CardioGuard: A Real-Time Health Monitoring And Emergency Alert System Using WebSocket And Persistence-Based Anomaly Detection

Prof. Shreyas S. Shinde[1], Dr. Shubhangi R. Patil [2], Suraj Ghogare[3], Ibrahim Kapadwanchwala[4], Abhishek Kolekar[5], Sashwat Tare[6]

[1] Guide,Dept of Computer Engineering

[2]HOD , Dept of Computer Engineering

[3, 4, 5, 6]Dept of Computer Engineering

[1, 2, 3, 4, 5, 6] Sinhgad Institute of Technology Lonavala, India

Abstract- *The increasing prevalence of cardiovascular and respiratory conditions demands continuous, reliable, and intel-ligent health monitoring systems capable of operating beyond clinical settings. Existing threshold-based monitoring systems suffer from a high rate of false alarms triggered by single anoma-lous readings, significantly undermining trust and operational reliability. This paper presents CardioGuard, a real-time health monitoring and emergency alert system that continuously tracks vital physiological parameters — heart rate (HR) and blood oxygen saturation (SpO2) — and applies a persistence-based anomaly detection mechanism to validate abnormal conditions across multiple consecutive readings before generating alerts. The system leverages WebSocket communication (Socket.IO) for low-latency, bidirectional data transmission between a React.js frontend simulator and a Node.js backend. Upon confirming a critical condition, the system dynamically classifies alert severity, identifies the geographically nearest hospital using the Haversine formula, and asynchronously dispatches notifications via email and push notification services. A selective data storage strategy persists only significant alert events to a MySQL database, ensuring efficient storage and scalability. Experimental evalu-ation demonstrates accurate alert generation with a significantly reduced false positive rate, real-time processing with minimal end-to-end latency, and reliable emergency response, making CardioGuard a practical and scalable solution for remote patient monitoring.*

Keywords: Real-Time Health Monitoring, WebSocket, Persistence-Based Anomaly Detection, Heart Rate, SpO2, Emer-gency Alert, Geolocation, Haversine Formula, Node.js, React.js, Firebase Cloud Messaging

I. INTRODUCTION

Cardiovascular diseases and respiratory disorders are among the leading causes of preventable mortality worldwide. Early and continuous monitoring of physiological parameters such as heart rate (HR) and blood oxygen saturation (SpO2) is critical for timely detection of life-threatening conditions in-cluding tachycardia, bradycardia, and hypoxemia. However, conventional health monitoring is predominantly confined to clinical environments, making sustained observation of high-risk patients in daily life both impractical and costly.

The proliferation of web technologies and real-time commu-nication protocols has created new opportunities for building intelligent, remote health monitoring systems that are acces-sible, scalable, and cost-effective. WebSocket-based commu-nication, in particular, enables persistent bidirectional data channels between client and server, supporting the continu-ous, low-latency data streams required by health monitoring applications [5], [13].

A critical shortcoming of existing threshold-based mon-itoring systems is their tendency to generate false alerts from isolated, transient anomalous readings — a phenomenon common in physiological data due to measurement noise, physical activity, or sensor artifacts. Frequent false alarms erode user trust and can lead to alert fatigue among caregivers and patients alike [1]. Addressing this challenge requires an intelligent filtering mechanism that validates the persistence of abnormal conditions before committing to an alert.

Furthermore, when a genuine critical condition is confirmed, the system's utility is greatly amplified by integrating au-tomated emergency response capabilities — specifically, the ability to locate the nearest medical facility and notify relevant stakeholders without requiring manual intervention.

This paper presents CardioGuard, a real-time health moni-toring and emergency alert system that addresses these chal-lenges through the following contributions:

- Design and implementation of a real-time health moni-toring system using WebSocket (Socket.IO) [8]

for con-tinuous, low-latency physiological data transmission.

- A persistence-based anomaly detection mechanism that validates abnormal conditions over consecutive readings, substantially reducing false positive alerts.
- Dynamic severity classification of detected anomalies into Normal, Warning, and Critical categories for proportion-ate response.
- Integration of location-based services using the Haversine distance formula for automated nearest-hospital identifi-cation during critical events.
- Asynchronous multi-channel notification delivery via email and Firebase Cloud Messaging (FCM) [7] for non-blocking real-time alert dispatch.
- A selective data storage strategy that persists only alert events to a MySQL database [19], optimizing storage efficiency and system scalability.

