Architecture Decision Record (ADR) Register

YourHealthNS – Solution Architecture

Prepared by: Suresh Kumar Balasubramanian

November 2025

Contents

Executive Summary

This Architecture Decision Record (ADR) Register consolidates key design decisions for the YourHealthNS platform. It captures nine approved architectural choices spanning frontend strategy, backend runtime, resilience patterns, CI/CD automation, and observability. Each ADR documents the problem context, considered alternatives, the final decision, and anticipated consequences. Together, these decisions establish a scalable, maintainable, and resilient foundation aligned with SOLID principles, Separation of Concerns, and the organization's security and compliance objectives.

This register forms part of the YourHealthNS Solution Architecture submission and should be read in conjunction with the main architecture document (v1.1).

ADR-001: Dual Backend for Frontend Pattern for Web and Mobile

Context

Both the React Native mobile app and Next.js web portal requires optimized payloads and specific backend integrations.

Considered Options

- **Option 1 – Single Unified BFF**: Shared logic for both clients; simple to deploy, but hard to maintain; poor separation of concerns.
- **Option 2 – Dual Channel-Specific BFFs (Web + Mobile)**: Each BFF focuses on channel-optimized APIs, payloads, and caching; cleaner boundaries and independent scalability.
- **Option 3 – Thin API Gateway + Client Logic**: Pushes orchestration logic to the frontend, complicating error handling and increasing coupling.

Decision Outcome

Chosen option: *Dual Channel-Specific BFFs implemented in .NET 8 (minimal api).*

Rationale:
Adheres to **Separation of Concerns** and **SOLID (Single Responsibility)** principles by isolating channel-specific logic within dedicated BFFs. This approach improves maintainability, enables independent deployments, and aligns with the overall orchestrator pattern. It also enhances user experience by allowing each channel to tailor interactions and data representations to its audience for example, a lab result can be summarized in a compact, mobile-friendly format in the app while being displayed with richer details in the web portal. In essence, the BFF acts as a user-experience orchestration layer that sits in front of the business logic services.

Consequences:
Two deployables to maintain, but simpler CI/CD, clearer ownership, and predictable scaling. Enables channel-specific evolution (e.g., mobile-first caching, web analytics integration)

ADR-002: Adoption of Orchestrator Pattern for Multi-Step Workflows

Context

Complex workflows (e.g., consent approval + appointment booking) span multiple services and data stores. A purely event-choreographed system led to traceability and compensation challenges during failure recovery.

Considered Options

- **Option 1 – Choreography via Events (Kafka):** High autonomy but low observability; difficult rollback.
- **Option 2 – Central Orchestrator per Business Workflow:** Deterministic flow, compensation handling, centralized audit trail.
- **Option 3 – Hybrid (Orchestrator for critical flows, events for async ops).**

Decision Outcome

Chosen option: *Hybrid approach—Orchestrator for critical workflows, Kafka for async/non-critical flows.*

Rationale:
Improves resilience and troubleshooting. The orchestrator manages synchronous transactions with compensation logic and leverages Kafka for decoupled downstream processing. For example, in an appointment booking flow, the orchestrator synchronously verifies consent and reserves the appointment slot, ensuring transactional integrity; once confirmed, it asynchronously publishes an event to Kafka for downstream actions such as sending notifications or updating audit logs without blocking the user response.

Consequences:
Increased coupling in orchestrator service but improved reliability and auditability. Requires coordination of state transitions and OTel tracing for correlation.

ADR-003: React Native for Mobile Development

Context

The platform requires iOS and Android apps with shared features and branding. Native development (Swift/Kotlin) would double effort and specialized skillsets for engineers.

Considered Options

- **Option 1 – Native Swift/Kotlin:** Best UX performance but separate codebases and duplicated effort.
- **Option 2 – React Native:** Cross-platform, single TypeScript codebase, OIDC-ready SDKs, shared UI components.
- **Option 3 – Flutter:** Strong cross-platform option but less TypeScript/React ecosystem alignment with web team.

Decision Outcome

Chosen option: *React Native.*

Rationale:
Delivers cross-platform consistency and faster feature parity by reusing the shared TypeScript design system and BFF integration models. Using a single codebase improves maintainability and allows engineers cross-trained in React and .NET ecosystems to contribute across both web and mobile channels. While React Native may not achieve the same frame-rate or animation fluidity as fully native Swift/Kotlin apps, this trade-off is acceptable since YourHealthNS is a healthcare application not a graphics-intensive or gaming experience where maintainability, security, and delivery velocity take precedence over minor UX performance differences.

Consequences:
Minor performance trade-offs but significant productivity and consistency gains. Requires strong testing discipline for device fragmentation.

