PubArx:2026.00005v1 [cs.SE] 4 Nov 2026 · Preprint

Detailed Development Document

Multi-Agent System Implementation

UCNS

University of Calgary Name Service

Blockchain-Based Multi-Agent System Development

Course: SENG 696 - Agent-Based Software Engineering

Instructor: Professor Behrouz Far

Institution: University of Calgary

Authors:

Ali Mohammadi Ruzbahani [30261140], Shuvam Agarwala [30290444]

Fall 2025

Abstract

This document provides comprehensive development specifications for the University of Calgary Name Service (UCNS) multi agent system implemented on the Polygon blockchain. While traditional multi agent frameworks like JADE and SPADE are designed for distributed computing environments with autonomous agents, UCNS adopts blockchain smart contracts as autonomous agents a novel approach that maintains agent oriented principles while leveraging blockchain's trustless execution model. This document details the complete development process including use case definitions for all participating agents, detailed class diagrams representing agent structures, message sequence charts showing inter agent protocols, data specifications using entity relationship modeling, deployment procedures, and testing strategies. The implementation translates GAIA design artifacts into executable Solidity smart contracts deployed on Polygon mainnet, demonstrating how agent based software engineering principles can be realized in decentralized blockchain environments. This document serves as both a technical specification for implementation and a comprehensive reference for understanding the system's architectural and behavioral characteristics.

Contents

1 Introduction **7**
1.1 Purpose and Scope . . . 7
1.2 Development Framework Context . . . 7
1.2.1 Traditional MAS Frameworks vs. Blockchain Agents . . . 7
1.2.2 Why Blockchain for Agent Implementation . . . 8
1.3 Development Methodology . . . 8
1.4 Relationship to GAIA Design (Report 1B) . . . 9
1.5 System Architecture Overview . . . 9
1.6 Document Organization . . . 10

2 Use Case Specifications **11**
2.1 Use Case Overview . . . 11
2.2 Use Case UC - R1: Register Domain . . . 11
2.3 Use Case UC - R3: Transfer Ownership . . . 13
2.4 Use Case UC - S2: Update Address Record . . . 14
2.5 Use Case UC - P1: Calculate Cost . . . 15
2.6 Use Case Diagram . . . 15

3 Class Diagrams **16**
3.1 Overview of Class Structure . . . 16
3.2 RegistryAgent Class Diagram . . . 16
3.3 ResolverAgent Class Diagram . . . 17
3.4 PricingAgent Class Diagram . . . 17
3.5 Complete System Class Diagram . . . 18
3.6 Smart Contract Interaction Diagram . . . 18
3.7 Interface Specifications . . . 18
3.7.1 IRegistryAgent Interface . . . 18
3.7.2 IResolverAgent Interface . . . 19

4 Sequence Diagrams **20**
4.1 Sequence Diagram: Domain Registration . . . 20
4.2 Sequence Diagram: Record Update with Authorization . . . 21
4.3 Sequence Diagram: Ownership Transfer . . . 21
4.4 Sequence Diagram: Complete Registration Flow with Error Handling . . . 22

5 Data Specifications **23**
5.1 Entity Relationship Diagram . . . 23
5.2 Data Dictionary . . . 23
5.3 Storage Layout Specifications . . . 24
5.3.1 RegistryAgent Storage Layout . . . 24
5.3.2 ResolverAgent Storage Layout . . . 25

6 Implementation Details **25**
6.1 Blockchain Environment Constraints . . . 25
6.2 Contract Structure . . . 26
6.2.1 RegistryAgent Implementation . . . 26

6.2.2 Registration Function Implementation . . . 28
6.3 ResolverAgent Implementation . . . 29
6.4 PricingAgent Implementation . . . 32

7 Deployment Procedures **33**
7.1 Deployment Architecture Overview . . . 33
7.2 Deployment Strategy . . . 34
7.2.1 Deployment Order . . . 34
7.2.2 Deployment Script (Hardhat) . . . 34
7.3 Configuration Files . . . 35
7.3.1 Hardhat Configuration . . . 35
7.4 Post Deployment Verification . . . 37
7.4.1 Verification Checklist . . . 37

8 Testing Strategy **37**
8.1 Test Categories . . . 37
8.1.1 Unit Tests . . . 37
8.1.2 Integration Tests . . . 39
8.2 Test Coverage Requirements . . . 41
8.2.1 Test Execution . . . 41

9 System Integration **41**
9.1 Web Interface Integration . . . 41
9.1.1 Frontend Architecture . . . 41
9.1.2 Web3 Configuration . . . 42
9.1.3 MetaMask Integration . . . 43
9.2 End to End User Flows . . . 45
9.2.1 Flow 1: Complete Domain Registration . . . 45
9.2.2 Flow 2: Domain Management . . . 45
9.3 Event Listening and Real Time Updates . . . 48

10 Production Deployment Information **50**
10.1 Live System Details . . . 50
10.1.1 Deployed Contracts . . . 50
10.1.2 Web Interface . . . 51
10.2 System Metrics . . . 51
10.2.1 Performance Benchmarks . . . 51
10.2.2 Cost Analysis . . . 51

11 Maintenance and Monitoring **52**
11.1 Monitoring Strategy . . . 52
11.1.1 Contract Monitoring . . . 52
11.2 Backup and Recovery . . . 53
11.2.1 Data Backup Strategy . . . 53
11.2.2 Disaster Recovery . . . 53

12 Conclusion **53**
12.1 Summary of Implementation 53
12.2 Agent-Oriented Implementation 54
12.3 Production Readiness 54
12.4 Lessons Learned 54
12.5 Future Enhancements 55

List of Figures

1 UCNS System Architecture Overview 10
2 UCNS Use Case Diagram 15
3 RegistryAgent Class Diagram 16
4 ResolverAgent Class Diagram 17
5 PricingAgent Class Diagram 17
6 Complete UCNS System Class Diagram 18
7 Smart Contract Interaction Diagram 18
8 Domain Registration Sequence Diagram 20
9 Record Update Sequence Diagram with Authorization 21
10 Ownership Transfer Sequence Diagram 21
11 Complete Registration Flow with Error Handling 22
12 UCNS Entity Relationship Diagram 23
13 UCNS Deployment Architecture 34
14 UCNS Frontend Architecture Layers 42
15 Complete Domain Registration Flow 45

List of Tables

1 Framework Comparison: JADE/SPADE vs. Blockchain Agents 8
2 Use Case Catalog 11
3 Complete Data Dictionary 23
4 Post Deployment Verification Checklist 37
5 Test Coverage Requirements by Component 41
6 Production Deployment Details 51
7 Production Performance Metrics 51
8 Common Issues and Solutions 56

1 Introduction

1.1 Purpose and Scope

This Detailed Development Document provides complete technical specifications for implementing the University of Calgary Name Service (UCNS) as a multi agent system on the Polygon blockchain. The document bridges the gap between high level GAIA design (Report 1B) and executable code, offering detailed development guidelines that ensure correct implementation of agent behaviors, interactions, and data structures.

Document Scope:

- **Use Case Specifications:** Detailed use cases for all agent interactions with actors, preconditions, postconditions, and exceptional flows
- **Class Diagrams:** Complete object oriented design showing agent classes, attributes, methods, and relationships
- **Sequence Diagrams:** Message sequence charts illustrating inter agent protocols and temporal ordering
- **Data Specifications:** Entity relationship models defining data structures and relationships
- **Implementation Details:** Contract structures, function signatures, modifiers, events, and state variables
- **Deployment Procedures:** Step by step deployment process with configuration and verification
- **Testing Specifications:** Unit tests, integration tests, and validation procedures

1.2 Development Framework Context

1.2.1 Traditional MAS Frameworks vs. Blockchain Agents

Traditional multi agent system frameworks like JADE (Java Agent Development Environment) and SPADE (Smart Python Agent Development Environment) provide runtime platforms for deploying autonomous software agents that:

- Execute as independent processes or threads
- Communicate via asynchronous message passing (typically FIPA ACL)
- Maintain internal state and knowledge bases
- Exhibit proactive and reactive behaviors
- Coordinate through negotiation and argumentation protocols

UCNS adopts a **blockchain native agent implementation** where agents are realized as smart contracts. This approach maintains core agent principles while adapting to blockchain constraints:

Table 1: Framework Comparison: JADE/SPADE vs. Blockchain Agents

Aspect	**JADE/SPADE**	**UCNS (Blockchain)**
Agent Runtime	JVM/Python processes	EVM smart contracts
Communication	Async message passing	Sync function calls
State Storage	In memory/database	Blockchain storage
Deployment	Central platform	Decentralized network
Trust Model	Platform mediated	Cryptographic/trustless
Immutability	Mutable agent code	Immutable contracts
Coordination	FIPA protocols	Transaction based
Discovery	Directory Facilitator	Fixed addresses
Persistence	Optional	Guaranteed by blockchain

1.2.2 Why Blockchain for Agent Implementation

Blockchain provides unique advantages for certain agent-based systems:

1. **Trustless Autonomy:** Agents operate without requiring trust in platform operators or other agents
2. **Verifiable Execution:** All agent actions are transparent and auditable on chain
3. **Guaranteed Persistence:** Agent state is replicated across thousands of nodes
4. **Economic Incentives:** Native cryptocurrency enables direct agent to agent value transfer
5. **Censorship Resistance:** No central authority can shut down or modify agents

For UCNS, these properties are essential. A naming service requires:

- Authoritative ownership records that cannot be unilaterally altered
- Transparent pricing and resolution logic
- Permanent availability without reliance on central servers
- Economic mechanisms for registration and renewal

1.3 Development Methodology

The development process follows these phases:

1. **Requirements to Use Cases:** Transform GAIA goals and roles into detailed use case specifications
2. **Design to Classes:** Map agent types and services to Solidity contract classes

3. **Protocols to Sequences:** Elaborate GAIA protocols into complete message sequence charts
4. **State to Data Models:** Define entity relationship models for agent state
5. **Implementation:** Code Solidity smart contracts implementing agent behaviors
6. **Testing:** Validate implementation against specifications
7. **Deployment:** Deploy to Polygon mainnet and verify

1.4 Relationship to GAIA Design (Report 1B)

Report 1B specified the UCNS system using the GAIA methodology in terms of roles, responsibilities, protocols, and organizational rules. This development document provides the concrete realization of those artifacts:

- **Roles to Agents:** The GAIA roles *RegistryAgent*, *ResolverAgent*, and *PricingAgent* are implemented as the Solidity contracts `UCNSRegistry`, `UCNSResolver`, and `PricingAgent`, respectively.
- **Protocols to Functions and Events:** GAIA interaction protocols (e.g., registration, ownership transfer, resolution, pricing) are mapped to explicit function calls and event emissions in the contracts, and are documented as sequence diagrams in Section **??**.
- **Liveness and Safety Properties:** GAIA liveness properties (such as "every valid registration eventually leads to a `DomainRegistered` event") and safety properties (such as "only the owner may transfer ownership") are encoded using Solidity `require` checks, modifiers, and carefully designed state transitions.
- **Organizational Rules to Constraints:** GAIA organizational rules (e.g., unique ownership, expiration policies, pricing tiers) are realized through the data model (Section **??**) and the blockchain constraints described in Section 6.1.

This ensures traceability from the abstract GAIA design in Report 1B to the concrete implementation decisions and artifacts presented in this development document.

1.5 System Architecture Overview

To provide a high-level view before diving into classes and protocols, Figure 1 shows the overall UCNS system architecture, including external actors, the web interface, the Web3 integration layer, the three smart-contract agents, and the underlying Polygon PoS network.