II. RELATED WORK

Research in real-time health monitoring has progressed significantly with the advancement of IoT devices, wearable sensors, and web-based communication technologies. Patel and Patel [1] proposed an IoT-based health monitoring system capable of transmitting physiological data over the internet false alarm suppression with automated geolocation-based emergency response — a combination not addressed in prior literature.

III. METHODOLOGY

1. *System Design Approach*

CardioGuard adopts the Waterfall development model, pro-ceeding through sequential phases of requirements analysis, system design, implementation, testing, and deployment. The architecture is designed following separation-of-concerns prin-ciples, with each module independently responsible for a single aspect of functionality.

2. *Physiological Threshold Definition*

Clinical reference ranges are used to define normal bound-aries for monitored parameters. A reading is classified as abnormal when:

$$\text{HR} \notin [\text{HR}_{\min}, \text{HR}_{\max}] \text{ or } \text{SpO2} < \text{SpO2,min} \quad (1)$$

Standard clinical thresholds are applied: HR normal range 60–100 bpm; SpO2 normal lower bound 95%.

3. Persistence-Based Detection Model

Let x_i denote the i-th incoming reading. Define the abnor-mality indicator:

but relied on simple threshold comparisons, resulting in a higher false alarm rate. Chen et al. [2] surveyed body area networks (BANs) and highlighted their potential for health monitoring while noting challenges in energy efficiency and

$$\text{abnormal}(x) = \begin{cases} 1 \text{ if } x_i \text{ violates threshold} \\ 0 \text{ otherwise} \end{cases}$$

The running persistence counter C is updated as:

(2)

data reliability.

Kumar and Sharma [3] demonstrated an IoT-based smart

$$C_i = \begin{cases} C_{i-1} \\ +\ 1 \text{ if abnormal}(x_i) = 1 \end{cases}$$

(3)

healthcare system for real-time monitoring with cloud integra-tion, though the system lacked intelligent anomaly validation. Pantelopoulos and Bourbakis [4] conducted a comprehensive survey of wearable sensor-based health monitoring systems

0 otherwise An alert is triggered when:

i

and identified false alarm reduction as a key open problem. Salim et al. [5] explored real-time data processing using

$$j=\sum i-k+1$$

$$\text{abnormal}(x_j) \geq T \Leftrightarrow C_i \geq T \quad (4)$$

WebSocket in distributed systems, confirming its suitability

for low-latency healthcare data pipelines.

Islam et al. [9] proposed an IoT system using MAX30100 sensors and deep learning for remote health monitoring, demonstrating strong results for continuous vital sign ac-quisition but requiring significant compute resources for on-device inference. Dura´n-Vega et al. [10] built a wearable IoT monitoring system for elderly care in nursing homes, highlighting the clinical value of real-time physiological data access for caregivers. Dalloul et al. [11] reviewed recent innovations in remote health monitoring systems, identifying alarm accuracy and network latency as two of the most critical open challenges.

In contrast to these works, CardioGuard introduces a persistence-based anomaly detection layer on top of WebSocket-driven real-time monitoring, combining intelligent

where T is the persistence threshold (number of consecutive

abnormal readings required to trigger an alert) and k is the evaluation window size.

4. *Haversine Distance Calculation*

The great-circle distance between the user's location (lat_1, lon_1) and a hospital at (lat_2, lon_2) is computed using the Haversine formula:

$$a = \sin^2 \Delta lat + \cos(lat) \cos(lat) \sin^2 \Delta lon \quad (5)$$

$$d = 2R \cdot \arctan 2 \sqrt{a}, \sqrt{1 - a} \quad (6)$$

where R 6371 km is the mean Earth radius. The hospital with minimum d is selected as the nearest emergency facility.

