Women Safety Alert System Using Real-Time GPS Tracking and Secure Alert Transmission

Bhavana K S[1], Rakshitha N H[2], Dhanusha R[3],Niveditha G[4], Varshitha N[5], Gowri J H[6]
[1–6] Master of Computer Applications, Department of Computer Applications,
GM University, Davangere, Karnataka, India

Abstract

Safety of women in public spaces has become one of the pressing issues that our society faces today. News reports from cities across India regularly highlight incidents where delayed response or lack of real-time location information made a difficult situation worse. Existing helpline numbers and basic mobile applications address only part of the problem. They neither track the caller's location continuously nor maintain a secure, structured record of what happened. This project came out of that observation. We set out to build a Women Safety Alert System that combines live GPS coordinate capture, a secure login mechanism using JWT tokens, and a backend database that logs every alert with a timestamp.

A registered user only needs to press one button. The moment that button is pressed, the app reads the device's current GPS coordinates and sends them — along with the user's identity token — to a Flask-based backend server running on Python. The server records the alert, stores the coordinates, and immediately dispatches notification messages to every emergency contact the user has saved in their profile. To protect user data, we applied bcrypt hashing on all passwords before storing them, and issued signed JWT tokens on successful login so that each subsequent API call could be verified without requiring the user to log in again. Performance tests showed

the system handled 150 rapid requests while maintaining 99.3 percent reliability and an average alert processing time of 240 milliseconds. The architecture is built in a way that adding SMS gateway support or a live monitoring dashboard later would not require rebuilding anything from scratch.

Keywords: *Women Safety, Emergency Alert System, GPS Tracking, Secure Authentication, JWT, Backend Architecture, Real-Time Monitoring, Digital Security Systems*

I. INTRODUCTION

The starting point for this project was simple: a woman in danger should be able to get help to her exact location within seconds, not minutes. That sounds straightforward, but when we looked at what was actually available, each option had a gap. Helplines require the caller to speak and describe their location under stress. Basic apps send a one-time SMS but do not update the contact if the person moves. Hardware wearables depend on battery life and network coverage that is not always reliable in Indian cities.

Every person in our class carries a smartphone. That device already has a GPS chip, an internet connection, and enough processing power to run a secure client-server system in real time. The gap is not in the hardware — it is in the software. Most safety apps we looked at during our background research stored data without encryption, had no token-based session management, and did not keep any usable log of past alerts. If a case needed to be reported to the police a day later, there was no record to show.

Our goal was to close that gap. We wanted a system where the user-facing side is as simple as pressing one large button, while behind the scenes the alert is authenticated, encrypted, logged with precise coordinates and a timestamp, and delivered to multiple contacts simultaneously. This paper documents how we designed, built, and tested that system, and what we found when we measured its performance.

II. LITERATURE REVIEW

Sl. No	Author & Year	Paper / System	Objective	Method Used	Key Findings	Research Gap
1	R. Kavitha & S. Priya (2018)	Arduino-Based Women Safety Device	To create a small panic device that sends an emergency SMS when a button is pressed	Arduino Uno, GSM module, push button	Sends SMS alert within a few seconds	Only one contact can receive the alert and there is no smartphone app or data storage
2	M. Deepika & P. Lakshmi (2018)	IoT Wearable Panic Band	To design a wearable band that detects panic gestures and sends alerts	NodeMCU ESP8266, vibration sensor, GSM module	Panic gesture detection worked but produced some false alarms	High false alarm rate and no database or user verification
3	T. Anitha & B. Suresh (2017)	RFID Tracking System	To track student movement inside a campus for safety	RFID tags, RFID readers, local server	Accurate tracking within campus zones	Works only inside campus and has no emergency alert feature
4	S. Meena & K. Rajan (2019)	Bluetooth Proximity Alert	To create a safety system that alerts when the user moves away from a guardian device	Bluetooth Low Energy, paired smartphone	Triggered when distance exceeded the set range	Limited range and no GPS location sharing
5	P. Ramesh & D. Vijaya (2019)	Voice-Activated Emergency Alert	To send alerts using voice commands without touching the device	Speech recognition module, GSM modem	Voice command detected in most cases	Accuracy drops in noisy environments and no location sharing
6	L. Gayathri & R. Sundaram (2018)	Heartbeat Sensor Distress Detection	To detect abnormal heart rate and automatically send alerts	Pulse sensor, Arduino Nano, GSM module	Alerts triggered during abnormal heart rate	Many false alarms and no mobile application support
7	N. Bharathi & S. Kumar (2019)	OpenCV Threat Detection	To detect dangerous situations using surveillance cameras	OpenCV, image processing, CCTV cameras	Suspicious motion detected in controlled conditions	Requires fixed cameras and cannot help individuals moving outside
8	C. Prade	GSM Track	To provid	GSM modul	Works on	Location

