Report 1B

GAIA-Based Design Document

UCNS

University of Calgary Name Service

Multi-Agent System Design Using GAIA Methodology

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 presents a comprehensive multi agent system design for the University of Calgary Name Service (UCNS) using the GAIA methodology. GAIA (Generic Architecture for Information Availability) provides a systematic approach to analyzing and designing agent-based systems through well defined phases: requirements analysis, architectural design, and detailed design. This report applies GAIA's modeling concepts including roles, protocols, agent types, services, and acquaintances to design a decentralized naming service implemented as smart contracts on the Polygon blockchain. The document details the complete design process from initial goal identification through agent internal architecture specification, demonstrating how blockchain based agents can be systematically designed using established agent oriented software engineering methodologies. The resulting design provides a blueprint for implementing three autonomous agents (Registry, Resolver, and Pricing) that coordinate to deliver comprehensive domain name registration and resolution services in a trustless, decentralized environment.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose and Scope

This design document applies the GAIA (Generic Architecture for Information Availability) methodology to systematically design the University of Calgary Name Service (UCNS) as a multi agent system. GAIA, developed by Wooldridge, Jennings, and Kinny, provides a structured approach to agent-based system development that bridges the gap between requirements analysis and implementation through explicit modeling of agent oriented concepts.

The primary purpose of this document is to transform the system specification presented in Report 1A into a detailed multi agent design that can be directly implemented as smart contracts on the Polygon blockchain. By applying GAIA's three phase approach analysis, architectural design, and detailed design, we demonstrate how theoretical agent oriented principles manifest in practical blockchain system architecture.

The scope encompasses:

- **Requirements Analysis:** Identification of system goals, decomposition into organizational roles, and definition of interaction protocols
- **Architectural Design:** Definition of agent types, assignment of roles to agents, and specification of inter agent communication patterns
- **Detailed Design:** Internal architecture of each agent, service specification, and coordination mechanisms

1.2 GAIA Methodology Overview

GAIA is specifically designed for analyzing and designing agent-based systems, providing constructs that directly capture agent oriented concepts rather than forcing them into object oriented or procedural paradigms. The methodology emphasizes:

Organization Oriented Decomposition: Systems are decomposed into organizational structures with defined roles and responsibilities rather than functional modules or object hierarchies.

Role Based Analysis: The fundamental unit of decomposition is the role, a collection of responsibilities, permissions, activities, and protocols. Roles capture what agents do independently of how they do it.

Protocol Centric Interaction: Agent interactions are modeled as protocols defining message sequences, constraints, and coordination patterns.

Clear Separation of Concerns: GAIA maintains distinct analysis (what the system does), architectural design (which agents exist and how they relate), and detailed design (how each agent works internally) phases.

The methodology produces several key artifacts:

1. **Role Models:** Schema definitions of organizational roles
2. **Interaction Models:** Protocol specifications for agent communication
3. **Agent Models:** Mapping of roles to agent instances

4. **Service Models:** Functional specifications of agent capabilities
5. **Acquaintance Models:** Communication topology between agents

1.3 UCNS Context

UCNS provides a decentralized blockchain based naming service where human-readable domain names map to Ethereum addresses and associated metadata. The system must handle domain registration, ownership tracking, resolution services, and dynamic pricing without centralized authority. This inherently multi agent problem space makes GAIA particularly suitable for design.

The blockchain context introduces unique constraints and opportunities:

- Agents are implemented as immutable smart contracts with deterministic behavior
- Communication occurs through function calls and events on the blockchain
- State is globally visible but cryptographically secured
- Coordination must be trustless, relying on protocol enforcement rather than trust assumptions
- Performance and cost considerations drive architectural decisions

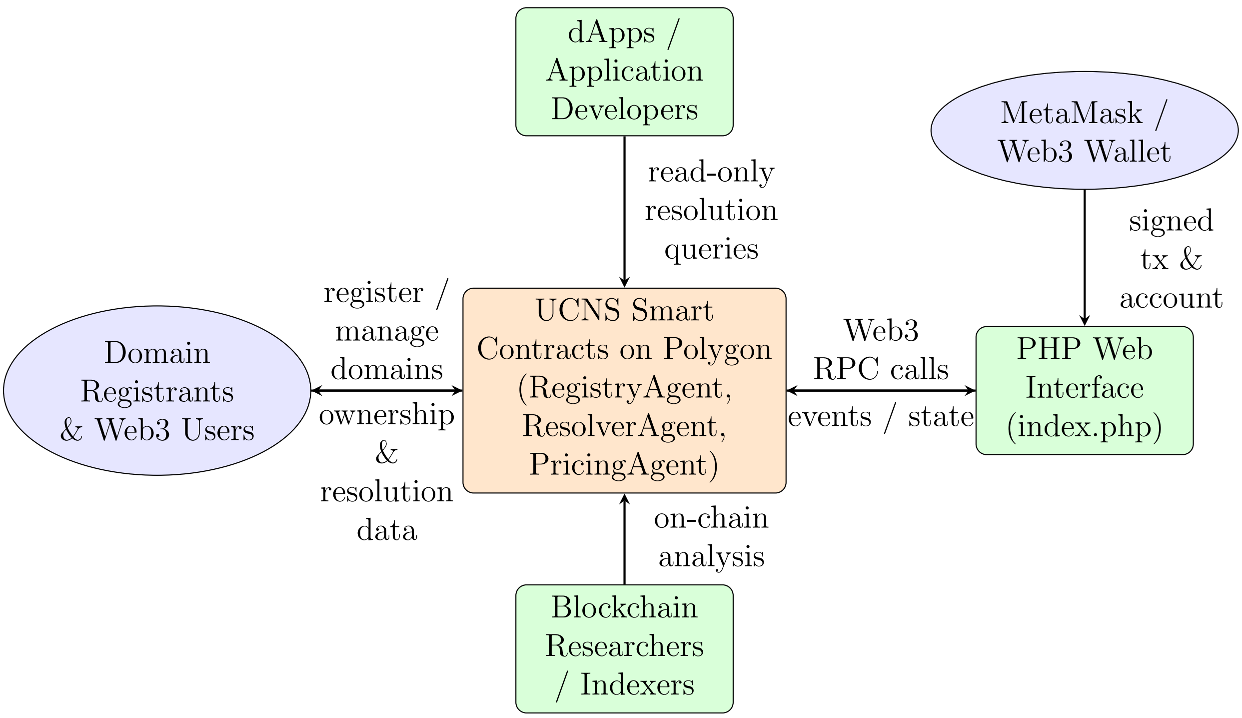

Figure 1: UCNS system context and external actor interactions

The UCNS platform operates within a broader ecosystem that includes human users, decentralized applications, researchers, and an off chain web interface. These actors interact with the system primarily through Web3 enabled wallets such as MetaMask and through the UCNS smart contracts deployed on the Polygon PoS blockchain. Figure 1 provides a high level context view, showing how domain registrants, dApps, indexers, and the PHP web frontend connect to the Registry, Resolver, and Pricing components of UCNS. This contextual model clarifies the system boundaries and sets the stage for the role and agent definitions developed in later sections.

1.4 Document Structure

This document follows GAIA's three phase structure:

Section 2: Requirements Analysis: Identifies system goals, decomposes them into roles, and specifies interaction protocols. This phase answers "what does the system do?"

Section 3: Architectural Design: Defines agent types, maps roles to agents, and establishes the system's organizational structure. This phase answers "what agents exist and how do they relate?"

Section 4: Detailed Design: Specifies internal agent architecture, service definitions, and coordination mechanisms. This phase answers "how does each agent work internally?"

Section 5: Design Validation: Evaluates the design against requirements and identifies potential issues.

Section 6: Implementation Mapping: Bridges the design to Solidity smart contract implementation.

2 Phase 1: Requirements Analysis

2.1 Goal Identification

GAIA's requirements analysis begins with identifying high level organizational goals that the multi agent system must achieve. Goals represent desired states or properties the organization should exhibit, independent of how those states are achieved.

2.1.1 Primary Goals

G1: Autonomous Domain Registration

Description: Enable users to register human readable domain names with cryptographic ownership guarantees without requiring trusted intermediaries.

Success Criteria:

- Registration completes atomically with payment and ownership record
- Ownership is cryptographically verifiable and tamper proof
- No centralized authority can revoke or alter registrations unilaterally
- Registration costs are deterministically computed and transparent

Rationale: This goal captures the core purpose of UCNS providing decentralized identity through blockchain based naming.

G2: Reliable Name Resolution

Description: Provide deterministic, tamper proof resolution of domain names to addresses and metadata.

Success Criteria:

- Resolution queries return current, authoritative data
- Only authorized parties can modify resolution records
- Resolution data integrity is cryptographically guaranteed
- Resolution logic is transparent and verifiable

Rationale: Resolution is the fundamental service consumers need from a naming system.

G3: Dynamic Economic Management

Description: Implement flexible pricing mechanisms that reflect domain value while remaining transparent and predictable.

Success Criteria:

- Pricing calculations are deterministic and publicly verifiable
- Pricing policy can adapt to market conditions without disrupting existing registrations
- Shorter domains command premium pricing reflecting higher perceived value
- Registration duration affects total cost proportionally

Rationale: Economic incentives shape system usage patterns and sustainability.

G4: Decentralized Ownership Management

Description: Enable complete lifecycle management of domain ownership including transfers and renewals without centralized coordination.

Success Criteria:

- Ownership can be transferred atomically and securely
- Expiration tracking is automatic and deterministic
- Ownership history is fully auditable
- Authorization delegation is supported for organizational use cases

Rationale: Ownership management is critical for domains to function as digital assets.

2.1.2 Secondary Goals

G5: System Transparency and Auditability

Description: Ensure all system actions are publicly visible and auditable.

Success Criteria:

- All state changes emit events
- Contract logic is verifiable through source code publication
- Historical actions are reconstructable from blockchain logs

G6: Gas Efficiency

Description: Minimize computational costs while maintaining functionality.

Success Criteria:

- Operations use optimized storage patterns
- Redundant computations are eliminated
- View functions enable free queries

G7: Extensibility

Description: Support future enhancements without requiring complete system redesign.

