HABIBI Protocol - Technical Design Document

NASA-STD-2202-93 Compliant

Document Version 1.0

Document Control

- **Project**: HABIBI Protocol Web Platform
- **Document ID**: HABIBI-TDD-001
- **Date**: November 2024
- **Classification**: Technical Implementation
- **Baseline**: Functional Requirements v1.0

---

1. INTRODUCTION

1.1 Purpose

This document provides comprehensive technical design specifications for implementing the HABIBI Protocol web platform, detailing system architecture, component interactions, data structures, algorithms, and implementation guidelines.

1.2 Scope

The design encompasses a multi-chain DeFi platform with real-time data processing, complex graph computations, and high-performance user interfaces supporting 10,000+ concurrent users.

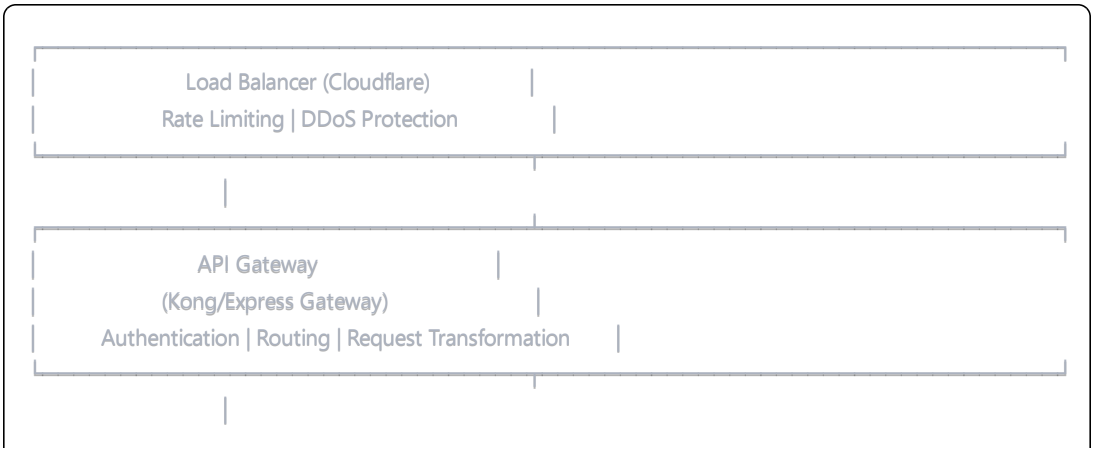
1.3 Design Methodology

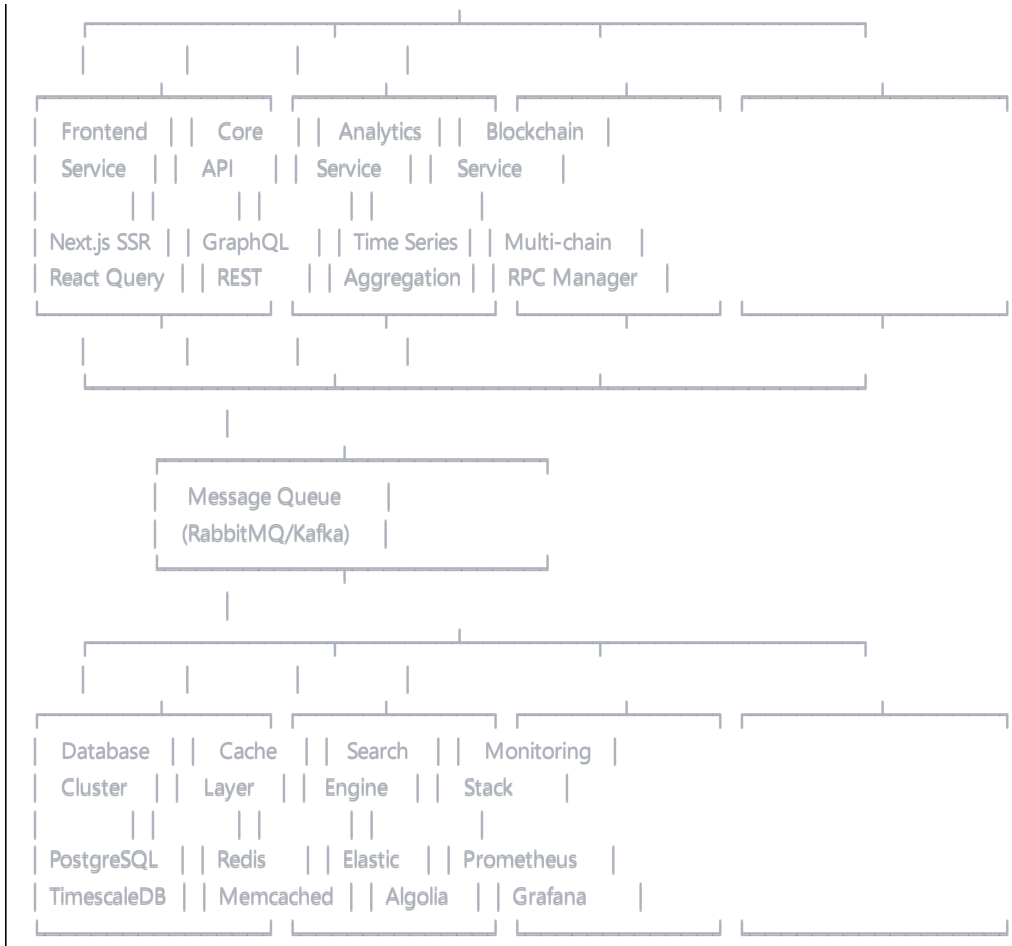
- **Architecture Pattern**: Microservices with event-driven communication
- **Development Paradigm**: Domain-Driven Design (DDD)
- **Deployment Model**: Kubernetes-orchestrated containers
- **Data Architecture**: CQRS with Event Sourcing

---

2. SYSTEM ARCHITECTURE

2.1 High-Level Architecture





2.2 Component Architecture

2.2.1 Frontend Service Architecture

typescript

```
// Component Hierarchy
├── src/
│   ├── app/                    // Next.js 14 App Router
│   │   ├── (marketing)/        // Static/SSG pages
│   │   │   ├── page.tsx        // Landing page
│   │   │   └── layout.tsx      // Marketing layout
│   │   ├── (app)/              // Dynamic/SSR pages
│   │   │   ├── dashboard/
│   │   │   ├── network/
│   │   │   └── rewards/
│   │   └── api/                // API routes
│   ├── components/
│   │   ├── ui/                 // Base components
│   │   ├── features/           // Feature components
│   │   └── layouts/            // Layout components
│   ├── hooks/                  // Custom React hooks
│   ├── lib/                    // Utilities
│   ├── services/               // API clients
│   └── stores/                 // Zustand stores

// State Management Architecture
interface AppState {
  user: UserState;
  wallet: WalletState;
  network: NetworkState;
  rebase: RebaseState;
  ui: UIState;
}

// Render Optimization Strategy
- Static Generation: Landing, docs
- Server-Side Rendering: Dashboard, analytics
- Client-Side Rendering: Network graph, real-time data
- Incremental Static Regeneration: Leaderboards (60s)
```

2.2.2 Core API Service Design

typescript

```
// GraphQL Schema Architecture
type Query {
  # User queries
  user(address: String!): User
  userNetwork(address: String!, depth: Int = 3): NetworkGraph
  userRewards(address: String!, timeframe: TimeFrame!): RewardHistory

  # Global queries
  globalStats: GlobalStatistics
  leaderboard(type: LeaderboardType!, limit: Int = 100): [LeaderboardEntry!]!
  rebaseHistory(chain: Chain!, limit: Int = 30): [RebaseEvent!]!
}

type Mutation {
  # Referral operations
  registerReferral(code: String!, signature: String!): RegisterResult!

  # User operations
  updateProfile(input: ProfileInput!): User!
  claimRewards(type: RewardType!): ClaimResult!
}

type Subscription {
  # Real-time updates
  balanceUpdates(address: String!): BalanceUpdate!
  rebaseEvents(chain: Chain!): RebaseEvent!
  networkGrowth(address: String!): NetworkUpdate!
}

// Service Layer Architecture
class UserService {
  constructor(
    private db: DatabaseService,
    private cache: CacheService,
    private blockchain: BlockchainService,
    private events: EventBus
  ) {}

  async getUserWithNetwork(address: string, depth: number = 3): Promise<UserNetwork> {
    // Check L1 cache (Redis)
    const cached = await this.cache.get(`user:network:${address}:${depth}`);
    if (cached) return cached;

    // Check L2 cache (Database materialized view)
    const materialized = await this.db.getMaterializedNetwork(address, depth);
    if (materialized && materialized.updatedAt > Date.now() - 300000) {
      await this.cache.set(`user:network:${address}:${depth}`, materialized, 300);
      return materialized;
    }

    // Compute fresh
    const network = await this.computeNetwork(address, depth);

    // Update caches asynchronously
```

```
    this.events.emit('network:computed', { address, depth, network });

    return network;
  }

  private async computeNetwork(address: string, depth: number): Promise<UserNetwork> {
    // Parallel data fetching
    const [user, referrals, balances] = await Promise.all([
      this.blockchain.getUser(address),
      this.db.getReferralTree(address, depth),
      this.blockchain.getBalances(referrals.map(r => r.address))
    ]);

    // Build network graph
    return this.buildNetworkGraph(user, referrals, balances);
  }
}
```