ADR-004: .NET 8 as Unified Backend and BFF Runtime

Context

Backend services, BFFs, and adapters require a runtime that supports async I/O, strong type safety, observability, and built-in security features

Considered Options

- **Option 1 – Node.js + TypeScript:** Common for BFFs, lightweight, but lacks strong typing and resilience tooling.
- **Option 2 – .NET 8 (ASP.NET Core):** High concurrency, mature ecosystem (Polly, Serilog, OTel), long-term LTS support.
- **Option 3 – Java Spring Boot:** Enterprise-grade, but heavier memory footprint and longer startup times.

Decision Outcome

Chosen option: *.NET 8 for all backend services and BFFs.*

Rationale:
Provides a unified CI/CD pipeline, consistent resilience patterns using *Polly*, and vendor-neutral observability through *OpenTelemetry*. Ensures security alignment across services via *OIDC* and *mTLS*. The .NET 8 platform adheres to SOLID and Separation of Concerns principles, promoting modularity and maintainability. As a mature, long-term-supported framework, it offers standardized patterns such as dependency injection, strong typing, and async I/O for high-performance workloads.

Adopting the provincial Identity Provider (IDP) with OIDC federation and PKCE ensures compliance with existing government identity standards and simplifies end-user onboarding. Implementing an in-house OAuth or Identity Server was evaluated but rejected due to the high operational complexity, ongoing compliance obligations (HIPAA, PIPEDA), and maintenance burden around token storage, key rotation, and audit logging.

Consequences:
Requires .NET expertise but delivers predictable performance and maintainability. Enables shared NuGet packages for telemetry, validation, and DTOs. However, if the platform were to evolve towards hosting its own identity infrastructure, .NET 8 provides mature support for OAuth 2.0, OIDC, and integrations with external providers such as Auth0, enabling a controlled and standards-compliant implementation when governance maturity allows.

ADR-005: ACID-Compliant Relational Database (PostgreSQL)

Context

Health data (consents, appointments, lab results) requires strict consistency, integrity, and traceability. NoSQL or eventual consistency stores could violate data correctness under concurrency.

Considered Options

- **Option 1 – PostgreSQL (ACID RDBMS):** Strong consistency, mature ecosystem, JSONB support.
- **Option 2 – MongoDB / CosmosDB:** Flexible schema, easier horizontal scaling, but eventual consistency.
- **Option 3 – Hybrid (Postgres + NoSQL cache):** Postgres for system of record, Redis/Kafka for async access.

Decision Outcome

Chosen option: *PostgreSQL as the primary system of record.*

Rationale:
Ensures ACID compliance and transactional safety for regulated data, with Redis and Kafka providing low-latency caching and async processing. Aligns with healthcare-grade audit and compliance needs. Ensures ACID compliance and transactional safety for regulated data while explicitly prioritizing Consistency and Availability over Partition Tolerance as per the CAP theorem, aligning with healthcare system requirements. Eventual consistency patterns are isolated to non-critical flows via Kafka and Redis. This balance guarantees data correctness for PHI transactions while maintaining horizontal scalability for analytical workloads.

Consequences:
Slightly higher write latency than NoSQL but guarantees data correctness. Requires schema governance and PITR backup strategy.

ADR-006: CI/CD Strategy Using Jenkins and FluxCD (GitOps)

Context

The solution requires automated, auditable, and repeatable deployments across multiple environments (Dev, QA, Prod) for microservices running on OpenShift. Manual deployment through pipelines leads to configuration drift and weak release traceability.

Considered Options

- **Option 1 – Jenkins for CI + FluxCD (GitOps) for CD:** Jenkins builds and scans artifacts; FluxCD syncs manifests from Git for immutable deployments.
- **Option 2 – Jenkins X (end-to-end CI/CD):** Single integrated tool, but limited maturity on OpenShift.
- **Option 3 – ArgoCD + Tekton Pipelines:** Modern alternative; better visualization but requires more setup and governance maturity.

Decision Outcome

Chosen option: *Jenkins for CI and FluxCD for GitOps-based CD.*

Rationale:
Combines proven CI automation (Jenkins) with GitOps declarative deployment (FluxCD), aligning with OpenShift's operator model. Provides traceability and rollback through Git history, with build and deploy stages handled by distinct systems. Helm charts define environment-specific parameters such as replicas, resource limits, and secrets using declarative syntax, ensuring full Infrastructure-as-Code (IaC) consistency across clusters. Integrates seamlessly with SAST scans, and automated test gates to enforce compliance and deployment quality before production promotion.

Consequences:
Requires GitOps discipline and environment-specific manifest repos. Delivers full auditability, rollback safety, and environment consistency. Reduces human error and drift between test and prod.

ADR-007: Enforcement of Separation of Concerns and SOLID Principles

Context

Microservices risk becoming tightly coupled over time as teams evolve and feature pressure grows. Without clear boundaries and design discipline, the platform could face regression risk, difficult onboarding, and maintainability issues.

