Implementation Roadmap: Application Service Framework

Version: 4.0 (Final)
Date: July 29, 2025

1. Introduction

This document provides the definitive, implementation-focused roadmap for the Application Service framework. It moves beyond high-level architecture to specify the granular, technical details required to build a fully reactive, functional, and behavior-driven system. This final version consolidates all previous architectural decisions, emphasizing a pure Semantic Web foundation, deterministic structural embeddings via Formal Concept Analysis (FCA), and a sophisticated, message-driven activation layer.

The appendices that follow provide an exhaustive exploration of the core models, the conversational API protocol, a complete federated use case, and the resulting business intelligence capabilities, serving as a complete blueprint for implementation.

Appendix A: The Unified Model Architecture

The framework's power lies in its layered, yet unified, model architecture. While conceptually distinct, the Reference, Graph, Activation, FCA, and Dimensional models are all implemented on a single underlying Semantic Web store (Apache Jena), with the Reference Model's reified statements acting as the unifying data structure.

A.1. The Reference Model (The Grammar of Identity)

This is the foundational layer, transforming raw data into formal, identifiable concepts.

- **Entities & Implementation**:
    - **ID**: Represents the canonical, context-free "idea" of a resource. It is immutable.
        - **Implementation**: A Java record ID(long primeID, String urn, String did).
    - **IDOccurrence**: Represents an ID appearing in a specific role within a context (a statement).
        - **Implementation**: A Java record IDOccurrence(ID occurringId, URI contextStatementUri). The contextStatementUri points to the reified rdf:Statement that represents the context.
- Unified Property Graph Representation (on Jena):
  We use RDF reification to create a property graph structure. A single resource (e.g., "Alice") is a URI (urn:appservice:id:937).

  - **Nodes**: A resource is created with rdf:type. <urn:appservice:id:937> rdf:type :Resource .
  - **Properties**: Standard triples. <urn:appservice:id:937> :hasName "Alice" .
  - **Edges/Statements**: A statement (Alice, worksFor, Google) is reified, and this reified statement becomes the context for its participants.
    ```
    @prefix : <urn:appservice:ont#> .
    @prefix id: <urn:appservice:id#> .

    :stmt_456 a rdf:Statement, :IDOccurrence ; # The statement is an
    IDOccurrence
            rdf:subject   id:937 ;
            rdf:predicate :worksFor ;
            rdf:object    id:101 .
    ```

  This approach allows us to attach properties to edges (statements) and query the graph based on the roles entities play, unifying all model layers.
- **Statement Types & Messages**:
  - **Schema Statements**: Define the rules. (Context, Role, Dataflow). They describe the *types* of interactions possible.
  - **Data Statements**: Represent concrete events. (Interaction, Actor, Transform). They describe *instances* of interactions.
- **Inference & Traversal**:
  - **Capability**: Find all contexts a specific user participated in.
  - Implementation: A SPARQL query that selects all rdf:Statements where the rdf:subject or rdf:object is the user's ID.
    ```
    SELECT ?stmt WHERE { ?stmt a rdf:Statement; ?p
    <urn:appservice:id:user_Alice> . }
    ```

A.2. The FCA Model (The Logic of Structure)

This model provides deterministic, explainable embeddings and type inference.

- **FCA Contexts & Prime ID Products**:
  - **Model**: We define three FCA context relations from a single statement: (predicate, subject, object), (subject, predicate, object), and (object, subject, predicate).
  - **Implementation**: The Index Service can construct these contexts on demand. For a given relation (e.g., the predicate :worksFor), the context objects are the subjects, and the attributes are the objects.
  - Embedding: The Contextual Prime Product Embedding (CPPE) for an entity within a context is the product of the primeIDs of the entities it's related to.

CPPE(Google, worksFor) = primeID(Alice) * primeID(Bob) * ...