Success Criteria:

- Clear agent boundaries enable independent evolution
- New agents can be added through defined interfaces
- Existing agents do not need modification for system extensions

2.1.3 Goal Hierarchy

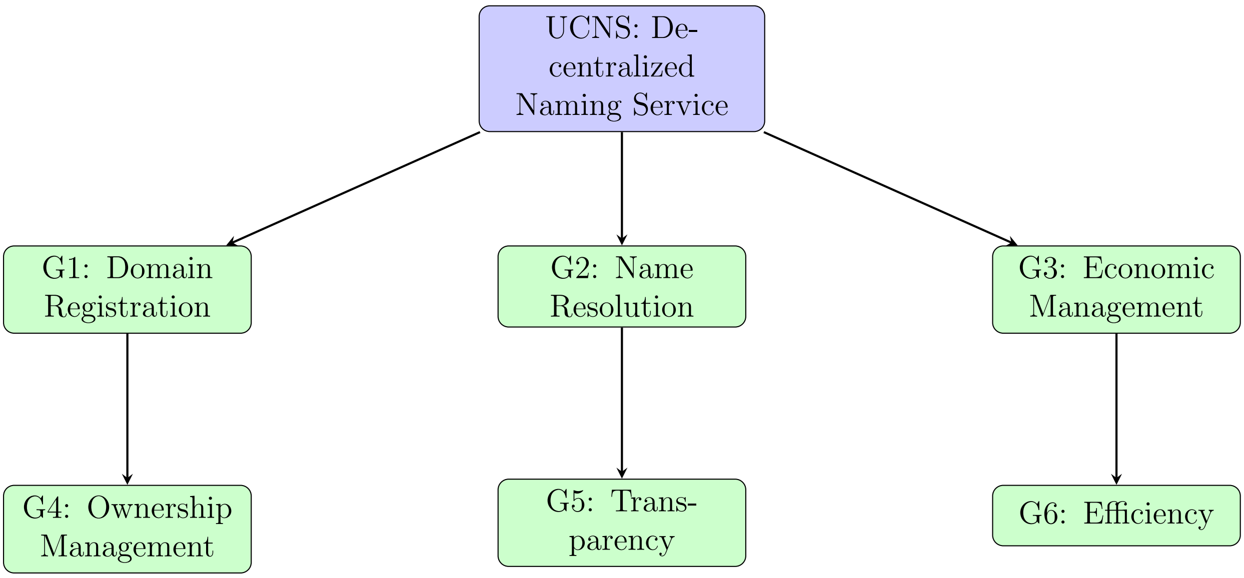

Figure 2: UCNS Goal Hierarchy

The goal hierarchy establishes the functional and non functional intentions that UCNS must satisfy. Moving from goals to operational behavior requires identifying the organizational roles responsible for achieving these objectives. In GAIA, roles act as the conceptual bridge between abstract goals and concrete agent behaviors. The next subsection formalizes these roles and their responsibilities, forming the basis for the interaction protocols developed later in Phase 2.

2.2 Role Identification

GAIA decomposes organizational goals into roles collections of responsibilities, permissions, activities, and protocols. Roles represent functional positions within the organization that agents will fulfill.

2.2.1 Role 1: Registry Manager

Description: Maintains authoritative ownership records for all registered domains, enforces naming rules, and coordinates registration processes.

Responsibilities:

1. Validate domain name syntax and availability
2. Record domain ownership with expiration timestamps
3. Process ownership transfers between parties
4. Track operator authorizations for delegated management
5. Enforce reserved name restrictions
6. Coordinate with pricing authority for cost determination
7. Emit events for all ownership state changes

Permissions:

- Read: All domain ownership records, reserved name list
- Write: Domain ownership mappings, expiration timestamps, operator authorizations
- Execute: Ownership transfer logic, registration validation
- Query: Pricing authority for cost calculations

Activities:

- `ValidateDomainName`: Verify name adheres to syntax rules
- `CheckAvailability`: Determine if domain is unregistered or expired
- `RegisterDomain`: Atomically record ownership with payment verification
- `TransferOwnership`: Update owner field with authorization check
- `AuthorizeOperator`: Grant management permissions to designated address
- `VerifyExpiration`: Check if domain has exceeded expiration timestamp

Protocols: RegistrationProtocol, OwnershipTransferProtocol, PricingQueryProtocol, OwnershipVerificationProtocol

Liveness Properties:

- Registry Manager processes all incoming registration requests
- Ownership records are updated within transaction finality time
- Expired domains become available for re registration immediately upon expiration

Safety Properties:

- No domain has multiple simultaneous owners
- Ownership can only change through authorized transfers or expiration
- Reserved names cannot be registered

2.2.2 Role 2: Resolution Provider

Description: Maps domain names to addresses and metadata, enforcing authorization for updates and providing query services.

Responsibilities:

1. Store resolution records mapping domains to addresses
2. Maintain extended metadata (email, avatar, social profiles)
3. Verify ownership authorization before accepting updates
4. Provide query interfaces for address resolution
5. Support multiple record types with type specific retrieval
6. Coordinate with registry for ownership validation

Permissions:

- Read: All resolution records, registry ownership data
- Write: Resolution record mappings for authorized domains
- Execute: Authorization verification, record update logic
- Query: Registry manager for ownership verification

Activities:

- `ResolveAddress`: Return primary Ethereum address for domain
- `UpdateRecord`: Modify resolution data after authorization check
- `GetTextRecord`: Retrieve specific metadata field
- `GetAllRecords`: Return complete resolution record structure
- `VerifyAuthorization`: Confirm caller owns domain via registry query

Protocols: RecordUpdateProtocol, ResolutionQueryProtocol, AuthorizationVerificationProtocol

Liveness Properties:

- Resolution queries always complete deterministically
- Record updates process within transaction finality time
- Authorization checks occur before every state modification

Safety Properties:

- Only domain owners can modify resolution records
- Resolution data cannot be altered by unauthorized parties
- Query operations never modify state (read only guarantee)

2.2.3 Role 3: Pricing Authority

Description: Computes registration costs based on domain characteristics and market parameters, providing transparent and deterministic pricing.

Responsibilities:

1. Calculate registration costs based on domain length
2. Apply duration based pricing for multi year registrations
3. Maintain configurable pricing tier structures
4. Provide cost preview capabilities for users
5. Enable owner controlled pricing policy updates

Permissions:

- Read: Pricing tier configuration, domain length
- Write: Pricing tier parameters (owner only)
- Execute: Cost calculation algorithms

Activities:

- `CalculateCost`: Compute total registration cost
- `GetBasePriceForLength`: Return tier specific base price
- `UpdatePricingTier`: Modify pricing configuration (owner only)
- `PreviewCost`: Provide cost estimate without state changes

Protocols: PricingQueryProtocol, PricingUpdateProtocol

Liveness Properties:

- Pricing calculations always terminate with deterministic results
- Pricing queries complete in constant time

Safety Properties:

- Pricing calculations are pure functions without side effects
- Only contract owner can modify pricing tiers
- Pricing changes do not affect existing registrations retroactively

2.2.4 Role Schema Summary

Table 1: Role Schema Summary

Role	Primary Goals	Key Activities
Registry Manager	G1, G4	Domain validation, ownership recording, transfer processing, expiration tracking
Resolution Provider	G2, G5	Address resolution, metadata management, authorization verification
Pricing Authority	G3, G6	Cost calculation, pricing tier management, transparent fee computation

The three core roles of UCNS: Registry Manager, Resolution Provider, and Pricing Authority capture the essential responsibilities of the system. Each role encapsulates a distinct viewpoint: ownership management, resolution services, and pricing governance. These roles are not yet agents; instead, they represent organizational abstractions that will later be mapped to agent types in Phase 2. Figure 3 summarizes the three roles and highlights their responsibilities and interdependencies, providing a clear foundation for defining the interaction protocols.

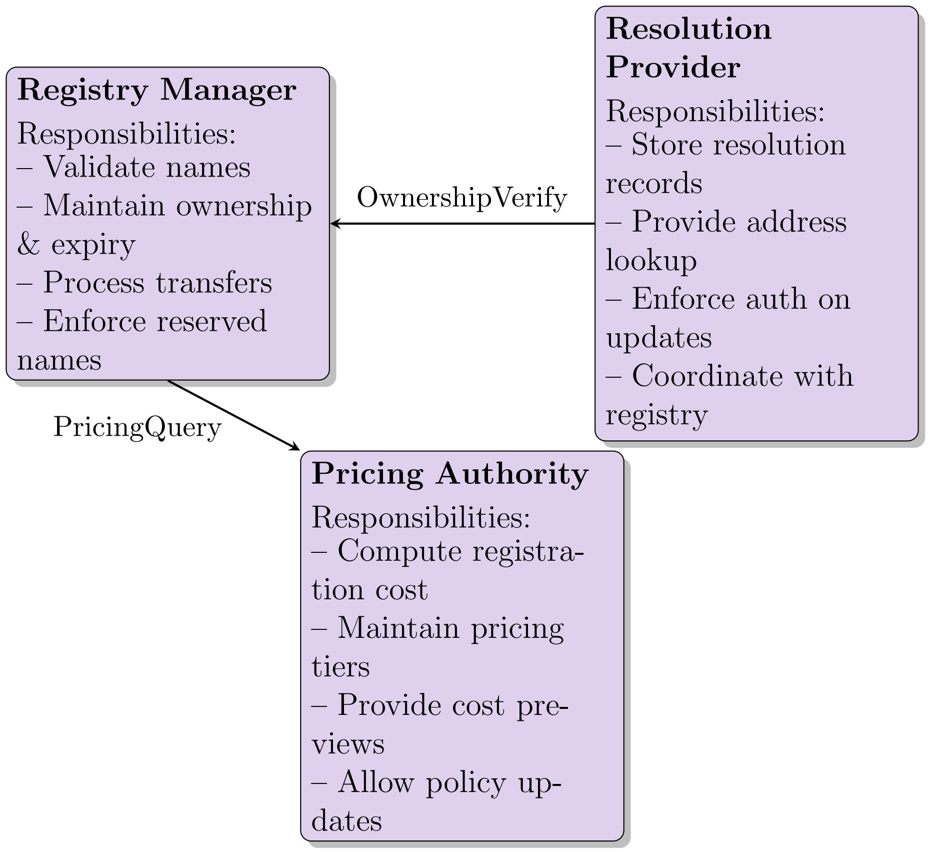

Figure 3: GAIA role model for UCNS organization

2.3 Interaction Protocols

Having defined the roles, GAIA next specifies how these roles interact through structured communication protocols. These protocols define permissible message flows, the conditions under which actions occur, and expected outcomes. They formalize the cooperative behavior of UCNS in a way that can later be translated into agent behaviour and smart contract functions. The following subsections detail the three principal protocols: Registration, Resolution, and Ownership Transfer and Figures 4 to 6 provide operational flowcharts to ground these interactions.

2.3.1 Protocol 1: RegistrationProtocol

The RegistrationProtocol governs how users register a new domain and how the Registry and Pricing roles coordinate to validate the request, compute costs, and update ownership. To supplement the message level description, Figure 4 illustrates the end to end control flow, including all validation, pricing, and failure branches.

Purpose: Coordinate domain registration between user, Registry Manager, and Pricing Authority.