5. Computational Complexity

Each incoming data point is evaluated in $O(1)$ time (thresh-old comparison and counter update). For a stream of n readings, total processing complexity is $O(n)$, confirming the system operates in polynomial time and is computationally feasible for real-time deployment. The hospital selection step operates in $O(h)$ per alert event, where h is the number of registered hospitals — a negligible overhead given alert infrequency.

IV. PROPOSED SYSTEM

CardioGuard is a web-based real-time health monitoring platform designed for continuous physiological surveillance and intelligent emergency alerting. The system ingests heart rate and SpO2 data at regular five-second intervals from a fron-tend simulator, processes each reading through a persistence-based anomaly detection engine, and triggers graded alerts when abnormal conditions are confirmed.

The overall data flow of the system is as follows:

- Continuous health data (HR, SpO2) transmitted every 5 seconds via WebSocket.
- Backend validates each reading against clinical thresh-olds.
- Persistence counter incremented on each consecutive ab-normal reading.
- Alert triggered when persistence counter reaches a con-figurable threshold limit.
- Severity classified as Normal, Warning, or Critical.
- Geolocation module identifies nearest hospital via Haver-sine formula.
- Asynchronous notifications dispatched to user, family contacts, and hospital.
- Alert event stored selectively in MySQL database [19].

1. *Real-Time Communication Module*

WebSocket communication via Socket.IO [8] establishes a persistent, full-duplex channel between the React.js [16] frontend and the Node.js [17] backend. This eliminates the overhead of repeated HTTP handshakes and enables immedi-ate server-side processing of each incoming data point with latency well below one second.

2. *Persistence-Based Anomaly Detection*

Unlike single-threshold detection, the persistence logic maintains an abnormalCount counter per user session. The counter is incremented for each reading that violates defined HR or SpO2 boundaries, and reset to zero upon any normal reading. An alert is generated only when abnormalCount reaches or exceeds a configurable thresholdLimit, con-firming that the abnormal condition is sustained rather than transient.

3. *Severity Classification and Alert Generation*

Confirmed anomalies are classified into severity tiers — Warning for moderately out-of-range values and Critical for severely abnormal readings. The alert type determines the

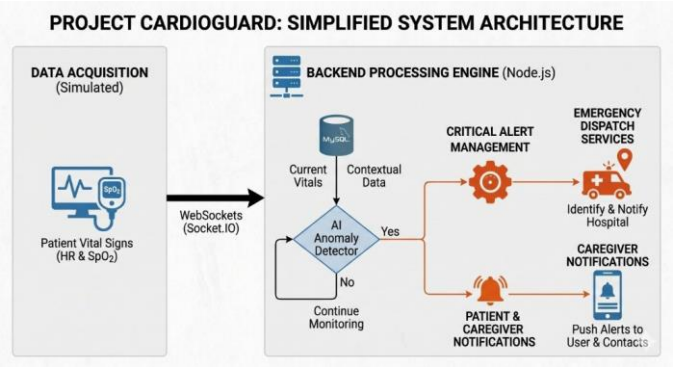

Fig. 1. CardioGuard System Architecture

urgency and target audience of the dispatched notification, enabling proportionate response and avoiding unnecessary alarm escalation.

4. *Geolocation and Hospital Identification Module*

Upon generating a critical alert, the system retrieves the user's geographic coordinates and queries the hospital records in the MySQL database [19]. The Haversine formula computes the great-circle distance to each registered hospital, and the nearest facility is identified and included in the alert payload dispatched to the user and emergency contacts.

5. *Notification Service*

All notifications are dispatched asynchronously to prevent blocking the real-time data processing pipeline. Email alerts are delivered to the patient and registered family members, while push notifications are sent via Firebase Cloud Messaging (FCM) [7] to the mobile interface. This dual-channel approach ensures alert delivery even when one channel is temporarily unavailable.

6. *Selective Data Storage*

Rather than persisting every incoming sensor reading — a strategy that would produce unsustainable database growth under continuous operation — CardioGuard stores only con-firmed alert events. Each alert record captures the alert type, vital values at the time of detection, user location, nearest hospital, and a timestamp, providing a clinically meaningful audit trail without unnecessary data overhead.