2.3 Data Architecture

2.3.1 Database Schema Design

```
sql
```

```sql
-- Core user table with partitioning
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    address VARCHAR(44) UNIQUE NOT NULL,
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    metadata JSONB NOT NULL DEFAULT '{}',

    -- Indexes
    INDEX idx_users_address (address),
    INDEX idx_users_created_at (created_at)
) PARTITION BY RANGE (created_at);

-- Create monthly partitions
CREATE TABLE users_2024_11 PARTITION OF users
    FOR VALUES FROM ('2024-11-01') TO ('2024-12-01');

-- Referral relationships with graph optimization
CREATE TABLE referrals (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    referrer_address VARCHAR(44) NOT NULL,
    referee_address VARCHAR(44) NOT NULL,
    chain_id INTEGER NOT NULL,
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    depth INTEGER NOT NULL DEFAULT 1,
    path_vector INTEGER[] NOT NULL,

    -- Constraints
    CONSTRAINT uk_referrals UNIQUE (referee_address),
    CONSTRAINT fk_referrer FOREIGN KEY (referrer_address)
        REFERENCES users(address) ON DELETE CASCADE,
    CONSTRAINT fk_referee FOREIGN KEY (referee_address)
        REFERENCES users(address) ON DELETE CASCADE,

    -- Indexes for graph queries
    INDEX idx_referrals_referrer (referrer_address),
    INDEX idx_referrals_referee (referee_address),
    INDEX idx_referrals_depth (depth),
    INDEX idx_referrals_path_gin (path_vector) USING GIN
);

-- Time-series data for analytics
CREATE TABLE user_balances (
    user_address VARCHAR(44) NOT NULL,
    chain_id INTEGER NOT NULL,
    balance NUMERIC(78, 18) NOT NULL,
    usd_value NUMERIC(20, 2),
    recorded_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),

    -- TimescaleDB hypertable
    PRIMARY KEY (user_address, chain_id, recorded_at)
);

SELECT create_hypertable('user_balances', 'recorded_at');
```

```sql
-- Materialized view for network statistics
CREATE MATERIALIZED VIEW mv_user_network_stats AS
WITH RECURSIVE network_tree AS (
    -- Base case: direct referrals
    SELECT
        referrer_address,
        referee_address,
        1 as depth,
        ARRAY[referrer_address, referee_address] as path,
        1 as descendant_count
    FROM referrals

    UNION ALL

    -- Recursive case: indirect referrals
    SELECT
        nt.referrer_address,
        r.referee_address,
        nt.depth + 1,
        nt.path || r.referee_address,
        nt.descendant_count + 1
    FROM network_tree nt
    JOIN referrals r ON r.referrer_address = nt.referee_address
    WHERE nt.depth < 7
        AND NOT r.referee_address = ANY(nt.path) -- Prevent cycles
)
SELECT
    referrer_address,
    COUNT(DISTINCT referee_address) as total_referrals,
    MAX(depth) as max_depth,
    SUM(CASE WHEN depth = 1 THEN 1 ELSE 0 END) as direct_referrals,
    array_agg(DISTINCT referee_address) as all_referrals
FROM network_tree
GROUP BY referrer_address;

-- Create indexes on materialized view
CREATE INDEX idx_mv_network_stats_address ON mv_user_network_stats(referrer_address);
CREATE INDEX idx_mv_network_stats_count ON mv_user_network_stats(total_referrals DESC);
```

2.3.2 Cache Strategy Design

typescript

```typescript
// Multi-tier caching architecture
interface CacheLayer {
  L0_Browser: {
    storage: 'localStorage' | 'sessionStorage';
    ttl: 300; // 5 minutes
    size: '5MB';
    data: ['user_profile', 'recent_transactions'];
  };

  L1_CDN: {
    provider: 'Cloudflare';
    ttl: 3600; // 1 hour
    strategy: 'Cache-Control headers';
    data: ['static_assets', 'api_responses'];
  };

  L2_Application: {
    provider: 'Redis Cluster';
    ttl: 900; // 15 minutes
    eviction: 'allkeys-lru';
    data: ['session_data', 'computed_networks', 'price_feeds'];
  };

  L3_Database: {
    provider: 'PostgreSQL';
    strategy: 'Materialized Views';
    refresh: 'CONCURRENTLY';
    data: ['network_statistics', 'leaderboards', 'analytics'];
  };
}

// Cache key patterns
const CACHE_KEYS = {
  USER_PROFILE: (address: string) => `user:profile:${address}`,
  USER_NETWORK: (address: string, depth: number) => `user:network:${address}:${depth}`,
  REBASE_HISTORY: (chain: string) => `rebase:history:${chain}`,
  GLOBAL_STATS: () => 'global:stats:current',
  PRICE_FEED: (token: string) => `price:${token}:usd`,
} as const;

// Cache invalidation strategy
class CacheInvalidator {
  constructor(private redis: Redis, private events: EventEmitter) {
    this.setupEventListeners();
  }

  private setupEventListeners() {
    // Invalidate on blockchain events
    this.events.on('rebase:completed', async (event) => {
      await this.invalidatePattern(`rebase:*:${event.chain}`);
      await this.invalidatePattern('global:stats:*');
    });

    // Invalidate on user actions
```

```
    this.events.on('referral:registered', async (event) => {
      await this.invalidatePattern(`user:network:${event.referrer}:*`);
      await this.invalidatePattern(`user:network:${event.referee}:*`);
    });
  }

  private async invalidatePattern(pattern: string) {
    const keys = await this.redis.keys(pattern);
    if (keys.length > 0) {
      await this.redis.del(...keys);
    }
  }
}
```

2.4 Blockchain Integration Architecture

2.4.1 Multi-Chain RPC Management

typescript

```
// RPC Provider management with failover
interface RPCConfig {
  chains: {
    solana: {
      primary: 'https://api.mainnet-beta.solana.com';
      fallback: ['https://solana-api.projectserum.com', 'https://rpc.ankr.com/solana'];
      timeout: 5000;
      retries: 3;
    };
    base: {
      primary: 'https://mainnet.base.org';
      fallback: ['https://base.llamarpc.com', 'https://base.blockpi.network/v1/rpc/public'];
      timeout: 10000;
      retries: 3;
    };
    arbitrum: {
      primary: 'https://arb1.arbitrum.io/rpc';
      fallback: ['https://arbitrum.llamarpc.com', 'https://arbitrum.blockpi.network/v1/rpc/public'];
      timeout: 10000;
      retries: 3;
    };
  };
}

// Health-check based RPC rotation
class RPCManager {
  private providers: Map<string, ProviderPool> = new Map();
  private healthScores: Map<string, number> = new Map();

  constructor(private config: RPCConfig) {
    this.initializeProviders();
    this.startHealthChecking();
  }

  async call<T>(chain: string, method: string, params: any[]): Promise<T> {
    const pool = this.providers.get(chain);
    if (!pool) throw new Error(`Unsupported chain: ${chain}`);

    // Try providers in order of health score
    const providers = pool.getHealthyProviders();

    for (const provider of providers) {
      try {
        const result = await this.executeWithTimeout(
          provider.call(method, params),
          this.config.chains[chain].timeout
        );

        // Update health score on success
        this.updateHealthScore(provider.url, true);

        return result;
      } catch (error) {
        // Update health score on failure
```

```
        this.updateHealthScore(provider.url, false);

        // Continue to next provider
        continue;
      }
    }

    throw new Error(`All RPC providers failed for ${chain}`);
  }

  private startHealthChecking() {
    setInterval(async () => {
      for (const [chain, pool] of this.providers) {
        for (const provider of pool.getAllProviders()) {
          try {
            // Chain-specific health check
            if (chain === 'solana') {
              await provider.getSlot();
            } else {
              await provider.getBlockNumber();
            }
            this.updateHealthScore(provider.url, true);
          } catch {
            this.updateHealthScore(provider.url, false);
          }
        }
      }
    }, 30000); // Every 30 seconds
  }
}
```

