Altheia Software Requirements Specification

Executive Summary

Altheia is a decentralized protocol designed to bring standardized risk and yield assessment to tokenized real-world assets (RWAs) in the DeFi ecosystem. The platform enables a network of validators to evaluate assets like tokenized real estate, invoices, and commodities based on transparent scoring methodologies. By leveraging blockchain technology, oracles, and a validation network, Altheia aims to become the “Standard & Poor’s for DeFi’s real-world asset layer.”

This Software Requirements Specification (SRS) document provides comprehensive technical guidance for developing the Altheia platform, covering all aspects from smart contracts to user interfaces. It is structured to serve as a complete reference for blockchain engineers, backend developers, and frontend developers working on the project.

Table of Contents

1. Introduction

1.1 Problem Statement

Real-world assets (RWAs) like real estate, invoices, and commodities are being tokenized on-chain. However, there is no standardized, verifiable risk rating system for these assets, making it difficult for investors to trust or compare them across protocols. Furthermore, yield estimates are scattered, unverifiable, and lack provenance.

1.2 Goal

Altheia aims to build a decentralized, transparent protocol to assess the risk and yield profile of tokenized RWAs through a network of validators, oracles, and curated scoring models.

1.3 Vision

To be the Standard & Poor's for DeFi's real-world asset layer — open, programmable, and credibly neutral.

1.4 Target Audience

- Web3 developers and builders
- Investors and institutions onboarding to RWA DeFi
- KYC and Oracle providers
- Tokenization platforms
- Validators and scoring professionals

2. Background Concepts

2.1 What is a Real World Asset (RWA)?

A real-world asset refers to a tangible or off-chain asset that is brought on-chain via tokenization. Examples:

- Real estate
- Invoices and receivables
- Gold, art, commodities

These assets can be represented as ERC721 (NFT) or ERC20 tokens with off-chain metadata.

Read More:

- Centrifuge RWAs
- Tokenization 101 – Consensys

2.2 Tokenization & Smart Contracts

Tokenization involves issuing a digital token that represents a share of the physical asset. This is done via smart contracts with metadata linking to:

- Legal ownership docs
- Geolocation info
- Asset-backed cashflows

Standards Used:

- ERC721 for unique assets
- ERC3643 for compliance
- ERC4626 for yield-bearing assets

2.3 Stakeholders in Altheia

Role	Description
Issuer	Submits tokenized asset for scoring
Validator	Reviews, scores and attests to RWA claims
Investor	Explores rated assets, allocates capital based on risk/yield preferences
Oracle	Fetches external data (e.g., deeds, history, APY, registry verification)
Curator	Optionally bundles RWAs into portfolios by theme, geography or score
Admin	Manages registry, integrations, and protocol-level parameters

3. Protocol Architecture Overview

3.1 Core Modules

- **Asset Submission Module**: Issuer uploads RWA, jurisdiction info, docs, token ref
- **KYC Integration**: Verifies issuer identity using services like Persona, Sumsub
- **Validator Registry**: List of approved validators with jurisdictional

alignment
- **Scoring Engine**: Aggregates validator input, oracle data to produce a canonical risk score
- **Yield Oracle**: Chainlink Functions fetch external APY/cashflow data
- **Score Feed API**: Returns risk/yield score in a consistent JSON format
- **Portfolio Builder**: Allows filtering & bundling assets by risk tier/geography

3.2 On-chain & Off-chain Boundary

- **On-chain**: Asset metadata hash, score registry, validator staking & slashing
- **Off-chain**: KYC, jurisdictional legal doc validation, historical cashflow analysis

3.3 Chainlink Functions

Chainlink will:

- Query external property registries
- Fetch historical yields from institutions
- Validate document hashes or licensing status

4. Functional Requirements

4.1 Issuer Flow

- Must connect wallet and complete KYC
- Upload asset token reference (ERC721/4626/20), jurisdiction, APY

metadata
- Submit relevant documents (deeds, agreements, yield history)
- View dashboard of submitted and rated assets

4.2 Validator Flow

- Register as validator by staking native token or passing eligibility criteria
- Assigned assets based on jurisdiction
- Review asset off-chain docs, run risk model, sign and submit score + justification
- Can be slashed for false or malicious scores

4.3 Investor Flow

- Browse asset marketplace
- Filter by risk, APY, geography
- Access risk report, validator history, oracle data
- Allocate capital externally to lending/issuance platforms or bundle assets into a managed portfolio

4.4 Curator Flow (Optional)

- Create custom portfolios based on strategies (e.g. low risk Indian real estate)
- Stake curation token to register bundle
- Earn share from investor deposits if adopted

4.5 Oracle & Off-Chain System Integration

- Chainlink Functions fetch jurisdiction registry, validator licenses, APY data
- Optional integration with:
    - Goldfinch for off-chain borrower profiles
    - Clearpool for historical repayment trends
    - Local government APIs (e.g., Indian land records, US county APIs)

4.6 Geographic & Regulatory Considerations

- Asset validation logic may differ by country
- Regional KYC data and licensing will be mapped per validator and issuer
- Jurisdiction-specific oracles integrated via Chainlink Functions for attested government APIs
- Validator and Issuer onboarding flows will vary depending on data availability and legal boundaries

5. User Flows

5.1 Issuer User Flow

1. **Wallet Connect**: User connects EVM wallet
2. **KYC Verification**: Redirected to Persona/Sumsub for identity verification
3. **Submit Tokenized Asset**:
    - Upload ERC token address
    - Choose asset class and jurisdiction
    - Upload documents: property deed, cashflow history, APY metadata
4. **Await Scoring**: Validator assigned automatically

5. **Receive Score**: Risk and APY data returned to issuer dashboard

5.2 Validator Flow

1. **Apply as Validator**: Submit credentials, jurisdiction, stake tokens
2. **Asset Assignment**: Platform matches validator by jurisdictional fit
3. **Document Review**: Off-chain examination and scoring
4. **Score Submission**:
    - Submit signed score and report hash
    - Slashable if inconsistent with majority score or data manipulation

5.3 Investor Flow

1. **Marketplace Access**: Browse assets with filters (risk, APY, location)
2. **Due Diligence**: View risk report, validator details, and oracle inputs
3. **Allocate Capital**:
    - Optionally bundle assets via curator portfolios
    - Redirect to issuer/lending platform with risk context

5.4 Oracle Flow

1. **Off-Chain Data Requests**: Chainlink calls to:
    - Government registries
    - Real estate databases
    - Bank/payment history APIs
2. **Standardized Response Format**: Returns JSON with attestation, timestamp, and hash of source

5.5 Curator Flow

1. **Stake to Curate**: Bond token to propose portfolio
2. **Portfolio Creation**: Pick assets based on theme/risk
3. **Public Display**: Investors subscribe
4. **Revenue Share**: Earns fees proportionate to portfolio allocation volume

6. Algorithms and Data Structures

6.1 Asset Data Structure

Assets in the Altheia platform are represented as structured data objects containing:

- Basic information (name, type, issuer)
- Financial metrics
- Risk assessment parameters
- Historical performance data
- Associated documents and certifications

6.1.1 Asset Object Schema (Backend)

```
interface Asset {
  id: string;                    // Unique identifier
  name: string;                  // Asset name
  type: AssetType;               // Enum: RealEstate, Invoice,
Commodity, Other
  issuer: {
    id: string;                  // Issuer wallet address
    name: string;                // Organization or individual
```

```
name
    jurisdiction: string;         // ISO country code
    kycVerified: boolean;         // Verification status
    credibilityScore: number;     // 0-100 score based on
history
  };
  financialMetrics: {
    expectedYield: number;        // Annual percentage (e.g.,
7.5%)
    historicalYield?: number[];   // Time series data of past
performance
    maturityDate?: Date;          // For fixed-term assets
    valuationCurrency: string;    // USD, EUR, etc.
    valuation: number;            // Current valuation in
specified currency
    paymentFrequency?: string;    // Monthly, quarterly, annual,
etc.
    occupancyRate?: number;       // For real estate (0-100%)
    debtToEquityRatio?: number;   // Leverage indicator
  };
  riskParameters: {
    liquidityScore: number;       // 0-100 (how easily
convertible to cash)
    volatilityScore: number;      // 0-100 (price stability)
    jurisdictionRiskScore: number; // Country/region risk
    collateralization?: number;   // For collateralized assets
(%)
    loanToValueRatio?: number;    // For debt instruments (%)
    defaultRisk: number;          // 0-100 (probability of
default)
  };
  performance: {
    historicalData: {
      timestamps: number[];       // UNIX timestamps
      values: number[];           // Value at corresponding
```

```
timestamp
    };
    benchmarkComparison?: {
      benchmarkId: string;
      outperformance: number;     // % above/below benchmark
    }[];
  };
  documents: {
    legalDocuments: {             // Legal proof of ownership,
etc.
      hash: string;               // IPFS hash
      type: string;               // Deed, contract,
certificate, etc.
      verificationStatus: boolean; // Verified by
oracle/validator
    }[];
    certifications: {
      type: string;               // Regulatory approval, audit,
etc.
      issuer: string;             // Organization that issued
certification
      expiryDate?: Date;          // When certification expires
      hash: string;               // IPFS hash of certification
document
    }[];
  };
  metadata: {
    createdAt: Date;
    updatedAt: Date;
    blockchain: string;           // Ethereum, Polygon, etc.
    tokenContract: string;        // Contract address
    tokenId?: string;             // For NFTs
    ipfsMetadataHash: string;     // IPFS hash of complete
metadata
  };
```

```
  status: AssetStatus;              // Pending, InReview, Scored,
Rejected
  riskScore?: {
    overall: number;                // 0-100 (higher = riskier)
    components: {
      category: string;             // Category of risk
      score: number;                // Score for this category
      weight: number;               // Weight in overall
calculation
    }[];
    analystId: string;              // Validator who assessed
    timestamp: Date;                // When score was calculated
    confidence: number;             // Confidence level in score
(0-100)
  };
}

enum AssetType {
  RealEstate = 'real_estate',
  Invoice = 'invoice',
  Commodity = 'commodity',
  Security = 'security',
  Loan = 'loan',
  Account = 'account_receivable',
  Other = 'other'
}

enum AssetStatus {
  Pending = 'pending',
  InReview = 'in_review',
  Scored = 'scored',
  Rejected = 'rejected'
}
```

