Public Health Engagement & Data Portal - System Design

Prepared by: Monty Dimkpa

Email: info@montydimkpa.fyi

Website: https://montydimkpa.fyi

Executive Summary

Public health programs across Africa face a persistent challenge: field workers operate in areas with unreliable connectivity, yet the data they collect is critical for disease surveillance, vaccination campaigns, and outbreak response. This system addresses that reality head-on.

We're building a portal that treats offline operation as the default, not an edge case. Community health workers can record patient encounters, log intervention activities, and coordinate with their teams regardless of network availability. When connectivity returns, the system synchronizes intelligently—resolving conflicts and pushing updates to national health information systems like DHIS2.

The platform serves multiple organizations (ministries of health, NGOs, research institutions) from a shared infrastructure while keeping their data strictly isolated. We've designed for the constraints we know we'll face: inconsistent power, shared devices, varying levels of technical literacy among users, and the need to integrate with legacy EMR systems that won't be replaced anytime soon.

Our technology choices reflect these priorities:

Layer	Choice	Why
Frontend	React + React Native	Single codebase expertise, strong PWA/offline support
Backend	Node.js	Team familiarity, good async I/O for sync operations
Primary Database	PostgreSQL	Mature, PostGIS for mapping, row-level security for tenants
Sync Queue	DynamoDB	Handles burst writes during mass sync events
Event Streaming	Kafka	Reliable delivery for health data—we can't afford dropped messages
Cloud	AWS	Best regional presence near Africa (eu-west-1, af-south-1)

We're following FHIR R4 for health data interoperability. It's not perfect, but it's what DHIS2 and most modern EMRs speak.

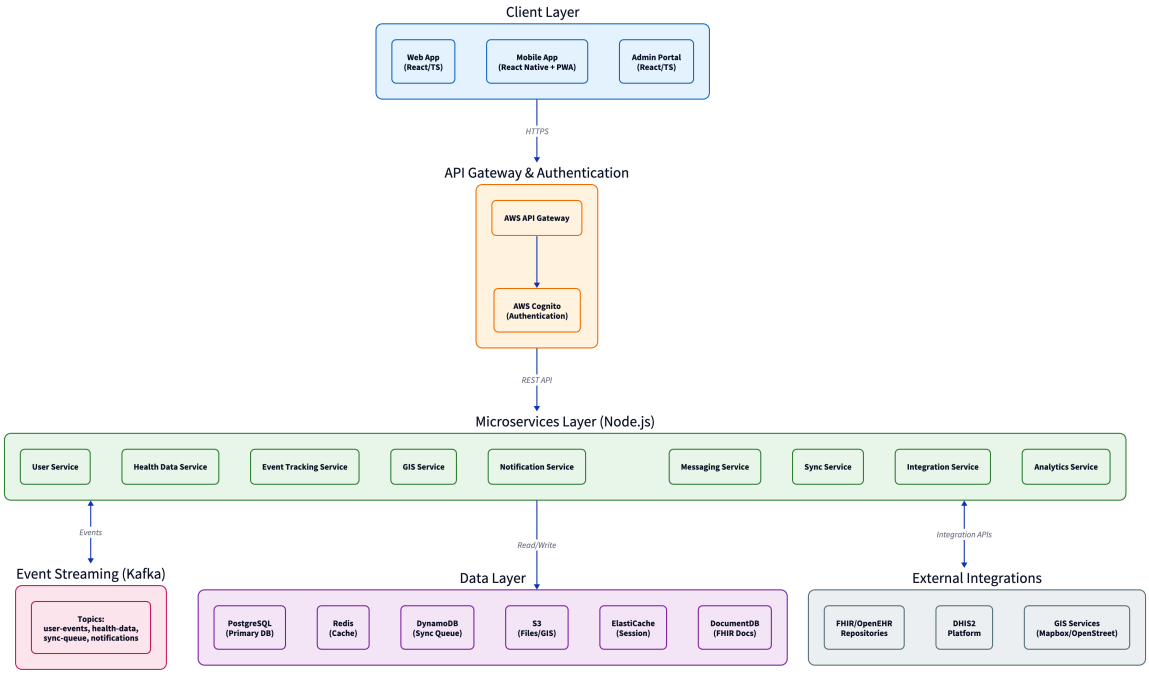

Figure 1: System Architecture

1. High-Level Architecture

1.1 Architecture Overview

1.2 How the Pieces Fit Together

The architecture follows a fairly standard microservices pattern, but a few decisions deserve explanation.

