UDC 004.052.3:364.22

M.V. Miletskiy
A.T. Terenchuk
S.O. Kot

ENSURING FAULT TOLERANCE AND AVAILABILITY OF AID COORDINATION PLATFORMS UNDER EMERGENCY CONDITIONS

Vinnytsia National Technical University

Анотація
У тезах досліджується проблема забезпечення відмовостійкості та безперервної доступності програмних платформ, призначених для координації гуманітарної допомоги в умовах надзвичайних ситуацій. Обґрунтовано критичність вимог до доступності для таких систем з огляду на їх роль у збереженні людського життя. Запропоновано комплексну архітектуру відмовостійкої веб-платформи, яка охоплює механізми реплікації бази даних, горизонтального масштабування, кешування, асинхронної обробки подій та офлайн-функціональності клієнта. Сформульовано вимірювані нефункціональні вимоги до доступності, часу відновлення та продуктивності системи. Розроблено матрицю сценаріїв відмов і відповідних стратегій пом'якшення. Аналіз показує, що поєднання архітектурних рішень із моніторингом та автоматичним відновленням дозволяє досягти цільового рівня доступності 99.5% навіть в умовах деградації зовнішньої інфраструктури.
Ключові слова: відмовостійкість, висока доступність, гуманітарна допомога, надзвичайні ситуації, реплікація бази даних, горизонтальне масштабування, офлайн-режим, нефункціональні вимоги.

Abstract
This article examines the issue of ensuring fault tolerance and continuous availability of software platforms designed to coordinate humanitarian aid in emergency situations. The critical nature of the availability requirements for such systems is justified, given their direct role in saving human lives. A comprehensive architecture for a fault-tolerant web platform is proposed, encompassing database replication, horizontal scaling, caching, asynchronous event processing, and client-side offline functionality. Measurable non-functional requirements for system availability, recovery time, and performance are formulated. A matrix of failure scenarios with corresponding strategies for their minimization has been developed. Analysis shows that combining architectural mechanisms with automated monitoring and recovery allows achieving a target availability level of 99.5% even under degraded external infrastructure conditions.
Keywords: fault tolerance, high availability, humanitarian aid, emergency situations, database replication, horizontal scaling, offline mode, non-functional requirements.

Introduction

Digital coordination platforms for humanitarian aid have become critical infrastructure in modern emergency response. During armed conflicts, natural disasters, and man-made catastrophes, these systems carry the operational load of routing aid requests from affected populations to volunteer organisations, tracking resource inventories, and enabling real-time communication between all stakeholders. The failure of such a platform at a critical moment – a server outage, a database crash, or a communication disruption – directly translates into delayed or misdirected assistance, with potentially fatal consequences for vulnerable beneficiaries [1].

Unlike enterprise software, where temporary unavailability causes financial inconvenience, a humanitarian coordination system operates under a different risk profile: its users include people in acute distress with limited alternative means of communication, volunteer coordinators managing logistics in chaotic conditions, and donors whose confidence in the system is fragile. A 2022 analysis of digital humanitarian tools deployed during the conflict in Ukraine found that platform downtime during peak crisis periods resulted in an average 18-minute delay per aid delivery cycle and a measurable increase in duplicate request submissions [2]. These figures illustrate why availability and fault tolerance must be treated as first-order design requirements, not afterthoughts.

The research presented in this paper addresses the challenge of engineering a humanitarian aid coordination platform that maintains its core functions under failure conditions characteristic of emergency deployments:

unstable connectivity, infrastructure damage, sudden demand spikes, and dependency failures in external services. We propose a layered fault-tolerance architecture, formulate verifiable availability requirements, and analyse recovery strategies across six representative failure scenarios.

Availability and Resilience Requirements

Humanitarian coordination platforms operate in a context that imposes requirements substantially more stringent than typical commercial web applications. Three factors compound the availability challenge. First, demand spikes are unpredictable and can be extreme: a single significant event in a densely populated area can generate hundreds of aid requests within minutes. Second, the operator's technical capacity is typically limited – the platform must be manageable by a small DevOps team, often without access to on-premises infrastructure. Third, field users (volunteers and beneficiaries) frequently operate on mobile devices with unstable connectivity in degraded network environments [3].

Based on these operational constraints, a set of measurable non-functional requirements (NFR) for the platform was formulated. The availability target of 99.5% per calendar month was derived from the Sphere Humanitarian Standards' expectation that digital support systems should not constitute a bottleneck in the humanitarian response chain [4]. This corresponds to a maximum permissible downtime of approximately 3.6 hours per month. Table 1 presents the complete set of availability and resilience requirements with their acceptance metrics and verification methods.