Participants:

- Initiator: User (external to agent system)
- Responder: Registry Manager
- Consulted: Pricing Authority

Inputs:

- `domainName`: Desired domain string
- `duration`: Registration period in years
- `payment`: MATIC amount sent with transaction

Outputs:

- `success`: Boolean indicating registration success
- `domainHash`: Keccak256 hash of registered domain
- `expirationTimestamp`: Unix timestamp of expiration

Message Sequence:

1. User → Registry Manager: `register(domainName, duration)` with payment
2. Registry Manager → Registry Manager: `validateName(domainName)`
3. Registry Manager → Registry Manager: `checkAvailability(domainHash)`
4. Registry Manager → Pricing Authority: `calculateCost(domainName, duration)`
5. Pricing Authority → Registry Manager: `cost(amount)`
6. Registry Manager → Registry Manager: `verifyPayment(payment, cost)`

7. Registry Manager → Registry Manager: `recordOwnership(domainHash, owner, expiration)`
8. Registry Manager → User: `DomainRegistered(domainName, owner, expiration)` event

Preconditions:

- Domain name adheres to syntax rules
- Domain is unregistered or expired
- Payment equals or exceeds calculated cost
- Duration is within allowed range (1 - 10 years)

Postconditions:

- Domain ownership recorded with caller as owner
- Expiration timestamp set to current time + duration
- Payment transferred to registry contract
- Registration event emitted

Failure Conditions:

- Invalid name format → Revert with "Invalid name"
- Domain unavailable → Revert with "Domain already registered"
- Insufficient payment → Revert with "Insufficient payment"

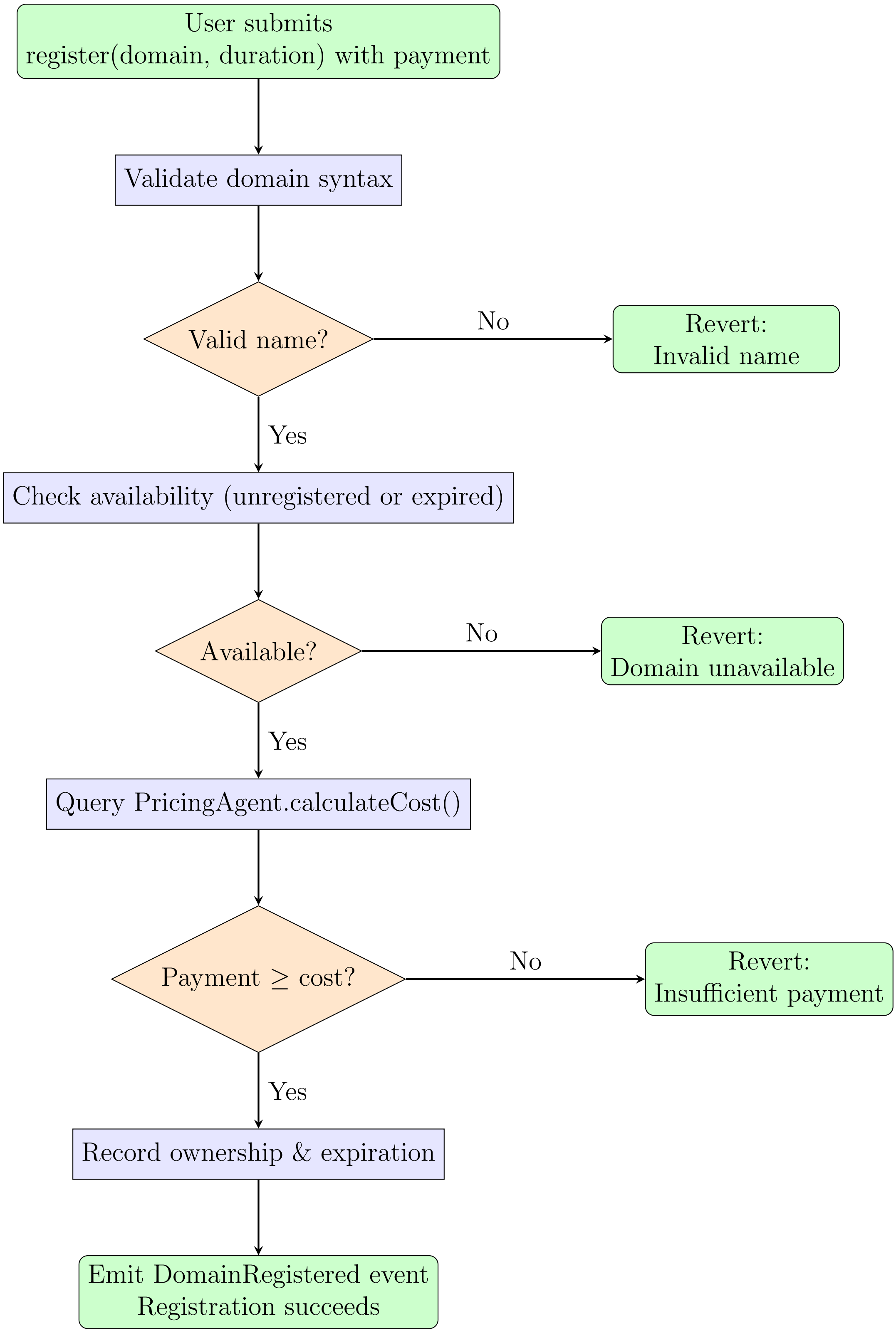

Figure 4: Flowchart of UCNS domain registration process

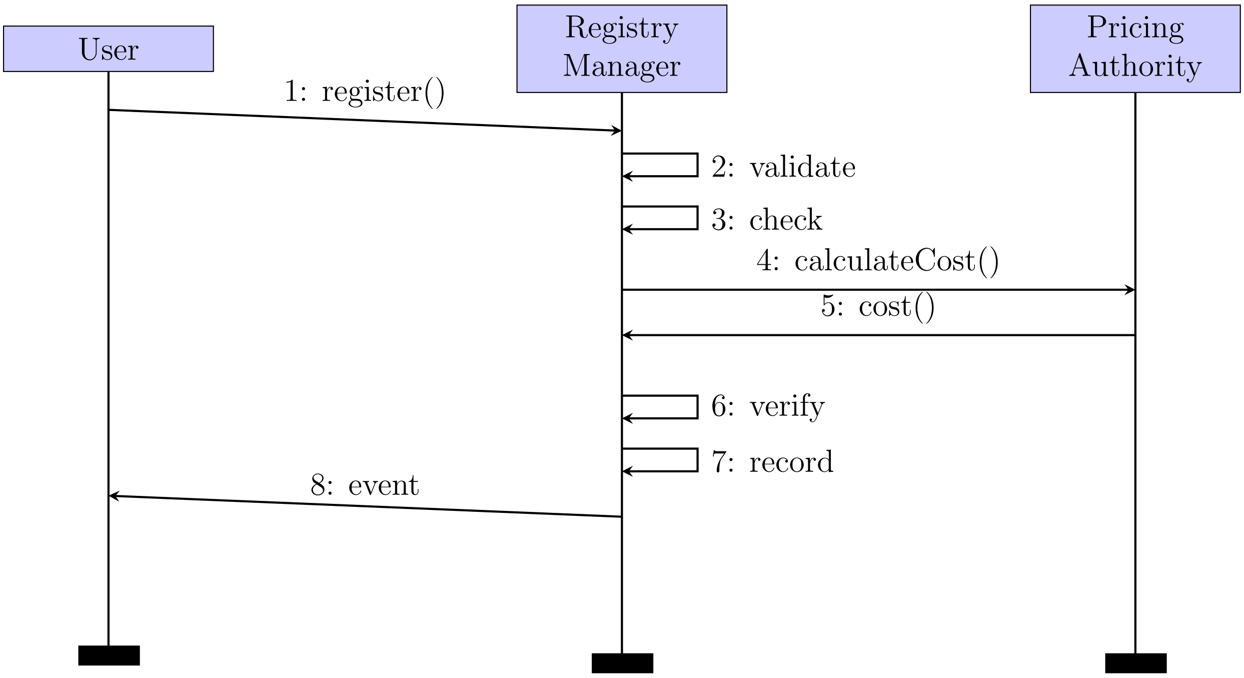

Figure 5: Registration Protocol Sequence Diagram

2.3.2 Protocol 2: RecordUpdateProtocol

Purpose: Enable authorized updates to domain resolution records with ownership verification.

Participants:

- Initiator: Domain Owner
- Responder: Resolution Provider
- Consulted: Registry Manager

Inputs:

- `domainName`: Target domain
- `recordType`: Type of record to update (address, email, etc.)
- `newValue`: Updated value for record

Outputs:

- `success`: Boolean indicating update success

Message Sequence:

1. Owner → Resolution Provider: `updateRecord(domainName, recordType, newValue)`
2. Resolution Provider → Registry Manager: `getOwner(domainHash)`
3. Registry Manager → Resolution Provider: `owner(address)`
4. Resolution Provider → Resolution Provider: `verifyOwnership(caller, owner)`

5. Resolution Provider → Registry Manager: `isExpired(domainHash)`
6. Registry Manager → Resolution Provider: `expired(boolean)`
7. Resolution Provider → Resolution Provider: `updateRecordInternal(domainHash, recordType, newValue)`
8. Resolution Provider → Owner: `RecordUpdated(domainName, recordType)` event

Preconditions:

- Caller is domain owner or authorized operator
- Domain has not expired
- New value is valid for record type

Postconditions:

- Resolution record updated with new value
- Update event emitted
- Previous value overwritten

Failure Conditions:

- Caller not owner → Revert with "Not authorized"
- Domain expired → Revert with "Domain expired"
- Invalid value format → Revert with "Invalid value"

2.3.3 Protocol 3: OwnershipTransferProtocol

Purpose: Transfer domain ownership from current owner to new owner atomically.
Participants:

- Initiator: Current Owner
- Responder: Registry Manager

Inputs:

- `domainName`: Domain to transfer
- `newOwner`: Address of new owner

Outputs:

- `success`: Boolean indicating transfer success

Message Sequence:

1. Owner → Registry Manager: `transferOwnership(domainName, newOwner)`
2. Registry Manager → Registry Manager: `verifyOwnership(caller, domainHash)`

3. Registry Manager → Registry Manager: `checkExpiration(domainHash)`
4. Registry Manager → Registry Manager: `validateNewOwner(newOwner)`
5. Registry Manager → Registry Manager: `updateOwner(domainHash, newOwner)`
6. Registry Manager → Owner: `OwnershipTransferred(domainName, oldOwner, newOwner)` event

Preconditions:

- Caller is current owner
- Domain has not expired
- New owner address is not zero address

Postconditions:

- Owner field updated to new owner address
- Transfer event emitted
- Expiration timestamp unchanged