Why we separated the Sync Service from other backend services: Synchronization is the hairiest part of this system. Conflict resolution, queue management, and retry logic are complex enough that we want a dedicated service we can scale independently. When 500 field workers come back online after a regional network outage, the Sync Service will bear the brunt of the load.

Why Kafka instead of simpler alternatives: We considered SQS and even Redis pub/sub. Kafka won because health data updates need guaranteed delivery with ordering. If a patient record is created and then updated, those events must be processed in sequence. Kafka's partition-based ordering gives us that. The operational complexity is real, but the data integrity requirements justify it.

Why DocumentDB for FHIR resources: FHIR resources are deeply nested JSON documents that don't map cleanly to relational tables. We store the canonical FHIR representation in DocumentDB and maintain searchable indexes in PostgreSQL. This adds complexity but avoids the nightmare of constantly transforming between FHIR JSON and relational schemas.

The frontend applications—web, mobile, and admin portal—all talk to the same API Gateway. We're using AWS Cognito for authentication because it handles the undifferentiated heavy lifting (password policies, MFA, token refresh) and integrates cleanly with API Gateway.

Core Services:

The system breaks down into nine services. Rather than describe each exhaustively, here's how they cluster

by concern:

Data management: User Service, Health Data Service, Analytics Service *Field operations:* Event Tracking Service, GIS Service, Sync Service *Communication:* Notification Service, Messaging Service *External systems:* Integration Service

Each service owns its data and exposes REST APIs. They communicate asynchronously through Kafka when the operation doesn't need an immediate response.

---

2. Service Details

This section covers the services that need more explanation. We won't belabor the obvious ones—the User Service manages users, the Notification Service sends notifications. Instead, we'll focus on the services with non-trivial design decisions.

2.1 Health Data Service

This service is the heart of the system. It handles FHIR R4 resources (Patient, Observation, Encounter, etc.) and needs to satisfy two competing requirements: store data in a standards-compliant format for interoperability, and query it efficiently for the UI.

Our approach: store the canonical FHIR JSON in DocumentDB, and maintain denormalized search indexes in PostgreSQL. When a FHIR resource is created or updated, we write to both stores within a transaction (using the Saga pattern for cross-database consistency).

We're also supporting OpenEHR for organizations that use it. The service includes transformation logic between FHIR and OpenEHR, though we expect most integrations to use FHIR.

2.2 Sync Service

The most complex service in the system. It manages the offline queue and handles what happens when a field worker reconnects after hours or days of offline work.

The sync flow works like this:

1. Client stores mutations locally in IndexedDB
2. When online, client pushes pending changes to the Sync Service
3. Sync Service writes to DynamoDB (fast, handles burst traffic)
4. Background workers process the queue, applying changes to the main database
5. Conflicts are detected by comparing timestamps and version numbers
6. Simple conflicts (e.g., two people edited different fields) are auto-merged
7. Complex conflicts (e.g., same field edited differently) are flagged for user resolution

Conflict resolution hierarchy:

We don't treat all users equally when resolving conflicts. A clinician's edits take precedence over a field officer's. An edit made while online beats one made offline (assuming the online user had fresher data). These rules handle 80% of conflicts automatically.

2.3 GIS Service

Mapping is central to public health work—visualizing disease spread, planning vaccination routes, identifying coverage gaps. This service handles geospatial data using PostGIS for queries and serving pre-rendered map tiles from S3.

We integrate with Mapbox for the base layer, with OpenStreetMap as a fallback. For offline use, we pre-cache map tiles for known operational areas.

A limitation to note: Real-time location tracking of field workers is intentionally excluded. The privacy implications are significant, and the operational benefit doesn't justify the risk.

2.4 Integration Service

This is our adapter layer to the outside world. Currently scoped to:

- **DHIS2:** Bidirectional sync of aggregate health data. Most ministries of health use DHIS2, so this integration is critical.
- **EMR systems:** Receive patient data via FHIR APIs or HL7 messages. We're not trying to replace EMRs—just consume their data.
- **GIS services:** Pull boundary data, facility locations, and population estimates.

Each integration is configurable per tenant. A ministry of health will have different DHIS2 credentials and data mappings than an NGO operating in the same region.