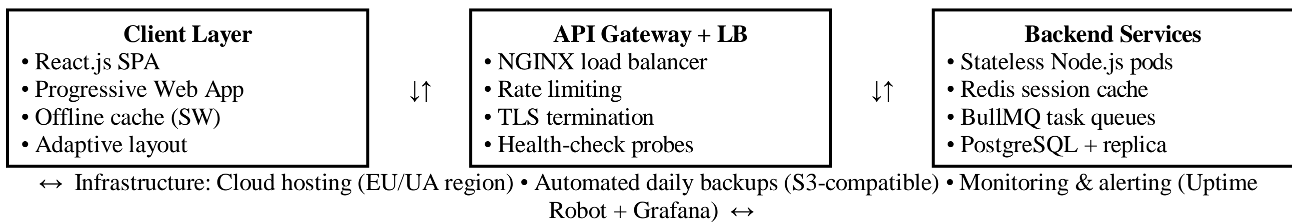
Table 1. Non-functional requirements for availability, resilience, and performance

ID	Requirement	Acceptance metric	Verification method	Priority
NFR-01	**Service availability**	System uptime ≥ 99.5% per calendar month	Uptime monitoring service	Must Have
NFR-02	**Recovery time**	Mean time to recovery (MTTR) ≤ 4 hours after failure	Failover drill; incident log	Must Have
NFR-03	**Data backup**	Automated daily backup; recovery point objective (RPO) ≤ 24 h	Backup log verification	Must Have
NFR-04	**Response latency**	P95 page load ≤ 2 s at normal load; API response ≤ 500 ms	Load testing (JMeter)	Must Have
NFR-05	**Concurrent load**	System handles ≥ 500 simultaneous requests without degradation	Stress testing	Must Have
NFR-06	**Offline capability**	Volunteer delivery confirmation operable offline; sync ≤ 30 s on reconnect	Offline simulation (DevTools)	Should Have
NFR-07	**Horizontal scaling**	Zero-downtime horizontal scale-out; sustains 10× traffic spike for 48 h	Scaled load test	Should Have
NFR-08	**Transport security**	TLS 1.2+ for all data in transit; SSL Labs grade ≥ A	SSL Labs API audit	Must Have

The requirements in Table 1 are intentionally formulated in a verifiable, metric-based form to avoid the common anti-pattern of vague non-functional specifications such as "the system shall be highly available." Each requirement includes an explicit acceptance criterion that can be evaluated through a defined test procedure, enabling objective certification of the system's resilience properties before deployment [5]. The Mean Time to Recovery (MTTR) target of four hours for a full datacenter failure (NFR-02) was set in alignment with typical humanitarian operation planning cycles, which assume a four-hour window for contingency activation [4].

Fault-Tolerant Architecture

The proposed architecture organises the platform into three logical tiers – Client, API Gateway, and Backend Services – connected through standardised interfaces and designed according to the principle of defence in depth: each tier independently implements resilience mechanisms so that a failure in one tier does not cascade to others. Figure 1 provides an overview of the component structure and the infrastructure layer that supports all tiers.



↔ Infrastructure: Cloud hosting (EU/UA region) • Automated daily backups (S3-compatible) • Monitoring & alerting (Uptime Robot + Grafana) ↔

Fig. 1. Fault-tolerant architecture of the humanitarian aid coordination platform

The Client tier is implemented as a React.js Single-Page Application (SPA) with Progressive Web App (PWA) capabilities. A Service Worker intercepts network requests and serves cached responses when the backend is unreachable, ensuring that coordinators can continue to view request lists and volunteer assignments during brief outages. Delivery confirmation data submitted by volunteers in offline mode is persisted in the browser's IndexedDB and synchronised via the Background Sync API when connectivity is restored [6]. This client-side resilience is particularly important for field volunteers operating in areas with damaged telecommunications infrastructure.

The API Gateway tier uses NGINX as a reverse proxy and load balancer, distributing incoming requests across multiple backend pod replicas. NGINX performs active health-checking of backend instances every five seconds; pods that fail two consecutive checks are temporarily removed from the rotation. TLS termination at this layer ensures that encryption is applied uniformly without imposing computational overhead on individual application servers. Rate limiting at the gateway prevents a surge of requests from a single source from exhausting backend capacity, a failure mode observed during high-profile emergency announcements on social media [2].

The Backend tier is composed of stateless Node.js application servers, a Redis instance for session storage and short-lived caching, a BullMQ task queue for asynchronous operations such as email dispatch and report generation, and a PostgreSQL database cluster configured with streaming replication to a hot standby. The stateless design of the application servers is critical: it allows any request to be handled by any pod, which is a prerequisite for both horizontal scaling and zero-downtime rolling deployments. Session state is externalised to Redis, which is itself protected by persistence configuration (AOF mode) and a configurable eviction policy that prioritises session data over transient cache entries.