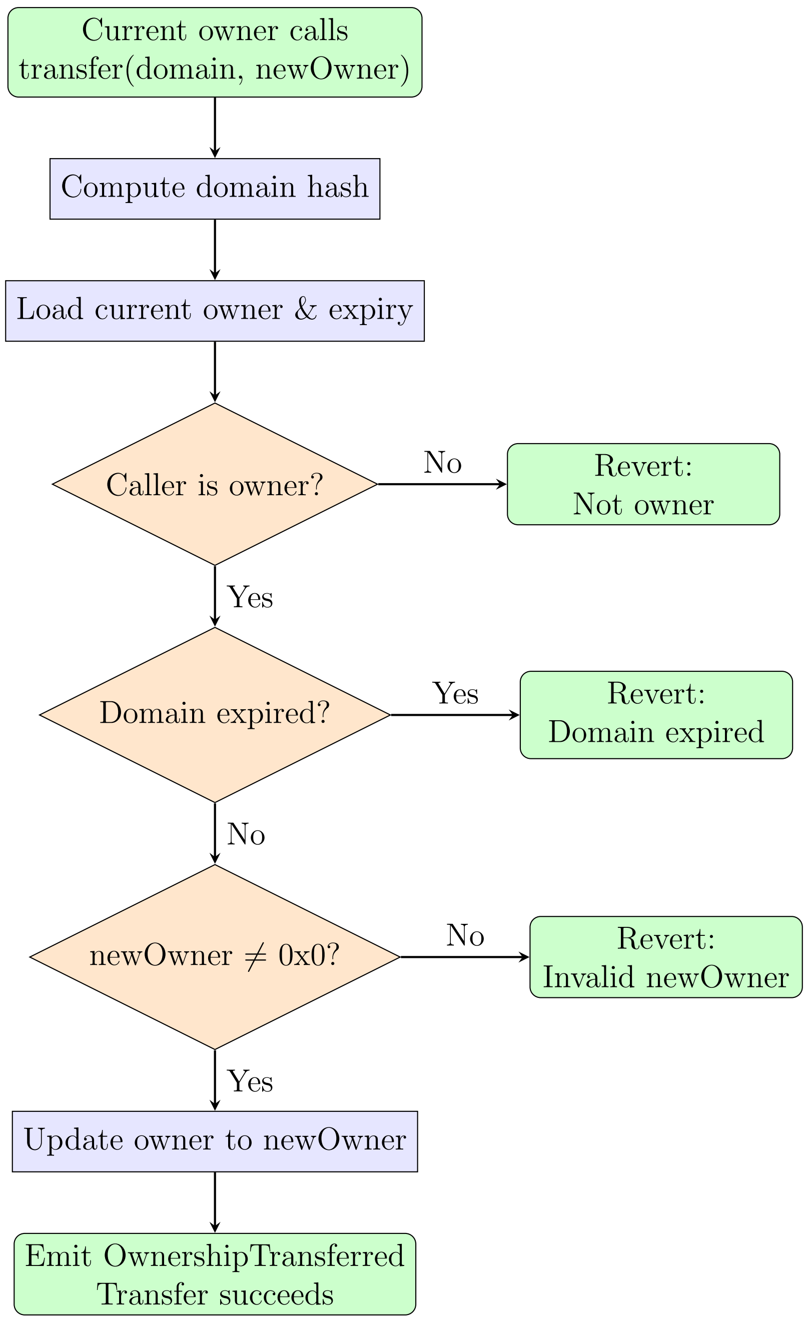

Figure 6: Flowchart of UCNS ownership transfer process

2.3.4 Protocol 4: PricingQueryProtocol

Purpose: Request cost calculation from Pricing Authority.

Participants:

- Initiator: Registry Manager or User
- Responder: Pricing Authority

Inputs:

- `domainName`: Domain for pricing
- `duration`: Registration duration in years

Outputs:

- `cost`: Total registration cost in MATIC

Message Sequence:

1. Initiator → Pricing Authority: `calculateCost(domainName, duration)`
2. Pricing Authority → Pricing Authority: `getLength(domainName)`
3. Pricing Authority → Pricing Authority: `getBasePriceForLength(length)`
4. Pricing Authority → Pricing Authority: `calculateDurationCost(duration)`
5. Pricing Authority → Pricing Authority: `computeTotal(basePrice, durationCost)`
6. Pricing Authority → Initiator: `return cost`

Preconditions:

- Domain name is non empty
- Duration is positive integer

Postconditions:

- Cost calculated deterministically
- No state changes occur (view function)

2.3.5 Protocol Summary Table

Table 2: Interaction Protocol Summary

Protocol	**Purpose**	**Participants**	**Key Messages**
Registration	Register new domain	User, Registry, Pricing	register(), calculateCost(), recordOwnership()
RecordUpdate	Modify resolution data	Owner, Resolver, Registry	updateRecord(), getOwner(), verifyAuth()
OwnershipTransfer	Transfer domain	Owner, Registry	transferOwnership(), updateOwner()
PricingQuery	Calculate costs	Any, Pricing	calculateCost(), return cost

2.4 Organizational Structure

GAIA models the organizational relationships between roles through an organizational structure that captures reporting relationships, communication patterns, and coordination mechanisms.

2.4.1 Hierarchy and Coordination

UCNS employs a *peer based organizational structure* where roles have equal authority and coordinate through defined protocols rather than hierarchical command chains. This reflects the decentralized nature of blockchain systems where no agent has superior authority over others.

Coordination Patterns:

- **Request Response:** Registry Manager queries Pricing Authority for costs
- **Consultation:** Resolution Provider verifies ownership with Registry Manager
- **Event Broadcasting:** All roles emit events for external observers
- **Atomic Transactions:** State changes occur within single blockchain transactions

2.4.2 Organizational Rules

Rule 1: Single Authority: Each role has exclusive authority over its domain of responsibility. Registry Manager alone can modify ownership records; Resolution Provider alone can update resolution data; Pricing Authority alone can calculate costs.

Rule 2: Verification Before Action: Roles must verify prerequisites through consultation before taking action. Resolution Provider must verify ownership before updates; Registry Manager must verify payment before registration.

Rule 3: Immutability: All state changes are permanent and irreversible (inherent blockchain property). This drives careful precondition checking and atomic transaction design.

Rule 4: Transparency: All actions must emit events providing visibility into role activities for external monitoring and audit.

Rule 5: Autonomy: Each role operates independently without relying on external coordinators or supervisors. Coordination occurs horizontally through protocols.

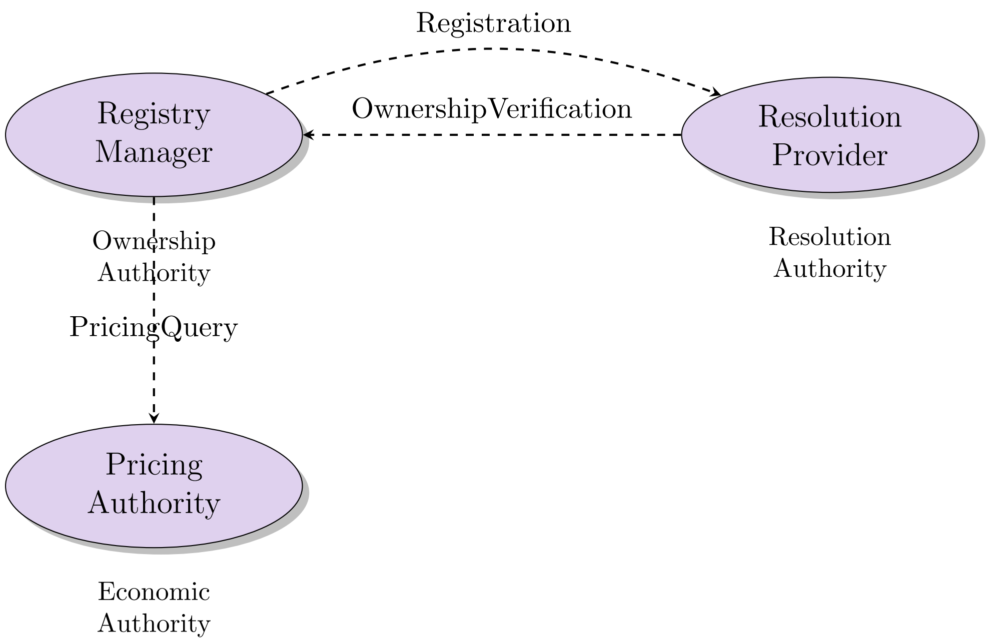

Figure 7: UCNS Organizational Structure

3 Phase 2: Architectural Design

Before introducing concrete agent types, it is important to understand how the GAIA artifacts from Phase 1 inform Phase 2. In GAIA, the *roles* and *interaction protocols* identified during requirements analysis serve as conceptual bridges between high level goals and low level implementation. Phase 2 systematically transforms these roles into *agent types*, defining each agent's responsibilities, the number of instances required, and the acquaintance relationships between agents. In the context of UCNS, the three core roles Registry Manager, Resolution Provider, and Pricing Authority map almost one to one to three singleton agent types: RegistryAgent, ResolverAgent, and PricingAgent. This mapping ensures a single source of truth for ownership, resolution, and pricing logic on the blockchain. Figure 8 provides a high level view of the UCNS agents and illustrates how they interact with external actors through the Web3 interface.

3.1 Agent Type Identification

In GAIA's architectural design phase, organizational roles are mapped to agent types instantiable templates that implement one or more roles. In UCNS, each role naturally corresponds to a single agent type to ensure authoritative and deterministic behavior. Because ownership, resolution, and pricing require a single source of truth, each agent type is instantiated exactly once as an immutable smart contract. Figure 9 formalizes the mapping from roles to agent types and their instances.

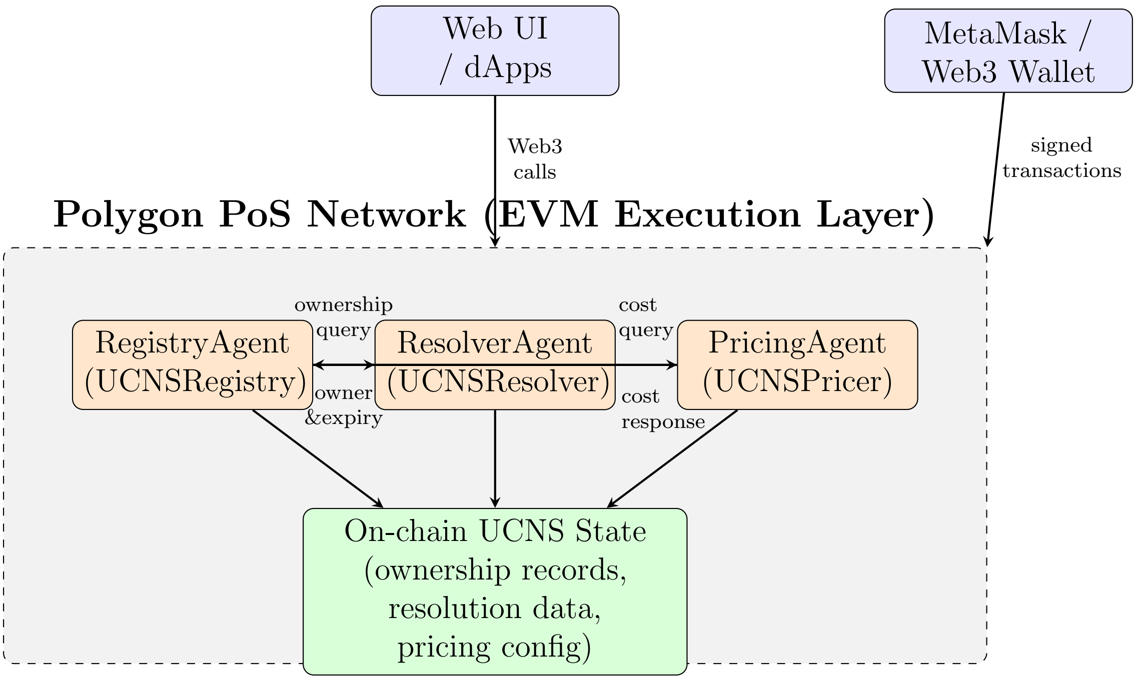

Figure 8: High-level UCNS agent architecture on Polygon PoS with properly aligned arrows and wrapped labels

3.1.1 Agent Type 1: RegistryAgent

Implemented Roles: Registry Manager

Description: Smart contract agent responsible for all domain ownership and lifecycle management. Implemented as UCNSRegistry.sol deployed on Polygon blockchain.

