SportiQue - XRPL Technical Design Document

1 SportiQue - XRPL Technical Design Document

1.1 Blockchain-Based Health Data Marketplace Platform

Project: SportiQue Healthcare Data Marketplace **Version**: 1.0 **Date**: October 2025 **Team**: SportiQue Development Team **Repository**: https://github.com/sportique/xrpl-core

---

1.2 Table of Contents

1.3 1. Executive Summary

1.3.1 1.1 Project Overview

SportiQue is a **blockchain-based health data marketplace** built on the **XRP Ledger (XRPL)**. The platform enables:

- **Healthcare providers** (nursing homes, hospitals) to manage patient data
- **Individual users** to track health activities and earn rewards
- **Enterprise buyers** (pharma, research) to purchase anonymized health data
- **Transparent, trustless data trading** using XRPL's native features

1.3.2 1.2 Core Innovation: XRPL Integration

Our platform leverages three key XRPL primitives:

1. **NFTokens** - Data access control and ownership proof
2. **Escrow** - Trustless data trading without platform intermediary
3. **Multi-Purpose Tokens (MPT)** - Reward point system with batching

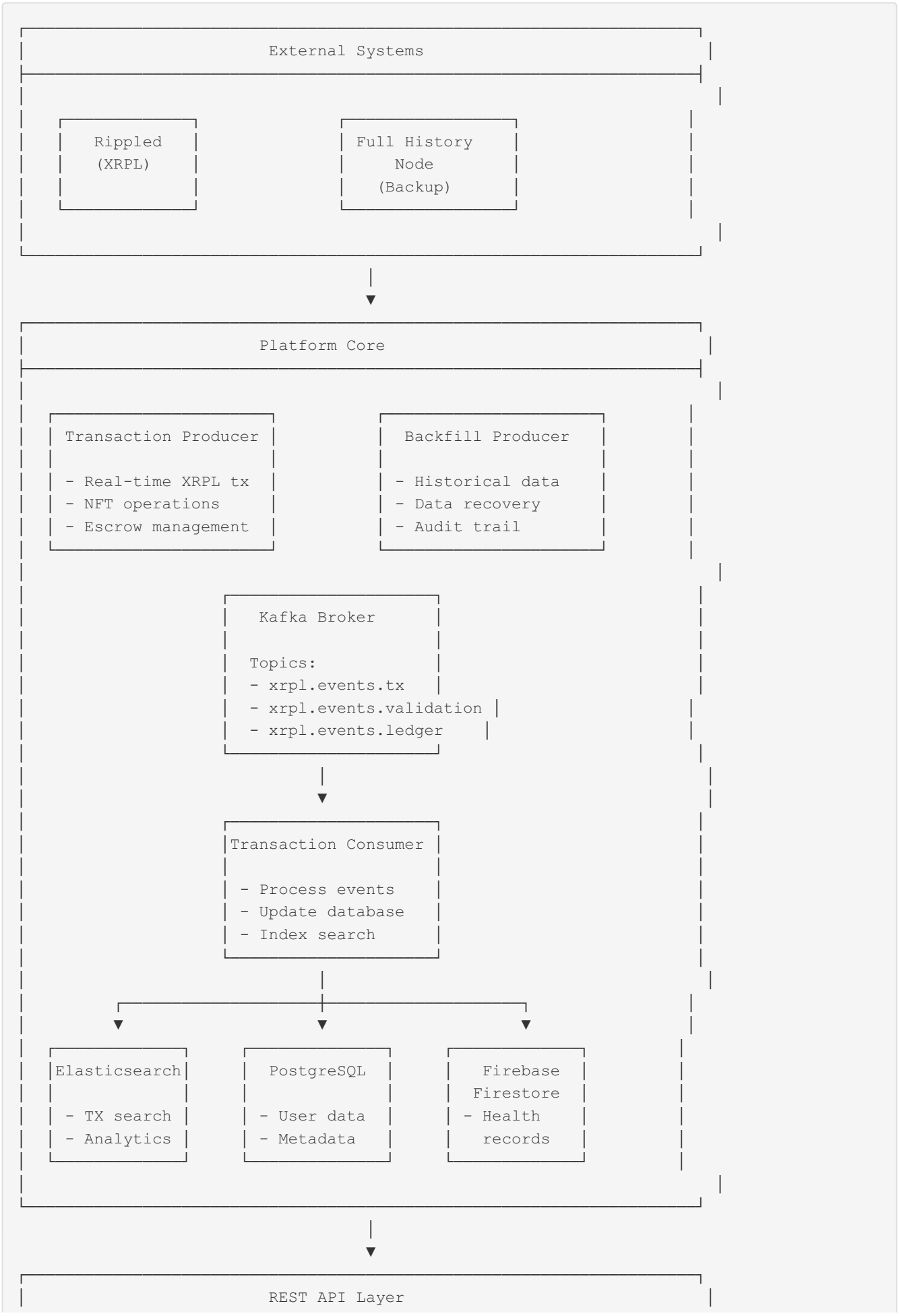
1.3.3 1.3 Key Metrics

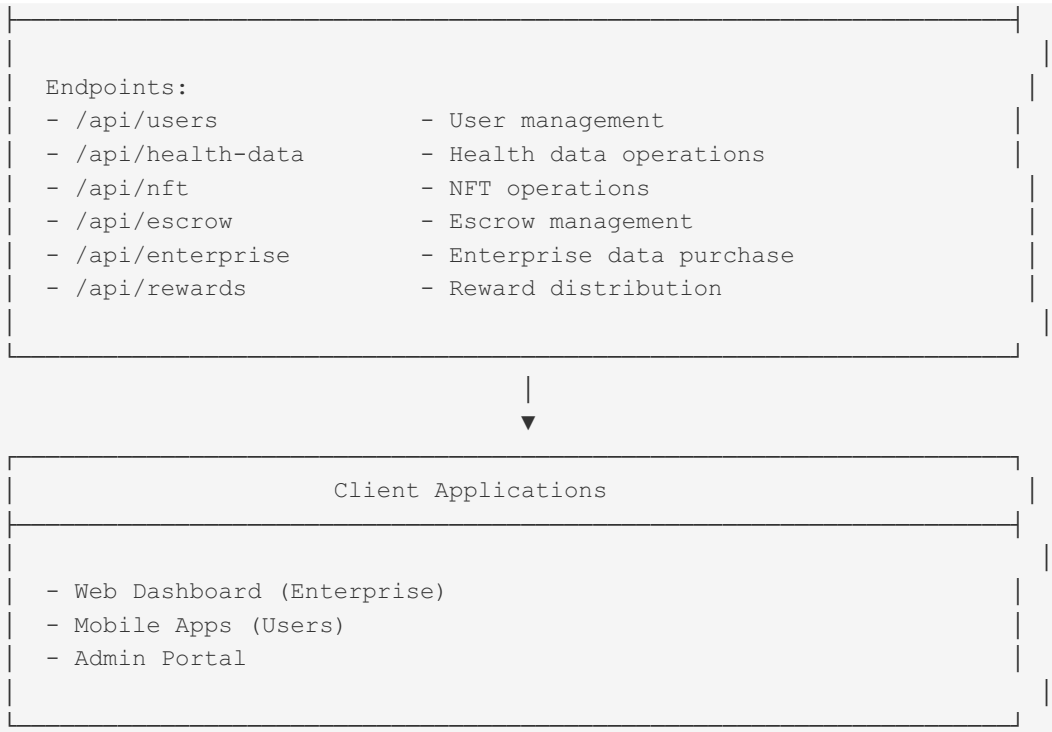
- **Target Users**: 350 nursing home residents + 5,000 individual users
- **Transaction Cost**: $0.00002 per XRPL transaction (99.9% cheaper than alternatives)
- **Data Security**: Zero plaintext health data on blockchain (privacy-first)
- **Transparency**: 100% of data sales traceable on public ledger