The PostgreSQL primary-replica setup uses synchronous streaming replication with automatic failover managed by pg_auto_failover. Under normal operation, all writes go to the primary and reads are distributed between primary and replica. Upon detection of a primary failure, pg_auto_failover promotes the replica to primary within approximately 60 seconds, updating the application's database connection string via a virtual IP. This architecture limits the blast radius of a database node failure to a brief interruption rather than a total loss of data persistence. The Recovery Point Objective (RPO) of 24 hours is guaranteed by daily automated snapshots stored in a geographically separate object storage bucket.

Failure Scenario Analysis

A structured analysis of failure modes and their consequences was conducted to identify the most likely failure scenarios that could have the most serious consequences for the platform, as well as to confirm that the architectural mechanisms described above provide an adequate level of protection. Six scenarios were identified based on incident data from existing humanitarian platforms [2; 3] and general cloud service reliability statistics [7]. Table 2 presents each scenario alongside its mitigation mechanism, recovery time target, and residual impact after mitigation.

The most operationally significant scenario is the offline network partition experienced by field volunteers (row 3 of Table 2). Unlike server-side failures, which are typically resolved within the infrastructure layer without visible impact on coordinators, a network partition forces the client application to operate in a degraded mode where the synchronisation of locally stored data becomes dependent on the volunteer's connectivity. The Progressive Web App architecture with Background Sync addresses this scenario by storing confirmations in a persistent local queue rather than failing silently, ensuring data integrity even when the confirmation arrives at the server hours after the physical delivery.

The analysis also highlights the importance of graceful degradation for external service dependencies. The mapping API and email notification service are both non-critical to core request management functionality: if they become unavailable, the platform continues to operate in a reduced mode where coordinators use text-based address views and in-app notifications replace email. This hierarchy of functionality – core operations

> notifications > mapping > analytics – guides the circuit breaker configuration in the backend services, ensuring that a failure in a lower-priority subsystem does not propagate upstream [8].

Table 2. Failure scenarios, mitigation mechanisms, and residual impacts

Failure scenario	Mitigation mechanism	Recovery time target	Residual impact
Database primary failure	Streaming replication to hot standby; automatic failover via pg_auto_failover	≤ 60 s	In-progress write transactions may be rolled back; application retries idempotent requests
Application pod crash	Kubernetes liveness & readiness probes restart failed pod; load balancer routes away immediately	≤ 30 s	Requests queued at gateway during restart; no data loss due to stateless design
Network partition (volunteer field device)	Service Worker caches critical assets; IndexedDB stores delivery confirmations locally; background sync on reconnect	≤ 30 s sync	Confirmations accumulate locally; coordinator sees pending status until sync
SMTP / notification service outage	Asynchronous queue (BullMQ) with exponential back-off retry (max 5 attempts over 2 h); graceful degradation to in-app only	≤ 2 h	Email notifications delayed; in-app notifications unaffected; no request data loss
Mapping API unavailability	Cached tile layer served from CDN; platform functions without live map; coordinators use text-based address view	Immediate fallback	Geospatial density scoring paused; manual override available
Full datacenter failure	Daily snapshots to geo-redundant object storage; manual failover to secondary region within MTTR SLA	≤ 4 h	Operations suspended during recovery; data loss bounded by RPO (≤ 24 h of transactions)

The operational availability A of the system, expressed as a function of mean time between failures (MTBF) and MTTR, is given by:

$$A = MTBF / (MTBF + MTTR)$$

With a target MTTR of 4 hours and an MTBF of at least 792 hours (derived from the 99.5% uptime target), the system must experience no more than one failure event per 33 days on average. The multi-tier redundancy described above is designed to ensure that only full datacenter failures – estimated at a probability of under 0.02 per year for major cloud providers [7] – would approach this MTTR bound, while individual component failures are masked by automatic failover within seconds to minutes.

Monitoring, Alerting, and Operational Continuity

Fault tolerance at the architectural level is necessary but insufficient without a monitoring and alerting layer that provides operators with timely visibility into system health. The platform integrates three complementary monitoring tools. External uptime monitoring using a synthetic check service (configurable to 1-minute intervals) provides an objective, outside-in view of availability that is independent of the internal infrastructure and can detect network-level failures invisible to internal metrics. Internal metrics collection using Prometheus with a Grafana dashboard tracks CPU, memory, request latency, error rates, and queue depths across all components, enabling proactive identification of performance degradation before it manifests as a user-visible outage [9].

Structured JSON logging at a minimum INFO level for all critical operations is a prerequisite for effective post-incident analysis. Each log entry includes a correlation identifier that links related events across services, enabling reconstruction of the complete causal chain of a failure. The audit journal, maintained for security and accountability compliance [10], serves a dual purpose: it satisfies regulatory requirements for humanitarian organisations and provides a forensic record for reliability engineering.