Considered Options

- **Option 1 – Enforce SoC + SOLID at code and architectural levels.** Each layer (UI, BFF, Domain, Data, Integration) has a single purpose and minimal cross-dependency.
- **Option 2 – Loose structure with developer discretion.** Faster delivery early but creates brittle interdependencies and future redesign effort.
- **Option 3 – Over-engineered abstraction everywhere.** Clean design but slower iteration and over-complex for smaller services.

Decision Outcome

Chosen option: *System-wide enforcement of SoC and SOLID principles.*

Rationale:
Adhering to SOLID principlaes improves modularity, testability, and ease of scaling. Separation of Concerns ensures each layer handles its bounded context:

- UI layer (React Native / Next.js) → presentation only
- BFFs (.NET) → orchestration, security, and caching
- Domain APIs (.NET) → core business rules
- Integration layer (FHIR, HL7, Kafka) → external communications
- Data layer (PostgreSQL, Redis) → persistence and consistency

CI/CD pipelines include static architecture validation, automated dependency scans, and code quality gates to ensure SOLID conformance. Unit and integration tests are implemented using xUnit/NUnit with Moq for dependency isolation and FluentAssertions for expressive validation, ensuring each component can be independently verified. This disciplined testing framework enforces architectural contracts early and strengthens confidence during refactoring or scaling.

Consequences:
Slightly more upfront governance effort but ensures long-term maintainability, reduced coupling, and predictable change propagation. Improves developer onboarding and system resilience through layered modularity.

ADR-008: Adoption of CDN for Static Assets and API Edge Caching

Context

The Next.js web portal and React Native app rely heavily on static assets (JavaScript bundles, CSS, images) and API calls to the BFFs. Without a Content Delivery Network (CDN), all users—including remote or rural healthcare providers—would experience higher latency due to single-region hosting.

Considered Options

- **Option 1 –** Use CDN (Akamai) for static and API caching: Global edge POPs, SSL offload, DDoS protection.
- **Option 2 –** Direct access to OpenShift ingress routers: Simpler but centralized; higher latency.
- **Option 3 –** Hybrid approach: CDN for static assets only; APIs handled directly.

Decision Outcome

Chosen option: Global CDN fronting both static web assets and selected BFF API routes.

Rationale:
CDN provides low-latency, edge-accelerated content delivery, SSL offloading, and edge caching for static resources and frequently accessed APIs. Improves Time to First Byte (TTFB) and First Contentful Paint (FCP). Reduces traffic on ingress routers, improving backend scalability and cost efficiency.

Consequences:
Adds CDN configuration and cache invalidation complexity, mitigated by CI/CD integration and cache-busting strategies. Delivers ~30–50% latency reduction, improved user experience, and greater resilience under load or regional outages.

ADR-009: Unified Resilience and Observability Strategy Using Polly and OpenTelemetry

Context

As the system scales across multiple .NET microservices, BFFs, and integrations (FHIR, HL7, Kafka), maintaining **consistent resilience**, **traceability**, and **SLO observability** becomes critical.
Without a unified approach, each service could implement its own retry or logging logic, leading to inconsistent behavior, debugging difficulty, and vendor lock-in with proprietary monitoring agents.

Considered Options

- **Option 1 – Individual libraries per service:** Each team chooses its own resilience and logging packages. Fast to start but inconsistent and untraceable across services.
- **Option 2 – Centralized Resilience & Observability Layer using Polly +.NET Resilience + OpenTelemetry:** Shared libraries for circuit-breakers, retries, and fallback logic, plus vendor-neutral tracing.
- **Option 3 – Managed APM vendor SDKs (e.g., Datadog, New Relic):** Rich UI but high licensing cost and vendor lock-in; limited on-prem OpenShift support.

Decision Outcome

Chosen option: *Unified Polly + OpenTelemetry strategy implemented across all .NET services.*

Rationale:
Polly provides circuit-breaker, retry, and fallback patterns embedded directly in .NET 8, ensuring predictable resilience at every network call (HTTP, DB, Kafka).
OpenTelemetry offers vendor-neutral, open-standard instrumentation for traces, metrics, and logs across .NET, Node.js, Kafka, and Istio enabling full request correlation (trace-ID propagation) Together, they:

- Deliver consistent fault-handling and automatic retries with exponential back-off. Enable end-to-end distributed tracing for debugging orchestrated workflows.
- Support SLO enforcement (P95 < 500 ms, 99.99 % availability).
- Maintain vendor neutrality—switch backends without code changes.

Consequences:
Slightly higher telemetry volume and CPU overhead; mitigated via 10 % sampling and retention policies.Requires developer discipline to instrument spans and apply standardized resilience policies. Results in uniform error semantics, faster root-cause detection, improved uptime, and observability compliance across the entire stack.