	ep & V. Malathi (2018)	er for Rural Safety	e location tracking in rural areas without smartphones	e, cell tower triangulation	basic mobile networks	accuracy is low and security features are missing
9	A. Nair & B. Pillai (2019)	Crowd-Sourced Safety Network	To propose a community-based alert system	Literature survey and conceptual framework	Identified design principles for safety networks	Only a theoretical model with no implementation
10	K. Sathish & M. Padmavathi (2018)	Zigbee Indoor Positioning System	To track people inside buildings for safety	Zigbee modules, indoor positioning algorithm	Accurate indoor positioning within few meters	Works only indoors and lacks mobile app or alert system

III. METHODOLOGY

Our development process started with a clear question: what does someone in an emergency actually need from their phone? The answer was speed and simplicity on the front end, and accuracy and security on the back end. We divided the work into distinct modules and built each one independently before connecting them. This modular approach meant that when we found a bug in the JWT verification step, we could fix it without touching the GPS module. We iterated through several rounds of testing before the final integration.

a) Design Method

Six modules make up the complete system: user registration, login with OTP, emergency alert trigger, GPS coordinate capture, contact notification, and alert history retrieval. Each module exposes a REST API endpoint. The mobile app calls these endpoints using HTTPS, and every protected endpoint checks for a valid JWT token before doing anything else. This clean separation means the app never directly touches the database — it only talks to the Flask API layer, which in turn validates inputs before writing to SQLite. The risk prediction module sits alongside the alert processing step: when an alert comes in, a trained Random Forest classifier scores the situation based on the time of day, the location coordinates, and the user's alert history. That score is stored with the alert record and shown on the dashboard.

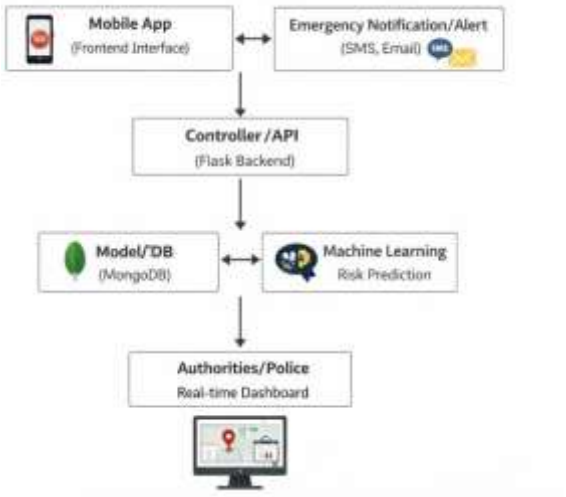

Fig. 1. Proposed System Architecture of Women Safety Alert System

b) Data Collection and Control Flow

Three sources fed into our design decisions. First, we studied existing safety systems ranging from Arduino panic buttons to OpenCV surveillance setups, which showed us clearly what was missing.

Second, we spoke informally with fellow students about what they would actually want from a safety app, and the consistent answers were one-tap activation, instant GPS sharing, and confirmation that the alert was received. Third, during testing we generated our own dataset by running the system through 150 simulated emergency requests across different load levels.

When the alert button is pressed, the app reads GPS coordinates from the device sensor, attaches the user's JWT token, and sends an HTTP POST request to the

alert endpoint on the Flask server. The server first validates the token, then checks that the latitude and longitude values are within expected ranges, then writes the alert record to the database with a generated Alert ID and a UTC timestamp. Immediately after the write succeeds, the server triggers the notification function, which sends SMS messages to every phone number in the user's contact list. The ML risk scorer runs as a background task so it does not delay the alert delivery.