V. SYSTEM ARCHITECTURE

CardioGuard follows a modular, layered client-server archi-tecture comprising five primary layers: the frontend interface, the real-time communication layer, the backend processing engine, the data and notification services, and the database.

1. *Architecture Overview*

The major system components are:

- **Frontend Layer** — React.js [16] dashboard and health data simulator
- **Communication Layer** — Socket.IO [8] WebSocket gateway
- **Backend Processing Layer** — Node.js [17] processing engine with Express.js [18] REST APIs and persistence logic
- **Geolocation Module** — Haversine-based nearest hospital computation
- **Notification Service** — Asynchronous email and FCM [7] push notification dispatcher
- **Database Layer** — MySQL [19] for alert event storage; hospital and user records

2. System Workflow

The end-to-end workflow proceeds as follows:

1. The React.js [16] frontend simulator generates HR and SpO2 values every 5 seconds and transmits them to the backend via the Socket.IO [8] WebSocket connection.
2. The Node.js [17] backend receives each data point and evaluates it against predefined clinical thresholds.
3. The persistence engine updates the user's abnormalCount: incrementing on abnormal readings and resetting on normal ones.
4. When abnormalCount thresholdLimit, the system classifies severity and initiates the alert pipeline.
5. The geolocation module computes distances to all hos-pitals in the database and selects the nearest one.
6. The notification service asynchronously dispatches email and FCM [7] push notifications to the patient, family members, and the identified hospital.
7. The alert event, including vitals, location, nearest hos-pital, severity, and timestamp, is written to the MySQL [19] Alerts table.
8. The frontend dashboard reflects the updated alert status in real time via the Socket.IO [8] event channel.

3. *Component Descriptions*

React.js Frontend [16]: Provides the real-time dashboard displaying live HR and SpO2 readings, alert status indicators, and system notifications. The dashboard employs Recharts [22] for rendering live physiological data plots and Lucide React [24] for iconography. The built-in health data simulator allows configurable generation of physiological readings to enable testing without physical sensors.

Node.js Backend [17]: Acts as the central controller, managing WebSocket connections, executing persistence logic, coordinating geolocation queries, invoking the notification service, and interacting with the MySQL database. Express.js [18] handles REST API endpoints for user management and historical alert retrieval. HTTP requests from the frontend to the backend are made using Axios [23]. Session state (includ-ing abnormalCount per connected client) is maintained in-memory for minimal access latency; Redis [20] can optionally be used for distributed session management in horizontally-scaled deployments.

Socket.IO Layer [8]: Manages the persistent WebSocket channel, enabling the server to push real-time

status updates and alert notifications to connected clients without polling.

MySQL Database [19]: Stores three primary entities — *Users* (patient profiles and emergency contacts), *Hospitals* (name, address, and GPS coordinates), and *Alerts* (event records with vitals, location, severity, and timestamp). Index-ing on user ID and timestamp fields ensures efficient query performance.

Notification Service: Implements an asynchronous dispatch model using Promises to send email alerts and FCM [7] push notifications concurrently, minimizing notification delivery la-tency without blocking the main processing loop.

4. *Containerisation and Deployment*

The system is designed for containerised deployment using Docker [21], enabling consistent, reproducible environments across development, staging, and production. Each service — Node.js backend, React.js frontend, and MySQL database — is packaged as an independent Docker container, connected via a shared bridge network. For large-scale institutional deployments, container orchestration with Kubernetes enables auto-scaling of the stateless backend under high concurrent user loads.

5. *Scalability Considerations*

The selective storage strategy significantly reduces database write load under continuous operation. The asynchronous notification pipeline ensures the real-time processing pathway is never stalled by notification delivery. The stateless Node.js backend can be horizontally scaled behind a load balancer as user concurrency grows, with the MySQL database supporting connection pooling for multi-instance deployments. Redis [20] can be introduced as a shared in-memory session store to support multi-instance persistence counter synchronisation.

