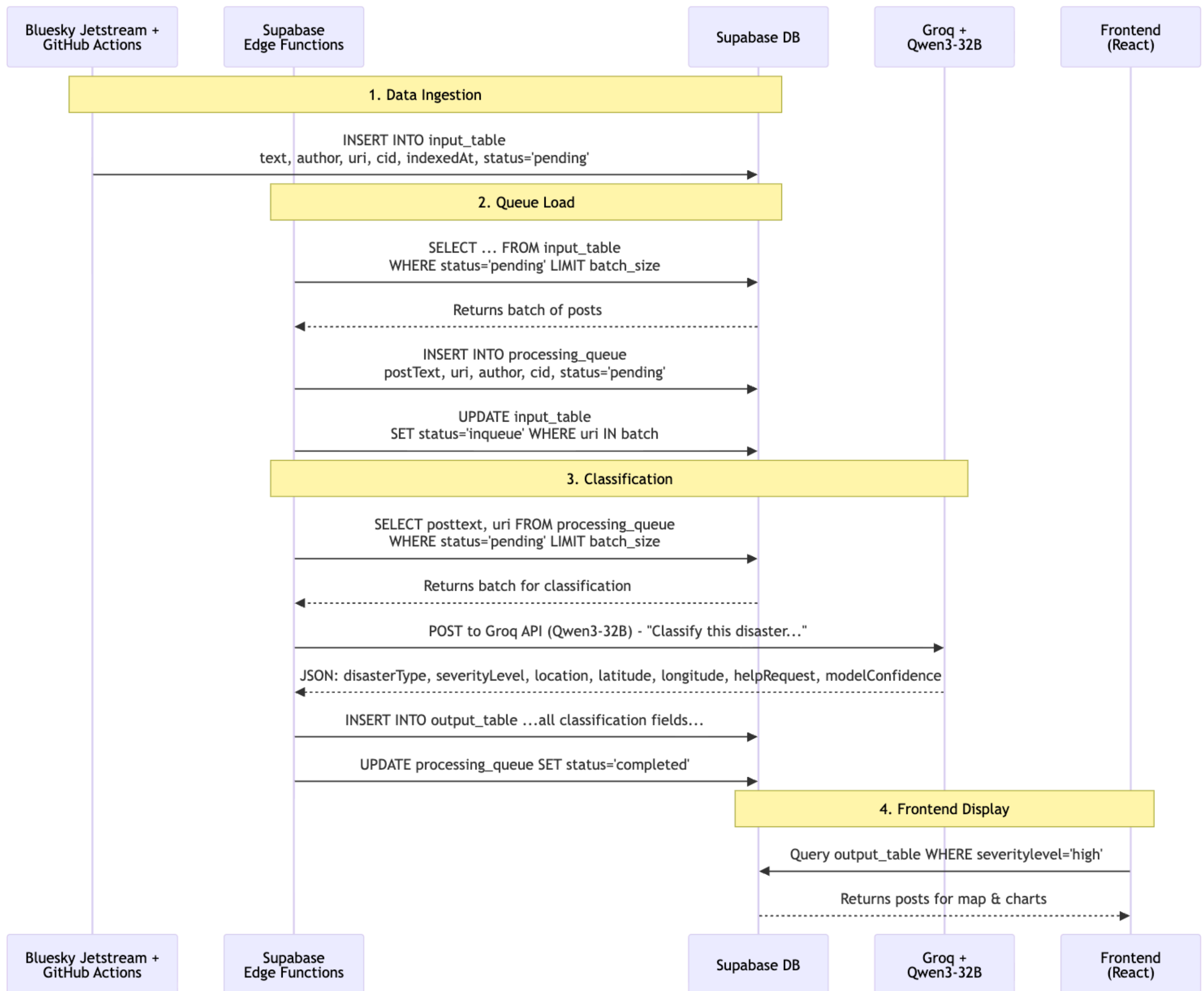


Fig. 5.1 UML Sequence Diagram

5.2 Database Design and Schema

The database schema employs a normalized design with clear separation between input, processing, output, and error handling tables. This design ensures data integrity while supporting efficient querying for both operational and analytical purposes.

