**MEDIMATE DEVELOPER DOCUMENTATION**

*Technical Specification & Implementation Guide*

**Version:** v1.0  
**Author:** ADELA  
**Date:** 11 October 2025

**Organization:** MediMate Development Team  
**Contact:** [dev@medimate.io](mailto:dev@medimate.io)

📘 Confidential Internal Use Only

# **REVISION HISTORY**

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Description of Change |
| 1.0 | 11 October 2025 | ADELA | Initial draft created for MediMate Developer Documentation. |
| 1.1 |  |  |  |
| 1.2 |  |  |  |

# **TABLE OF CONTENTS**

Contents

[**REVISION HISTORY** 2](#_Toc212156904)

[**TABLE OF CONTENTS** 3](#_Toc212156905)

[**1.0** **INTRODUCTION** 12](#_Toc212156906)

[**1.1 Purpose of the Document** 12](#_Toc212156907)

[**1.2 About MediMate** 12](#_Toc212156908)

[**1.3 Document Structure** 12](#_Toc212156909)

[**1.4 Intended Audience** 14](#_Toc212156910)

[**1.5 Goals and Objectives** 14](#_Toc212156911)

[**2.0 – Scope and Objectives** 15](#_Toc212156912)

[**2.1 Overview** 15](#_Toc212156913)

[**2.2 Core Scope** 15](#_Toc212156914)

[**2.3 Integration Between Medication and Mood Scopes** 16](#_Toc212156915)

[**2.4 Objectives** 17](#_Toc212156916)

[**2.5 Out of Scope (For Current Phase)** 17](#_Toc212156917)

[**2.6 Summary** 18](#_Toc212156918)

[**3.0 – Target Audience** 19](#_Toc212156919)

[**3.1 Overview** 19](#_Toc212156920)

[**3.2 Primary User Groups** 19](#_Toc212156921)

[**3.3 Secondary User Groups** 21](#_Toc212156922)

[**3.4 Stakeholder Relationships** 21](#_Toc212156923)

[**3.5 Accessibility Considerations** 22](#_Toc212156924)

[**3.6 Target Audience Summary** 22](#_Toc212156925)

[**4.0 – Development Phases** 24](#_Toc212156926)

[**4.1 Overview** 24](#_Toc212156927)

[**4.2 Phase Structure Overview** 24](#_Toc212156928)

[**4.3 Phase 1 – Core Medication System (MVP)** 24](#_Toc212156929)

[**4.4 Phase 2 – Mood & Emotional Wellness** 25](#_Toc212156930)

[**4.5 Phase 3 – Caregiver & Shared Access** 26](#_Toc212156931)

[**4.6 Phase 4 – AI Insights & Predictive Analytics** 27](#_Toc212156932)

[**4.7 Phase 5 – Clinical & Cloud Expansion** 28](#_Toc212156933)

[**4.8 Phase Summary and Developer Notes** 28](#_Toc212156934)

[**4.9 Summary** 29](#_Toc212156935)

[**5.0 – System Architecture** 30](#_Toc212156936)

[**5.1 Overview** 30](#_Toc212156937)

[**5.2 High-Level Architecture Diagram** 30](#_Toc212156938)

[**5.3 Frontend Architecture** 31](#_Toc212156939)

[**5.4 Backend Architecture** 32](#_Toc212156940)

[**5.5 Database Design** 33](#_Toc212156941)

[**5.6 External Services Integration** 34](#_Toc212156942)

[**5.7 Security Architecture** 34](#_Toc212156943)

[**5.8 Scalability and Deployment Architecture** 35](#_Toc212156944)

[**5.9 Error Handling and Monitoring** 35](#_Toc212156945)

[**5.10 Data Flow Summary** 35](#_Toc212156946)

[**5.11 Summary** 36](#_Toc212156947)

[**6.0 – Technical Stack & Tools** 37](#_Toc212156948)

[**6.1 Overview** 37](#_Toc212156949)

[**6.2 Stack Overview Table** 37](#_Toc212156950)

[**6.3 Frontend Stack (Web & Mobile)** 38](#_Toc212156951)

[**6.4 Backend Stack** 39](#_Toc212156952)

[**6.5 Database Stack** 40](#_Toc212156953)

[**6.6 Cloud & Hosting Infrastructure** 41](#_Toc212156954)

[**6.7 AI & Analytics (Later Phase Integration)** 42](#_Toc212156955)

[**6.8 Security & Compliance Tools** 42](#_Toc212156956)

[**6.9 Development & Testing Tools** 43](#_Toc212156957)

[**6.10 Why This Stack?** 43](#_Toc212156958)

[**6.11 Summary** 44](#_Toc212156959)

[**7.0 – Security and Privacy Framework** 45](#_Toc212156960)

[**7.1 Overview** 45](#_Toc212156961)

[**7.2 Core Security Objectives** 45](#_Toc212156962)

[**7.3 Authentication & Authorization** 45](#_Toc212156963)

[**7.4 Data Encryption & Protection** 47](#_Toc212156964)

[**7.5 Data Privacy & User Consent** 48](#_Toc212156965)

[**7.6 Logging, Auditing & Monitoring** 48](#_Toc212156966)

[**7.7 Secure Development Practices** 49](#_Toc212156967)

[**7.8 Compliance & Regulatory Standards** 50](#_Toc212156968)

[**7.9 Data Backup, Recovery & Incident Response** 50](#_Toc212156969)

[**7.10 Developer Guidelines for Security Testing** 51](#_Toc212156970)

[**7.11 Summary** 51](#_Toc212156971)

[**8.0 – Features and Functional Workflows** 52](#_Toc212156972)

[**8.1 Overview** 52](#_Toc212156973)

[**8.2 Core Feature Categories** 52](#_Toc212156974)

[**8.3 Feature 1 – Medication Management System** 53](#_Toc212156975)

[**8.4 Feature 2 – Smart Reminders and Notifications** 54](#_Toc212156976)

[**8.5 Feature 3 – Mood Tracking and Journaling** 54](#_Toc212156977)

[**8.6 Feature 4 – Adherence Analytics and Reports** 55](#_Toc212156978)

[**8.7 Feature 5 – Caregiver and Shared Access** 56](#_Toc212156979)

[**8.8 Feature 6 – Export and Reporting Tools** 56](#_Toc212156980)

[**8.9 Feature 7 – Settings and Privacy Controls** 57](#_Toc212156981)

[**8.10 Inter-Feature Relationships** 58](#_Toc212156982)

[**8.11 Feature Development Best Practices** 58](#_Toc212156983)

[**8.12 Summary** 58](#_Toc212156984)

[**Section 9.0 – UI/UX Design Principles** 60](#_Toc212156985)

[**9.1 Overview** 60](#_Toc212156986)

[**9.2 Core Design Philosophy** 60](#_Toc212156987)

[**9.3 Visual Design System** 61](#_Toc212156988)

[**9.4 UX Interaction Guidelines** 61](#_Toc212156989)

[**9.5 Light and Dark Theme Design** 62](#_Toc212156990)

[**9.6 Component Standards** 63](#_Toc212156991)

[**9.7 Usability Testing & Feedback Loops** 63](#_Toc212156992)

[**9.8 Emotional Design Integration** 64](#_Toc212156993)

[**9.9 Developer Best Practices** 64](#_Toc212156994)

[**9.10 Summary** 64](#_Toc212156995)

[**Section 10.0 – Database Schema** 65](#_Toc212156996)

[**10.1 Overview** 65](#_Toc212156997)

[**10.2 Database Design Philosophy** 65](#_Toc212156998)

[**10.3 Core Collections and Relationships** 65](#_Toc212156999)

[**10.4 Relationships Overview** 69](#_Toc212157000)

[**10.5 Indexing and Optimization** 70](#_Toc212157001)

[**10.6 Example Data Flow** 70](#_Toc212157002)

[**10.7 Developer Best Practices** 71](#_Toc212157003)

[**10.8 Future Expansion Considerations** 71](#_Toc212157004)

[**10.9 Summary** 71](#_Toc212157005)

[**Section 11.0 – API Documentation** 72](#_Toc212157006)

[**11.1 Overview** 72](#_Toc212157007)

[**11.2 API Design Philosophy** 72](#_Toc212157008)

[**11.3 API Base URL** 72](#_Toc212157009)

[**11.4 Authentication Endpoints** 73](#_Toc212157010)

[**11.5 User Management Endpoints** 74](#_Toc212157011)

[**11.6 Mood Tracking Endpoints** 75](#_Toc212157012)

[**11.7 Messaging Endpoints** 76](#_Toc212157013)

[**11.8 Notification Endpoints** 77](#_Toc212157014)

[**11.9 Reporting & Export Endpoints** 78](#_Toc212157015)

[**11.10 Error Handling and Status Codes** 79](#_Toc212157016)

[**11.11 API Security and Privacy Controls** 80](#_Toc212157017)

[**11.12 Developer Usage Guidelines** 80](#_Toc212157018)

[**11.13 Example Integration (Frontend)** 81](#_Toc212157019)

[**11.14 Summary** 81](#_Toc212157020)

[**Section 12.0 – Notification System (Hybrid Online + Offline)** 82](#_Toc212157021)

[**12.1 Overview** 82](#_Toc212157022)

[**12.2 Objectives** 82](#_Toc212157023)

[**12.3 Hybrid Notification Architecture** 82](#_Toc212157024)

[**12.4 Notification Categories** 83](#_Toc212157025)

[**12.5 Hybrid Notification Flow** 83](#_Toc212157026)

[**12.6 Database Schema** 84](#_Toc212157027)

[**12.7 Local Notification Scheduling (Offline)** 85](#_Toc212157028)

[**12.8 Cloud Notification Delivery (Online)** 85](#_Toc212157029)

[**12.9 Offline Logging and Sync Service** 86](#_Toc212157030)

[**12.10 Scheduler and Automation** 87](#_Toc212157031)

[**12.11 Privacy and Control** 87](#_Toc212157032)

[**12.12 Error Handling & Retry Logic** 87](#_Toc212157033)

[**12.13 Developer Implementation Guidelines** 88](#_Toc212157034)

[**12.14 Example Use Cases** 88](#_Toc212157035)

[**12.15 Analytics and Monitoring** 88](#_Toc212157036)

[**12.16 Summary** 89](#_Toc212157037)

[**Section 13.0 – Export and Reporting Feature** 90](#_Toc212157038)

[**13.1 Overview** 90](#_Toc212157039)

[**13.2 Core Objectives** 90](#_Toc212157040)

[**13.3 Key Functionalities** 90](#_Toc212157041)

[**13.4 Workflow and Process Design** 91](#_Toc212157042)

[**13.5 Developer Implementation Notes** 92](#_Toc212157043)

[**13.6 Offline and Hybrid Operation** 93](#_Toc212157044)

[**13.7 Security and Privacy Considerations** 93](#_Toc212157045)

[**13.8 Future Enhancements** 93](#_Toc212157046)

[**13.9 Example Use Cases** 94](#_Toc212157047)

[**13.10 Developer Checklist** 94](#_Toc212157048)

[**Section 14.0 – AI and Analytics (Future Phase)** 95](#_Toc212157049)

[**14.1 Purpose** 95](#_Toc212157050)

[**14.2 Architecture Overview** 95](#_Toc212157051)

[**14.2.1 Data Ingestion Layer** 95](#_Toc212157052)

[**14.2.2 Preprocessing & Feature Engineering Service** 95](#_Toc212157053)

[**14.2.3 Model Training Service** 96](#_Toc212157054)

[**14.2.4 Model Serving & Inference Service** 96](#_Toc212157055)

[**14.2.5 Analytics & Visualization Layer** 96](#_Toc212157056)

[**14.3 Data Privacy & Security** 97](#_Toc212157057)

[**14.4 Development Workflow** 97](#_Toc212157058)

[**14.5 Use Cases & Developer Notes** 97](#_Toc212157059)

[**14.6 Future Considerations** 98](#_Toc212157060)

[**Section 15.0 – DevOps and CI/CD Pipeline** 99](#_Toc212157061)

[**15.1 Purpose** 99](#_Toc212157062)

[**15.2 Architecture Overview** 99](#_Toc212157063)

[15.2.1 Version Control 99](#_Toc212157064)

[**15.2.2 Continuous Integration (CI)** 99](#_Toc212157065)

[**15.2.3 Continuous Deployment/Delivery (CD)** 100](#_Toc212157066)

[**15.3 Cloud Environment Setup** 100](#_Toc212157067)

[**15.4 DevOps Workflow** 100](#_Toc212157068)

[**15.5 Key Developer Notes** 100](#_Toc212157069)

[**15.6 Future Considerations** 101](#_Toc212157070)

[**Section 16.0 – Coding Standards and Best Practices** 102](#_Toc212157071)

[**16.1 Overview** 102](#_Toc212157072)

[**16.2 Objectives** 102](#_Toc212157073)

[**16.3 Code Structure and Organization** 102](#_Toc212157074)

[**a. Folder Structure Guidelines** 102](#_Toc212157075)

[**16.4 Naming Conventions** 103](#_Toc212157076)

[**16.5 Coding Style Standards** 103](#_Toc212157077)

[**a. General** 103](#_Toc212157078)

[**b. Language-Specific** 103](#_Toc212157079)

[**16.6 Code Review and Collaboration Process** 104](#_Toc212157080)

[**Commit Message Format** 104](#_Toc212157081)

[**16.7 Testing and Quality Assurance** 104](#_Toc212157082)

[**16.8 Documentation and Comments** 105](#_Toc212157083)

[**16.9 Security and Privacy Considerations** 105](#_Toc212157084)

[**16.10 Performance and Optimization** 105](#_Toc212157085)

[**16.11 Version Control and Branching Strategy** 105](#_Toc212157086)

[**16.12 Code Review Checklist** 106](#_Toc212157087)

[**16.13 Continuous Improvement** 106](#_Toc212157088)

[**16.14 Tools and Enforcement** 107](#_Toc212157089)

[**16.15 Summary** 107](#_Toc212157090)

[**Section 17.0 – Environment Setup** 108](#_Toc212157091)

[**17.1 Overview** 108](#_Toc212157092)

[**17.2 Objectives** 108](#_Toc212157093)

[**17.3 Prerequisites** 108](#_Toc212157094)

[**17.4 Repository Structure** 109](#_Toc212157095)

[17.5 Local Development Setup 109](#_Toc212157096)

[Step 1: Clone the Repository 109](#_Toc212157097)

[Step 2: Install Dependencies 109](#_Toc212157098)

[**17.6 Environment Configuration** 110](#_Toc212157099)

[Example: .env (Backend) 110](#_Toc212157100)

[Example: .env (Frontend / Mobile) 110](#_Toc212157101)

[Example: .env (AI Module) 111](#_Toc212157102)

[**17.7 Database Setup** 111](#_Toc212157103)

[Option 1: Local PostgreSQL 111](#_Toc212157104)

[Option 2: MongoDB (for NoSQL modules) 111](#_Toc212157105)

[Option 3: Cloud Database 111](#_Toc212157106)

[**17.8 Running the Project** 111](#_Toc212157107)

[Start Backend API (Node.js) 111](#_Toc212157108)

[Start Backend (Python FastAPI) 111](#_Toc212157109)

[Run Web App 111](#_Toc212157110)

[Run Mobile App 111](#_Toc212157111)

[Run AI/ML Module 112](#_Toc212157112)

[Run All via Docker 112](#_Toc212157113)

[**17.9 Folder Permissions and Access Control** 112](#_Toc212157114)

[**17.10 Environment Profiles** 112](#_Toc212157115)

[**17.11 Docker Setup (Recommended for Consistency)** 112](#_Toc212157116)

[**Docker Compose Example** 112](#_Toc212157117)

[**17.12 Troubleshooting Guide** 113](#_Toc212157118)

[**17.13 Security and Secrets Management** 114](#_Toc212157119)

[**17.14 Environment Verification Checklist** 114](#_Toc212157120)

[**17.15 Summary** 114](#_Toc212157121)

[**Section 18.0 – Developer Onboarding and Git Workflow** 115](#_Toc212157122)

[**18.1 Overview** 115](#_Toc212157123)

[**18.2 Developer Onboarding Process** 115](#_Toc212157124)

[**18.3 Git Repository Structure** 117](#_Toc212157125)

[**18.4 Git Branching Strategy** 117](#_Toc212157126)

[**18.5 Commit and Pull Request Guidelines** 118](#_Toc212157127)

[**18.6 Code Review and Merging Protocol** 118](#_Toc212157128)

[**18.7 Version Control and Tagging** 119](#_Toc212157129)

[**18.8 Collaboration and Issue Tracking** 119](#_Toc212157130)

[**18.9 Continuous Integration Alignment** 119](#_Toc212157131)

[**18.10 Best Practices Summary** 120](#_Toc212157132)

[**18.11 Example Developer Flow** 120](#_Toc212157133)

[**18.12 Onboarding Checklist** 120](#_Toc212157134)

[**18.13 Summary** 121](#_Toc212157135)

[**Section 19.0 – Troubleshooting and Maintenance** 122](#_Toc212157136)

[**19.1 Overview** 122](#_Toc212157137)

[**19.2 Common Issue Categories** 122](#_Toc212157138)

[**19.3 General Troubleshooting Workflow** 123](#_Toc212157139)

[**19.4 Logging and Error Monitoring** 123](#_Toc212157140)

[**19.5 Known Issues and Fix References** 124](#_Toc212157141)

[**19.6 Debugging by Module** 124](#_Toc212157142)

[**19.7 Maintenance Routines** 126](#_Toc212157143)

[**19.8 Preventive Maintenance and Alerts** 127](#_Toc212157144)

[**19.9 Escalation and Support Procedure** 127](#_Toc212157145)

[**19.10 Backup and Recovery Plan** 127](#_Toc212157146)

[**19.11 Summary** 127](#_Toc212157147)

# **INTRODUCTION**

## **1.1 Purpose of the Document**

The purpose of this Developer Documentation is to provide a comprehensive technical reference for the design, architecture, development, deployment, and maintenance of the MediMate system.  
It serves as a central resource for developers, QA engineers, DevOps specialists, and future contributors to understand the system’s core structure and behavior.

## **1.2 About MediMate**

MediMate is a cross-platform digital health solution focused on mood monitoring and mental-wellness management.  
It allows patients to log moods, visualize emotional trends, and communicate securely with caregivers.  
The application will evolve to include AI-driven insights and professional doctor support in later development phases.

* Platforms**:** Web & Mobile (React + React Native)
* Backend**:** Node.js (REST API) + Python microservices (AI)
* Database**:** MongoDB Atlas
* Notifications**:** Firebase Cloud Messaging
* Media **Storage:** Cloudinary
* Hosting **/ Cloud:** Google Cloud Platform (GCP)

## **1.3 Document Structure**

This document is organized into 22 major sections:

1. **Introduction** – Overview of MediMate, its purpose, and the structure of this documentation.
2. **Scope and Objectives** – Defines project boundaries, features included/excluded, and high-level goals.
3. **Target Audience** – Identifies the intended readers and contributors to this documentation.
4. **Development Phases** – Outlines each planned development phase (MVP, AI integration, full ecosystem).
5. **System Architecture** – Describes the software architecture, modules, and interactions.
6. **Technology Stack** – Lists all core technologies, frameworks, and cloud services used.
7. **Features** – Provides detailed descriptions and workflows for all app features.
8. **UI/UX Design Principles** – Establishes design philosophy, accessibility, and theme support.
9. **Database Schema** – Documents collections, relationships, and example data structures.
10. **API Documentation** – Details all REST API endpoints, parameters, and sample requests/responses.
11. **Security and Privacy** – Explains authentication, authorization, and data-protection mechanisms.
12. **Notification System** – Covers Firebase integration, delivery flow, and privacy considerations.
13. **Export and Reporting Features** – Describes how user data and reports are exported (CSV/PDF).
14. **AI and Analytics (Future Phase)** – Outlines AI model training, microservices, and predictive insights.
15. **DevOps and CI/CD Pipeline** – Defines continuous integration, deployment, and cloud environment setup.
16. **Coding Standards and Best Practices** – Specifies code structure, naming conventions, and review rules.
17. **Environment Setup** – Provides installation, configuration, and environment variable details.
18. **Developer Onboarding and Git Workflow** – Describes collaboration, branching, and version-control strategy.
19. **Troubleshooting and Maintenance** – Lists known issues, debugging steps, and system maintenance routines.
20. **Testing Strategy** – Defines test levels, frameworks, and coverage requirements.
21. **Glossary** – Provides definitions of all technical terms and acronyms.
22. **References and Resources** – Lists all external sources, integration links, and supporting documentation.

## **1.4 Intended Audience**

This documentation is intended for:

* **Frontend developers** responsible for React/React Native implementation
* **Backend engineers** managing Node.js APIs and microservices
* **QA/test engineers** validating functional and integration layers
* **DevOps personnel** handling CI/CD pipelines and cloud deployment
* **Future AI/ML engineers** integrating predictive analytics modules

## **1.5 Goals and Objectives**

* Deliver a modular, secure, and scalable digital-health platform
* Maintain consistent code quality and documentation standards
* Ensure compliance with data-privacy regulations (HIPAA/GDPR readiness)
* Enable easy onboarding for future contributors through clear documentation

## **2.0 – Scope and Objectives**

### **2.1 Overview**

The **scope** of MediMate covers a unified digital health ecosystem that bridges **medication management** and **mental well-being support** within a single mobile and web platform. It is designed for patients, caregivers, and healthcare professionals to collaborate effectively while simplifying medication routines and tracking emotional health patterns.

MediMate’s purpose is to improve **treatment adherence**, **personal health awareness**, and **emotional stability** by combining practical medication tools with mental health insights. The platform promotes a holistic approach to wellness focusing not only on **what** users take (medication) but also on **how** they feel (mood and mental state).

### **2.2 Core Scope**

MediMate’s system is divided into **two primary functional scopes** that work in synergy:

#### **A. Medication Management Scope**

This core scope focuses on enabling users to take control of their medication journey.  
Key components include:

* **Medication Scheduling and Reminders**  
  Users can create custom medication plans by specifying drug names, dosages, times, and frequency. The app automatically triggers smart notifications to ensure adherence without overwhelming users.
* **Smart Reminders & Snooze Function**  
  Intelligent reminders adjust based on user patterns (e.g., missed doses or late acknowledgment). A “snooze” or “taken later” option offers flexibility while maintaining accountability.
* **Refill Alerts and Inventory Tracking**  
  Users are notified when medication supply is running low, with options to add refill reminders or link to pharmacy services (future integration).
* **Medication Logs and Adherence Reports**  
  Every dose taken, missed, or delayed is recorded. Analytics charts visualize adherence rates over time, allowing users, caregivers, or medical professionals to identify patterns and intervene early.
* **Caregiver and Multi-User Access (Later Phase)**  
  Caregivers can monitor adherence data remotely and provide assistance or encouragement while maintaining data privacy and consent-based access.
* **Export and Data Sharing**  
  Users can export medication history in standard formats (CSV, PDF) for sharing with healthcare providers or for personal records.

#### **B. Mental Health & Mood Monitoring Scope**

Beyond medication tracking, MediMate incorporates a proactive emotional wellness layer.  
This scope ensures that the platform supports users’ **psychological health** alongside their **physical medication routine**.

Key components include:

* **Mood Check-Ins**  
  Users can log their emotional state daily or multiple times per day using an intuitive mood scale (emoji-based or color-coded).
* **Mood Journaling**  
  Optionally, users can write brief notes about how they feel, factors affecting their mood, or medication side effects.
* **Mood Trends & Insights**  
  Visual charts display mood progression over days, weeks, or months enabling users and caregivers to spot triggers, improvements, or potential declines.
* **AI-Powered Emotional Insights (Later Phase)**  
  Using cloud-based analysis, MediMate will provide mood–medication correlation insights, highlighting possible links between treatment adherence and emotional fluctuations.
* **Privacy & Data Control**  
  All emotional and mental health data are user-owned, encrypted, and accessible only under consent. Users can control what information is shared and with whom.

### **2.3 Integration Between Medication and Mood Scopes**

A key differentiator of MediMate is the **interconnection** between both domains.  
The system will intelligently correlate **medication adherence** with **mood variations**, helping users and clinicians understand whether certain medications or patterns influence mental well-being.