---

1.4 2. System Architecture Overview

1.4.1 2.1 High-Level Architecture Diagram


External Systems
Rippled (XRPL)
Full History Node (Backup)
Platform Core
Transaction Producer
- Real-time XRPL tx
- NFT operations
- Escrow management
Backfill Producer
- Historical data
- Data recovery
- Audit trail
Kafka Broker
Topics:
- xrpl.events.tx
- xrpl.events.validation
- xrpl.events.ledger
Transaction Consumer
- Process events
- Update database
- Index search
Elasticsearch
- TX search
- Analytics
PostgreSQL
- User data
- Metadata
Firebase Firestore
- Health records
REST API Layer



1.4.2 2.2 Architecture Principles

1.4.2.1 2.1 Event-Driven Architecture

Our platform uses **Kafka** as the central message broker to decouple XRPL events from application logic:

- **Transaction Producer**: Listens to XRPL websocket, publishes transactions to Kafka
- **Backfill Producer**: Syncs historical data from full-history nodes
- **Consumer**: Processes events and updates application databases

Benefits: - Scalability: Add consumers without changing producers - Reliability: Kafka persists messages (no data loss) - Real-time: Sub-second event processing

1.4.2.2 2.2 Privacy-First Design

Zero health data on blockchain: - XRPL stores only: NFT IDs, transaction hashes, wallet addresses - Health data encrypted in Firebase Firestore - NFT URI points to encrypted data location - Decryption keys managed separately (AWS KMS)

1.4.2.3 2.3 Trustless Trading

Escrow eliminates platform risk: - Enterprise funds locked in XRPL escrow (not platform wallet) - Auto-release when user transfers NFT - Auto-refund if timeout (no NFT transfer) - Platform cannot steal funds

---

1.5 3. XRPL Integration Points

1.5.1 3.1 XRPL Primitives Used

XRPL Feature	Use Case	Implementation File
NFTokenMint	Data access rights	`core/nft.ts`
NFTokenTransfer	Data sales	`transactions/NFTAccessControl.ts`
NFTokenBurn	Access revocation	`core/nft.ts`
EscrowCreate	Lock purchase funds	`core/escrow.ts`
EscrowFinish	Release funds to seller	`platform/xrpl/XrplEscrow.ts`
EscrowCancel	Refund buyer	`platform/xrpl/XrplEscrow.ts`
MPTokenIssuanceCreate	Issue reward points	`core/mpt.ts`
Payment (MPT)	Distribute rewards	`transactions/DataReward.ts`
TrustSet	Enable MPT receiving	`core/wallet.ts`

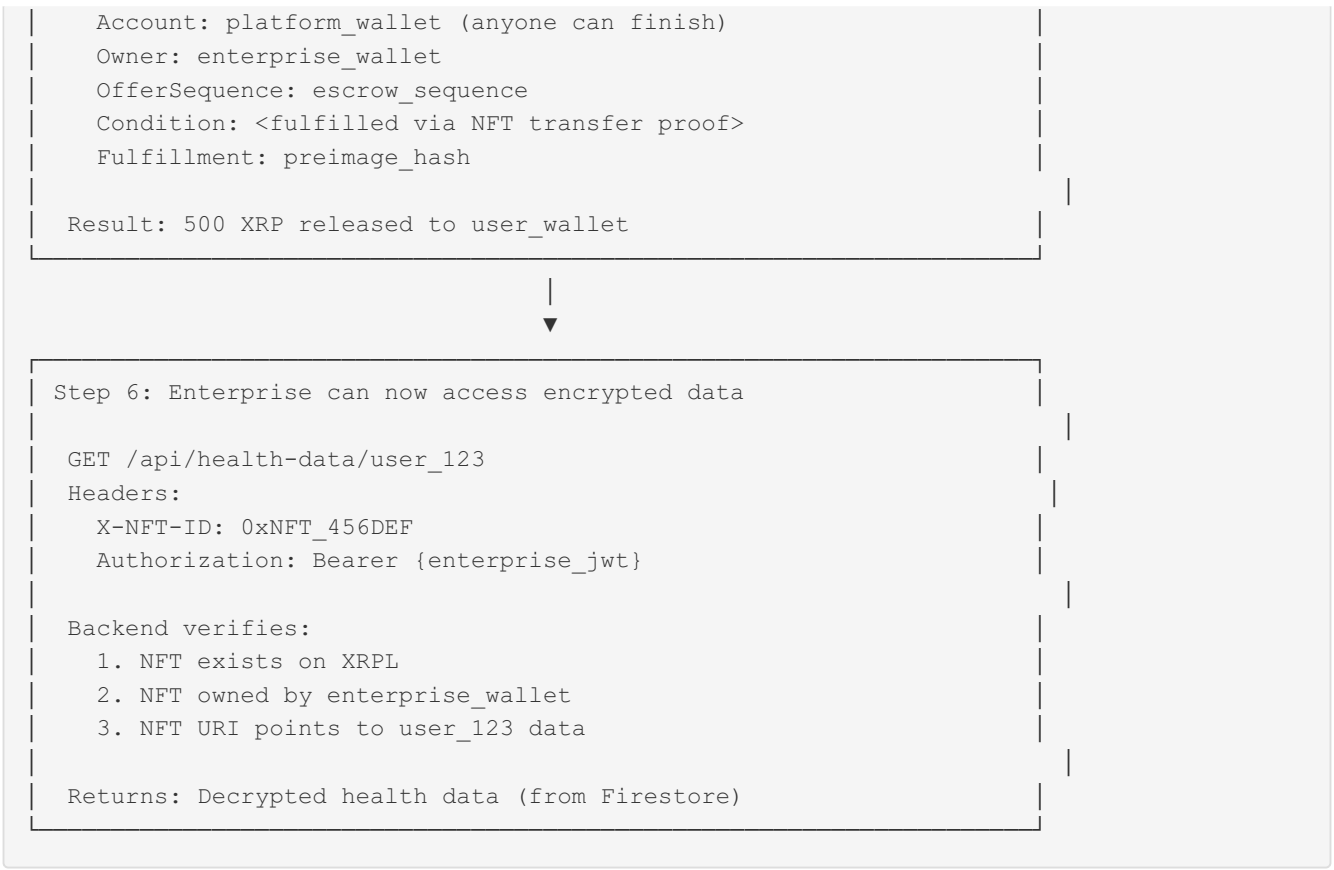
1.5.2 3.2 XRPL Network Configuration

```
// core/config.ts

export const XRPL_CONFIG = {
  // Primary connection (mainnet)
  network: 'wss://xrplcluster.com',

  // Fallback connections
  fallbackNetworks: [
    'wss://s2.ripple.com',
    'wss://xrpl.ws'
  ],

  // Full-history node for backfilling
  fullHistoryNode: 'https://xrplcluster.com',

  // Network settings
  networkId: 0, // Mainnet

  // Transaction settings
  fee: '12', // 12 drops = 0.000012 XRP
  maxLedgerOffset: 20,

  // MPT Configuration
  mptIssuer: process.env.MPT_ISSUER_ADDRESS,
  mptTaxon: 0,

  // NFT Configuration
  nftTaxon: 1, // Health data category
  transferFee: 0, // No royalties
};
```