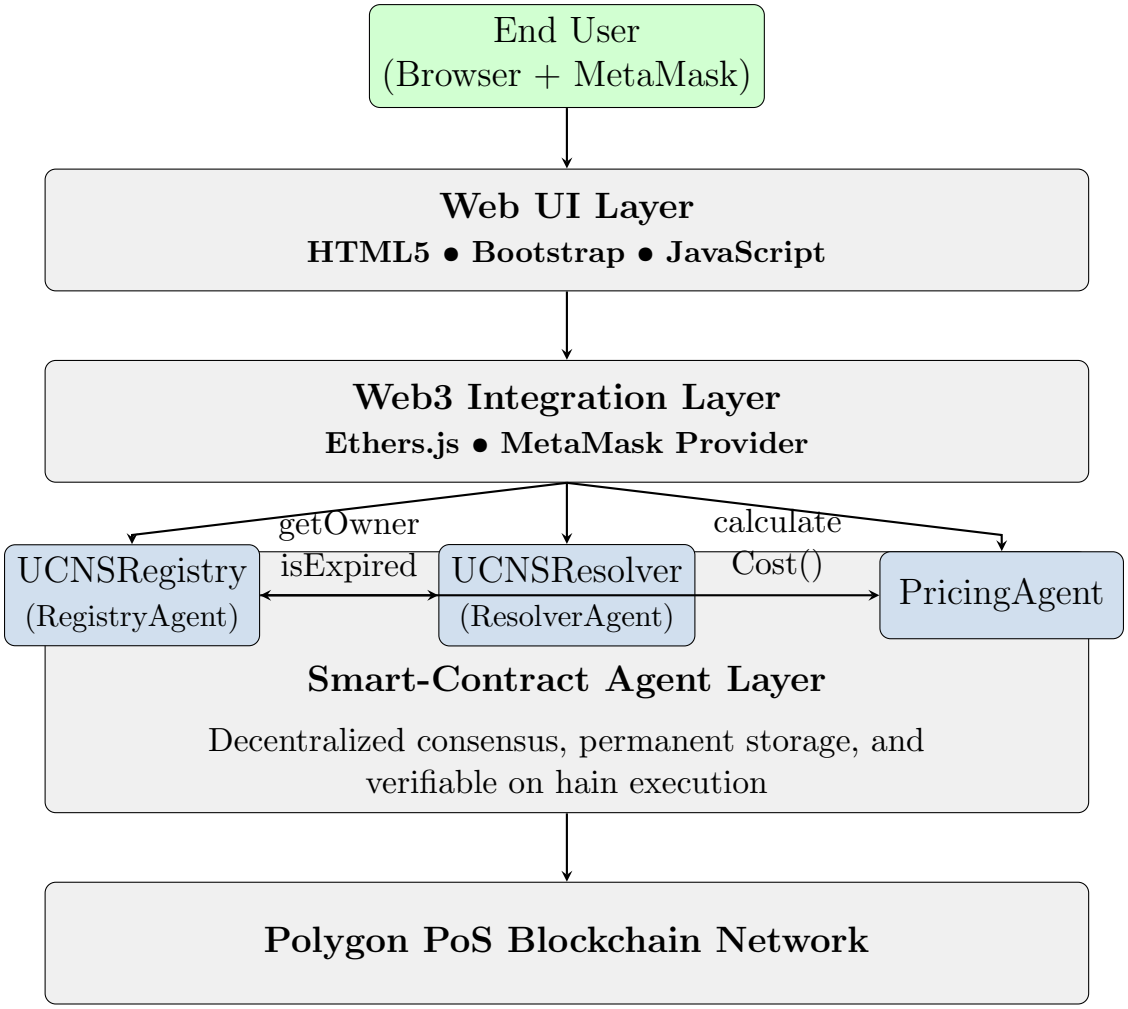


Figure 1: UCNS System Architecture Overview

1.6 Document Organization

Section 2 - Use Case Specifications: Comprehensive use cases for all agent functionalities

Section 3 - Class Diagrams: Object oriented design of agent contracts with complete class structures

Section 4 - Sequence Diagrams: Detailed message sequence charts for all interaction protocols

Section 5 - Data Specifications: Entity relationship models and data structure definitions

Section 6 - Implementation Details: Contract code structures, functions, modifiers, and events

Section 7 - Deployment Procedures: Step by step deployment and configuration

Section 8 - Testing Strategy: Comprehensive testing approach with test cases

Section 9 - System Integration: Web interface integration and end to end flows

2 Use Case Specifications

2.1 Use Case Overview

UCNS defines 15 primary use cases across three agent types. Use cases are categorized by agent and complexity level.

Table 2: Use Case Catalog

ID	Use Case	Agent	Complexity
UC-R1	Register Domain	Registry	High
UC-R2	Check Availability	Registry	Low
UC-R3	Transfer Ownership	Registry	Medium
UC-R4	Approve Operator	Registry	Medium
UC-R5	Get Owner	Registry	Low
UC-R6	Check Expiration	Registry	Low
UC-S1	Resolve Address	Resolver	Low
UC-S2	Update Address Record	Resolver	Medium
UC-S3	Update Text Record	Resolver	Medium
UC-S4	Get Text Record	Resolver	Low
UC-S5	Get All Records	Resolver	Low
UC-P1	Calculate Cost	Pricing	Low
UC-P2	Update Pricing Tier	Pricing	Medium
UC-P3	Preview Cost	Pricing	Low

2.2 Use Case UC - R1: Register Domain

Use Case ID	UC - R1
Use Case Name	Register Domain
Agent	RegistryAgent (UCNSRegistry)
Primary Actor	Domain Registrant (User)
Secondary Actors	PricingAgent
Brief Description	User registers a new domain name by providing payment and desired registration duration. RegistryAgent validates the name, checks availability, queries pricing, verifies payment, and records ownership atomically.
Preconditions	• User has MetaMask wallet with MATIC balance • Domain name adheres to naming rules • Domain is unregistered or expired • Duration is 1 - 10 years
Postconditions	• Domain ownership recorded with user as owner • Expiration timestamp set • Payment transferred to registry • DomainRegistered event emitted

Main Flow	1. User initiates registration via web interface 2. User enters domain name and selects duration 3. Web interface queries PricingAgent for cost preview 4. User confirms registration and signs transaction 5. Transaction submitted to RegistryAgent.register() 6. RegistryAgent validates domain name format 7. RegistryAgent checks naming rule compliance 8. RegistryAgent verifies domain availability 9. RegistryAgent queries PricingAgent for official cost 10. RegistryAgent verifies payment amount 11. RegistryAgent records ownership with expiration 12. RegistryAgent emits DomainRegistered event 13. Transaction confirmed on blockchain 14. Web interface updates to show successful registration
Alternative Flows	**A1 - Invalid Name Format:** • At step 6, if name format invalid, revert with error • User receives error message • User corrects name and retries **A2 - Domain Unavailable:** • At step 8, if domain already registered, revert • User informed domain is taken • User tries different name **A3 - Insufficient Payment:** • At step 10, if payment below cost, revert • User informed of required amount • User resubmits with correct payment
Exception Flows	**E1 - Transaction Failure:** • Network error during submission • User retries transaction **E2 - Gas Limit Exceeded:** • Transaction runs out of gas • User increases gas limit and retries
Special Requirements	• Transaction must complete in ¡5 seconds • Gas cost must not exceed 200,000 gas • Error messages must be user friendly

2.3 Use Case UC - R3: Transfer Ownership

Use Case ID	UC - R3
Use Case Name	Transfer Ownership
Agent	RegistryAgent
Primary Actor	Domain Owner
Brief Description	Current domain owner transfers ownership to a new Ethereum address.
Preconditions	• Caller is current owner • Domain has not expired • New owner address is valid (not zero)
Postconditions	• Owner field updated to new address • Expiration timestamp unchanged • OwnershipTransferred event emitted
Main Flow	1. Owner navigates to domain management page 2. Owner clicks "Transfer Ownership" 3. Owner enters new owner address 4. Owner confirms transfer 5. Transaction sent to RegistryAgent.transferOwnership() 6. RegistryAgent verifies caller is current owner 7. RegistryAgent checks domain not expired 8. RegistryAgent validates new owner address 9. RegistryAgent updates owner field 10. RegistryAgent emits OwnershipTransferred event 11. Transaction confirmed 12. UI reflects new ownership
Alternative Flows	**A1 - Not Owner:** • At step 6, if caller not owner, revert • User informed they lack permission **A2 - Domain Expired:** • At step 7, if expired, revert • User informed domain must be renewed first **A3 - Invalid Address:** • At step 8, if address is 0x0, revert • User corrects address

2.4 Use Case UC - S2: Update Address Record

Use Case ID	UC - S2
Use Case Name	Update Address Record
Agent	ResolverAgent
Primary Actor	Domain Owner
Secondary Actors	RegistryAgent
Brief Description	Owner updates the Ethereum address associated with their domain. ResolverAgent verifies ownership with RegistryAgent before updating.
Preconditions	• Caller owns the domain • Domain has not expired • New address is valid
Postconditions	• Resolution record updated with new address • AddressUpdated event emitted
Main Flow	1. Owner navigates to domain settings 2. Owner enters new Ethereum address 3. Owner submits update 4. Transaction sent to ResolverAgent.updateAddress() 5. ResolverAgent queries RegistryAgent.getOwner() 6. RegistryAgent returns owner address 7. ResolverAgent verifies caller matches owner 8. ResolverAgent queries RegistryAgent.isExpired() 9. RegistryAgent returns expiration status 10. ResolverAgent verifies domain not expired 11. ResolverAgent updates address record 12. ResolverAgent emits AddressUpdated event 13. Transaction confirmed 14. UI shows updated address
Alternative Flows	**A1 - Not Authorized:** • At step 7, if caller not owner, revert • User informed of authorization failure **A2 - Domain Expired:** • At step 10, if domain expired, revert • User informed to renew domain first

2.5 Use Case UC - P1: Calculate Cost

Use Case ID	UC - P1
Use Case Name	Calculate Cost
Agent	PricingAgent
Primary Actor	User or RegistryAgent
Brief Description	Calculate total registration cost based on domain length and duration.
Preconditions	• Domain name provided • Duration specified in years
Postconditions	• Cost returned in MATIC (wei) • No state changes (view function)
Main Flow	1. Caller invokes PricingAgent.calculateCost() 2. PricingAgent computes domain name length 3. PricingAgent determines pricing tier from length 4. PricingAgent retrieves base price for tier 5. PricingAgent calculates duration cost 6. PricingAgent sums base + duration costs 7. PricingAgent returns total cost
Alternative Flows	None (pure function, always succeeds)
Special Requirements	• Must execute in constant time O(1) • Must be deterministic • Zero gas cost (view function)

2.6 Use Case Diagram

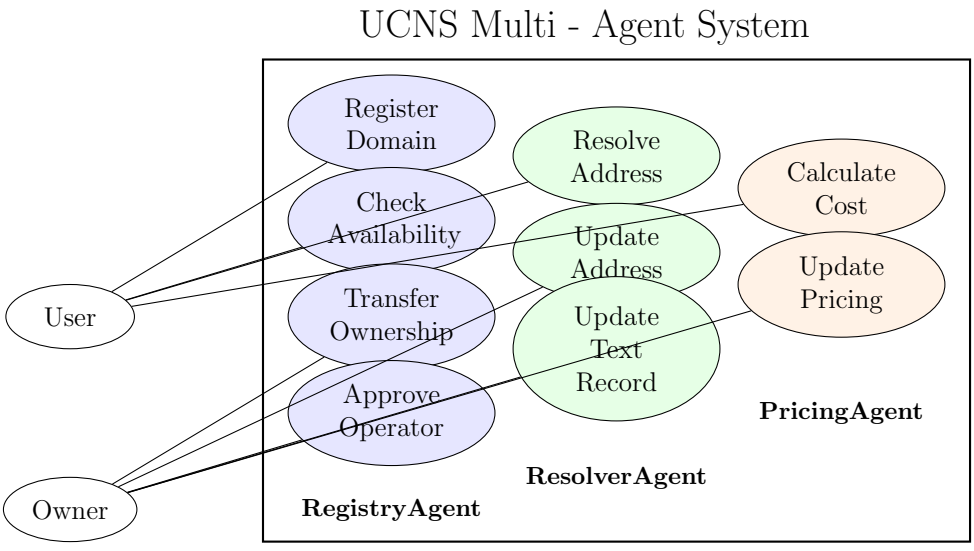


Figure 2: UCNS Use Case Diagram

3 Class Diagrams

3.1 Overview of Class Structure

UCNS implements three primary agent classes (smart contracts) with supporting structs, enums, and interfaces. The class diagram employs UML notation adapted for Solidity:

- **Contracts** → Classes
- **State Variables** → Attributes
- **Functions** → Methods
- **Events** → Signals
- **Modifiers** → Constraints
- **Structs** → Composite Data Types