2.4.2 Smart Contract Integration Layer

typescript

```typescript
// Contract abstraction layer
abstract class ContractAdapter {
  abstract async getUserBalance(address: string): Promise<BigNumber>;
  abstract async getReferralData(address: string): Promise<ReferralInfo>;
  abstract async registerReferral(referee: string, referrer: string): Promise<TransactionResult>;
}

// Solana implementation
class SolanaContractAdapter extends ContractAdapter {
  constructor(
    private connection: Connection,
    private programId: PublicKey
  ) {
    super();
  }

  async getUserBalance(address: string): Promise<BigNumber> {
    const pubkey = new PublicKey(address);
    const accountInfo = await this.connection.getAccountInfo(pubkey);

    if (!accountInfo) return BigNumber.from(0);

    // Deserialize account data
    const decoded = UserAccount.decode(accountInfo.data);
    return BigNumber.from(decoded.balance.toString());
  }

  async getReferralData(address: string): Promise<ReferralInfo> {
    const [pda] = await PublicKey.findProgramAddress(
      [Buffer.from('referral'), new PublicKey(address).toBuffer()],
      this.programId
    );

    const account = await this.connection.getAccountInfo(pda);
    if (!account) return null;

    return ReferralAccount.decode(account.data);
  }

  async registerReferral(referee: string, referrer: string): Promise<TransactionResult> {
    const instruction = await this.program.methods
      .registerReferral()
      .accounts({
        referee: new PublicKey(referee),
        referrer: new PublicKey(referrer),
        referralAccount: await this.getReferralPDA(referee),
        systemProgram: SystemProgram.programId,
      })
      .instruction();

    const transaction = new Transaction().add(instruction);

    // Add priority fee for better inclusion
    transaction.add(
```

```
      ComputeBudgetProgram.setComputeUnitPrice({
        microLamports: 1000,
      })
    );

    return this.sendTransaction(transaction);
  }
}

// EVM implementation with gas optimization
class EVMContractAdapter extends ContractAdapter {
  private contract: Contract;
  private gasEstimator: GasEstimator;

  constructor(
    private provider: Provider,
    private contractAddress: string,
    private abi: any[]
  ) {
    super();
    this.contract = new Contract(contractAddress, abi, provider);
    this.gasEstimator = new GasEstimator(provider);
  }

  async registerReferral(referee: string, referrer: string): Promise<TransactionResult> {
    // Estimate gas with 10% buffer
    const estimatedGas = await this.contract.estimateGas.registerReferral(referee, referrer);
    const gasLimit = estimatedGas.mul(110).div(100);

    // Get optimal gas price
    const gasPrice = await this.gasEstimator.getOptimalGasPrice();

    // Execute with retry logic
    return this.executeWithRetry(async () => {
      const tx = await this.contract.registerReferral(referee, referrer, {
        gasLimit,
        gasPrice,
      });

      return {
        hash: tx.hash,
        wait: () => tx.wait(),
      };
    });
  }
}
```

2.5 Real-Time Data Architecture

2.5.1 WebSocket Service Design

typescript

```
// WebSocket server with horizontal scaling
class RealtimeService {
  private io: Server;
  private redis: RedisAdapter;
  private subscriptions: Map<string, Set<string>> = new Map();

  constructor(server: HttpServer) {
    this.io = new Server(server, {
      cors: {
        origin: process.env.ALLOWED_ORIGINS?.split(','),
        credentials: true,
      },
      transports: ['websocket', 'polling'],
    });

    // Enable Redis adapter for multi-instance support
    this.redis = new RedisAdapter({
      pubClient: createRedisClient(),
      subClient: createRedisClient(),
    });

    this.io.adapter(this.redis);
    this.setupEventHandlers();
  }

  private setupEventHandlers() {
    this.io.on('connection', async (socket) => {
      const { address } = socket.handshake.auth;

      if (!address) {
        socket.disconnect();
        return;
      }

      // Join user-specific room
      socket.join(`user:${address}`);

      // Handle subscriptions
      socket.on('subscribe', async (event: SubscribeEvent) => {
        switch (event.type) {
          case 'balance':
            await this.subscribeToBalance(socket, address);
            break;
          case 'network':
            await this.subscribeToNetwork(socket, address);
            break;
          case 'rebase':
            await this.subscribeToRebase(socket, event.chain);
            break;
        }
      });

      // Cleanup on disconnect
      socket.on('disconnect', () => {
```

```
      this.cleanupSubscriptions(socket.id);
    });
  });
}

private async subscribeToBalance(socket: Socket, address: string) {
  // Send initial balance
  const balance = await this.getBalance(address);
  socket.emit('balance:update', balance);

  // Subscribe to updates
  const subscription = `balance:${address}`;
  this.addSubscription(socket.id, subscription);

  // Listen for blockchain events
  this.blockchainEvents.on(subscription, (data) => {
    socket.emit('balance:update', data);
  });
}

// Efficient broadcast for global events
async broadcastRebase(chain: string, event: RebaseEvent) {
  // Use Redis pub/sub for multi-instance coordination
  await this.redis.publish('rebase:event', JSON.stringify({ chain, event }));

  // Local broadcast
  this.io.to(`rebase:${chain}`).emit('rebase:completed', event);

  // Update affected user balances
  const affectedUsers = await this.getAffectedUsers(chain);

  // Batch updates for efficiency
  const chunks = this.chunkArray(affectedUsers, 100);
  for (const chunk of chunks) {
    const updates = await this.calculateBalanceUpdates(chunk, event);

    for (const update of updates) {
      this.io.to(`user:${update.address}`).emit('balance:update', update);
    }
  }
}
```

2.5.2 Event Streaming Architecture

typescript

```typescript
// Event sourcing for audit trail and replay capability
interface DomainEvent {
  id: string;
  type: string;
  aggregateId: string;
  payload: any;
  metadata: {
    timestamp: number;
    userId?: string;
    correlationId: string;
  };
}

class EventStore {
  constructor(
    private kafka: Kafka,
    private db: Database
  ) {
    this.producer = kafka.producer();
    this.setupConsumers();
  }

  async append(event: DomainEvent): Promise<void> {
    // Store in database for persistence
    await this.db.query(
      `INSERT INTO events (id, type, aggregate_id, payload, metadata, created_at)
       VALUES ($1, $2, $3, $4, $5, NOW())`,
      [event.id, event.type, event.aggregateId, event.payload, event.metadata]
    );

    // Publish to Kafka for real-time processing
    await this.producer.send({
      topic: `events.${event.type}`,
      messages: [{
        key: event.aggregateId,
        value: JSON.stringify(event),
        headers: {
          'correlation-id': event.metadata.correlationId,
        },
      }],
    });
  }

  // Event replay for debugging or recovery
  async replay(aggregateId: string, fromTimestamp?: number): Promise<DomainEvent[]> {
    const query = fromTimestamp
      ? `SELECT * FROM events WHERE aggregate_id = $1 AND created_at >= $2 ORDER BY created_at`
      : `SELECT * FROM events WHERE aggregate_id = $1 ORDER BY created_at`;

    const result = await this.db.query(query,
      fromTimestamp ? [aggregateId, new Date(fromTimestamp)] : [aggregateId]
    );

    return result.rows.map(row => ({
```

```
      id: row.id,
      type: row.type,
      aggregateId: row.aggregate_id,
      payload: row.payload,
      metadata: row.metadata,
    }));
  }
}

// Event handlers for different domains
class ReferralEventHandler {
  async handle(event: DomainEvent): Promise<void> {
    switch (event.type) {
      case 'referral.registered':
        await this.handleReferralRegistered(event);
        break;
      case 'referral.transferred':
        await this.handleReferralTransferred(event);
        break;
    }
  }

  private async handleReferralRegistered(event: DomainEvent) {
    const { referee, referrer, chain } = event.payload;

    // Update graph database
    await this.graphDb.addEdge(referrer, referee, {
      chain,
      timestamp: event.metadata.timestamp,
    });

    // Invalidate caches
    await this.cache.invalidate([
      `network:${referrer}:*`,
      `network:${referee}:*`,
    ]);

    // Calculate rewards
    const rewards = await this.rewardCalculator.calculateReferralRewards(
      referrer,
      referee
    );

    // Emit reward events
    for (const reward of rewards) {
      await this.eventStore.append({
        id: generateId(),
        type: 'reward.calculated',
        aggregateId: reward.recipient,
        payload: reward,
        metadata: {
          timestamp: Date.now(),
          correlationId: event.metadata.correlationId,
        },
      });
```

```
    }
  }
}
```

2.6 Network Graph Computation

2.6.1 Graph Algorithm Implementation

typescript

```typescript
// Efficient network strength calculation
class NetworkCalculator {
  private graphDb: Neo4j.Driver;
  private cache: RedisCache;

  async calculateNetworkStrength(address: string): Promise<NetworkStrength> {
    // Use Neo4j for complex graph queries
    const session = this.graphDb.session();

    try {
      // Get network metrics in single query
      const result = await session.run(`
        MATCH (root:User {address: $address})
        OPTIONAL MATCH path = (root)-[:REFERRED*1..7]->(descendant:User)
        WITH root, descendant, length(path) as depth
        RETURN
          count(DISTINCT descendant) as totalDescendants,
          sum(descendant.balance * power(0.7, depth)) as weightedBalance,
          max(depth) as maxDepth,
          collect({
            address: descendant.address,
            balance: descendant.balance,
            depth: depth
          })[0..100] as topDescendants
      `, { address });

      const record = result.records[0];

      // Calculate network strength score
      const strength = this.computeStrength({
        descendants: record.get('totalDescendants'),
        weightedBalance: record.get('weightedBalance'),
        maxDepth: record.get('maxDepth'),
      });

      // Cache result
      await this.cache.set(
        `network:strength:${address}`,
        strength,
        300 // 5 minutes
      );

      return strength;
    } finally {
      await session.close();
    }
  }

  private computeStrength(metrics: NetworkMetrics): number {
    // Logarithmic scaling to prevent extreme values
    const descendantScore = Math.log10(metrics.descendants + 1) * 100;
    const balanceScore = Math.log10(metrics.weightedBalance + 1) * 50;
    const depthScore = metrics.maxDepth * 10;
```

```
    return Math.min(descendantScore + balanceScore + depthScore, 10000);
  }

  // Batch processing for rebase calculations
  async calculateRebaseRewards(epoch: number): Promise<RebaseDistribution[]> {
    const session = this.graphDb.session();

    try {
      // Get all users with their network strength
      const result = await session.run(`
        MATCH (user:User)
        WHERE user.balance > $minBalance
        OPTIONAL MATCH (user)-[:REFERRED*1..7]->(descendant:User)
        WITH user,
             count(DISTINCT descendant) as networkSize,
             sum(descendant.balance * 0.7 ^ length((user)-[:REFERRED*]->(descendant))) as networkValue
        RETURN user.address as address,
               user.balance as balance,
               networkSize,
               networkValue,
               user.isVerified as isVerified
        ORDER BY networkValue DESC
      `, { minBalance: 100 });

      // Calculate total distribution
      const totalSupply = await this.getTotalSupply();
      const rebaseAmount = totalSupply * 0.02; // 2% daily

      // Weight calculation
      let totalWeight = 0;
      const weights = result.records.map(record => {
        const balance = record.get('balance');
        const networkValue = record.get('networkValue');
        const isVerified = record.get('isVerified');

        // Base weight from balance
        let weight = balance / totalSupply;

        // Network multiplier
        weight *= (1 + Math.log10(networkValue + 1) * 0.5);

        // Verification bonus
        if (isVerified) weight *= 1.25;

        totalWeight += weight;

        return {
          address: record.get('address'),
          weight,
        };
      });

      // Distribute rewards proportionally
      return weights.map(({ address, weight }) => ({
        address,
```

```
        amount: (weight / totalWeight) * rebaseAmount,
        epoch,
      }));

    } finally {
      await session.close();
    }
  }
}
```