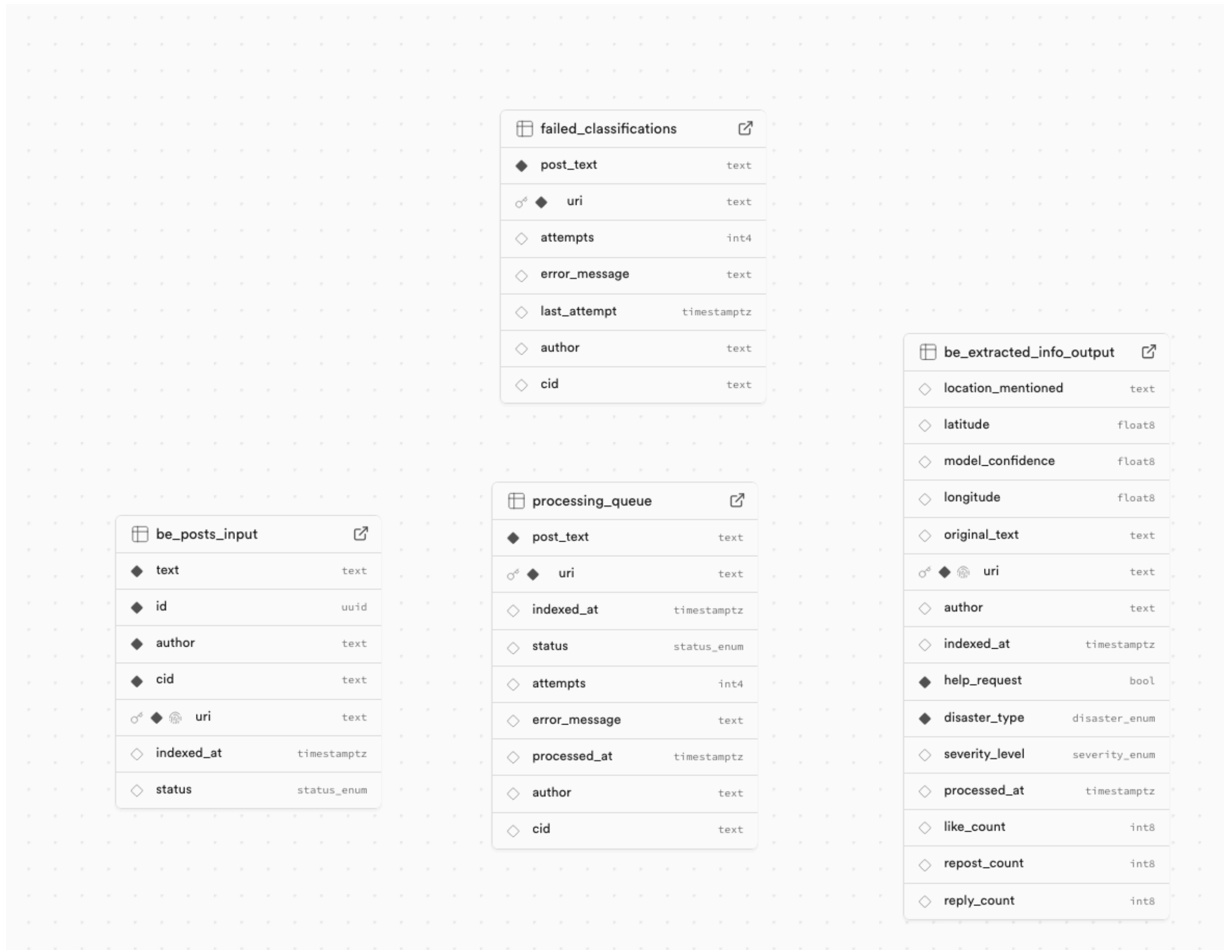


Fig. 5.2 Database Tables Schema Diagram

Table	SQL CREATE Statement
be_posts_input	create table public.be_posts_input (text text not null, author text not null, cid text not null, uri text not null, indexed_at timestamp with time zone null default now(), status public.status_enum null default 'pending'::status_enum, constraint be - posts_input_pkey primary key (uri),

	constraint posts_uri_key unique (uri)) TABLESPACE pg_default;
processing_queue	create table public.processing_queue (post_text text not null, uri text not null, indexed_at timestamp with time zone null default now(), status public.status_enum null default 'pending'::status_enum, attempts integer null default 0, error_message text null, processed_at timestamp with time zone null default now(), author text null, cid text null, constraint processing_queue_pkey primary key (uri)) TABLESPACE pg_default;
be_extracted_info_output	create table public.be_extracted_info_output (location_mentioned text null, latitude double precision null, model_confidence double precision null, longitude double precision null, original_text text null, uri text not null, author text null, indexed_at timestamp with time zone null, help_request boolean not null default false, disaster_type public.disaster_enum not null default 'not_relevant'::disaster_enum, severity_level public.severity_enum null, processed_at timestamp with time zone null, like_count bigint null default '0'::bigint, repost_count bigint null default '0'::bigint, reply_count bigint null default '0'::bigint,

	constraint be - extracted_info_output_pkey primary key (uri), constraint be_extracted_info_output_uri_key unique (uri)) TABLESPACE pg_default;
failed_classifications	create table public.failed_classifications (post_text text not null, uri text not null, attempts integer null default 0, error_message text null, last_attempt timestamp with time zone null default now(), author text null, cid text null, constraint failed_classifications_pkey primary key (uri)) TABLESPACE pg_default;

Fig 5.3 SQL Definitions of Tables in Data Pipeline

The schema utilizes enumerated types for critical domains including disaster categories (disaster_enum with 12 values), severity levels (severity_enum), and processing status (status_enum). This ensures data consistency and enables efficient filtering and aggregation in front-end visualizations.

NAME	VALUES
status_enum	pending, processing, completed, failed
disaster_enum	not_relevant, auto_accident, fire, flood, earthquake, severe_storm, shooting, tornado, hurricane, extreme_heat, tropical_storm, other_disaster
severity_enum	low, medium, high

Fig 5.4 Enumerated Types in Supabase

5.3 Real-time Processing Pipeline

The processing pipeline operates as a series of coordinated batch operations that simulate real-time processing through frequent execution. The pipeline manages data flow through multiple stages with built-in error handling and retry mechanisms.

The pipeline timing is carefully coordinated to maximize throughput while respecting external API rate limits. With cron jobs executing every minute and batch sizes of 5 posts, the system achieves a theoretical maximum throughput of 300 posts per hour while maintaining sustainable operation within Groq API constraints.

The data in the posts_input table moves to the processing_queue table in batches so that it can be processed using an edge function that utilizes the Groq API. Once this data has been processed, disaster-relevant data enters the extracted_info_output table and the non-relevant posts are deleted. For the posts in the output table, their URIs are used to gather the repost, reply, and like counts through HTTP requests made using the AT Protocol API. This information is only relevant for posts in this table since it is only relevant to the front-end display.

5.4 Edge Functions and Cron Jobs

Our serverless architecture utilizes Supabase Edge Functions written in TypeScript, triggered by scheduled cron jobs that coordinate the classification pipeline. Edge Functions are serverless functions that run in a distributed global network, providing automatic scaling, low latency, and minimal operational overhead. We chose this approach over traditional server-based solutions because it eliminates server management, scales automatically with demand, and provides built-in deployment tooling. Each function executes in isolation with defined resources, ensuring consistent performance and security.

Cron jobs provide scheduled, recurring execution of our Edge Functions, creating a pseudo-real-time processing pipeline without the complexity of continuous streaming. We selected cron-based scheduling over event-driven triggers or message queues because it offers predictable resource usage, simpler error handling, and better alignment with Groq API rate limits. The minute-based scheduling with small batch sizes (6 posts) provides near real-time processing while staying well within API constraints, avoiding the complexity and cost of true real-time streaming systems.

This combination of serverless functions and scheduled execution provides an optimal balance of responsiveness, cost-effectiveness, and operational simplicity. The architecture handles variable workloads gracefully, scales automatically during high-volume periods, and maintains consistent performance without manual intervention, making it ideal for a crisis detection system that must remain operational with minimal maintenance.