1.5.3 3.3 Transaction Flow Examples

1.5.3.1 3.3.1 NFT-Based Data Purchase Flow


Step 1: Enterprise browses data marketplace
(Anonymous, aggregated samples only)
Step 2: Enterprise initiates purchase
POST /api/enterprise/purchase
{
"userId": "user_123",
"dataType": "health_record",
"priceXRP": "500"
}
Step 3: Platform creates escrow (enterprise → user)
XRPL Transaction:
TransactionType: "EscrowCreate"
Account: enterprise_wallet
Destination: user_wallet
Amount: "500000000" (500 XRP)
FinishAfter: rippleTimeNow() + 604800 (7 days)
Condition: crypto_condition(nft_transfer)
Result: Escrow ID = 0xESCROW_789ABC
Step 4: User approves and transfers NFT
XRPL Transaction:
TransactionType: "NFTokenCreateOffer"
Account: user_wallet
NFTokenID: "0xNFT_456DEF"
Destination: enterprise_wallet
Amount: "0" (already paid via escrow)
Then:
TransactionType: "NFTokenAcceptOffer"
Account: enterprise_wallet
Step 5: Platform monitors and finishes escrow
Kafka Consumer detects NFTokenAcceptOffer
Verifies: NFT now owned by enterprise_wallet
XRPL Transaction:
TransactionType: "EscrowFinish"


Account: platform_wallet (anyone can finish)
Owner: enterprise_wallet
OfferSequence: escrow_sequence
Condition: <fulfilled via NFT transfer proof>
Fulfillment: preimage_hash
Result: 500 XRP released to user_wallet
Step 6: Enterprise can now access encrypted data
GET /api/health-data/user_123
Headers:
X-NFT-ID: 0xNFT_456DEF
Authorization: Bearer {enterprise_jwt}
Backend verifies:
1. NFT exists on XRPL
2. NFT owned by enterprise_wallet
3. NFT URI points to user_123 data
Returns: Decrypted health data (from Firestore)

1.5.3.2 3.3.2 MPT Reward Distribution Flow

1.6 4. Code Organization

1.6.1 4.1 Project Structure

```
sportique-xrpl-core/
│
├── core/                          # XRPL primitive wrappers
│   ├── config.ts                  # Network configuration
│   ├── wallet.ts                  # Wallet management
│   ├── nft.ts                     # NFToken operations
│   ├── escrow.ts                  # Escrow operations
│   ├── payment.ts                 # XRP/MPT payments
│   ├── mpt.ts                     # Multi-Purpose Tokens
│   ├── dataPool.ts                # Data pool utilities
│   ├── firebase.ts                # Firebase integration
│   ├── types.ts                   # Core type definitions
│   ├── utils.ts                   # Helper functions
│   └── index.ts                   # Module exports
│
├── platform/                      # Business logic layer
│   │
│   ├── types/                     # Platform type definitions
│   │   ├── user.ts                # User types
│   │   ├── health.ts              # Health data types
│   │   ├── enterprise.ts          # Enterprise types
│   │   └── index.ts               # Type exports
│   │
│   ├── user/                      # User domain
│   │   ├── UserAuth.ts            # User authentication
│   │   └── HealthDataManager.ts   # Health data CRUD
│   │
│   ├── enterprise/                # Enterprise domain
│   │   ├── EnterpriseAuth.ts      # Enterprise auth
│   │   ├── DataPoolManager.ts     # Data marketplace
│   │   └── SubscriptionManager.ts # Subscription handling
│   │
│   ├── platform/                  # Platform operations
│   │   ├── DataPoolManager.ts     # Pool administration
│   │   ├── RewardCalculator.ts    # Reward algorithms
│   │   └── SystemConfig.ts        # Platform config
│   │
│   ├── xrpl/                      # XRPL integration
│   │   ├── XrplWallet.ts          # Wallet service
│   │   ├── XrplEscrow.ts          # Escrow service
│   │   └── NftGenerator.ts        # NFT metadata
│   │
│   └── data/                      # Data operations
│       └── QualityEvaluator.ts    # Data quality scoring
│
├── transactions/                  # High-level transaction flows
│   ├── NFTAccessControl.ts        # NFT-based access
│   ├── SubscriptionNFT.ts         # Subscription NFTs
│   ├── SubscriptionEscrow.ts      # Subscription payments
│   ├── PoolNFT.ts                 # Data pool NFTs
│   ├── DataPoolParticipation.ts   # Pool joining
│   ├── DataReward.ts              # Reward distribution
│   ├── index.ts                   # Transaction exports
│   │
│   └── scripts/                   # Demo & utilities
│       ├── runDemo.ts             # Full demo flow
│       ├── seedFirestore.ts       # Database seeding
```

```
│       ├── dataPoolScenario.ts    # Pool demo
│       └── mptScenario.ts         # MPT demo
│
├── tests/                         # Test suites
│   ├── unit/                      # Unit tests
│   ├── integration/               # Integration tests
│   └── e2e/                       # End-to-end tests
│
├── package.json                   # Dependencies
├── tsconfig.json                  # TypeScript config
├── .env.example                   # Environment template
└── README.md                      # Documentation
```

1.6.2 4.2 Module Responsibilities

1.6.2.1 4.2.1 Core Layer (`core/`)

Purpose: Direct XRPL interaction wrappers

Key Modules:

- `wallet.ts`: Wallet generation, seed management, signing

```typescript
export async function createWallet(seed?: string): Promise<Wallet>
export async function fundWallet(wallet: Wallet, amount: string): Promise<void>
export async function getBalance(address: string): Promise<string>
```

- `nft.ts`: NFToken minting, transferring, burning

```typescript
export async function mintNFT(
  wallet: Wallet,
  uri: string,
  transferFee?: number
): Promise<string> // Returns NFTokenID

export async function transferNFT(
  wallet: Wallet,
  nftId: string,
  destination: string
): Promise<void>

export async function burnNFT(
  wallet: Wallet,
  nftId: string
): Promise<void>
```

- `escrow.ts`: Escrow creation, finishing, canceling

```
export async function createEscrow(
  wallet: Wallet,
  destination: string,
  amount: string,
  finishAfter: number,
  condition?: string
): Promise<string> // Returns escrow sequence

export async function finishEscrow(
  wallet: Wallet,
  owner: string,
  sequence: number,
  fulfillment?: string
): Promise<void>
```

- `mpt.ts`: Multi-Purpose Token operations

```
export async function createMPToken(
  wallet: Wallet,
  maxAmount?: string
): Promise<string> // Returns MPT ID

export async function sendMPT(
  wallet: Wallet,
  destination: string,
  amount: string,
  mptId: string
): Promise<void>
```

1.6.2.2 4.2.2 Platform Layer (`platform/`)

Purpose: Business logic and domain models

Key Modules:

- `platform/user/HealthDataManager.ts`: Health data operations

```
class HealthDataManager {
  async uploadHealthData(
    userId: string,
    data: HealthRecord
  ): Promise<string> // Returns data ID

  async getHealthData(
    userId: string,
    nftId: string // Proves access rights
  ): Promise<HealthRecord>

  async generateDataNFT(
    userId: string,
    dataId: string
  ): Promise<string> // Returns NFTokenID
}
```

- `platform/enterprise/DataPoolManager.ts`: Data marketplace

```
class DataPoolManager {
  async createDataPool(
    criteria: DataCriteria,
    pricePerRecord: string
  ): Promise<string> // Returns pool ID

  async purchaseDataAccess(
    enterpriseId: string,
    poolId: string
  ): Promise<PurchaseResult> // Creates escrow + NFT transfer
}
```

- `platform/platform/RewardCalculator.ts`: Reward algorithms

```
class RewardCalculator {
  calculateActivityReward(activity: Activity): number
  calculateDataQualityBonus(data: HealthRecord): number
  calculateDailyRewards(userId: string): RewardSummary
}
```

1.6.2.3 4.2.3 Transactions Layer (`transactions/`)

Purpose: Complex multi-step transaction flows

Key Modules:

- `NFTAccessControl.ts`: NFT-based authorization

```
class NFTAccessControl {
  async verifyAccess(
    nftId: string,
    requesterAddress: string,
    resourceId: string
  ): Promise<boolean>

  async grantAccess(
    ownerId: string,
    resourceId: string,
    expiresAt: Date
  ): Promise<string> // Mints access NFT
}
```