2.6.2 Graph Visualization Engine

typescript

```typescript
// 3D network visualization with WebGL
class NetworkVisualizer {
  private scene: THREE.Scene;
  private renderer: THREE.WebGLRenderer;
  private camera: THREE.PerspectiveCamera;
  private nodes: Map<string, THREE.Mesh> = new Map();
  private edges: THREE.LineSegments[] = [];

  constructor(container: HTMLElement) {
    this.initializeScene(container);
    this.setupInteractions();
  }

  async renderNetwork(data: NetworkData) {
    // Clear existing visualization
    this.clearScene();

    // Create desert rose center node
    const centerNode = this.createRoseNode(data.root, 20);
    centerNode.position.set(0, 0, 0);
    this.scene.add(centerNode);
    this.nodes.set(data.root.address, centerNode);

    // Layout algorithm for optimal positioning
    const positions = this.calculateLayout(data);

    // Render nodes with animation
    for (const node of data.nodes) {
      if (node.address === data.root.address) continue;

      const position = positions.get(node.address);
      const mesh = this.createNode(node, position);

      // Animate node appearance
      gsap.from(mesh.scale, {
        duration: 0.5,
        x: 0,
        y: 0,
        z: 0,
        ease: 'back.out',
        delay: node.depth * 0.1,
      });

      this.scene.add(mesh);
      this.nodes.set(node.address, mesh);
    }

    // Render edges with gradient
    for (const edge of data.edges) {
      const line = this.createEdge(
        positions.get(edge.from),
        positions.get(edge.to),
        edge.weight
      );
```

```
    // Animate edge drawing
    const material = line.material as THREE.ShaderMaterial;
    gsap.from(material.uniforms.progress, {
      duration: 1,
      value: 0,
      ease: 'power2.inOut',
      delay: edge.depth * 0.1,
    });

    this.scene.add(line);
    this.edges.push(line);
  }

  // Start render loop
  this.animate();
}

private calculateLayout(data: NetworkData): Map<string, THREE.Vector3> {
  const positions = new Map<string, THREE.Vector3>();

  // Use force-directed layout with constraints
  const simulation = d3.forceSimulation(data.nodes)
    .force('link', d3.forceLink(data.edges).id(d => d.address).distance(50))
    .force('charge', d3.forceManyBody().strength(-300))
    .force('center', d3.forceCenter(0, 0))
    .force('radial', d3.forceRadial(d => d.depth * 100, 0, 0))
    .stop();

  // Run simulation
  for (let i = 0; i < 100; i++) {
    simulation.tick();
  }

  // Convert to 3D positions with depth layers
  data.nodes.forEach(node => {
    positions.set(node.address, new THREE.Vector3(
      node.x,
      node.y,
      -node.depth * 30 // Z-depth based on referral depth
    ));
  });

  return positions;
}

private createRoseNode(node: NodeData, size: number): THREE.Mesh {
  // Custom shader for animated desert rose
  const geometry = new THREE.IcosahedronGeometry(size, 2);
  const material = new THREE.ShaderMaterial({
    uniforms: {
      time: { value: 0 },
      color1: { value: new THREE.Color(0xe8b4b8) }, // Rose gold
      color2: { value: new THREE.Color(0x6b5b95) }, // Deep purple
    },
```

```
      vertexShader: `
        varying vec3 vNormal;
        varying vec3 vPosition;

        void main() {
          vNormal = normal;
          vPosition = position;
          gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
        }
      `,
      fragmentShader: `
        uniform float time;
        uniform vec3 color1;
        uniform vec3 color2;

        varying vec3 vNormal;
        varying vec3 vPosition;

        void main() {
          float pulse = sin(time * 2.0) * 0.5 + 0.5;
          vec3 color = mix(color1, color2, pulse);

          float fresnel = pow(1.0 - dot(vNormal, vec3(0.0, 0.0, 1.0)), 2.0);

          gl_FragColor = vec4(color * (0.7 + fresnel * 0.3), 0.9);
        }
      `,
      transparent: true,
    });

    return new THREE.Mesh(geometry, material);
  }
}
```

2.7 Performance Optimization

2.7.1 Frontend Performance

typescript

```
// React component optimization
const NetworkDashboard = memo(({ address }: { address: string }) => {
  // Split code for lazy loading
  const NetworkGraph = lazy(() => import('./NetworkGraph'));
  const RebaseHistory = lazy(() => import('./RebaseHistory'));

  // Virtualized list for large datasets
  const VirtualizedLeaderboard = ({ data }: { data: LeaderboardEntry[] }) => {
    const rowRenderer = useCallback(({ index, style }) => {
      const entry = data[index];
      return (
        <div style={style} className="flex items-center p-4">
          <RankDisplay rank={index + 1} />
          <UserDisplay user={entry} />
          <BalanceDisplay balance={entry.balance} />
        </div>
      );
    }, [data]);

    return (
      <AutoSizer>
        {({ height, width }) => (
          <List
            height={height}
            width={width}
            rowCount={data.length}
            rowHeight={80}
            rowRenderer={rowRenderer}
            overscanRowCount={5}
          />
        )}
      </AutoSizer>
    );
  };

  // Optimistic updates for better UX
  const { mutate: registerReferral } = useMutation({
    mutationFn: api.registerReferral,
    onMutate: async (variables) => {
      // Cancel outgoing queries
      await queryClient.cancelQueries(['network', address]);

      // Snapshot previous value
      const previousNetwork = queryClient.getQueryData(['network', address]);

      // Optimistically update
      queryClient.setQueryData(['network', address], (old) => ({
        ...old,
        referrals: [...old.referrals, variables.referee],
      }));

      return { previousNetwork };
    },
    onError: (err, variables, context) => {
```

```
      // Rollback on error
      queryClient.setQueryData(['network', address], context.previousNetwork);
    },
    onSettled: () => {
      // Refetch to ensure consistency
      queryClient.invalidateQueries(['network', address]);
    },
  });

  return (
    <Suspense fallback={<LoadingSpinner />}>
      <div className="grid grid-cols-12 gap-6">
        <div className="col-span-8">
          <NetworkGraph address={address} />
        </div>
        <div className="col-span-4">
          <RebaseHistory />
          <VirtualizedLeaderboard data={leaderboardData} />
        </div>
      </div>
    </Suspense>
  );
});

// Image optimization
const OptimizedImage = ({ src, alt, ...props }: ImageProps) => {
  return (
    <Image
      src={src}
      alt={alt}
      loading="lazy"
      placeholder="blur"
      blurDataURL={generateBlurDataURL(src)}
      sizes="(max-width: 768px) 100vw, (max-width: 1200px) 50vw, 33vw"
      {...props}
    />
  );
};
```