Function	Purpose	Batch Size	Frequency	Key Logic
queue-loader	Moves batches of posts from input table to processing queue	5 posts/ execution	Every minute (cron)	Checks for existing rows, prevents duplicates, maintains data lineage
classification-worker	Processes queued posts through Groq AI classification	5 posts/ execution	Every minute (cron)	Calls Qwen model, handles JSON parsing, routes results appropriately
recover-failed	Implements retry logic for failed classifications	5 posts/ execution	Every minute (cron)	Tracks attempt counts, prevents infinite retry loops

Fig. 5.5 Edge Function Deployment Summary

Job Name	Schedule	Batch Size	Avg Processing Time
queue-loader	*/5 * * * *	6 posts	400-500ms
classify_batch	*/1 * * * *	6 posts	1800-2000ms
recover-failed	*/1 * * * *	6 posts	400-500ms

Fig. 5.6 Cron Job Scheduling and Performance Characteristics

5.5 Front-end and Back-end Integration

The front-end application, deployed on Vercel, interfaces with the back-end through Supabase's client library. The integration supports multiple critical features:

Up-to-Date Data Synchronization: The front-end applies the database changes using Supabase client, enabling immediate updates when new disaster classifications are processed and front-end pages are reloaded.

Geospatial Visualization: Leveraging latitude and longitude coordinates extracted during classification, the front-end displays disaster posts on a heatmap with clustering and filtering capabilities. The clustering of Bluesky post points is based on the ML model's confidence and the severity level of the disaster.

Post Filtering: The tuple various fields from our database output table enables users to filter results by severity level, disaster category, date range, and Help Request classification. This functionality supports quality-focused visualization and analysis of disaster events.

Urgency-based Prioritization: The severity_level and help_request fields from our database output table allows emergency responders to prioritize high-urgency posts while maintaining awareness of developing situations.

The integration maintains full data lineage from original Bluesky posts through classification results, enabling traceability and audit capabilities. The use of URI foreign keys throughout the

pipeline ensures that each classified post can be traced back to its source, supporting credibility assessment and source verification.

Preprocessing and Normalization before exposure to the front-end: Before data is made available to the dashboard, the back-end performs text normalization, timestamp standardization (UTC), and ensures all extracted fields conform to expected data types. This preprocessing step prevents inconsistent data from reaching the UI and reduces the need for defensive code on the front-end

Staged Data Flow Through Supabase Tables: The back-end employs a multi-table flow (‘be_posts_input’ feeds into ‘processing_queue’ feeds into ‘be_extracted_info_output’) to maintain clarity about each post’s state in the pipeline. This enables the front-end to rely solely on the final output table without needing to understand intermediate processing stages.

Automated Error Recovery for Reliable front-end Data: Failed classification attempts are routed to a dedicated retry table, where a scheduled recovery function reprocesses them. This ensures the front-end receives complete datasets without silent drops or missing posts caused by temporary API failures.

Back-end Rate Limiting and Batching: Classification is executed in controlled batch sizes to remain within Groq rate limits. This back-end design choice prevents sudden spikes in request volume from affecting data availability on the front-end or causing inconsistent update patterns.

5.6 Back-end Framework and APIs

Jetstream: The @skyware/Jetstream library allows us to create Jetstream objects that ingest data from Bluesky. Since Jetstream is a data service and not an API, the events it gathers are raw data that must be stored in the appropriate columns in the database.

AT Protocol API: The @atproto/api package allows us to create an Agent that makes HTTP requests to Bluesky to get relevant and necessary information.

Supabase: This platform serves as our PostgreSQL database and back-end framework. It provides authentication, storage, and serverless edge functions. Because Supabase uses PostgREST and automatically generates a RESTful API, we can easily perform CRUD operations on the data tables.

Vercel: Hosts and deploys our frontend dashboard, providing fast global delivery and automatic builds from GitHub. The frontend uses Vercel to securely access Supabase and display the processed disaster data without needing any additional server infrastructure.

5.7 Deployment Architecture

The deployment architecture centers on a fully serverless model that separates the front-end interface from the back-end processing pipeline while allowing both to scale independently. The front-end dashboard is deployed on Vercel, which provides automatic builds from the project repository and efficient routing of client side requests to Supabase. This ensures fast page loads, consistent data fetching, and minimal operational overhead for maintaining the user facing application.

Front-end Deployment – Vercel

- React/Vite app deployed on Vercel
- Automatic builds triggered by GitHub commits
- Environment variables managed via Vercel dashboard

Benefits:

- Zero server maintenance
- Global edge network
- Automatic HTTPS and caching

Back-end Deployment – Supabase

- PostgreSQL database
- REST API
- Edge Functions
- Auth. and storage
- Real-time listeners

Note: Edge Functions are deployed via Supabase CLI and run on Deno's global edge runtime.

Cron Scheduling

Two scheduling approaches are used:

- Supabase Cron: for queue movement and classification
- GitHub Actions: for Jetstream ingestion scripts

This hybrid design keeps ingestion and processing decoupled.

External APIs

- Groq API keys managed through environment variable
- AT Protocol rate limits handled through timed requests
- All service outages handled with retry strategies

Deployment Challenges

- Groq API daily rate limits required key rotation
- Jetstream instability required retry logic
- Some functions required staggered scheduling to avoid hitting RPM limits
- Cloudflare outage (Nov. 18) temporarily disrupted all edge functions

Chapter 6: Performance Analysis and Results

6.1 Production Pipeline Performance

The operational crisis detection system demonstrated robust scalability and resilience, processing over **~50,000** Bluesky posts during a 12-day operational period from November 5-17. The system successfully classified **10,372 disaster-related events**, representing a **29.3% positive classification rate** that reflects both the quality of our model and the disaster-focused nature of the data stream. This substantial volume of processed data validates the system's capability to handle real-world social media throughput while maintaining consistent classification performance.

The pipeline exhibited excellent stability despite multiple API key rotations required to manage Groq's rate limiting constraints. Seven strategic API key changes were executed throughout the testing period, allowing the system to maintain continuous operation and process an average of **2,900 posts per day**. The gradual progression from 2,627 to 10,372 output classifications demonstrates sustained processing capability and effective queue management.