For example:

* A missed antidepressant dose could trigger a reminder and a mood-check prompt the next day.
* Consistent adherence patterns may reflect improved mood stability, helping doctors fine-tune treatment.

This synergy transforms MediMate from a simple reminder app into a **personal health companion** that evolves with the user.

### **2.4 Objectives**

The main objectives of MediMate are:

1. **Enhance Treatment Adherence**  
   Ensure users consistently take prescribed medications through intelligent reminders, tracking, and user-friendly notifications.
2. **Promote Mental Health Awareness**  
   Encourage users to monitor, understand, and improve their emotional well-being alongside their physical health journey.
3. **Enable Data-Driven Insights**  
   Use analytics and visualizations to offer actionable insights into adherence rates and emotional trends.
4. **Support Caregiver and Clinical Collaboration**  
   Facilitate responsible data sharing between patients, caregivers, and healthcare providers through secure, consent-based access.
5. **Ensure Privacy, Security, and Compliance**  
   Adhere to medical data protection laws (e.g., HIPAA, GDPR) and maintain full data integrity through encryption, authentication, and controlled sharing.
6. **Lay the Foundation for AI-Enhanced Health Management**  
   Build a scalable foundation that will later support AI-driven recommendations, predictive insights, and anomaly detection for personalized health support.

### **2.5 Out of Scope (For Current Phase)**

While MediMate’s long-term vision is expansive, the **initial development phase (MVP)** will **exclude** the following features:

* Direct doctor or pharmacist consultation features
* AI-driven predictive analytics
* Pharmacy purchase integrations
* Image or prescription upload
* Advanced chat or telemedicine modules

These will be introduced gradually in later versions once the core functionalities **Medication Tracking**, **Reminders**, and **Mood Monitoring** are fully stable and tested.

### **2.6 Summary**

In summary, MediMate’s scope spans across both **medical and emotional wellness**, ensuring users not only stay on schedule with their treatments but also remain mentally balanced and informed. By combining **Medication Management** and **Mood Monitoring**, MediMate provides a complete ecosystem for preventive, active, and supportive healthcare.

## **3.0 – Target Audience**

### **3.1 Overview**

The **target audience** of MediMate encompasses individuals and stakeholders involved in **personal health management**, **caregiving**, and **digital healthcare innovation**. The platform is designed to serve a **diverse user base** from patients managing multiple prescriptions to caregivers monitoring dependents, and eventually healthcare professionals who need reliable adherence data for clinical insights.

By catering to these varied audiences, MediMate aims to simplify healthcare engagement, improve medication adherence, enhance emotional well-being, and foster better collaboration between patients, caregivers, and healthcare systems.

### **3.2 Primary User Groups**

#### **A. Patients (Core Users)**

Patients are MediMate’s **primary and central users**. They are individuals who take one or more medications regularly and may struggle with adherence, emotional regulation, or the complexity of managing multiple prescriptions.

**Key User Types Include:**

* **Chronic Condition Patients:** Individuals managing long-term illnesses such as hypertension, diabetes, or depression who require strict adherence to treatment schedules.
* **Elderly Users:** Older adults who often need assistance with remembering doses, refills, or tracking side effects.
* **Post-Treatment Patients:** Users in recovery who need temporary support for medication adherence and progress tracking.
* **Mental Health Patients:** Individuals on antidepressants, anxiety medication, or therapy schedules who benefit from the combined mood and medication tracking.

**Primary Needs:**

* Simple, reliable medication reminders
* Easy dose tracking and refill alerts
* Private mood journaling and reflection
* Data visualization for adherence and progress
* Motivation to stay consistent with treatment plans

#### **B. Caregivers and Family Members**

Caregivers play an essential role in ensuring that patients especially the elderly, minors, or individuals with cognitive challenges follow their prescribed regimens correctly. MediMate empowers caregivers with controlled, consent-based visibility into patient health data.

**Key User Types Include:**

* **Family Caregivers:** Parents, spouses, or guardians responsible for dependent patients.
* **Professional Caregivers:** Nurses or attendants managing medications for multiple patients in home or institutional settings.

**Primary Needs:**

* Secure access to adherence and mood summaries
* Notifications for missed doses or emotional changes
* Ability to set reminders or assist with scheduling
* Reliable communication channel (later phase)

MediMate ensures caregivers act as **supportive partners** rather than overseers encouraging empathy, communication, and better health outcomes.

#### **C. Healthcare Professionals (Future Integration)**

In later phases, MediMate will expand access to **clinicians, doctors, and mental health professionals**, enabling them to review adherence data and mood patterns for better treatment planning.

**Primary Needs:**

* Accurate adherence reports and medication logs
* Correlation between medication intake and patient-reported mood
* Exportable or sharable insights for electronic medical record (EMR) systems
* Ability to provide personalized adjustments based on tracked data

This user group, while not part of the MVP release, represents a **critical growth direction** for MediMate’s data-driven ecosystem.

### **3.3 Secondary User Groups**

#### **A. Developers and Technical Teams**

The MediMate platform will be open to collaboration and scaling by **software developers, data scientists, and product engineers** who contribute to new features or integrations (e.g., AI analytics, cloud interoperability, wearable device data sync).

**Primary Needs:**

* Well-documented APIs and SDKs
* Secure and scalable architecture
* Cloud-based data processing support
* CI/CD pipeline for continuous updates

This audience ensures the long-term sustainability and innovation of MediMate as a scalable, secure digital health ecosystem.

#### **B. Administrators and System Managers**

Administrators are responsible for overseeing MediMate’s operations, security compliance, and user management. They maintain the system’s integrity and ensure adherence to healthcare data laws.

**Primary Needs:**

* Role-based access control (RBAC) management
* Data privacy and audit logging
* Incident reporting and account verification tools
* Secure backup and recovery systems

Their oversight ensures MediMate remains compliant, safe, and reliable as the user base grows.

### **3.4 Stakeholder Relationships**

MediMate’s design philosophy is centered around **collaboration and interconnected care**. The platform aligns each stakeholder’s needs as follows:

| **Stakeholder** | **Interaction Focus** | **Value Gained** |
| --- | --- | --- |
| **Patients** | Medication adherence & emotional tracking | Better health outcomes, empowerment |
| **Caregivers** | Support and oversight | Peace of mind, accountability |
| **Healthcare Professionals** | Review and analysis | Data-driven insights, improved patient care |
| **Administrators** | Compliance and system management | Security, reliability, scalability |
| **Developers/Tech Teams** | Feature extension & innovation | Platform growth, technical excellence |

This interconnected structure ensures that MediMate remains **patient-centered** while supporting every stakeholder’s goals in a secure and coordinated manner.

### **3.5 Accessibility Considerations**

MediMate is developed with inclusivity in mind. Every target user regardless of age, device literacy, or physical limitations must be able to use the platform effectively.

**Key Accessibility Goals:**

* Intuitive UI with large, legible fonts and clear icons
* Support for **light and dark themes** to reduce eye strain
* Voice prompts or read-aloud features for visually impaired users (future phase)
* Simplified onboarding flow with guided setup for medication schedules
* Localization and multi-language support for broader reach

Accessibility is not treated as an add-on but as a **core design principle**, ensuring equal usability for all.

### **3.6 Target Audience Summary**

MediMate primarily serves:

* **Patients** seeking simplified medication and mood management
* **Caregivers** providing daily or remote assistance
* **Healthcare professionals** (in future updates) using data for care optimization
* **Developers and administrators** ensuring performance, security, and expansion

Together, these audiences form a complete healthcare network built around **trust, simplicity, and continuity of care**. MediMate’s inclusive design and feature scope ensure that every stakeholder from patient to provider benefits from consistent, data-informed health engagement.

## **4.0 – Development Phases**

### **4.1 Overview**

The development of **MediMate** follows a **phased and modular approach**, ensuring a smooth evolution from a core Minimum Viable Product (MVP) to a fully integrated, AI-powered digital health ecosystem.

Each phase is strategically structured to:

* Establish a **stable foundation** before introducing advanced features.
* Allow **continuous testing and validation** through user feedback.
* Maintain **scalability**, **security**, and **compliance** throughout.
* Enable **progressive enhancement** rather than abrupt feature additions.

This phased model allows the development team to focus on **core stability first**, then expand iteratively into emotional analytics, cloud intelligence, and clinician connectivity.

### **4.2 Phase Structure Overview**

| **Phase** | **Title** | **Core Focus** | **Key Stakeholders** |
| --- | --- | --- | --- |
| **Phase 1** | Core Medication System (MVP) | Medication tracking, scheduling, and reminders | Patients |
| **Phase 2** | Mood & Emotional Wellness | Mood check-ins, mood charts, emotional analytics | Patients |
| **Phase 3** | Caregiver & Shared Access | Multi-user roles, caregiver integration, shared dashboards | Patients, Caregivers |
| **Phase 4** | AI Insights & Predictive Analytics | Smart adherence prediction, AI correlation between mood and medication | Patients, Healthcare Professionals |
| **Phase 5** | Clinical & Cloud Expansion | Doctor portal, data export, EMR integration, cloud scaling | Clinicians, Admins |

### **4.3 Phase 1 – Core Medication System (MVP)**

#### **Objective**

Establish a robust foundation that provides the **essential medication management features** ensuring accuracy, usability, and data integrity.

#### **Key Features**

* **User Authentication & Profiles:** Secure account creation using JWT (HttpOnly cookies) for patient privacy.
* **Medication Scheduler:** Define medication name, dosage, time, and recurrence (daily, weekly, custom).
* **Smart Reminders:** Firebase-based notifications with optional snooze or “taken later” responses.
* **Medication Logs:** Auto-record each acknowledgment (Taken, Missed, Skipped).
* **Refill Alerts:** Prompt users when their medication supply is running low.
* **Adherence Reports:** Basic visual analytics (charts and completion percentages).
* **Offline Capability (Optional):** Local data sync using IndexedDB or device storage.

#### **Development Goals**

* Build backend using **Node.js + Express** (or FastAPI alternative).
* Database: **MongoDB** with secure schema design for users, medications, and logs.
* Notifications: **Firebase Cloud Messaging (FCM)**.
* Media Storage (future use): **Cloudinary** for secure image management.
* Authentication: JWT-based with role-based access control (RBAC).

#### **Outcome**

A **fully functional MVP** capable of handling medication scheduling, reminders, and reporting ready for user testing and iterative improvement.

### **4.4 Phase 2 – Mood & Emotional Wellness**

#### **Objective**

Integrate mental health tracking into MediMate, enabling users to monitor **mood patterns, stress levels, and emotional triggers** alongside medication adherence.

#### **Key Features**

* **Mood Check-In System:**  
  Users can log moods using emoji scales, color gradients, or descriptive tags (e.g., Happy, Neutral, Sad, Anxious).
* **Mood Journal:**  
  Optional text-based journaling for users to describe their day, symptoms, or experiences.
* **Mood Chart & Analytics:**  
  Visual graph (bar/line/emoji-based) showing trends over days, weeks, and months.
* **Mood–Medication Correlation (Basic):**  
  A simple algorithm linking mood changes with medication adherence trends (e.g., missed doses vs. mood dips).
* **Privacy Controls:**  
  All mood entries encrypted and accessible only by the user (unless shared).

#### **Developer Focus**

* Frontend integration of **charting libraries** (e.g., Chart.js, Recharts).
* Backend schema expansion: mood\_entries, linked by user\_id and timestamp.
* Secure data relationships between medication\_logs and mood\_logs.
* Lightweight analytics endpoint to compute averages or correlations.

#### **Outcome**

A holistic **Mind–Medication tracking ecosystem** that encourages self-awareness and bridges emotional health with medical adherence paving the way for data-driven insights in later phases.

### **4.5 Phase 3 – Caregiver & Shared Access**

#### **Objective**

Introduce **caregiver functionality** and **multi-user role-based access** to enhance support for dependents, elderly users, or patients under supervision.

#### **Key Features**

* **Role-Based Access (RBAC):**  
  Define roles such as Patient, Caregiver, and Admin.
* **Caregiver Dashboard:**  
  Allows caregivers to view patient medication adherence summaries, missed-dose alerts, and mood charts (with consent).
* **Secure Access Control:**  
  Caregivers access only authorized patient data using permission tokens.
* **Chat & Notification System (Later Addition):**  
  Secure text communication and supportive messaging between caregiver and patient.
* **Audit Logs:**  
  Every caregiver access or change is logged for compliance.

#### **Developer Focus**

* Implement **RBAC Middleware** for access verification.
* Extend existing MongoDB schema to handle multi-user data relationships.
* UI components for caregiver dashboards and summary cards.
* Notification linking via Firebase for both sides (patient ↔ caregiver).

#### **Outcome**

A **collaborative care environment**, where family members and caregivers can assist patients without compromising data privacy or control.

### **4.6 Phase 4 – AI Insights & Predictive Analytics**

#### **Objective**

Elevate MediMate from a passive tracker into an **intelligent, predictive health assistant** using artificial intelligence and cloud computing.

#### **Key Features**

* **AI Adherence Prediction:**  
  Identify potential non-adherence patterns using historical data.
* **Mood–Medication Correlation Analysis:**  
  Machine learning models trained to detect relationships between adherence rates and emotional states.
* **Smart Recommendations:**  
  Suggest optimal medication times, reminders, or lifestyle tips based on behavior patterns.
* **Natural Language Processing (NLP):**  
  Analyze mood journal text entries for sentiment and stress indicators.
* **Anomaly Alerts:**  
  Detect irregular behaviors, such as sudden emotional declines or skipped medications.

#### **Developer Focus**

* Use **Python (TensorFlow / PyTorch)** for AI model training (hosted on GCP).
* Data pipelines to collect and anonymize user data before processing.
* Integration with **Docker** and **Kubernetes** for scalable cloud deployments.
* Ensure compliance with data governance frameworks (HIPAA, GDPR).

#### **Outcome**

A **smart health assistant** capable of early detection, personalized insights, and AI-enhanced emotional intelligence marking MediMate’s entry into digital precision care.

### **4.7 Phase 5 – Clinical & Cloud Expansion**

#### **Objective**

Extend MediMate into clinical environments and ensure its readiness for **enterprise-level scalability, interoperability, and cloud integration**.

#### **Key Features**

* **Doctor/Professional Portal:**  
  Secure interface for clinicians to view patient adherence and mood summaries.
* **Data Export & EMR Integration:**  
  Support for PDF, CSV, and HL7/FHIR data formats for compatibility with hospital systems.
* **Cloud Scalability:**  
  Full backend migration to **Google Cloud Platform (GCP)** with container orchestration via **Kubernetes**.
* **Admin Panel:**  
  Central dashboard for monitoring user activities, managing roles, and handling compliance checks.
* **Advanced Encryption:**  
  End-to-end encryption for sensitive health and emotional data.

#### **Developer Focus**

* Design modular APIs to allow third-party integration (telemedicine, pharmacy, or insurance).
* Optimize system for high concurrency and large-scale data processing.
* Implement full **CI/CD pipeline** using GitHub Actions and containerized deployments.

#### **Outcome**

A mature, cloud-ready, compliant healthcare system capable of serving large user bases and integrating with existing medical infrastructures globally.

### **4.8 Phase Summary and Developer Notes**

| **Phase** | **Focus** | **Technology Stack** | **Output** |
| --- | --- | --- | --- |
| **1** | Core Medication Management | Node.js, MongoDB, Firebase | MVP foundation |
| **2** | Mood & Emotional Health | React, Chart.js, MongoDB | Mood analytics module |
| **3** | Caregiver Support | RBAC, Firebase | Shared care experience |
| **4** | AI Insights | Python, TensorFlow, GCP | Predictive intelligence |
| **5** | Clinical & Cloud Scaling | Docker, K8s, GCP, API Integrations | Enterprise-grade product |

**Developer Notes:**

* Each phase should undergo **unit, integration, and user acceptance testing (UAT)**.
* Use **modular version control branching** (e.g., phase-1-core, phase-2-mood) for clarity.
* Maintain **consistent UI/UX principles** across mobile and web versions.
* Emphasize **data integrity** especially when transitioning between phases.
* Prepare for future **AI dataset expansion** by maintaining clean and anonymized user records.

### **4.9 Summary**

The phased development strategy ensures that MediMate evolves from a **simple medication tracker** into a **comprehensive health companion** capable of emotional analysis, intelligent prediction, and clinical-grade integration.

By prioritizing **stability first**, **intelligence next**, and **scalability last**, MediMate sets a sustainable path for both developers and users, ensuring a consistent, secure, and meaningful digital health experience.

## **5.0 – System Architecture**

### **5.1 Overview**

The **MediMate system architecture** follows a **modular, service-oriented design** optimized for scalability, security, and maintainability across both **mobile** and **web** platforms.  
It leverages a **three-tier architecture** Frontend, Backend (API layer), and Database/Services all interconnected via secure, stateless communication using RESTful APIs and WebSockets (for notifications).

The architecture ensures:

* **Scalable deployment** through containerization (Docker / Kubernetes).
* **Consistent user experience** across platforms (React / React Native).
* **Strict security** with JWT authentication, encrypted tokens, and HTTPS endpoints.
* **Seamless integration** with third-party services like Firebase (notifications) and Cloudinary (media storage).

### **5.2 High-Level Architecture Diagram**

┌────────────────────────────────────────────────────────────┐

│ FRONTEND (Web / Mobile) │

│────────────────────────────────────────────────────────────│

│ React / React Native UI │ Axios / Fetch API │ Firebase │

└───────────────┬────────────────────────────────────────────┘

│ HTTPS / REST API + WebSocket (Secure TLS)

┌───────────────┴────────────────────────────────────────────┐

│ BACKEND (API Layer) │

│────────────────────────────────────────────────────────────│

│ Node.js (Express) / FastAPI │

│ Business Logic / RBAC Middleware / Validation Layer │

│ Notification Service (Firebase SDK) │

│ Data Processing & Analytics Engine │

│ Logging & Monitoring (Winston / PM2) │

└───────────────┬────────────────────────────────────────────┘

│ Mongoose / ORM

┌───────────────┴────────────────────────────────────────────┐

│ DATABASE & STORAGE │

│────────────────────────────────────────────────────────────│

│ MongoDB Atlas │ Cloudinary │ Firebase │ GCP (AI Phase) │

│ Medication, Mood, Logs, User, Caregiver Schemas │

│ Encrypted fields (PHI) + Backup + Replication │

└────────────────────────────────────────────────────────────┘

### **5.3 Frontend Architecture**

#### **Frameworks & Platforms**

* **Web Application:** React.js (SPA)
* **Mobile Application:** React Native (Android + iOS)
* **State Management:** Redux / Recoil
* **Routing:** React Router / Expo Router
* **UI Library:** Tailwind CSS / ShadCN UI components
* **Charting:** Recharts / Chart.js for analytics visualization

#### **Frontend Responsibilities**

1. **User Interface & Experience (UI/UX):**  
   Build a clean, accessible, theme-adaptive interface (light/dark mode support).
2. **Data Interaction:**  
   Communicate with backend via REST APIs using secure HTTPS (Axios).
3. **Authentication Handling:**  
   Store only non-sensitive session tokens in memory; sensitive JWTs handled by HttpOnly cookies.
4. **Push Notifications:**  
   Firebase SDK integration for reminders and status updates.
5. **Offline Mode:**  
   Local cache using IndexedDB / AsyncStorage for medication data when offline.
6. **Visualization:**  
   Display adherence trends, mood charts, and progress summaries.

#### **Developer Notes**

* Use **component modularization** (atomic design: atoms → molecules → organisms).
* Follow **responsive layout** principles for both mobile and web.
* Apply **form validation** using libraries like Yup + Formik.
* Implement **error boundaries** to handle API or network failures gracefully.

### **5.4 Backend Architecture**

#### **Technology Stack**

* **Primary Language:** JavaScript / TypeScript
* **Framework:** Node.js (Express.js) (optionally Python FastAPI for microservices)
* **Authentication:** JWT (HttpOnly cookies) + bcrypt for password hashing
* **RBAC:** Role-based Access Control (Patient, Caregiver, Admin, Doctor – future)
* **Notification Engine:** Firebase Admin SDK
* **Logging:** Winston + Morgan + PM2 monitoring

#### **Core Backend Modules**

| **Module** | **Purpose** |
| --- | --- |
| **User Module** | Handles authentication, registration, and profile management |
| **Medication Module** | CRUD operations for medications, schedules, reminders |
| **Mood Module** | Stores and retrieves mood entries, analytics, and charts |
| **Caregiver Module** | Manages caregiver assignments and shared access |
| **Notification Module** | Manages reminders, alerts, and FCM push triggers |
| **Analytics Module** | Generates adherence and mood correlation reports |
| **Admin Module** | Monitors logs, flags, and system configurations |

#### **Data Flow Example**

1. **User creates medication schedule →**  
   Request sent from frontend via HTTPS POST /api/medications.
2. **Backend validates data →**  
   Saves medication details to MongoDB.
3. **Firebase Notification Scheduler →**  
   Registers reminders based on time, frequency, and timezone.
4. **When user confirms intake →**  
   Log entry created; adherence analytics updated in real-time.

#### **Developer Notes**

* Implement **input validation** using Joi / Zod.
* Separate **business logic** from **controller logic** for clarity.
* Use **async/await** and **try/catch** consistently.
* Employ **environment-based configuration** (dotenv for dev, GCP Secrets Manager for production).

### **5.5 Database Design**

#### **Database Platform**

* **MongoDB Atlas** (NoSQL, cloud-hosted)
* Collections designed for scalability, referencing, and flexible schemas.

#### **Core Collections**

| **Collection** | **Primary Fields** | **Description** |
| --- | --- | --- |
| users | name, email, passwordHash, role, preferences | Core user accounts |
| medications | userId, drugName, dosage, schedule, startDate, endDate | Medication records |
| logs | userId, medicationId, timestamp, status | Adherence events |
| moods | userId, moodLevel, note, timestamp | Emotional tracking |
| caregivers | patientId, caregiverId, permissions | Shared-access control |
| notifications | userId, type, message, sentAt | Reminder history |
| admin\_audit | action, performedBy, timestamp | Audit logs |

#### **Database Security**

* Field-level encryption for PII/PHI.
* Role-based access via JWT.
* Scheduled backups and cloud replication.
* Sanitized queries to prevent injection attacks.

### **5.6 External Services Integration**

| **Service** | **Purpose** | **Integration Notes** |
| --- | --- | --- |
| **Firebase Cloud Messaging (FCM)** | Push notifications for reminders and updates | Linked via Firebase Admin SDK in backend |
| **Cloudinary** | Secure media storage for user images (profile, prescription – later phase) | Configured via environment keys |
| **Google Cloud Platform (GCP)** | AI model hosting, analytics pipelines, and storage | Used in Phase 4+ |
| **GitHub Actions** | Continuous Integration/Deployment | Automates test + build workflows |
| **Docker / Kubernetes** | Containerization + orchestration | Manages scaling and uptime |
| **Stripe / Billing (Future)** | Optional subscription model | Not part of MVP |

### **5.7 Security Architecture**

1. **Authentication & Authorization**
   * JWT stored as HttpOnly cookie for web; Secure AsyncStorage for mobile.
   * Role validation middleware for each API route.
2. **Data Encryption**
   * **At Rest:** MongoDB field-level and database encryption.
   * **In Transit:** HTTPS with TLS 1.2+.
3. **Input Validation & Sanitization**
   * Validate all payloads server-side.
   * Sanitize text fields to prevent XSS or injection.
4. **Logging & Auditing**
   * Every sensitive action logged to admin\_audit.
   * PM2 and Winston rotate logs for performance.
5. **Compliance**
   * Design aligned with **HIPAA** and **GDPR** standards.
   * Explicit user consent for data sharing.

### **5.8 Scalability and Deployment Architecture**

#### **Deployment Model**

* **Development:** Local Docker containers (frontend, backend, DB).
* **Staging:** GCP Compute Engine with test database.
* **Production:** Kubernetes cluster with load-balanced microservices.