2.5 How Services Communicate

Services talk to each other in two ways:

Synchronous (REST): When the caller needs an immediate response. User authentication, data validation, queries that power UI screens.

Asynchronous (Kafka): When the operation can happen eventually. A new patient record triggers notifications, analytics updates, and DHIS2 sync—none of which need to block the original request.

We use AWS ECS Service Discovery for service-to-service calls, with an Application Load Balancer handling external traffic through the API Gateway.

---

3. Multi-Tenancy Architecture

We need to support multiple organizations on shared infrastructure without any risk of data leakage between them. A ministry of health must never see an NGO's data, even if both are operating in the same country.

3.1 The Approach We Chose (and Why)

There are three common patterns for multi-tenancy:

Approach	Isolation	Cost	Operational Complexity
Separate databases per tenant	Strongest	Highest	High (many DB instances)
Shared database, separate schemas	Strong	Medium	Medium
Shared tables with tenant_id column	Weakest	Lowest	Low

We're using **separate schemas** as our primary approach. Each tenant gets their own PostgreSQL schema with identical table structures. This gives us strong isolation (a bug in a query can't accidentally leak data across schemas) while keeping operational costs reasonable.

For very large tenants or those with strict data residency requirements, we can provision a dedicated database. The application code doesn't change—just the connection routing.

3.2 How It Works in Practice

Authentication flow: 1. User logs in through Cognito 2. JWT token includes a `tenant_id` custom claim 3. Every API request extracts tenant_id from the token 4. Database connections are scoped to that tenant's schema 5. Row-level security policies act as a safety net (defense in depth)

Kafka topic strategy: We use a single set of topics with tenant_id in message headers, rather than per-tenant topics. With potentially hundreds of tenants, per-tenant topics would create operational overhead in Kafka cluster management. The consumer code filters by tenant_id.

3.3 Limitations

This model has constraints we've accepted:

- **Cross-tenant analytics are harder.** Aggregating data across tenants (e.g., for a funder who supports multiple organizations) requires explicit data sharing agreements and separate ETL jobs.
- **Schema migrations affect everyone.** We can't have tenant-specific schema customizations—everyone gets the same tables.
- **Noisy neighbor risk.** A tenant running heavy queries can impact others. We mitigate this with connection pooling limits and query timeouts, but it's not eliminated.

---

4. Offline-First Design

This is the defining technical challenge of the system. In many deployment contexts, connectivity isn't just unreliable—it's the exception. A community health worker might have signal for 30 minutes in the morning when they're near the district office, then nothing for the rest of the day while visiting households.

We've designed assuming the client will be offline most of the time.

4.1 What "Offline-First" Means Concretely

Every feature must work without connectivity. If a feature requires real-time server communication, it either gets redesigned or we're explicit that it's only available online.

Local storage is the source of truth for the user. When offline, the app reads from and writes to IndexedDB. The user doesn't see spinners or error states—the app just works.

Sync is background and incremental. When connectivity appears, the app syncs in the background without interrupting the user's workflow. It only syncs what changed, not the entire dataset.

4.2 The Hard Part: Conflict Resolution

Conflicts are inevitable. Two users might edit the same patient record while both are offline. Someone might update a record on the web while a field worker has a stale copy on their phone.

Our resolution strategy is tiered:

Automatic resolution (most cases): - Different fields edited → merge both changes - Same field, same value → no conflict - Metadata changes (timestamps, audit fields) → server always wins - User role hierarchy → clinician > nurse > field officer > data clerk

User resolution (when we can't decide): - Same field, different values, same role level - Deletions conflicting with edits - Any conflict involving diagnosis or treatment data

The UI shows conflicting changes side-by-side and lets the user pick. We store both versions until resolved—no data is lost.

4.3 Bandwidth Constraints

Many users will be on 2G connections or expensive mobile data. We've optimized for this:

- API responses are gzip compressed (typically 70-80% reduction)
- Sync payloads only include changed fields, not full records
- Images and attachments sync separately, with user control over timing
- Map tiles are pre-cached for known operational areas

4.4 What We're Not Solving

Real-time collaboration (Google Docs-style concurrent editing) is out of scope. The complexity isn't justified for the use cases we're targeting. Users will occasionally see "this record was modified by someone else" messages, and that's acceptable.