- `DataReward.ts`: Batch reward distribution

```
class DataReward {
  async distributeRewards(
    rewards: Map<string, number>
  ): Promise<BatchResult>

  async claimReward(
    userId: string,
    rewardId: string
  ): Promise<void>
}
```

1.7 5. Technology Stack

1.7.1 5.1 Core Technologies

Layer	Technology	Version	Purpose
Blockchain	XRP Ledger	Mainnet	Trustless transactions, NFTs, escrow
Backend Runtime	Node.js	20.x	JavaScript runtime
Language	TypeScript	5.3+	Type safety, developer experience
Message Broker	Apache Kafka	3.5+	Event streaming
Database	PostgreSQL	15+	Relational data (users, transactions)
NoSQL	Firebase Firestore	10.x	Health records (encrypted)
Search	Elasticsearch	8.x	Transaction search, analytics
Cache	Redis	7.x	Session management, batching queues

1.7.2 5.2 Key Libraries & Frameworks

1.7.2.1 5.2.1 XRPL Integration

```
{
  "dependencies": {
    "xrpl": "^2.14.0"
  }
}
```

Features Used: - **Client API**: Websocket connection to rippled - **NFTokenMint**: Create NFTs with metadata URIs - **EscrowCreate/Finish**: Conditional payments - **Payment**: XRP and MPT transfers - **Account queries**: NFT ownership, balances

Example Usage:

```
import { Client, Wallet, NFTokenMint } from 'xrpl';

const client = new Client('wss://xrplcluster.com');
await client.connect();

const wallet = Wallet.fromSeed('sXXXXXXXXXXXXXXXXXXXXXXXXXXXX');

const mintTx: NFTokenMint = {
  TransactionType: 'NFTokenMint',
  Account: wallet.address,
  URI: '68747470733A2F2F73706F7274697175652E696F2F6E66742F313233', // hex-encoded
  Flags: 8, // tfTransferable
  NFTokenTaxon: 1
};

const result = await client.submitAndWait(mintTx, { wallet });
console.log('NFT Minted:', result.result.meta.nftoken_id);
```

1.7.2.2 5.2.2 Database & Storage

```
{
  "dependencies": {
    "firebase": "^10.7.0",
    "firebase-admin": "^11.11.0"
  }
}
```

Firebase Usage: - **Firestore**: Encrypted health records storage - **Authentication**: User identity (linked to XRPL wallet) - **Security Rules**: NFT-based access control

Example Firestore Structure:

```
// Collection: health_records
{
  "user_123": {
    "records": [
      {
        "id": "record_456",
        "type": "blood_pressure",
        "value": {
          "systolic": 120,
          "diastolic": 80
        },
        "timestamp": "2025-10-20T10:00:00Z",
        "encrypted": true,
        "nftId": "0xNFT_789ABC", // Access control
        "encryptionKey": "kms://key_123" // AWS KMS reference
      }
    ]
  }
}
```

1.7.2.3 5.2.3 Cryptography

```
{
  "dependencies": {
    "crypto-js": "^4.2.0"
  }
}
```

Purpose: Encrypt health data before storage

Example:

```
import CryptoJS from 'crypto-js';

function encryptHealthData(
  data: HealthRecord,
  key: string
): string {
  return CryptoJS.AES.encrypt(
    JSON.stringify(data),
    key
  ).toString();
}

function decryptHealthData(
  encrypted: string,
  key: string
): HealthRecord {
  const bytes = CryptoJS.AES.decrypt(encrypted, key);
  const decrypted = bytes.toString(CryptoJS.enc.Utf8);
  return JSON.parse(decrypted);
}
```

1.7.3 5.3 Development Tools

Tool	Purpose
TypeScript	Static typing, IntelliSense
ts-node	Run TypeScript without compiling
Jest	Unit & integration testing
ESLint	Code linting
Prettier	Code formatting
Husky	Git hooks for pre-commit checks

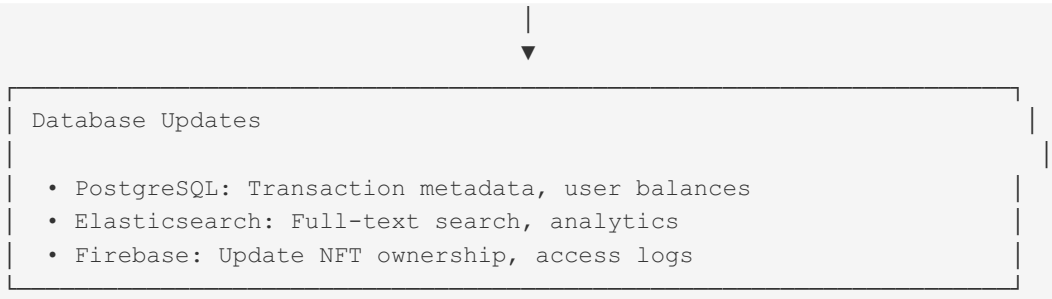
1.7.4 5.4 Infrastructure (Deployment)

Service	Provider	Purpose
Compute	AWS EC2 / ECS	Docker containers
Load Balancer	AWS ALB	Traffic distribution
CDN	CloudFront	Static asset delivery
Monitoring	Datadog	Logs, metrics, APM
CI/CD	GitHub Actions	Automated testing & deployment
Secrets	AWS Secrets Manager	Environment variables, keys

1.8 6. Data Flow Architecture

1.8.1 6.1 Real-Time Transaction Processing


XRPL Network
• New transaction validated
• Ledger closes every 3-5 seconds
Websocket subscription
Transaction Producer (Node.js)
const client = new Client('wss://xrplcluster.com');
client.on('transaction', (tx) => {
if (isRelevantTransaction(tx)) {
kafka.produce('xrpl.events.tx', tx);
}
});
Publish to Kafka
Kafka Topics
• xrpl.events.tx (partitioned by account address)
• xrpl.events.validation
• xrpl.events.ledger
Consumer reads
Transaction Consumer (Node.js)
kafka.consume('xrpl.events.tx', async (msg) => {
const tx = JSON.parse(msg.value);
// Route by transaction type
switch (tx.TransactionType) {
case 'NFTokenMint':
await handleNFTMint(tx);
break;
case 'NFTokenAcceptOffer':
await handleNFTTransfer(tx);
break;
case 'EscrowCreate':
await handleEscrowCreate(tx);
break;
case 'EscrowFinish':
await handleEscrowFinish(tx);
break;
case 'Payment':
await handlePayment(tx);
break;
}
});


Database Updates
• PostgreSQL: Transaction metadata, user balances
• Elasticsearch: Full-text search, analytics
• Firebase: Update NFT ownership, access logs

1.8.2 6.2 Data Purchase Flow (Detailed)