2.7.2 API Performance

typescript

```
// Request batching and deduplication
class DataLoader {
  private batchQueue: Map<string, Set<string>> = new Map();
  private batchTimeout: NodeJS.Timeout;

  async loadUser(address: string): Promise<User> {
    return this.load('user', address);
  }

  private async load(type: string, id: string): Promise<any> {
    // Check cache first
    const cached = await this.cache.get(`${type}:${id}`);
    if (cached) return cached;

    // Add to batch queue
    if (!this.batchQueue.has(type)) {
      this.batchQueue.set(type, new Set());
    }
    this.batchQueue.get(type).add(id);

    // Schedule batch execution
    if (!this.batchTimeout) {
      this.batchTimeout = setTimeout(() => this.executeBatch(), 10);
    }

    // Return promise that resolves when batch completes
    return new Promise((resolve, reject) => {
      this.pendingRequests.set(`${type}:${id}`, { resolve, reject });
    });
  }

  private async executeBatch() {
    const batches = new Map(this.batchQueue);
    this.batchQueue.clear();
    this.batchTimeout = null;

    // Execute batches in parallel
    await Promise.all(
      Array.from(batches.entries()).map(async ([type, ids]) => {
        try {
          const results = await this.batchFetch(type, Array.from(ids));

          // Resolve individual promises
          results.forEach((result, index) => {
            const id = Array.from(ids)[index];
            const key = `${type}:${id}`;

            // Cache result
            this.cache.set(key, result, 300);

            // Resolve promise
            const pending = this.pendingRequests.get(key);
            if (pending) {
              pending.resolve(result);
```

```
            this.pendingRequests.delete(key);
          }
        });
      } catch (error) {
        // Reject all promises in batch
        ids.forEach(id => {
          const key = `${type}:${id}`;
          const pending = this.pendingRequests.get(key);
          if (pending) {
            pending.reject(error);
            this.pendingRequests.delete(key);
          }
        });
      }
    })
  );
 }
}

// Query optimization with prepared statements
class OptimizedDatabase {
 private statements: Map<string, any> = new Map();

 async initialize() {
  // Prepare frequently used statements
  this.statements.set('getUserNetwork', await this.db.prepare(`
   WITH RECURSIVE network AS (
    SELECT
     u.address,
     u.balance,
     0 as depth,
     ARRAY[u.address] as path
    FROM users u
    WHERE u.address = $1

    UNION ALL

    SELECT
     u.address,
     u.balance,
     n.depth + 1,
     n.path || u.address
    FROM users u
    JOIN referrals r ON r.referee_address = u.address
    JOIN network n ON n.address = r.referrer_address
    WHERE n.depth < $2
     AND NOT u.address = ANY(n.path)
   )
   SELECT * FROM network;
  `));
 }

 async getUserNetwork(address: string, maxDepth: number = 7): Promise<NetworkNode[]> {
  const statement = this.statements.get('getUserNetwork');
  return statement.all(address, maxDepth);
```

```
  }
}
```

2.8 Security Architecture

2.8.1 Authentication & Authorization

typescript

```typescript
// JWT-based authentication with refresh tokens
interface AuthToken {
  address: string;
  chainId: number;
  nonce: string;
  issuedAt: number;
  expiresAt: number;
  signature: string;
}

class AuthService {
  private readonly ACCESS_TOKEN_TTL = 900; // 15 minutes
  private readonly REFRESH_TOKEN_TTL = 604800; // 7 days

  async authenticate(request: AuthRequest): Promise<AuthResponse> {
    // Verify signature
    const message = `Sign this message to authenticate with HABIBI Protocol.\n\nNonce: ${request.nonce}`;
    const recovered = ethers.utils.verifyMessage(message, request.signature);

    if (recovered.toLowerCase() !== request.address.toLowerCase()) {
      throw new UnauthorizedError('Invalid signature');
    }

    // Check nonce to prevent replay attacks
    const usedNonce = await this.redis.get(`nonce:${request.nonce}`);
    if (usedNonce) {
      throw new UnauthorizedError('Nonce already used');
    }

    // Mark nonce as used
    await this.redis.setex(`nonce:${request.nonce}`, 3600, '1');

    // Generate tokens
    const accessToken = this.generateAccessToken(request.address);
    const refreshToken = this.generateRefreshToken(request.address);

    // Store refresh token
    await this.redis.setex(
      `refresh:${refreshToken}`,
      this.REFRESH_TOKEN_TTL,
      JSON.stringify({
        address: request.address,
        createdAt: Date.now(),
      })
    );

    return {
      accessToken,
      refreshToken,
      expiresIn: this.ACCESS_TOKEN_TTL,
    };
  }

  // Rate limiting middleware
```

```
  createRateLimiter(options: RateLimitOptions) {
    return async (req: Request, res: Response, next: NextFunction) => {
      const key = options.keyGenerator(req);
      const limit = options.limit;
      const window = options.window;

      // Use sliding window algorithm
      const now = Date.now();
      const windowStart = now - window * 1000;

      // Get request timestamps
      const timestamps = await this.redis.zrangebyscore(
        `ratelimit:${key}`,
        windowStart,
        now
      );

      if (timestamps.length >= limit) {
        const resetTime = parseInt(timestamps[0]) + window * 1000;
        res.setHeader('X-RateLimit-Limit', limit.toString());
        res.setHeader('X-RateLimit-Remaining', '0');
        res.setHeader('X-RateLimit-Reset', Math.floor(resetTime / 1000).toString());

        return res.status(429).json({
          error: 'Too many requests',
          retryAfter: Math.ceil((resetTime - now) / 1000),
        });
      }

      // Add current request
      await this.redis.zadd(`ratelimit:${key}`, now, `${now}:${crypto.randomUUID()}`);
      await this.redis.expire(`ratelimit:${key}`, window);

      // Clean old entries
      await this.redis.zremrangebyscore(`ratelimit:${key}`, 0, windowStart);

      // Set headers
      res.setHeader('X-RateLimit-Limit', limit.toString());
      res.setHeader('X-RateLimit-Remaining', (limit - timestamps.length - 1).toString());

      next();
    };
  }
}

// RBAC implementation
class AuthorizationService {
  private permissions = {
    user: ['read:own_profile', 'read:own_network', 'write:own_referral'],
    verified: ['read:leaderboard', 'write:claim_rewards'],
    admin: ['read:all_users', 'write:system_config', 'execute:manual_rebase'],
  };

  async authorize(user: User, action: string, resource?: any): Promise<boolean> {
    const userPermissions = this.getUserPermissions(user);
```

```
    // Check direct permission
    if (userPermissions.includes(action)) {
      return true;
    }

    // Check resource-based permissions
    if (resource && action.startsWith('read:own_') || action.startsWith('write:own_')) {
      return resource.address === user.address;
    }

    return false;
  }

  private getUserPermissions(user: User): string[] {
    const roles = ['user'];

    if (user.isVerified) roles.push('verified');
    if (user.isAdmin) roles.push('admin');

    return roles.flatMap(role => this.permissions[role] || []);
  }
}
```

2.8.2 Smart Contract Security

typescript

```
// Contract interaction security layer
class SecureContractService {
  private readonly MAX_SLIPPAGE = 0.05; // 5%
  private readonly SIMULATION_REQUIRED = true;

  async executeTransaction(params: TransactionParams): Promise<TransactionResult> {
    // 1. Validate contract address
    if (!this.isKnownContract(params.to)) {
      throw new SecurityError('Unknown contract address');
    }

    // 2. Simulate transaction
    if (this.SIMULATION_REQUIRED) {
      const simulation = await this.simulateTransaction(params);

      // Check for unexpected outcomes
      if (simulation.reverted) {
        throw new TransactionError('Transaction would revert', simulation.reason);
      }

      // Check for excessive value transfer
      if (simulation.valueTransferred > params.expectedValue * (1 + this.MAX_SLIPPAGE)) {
        throw new SecurityError('Excessive value transfer detected');
      }
    }

    // 3. Check for reentrancy
    const reentrancyKey = `reentrancy:${params.from}:${params.to}`;
    const isReentrant = await this.redis.get(reentrancyKey);
    if (isReentrant) {
      throw new SecurityError('Reentrancy detected');
    }

    // Set reentrancy lock
    await this.redis.setex(reentrancyKey, 30, '1');

    try {
      // 4. Execute with monitoring
      const tx = await this.sendTransaction(params);

      // 5. Monitor for frontrunning
      this.monitorTransaction(tx.hash);

      return tx;
    } finally {
      // Release reentrancy lock
      await this.redis.del(reentrancyKey);
    }
  }

  private async monitorTransaction(txHash: string) {
    const tx = await this.provider.getTransaction(txHash);
    const block = await this.provider.getBlock('pending');
```

```
    // Check if transaction was frontrun
    const suspiciousTransactions = block.transactions.filter(hash => {
      // Logic to detect similar transactions with higher gas price
      return this.isSuspiciousFrontrun(hash, tx);
    });

    if (suspiciousTransactions.length > 0) {
      this.alertService.send({
        type: 'FRONTRUN_DETECTED',
        transaction: txHash,
        suspicious: suspiciousTransactions,
      });
    }
  }
}

// Input validation and sanitization
class ValidationService {
  // Comprehensive input validation schemas
  private schemas = {
    address: z.string()
      .regex(/^0x[a-fA-F0-9]{40}$/, 'Invalid Ethereum address')
      .or(z.string().regex(/^[1-9A-HJ-NP-Za-km-z]{32,44}$/, 'Invalid Solana address')),

    referralCode: z.string()
      .regex(/^HABIBI-[A-Z]{3,5}-[A-Z0-9]{8}$/, 'Invalid referral code format'),

    amount: z.string()
      .regex(/^\d+(\.\d{1,18})?$/, 'Invalid amount format')
      .refine(val => parseFloat(val) > 0, 'Amount must be positive')
      .refine(val => parseFloat(val) < 1e12, 'Amount too large'),
  };

  validate<T>(schema: keyof typeof this.schemas, data: unknown): T {
    try {
      return this.schemas[schema].parse(data) as T;
    } catch (error) {
      if (error instanceof z.ZodError) {
        throw new ValidationError(error.errors[0].message);
      }
      throw error;
    }
  }

  // XSS prevention for user-generated content
  sanitizeHTML(input: string): string {
    return DOMPurify.sanitize(input, {
      ALLOWED_TAGS: ['b', 'i', 'em', 'strong', 'a'],
      ALLOWED_ATTR: ['href'],
      ALLOW_DATA_ATTR: false,
    });
  }

  // SQL injection prevention (beyond parameterized queries)
  sanitizeIdentifier(identifier: string): string {
```

```
    if (!/^[a-zA-Z_][a-zA-Z0-9_]*$/.test(identifier)) {
      throw new ValidationError('Invalid identifier');
    }
    return identifier;
  }
}
```