#### **Scaling Components**

* **Horizontal Scaling:** Duplicate Node.js pods under load balancer.
* **Vertical Scaling:** Increase resources for MongoDB and FCM handling.
* **Caching (Future):** Redis layer for faster analytics queries.

#### **CI/CD Workflow**

1. **Push to GitHub** → triggers automated tests.
2. **GitHub Actions** builds Docker image.
3. **CI pipeline** deploys to GCP staging cluster.
4. **Manual approval** → promotion to production.

### **5.9 Error Handling and Monitoring**

* **Error Middleware:** Centralized Express error handler returning standardized JSON.
* **Logging:** Winston for file logs; PM2 Dashboard for live metrics.
* **Crash Recovery:** Automatic restarts using PM2 watchdog.
* **Health Checks:** /health endpoint returning system status.
* **Alerting (Future):** Integration with PagerDuty / Slack for critical issues.

### **5.10 Data Flow Summary**

**Example: “Medication Reminder” Flow**

1. **User adds medication** → Frontend sends data to /api/medications.
2. **Backend validates & stores record** in MongoDB.
3. **Scheduler service** registers a reminder in Firebase FCM.
4. **Notification** triggers on user device (mobile / web).
5. **User confirms dose** → API updates adherence log.
6. **Analytics module** refreshes charts and reports in real-time.

This end-to-end flow represents the reliability and modular design philosophy behind MediMate’s system.

### **5.11 Summary**

The MediMate System Architecture provides a **secure, modular, and scalable framework** built for both patient usability and developer efficiency.  
By clearly separating responsibilities between **Frontend**, **Backend**, **Database**, and **External Services**, the system enables easy debugging, feature expansion, and future integration with AI and clinical modules.

This architecture ensures that MediMate evolves smoothly from an MVP medication reminder into a **cloud-native, AI-driven health companion** robust enough for individual users and enterprise-level healthcare systems alike.

## **6.0 – Technical Stack & Tools**

### **6.1 Overview**

The **MediMate Technical Stack** defines the foundational technologies, frameworks, libraries, and tools that power both the **mobile** and **web** applications.

It has been carefully selected to balance:

* **Performance** – optimized response times and smooth UI experiences
* **Scalability** – support for future AI and cloud integrations
* **Security** – full compliance with healthcare data standards (HIPAA, GDPR)
* **Cross-Platform Efficiency** – shared codebase for mobile and web
* **Developer Productivity** – easy maintenance, modular components, and CI/CD automation

This stack supports the MediMate ecosystem from **frontend interaction** to **backend processing**, **database management**, **cloud operations**, and **deployment pipelines**.

### **6.2 Stack Overview Table**

| **Layer** | **Technology** | **Purpose / Role** |
| --- | --- | --- |
| **Frontend (Web)** | React.js | Web-based interface (browser) |
| **Frontend (Mobile)** | React Native | Mobile app (Android/iOS) |
| **State Management** | Redux / Recoil | Global state and data flow |
| **Backend API** | Node.js (Express) / FastAPI (optional) | Core business logic, routing |
| **Database** | MongoDB Atlas | NoSQL data storage (users, medications, moods) |
| **Notifications** | Firebase Cloud Messaging (FCM) | Push notifications for reminders |
| **Media Storage** | Cloudinary | Secure storage for images/media |
| **Hosting / Cloud** | Google Cloud Platform (GCP) | Backend hosting, AI models, scaling |
| **AI / ML (Later Phase)** | Python (TensorFlow, PyTorch) | Predictive analytics and mood correlation |
| **Containerization** | Docker + Kubernetes | Deployment, scaling, and orchestration |
| **CI/CD** | GitHub Actions | Continuous integration & deployment |
| **Security** | JWT (HttpOnly cookies), HTTPS, bcrypt | Authentication and data protection |
| **Version Control** | Git / GitHub | Source code management |
| **Analytics & Charts** | Recharts / Chart.js | Visual representation of mood and adherence data |
| **Testing** | Jest / Mocha / Cypress | Unit, integration, and E2E testing |
| **Documentation & Collaboration** | Notion / Swagger / Postman | API documentation, team collaboration |

### **6.3 Frontend Stack (Web & Mobile)**

#### **Frameworks**

* **React.js (Web)**
  + Chosen for its **component-based architecture** and **virtual DOM efficiency**.
  + Enables **single-page application (SPA)** structure for smooth navigation without page reloads.
  + Integrates seamlessly with REST APIs and supports real-time data updates.
* **React Native (Mobile)**
  + Used for **cross-platform mobile app** (Android and iOS) using a single JavaScript codebase.
  + Delivers near-native performance with native modules (notifications, vibration, camera).
  + Integrates directly with the backend via the same RESTful endpoints used by the web app.

#### **UI & Styling**

* **Tailwind CSS**: For rapid, responsive UI styling with utility-first classes.
* **ShadCN UI**: Provides pre-styled, accessible components (cards, buttons, forms).
* **Framer Motion**: Enables smooth animations and transitions to enhance UX.

#### **State & Data Management**

* **Redux Toolkit** or **Recoil**: Centralized state container to synchronize user data (medications, mood, profile).
* **React Query**: Handles asynchronous data fetching, caching, and synchronization between frontend and backend.

#### **Data Visualization**

* **Recharts / Chart.js**: Used for plotting mood trends, adherence statistics, and medication timelines.

#### **Developer Tools**

* **Vite / Expo CLI** for fast build and development environments.
* **ESLint + Prettier** for code linting and formatting consistency.
* **Jest** for UI component testing.

### **6.4 Backend Stack**

#### **Core Framework**

* **Node.js (Express.js)**
  + Provides an efficient, event-driven environment ideal for real-time applications.
  + Handles routing, business logic, validation, and API responses.
  + Highly scalable and integrates easily with MongoDB and Firebase.
* (Optional) **FastAPI (Python)**
  + Used for future AI microservices (e.g., predictive analytics, NLP processing).
  + Offers high performance with async endpoints and automatic OpenAPI documentation.

#### **API Design**

* RESTful architecture:
  + CRUD endpoints for medications, moods, caregivers, and analytics.
  + Versioned API routes (e.g., /api/v1/medications).
  + JSON-based request and response schema.

#### **Authentication**

* **JWT (JSON Web Tokens):**
  + Stored in **HttpOnly cookies** (web) or **Secure AsyncStorage** (mobile).
  + Protects session data from XSS attacks.
* **bcrypt:** For password hashing and secure credential management.

#### **Notifications**

* **Firebase Admin SDK:**
  + Handles scheduled and on-demand push notifications for medication reminders.
  + Integrates with user preferences and timezone configurations.

#### **Validation & Middleware**

* **Joi / Zod:** Schema-based input validation.
* **Helmet.js:** Adds HTTP headers for app security.
* **Cors:** Configured for secure cross-origin resource sharing.
* **Morgan + Winston:** For structured logging and request tracking.

### **6.5 Database Stack**

#### **Database Engine**

* **MongoDB Atlas (NoSQL, Cloud-Hosted)**
  + Offers flexible document-based data storage ideal for variable medical data.
  + Automatically scales, supports indexing, and handles unstructured data efficiently.
  + Stores complex data relationships (medications ↔ moods ↔ users).

#### **ODM Layer**

* **Mongoose**
  + Simplifies schema creation, validation, and CRUD operations.
  + Enables relationships between collections (e.g., linking user ID with medication and mood logs).

#### **Database Features**

* **Encryption:** Field-level and full-database encryption for sensitive data.
* **Backups:** Daily backups with retention policy.
* **Replication:** Multi-region replication for high availability.
* **Indexing:** Optimized for frequent queries (e.g., reminders by time, user mood logs).

#### **Example Collections**

| **Collection** | **Use** |
| --- | --- |
| users | Profile data, roles, and preferences |
| medications | Drug details, schedule, and frequency |
| logs | History of medication intake |
| moods | Emotional state records |
| caregivers | Linked caregiver access and permissions |
| notifications | Log of reminders and alert events |

### **6.6 Cloud & Hosting Infrastructure**

#### **Google Cloud Platform (GCP)**

* **Compute Engine / Cloud Run:** Host backend services and APIs.
* **Cloud Storage:** Handles logs, exports, and report files.
* **AI Platform (Phase 4+):** Runs predictive models for adherence and mood analytics.
* **Cloud Monitoring:** Tracks uptime, usage, and performance metrics.

#### **Firebase**

* **FCM (Firebase Cloud Messaging):** Sends medication reminders and notifications.
* **Authentication (optional use):** Could integrate for passwordless login in future versions.

#### **Cloudinary**

* Media storage for user uploads (profile images, prescriptions, notes).
* Automatically compresses and secures uploaded files with URLs.

#### **Docker & Kubernetes**

* **Docker:** Packages the entire application into containers for consistency across environments.
* **Kubernetes (GKE):** Manages load balancing, scaling, and container orchestration for production.

#### **GitHub Actions (CI/CD)**

* Automates testing, building, and deployment pipelines.
* Runs static analysis, unit tests, and builds Docker images.
* Triggers deployment workflows on commit to main or release branches.

### **6.7 AI & Analytics (Later Phase Integration)**

#### **Core AI Tools**

* **Python (TensorFlow / PyTorch):** For model training and inference.
* **Pandas / NumPy / Scikit-learn:** For preprocessing, feature extraction, and data correlation.
* **NLP Models:** Analyze user mood journals for sentiment and pattern detection.

#### **Use Cases**

* Predict missed medication patterns.
* Identify emotional trends tied to medication schedules.
* Generate AI-driven adherence recommendations.

#### **Integration Plan**

* AI models trained and hosted on GCP AI Platform.
* Secure API endpoints to serve predictions to Node.js backend.
* Asynchronous task handling using Celery / Google Pub/Sub.

### **6.8 Security & Compliance Tools**

| **Tool / Method** | **Purpose** |
| --- | --- |
| **JWT + RBAC** | Secure user access and permissions |
| **bcrypt** | Secure password hashing |
| **Helmet.js / HTTPS** | Protects against known web vulnerabilities |
| **TLS Certificates** | Enforces encrypted API communication |
| **Audit Logs** | Tracks access, changes, and actions |
| **Env Variables (.env)** | Prevents credential exposure in codebase |
| **HIPAA & GDPR Compliance Frameworks** | Guides design for medical data handling |

### **6.9 Development & Testing Tools**

| **Tool** | **Purpose** |
| --- | --- |
| **Postman / Swagger UI** | API documentation and testing |
| **Jest / Mocha / Chai** | Unit and integration testing |
| **Cypress** | End-to-end UI testing for React apps |
| **Notion / Trello / ClickUp** | Project and sprint management |
| **VS Code / WebStorm** | IDEs for coding and debugging |
| **Prettier + ESLint** | Enforce code style and linting rules |
| **Git + GitHub** | Version control, code reviews, and collaboration |

#### **Testing Goals**

* **Unit Testing:** For backend routes, controllers, and UI components.
* **Integration Testing:** Endpoints and database interaction validation.
* **UI Testing:** Simulate user journeys for reminders, check-ins, and logs.
* **Load Testing:** Ensure stability during high concurrent requests (Phase 5).

### **6.10 Why This Stack?**

1. **Cross-Platform Efficiency:**  
   React + React Native allow a unified codebase for web and mobile.
2. **Scalable Backend:**  
   Node.js (and optional Python microservices) allow modular scaling and integration with AI.
3. **Flexible Database:**  
   MongoDB’s schema-less design suits variable health data formats and evolving features.
4. **Real-Time Communication:**  
   Firebase enables instant reminders and interaction between patients and caregivers.
5. **Secure and Compliant:**  
   JWT, HTTPS, and role-based access ensure safe handling of medical data.
6. **Future-Ready Cloud Infrastructure:**  
   GCP and Docker prepare MediMate for global-scale deployment and continuous delivery.

### **6.11 Summary**

The **MediMate Technical Stack** combines proven, modern technologies that prioritize:

* Developer productivity
* Data integrity
* User security
* Scalability and performance

From the **React-based frontend** to the **Node.js backend** and **GCP cloud environment**, every choice is intentional aligning with MediMate’s mission to provide a secure, intelligent, and reliable health companion app for users worldwide.

This stack provides developers with a clear foundation to **build, extend, test, and maintain** the MediMate ecosystem confidently and efficiently.

## **7.0 – Security and Privacy Framework**

### **7.1 Overview**

The **Security and Privacy Framework** of **MediMate** forms the backbone of its compliance, data integrity, and user trust.

Given the platform’s focus on handling **sensitive medical and emotional data**, it adheres to industry-standard healthcare regulations such as:

* **HIPAA (Health Insurance Portability and Accountability Act)** for U.S. users
* **GDPR (General Data Protection Regulation)** for EU users
* **ISO/IEC 27001** principles for data security management

This framework defines the **authentication**, **authorization**, **encryption**, and **data-handling** mechanisms that safeguard user information across all MediMate systems — web, mobile, and cloud.

Security in MediMate is **proactive**, not reactive — every layer (frontend, backend, database, and cloud infrastructure) implements dedicated controls to protect confidentiality, integrity, and availability of user data.

### **7.2 Core Security Objectives**

MediMate’s security design is built around five key objectives:

1. **Confidentiality** – Ensure that user data (medication, mood, caregiver info) is accessible only to authorized parties.
2. **Integrity** – Protect data from unauthorized modification or tampering.
3. **Availability** – Guarantee that systems and data remain available during operation.
4. **Accountability** – Maintain detailed logs and audit trails for all user and system actions.
5. **Compliance** – Fulfill legal and ethical standards for healthcare data management.

### **7.3 Authentication & Authorization**

#### **A. Authentication Flow**

MediMate uses **JWT (JSON Web Token)**–based authentication with **HttpOnly cookies** on the web platform and **Secure AsyncStorage** on mobile.  
This approach ensures tokens are never directly accessible by client-side JavaScript, reducing XSS vulnerability.

**Flow Summary:**

1. User logs in → credentials validated via /api/auth/login.
2. Backend generates JWT with claims: { userId, role, exp }.
3. Token stored in:
   * **HttpOnly cookie** (web)
   * **Secure AsyncStorage** (mobile)
4. Each request thereafter attaches the token automatically.
5. Backend middleware validates token before granting resource access.

**JWT Expiry:**

* Access tokens: ~24 hours
* Refresh tokens: ~7 days (renewable securely)

**Developer Notes:**

* Use jsonwebtoken for token generation and verification.
* Validate every protected route using verifyToken middleware.
* Avoid exposing JWTs in URL queries or localStorage.

#### **B. Role-Based Access Control (RBAC)**

Authorization in MediMate is enforced through **RBAC**, ensuring that each user role accesses only what’s necessary.

**Defined Roles:**

| **Role** | **Access Scope** |
| --- | --- |
| **Patient** | Manage personal medication and mood data |
| **Caregiver** | View patient summaries, receive alerts (with consent) |
| **Admin** | Monitor system health, manage users, audit logs |
| **Doctor (Future)** | View patient trends and adherence reports |

**Implementation Approach:**

* Role encoded in JWT payload.
* Middleware checks req.user.role against allowed roles per endpoint.
* Routes grouped by access level (e.g., /api/patient/\*, /api/caregiver/\*).

**Developer Tools:**

// Example RBAC Middleware

function authorizeRoles(...roles) {

return (req, res, next) => {

if (!roles.includes(req.user.role))

return res.status(403).json({ message: "Access Denied" });

next();

};

}

**Best Practice:**  
Keep roles and permissions in a centralized roles.config.js file for easier expansion (e.g., adding doctors or admin sub-roles later).

### **7.4 Data Encryption & Protection**

#### **A. Encryption at Rest**

All sensitive data stored in **MongoDB Atlas** is encrypted using **AES-256** at the storage layer.

* **MongoDB Atlas Encryption:** Handles automatic field-level encryption for PHI (Protected Health Information) such as names, emails, and mood entries.
* **Custom Field Encryption:** Implemented using the crypto library for highly sensitive fields before saving to the database (e.g., user notes).

#### **B. Encryption in Transit**

All communication between frontend, backend, and third-party services (Firebase, GCP, Cloudinary) uses **HTTPS with TLS 1.2+**.  
Self-signed certificates are used in development, and verified SSL certificates in production (via Let’s Encrypt or GCP-managed SSL).

#### **C. Password Security**

* Passwords are **never stored in plain text**.
* Each password is hashed with **bcrypt (saltRounds = 10)**.
* Implement rate limiting on login endpoints to prevent brute-force attacks.

#### **D. API Key Management**

* Third-party API keys (Firebase, Cloudinary, GCP) are stored in **environment variables** (.env) or **GCP Secrets Manager**.
* Never hard-coded or exposed in version control.
* Accessed through environment-specific configurations only.

### **7.5 Data Privacy & User Consent**

1. **User Consent Control**
   * Before collecting data (e.g., mood logs, medication schedules), users are explicitly informed about storage and processing.
   * A digital consent screen appears during onboarding.
2. **Data Ownership**
   * Users retain full ownership of their data.
   * MediMate only processes and stores information as required for app functionality.
3. **Right to Erasure (GDPR)**
   * Users can permanently delete their accounts.
   * A delete request triggers data removal from MongoDB, Firebase, and Cloudinary storage.
4. **Anonymized Analytics**
   * AI and analytics phases only use anonymized data (no personal identifiers).
   * Aggregated datasets are used for research insights.
5. **Consent for Caregiver Sharing**
   * Patients grant permission explicitly for caregiver access.
   * Each permission token has an expiry and can be revoked instantly.

### **7.6 Logging, Auditing & Monitoring**

To ensure **accountability** and **traceability**, MediMate logs all critical events.

#### **A. Logging**

* **Winston + PM2** used for structured application logging.
* Log events include:
  + User authentication
  + Data modification
  + API errors or failed access attempts

#### **B. Audit Trails**

* All sensitive actions stored in an admin\_audit collection:
* {
* action: "DELETE\_USER",
* performedBy: "adminID",
* target: "userID",
* timestamp: "2025-10-11T12:30:00Z"
* }
* Helps administrators monitor changes and investigate potential breaches.

#### **C. Monitoring**

* **PM2 Dashboard:** Tracks uptime, memory usage, CPU performance.
* **GCP Cloud Monitoring:** Provides real-time health metrics and alerts.
* **Slack/PagerDuty Integration (Future):** Sends alerts for unusual activity or downtime.

### **7.7 Secure Development Practices**

Developers must follow strict security and privacy best practices during all phases of development.

#### **Code-Level Practices**

* Sanitize user input (prevent XSS and injection).
* Use parameterized queries for MongoDB.
* Avoid hard-coded credentials or tokens.
* Perform static code analysis before deployment.

#### **Dependency Management**

* Use npm audit to detect vulnerabilities.
* Update dependencies regularly and avoid deprecated packages.
* Use **Dependabot** or similar tools for automatic patch management.

#### **Environment Handling**

* All environment variables stored in .env (local) or GCP Secret Manager (production).
* Ensure .env is in .gitignore.
* Separate configurations per environment (dev, staging, prod).

#### **Data Minimization**

* Collect only necessary data for functionality.
* Archive inactive records and auto-purge after retention period (per policy).

### **7.8 Compliance & Regulatory Standards**

| **Regulation / Framework** | **Purpose** | **Implementation in MediMate** |
| --- | --- | --- |
| **HIPAA** | Protects medical information and privacy | PHI encryption, consent control, access logging |
| **GDPR** | EU data protection and privacy | Right to erasure, consent, anonymized data |
| **ISO/IEC 27001** | Information security management | Role-based access, continuous risk assessment |
| **OWASP Top 10** | Common web vulnerabilities | Prevents XSS, SQLi, CSRF, broken authentication |

MediMate’s development pipeline includes **security review gates** before deployment to ensure all modules meet compliance.

### **7.9 Data Backup, Recovery & Incident Response**

#### **A. Backup Strategy**

* **Daily backups** of MongoDB Atlas with 7-day retention.
* Cloudinary and Firebase automatically maintain media redundancy.

#### **B. Disaster Recovery**

* **Multi-zone GCP replication** ensures high availability.
* Automated failover in case of database or API downtime.
* Regular recovery tests performed in staging environments.

#### **C. Incident Response**

1. Detect anomaly via monitoring tools.
2. Lock affected accounts (if user-specific breach).
3. Investigate root cause and isolate impact.
4. Notify affected users and regulatory bodies within 72 hours (GDPR standard).
5. Patch and redeploy the fixed version after code review.

### **7.10 Developer Guidelines for Security Testing**

1. **Penetration Testing:**  
   Perform simulated attacks on staging environments quarterly.
2. **Static Application Security Testing (SAST):**  
   Automated scanning of codebase for security flaws before merges.
3. **Dynamic Application Security Testing (DAST):**  
   Scan deployed apps for runtime vulnerabilities (via OWASP ZAP or Burp Suite).
4. **Data Protection Testing:**  
   Verify encryption and decryption workflows in development and staging.
5. **Access Testing:**  
   Validate RBAC logic — ensure role isolation and permission revocation works correctly.

### **7.11 Summary**

MediMate’s **Security and Privacy Framework** ensures that every user interaction — from login to mood logging — is governed by strong protection measures.  
For developers, it provides **clear implementation standards** for building secure modules and handling sensitive health data responsibly.

This framework not only safeguards users but also establishes MediMate’s reliability as a compliant, trustworthy digital health ecosystem capable of scaling globally while respecting individual privacy.

## **8.0 – Features and Functional Workflows**

### **8.1 Overview**

The **Features and Functional Workflows** section provides a complete breakdown of all the major features integrated into **MediMate**, explaining their **purpose, logic, user flow, and developer implementation**.

MediMate’s functionality centers around **three pillars**:

1. **Medication Management** – Smart scheduling, reminders, and adherence tracking.
2. **Mood & Emotional Health** – Daily emotional logging and visual trend analysis.
3. **Caregiver Integration & Collaboration** – Secure data sharing and supportive care monitoring.

Each feature is designed to work **independently yet interconnectedly**, ensuring that users receive a seamless experience across mobile and web platforms.

For developers, this section outlines **how these features interact across the system architecture**, including their relationship with backend APIs, databases, and notification services.

### **8.2 Core Feature Categories**

| **Feature Category** | **Description** | **Primary Stakeholders** |
| --- | --- | --- |
| **Medication Management** | Enables users to add, manage, and receive reminders for medications. | Patients |
| **Mood Monitoring & Journaling** | Allows users to log emotions, moods, and reflections. | Patients |
| **Adherence Analytics & Reports** | Visualizes user adherence and emotional health trends. | Patients, Caregivers |
| **Caregiver Access & Shared View** | Provides authorized caregivers access to summary data. | Caregivers, Patients |
| **Notifications & Alerts** | Sends push notifications and reminders using Firebase. | All Users |
| **Export & Reporting Tools** | Generates downloadable health summaries and adherence reports. | Patients, Healthcare Professionals |
| **Settings & Privacy Controls** | Gives users control over preferences, visibility, and data permissions. | All Users |

### **8.3 Feature 1 – Medication Management System**

#### **Purpose**

To provide users with a simple, intelligent, and flexible system for managing medications — reducing missed doses and improving treatment adherence.

#### **Functional Flow**

**User Workflow:**

1. User adds a new medication (name, dosage, frequency, start/end date).
2. Backend validates schedule and stores entry in medications collection.
3. System registers medication events with Firebase Scheduler.
4. Push notifications are sent at each scheduled time.
5. User confirms the intake → updates logs collection with status (“Taken”, “Missed”, “Skipped”).
6. Dashboard analytics update adherence rates automatically.

**Developer Implementation:**

* **Frontend:**
  + React form for medication entry with real-time validation (Yup + Formik).
  + Local state syncs with Redux and triggers POST /api/medications.
* **Backend:**
  + Controller: createMedication() handles validation and database write.
  + Scheduler: Firebase Cloud Function triggers reminders based on UTC.
  + Log Update: updateLog() writes user responses (Taken/Missed) to logs.
* **Database Relations:**
  + users → medications (One-to-Many)
  + medications → logs (One-to-Many)

**Key Developer Notes:**

* Use **cron-like logic** or Firebase Scheduled Triggers for repeating reminders.
* Handle **timezone normalization** using moment-timezone.
* Add **retry logic** for failed notification sends.