Enterprise Browser
1. Browse anonymized data samples
GET /api/data-pools
REST API
Response: [
{
"poolId": "pool_123",
"criteria": { "ageMin": 65, "condition": "diabetes" },
"recordCount": 150,
"pricePerRecord": "10", // XRP
"sampleData": { /* anonymized */ }
}
]
2. Purchase pool access
POST /api/enterprise/purchase-pool
{ "poolId": "pool_123" }
Backend: Data Pool Service
1. Calculate total price: 150 records × 10 XRP = 1500 XRP
2. Generate pool access NFT metadata
3. Create escrow: enterprise → platform
4. Await NFT transfer confirmation
3. XRPL: EscrowCreate
XRPL Ledger
Escrow {
account: enterprise_wallet,
destination: platform_wallet,
amount: 1500 XRP,
finishAfter: now + 7 days,
condition: sha256(pool_nft_transfer)
}
4. Backend mints pool access NFT
XRPL: NFTokenMint
NFT {
uri: "https://sportique.io/nft/pool_123_access",
taxon: 1,


flags: tfTransferable
}
NFT ID: 0xNFT_POOL_456
5. Transfer NFT to enterprise
XRPL: NFTokenCreateOffer + NFTokenAcceptOffer
Offer {
nftId: 0xNFT_POOL_456,
destination: enterprise_wallet,
amount: 0 // Already paid via escrow
}
6. Kafka event: NFTokenAcceptOffer detected
Transaction Consumer
async function handleNFTTransfer(tx) {
const nftId = tx.NFTokenID;
const newOwner = tx.Account;
// Check if this NFT is linked to an escrow
const escrow = await db.findEscrowByNFT(nftId);
if (escrow) {
// Finish escrow (release funds)
await xrpl.finishEscrow(escrow.id, nftId);
// Grant data access
await db.grantAccess(newOwner, escrow.poolId);
}
}
7. XRPL: EscrowFinish
XRPL Ledger
EscrowFinish {
fulfillment: preimage(pool_nft_transfer),
result: 1500 XRP → platform_wallet
}
8. Enterprise can now access data
GET /api/data-pools/pool_123/records
Header: X-NFT-ID: 0xNFT_POOL_456

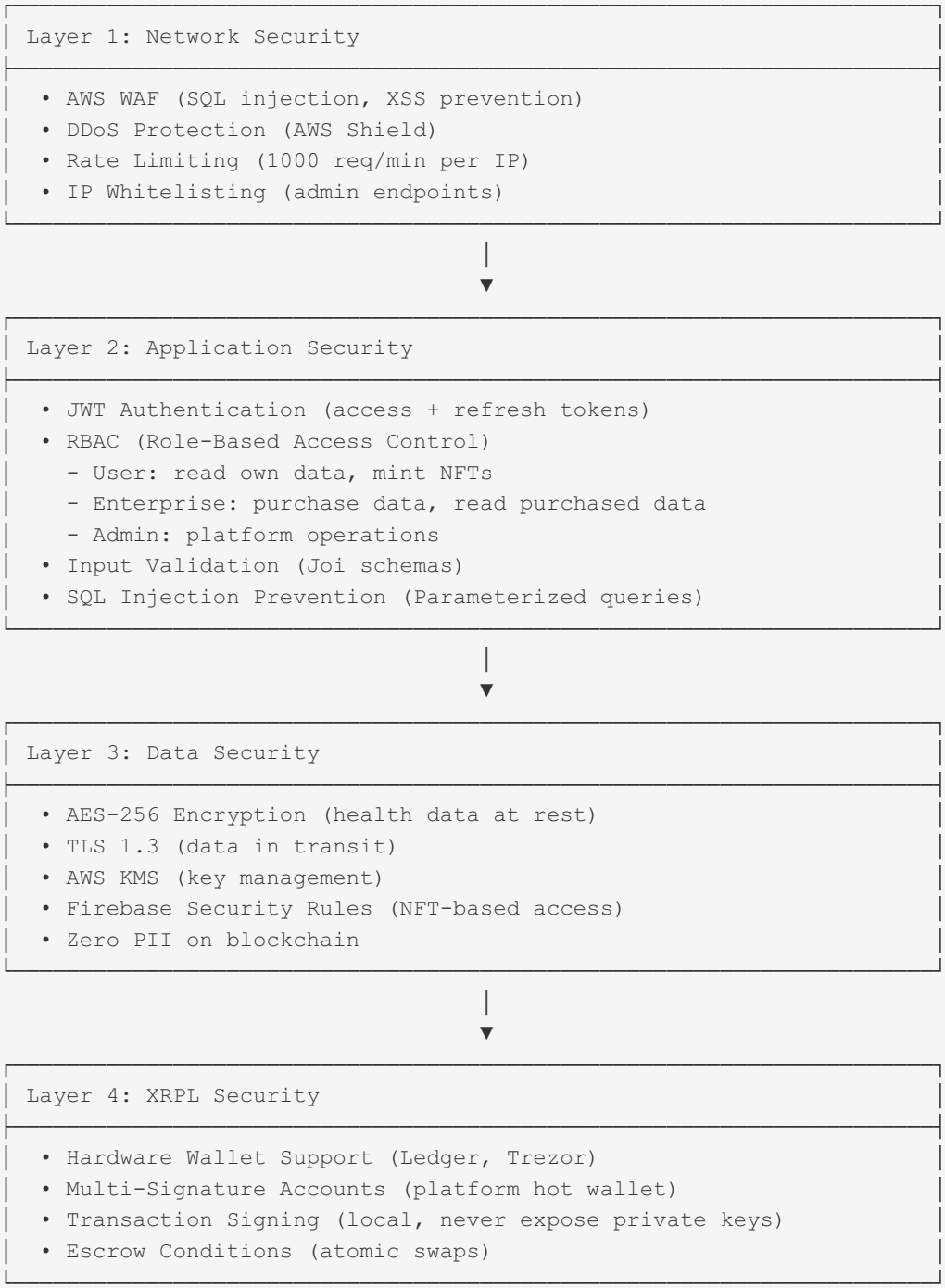
```
│  Middleware verifies:                                          │
│    - NFT exists on XRPL                                        │
│    - NFT owned by requester                                    │
│    - NFT URI matches pool_123                                  │
│                                                                │
│  Response: Decrypted health records (150 records)              │
└────────────────────────────────────────────────────────────────┘
```

1.8.3 6.3 Reward Distribution Flow

```
Daily Cron Job (00:00 UTC)
         │
         ▼
┌────────────────────────────────────────────────────────────────┐
│  Reward Aggregation Service                                    │
│                                                                │
│  1. Query all user activities (yesterday)                      │
│     SELECT userId, SUM(points) FROM activities                 │
│     WHERE date = YESTERDAY                                     │
│     GROUP BY userId;                                           │
│                                                                │
│  2. Calculate rewards:                                         │
│     user_1 → 120 points → 120 MPT                              │
│     user_2 → 95 points → 95 MPT                                │
│     ...                                                        │
│     user_350 → 110 points → 110 MPT                            │
│                                                                │
│  3. Prepare batch MPT transfers                                │
└────────────────────────────────────────────────────────────────┘
         │
         ▼
┌────────────────────────────────────────────────────────────────┐
│  MPT Batch Transfer (XRPL)                                     │
│                                                                │
│  for (const user of users) {                                   │
│    await xrpl.sendMPT({                                        │
│      from: platform_issuer,                                    │
│      to: user.wallet,                                          │
│      amount: user.points,                                      │
│      mptId: SPORTIQUE_MPT_ID                                   │
│    });                                                         │
│  }                                                             │
│                                                                │
│  Cost: 350 tx × 0.000012 XRP = 0.0042 XRP (~$0.002)            │
└────────────────────────────────────────────────────────────────┘
         │
         ▼
┌────────────────────────────────────────────────────────────────┐
│  User Notification                                             │
│                                                                │
│  Send push notification:                                       │
│  "You earned 120 SportiQue points yesterday!                   │
│   View balance: https://app.sportique.io/rewards"              │
└────────────────────────────────────────────────────────────────┘
```