3.2 RegistryAgent Class Diagram

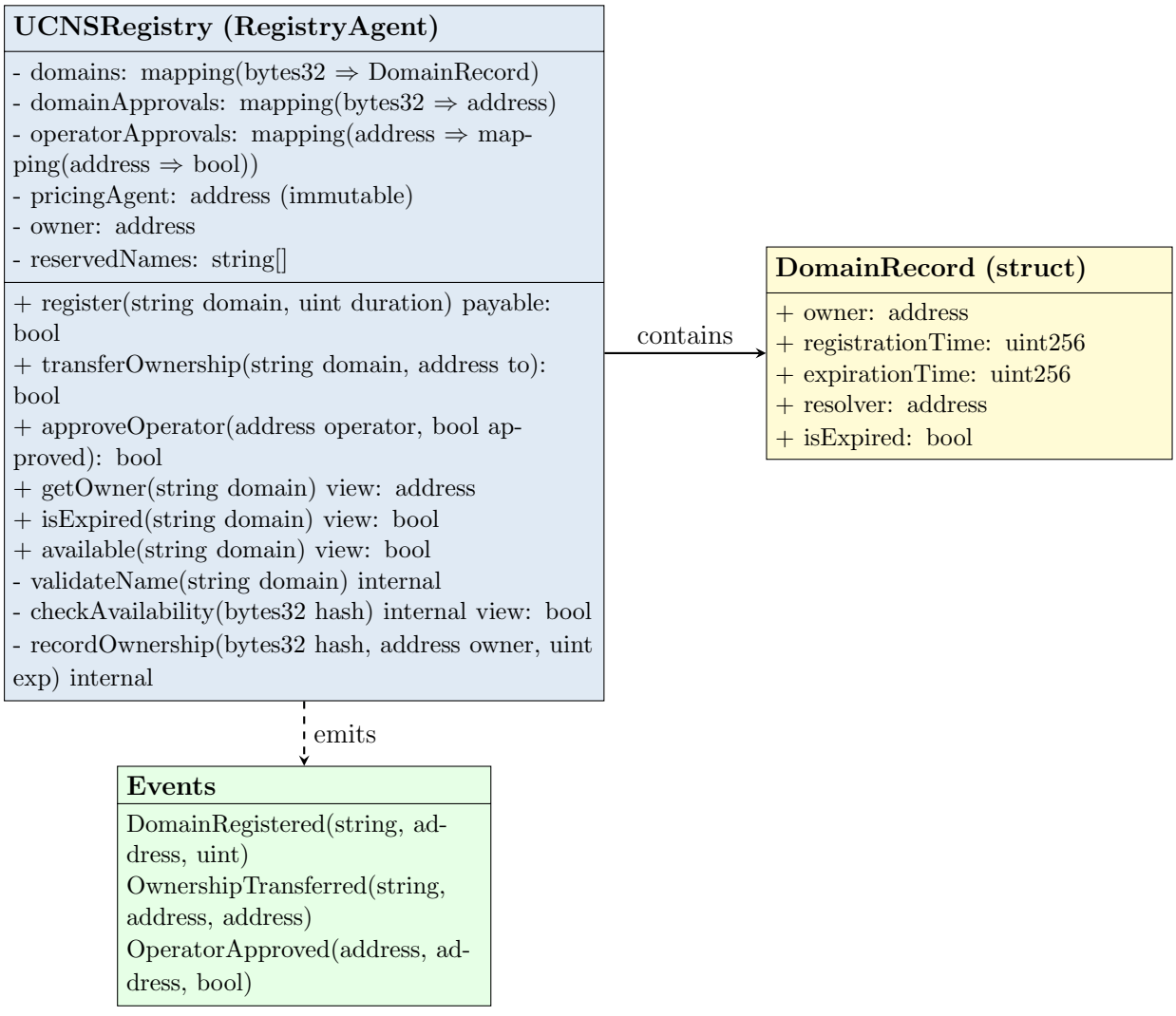


Figure 3: RegistryAgent Class Diagram

3.3 ResolverAgent Class Diagram

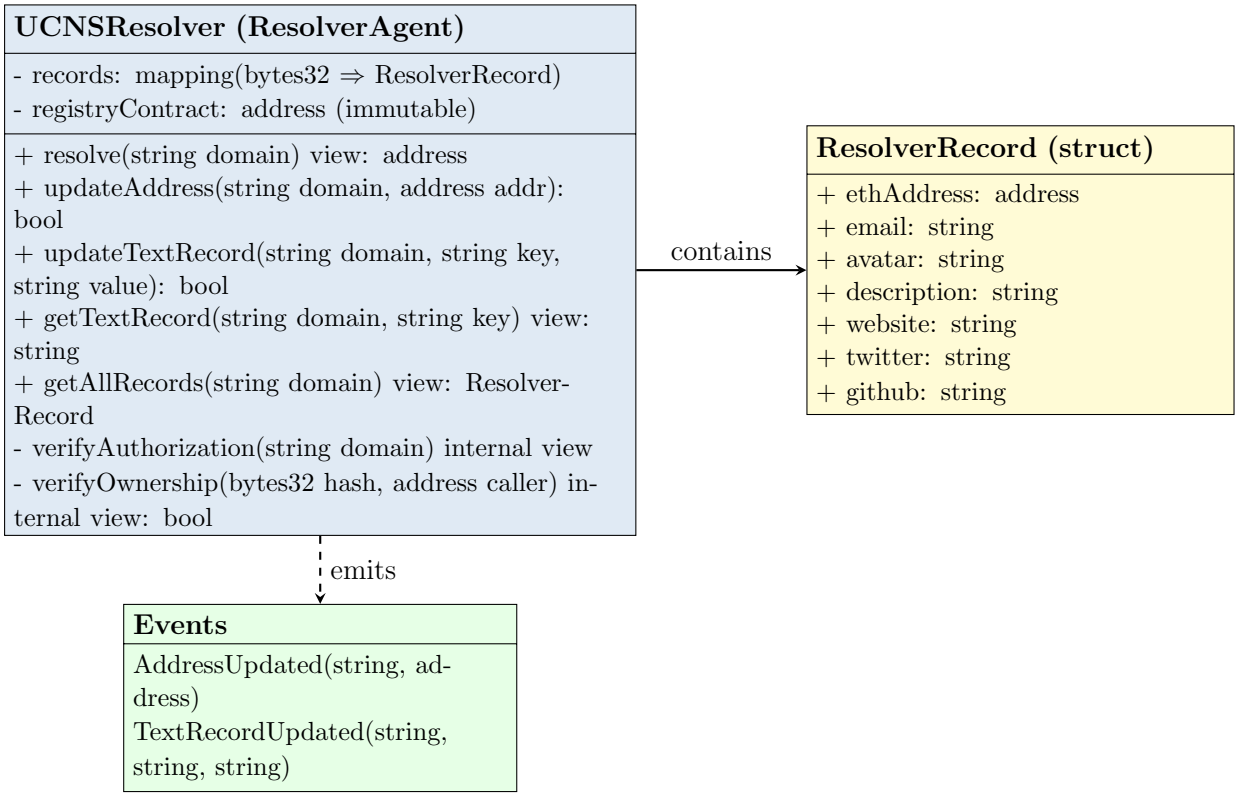


Figure 4: ResolverAgent Class Diagram

3.4 PricingAgent Class Diagram

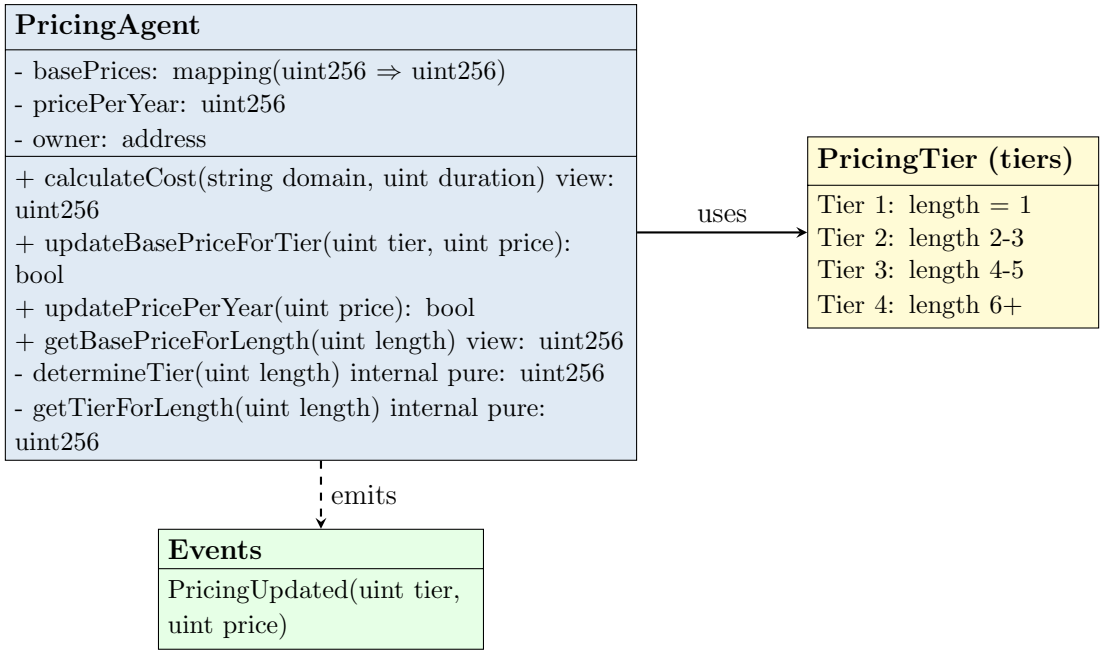


Figure 5: PricingAgent Class Diagram

3.5 Complete System Class Diagram

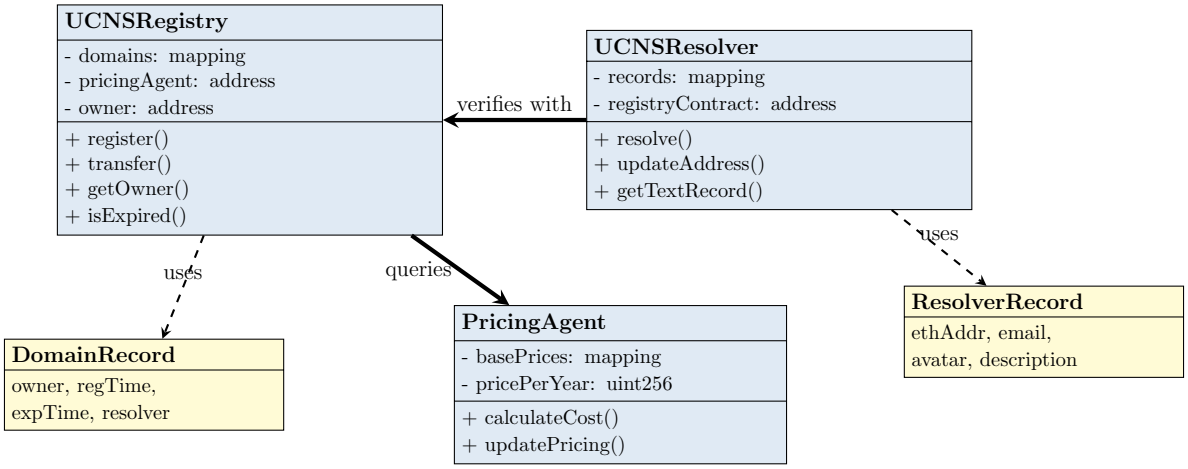


Figure 6: Complete UCNS System Class Diagram

3.6 Smart Contract Interaction Diagram

While Figures 3–6 focus on internal structure, Figure 7 summarizes the main interaction channels between the three smart-contract agents and the external Web3 client.

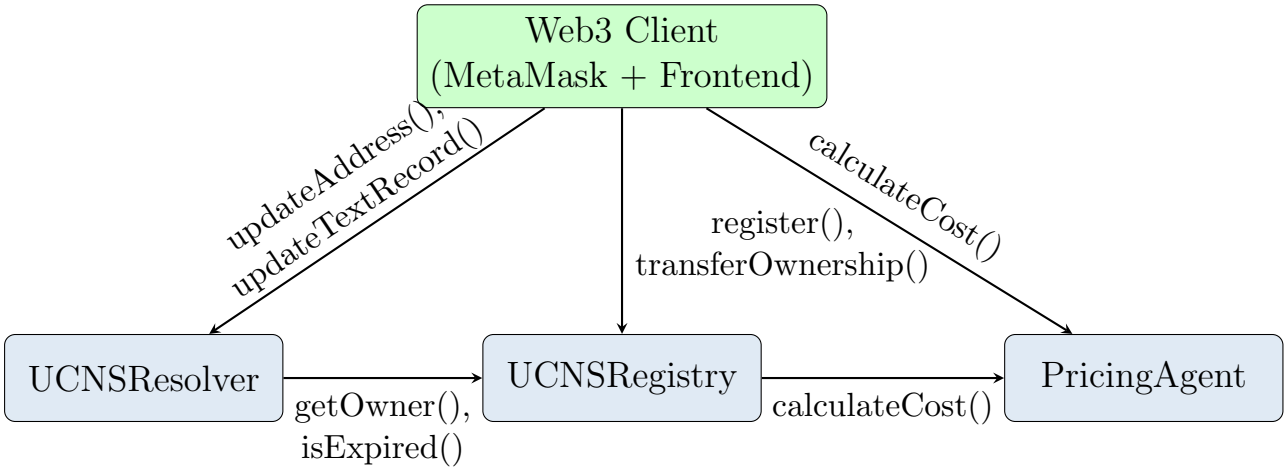


Figure 7: Smart Contract Interaction Diagram

3.7 Interface Specifications

3.7.1 IRegistryAgent Interface

```
interface IRegistryAgent {
    // Core registration functions
    function register(
        string memory domainName,
        uint256 duration
    ) external payable returns (bool);

    function transferOwnership(
```

```
        string memory domainName,
        address newOwner
    ) external returns (bool);

    function approveOperator(
        address operator,
        bool approved
    ) external returns (bool);

    // Query functions
    function getOwner(string memory domainName)
        external view returns (address);

    function isExpired(string memory domainName)
        external view returns (bool);

    function available(string memory domainName)
        external view returns (bool);

    // Events
    event DomainRegistered(
        string indexed domainName,
        address indexed owner,
        uint256 expirationTime
    );

    event OwnershipTransferred(
        string indexed domainName,
        address indexed from,
        address indexed to
    );
}
```

Listing 1: RegistryAgent Public Interface

3.7.2 IResolverAgent Interface

```
interface IResolverAgent {
    // Resolution functions
    function resolve(string memory domainName)
        external view returns (address);

    function updateAddress(
        string memory domainName,
        address newAddress
    ) external returns (bool);

    function updateTextRecord(
        string memory domainName,
        string memory key,
        string memory value
    ) external returns (bool);

    function getTextRecord(
        string memory domainName,
        string memory key
    ) external view returns (string memory);
```

```
21
22      // Events
23      event AddressUpdated(
24          string indexed domainName,
25          address newAddress
26      );
27
28      event TextRecordUpdated(
29          string indexed domainName,
30          string key,
31          string value
32      );
33  }
```

Listing 2: ResolverAgent Public Interface

4 Sequence Diagrams

4.1 Sequence Diagram: Domain Registration

Figure 8 details the end to end interaction for the *Register Domain* use case (UC - R1). It shows how the user, web interface, RegistryAgent, and PricingAgent collaborate to validate the domain, compute the cost, verify payment, record ownership, and propagate the success back to the user.