- **Inference & Traversal**:
    - **Similarity**: Similarity(A, B) = GCD(CPPE(A), CPPE(B)). The Greatest Common Divisor reveals the product of shared relations, providing a quantitative similarity score.
    - **Role Inference**: Complex roles like "uncle" can be inferred by composing CPPEs. Find "Person A" who is a :brotherOf "Person B", and "Person B" who is a :fatherOf "Person C". The system can infer a new triple: (A, :uncleOf, ChildOfB). This is a deterministic graph traversal and arithmetic operation, far more efficient and explainable than vector-based similarity.

A.3. The Graph Model (The Semantics of Sets)

This model elevates the Reference Model by reifying statements into higher-order Kinds.

- **Entities & Implementation**:
    - **Kind**: A set of IDOccurrences that share common structural properties. A SubjectKind like :Customer is formed by grouping all Subjects that interact with a similar set of (Predicate, Object) pairs.
    - **Implementation**: A Kind is a rdfs:Class that is a rdfs:subClassOf its base type (e.g., <:Customer> rdfs:subClassOf <:SubjectKind>).
- **Inference & Traversal (Functional Interfaces)**:
    - **Capability**: "Given the :Customer type, what types of actions can they perform?"
    - **Implementation (Function<URI, Set<URI>>)**: A SPARQL query that finds all PredicateKinds connected to the given SubjectKind in schema statements.

A.4. The Activation Model (The Pragmatics of Behavior)

This model enables the execution of business processes using DCI, DDD, and a Dynamic Object Model.

- **Entities & Implementation**:
    - **DOM**: An Actor's data is a flexible Instance (record Instance(ID id, Class clazz, Map<URI, Instance> attributes)).
    - **DCI**: A Context (record Context(URI id, List<Role> roles)) defines a use case. An Interaction is an instance of a Context.
    - **Actor-Role**: An Actor is an Instance playing a Role. Its state is defined by its available Transforms (record ActorState(Transform previous, Transform current, List<Transform> next)).
- **Inference & Traversal**:
    - **Capability**: "Given a desired outcome (e.g., ProductShipped), what sequence

    of Transforms is required?"
  - **Implementation**: A backward-chaining graph traversal (SPARQL query) starting from the desired final state and traversing the :precededBy relationships defined in the Dataflow schemas to construct a valid plan.

A.5. The Dimensional Model (The Analytics of Measurement)

This model provides powerful, OLAP-style analytics directly on the live graph.

- **Entities & Implementation**:
  - **Dimensional Context Statement**: We encode dimensional data using RDF reification to create explicit, queryable paths. A single sale event is encoded as a series of nested statements.
    - stmt1: The core fact (order_456 hasValue 118.00).
    - stmt2: Slices stmt1 by the Time dimension.
    - stmt3: Slices stmt2 by the Product dimension.
    - stmt4: Slices stmt3 by the Region dimension.
- **Inference & Traversal (Functional Retrieval)**:
  - **Capability**: "Show me all sales for the Pro-Lite shoe in Boston."
  - **Implementation**: A SPARQL SELECT query that finds all paths matching the nested statement pattern, filtering at each dimensional level. This provides a functional, real-time BI capability directly on the operational graph.

Appendix B: The Semantic Engine & COST/HAL Protocol

The Activation layer protocol is a conversational state dataflow, COST (COnversational State Transfer), implemented with HAL. It uses placeholders to guide the interaction, making the client dynamic and the server's logic flexible.

B.1. Encoding Behavior: Dataflow and Transforms

A Role's behavior (Dataflow) is defined as a sequence of declarative Transform definitions stored in the Jena graph.