The processing queue management proved highly effective, maintaining an average of 21,000 posts in the processing queue while successfully clearing the backlog through systematic batch processing. The remarkably low failure rate, peaking at only 16,570 failed classifications early in the testing period and rapidly declining to near-zero, demonstrates the resilience of our error handling and retry mechanisms. By November 17, failed classifications were reduced to just 1 record, indicating excellent system reliability and recovery capabilities.

The progressive improvement in success rates from 18.8% to 46.7% over the operational period suggests potential model adaptation to the data stream characteristics and optimized pipeline tuning. This performance trajectory confirms the system's capacity for sustained high-volume processing while maintaining classification quality, establishing a solid foundation for production deployment in emergency response scenarios.

6.2 Disaster Type Distribution Analysis

The classification system identified diverse disaster types from production data, demonstrating comprehensive coverage across natural and human-made crises. The distribution reflects realistic emergency patterns that would be encountered in real-world deployment scenarios. The system's ability to categorize posts across multiple disaster types while maintaining contextual understanding of each category's characteristics validates the effectiveness of our 12-category taxonomy and model training approach.

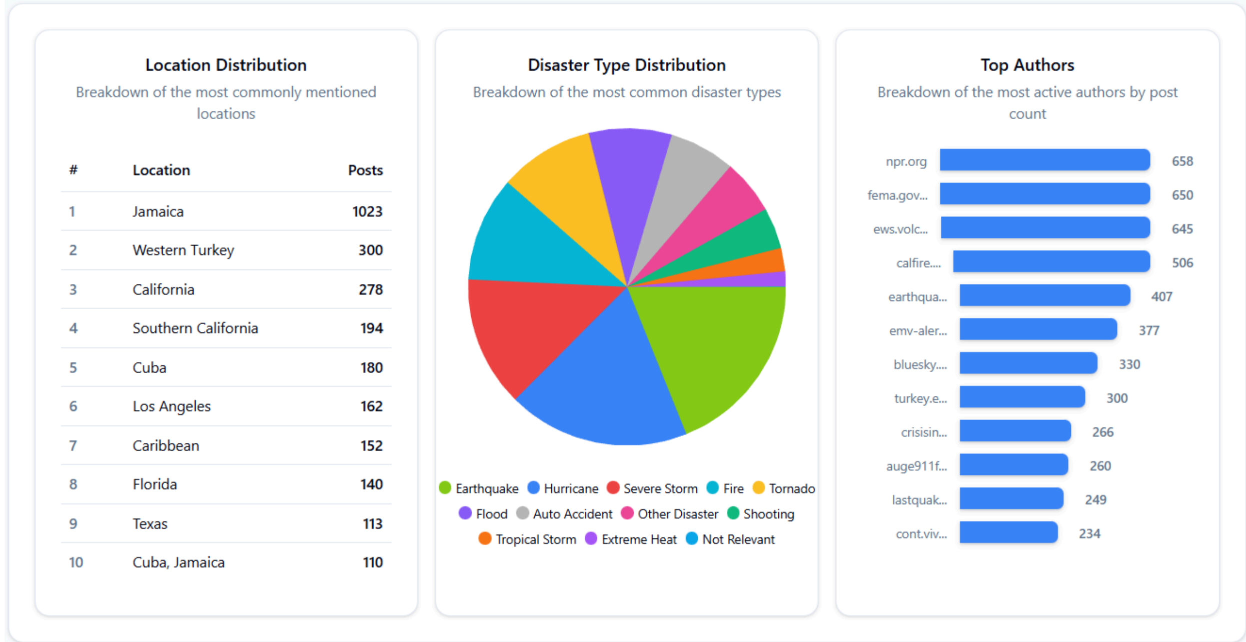


Fig. 6.1 Current Disaster Distribution as of December 2, 2025

6.3 Model Confidence and Severity Analysis

The Qwen model demonstrated strong overall confidence in its classifications, with an average confidence score of 0.917 across all disaster posts. Confidence levels varied meaningfully by disaster type, with fire incidents receiving the highest average confidence (0.92) while "other_disaster" categories showed more conservative confidence scores (0.81). This pattern suggests the model correctly calibrates its certainty based on classification difficulty and training data representation.

Severity level distribution revealed a balanced alert pattern, with 38.46% of incidents classified as high severity, 33.33% as medium, and 28.21% as low severity. The strong correlation between severity levels and confidence scores (high severity events averaged 0.92 confidence vs 0.87 for low severity) indicates the model's logical reasoning capabilities in assessing both event type and potential impact.

6.4 Geographic and Temporal Patterns

The system successfully extracted location information from 75+ distinct geographic references, demonstrating robust geographical entity recognition capabilities. Location mentions ranged from specific cities and neighborhoods to regional references, with clustering around populated areas that correlates with both population density and social media usage patterns. This geographic intelligence provides crucial context for emergency responders, enabling targeted resource allocation and situation awareness.

Temporal analysis revealed consistent processing throughout the operational period, with the system maintaining stable performance across varying load conditions. The classification pipeline demonstrated reliable throughput with posts processed within minutes of ingestion, meeting our objective of near real-time crisis detection for emergency response applications.

6.5 System Limitations and Bottlenecks

Several technical challenges emerged during system operation, primarily centered around data quality and model calibration. The absence of hurricane and tropical storm classifications reflects seasonal and geographic factors in the data collection period, while the presence of "not_relevant" posts in the output table indicates occasional classification errors that were successfully identified through manual review processes.

Data quality challenges included inconsistent location formatting in social media posts and occasional ambiguous disaster descriptions that complicated classification. The system's error handling and retry mechanisms proved effective in maintaining stability, with failed classifications successfully recovering through automated retry processes. The absence of help requests in the output data (0 count) suggests either low incidence of such posts in the source data or potential limitations in the help request detection logic, indicating an area for model prompt refinement in future iterations.

6.6 Rate Limiting Analysis and Adaptive Optimization

Early in the operational period, the system encountered significant challenges with API rate limiting that directly impacted performance metrics. The initial failure peak of 16,570 failed classifications on November 5th was primarily attributed to exceeding Groq API's daily request limits, which created a cascade effect throughout the processing pipeline. This bottleneck manifested as HTTP 429 errors and temporary service disruptions that required immediate system intervention.

