1. Introduction

1.1 Purpose

This Site Reliability Engineering (SRE) plan ensures MediHelp+ stays reliable, fast, and user-friendly across its **AI-enhanced** backend (Django) and frontend (React) during the hackathon and early production.

1.2 Document Conventions

Follows IEEE SRS style, adapted for SRE: split into reliability requirements, system functions, monitoring structures, recovery systems, and timelines.

1.3 Intended Audience

- MediHelp+ frontend and backend developers
- Hackathon evaluators
- Future maintainers of the MediHelp+ platform

1.4 Scope

Covers:

- Frontend and backend uptime, performance, and error monitoring
- AI service monitoring (OpenAI, Gemini APIs)
- Lightweight auto-recovery
- User-centric performance observability
- Fast failure detection and simple alerts

---

2. Overall Description

2.1 Product Perspective

The SRE system is a monitoring and auto-recovery layer around the Django backend and React frontend, tailored to MediHelp+’s use of AI services.

2.2 Product Functions

- Backend API uptime checks and auto alerts
- Frontend error and performance monitoring
- AI API (OpenAI/Gemini) health checking
- Simple database availability checks
- Lightweight process auto-recovery
- Real-time Discord alerting

2.3 User Characteristics

- Developers: need fast, actionable alerts
- Hackathon judges: evaluating resilience
- Users: expect quick, reliable responses

2.4 Constraints

- Must fit a 7-day timeline
- Prefer free or open-source tools
- Minimal server overhead
- Monitor both app and AI reliability

3. Specific Requirements

3.1 Functional Requirements

3.1.1 Backend Monitoring

- Uptime Kuma monitors Django API `/api/healthz`
- Basic pg_isready database check every 2 minutes
- Monitor AI API usage:
    - Latency over 2s triggers warning
    - Failed AI calls (non-2xx response) logged separately

3.1.2 Frontend Monitoring

- Sentry captures unhandled React errors
- Core Web Vitals (LCP, FID, CLS) logged
- Skip rrweb-lite (optional if ahead of schedule)

3.1.3 Alerting

- Use Healthchecks.io or similar bots for Discord alerts
- Uptime failures, 5xx spikes, AI API failures push Discord alerts immediately

3.1.4 Recovery Systems

- Backend managed by PM2 or systemd, auto-restarts on failure
- Scheduled server health reports (simple scripts + Discord)

3.2 Non-Functional Requirements

- Reliability: ≥99% uptime during hackathon
- Performance: Monitor overhead ≤5% of resource usage
- Scalability: Framework extensible beyond hackathon
- Usability: Alerts in simple English with direct action items
- Security: API keys (OpenAI, Gemini) hidden from logs

---

4. Monitoring Structures

4.1 Backend Monitoring

Metric	Tool	Threshold	Action
API Health (/api/healthz)	Uptime Kuma	Fail 2+ checks	Discord alert
PostgreSQL Availability	`pg_isready` cron	No response 2 min	Discord alert
API Error Rate (500s)	Django log analysis	>5% errors	Discord alert
AI API Latency	Custom Django middleware	>2s response	Log warning
AI API Failures	Django middleware	>1 failure/min	Discord alert

4.2 Frontend Monitoring

Metric	Tool	Threshold	Action
Javascript Errors	Sentry	Any critical error	Discord alert
LCP (Largest Contentful Paint)	Web Vitals API	>2.5s	Log warning
CLS (Cumulative Layout Shift)	Web Vitals API	>0.1	Log warning
FID (First Input Delay)	Web Vitals API	>100ms	Log warning

---

5. Recovery and Automation

5.1 Backend Recovery

- PM2 or systemd watches Django server
- Auto-restart on crash within 10 seconds
- Optional: Healthchecks.io monitor server reboots

5.2 Frontend Crash Handling

- Critical Sentry errors suggest users refresh (basic modal popup)
- Frontend does **not** auto-restart, but user guidance will mitigate.

5.3 AI API Degradation

- On failed AI response, show "Service Temporarily Unavailable" graceful message instead of crashing the app.

---

6. Milestones and Timeline

Milestone	Deliverable	Deadline
Monitoring Setup	Uptime Kuma, Healthchecks.io bot, Sentry	Day 1
API Health Checks	`/api/healthz`, pg_isready cron job	Day 2
Basic Alerting	Discord alerts for backend/frontend failures	Day 3
AI API Monitoring	Latency + failure monitoring for OpenAI, Gemini	Day 4

Automated Recovery	PM2 auto-restart backend	Day 5
Final Observability	Web Vitals tracking in production build	Day 6
Chaos Testing + Polish	Simulated failures and final thresholds tuning	Day 7

---

7. Appendices

Technology Stack

- **Backend Monitoring:** Uptime Kuma, Healthchecks.io, PM2/systemd
- **Frontend Monitoring:** Sentry, Web Vitals API
- **Database Monitoring:** `pg_isready` (PostgreSQL native)
- **Alerting:** Discord webhooks (via Healthchecks.io)
- **AI Monitoring:** Custom Django middleware for OpenAI/Gemini APIs
- **Deployment/Recovery:** PM2, simple Bash scripts

✅ Key Improvements from Previous Version

- 🚀 **Much faster to implement**
- 🔥 **Includes AI service monitoring (critical!)**
- 🧹 **Removed complex session replay**
- 📩 **Simplified alerts with Healthchecks.io bots**

- 💻 **Minimal server load**
- 🛡️ **More focused on protecting user experience during failures**