- **TransformDef Record**:
```
record TransformDef(
   String name,
   OperationType operation, // e.g., ASSIGN, TRANSFER, COMPUTE,
INVOKE_TOOL
   Map<String, FieldDef> inputs, // Named inputs for the operation
   Map<String, FieldDef> outputs // Named outputs produced
) {}
```

```
record FieldDef(String role, String field) {}
```

- Example: ProductBuy Context Dataflow
  The business logic aBuyer.accountBalance -= anAmount.amount is encoded as a TransformDef:
  ```
  {
    "name": "DebitBuyerAccount",
    "operation": "COMPUTE",
    "inputs": {
      "currentBalance": { "role": "Buyer", "field": "accountBalance" },
      "debitAmount": { "role": "Amount", "field": "amount" }
    },
    "outputs": {
      "newBalance": { "role": "Buyer", "field": "accountBalance" }
    }
  }
  ```
  The Activation Service reads this definition, populates it with actual Actor IDs, and dispatches it as an executable Transform message.

B.2. The COST/HAL Conversation with Placeholders

The power of COST lies in embedding placeholders within the HAL _links, turning the API into a guided conversation.

1. **Server Initiates**: The server starts a ProductBuy Interaction. It sends the initial HAL state. The _links.next array contains an action template.
   ```
   // Part of a HAL response for an Interaction
   "_links": {
     "next": [
       {
         "href": "/interactions/123/transform",
         "method": "POST",
         "name": "SelectProductForPurchase",
         "title": "Select a Product to Buy",
         "schema": { // The placeholder definition
           "type": "object",
           "properties": {
             "selectedProduct": {
               "type": "string",
               "description": "The DID of the product to purchase.",
   ```

```
            "_links": {
              "possibleValues": { "href": "/interactions/123/roles/Product/possibleActors"
}
            }
          }
        }
      }
    }
  ]
}
```

2. **Client Discovers Options (Client-Side Population)**:
   - The client UI sees the SelectProductForPurchase action and its schema.
   - It sees that the selectedProduct placeholder has a possibleValues link.
   - It performs a GET on /interactions/123/roles/Product/possibleActors.
   - The server responds with a list of available products (potential Actors for the Product Role). The UI renders this as a dropdown.
3. **Client Responds with Data**:
   - The user selects a product (e.g., did:ion:product_456).
   - The client constructs the Transform message body according to the schema: { "transformName": "SelectProductForPurchase", "payload": { "selectedProduct": "did:ion:product_456" } }.
   - It POSTs this body to the href.
4. **Server Infers and Prompts for Confirmation (Server-Side Population)**:
   - The server receives the selected product. Its internal logic determines that for this product, "FedEx" is the only valid shipping partner.
   - Instead of asking the client to choose, it pre-populates the next step and asks for confirmation.
   - It responds with a new HAL state where the next action is ConfirmShippingPartner, and the response body already contains the populated data:
     ```
     {
       "shippingPartner": {
         "name": "FedEx",
         "did": "did:ion:shipper_fedex"
       }
     }
     ```
5. **Client Confirms**: The UI displays "Shipping with: FedEx" and a "Confirm" button.

Clicking it sends the ConfirmShippingPartner Transform message, and the conversation continues. This conversational, placeholder-driven approach makes the interaction incredibly flexible and decoupled.

Appendix C: End-to-End Integration Use Case: A Federated Supply Chain

This appendix depicts a complete, multi-organization use case, demonstrating how three independent entities can form a seamless, automated, and intelligent supply chain.

C.1. The Participants & Their Systems

- **Manufacturer**: SportProducts Manufacturing Inc. (SPM)
  - **Systems**: SCM for production, ERP for orders.
  - **AppService Instance**: spm.appservice.com
- **Consumer**: Sport and Fitness Stores (SFS)
  - **Systems**: Legacy ERP for inventory, modern CRM for sales.
  - **AppService Instance**: sfs.appservice.com
- **Provider**: Sports Goods Raw Materials LLC (SGRM)
  - **Systems**: Simple order management database.
  - **AppService Instance**: sgrm.appservice.com

C.2. The Use Case: From Low Stock to Federated BI

Step 1: Low Inventory Trigger at the Retailer (SFS)