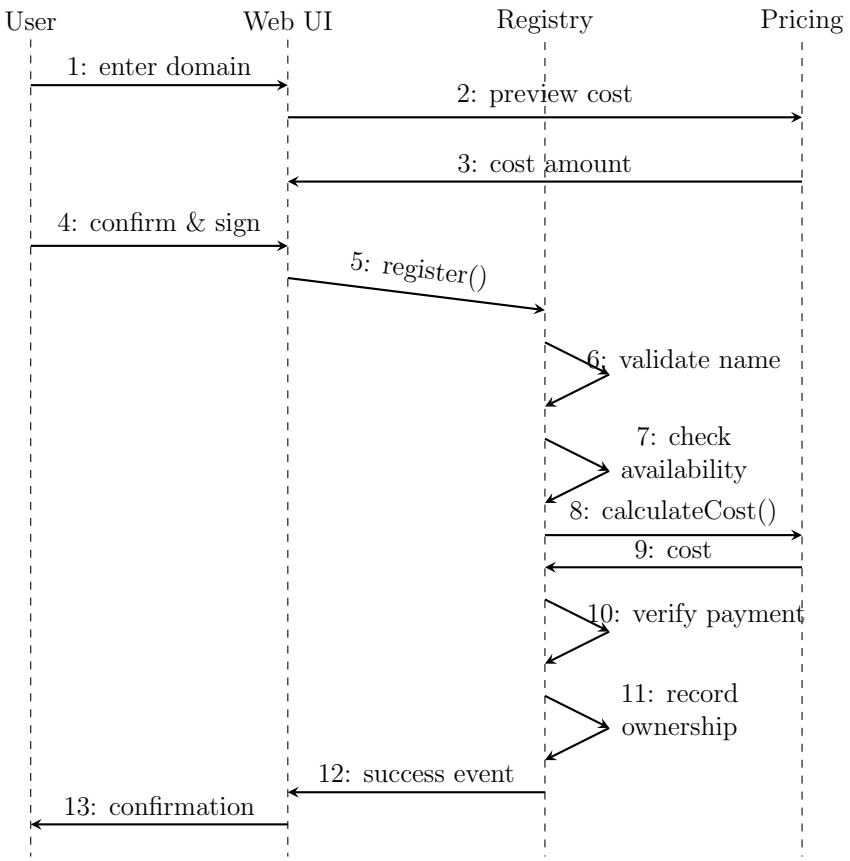


Figure 8: Domain Registration Sequence Diagram

4.2 Sequence Diagram: Record Update with Authorization

Figure 9 shows how ResolverAgent and RegistryAgent collaborate to enforce authorization when a domain owner attempts to update the address record. The diagram makes explicit that ResolverAgent always consults RegistryAgent for ownership and expiration checks before persisting any changes.

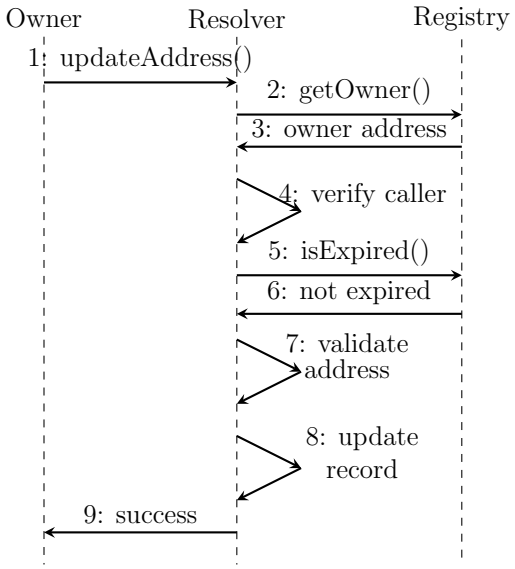


Figure 9: Record Update Sequence Diagram with Authorization

4.3 Sequence Diagram: Ownership Transfer

Figure 10 focuses on the *Transfer Ownership* use case (UC - R3). It highlights that only the current owner can initiate the transfer and that the RegistryAgent enforces both non expiration and address validity before updating the on chain owner field.

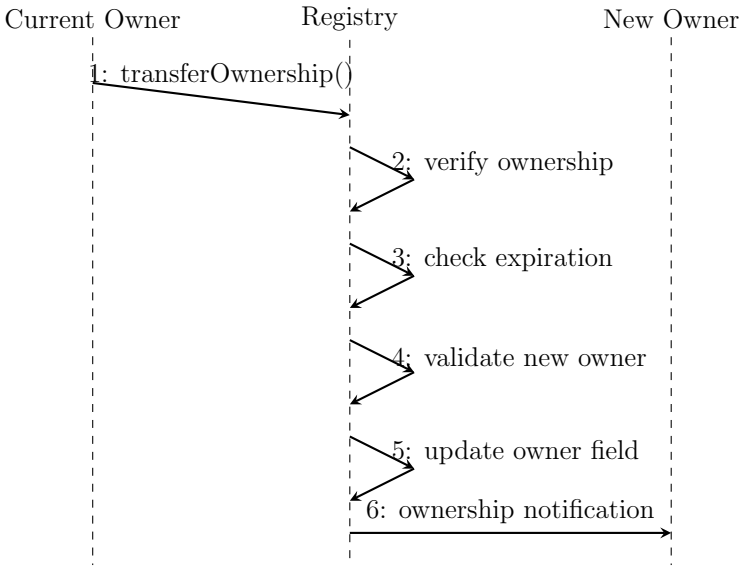


Figure 10: Ownership Transfer Sequence Diagram

4.4 Sequence Diagram: Complete Registration Flow with Error Handling

Figure 11 extends the basic registration flow by explicitly modeling error paths (invalid domain, insufficient payment) and the interaction with the underlying blockchain. This diagram ties together user actions, MetaMask signature, transaction submission, on chain execution, and frontend error reporting.

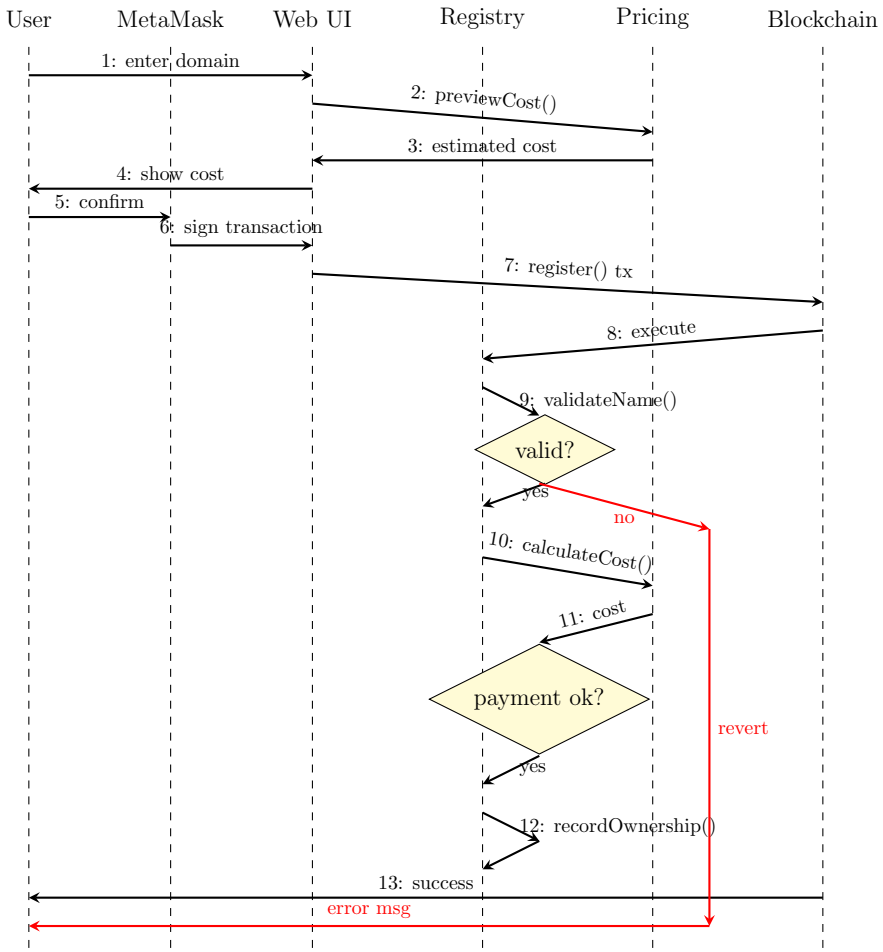


Figure 11: Complete Registration Flow with Error Handling

5 Data Specifications

5.1 Entity Relationship Diagram

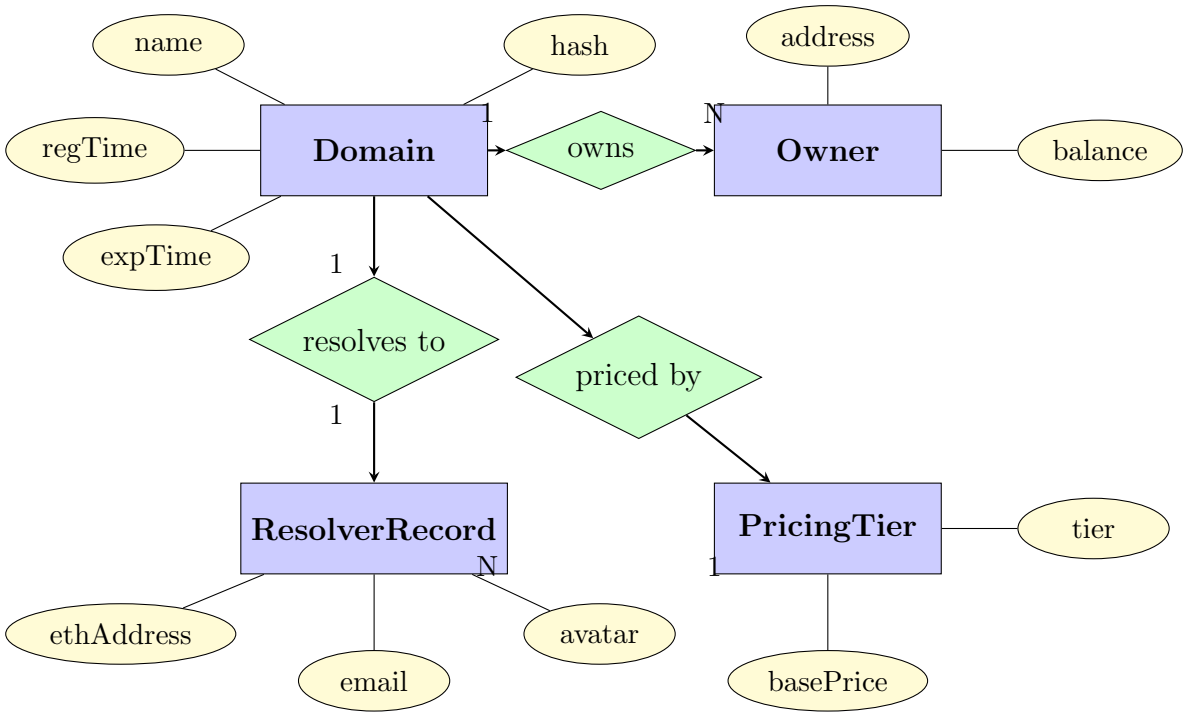


Figure 12: UCNS Entity Relationship Diagram

5.2 Data Dictionary

Table 3: Complete Data Dictionary

Entity/Attribute	Type	Description
Domain Entity		
domainName	string	Human readable domain name (e.g., "example")
domainHash	bytes32	Keccak256 hash of lowercase domain name
owner	address	Ethereum address of current owner
registrationTime	uint256	Unix timestamp of registration
expirationTime	uint256	Unix timestamp when domain expires
resolver	address	Address of resolver contract for this domain
isExpired	bool	Cached expiration status flag
Owner Entity		
address	address	20-byte Ethereum address
balance	uint256	MATIC balance in wei
domains	bytes32[]	Array of domain hashes owned
operators	address[]	Addresses with operator permissions
ResolverRecord Entity		