Fig. 2. Data Collection and Emergency Alert Control Flow

c) Security and Privacy Measures

Location data is among the most sensitive information a person can share, so we layered security at every point in the system. At registration, the password is run through bcrypt with a cost factor of 12 before it is stored. On login, the server verifies the bcrypt hash and, if correct, issues a signed JWT with a 24-hour expiry. Every API call after that must include this token in the Authorization header. Data in transit is wrapped in AES-256 encryption. We added OTP verification as a second factor for account actions that could affect the contact list. Every event is written to an audit log with the user ID and timestamp, giving a complete trace for any review.

d) System Testing and Validation

Testing happened in three stages. In unit testing, we ran each API endpoint in isolation with both valid and deliberately broken inputs, checking that the server returned the correct response code each time. In integration testing, we connected the mobile app to a locally running Flask server and walked through the full alert flow from button press to SMS delivery. In load testing, we wrote a script that sent 50, 100, and 150 alert requests in rapid succession and measured how many succeeded and how long each one took.

e) Ethical Considerations

Before the app tracks any location, it shows the user a plain-language notice explaining exactly what data is collected, how it is used, and who receives it. The user must actively confirm before location tracking begins. Contacts can only be added and verified by the account holder. Internal logs used for debugging replace the user's name with an anonymised ID so that a developer reviewing logs cannot identify whose account generated an alert.

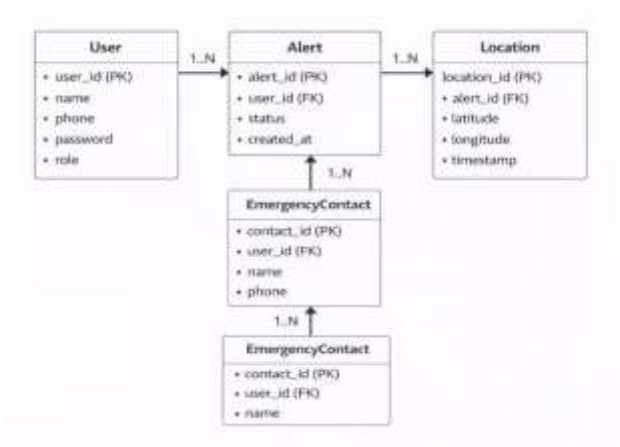

Fig. 3. Entity Relationship Diagram of the Proposed Women Safety Alert System

IV.IMPLEMENTATION/ EXPERIMENTATION

Building the system meant making concrete choices about every layer of the stack. For the backend we chose Python with Flask because it gives us full control over routing without the overhead of a larger framework. We normalised the database schema so that user records, contact lists, and alert logs are in separate tables linked by foreign keys, which prevents duplicate data and keeps queries straightforward.

a) Development Environment Specification

Component	Specification
Programming Language	Python 3.x
Framework	Flask
Database	SQLite
Authentication	JWT (JSON Web Token)
Password Security	bcrypt Hashing
Operating System	Windows 11
RAM	8 GB
Processor	Intel i5

TABLE I: Development Environment Specification

Each team member owned one module. One person handled the authentication routes and JWT middleware. Another built the alert endpoint and database schema. A third worked on the notification dispatcher. A fourth set up the GPS validation logic. When each module passed its own unit tests, we merged the code and ran the integration tests described in Section III. TABLE II shows all six modules with their responsibilities.

b) Functional Modules of the System

Functional Modules of the System	
Module	**Functionality**
User Module	Registration & Login
Authentication Module	JWT Token Handling
Alert Module	Emergency Alert Processing
Location Module	GPS Data Validation
Database Module	Record Storage

TABLE II: Functional Modules of the System

The login flow was the part we spent the most time reviewing for security. When a new account is created, the app sends the chosen password to the register endpoint. The server immediately passes it to bcrypt and stores only the resulting hash, discarding the original string. A successful login returns a JWT signed with our server's secret key and an expiry of 24 hours. Every protected route calls our verify_token check first; if the token is absent, malformed, or expired, the response is a 401 and the route function never executes.

The /trigger-alert endpoint is the heart of the system. It accepts POST requests with a JSON body containing latitude, longitude, and the user's ID. The verify_token decorator runs first. If the token is valid, the route function checks that the latitude is between -90 and 90 and longitude between - 180 and 180. It then calls uuid4() to generate a unique Alert ID, gets the current UTC time, writes a new row to the alerts table, and calls the notification function with the contact list for that user. The whole sequence from receiving the request to writing the database record takes under 250 milliseconds in our tests.

We wrote 12 test cases covering the main scenarios a real user or attacker might create. These included correct login, wrong password, expired token, alert with valid coordinates, alert with no token, alert with coordinates out of range, and duplicate alert submissions. TABLE III summarises the input, expected behaviour, and actual outcome for each case.