- **Services Layout & Roles**:
  - SFS.Datasource: Continuously ingests inventory levels from SFS's ERP via its JCA adapter.
  - SFS.Alignment: Aligns the raw stock number into a canonical Measure. It has previously aligned the "Pro-Lite Running Shoe" from the ERP with SPM's official product definition, creating an owl:sameAs link between did:sfs:product_789 and did:spm:product_ProLite.
  - SFS.Activation: An internal Context named MonitorInventory is constantly running.
- **Messages & Dataflow**:
  1. SFS.Datasource produces a raw statement: ("store_boston", "stock_PLRS", "49").
  2. SFS.Aggregation/Alignment processes this into a Graph Model statement.
  3. The SFS.Activation engine's MonitorInventory Interaction evaluates a rule. The stock level is below the threshold, so it starts a new ReplenishStock Interaction.

Step 2: Automated Order Placement via MCP (SFS -> SPM)

- **Services Layout & Roles**:
  - SFS.Activation: The ReplenishStock Interaction needs to place an order. It knows the canonical product is from SPM. It now acts as an **MCP Client**.
  - SPM.Activation: Exposes its capabilities as an **MCP Server**.
- **Messages & Dataflow**:
  1. The SFS AppService authenticates to the SPM AppService using **DID-Auth**.
  2. SFS queries SPM's MCP endpoint for available Tools related to did:spm:product_ProLite.
  3. SPM's MCP server responds, offering a PurchaseOrder Tool.
  4. SFS sends a standardized ToolCall message via MCP to SPM's endpoint, containing the required quantity.
  5. SPM.Activation receives this, starts a FulfillOrder Interaction, and begins a COST/HAL conversation back with the SFS agent to confirm pricing and delivery dates.

Step 3: Manufacturer Checks Raw Materials (SPM)

- **Services Layout & Roles**:
  - SPM.Activation: The FulfillOrder Interaction's Dataflow includes a Transform to check internal stock for raw materials (did:sgrm:material_eva_foam).
  - SPM.Datasource: Provides access to SPM's SCM system data.
- **Messages & Dataflow**:
  1. It queries its own Graph Model and finds the stock of "EVA Foam" is insufficient.
  2. This triggers a new internal Interaction: ProcureMaterials. This Interaction identifies the supplier for did:sgrm:material_eva_foam as SGRM. SPM's AppService now prepares to act as an MCP Client.

Step 4: Manufacturer Orders Raw Materials via MCP (SPM -> SGRM)

- The process from Step 2 repeats, with SPM acting as the MCP Client and SGRM as the MCP Server.

C.3. Business Intelligence Leverage

- **Internal BI**:
  - **SFS**: Can analyze its sales data, sliced by store, product, and time, to optimize marketing. They can create a KPI for "Sell-Through Rate".
  - **SPM**: Can analyze production data. By correlating FulfillOrder Interaction times with the ProcureMaterials Interaction times, they can create an indicator for "Production Delay due to Material Shortage."
- **Federated BI**:

    - Because all entities use a common (though decentralized) identity system (DIDs) and a common dimensional model, they can agree to share specific, anonymized ContextMeasure data.
    - They can build a federated view of the entire supply chain's health, analyzing end-to-end efficiency from raw material order to final consumer sale, without exposing sensitive internal operational data.

Appendix D: Advanced Relational Inference with FCA and Prime Semantics

This appendix details the algorithms and methods for inferring and materializing implicit relational knowledge from the explicit data ingested into the framework. This is a core function of the **Alignment Service**.

D.1. FCA-based Relational Schema Inference

The system can infer relational schemas (rules or "upper concepts") from the structure of the data itself using FCA.