The root cause analysis revealed that our initial timing specifications, while theoretically sound, did not adequately account for the cumulative effect of continuous processing across multiple cron jobs. The simultaneous operation of queue-loader, classification-worker, and recover-failed functions, each running with 6-post batches, created a sustained request pattern that quickly consumed available API quotas despite individual batch sizes appearing conservative. In response to these challenges, we implemented a comprehensive adaptive optimization strategy:

Multi-API Key Rotation: We established a systematic key rotation protocol across four distinct Groq API keys, distributing the processing load and effectively quadrupling our daily capacity from 1,000 to 4,000 posts per day.

Timing Specification Refinement: We recalibrated the cron job scheduling to introduce strategic offsets, ensuring that parallel functions would not execute simultaneously and create

request spikes. The implementation of staggered execution patterns reduced instantaneous load while maintaining overall throughput.

Intelligent Backlog Management: The system was enhanced with dynamic batch sizing that could adjust based on real-time queue depth and recent error rates, providing built-in throttling during high-load periods while maximizing utilization during optimal conditions.

The effectiveness of these adaptations is clearly demonstrated in the rapid decline of failure rates, dropping from thousands of failed classifications to single digits within four days of implementation. By November 9th, failed classifications were reduced to just 5 records, and the system maintained near-perfect reliability throughout the remainder of the operational period. This successful resolution underscores the importance of real-time monitoring and adaptive configuration in cloud-based AI service integration.

Despite the robust adaptive optimizations implemented for rate limiting, the system's operational continuity remained inherently dependent on the stability of external API providers. This dependency was starkly illustrated on November 18th, when Cloudflare experienced a significant back-end outage, ultimately affecting the Groq API and our edge functions as well. Unlike rate-limiting errors which could be mitigated through key rotation and scheduling, a complete service failure at the provider level presented a challenge beyond the system's internal control. During such events, the automated retry mechanisms and fallback protocols were ineffective, and production was temporarily paused with no alternative but to wait for the upstream provider to restore service. This incident highlighted a fundamental operational constraint: while system design can optimize for predictable constraints like quotas, it remains vulnerable to the availability of the underlying third-party services it is built upon.

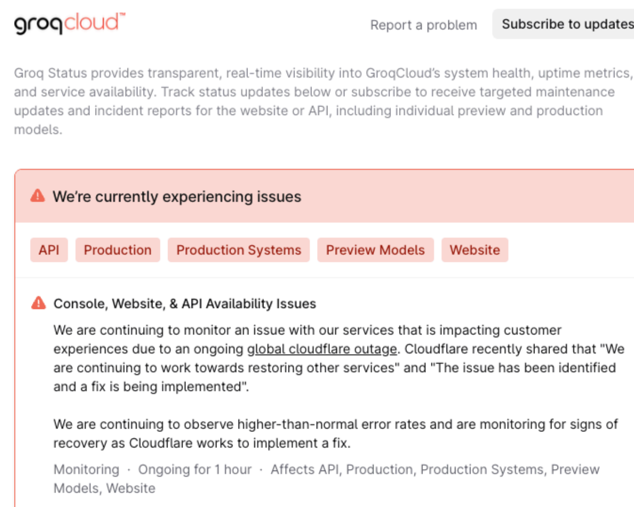


Fig. 6.4 Groq API Service Error

Chapter 7: User Interface and Dashboard

7.1 Front-end Component Architecture

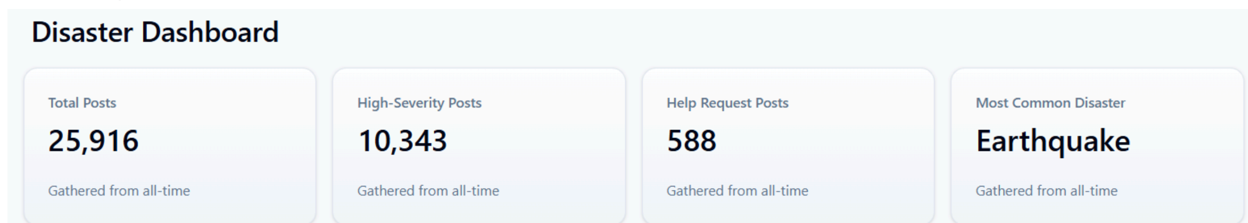
7.1.1 Front-end Tools

The front-end was built using **React.js** and **Vite** in addition to the **Leaflet**, **Tailwind CSS**, **Lucide Icon**, and **Shadcn UI** libraries for their ease of integration and modern and accessible designs.

Once all the front-end was developed, the front-end team decided to divide the main components into their respective pages for better cognitive ease and clarity of each component's goal.

7.1.2 Front-end Components

Breaking Numbers



The Breaking Numbers component displays the following: the total number of disaster posts classified and stored in our database, the total number of posts classified and stored as "high severity", the total number of posts classified and stored as a help request, and the most common disaster among the classified posts. Each of the breaking numbers takes into account all of the posts stored in our database.

For the first three Breaking Numbers, we accomplished the functionality through state management and Typescript's built-in functions. For the last Breaking Number, we accomplished its functionality using React's useMemo() hook to find and return the most common disaster type.

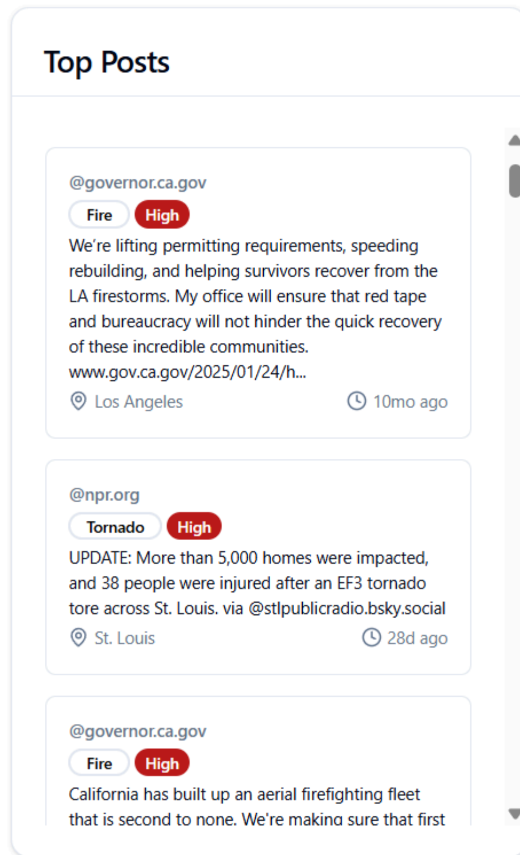
Heat Map, Markers, and Popup

The Heat Map component provides a visual representation of disaster posts. Here, a Leaflet map of the United States is displayed, featuring circle markers placed on the map according to coordinates extracted from the model and stored in the database. The marker color is indicative of the disaster type explained in the key below the map, and the disaster and severity level can both be seen when hovering over the marker. Clicking on the marker will display all details of the relevant post via a popup. Filters can be used to narrow the scope of posts displayed via date of posting, disaster type, help request status, or severity level.