### **8.4 Feature 2 – Smart Reminders and Notifications**

#### **Purpose**

To intelligently notify users of medication times, refill alerts, and other health-related activities, using Firebase Cloud Messaging (FCM).

#### **Functional Flow**

1. System retrieves all scheduled medication times from the backend.
2. Firebase Scheduler queues reminders based on user timezone.
3. Notification payload includes medication name, dose, and “Mark as Taken” action.
4. User interaction updates backend log via secure API call.
5. Missed notifications trigger a follow-up reminder within 30 minutes (configurable).

#### **Developer Implementation**

* **Backend:**
  + reminderService.js schedules FCM messages.
  + onNotificationClick() API endpoint updates user status.
* **Frontend:**
  + React Native Firebase SDK listens for FCM events and displays system notifications.
  + “Mark as Taken” button triggers backend update.

#### **Developer Notes**

* Ensure **notification tokens** are refreshed periodically (stored in users.deviceToken).
* Use **Firebase Topics** to group reminders by user ID.
* Test all flows using **Firebase Emulator Suite** during development.

### **8.5 Feature 3 – Mood Tracking and Journaling**

#### **Purpose**

To allow users to record emotional states, reflect on mood changes, and visualize emotional progress alongside medication adherence.

#### **Functional Flow**

1. User selects a mood level (e.g., 😄 Happy, 😐 Neutral, 😞 Sad).
2. Optional: user adds a note or short journal entry.
3. Data is submitted to /api/moods → saved in moods collection.
4. Analytics module aggregates moods over time.
5. Mood charts update to show daily and weekly trends.

#### **Developer Implementation**

* **Frontend:**
  + Mood selection built using icons or emojis.
  + Uses Recharts or Chart.js for trend visualization.
* **Backend:**
  + Controller: addMoodEntry() validates and stores data.
  + Optional NLP (Phase 4): analyze note text for sentiment via Python service.
* **Database Relation:**
  + users → moods (One-to-Many)
  + moods correlated with logs (for adherence analysis).

#### **Developer Notes**

* Encrypt journal notes before storing (crypto AES-256).
* Limit daily mood entries to prevent data flooding.
* Use **aggregation pipelines** for computing average mood scores over time.

### **8.6 Feature 4 – Adherence Analytics and Reports**

#### **Purpose**

To provide users and caregivers with visual insights into medication adherence, emotional patterns, and progress trends.

#### **Functional Flow**

1. System fetches medication and mood logs for the selected date range.
2. Backend computes adherence percentage = (taken / total scheduled) \* 100.
3. Mood averages and adherence metrics plotted side-by-side.
4. Dashboard displays visual insights and alerts for missed doses or mood dips.

#### **Developer Implementation**

* **Frontend:**
  + Charts using Recharts with combined line and bar graphs.
  + Filters: by week, month, or custom date range.
* **Backend:**
  + Endpoint /api/analytics/adherence runs aggregation queries in MongoDB.
  + Returns dataset in JSON for frontend chart rendering.

#### **Developer Notes**

* Cache computed analytics using **Redis (future)** for faster load times.
* Provide **export API** to generate downloadable charts as PDF (Phase 5).

### **8.7 Feature 5 – Caregiver and Shared Access**

#### **Purpose**

To enable caregivers (e.g., family members, nurses) to support patients by monitoring medication adherence and emotional patterns under secure, consent-based access.

#### **Functional Flow**

1. Patient invites caregiver via email/QR link → /api/caregiver/invite.
2. Caregiver accepts → access token assigned with role = “caregiver”.
3. Caregiver views summary dashboard (adherence rates, mood trends).
4. Patient can revoke caregiver access anytime.

#### **Developer Implementation**

* **Backend:**
  + caregiverController.js manages invitations, verification, and revocations.
  + RBAC middleware ensures caregivers cannot modify patient data.
* **Frontend:**
  + Separate “Caregiver View” dashboard with summary cards.
  + Consent management toggle under “Settings → Data Sharing”.

#### **Developer Notes**

* Implement **token expiration** for caregiver access links.
* Maintain logs for all shared-data events in admin\_audit.
* All caregiver actions must be **read-only** unless explicitly granted write access.

### **8.8 Feature 6 – Export and Reporting Tools**

#### **Purpose**

To allow users to download or share adherence and emotional progress reports in common file formats (PDF, CSV).

#### **Functional Flow**

1. User selects “Export Report” from dashboard.
2. Frontend triggers /api/export/report with date range.
3. Backend compiles adherence + mood analytics into formatted document.
4. File generated using **ReportLab (PDF)** or **pandas (CSV)** and stored temporarily.
5. Download link returned via secure URL (expires in 1 hour).

#### **Developer Implementation**

* **Backend Libraries:**
  + reportlab for PDFs, pandas for CSV exports.
  + GCP Cloud Storage used for temporary file hosting.
* **Frontend:**
  + Download triggered via secure link with progress feedback.

#### **Developer Notes**

* Implement **token-based download URLs** to prevent unauthorized access.
* Store generated files for limited duration only (auto-delete after 24 hours).

### **8.9 Feature 7 – Settings and Privacy Controls**

#### **Purpose**

To empower users with control over their data, notifications, themes, and privacy preferences.

#### **Functional Flow**

1. User opens Settings panel → retrieves preferences via /api/settings.
2. Toggles notification, theme (light/dark), or caregiver sharing options.
3. Updates saved instantly using PATCH /api/settings/update.
4. Changes reflected in both frontend and backend user profiles.

#### **Developer Implementation**

* **Frontend:**
  + Controlled form components bound to Redux store.
  + Real-time UI switch for dark/light mode.
* **Backend:**
  + User preferences stored in users.preferences.
  + Each preference has default fallback value.

#### **Developer Notes**

* Respect **privacy-first defaults** — all sharing OFF unless user enables.
* Apply **schema validation** for preferences to avoid malformed data.

### **8.10 Inter-Feature Relationships**

The features in MediMate are not siloed — they interact through a shared logic model.

| **Feature** | **Connected To** | **Purpose of Connection** |
| --- | --- | --- |
| Medication | Notifications | Reminder scheduling |
| Mood | Analytics | Correlate mood with adherence |
| Caregiver | Settings | Consent and permission control |
| Export | Analytics | Generate visual reports |
| Notifications | Logs | Update adherence automatically |

These relationships are managed at the **backend layer**, ensuring all data flows remain secure, validated, and consistent across user actions.

### **8.11 Feature Development Best Practices**

* Develop each feature in **isolation (modular approach)** with its own API endpoints, models, and components.
* Maintain consistent naming conventions: featureNameController, featureNameRoutes.
* All feature branches must pass **unit tests** before merging.
* Document API changes in **Swagger** for easy integration testing.
* Keep features backward-compatible during phase rollouts.

### **8.12 Summary**

The **Features and Functional Workflows** section provides developers with a clear map of how MediMate’s systems operate in real-world use.

By integrating medication tracking, emotional wellness, and caregiver collaboration under a secure and modular architecture, MediMate ensures that every feature — from reminders to reports — is reliable, privacy-conscious, and user-centric.

For developers, this section serves as the **functional backbone** of MediMate, guiding implementation, testing, and continuous improvement across all phases of development.

## **Section 9.0 – UI/UX Design Principles**

### **9.1 Overview**

The **UI/UX Design Principles** define the visual and interactive foundation of MediMate.  
They ensure that every interface—across **mobile** and **web**—is intuitive, accessible, and emotionally supportive for users navigating mental and medical wellness journeys.

Since MediMate caters to **patients**, **caregivers**, and **administrators**, the interface must combine simplicity, clarity, and trustworthiness. These principles guide designers and developers in achieving a seamless, consistent, and human-centered experience across platforms.

### **9.2 Core Design Philosophy**

MediMate’s UI/UX design philosophy centers around the following guiding principles:

| **Principle** | **Description** | **Developer Application** |
| --- | --- | --- |
| **Simplicity First** | The interface avoids clutter and unnecessary actions. Each screen focuses on a single primary goal. | Developers should minimize steps per task, avoid overloading screens with buttons, and use clear labels. |
| **Empathy & Calmness** | Since users might be emotionally sensitive, the UI tone must feel calm and non-intrusive. | Choose soft colors, rounded shapes, and smooth transitions. Avoid loud animations or alert tones. |
| **Consistency** | Users should experience predictable navigation and component behavior. | Use shared UI components, typography, and spacing constants across modules. |
| **Accessibility** | The interface must be usable by all individuals, including those with impairments. | Ensure readable color contrast, scalable text, proper alt texts, and voice-over compatibility. |
| **Responsiveness** | Both web and mobile versions must adapt naturally to all screen sizes. | Implement responsive layouts with flexible grids and test across multiple devices. |
| **Data Privacy Awareness** | UI must subtly communicate confidentiality and safety. | Mask sensitive information in notifications, display privacy icons, and respect “incognito” or privacy mode. |

### **9.3 Visual Design System**

To maintain a unified experience, MediMate follows a structured **design system** including:

#### **a. Color Palette**

* **Primary Colors:** Soothing shades of blue and teal (symbolizing trust and calm).
* **Accent Colors:** Muted greens for positivity, subtle orange for alerts.
* **Dark Mode:** Deep charcoal backgrounds with desaturated text tones to reduce glare.

Developers should define these colors in a shared global style file (theme.ts or theme.js) to ensure maintainability and consistency.

#### **b. Typography**

* **Primary Font:** Segoe UI
* **Style Hierarchy:**
  + **H1:** 24–28px – Section titles
  + **H2:** 20–22px – Sub-headings
  + **Body Text:** 14–16px – Main content
  + **Caption/Hint:** 12px – Helper or secondary text

Use consistent font weights and spacing to ensure readability and aesthetic balance.

#### **c. Iconography**

* Use minimal, line-based icons (e.g., from **Lucide** or **Material Icons**).
* Icons should reinforce meaning without overwhelming the interface.
* Color-coded states (e.g., green = success, red = alert, gray = disabled).

#### **d. Imagery and Illustration**

* Images should convey support, hope, and connection.
* Avoid medical imagery that might trigger anxiety.
* Use illustrations or abstract art to maintain emotional neutrality.

### **9.4 UX Interaction Guidelines**

To ensure fluid and supportive user journeys, developers should adhere to the following interaction design rules:

#### **1. Navigation**

* Use a **bottom tab bar** for patients/caregivers (Home, Chat, Tracker, Profile).
* Use a **sidebar layout** for admin dashboards (Reports, Users, Settings).
* Each navigation element must have **active state highlights**.

#### **2. Animations and Transitions**

* All animations should be **soft and purpose-driven** (200–400ms range).
* Use fade-ins or slide-ins for modal views.
* Avoid “bouncy” or distracting motion effects.

#### **3. Forms and Inputs**

* Limit visible fields to essentials.
* Display **inline validation messages** (e.g., “Email format invalid”).
* Use progress indicators for multi-step processes.

#### **4. Feedback and Notifications**

* Success: Green toast/snackbar (“Your entry was saved successfully.”)
* Error: Red toast/snackbar (“Something went wrong. Try again.”)
* Privacy Mode: Show neutral messages (“New update available.” instead of message text).

#### **5. Accessibility Actions**

* All actionable items must have:
  + aria-labels
  + Keyboard focus indicators
  + Proper tab order
  + Voice-over compatibility (for mobile)

### **9.5 Light and Dark Theme Design**

MediMate supports **dual theme modes** to enhance comfort:

| **Theme** | **Use Case** | **Implementation** |
| --- | --- | --- |
| **Light Mode** | Daytime use, clinical settings, or bright environments. | Default palette with soft blues and white backgrounds. |
| **Dark Mode** | Nighttime or stress-reduction mode. | Reversed palette, preserving accessibility and contrast. |

Developers must ensure **theme persistence** across sessions using local storage or user preference saved in backend.

### **9.6 Component Standards**

Each major component (button, card, modal, form field) must adhere to standardized design tokens:

| **Component** | **Behavior** | **Example** |
| --- | --- | --- |
| **Button** | Rounded corners (radius 12–16px), ripple feedback, text in sentence case. | Primary: “Save”, Secondary: “Cancel”. |
| **Card** | Shadow elevation: low, corner radius: 16px, padding: 12–16px. | Used for mood logs, chat previews, or insights. |
| **Modal/Dialog** | Dimmed background, clear close button, focus trapped inside modal. | Used for settings and privacy prompts. |

Create reusable React (web) or Flutter (mobile) components linked to the shared design tokens to avoid UI drift.

### **9.7 Usability Testing & Feedback Loops**

Developers and designers must collaborate through **UI/UX validation sprints**:

* Conduct **A/B testing** for new layouts or interaction flows.
* Collect user feedback through **in-app surveys** or **Firebase Analytics**.
* Track user behavior (screen visits, task completion) to identify friction points.
* Continuously refine interface behavior to improve satisfaction and retention.

### **9.8 Emotional Design Integration**

Because MediMate addresses mental and emotional health, tone and presentation are crucial:

* Always use positive reinforcement (e.g., “Good job!” instead of “Task completed.”)
* Use microinteractions that celebrate progress subtly (gentle confetti, glowing icons).
* Display friendly avatars or wellness illustrations on onboarding screens.

### **9.9 Developer Best Practices**

To maintain design integrity:

1. **Follow the global design tokens** and shared style libraries.
2. **Never hard-code colors or font sizes.**
3. **Respect padding and margin standards.**
4. **Test UI across screen sizes and contrast ratios.**
5. **Collaborate using Figma/Zeplin assets** for accuracy.
6. **Commit reusable UI components** into a shared repo (e.g., /ui-library).

### **9.10 Summary**

The MediMate UI/UX framework is built on **clarity, empathy, and accessibility**.  
By adhering to these principles, developers ensure that each user—whether a patient seeking calm, a caregiver offering support, or an admin reviewing data—feels connected, safe, and understood.

MediMate’s visual and interaction design is not just about how it looks, but how it feels — building digital trust through thoughtful design.

## **Section 10.0 – Database Schema**

### **10.1 Overview**

The **MediMate Database Schema** defines the structural foundation for storing, managing, and retrieving all data across both mobile and web platforms.  
MediMate uses a **NoSQL document-based database (MongoDB)** for flexibility, scalability, and ease of integration with APIs and cloud services.

The schema is designed around collections instead of rigid tables, ensuring that each feature module—such as mood tracking, caregiver communication, and health insights—can evolve independently as the platform expands.

### **10.2 Database Design Philosophy**

MediMate’s database follows three guiding principles:

| **Principle** | **Description** | **Implementation Guidance** |
| --- | --- | --- |
| **Modularity** | Each collection handles a distinct domain (e.g., users, mood logs, chat messages). | Developers can modify or extend individual collections without impacting the others. |
| **Scalability** | Designed for vertical and horizontal scaling in cloud environments. | Use sharding and indexing on high-read/write collections like messages or mood\_logs. |
| **Security-Centric** | Sensitive data (personal info, tokens, reports) are stored securely and encrypted where applicable. | Use field-level encryption and restrict access via role-based control (RBAC). |

### **10.3 Core Collections and Relationships**

Below are the main collections in MediMate and how they relate:

#### **a. users**

Stores the profiles for all system users (patients, caregivers, admins, and future doctors).

**Schema Example:**

{

"\_id": "64ae234abc98d1e7f9c1d112",

"fullName": "Jane Doe",

"email": "jane@example.com",

"passwordHash": "hashed\_password\_string",

"role": "patient",

"gender": "female",

"dob": "1998-03-25",

"profileImage": "https://cloudinary.com/media/jane.png",

"contactNumber": "+447700900123",

"linkedCaregiver": "64ae234abc98d1e7f9c1d113",

"privacyMode": true,

"createdAt": "2025-09-01T09:12:23Z",

"updatedAt": "2025-09-12T09:12:23Z"

}

**Notes for Developers:**

* **Role-based access (RBAC)** is essential for API authorization.
* Store passwords as **bcrypt hashes** (never plain text).
* Link relationships using the \_id reference (ObjectId) instead of embedding full documents.

#### **b. caregivers**

Holds caregiver data and relationships to patients.

**Schema Example:**

{

"\_id": "64ae234abc98d1e7f9c1d113",

"userId": "64ae234abc98d1e7f9c1d112",

"assignedPatients": ["64ae234abc98d1e7f9c1d115", "64ae234abc98d1e7f9c1d116"],

"availabilityStatus": "online",

"specialization": "mental health support",

"lastActive": "2025-10-14T10:23:42Z"

}

**Developer Guidance:**

* userId links to a record in the users collection.
* Ensure two-way relationships: patients can see their caregivers; caregivers can view assigned patients.
* Track activity for analytics and availability indicators.

#### **c. mood\_logs**

Stores daily mood check-ins and emotional states of patients.

**Schema Example:**

{

"\_id": "64ae234abc98d1e7f9c1d210",

"userId": "64ae234abc98d1e7f9c1d112",

"mood": "anxious",

"intensity": 6,

"notes": "Felt stressed about upcoming exam.",

"tags": ["stress", "school"],

"createdAt": "2025-10-14T08:12:12Z"

}

**Developer Guidance:**

* Moods are represented by **string enums** (“happy”, “sad”, “anxious”, “neutral”, etc.).
* Use this data to render charts and trend analytics.
* Keep each entry immutable; corrections should create a new record to preserve logs.

#### **d. chat\_messages**

Handles all one-to-one and caregiver-patient chat exchanges.

**Schema Example:**

{

"\_id": "64ae234abc98d1e7f9c1d300",

"senderId": "64ae234abc98d1e7f9c1d112",

"receiverId": "64ae234abc98d1e7f9c1d113",

"messageText": "Hey, how are you feeling today?",

"attachments": [],

"timestamp": "2025-10-14T10:45:00Z",

"isRead": false,

"deletedForBoth": false

}

**Developer Guidance:**

* Messages are soft-deleted for privacy—no permanent deletion until retention period expires (e.g., 90 days).
* Push notifications are triggered through Firebase when newMessage events occur.
* Limit message attachments (MVP: text + emoji only).

#### **e. notifications**

Handles all system and in-app alerts for users.

**Schema Example:**

{

"\_id": "64ae234abc98d1e7f9c1d400",

"userId": "64ae234abc98d1e7f9c1d112",

"type": "reminder",

"title": "Daily Mood Check-in",

"body": "How are you feeling today?",

"isRead": false,

"createdAt": "2025-10-14T08:00:00Z"

}

**Developer Guidance:**

* Notifications can originate from system tasks or caregiver messages.
* Integrate Firebase Cloud Messaging (FCM) for cross-platform delivery.
* Always respect **privacy mode** (no sensitive message content shown in previews).

#### **f. reports**

Stores exported or generated reports (mood summaries, caregiver feedback, etc.)

**Schema Example:**

{

"\_id": "64ae234abc98d1e7f9c1d500",

"userId": "64ae234abc98d1e7f9c1d112",

"reportType": "monthly\_mood\_summary",

"fileUrl": "https://cloudinary.com/reports/summary\_Sept2025.pdf",

"generatedBy": "system",

"createdAt": "2025-09-30T23:59:59Z"

}

**Developer Guidance:**

* Generated via background jobs or user request.
* Export formats supported: **CSV, PDF**.
* Use this collection for audit logs and download tracking.

#### **g. sessions**

Manages authentication tokens and login states.

**Schema Example:**

{

"\_id": "64ae234abc98d1e7f9c1d600",

"userId": "64ae234abc98d1e7f9c1d112",

"sessionToken": "JWT\_TOKEN\_STRING",

"ipAddress": "192.168.1.105",

"device": "iPhone 15 Pro",

"expiresAt": "2025-10-21T09:00:00Z"

}

**Developer Guidance:**

* Tokens stored as **HttpOnly cookies** on the client for web.
* Mobile apps may store tokens using secure storage (Keychain / EncryptedSharedPreferences).
* Revoke tokens automatically after expiry or on logout.

### **10.4 Relationships Overview**

Even though MongoDB is schema-less, MediMate enforces **logical relationships** through references:

| **Relationship** | **Description** | **Example** |
| --- | --- | --- |
| **User ↔ Caregiver** | Each patient can be linked to one or more caregivers. | user.linkedCaregiver → caregivers.\_id |
| **User ↔ Mood Logs** | Each mood log belongs to a user. | mood\_logs.userId → users.\_id |
| **User ↔ Messages** | Sender and receiver are both users. | chat\_messages.senderId / receiverId → users.\_id |
| **User ↔ Reports** | Reports are generated per user. | reports.userId → users.\_id |
| **User ↔ Notifications** | Notifications are specific to a user. | notifications.userId → users.\_id |

These relationships help simplify aggregation queries (e.g., show patient moods and chat activity in a single dashboard view).

### **10.5 Indexing and Optimization**

To maintain performance and scalability:

* **Create Indexes** on:
  + email (unique)
  + createdAt (for sorting)
  + userId (for frequent lookups)
* **Use TTL Indexes** on time-sensitive collections like sessions and temporary logs.
* **Aggregate Pipelines** for analytics dashboards (e.g., average mood over time).

### **10.6 Example Data Flow**

1. **User Registration:**  
   A new record is added to users.
2. **Mood Check-in:**  
   Creates a mood\_logs entry linked to the user.
3. **Notification Trigger:**  
   Firebase sends a reminder → stored in notifications.
4. **Caregiver Message:**  
   Sent → logged in chat\_messages → triggers Firebase push → visible to receiver.
5. **Monthly Report Generation:**  
   Cron job aggregates mood\_logs → creates PDF → uploads to Cloudinary → record added to reports.

### **10.7 Developer Best Practices**

1. **Never expose internal \_id references** directly in public APIs. Use safe UUIDs where needed.
2. **Validate all incoming data** at the controller level before saving.
3. **Encrypt sensitive fields** (e.g., phone, caregiver notes).
4. **Use transactions** (where needed) when multiple writes depend on each other.
5. **Backup database daily** using automated cloud backup tools.

### **10.8 Future Expansion Considerations**

Future schema extensions will support:

* **Doctor module** (appointments, prescriptions, recommendations)
* **AI insights** collection (predictive emotional patterns)
* **Audit logs** for compliance tracking

### **10.9 Summary**

The MediMate database schema provides a modular, scalable, and secure foundation for all app functionality.  
Each collection reflects a real-world entity—patient, caregiver, report, or message—ensuring logical data flow and long-term maintainability.

By following the schema conventions and referencing rules, developers can maintain **data integrity**, improve **query performance**, and ensure the app’s evolution remains stable and efficient.

## **Section 11.0 – API Documentation**

### **11.1 Overview**

The **MediMate API Layer** serves as the communication bridge between the frontend (mobile/web) and backend services.  
It is implemented as a **RESTful API**, built with **Node.js (Express)**, using **JWT-based authentication** and **role-based access control (RBAC)** to ensure secure and organized data exchange.

All endpoints follow **REST conventions**, use **JSON** as the data format, and include clear HTTP status codes for error handling and debugging.

### **11.2 API Design Philosophy**

| **Principle** | **Description** | **Developer Application** |
| --- | --- | --- |
| **Consistency** | Endpoints follow a predictable naming pattern and use plural nouns. | Example: /api/users, /api/mood-logs |
| **Security First** | All sensitive operations require JWT authentication and role validation. | Include token in the Authorization: Bearer <token> header. |
| **Statelessness** | Each request is independent and contains all necessary data for processing. | Avoid storing session state on the server. |
| **Scalability** | Designed for distributed cloud environments (GCP, AWS). | Support horizontal scaling through load balancers. |
| **Versioning** | Version prefix included for future backward compatibility. | Example: /api/v1/... |

### **11.3 API Base URL**

| **Environment** | **Base URL** |
| --- | --- |
| **Development** | https://dev.api.medimate.com/api/v1/ |
| **Production** | https://api.medimate.com/api/v1/ |

All requests must include:

Content-Type: application/json

Authorization: Bearer <JWT\_TOKEN>

### **11.4 Authentication Endpoints**

#### **1. POST /auth/register**

Registers a new user (patient or caregiver).

**Request:**

{

"fullName": "Jane Doe",

"email": "jane@example.com",

"password": "mypassword123",

"role": "patient"

}

**Response:**

{

"message": "User registered successfully",

"userId": "64ae234abc98d1e7f9c1d112"

}

**Notes for Developers:**

* Passwords are hashed using **bcrypt** before saving.
* Default role is **patient** unless otherwise specified.
* Email uniqueness enforced at the DB level.

