UDC [004.774:615.015.38](043.2)

I.S. Kovalchuk
O.M. Reyda

METHODS FOR IMPLEMENTING MEDICATION REMINDER SYSTEMS IN WEB-BASED APPLICATIONS

Vinnytsia National Technical University

Анотація

У тезах розглядаються методи реалізації систем нагадування про прийом медикаментів у веб-орієнтованих застосунках. Проаналізовано та порівняно основні канали доставки сповіщень – email, браузерні push-сповіщення, SMS та внутрішні оповіщення застосунку. Детально розглянуто архітектурні підходи до планування задач: від простих cron-механізмів до повноцінних систем черг повідомлень на основі BullMQ та Redis із підтримкою повторних спроб та горизонтального масштабування. Описано рекомендовану трирівневу архітектуру системи та особливості інтеграції компонента Scheduler із диспетчером сповіщень. Розглянуто питання захисту персональних медичних даних у процесі доставки сповіщень та підходи до класифікації статусу прийому препаратів.

Ключові слова: веб-застосунок, нагадування про ліки, push-сповіщення, планувальник задач, черга повідомлень, Node.js, Redis, прихильність до лікування.

Abstract

This paper examines methods for implementing medication reminder systems in web-based applications. The principal notification delivery channels – email, browser push notifications, SMS, and in-app alerts – are analyzed and compared. Architectural approaches to task scheduling are reviewed in detail, ranging from simple cron-based mechanisms to fully-featured message queue systems built on BullMQ and Redis with retry support and horizontal scalability. The recommended three-tier architecture for such a system is described, along with the integration of the Scheduler component with the notification dispatcher. Security considerations for personal health data during the notification delivery pipeline are discussed, and approaches to dose status classification are outlined.

Keywords: web application, medication reminders, push notifications, task scheduler, message queue, Node.js, Redis, medication adherence.

Introduction

Medication non-adherence remains one of the most critical challenges in contemporary healthcare. The World Health Organization estimates that approximately 50% of patients with chronic diseases do not follow their prescribed treatment regimens [1]. This problem leads to disease progression, treatment failure, increased hospitalization rates, and significant economic burden on health systems worldwide. Non-adherence is especially pronounced among elderly patients and those managing complex multi-drug schedules that require taking several different medications at different times throughout the day.

Digital health technologies represent a promising approach to improving medication adherence. Smartphone applications and web-based platforms can provide automated, timely reminders that replace unreliable paper-based records and memory alone. A systematic review by Thakkar et al. [2] demonstrated that SMS and app-based reminder interventions increased medication adherence by 17–21% compared to control groups across multiple chronic disease categories, including hypertension, diabetes, and HIV treatment.

Web-based applications, in particular, offer significant advantages over native mobile applications in terms of accessibility and cross-device reach. A web application requires no installation, runs on any modern browser, and is immediately accessible on desktop computers, tablets, and smartphones alike. This universality makes web-based reminder systems well-suited for deployment in settings where patients may not have the latest smartphones or where organizations do not wish to maintain separate iOS and Android codebases [3].

The technical implementation of a medication reminder system within a web application involves several non-trivial engineering challenges. The core challenge is reliably delivering a time-sensitive notification to a user at a precise moment regardless of their current activity – even when the application is not open in the browser. Beyond notification delivery, the system must maintain a persistent record of each dose event, compute adherence statistics, and expose data to both patients and their healthcare providers. Each of these requirements places specific demands on the system's architecture, particularly on the scheduling and

notification subsystems.

This paper analyzes the principal technical methods for implementing the reminder and notification functionality of a web-based medication monitoring system. Section 2 compares notification delivery channels available to web applications. Section 3 reviews architectural approaches to task scheduling. Section 4 describes a recommended three-tier system architecture that integrates these components. The paper concludes with a synthesis of findings and directions for future work.

Notification Delivery Methods

The choice of notification delivery channel is the first fundamental design decision in building a medication reminder system. Different channels offer different trade-offs between reliability, immediacy, user experience, and implementation effort. Table 1 presents a structured comparison of the five main notification delivery methods applicable to web-based systems.