The map allows the user to move around, zoom in, zoom out, and toggle to show and hide the markers to separate from the heat layer. The heat layer is indicative of the number of disasters by location and is determined by the severity of the disaster multiplied by the confidence score of the classification calculated by the model.

This functionality was accomplished through Leaflet's map, React's state management and filtering, and Typescript's .map() functionality to create post cards with the relevant posts' information.

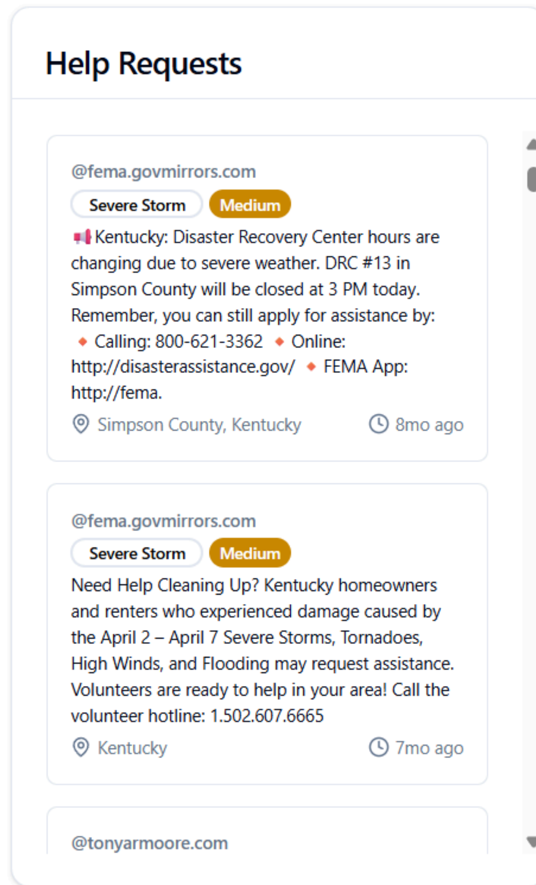
Top Posts Widget



The Top Posts Widget is a frame of minimized cards that display essential post information; the cards are sorted in order of the post's like count at the time it was retrieved through the Bluesky API. The post cards displayed correspond to the filters applied to heatmap, meaning that the post cards shown are reflective of the post markers on the heatmap. Additionally, a limit of the top 50 post cards by like count was applied to improve load time. The Top Posts Widget is displayed when the Help Request option is toggled off.

This functionality was accomplished through React's state management and sorting the posts applicable to the filters by like count, in addition to Typescript's .map() functionality to create post cards with the relevant posts' information.

Help Requests Widget



The Help Requests Widget is a frame of minimized cards that display essential post information for posts classified as help requests. The post cards displayed correspond to the filters applied to heatmap, meaning that the post cards shown are reflective of the post markers on the heatmap. Therefore, the Help Requests Widget is displayed only when the Help Request option is toggled on.

This functionality was accomplished through React's state management and Typescript's .map() functionality to create post cards with the relevant help request posts' information.

Data Table

All Activity

Filter by post text...

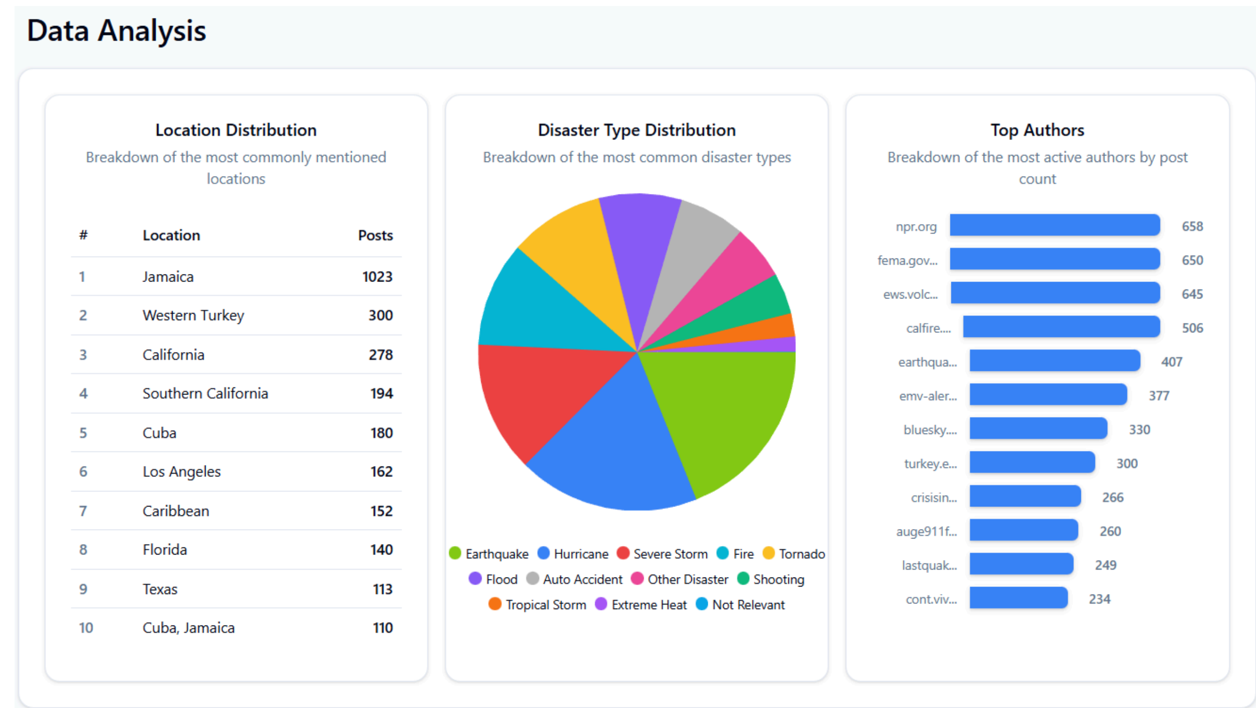
Select date | All Severities | All Disasters | Help Requests Only