---

5. Data Storage

We're using multiple databases, each chosen for a specific purpose. This adds operational complexity, but the alternative—forcing everything into PostgreSQL—would create worse problems.

5.1 Why Multiple Databases

Store	Purpose	Why Not Just Postgres?
PostgreSQL	Relational data, search indexes, tenant schemas	It's our primary store—no argument here
DocumentDB	FHIR resources (complex nested JSON)	Deeply nested documents are painful in relational tables
DynamoDB	Sync queue	Handles 10,000+ writes/second during sync storms
Redis	Caching, sessions, rate limits	Sub-millisecond reads for hot data
S3	Files, GIS tiles, backups	Blob storage doesn't belong in databases

5.2 PostgreSQL Details

Each tenant gets a schema. Within that schema, the core tables are:

- `users`, `roles`, `permissions` — standard RBAC
- `health_records` — searchable index of FHIR resources (not the full documents)
- `events` — community health events and interventions
- `messages` — team communication
- `sync_metadata` — tracking what's been synced

We partition large tables by date. Health records older than 2 years move to archive partitions with slower storage.

Backup strategy: Daily snapshots via RDS, point-in-time recovery with a 7-day window. Critical tables replicate to a DR region.

5.3 The DynamoDB Sync Queue

This deserves special mention. When 500 field workers come online simultaneously after a network outage, they'll all try to sync at once. PostgreSQL would buckle under that write load.

DynamoDB handles it. We write sync operations to a DynamoDB table (partition key: tenant_id + user_id, sort key: timestamp), then background workers process the queue at a sustainable rate. Items auto-delete after 30 days via TTL.

---

6. External Integrations

Integrations are where projects like this usually get messy. External systems have their own release schedules, their own bugs, and their own interpretations of "standards."

6.1 DHIS2 — The Critical Integration

Most African ministries of health use DHIS2 for health information management. Our integration needs to:

1. Push aggregate data (e.g., "47 children vaccinated in District X this week")
2. Pull reference data (facility lists, organization units, data element definitions)
3. Handle the reality that every DHIS2 instance is configured differently

We've built an abstraction layer with configurable field mappings per tenant. When a tenant onboards, someone manually maps their DHIS2 data elements to our internal schema. It's not elegant, but DHIS2's flexibility makes full automation impractical.

Sync frequency: Configurable per tenant, typically every 6-24 hours. We batch changes and retry failures with exponential backoff.

6.2 EMR Systems

EMRs are harder than DHIS2. There's no dominant system—OpenMRS, DHIS2 Tracker, proprietary systems, paper records being digitized. We support:

- **FHIR R4 APIs** for modern EMRs that expose them
- **HL7 v2 messages** for legacy systems (via a translation layer)
- **Flat file imports** as a last resort

We're consumers, not producers. The EMR remains the source of truth for clinical data; we receive updates via webhooks or periodic polling.

6.3 GIS Services

We pull geographic data from external services:

- **Mapbox** for base map tiles and geocoding
- **OpenStreetMap** as a fallback and for offline maps
- **National statistical offices** for administrative boundaries (manually imported)

The GIS service caches everything aggressively. A map tile requested once is stored in S3 and served from CloudFront thereafter.

---

7. Security & Compliance

Health data is sensitive. A breach doesn't just violate regulations—it can destroy trust in a health program and put patients at risk. We've designed security as a core concern, not an afterthought.

7.1 Authentication

All authentication flows through AWS Cognito. We're not rolling our own auth—the risks aren't worth the customization benefits.

User roles: - **Admin** — full system access, tenant management - **Clinician** — read/write health data, approve conflicts - **Field Officer** — create records, limited editing - **Viewer** — read-only dashboards and reports

MFA is required for admin and clinician roles. Field officers can use password-only auth (pragmatic concession to device sharing in the field).

7.2 Data Protection

Encryption everywhere: - At rest: AWS KMS-managed keys for all databases and S3 buckets - In transit: TLS 1.3 for all connections (no exceptions) - Field-level: Patient identifiers encrypted at the application layer (defense in depth)

Audit logging: Every data access is logged. Who viewed what record, when, from where. Logs are immutable (CloudWatch Logs with restricted deletion policies) and retained for 7 years.