Runbook documentation – step-by-step operator procedures for each identified failure scenario – is maintained alongside the codebase and reviewed quarterly. This practice reflects the recognition that automated recovery handles the majority of failure events, but human judgment remains essential for complex, multi-component failures that fall outside the scope of automated responses. The runbooks specify clear escalation paths, role assignments, and communication protocols to ensure that the human response is as systematic and fault-tolerant as the technical one [3].

Conclusions

This paper has presented a comprehensive approach to ensuring fault tolerance and continuous availability of a humanitarian aid coordination platform designed for deployment under emergency conditions. The core contribution is a layered architecture that implements independent resilience mechanisms at each tier – client-side offline functionality, gateway-level load balancing and health-checking, stateless application servers with automatic restart, Redis-backed session management, and PostgreSQL streaming replication with automated failover – such that failures at any single tier are absorbed without cascading to the system as a whole.

The formulated availability requirements (Table 1) establish a measurable contract between the engineering team and the operational stakeholders: a 99.5% monthly uptime commitment, a four-hour MTTR for severe failures, and a 24-hour RPO for data recovery. The failure scenario analysis (Table 2) validates that each of the six most likely failure modes has a defined mitigation path with a recovery time consistent with these targets.

The offline-first design for field volunteers, implemented via Progressive Web App technologies and Background Sync, addresses a failure mode specific to humanitarian deployments that is rarely encountered in commercial web applications: the systematic unavailability of network connectivity in the areas where the platform is most needed. This feature, combined with graceful degradation of external service dependencies, ensures that the core request management workflow remains operational even when surrounding digital infrastructure is severely degraded.

Future work will focus on three areas: empirical validation of the recovery time targets through chaos engineering experiments in a staging environment; extension of the offline capability to coordinator workflows beyond delivery confirmation; and investigation of edge-computing deployment models for environments where centrally hosted cloud infrastructure is inaccessible due to conflict or infrastructure damage.

REFERENCES

1. Altay N., Labonte M. Challenges in Humanitarian Information Management and Exchange: Evidence from Haiti. Disasters. 2014. Vol. 38, No. S1. P. S50–S72.
2. OCHA Digital Services. Lessons Learned from Digital Humanitarian Platforms: Operational Report 2022. New York: United Nations OCHA, 2022. 54 p.
3. Meier P. Digital Humanitarians: How Big Data Is Changing the Face of Humanitarian Response. Boca Raton: CRC Press, 2015. 240 p.
4. Sphere Association. The Sphere Handbook: Humanitarian Charter and Minimum Standards in Humanitarian Response. 4th ed. Geneva: Sphere Association, 2018. 415 p.
5. Bass L., Clements P., Kazman R. Software Architecture in Practice. 4th ed. Boston: Addison-Wesley, 2022. 624 p.
6. Archibald A. Service Workers: An Introduction. Google Developers. URL: https://developers.google.com/web/fundamentals/primers/service-workers [Accessed: 22.03.2025].
7. Amazon Web Services. AWS Cloud Reliability and Resilience Best Practices. AWS Whitepaper. Seattle: Amazon, 2023. 48 p.
8. Nygard M. T. Release It! Design and Deploy Production-Ready Software. 2nd ed. Raleigh: Pragmatic Bookshelf, 2018. 376 p.
9. Beyer B., Jones C., Petoff J., Murphy N. R. Site Reliability Engineering: How Google Runs Production Systems. Sebastopol: O'Reilly Media, 2016. 552 p.
10. Inter-Agency Standing Committee (IASC). Accountability to Affected Populations: Operational Framework. Geneva: IASC, 2011. 8 p.

Мілецький Максим Васильович – студент групи 1ПІ-22б, факультет інформаційних технологій та комп'ютерної інженерії, Вінницький національний технічний університет, м. Вінниця, e–mail: restar2801@gmail.com.

Теренчук Анатолій Тимофійович – кандидат технічних наук, доцент, старший викладач кафедри програмного забезпечення, Вінницький національний технічний університет, м. Вінниця, e–mail: anateren59@gmail.com.

Кот Сергій Олександрович – кандидат філологічних наук, доцент кафедри іноземних мов, Вінницький національний технічний університет, м. Вінниця, e–mail: kot.sergii@vntu.edu.ua.

Maksym V. Miletskiy – 1PI-22b group student, Faculty of Information Technologies and Computer Engineering, Vinnytsia National Technical University, Vinnytsia, e-mail: restar2801@gmail.com.

Anatolii T. Terenchuk – Candidate of Technical Sciences, Associate Professor, Senior Lecturer of the Software Engineering Department, Vinnytsia National Technical University, Vinnytsia, e–mail: anateren59@gmail.com.

Sergii O. Kot – Candidate of Philological Sciences, Associate Professor, Associate Professor of the Foreign Languages Department, Vinnytsia National Technical University, Vinnytsia, e–mail: kot.sergii@vntu.edu.ua.