1.9 7. Security Architecture

1.9.1 7.1 Multi-Layer Security Model



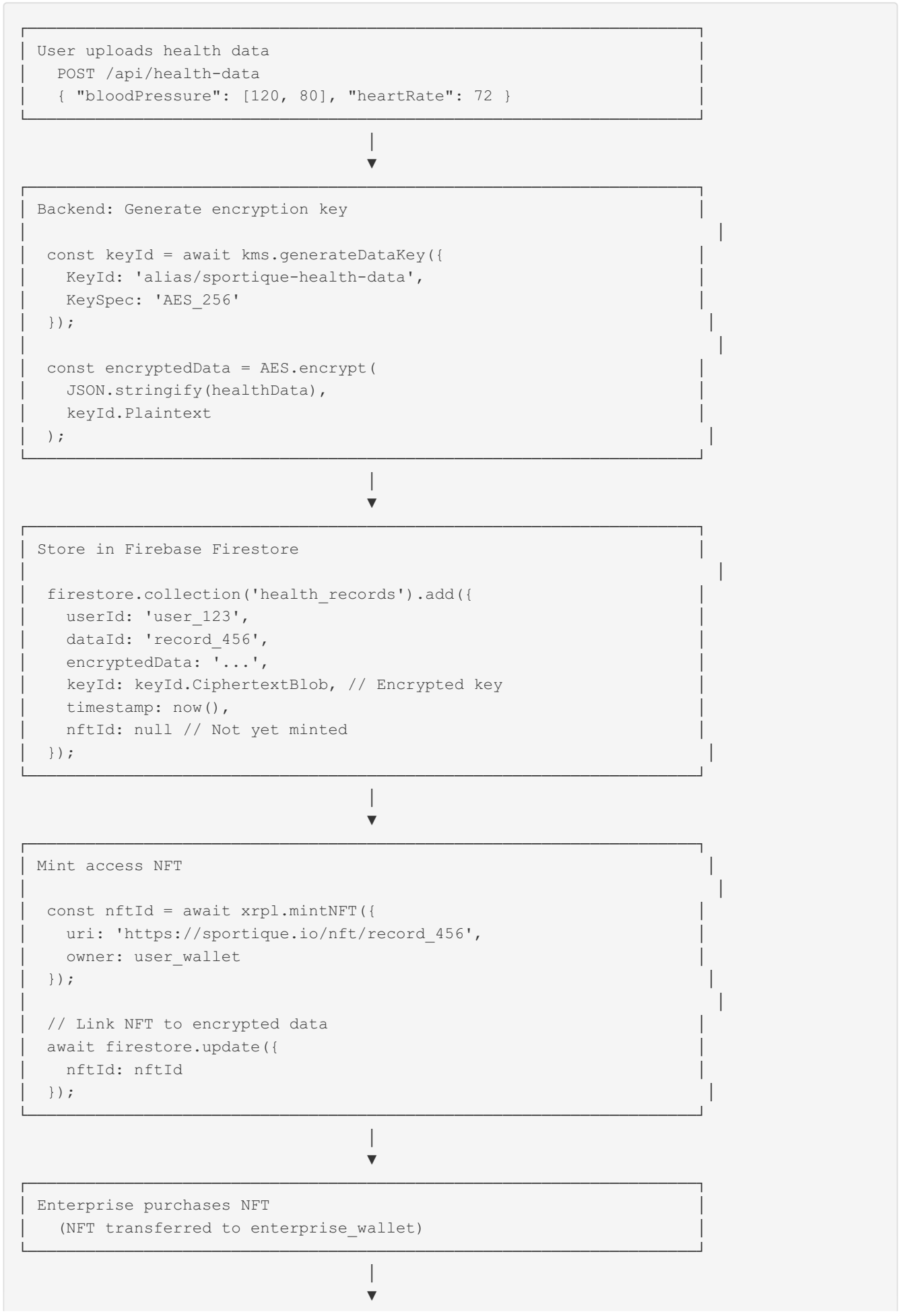
1.9.2 7.2 NFT-Based Access Control

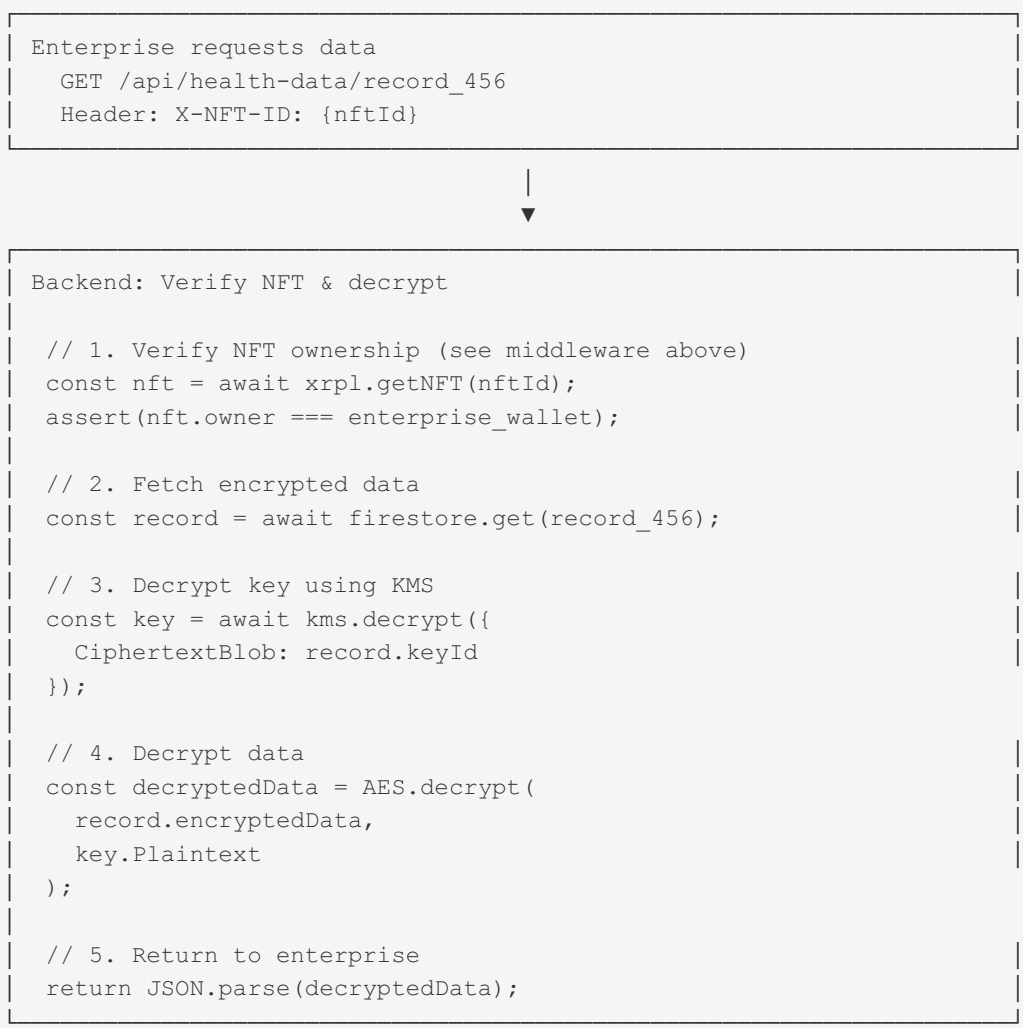
Problem: Traditional databases allow anyone with credentials to query data

Solution: NFT ownership gates database queries

```
// Middleware: Verify NFT ownership before database query

export async function verifyNFTAccess(
  req: Request,
  res: Response,
  next: NextFunction
) {
  const nftId = req.headers['x-nft-id'];
  const userId = req.params.userId;

  // 1. Query XRPL for NFT ownership
  const nft = await xrpl.getNFT(nftId);

  if (!nft) {
    return res.status(404).json({ error: 'NFT not found' });
  }

  // 2. Verify requester owns the NFT
  if (nft.owner !== req.user.walletAddress) {
    return res.status(403).json({ error: 'NFT not owned by requester' });
  }

  // 3. Parse NFT URI to check resource access
  const metadata = await fetchNFTMetadata(nft.URI);

  if (metadata.userId !== userId) {
    return res.status(403).json({ error: 'NFT not valid for this user' });
  }

  // 4. Check expiration
  if (metadata.expiresAt < Date.now()) {
    return res.status(403).json({ error: 'NFT expired' });
  }

  // Access granted
  req.nftMetadata = metadata;
  next();
}
```