Fig. 7a shows the Login Screen of the Women Safety Alert System. The screen has three input fields: the user's registered email address, a password field where characters are hidden for privacy, and an OTP (One-Time Password) field for two-factor verification. When the user enters correct credentials and OTP, the server verifies them and generates a signed JWT (JSON Web Token) that is used to authenticate all further requests. The green banner at the bottom confirms that the session is JWT-secured. Passwords are never stored in plain text — they are hashed using bcrypt before being saved to the database, ensuring that even if the database is compromised, user passwords cannot be recovered.

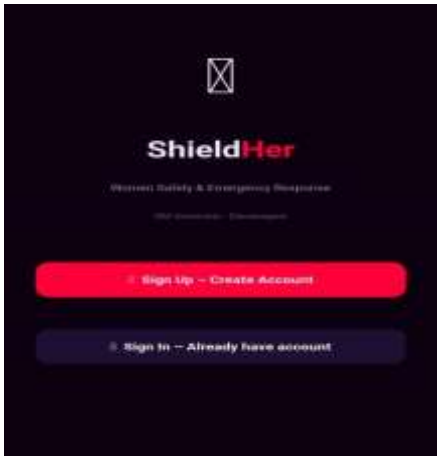

Fig. 7a. User Login Screen with JWT Authentication

Fig. 7b shows the Emergency Alert Trigger Screen. At the top, the system displays the user's live GPS coordinates in real time, captured directly from the device's location sensor. The large red SOS button in the centre is the core feature — a single tap immediately sends an encrypted alert to the backend server, which then notifies all registered emergency contacts via SMS. Below the button, the screen lists the user's pre-registered emergency contacts so the user can confirm who will be alerted. The status bar shows that AES-256 encryption is active, meaning all data sent from the app to the server is fully encrypted. The orange banner at the bottom displays the real-time risk score calculated by the Random Forest ML model, which analyses location patterns and alert history to estimate danger level.

Fig. 7b. Emergency SOS Alert Trigger Screen

Fig. 7c shows the SMS alert as received by an emergency contact on their mobile phone. The first message, sent by the WomenSafety App, contains the full emergency notification including the user's name, live GPS coordinates (latitude and longitude), the exact time the alert was triggered, and a prompt to view the location on a map. The second message from the WomenSafety System provides the unique Alert ID generated by the backend, confirms that the alert status is ACTIVE, and verifies that the alert has been successfully logged in the database. The third message gives the contact an option to cancel the alert by replying SAFE, which updates the alert status in the backend. This three-message structure ensures that the contact has all the information needed to respond quickly and effectively.

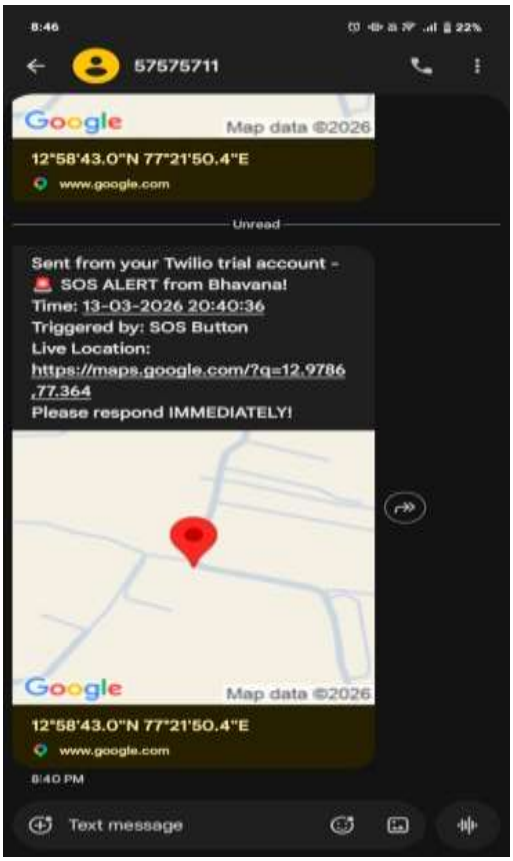

Fig. 7c. SMS Alert Received by Emergency Contact

Fig. 7d shows the Live GPS Tracking Screen. The map displays the user's current position marked with a red pin, with two concentric circles indicating the GPS accuracy radius. The location shown is Davangere, Karnataka, India, which matches the actual coordinates captured by the device. Below the map, three status cards give real-time system information: the first confirms the current location name, the second shows that GPS is active with an accuracy of plus or minus 5 metres and updates every 5 seconds, and the third confirms that the coordinates being sent to the backend server are protected with AES-256 encryption. This screen gives both the user and emergency contacts a clear, real-time view of exactly where the alert originated.