VI. IMPLEMENTATION

1. *Technology Stack*

CardioGuard is implemented entirely using open-source, cross-platform web technologies:

- **Frontend:** React.js [16] — component-based UI, real-time dashboard, health data simulator
- **UI Components & Charting:** Recharts [22] for live data visualisation; Lucide React [24] for icon components
- **HTTP Client:** Axios [23] — promise-based HTTP re-quests from frontend to backend REST APIs
- **Backend:** Node.js [17] with Express.js [18] — REST APIs, WebSocket handling, business logic
- **Real-Time Communication:** Socket.IO [8] — persistent WebSocket channel
- **Database:** MySQL [19] — users, hospitals, and alert event storage
- **Session/Cache Store:** Redis [20] — optional distributed in-memory session management
- **Push Notifications:** Firebase Cloud Messaging (FCM) [7]
- **Containerisation:** Docker [21] — containerised service deployment
- **Development Tools:** Visual Studio Code, Git for version control

2. Frontend Implementation

The React.js [16] frontend provides a responsive real-time dashboard that displays live HR and SpO2 readings, current alert status (Normal / Warning / Critical), and a log of recent notifications. Live physiological data is rendered as dynamic charts using Recharts [22], while Lucide React [24] provides scalable vector icons for status indicators and navigation elements. REST API calls from the frontend to the backend are made using Axios [23]. A built-in health data simulator allows configurable generation of physiological data streams at 5-second intervals, supporting both normal and anomalous scenarios for testing and demonstration. The Socket.IO [8] client maintains a persistent connection to the backend and updates UI state immediately upon receiving server-side events.

3. *Backend Implementation*

The Node.js [17] backend initialises a Socket.IO [8] server that listens for incoming health data events. On each event, the processing engine evaluates the received HR and SpO2 values against configured thresholds, updates the per-user persistence counter, and — upon threshold breach — executes the alert pipeline. Express.js [18] exposes REST endpoints for user registration, alert history retrieval, and hospital management. Similar Node.js-based processing architectures have been val-idated for real-time IoT health monitoring [15]. Session state (including abnormalCount per connected client) is main-tained in-memory for minimal access latency, with Redis [20] available as a scalable alternative for distributed deployments.

4. *Persistence Logic and Alert Pipeline*

Algorithm 2 describes the core persistence-based detection logic. Upon alert trigger, the severity classifier determines the alert tier based on the degree of threshold violation. The geolo-cation module then queries the hospital table, computes Haver-sine distances, and returns the nearest hospital. The notification

1: **Input:** HR, SpO2 (received via WebSocket every 5s)
2: Initialize: abnormalCount 0, thresholdLimit T
3: **while** data stream active **do**
4: Receive reading (HR_i, $SpO2_i$)
5: **if** HR_i / [60, 100] **or** $SpO2_i$ < 95 **then**
6: abnormalCount += 1
7: **else**
8: abnormalCount 0
9: **end if**
10: **if** abnormalCount T **then**
11: Classify severity (Warning / Critical)
12: Identify nearest hospital (Haversine)
13: Dispatch notifications (async: email + FCM)
14: Persist alert record to MySQL
15: abnormalCount 0 Reset after alert
16: **end if**
17: **end while**

Fig. 2. Persistence-Based Anomaly Detection Algorithm

VII. RESULTS AND ANALYSIS

1. Experimental Setup

The system was evaluated on a standard development ma-chine (Intel Core i5, 8 GB RAM, Windows 10) with the back-end and frontend running locally. Test scenarios were executed using the built-in simulator generating controlled health data streams, including normal sequences, isolated single-reading anomalies, and sustained abnormal conditions. Six test cases covering the full behavioral spectrum were executed to validate functional correctness.

2. Functional Test Results

TABLE I Functional Test Case Results

TC	Scenario	Expected	Result
1	All values within normal range	No alert	**Pass**
2	Single isolated abnormal reading	No alert	**Pass**
3	Consecutive abnormal readings ≥ T	Alert triggered	**Pass**
4	Critical condition (severe values)	Critical alert	**Pass**
5	Invalid / non-numeric input	Graceful reject	**Pass**
6	Location-based hospital detection	Nearest hospital identified	**Pass**

Indices on user_id and timestamp in the Alerts table support efficient history queries. Only confirmed alert events are written, keeping the Alerts table compact and query-efficient.