1.9.3 7.3 Encryption Flow

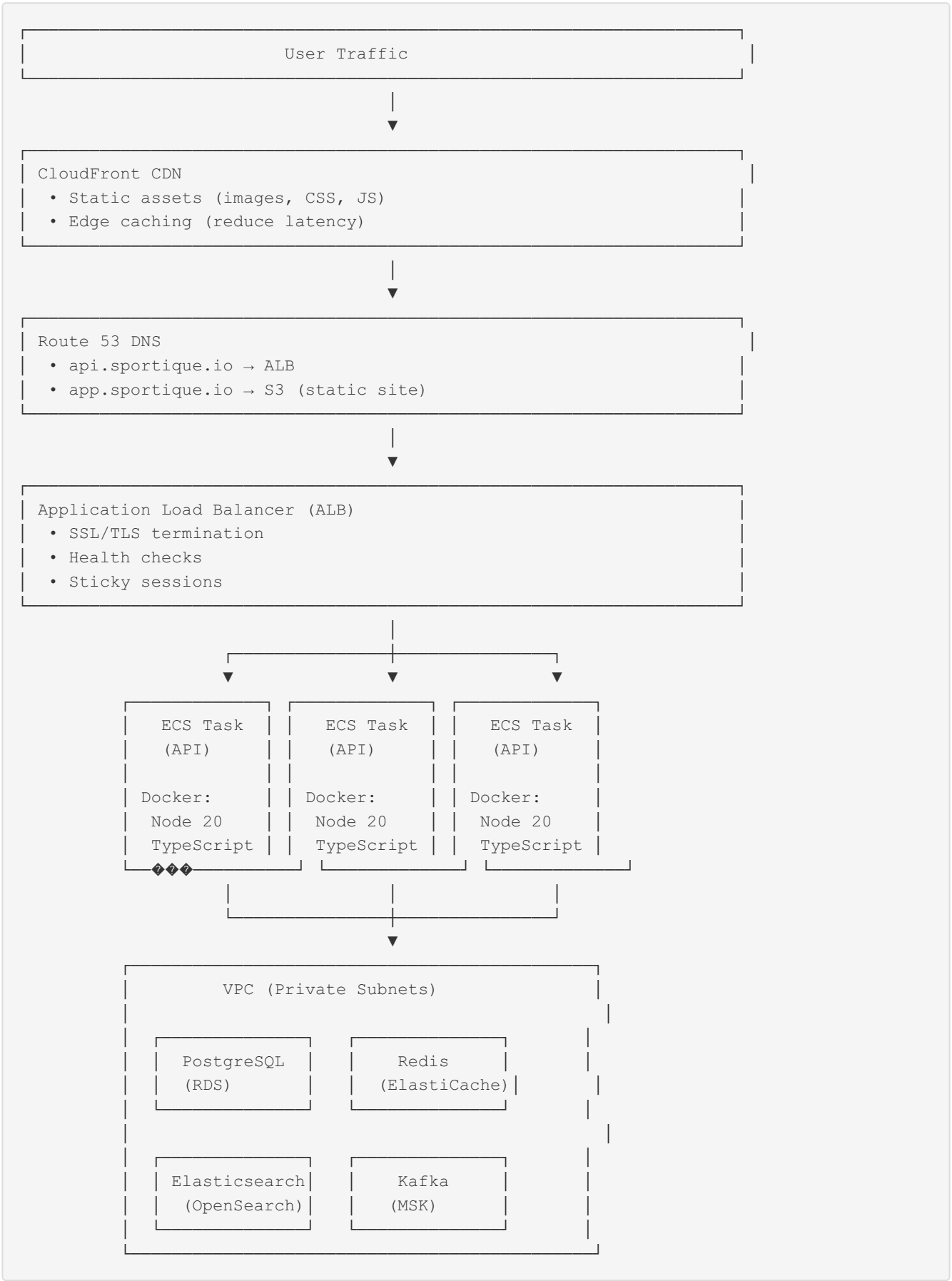

User uploads health data
POST /api/health-data
{ "bloodPressure": [120, 80], "heartRate": 72 }
Backend: Generate encryption key
const keyId = await kms.generateDataKey({
KeyId: 'alias/sportique-health-data',
KeySpec: 'AES_256'
});
const encryptedData = AES.encrypt(
JSON.stringify(healthData),
keyId.Plaintext
);
Store in Firebase Firestore
firestore.collection('health_records').add({
userId: 'user_123',
dataId: 'record_456',
encryptedData: '...',
keyId: keyId.CiphertextBlob, // Encrypted key
timestamp: now(),
nftId: null // Not yet minted
});
Mint access NFT
const nftId = await xrpl.mintNFT({
uri: 'https://sportique.io/nft/record_456',
owner: user_wallet
});
// Link NFT to encrypted data
await firestore.update({
nftId: nftId
});
Enterprise purchases NFT
(NFT transferred to enterprise_wallet)



Key Security Properties: - ✅ Health data never stored in plaintext - ✅ Encryption keys managed by AWS KMS (FIPS 140-2 Level 3) - ✅ NFT ownership verified on-chain before decryption - ✅ Zero PII on XRPL blockchain - ✅ Immutable audit trail (all NFT transfers logged)