#### **2. POST /auth/login**

Authenticates a user and returns a JWT token.

**Request:**

{

"email": "jane@example.com",

"password": "mypassword123"

}

**Response:**

{

"token": "JWT\_TOKEN\_HERE",

"user": {

"id": "64ae234abc98d1e7f9c1d112",

"role": "patient",

"fullName": "Jane Doe"

}

}

**Notes for Developers:**

* Tokens are valid for 7 days.
* Use **HttpOnly cookies** for web sessions.
* Revoke old tokens on password reset.

#### **3. POST /auth/logout**

Logs out a user and invalidates the session.

**Response:**

{ "message": "User logged out successfully" }

### **11.5 User Management Endpoints**

#### **1. GET /users/profile**

Fetches the profile of the authenticated user.

**Response:**

{

"id": "64ae234abc98d1e7f9c1d112",

"fullName": "Jane Doe",

"email": "jane@example.com",

"role": "patient",

"linkedCaregiver": "64ae234abc98d1e7f9c1d113",

"privacyMode": true

}

**Developer Notes:**

* Token required in headers.
* Use this endpoint to populate user dashboards or profile pages.

#### **2. PATCH /users/update**

Updates user profile details.

**Request:**

{

"fullName": "Jane A. Doe",

"privacyMode": false

}

**Response:**

{ "message": "Profile updated successfully" }

**Developer Notes:**

* Validate editable fields (avoid changing role/email directly).
* Enforce access control: a user can only edit their own profile.

### **11.6 Mood Tracking Endpoints**

#### **1. POST /mood-logs**

Logs the user’s current mood.

**Request:**

{

"mood": "calm",

"intensity": 4,

"notes": "Had a relaxing walk in the park"

}

**Response:**

{

"message": "Mood logged successfully",

"moodId": "64ae234abc98d1e7f9c1d210"

}

**Developer Notes:**

* Auto-attaches userId from the JWT token.
* Timestamp added automatically in backend.
* Used in mood tracking charts and analytics.

#### **2. GET /mood-logs**

Fetches all mood entries for the logged-in user.

**Response:**

[

{

"mood": "anxious",

"intensity": 7,

"createdAt": "2025-10-12T08:22:00Z"

},

{

"mood": "happy",

"intensity": 9,

"createdAt": "2025-10-14T10:10:00Z"

}

]

**Developer Notes:**

* Use query filters (?from=2025-10-01&to=2025-10-15) for range-based retrieval.
* Sort moods chronologically for chart generation.

### **11.7 Messaging Endpoints**

#### **1. POST /messages/send**

Sends a new message between a patient and caregiver.

**Request:**

{

"receiverId": "64ae234abc98d1e7f9c1d113",

"messageText": "Hey, I’m feeling better today."

}

**Response:**

{

"message": "Message sent successfully",

"chatId": "64ae234abc98d1e7f9c1d300"

}

**Developer Notes:**

* Triggers Firebase push notification automatically.
* Respect privacy: no message previews on lock screens.

#### **2. GET /messages/conversation/:receiverId**

Fetches the conversation thread between two users.

**Response:**

[

{

"senderId": "64ae234abc98d1e7f9c1d112",

"messageText": "Hey!",

"timestamp": "2025-10-14T09:30:00Z"

},

{

"senderId": "64ae234abc98d1e7f9c1d113",

"messageText": "Hi there! How are you feeling?",

"timestamp": "2025-10-14T09:32:00Z"

}

]

**Developer Notes:**

* Messages auto-marked as read once retrieved.
* Limit messages per query for pagination (?limit=20&page=2).

### **11.8 Notification Endpoints**

#### **1. GET /notifications**

Fetches all notifications for the logged-in user.

**Response:**

[

{

"type": "reminder",

"title": "Daily Mood Check-in",

"isRead": false,

"createdAt": "2025-10-14T07:00:00Z"

}

]

**Developer Notes:**

* Integrates with Firebase Cloud Messaging (FCM).
* Use unread count for notification badges in the UI.

#### **2. PATCH /notifications/mark-read**

Marks notifications as read.

**Request:**

{ "notificationIds": ["64ae234abc98d1e7f9c1d400"] }

**Response:**

{ "message": "Notifications marked as read" }

### **11.9 Reporting & Export Endpoints**

#### **1. GET /reports**

Fetches all generated reports for a user.

**Response:**

[

{

"reportType": "monthly\_mood\_summary",

"fileUrl": "https://cloudinary.com/reports/summary\_Sept2025.pdf",

"createdAt": "2025-09-30T23:59:59Z"

}

]

**Developer Notes:**

* Files hosted on Cloudinary or cloud storage.
* Used for viewing or exporting analytics data.

#### **2. POST /reports/generate**

Generates a new user report manually.

**Request:**

{

"reportType": "weekly\_summary"

}

**Response:**

{

"message": "Report generated successfully",

"fileUrl": "https://cloudinary.com/reports/weekly\_summary\_2025-10-14.pdf"

}

**Developer Notes:**

* Backend aggregates data from mood\_logs.
* Asynchronous process — return status or file link upon completion.

### **11.10 Error Handling and Status Codes**

All MediMate APIs use standard HTTP response codes for clarity.

| **Code** | **Meaning** | **Usage** |
| --- | --- | --- |
| **200 OK** | Successful request. | Standard successful response. |
| **201 Created** | Resource created successfully. | Used after creating users, messages, or logs. |
| **400 Bad Request** | Invalid input data. | Missing required fields or wrong format. |
| **401 Unauthorized** | Invalid or expired token. | JWT missing or invalid. |
| **403 Forbidden** | Access denied due to role restrictions. | Non-admin trying to access admin-only endpoints. |
| **404 Not Found** | Resource not found. | Record does not exist. |
| **500 Internal Server Error** | Unexpected server issue. | Logged for debugging. |

### **11.11 API Security and Privacy Controls**

MediMate APIs enforce strict security measures:

| **Mechanism** | **Description** |
| --- | --- |
| **JWT Authentication** | Tokens signed with a private key and verified on every request. |
| **RBAC (Role-Based Access Control)** | Limits access per role (patient, caregiver, admin). |
| **Rate Limiting** | Protects from API abuse by limiting requests per IP. |
| **HTTPS/TLS** | All API calls must be encrypted. |
| **CORS Rules** | Only approved domains can access the API endpoints. |
| **Input Validation** | Sanitizes request bodies and parameters. |

### **11.12 Developer Usage Guidelines**

1. **Always test endpoints using Postman or Insomnia** before frontend integration.
2. **Store base URLs in environment variables** (.env), not in code.
3. **Use Axios or Fetch API wrappers** for all HTTP requests.
4. **Implement retry logic** for transient network errors.
5. **Cache static data** (like roles or preferences) where applicable.
6. **Log API calls** using Winston/Morgan for debugging and monitoring.

### **11.13 Example Integration (Frontend)**

**React Example using Axios:**

import axios from "axios";

const API = axios.create({

baseURL: "https://api.medimate.com/api/v1",

headers: {

"Content-Type": "application/json"

}

});

// Attach token automatically

API.interceptors.request.use(config => {

const token = localStorage.getItem("token");

if (token) config.headers.Authorization = `Bearer ${token}`;

return config;

});

// Example: Get user profile

export const getUserProfile = async () => {

const { data } = await API.get("/users/profile");

return data;

};

### **11.14 Summary**

The MediMate REST API provides a **modular, secure, and scalable** backbone for communication between all app layers.  
By following the documented standards, developers can easily integrate features across **web and mobile**, ensuring **data consistency**, **security**, and **seamless user experience**.

Each endpoint is designed to be intuitive and predictable, promoting faster development and maintainability across all MediMate environments.

## **Section 12.0 – Notification System (Hybrid Online + Offline)**

### **12.1 Overview**

The **MediMate Notification System** is designed as a **hybrid model**, combining both **online push notifications** (via Firebase Cloud Messaging) and **offline local notifications** (via device OS scheduling).

This ensures that users receive **timely medication and mood reminders**, **caregiver messages**, and **system alerts** — even when they are **not connected to the internet**.

By blending cloud and device-side notification logic, MediMate guarantees **reliability, privacy, and consistency**, regardless of a user’s connectivity status or platform (mobile/web).

### **12.2 Objectives**

| **Objective** | **Description** |
| --- | --- |
| **Continuous Engagement** | Notify users of critical actions (medication, mood check-ins) online or offline. |
| **Hybrid Reliability** | Combine local scheduling with cloud delivery to ensure no missed reminders. |
| **Cross-Platform Support** | Unified experience for Android, iOS, and Web. |
| **Data Sync Integrity** | Synchronize offline logs once the user reconnects to the internet. |
| **Privacy & Control** | Deliver sensitive alerts safely under privacy and mute settings. |

### **12.3 Hybrid Notification Architecture**

MediMate uses a **dual-layer notification system**:

| **Layer** | **Mode** | **Function** | **Works Offline?** | **Trigger Source** |
| --- | --- | --- | --- | --- |
| **Local Notification Engine** | Device-level | Schedules and fires medication/mood reminders. | ✅ Yes | App (client-side) |
| **Cloud Notification Engine** | Server-side | Sends chat messages, system alerts, and cross-device updates. | ❌ No | Backend (Firebase + API) |
| **Sync Layer** | Middleware | Syncs local events (delivered, dismissed, completed) when back online. | ✅ (queued) | App background task |
| **Backend Storage** | MongoDB | Stores notification logs, history, and analytics. | ✅ (syncs later) | Server |

### **12.4 Notification Categories**

| **Category** | **Purpose** | **Works Offline?** | **Example** |
| --- | --- | --- | --- |
| **Medication Reminder** | Remind users to take prescribed medications. | ✅ | “Time to take 1 tablet of Lisinopril 10mg.” |
| **Mood Check-In Reminder** | Encourage emotional self-tracking. | ✅ | “How are you feeling today? Record your mood.” |
| **Medication Refill Alert** | Warn users before prescription depletion. | ❌ | “Your Sertraline prescription is running low.” |
| **Caregiver Message** | New message or advice from caregiver. | ❌ | “New message from Dr. Lee.” |
| **Report Generated** | Notify when summary or adherence report is ready. | ❌ | “Your weekly medication report is available.” |
| **System Alert** | Platform updates or app changes. | ❌ | “System maintenance scheduled for 11 PM.” |

### **12.5 Hybrid Notification Flow**

#### **1. When Online**

[Trigger Event] → [Backend Logic] → [Firebase Cloud Messaging] → [User Device]

* Chat messages, refills, and system updates are sent from backend via Firebase.
* Medication and mood reminders are **also scheduled locally** for redundancy.

#### **2. When Offline**

[User Action or Pre-set Reminder] → [Local Scheduler] → [Device Notification Center]

* Local reminders (saved in device storage) fire at the scheduled time even without internet.
* User interactions (e.g., “Taken”, “Skipped”, “Snoozed”) are **logged locally** for later sync.

#### **3. Upon Reconnection**

[App Startup / Background Sync] → [Sync Service] → [Backend API] → [Update Notifications Collection]

* Local logs (delivered reminders, completion states) are uploaded to the server.
* Missed Firebase pushes are fetched and displayed as in-app notifications.

### **12.6 Database Schema**

**Collection:** notifications

{

"\_id": "64ae234abc98d1e7f9c1d400",

"userId": "64ae234abc98d1e7f9c1d112",

"type": "medication\_reminder",

"title": "Morning Dose",

"body": "Take 1 tablet of Paroxetine 20mg.",

"isRead": false,

"deliveryMode": "local",

"synced": false,

"scheduledTime": "2025-10-16T08:00:00Z",

"createdAt": "2025-10-16T07:59:00Z"

}

**Developer Notes:**

* deliveryMode: "local" or "cloud".
* synced: false → true after the reminder is uploaded from device to server.
* Local logs (offline events) are inserted with synced: false and updated post-connection.

### **12.7 Local Notification Scheduling (Offline)**

Local notifications are scheduled and managed directly by the device OS or framework (Flutter, React Native, Swift, Kotlin).

**Example (JavaScript / React Native):**

import PushNotification from "react-native-push-notification";

PushNotification.localNotificationSchedule({

title: "Medication Reminder",

message: "Take 1 tablet of Lisinopril 10mg.",

date: new Date(Date.now() + 60 \* 1000), // 1 min from now

allowWhileIdle: true,

repeatType: "day"

});

**Example (Flutter):**

await flutterLocalNotificationsPlugin.zonedSchedule(

0,

'Mood Check-In',

'How are you feeling today?',

scheduledTime,

NotificationDetails(

android: AndroidNotificationDetails('reminder', 'Reminders')

),

androidAllowWhileIdle: true,

uiLocalNotificationDateInterpretation:

UILocalNotificationDateInterpretation.absoluteTime,

);

**Offline Logic:**

* The schedule is saved locally (in SQLite/AsyncStorage).
* Even without internet, the OS triggers it at the correct time.
* A local log entry is queued for syncing when reconnected.

### **12.8 Cloud Notification Delivery (Online)**

Uses **Firebase Cloud Messaging (FCM)** for real-time server-to-device delivery.

**Backend (Node.js) Example:**

const admin = require("firebase-admin");

admin.initializeApp({ credential: admin.credential.cert(process.env.FIREBASE\_CREDENTIALS) });

async function sendCloudNotification(token, title, body, data) {

await admin.messaging().send({

token,

notification: { title, body },

data

});

}

* Cloud notifications are used for **caregiver messages**, **system alerts**, **refill reminders**, and **report availability**.
* If the device is offline, FCM queues the message for up to 4 weeks until reconnection.

### **12.9 Offline Logging and Sync Service**

When offline, MediMate stores pending notifications and actions in a local log:

{

"id": "local\_17345",

"userId": "64ae234abc98d1e7f9c1d112",

"type": "medication\_reminder",

"status": "delivered",

"timestamp": "2025-10-16T08:00:00Z",

"synced": false

}

#### **Sync Process (On Reconnection):**

1. App detects network restoration.
2. Checks unsynced local entries (synced == false).
3. Sends batch sync request:
4. POST /api/v1/notifications/sync
5. Server merges entries, updates analytics, and marks logs as synced.

### **12.10 Scheduler and Automation**

| **Task** | **Engine** | **Frequency** | **Description** |
| --- | --- | --- | --- |
| **Medication Reminder Scheduler** | node-cron / device OS | Configurable | Fires per dosage schedule. |
| **Mood Check-in Reminder** | node-cron / device OS | Daily | Default 9 AM or user-defined time. |
| **Refill Reminder** | Server | Weekly | Alerts before prescription depletion. |
| **Sync Task** | App Service Worker / Background Task | Periodic | Syncs local logs to backend when connected. |

### **12.11 Privacy and Control**

| **Control** | **Description** | **Developer Implementation** |
| --- | --- | --- |
| **Privacy Mode** | Hides sensitive medication/mood info in alerts. | Display neutral messages (“You have a new health update”). |
| **Mute Mode** | Temporarily disables all reminders. | Check muteUntil before scheduling/sending. |
| **Notification Preferences** | Users toggle which alerts they receive. | Store preferences in user\_settings collection. |
| **Local Data Encryption** | Protects stored offline logs. | Use AES encryption before saving locally. |

### **12.12 Error Handling & Retry Logic**

| **Scenario** | **Behavior** | **Action** |
| --- | --- | --- |
| No Internet | Fire local notification; mark unsynced. | Log event locally. |
| Firebase Token Expired | Remove token, re-register on next login. | Regenerate via FCM SDK. |
| Delivery Failure | Retry with exponential backoff (max 3). | Log failure in logs collection. |
| Sync Conflict | Server resolves duplicates by timestamp. | Keep latest state. |

### **12.13 Developer Implementation Guidelines**

1. Always **schedule a local backup** for every time-based reminder.
2. Use **offline-first data models** — sync later when possible.
3. Store **FCM tokens securely** and refresh when changed.
4. Implement **background sync** for Android and iOS.
5. Respect **time zones** and local daylight savings for reminders.
6. Use **notification grouping** for multiple alerts to prevent clutter.
7. Integrate **delivery analytics** (sent, fired, opened, dismissed).

### **12.14 Example Use Cases**

#### **1. Offline Medication Reminder**

* User schedules “8 AM medication”.
* Internet disconnects overnight.
* OS fires local notification at 8 AM.
* When reconnected, log syncs:
* { type: "medication\_reminder", status: "delivered", synced: true }

#### **2. Offline Mood Check-In**

* User travels with no signal.
* Local scheduler triggers 9 AM check-in.
* User logs mood locally → synced automatically when online.

#### **3. Caregiver Message While Offline**

* Sent via Firebase → queued until device reconnects.
* On reconnect, push received + in-app message synced.

### **12.15 Analytics and Monitoring**

Track both online and offline notification performance:

| **Metric** | **Description** | **Source** |
| --- | --- | --- |
| **Delivery Rate** | % of all notifications delivered successfully. | FCM + local logs |
| **Open Rate** | % of notifications user interacted with. | App analytics |
| **Sync Success** | % of offline logs synced successfully. | Backend API |
| **Reminder Adherence** | % of medication/mood logs completed on time. | Combined reports |

### **12.16 Summary**

The **Hybrid Notification System** is a cornerstone of MediMate’s user engagement strategy.  
It ensures that every **reminder, update, or message** is delivered — whether online or offline.

By combining:

* **Local Scheduling (offline resilience)**
* **Firebase Cloud Messaging (real-time delivery)**
* **Sync Engine (data reconciliation)**

MediMate guarantees **consistent, reliable, and secure communication** — supporting both the **mood** and **medication** management pillars of user wellness.

Even when disconnected, users remain supported.  
Once reconnected, MediMate synchronizes seamlessly — ensuring nothing is missed.

## **Section 13.0 – Export and Reporting Feature**

### **13.1 Overview**

The **Export and Reporting Feature** in **MediMate** empowers users to generate, view, and share personalized reports related to their **mood tracking**, **medication adherence**, and **health analytics**.  
This feature is designed for **patients**, **caregivers**, and **clinicians** (when available in future phases), enabling data-driven insights while maintaining privacy and data integrity.

It supports **offline generation** (local cache-based) and **online synchronization** (cloud export), ensuring usability even without active internet connectivity.

### **13.2 Core Objectives**

1. **Empower Users:** Allow users to view and analyze their data trends (mood charts, medication history, reminders).
2. **Simplify Data Sharing:** Enable exporting to common formats (PDF, CSV, and in later phases, XLSX or JSON).
3. **Support Caregiver Collaboration:** Facilitate sharing reports securely with caregivers or healthcare providers.
4. **Ensure Privacy:** Implement data redaction and encryption mechanisms for sensitive information.
5. **Enable Hybrid Operation:** Allow offline report creation and deferred cloud syncing when connectivity resumes.

### **13.3 Key Functionalities**

| **Feature** | **Description** |
| --- | --- |
| **Export Options** | Users can export data in formats: PDF (visual summary), CSV (raw data), and JSON (for system integration). |
| **Report Categories** | - Mood Progress Reports - Medication History - Reminder Compliance Logs - AI Insights (future phase) - System Usage Analytics (admin only) |
| **Data Filters** | Exports can be filtered by date range, category (mood/medication), or tags. |
| **Visual Analytics** | Charts, graphs, and trends (line, bar, pie) embedded in exported reports. |
| **Offline Mode Support** | Export requests initiated offline are stored locally and processed once the device reconnects. |
| **Secure Sharing** | Users can share export links (expiring URLs) or download encrypted report files. |
| **Audit Logging** | Every export action is logged (who exported, timestamp, data type). |
| **Multi-Platform Sync** | Reports are available across mobile and web through cloud synchronization. |

### **13.4 Workflow and Process Design**

#### **Step 1 – Data Retrieval**

* Data is fetched from the **local database** (for offline mode) or **cloud Firestore/DB** when online.
* Each module (Mood, Medication, Reminders) has its data source endpoint.

#### **Step 2 – Data Processing**

* Data is normalized and formatted based on export type.
* Charts are rendered using libraries like **Chart.js** or **Recharts** (React Web) and **MPAndroidChart** or **Charts (iOS)** for mobile.

#### **Step 3 – Report Generation**

* Generated via the **Report Builder Module**, which uses templates (e.g., HTML-to-PDF or markdown parsers).
* Libraries:
  + **PDF Generation:** pdfkit / reportlab (backend)
  + **CSV Generation:** Node json2csv, Python pandas
  + **Offline Storage:** Local SQLite or IndexedDB

#### **Step 4 – Export/Save Options**

* **Download to device**
* **Save to cloud** (Firebase Storage or Cloudinary)
* **Share as link/email attachment** (secured tokenized link)

#### **Step 5 – Logging & Sync**

* Each export action is logged to:
  + Local log (if offline)
  + Cloud Firestore logs (when online)
* Once online, pending logs are auto-synced with timestamp validation.

### **13.5 Developer Implementation Notes**

#### **Frontend**

* **React (Web)** and **Flutter/React Native (Mobile)** handle user interactions and UI visualization.
* Use background threads for report generation to avoid UI freezing.
* Display real-time progress and allow “cancel export” options.

#### **Backend**

* Export service endpoint: /api/v1/reports/export
* Responsible for:
  + Querying user data (MongoDB)
  + Applying filters (date range, tags)
  + Generating output file (PDF/CSV)
  + Returning a downloadable file or signed URL

#### **Database Layer**

* **MongoDB** stores:
  + Export metadata: { userId, type, format, timestamp, status }
  + Report logs and activity records
* Indexed for faster queries.

#### **Authentication**

* All exports are verified using **JWT tokens** stored in **HttpOnly cookies**.
* Role-based restrictions:
  + Patients: Personal data only
  + Caregivers: Linked users’ reports (with consent)
  + Admin: System-wide analytics (future)

#### **Cloud Integration**

* Cloud Storage (Firebase or Cloudinary) for report backup and sharing.
* Cloud Functions handle background exports and cleanup of old reports.

### **13.6 Offline and Hybrid Operation**

* Exports initiated while offline:
  + Stored locally as “Pending Reports”.
  + When internet is restored:
    - Automatically synced and uploaded.
    - User notified via **Local → Push Notification** sync bridge.
* All exports and logs maintain **integrity timestamps** to ensure consistency after reconnection.

### **13.7 Security and Privacy Considerations**

1. **Data Encryption:**
   * Reports are encrypted with AES before storage/upload.
2. **Access Tokens:**
   * Exported URLs are protected with expiring tokens (24h default).
3. **Anonymization:**
   * Identifiers can be redacted when sharing externally.
4. **Role-based Access Control (RBAC):**
   * Prevent unauthorized report generation or download.
5. **GDPR/PHI Compliance:**
   * Ensure user consent for all exports involving sensitive health data.

### **13.8 Future Enhancements**

* **Automated Periodic Reports:** Monthly or weekly auto-generated summaries.
* **Custom Report Templates:** User-selectable design or format themes.
* **AI-Powered Insights:** Trend analysis, medication impact correlation.
* **Clinician Dashboard Integration:** Direct doctor access to reports (with patient permission).
* **Integration with Cloud Drive APIs:** e.g., Google Drive, OneDrive.

### **13.9 Example Use Cases**

| **Use Case** | **Description** |
| --- | --- |
| **Patient Report** | A patient exports their last 30 days of mood and medication data as a PDF to discuss with their doctor. |
| **Caregiver Summary** | A caregiver generates a CSV export of all adherence logs for analysis. |
| **Offline Export Sync** | User generates an export offline; once the device reconnects, the report uploads automatically. |
| **Admin Analytics** | Admin retrieves a system-wide usage summary (only in later phase). |

### **13.10 Developer Checklist**

* Implement /api/v1/reports/export endpoint
* Add export triggers in frontend dashboards
* Build local queue for offline export tasks
* Add AES encryption to exported files
* Create report templates (HTML/PDF)
* Enable background syncing with Firebase
* Write logs to both local and cloud audit trail

# **Section 14.0 – AI and Analytics (Future Phase)**

## **14.1 Purpose**

The AI and Analytics module is a **future-phase extension** of MediMate designed to transform raw data into actionable insights. Its primary goals are:

1. **Predictive Mood Insights:** Use historical mood check-ins, sleep patterns, and behavioral data to forecast mood trends and potential mental health events.
2. **Medication Guidance:** Predict adherence issues, detect possible drug-drug interactions, and anticipate side effects based on user medication history.
3. **Personalized Self-Care Recommendations:** Suggest lifestyle adjustments, therapy prompts, or medication adjustments based on predictive modeling.
4. **Operational Analytics:** Offer dashboards for administrators and healthcare providers to monitor app usage, engagement, and intervention effectiveness.
5. **Proactive Health Alerts:** Detect anomalies in behavior, mood, or medication adherence and trigger automated alerts or notifications.