7.3 Compliance Landscape

We're building for a complex regulatory environment:

Regulation	Applicability	Our Approach
HIPAA	US-funded programs	AWS BAA, encryption, audit logs
GDPR	EU-funded programs, EU citizens	Consent management, right to deletion, data portability
Local data protection laws	Varies by country	Data residency options, configurable retention

The compliance configuration is per-tenant. A ministry of health in Kenya has different requirements than an EU-based research institution.

7.4 What Keeps Us Up at Night

Shared devices: Field workers often share tablets. We mitigate with session timeouts, PIN locks, and no persistent storage of credentials.

Lost devices: When a device is lost, we can remotely wipe the local IndexedDB data on next connection. Until then, local encryption protects the data.

Insider threats: Audit logs and anomaly detection. If someone exports an unusual volume of records, we'll notice.

---

8. Performance & Scalability

The system needs to handle everything from a pilot with 50 users to a national deployment with 100,000.

8.1 Scaling Strategy

Services run in AWS ECS with auto-scaling based on CPU and request count. During normal operation, we run lean. When load spikes (typically Monday mornings when field workers sync their weekend data), capacity increases automatically.

Database scaling: - PostgreSQL: Read replicas for heavy query loads - DynamoDB: On-demand capacity (we pay for what we use) - Redis: Cluster mode for high availability

8.2 Performance Targets

Operation	Target	Why This Matters
API response time (p95)	<500ms	Users on slow connections notice delays
Sync initiation	<2 seconds	Field worker shouldn't wait to start work
Dashboard load	<3 seconds	Decision-makers have limited patience
Conflict resolution	<100ms	Shouldn't interrupt workflow

8.3 Capacity Planning

Initial deployment: 100K users, 1M health records, 10 tenants Target scale: 1M users, 50M health records, 100+ tenants

The jump is significant. We'll need to revisit database sharding and potentially move some workloads to dedicated infrastructure before reaching full scale.

---

9. Observability

When something goes wrong in a distributed system, finding the problem is half the battle.

9.1 The Three Pillars

Logs: Structured JSON to CloudWatch. Every log entry includes request_id, tenant_id, user_id, and service name. We can trace a single user action across all services.

Metrics: CloudWatch metrics for infrastructure (CPU, memory, connections) plus custom business metrics (sync success rate, conflict rate, integration health).

Traces: AWS X-Ray for distributed tracing. When a request is slow, we can see exactly which service call took too long.

9.2 Alerting Philosophy

We alert on symptoms, not causes. "Error rate exceeded 1%" is an alert. "CPU at 80%" is not—high CPU is fine if requests are succeeding.

Critical alerts (pages the on-call engineer): - Error rate >5% for 5 minutes - Sync queue backlog >10,000 items - Database connection pool exhaustion - Integration health check failures

Everything else goes to a dashboard for review during business hours.

---

10. API Design

The full API specification will live in OpenAPI format elsewhere. Here we'll cover the design principles and a few non-obvious decisions.

10.1 Design Principles

Tenant isolation is implicit. The tenant_id comes from the JWT token, never the request body. There's no way for a client to accidentally (or maliciously) access another tenant's data by manipulating a request.

FHIR endpoints follow the spec. For health data, we implement standard FHIR R4 REST endpoints (`/fhir/Patient`, `/fhir/Observation`, etc.). This isn't optional—interoperability depends on it.

Everything else is pragmatic REST. We're not purists. If a resource fits REST conventions, great. If an RPC-style endpoint makes more sense (like `POST /sync/initiate`), we use that.

10.2 Key Endpoints

Area	Key Endpoints	Notes
Auth	`/auth/login`, `/auth/refresh`	Cognito handles the heavy lifting
Health Data	`/fhir/Patient`, `/fhir/Observation`	FHIR R4 compliant
Sync	`/sync/initiate`, `/sync/conflicts`	The offline magic happens here
Events	`/events`, `/events/{id}/interventions`	Community health activities
Integrations	`/integrations/dhis2/sync`	Trigger external syncs

10.3 Versioning

URL-based versioning: `/api/v1/`, `/api/v2/`. We'll maintain backward compatibility for 12 months after a new version releases, with 6 months notice before deprecation.

---

11. Testing & Quality

11.1 Testing Approach

We test at three levels:

Unit tests (Jest): Fast, isolated, run on every commit. Target: 80% coverage on business logic. We don't chase 100%—diminishing returns on boilerplate code.