Table 1 – Comparison of Notification Delivery Methods

Method	Delivery Channel	Reliability	Implementation Complexity	User Interaction Required
Email Notification	SMTP / Email API	Medium	Low	No
Browser Push (Web Push API)	Service Worker	High (browser online)	Medium	Yes (one-time permission)
SMS / Messaging	Telecom gateway	Very High	High	No
In-app Alert (WebSocket)	WebSocket / Polling	High (app open only)	Medium	No
Hybrid: Cron + Email/Push	Server-side scheduler	High	Medium–High	Partial

Email-based notifications are the most straightforward to implement, requiring only an SMTP connection or integration with a transactional email service such as SendGrid or Mailgun. They function reliably on all devices and require no special browser permissions. However, email is poorly suited as a primary channel for time-critical reminders due to delivery delays, unpredictable inbox delivery rates, and the requirement for the user to proactively check their inbox. Studies indicate that patients frequently overlook or delay reading email reminders, particularly when the email volume from other sources is high [2].

Browser Push Notifications, enabled through the Web Push API and Service Workers, represent the most technically suitable channel for web-based medication reminders. Once a user grants permission, the browser can receive push messages delivered via a push service (such as Google's FCM for Chrome or Mozilla's autopush for Firefox) even when the web application is not open. This behavior closely approximates that of native mobile push notifications. The Service Worker, a JavaScript file registered in the browser that runs in the background, intercepts incoming push events and displays a system notification [4]. The primary limitation is the one-time permission request: if the user declines, push notifications cannot be sent. In practice, user consent rates for health-related applications tend to be higher than average, as users who install a medication tracking tool have explicit motivation to receive reminders [5].

In-app alerts delivered via WebSocket connections provide real-time visual notifications within the application interface. They are suitable as a secondary, supplementary channel to reinforce push notifications when the user happens to have the application open. They cannot serve as a primary reminder channel, since they are ineffective when the application tab is closed or the device is idle.

SMS-based notifications offer the highest delivery reliability, as they function on any mobile phone regardless of internet connectivity. However, their implementation requires integration with a telecommunications gateway (e.g., Twilio, Vonage), which introduces per-message costs and regulatory compliance considerations. For large-scale deployment, SMS is typically reserved as a fallback channel for users who cannot receive browser push notifications.

For most web-based medication reminder systems, the recommended primary channel is Browser Push Notification, with email serving as a reliable fallback. This combination covers the vast majority of use cases: users who have granted push permission receive immediate browser notifications, while those who have not still receive email reminders as a safety net.

Task Scheduling Architectures

The scheduling subsystem is responsible for triggering notification dispatch at the precise time defined in each user's medication schedule. Unlike a simple daily newsletter, medication reminders are highly personalized: each patient may have dozens of individual drug-dose events per day, each at a different time and potentially with a different recurrence pattern (once daily, twice daily, every 8 hours, etc.). This dynamic, per-user nature of scheduling makes it substantially more complex than standard application background jobs [6]. Four main architectural approaches to task scheduling are evaluated in Table 2.

Table 2 – Comparison of Scheduling Approaches for Notification Systems

Approach	Advantages	Limitations
Cron-based scheduler (node-cron, cron-job.org)	Simple setup; low resource usage; suitable for global, low-frequency reminders	Not suitable for per-user dynamic schedules; single-node dependency; polling latency
Message queue (Bull/BullMQ + Redis)	Persistent jobs; automatic retry on failure; horizontal scalability; precise per-job delay	Requires Redis infrastructure; higher initial setup complexity
Cloud scheduler (AWS Event Bridge, Google Cloud Tasks)	Fully managed; built-in reliability; no infrastructure maintenance required	Vendor lock-in; per-invocation cost; latency for high-frequency per-user tasks
Database-driven polling	No additional infrastructure; all data in one place; straightforward implementation	High database load at scale; imprecise timing limited by polling interval

A cron-based scheduler (e.g., node-cron in a Node.js environment) runs at fixed intervals – typically every minute – and queries the database to find any scheduled reminders that are due within the current time window. While simple to implement, this approach introduces a timing imprecision of up to one polling interval. More importantly, it does not natively support per-job retry logic: if a notification fails to deliver, the cron job does not automatically retry it. Additionally, a cron-based scheduler running on a single application instance is not inherently fault-tolerant; if the server restarts, in-flight jobs may be lost.