This module aims to provide **predictive, rather than reactive, insights** to improve patient outcomes and engagement.

## **14.2 Architecture Overview**

The AI & Analytics system is **microservices-based** to enable scalability, modularity, and independent deployment. Key components:

### **14.2.1 Data Ingestion Layer**

* Collects anonymized data from:
  + Mood check-ins
  + Medication logs
  + Sleep patterns and activity metrics
  + Optional sensor data (heart rate, step count)
* **Offline Support:** Local caching of user activity; syncs to cloud when back online.
* **Tech Stack:** Firebase, MongoDB, secure REST APIs.

### **14.2.2 Preprocessing & Feature Engineering Service**

* Cleans, normalizes, and encodes data for model training.
* Handles missing or inconsistent offline submissions.
* Derives features such as:
  + Mood trend vectors
  + Medication adherence ratios
  + Drug-drug and drug-side effect matrices
* **Tools:** Python (Pandas, NumPy), Apache Spark (batch processing if needed).

### **14.2.3 Model Training Service**

Supports multiple AI/ML models:

| **Model Type** | **Purpose** | **Notes for Devs** |
| --- | --- | --- |
| Mood Prediction | Forecast mood trends (classification or regression) | LSTM/GRU for time-series, lightweight Random Forest for MVP |
| Medication Adherence | Predict likelihood of missed doses | Use historical intake + context features (time, user routine) |
| Drug Interaction & Side-Effect Detection | Detect potential harmful combinations or side-effects | Integrate publicly available drug databases; use NLP to map symptoms & interactions |
| Anomaly Detection | Flag sudden deviations in mood or behavior | Can trigger alerts to user or caregiver |
| Personalized Recommendation Engine | Suggest self-care activities, therapy prompts | Content can be rule-based initially, later AI-driven |

* **Environment:** Cloud GPU servers (GCP AI Platform, AWS Sagemaker).
* **Pipeline:** Dockerized containers; automated CI/CD using GitHub Actions.

### **14.2.4 Model Serving & Inference Service**

* Exposes models via **REST/gRPC microservices**.
* Supports **real-time inference** for individual users and **batch inference** for analytics dashboards.
* Scalable deployment via Kubernetes (K8s).

### **14.2.5 Analytics & Visualization Layer**

* Dashboards for:
  + Users: Mood trends, predicted adherence issues, self-care suggestions.
  + Caregivers: Alerts for potential mood deterioration, missed medications.
  + Admins: Engagement analytics, retention, feature usage.
* **Tools:** Plotly Dash, Grafana, or Power BI; MongoDB aggregations for metrics.

## **14.3 Data Privacy & Security**

* **Anonymization:** All AI processing uses de-identified user data.
* **Encryption:** TLS 1.3 for data in transit; AES-256 at rest.
* **Regulatory Compliance:** HIPAA, GDPR, local health laws.
* **Role-Based Access Control (RBAC):** Separate permissions for patients, caregivers, doctors, and admins.
* **Audit Logging:** Track all AI model queries and data access.

## **14.4 Development Workflow**

1. **Data Collection & Syncing**
   * Ensure offline events (mood check-ins, medication logs) are cached locally.
   * Synchronize securely when online.
2. **Data Preprocessing & Feature Engineering**
   * Transform raw logs into structured datasets.
   * Encode drug combinations for interaction detection; map symptoms for side-effect prediction.
3. **Model Training**
   * Start with classical ML (Random Forest, Gradient Boosted Trees).
   * Progress to deep learning (LSTM, GRU) for time-series mood prediction.
   * Train drug interaction models using NLP on medical datasets.
4. **Deployment & Microservices**
   * Containerize AI models with Docker.
   * Deploy via K8s or serverless functions for real-time prediction.
5. **Feedback Loop & Continuous Learning**
   * Incorporate user feedback, adherence outcomes, and new data to retrain models.
   * Maintain versioned models with rollback capability.

## **14.5 Use Cases & Developer Notes**

| **Use Case** | **Description** | **Developer Notes** |
| --- | --- | --- |
| Mood Prediction | Forecast future mood states | LSTM-based sequence modeling; offline data sync required |
| Medication Adherence Prediction | Detect potential missed doses | Can trigger reminders; integrate with notification service |
| Drug Interaction & Side-Effect Detection | Identify high-risk medication combinations | Requires integration with drug database (e.g., FDA, MedlinePlus); NLP for symptom mapping |
| Personalized Self-Care Suggestions | Suggest therapy, lifestyle adjustments | Can combine AI prediction with rule-based recommendations |
| Anomaly Detection | Alert on sudden behavioral or mood changes | Threshold-based alerts; notify caregiver/admin if critical |
| Analytics Dashboards | Visualize trends and engagement | Use pre-aggregated metrics to minimize latency |

## **14.6 Future Considerations**

* **On-device inference** for privacy-preserving predictions.
* **Federated learning** to train models across devices without centralizing sensitive data.
* **Multimodal AI:** Integrate voice journaling, textual input, and biometric sensors.
* **Explainable AI (XAI):** Make predictions interpretable for users and caregivers.
* **Adaptive Recommendations:** Continuously refine self-care suggestions based on predictive success.

# **Section 15.0 – DevOps and CI/CD Pipeline**

## **15.1 Purpose**

The DevOps and CI/CD pipeline for MediMate ensures **rapid, reliable, and automated deployment** of both mobile and web applications. Its objectives include:

1. **Automated Builds & Testing**: Ensure that all code changes are verified via unit, integration, and UI tests before deployment.
2. **Continuous Integration (CI)**: Merge code changes from multiple developers frequently, detecting integration issues early.
3. **Continuous Deployment/Delivery (CD)**: Automatically deploy stable builds to staging and production environments.
4. **Infrastructure as Code (IaC)**: Standardize environment setup for local, staging, and production using reproducible scripts.
5. **Monitoring & Rollbacks**: Detect failures quickly, monitor application health, and enable safe rollbacks if needed.

## **15.2 Architecture Overview**

The DevOps setup integrates **source control, CI/CD pipelines, automated testing, containerization, and cloud deployment**.

### **15.2.1 Version Control**

* **GitHub** is used as the main source code repository for both web and mobile apps.
* Branching strategy:
  + main – Production-ready code
  + develop – Integration of new features
  + Feature branches – Individual feature development

### **15.2.2 Continuous Integration (CI)**

* Automated build pipeline triggers **on every push or pull request**.
* Key steps:
  1. **Code Checkout:** Pull latest code from GitHub branch.
  2. **Dependency Installation:** Install required packages (Node.js, Python, etc.).
  3. **Static Code Analysis:** Linting and code quality checks.
  4. **Unit & Integration Testing:** Run automated tests for backend, frontend, and AI microservices.
* Tools: GitHub Actions, Jenkins, or GitLab CI.

### **15.2.3 Continuous Deployment/Delivery (CD)**

* Deployment pipeline automatically promotes **CI-verified builds** to staging or production.
* Steps:
  1. **Build Docker Images** for backend, AI microservices, and frontend.
  2. **Push Images** to container registry (e.g., DockerHub, GCP Artifact Registry).
  3. **Deploy to Cloud Environment** using Kubernetes or serverless functions.
  4. **Run Smoke Tests** to verify deployment success.
  5. **Promote to Production** after approval or automatic verification.

## **15.3 Cloud Environment Setup**

* **Cloud Provider:** GCP (Google Cloud Platform) for hosting web, backend, and AI microservices.
* **Services Used:**
  + **Compute:** GKE (Kubernetes Engine) for microservices deployment
  + **Storage:** Cloud Storage for media; MongoDB Atlas for database
  + **Messaging/Notifications:** Firebase Cloud Messaging
  + **Secrets Management:** Secret Manager for API keys, DB credentials
* **Infrastructure as Code (IaC):** Terraform or Google Deployment Manager to define and provision cloud resources programmatically.

## **15.4 DevOps Workflow**

1. **Code Commit:** Developer pushes feature or bug-fix branch.
2. **CI Pipeline Trigger:** GitHub Actions triggers linting, tests, and builds.
3. **Build Artifacts:** Docker images for services are built and tagged.
4. **Staging Deployment:** Images deployed to a staging environment for QA testing.
5. **Automated Tests:** Integration, regression, and smoke tests run automatically.
6. **Approval & Production Deployment:** After QA approval or automated verification, deployment proceeds to production.
7. **Monitoring & Alerts:** Application and infrastructure health monitored via Prometheus, Grafana, or Cloud Monitoring.

## **15.5 Key Developer Notes**

* **Environment Parity:** Ensure local, staging, and production environments are consistent via Docker and IaC.
* **Secrets & Configs:** Never hardcode credentials; use environment variables or cloud secret managers.
* **Rollback Mechanism:** Maintain previous stable Docker images to allow instant rollback if deployment fails.
* **Automated Testing Coverage:** Critical to prevent regression, especially for AI microservices and offline syncing features.
* **Scalability:** Microservices architecture allows scaling individual components (AI, notification, backend) independently.

## **15.6 Future Considerations**

* **Blue-Green Deployment:** Minimize downtime during production updates.
* **Canary Releases:** Gradually roll out features to a subset of users to monitor performance and errors.
* **CI/CD for AI Models:** Automate retraining, testing, and deployment of AI/ML models as part of the pipeline.
* **Monitoring & Logging Enhancements:** Implement centralized logging (ELK Stack) and AI-powered anomaly detection in infrastructure.

# **Section 16.0 – Coding Standards and Best Practices**

## **16.1 Overview**

This section defines the **coding standards, architectural conventions, documentation rules, and best practices** adopted throughout the **MediMate development ecosystem** (mobile, web, and backend).  
Its goal is to ensure **clean, consistent, and maintainable code**, improve **team collaboration**, and guarantee **scalability and security** across the entire system.

The standards apply to **all contributors**, including frontend developers, backend engineers, data scientists, AI/ML specialists, and DevOps teams.

## **16.2 Objectives**

* To establish a **uniform coding style** across all MediMate repositories.
* To enforce **best engineering principles** such as modularity, readability, and performance optimization.
* To ensure **code quality through automated linting, peer review, and CI/CD pipelines.**
* To simplify onboarding of new developers by maintaining **predictable and documented code patterns.**

## **16.3 Code Structure and Organization**

### **a. Folder Structure Guidelines**

Each repository (mobile, web, backend, or AI) should adopt a **modular structure** for scalability:

/src

/components → Reusable UI components

/screens → Page or feature-specific UI

/services → API and business logic

/models → Data models or interfaces

/hooks → Custom React hooks (for web/mobile)

/assets → Images, fonts, icons

/utils → Helper or utility functions

/tests

/unit → Unit test cases

/integration → Integration test cases

**Note:** For backend or API services, similar structure applies but may include /controllers, /routes, /middlewares, and /database.

## **16.4 Naming Conventions**

| **Element** | **Convention** | **Example** |
| --- | --- | --- |
| **Files/Folders** | lower-case-with-hyphens | user-profile.js, medication-service.ts |
| **Variables** | camelCase | userProfile, medicationList |
| **Constants** | UPPER\_CASE | MAX\_RETRY\_LIMIT |
| **Classes / Components** | PascalCase | MedicationCard, UserModel |
| **Functions** | camelCase (verbNoun) | fetchUserData(), calculateDosage() |
| **Database Tables** | snake\_case (plural) | user\_profiles, medication\_logs |
| **AI/ML Models** | Descriptive lower\_case | drug\_interaction\_model.pkl |

Avoid abbreviations unless they are universally understood (e.g., API, URL, ID).

## **16.5 Coding Style Standards**

### **a. General**

* Write **self-explanatory code**; minimize inline comments by using **clear names and modular logic**.
* Maintain a **maximum function length of 40 lines** where possible.
* Keep **each file focused on one purpose** (Single Responsibility Principle).
* Always include **error handling and null checks**.

### **b. Language-Specific**

#### **JavaScript / TypeScript (Frontend & API)**

* Follow **ESLint + Prettier** style rules.
* Prefer **TypeScript** for strong typing.
* Use **async/await** over callbacks or promises chaining.
* Validate all API inputs using schemas (e.g., **Yup**, **Zod**, or **Joi**).

#### **Python (AI / Backend)**

* Follow **PEP 8** style conventions.
* Use **black** or **flake8** for linting.
* Document functions with **docstrings**:
* def predict\_drug\_interaction(drug\_a, drug\_b):
* """
* Predicts potential drug interaction risks between two medications.
* Returns interaction level (low, moderate, high).
* """
* Prefer **f-strings** for formatting and **type hints** for clarity:
* def get\_user\_data(user\_id: str) -> dict:
* ...

## **16.6 Code Review and Collaboration Process**

| **Stage** | **Action** | **Responsible** |
| --- | --- | --- |
| Code Commit | Follow commit message format and run pre-commit tests | Developer |
| Pull Request | Submit PR with description and screenshots (if UI) | Developer |
| Code Review | Peer or lead reviews for logic, style, and test coverage | Reviewer |
| Merge | Only after approval + all CI tests passed | Maintainer |
| Deployment | Automated via CI/CD pipeline | DevOps |

### **Commit Message Format**

<type>(<scope>): <short summary>

[optional body]

[optional references or issue ID]

**Examples:**

* feat(auth): add Google OAuth login
* fix(api): resolve null pointer in medication endpoint
* chore(ui): update color scheme for dark mode

## **16.7 Testing and Quality Assurance**

All code must include **automated test coverage**:

* **Unit tests:** Minimum 80% coverage for core logic.
* **Integration tests:** Validate API endpoints, DB, and UI flow.
* **End-to-end tests:** Simulate full user actions using Cypress or Playwright.
* **AI modules:** Include validation datasets and performance metrics (accuracy, precision, recall).

CI/CD pipelines will automatically reject builds with failing tests or coverage < threshold.

## **16.8 Documentation and Comments**

* Each module, function, and class should include **concise docstrings** or **JSDoc** comments.
* Use **README.md** files in each module directory to describe functionality and usage.
* Maintain up-to-date **API documentation** using **Swagger (OpenAPI)** for backend endpoints.
* Keep **inline comments minimal** and only where logic is non-trivial.

## **16.9 Security and Privacy Considerations**

* Never commit credentials, API keys, or tokens to the repository.
* Use **environment variables** and .env files (excluded via .gitignore).
* Follow **OWASP guidelines** for input validation, data encryption, and session management.
* Sanitize all user inputs to prevent SQL injection or XSS attacks.
* For AI modules, **anonymize sensitive health data** before model training or analytics.

## **16.10 Performance and Optimization**

* Use **lazy loading** for UI components and data-heavy modules.
* Implement **caching** (Redis, Cloud Cache) for repeated backend queries.
* Optimize database queries and indexes.
* Avoid large dependencies; prefer **native or lightweight libraries.**
* Monitor API latency and optimize with async I/O (e.g., FastAPI or Node async routes).

## **16.11 Version Control and Branching Strategy**

MediMate follows **Gitflow** workflow:

| **Branch** | **Purpose** |
| --- | --- |
| main | Production-ready code |
| develop | Ongoing stable development |
| feature/\* | New feature implementations |
| fix/\* | Hotfix or bug patch |
| release/\* | Pre-production staging |

Always create branches from develop and submit pull requests back to it.

## **16.12 Code Review Checklist**

Before submitting any PR, verify:

* ✅ Code passes lint and unit tests.
* ✅ Function and variable names are meaningful.
* ✅ No hard-coded credentials or endpoints.
* ✅ Error handling and logging included.
* ✅ UI follows design guidelines (for frontend).
* ✅ Documentation updated if feature or API changes.

## **16.13 Continuous Improvement**

Coding standards will evolve with technology updates and team feedback.  
Each sprint’s retrospective should evaluate:

* Code readability
* Performance metrics
* Security compliance
* Maintainability score

All adjustments are documented in the **“Engineering Guidelines” wiki** for consistency across teams.

## **16.14 Tools and Enforcement**

| **Tool** | **Purpose** |
| --- | --- |
| ESLint / Prettier | JS/TS code linting and formatting |
| Black / Flake8 | Python linting |
| SonarQube | Code quality & vulnerability scanning |
| Husky + Pre-commit Hooks | Enforce lint/tests before commits |
| GitHub Actions | Automated CI/CD testing and review gates |
| Swagger / Postman | API documentation and testing |

## **16.15 Summary**

By adhering to these coding standards, MediMate ensures:

* High-quality, maintainable, and secure codebases.
* Consistent developer experience across teams and technologies.
* Streamlined collaboration, faster onboarding, and fewer production bugs.

These standards form the **technical foundation** for all future MediMate releases, ensuring reliability and scalability as the platform evolves.

# **Section 17.0 – Environment Setup**

## **17.1 Overview**

This section provides detailed **installation, configuration, and environment setup guidelines** required to build, run, and maintain the **MediMate system** (Mobile, Web, Backend, and AI/Analytics modules).  
It ensures all developers, testers, and DevOps engineers have a **consistent local environment** aligned with the production setup.

MediMate’s system is **multi-platform**, integrating:

* **Mobile (React Native)**
* **Web (React + TypeScript)**
* **Backend (Node.js / FastAPI)**
* **AI/ML Services (Python)**
* **Cloud & DevOps tools (Docker, Kubernetes, GCP)**

## **17.2 Objectives**

* Provide a **step-by-step setup guide** for all modules of MediMate.
* Define **environment variables and configuration rules** for development, staging, and production.
* Ensure **consistency and reproducibility** across all developer machines.
* Enable smooth onboarding for new developers and collaborators.

## **17.3 Prerequisites**

Before beginning setup, ensure the following dependencies are installed:

| **Tool/Dependency** | **Recommended Version** | **Purpose** |
| --- | --- | --- |
| Node.js | ≥ 18.x | Frontend and Backend runtime |
| npm / yarn | Latest | Package management |
| Python | ≥ 3.10 | AI/ML and backend services |
| Git | ≥ 2.40 | Version control |
| Docker & Docker Compose | Latest | Containerization |
| PostgreSQL / MongoDB | Latest | Database engine |
| Android Studio / Xcode | Latest | Mobile app build & testing |
| VS Code / JetBrains IDE | Latest | Development environment |
| Google Cloud SDK | Latest | Cloud service interaction |
| Postman / Swagger | Latest | API testing and documentation |

## **17.4 Repository Structure**

Each major MediMate component has a dedicated repository:

/medimate-root

├── medimate-web/ → Web client (React + TS)

├── medimate-mobile/ → Mobile app (React Native)

├── medimate-api/ → Backend APIs (Node.js / FastAPI)

├── medimate-ai/ → AI & analytics models (Python)

├── medimate-infra/ → Docker, CI/CD, and Kubernetes configs

└── docs/ → Documentation and design references

## **17.5 Local Development Setup**

### **Step 1: Clone the Repository**

git clone https://github.com/<organization>/medimate-root.git

cd medimate-root

### **Step 2: Install Dependencies**

Each subproject must install its respective dependencies.

**Frontend (Web):**

cd medimate-web

npm install

**Mobile:**

cd medimate-mobile

npm install

npx pod-install # For iOS only

**Backend:**

cd medimate-api

npm install # For Node.js

# or

pip install -r requirements.txt # For Python FastAPI variant

**AI/ML Module:**

cd medimate-ai

pip install -r requirements.txt

## **17.6 Environment Configuration**

MediMate uses .env files to manage sensitive variables.  
Each environment (development, staging, production) has its own .env file stored **outside version control**.

### **Example: .env (Backend)**

# Server Config

PORT=5000

NODE\_ENV=development

# Database Config

DB\_HOST=localhost

DB\_PORT=5432

DB\_USER=medimate\_user

DB\_PASSWORD=secure\_password

DB\_NAME=medimate\_db

# API Keys

JWT\_SECRET=your\_jwt\_secret

GOOGLE\_API\_KEY=your\_google\_key

# Cloud & Storage

GCP\_BUCKET=medimate-data

CLOUD\_REGION=europe-west1

# Logging

LOG\_LEVEL=debug

### **Example: .env (Frontend / Mobile)**

API\_BASE\_URL=https://dev.api.medimate.com

APP\_ENV=development

ENABLE\_ANALYTICS=false

SENTRY\_DSN=

### **Example: .env (AI Module)**

MODEL\_PATH=/models/drug\_interaction\_model.pkl

DATASET\_PATH=/datasets/training\_data.csv

AI\_API\_KEY=your\_ai\_key

ENABLE\_LOGGING=True

⚠️ **Important:**  
.env files **must never be committed** to Git.  
Add them to .gitignore and share via **secure channels** (e.g., Vault, Google Secret Manager).

## **17.7 Database Setup**

### **Option 1: Local PostgreSQL**

docker run --name medimate-db -e POSTGRES\_PASSWORD=secure\_password -p 5432:5432 -d postgres

Then connect using credentials defined in .env.

### **Option 2: MongoDB (for NoSQL modules)**

docker run --name medimate-mongo -p 27017:27017 -d mongo

### **Option 3: Cloud Database**

For production, databases are hosted on **Google Cloud SQL** or **Atlas MongoDB**, accessed securely via service accounts.

## **17.8 Running the Project**

### **Start Backend API (Node.js)**

cd medimate-api

npm run dev

### **Start Backend (Python FastAPI)**

uvicorn main:app --reload

### **Run Web App**

cd medimate-web

npm start

### **Run Mobile App**

cd medimate-mobile

npm start

Then select the device:

* Press **“a”** to open Android emulator
* Press **“i”** to open iOS simulator

### **Run AI/ML Module**

cd medimate-ai

python app.py

### **Run All via Docker**

docker-compose up --build

## **17.9 Folder Permissions and Access Control**

* Application logs, cache, and dataset folders require **read/write access**.
* Avoid running applications as root.
* Use **least privilege principles** for environment credentials.
* Developers requiring elevated access (e.g., AI model uploads) must request temporary tokens via the **DevOps team**.

## **17.10 Environment Profiles**

| **Profile** | **Purpose** | **Key Differences** |
| --- | --- | --- |
| **Development** | Local setup for feature coding | Debug mode, local DB, verbose logging |
| **Staging** | Testing environment (pre-production) | Connected to cloud DB, staging APIs |
| **Production** | Live deployment | Secure APIs, minified code, monitoring enabled |

Each profile corresponds to .env.dev, .env.staging, .env.prod.

## **17.11 Docker Setup (Recommended for Consistency)**

### **Docker Compose Example**

version: '3.8'

services:

web:

build: ./medimate-web

ports:

- "3000:3000"

env\_file:

- ./medimate-web/.env

api:

build: ./medimate-api

ports:

- "5000:5000"

env\_file:

- ./medimate-api/.env

ai:

build: ./medimate-ai

ports:

- "8000:8000"

env\_file:

- ./medimate-ai/.env

db:

image: postgres

environment:

POSTGRES\_PASSWORD: secure\_password

ports:

- "5432:5432"

Run everything:

docker-compose up --build

## **17.12 Troubleshooting Guide**

| **Issue** | **Possible Cause** | **Solution** |
| --- | --- | --- |
| Port already in use | Previous process not closed | Kill process or change port in .env |
| Database connection error | Incorrect credentials or container not running | Verify .env and restart Docker |
| API not responding | CORS or base URL mismatch | Update API\_BASE\_URL in frontend .env |
| Build fails | Missing dependency or incompatible Node/Python version | Run npm ci / recreate virtual env |
| Emulator not starting | Android SDK path missing | Reinstall Android Studio or set ANDROID\_HOME |

## **17.13 Security and Secrets Management**

* Store API keys and credentials in **Google Secret Manager** or **AWS Secrets Manager**.
* Encrypt sensitive .env files before sharing (e.g., using **gpg**).
* Rotate credentials quarterly or after staff changes.
* Never hardcode credentials inside the codebase.
* Follow least-privilege access for all cloud service accounts.