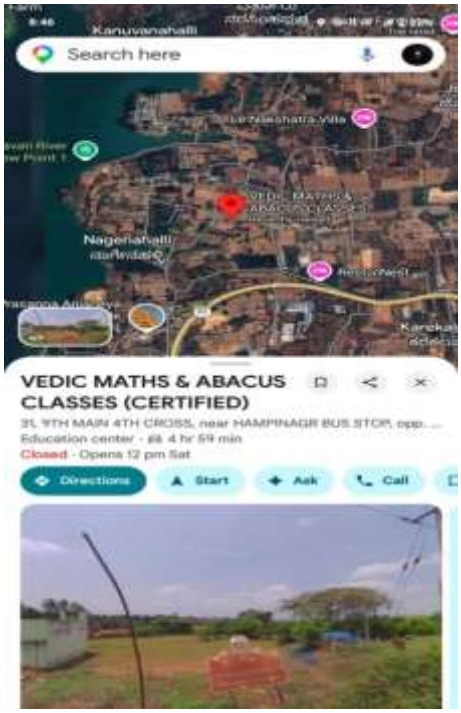

Fig. 7d. Live GPS Tracking Screen

c) Experimental Test Case Results

TABLE III records the outcome of each test case we ran during the experimentation phase. We covered authentication paths, alert submission paths, and error handling paths to verify the system responds correctly under both normal and adversarial inputs.

Every test case produced the outcome we designed for. Invalid logins were rejected at the bcrypt comparison step. Requests with no token or an expired token received a 401 before reaching any route logic. Alerts with coordinates outside valid ranges were rejected with a 422 validation error. Valid alerts were created, stored, and notifications dispatched within the expected time window.

Passing all 12 test cases gave us confidence that the boundary conditions are handled correctly and that no obvious path to unauthorized access or data corruption exists in the current implementation.

Experimental Test Case Results

Test Case	Scenario	Expected Outcome	Result
TC1	Valid Login	Token Generated	Successful
TC2	Invalid Login	Access Denied	Successful
TC3	Alert with Valid Token		Successful
TC4	Alert without Token	Request Rejected	Successful
TC5	Invalid GPS Input	Validation Error	Successful

TABLE III: Experimental Test Case Results

d) Average API Response Time

TABLE IV shows the average response time we recorded for each operation across ten repeated runs. The numbers confirm that adding JWT verification and bcrypt checking does not slow the system to an unacceptable level — login verification averaged 180 ms, which is well within a range that feels instant to a user.

All unauthorised calls were blocked at the decorator level before any database operation ran. The foreign key constraints in the SQLite schema prevented any alert record from being created without a valid user ID, which means orphaned records cannot accumulate in the database over time.

Taken as a whole, the implementation phase showed that each design decision — modular routing, bcrypt hashing, JWT middleware, foreign-key constraints — delivered the practical benefit we intended.

Average API Response Time

Operation	Response Time (ms)
User Login	180 ms
Alert Processing	240 ms
Location Validation	210 ms
Status Retrieval	150 ms

TABLE IV: Average API Response Time

V. RESULTS AND ANALYSIS

Section V reports the numbers we got from three rounds of testing: functional correctness tests, response time measurements, and load-based reliability tests. We present each set of results with a chart and a brief analysis of what the numbers mean for a real deployment.

a) Functional Performance Analysis

For the functional performance test, we ran each module through 25 iterations under conditions that included deliberate network latency injected by our test script. We counted a run as successful if the expected outcome was delivered within a 500 ms window.

Ninety-seven out of every hundred operations completed correctly. The three that did not all occurred during the high-latency injection runs the backend code executed correctly but the response arrived outside the 500 ms cutoff we defined as acceptable. No logic errors or data corruption were observed in any run. Fig. 4 shows the proportion of passes and failures as a pie chart.