2.9 Monitoring and Observability

2.9.1 Metrics Collection

typescript

```typescript
// Prometheus metrics setup
class MetricsCollector {
  private register: Registry;
  private metrics: {
    httpDuration: Histogram<string>;
    activeUsers: Gauge<string>;
    referralCount: Counter<string>;
    rebaseAmount: Histogram<string>;
    errorRate: Counter<string>;
  };

  constructor() {
    this.register = new Registry();

    // HTTP request duration
    this.metrics.httpDuration = new Histogram({
      name: 'habibi_http_duration_seconds',
      help: 'Duration of HTTP requests in seconds',
      labelNames: ['method', 'route', 'status_code'],
      buckets: [0.1, 0.5, 1, 2, 5],
      registers: [this.register],
    });

    // Active users gauge
    this.metrics.activeUsers = new Gauge({
      name: 'habibi_active_users',
      help: 'Number of active users',
      labelNames: ['chain'],
      registers: [this.register],
    });

    // Referral counter
    this.metrics.referralCount = new Counter({
      name: 'habibi_referrals_total',
      help: 'Total number of referrals',
      labelNames: ['chain', 'status'],
      registers: [this.register],
    });

    // Rebase distribution
    this.metrics.rebaseAmount = new Histogram({
      name: 'habibi_rebase_amount',
      help: 'Distribution of rebase amounts',
      labelNames: ['chain'],
      buckets: [1, 10, 100, 1000, 10000],
      registers: [this.register],
    });

    // Error rate
    this.metrics.errorRate = new Counter({
      name: 'habibi_errors_total',
      help: 'Total number of errors',
      labelNames: ['type', 'severity'],
      registers: [this.register],
```

```
    });
  }

  // Express middleware for automatic HTTP metrics
  httpMiddleware() {
    return (req: Request, res: Response, next: NextFunction) => {
      const start = Date.now();

      res.on('finish', () => {
        const duration = (Date.now() - start) / 1000;

        this.metrics.httpDuration
          .labels(req.method, req.route?.path || 'unknown', res.statusCode.toString())
          .observe(duration);
      });

      next();
    };
  }

  // Custom business metrics
  recordReferral(chain: string, success: boolean) {
    this.metrics.referralCount
      .labels(chain, success ? 'success' : 'failed')
      .inc();
  }

  recordRebase(chain: string, amount: number) {
    this.metrics.rebaseAmount
      .labels(chain)
      .observe(amount);
  }
}

// Distributed tracing with OpenTelemetry
class TracingService {
  private tracer: Tracer;

  constructor() {
    const provider = new NodeTracerProvider({
      resource: new Resource({
        [SemanticResourceAttributes.SERVICE_NAME]: 'habibi-api',
        [SemanticResourceAttributes.SERVICE_VERSION]: process.env.VERSION || '1.0.0',
      }),
    });

    // Add exporters
    provider.addSpanProcessor(
      new BatchSpanProcessor(
        new JaegerExporter({
          endpoint: process.env.JAEGER_ENDPOINT,
        })
      )
    );
```

```
    provider.register();
    this.tracer = provider.getTracer('habibi-api');
  }

  // Trace async operations
  async traceAsync<T>(
    name: string,
    fn: (span: Span) => Promise<T>,
    attributes?: Attributes
  ): Promise<T> {
    const span = this.tracer.startSpan(name, { attributes });

    try {
      const result = await fn(span);
      span.setStatus({ code: SpanStatusCode.OK });
      return result;
    } catch (error) {
      span.setStatus({
        code: SpanStatusCode.ERROR,
        message: error.message,
      });
      span.recordException(error);
      throw error;
    } finally {
      span.end();
    }
  }
}
```

2.9.2 Logging Architecture

typescript

```
// Structured logging with context
class Logger {
  private winston: winston.Logger;
  private context: Map<string, any> = new Map();

  constructor() {
    this.winston = winston.createLogger({
      level: process.env.LOG_LEVEL || 'info',
      format: winston.format.combine(
        winston.format.timestamp(),
        winston.format.errors({ stack: true }),
        winston.format.json()
      ),
      defaultMeta: {
        service: 'habibi-api',
        environment: process.env.NODE_ENV,
      },
      transports: [
        // Console for development
        new winston.transports.Console({
          format: winston.format.combine(
            winston.format.colorize(),
            winston.format.simple()
          ),
        }),

        // File for production
        new winston.transports.File({
          filename: 'logs/error.log',
          level: 'error',
          maxsize: 10485760, // 10MB
          maxFiles: 5,
        }),

        // CloudWatch for AWS
        new WinstonCloudWatch({
          logGroupName: '/aws/ecs/habibi-api',
          logStreamName: `${process.env.NODE_ENV}-${process.env.HOSTNAME}`,
          awsRegion: process.env.AWS_REGION,
        }),
      ],
    });
  }

  // Contextual logging
  child(context: Record<string, any>): Logger {
    const child = Object.create(this);
    child.context = new Map([...this.context, ...Object.entries(context)]);
    return child;
  }

  private log(level: string, message: string, meta?: any) {
    const contextObj = Object.fromEntries(this.context);
```

```
    this.winston.log(level, message, {
      ...contextObj,
      ...meta,
      timestamp: new Date().toISOString(),
    });
  }

  // Typed logging methods
  info(message: string, meta?: any) {
    this.log('info', message, meta);
  }

  error(message: string, error?: Error, meta?: any) {
    this.log('error', message, {
      ...meta,
      error: error ? {
        message: error.message,
        stack: error.stack,
        name: error.name,
      } : undefined,
    });
  }

  // Performance logging
  time(label: string): () => void {
    const start = process.hrtime.bigint();

    return () => {
      const end = process.hrtime.bigint();
      const duration = Number(end - start) / 1e6; // Convert to ms

      this.info(`Timing: ${label}`, {
        duration_ms: duration,
        label,
      });
    };
  }
}

// Audit logging for compliance
class AuditLogger {
  async logUserAction(action: AuditAction) {
    const entry: AuditEntry = {
      id: generateId(),
      timestamp: new Date(),
      userId: action.userId,
      action: action.type,
      resource: action.resource,
      result: action.result,
      metadata: action.metadata,
      ipAddress: action.ipAddress,
      userAgent: action.userAgent,
    };

    // Store in append-only audit table
```

```
    await this.db.query(
      `INSERT INTO audit_log
       (id, timestamp, user_id, action, resource, result, metadata, ip_address, user_agent)
       VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9)`,
      Object.values(entry)
    );

    // Also send to SIEM if configured
    if (this.siemEndpoint) {
      await this.sendToSIEM(entry);
    }
  }
}
```

2.10 Deployment Architecture

2.10.1 Container Configuration

yaml

```yaml
# docker-compose.yml for local development
version: '3.8'

services:
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
      target: development
    environment:
      - NEXT_PUBLIC_API_URL=http://localhost:4000
      - NEXT_PUBLIC_WS_URL=ws://localhost:4000
    volumes:
      - ./frontend:/app
      - /app/node_modules
      - /app/.next
    ports:
      - "3000:3000"
    command: npm run dev

  api:
    build:
      context: ./api
      dockerfile: Dockerfile
      target: development
    environment:
      - NODE_ENV=development
      - DATABASE_URL=postgresql://user:pass@postgres:5432/habibi
      - REDIS_URL=redis://redis:6379
      - JWT_SECRET=dev-secret
    volumes:
      - ./api:/app
      - /app/node_modules
    ports:
      - "4000:4000"
    command: npm run dev
    depends_on:
      - postgres
      - redis

  postgres:
    image: timescale/timescaledb:latest-pg14
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=pass
      - POSTGRES_DB=habibi
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./database/init.sql:/docker-entrypoint-initdb.d/init.sql
    ports:
      - "5432:5432"

  redis:
    image: redis:7-alpine
```

```
    command: redis-server --appendonly yes
    volumes:
      - redis_data:/data
    ports:
      - "6379:6379"

  kafka:
    image: confluentinc/cp-kafka:latest
    environment:
      - KAFKA_ZOOKEEPER_CONNECT=zookeeper:2181
      - KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://localhost:9092
      - KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR=1
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"

  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    environment:
      - ZOOKEEPER_CLIENT_PORT=2181
      - ZOOKEEPER_TICK_TIME=2000

volumes:
  postgres_data:
  redis_data:

# Kubernetes production deployment
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: habibi-api
  namespace: habibi
spec:
  replicas: 3
  selector:
    matchLabels:
      app: habibi-api
  template:
    metadata:
      labels:
        app: habibi-api
    spec:
      containers:
      - name: api
        image: habibi/api:latest
        ports:
        - containerPort: 4000
        env:
        - name: NODE_ENV
          value: "production"
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
```

```
            name: habibi-secrets
            key: database-url
        resources:
          requests:
            memory: "512Mi"
            cpu: "500m"
          limits:
            memory: "1Gi"
            cpu: "1000m"
        livenessProbe:
          httpGet:
            path: /health
            port: 4000
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /ready
            port: 4000
          initialDelaySeconds: 5
          periodSeconds: 5
---
apiVersion: v1
kind: Service
metadata:
  name: habibi-api
  namespace: habibi
spec:
  selector:
    app: habibi-api
  ports:
  - port: 80
    targetPort: 4000
  type: ClusterIP
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: habibi-ingress
  namespace: habibi
  annotations:
    kubernetes.io/ingress.class: nginx
    cert-manager.io/cluster-issuer: letsencrypt-prod
    nginx.ingress.kubernetes.io/rate-limit: "100"
spec:
  tls:
  - hosts:
    - api.habibi.finance
    secretName: habibi-tls
  rules:
  - host: api.habibi.finance
    http:
      paths:
      - path: /
        pathType: Prefix
```

```
      backend:
        service:
          name: habibi-api
          port:
            number: 80
```