## **17.14 Environment Verification Checklist**

Before running or deploying, ensure:

* ✅ All dependencies installed.
* ✅ .env configured correctly for current environment.
* ✅ Database containers running.
* ✅ API reachable locally.
* ✅ Frontend connects to correct API URL.
* ✅ Linting and tests pass successfully.

## **17.15 Summary**

The environment setup ensures that MediMate developers operate within a **controlled, predictable, and secure environment**, mirroring the cloud infrastructure.  
By following these setup standards, teams can:

* Reduce configuration conflicts,
* Accelerate onboarding,
* Improve debugging and testing efficiency,
* Maintain consistent deployments across development, staging, and production.

## **Section 18.0 – Developer Onboarding and Git Workflow**

**Purpose:**  
This section outlines the procedures, tools, and standards for new developer onboarding, version control, and collaborative workflows within the MediMate development environment. It ensures consistency, efficiency, and traceability throughout all development phases.

### **18.1 Overview**

The **Developer Onboarding and Git Workflow** establishes a unified approach to project collaboration. It defines how new team members join the project, set up their environments, manage source code through Git, and contribute efficiently while maintaining code integrity and project stability.

This framework promotes:

* Consistent branching and merging practices
* Clear commit and review policies
* Secure and traceable collaboration using Git and GitHub
* Efficient onboarding with environment and access guidelines

### **18.2 Developer Onboarding Process**

#### **18.2.1 Pre-requisites**

Before onboarding, every developer should have:

* Basic understanding of **Git**, **GitHub**, and **CLI (Command Line Interface)**
* Familiarity with the **MediMate tech stack** (React Native, Node.js, Python, Firebase, MongoDB, etc.)
* Access to required tools and environments (listed below)

#### **18.2.2 Required Tools and Accounts**

Developers must install and configure the following:

* **Git** – Version control system (<https://git-scm.com/>)
* **GitHub Account** – Access to MediMate repository
* **Node.js & npm** – For frontend and API dependencies
* **Python (3.10+)** – For AI and backend microservices
* **Docker** – For containerization and local testing
* **VS Code** (recommended) or preferred IDE
* **Postman** – For API testing
* **Firebase CLI / MongoDB Compass** – For database interaction
* **Slack or Teams** – For internal communication

#### **18.2.3 Access and Permissions**

* Repository access is granted via **GitHub Organization (MediMate Org)**.
* Each developer is assigned to one or more repositories (e.g., medimate-mobile, medimate-api, medimate-ai).
* Permissions:
  + **Admin:** Core maintainers
  + **Write:** Feature developers
  + **Read:** Reviewers or documentation contributors

#### **18.2.4 Environment Setup**

Each new developer must:

1. Clone the base repository using
2. git clone https://github.com/medimate-org/medimate.git
3. Create and configure the .env file using templates provided in /env-samples/.
4. Run local setup commands:
5. npm install
6. npm start

or, for backend:

python -m venv venv

source venv/bin/activate

pip install -r requirements.txt

1. Validate setup by running test suites:
2. npm run test

or

pytest

#### **18.2.5 Orientation and Documentation**

New developers should review:

* README.md and /docs/ folder for module descriptions
* The **Architecture Overview** (Section 5.0)
* The **Coding Standards and Best Practices** (Section 16.0)
* The **Environment Setup Guide** (Section 17.0)

### **18.3 Git Repository Structure**

The MediMate codebase follows a **modular monorepo** structure where independent services (mobile, API, AI, etc.) reside under a unified organization.

Example layout:

medimate/

│

├── mobile/ # React Native frontend

├── api/ # Node.js/Express backend

├── ai/ # Python-based AI microservices

├── docs/ # Developer documentation

├── scripts/ # Utility scripts and CI/CD configs

└── .github/ # Workflows, issue templates

### **18.4 Git Branching Strategy**

#### **18.4.1 Branch Types**

| **Branch Type** | **Purpose** | **Example** |
| --- | --- | --- |
| main | Production-ready, stable code | main |
| develop | Active development integration | develop |
| feature/\* | Individual feature or module work | feature/mood-tracker |
| bugfix/\* | Quick fixes on development branch | bugfix/login-error |
| hotfix/\* | Urgent production-level fixes | hotfix/api-auth |
| release/\* | Staging for release versions | release/v1.2.0 |

#### **18.4.2 Workflow Diagram**

main ← release ← develop ← feature/bugfix

↑

hotfix

### **18.5 Commit and Pull Request Guidelines**

#### **18.5.1 Commit Message Format**

All commits must follow the **Conventional Commit Standard** for clarity and automation:

<type>(scope): <short description>

**Examples:**

feat(auth): add JWT token-based authentication

fix(api): resolve null pointer in medication controller

docs(readme): update setup instructions

chore(ci): update GitHub workflow triggers

**Commit types:** feat, fix, docs, style, refactor, test, chore.

#### **18.5.2 Pull Requests (PRs)**

* Always branch off develop unless doing a hotfix from main.
* Keep PRs **small, descriptive, and atomic**.
* Include:
  + Summary of changes
  + Related issue ID (Closes #123)
  + Test results or screenshots
* At least **one reviewer approval** is required before merging.
* CI/CD pipeline will automatically run tests and lint checks on all PRs.

### **18.6 Code Review and Merging Protocol**

1. **Open PR → Assign Reviewer(s)**
2. **Automated Checks** (CI/CD, linting, tests)
3. **Manual Review** – Focus on functionality, readability, and security
4. **Approval & Merge:**
   * Merge using **Squash and Merge** for cleaner history.
5. **Delete Feature Branch** after successful merge.

**Note:** Only maintainers can merge into main. Developers merge into develop after approval.

### **18.7 Version Control and Tagging**

* Versions follow **Semantic Versioning (SemVer)**:  
  MAJOR.MINOR.PATCH → e.g., v1.4.2
* Tags are applied when a release branch is merged into main.
* git tag -a v1.4.2 -m "Release v1.4.2"
* git push origin v1.4.2

### **18.8 Collaboration and Issue Tracking**

#### **18.8.1 Project Boards**

* Managed in **GitHub Projects** or **Jira**
* Columns: Backlog → In Progress → In Review → Done

#### **18.8.2 Issues and Labels**

Standard labels:

* bug, feature, enhancement, documentation, help wanted, urgent  
  Each issue must include:
* Clear title and description
* Acceptance criteria
* Assignee and milestone

#### **18.8.3 Communication**

* **Slack/Teams:** for daily updates and quick discussions
* **Weekly Sync Meetings:** project updates and blockers
* **Code Review Channels:** for peer discussions and improvement

### **18.9 Continuous Integration Alignment**

Git workflow integrates seamlessly with **CI/CD pipelines** (see Section 15.0).

* Each PR triggers build and test runs via **GitHub Actions**.
* Merges to main auto-deploy to the **staging environment**.
* Version tags trigger production deployment.

### **18.10 Best Practices Summary**

✅ Keep branches small and focused  
✅ Use descriptive commit messages  
✅ Pull latest from develop before starting new work  
✅ Never push directly to main  
✅ Run local tests before pushing  
✅ Review and document major changes

### **18.11 Example Developer Flow**

# Step 1: Create a feature branch

git checkout develop

git pull origin develop

git checkout -b feature/medication-reminder

# Step 2: Develop and commit

git add .

git commit -m "feat(reminder): add medication reminder scheduling"

# Step 3: Push and open PR

git push origin feature/medication-reminder

# Step 4: Create Pull Request → Review → Merge into develop

### **18.12 Onboarding Checklist**

| **Task** | **Status** |
| --- | --- |
| GitHub account created | ☐ |
| Repo cloned and environment setup | ☐ |
| .env configured | ☐ |
| Dependencies installed | ☐ |
| Access to Slack/Teams confirmed | ☐ |
| Reviewed Docs: Sections 5, 16, 17 | ☐ |
| First PR submitted | ☐ |

### **18.13 Summary**

The Developer Onboarding and Git Workflow ensures structured collaboration, maintainable source control, and a smooth developer experience. By following this process, MediMate developers uphold high standards of productivity, version traceability, and release stability across all environments.

## **Section 19.0 – Troubleshooting and Maintenance**

**Purpose:**  
This section provides a structured guide for identifying, diagnosing, and resolving technical issues within the MediMate platform. It also defines the standard maintenance routines and preventive measures that ensure long-term system stability, performance, and reliability.

### **19.1 Overview**

The **Troubleshooting and Maintenance** framework ensures the MediMate system remains stable, secure, and high-performing. It enables developers and support engineers to:

* Quickly isolate and resolve system or application issues
* Maintain uptime through regular system health checks
* Apply updates, patches, and optimizations systematically
* Document and track incidents for future prevention

This process applies to all MediMate modules — including **Mobile App**, **Web App**, **Backend APIs**, and **AI Microservices**.

### **19.2 Common Issue Categories**

MediMate issues are grouped into four main categories:

| **Category** | **Description** | **Examples** |
| --- | --- | --- |
| **Application Errors** | Functional or logic-related bugs in frontend/backend | App crash on login, reminder not saving |
| **Integration/Network Issues** | API communication failures, timeouts, or data sync problems | Firebase sync delay, failed AI service response |
| **Database/Storage Issues** | Problems with data retrieval, corruption, or connectivity | MongoDB connection timeout, inconsistent records |
| **Environment/Dependency Issues** | Conflicts in versions, configurations, or external libraries | Node/Python version mismatch, missing environment variables |

### **19.3 General Troubleshooting Workflow**

A systematic debugging and resolution workflow ensures consistency and traceability:

1. **Identify the Problem**
   * Observe error messages or malfunction symptoms.
   * Review recent code merges, deployments, or environment changes.
2. **Check Logs and Monitoring Tools**
   * Access logs via **Firebase Console**, **Docker logs**, or **Cloud Logging**.
   * Use monitoring dashboards (Grafana, CloudWatch, or Firebase Crashlytics).
3. **Reproduce the Issue**
   * Attempt to replicate the issue locally or in a staging environment.
   * Document exact conditions leading to the failure.
4. **Isolate the Cause**
   * Test related components independently (API, DB, AI).
   * Roll back recent code or configuration changes if necessary.
5. **Fix and Validate**
   * Apply a fix in a local branch.
   * Run automated test suites and manual regression tests.
6. **Document and Deploy**
   * Log the issue and solution in /docs/issues.md or GitHub Issues.
   * Merge fix following Git workflow (see Section 18.0).
7. **Monitor Post-Fix Behavior**
   * Verify that issue recurrence or new regressions do not appear.
   * Track metrics (API uptime, app crash rate) for 24–48 hours post-deploy.

### **19.4 Logging and Error Monitoring**

#### **19.4.1 Logging Standards**

All MediMate modules implement structured logging for visibility:

// Example (Node.js)

logger.info("Medication added successfully", { userId, medicationId });

logger.error("API failure: ", { endpoint: "/v1/drugs", error });

**Log Levels:**

* info – General events (successful actions, operations)
* warn – Non-critical issues or potential risks
* error – Failures affecting user functionality
* critical – System-wide crashes or security events

#### **19.4.2 Monitoring Tools**

| **Tool** | **Purpose** | **Module** |
| --- | --- | --- |
| **Firebase Crashlytics** | Mobile crash reports | Mobile App |
| **Sentry** | Error tracking and alerting | API & Frontend |
| **Google Cloud Logging / Grafana** | Real-time performance and system metrics | Backend & AI Services |
| **GitHub Actions Logs** | Build and deployment diagnostics | CI/CD Pipeline |

### **19.5 Known Issues and Fix References**

| **ID** | **Issue Description** | **Root Cause** | **Resolution** | **Status** |
| --- | --- | --- | --- | --- |
| #001 | Notification delay when offline | Local queue not syncing post-reconnect | Implement hybrid notification cache sync | ✅ Fixed |
| #002 | AI drug-interaction check timing out | API call exceeding threshold | Optimize service response and add async queue | ✅ Fixed |
| #003 | Docker build failure on Windows | Permission conflicts | Adjust Dockerfile permissions and path mapping | ✅ Fixed |
| #004 | Intermittent login token error | JWT expiration misconfigured | Extend refresh token duration | 🟡 In Testing |

**Note:** Developers should update /docs/known\_issues.md after resolving or discovering new issues.

### **19.6 Debugging by Module**

#### **19.6.1 Frontend (React Native / Web)**

**Common Issues:**

* Blank screens or crash after launch
* Component rendering errors
* API response undefined

**Debugging Steps:**

1. Run app in debug mode:
2. npm start -- --reset-cache
3. Use **React Developer Tools** or **Flipper** for inspection.
4. Check for API response errors via **Network tab** or **Postman**.
5. Validate .env and endpoint configurations.
6. Reinstall dependencies if version conflicts occur:
7. npm ci

#### **19.6.2 Backend (Node.js / Express)**

**Common Issues:**

* API returning 500 or 404 errors
* Database connection failures
* Invalid token or authorization error

**Debugging Steps:**

1. View logs:
2. docker logs medimate-api
3. Test endpoints locally using Postman.
4. Validate environment variables (especially MONGO\_URI, JWT\_SECRET).
5. Run linting and unit tests:
6. npm run lint
7. npm test
8. If persistent, restart containers or rebuild images:
9. docker-compose up --build

#### **19.6.3 AI Microservices (Python / Flask / FastAPI)**

**Common Issues:**

* Model loading failure
* Slow inference times
* API unresponsive after timeout

**Debugging Steps:**

1. Check logs:
2. docker logs medimate-ai
3. Validate model paths and configuration files.
4. Test locally using curl or Postman.
5. Monitor GPU/CPU usage to detect overload.
6. Restart services using Docker Compose or Kubernetes pod restart.

### **19.7 Maintenance Routines**

#### **19.7.1 System Health Checks**

Perform weekly or automated checks on:

* **API uptime** (≥ 99%)
* **Database response times**
* **Crash analytics reports**
* **Server CPU and memory usage**

Use tools such as **Grafana dashboards**, **Firebase Analytics**, and **GCP Monitoring**.

#### **19.7.2 Database Maintenance**

* Backup MongoDB and Firebase data weekly.
* Archive old records to optimize query performance.
* Run integrity checks monthly to detect corruption or orphaned data.
* Use scheduled scripts (/scripts/db\_backup.js) to automate.

#### **19.7.3 Security and Dependency Updates**

* Update Node and Python dependencies monthly.
* Run automated vulnerability checks using:
* npm audit fix
* pip-audit
* Rotate environment secrets and API keys every quarter.
* Review access control lists and permission scopes.

#### **19.7.4 Infrastructure Maintenance**

* Restart containers weekly or after major updates.
* Clear cached Docker images to reclaim space.
* Validate SSL certificates and domain DNS configuration.
* Monitor billing and storage thresholds on GCP/Firebase.

### **19.8 Preventive Maintenance and Alerts**

* **Automated Alerts:** Set up real-time notifications via Slack/Email for downtime or anomalies.
* **Proactive Testing:** Regularly execute integration and regression test suites.
* **Incident Reports:** Maintain logs of system failures and corrective actions.
* **Scheduled Audits:** Conduct monthly reviews of logs, error rates, and system performance.

### **19.9 Escalation and Support Procedure**

1. **Immediate Fix (Severity 1):**
   * Trigger rollback or failover deployment.
   * Notify DevOps and QA leads.
2. **Moderate Issues (Severity 2):**
   * Open a GitHub issue tagged urgent.
   * Assign to the responsible developer.
3. **Minor Issues (Severity 3):**
   * Log for next sprint or patch release.

All escalations should be documented in the **Incident Register** under /docs/incidents.md.

### **19.10 Backup and Recovery Plan**

* **Automated Daily Backups** for all databases and logs.
* **Cloud Storage Replication** on multiple regions for redundancy.
* **Disaster Recovery Scripts** to restore environments within 30 minutes.
* **Periodic Restoration Tests** to verify backup integrity.

### **19.11 Summary**

The troubleshooting and maintenance framework ensures the MediMate system remains resilient and dependable. By following structured debugging steps, maintaining proactive health checks, and documenting all incidents, the team ensures minimal downtime and high operational reliability across all environments — from development to production.

## **Section 20.0 – Testing Strategy**

**Purpose:**  
This section defines the **testing framework, methodologies, and coverage expectations** applied across all MediMate components. It ensures that every release maintains reliability, usability, and security standards through automated and manual testing layers.

### **20.1 Overview**

The **Testing Strategy** outlines how MediMate validates application functionality, stability, and performance before each release.  
Testing spans across all modules — **Mobile App**, **Web App**, **Backend APIs**, and **AI Microservices** — and integrates with the CI/CD pipeline (see Section 15.0).

The key goals are to:

* Detect defects early and reduce regression risk.
* Maintain high test coverage across all layers.
* Automate repetitive validation tasks.
* Ensure each module meets defined acceptance and quality criteria.

### **20.2 Testing Objectives**

1. Guarantee application stability across supported devices and environments.
2. Validate data integrity and synchronization across APIs, database, and cloud services.
3. Ensure security, compliance, and privacy (aligning with Section 7.0).
4. Validate AI prediction accuracy and ensure non-biased outcomes.
5. Support continuous integration with automated test suites.

### **20.3 Test Levels**

MediMate employs **five structured test levels** to ensure complete quality assurance coverage:

| **Level** | **Scope** | **Primary Tools/Frameworks** | **Responsible Team** |
| --- | --- | --- | --- |
| **Unit Testing** | Individual functions or modules | Jest (JS), Pytest (Python), Mocha + Chai | Developers |
| **Integration Testing** | Combined components (API ↔ DB, Mobile ↔ Backend) | Supertest, Postman Collections | Developers |
| **System Testing** | Full application validation | Cypress, Appium, Selenium | QA Engineers |
| **User Acceptance Testing (UAT)** | End-user scenarios and feature validation | Manual + Automated scripts | Product Team / QA |
| **Regression Testing** | Verify existing features post-change | Automated pipelines (CI/CD) | QA + DevOps |

### **20.4 Test Environments**

Each test phase runs within controlled, reproducible environments:

| **Environment** | **Purpose** | **Hosted On** |
| --- | --- | --- |
| **Local Dev** | Developer sandbox for unit + integration testing | Local machine / Docker |
| **Staging / Pre-Prod** | UAT & System testing; mirrors production | Cloud (GCP / Firebase / Kubernetes) |
| **Production** | Live user environment | Cloud (GCP / Firebase) |

Environment variables and configurations are isolated via .env.\* files to prevent credential leaks.

### **20.5 Testing Frameworks and Tools**

| **Category** | **Technology / Tool** | **Description** |
| --- | --- | --- |
| **Frontend Testing** | Jest, React Testing Library, Cypress | Component/unit tests and UI automation |
| **Mobile App Testing** | Appium, Detox | Cross-platform (Android/iOS) UI tests |
| **API Testing** | Supertest, Postman, Newman | REST endpoint validation and automated API collections |
| **Backend Testing** | Mocha + Chai (JS), Pytest (Python) | Logic validation and microservice testing |
| **AI Model Testing** | Pytest, Jupyter Validation Scripts | Model accuracy, drift detection, bias analysis |
| **Performance Testing** | Locust, K6 | Load and stress simulation |
| **Security Testing** | OWASP ZAP, npm audit, Bandit | Vulnerability scanning and dependency checks |
| **CI/CD Integration** | GitHub Actions, Jest Coverage Reporter | Automated testing in pipeline runs |

### **20.6 Test Coverage Requirements**

| **Layer** | **Minimum Coverage** | **Notes** |
| --- | --- | --- |
| **Unit Tests** | ≥ 85 % of core functions | Each commit should increase coverage trend |
| **Integration Tests** | ≥ 70 % of API routes and service interactions | Must run before any merge to develop |
| **End-to-End (UI) Tests** | ≥ 60 % of critical flows | Login, Reminders, Medication Logs, AI Checks |
| **Security Tests** | 100 % of critical endpoints scanned | No deployment if critical risk found |
| **AI Validation Tests** | ≥ 90 % accuracy consistency post-training | Flag deviations > 5 % for review |

Coverage reports are automatically generated and stored under /reports/coverage.

### **20.7 Test Data Management**

* **Synthetic Data:** Used in non-production environments to protect real user information.
* **Mock Data Generators:** Tools like Faker.js and Factory Boy generate consistent test data.
* **Seed Scripts:** Create baseline database states (/scripts/seed\_data.js).
* **Data Anonymization:** Production data samples used for performance tests are anonymized to comply with privacy laws.

### **20.8 Automated Testing Integration (CI/CD)**

All tests run automatically during the CI/CD pipeline (Section 15.0):

1. **Pull Request Opened** → Trigger **Unit & Lint Tests**
2. **Develop Merge** → Run **Integration Tests**
3. **Release Branch Created** → Execute **System & Regression Tests**
4. **Production Tag** → Perform **Smoke & Security Tests**

# Example GitHub Actions snippet

jobs:

test:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- run: npm install

- run: npm test -- --coverage

- run: pytest --maxfail=1 --disable-warnings

### **20.9 Manual Testing Protocols**

Manual tests complement automation for exploratory and UX-driven validation:

* Cross-browser and cross-device checks (Chrome, Safari, Android, iOS).
* Accessibility testing using **Lighthouse** and **Screen Reader tools**.
* Visual consistency checks (UI/UX alignment with design guidelines).
* User flow simulations for critical scenarios (e.g., offline reminder sync, account recovery).

Results are logged in /docs/manual\_test\_cases.xlsx.

### **20.10 Regression and Smoke Testing**

* **Regression Tests:** Run nightly and before every release build to verify that new commits haven’t broken existing features.
* **Smoke Tests:** Lightweight sanity checks post-deployment verifying:
  + App launches successfully
  + APIs return valid responses
  + Database connections active
  + Authentication functional

Automated via GitHub Actions and triggered by deployment success events.

### **20.11 Performance and Load Testing**

To ensure system scalability and responsiveness:

* **Load Testing:** Validate performance under typical and peak load using Locust/K6.
* **Stress Testing:** Identify system limits under extreme load.
* **Metrics Monitored:** Response time < 2 s, Error rate < 1 %, Throughput > 200 req/s.
* **Frequency:** Monthly or before major releases.

Reports stored in /reports/performance/.

### **20.12 AI Model Testing**

AI components (drug interaction detector, side effect predictor, mood insight engine) follow specialized validation:

1. **Model Accuracy Tests** – Validate prediction outputs against labeled data.
2. **Bias and Fairness Analysis** – Ensure no demographic or clinical bias.
3. **Drift Detection Monitoring** – Continuous model performance tracking in production.
4. **Retraining Validation** – Compare new vs previous model results (accuracy and precision differences ≤ 5 %).

### **20.13 Test Reporting and Documentation**

All test executions automatically produce structured reports:

* **Coverage Reports:** /reports/coverage/index.html
* **Test Results:** /reports/results.xml
* **Performance Metrics:** /reports/performance/summary.json
* **Error Logs:** /logs/tests/

The CI/CD pipeline uploads summaries to GitHub Artifacts or the project dashboard.

### **20.14 Roles and Responsibilities**

| **Role** | **Key Testing Responsibilities** |
| --- | --- |
| **Developers** | Write unit & integration tests, fix failing cases before merge |
| **QA Engineers** | Design test plans, execute system & UAT tests, track bugs |
| **DevOps Team** | Maintain test environments and CI/CD automation |
| **Product Owners** | Validate UAT outcomes and approve release candidates |

### **20.15 Quality Gates**

To maintain high release standards, the CI/CD pipeline enforces **quality gates**:

* Builds fail if unit test coverage < 85 %.
* Security scan must return no critical vulnerabilities.
* All automated tests must pass before merge to main.
* Manual UAT approval required for release branch merge.

### **20.16 Continuous Quality Improvement**

* **Retrospective Testing Reviews:** Conducted after each release to analyze defects and improve coverage.
* **Test Case Repository:** Centralized and version-controlled in /tests/docs/.
* **Automation Backlog:** QA team adds new automatable cases each sprint.
* **AI-Driven Testing (Planned):** Future integration for predictive defect detection and smart test selection.