Post Text	Disaster	Severity	Location	Timestamp	Help Requested	Handle
NPR's Scott Simon recounts the heroic actions of a Chicago bus driver and his passengers, who saw buildings on fire at 2 a.m. and raced to warn residents.	Fire	High	Chicago	Mar 9. 2024, 2:26 PM		npr.org
Three National Guard soldiers and a Border Patrol agent were on board when a helicopter crashed near the Mexican border. A soldier was seriously injured.	Other	High	Mexican border	Mar 9. 2024, 11:01 AM		npr.org
A dangerous blizzard expected to bring up to 10 feet of snow to parts of the Sierra Nevada has forced the closure of Yosemite National Park, at least nine Lake Tahoe ski resorts and a major interstate and disrupted power to thousands.	Severe Storm	High	Sierra Nevada, Yosemite National Park, Lake Tahoe	Mar 2. 2024, 11:07 AM		npr.org
As the wildfires in the Texas Panhandle continue leaving scorched homes, barns and livestock in their wake, some residents are beginning to glimpse what recovery will look like when the blazes are finally extinguished.	Other	High	Texas Panhandle	Mar 1. 2024, 6:18 PM		npr.org

The Data Table component displays the information of each post that has been extracted, classified, and stored in the database, including post text, disaster, severity, location, timestamp, help status, and handle. This is done through a table with sortable columns and filters for disaster, severity, posting date, and help request status. The table pulls data from Supabase and displays each post as its own row. The posts are displayed in groups of ten per page to increase readability and the user's experience.

This functionality was accomplished through React's state management and table sorting, filtering, pagination, and column properties. TypeScript was also used via .map() and table functions to sort, filter, and display data from Supabase.

Analysis Charts



The Analysis Charts refer to three data visualizations that analyze the classified and stored contents of the database. The first is a **Location Distribution** that lists the ten most frequently mentioned locations in the dataset, along with the number of mentions. This is done by sorting the dataset to count repeated locations.

The next is a **Disaster Type Distribution** that uses a pie chart to represent how the different disasters are distributed across the database. Each colored section refers to a disaster type as indicated on the key below, and hovering over the slice provides amount and volume statistics.

The last is a **Top Authors Chart** that displays the most active accounts in the database via a horizontal bar graph with count and handle information. This is done by sorting the database by handles based on post frequency.

This functionality was accomplished through React's state management capabilities and Typescript's .map(), .reduce(), and .slice() functionalities to create tables, pie charts, and graphs.

Resources

Disaster Preparedness Guide

Fire

Wildfires and house fires can spread quickly. Stay alert and act fast.

Supplies
- Fire extinguisher
- Mask or cloth to cover mouth
- Important documents
- Water and first aid kit

Actions
- Know at least two exits in your home.
- Avoid smoke inhalation and stay low if indoors.
- Evacuate early if advised by authorities.

Flood

Flooding can occur from heavy rain, storms, or overflowing rivers.

Supplies
- Bottled water
- Waterproof clothing
- Flashlight with batteries
- Emergency radio

Actions
- Move to higher ground immediately.
- Avoid walking or driving through flood water.
- Turn off electricity if safe to do so.

Earthquake

Sudden shaking of the ground caused by movement along fault lines.

Supplies
- First aid kit
- Sturdy shoes
- Flashlight
- Whistle for signaling

Actions
- Drop, cover, and hold on during shaking.
- Stay away from windows and heavy objects.
- After shaking stops, move to open areas.

Extreme Heat

Prolonged high temperatures can cause heat exhaustion or stroke.

Supplies
- Water bottles
- Cool clothing
- Electrolyte drinks
- Cooling packs or fans

Actions
- Stay hydrated and avoid direct sun.
- Go to cooling centers or shaded areas.
- Never leave children or pets in vehicles.

Hurricane

Strong tropical storm with heavy rain and winds exceeding 75 mph.

Supplies
- Water and nonperishable food to last 3+ days
- Battery-powered radio
- Extra clothing and blankets
- Evacuation plan

Actions
- Follow evacuation orders immediately.
- Board up windows and secure outdoor items.
- Shelter in an interior room away from windows.

Tornado

Fast rotating air extending from a thunderstorm to the ground.

Supplies
- Flashlight
- Whistle
- Sturdy shoes

Actions
- Go to a basement or interior room without windows.
- Cover your head and neck.
- Avoid mobile homes and vehicles.

This Resources page features cards that aim to inform users about different disaster types through brief descriptions, lists of needed supplies, and sections on how to prepare. The information presented on the page came from government resources such as FEMA, and can be seen credited in the footnote.

This functionality was accomplished through React capabilities and TypeScript’s .map() functionalities, creating a card for each disaster in the database.

Appendix

A.1 UML Diagrams
Located in Appendix Folder → Appendix/A.1
These diagrams were created to understand how the different components work together, and include multiple versions to reflect the changes made throughout the semester.