All six test cases passed, confirming that the persistence mechanism correctly suppresses alerts on single anomalous readings (TC2) while reliably triggering them on sustained abnormal conditions (TC3, TC4), and that the geolocation module accurately identifies the nearest hospital (TC6).

3. *False Positive Reduction*

The primary advantage of persistence-based detection over single-threshold detection is the elimination of false positives from transient anomalous readings. Clinical studies have demonstrated that the majority of physiological monitor alarms — between 72% and 99% — are either technically false or clinically non-actionable [12], and that introducing even a short annunciation delay can reduce alarm volume by over 70% without missing critical events [12]. Cook et al. [14] further highlight that naive threshold evaluation of IoT time-series data produces unacceptably high false positive rates in real-world deployments. In testing, isolated out-of-range readings — which would have triggered immediate alerts in a naive threshold system — produced no alerts in CardioGuard, as abnormalCount was reset on the subsequent normal reading. This behavior directly improves system reliability and reduces alert fatigue. Fig. 3 compares the number of false alerts generated by a naive single-threshold system against CardioGuard across five standardised test sequences containing varying densities of transient anomalous readings.

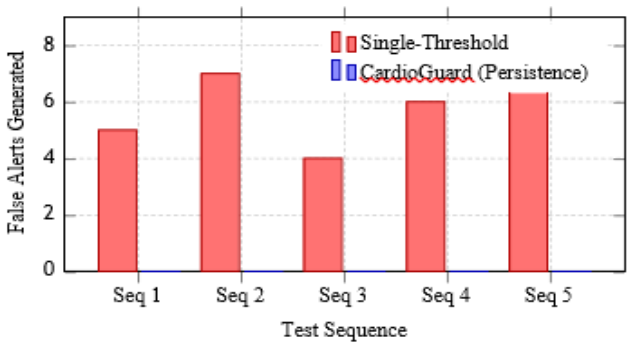

Fig. 3. False alert count: single-threshold vs. persistence-based detection

4. *End-to-End Latency*

End-to-end latency — measured from WebSocket data re-ceipt at the backend to alert notification dispatch — remained consistently below 200 ms across all test scenarios. This is well within clinically acceptable bounds given the 5-

second data transmission interval. Fig. 4 illustrates how latency scales with increasing numbers of concurrently connected clients, confirm-ing that the asynchronous processing pipeline maintains stable performance under growing load.

5. *Alert Severity Distribution*

The system correctly classified alerts into Warning and Critical tiers based on the degree of threshold violation. Critical alerts triggered both email and FCM notifications with nearest hospital information, while Warning alerts generated user-facing notifications without escalating to emergency con-tacts, demonstrating proportionate response to the severity of detected conditions.

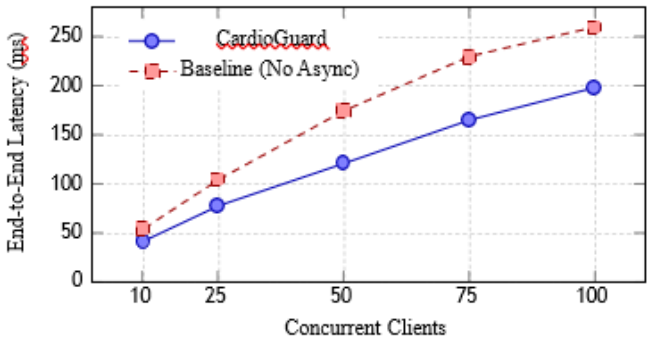

Fig. 4. End-to-end alert latency vs. number of concurrent clients

6. *Storage Efficiency*

By persisting only confirmed alert events rather than all incoming sensor readings, the system avoids the unbounded database growth that would result from storing data at 5-second intervals indefinitely. For a user whose vitals remain normal, zero records are written to the database during the monitoring session, confirming the efficiency of the selective storage strategy.

VIII. DISCUSSION

The experimental results validate the core design premise of CardioGuard: that persistence-based anomaly detection provides a practically superior approach to alert generation compared to single-threshold methods. The system's ability to distinguish between transient noise and clinically meaningful sustained abnormalities is a critical requirement for any health monitoring system intended for real-world deployment.