6.1.2 On-Chain vs Off-Chain Data Storage Strategy

Data Category	Storage Location	Rationale
Basic Asset Info	On-Chain	Core information needed for smart contract execution
Asset Identifier	On-Chain	ERC721/ERC20 token reference
Ownership	On-Chain	Essential for trustless verification
Risk Score	On-Chain	Final calculated score needed for on-chain actions
Document Hashes	On-Chain	Verification proof without exposing content
KYC Status	On-Chain	Binary verification without personal data
Full Documents	Off-Chain (IPFS)	Privacy concerns and gas cost limitations
Historical Data	Off-Chain DB	Volume of data makes on-chain storage impractical
Analytics	Off-Chain DB	Computed data not required for consensus
Detailed Reports	Off-Chain (IPFS)	Size constraints, with on-chain hash reference

6.2 Risk Scoring Algorithm

The risk scoring system utilizes a multi-factor analysis considering:

- Historical performance metrics
- Market volatility indicators
- Issuer credibility scores
- External rating agency inputs
- Environmental and social impact factors

6.2.1 Detailed Risk Scoring Methodology

The risk score is calculated using a weighted average of multiple risk factors:

```
// Pseudocode for risk score calculation
function calculateRiskScore(asset: Asset, marketData:
MarketData, issuerData: IssuerData): RiskScore {
  // 1. Initialize component scores
  const components = [
    // Historical performance (lower performance = higher risk)
    {
      category: "historical_performance",
      score:
calculateHistoricalPerformanceRisk(asset.performance.historical
Data),
      weight: 0.25
    },

    // Market volatility
    {
      category: "market_volatility",
      score: calculateMarketVolatilityRisk(asset.type,
marketData),
```

```
      weight: 0.15
    },

    // Issuer credibility
    {
      category: "issuer_credibility",
      score: calculateIssuerCredibilityRisk(asset.issuer,
issuerData),
      weight: 0.20
    },

    // Jurisdiction risk
    {
      category: "jurisdiction",
      score:
calculateJurisdictionRisk(asset.issuer.jurisdiction),
      weight: 0.10
    },

    // Asset type risk (inherent to asset class)
    {
      category: "asset_type",
      score: getBaseRiskForAssetType(asset.type),
      weight: 0.15
    },

    // Liquidity risk
    {
      category: "liquidity",
      score: calculateLiquidityRisk(asset),
      weight: 0.15
    }
  ];

  // Add asset-specific components based on type
```

```
  if (asset.type === AssetType.RealEstate) {
    components.push({
      category: "occupancy",
      score:
calculateOccupancyRisk(asset.financialMetrics.occupancyRate),
      weight: 0.10
    });
    // Adjust weights to ensure sum = 1
    normalizeWeights(components);
  }

  // 2. Calculate weighted score
  const overallScore = components.reduce(
    (sum, component) => sum + (component.score *
component.weight),
    0
  );

  // 3. Calculate confidence level based on data quality
  const confidence = calculateConfidenceLevel(asset,
components);

  return {
    overall: Math.round(overallScore),
    components,
    analystId: getCurrentValidator().id,
    timestamp: new Date(),
    confidence
  };
}
```

6.2.2 Risk Factor Calculation Methods

Historical Performance Risk

```
function calculateHistoricalPerformanceRisk(historicalData:
{timestamps: number[], values: number[]}): number {
  if (!historicalData || historicalData.values.length < 2) {
    return 50; // Neutral score for insufficient data
  }

  // Calculate volatility using standard deviation of returns
  const returns = calculateReturns(historicalData.values);
  const volatility = calculateStandardDeviation(returns);

  // Calculate downside deviation (only negative returns)
  const downsideReturns = returns.filter(r => r < 0);
  const downsideDeviation = downsideReturns.length > 0
    ? calculateStandardDeviation(downsideReturns)
    : 0;

  // Calculate average return
  const avgReturn = calculateAverage(returns);

  // Higher volatility, higher downside deviation, and lower
returns = higher risk
  const volatilityComponent = mapToScale(volatility, 0, 0.5, 0,
100);
  const downsideComponent = mapToScale(downsideDeviation, 0,
0.3, 0, 100);
  const returnComponent = mapToScale(avgReturn, -0.1, 0.3, 100,
0);

  // Weighted combination of components
  return (volatilityComponent * 0.4) + (downsideComponent *
0.4) + (returnComponent * 0.2);
}
```

Market Volatility Risk

```
function calculateMarketVolatilityRisk(assetType: AssetType,
marketData: MarketData): number {
  // Get relevant market index based on asset type
  const relevantIndex = getRelevantMarketIndex(assetType,
marketData);

  // Calculate market volatility
  const marketVolatility =
calculateVolatility(relevantIndex.historicalValues);

  // Map volatility to 0-100 scale
  return mapToScale(marketVolatility, 0, 0.3, 0, 100);
}
```

Issuer Credibility Risk

```
function calculateIssuerCredibilityRisk(issuer: Issuer,
issuerData: IssuerData): number {
  // Factors affecting issuer credibility
  const factors = [
    // KYC verification status
    issuer.kycVerified ? 0 : 100,

    // Previous assets performance
    calculatePreviousAssetsPerformance(issuer.id, issuerData),

    // Length of history
    calculateHistoryFactor(issuer.id, issuerData),

    // External ratings if available
    issuerData.externalRatings ?
mapExternalRatingToRiskScore(issuerData.externalRatings) : 50
  ];

  // Average of factors
  return factors.reduce((sum, factor) => sum + factor, 0) /
factors.length;
}
```

6.2.3 Risk Score Calibration

The risk scoring system is calibrated using:

1. **Training Data Set**:
    - Historical performance of similar assets across multiple markets
    - Known defaults and success cases as ground truth
    - Market benchmark indices for each asset class

2. **Validator Consensus Mechanism**:
    - Multiple validators score the same asset
    - Outlier detection with Tukey's method (scores outside 1.5 * IQR)
    - Weighted average based on validator history and stake
3. **Periodic Recalibration Process**:
    - Quarterly review of scoring accuracy
    - Backtesting against actual asset outcomes
    - Model adjustment based on accuracy metrics

6.3 Data Validation Process

All submitted asset data undergoes a three-tier validation process:

1. Automated syntax and format checks
2. Statistical anomaly detection
3. Manual review by qualified validators

6.3.1 Automated Validation Rules

Syntax and Format Checks

```
// Example validation schema for asset submission
const assetSubmissionSchema = {
  name: {
    type: 'string',
    required: true,
    minLength: 3,
    maxLength: 100,
    pattern: '^[a-zA-Z0-9\\s\\-_]+{{content}}#x27;
  },
```

```
  type: {
    type: 'string',
    required: true,
    enum: Object.values(AssetType)
  },
  tokenContract: {
    type: 'string',
    required: true,
    pattern: '^0x[a-fA-F0-9]{40}{{content}}#x27;, // Ethereum
address format
    custom: validateContractExists
  },
  financialMetrics: {
    type: 'object',
    required: true,
    properties: {
      expectedYield: {
        type: 'number',
        required: true,
        min: 0,
        max: 100
      },
      valuationCurrency: {
        type: 'string',
        required: true,
        enum: ['USD', 'EUR', 'GBP', 'JPY', 'INR']
      },
      valuation: {
        type: 'number',
        required: true,
        min: 0
      }
      // Other metrics with appropriate validation
    }
  },
```

```
  documents: {
    type: 'array',
    required: true,
    minItems: 1,
    items: {
      type: 'object',
      properties: {
        hash: {
          type: 'string',
          required: true,
          pattern: '^[a-zA-Z0-9]+{{content}}#x27;
        },
        type: {
          type: 'string',
          required: true
        }
      }
    }
  }
};
```