Instance Cardinality: Singleton (exactly one instance per deployment)

Rationale: Ownership records require a single source of truth. Multiple registry instances would create inconsistent ownership state. The singleton pattern ensures all ownership queries return authoritative results.

State Variables:

- `domains`: Mapping from domain hash to ownership record
- `domainApprovals`: Mapping for approved operators
- `operatorApprovals`: Mapping for operator authorizations
- `owner`: Contract owner address
- `pricingAgent`: Address of Pricing Authority

Public Interfaces:

- `register(string domain, uint duration) payable returns (bool)`
- `transfer(string domain, address newOwner) returns (bool)`
- `getOwner(string domain) view returns (address)`
- `isExpired(string domain) view returns (bool)`
- `approveOperator(address operator, bool approved) returns (bool)`

Events:

- `DomainRegistered(string domain, address owner, uint expiration)`
- `OwnershipTransferred(string domain, address from, address to)`
- `OperatorApproved(address owner, address operator, bool approved)`

3.1.2 Agent Type 2: ResolverAgent

Implemented Roles: Resolution Provider

Description: Smart contract agent managing domain to address resolution and metadata storage. Implemented as UCNSResolver.sol deployed on Polygon blockchain.

Instance Cardinality: Singleton (exactly one instance per deployment)

Rationale: Resolution records need consistent global accessibility. A single resolver ensures all resolution queries return uniform results and simplifies client implementation.

State Variables:

- `records`: Mapping from domain hash to resolution record structure
- `registryContract`: Address of Registry Authority for ownership verification

Public Interfaces:

- `resolve(string domain) view returns (address)`

- `updateAddress(string domain, address newAddress) returns (bool)`
- `updateTextRecord(string domain, string key, string value) returns (bool)`
- `getTextRecord(string domain, string key) view returns (string)`
- `getAllRecords(string domain) view returns (ResolverRecord)`

Events:

- `AddressUpdated(string domain, address newAddress)`
- `TextRecordUpdated(string domain, string key, string value)`

3.1.3 Agent Type 3: PricingAgent

Implemented Roles: Pricing Authority

Description: Smart contract agent computing registration costs based on domain characteristics. Implemented as PricingAgent.sol deployed on Polygon blockchain.

Instance Cardinality: Singleton (exactly one instance per deployment)

Rationale: Unified pricing logic ensures consistent cost calculation across the system. Single instance simplifies pricing policy updates and prevents pricing arbitrage.

State Variables:

- `basePrices`: Mapping from length tier to base price
- `pricePerYear`: Fixed annual cost component
- `owner`: Contract owner for pricing updates

Public Interfaces:

- `calculateCost(string domain, uint duration) view returns (uint)`
- `updateBasePriceForTier(uint tier, uint price) returns (bool)`
- `updatePricePerYear(uint price) returns (bool)`

Events:

- `PricingUpdated(uint tier, uint newPrice)`

3.2 Agent Model

The Agent Model captures the complete set of agent type instances, their role assignments, and instance relationships.

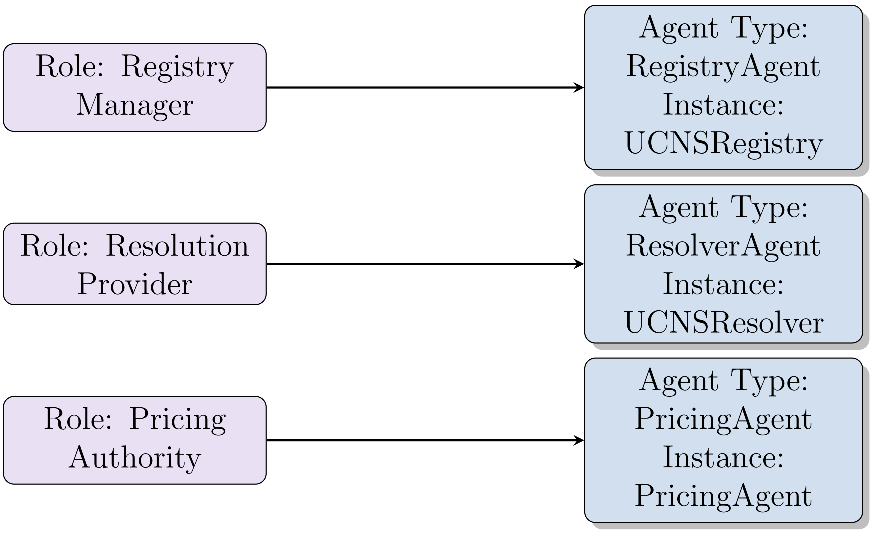

Figure 9: GAIA agent model: mapping roles to agent types and instances

3.2.1 Agent Instance Definitions

Table 3: Agent Instance Model

Agent Type	Instance Name	Roles	Cardinality
RegistryAgent	UCNSRegistry	Registry Manager	1
ResolverAgent	UCNSResolver	Resolution Provider	1
PricingAgent	PricingAgent	Pricing Authority	1

3.2.2 Agent Relationships

Dependency Graph:

- RegistryAgent *depends on* PricingAgent for cost calculations
- ResolverAgent *depends on* RegistryAgent for ownership verification
- PricingAgent *independent* of other agents (provides service only)

Communication Topology:

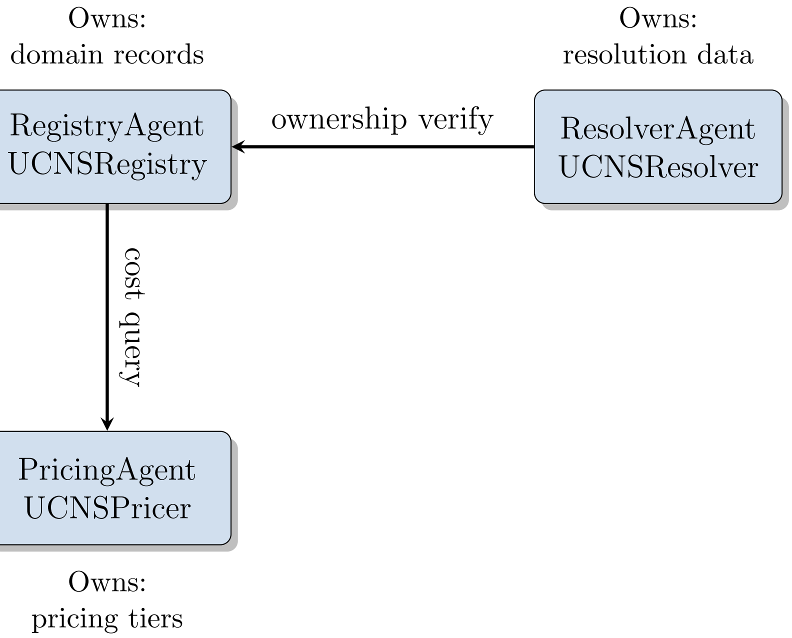

Figure 10: Agent Communication Topology

3.3 Acquaintance Model

The Acquaintance Model defines which agents are aware of one another and establishes the communication pathways in the system. In GAIA, acquaintance relationships indicate which agents must know of each other to perform their tasks. In the context of UCNS, these relationships correspond to stored smart contract addresses, since contracts interact directly via known addresses. Figure 11 illustrates these directed dependencies for example, the ResolverAgent relies on the RegistryAgent for ownership verification, while the RegistryAgent relies on the PricingAgent for cost retrieval.

3.3.1 Acquaintance Definitions

Table 4: Agent Acquaintance Model

Agent	**Acquainted With**	**Purpose**
RegistryAgent	PricingAgent	Cost calculation queries
RegistryAgent	ResolverAgent	Initial resolver assignment (optional)
ResolverAgent	RegistryAgent	Ownership verification queries

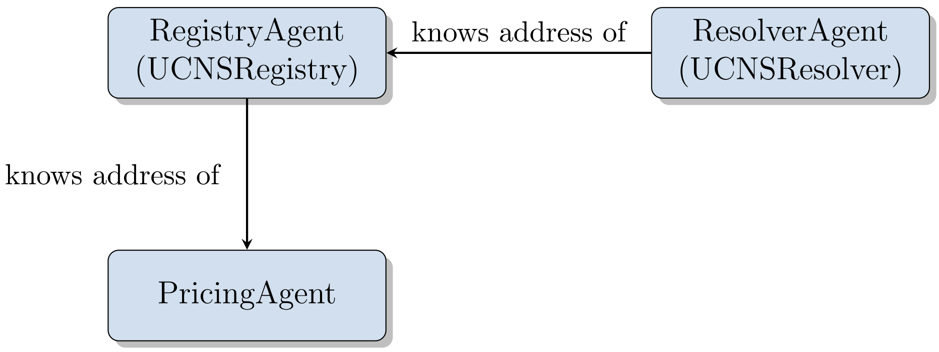

Figure 11: Agent acquaintance model: who knows whom

Implementation Mechanism:

Acquaintances are implemented through address storage:

- RegistryAgent stores `pricingAgentAddress` as immutable state variable
- ResolverAgent stores `registryAddress` as immutable state variable
- Addresses set during contract deployment and cannot be changed (immutability constraint)

Rationale for Fixed Acquaintances:

The immutable acquaintance model reflects blockchain contract immutability. While this limits runtime flexibility, it provides several benefits:

- Prevents malicious address substitution attacks
- Ensures predictable agent coordination patterns
- Simplifies security analysis and formal verification
- Aligns with smart contract best practices

3.3.2 Communication Patterns

Synchronous Calls: All inter agent communication occurs through synchronous function calls within single transactions. Example: RegistryAgent calls PricingAgent.calculateCost() and receives immediate response.