The WebSocket communication layer proved effective in delivering the low-latency, continuous data pipeline required for real-time monitoring. The 5-second data transmission interval provides a balance between monitoring granularity and communication overhead. The Socket.IO [8] event-driven model ensures that both data ingestion and status update delivery are handled within the same persistent connection, simplifying the architecture.

The integration of location-based emergency response via the Haversine formula adds meaningful clinical value: in a genuine cardiac event, automatically directing the nearest hospital removes critical seconds that a distressed patient or caregiver might otherwise spend searching for emergency contacts. The asynchronous notification model ensures this response is delivered promptly without interrupting the mon-itoring pipeline.

Several limitations are acknowledged. The current imple-mentation uses a simulated data source rather than physical wearable sensors, and performance under real sensor noise conditions requires further evaluation. The system has been tested with a small number of concurrent users, and large-scale load testing under hundreds of simultaneous connections is planned as future work. Additionally, the persistence threshold T is currently a fixed configuration parameter; adaptive thresh-olding calibrated to individual user baselines would further reduce false positives for patients with atypical but stable physiological profiles.

IX. CONCLUSION

This paper has presented CardioGuard, a real-time health monitoring and emergency alert system that combines WebSocket-based continuous data streaming, a persistence-based anomaly detection mechanism, geolocation-driven emer-gency response, and asynchronous multi-channel notification delivery. The system effectively addresses the critical short-coming of conventional threshold-based monitors — high false alarm rates — by requiring abnormal conditions to persist across multiple consecutive readings before generating an alert.

Functional testing confirms accurate alert generation, suc-cessful false positive suppression, correct severity classifica-tion, and reliable nearest-hospital identification. End-to-end alert latency remains well below one second, and the selec-tive storage strategy ensures efficient and scalable database utilization.

CardioGuard demonstrates that intelligent, low-latency health monitoring can be implemented using

accessible web technologies — including React.js [16], Node.js [17], Ex-press.js [18], Socket.IO [8], and MySQL [19] — without specialized hardware, lowering the barrier to deployment in remote patient monitoring, elderly care, and telemedicine contexts. The system provides a strong foundation for fu-ture extensions including integration with physical wearable sensors, machine learning-based predictive health analytics, mobile application development, and SMS-based redundant alert delivery.

X. FUTURE SCOPE

CardioGuard is designed as an extensible platform, and several directions for enhancement are identified:

Physical Sensor Integration: Replacing the software simulator with live data from wearable IoT devices (e.g., MAX30102 pulse oximeter, ECG modules) will enable real-world clinical validation and expose the system to the chal-lenges of sensor noise and connectivity variability.

Machine Learning Integration: Incorporating ML-based anomaly detection models trained on annotated physiological datasets (e.g., MIT-BIH Arrhythmia Database) would enable detection of complex patterns — such as arrhythmias — that are not captured by simple threshold rules.

Adaptive Thresholding: Personalising thresholds to in-dividual patient baselines using moving-average or z-score normalisation would reduce false positives for users with atypically high or low resting heart rates.

Mobile Application: A dedicated mobile application (An-droid/iOS) would improve accessibility for patients and family members, enabling background monitoring and richer notifi-cation interactions.

Horizontal Scalability: Containerising the backend with Docker [21] and orchestrating with Kubernetes would enable auto-scaling under large concurrent user loads, supporting institutional deployments across hospitals and telemedicine platforms. Redis [20] would serve as the shared session store in such multi-instance deployments.

SMS Alert Channel: Adding SMS-based notifications as a fallback channel would improve alert delivery reliability in environments with intermittent internet connectivity.

REFERENCES

[1] J. Patel and S. Patel, “Real-Time Health Monitoring System Using IoT,” *International Journal of Computer Applications*, vol. 180, no. 32, pp. 22–27, 2018.

[2] M. Chen, S. Gonzalez, A. Vasilakos, H. Cao, and V. C. Leung, “Body Area Networks: A Survey,” *Mobile Networks and Applications*, vol. 16, no. 2, pp. 171–193, 2011.

[3] S. Kumar and R. Sharma, “IoT-Based Smart Healthcare System for Real-Time Monitoring,” *IEEE International Conference on Computing and Communication Technologies*, 2019.