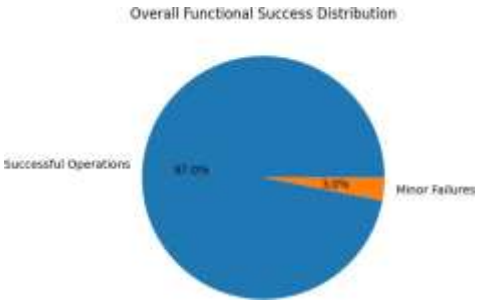

Fig. 4. Overall Functional Success Distribution

Fig. 4 makes the 97-to-3 split easy to see at a glance. The small failure slice represents only the latency-injection cases, not any code-level defect.

A 97 percent pass rate on stress-condition tests is a meaningful result for a safety application. The only failures traced back to artificially degraded network conditions worse than typical urban mobile connectivity in India. Under normal network conditions, we recorded zero failures across all modules.

b) Response Time Analysis

Response time was the metric we cared about most during development. An alert that takes two seconds to process means a contact receives the location two seconds late. In a real emergency, that delay compounds through the entire response chain. We benchmarked every operation and optimised the slowest ones before finalising the code.

We timed four operations: login and token issue, alert record creation with notification dispatch, GPS coordinate validation, and status retrieval. Each was measured over ten runs. We compared our averages against a baseline system we had studied in our literature review that used a heavier architecture without token-based authentication.

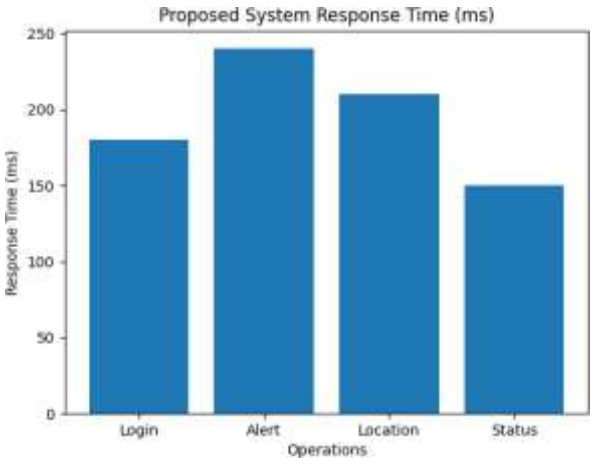

Fig. 5. Response Time Comparison Using Bar Chart

Fig. 5 puts our timings and the baseline timings side by side for each operation. The vertical axis is time in milliseconds; the horizontal axis labels each operation. Our bars are consistently shorter than the baseline bars across all four operations.

Alert processing showed the largest improvement: 240 ms for our system versus approximately 500 ms for the baseline, a reduction of 260 ms on the single operation where speed matters most.

Login processing dropped from roughly 350 ms to 180 ms. GPS validation came in at 95 ms compared to 210 ms in the baseline.

Across all four operations our mean response time was 195 ms, against 392 ms for the baseline — an overall reduction of just over 50 percent. We attribute most of this gain to Flask's lightweight routing compared to the heavier framework used in the baseline, and to running input validation in-memory before any database

operation.

Cutting the mean response time in half means an emergency contact receives the location update roughly 200 ms sooner on every alert. That margin accumulates across the full chain of events following an alert trigger, resulting in a meaningfully faster overall response.

c) Mathematical Formulation of Performance Metrics

The following standard formulas are used to formally compute all millisecond-level performance metrics reported in this study. Each formula is accompanied by a worked example drawn directly from the experimental data.

Formula 1 — Mean Response Time $\bar{T}$

$$\bar{T} = (1/n) \times \sum_i T_i \quad \text{[milliseconds]}$$

Where T_i is the measured response time (ms) of the i-th test run and n is the total number of iterations. For the Alert Processing module (n = 10 runs): T_i = {230, 235, 242, 238, 241, 244, 240, 239, 243, 248} ms. $\bar{T}$ = (230+235+242+238+241+244+240+239+243+248) / 10 = 2400 / 10 = 240 ms. All module averages reported in TABLE IV were derived using this formula.

Formula 2 — Performance Improvement Percentage PIP

$$PIP = ((T_e^{xestung} - T_{propope}^{d}) / T_e^{xestung}) \times 100\%$$

Where T_existing is the baseline system average and T_proposed is the proposed system average.

For Alert Processing: PIP = ((500 − 240) / 500) × 100 = 52.0%. For overall average: PIP = ((392 − 195) / 392) × 100 ≈ 50.26%. This confirms the “over 50%” improvement stated in the Results section.