Integration tests (Supertest + Testcontainers): Verify services work together. Each service has a test suite that runs against real PostgreSQL, Redis, and Kafka instances (via Testcontainers). These run on PR merges.

End-to-end tests (Cypress/Detox): Cover critical user journeys. Not exhaustive—we pick the paths that would hurt most if broken. Login, create patient, sync data, resolve conflict.

11.2 Security Testing

Static analysis (SonarQube) runs on every PR. Dynamic analysis (OWASP ZAP) runs weekly against staging. Annual penetration testing by a third party—non-negotiable for health data systems.

---

12. Operations

12.1 Backup & Recovery

Data	Backup Frequency	Retention	Recovery Target
PostgreSQL	Continuous (PITR)	7 days	<1 hour (RPO)
S3	Versioned	30 days	Instant
DynamoDB	On-demand backups	7 days	<4 hours
Configuration	Git (Infrastructure as Code)	Forever	<1 hour

Disaster recovery: Primary region is eu-west-1 (closest to Africa with full AWS service availability). DR region is af-south-1 (Cape Town) where available, otherwise us-east-1. RTO is 4 hours for full recovery.

12.2 Deployment

Zero-downtime deployments via rolling updates in ECS. Database migrations run before code deploys and must be backward-compatible (no breaking changes until old code is fully drained).

We deploy to staging automatically on merge to main. Production deploys are manual but routine—typically 2-3 times per week.

---

13. Team & Skills

Building this system requires a mix of skills that's hard to find in one person. Here's what the team needs collectively:

13.1 Must-Have Technical Skills

- **Node.js/TypeScript** — our entire backend
- **React/React Native** — frontend and mobile
- **PostgreSQL** — deep knowledge, not just basic queries
- **AWS** — ECS, RDS, DynamoDB, Cognito, at minimum
- **Offline-first patterns** — IndexedDB, Service Workers, sync logic

13.2 Must-Have Domain Knowledge

- **FHIR R4** — at least one team member who really understands it
- **DHIS2** — someone who's integrated with it before
- **Public health workflows** — ideally someone with field experience

13.3 Nice-to-Have

- Kafka operations experience
- PostGIS and geospatial data
- Healthcare compliance (HIPAA/GDPR)
- Experience in low-connectivity environments

13.4 Realistic Team Composition

For initial development: 4-5 engineers (2 backend, 1-2 frontend, 1 full-stack/DevOps), 1 product manager with health domain expertise, part-time compliance advisor.

For operations at scale: Add 1-2 SRE/DevOps, dedicated QA, potentially split into feature teams.

---

14. Risks We're Watching

Every project has risks. Here are the ones we've identified and how we're addressing them.

14.1 Technical Risks

Offline sync complexity (High probability, High impact)

This is our biggest technical risk. Conflict resolution across intermittent connectivity is genuinely hard. Our mitigation: invest heavily in the sync service, build comprehensive test scenarios for conflict cases, and design the UI to handle edge cases gracefully. We're also starting with simpler conflict rules and adding sophistication based on real-world feedback.

Kafka operational overhead (Medium probability, High impact)

Kafka is powerful but operationally demanding. If the team lacks Kafka experience, we risk misconfigurations that cause data loss or performance issues. Mitigation: use AWS MSK (managed Kafka), invest in training, and start with a simple topic structure.

Multi-tenant data leakage (Low probability, Critical impact)

A bug that exposes one tenant's data to another would be catastrophic. Mitigation: defense in depth (schema separation + row-level security), comprehensive automated testing, and security-focused code reviews for any tenant-boundary code.

14.2 Organizational Risks

Domain knowledge gaps

Building health informatics systems without health informatics expertise leads to wrong assumptions. Mitigation: ensure at least one team member has FHIR/DHIS2 experience, and establish relationships with clinical advisors who can review designs.

Scope creep from stakeholders

Health ministries and NGOs always have more requirements than budget allows. Mitigation: define clear MVP scope upfront, document what's out of scope, and resist pressure to add features before core functionality is solid.

14.3 External Dependencies

DHIS2 integration variability

Every DHIS2 instance is configured differently. What works in one country may fail in another. Mitigation: build flexible field mappings, budget significant time for per-tenant integration work, and don't promise turnkey DHIS2 integration.