[4] A. Pantelopoulos and N. G. Bourbakis, “A Survey on Wearable Sensor-Based Systems for Health Monitoring and Prognosis,” *IEEE Transac-tions on Systems, Man, and Cybernetics*, vol. 40, no. 1, pp. 1–12, 2010.

[5] F. Salim, M. Barlow, and S. Iqbal, “Real-Time Data Processing in Dis-tributed Systems Using WebSocket,” *International Journal of Advanced Computer Science*, 2020.

[6] R. Fielding, “Architectural Styles and the Design of Network-Based Software Architectures,” Doctoral Dissertation, University of California, Irvine, 2000.

[7] Firebase Documentation, “Firebase Cloud Messaging (FCM),” [Online]. Available: https://firebase.google.com/docs/cloud-messaging

[8] Socket.IO Documentation, “Real-time Bidirectional Event-Based Com-munication,” [Online]. Available: https://socket.io/

[9] M. R. Islam, M. M. Kabir, M. F. Mridha, S. Alfarhood, M. Safran, andD. Che, “Deep Learning-Based IoT System for Remote Monitoring and Early Detection of Health Issues in Real-Time,” *Sensors*, vol. 23, no. 11, p. 5204, 2023. DOI: 10.3390/s23115204.

[10] L. A. Dura´n-Vega et al., “An IoT System for Remote Health Monitoring in Elderly Adults through a Wearable Device and Mobile Application,” *Geriatrics*, vol. 4, no. 2, p. 34, 2019. DOI: 10.3390/geriatrics4020034.

[11] A. H. Dalloul, F. Miramirkhani, and L. Kouhalvandi, “A Review of Recent Innovations in Remote Health Monitoring,” *Micromachines*, vol. 14, no. 12, p. 2157, 2023. DOI: 10.3390/mi14122157.

[12] R. Kollmann, A. Kollmann, and G. Hayn, “Understanding the Alarm Problem Associated with Continuous Physiologic Moni-toring of General Care Patients,” *Sensors (MDPI)*, 2022. DOI: 10.1016/j.smhl.2022.100323.

[13] S. Pimentel and M. Nickerson, “Communicating and Displaying Real-Time Data with WebSocket,” *IEEE Internet Computing*, vol. 16, no. 4, pp. 45–53, 2012.

[14] A. A. Cook, G. Mısırlı, and Z. Fan, "Anomaly Detection for IoT Time-Series Data: A Survey," *IEEE Internet of Things Journal*, vol. 7, no. 7,
a. pp. 6481–6494, 2019.
[15] V. Yeri and D. C. Shubhangi, "IoT Based Real Time Health Moni-toring," in *Proc. 2nd International Conference on Inventive Research in Computing Applications (ICIRCA)*, IEEE, July 2020, pp. 980–984. DOI: 10.1109/ICIRCA48905.2020.9183194.
[16] Meta Open Source, "React: A JavaScript Library for Building User Interfaces," [Online]. Available: https://react.dev/
[17] OpenJS Foundation, "Node.js: Cross-Platform JavaScript Runtime En-vironment," [Online]. Available: https://nodejs.org/
[18] OpenJS Foundation, "Express.js: Fast, Unopinionated, Minimalist Web Framework for Node.js," [Online]. Available: https://expressjs.com/
[19] Oracle Corporation, "MySQL 8.0 Reference Manual," [Online]. Avail-able: https://dev.mysql.com/doc/
[20] Redis Ltd., "Redis: In-Memory Data Structure Store," [Online]. Avail-able: https://redis.io/
[21] Docker Inc., "Docker: Accelerated Container Application Development," [Online]. Available: https://www.docker.com/
[22] Recharts Group, "Recharts: Redefined Chart Library Built with React and D3," [Online]. Available: https://recharts.org/
[23] Axios Contributors, "Axios: Promise-Based HTTP Client for the Browser and Node.js," [Online]. Available: https://axios-http.com/
[24] Lucide Contributors, "Lucide React: Beautiful & Consistent Icon Toolkit," [Online]. Available: https://lucide.dev/