Data Consistency Checks

```
// Example consistency validation function
function validateAssetConsistency(asset: Asset):
ValidationResult {
  const issues = [];

  // Check that dates are chronologically valid
  if (asset.financialMetrics.maturityDate &&
      new Date(asset.financialMetrics.maturityDate) <= new
Date()) {
    issues.push('Maturity date must be in the future');
  }
```

```
  // Check that historical data is chronologically ordered
  if (asset.performance?.historicalData?.timestamps) {
    for (let i = 1; i < 
asset.performance.historicalData.timestamps.length; i++) {
      if (asset.performance.historicalData.timestamps[i] <= 
          asset.performance.historicalData.timestamps[i-1]) {
        issues.push('Historical data timestamps must be in 
ascending order');
        break;
      }
    }
  }

  // Verify that arrays have matching lengths
  if (asset.performance?.historicalData?.timestamps?.length !== 
      asset.performance?.historicalData?.values?.length) {
    issues.push('Historical data timestamps and values arrays 
must have the same length');
  }

  return {
    valid: issues.length === 0,
    issues
  };
}
```

6.3.2 Statistical Anomaly Detection

The system employs several statistical methods to identify potentially fraudulent or erroneous data:

1. **Z-Score Analysis**:

- Calculate Z-scores for numerical metrics against peer group
- Flag outliers beyond 3 standard deviations

2. **Benford's Law Analysis**:
   - Apply to financial figures to detect fabricated numbers
   - Particularly useful for invoice and cashflow validation
3. **Time Series Consistency**:
   - Check for unnatural patterns in historical data
   - Detect sudden changes inconsistent with asset class behavior
4. **Cross-Reference Validation**:
   - Compare submitted data with external sources where available
   - Flag significant discrepancies for manual review

```
// Example anomaly detection function for yield data
function detectYieldAnomalies(asset: Asset, peerAssets:
Asset[]): AnomalyResult {
  // Extract yields from peer group (same asset type and
jurisdiction)
  const peerYields = peerAssets
    .filter(p => p.type === asset.type && p.issuer.jurisdiction
=== asset.issuer.jurisdiction)
    .map(p => p.financialMetrics.expectedYield);

  if (peerYields.length < 5) {
    return { hasAnomalies: false, message: 'Insufficient peer
data for comparison' };
  }

  // Calculate statistics
```

```
  const mean = calculateMean(peerYields);
  const stdDev = calculateStdDev(peerYields);

  // Calculate z-score
  const zScore = Math.abs((asset.financialMetrics.expectedYield
- mean) / stdDev);

  // Check for anomaly
  if (zScore > 3) {
    return {
      hasAnomalies: true,
      message: `Expected yield
(${asset.financialMetrics.expectedYield}%) is unusually ${
        asset.financialMetrics.expectedYield > mean ? 'high' :
'low'
      } compared to similar assets (avg: ${mean.toFixed(2)}%)`,
      severity: zScore > 5 ? 'high' : 'medium',
      details: {
        metric: 'expectedYield',
        value: asset.financialMetrics.expectedYield,
        mean,
        stdDev,
        zScore
      }
    };
  }

  return { hasAnomalies: false };
}
```

6.3.3 Manual Validator Review Process

Qualified validators follow a structured review process:

1. **Document Verification**:
    - Authenticate document existence and validity
    - Cross-reference with official sources where possible
    - Verify document hashes match IPFS content
2. **Financial Assessment**:
    - Apply domain expertise to evaluate financial metrics
    - Validate yield projections against market conditions
    - Assess risk factors specific to asset class and jurisdiction
3. **Report Generation**:
    - Produce standardized scoring report with justifications
    - Document assumptions and limitations
    - Highlight areas of uncertainty
4. **Consensus Building**:
    - Multiple validators review high-value or complex assets
    - Differences beyond threshold trigger dispute resolution
    - Final score aggregated from validated reviews

7. User Interface Requirements

7.1 General UI Requirements

- Clean, modern interface adhering to material design principles
- Responsive design compatible with desktop, tablet, and mobile devices
- Accessibility compliance with WCAG 2.1 AA standards

- Dark/light mode toggle functionality
- Consistent branding elements across all pages

7.2 Dashboard Requirements

- Role-specific dashboards for investors, issuers, analysts, and validators
- Customizable widget layout for personalized experience
- Real-time data updates and notifications
- Interactive data visualizations for risk-yield analysis
- Filtering and sorting capabilities for all data tables

7.3 Asset Submission Interface

- Multi-step submission form with progress tracking
- Document upload functionality with support for common file formats
- Auto-save capability for partial submissions
- In-line validation with helpful error messages
- Preview option before final submission

7.4 Analytics Interface

- Interactive charts for risk-yield comparison
- Historical performance tracking with customizable date ranges
- Correlation analysis between different asset metrics
- Export functionality for reports and raw data
- Benchmark comparison capabilities

8. Security Requirements

8.1 Authentication and Authorization

- Web3 wallet integration for decentralized authentication
- Role-based access control system
- Multi-factor authentication option for critical operations
- Session management with automatic timeout
- Audit logging of all authentication events

8.2 Data Security

- End-to-end encryption for all sensitive data
- Secure storage of cryptographic keys
- Data anonymization for reporting and analytics
- Regular security audits and penetration testing
- GDPR and CCPA compliance measures

8.3 Smart Contract Security

- Formal verification of all deployed contracts
- Rate limiting to prevent denial-of-service attacks
- Multi-signature requirements for critical operations
- Secure upgrade paths for contract versions
- External security audit certification

9. Performance Requirements

9.1 Response Time

- Page load time under 2 seconds for standard operations
- API response time under 500ms for common queries
- Real-time data synchronization with blockchain (≤15 second delay)
- Search results returned within 1 second
- Report generation completed within 5 seconds

9.2 Scalability

- Support for up to 10,000 concurrent users
- Capability to handle 100,000 assets in the database
- Elastic scaling during peak usage periods
- Performance degradation not exceeding 10% at maximum load
- Batch processing capabilities for high-volume operations

9.3 Availability

- 99.9% uptime guarantee during business hours
- Scheduled maintenance windows with advance notice
- Automated failover mechanisms
- Geographic redundancy for critical services
- Real-time system health monitoring and alerts

10. Compliance and Regulatory Requirements

10.1 Financial Regulations

- Compliance with SEC reporting requirements
- KYC/AML verification processes

- Support for international financial reporting standards (IFRS)
- Audit trails for all financial transactions
- Regulatory reporting automation

10.2 Data Protection

- GDPR compliance for EU users
- CCPA compliance for California residents
- Data retention policies with automatic enforcement
- Consent management for user data processing
- Data subject access request (DSAR) handling system

10.3 Industry Standards

- Compliance with ISO 27001 for information security
- Adherence to AICPA SOC 2 controls
- Implementation of NIST cybersecurity framework
- Support for OpenID Connect authentication standards
- Blockchain interoperability standards compliance

11. Integration Requirements

11.1 Blockchain Integration

- Support for Ethereum and compatible EVM chains
- Integration with popular Web3 wallets (MetaMask, WalletConnect)
- Smart contract event listeners for real-time updates
- Transaction signing and verification interfaces

- Gas fee estimation and optimization

11.2 External Data Sources

- Integration with market data providers
- Connections to credit rating agency APIs
- ESG data source integration
- Financial news aggregation services
- Regulatory database access

11.3 Export/Import Capabilities

- Support for CSV, JSON, and Excel file formats
- Automated data import validation
- Scheduled data export functionality
- API-based data exchange capabilities
- Blockchain transaction history exports

12. User Experience Requirements

12.1 Onboarding Process

- Interactive tutorial for first-time users
- Role-specific guidance and tooltips
- Progressive disclosure of complex features
- Quick-start templates for common operations
- Contextual help resources

12.2 Feedback Mechanisms

- In-app feedback collection tools
- Issue reporting with screenshot capability
- Usage analytics to identify friction points
- A/B testing framework for UX improvements
- User satisfaction measurement

12.3 Accessibility Features

- Screen reader compatibility
- Keyboard navigation support
- Color contrast compliance
- Resizable text and UI elements
- Alternative text for all images and charts

13. Documentation Requirements

13.1 User Documentation

- Role-based user manuals
- Interactive walkthrough guides
- Searchable knowledge base
- Video tutorials for complex procedures
- Frequently asked questions (FAQ) section