- **FCA Contexts for Relational Analysis**: We use three types of FCA contexts to analyze relationships from different perspectives:
    1. **Predicate-as-Context**: (G: Subjects, M: Objects, I: relation). This context reveals which types of subjects relate to which types of objects for a given predicate.
    2. **Subject-as-Context**: (G: Predicates, M: Objects, I: relation). This reveals all the relationships and objects associated with a given subject, defining its role.
    3. **Object-as-Context**: (G: Subjects, M: Predicates, I: relation). This reveals all the subjects and actions that affect a given object.
- **Algorithm: Inferring Relational Schema**:
    1. **Select Context**: For a given predicate P (e.g., :worksOn), the Alignment Service constructs the Predicate-as-Context.
    2. **Build Lattice**: It uses an FCA library (e.g., fcalib) to compute the concept lattice from this context.
    3. **Identify Formal Concepts**: Each node in the lattice is a *formal concept* (A, B), where A is a set of subjects (the "extent") and B is the set of objects they all share (the "intent").
    4. **Materialize Schema**: Each formal concept represents an inferred relational schema or "upper concept". The system creates a new RDF class for this concept. For a concept where the extent is {dev1, dev2} (both :Developers) and the intent is {projA, projB} (both :Projects), the system can materialize a schema:

```
:DeveloperWorksOnProject a rdfs:Class, :RelationalSchema ;
   :hasDomain :Developer ;
   :hasRange :Project .
```

D.2. Materializing Relational Instances

Once a schema is inferred, the system can find and explicitly link all data instances that conform to it.

- **Algorithm: Materializing Instances**:
  1. **Iterate Schemas**: The Alignment Service iterates through the newly inferred :RelationalSchema classes.
  2. **Construct Query**: For each schema (e.g., :DeveloperWorksOnProject), it constructs a SPARQL query to find all raw statements that fit the schema's domain and range.
     ```
     SELECT ?statement WHERE {
        ?statement a rdf:Statement ;
                rdf:subject ?s ;
                rdf:object ?o .
        ?s a :Developer .
        ?o a :Project .
     }
     ```
  3. Link Instance to Schema: For each matching ?statement, it materializes a new triple, explicitly typing the statement as an instance of the inferred schema.
     ```
     <:stmt_123> rdf:type :DeveloperWorksOnProject .
     ```
     This makes future queries for "all instances of developers working on projects" much faster, as it becomes a simple rdf:type lookup.

D.3. Inferring and Materializing Order Relations

Order is inferred from type/state hierarchies and then materialized as explicit links.

- **Schema Inference**:
  - **Method**: The Alignment Service analyzes state transition data. It observes that instances of :Order consistently transition from a state of :Placed to :Paid to :Shipped.
  - **Materialization**: It creates a directed graph of the state schema using a custom property:
    ```
    :Paid :isPrecededBy :Placed .
    :Shipped :isPrecededBy :Paid .
    ```

- **Instance Materialization**:
  - **Method**: The service listens for events. When it sees an OrderPaid event for order_789 at T1 and later an OrderShipped event for the same order at T2, it can materialize the instance of the order relation.
  - Materialization: It creates a direct link between the reified event statements: <:event_ship_789> :isPrecededBy <:event_paid_789> .

D.4. Inferring Domain-Specific & Transitive Relations

This is where the system learns complex, multi-hop rules from the data.

- **Attribute Closure (e.g., knowsLanguage)**:
  - **Schema Inference Algorithm**:
    1. Run a SPARQL query to find recurring paths of length two: SELECT ?r1 ?r2 WHERE { ?a ?r1 ?b . ?b ?r2 ?c . }.
    2. Aggregate the results. A frequent co-occurrence of (:Developer)-[:worksOn]->(:Project) and (:Project)-[:usesLanguage]->(:Language) suggests a new potential relation.
    3. Materialize the new relational schema: :knowsLanguage owl:propertyChainAxiom (:worksOn :usesLanguage) . This uses OWL 2's powerful property chain axiom to formally define the rule.
  - **Instance Materialization**: Once the rule is defined, the Jena reasoner can automatically infer the (:Developer)-[:knowsLanguage]->(:Language) triples without needing to materialize them explicitly. Queries for :knowsLanguage will return results as if the triples existed.
- **Link Inference (e.g., uncleOf)**:
  - **Schema Inference**: The process is identical to attribute closure. The system detects the frequent path (:Person)-[:brotherOf]->(:Person)-[:fatherOf]->(:Person) and defines the uncleOf property chain.
  - **Instance Materialization**: Again, the OWL reasoner handles the inference.
- **Transitivity (e.g., partOf)**:
  - **Property Inference Algorithm**:
    1. The system queries for instances of the pattern (?a :partOf ?b) , (?b :partOf ?c).
    2. For each result, it checks if the triple (?a :partOf ?c) also exists.
    3. If this holds true for a high percentage of cases, the system can infer that the :partOf relation is likely transitive.
  - **Schema Materialization**: It formally declares the property as transitive: :partOf rdf:type owl:TransitiveProperty .
  - **Instance Inference (Transitive Closure)**: The OWL reasoner will automatically