Entity/Attribute	Type	Description
ethAddress	address	Primary Ethereum address for domain
email	string	Email address (not verified)
avatar	string	URL to avatar image
description	string	Profile description text
website	string	Personal/organization website URL
twitter	string	Twitter handle (without @)
github	string	GitHub username
PricingTier Entity		
tier	uint256	Pricing tier number (1 - 4)
lengthMin	uint256	Minimum domain length for tier
lengthMax	uint256	Maximum domain length for tier
basePrice	uint256	Base registration price in wei
pricePerYear	uint256	Additional cost per year in wei

5.3 Storage Layout Specifications

5.3.1 RegistryAgent Storage Layout

```
contract UCNSRegistry {
    // Storage slot 0: owner address (20 bytes)
    address public owner;

    // Storage slot 1: pricing agent address (immutable, not in storage
        after init)
    address public immutable pricingAgent;

    // Storage slots 2+: nested mapping
    // domainHash => DomainRecord struct (5 slots per record)
    mapping(bytes32 => DomainRecord) private domains;

    // Storage slots vary: nested mapping
    // domainHash => approved address
    mapping(bytes32 => address) private domainApprovals;

    // Storage slots vary: doubly nested mapping
    // owner => operator => approved bool
    mapping(address => mapping(address => bool)) private
        operatorApprovals;

    // Reserved names array (dynamic size)
    string[] private reservedNames;

    struct DomainRecord {
        address owner;           // slot 0
        uint256 registrationTime; // slot 1
        uint256 expirationTime;   // slot 2
        address resolver;         // slot 3
        bool isExpired;          // slot 4
    }
}
```

Listing 3: RegistryAgent Storage Variables

5.3.2 ResolverAgent Storage Layout

```
contract UCNSResolver {
    // Storage slot 0: registry contract address (immutable)
    address public immutable registryContract;

    // Storage slots 1+: nested mapping
    // domainHash => ResolverRecord struct (7 slots per record)
    mapping(bytes32 => ResolverRecord) private records;

    struct ResolverRecord {
        address ethAddress;      // slot 0
        string email;            // slot 1 (dynamic)
        string avatar;           // slot 2 (dynamic)
        string description;      // slot 3 (dynamic)
        string website;          // slot 4 (dynamic)
        string twitter;          // slot 5 (dynamic)
        string github;           // slot 6 (dynamic)
    }
}
```

Listing 4: ResolverAgent Storage Variables

6 Implementation Details

6.1 Blockchain Environment Constraints

Before presenting the concrete contract code, it is important to make the blockchain specific constraints explicit, since they strongly influence the implementation:

- **Gas Costs and Computational Limits:** All on chain execution must be bounded and predictable. Loops are kept short, mappings are preferred over arrays for lookups, and cost critical functions such as `calculateCost()` are implemented in $O(1)$ time.
- **Immutability of Deployed Code:** Once deployed, contract bytecode cannot be changed. Upgradeability is therefore handled at the architectural level (e.g., via redeployment and configuration) rather than by modifying contracts in place.
- **Persistent, Public State:** All contract storage is globally replicated and publicly visible. The data model is designed to expose only information necessary for correct UCNS operation while avoiding redundant or sensitive data.
- **Synchronous, Transaction-Based Interaction:** Agent to agent communication (e.g., RegistryAgent to PricingAgent) occurs through synchronous function calls within a single transaction, not asynchronous message queues. Sequence diagrams therefore show direct call chains rather than delayed messaging.
- **Event Driven Observability:** Off chain components (frontend, monitoring tools) cannot directly "pull" state changes from contracts. Instead, they subscribe to events such as `DomainRegistered`, `OwnershipTransferred`, and `AddressUpdated` to maintain a consistent local view.

- **Security and Reentrancy Resistance:** State updates follow a "checks effects interactions" pattern; external calls are minimized and carefully ordered to avoid reentrancy issues.

These constraints motivate many of the design choices in the subsequent subsections, including the contract structure, storage layout, and event design.

6.2 Contract Structure

6.2.1 RegistryAgent Implementation

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract UCNSRegistry {
    // State variables
    struct DomainRecord {
        address owner;
        uint256 registrationTime;
        uint256 expirationTime;
        address resolver;
        bool isExpired;
    }

    mapping(bytes32 => DomainRecord) private domains;
    mapping(bytes32 => address) private domainApprovals;
    mapping(address => mapping(address => bool)) private
       operatorApprovals;

    address public immutable pricingAgent;
    address public owner;
    string[] private reservedNames;

    // Events
    event DomainRegistered(
        string indexed domainName,
        address indexed owner,
        uint256 expirationTime
    );

    event OwnershipTransferred(
        string indexed domainName,
        address indexed from,
        address indexed to
    );

    event OperatorApproved(
        address indexed owner,
        address indexed operator,
        bool approved
    );

    // Modifiers
    modifier onlyOwner() {
        require(msg.sender == owner, "Not contract owner");
        _;
```

```
    }

    modifier onlyDomainOwner(string memory domainName) {
        bytes32 hash = keccak256(bytes(toLowerCase(domainName)));
        require(domains[hash].owner == msg.sender, "Not domain owner");
        _;
    }

    modifier validDomain(string memory domainName) {
        require(bytes(domainName).length > 0 &&
                bytes(domainName).length <= 64, "Invalid length");
        require(isValidName(domainName), "Invalid name format");
        _;
    }

    // Constructor
    constructor(address _pricingAgent) {
        owner = msg.sender;
        pricingAgent = _pricingAgent;

        // Initialize reserved names
        reservedNames.push("admin");
        reservedNames.push("ucns");
        reservedNames.push("root");
        reservedNames.push("system");
    }

    // Core functions
    function register(
        string memory domainName,
        uint256 duration
    ) external payable validDomain(domainName) returns (bool) {
        // Implementation details in next section
    }

    function transferOwnership(
        string memory domainName,
        address newOwner
    ) external onlyDomainOwner(domainName) returns (bool) {
        // Implementation
    }

    // Query functions
    function getOwner(string memory domainName)
        external view returns (address) {
        bytes32 hash = keccak256(bytes(toLowerCase(domainName)));
        return domains[hash].owner;
    }

    function isExpired(string memory domainName)
        external view returns (bool) {
        bytes32 hash = keccak256(bytes(toLowerCase(domainName)));
        return block.timestamp > domains[hash].expirationTime;
    }

    // Internal helper functions
    function toLowerCase(string memory str)
        internal pure returns (string memory) {
```

```
        bytes memory bStr = bytes(str);
        bytes memory bLower = new bytes(bStr.length);

        for (uint i = 0; i < bStr.length; i++) {
            if ((uint8(bStr[i]) >= 65) && (uint8(bStr[i]) <= 90)) {
                bLower[i] = bytes1(uint8(bStr[i]) + 32);
            } else {
                bLower[i] = bStr[i];
            }
        }
        return string(bLower);
    }

    function isValidName(string memory domainName)
        internal pure returns (bool) {
        bytes memory b = bytes(domainName);

        // Check first and last character not hyphen
        if (b[0] == "-" || b[b.length - 1] == "-") return false;

        // Check all characters are valid
        for (uint i = 0; i < b.length; i++) {
            bytes1 char = b[i];

            bool isLowercase = (char >= "a" && char <= "z");
            bool isDigit = (char >= "0" && char <= "9");
            bool isHyphen = (char == "-");

            if (!(isLowercase || isDigit || isHyphen)) {
                return false;
            }
        }

        return true;
    }
}
```

Listing 5: UCNSRegistry.sol Core Structure

6.2.2 Registration Function Implementation

```
function register(
    string memory domainName,
    uint256 duration
) external payable validDomain(domainName) returns (bool) {
    // 1. Normalize domain name
    string memory normalizedName = toLowerCase(domainName);
    bytes32 domainHash = keccak256(bytes(normalizedName));

    // 2. Check reserved names
    for (uint i = 0; i < reservedNames.length; i++) {
        require(
            keccak256(bytes(reservedNames[i])) != domainHash,
            "Reserved name"
        );
    }

```

```
    // 3. Check availability
    require(
        domains[domainHash].owner == address(0) ||
        block.timestamp > domains[domainHash].expirationTime,
        "Domain unavailable"
    );

    // 4. Validate duration (1-10 years in seconds)
    require(
        duration >= 365 days && duration <= 3650 days,
        "Invalid duration"
    );

    // 5. Query pricing agent for cost
    IPricingAgent pricing = IPricingAgent(pricingAgent);
    uint256 cost = pricing.calculateCost(normalizedName, duration / 365
        days);

    // 6. Verify payment
    require(msg.value >= cost, "Insufficient payment");

    // 7. Record ownership
    domains[domainHash] = DomainRecord({
        owner: msg.sender,
        registrationTime: block.timestamp,
        expirationTime: block.timestamp + duration,
        resolver: address(0), // Can be set later
        isExpired: false
    });

    // 8. Refund excess payment
    if (msg.value > cost) {
        payable(msg.sender).transfer(msg.value - cost);
    }

    // 9. Emit event
    emit DomainRegistered(
        domainName,
        msg.sender,
        block.timestamp + duration
    );

    return true;
}
```

Listing 6: Domain Registration Implementation with Full Logic

6.3 ResolverAgent Implementation

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

interface IUCNSRegistry {
    function getOwner(string memory domainName)
        external view returns (address);
    function isExpired(string memory domainName)
        external view returns (bool);
```

```
    function isApprovedOperator(address owner, address operator)
        external view returns (bool);
}

contract UCNSResolver {
    struct ResolverRecord {
        address ethAddress;
        string email;
        string avatar;
        string description;
        string website;
        string twitter;
        string github;
    }

    mapping(bytes32 => ResolverRecord) private records;
    address public immutable registryContract;

    event AddressUpdated(
        string indexed domainName,
        address newAddress
    );

    event TextRecordUpdated(
        string indexed domainName,
        string key,
        string value
    );

    modifier onlyAuthorized(string memory domainName) {
        IUCNSRegistry registry = IUCNSRegistry(registryContract);
        address owner = registry.getOwner(domainName);

        require(
            msg.sender == owner ||
            registry.isApprovedOperator(owner, msg.sender),
            "Not authorized"
        );

        require(
            !registry.isExpired(domainName),
            "Domain expired"
        );
        _;
    }

    constructor(address _registryContract) {
        registryContract = _registryContract;
    }

    function resolve(string memory domainName)
        external view returns (address) {
        bytes32 hash = keccak256(bytes(toLowerCase(domainName)));
        return records[hash].ethAddress;
    }

    function updateAddress(
        string memory domainName,
```

```
        address newAddress
    ) external onlyAuthorized(domainName) returns (bool) {
        bytes32 hash = keccak256(bytes(toLowerCase(domainName)));
        records[hash].ethAddress = newAddress;

        emit AddressUpdated(domainName, newAddress);
        return true;
    }

    function updateTextRecord(
        string memory domainName,
        string memory key,
        string memory value
    ) external onlyAuthorized(domainName) returns (bool) {
        bytes32 hash = keccak256(bytes(toLowerCase(domainName)));

        if (keccak256(bytes(key)) == keccak256(bytes("email"))) {
            records[hash].email = value;
        } else if (keccak256(bytes(key)) == keccak256(bytes("avatar"))) {
            records[hash].avatar = value;
        } else if (keccak256(bytes(key)) == keccak256(bytes("description"))) {
            records[hash].description = value;
        } else if (keccak256(bytes(key)) == keccak256(bytes("website"))) {
            records[hash].website = value;
        } else if (keccak256(bytes(key)) == keccak256(bytes("twitter"))) {
            records[hash].twitter = value;
        } else if (keccak256(bytes(key)) == keccak256(bytes("github"))) {
            records[hash].github = value;
        } else {
            revert("Invalid key");
        }

        emit TextRecordUpdated(domainName, key, value);
        return true;
    }

    function getTextRecord(
        string memory domainName,
        string memory key
    ) external view returns (string memory) {
        bytes32 hash = keccak256(bytes(toLowerCase(domainName)));

        if (keccak256(bytes(key)) == keccak256(bytes("email"))) {
            return records[hash].email;
        } else if (keccak256(bytes(key)) == keccak256(bytes("avatar"))) {
            return records[hash].avatar;
        }
        // ... other keys

        return "";
    }