13.2 Technical Documentation

- API reference documentation
- Smart contract specifications
- System architecture diagrams
- Database schema documentation
- Integration guides for third-party developers

13.3 Operational Documentation

- Installation and deployment guides
- Backup and recovery procedures
- Performance tuning recommendations
- Security best practices
- Troubleshooting guides

14. Testing Requirements

14.1 Unit Testing

- Comprehensive test coverage for all business logic
- Automated test suite for continuous integration
- Mocking framework for external dependencies
- Test-driven development approach for critical components
- Regular regression testing

14.2 Integration Testing

- End-to-end testing of user workflows
- API contract testing

- Blockchain interaction testing
- Cross-browser and cross-device testing
- Performance and load testing

14.3 User Acceptance Testing

- Role-based testing scenarios
- Beta testing program for early adopters
- Usability testing with representative users
- Accessibility compliance testing
- Security penetration testing

6.4 Issuer Module

6.4.1 Data Structures

```
struct Issuer {
    address walletAddress;
    string kycId;           // KYC provider unique ID
    string jurisdiction;    // ISO country code (e.g., IN, US)
    bool isVerified;        // KYC completed or not
    uint256[] submittedAssetIds;
}

struct Asset {
    uint256 assetId;
    address issuer;
    string tokenAddress;    // ERC721/ERC20 contract address
representing RWA
    string assetType;       // e.g., RealEstate, Invoice,
Commodity
    string jurisdiction;    // Asset location country code
    string documentHashes; // IPFS or other hash of uploaded
docs (JSON encoded)
    uint256 timestamp;      // submission time
    uint256 latestRiskScoreId;
    uint256 latestYieldScoreId;
}
```

6.4.2 Issuer Onboarding Algorithm

1. **Connect Wallet**
2. **KYC Flow**: Redirect user to integrated KYC provider (e.g., Persona).
3. **Receive KYC callback** with `kycId` and `verificationStatus`.
4. **Save Issuer info**:
    - walletAddress
    - kycId
    - jurisdiction (from KYC or user input)

- isVerified = true if KYC passed

5. **Allow asset submission** only if `isVerified == true`.
6. **On Asset Submission**:
    - Validate token contract address conforms to ERC standards.
    - Store metadata hashes.
    - Store timestamp and jurisdiction.
    - Emit event for new asset submission.

6.5 Validator Module

6.5.1 Data Structures

```
struct Validator {
    address walletAddress;
    string jurisdiction;  // Validator's area of expertise or
legal domain
    uint256 stakeAmount;  // tokens staked as collateral
    bool isActive;        // validated eligibility
    uint256[] reviewedAssetIds;
}

struct Score {
    uint256 scoreId;
    uint256 assetId;
    address validator;
    uint8 riskScore;       // 0-100 scale risk rating
    uint8 yieldScore;      // 0-100 scale yield confidence
    string justificationHash;  // IPFS hash of off-chain
detailed report
    uint256 timestamp;
}
```

6.5.2 Validator Scoring Algorithm

1. **Validator Registration**:
    - Stake required native tokens.
    - Provide jurisdiction and credentials off-chain.
    - Platform admin or DAO verifies validator eligibility.
    - Set `isActive = true`.
2. **Asset Assignment**:
    - Match asset jurisdiction with validator jurisdiction.
    - Assign asset to validator for review.

3. **Scoring Process**:
    - Validator downloads off-chain docs via IPFS hash.
    - Runs risk/yield analysis (off-chain).
    - Generates report, uploads to IPFS, stores hash.
    - Submits riskScore, yieldScore, and report hash on-chain.
4. **Score Verification**:
    - Platform aggregates multiple validator scores.
    - Flags outliers.
    - Slashing conditions triggered for false reports.

7. APIs & Interfaces

7.1 Public APIs

- `GET /assets`: List all tokenized RWA assets with metadata
- `GET /asset/{id}`: Full risk and yield profile
- `GET /validators`: Available and active validators
- `GET /portfolios`: Curated bundles
- `GET /score-feed/{tokenAddress}`: Canonical Chainlink-compatible feed for any RWA

7.2 Internal APIs (Web2 to Web3 bridges)

- `POST /issuer/kyc`: Accepts webhook from KYC provider with verified user info
- `POST /validator/submit-score`: Signed risk/yield score + IPFS report

hash

- `POST /oracle/submit-proof`: Oracle submission JSON

---

8. Smart Contract Architecture

8.1 Contract Types

Contract	Purpose
`AltheaIssuer.sol`	Handles issuer KYC, asset submission
`AltheaValidator.sol`	Validator registry, staking, score review
`AltheaScoreRegistry.sol`	Stores canonical risk/yield scores
`AltheaOracleBridge.sol`	Accepts Chainlink proof payloads
`PortfolioFactory.sol`	Curator portfolio creation and staking

8.2 Access Control

- Role-based access (e.g., Issuer, Validator, Curator)
- Multi-sig for admin updates
- Upgradable contracts (UUPS pattern)

---

9. Security Considerations

- Validators must stake tokens (slashing for false scores)
- Issuer KYC mandatory
- Chainlink oracle proofs only accepted with attestation
- Off-chain metadata must be IPFS-verified
- Score disputes resolved via majority quorum logic

---

10. Sample Data Schemas

10.1 Asset Metadata (IPFS JSON)

```
{
  "assetType": "RealEstate",
  "jurisdiction": "IN",
  "location": "Bangalore, India",
  "deedHash": "QmXYZ123...",
  "cashflowHistory": "QmABC456...",
  "apy": "7.5"
}
```

10.2 Score Submission

```
{
  "assetId": 112,
  "validator": "0xAbc...",
  "riskScore": 42,
  "yieldScore": 75,
  "ipfsReportHash": "QmReport123..."
}
```

---

11. Deployment Strategy

- Phase 1: Testnet deployment with mock Chainlink functions
- Phase 2: Regional pilot with India/US real estate use cases
- Phase 3: Mainnet launch with validator DAO governance

---

12. Monitoring & Metrics

- Risk score variance tracking
- Validator score history & slashing logs
- Issuer KYC pass/fail ratio
- Oracle response latency
- Smart contract gas usage trends

---

13. Future Extensions

- ZK-KYC for anonymous but verified issuers
- Score prediction ML models based on oracle + validator data
- DAO-based validator onboarding & slashing decisions
- Tokenized rating agency backed by Althea scores