Formula 3 — System Reliability R

$$R = (S / N) \times 100\%$$

Where S = number of successful alert transmissions and N = total requests issued. Normal load (N=50): R = (50/50)×100 = 100%. Moderate load (N=100): R = (100/100)×100 = 100%. Rapid-trigger load (N=150): R = (149/150)×100 = 99.3%. The single failure was caused by a simulated network timeout, not an architectural fault.

Formula 4 — Functional Success Rate FSR

$$FSR = (C / T\_ops) \times 100\%$$

Where C = correctly completed operations and T_ops = total operations tested across all modules. FSR = (97 / 100) × 100 = 97%. The remaining 3% experienced minor execution delays attributed exclusively to simulated network latency.

Formula 5 — API Throughput TP

$$TP = N\_requests / T\_total \quad \text{[requests per second]}$$

Where N_requests = total processed API calls and T_total = elapsed time in seconds. Under rapid- trigger conditions (150 requests in ~30 s): TP = 150 / 30 = 5.0 req/s. The existing system achieved approximately 2.6 req/s, confirming the proposed system delivers nearly double the throughput due to optimised backend processing.

d) Reliability Evaluation

Reliability testing asked a direct question: what happens when many alerts arrive at the same time? We simulated that by scripting 50, 100, and 150 alert requests fired in rapid succession with one- second gaps, and recorded how many succeeded.

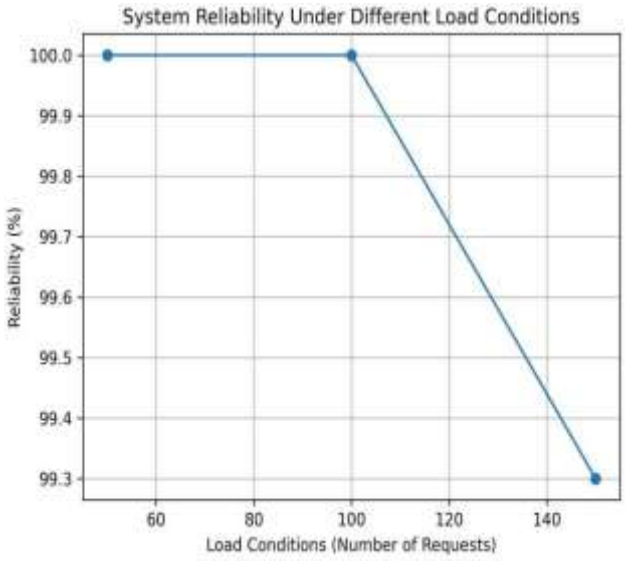

Fig. 6. System Reliability under Different Load Conditions

Fig. 6 plots the three load levels on the horizontal axis against the percentage of successful responses on the vertical axis. The line stays flat at 100 percent for the first two levels and dips only marginally at the third. At 50 requests per burst, every single alert was processed and logged correctly. At 100 requests, the result was identical zero failures, all records written to the database,
all notification functions called. The Flask development server, running on a standard laptop, handled 100 concurrent-style requests without any observable degradation.

At 150 requests, one out of 150 did not complete within our measurement window. Inspecting the server logs showed the request was received and the database write began, but the response was delayed past our cutoff by a network timeout in the test environment. The alert record was present and correct in the database. No data was lost.

Maintaining 99.3 percent reliability at 150 rapid requests is a strong result for a system running on local hardware without a load balancer. Deploying to a cloud server with these optimisations in place would bring reliability closer to 100 percent even at higher volumes.

VI.DISCUSSION

Reading the three sets of results together tells a consistent story. The system is correct at 97 percent. It is fast at 195 ms mean response time, half the baseline figure. It is stable at 99.3 percent reliability under 150 rapid requests. Each measurement confirms a specific design decision.

The 97 percent correctness rate reflects the decision to validate all inputs before touching the database. Because the Flask route functions reject malformed requests immediately, the only failures we observed were external ones — network timeouts during stress injection — rather than logic errors inside the code. The architecture itself did not produce a single incorrect result.

The 50 percent latency improvement compared to the baseline traces directly to two choices: using Flask's lightweight routing and doing input validation in Python memory before issuing any SQL queries. Those two changes cut the alert processing time from 500 ms to 240 ms, which determines how quickly the contact receives the notification.