2.10.2 CI/CD Pipeline

yaml

```yaml
# .github/workflows/deploy.yml
name: Deploy to Production

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3

    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '18'
        cache: 'npm'

    - name: Install dependencies
      run: npm ci

    - name: Run tests
      run: npm test -- --coverage

    - name: Run security audit
      run: npm audit --production

    - name: SonarCloud Scan
      uses: SonarSource/sonarcloud-github-action@master
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}

  build:
    needs: test
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write

    steps:
    - uses: actions/checkout@v3

    - name: Log in to Container Registry
      uses: docker/login-action@v2
      with:
        registry: ${{ env.REGISTRY }}
```

```
      username: ${{ github.actor }}
      password: ${{ secrets.GITHUB_TOKEN }}

  - name: Build and push Docker image
    uses: docker/build-push-action@v4
    with:
      context: .
      push: true
      tags: |
        ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:latest
        ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:${{ github.sha }}
      cache-from: type=gha
      cache-to: type=gha,mode=max

deploy:
  needs: build
  runs-on: ubuntu-latest
  if: github.ref == 'refs/heads/main'

  steps:
  - name: Deploy to Kubernetes
    uses: azure/k8s-deploy@v4
    with:
      manifests: |
        k8s/deployment.yaml
        k8s/service.yaml
      images: |
        ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:${{ github.sha }}
      kubeconfig: ${{ secrets.KUBE_CONFIG }}

  - name: Run database migrations
    run: |
      kubectl exec -it deployment/habibi-api -- npm run migrate:prod

  - name: Verify deployment
    run: |
      kubectl rollout status deployment/habibi-api
      kubectl get services

  - name: Run smoke tests
    run: |
      npm run test:smoke -- --url=https://api.habibi.finance
```

2.11 Testing Strategy

2.11.1 Unit Testing

typescript

```
// Example unit test for network calculator
describe('NetworkCalculator', () => {
  let calculator: NetworkCalculator;
  let mockGraphDb: jest.Mocked<Neo4j.Driver>;
  let mockCache: jest.Mocked<RedisCache>;

  beforeEach(() => {
    mockGraphDb = createMockGraphDb();
    mockCache = createMockCache();
    calculator = new NetworkCalculator(mockGraphDb, mockCache);
  });

  describe('calculateNetworkStrength', () => {
    it('should return cached value if available', async () => {
      const cachedStrength = 1500;
      mockCache.get.mockResolvedValue(cachedStrength);

      const result = await calculator.calculateNetworkStrength('0x123');

      expect(result).toBe(cachedStrength);
      expect(mockGraphDb.session).not.toHaveBeenCalled();
    });

    it('should calculate strength from graph if not cached', async () => {
      mockCache.get.mockResolvedValue(null);

      const mockSession = {
        run: jest.fn().mockResolvedValue({
          records: [{
            get: jest.fn()
              .mockReturnValueOnce(100) // totalDescendants
              .mockReturnValueOnce(50000) // weightedBalance
              .mockReturnValueOnce(5), // maxDepth
          }],
        }),
        close: jest.fn(),
      };

      mockGraphDb.session.mockReturnValue(mockSession);

      const result = await calculator.calculateNetworkStrength('0x123');

      expect(mockSession.run).toHaveBeenCalledWith(
        expect.stringContaining('MATCH (root:User {address: $address})'),
        { address: '0x123' }
      );

      expect(result).toBeGreaterThan(0);
      expect(result).toBeLessThanOrEqual(10000);

      expect(mockCache.set).toHaveBeenCalledWith(
        'network:strength:0x123',
        result,
        300
```

```
      );
    });

    it('should handle network with no descendants', async () => {
      mockCache.get.mockResolvedValue(null);

      const mockSession = {
        run: jest.fn().mockResolvedValue({
          records: [{
            get: jest.fn()
              .mockReturnValueOnce(0) // totalDescendants
              .mockReturnValueOnce(0) // weightedBalance
              .mockReturnValueOnce(0), // maxDepth
          }],
        }),
        close: jest.fn(),
      };

      mockGraphDb.session.mockReturnValue(mockSession);

      const result = await calculator.calculateNetworkStrength('0x123');

      expect(result).toBe(0);
    });
  });
});

// Integration test for referral registration
describe('Referral Registration Integration', () => {
  let app: Application;
  let db: DatabaseService;
  let redis: RedisService;

  beforeAll(async () => {
    // Setup test environment
    app = await createTestApp();
    db = app.get(DatabaseService);
    redis = app.get(RedisService);

    // Run migrations
    await db.migrate();
  });

  afterEach(async () => {
    // Clean up test data
    await db.query('TRUNCATE TABLE users, referrals CASCADE');
    await redis.flushall();
  });

  afterAll(async () => {
    await app.close();
  });

  it('should register referral and update network', async () => {
    // Create referrer
```

```
    const referrer = await createTestUser(db, {
      address: '0xreferrer',
      balance: 10000,
    });

    // Register referee
    const response = await request(app)
      .post('/api/referrals/register')
      .send({
        refereeAddress: '0xreferee',
        referralCode: 'HABIBI-SOL-12345678',
        signature: await signMessage('0xreferee', 'Register referral'),
      })
      .expect(200);

    expect(response.body).toMatchObject({
      success: true,
      referee: '0xreferee',
      referrer: '0xreferrer',
    });

    // Verify database state
    const referral = await db.query(
      'SELECT * FROM referrals WHERE referee_address = $1',
      ['0xreferee']
    );

    expect(referral.rows).toHaveLength(1);
    expect(referral.rows[0]).toMatchObject({
      referrer_address: '0xreferrer',
      referee_address: '0xreferee',
      depth: 1,
    });

    // Verify cache invalidation
    const cachedNetwork = await redis.get('network:0xreferrer:3');
    expect(cachedNetwork).toBeNull();
  });
});
```

2.11.2 Load Testing

```
javascript
```

```
// k6 load test script
import http from 'k6/http';
import { check, sleep } from 'k6';
import { Rate } from 'k6/metrics';

const errorRate = new Rate('errors');

export const options = {
  stages: [
    { duration: '2m', target: 100 }, // Ramp up
    { duration: '5m', target: 1000 }, // Stay at 1000 users
    { duration: '2m', target: 5000 }, // Spike test
    { duration: '5m', target: 1000 }, // Back to normal
    { duration: '2m', target: 0 }, // Ramp down
  ],
  thresholds: {
    http_req_duration: ['p(95)<500'], // 95% of requests under 500ms
    errors: ['rate<0.01'], // Error rate under 1%
  },
};

const BASE_URL = 'https://api.habibi.finance';

export default function () {
  // Simulate user journey
  const userAddress = `0x${Math.random().toString(16).substr(2, 40)}`;

  // 1. Get user profile
  const profileRes = http.get(`${BASE_URL}/users/${userAddress}`);
  check(profileRes, {
    'profile status 200': (r) => r.status === 200,
    'profile response time < 200ms': (r) => r.timings.duration < 200,
  });
  errorRate.add(profileRes.status !== 200);

  sleep(1);

  // 2. Get network data
  const networkRes = http.get(`${BASE_URL}/users/${userAddress}/network?depth=3`);
  check(networkRes, {
    'network status 200': (r) => r.status === 200,
    'network response time < 500ms': (r) => r.timings.duration < 500,
  });
  errorRate.add(networkRes.status !== 200);

  sleep(2);

  // 3. Register referral (write operation)
  const referralRes = http.post(
    `${BASE_URL}/referrals/register`,
    JSON.stringify({
      refereeAddress: `0x${Math.random().toString(16).substr(2, 40)}`,
      referralCode: 'HABIBI-SOL-TEST1234',
    }),
```

```
    {
      headers: {
        'Content-Type': 'application/json',
        'Authorization': `Bearer ${__ENV.TEST_TOKEN}`,
      },
    }
  );

  check(referralRes, {
    'referral status 200': (r) => r.status === 200,
    'referral response time < 1000ms': (r) => r.timings.duration < 1000,
  });
  errorRate.add(referralRes.status !== 200);

  sleep(Math.random() * 3);
}

// Spike test for WebSocket connections
export function websocketTest() {
  const url = 'wss://api.habibi.finance/ws';
  const params = {
    headers: {
      'Authorization': `Bearer ${__ENV.TEST_TOKEN}`,
    },
  };

  const ws = http.connect(url, params, function (socket) {
    socket.on('open', () => {
      console.log('WebSocket connected');

      // Subscribe to balance updates
      socket.send(JSON.stringify({
        type: 'subscribe',
        channel: 'balance',
        address: `0x${Math.random().toString(16).substr(2, 40)}`,
      }));
    });

    socket.on('message', (data) => {
      const message = JSON.parse(data);
      check(message, {
        'received balance update': (m) => m.type === 'balance:update',
      });
    });

    socket.on('error', (e) => {
      console.error('WebSocket error:', e);
      errorRate.add(1);
    });

    // Keep connection open for 30 seconds
    socket.setTimeout(() => {
      socket.close();
    }, 30000);
  });
```

```
  check(ws, {
    'WebSocket connection established': (ws) => ws.status === 101,
  });
}
```