---

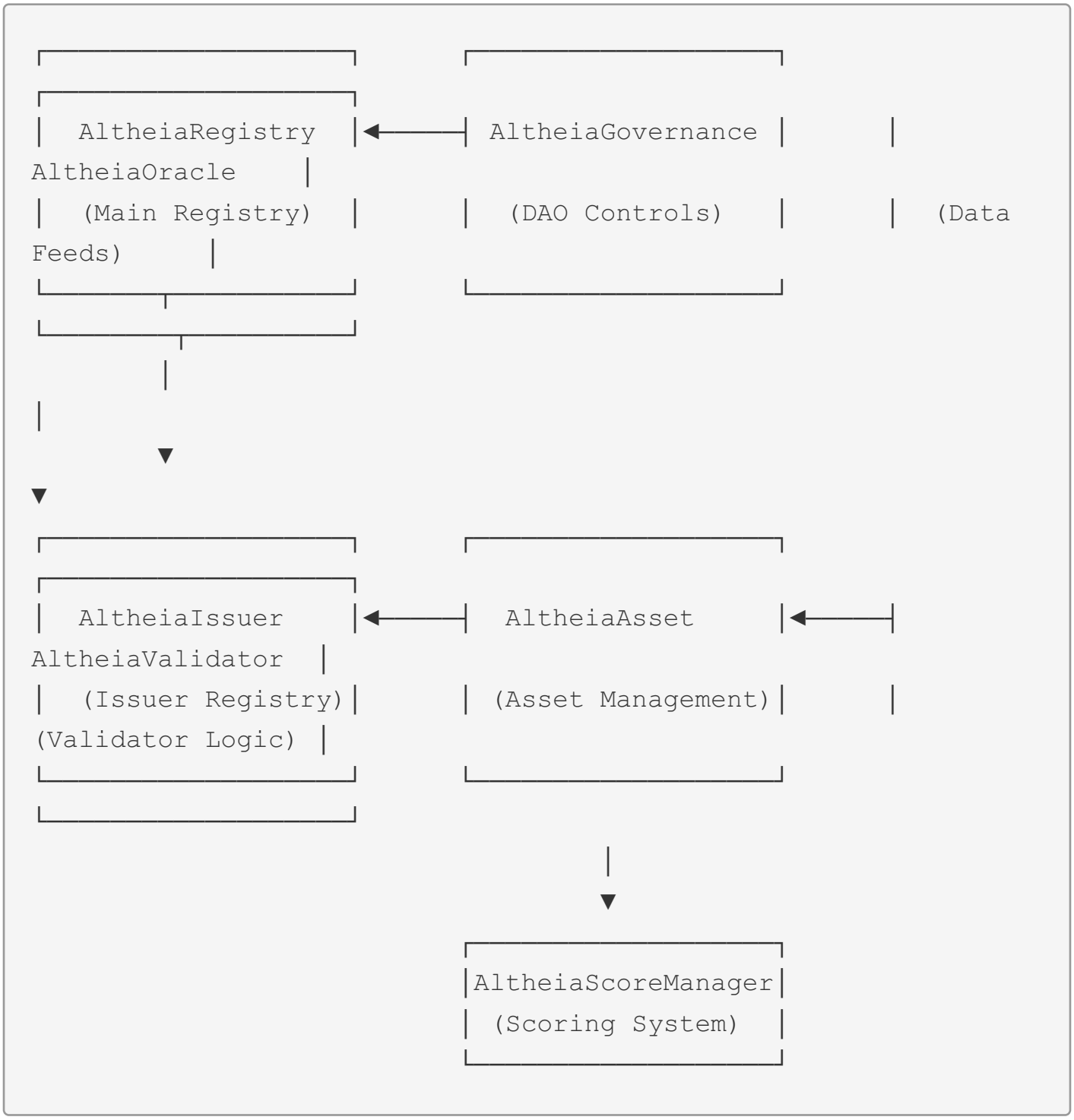
14. Developer Implementation Guide

This section provides detailed technical guidance for different developers working on the Altheia platform.

14.1 Smart Contract Developer Guide

14.1.1 Contract Architecture Overview

The following diagram illustrates the relationship between smart contracts in the Altheia ecosystem:



14.1.2 Contract Implementation Specifications

AltheiaRegistry.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;
```

```
import "@openzeppelin/contracts-
upgradeable/access/AccessControlUpgradeable.sol";
import "@openzeppelin/contracts-
upgradeable/proxy/utils/Initializable.sol";
import "@openzeppelin/contracts-
upgradeable/proxy/utils/UUPSUpgradeable.sol";

/**
 * @title AltheiaRegistry
 * @dev Central registry contract that manages Altheia protocol
 */
contract AltheiaRegistry is Initializable,
AccessControlUpgradeable, UUPSUpgradeable {
    bytes32 public constant ADMIN_ROLE =
keccak256("ADMIN_ROLE");
    bytes32 public constant UPGRADER_ROLE =
keccak256("UPGRADER_ROLE");
    bytes32 public constant VALIDATOR_ROLE =
keccak256("VALIDATOR_ROLE");
    bytes32 public constant ISSUER_ROLE =
keccak256("ISSUER_ROLE");

    // Address mappings for protocol contracts
    address public issuerContract;
    address public assetContract;
    address public validatorContract;
    address public scoreManagerContract;
    address public oracleContract;
    address public governanceContract;

    // Protocol settings
    uint256 public validatorStakeAmount;
    uint256 public disputeThreshold;
    mapping(string => uint256) public jurisdictionRiskFactors;
```

```
    event ContractUpdated(string contractName, address indexed
contractAddress);
    event ProtocolSettingUpdated(string settingName, uint256
value);

    function initialize(address admin) public initializer {
        __AccessControl_init();
        __UUPSUpgradeable_init();

        _setupRole(DEFAULT_ADMIN_ROLE, admin);
        _setupRole(ADMIN_ROLE, admin);
        _setupRole(UPGRADER_ROLE, admin);

        // Initial settings
        validatorStakeAmount = 1000 ether; // Example amount
        disputeThreshold = 20; // 20% deviation triggers
dispute
    }

    function setContractAddress(
        string calldata contractName,
        address contractAddress
    ) external onlyRole(ADMIN_ROLE) {
        if (keccak256(abi.encodePacked(contractName)) ==
keccak256("issuer")) {
            issuerContract = contractAddress;
        } else if (keccak256(abi.encodePacked(contractName)) ==
keccak256("asset")) {
            assetContract = contractAddress;
        } else if (keccak256(abi.encodePacked(contractName)) ==
keccak256("validator")) {
            validatorContract = contractAddress;
        } else if (keccak256(abi.encodePacked(contractName)) ==
keccak256("scoreManager")) {
```

```
            scoreManagerContract = contractAddress;
        } else if (keccak256(abi.encodePacked(contractName)) ==
keccak256("oracle")) {
            oracleContract = contractAddress;
        } else if (keccak256(abi.encodePacked(contractName)) ==
keccak256("governance")) {
            governanceContract = contractAddress;
        } else {
            revert("Invalid contract name");
        }

        emit ContractUpdated(contractName, contractAddress);
    }

    function updateProtocolSetting(
        string calldata settingName,
        uint256 value
    ) external onlyRole(ADMIN_ROLE) {
        if (keccak256(abi.encodePacked(settingName)) ==
keccak256("validatorStake")) {
            validatorStakeAmount = value;
        } else if (keccak256(abi.encodePacked(settingName)) ==
keccak256("disputeThreshold")) {
            disputeThreshold = value;
        } else {
            revert("Invalid setting name");
        }

        emit ProtocolSettingUpdated(settingName, value);
    }

    function setJurisdictionRiskFactor(
        string calldata jurisdiction,
        uint256 riskFactor
    ) external onlyRole(ADMIN_ROLE) {
```

```
        require(riskFactor <= 100, "Risk factor must be between
0-100");
        jurisdictionRiskFactors[jurisdiction] = riskFactor;
    }

    function _authorizeUpgrade(address newImplementation)
internal override onlyRole(UPGRADER_ROLE) {}
}
```

AltheiaAsset.sol (Partial)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import "./AltheiaRegistry.sol";
import "@openzeppelin/contracts-
upgradeable/proxy/utils/Initializable.sol";

/**
 * @title AltheiaAsset
 * @dev Manages asset submissions and metadata
 */
contract AltheiaAsset is Initializable {
    AltheiaRegistry public registry;

    struct Asset {
        uint256 id;
        address issuer;
        string assetType;
        string jurisdiction;
        string metadataURI;  // IPFS URI to full asset metadata
        string tokenAddress;  // Address of ERC20/ERC721 token
representing RWA
        uint256 timestamp;
```

```
        uint256 scoreId;     // Latest score ID
        AssetStatus status;
    }

    enum AssetStatus {
        Pending,
        InReview,
        Scored,
        Rejected
    }

    uint256 public nextAssetId;
    mapping(uint256 => Asset) public assets;
    mapping(address => uint256[]) public issuerAssets;
    mapping(string => uint256[]) public assetsByType;
    mapping(string => uint256[]) public assetsByJurisdiction;

    event AssetSubmitted(uint256 indexed assetId, address
indexed issuer);
    event AssetStatusUpdated(uint256 indexed assetId,
AssetStatus status);

    function initialize(address registryAddress) public
initializer {
        registry = AltheiaRegistry(registryAddress);
        nextAssetId = 1;
    }

    function submitAsset(
        string calldata assetType,
        string calldata jurisdiction,
        string calldata metadataURI,
        string calldata tokenAddress
    ) external returns (uint256) {
        // Check if caller is a registered issuer
```

```
        require(
            registry.hasRole(registry.ISSUER_ROLE(),
msg.sender),
            "Caller is not a registered issuer"
        );

        uint256 assetId = nextAssetId;
        nextAssetId++;

        Asset memory newAsset = Asset({
            id: assetId,
            issuer: msg.sender,
            assetType: assetType,
            jurisdiction: jurisdiction,
            metadataURI: metadataURI,
            tokenAddress: tokenAddress,
            timestamp: block.timestamp,
            scoreId: 0,  // No score yet
            status: AssetStatus.Pending
        });

        assets[assetId] = newAsset;
        issuerAssets[msg.sender].push(assetId);
        assetsByType[assetType].push(assetId);
        assetsByJurisdiction[jurisdiction].push(assetId);

        emit AssetSubmitted(assetId, msg.sender);
        return assetId;
    }

    // Additional functions would include:
    // - updateAssetStatus
    // - getAssetDetails
    // - assignValidator
    // - linkScore
```

```
    // - getAssetsByFilters
}
```