```

```
    function getAllRecords(string memory domainName)
        external view returns (ResolverRecord memory) {
        bytes32 hash = keccak256(bytes(toLowerCase(domainName)));
        return records[hash];
    }

    function toLowerCase(string memory str)
        internal pure returns (string memory) {
        // Same implementation as Registry
    }
}
```

Listing 7: UCNSResolver.sol Core Implementation

6.4 PricingAgent Implementation

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract PricingAgent {
    mapping(uint256 => uint256) public basePrices;
    uint256 public pricePerYear;
    address public owner;

    event PricingUpdated(uint256 tier, uint256 newPrice);

    modifier onlyOwner() {
        require(msg.sender == owner, "Not owner");
        _;
    }

    constructor() {
        owner = msg.sender;

        // Initialize pricing tiers (in wei)
        basePrices[1] = 1 ether;     // 1-char domains: 1 MATIC
        basePrices[2] = 0.5 ether;   // 2-3 char: 0.5 MATIC
        basePrices[3] = 0.1 ether;   // 4-5 char: 0.1 MATIC
        basePrices[4] = 0.05 ether;  // 6+ char: 0.05 MATIC

        pricePerYear = 0.01 ether;   // 0.01 MATIC per year
    }

    function calculateCost(
        string memory domainName,
        uint256 durationYears
    ) external view returns (uint256) {
        uint256 length = bytes(domainName).length;
        uint256 tier = getTierForLength(length);
        uint256 basePrice = basePrices[tier];
        uint256 durationCost = durationYears * pricePerYear;

        return basePrice + durationCost;
    }

    function getTierForLength(uint256 length)
        internal pure returns (uint256) {
```

```
        if (length == 1) return 1;
        if (length <= 3) return 2;
        if (length <= 5) return 3;
        return 4;
    }

    function getBasePriceForLength(uint256 length)
        external view returns (uint256) {
        uint256 tier = getTierForLength(length);
        return basePrices[tier];
    }

    function updateBasePriceForTier(
        uint256 tier,
        uint256 newPrice
    ) external onlyOwner returns (bool) {
        require(tier >= 1 && tier <= 4, "Invalid tier");
        basePrices[tier] = newPrice;

        emit PricingUpdated(tier, newPrice);
        return true;
    }

    function updatePricePerYear(uint256 newPrice)
        external onlyOwner returns (bool) {
        pricePerYear = newPrice;
        return true;
    }
}
```

Listing 8: PricingAgent.sol Complete Implementation

7 Deployment Procedures

7.1 Deployment Architecture Overview

Figure 13 summarizes how the development environment, deployment scripts, Polygon RPC endpoints, PolygonScan, and the production frontend fit together during deployment and operation.

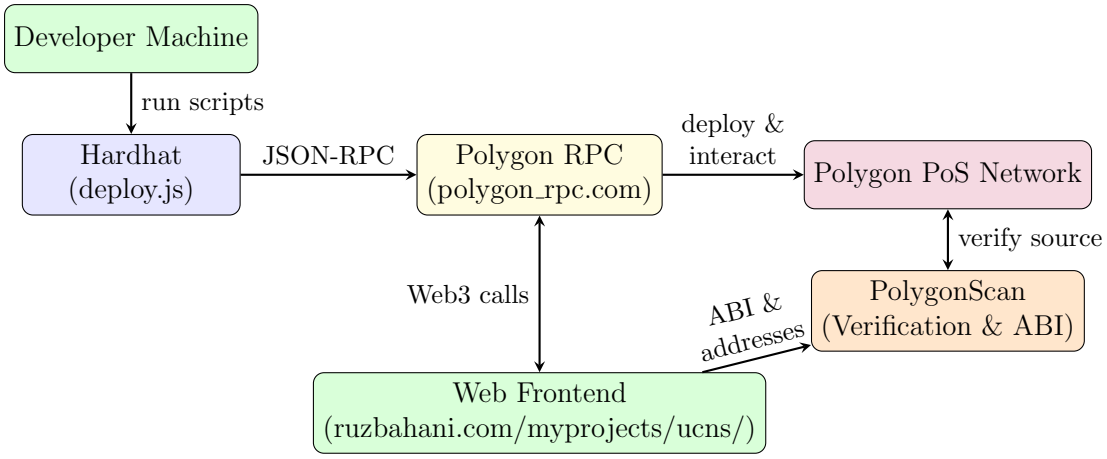


Figure 13: UCNS Deployment Architecture

7.2 Deployment Strategy

7.2.1 Deployment Order

Contracts must be deployed in specific order due to dependencies:

1. **Deploy PricingAgent** - No dependencies, deploy first
2. **Deploy UCNSRegistry** - Requires PricingAgent address in constructor
3. **Deploy UCNSResolver** - Requires Registry address in constructor
4. **Verify all contracts** on PolygonScan for transparency

7.2.2 Deployment Script (Hardhat)

```
const hre = require("hardhat");

async function main() {
    console.log("Starting UCNS deployment to Polygon...");

    // Get deployer account
    const [deployer] = await hre.ethers.getSigners();
    console.log("Deploying with account:", deployer.address);
    console.log("Account balance:",
        (await deployer.getBalance()).toString());

    // 1. Deploy PricingAgent
    console.log("\n1. Deploying PricingAgent...");
    const PricingAgent = await hre.ethers.getContractFactory("
       PricingAgent");
    const pricingAgent = await PricingAgent.deploy();
    await pricingAgent.deployed();
    console.log("PricingAgent deployed to:", pricingAgent.address);

    // Wait for confirmations
    await pricingAgent.deployTransaction.wait(5);

    // 2. Deploy UCNSRegistry
```

```
    console.log("\n2. Deploying UCNSRegistry...");
    const UCNSRegistry = await hre.ethers.getContractFactory("
        UCNSRegistry");
    const registry = await UCNSRegistry.deploy(pricingAgent.address);
    await registry.deployed();
    console.log("UCNSRegistry deployed to:", registry.address);

    await registry.deployTransaction.wait(5);

    // 3. Deploy UCNSResolver
    console.log("\n3. Deploying UCNSResolver...");
    const UCNSResolver = await hre.ethers.getContractFactory("
        UCNSResolver");
    const resolver = await UCNSResolver.deploy(registry.address);
    await resolver.deployed();
    console.log("UCNSResolver deployed to:", resolver.address);

    await resolver.deployTransaction.wait(5);

    // 4. Verify contracts on PolygonScan
    console.log("\n4. Verifying contracts on PolygonScan...");

    await hre.run("verify:verify", {
        address: pricingAgent.address,
        constructorArguments: []
    });

    await hre.run("verify:verify", {
        address: registry.address,
        constructorArguments: [pricingAgent.address]
    });

    await hre.run("verify:verify", {
        address: resolver.address,
        constructorArguments: [registry.address]
    });

    console.log("\n=== Deployment Complete ===");
    console.log("PricingAgent:", pricingAgent.address);
    console.log("UCNSRegistry:", registry.address);
    console.log("UCNSResolver:", resolver.address);
    console.log("\nUpdate frontend config with these addresses.");
}

main()
    .then(() => process.exit(0))
    .catch((error) => {
        console.error(error);
        process.exit(1);
    });
```

Listing 9: deploy.js - Hardhat Deployment Script

7.3 Configuration Files

7.3.1 Hardhat Configuration

```
require("@nomicfoundation/hardhat-toolbox");
require("@nomiclabs/hardhat-etherscan");
require("dotenv").config();

module.exports = {
  solidity: {
    version: "0.8.20",
    settings: {
      optimizer: {
        enabled: true,
        runs: 200
      }
    }
  },
  networks: {
    polygon: {
      url: process.env.POLYGON_RPC_URL || "https://polygon-rpc.com",
      accounts: [process.env.PRIVATE_KEY],
      chainId: 137
    },
    mumbai: {
      url: process.env.MUMBAI_RPC_URL ||
           "https://rpc-mumbai.maticvigil.com",
      accounts: [process.env.PRIVATE_KEY],
      chainId: 80001
    }
  },
  etherscan: {
    apiKey: {
      polygon: process.env.POLYGONSCAN_API_KEY,
      polygonMumbai: process.env.POLYGONSCAN_API_KEY
    }
  },
  paths: {
    sources: "./contracts",
    tests: "./test",
    cache: "./cache",
    artifacts: "./artifacts"
  }
};
```

Listing 10: hardhat.config.js

7.4 Post Deployment Verification

7.4.1 Verification Checklist

Table 4: Post Deployment Verification Checklist

Verification Step	Status	Notes
PricingAgent deployed successfully		0x50F50124...
UCNSRegistry deployed with correct PricingAgent address		0xc9eD4B38...
UCNSResolver deployed with correct Registry address		0x2De89713...
All contracts verified on PolygonScan		Source code visible
PricingAgent.calculateCost() works		Returns expected costs
Registry can query PricingAgent		Inter contract call successful
Resolver can query Registry		Authorization works
Test domain registration succeeds		Gas usage acceptable
Events emitted correctly		Visible in transactions
Frontend can connect to contracts		Web3 integration working

8 Testing Strategy

8.1 Test Categories

8.1.1 Unit Tests

Test individual contract functions in isolation:

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("UCNSRegistry", function () {
    let registry, pricing, owner, user1, user2;

    beforeEach(async function () {
        [owner, user1, user2] = await ethers.getSigners();

        // Deploy PricingAgent
        const PricingAgent = await ethers.getContractFactory("
            PricingAgent");
        pricing = await PricingAgent.deploy();

        // Deploy Registry
        const Registry = await ethers.getContractFactory("UCNSRegistry"
            );
        registry = await Registry.deploy(pricing.address);
    });

    describe("Domain Registration", function () {
```

```
20          it("Should register a valid domain", async function () {
21              const domain = "test";
22              const duration = 365 * 24 * 60 * 60; // 1 year
23              const cost = await pricing.calculateCost(domain, 1);
24
25              await expect(
26                  registry.connect(user1).register(domain, duration, {
27                      value: cost
28                  })
29              ).to.emit(registry, "DomainRegistered")
30               .withArgs(domain, user1.address, anyValue);
31
32              expect(await registry.getOwner(domain)).to.equal(user1.
                    address);
33          });
34
35          it("Should reject invalid domain names", async function () {
36              const invalidDomains = [
37                  "-test",    // Leading hyphen
38                  "test-",    // Trailing hyphen
39                  "Test",     // Uppercase
40                  "te st",    // Space
41                  "te@st"     // Special char
42              ];
43
44              for (const domain of invalidDomains) {
45                  await expect(
46                      registry.connect(user1).register(
47                          domain,
48                          365 * 24 * 60 * 60,
49                          { value: ethers.utils.parseEther("1") }
50                      )
51                  ).to.be.reverted;
52              }
53          });
54
55          it("Should reject registration of taken domain", async function
                () {
56              const domain = "test";
57              const duration = 365 * 24 * 60 * 60;
58              const cost = await pricing.calculateCost(domain, 1);
59
60              // First registration
61              await registry.connect(user1).register(domain, duration, {
62                  value: cost
63              });
64
65              // Second attempt should fail
66              await expect(
67                  registry.connect(user2).register(domain, duration, {
68                      value: cost
69                  })
70              ).to.be.revertedWith("Domain unavailable");
71          });
72
73          it("Should reject insufficient payment", async function () {
74              const domain = "test";
75              const duration = 365 * 24 * 60 * 60;
```

```
            const cost = await pricing.calculateCost(domain, 1);
            const insufficientPayment = cost.div(2);

            await expect(
                registry.connect(user1).register(domain, duration, {
                    value: insufficientPayment
                })
            ).to.be.revertedWith("Insufficient payment");
        });
    });

    describe("Ownership Transfer", function () {
        beforeEach(async function () {
            const domain = "test";
            const duration = 365 * 24 * 60 * 60;
            const cost = await pricing.calculateCost(domain, 1);

            await registry.connect(user1).register(domain, duration, {
                value: cost
            });
        });

        it("Should transfer ownership to new address", async function
            () {
            const domain = "test";

            await expect(
                registry.connect(user1).transferOwnership(domain, user2
                    .address)
            ).to.emit(registry, "OwnershipTransferred")
             .withArgs(domain, user1.address, user2.address);

            expect(await registry.getOwner(domain)).to.equal(user2.
               address);
        });

        it("Should reject transfer by non-owner", async function () {
            const domain = "test";

            await expect(
                registry.connect(user2).transferOwnership(domain, user2
                    .address)
            ).to.be.revertedWith("Not domain owner");
        });
    });
});
```