A.2 Complete SQL Schema and Queries
https://github.com/senior-design-crisis-analysis/SeniorDesign/tree/master/SQLQueries

A.3 Experimental Artifacts

Edge Function Source Code	
queue-loader	https://tinyurl.com/queue-loader
classification-worker	https://tinyurl.com/classification-worker
recover-failed	https://tinyurl.com/recover-failed
monitor-dashboard	https://tinyurl.com/monitor-dashboard

Postman Collection (LLM Benchmarking):
https://github.com/senior-design-crisis-analysis/SeniorDesign/blob/master/postman_benchmarking/postman_benchmarking.json

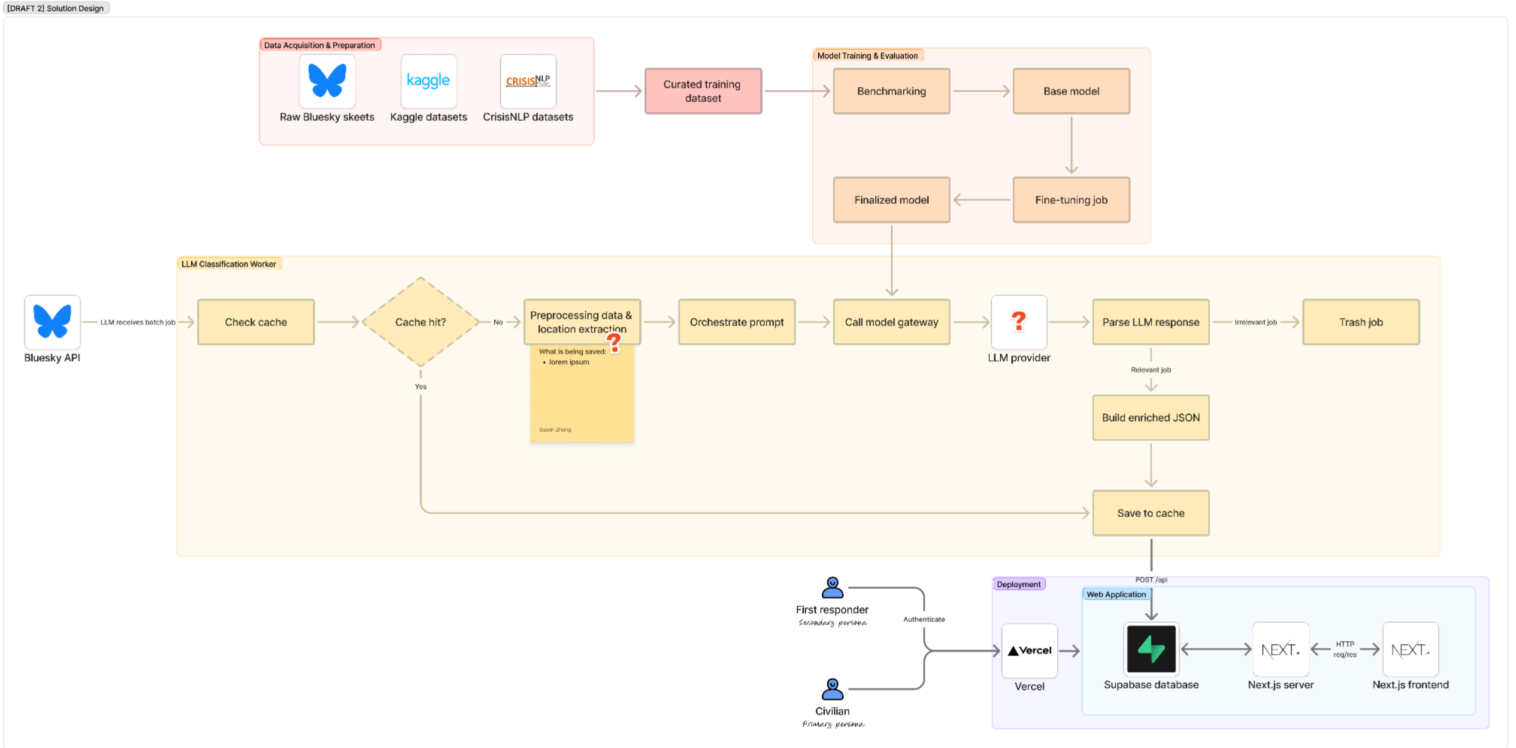
A.4 API Configuration Details

Groq API Free-Tier Limits for the qwen/qwen3-32b Model	
Requests Per Minute (RPM)	60
Requests Per Day (RPD)	1,000
Tokens Per Minute (TPM)	6,000
Tokens Per Day (TPD)	50,000

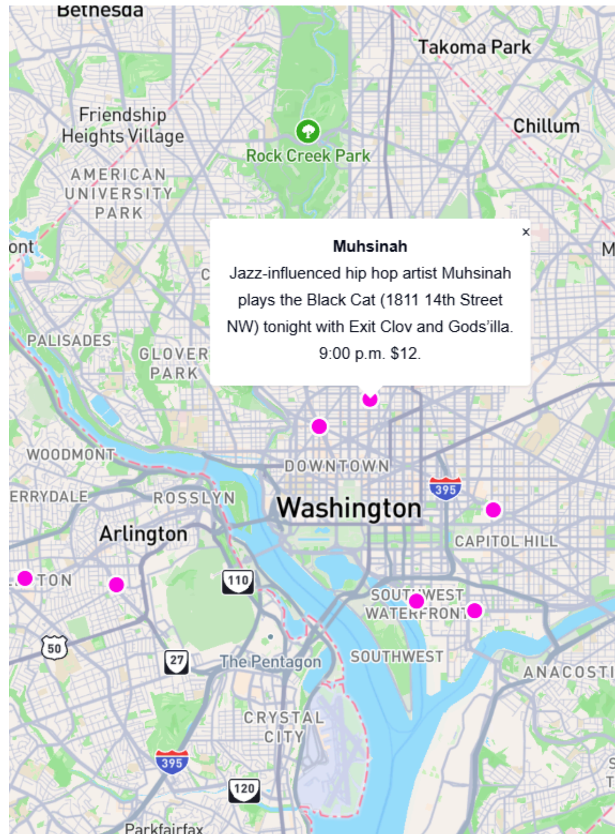
A.5 Project Management Artifacts
Located in Appendix folder → Appendix/A.5

- Original Project Proposal
- Meeting Minutes
- Weekly Sprint Report

A.6 Alternative Designs



An alternative, prior version of our solution design, where the team utilized Next.js for our front-end, had not yet begun benchmarking, considered using a cache, and were unsure of what features the ML model would be classifying.



An older version of the heatmap used Mapbox GL, but we converted it to Leaflet for its ability to display more information via marker colors, disaster type, post information, severity, and a separate heat layer.

A.7 Datasets

Located in Appendix folder → Appendix/A.7

- Datasets used for benchmarking and validation