Event Broadcasting: Agents emit events for external observers (users, dApps, indexers) but do not directly communicate via events. Events are informational only.

No Asynchronous Messaging: The blockchain execution model enforces synchronous call semantics. There are no message queues or asynchronous callbacks between agents.

3.4 Service Model

Services represent the functional capabilities agents provide through well defined interfaces. The Service Model catalogs available services, their signatures, and quality of service properties.

3.4.1 RegistryAgent Services

Table 5: RegistryAgent Service Catalog

Service	Signature	Description
RegisterDomain	`register(string domain, uint duration) payable returns (bool)`	Registers new domain with payment verification
TransferOwnership	`transfer(string domain, address to) returns (bool)`	Transfers domain to new owner
GetOwner	`getOwner(string domain) view returns (address)`	Returns current owner address
CheckExpiration	`isExpired(string domain) view returns (bool)`	Checks if domain has expired
ApproveOperator	`approveOperator(address operator, bool approved) returns (bool)`	Grants management permissions
CheckAvailability	`available(string domain) view returns (bool)`	Determines registration availability

3.4.2 ResolverAgent Services

Table 6: ResolverAgent Service Catalog

Service	Signature	Description
ResolveAddress	`resolve(string domain) view returns (address)`	Returns primary address for domain
UpdateAddress	`updateAddress(string domain, address addr) returns (bool)`	Updates resolution address
GetTextRecord	`getTextRecord(string domain, string key) view returns (string)`	Retrieves specific text record

Service	Signature	Description
UpdateTextRecord	`updateText(string domain, string key, string value) returns (bool)`	Updates text record field
GetAllRecords	`getAllRecords(string domain) view returns (Record)`	Returns complete record structure

3.4.3 PricingAgent Services

Table 7: PricingAgent Service Catalog

Service	Signature	Description
CalculateCost	`calculateCost(string domain, uint duration) view returns (uint)`	Computes registration cost
UpdateBaseTier	`updateBasePriceForTier(uint tier, uint price) returns (bool)`	Modifies tier pricing
UpdateYearlyPrice	`updatePricePerYear(uint price) returns (bool)`	Adjusts annual cost component
GetBasePriceForLength	`getBasePriceForLength(uint length) view returns (uint)`	Returns tier base price

3.4.4 Service Properties

QoS Characteristics:

Table 8: Service Quality-of-Service Properties

Service	Latency	Cost	Idempotent	Deterministic
RegisterDomain	2-5s	150k gas	No	Yes
TransferOwnership	2-5s	80k gas	No	Yes
GetOwner	¡100ms	0 gas	Yes	Yes
ResolveAddress	¡100ms	0 gas	Yes	Yes
UpdateAddress	2-5s	50k gas	No	Yes
CalculateCost	¡100ms	0 gas	Yes	Yes

3.5 System Architecture

The system architecture decomposes UCNS into modular agents, each implemented as a standalone smart contract specializing in a well defined service. The architectural model closely mirrors the GAIA agent model from Phase 2, but shifts the perspective toward deployment and functional decomposition. Before discussing each agent's internal behaviors, Figure 12 presents a consolidated data model that captures the records managed by the Registry, Resolver, and Pricing agents.

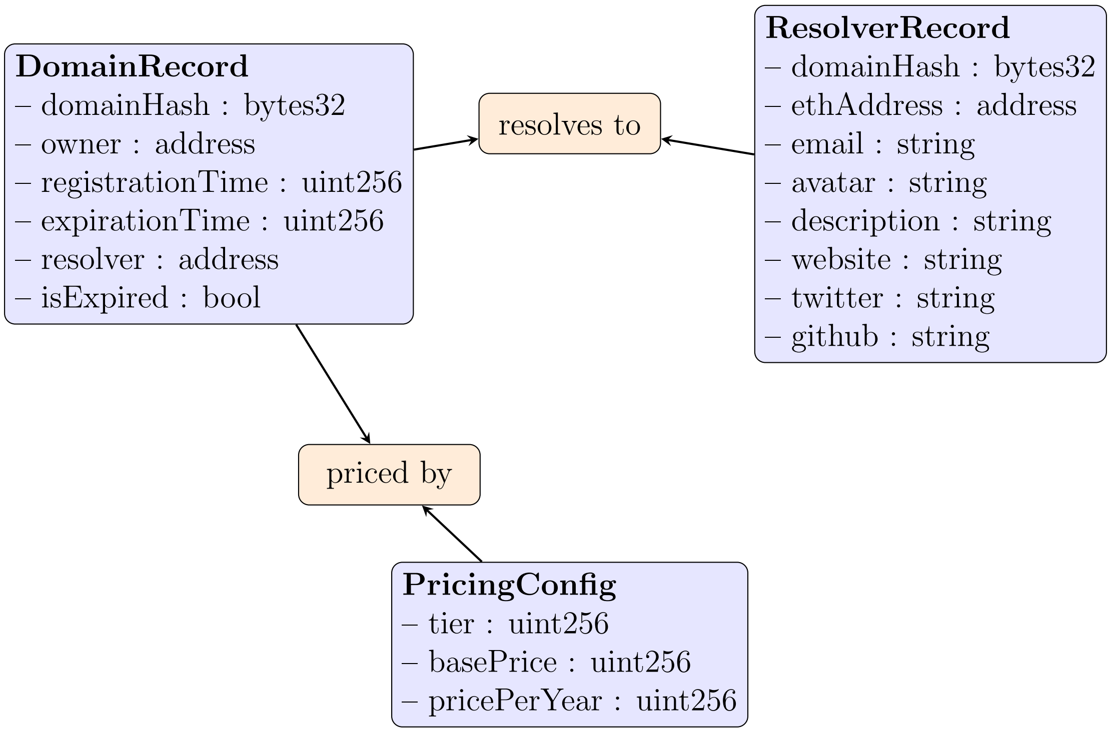

Figure 12: Conceptual data model for UCNS domain, resolution, and pricing records

4 Phase 3: Detailed Design

4.1 Agent Internal Architecture

GAIA's detailed design phase specifies the internal structure of each agent type, including decision-making logic, state management, and service implementation.

4.1.1 RegistryAgent Internal Architecture

State Representation:

```
struct DomainRecord {
    address owner;
    uint256 registrationTime;
    uint256 expirationTime;
    address resolver;
    bool isExpired;
}

mapping(bytes32 => DomainRecord) domains;
mapping(bytes32 => address) domainApprovals;
mapping(address => mapping(address => bool)) operatorApprovals;
address immutable pricingAgent;
```

```
address public owner;
string[] reservedNames;
```

Listing 1: RegistryAgent Internal State

Decision Making Components:

Component 1: Name Validator

Evaluates domain names against syntax rules and reserved name list.

Algorithm 1 RegistryAgent: Name Validation

Require: $domainName$
Ensure: Valid or revert

$length \leftarrow strlen(domainName)$
if $length < 1$ **or** $length > 64$ **then**
  revert "Invalid length"
end if
if $domainName[0] =' -'$ **or** $domainName[length - 1] =' -'$ **then**
  revert "Invalid hyphen position"
end if
for $i \leftarrow 0$ **to** $length - 1$ **do**
  $c \leftarrow domainName[i]$
  if $\neg(isLowerAlpha(c)$ **or** $isDigit(c)$ **or** $c =' -')$ **then**
    revert "Invalid character"
  end if
end for
if $domainName \in reservedNames$ **then**
  revert "Reserved name"
end if
return `valid`

Component 2: Availability Checker

Determines if domain can be registered.

Algorithm 2 RegistryAgent: Availability Check

Require: $domainName$
Ensure: Available or unavailable

$hash \leftarrow keccak256(domainName)$
if $domains[hash].owner = 0x0$ **then**
  return `available` ▷ Never registered
end if
if $block.timestamp > domains[hash].expirationTime$ **then**
  return `available` ▷ Expired
end if
return `unavailable`

Component 3: Registration Processor

Coordinates complete registration workflow.

Algorithm 3 RegistryAgent: Registration Processing

Require: $domainName$, $duration$, $payment$
Ensure: Registration success or revert

```
validateName(domainName)
hash ← keccak256(domainName)
if ¬available(domainName) then
    revert "Domain unavailable"
end if
cost ← pricingAgent.calculateCost(domainName, duration)
if payment < cost then
    revert "Insufficient payment"
end if
domains[hash].owner ← msg.sender
domains[hash].registrationTime ← block.timestamp
domains[hash].expirationTime ← block.timestamp + duration
domains[hash].isExpired ← false
emit DomainRegistered(domainName, msg.sender, expirationTime)
return success
```

Concurrency Model:

RegistryAgent relies on blockchain transaction atomicity for concurrency control. Multiple users can attempt simultaneous registration of the same domain, but blockchain consensus ensures only one transaction succeeds. Race conditions are prevented by:

- Atomic read, modify, write within single transactions
- Blockchain sequential transaction ordering
- State locking during transaction execution

Error Handling:

All invalid inputs or constraint violations result in transaction reversion with descriptive error messages. Partial state updates are impossible due to transaction atomicity.

4.1.2 ResolverAgent Internal Architecture

State Representation:

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

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

Listing 2: ResolverAgent Internal State

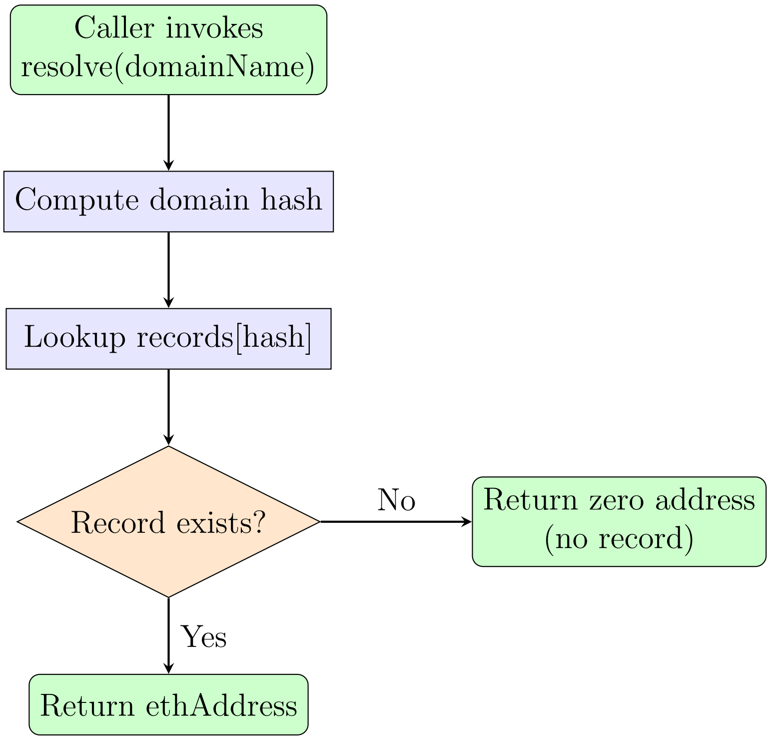

Figure 13: Flowchart of UCNS address resolution via ResolverAgent

Decision Making Components:
Component 1: Authorization Verifier
Confirms caller has permission to modify records.