1.10 8. Deployment Architecture

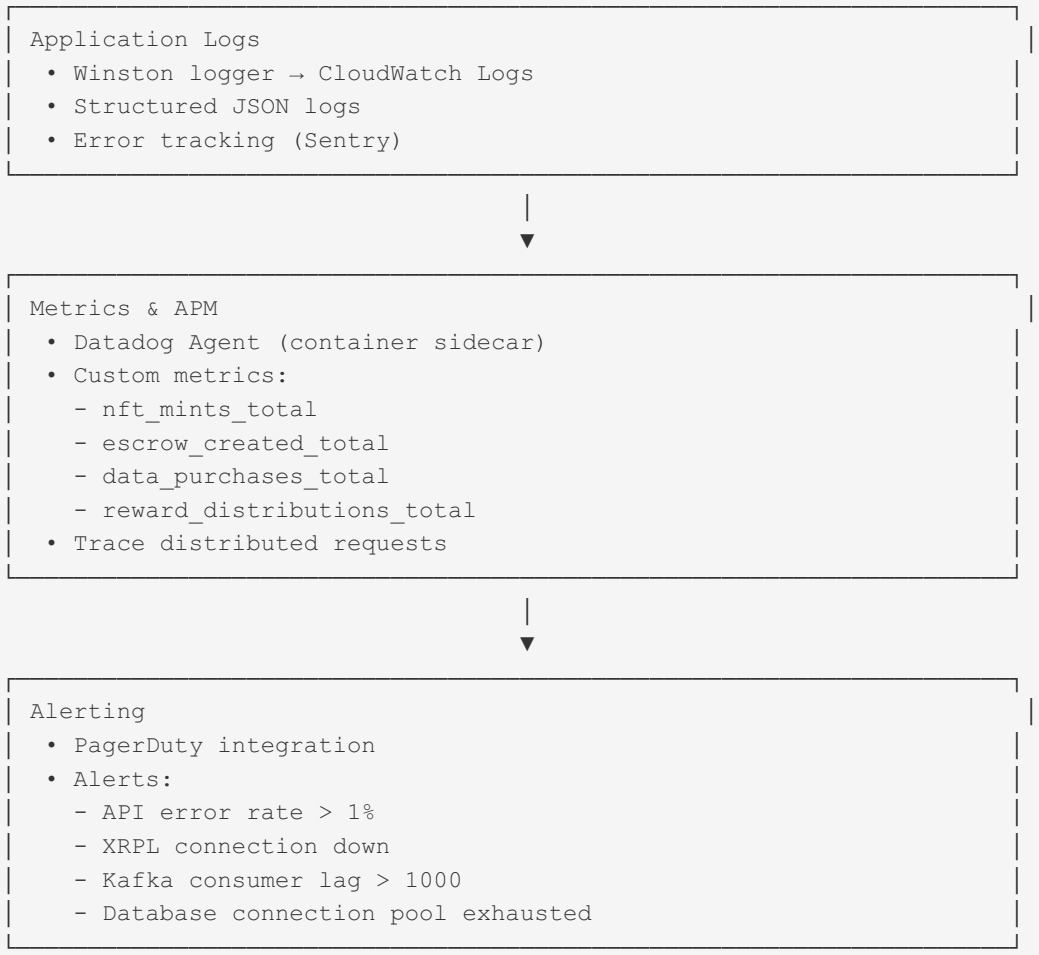
1.10.1 8.1 Production Infrastructure


User Traffic
CloudFront CDN
• Static assets (images, CSS, JS)
• Edge caching (reduce latency)
Route 53 DNS
• api.sportique.io → ALB
• app.sportique.io → S3 (static site)
Application Load Balancer (ALB)
• SSL/TLS termination
• Health checks
• Sticky sessions
ECS Task (API)
Docker: Node 20 TypeScript
ECS Task (API)
Docker: Node 20 TypeScript
ECS Task (API)
Docker: Node 20 TypeScript
VPC (Private Subnets)
PostgreSQL (RDS)
Redis (ElastiCache)
Elasticsearch (OpenSearch)
Kafka (MSK)

1.10.2 8.2 CI/CD Pipeline

```
GitHub Repository (main branch)
         │
         │ git push
         ▼
GitHub Actions Workflow
         │
         ├─ Step 1: Lint & Type Check
         │    npm run lint
         │    tsc --noEmit
         │
         ├─ Step 2: Run Tests
         │    npm run test
         │    npm run test:integration
         │
         ├─ Step 3: Build Docker Image
         │    docker build -t sportique-api:${SHA}
         │
         ├─ Step 4: Push to ECR
         │    aws ecr push sportique-api:${SHA}
         │
         └─ Step 5: Deploy to ECS
              aws ecs update-service \
                --cluster sportique-prod \
                --service api \
                --force-new-deployment
```

1.10.3 8.3 Monitoring & Observability



1.10.4 8.4 Disaster Recovery

Scenario	Recovery Strategy	RTO	RPO
API Server Down	Auto-scaling + health checks (ECS)	5 min	0
Database Failure	RDS Multi-AZ failover	2 min	0
XRPL Node Down	Fallback to secondary node	10 sec	0
Kafka Broker Down	Replicated partitions (3x)	1 min	0
Region Outage	Multi-region failover (manual)	30 min	5 min

1.11 Appendix A: XRPL Transaction Examples

1.11.1 A.1 NFT Mint Transaction

```
{
  "TransactionType": "NFTokenMint",
  "Account": "rUserWalletAddress123...",
  "URI": "68747470733A2F2F73706F7274697175652E696F2F6E66742F726563",
  "Flags": 8,
  "TransferFee": 0,
  "NFTokenTaxon": 1,
  "Fee": "12",
  "Sequence": 1234567,
  "LastLedgerSequence": 1234587
}
```

1.11.2 A.2 Escrow Create Transaction

```
{
  "TransactionType": "EscrowCreate",
  "Account": "rEnterpriseWallet456...",
  "Destination": "rUserWalletAddress123...",
  "Amount": "500000000",
  "FinishAfter": 750000000,
  "Condition":
        "A0258020E3B0C44298FC1C149AFBF4C8996FB92427AE41E4649B934CA495991B7852B855810100",

  "Fee": "12",
  "Sequence": 7890123
}
```

1.11.3 A.3 MPT Payment Transaction

```
{
  "TransactionType": "Payment",
  "Account": "rPlatformMPTIssuer789...",
  "Destination": "rUserWalletAddress123...",
  "Amount": {
    "currency": "MPT",
    "value": "120",
    "issuer": "rPlatformMPTIssuer789..."
  },
  "Fee": "12",
  "Memos": [
    {
      "Memo": {
        "MemoType": "726577617264",
        "MemoData": "4461696C792061637469766974792072657617264"
      }
    }
  ]
}
```

1.12 Appendix B: API Endpoints Reference

1.12.1 B.1 User Endpoints

Method	Endpoint	Description
POST	`/api/users/register`	Create user account
POST	`/api/users/login`	Authenticate user
GET	`/api/users/me`	Get current user
POST	`/api/health-data`	Upload health data
GET	`/api/health-data/:id`	Get health data (NFT required)
POST	`/api/nft/mint`	Mint data access NFT

1.12.2 B.2 Enterprise Endpoints

Method	Endpoint	Description
POST	`/api/enterprise/register`	Register enterprise
GET	`/api/data-pools`	Browse data pools
POST	`/api/enterprise/purchase-pool`	Purchase pool access
GET	`/api/enterprise/purchases`	List purchases
GET	`/api/data-pools/:id/records`	Get purchased records

1.12.3 B.3 Admin Endpoints

Method	Endpoint	Description
GET	`/api/admin/stats`	Platform statistics
POST	`/api/admin/rewards/distribute`	Trigger reward distribution
GET	`/api/admin/escrows`	List all escrows
POST	`/api/admin/escrows/:id/finish`	Manually finish escrow

1.13 Appendix C: Database Schema

1.13.1 C.1 PostgreSQL Tables

```sql
-- Users table
CREATE TABLE users (
  id UUID PRIMARY KEY,
  wallet_address VARCHAR(34) UNIQUE NOT NULL,
  email VARCHAR(255),
  role VARCHAR(20) CHECK (role IN ('user', 'enterprise', 'admin')),
  created_at TIMESTAMP DEFAULT NOW()
);

-- NFTs table
CREATE TABLE nfts (
  id UUID PRIMARY KEY,
  nft_id VARCHAR(64) UNIQUE NOT NULL,
  owner_address VARCHAR(34) NOT NULL,
  uri TEXT,
  metadata JSONB,
  created_at TIMESTAMP DEFAULT NOW(),
  transferred_at TIMESTAMP
);

-- Escrows table
CREATE TABLE escrows (
  id UUID PRIMARY KEY,
  escrow_sequence INTEGER NOT NULL,
  creator_address VARCHAR(34) NOT NULL,
  destination_address VARCHAR(34) NOT NULL,
  amount VARCHAR(20) NOT NULL,
  finish_after INTEGER,
  condition TEXT,
  status VARCHAR(20) CHECK (status IN ('active', 'finished', 'cancelled')),
  created_at TIMESTAMP DEFAULT NOW()
);

-- Transactions table
CREATE TABLE transactions (
  id UUID PRIMARY KEY,
  tx_hash VARCHAR(64) UNIQUE NOT NULL,
  tx_type VARCHAR(50) NOT NULL,
  account VARCHAR(34) NOT NULL,
  ledger_index INTEGER NOT NULL,
  timestamp TIMESTAMP NOT NULL,
  metadata JSONB,
  indexed_at TIMESTAMP DEFAULT NOW()
);
```

1.14 Summary

This technical design document demonstrates SportiQue's comprehensive integration with the XRP Ledger:

1. **System Architecture**: Event-driven architecture using Kafka for scalable XRPL event processing
2. **XRPL Integration**: Native use of NFTokens, Escrow, and MPT for trustless data trading
3. **Code Organization**: Modular TypeScript codebase with clear separation of concerns
4. **Technology Stack**: Modern, production-ready stack (Node.js, TypeScript, PostgreSQL, Firebase, Kafka)
5. **Security**: Multi-layer security with NFT-based access control and end-to-end encryption
6. **Deployment**: Cloud-native infrastructure on AWS with CI/CD automation

The platform uniquely leverages XRPL's features to create a **privacy-preserving, transparent, and trustless healthcare data marketplace** that benefits users, enterprises, and society.