### **20.17 Summary**

The MediMate **Testing Strategy** ensures that every module, service, and user workflow is validated under rigorous automated and manual testing layers.  
By maintaining strict coverage thresholds, continuous integration with CI/CD, and multi-environment validation, MediMate achieves a robust, secure, and high-quality delivery pipeline that guarantees user trust and system reliability.

## **Section 21.0 – Glossary**

**Purpose:**  
This section defines all technical terms, acronyms, and key concepts used across the **MediMate Development Documentation**. It serves as a unified reference for developers, testers, designers, analysts, and product managers to ensure consistent understanding of project terminology.

Each term includes a **clear definition**, **practical context**, and — where relevant — an example or usage within the MediMate ecosystem.

### **21.1 General Software and Development Terms**

| **Term / Acronym** | **Definition / Description** | **Example / Context in MediMate** |
| --- | --- | --- |
| **Application (App)** | A software program designed for end users, either web-based or mobile. | MediMate Mobile App (React Native) and MediMate Web Portal (React.js). |
| **Architecture** | The structural design of software components, their relationships, and interactions. | MediMate follows a microservices + modular architecture (Section 5.0). |
| **API (Application Programming Interface)** | A defined communication method allowing systems to exchange data. | The backend exposes REST APIs for user, medication, and AI endpoints. |
| **Backend** | The server-side logic, APIs, and database layer. | Node.js Express APIs handle all backend operations. |
| **Frontend** | The user interface and client-side logic visible to users. | The React Native UI for the mobile application. |
| **Bug / Defect** | An error causing incorrect system behavior. | “Medication reminder not triggering offline.” |
| **Build** | A compiled or packaged version of code ready for deployment. | medimate-v1.5.0.apk generated after CI/CD pipeline build. |
| **Cache** | Temporary storage that speeds up data retrieval. | Cached reminders are stored locally when offline. |
| **Commit** | A recorded change in a Git repository. | feat(ui): add offline mode toggle |
| **Containerization** | Packaging software and dependencies into isolated environments. | MediMate’s API and AI services run inside Docker containers. |
| **Dependency** | External library or module required by a project. | React Native, Axios, Firebase SDK. |
| **Deployment** | Moving software from development to a live or staging environment. | GitHub Actions auto-deploy to GCP. |
| **Dev Environment** | Local or remote environment used for active development. | Local Docker containers or Firebase emulator. |
| **Feature Branch** | A temporary Git branch for developing specific features. | feature/mood-checkin-analytics |
| **Framework** | A reusable software foundation for building applications. | React Native (Frontend), Node.js (Backend), FastAPI (AI services). |
| **Hotfix** | An urgent fix deployed directly to production. | Fixing a medication reminder crash without waiting for the next release. |
| **Integration** | Connecting separate systems to work as one. | Mobile app ↔ API ↔ Firebase integration. |
| **Library** | A reusable set of functions or components. | Moment.js for time formatting, Mongoose for MongoDB models. |
| **Middleware** | Logic that processes requests/responses between client and server. | Authentication middleware verifying JWT tokens. |
| **Module** | A self-contained functional unit in the codebase. | NotificationModule, UserProfileModule, DrugInteractionService. |
| **Monorepo** | A single repository containing multiple related projects. | MediMate stores API, mobile, and AI services in one GitHub monorepo. |
| **Pull Request (PR)** | Proposal to merge code changes into a branch after review. | Developer submits PR to merge feature branch into develop. |
| **Repository (Repo)** | A storage location for code, managed by Git. | https://github.com/medimate-org/medimate-api |
| **Rollback** | Reverting to a stable version after a failed deployment. | Restoring build v1.3.0 after CI/CD pipeline failure. |
| **Scalability** | The system’s ability to handle increased workload or users efficiently. | Auto-scaling of AI services using Kubernetes. |
| **Version Control** | System tracking changes to code over time. | Managed via Git and GitHub. |
| **Workflow** | A defined process or sequence of development steps. | The Git branching and review process outlined in Section 18.0. |

### **21.2 Testing and Quality Assurance Terms**

| **Term / Acronym** | **Definition / Description** | **Example / Context in MediMate** |
| --- | --- | --- |
| **Automated Testing** | Tests executed automatically by scripts or CI tools. | Jest, Cypress, and Pytest integrated in CI/CD pipeline. |
| **Manual Testing** | Human-led testing process to validate system behavior. | QA manually tests mood-checkin flows on mobile devices. |
| **Unit Test** | Tests a single function or component in isolation. | Testing calculateDosage() function in API. |
| **Integration Test** | Verifies interaction between multiple modules or APIs. | Testing AIService ↔ API ↔ Database. |
| **System Test** | Validates end-to-end system behavior. | Running MediMate app end-to-end via Cypress. |
| **Regression Test** | Ensures new updates don’t break existing functionality. | Run automatically on each develop merge. |
| **Smoke Test** | A quick validation ensuring core functionalities work. | Verifying login, API status, and reminders post-deployment. |
| **UAT (User Acceptance Testing)** | User-focused test to confirm system meets requirements. | Product team verifies that reminder feature works as expected. |
| **Test Case** | A documented scenario with input, expected output, and result. | Test case: “User adds new medication; system saves and displays it.” |
| **Test Coverage** | Measure of how much of the codebase is tested. | Target: ≥ 85% coverage for backend code. |
| **Test Data** | Controlled or mock data used for testing features. | Fake users generated using Faker.js. |
| **Mock Server** | Simulated backend used for frontend development/testing. | Mock API returning fake medication data. |
| **Bug Report** | Documentation of an issue with steps to reproduce. | Logged in GitHub Issues. |

### **21.3 AI, Analytics, and Data Science Terms**

| **Term / Acronym** | **Definition / Description** | **Example / Context in MediMate** |
| --- | --- | --- |
| **AI (Artificial Intelligence)** | Machines performing tasks requiring human intelligence. | Predicting drug interactions or mood patterns. |
| **ML (Machine Learning)** | Algorithms that learn from data to make predictions. | Predicting likelihood of side effects. |
| **Dataset** | A structured collection of data used for model training or analysis. | Medication adherence dataset from anonymized logs. |
| **Training Data** | Data used to teach an AI model. | Mood logs from consenting users. |
| **Testing Data** | Separate dataset used to validate model performance. | Used in accuracy and bias validation. |
| **Model Inference** | Generating predictions from a trained model. | AI service predicting potential drug interaction risk. |
| **Feature Engineering** | Selecting and transforming input variables for models. | Age, dosage, and time used to predict adherence rate. |
| **Bias (AI)** | Systematic deviation in model predictions due to unbalanced training data. | Gender-based disparity in mood prediction accuracy. |
| **Model Drift** | Degradation of model accuracy over time as data patterns evolve. | AI retraining triggered monthly if accuracy < 90%. |
| **Neural Network** | A machine learning model inspired by the human brain’s structure. | Used for advanced drug interaction prediction. |
| **NLP (Natural Language Processing)** | AI’s ability to process and understand text. | Analyzing symptom descriptions entered by users. |
| **Predictive Analytics** | Data-driven prediction of future outcomes. | Forecasting missed medication likelihood. |
| **Data Pipeline** | Automated data flow for processing and analysis. | ETL pipeline syncing data between Firebase and AI service. |

### **21.4 Security and Compliance Terms**

| **Term / Acronym** | **Definition / Description** | **Example / Context in MediMate** |
| --- | --- | --- |
| **Authentication** | Verifying a user’s identity before granting access. | Email/password or OAuth login. |
| **Authorization** | Defining which actions a user can perform. | Only doctors can view analytics dashboard. |
| **Encryption** | Encoding data for secure storage or transmission. | AES-256 encryption used for sensitive medical data. |
| **Decryption** | Converting encrypted data back to readable form. | Occurs on secure endpoints during retrieval. |
| **Hashing** | Irreversible transformation of sensitive data. | Passwords hashed using bcrypt. |
| **JWT (JSON Web Token)** | Token-based mechanism for secure session handling. | Used for mobile app authentication. |
| **OAuth 2.0** | Authorization protocol enabling secure API access. | MediMate may integrate with third-party health APIs. |
| **RBAC (Role-Based Access Control)** | User permissions defined by roles. | Admin, Patient, Doctor roles in MediMate. |
| **TLS (Transport Layer Security)** | Encryption protocol for securing HTTPS communication. | Protects all MediMate API requests. |
| **GDPR / HIPAA Compliance** | Legal standards for protecting medical data. | MediMate anonymizes and encrypts all health data. |
| **Access Token** | Temporary credential authorizing access to protected resources. | Expires after 1 hour for security. |
| **Refresh Token** | Token used to obtain new access tokens. | Automatically renewed without re-login. |
| **Vulnerability Scan** | Automated process for detecting security flaws. | Run using OWASP ZAP and npm audit. |

### **21.5 DevOps, Infrastructure, and Cloud Terms**

| **Term / Acronym** | **Definition / Description** | **Example / Context in MediMate** |
| --- | --- | --- |
| **CI/CD (Continuous Integration / Continuous Deployment)** | Automated process for building, testing, and deploying code. | Managed via GitHub Actions pipelines. |
| **Docker** | Platform for creating lightweight, portable containers. | medimate-api runs inside Docker containers. |
| **Kubernetes (K8s)** | Orchestrates container deployment, scaling, and management. | MediMate AI services auto-scale using Kubernetes. |
| **Image** | Snapshot of an application and its dependencies. | medimate-api:v1.2.0 Docker image. |
| **Container** | Running instance of a Docker image. | Isolated instance of MediMate AI microservice. |
| **Pod** | Smallest deployable unit in Kubernetes, containing containers. | AI predictor pod deployed on GCP cluster. |
| **Load Balancer** | Distributes traffic evenly among multiple servers. | Ensures stable backend response under heavy load. |
| **Auto Scaling** | Adjusting resources dynamically based on usage. | AI pods scale up during high traffic periods. |
| **Monitoring** | Continuous tracking of metrics, logs, and health checks. | Grafana dashboards display system uptime and response times. |
| **Artifact** | Build output stored for deployment. | Test coverage reports or compiled builds. |
| **Infrastructure as Code (IaC)** | Managing infrastructure using code templates. | GCP deployment managed through YAML and Terraform scripts. |

### **21.6 UI/UX and User Interaction Terms**

| **Term / Acronym** | **Definition / Description** | **Example / Context in MediMate** |
| --- | --- | --- |
| **UI (User Interface)** | The visible, interactive layer of an app. | MediMate’s dashboard and mood-tracking pages. |
| **UX (User Experience)** | The user’s overall experience and satisfaction using the app. | Simplicity of mood check-in process improves UX. |
| **Wireframe** | Low-fidelity sketch showing layout structure. | Early design of reminder setup screen. |
| **Prototype** | Interactive model simulating app flow before coding. | Created using Figma for design validation. |
| **Responsive Design** | UI adapts to different screen sizes and devices. | Web and mobile UI scale automatically. |
| **Dark Mode / Light Mode** | UI color schemes optimized for comfort and accessibility. | User can toggle between dark and light themes. |
| **Accessibility (A11y)** | Ensuring usability for users with disabilities. | Screen reader support for visually impaired users. |
| **Onboarding Flow** | Initial setup screens guiding new users. | Steps to configure medication reminders after signup. |
| **Gesture Control** | Mobile UI interactions like swipes, taps, or long-presses. | Swipe left to delete medication. |
| **Modal** | Pop-up dialog window requiring user action. | Confirmation modal for deleting a medication. |

### **21.7 Project and Management Terms**

| **Term / Acronym** | **Definition / Description** | **Example / Context in MediMate** |
| --- | --- | --- |
| **Sprint** | A time-boxed development cycle (1–2 weeks). | New features or bug fixes scheduled per sprint. |
| **Epic** | A large body of work that can be broken into smaller tasks. | “AI Prediction Module” is an epic with multiple sub-features. |
| **Task** | A specific actionable item within a sprint. | “Create drug interaction controller.” |
| **Milestone** | Major project checkpoint or deliverable. | “v1.0 release” or “AI phase beta testing.” |
| **Kanban Board** | Workflow visualization tool used to manage tasks. | Used in GitHub Projects for tracking development. |
| **Backlog** | List of pending or planned tasks/features. | “Export reports to PDF” feature awaiting prioritization. |

### **21.8 Common Abbreviations**

| **Abbreviation** | **Meaning** |
| --- | --- |
| AI | Artificial Intelligence |
| API | Application Programming Interface |
| CI/CD | Continuous Integration / Continuous Deployment |
| CRUD | Create, Read, Update, Delete |
| DB | Database |
| JSON | JavaScript Object Notation |
| JWT | JSON Web Token |
| ML | Machine Learning |
| NLP | Natural Language Processing |
| QA | Quality Assurance |
| SDK | Software Development Kit |
| UI | User Interface |
| UX | User Experience |
| UAT | User Acceptance Testing |
| IDE | Integrated Development Environment |
| KPI | Key Performance Indicator |
| DBMS | Database Management System |
| ORM | Object Relational Mapper |
| CLI | Command Line Interface |

### **21.9 Summary**

The **MediMate Glossary** provides unified terminology that underpins communication between developers, designers, analysts, and AI engineers.  
By maintaining a shared vocabulary — from infrastructure and testing to AI and UX — the team minimizes ambiguity, accelerates onboarding, and promotes consistent documentation across the entire MediMate ecosystem.

## **Section 22.0 – References and Resources**

**Purpose:**  
This section compiles all external references, documentation sources, libraries, frameworks, and integration resources that support the design, development, testing, and deployment of the MediMate system.

It serves as a quick-access index for developers, designers, and AI engineers — ensuring all project dependencies and related documentation are discoverable, standardized, and easy to maintain.

### **22.1 Primary Development References**

| **Category** | **Technology / Platform** | **Reference / Documentation Link** | **Usage in MediMate** |
| --- | --- | --- | --- |
| **Frontend Framework** | React Native | https://reactnative.dev/docs | Building cross-platform mobile UI (Android + iOS). |
| **Web Framework** | React.js | https://react.dev/ | Web dashboard and responsive interfaces. |
| **Backend Framework** | Node.js / Express | https://expressjs.com/ | RESTful APIs, authentication, and business logic. |
| **Database** | MongoDB | <https://www.mongodb.com/docs/> | Primary data storage for users, medications, and logs. |
| **Cloud Platform** | Firebase (Google Cloud) | https://firebase.google.com/docs | Authentication, Cloud Firestore, push notifications. |
| **AI/ML Framework** | Python / FastAPI | https://fastapi.tiangolo.com/ | AI microservices and predictive analytics. |
| **Model Training / ML Library** | TensorFlow / Scikit-learn / PyTorch | https://www.tensorflow.org/ / https://scikit-learn.org/ | Used for training and serving predictive models (e.g., mood analysis, side effect detection). |
| **Containerization** | Docker | https://docs.docker.com/ | Packaging backend, AI, and services for consistent deployment. |
| **Orchestration** | Kubernetes (K8s) | https://kubernetes.io/docs/home/ | Managing container clusters for scalability and resilience. |
| **Version Control** | Git / GitHub | https://docs.github.com/ | Source control, collaboration, and CI/CD integration. |
| **Scripting Language** | JavaScript / Python | https://developer.mozilla.org/en-US/docs/Web/JavaScript / https://docs.python.org/3/ | Core scripting languages used across the system. |

### **22.2 Testing and Quality Assurance Resources**

| **Category** | **Tool / Framework** | **Reference Link** | **Purpose / Use Case** |
| --- | --- | --- | --- |
| **Unit Testing (Frontend)** | Jest | https://jestjs.io/docs/getting-started | Testing UI components and logic. |
| **Integration / E2E Testing** | Cypress | https://www.cypress.io/ | Automated end-to-end testing for frontend. |
| **Mobile Testing** | Detox / Appium | https://wix.github.io/Detox/ / https://appium.io/docs/en/2.0/ | Testing React Native apps across Android/iOS. |
| **Backend Testing** | Mocha + Chai | https://mochajs.org/ / https://www.chaijs.com/ | API logic and response validation. |
| **Python Testing** | Pytest | https://docs.pytest.org/ | Unit and integration testing of AI modules. |
| **API Testing** | Postman / Newman | https://learning.postman.com/docs/ | Automated API tests and collections. |
| **Performance Testing** | Locust / K6 | https://locust.io/ / https://k6.io/docs/ | Load, stress, and performance evaluation. |
| **Security Testing** | OWASP ZAP / Bandit | https://www.zaproxy.org/docs/ / https://bandit.readthedocs.io/ | Vulnerability scanning and code auditing. |
| **Coverage Reports** | Istanbul / Coverage.py | https://istanbul.js.org/ | Generating code coverage metrics. |

### **22.3 DevOps and Deployment Resources**

| **Category** | **Tool / Platform** | **Documentation / Reference** | **Usage in MediMate** |
| --- | --- | --- | --- |
| **CI/CD Automation** | GitHub Actions | https://docs.github.com/actions | Automating build, test, and deployment pipelines. |
| **Cloud Provider** | Google Cloud Platform (GCP) | https://cloud.google.com/docs | Hosting API, AI microservices, and storage. |
| **Monitoring / Logging** | Grafana / Prometheus / Firebase Crashlytics | https://grafana.com/docs/ / https://prometheus.io/docs/ | Monitoring performance and logs. |
| **Secrets Management** | Google Secret Manager / dotenv | https://cloud.google.com/secret-manager/docs | Securing environment variables and API keys. |
| **Container Registry** | Docker Hub / GCP Artifact Registry | https://hub.docker.com/ | Hosting Docker images for deployments. |
| **Backup and Recovery** | GCP Storage / Firebase Backup | https://cloud.google.com/storage/docs | Database and log backups for disaster recovery. |

### **22.4 AI and Analytics Resources**

| **Category** | **Framework / Tool** | **Reference / Documentation Link** | **Use in MediMate** |
| --- | --- | --- | --- |
| **ML Framework** | TensorFlow | https://www.tensorflow.org/guide | AI model training (e.g., drug-interaction prediction). |
| **Statistical Modeling** | Scikit-learn | https://scikit-learn.org/stable/documentation.html | Classification and clustering tasks for predictive analysis. |
| **Deep Learning Framework** | PyTorch | https://pytorch.org/docs/ | Advanced mood and side-effect modeling. |
| **API Hosting (AI)** | FastAPI / Flask | https://fastapi.tiangolo.com/ | Serving trained models via REST API. |
| **Data Manipulation** | Pandas / NumPy | https://pandas.pydata.org/docs/ | Data cleaning, transformation, and preparation. |
| **Visualization** | Matplotlib / Plotly | https://matplotlib.org/stable/contents.html / https://plotly.com/python/ | Model insight visualization in analytics dashboards. |
| **Model Storage & Versioning** | MLflow / DVC | https://mlflow.org/docs/ | Managing and tracking AI model versions. |

### **22.5 Security, Privacy, and Compliance References**

| **Category** | **Standard / Tool** | **Reference Link** | **Purpose** |
| --- | --- | --- | --- |
| **Encryption** | AES / RSA Standards | https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf | Encryption standard for sensitive data protection. |
| **Authentication Protocols** | OAuth 2.0 / JWT | https://oauth.net/2/ / https://jwt.io/introduction | Secure access control for API endpoints. |
| **GDPR Compliance** | General Data Protection Regulation | https://gdpr-info.eu/ | Data privacy and user consent compliance. |
| **HIPAA Compliance** | Health Insurance Portability and Accountability Act | https://www.hhs.gov/hipaa/ | Health data protection compliance for MediMate medical features. |
| **OWASP Security Guide** | OWASP Foundation | https://owasp.org/Top10/ | Framework for identifying and mitigating security risks. |

### **22.6 Design and UX/UI Resources**

| **Category** | **Tool / Resource** | **Documentation / Reference** | **Usage in MediMate** |
| --- | --- | --- | --- |
| **Design Tool** | Figma | https://help.figma.com/hc/en-us | Prototyping, wireframing, and interface design. |
| **UI Library** | Shadcn / Tailwind CSS | https://ui.shadcn.com/docs / <https://tailwindcss.com/docs> | Building responsive, accessible, and consistent UI. |
| **Icon Set** | Lucide Icons | https://lucide.dev/docs/ | Unified iconography across web and mobile platforms. |
| **Accessibility Guidelines** | WCAG 2.1 | <https://www.w3.org/WAI/WCAG21/quickref/> | Ensuring app meets accessibility standards (A11y). |
| **Color and Typography Resources** | Google Fonts / Contrast Checker | https://fonts.google.com/ / https://contrast-ratio.com/ | Maintaining visual consistency and readability. |

### **22.7 Collaboration and Documentation Tools**

| **Category** | **Tool / Platform** | **Reference / Documentation** | **Purpose** |
| --- | --- | --- | --- |
| **Version Control Platform** | GitHub | https://docs.github.com/ | Code hosting and issue tracking. |
| **Documentation Platform** | Markdown / Notion / Confluence | https://www.markdownguide.org/ | Writing and maintaining project documentation. |
| **Communication** | Slack / Microsoft Teams | https://slack.com/help / https://learn.microsoft.com/en-us/microsoftteams/ | Developer collaboration and team discussions. |
| **Issue Tracking** | GitHub Projects / Jira | https://www.atlassian.com/software/jira | Task management, sprint tracking, and bug reporting. |
| **File Sharing** | Google Drive / Notion | https://drive.google.com/ | Centralized storage for design files and assets. |

### **22.8 Educational and Learning Resources**

| **Domain** | **Resource** | **Link** | **Learning Focus** |
| --- | --- | --- | --- |
| **React Native** | React Native Crash Course (Traversy Media) | <https://www.youtube.com/watch?v=0-S5a0eXPoc> | Building and debugging cross-platform apps. |
| **Node.js / Express** | Node.js Full Course (freeCodeCamp) | <https://www.youtube.com/watch?v=Oe421EPjeBE> | API development and middleware usage. |
| **Python for AI** | Google Machine Learning Crash Course | https://developers.google.com/machine-learning/crash-course | Understanding ML fundamentals. |
| **Docker / Kubernetes** | Docker & K8s Practical Guide | https://kubernetes.io/docs/tutorials/ | Containerization and orchestration for deployments. |
| **CI/CD** | GitHub Actions Workflow Guide | https://docs.github.com/actions | Setting up pipelines for automated testing and deployment. |
| **Cybersecurity** | OWASP Foundations | https://owasp.org/Top10/ | Secure development and threat modeling. |
| **UI/UX** | Google Material Design Guidelines | https://m3.material.io/ | Designing intuitive and accessible interfaces. |

### **22.9 Internal Documentation Index**

| **Document Name** | **Description** | **File Path / Location** |
| --- | --- | --- |
| **README.md** | Project overview and quick setup guide. | / root directory |
| **Architecture Overview** | System and module diagram with data flow. | /docs/architecture.md |
| **Coding Standards & Best Practices** | Code style, naming conventions, and review policies. | /docs/coding\_standards.md |
| **API Reference** | Full REST API documentation (Swagger/OpenAPI). | /docs/api\_reference.yaml |
| **Environment Setup Guide** | Installation and configuration instructions. | /docs/environment\_setup.md |
| **Testing Strategy Document** | Testing frameworks, coverage targets, and QA flows. | /docs/testing\_strategy.md |
| **Troubleshooting & Maintenance** | Common errors, fixes, and maintenance schedules. | /docs/troubleshooting.md |
| **AI Model Training Guide** | AI architecture, model lifecycle, and retraining workflow. | /ai/docs/model\_training.md |
| **Release Notes / Changelog** | Version history and updates. | /docs/changelog.md |

### **22.10 Summary**

The **References and Resources** section acts as the MediMate team’s definitive technical library.  
It centralizes every critical link — from frameworks and APIs to compliance, testing, and design references — ensuring all contributors can quickly find validated, official sources when developing, deploying, or expanding the MediMate ecosystem.

Maintaining this section up-to-date is a **key DevOps and documentation task** — new integrations, third-party APIs, or SDKs should be appended here with every major version release.

✅ **End of Developer Documentation**  
(Sections 1.0 – 22.0 complete: Comprehensive MediMate Developer Handbook)