A message queue architecture using BullMQ with a Redis backend addresses these limitations. When a patient creates or modifies a medication schedule, the API server enqueues a delayed job for each upcoming dose event. The delay is calculated as the difference between the current timestamp and the scheduled dose time. When the delay expires, BullMQ moves the job from the waiting state to the active state and delivers it to a worker process that performs the actual notification dispatch. This approach offers millisecond-level precision, automatic retry with configurable backoff strategies, and persistence: jobs survive application restarts because they are stored in Redis. BullMQ also provides a built-in dashboard (Bull Board) for monitoring queue health, which is valuable for production operations [7].

A practical retry policy for notification delivery should define a maximum number of attempts (typically 3–5) and an inter-attempt delay. An exponential backoff strategy – where the delay between attempts increases with each failure (e.g., 1 minute, 5 minutes, 25 minutes) – is preferable to fixed-interval retries, as it reduces the load on a temporarily unavailable downstream service (email provider, push service) while still ensuring eventual delivery within a clinically acceptable time window [6].

Cloud-managed schedulers such as AWS EventBridge Scheduler or Google Cloud Tasks are fully managed services that eliminate the need to maintain Redis infrastructure. They are well-suited for low-frequency, large-scale batch jobs. However, for per-user, high-frequency reminders (potentially hundreds of events per minute across all active users), the per-invocation cost model and API rate limits of cloud schedulers can become prohibitive. They are best used as a supplementary mechanism – for example, for daily adherence digest emails – rather than as the primary real-time notification trigger.

System Architecture and Integration

Figure 1 illustrates the recommended three-tier architecture for a web-based medication reminder system that integrates the notification delivery and scheduling components described in the preceding sections.

The client layer (React SPA running in the browser) handles all user-facing interactions: configuring the medication list and dose schedule, marking doses as taken or missed, viewing adherence history, and granting push notification permission to the browser. The client communicates with the server layer exclusively via a RESTful API over HTTPS. The Service Worker, registered by the client, operates independently in the browser background and handles incoming Web Push messages even when the application tab is closed.

The server layer (Node.js REST API) manages all business logic: user authentication and authorization,

medication and schedule persistence, and job queue management. When a patient creates a new medication schedule or updates an existing one, the API server calculates the timestamps of upcoming dose events and enqueues corresponding delayed jobs in the BullMQ queue. For recurring schedules, jobs for the next 24–48 hours are pre-enqueued, with a background process that continuously extends the horizon.

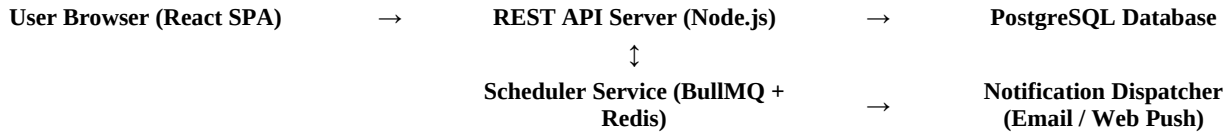

Figure 1 – Recommended three-tier architecture for a web-based medication reminder system

The Scheduler Service is a BullMQ worker process that monitors the Redis queue for jobs whose delay has expired. Upon dequeuing a job, it calls the Notification Dispatcher, which reads user preferences from the database and selects the appropriate delivery channel. If the user has an active Web Push subscription, the dispatcher sends a push message via the Web Push protocol; otherwise, it falls back to email. The dispatcher records the delivery attempt outcome in the database, and BullMQ's retry mechanism re-queues failed jobs automatically.

A critical security requirement throughout this pipeline is the protection of personal health information. Push notification payloads are end-to-end encrypted by the Web Push protocol between the server and the browser, but their content may briefly appear on the device lock screen. For this reason, notification content should be designed to minimize exposure of sensitive details: a generic message such as 'It is time for your medication' is preferable to one that includes the specific drug name and dosage. Complete medication information should be accessible only within the authenticated application session [5].