Listing 11: test/Registry.test.js - Sample Unit Tests

8.1.2 Integration Tests

Test inter agent communication:

```
describe("Agent Integration Tests", function () {
    let registry, resolver, pricing, user;

    beforeEach(async function () {
        [user] = await ethers.getSigners();
```

```
6
7           // Deploy all agents
8           const PricingAgent = await ethers.getContractFactory("
                PricingAgent");
9           pricing = await PricingAgent.deploy();
10
11          const Registry = await ethers.getContractFactory("UCNSRegistry"
                );
12          registry = await Registry.deploy(pricing.address);
13
14          const Resolver = await ethers.getContractFactory("UCNSResolver"
                );
15          resolver = await Resolver.deploy(registry.address);
16      });
17
18      it("Should complete full registration and resolution flow", async
            function () {
19          const domain = "test";
20          const duration = 365 * 24 * 60 * 60;
21          const testAddress = "0x1234567890123456789012345678901234567890
                ";
22
23          // 1. Calculate cost via PricingAgent
24          const cost = await pricing.calculateCost(domain, 1);
25          expect(cost).to.be.gt(0);
26
27          // 2. Register via Registry
28          await registry.connect(user).register(domain, duration, {
29              value: cost
30          });
31
32          // 3. Verify ownership
33          expect(await registry.getOwner(domain)).to.equal(user.address);
34
35          // 4. Update resolver record
36          await resolver.connect(user).updateAddress(domain, testAddress)
                ;
37
38          // 5. Resolve domain
39          const resolved = await resolver.resolve(domain);
40          expect(resolved).to.equal(testAddress);
41      });
42
43      it("Should enforce authorization between Resolver and Registry",
            async function () {
44          const domain = "test";
45          const duration = 365 * 24 * 60 * 60;
46          const cost = await pricing.calculateCost(domain, 1);
47
48          const [owner, attacker] = await ethers.getSigners();
49
50          // Register as owner
51          await registry.connect(owner).register(domain, duration, {
52              value: cost
53          });
54
55          // Attempt unauthorized update
56          await expect(
```

```
            resolver.connect(attacker).updateAddress(
                domain,
                attacker.address
            )
        ).to.be.revertedWith("Not authorized");
    });
});
```

Listing 12: test/Integration.test.js

8.2 Test Coverage Requirements

Table 5: Test Coverage Requirements by Component

Component	Target Coverage	Critical Paths	Edge Cases
RegistryAgent	>95%	Registration, Transfer	Invalid names, Reserved
ResolverAgent	>90%	Update, Resolve	Unauthorized, Expired
PricingAgent	100%	Cost calculation	All tier boundaries
Integration	>85%	Full flows	Inter-agent failures

8.2.1 Test Execution

```
# Run all tests
npx hardhat test

# Run specific test file
npx hardhat test test/Registry.test.js

# Run with coverage
npx hardhat coverage

# Run with gas reporting
REPORT_GAS=true npx hardhat test

# Generate coverage report
npx hardhat coverage --testfiles "test/**/*.js"
```

Listing 13: Running Tests

9 System Integration

Note that Figure 1 in Section 1.5 provides the global architecture overview, while Figure 14 in this section focuses specifically on the layered frontend and Web3 integration viewpoint.

9.1 Web Interface Integration

9.1.1 Frontend Architecture

The web interface serves as the user-facing layer that interacts with the three smart contract agents:

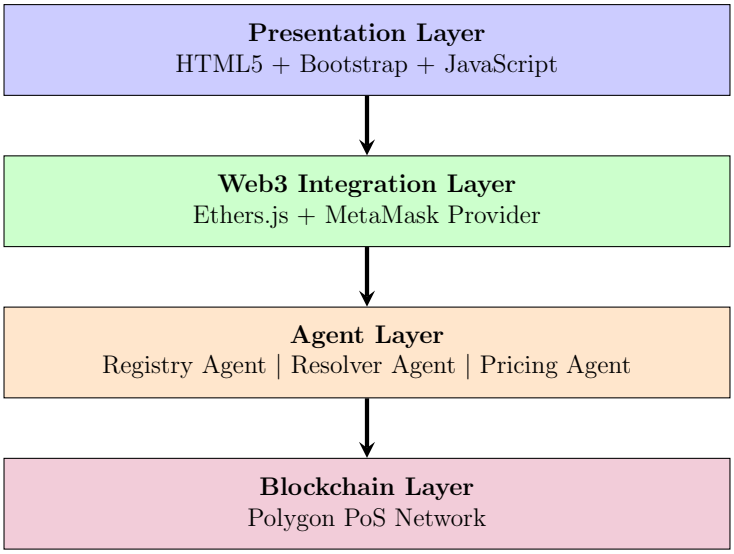


Figure 14: UCNS Frontend Architecture Layers

9.1.2 Web3 Configuration

```
// Contract addresses on Polygon Mainnet
const CONTRACT_ADDRESSES = {
    registry: '0xc9eD4B38E29C64d37cb83819D5eEcFD34EFdce0C',
    resolver: '0x2De897131ee8AC0538585887989E2314034F0b71',
    pricing: '0x50F50124Ee00002379142cff115b0550240898B3'
};

// Network configuration
const NETWORK_CONFIG = {
    chainId: 137,
    chainName: 'Polygon Mainnet',
    nativeCurrency: {
        name: 'MATIC',
        symbol: 'MATIC',
        decimals: 18
    },
    rpcUrls: ['https://polygon-rpc.com'],
    blockExplorerUrls: ['https://polygonscan.com']
};

// Contract ABIs (abbreviated)
const REGISTRY_ABI = [
    'function register(string domain, uint256 duration) payable returns (bool)',
    'function getOwner(string domain) view returns (address)',
    'function transferOwnership(string domain, address to) returns (bool)',
    'function isExpired(string domain) view returns (bool)',
    'event DomainRegistered(string indexed domain, address indexed owner, uint256 expiration)'
];

const RESOLVER_ABI = [
    'function resolve(string domain) view returns (address)',
```

```
    'function updateAddress(string domain, address addr) returns (bool)',
    'function updateTextRecord(string domain, string key, string value) returns (bool)',
    'function getTextRecord(string domain, string key) view returns (string)',
    'event AddressUpdated(string indexed domain, address newAddress)'
];

const PRICING_ABI = [
    'function calculateCost(string domain, uint256 duration) view returns (uint256)'
];
```

Listing 14: config.js - Frontend Configuration

9.1.3 MetaMask Integration

```
// Initialize Web3 provider
async function initWeb3() {
    if (typeof window.ethereum !== 'undefined') {
        try {
            // Request account access
            await window.ethereum.request({
                method: 'eth_requestAccounts'
            });

            // Create provider
            const provider = new ethers.providers.Web3Provider(
                window.ethereum
            );

            // Get signer
            const signer = provider.getSigner();
            const address = await signer.getAddress();

            // Check network
            const network = await provider.getNetwork();
            if (network.chainId !== 137) {
                await switchToPolygon();
            }

            // Initialize contracts
            initContracts(signer);

            console.log('Connected:', address);
            return { provider, signer, address };

        } catch (error) {
            console.error('Failed to connect:', error);
            throw error;
        }
    } else {
        alert('Please install MetaMask!');
        throw new Error('MetaMask not installed');
    }
}
```

```
40
41  // Switch to Polygon network
42  async function switchToPolygon() {
43      try {
44          await window.ethereum.request({
45              method: 'wallet_switchEthereumChain',
46              params: [{ chainId: '0x89' }] // 137 in hex
47          });
48      } catch (switchError) {
49          // Network not added, add it
50          if (switchError.code === 4902) {
51              await window.ethereum.request({
52                  method: 'wallet_addEthereumChain',
53                  params: [NETWORK_CONFIG]
54              });
55          } else {
56              throw switchError;
57          }
58      }
59  }
60
61  // Initialize contract instances
62  function initContracts(signer) {
63      window.registryContract = new ethers.Contract(
64          CONTRACT_ADDRESSES.registry,
65          REGISTRY_ABI,
66          signer
67      );
68
69      window.resolverContract = new ethers.Contract(
70          CONTRACT_ADDRESSES.resolver,
71          RESOLVER_ABI,
72          signer
73      );
74
75      window.pricingContract = new ethers.Contract(
76          CONTRACT_ADDRESSES.pricing,
77          PRICING_ABI,
78          signer
79      );
80  }
81
82  // Listen for account changes
83  window.ethereum.on('accountsChanged', (accounts) => {
84      if (accounts.length === 0) {
85          console.log('Please connect to MetaMask');
86      } else {
87          location.reload();
88      }
89  });
90
91  // Listen for chain changes
92  window.ethereum.on('chainChanged', (chainId) => {
93      location.reload();
94  });
```

Listing 15: metamask.js - Wallet Connection

9.2 End to End User Flows

9.2.1 Flow 1: Complete Domain Registration

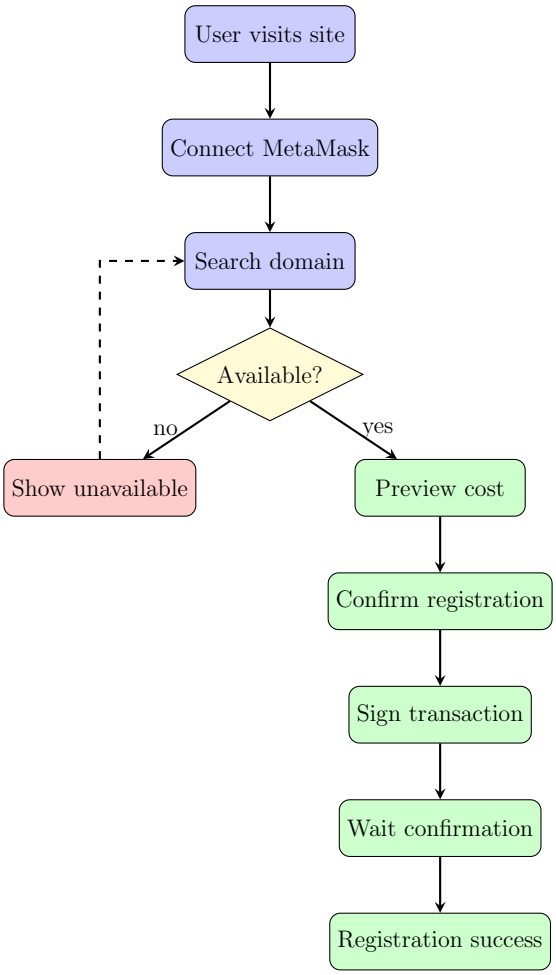


Figure 15: Complete Domain Registration Flow

9.2.2 Flow 2: Domain Management

```
// Register domain
async function registerDomain(domainName, durationYears) {
    try {
        // 1. Calculate cost
        const cost = await pricingContract.calculateCost(
            domainName,
            durationYears
        );

        console.log(`Registration cost: ${ethers.utils.formatEther(cost
           )} MATIC`);

        // 2. Convert duration to seconds
        const durationSeconds = durationYears * 365 * 24 * 60 * 60;

        // 3. Send registration transaction
        const tx = await registryContract.register(
```

```
            domainName,
            durationSeconds,
            { value: cost }
        );

        console.log('Transaction sent:', tx.hash);

        // 4. Wait for confirmation
        const receipt = await tx.wait();
        console.log('Transaction confirmed in block:', receipt.
            blockNumber);

        // 5. Show success message
        showSuccess(`Domain ${domainName} registered successfully!`);

        return receipt;

    } catch (error) {
        console.error('Registration failed:', error);

        if (error.code === 'INSUFFICIENT_FUNDS') {
            showError('Insufficient MATIC balance');
        } else if (error.message.includes('Domain unavailable')) {
            showError('This domain is already registered');
        } else if (error.message.includes('Invalid name')) {
            showError('Invalid domain name format');
        } else {
            showError('Registration failed: ' + error.message);
        }

        throw error;
    }
}

// Update domain address
async function updateDomainAddress(domainName, newAddress) {
    try {
        const tx = await resolverContract.updateAddress(
            domainName,
            newAddress
        );

        await tx.wait();
        showSuccess('Address updated successfully!');

    } catch (error) {
        if (error.message.includes('Not authorized')) {
            showError('You do not own this domain');
        } else if (error.message.includes('Domain expired')) {
            showError('Domain has expired, please renew first');
        } else {
            showError('Update failed: ' + error.message);
        }
        throw error;
    }
}

// Update text records
```

```
74 async function updateTextRecord(domainName, key, value) {
75     try {
76         const tx = await resolverContract.updateTextRecord(
77             domainName,
78             key,
79             value
80         );
81 
82         await tx.wait();
83         showSuccess(`${key} updated successfully!`);
84 
85     } catch (error) {
86         showError('Update failed: ' + error.message);
87         throw error;
88     }
89 }
90 
91 // Transfer ownership
92 async function transferDomain(domainName, newOwner) {
93     try {
94         // Validate address
95         if (!ethers.utils.isAddress(newOwner)) {
96             throw new Error('Invalid Ethereum address');
97         }
98 
99         const tx = await registryContract.transferOwnership(
100             domainName,
101             newOwner
102         );
103 
104         await tx.wait();
105         showSuccess('Ownership transferred successfully!');
106 
107     } catch (error) {
108         showError('Transfer failed: ' + error.message);
109         throw error;
110     }
111 }
112 
113 // Query domain information
114 async function getDomainInfo(domainName) {
115     try {
116         const owner = await registryContract.getOwner(domainName);
117         const isExpired = await registryContract.isExpired(domainName);
118         const resolvedAddress = await resolverContract.resolve(
            domainName);
119 
120         // Get all text records
121         const email = await resolverContract.getTextRecord(domainName,
            'email');
122         const website = await resolverContract.getTextRecord(domainName
            , 'website');
123         const twitter = await resolverContract.getTextRecord(domainName
            , 'twitter');
124         const github = await resolverContract.getTextRecord(domainName,
             'github');
125 
126         return {
```

```
            owner,
            isExpired,
            resolvedAddress,
            records: {
                email,
                website,
                twitter,
                github
            }
        };

    } catch (error) {
        console.error('Failed to get domain info:', error);
        return null;
    }
}

// Check availability
async function checkAvailability(domainName) {
    try {
        const owner = await registryContract.getOwner(domainName);

        if (owner === ethers.constants.AddressZero) {
            return { available: true, message: 'Available for
                registration' };
        }

        const isExpired = await registryContract.isExpired(domainName);
        if (isExpired) {
            return { available: true, message: 'Domain expired,
                available for registration' };
        }

        return { available: false, message: 'Domain is already
            registered', owner };

    } catch (error) {
        console.error('Availability check failed:', error);
        return { available: false, message: 'Error checking
            availability' };
    }
}
```