handle the transitive closure. A query for all parts of C will correctly return A even if only the (A, B) and (B, C) links are explicit.

D.5. Advanced Calculations and Further Inferences

- **Symmetry/Reflexivity**: The same statistical analysis used for transitivity can be used to infer if a property is symmetric (e.g., :spouseOf) or reflexive. These are also declared using OWL (owl:SymmetricProperty, owl:ReflexiveProperty).
- **Analogous Relationships via CPPE**: The GCD of Contextual Prime Product Embeddings can reveal more than just direct similarity. If GCD(CPPE(A, :manages), CPPE(B, :mentors)) is high, it suggests that A and B have similar teams, implying an analogous relationship between "managing" and "mentoring" in this domain. This can be flagged for a human expert to review and potentially create a new rdfs:subPropertyOf axiom.
- **Query Acceleration**: All materialized relationships (both schema and instance) act as "shortcuts" in the graph. A complex, multi-hop query for "uncles" becomes a single-hop query for rdf:type :UncleOfRelation, which is dramatically faster and less computationally expensive, directly benefiting the Activation and BI layers.

Appendix E: Numerical Representation and Inference of Relational Schemas

This appendix details a novel method for representing both relational schemas (rules) and instances (data) as numerical vectors, enabling inference, querying, and traversal through direct mathematical operations. This elevates the CPPE concept to a new level of abstraction.

E.1. The Relational Context Vector (RCV)

The core of this approach is the **Relational Context Vector (RCV)**. For any given statement (a reified triple), we compute a vector of three BigInteger values, (S, P, O). Each component is a CPPE calculated from one of the three FCA context perspectives, providing a holistic numerical signature of the statement's role in the graph.

- **RCV Definition**: RCV(statement) = (S, P, O)
    - **S (Subject Context Embedding)**: The CPPE of the statement's **subject** from the **Subject-as-Context** perspective. This number encodes *everything the subject does*.
        - S = calculateCPPE(statement.subject, SubjectAsContext)
    - **P (Predicate Context Embedding)**: The CPPE of the statement's **predicate** from the **Predicate-as-Context** perspective. This number encodes *every subject-object pair the predicate connects*.

        - P = calculateCPPE(statement.predicate, PredicateAsContext)
    - **O (Object Context Embedding)**: The CPPE of the statement's **object** from the **Object-as-Context** perspective. This number encodes *everything that happens to the object*.
        - O = calculateCPPE(statement.object, ObjectAsContext)
- **Implementation**: A Java record RCV(BigInteger s, BigInteger p, BigInteger o). The **Index Service** is responsible for calculating and caching the RCV for every reified statement in the graph.

E.2. Numerical Representation of Schema vs. Instance

This dual representation is key to performing inference.