Connectivity worse than expected

If field conditions are worse than our assumptions (completely offline for weeks, not hours), some features won't work. Mitigation: be honest with stakeholders about offline limitations, design for graceful degradation.

---

15. Implementation Approach

We'll build this in phases, with each phase delivering usable functionality. The goal is to get something in front of real users as early as possible.

Phase 1: Foundation

Focus: Core infrastructure, authentication, basic health data

Build the AWS infrastructure, get Cognito working, implement the User Service and a minimal Health Data Service. At the end of this phase, we can authenticate users and store FHIR resources. Not exciting, but necessary.

What we'll learn: Whether our multi-tenant schema approach works, how the team handles AWS infrastructure, any early performance concerns.

Phase 2: Field Operations

Focus: Event tracking, GIS, notifications, web frontend

This phase delivers the features field workers actually use day-to-day: logging health events, viewing maps, receiving alerts. The web application becomes usable for basic workflows.

What we'll learn: User feedback on workflows, GIS performance with real map data, notification delivery reliability.

Phase 3: Offline & Mobile

Focus: Sync service, mobile app, offline support

The hardest phase. We build the sync infrastructure, implement conflict resolution, and launch the mobile app with offline capability. This is where the offline-first design gets tested against reality.

What we'll learn: Real-world sync conflict patterns, mobile app performance on low-end devices, bandwidth consumption in the field.

Phase 4: Integrations

Focus: DHIS2, EMR connections

Connect to external systems. This phase is highly variable—integration complexity depends entirely on the specific DHIS2 instances and EMR systems we're connecting to.

What we'll learn: Integration patterns that work (and don't), data quality issues from external systems, realistic sync frequencies.

Phase 5: Polish & Scale

Focus: Analytics, performance optimization, security hardening

Build dashboards for decision-makers, optimize performance based on real usage data, complete security audit and compliance validation.

What we'll learn: What metrics actually matter to users, performance bottlenecks at scale, any compliance gaps.

Effort & Team Sizing

Total effort is roughly 50-65 person-months, depending on integration complexity. A team of 4-5 engineers can deliver the core system, with additional support needed for domain expertise and compliance.

We're not providing specific timeline estimates—those depend too heavily on team composition, stakeholder availability, and external system readiness. The phases above are ordered by dependency, not calendar time.

---

16. Assumptions & Dependencies

This design assumes certain things are true. If they're not, we'll need to adjust.

What We're Assuming About Infrastructure

- AWS account with appropriate service limits (some services need limit increases)
- Budget for initial cloud setup ($1,500-3,000 one-time for dev/staging environments)
- Development team has reliable internet access

What We're Assuming About External Systems

- DHIS2 instances are accessible via API (not all are)
- EMR systems either support FHIR or can export data in a usable format
- Mapbox or similar GIS services are available and affordable for the region

What We're Assuming About Users

- Field workers have smartphones capable of running a React Native app
- Some connectivity exists—we're designing for intermittent, not zero
- Users are willing to resolve sync conflicts when necessary

What We're Assuming About the Team

- Core competency in Node.js and React (can learn the rest)
- Access to at least one person with FHIR/health informatics experience
- Willingness to invest in Kafka and AWS training

---

17. What's Not in This Document

A few things are intentionally left for later:

Detailed UI/UX design: This document covers backend architecture. Frontend design decisions (component libraries, state management patterns, accessibility standards) will be documented separately.

Operational runbooks: Incident response procedures, on-call rotations, and detailed troubleshooting guides come after the system is built.

Cost optimization: Initial builds prioritize correctness over cost efficiency. We'll optimize AWS spending once we understand real usage patterns.

Detailed API specifications: OpenAPI specs will be maintained alongside the codebase, not in this document.

---

18. Final Thoughts

This system is ambitious. Offline-first, multi-tenant health informatics with complex external integrations—any one of those would be challenging. We're doing all three.

The architecture reflects pragmatic choices. We've picked boring technologies where possible (PostgreSQL, React, REST APIs) and accepted complexity only where the requirements demand it (Kafka for reliable event streaming, DynamoDB for sync burst handling).

The biggest risk isn't technical—it's building the wrong thing. Field workers, clinicians, and ministry officials have different needs. The phased approach exists partly to get feedback early and adjust before we've built too much.

If we execute well, this system can meaningfully improve how public health programs operate in challenging environments. That's worth the complexity.