AltheiaValidator.sol (Key Functions)

```
// Key functions in the validator contract
function stakeAndRegister(string calldata jurisdiction)
external payable {
    require(msg.value >= registry.validatorStakeAmount(),
"Insufficient stake");
    require(!validators[msg.sender].isRegistered, "Already
registered");

    validators[msg.sender] = Validator({
        walletAddress: msg.sender,
        jurisdiction: jurisdiction,
        stakeAmount: msg.value,
        isRegistered: true,
        isActive: false,  // Requires admin approval
        reputation: 50,   // Starting reputation score (0-100)
        lastActivityTime: 0,
        reviewedAssetCount: 0
    });

    emit ValidatorRegistered(msg.sender, jurisdiction);
}

function submitAssetScore(
    uint256 assetId,
    uint8 riskScore,
    uint8 yieldScore,
    string calldata reportURI
) external onlyActiveValidator {
    // Validate scores are in range 0-100
```

```
        require(riskScore <= 100, "Risk score out of range");
        require(yieldScore <= 100, "Yield score out of range");

        // Check if validator is assigned to this asset
        require(assetValidatorAssignments[assetId] == msg.sender,
"Not assigned to this asset");

        // Forward to score manager to store the score
        IAltheiaScoreManager scoreManager =
IAltheiaScoreManager(registry.scoreManagerContract());
        uint256 scoreId = scoreManager.recordScore(
            assetId,
            msg.sender,
            riskScore,
            yieldScore,
            reportURI
        );

        // Update validator stats
        validators[msg.sender].reviewedAssetCount++;
        validators[msg.sender].lastActivityTime = block.timestamp;

        emit ScoreSubmitted(assetId, msg.sender, scoreId);
    }
```

14.1.3 Deployment Sequence

Smart contracts should be deployed in this order:

1. Deploy `AltheiaRegistry` with admin multisig as owner
2. Deploy all peripheral contracts passing registry address to constructor
3. Register all contract addresses in the registry
4. Set initial protocol parameters
5. Add initial admin and validator roles

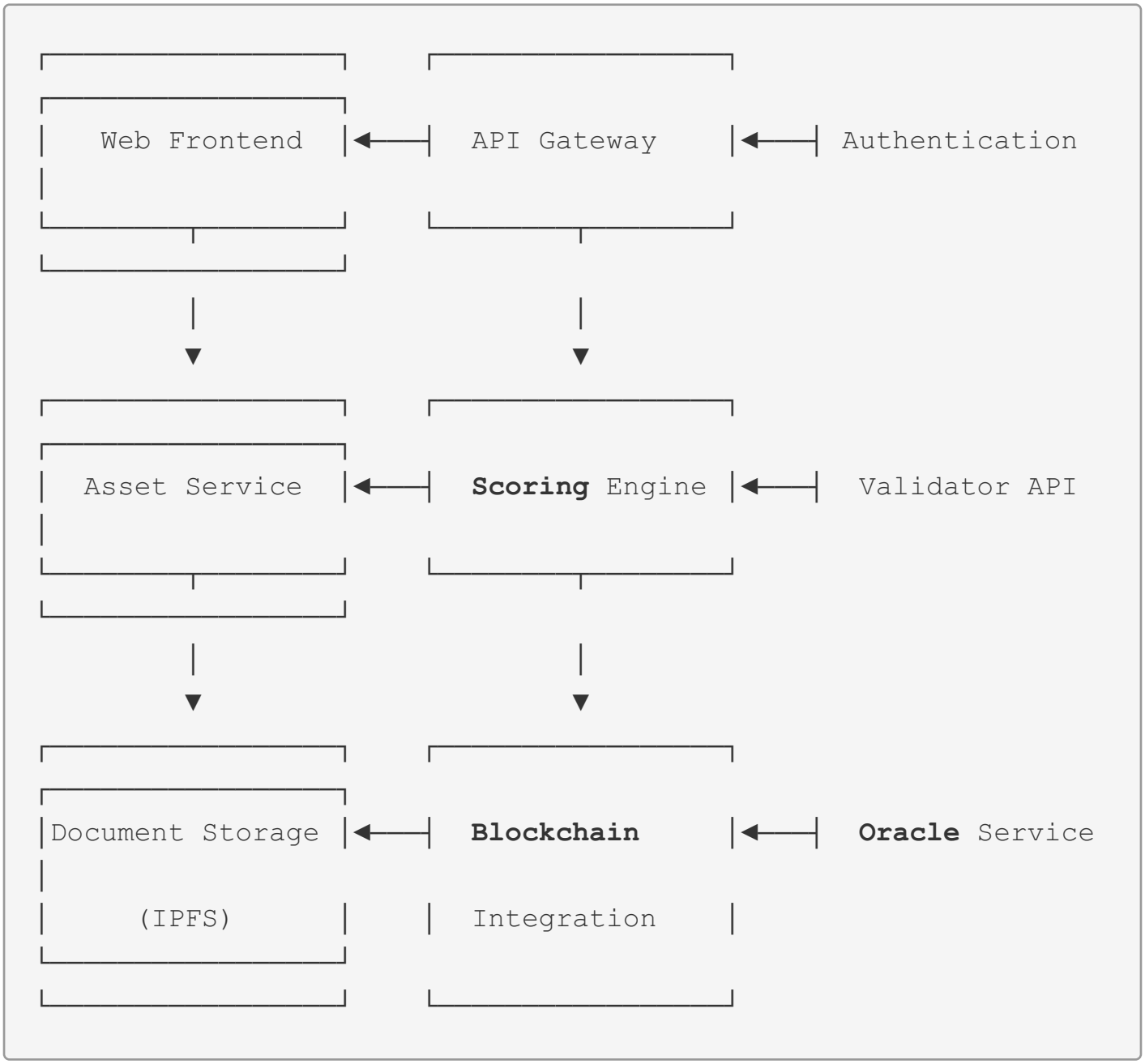
6. Transfer ownership to governance contract (once tested)

14.1.4 Security Considerations

1. **Re-entrancy Protection**: Use OpenZeppelin's ReentrancyGuard for state-changing functions
2. **Access Control**: Strictly enforce role-based access
3. **Proxies**: Use UUPS pattern for upgradeability
4. **Front-Running**: Use commit-reveal for sensitive operations
5. **Event Monitoring**: Implement comprehensive event emissions for off-chain monitoring
6. **Gas Optimization**: Batch operations when possible
7. **Formal Verification**: Verify critical functions using formal verification tools

14.2 Backend Developer Guide

14.2.1 System Architecture



14.2.2 API Endpoints

Asset Management

```
// Asset submission endpoint
// POST /api/assets
interface AssetSubmissionRequest {
  name: string;
  assetType: string;
```

```
  jurisdiction: string;
  tokenAddress: string;
  tokenId?: string;
  financialMetrics: {
    expectedYield: number;
    valuationCurrency: string;
    valuation: number;
    [key: string]: any;
  };
  documents: Array<{
    type: string;
    file: File;  // Binary file for upload
  }>;
}

interface AssetSubmissionResponse {
  success: boolean;
  assetId?: number;
  status?: string;
  transactionHash?: string;
  errors?: string[];
}

// GET /api/assets
interface AssetListRequest {
  page?: number;
  limit?: number;
  assetType?: string;
  jurisdiction?: string;
  status?: string;
  minRiskScore?: number;
  maxRiskScore?: number;
  minYield?: number;
  issuer?: string;
  sortBy?: 'yield' | 'risk' | 'date' | 'valuation';
```

```
  sortDirection?: 'asc' | 'desc';
}

interface AssetListResponse {
  assets: AssetSummary[];
  total: number;
  page: number;
  limit: number;
}

// GET /api/assets/:id
interface AssetDetailResponse {
  asset: Asset;  // Full asset object
  scores: Score[];
  analysts: AnalystSummary[];
  documents: DocumentReference[];
  relatedAssets?: AssetSummary[];
}
```

Validator API

```
// POST /api/validators/register
interface ValidatorRegistrationRequest {
  jurisdiction: string;
  credentials: {
    licenseNumber?: string;
    certifications?: string[];
    experience?: string;
  };
  walletAddress: string;
}

// POST /api/assets/:id/score
interface ScoreSubmissionRequest {
  riskScore: number;
  yieldScore: number;
  components: Array<{
    category: string;
    score: number;
    weight: number;
    justification: string;
  }>;
  report: File; // Detailed PDF report
}
```

Oracle Integration

```
// POST /api/oracles/submit
interface OracleSubmissionRequest {
  assetId: number;
  oracleType: 'property_registry' | 'yield_data' |
'market_data';
  data: Record<string, any>;
  sourceReference: string;
  attestation: {
    signature: string;
    timestamp: number;
    nodeId: string;
  };
}
```