For the adherence tracking component, each dose event is recorded in the database with a timestamp and a status label. A dose should be classified as 'on time' if the patient confirms it within a defined time window after the scheduled moment – a threshold of 60 minutes is a commonly used clinical benchmark that balances strictness with practical tolerance for real-life delays [8]. Doses confirmed after this threshold are recorded as 'late', and doses with no confirmation are eventually marked as 'missed' by a background cleanup process. This granular event log forms the foundation for computing adherence statistics and generating reports for patients and their physicians.

Conclusions

This paper has reviewed and compared the principal technical methods for implementing medication reminder functionality in web-based applications. The analysis covers notification delivery channels, scheduling architectures, system integration patterns, and security considerations.

Among the available notification delivery channels, Browser Push Notification via the Web Push API represents the most appropriate primary channel for web-based medication reminders, offering near-native mobile alert behavior without requiring a dedicated mobile application. Email should be implemented as a reliable fallback for users who have not granted push permission. SMS integration is recommended for safety-critical cases where maximum delivery reliability is required, despite its higher implementation and operational cost.

For the scheduling subsystem, a message queue architecture based on BullMQ and Redis provides the optimal combination of timing precision, fault tolerance, and horizontal scalability. It is recommended for any production system expected to serve more than a few hundred users. Cron-based scheduling is acceptable for prototyping or low-user-count deployments, while cloud-managed schedulers are best reserved for low-frequency batch notification tasks.

The proposed three-tier architecture, with a clearly separated Scheduler Service and Notification Dispatcher, provides a clean separation of concerns that facilitates independent scaling and maintenance of each component. Future research directions include adaptive reminder timing using machine learning models trained on individual adherence behavior patterns, integration with wearable sensors for passive dose detection, and investigation of gamification techniques to sustain long-term patient engagement with reminder systems.

REFERENCES

1. World Health Organization. Adherence to Long-Term Therapies: Evidence for Action. Geneva: WHO, 2003. 194 p. URL: https://www.who.int/chp/knowledge/publications/adherence_introduction.pdf [Accessed: 18.03.2026].

2. Thakkar J. et al. Mobile Telephone Text Messaging for Medication Adherence in Chronic Disease. JAMA Internal Medicine. 2016. Vol. 176, No. 3. P. 340–349. DOI: 10.1001/jamainternmed.2015.7667
3. Gaedke M., Meinecke J., Nussbaumer M. Concepts and Technologies for a Comprehensive Resource-Oriented Web Application Framework. Proceedings of the IADIS International Conference WWW/Internet. 2005. P. 1–9.
4. Mozilla Developer Network. Service Worker API. URL: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API [Accessed: 18.03.2026].
5. He D., Naveed M., Gunter C. A., Nahrstedt K. Security Concerns in Android mHealth Apps. AMIA Annual Symposium Proceedings. 2014. P. 645–654.
6. Fowler M. Patterns of Enterprise Application Architecture. Boston: Addison-Wesley, 2002. 560 p.
7. BullMQ Documentation. Premium Message Queue for Node.js. URL: https://docs.bullmq.io [Accessed: 18.03.2026].
8. Vrijens B. et al. A New Taxonomy for Describing and Defining Adherence to Medications. British Journal of Clinical Pharmacology. 2012. Vol. 73, No. 5. P. 691–705. DOI: 10.1111/j.1365-2125.2012.04167.x

Ковальчук Іван Сергійович – студент групи 1ПІ-22б, факультет інформаційних технологій та комп'ютерної інженерії, Вінницький національний технічний університет, м. Вінниця, e–mail: ivan.kovalchuk1338@gmail.com.
Рейда Олександр Миколайович – кандидат технічних наук, доцент кафедри програмного забезпечення, Вінницький національний технічний університет, м. Вінниця, e–mail: reyda@vntu.edu.ua.

Kovalchuk Ivan S. – student of group 1PI-22b, Faculty of Information Technologies and Computer Engineering, Vinnytsia National Technical University, Vinnytsia, e-mail: ivan.kovalchuk1338@gmail.com.
Reyda Oleksandr M. – Candidate of Technical Sciences, Associate Professor of the Software Engineering Department, Vinnytsia National Technical University, Vinnytsia, e–mail: reyda@vntu.edu.ua.