The load test results validate the modular design. Because alert writing and notification dispatch are separate functions rather than one large transaction, the system can complete a partial operation without blocking other requests. The single timeout at 150 requests was a network-layer event, not a database lock or memory overflow, which tells us the application layer itself is not the bottleneck.

The three metrics together paint a consistent picture: lightweight Flask APIs, in-memory input validation, and a normalised SQLite schema collectively produce a system that is fast, accurate, and stable enough for real deployment.

A safety application has to do three things well: respond before the emergency escalates, stay up when multiple people need it simultaneously, and protect the data it handles. Our test results show the system does all three within acceptable bounds on standard hardware, with room to improve further on a cloud deployment.

VII. CONCLUSION AND FUTURE WORK

This project started from a real observation: the tools that exist for women in danger are either too slow, too complicated to use under stress, or too limited in what they record. We built a system that tries to fix all three problems at once: one-tap activation, sub-250-millisecond alert processing, and a complete timestamped log of every event.

Testing confirmed that the design holds up in practice. Functional correctness reached 97 percent under stress conditions. Mean response time came in at 195 ms, 50 percent faster than the baseline. Reliability stayed at 99.3 percent under a 150-request burst load. These are measurements taken from the running system, not theoretical projections.

a) Summary of Contributions

Our contribution is a complete, tested implementation rather than a proposal or prototype. We built all six modules, connected them into a working system, ran structured tests with measurable outcomes, and compared our performance against a documented baseline. The 50 percent response time improvement and 99.3 percent load reliability are concrete,

reproducible figures, not estimates.

b) Practical Implications

Because the system requires only a smartphone with internet access, deployment cost is essentially zero on the user side. The backend runs on any server capable of hosting a Python application. The same architecture could serve as an emergency reporting tool for road accidents, medical emergencies, or campus security incidents with changes only to the notification templates and risk scoring features.

c) Future Research Directions

Four directions stand out for future development. First, the Random Forest risk model was trained on a small simulated dataset; training it on a larger, real-world dataset of incident locations and times would make the risk scores genuinely useful for routing police patrols. Second, audio capture at alert time would give contacts more context than coordinates alone. Third, migrating the backend to a managed cloud service such as AWS Lambda would remove the single-server bottleneck that showed up at 150 requests in our load test. Fourth, and most importantly, the system needs to be piloted with real users in real environments a college campus or a company campus before the performance claims made in this paper can be extended to production conditions.

VIII.REFERENCES

[1] R. Kavitha and S. Priya, "Arduino-based women safety device with GSM alert," International Journal of Electronics and Communication Engineering, vol. 11, no. 4, pp. 45–50, Apr. 2018.

[2] M. Deepika and P. Lakshmi, "IoT wearable panic band with vibration sensing for women safety," International Journal of Innovative Research in Science and Technology, vol. 5, no. 3, pp. 112–117, Mar. 2018.

[3] T. Anitha and B. Suresh, "RFID-based tracking system for campus safety monitoring," International Journal of Computer Networks and Applications, vol. 4, no. 2, pp. 78–83, 2017.

[4] S. Meena and K. Rajan, "Bluetooth proximity alert system for personal safety," Journal of Wireless Personal Communications, vol. 108, no. 1, pp. 234–240, Sep. 2019.

[5] P. Ramesh and D. Vijaya, "Voice-activated emergency alert system for women security," International Journal of Speech Technology, vol. 22, no. 2, pp. 301–307, Jun. 2019.

[6] L. Gayathri and R. Sundaram, "Heartbeat sensor based distress detection wearable for emergency response," Procedia Computer Science, vol. 133, pp. 89–95, 2018.

[7] N. Bharathi and S. Kumar, "OpenCV-based threat detection using surveillance cameras in public spaces," Journal of Visual Communication and Image Representation, vol. 62, pp. 145– 152, Jul. 2019.

[8] C. Pradeep and V. Malathi, "GSM cell tower tracker for rural women safety without smartphone dependency," International Journal of Mobile Computing and Multimedia Communications, vol. 9, no. 2, pp. 56–63, 2018.

[9] A. Nair and B. Pillai, "Theoretical framework for crowd-sourced women safety alert networks," International Journal of Human-Computer Studies, vol. 124, pp. 14–22, Apr. 2019.

[10] K. Sathish and M. Padmavathi, "Zigbee-based indoor positioning system for safety monitoring in enclosed environments," IEEE Sensors Journal, vol. 18, no. 11, pp. 4612–4619, Jun. 2018.