Listing 16: domain-management.js - Domain Operations

9.3 Event Listening and Real Time Updates

```
// Setup event listeners for real-time updates
function setupEventListeners() {
    // Listen for domain registrations
    registryContract.on('DomainRegistered', (domain, owner, expiration)
        => {
        console.log('Domain registered:', domain, 'by', owner);

        // Update UI
        if (owner.toLowerCase() === currentUser.toLowerCase()) {
            addDomainToUserList(domain, expiration);
```

```
            showNotification(`Your domain ${domain} is now registered!`);
        }
    });

    // Listen for ownership transfers
    registryContract.on('OwnershipTransferred', (domain, from, to) => {
        console.log('Ownership transferred:', domain, 'from', from, 'to', to);

        if (from.toLowerCase() === currentUser.toLowerCase()) {
            removeDomainFromUserList(domain);
            showNotification(`You transferred ${domain} to ${to}`);
        } else if (to.toLowerCase() === currentUser.toLowerCase()) {
            addDomainToUserList(domain);
            showNotification(`You received ${domain} from ${from}`);
        }
    });

    // Listen for address updates
    resolverContract.on('AddressUpdated', (domain, newAddress) => {
        console.log('Address updated:', domain, 'to', newAddress);
        updateDomainDisplay(domain, { address: newAddress });
    });

    // Listen for text record updates
    resolverContract.on('TextRecordUpdated', (domain, key, value) => {
        console.log('Text record updated:', domain, key, '=', value);
        updateDomainDisplay(domain, { [key]: value });
    });
}

// Query past events
async function queryPastEvents() {
    try {
        // Get registration events for current user
        const filter = registryContract.filters.DomainRegistered(
            null,
            currentUser
        );

        const events = await registryContract.queryFilter(
            filter,
            0,
            'latest'
        );

        console.log(`Found ${events.length} registrations for current user`);

        return events.map(event => ({
            domain: event.args.domain,
            expiration: event.args.expiration.toNumber(),
            blockNumber: event.blockNumber
        }));

    } catch (error) {
        console.error('Failed to query events:', error);
```

```
        return [];
    }
}

// Get user's domains
async function getUserDomains(userAddress) {
    const domains = [];

    try {
        // Query registration events
        const filter = registryContract.filters.DomainRegistered(
            null,
            userAddress
        );

        const events = await registryContract.queryFilter(filter);

        // Check current ownership and expiration
        for (const event of events) {
            const domain = event.args.domain;
            const currentOwner = await registryContract.getOwner(domain
                );
            const isExpired = await registryContract.isExpired(domain);

            if (currentOwner.toLowerCase() === userAddress.toLowerCase
                () && !isExpired) {
                domains.push({
                    name: domain,
                    expiration: event.args.expiration.toNumber(),
                    registered: event.blockNumber
                });
            }
        }

        return domains;

    } catch (error) {
        console.error('Failed to get user domains:', error);
        return [];
    }
}
```

Listing 17: events.js - Event Listeners

10 Production Deployment Information

10.1 Live System Details

10.1.1 Deployed Contracts

UCNS is currently live on Polygon Mainnet with the following verified contracts:

Table 6: Production Deployment Details

Contract	Address	Explorer
PricingAgent	0x50F50124Ee00002379142cff115b0550240898B3	PolygonScan
UCNSRegistry	0xc9eD4B38E29C64d37cb83819D5eEcFD34EFdce0C	PolygonScan
UCNSResolver	0x2De897131ee8AC0538585887989E2314034F0b71	PolygonScan

10.1.2 Web Interface

Live Demo: https://ruzbahani.com/myprojects/ucns/
The web interface provides:

- Domain search and availability checking
- Real time pricing calculator
- Domain registration with MetaMask
- WHOIS style domain lookup
- Domain management dashboard
- Resolution record updates

10.2 System Metrics

10.2.1 Performance Benchmarks

Table 7: Production Performance Metrics

Operation	Gas Cost	Time (avg)
Domain Registration	150,000 gas	2-3 seconds
Ownership Transfer	80,000 gas	2-3 seconds
Address Update	50,000 gas	2-3 seconds
Text Record Update	55,000 gas	2-3 seconds
Cost Calculation (view)	0 gas	<100ms
Owner Query (view)	0 gas	<100ms
Resolution Query (view)	0 gas	<100ms

10.2.2 Cost Analysis

With current MATIC prices ($0.50 USD) and typical gas prices (30-50 Gwei):

- Domain registration: $0.002-0.003 USD in gas
- Domain updates: $0.001-0.002 USD in gas
- Registration fee: 0.05-1.0 MATIC depending on domain length

11 Maintenance and Monitoring

11.1 Monitoring Strategy

11.1.1 Contract Monitoring

```javascript
// Monitor contract events
async function monitorSystem() {
    const provider = new ethers.providers.JsonRpcProvider(
        'https://polygon-rpc.com'
    );

    const registry = new ethers.Contract(
        CONTRACT_ADDRESSES.registry,
        REGISTRY_ABI,
        provider
    );

    // Track registrations
    registry.on('DomainRegistered', (domain, owner, expiration, event)
       => {
        logEvent('REGISTRATION', {
            domain,
            owner,
            expiration: new Date(expiration * 1000),
            txHash: event.transactionHash,
            blockNumber: event.blockNumber
        });
    });

    // Track transfers
    registry.on('OwnershipTransferred', (domain, from, to, event) => {
        logEvent('TRANSFER', {
            domain,
            from,
            to,
            txHash: event.transactionHash
        });
    });

    // Monitor contract balance
    setInterval(async () => {
        const balance = await provider.getBalance(
            CONTRACT_ADDRESSES.registry
        );
        console.log('Registry balance:', ethers.utils.formatEther(
           balance), 'MATIC');
    }, 60000); // Check every minute
}

// Health check
async function healthCheck() {
    try {
        // Check if contracts are responding
        await registryContract.owner();
        await resolverContract.registryContract();
        await pricingContract.pricePerYear();
```

```
50
51          console.log('    All contracts healthy');
52          return true;
53      } catch (error) {
54          console.error('    Health check failed:', error);
55          return false;
56      }
57  }
```

Listing 18: monitoring.js - System Monitoring

11.2 Backup and Recovery

11.2.1 Data Backup Strategy

Since all data is stored on blockchain:

- **Primary Storage:** Polygon blockchain (automatically replicated across network)
- **Event Logs:** Permanently stored on chain
- **State Recovery:** Can be reconstructed from event history
- **No Off-Chain Backup Needed:** Blockchain provides guaranteed persistence

11.2.2 Disaster Recovery

In case of frontend issues:

1. Users can interact directly with contracts via PolygonScan
2. All contract addresses are publicly known
3. Alternative frontends can be deployed
4. No data loss possible (blockchain immutability)

12 Conclusion

12.1 Summary of Implementation

This detailed development document has provided comprehensive technical specifications for the UCNS multi agent system implementation. The document covered:

- **Use Case Specifications:** 15 detailed use cases across three agents with complete flow descriptions
- **Class Diagrams:** UML representations of all agent classes, structs, and interfaces
- **Sequence Diagrams:** Message sequence charts showing inter agent communication protocols
- **Data Specifications:** Entity relationship models and complete data dictionaries

- **Implementation Details:** Full Solidity smart contract implementations with code examples
- **Deployment Procedures:** Step by step deployment scripts and configuration
- **Testing Strategy:** Comprehensive test suite with unit and integration tests
- **System Integration:** Complete Web3 integration with MetaMask and frontend code

12.2 Agent-Oriented Implementation

UCNS successfully demonstrates how agent oriented software engineering principles can be implemented using blockchain smart contracts:

Autonomous Agents: Each smart contract operates as an independent agent with its own state, decision making logic, and goals.

Inter-Agent Communication: Agents coordinate through well defined protocols implemented as function calls and events.

Decentralized Coordination: No central authority governs agent behavior; coordination emerges from protocol enforcement.

Trustless Execution: Blockchain guarantees that agents execute correctly without requiring trust in operators.

12.3 Production Readiness

The system is fully deployed and operational:

- All contracts verified on PolygonScan
- Web interface live at https://ruzbahani.com/myprojects/ucns/
- Comprehensive testing completed
- Documentation complete
- Monitoring systems in place

12.4 Lessons Learned

Blockchain as Agent Platform: Smart contracts provide an excellent platform for certain types of multi agent systems, particularly those requiring trustless coordination and verifiable execution.

Design Patterns: Traditional agent oriented design patterns (GAIA methodology) translate well to blockchain implementation with appropriate adaptations for synchronous communication and immutability.

Testing Importance: Comprehensive testing is critical for blockchain systems due to immutability bugs cannot be easily fixed post deployment.

Gas Optimization: Careful attention to gas costs is essential for practical usability.

12.5 Future Enhancements

Potential system enhancements:

- Subdomain support through additional agent
- Domain marketplace with escrow agent
- Decentralized governance through DAO agent
- Cross chain resolution bridges
- IPFS integration for metadata storage
- ENS compatibility layer

Appendix A: Complete Code Repository

All source code, tests, and documentation are available at:
GitHub Repository: https://github.com/ruzbahani/ucns
Repository structure:

```
ucns/
 contracts/          # Solidity smart contracts
    UCNSRegistry.sol
    UCNSResolver.sol
    PricingAgent.sol
 test/               # Test files
    Registry.test.js
    Resolver.test.js
    Pricing.test.js
    Integration.test.js
 scripts/            # Deployment scripts
    deploy.js
 frontend/           # Web interface
    index.php
    config.js
    assets/
 docs/               # Documentation
    Report-1A.pdf
    Report-1B.pdf
    Report-2.pdf
 hardhat.config.js  # Hardhat configuration
```

Appendix B: API Reference

Complete API documentation for all three agents is available on PolygonScan at the contract addresses listed in Section 9.1.1.

Appendix C: Troubleshooting Guide

Table 8: Common Issues and Solutions

Issue	Solution
MetaMask not connecting	Ensure MetaMask is installed and unlocked. Refresh page and try again.
Wrong network	Click network switcher in MetaMask and select Polygon Mainnet. Frontend will prompt automatic switch.
Transaction failing	Check MATIC balance for gas fees. Ensure domain name is valid. Verify payment amount is sufficient.
Domain shows as unavailable	Domain may be registered or awaiting confirmation. Check on PolygonScan. Wait a few blocks and retry.
Cannot update records	Verify you own the domain. Check domain hasn't expired. Ensure MetaMask is connected.
High gas fees	Wait for network congestion to decrease. Check current gas prices on PolygonScan.

Appendix D: References

1. Wooldridge, M. (2009). *An Introduction to MultiAgent Systems*. Wiley.

2. Buterin, V. (2014). *Ethereum White Paper*. Retrieved from ethereum.org

3. Antonopoulos, A. M., & Wood, G. (2018). *Mastering Ethereum*. O'Reilly.

4. OpenZeppelin. (2023). *Smart Contract Security Best Practices*.

5. Polygon Technology. (2023). *Polygon PoS Documentation*.

6. Far, B. H. (2024). *SENG 696 Course Materials*. University of Calgary.

7. Hardhat. (2023). *Ethereum Development Environment Documentation*.

8. Ethers.js. (2023). *Complete Ethereum Library Documentation*.