- **Instance RCV**: The RCV calculated for a specific, concrete statement (e.g., stmt_123: (dev:Alice, :worksOn, proj:Orion)) is its unique numerical signature. It represents a single data point.
- **Schema RCV (Archetype)**: The RCV for a *relational schema* (e.g., the :DeveloperWorksOnProject schema) is an "archetype" vector. It is calculated by finding the **Least Common Multiple (LCM)** of the corresponding components of all instance RCVs that belong to that schema.
    - **Algorithm: calculateSchemaRCV(schemaURI)**
        1. Find all instance statements s_i where s_i rdf:type schemaURI.
        2. For each instance s_i, retrieve its cached RCV_i = (S_i, P_i, O_i).
        3. Calculate the schema RCV components:
            - S_schema = LCM(S_1, S_2, ..., S_n)
            - P_schema = LCM(P_1, P_2, ..., P_n)
            - O_schema = LCM(O_1, O_2, ..., O_n)
        4. The result (S_schema, P_schema, O_schema) is the numerical archetype for the schema. The LCM ensures that the schema's numerical signature is "divisible" by all of its instances.

E.3. Inference via Mathematical Operators

With schemas and instances represented numerically, inference becomes a set of direct mathematical tests.

E.3.1. Subsumption / Instance Checking (rdf:type)

- **Concept**: An instance belongs to a schema if the instance's RCV "divides into" the schema's RCV.
- **Algorithm: isInstanceOf(instanceRCV, schemaRCV)**
    1. Perform a component-wise modulo operation.
    2. boolean isS = schemaRCV.s.mod(instanceRCV.s).equals(BigInteger.ZERO);

3. boolean isP = schemaRCV.p.mod(instanceRCV.p).equals(BigInteger.ZERO);
4. boolean isO = schemaRCV.o.mod(instanceRCV.o).equals(BigInteger.ZERO);
5. Return isS && isP && isO.

- **Use Case**: This is a high-speed, purely numerical method for checking type constraints, which can be performed in memory without a complex graph query.

E.3.2. Relational Composition / Link Inference (uncleOf)

- **Concept**: We can "compose" the RCVs of two adjacent relationships to calculate the RCV of the inferred, transitive relationship.
- **Algorithm: compose(RCV1, RCV2)**
  - Let RCV1 be for (A, :brotherOf, B).
  - Let RCV2 be for (B, :fatherOf, C).
  - The inferred relation is (A, :uncleOf, C).
  1. **Subject Component (S_uncle)**: The subject of the new relation is A. Its context is now expanded by the context of B's role in the second relation. We can approximate this by combining their S components. A simple multiplication is a reasonable starting point: S_uncle = RCV1.s.multiply(RCV2.s).
  2. **Object Component (O_uncle)**: The object of the new relation is C. Its context is expanded by the context of B's role in the first relation. O_uncle = RCV1.o.multiply(RCV2.o).
  3. **Predicate Component (P_uncle)**: The predicate is the most complex. It represents the combination of the two original predicate contexts. P_uncle = RCV1.p.multiply(RCV2.p).
  4. The resulting RCV_uncle = (S_uncle, P_uncle, O_uncle) is the calculated numerical signature for the inferred uncleOf relationship instance between A and C.
- **Use Case**: The system can now find all "uncles" of C by calculating the target RCV_uncle and then searching the Index Service for existing statements whose RCVs are mathematically similar (e.g., have a high GCD) to the target. This turns a complex path-finding query into a numerical index lookup.

E.4. Querying and Traversal by Numerical Properties

This numerical representation unlocks new ways to query the graph.

- **Find by Relational Role**: "Find all entities that have acted as a :Developer".
  - Instead of ?x a :Developer, we can query numerically. First, calculate the archetypal RCV for the :DeveloperWorksOnProject schema. Let this be RCV_dev_schema.
  - The query becomes: "Find all statements whose instanceRCV.s component divides RCV_dev_schema.s." This finds all statements where the subject is

    playing a role consistent with being a developer.

- **Traversal by Numerical Similarity**:
    - Start at a given statement stmt_A. Calculate its RCV_A.
    - The next step in the traversal could be: "Find the statement stmt_B in the graph whose RCV_B has the highest GCD with RCV_A."
    - This allows for a novel form of graph traversal where the path is not defined by explicit links, but by moving from one numerically similar relational context to the next. This could be used to discover analogous processes or events across different domains within the integrated enterprise data.