14.2.3 Database Schema

PostgreSQL Schema

```
-- Users Table
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  wallet_address VARCHAR(42) NOT NULL UNIQUE,
  role VARCHAR(20) NOT NULL,
  jurisdiction VARCHAR(2),
  kyc_status BOOLEAN DEFAULT FALSE,
  kyc_provider VARCHAR(50),
  kyc_id VARCHAR(100),
  created_at TIMESTAMP DEFAULT NOW(),
  metadata JSONB
);

-- Assets Table
CREATE TABLE assets (
```

```
  id SERIAL PRIMARY KEY,
  blockchain_asset_id INTEGER,
  name VARCHAR(255) NOT NULL,
  asset_type VARCHAR(50) NOT NULL,
  jurisdiction VARCHAR(2) NOT NULL,
  issuer_id INTEGER REFERENCES users(id),
  token_address VARCHAR(42) NOT NULL,
  token_id VARCHAR(100),
  metadata_uri VARCHAR(255),
  status VARCHAR(20) DEFAULT 'pending',
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW(),
  financial_metrics JSONB,
  UNIQUE(token_address, token_id)
);

-- Scores Table
CREATE TABLE scores (
  id SERIAL PRIMARY KEY,
  blockchain_score_id INTEGER,
  asset_id INTEGER REFERENCES assets(id),
  validator_id INTEGER REFERENCES users(id),
  risk_score SMALLINT NOT NULL CHECK (risk_score BETWEEN 0 AND
100),
  yield_score SMALLINT NOT NULL CHECK (yield_score BETWEEN 0
AND 100),
  confidence_score SMALLINT CHECK (confidence_score BETWEEN 0
AND 100),
  report_uri VARCHAR(255),
  components JSONB,
  created_at TIMESTAMP DEFAULT NOW(),
  transaction_hash VARCHAR(66)
);

-- Documents Table
```

```
CREATE TABLE documents (
  id SERIAL PRIMARY KEY,
  asset_id INTEGER REFERENCES assets(id),
  document_type VARCHAR(50) NOT NULL,
  ipfs_hash VARCHAR(100) NOT NULL,
  original_filename VARCHAR(255),
  mime_type VARCHAR(100),
  verification_status BOOLEAN DEFAULT FALSE,
  uploaded_at TIMESTAMP DEFAULT NOW(),
  verified_at TIMESTAMP
);

-- Oracle Data Table
CREATE TABLE oracle_data (
  id SERIAL PRIMARY KEY,
  asset_id INTEGER REFERENCES assets(id),
  oracle_type VARCHAR(50) NOT NULL,
  data JSONB NOT NULL,
  source_reference VARCHAR(255),
  attestation JSONB,
  created_at TIMESTAMP DEFAULT NOW()
);

-- Validator Stakes Table
CREATE TABLE validator_stakes (
  id SERIAL PRIMARY KEY,
  validator_id INTEGER REFERENCES users(id),
  amount NUMERIC NOT NULL,
  transaction_hash VARCHAR(66),
  staked_at TIMESTAMP DEFAULT NOW()
);

-- Events and status changes (audit log)
CREATE TABLE audit_log (
  id SERIAL PRIMARY KEY,
```

```
  entity_type VARCHAR(50) NOT NULL,
  entity_id INTEGER NOT NULL,
  action VARCHAR(50) NOT NULL,
  before_state JSONB,
  after_state JSONB,
  actor_id INTEGER REFERENCES users(id),
  ip_address VARCHAR(45),
  user_agent TEXT,
  created_at TIMESTAMP DEFAULT NOW()
);
```

14.2.4 Data Processing Pipeline

```
// Document Processing Pipeline
async function processDocumentUpload(file: File, assetId:
number, docType: string): Promise<string> {
  // 1. Validate file type and scan for viruses
  const validationResult = await validateFile(file);
  if (!validationResult.valid) {
    throw new Error(`Invalid file:
${validationResult.reason}`);
  }

  // 2. Extract metadata from document
  const metadata = await extractMetadata(file);

  // 3. Upload to IPFS
  const ipfsHash = await ipfsService.upload(file);

  // 4. Record document in database
  await documentsRepository.create({
    assetId,
    documentType: docType,
    ipfsHash,
    originalFilename: file.name,
    mimeType: file.type,
    metadata
  });

  // 5. For certain document types, trigger oracle verification
  if (docType === 'deed' || docType === 'title') {
    await queueOracleVerification(ipfsHash, assetId, docType);
  }

  return ipfsHash;
}
```

14.2.5 Blockchain Integration

```
// Submitting an asset to blockchain
async function submitAssetToBlockchain(asset: Asset, wallet:
ethers.Wallet): Promise<string> {
  const contract = new ethers.Contract(
    config.contracts.assetContract,
    AltheiaAssetABI,
    wallet
  );

  const tx = await contract.submitAsset(
    asset.assetType,
    asset.jurisdiction,
    asset.metadataURI,
    asset.tokenAddress,
    {
      gasLimit: 500000
    }
  );

  // Wait for transaction confirmation
  const receipt = await tx.wait();

  // Extract asset ID from event logs
  const event = receipt.events?.find(e => e.event ===
'AssetSubmitted');
  const assetId = event?.args?.assetId.toNumber();

  return {
    assetId,
    transactionHash: receipt.transactionHash
  };
}
```

14.3 Frontend Developer Guide

14.3.1 Component Architecture

```
App
├── Layout
│   ├── Header
│   │   ├── Logo
│   │   ├── MainNav
│   │   └── ConnectWallet
│   └── Footer
├── Pages
│   ├── Landing
│   ├── Login
│   ├── Dashboard
│   │   ├── RoleBasedDashboard
│   │   │   ├── InvestorDashboard
│   │   │   ├── IssuerDashboard
│   │   │   ├── ValidatorDashboard
│   │   │   └── AnalystDashboard
│   │   └── StatusOverview
│   ├── AssetDetails
│   │   ├── AssetMetrics
│   │   ├── RiskYieldChart
│   │   ├── DocumentList
│   │   └── ValidationHistory
│   ├── SubmitPage
│   │   ├── AssetSubmissionForm
│   │   └── DocumentUpload
│   └── InsightsPage
│       ├── FilterPanel
│       └── AssetList
└── Components
    ├── UI
```

```
    │   ├── Badge
    │   ├── Button
    │   ├── Card
    │   └── Modal
    ├── Charts
    │   └── RiskYieldChart
    └── Dashboard
        ├── AssetCard
        ├── ModelCard
        └── StatusOverview
```

14.3.2 State Management

Store Structure

```
// src/store/index.ts
import { create } from 'zustand';
import { Asset, AssetStatus, Score, User, ValidationStatus }
from '../types';

interface AltheiaStore {
  // Auth State
  user: User | null;
  walletConnected: boolean;
  selectedRole: string | null;

  // Asset Data
  assets: Asset[];
  filteredAssets: Asset[];
  selectedAsset: Asset | null;
  assetSubmissionStatus: 'idle' | 'pending' | 'success' |
'error';

  // Filters
```

```
  selectedAssetTypes: string[];
  selectedJurisdictions: string[];
  riskRange: [number, number];
  yieldRange: [number, number];
  searchQuery: string;

  // UI State
  currentTab: string;
  dashboardView: 'list' | 'grid';
  isAssetSubmissionModalOpen: boolean;
  isMobileMenuOpen: boolean;

  // Actions
  setUser: (user: User | null) => void;
  connectWallet: () => Promise<boolean>;
  disconnectWallet: () => void;
  setSelectedRole: (role: string | null) => void;

  fetchAssets: () => Promise<void>;
  fetchAssetDetails: (assetId: string) => Promise<Asset |
null>;
  submitAsset: (asset: Partial<Asset>) => Promise<string |
null>;
  submitScore: (assetId: string, score: Partial<Score>) =>
Promise<boolean>;

  setFilter: (filterType: string, value: any) => void;
  clearFilters: () => void;
}

export const useStore = create<AltheiaStore>((set, get) => ({
  // Initial state and action implementations here
}));
```

14.3.3 Key Component Implementations

ConnectWallet.tsx

```
import React, { useState } from 'react';
import { Button } from '../ui/Button';
import { useStore } from '../../store';
import { RoleSelector } from './RoleSelector';

export const ConnectWallet: React.FC = () => {
  const { walletConnected, connectWallet, disconnectWallet,
user } = useStore();
  const [isConnecting, setIsConnecting] = useState(false);
  const [showRoleSelector, setShowRoleSelector] =
useState(false);

  const handleConnect = async () => {
    setIsConnecting(true);
    try {
      const connected = await connectWallet();
      if (connected) {
        setShowRoleSelector(true);
      }
    } catch (error) {
      console.error("Failed to connect wallet:", error);
    } finally {
      setIsConnecting(false);
    }
  };

  const handleRoleSelect = (role: string) => {
    useStore.getState().setSelectedRole(role);
    setShowRoleSelector(false);
    // Navigate to appropriate dashboard
    window.location.href = '/dashboard';
```

```
  };

  const handleDisconnect = () => {
    disconnectWallet();
    setShowRoleSelector(false);
  };

  return (
    <div className="flex flex-col items-center justify-center
space-y-6 p-8 max-w-md mx-auto">
      {!walletConnected ? (
        <Button
          variant="primary"
          size="lg"
          onClick={handleConnect}
          disabled={isConnecting}
        >
          {isConnecting ? 'Connecting...' : 'Connect Wallet'}
        </Button>
      ) : showRoleSelector ? (
        <RoleSelector onRoleSelect={handleRoleSelect} />
      ) : (
        <div className="flex flex-col items-center space-y-4">
          <div className="bg-secondary p-4 rounded-lg text-
center">
            <p className="font-medium">Connected as</p>
            <p className="text-sm opacity-80 truncate max-w-
xs">{user?.walletAddress}</p>
          </div>
          <Button
            variant="secondary"
            size="sm"
            onClick={handleDisconnect}
          >
            Disconnect
```

```
            </Button>
          </div>
        )}
      </div>
    );
};
```