Algorithm 4 ResolverAgent: Authorization Verification

Require: $domainName$, $caller$
Ensure: Authorized or revert

$hash \leftarrow keccak256(domainName)$
$owner \leftarrow registryContract.getOwner(hash)$
if $caller \neq owner$ **then**
    $isOperator \leftarrow registryContract.isApprovedOperator(owner, caller)$
    if $\neg isOperator$ **then**
        revert "Not authorized"
    end if
end if
$expired \leftarrow registryContract.isExpired(hash)$
if $expired$ **then**
    revert "Domain expired"
end if
return `authorized`

Component 2: Record Updater
Modifies resolution records after authorization.

Algorithm 5 ResolverAgent: Record Update

Require: $domainName$, $recordType$, $newValue$
Ensure: Update success or revert

1: $verifyAuthorization(domainName, msg.sender)$
2: $hash \leftarrow keccak256(domainName)$
3: $validateValue(recordType, newValue)$
4: **if** $recordType =$ "*address*" **then**
5: $records[hash].ethAddress \leftarrow newValue$
6: **else if** $recordType =$ "*email*" **then**
7: $records[hash].email \leftarrow newValue$
8: **else if** $recordType =$ "*avatar*" **then**
9: $records[hash].avatar \leftarrow newValue$
10: **end if**
11: **emit** RecordUpdated($domainName$, $recordType$, $newValue$)
12: **return** `success`

Caching Strategy:

ResolverAgent maintains no caches. All queries read current state directly from blockchain storage. This ensures:

- Resolution always returns latest data
- No cache invalidation complexity
- Strong consistency guarantees
- Simplified implementation

4.1.3 PricingAgent Internal Architecture

State Representation:

```
mapping(uint256 => uint256) public basePrices;
uint256 public pricePerYear;
address public owner;

// Pricing tiers initialized in constructor:
// tier 1 (length 1): 1.0 MATIC
// tier 2 (length 2-3): 0.5 MATIC
// tier 3 (length 4-5): 0.1 MATIC
// tier 4 (length 6+): 0.05 MATIC
```

Listing 3: PricingAgent Internal State

Decision Making Components:

Component 1: Cost Calculator

Pure function computing total cost.

Algorithm 6 PricingAgent: Cost Calculation

Require: $domainName$, $durationYears$
Ensure: $totalCost$
$length \leftarrow strlen(domainName)$
$tier \leftarrow determineTier(length)$
$basePrice \leftarrow basePrices[tier]$
$durationCost \leftarrow durationYears \times pricePerYear$
$totalCost \leftarrow basePrice + durationCost$
return $totalCost$

Component 2: Tier Determiner
Maps domain length to pricing tier.

Algorithm 7 PricingAgent: Tier Determination

Require: $length$
Ensure: $tier$

```
if length = 1 then
    return 1
else if length ≤ 3 then
    return 2
else if length ≤ 5 then
    return 3
else
    return 4
end if
```

Statelessness:
Cost calculation is a pure function with no side effects. This enables:

- Zero gas preview queries via view functions
- Deterministic cost calculation
- Easy formal verification
- No transaction ordering dependencies

4.2 Service Implementation Specifications

4.2.1 RegisterDomain Service

Service Name: RegisterDomain
Agent: RegistryAgent
Inputs:

- `domainName: string` - Desired domain name
- `duration: uint256` - Registration period in seconds
- `msg.value: uint256` - Payment amount in wei

Outputs:

- `success: bool` - Registration success indicator
- `DomainRegistered event` - Emitted on success

Preconditions:

- Domain name valid per naming rules
- Domain available (unregistered or expired)
- Payment sufficient for calculated cost
- Duration within 1-10 year range

Postconditions:

- Domain ownership recorded with caller as owner
- Expiration timestamp set to registration time + duration
- Payment transferred to contract
- Registration event emitted

Exceptions:

- InvalidNameFormat: Revert if name violates syntax rules
- DomainUnavailable: Revert if domain already registered
- InsufficientPayment: Revert if payment below required cost

Performance:

- Gas Cost: Approximately 150,000 gas
- Latency: 2 - 5 seconds (blockchain confirmation time)
- Deterministic: Yes

4.2.2 ResolveAddress Service

Service Name: ResolveAddress
Agent: ResolverAgent
Inputs:

- `domainName: string` - Domain to resolve

Outputs:

- `address: address` - Resolved Ethereum address

Preconditions:

- Domain name non-empty

Postconditions:

- Returns address associated with domain
- Returns zero address if no record exists
- No state modifications

Exceptions:

- None (returns zero address for non existent domains)

Performance:

- Gas Cost: 0 (view function)
- Latency: ¡100ms
- Deterministic: Yes

4.2.3 CalculateCost Service

Service Name: CalculateCost
Agent: PricingAgent
Inputs:

- `domainName: string` - Domain for cost calculation
- `duration: uint256` - Registration duration in years

Outputs:

- `cost: uint256` - Total registration cost in wei

Preconditions:

- Domain name non-empty
- Duration positive

Postconditions:

- Returns deterministically computed cost
- No state modifications

Exceptions:

- None (pure function)

Performance:

- Gas Cost: 0 (view function)
- Latency: $<$ 10 ms
- Deterministic: Yes

4.3 Coordination Mechanisms

4.3.1 Transaction Based Coordination

All inter agent coordination occurs within atomic blockchain transactions. This provides:

Atomicity: Either all agent interactions succeed or none do. Partial coordination states are impossible.

Consistency: Agents always observe consistent global state. No dirty reads or phantom updates.

Isolation: Concurrent transactions execute in total order established by blockchain consensus.

Durability: Coordination results are permanently recorded on blockchain.

4.3.2 Protocol Execution Model

Synchronous Call Chain:

Registration protocol execution follows deterministic call sequence:

1. User submits transaction to RegistryAgent.register()
2. RegistryAgent executes validation internally
3. RegistryAgent calls PricingAgent.calculateCost() synchronously
4. PricingAgent returns cost immediately
5. RegistryAgent completes registration if payment sufficient
6. Transaction commits atomically

No Asynchronous Messaging:

Unlike traditional multi agent systems with message queues, UCNS agents communicate only through synchronous function calls. This reflects EVM execution model constraints.

Event Based Notification:

Events provide one way notification to external observers but do not trigger agent actions. Events are informational only.

4.4 Error Recovery and Fault Tolerance

4.4.1 Transaction Reversion

All errors trigger transaction reversion, rolling back state changes atomically. This provides:

- Guaranteed consistency (no partial updates)
- Simplified error handling (no cleanup code needed)
- Clear failure semantics (transaction either succeeds completely or fails completely)

4.4.2 Failure Modes

Invalid Input Failures:

Malformed inputs detected during validation phase before state changes occur. Transaction reverts with descriptive error message.

Authorization Failures:

Unauthorized operations detected through ownership verification. Transaction reverts preventing unauthorized state access.

Economic Failures:

Insufficient payment detected before ownership recording. Transaction reverts returning funds to sender.

Network Failures:

Blockchain node failures handled by network redundancy. Clients retry transactions on failure. Blockchain consensus ensures exactly once execution.

4.4.3 Invariant Maintenance

Critical system invariants are maintained through careful precondition checking:

Invariant 1: Every registered domain has exactly one owner.
Enforcement: Ownership field updated only through verified transfers or registrations.
Invariant 2: Only authorized parties modify resolution records.
Enforcement: All updates verify ownership before proceeding.
Invariant 3: Pricing calculations are deterministic and consistent.
Enforcement: Pure functions with no external dependencies.
Invariant 4: Reserved names cannot be registered.
Enforcement: Validation checks reserved name list before registration.

5 Design Validation

5.1 Goal Satisfaction Analysis

5.1.1 Goal Coverage

Table 9: Goal Satisfaction Matrix

Goal	Satisfied By	Mechanism
G1: Autonomous Registration	RegistryAgent	RegisterDomain service with cryptographic ownership
G2: Reliable Resolution	ResolverAgent	ResolveAddress service with authorization
G3: Dynamic Economics	PricingAgent	CalculateCost service with configurable tiers
G4: Ownership Management	RegistryAgent	TransferOwnership and ApproveOperator services
G5: Transparency	All Agents	Event emission for all state changes
G6: Gas Efficiency	All Agents	Optimized storage patterns, view functions
G7: Extensibility	Architecture	Clear agent boundaries, protocol based interaction

5.2 Protocol Verification

5.2.1 Protocol Completeness

Each identified interaction protocol has complete specification including:

- Participants clearly identified
- Message sequences fully specified
- Preconditions and postconditions defined
- Failure modes documented
- Sequence diagrams provided

5.2.2 Protocol Consistency

Protocol specifications are consistent with:

- Agent role responsibilities
- Service interface definitions
- Acquaintance model constraints
- Blockchain execution model

5.3 Architectural Quality Assessment

5.3.1 Modularity

Metric: Degree of independence between agents
Evaluation: High modularity achieved through:

- Clear role boundaries without overlap
- Minimal inter agent dependencies
- Service based interaction model
- Independent state management per agent

5.3.2 Cohesion

Metric: Degree of relatedness of agent responsibilities
Evaluation: High cohesion achieved through:

- Single purpose agent roles
- Focused service offerings per agent
- Logical grouping of related functionality

5.3.3 Coupling

Metric: Degree of interdependence between agents
Evaluation: Low coupling achieved through:

- Protocol based interaction only
- No shared state between agents
- Immutable acquaintance relationships
- Service abstraction hiding implementation details

5.4 Scalability Analysis

5.4.1 Domain Registration Scalability

Operations Per Second: Limited by blockchain throughput (65k TPS theoretical on Polygon)

Storage Scalability: Linear growth with number of domains. Each domain occupies fixed storage slots.

Query Scalability: View function queries scale independently of blockchain load. RPC providers can cache results.

5.4.2 Agent Communication Scalability

Intra Transaction Calls: No scalability concerns. Synchronous calls complete within single transaction.

External Queries: Unlimited scalability for view functions. No blockchain state modification required.

5.5 Security Analysis

5.5.1 Threat Model

Threat 1: Unauthorized Ownership Changes

Mitigation: Cryptographic signature verification ensures only rightful owners can transfer domains.

Threat 2: Resolution Record Tampering

Mitigation: ResolverAgent verifies ownership with RegistryAgent before accepting updates.

Threat 3: Pricing Manipulation

Mitigation: PricingAgent calculations are deterministic and publicly verifiable. Only contract owner can modify pricing tiers.