2.12 Disaster Recovery

2.12.1 Backup Strategy

bash

```bash
#!/bin/bash
# backup.sh - Automated backup script

set -euo pipefail

# Configuration
BACKUP_DIR="/backups"
RETENTION_DAYS=30
S3_BUCKET="habibi-backups"
ENCRYPTION_KEY_ID="arn:aws:kms:us-east-1:123456789012:key/12345678-1234-1234-1234-123456789012"

# Create backup directory
TIMESTAMP=$(date +%Y%m%d_%H%M%S)
BACKUP_PATH="${BACKUP_DIR}/${TIMESTAMP}"
mkdir -p "${BACKUP_PATH}"

# Backup PostgreSQL
echo "Backing up PostgreSQL..."
PGPASSWORD="${POSTGRES_PASSWORD}" pg_dump \
  -h "${POSTGRES_HOST}" \
  -U "${POSTGRES_USER}" \
  -d "${POSTGRES_DB}" \
  -Fc \
  -f "${BACKUP_PATH}/postgres.dump"

# Backup Redis
echo "Backing up Redis..."
redis-cli -h "${REDIS_HOST}" --rdb "${BACKUP_PATH}/redis.rdb"

# Backup Neo4j (if using for graph)
echo "Backing up Neo4j..."
neo4j-admin backup \
  --database=neo4j \
  --backup-dir="${BACKUP_PATH}/neo4j"

# Create encrypted archive
echo "Creating encrypted archive..."
tar -czf - -C "${BACKUP_PATH}" . | \
  openssl enc -aes-256-cbc -salt -pass pass:"${BACKUP_PASSWORD}" \
  > "${BACKUP_PATH}.tar.gz.enc"

# Upload to S3 with server-side encryption
echo "Uploading to S3..."
aws s3 cp "${BACKUP_PATH}.tar.gz.enc" \
  "s3://${S3_BUCKET}/backups/${TIMESTAMP}.tar.gz.enc" \
  --storage-class GLACIER \
  --server-side-encryption aws:kms \
  --ssekms-key-id "${ENCRYPTION_KEY_ID}"

# Clean up local files
rm -rf "${BACKUP_PATH}" "${BACKUP_PATH}.tar.gz.enc"

# Remove old backups from S3
echo "Cleaning up old backups..."
```

```
aws s3 ls "s3://${S3_BUCKET}/backups/" | \
  while read -r line; do
    createDate=$(echo "$line" | awk '{print $1" "$2}')
    createDate=$(date -d "$createDate" +%s)
    olderThan=$(date -d "${RETENTION_DAYS} days ago" +%s)

    if [[ $createDate -lt $olderThan ]]; then
      fileName=$(echo "$line" | awk '{print $4}')
      echo "Deleting $fileName"
      aws s3 rm "s3://${S3_BUCKET}/backups/$fileName"
    fi
  done

echo "Backup completed successfully"

# Send notification
curl -X POST "${SLACK_WEBHOOK_URL}" \
  -H 'Content-Type: application/json' \
  -d "{\"text\":\"Backup completed successfully: ${TIMESTAMP}\"}"
```

2.12.2 Recovery Procedures

typescript

```typescript
// Disaster recovery orchestration
class DisasterRecoveryService {
  async performRecovery(backupId: string): Promise<RecoveryResult> {
    const steps: RecoveryStep[] = [
      {
        name: 'Validate Backup',
        action: () => this.validateBackup(backupId),
        rollback: null,
      },
      {
        name: 'Stop Services',
        action: () => this.stopServices(),
        rollback: () => this.startServices(),
      },
      {
        name: 'Restore Database',
        action: () => this.restoreDatabase(backupId),
        rollback: () => this.restoreDatabaseFromLatest(),
      },
      {
        name: 'Restore Cache',
        action: () => this.restoreCache(backupId),
        rollback: null, // Cache can be rebuilt
      },
      {
        name: 'Verify Data Integrity',
        action: () => this.verifyDataIntegrity(),
        rollback: null,
      },
      {
        name: 'Rebuild Materialized Views',
        action: () => this.rebuildMaterializedViews(),
        rollback: null,
      },
      {
        name: 'Warm Caches',
        action: () => this.warmCaches(),
        rollback: null,
      },
      {
        name: 'Start Services',
        action: () => this.startServices(),
        rollback: () => this.stopServices(),
      },
      {
        name: 'Run Health Checks',
        action: () => this.runHealthChecks(),
        rollback: null,
      },
    ];

    const executedSteps: RecoveryStep[] = [];

    try {
```

```
    for (const step of steps) {
      this.logger.info(`Executing recovery step: ${step.name}`);

      await step.action();
      executedSteps.push(step);

      this.notifyProgress({
        step: step.name,
        status: 'completed',
        progress: (executedSteps.length / steps.length) * 100,
      });
    }

    return {
      success: true,
      duration: Date.now() - startTime,
      stepsCompleted: steps.length,
    };

  } catch (error) {
    this.logger.error(`Recovery failed at step: ${currentStep}`, error);

    // Rollback executed steps in reverse order
    for (const step of executedSteps.reverse()) {
      if (step.rollback) {
        try {
          await step.rollback();
        } catch (rollbackError) {
          this.logger.error(`Rollback failed for step: ${step.name}`, rollbackError);
        }
      }
    }

    throw new RecoveryError(`Recovery failed: ${error.message}`, error);
  }
}

private async verifyDataIntegrity(): Promise<void> {
  const checks = [
    // Check referral graph consistency
    {
      name: 'Referral Graph Consistency',
      query: `
        SELECT COUNT(*) as orphaned
        FROM referrals r
        LEFT JOIN users u1 ON r.referrer_address = u1.address
        LEFT JOIN users u2 ON r.referee_address = u2.address
        WHERE u1.address IS NULL OR u2.address IS NULL
      `,
      expectedResult: (result) => result.rows[0].orphaned === 0,
    },

    // Check balance consistency
    {
      name: 'Balance Consistency',
```

```
        query: `
          SELECT
            SUM(balance) as total_balance,
            (SELECT total_supply FROM global_stats) as recorded_supply
          FROM user_balances
          WHERE chain_id = $1
        `,
        params: [1], // Solana
        expectedResult: (result) => {
          const diff = Math.abs(result.rows[0].total_balance - result.rows[0].recorded_supply);
          return diff < 0.01; // Allow 0.01 token difference
        },
      },

      // Check for circular references
      {
        name: 'Circular Reference Check',
        query: `
          WITH RECURSIVE ref_path AS (
            SELECT
              referrer_address,
              referee_address,
              ARRAY[referrer_address, referee_address] as path
            FROM referrals

            UNION ALL

            SELECT
              rp.referrer_address,
              r.referee_address,
              rp.path || r.referee_address
            FROM ref_path rp
            JOIN referrals r ON rp.referee_address = r.referrer_address
            WHERE NOT r.referee_address = ANY(rp.path)
              AND array_length(rp.path, 1) < 10
          )
          SELECT COUNT(*) as circular_refs
          FROM ref_path
          WHERE referrer_address = referee_address
        `,
        expectedResult: (result) => result.rows[0].circular_refs === 0,
      },
    ];

    for (const check of checks) {
      const result = await this.db.query(check.query, check.params || []);

      if (!check.expectedResult(result)) {
        throw new IntegrityError(`Data integrity check failed: ${check.name}`);
      }
    }
  }
}
```

3. APPENDICES

A. Technology Stack Summary

- **Frontend**: Next.js 14, TypeScript, Tailwind CSS, Framer Motion
- **Backend**: Node.js, GraphQL, REST API
- **Databases**: PostgreSQL, TimescaleDB, Redis, Neo4j
- **Blockchain**: Solana Web3.js, Ethers.js, LayerZero
- **Infrastructure**: Kubernetes, Docker, AWS/GCP
- **Monitoring**: Prometheus, Grafana, OpenTelemetry

B. Performance Benchmarks

- Page Load: <2s (P95)
- API Response: <200ms (P95)
- WebSocket Latency: <50ms
- Concurrent Users: 10,000+
- Transactions/sec: 1,000+

C. Security Compliance

- SOC 2 Type II ready
- GDPR compliant architecture
- Smart contract audits required
- Penetration testing quarterly

D. Estimated Resources

- Development Team: 8-10 engineers
- Timeline: 4-6 months
- Infrastructure Cost: $10-20k/month
- Third-party Services: $5-10k/month