AssetSubmissionForm.tsx

```
import React, { useState } from 'react';
import { Button } from '../ui/Button';
import { useStore } from '../../store';
import { AssetType } from '../../types';

export const AssetSubmissionForm: React.FC = () => {
  const { submitAsset, assetSubmissionStatus } = useStore();

  const [formData, setFormData] = useState({
    name: '',
    assetType: '',
    jurisdiction: '',
    tokenAddress: '',
    tokenId: '',
    expectedYield: '',
    valuation: '',
    valuationCurrency: 'USD',
  });

  const [currentStep, setCurrentStep] = useState(1);
  const [documents, setDocuments] = useState<File[]>([]);

  const handleInputChange = (e:
React.ChangeEvent<HTMLInputElement | HTMLSelectElement>) => {
    const { name, value } = e.target;
```

```
    setFormData(prev => ({ ...prev, [name]: value }));
  };

  const handleDocumentUpload = (e:
React.ChangeEvent<HTMLInputElement>) => {
    if (e.target.files) {
      setDocuments(Array.from(e.target.files));
    }
  };

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();

    try {
      // Format data for submission
      const assetData = {
        ...formData,
        expectedYield: parseFloat(formData.expectedYield),
        valuation: parseFloat(formData.valuation),
        financialMetrics: {
          expectedYield: parseFloat(formData.expectedYield),
          valuation: parseFloat(formData.valuation),
          valuationCurrency: formData.valuationCurrency
        }
      };

      // Create form data for file uploads
      const formDataWithFiles = new FormData();
      documents.forEach((doc, index) => {
        formDataWithFiles.append(`documents[${index}]`, doc);
      });
      formDataWithFiles.append('assetData',
JSON.stringify(assetData));

      // Submit asset
```

```
      const assetId = await submitAsset(assetData);
      if (assetId) {
        // Navigate to success page or asset details
      }
    } catch (error) {
      console.error("Asset submission failed:", error);
    }
  };

  const nextStep = () => setCurrentStep(prev => prev + 1);
  const prevStep = () => setCurrentStep(prev => prev - 1);

  // Render appropriate form step based on currentStep
  // ...
};
```

14.3.4 API Integration

```
// src/api/assets.ts
import axios from 'axios';
import { API_BASE_URL } from '../config';
import { Asset, AssetSubmissionRequest, AssetListRequest } from
'../types';

export const assetsApi = {
  async getAll(params: AssetListRequest = {}) {
    const response = await axios.get(`${API_BASE_URL}/assets`,
{ params });
    return response.data;
  },

  async getById(id: string) {
    const response = await
axios.get(`${API_BASE_URL}/assets/${id}`);
```

```
    return response.data.asset;
  },

  async submit(data: AssetSubmissionRequest) {
    const response = await axios.post(`${API_BASE_URL}/assets`,
data, {
      headers: {
        'Content-Type': 'multipart/form-data'
      }
    });
    return response.data;
  },

  async getValidators(assetId: string) {
    const response = await
axios.get(`${API_BASE_URL}/assets/${assetId}/validators`);
    return response.data.validators;
  }
};
```

14.3.5 UI Implementation Guidelines

1. **Color Palette**
   - Primary: #3B82F6 (blue-500)
   - Secondary: #10B981 (emerald-500)
   - Accent: #8B5CF6 (violet-500)
   - Background: #F9FAFB (gray-50)
   - Text: #1F2937 (gray-800)
   - Risk indicators: Green (#10B981), Yellow (#FBBF24), Red (#EF4444)
2. **Typography**

- Headings: Inter, 700 weight
- Body: Inter, 400 weight
- Monospace (for addresses): Roboto Mono

3. **Responsive Design Breakpoints**
   - Mobile: < 640px
   - Tablet: 640px - 1024px
   - Desktop: > 1024px
4. **Component Variants**
   - Button variants: primary, secondary, accent, outline, ghost
   - Card variants: default, interactive, dashboard, asset
   - Badge variants: primary, secondary, accent, success, warning, error, neutral

15. Implementation Roadmap

This section outlines the development phases, milestones, and timelines for implementing the Altheia platform.

15.1 Phase 1: Foundation (Months 1-2)

Smart Contract Development

- Develop and test base contracts (Registry, Asset, Issuer)
- Implement basic access control and roles
- Deploy to test network

Backend Development

- Set up API server architecture
- Implement user authentication
- Create database schema
- Build basic CRUD operations for assets

Frontend Development

- Design system implementation
- Wallet connection flow
- Role selection components
- Basic dashboard layout

Deliverables

- Working wallet connection with role selection
- Testnet contract deployment
- Basic asset submission form
- Simple dashboard for each user role

15.2 Phase 2: Core Functionality (Months 3-4)

Smart Contract Development

- Implement validator staking and registration
- Develop score registry and verification logic
- Add oracle integration points
- Security audits (initial)

Backend Development

- IPFS integration for document storage
- Asset validation pipeline
- Scoring engine implementation
- API endpoints for validators

Frontend Development

- Asset submission workflow with document upload
- Validator dashboard with review interface
- Investor view with asset listings
- Basic risk-yield visualization

Deliverables

- End-to-end asset submission flow
- Document upload and IPFS storage
- Validator assignment and scoring interface
- Simple asset marketplace for investors

15.3 Phase 3: Enhanced Features (Months 5-6)

Smart Contract Development

- Portfolio curation contracts
- Advanced validation mechanisms
- Dispute resolution system
- Final security audit

Backend Development

- Analytics engine for risk assessment
- Oracle service integration
- Jurisdiction-specific validation rules
- Advanced filtering and search

Frontend Development

- Interactive risk-yield charts
- Advanced filtering interface
- Portfolio creation and management
- Enhanced document viewer

Integrations

- KYC provider integration (Persona/Sumsub)
- Chainlink Functions setup
- External data source connections

Deliverables

- Complete scoring system with validation
- KYC verification flow
- Oracle data integration
- Portfolio creation features

15.4 Phase 4: Production Launch (Month 7)

Smart Contract Development

- Mainnet deployment
- Post-launch monitoring
- Governance implementation

Backend Development

- Performance optimization
- Scaling infrastructure
- Final security hardening

Frontend Development

- UX refinements based on testing
- Accessibility improvements
- Mobile responsiveness enhancements

Testing & QA

- End-to-end testing
- Security penetration testing
- User acceptance testing

Deliverables

- Production-ready platform
- Documentation for all user roles
- Monitoring and analytics dashboard
- Initial governance framework

15.5 Development Dependencies & Critical Path

```
gantt
    title Altheia Platform Development Timeline
    dateFormat  YYYY-MM-DD
    section Smart Contracts
    Base Contracts           :a1, 2023-06-01, 30d
    Validator & Scoring      :a2, after a1, 30d
    Security Audit           :a3, after a2, 15d
    Portfolio & Governance   :a4, after a3, 30d
    Mainnet Deployment       :a5, after a4, 15d
    section Backend
    API Architecture         :b1, 2023-06-01, 20d
    Database Implementation  :b2, after b1, 20d
    Scoring Engine           :b3, after b2, 30d
    Oracle Integration       :b4, after b3, 25d
    Performance Optimization :b5, after b4, 20d
    section Frontend
    Design System            :c1, 2023-06-15, 25d
    Authentication Flow      :c2, after c1, 15d
    Asset Submission         :c3, after c2, 20d
    Dashboards               :c4, after c3, 30d
    Charts & Visualizations  :c5, after c4, 25d
    section Integration
    KYC Provider             :d1, 2023-08-01, 20d
    Chainlink Setup          :d2, after d1, 25d
    IPFS Configuration       :d3, after d2, 15d
    section Testing
    Smart Contract Testing   :e1, 2023-07-15, 60d
    E2E Testing              :e2, 2023-09-15, 45d
    UAT & Beta Testing       :e3, after e2, 30d
```

16. References & Learning Resources

- Chainlink Functions Docs (https://docs.chain.link/chainlink-functions)
- Centrifuge RWA (https://centrifuge.io/)
- Goldfinch Credit Evaluation (https://docs.goldfinch.finance/)
- Clearpool (https://clearpool.finance/)
- ERC Standards (https://eips.ethereum.org/)
- Sumsub (https://sumsub.com/kyc/)
- Persona (https://withpersona.com/)
- IPFS (https://docs.ipfs.tech/)