Threat 4: Reentrancy Attacks

Mitigation: Checks Effects Interactions pattern followed. External calls occur after state updates.

Threat 5: Front Running

Mitigation: Accepted risk in public blockchain. Users can increase gas prices for priority. No critical vulnerability from front running domain registrations.

5.5.2 Access Control Verification

Table 10: Access Control Matrix

Operation	Agent	Allowed Callers	Verification
Register Domain	Registry	Any (with payment)	Payment verification
Transfer Domain	Registry	Owner only	Ownership check
Update Records	Resolver	Owner/Operator	Ownership verification via Registry
Update Pricing	Pricing	Contract owner	Owner modifier
Query Services	Any	Public	No restrictions

6 Implementation Mapping

6.1 GAIA to Solidity Mapping

6.1.1 Agent to Smart Contract

Each GAIA agent type maps to one Solidity smart contract:

Table 11: Agent Contract Mapping

GAIA Agent Type	Solidity Contract	File
RegistryAgent	UCNSRegistry	UCNSRegistry.sol
ResolverAgent	UCNSResolver	UCNSResolver.sol
PricingAgent	PricingAgent	PricingAgent.sol

6.1.2 Role to Contract Functions

Agent roles map to collections of contract functions:

- Role *responsibilities* → Public functions
- Role *activities* → Internal/private functions
- Role *permissions* → Access control modifiers
- Role *protocols* → Function call sequences

6.1.3 Service to Function Interface

Each service specification becomes a public function:

```
// GAIA Service: RegisterDomain
// Maps to Solidity function:

function register(
    string memory domainName,
    uint256 duration
) public payable returns (bool) {
    // Service implementation
}
```

Listing 4: Service to Function Mapping Example

6.1.4 Protocol to Transaction Flow

Interaction protocols map to multi contract transaction flows:

- Protocol *messages* → Function calls
- Protocol *participants* → Contract addresses
- Protocol *sequence* → Call order within transaction
- Protocol *failures* → Revert conditions

6.2 Deployment Architecture

6.2.1 Contract Deployment Order

1. Deploy PricingAgent (no dependencies)
2. Deploy RegistryAgent with PricingAgent address
3. Deploy ResolverAgent with RegistryAgent address
4. Verify all contracts on PolygonScan

6.2.2 Deployed Addresses

Table 12: Production Deployment Addresses (Polygon Mainnet)

Contract	Address
PricingAgent	0x50F50124Ee00002379142cff115b0550240898B3
UCNSRegistry	0xc9eD4B38E29C64d37cb83819D5eEcFD34EFdce0C
UCNSResolver	0x2De897131ee8AC0538585887989E2314034F0b71

Verification Status: All contracts verified and publicly viewable on PolygonScan.

6.3 Integration with Web Interface

6.3.1 Agent Access from Web Layer

Web interface (index.php) accesses agents through Web3.js:

```
// JavaScript code in frontend

const registryABI = [...]; // Contract ABI
const registryAddress = "0xc9eD4B38...";

const registryContract = new ethers.Contract(
    registryAddress,
    registryABI,
    signer
);

// Call RegisterDomain service
await registryContract.register(
    domainName,
    duration,
    { value: cost }
);
```

Listing 5: Web3 Agent Integration

6.3.2 Event Monitoring

Frontend monitors agent events for state updates:

```
// Listen for DomainRegistered events
registryContract.on("DomainRegistered",
    (domain, owner, expiration) => {
        updateUI(domain, owner, expiration);
    }
);
```

Listing 6: Event Monitoring

7 Conclusion

7.1 Summary of GAIA Application

This document has systematically applied the GAIA methodology to design the University of Calgary Name Service as a multi agent system. The three phase GAIA process requirements analysis, architectural design, and detailed design provided structured frameworks for transforming high level goals into implementable agent specifications.

Requirements Analysis Phase identified seven system goals and decomposed them into three organizational roles (Registry Manager, Resolution Provider, Pricing Authority) with clearly defined responsibilities, permissions, and interaction protocols. This phase established what the system must accomplish independently of how it is implemented.

Architectural Design Phase defined three agent types (RegistryAgent, ResolverAgent, PricingAgent) that implement the identified roles. The phase specified agent relationships through acquaintance models and cataloged services each agent provides. This phase established the system's organizational structure and communication topology.

Detailed Design Phase specified internal agent architecture including decision making algorithms, state management strategies, and service implementations. This phase provided blueprints for implementing each agent as Solidity smart contracts on Polygon blockchain.

7.2 Benefits of Agent Oriented Design

The GAIA methodology provided several concrete benefits for UCNS development:

Systematic Decomposition: GAIA's role based analysis provided principled criteria for decomposing naming service functionality into cohesive, loosely coupled agents.

Clear Separation of Concerns: Each agent has well defined responsibilities without overlap or ambiguity. This modularity simplifies both implementation and maintenance.

Explicit Interaction Modeling: Protocol specifications make agent coordination patterns explicit and verifiable. This reduces integration errors and facilitates testing.

Traceability: Clear mapping from goals through roles to agent implementations enables verification that design satisfies requirements.

Reusability: Service oriented agent design enables components to be reused in related systems or upgraded independently.

Formal Reasoning: GAIA's structured models support formal verification of system properties including safety invariants and liveness properties.

7.3 Blockchain Specific Adaptations

Applying GAIA to blockchain systems required several adaptations:

Synchronous Communication Model: Traditional multi agent systems often assume asynchronous messaging. Blockchain smart contracts use synchronous function calls, affecting protocol design.

Immutability Constraints: Deployed contracts cannot be modified. This influenced acquaintance model design (fixed addresses) and emphasizes careful design validation before deployment.

Economic Considerations: Gas costs affect architectural decisions. GAIA does not explicitly model computational costs, requiring additional analysis for blockchain contexts.

Trustless Coordination: Blockchain enables coordination without trust assumptions. GAIA's organizational rules map naturally to cryptographic enforcement mechanisms.

7.4 Lessons Learned

Agent Granularity: Three agents provide appropriate granularity for UCNS. Finer grained decomposition would increase coordination overhead without corresponding modularity benefits.

Protocol Completeness: Complete protocol specifications are essential. Incomplete specifications lead to integration issues during implementation.

Early Validation: Validating design against requirements before implementation saves significant development effort. GAIA's structured approach facilitates early validation.

Role Evolution: Initial role definitions may require refinement as understanding deepens. GAIA's iterative refinement of role models accommodates this evolution.

7.5 Future Extensions

The agent-based design supports several future enhancements:

Marketplace Agent: A new agent type could be added to facilitate domain trading without modifying existing agents.

Governance Agent: Decentralized governance could be implemented through a dedicated agent coordinating with existing agents via defined protocols.

Storage Agent: Integration with IPFS for metadata storage could be encapsulated in a separate agent.

Bridge Agents: Cross chain resolution could be supported by bridge agents that coordinate across blockchain networks.

These extensions would leverage GAIA's modular architecture, adding new agent types and protocols without disrupting existing components.

7.6 Final Remarks

The GAIA methodology successfully guided UCNS design from initial goals through detailed implementation specifications. The resulting multi agent architecture exhibits strong modularity, clear responsibilities, and well defined coordination patterns. The systematic design process reduced ambiguity, facilitated validation, and produced comprehensive documentation supporting both implementation and future evolution.

The project demonstrates that agent oriented software engineering methodologies, originally developed for distributed AI systems, adapt effectively to blockchain contexts when appropriately tailored to smart contract constraints. The alignment between agent autonomy principles and blockchain's trustless execution model creates natural synergies that simplify design and enhance system properties.

UCNS serves as both a functional decentralized naming service and an exemplar of GAIA methodology application in blockchain system design, contributing to the growing body of knowledge on agent-based approaches to decentralized application development.

Appendix A: GAIA Modeling Constructs

Role Schema Template

```
Role Schema: <RoleName>
Description: <Brief description>
Protocols and Activities: <List of protocols and activities>
Permissions: <Read/Write/Execute permissions>
Responsibilities: <List of responsibilities>
Liveness: <Liveness properties>
Safety: <Safety properties>
```

Protocol Schema Template

```
Protocol: <ProtocolName>
Purpose: <Description>
Initiator: <Role>
Responder: <Role>
Inputs: <List>
Outputs: <List>
Processing: <Message sequence>
```

Agent Schema Template

```
Agent: <AgentName>
Roles: <List of roles>
Services: <List of provided services>
Acquaintances: <List of known agents>
```

Appendix B: Complete Service Catalog

Table 13: Complete UCNS Service Catalog

Service	Agent	Description
RegisterDomain	Registry	Registers new domain with ownership
TransferOwnership	Registry	Transfers domain to new owner
GetOwner	Registry	Returns current domain owner
CheckExpiration	Registry	Verifies domain expiration status
ApproveOperator	Registry	Grants operator permissions
CheckAvailability	Registry	Determines if domain is available
ResolveAddress	Resolver	Returns primary address for domain
UpdateAddress	Resolver	Updates domain resolution address
GetTextRecord	Resolver	Retrieves specific metadata field
UpdateTextRecord	Resolver	Updates text record value
GetAllRecords	Resolver	Returns complete resolution record
CalculateCost	Pricing	Computes registration cost
UpdateBaseTier	Pricing	Modifies pricing tier
UpdateYearlyPrice	Pricing	Adjusts annual cost component
GetBasePriceForLength	Pricing	Returns tier specific base price

Appendix C: Interaction Protocol Specifications

Complete protocol specifications with message sequences, preconditions, postconditions, and failure modes for all four protocols (Registration, RecordUpdate, OwnershipTransfer, PricingQuery) are detailed in Section 2.3.

Appendix D: References

1. Wooldridge, M., Jennings, N. R., & Kinny, D. (2000). The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and multi-agent systems*, 3(3), 285-312.

2. Zambonelli, F., Jennings, N. R., & Wooldridge, M. (2003). Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(3), 317-370.

3. Wooldridge, M. (2009). *An introduction to multiagent systems.* John Wiley & Sons.

4. Jennings, N. R. (2001). An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4), 35-41.

5. Buterin, V. (2014). *A next-generation smart contract and decentralized application platform.* Ethereum White Paper.

6. Antonopoulos, A. M., & Wood, G. (2018). *Mastering Ethereum: Building smart contracts and DApps.* O'Reilly Media.

7. Far, B. H. (2024). *SENG 696 Course Materials - Agent-Based Software Engineering.* University of Calgary.
8. OpenZeppelin. (2023). *OpenZeppelin Contracts Documentation.* Retrieved from https://docs.openzeppelin.com/
9. Polygon Technology. (2021). *Polygon PoS Chain Documentation.* Retrieved from https://docs.polygon.technology/
10. Johnson, N. F. (2009). *Simply complexity: A clear guide to complexity theory.* Oneworld Publications.