

Implementation Roadmap: Application Service Framework

Version: 1.4

Date: 2025-07-25

1. Introduction

This document provides a comprehensive, implementation-focused roadmap for the Application Service framework. It breaks down each phase into specific technical tasks, architectural decisions, and technology choices. This version provides a deep dive into the **reactive and functional programming paradigms** central to the architecture, ensuring a non-blocking, event-driven dataflow. It offers detailed explanations and examples of the practical application of key patterns and frameworks like **DCI, DDD, Spring AI, and a significant focus on the Reference Model, Formal Concept Analysis (FCA), the Set-Oriented Graph Model, Dimensional Ordering, and the Activation Model** and the agentic communication layer enabled by the **Model Context Protocol (MCP)** and **W3C Decentralized Identifiers (DIDs)**., with inline citations to the provided reference materials.

Phase 1: Core Infrastructure & Data Ingestion (Months 1-3)

Objective: Establish a robust, scalable, and fully reactive microservices foundation and a versatile data ingestion pipeline.

1.1. Components & Implementation Details:

- **Datasource Service (Java, Spring Boot):**
 - **Core Logic:** Implement a DataSourceAdapter interface with concrete strategies for each data source type.
 - JdbcAdapter: Use spring-boot-starter-data-jdbc and JdbcTemplate for direct SQL execution. A configuration file will map tables and columns to predicate names (e.g., users.name -> hasName). The adapter will dynamically query table metadata to handle schema evolution.
 - RestApiAdapter: Use spring-webflux's non-blocking WebClient. It will support paginated APIs by following next links in response headers or bodies.
 - FileAdapter: Use the Jackson library for JSON/XML parsing. It will watch a designated directory for new or updated files.
 - **Transformation:** The core transformation logic will convert source entities into SPO triples. For a database row (PK=123, table='Product', column='Name', value='Laptop'), the output will be a message: ("product:123", "hasName", "Laptop", "source:db1"). The subject URI is a composite of the entity type and

its primary key.

- **Synchronization:** Implement a polling mechanism using @Scheduled annotations in Spring for sources without push notifications. For event-driven sources, it will expose a webhook endpoint to receive update events. Provenance is maintained by adding a context string (e.g., the source application's name) to each triple.
- **Augmentation Service (Java, Spring Cloud Stream):**
 - **Message Bus:** Use Apache Kafka as the backbone. Define clear, versioned Avro schemas for all message types to ensure compatibility.
 - **Topics:**
 - datasource-raw-triples-v1: For raw data from the Datasource Service.
 - aggregation-reference-model-v1: For typed and identified data.
 - alignment-graph-model-v1: For semantically enriched data.
 - activation-dci-model-v1: For executable use cases.
 - **Orchestration & Saga Pattern:** Implement the Saga pattern using a state machine. For a multi-step process like "Ingest and Align," the service listens for a RAW_TRIPLE_INGESTED event, triggers the Aggregation Service, then listens for an AGGREGATION_COMPLETE event to trigger the Alignment Service. State transitions and compensating actions (e.g., deleting partially processed data on failure) are logged to a dedicated Kafka topic (saga-log-v1).
 - **Resiliency:** Use Spring Retry for transient failures and a dead-letter queue (DLQ) pattern for messages that repeatedly fail processing.
- **Registry Service (Helper Service - Java, Spring Boot, Neo4j):**
 - **Database:** Use a Neo4j graph database.
 - **Schema:** Nodes will have the label :Resource and a uri property (which is indexed). Relationships will represent the predicates.
 - **API:** A RESTful API built with Spring Boot and spring-data-neo4j.
 - POST /v1/graph/statements: A batch endpoint that accepts a list of triples and executes a single, optimized Cypher UNWIND ... MERGE query for high-performance writes.
 - GET /v1/graph/resource?uri={uri}: Retrieves a resource and its immediate relationships.
 - **Provenance:** Store provenance data (e.g., sourceApplication, ingestionTimestamp) as properties on the nodes and relationships.

Phase 2: Semantic Core & Knowledge Representation (Months 4-7)

Objective: Transform raw data into an interconnected, semantically rich knowledge

graph.

2.1. Deep Dive: The Reference Model and Prime Number Semantics

The **Reference Model**, produced by the **Aggregation Service**, moves from string-based URIs to a formal, mathematically grounded identification system.

- **ID & IDOccurrence:** An ID is the canonical concept of an entity, identified by a unique primeID. An IDOccurrence is an ID appearing in a specific role within a specific context (e.g., as the subject of a statement).
- **Prime Number Semantics:** We leverage the **Fundamental Theorem of Arithmetic**. An IDOccurrence's "embedding" is a set of prime IDs defining its full context. Similarity is a deterministic Jaccard Index on these sets.
- **FCA with Prime IDs:** In Formal Concept Analysis, we use primeIDs for objects and attributes. A concept's "intent" (its set of shared attributes) can be uniquely identified by the **product of its attribute primeIDs**. This allows for hyper-efficient subsumption checking: **a concept C1 is a sub-concept of C2 if C1's intent-product is cleanly divisible by C2's intent-product**. This transforms expensive set logic into simple integer arithmetic, a technique vital for large-scale inference as explored in works like "Formal Concept Analysis for Knowledge Discovery and Data Mining".

2.1. Components & Implementation Details:

- **Aggregation Service (Java, Spring AI, Python):**
 - **Core Logic:** Consumes from the datasource-raw-triples-v1 topic.
 - **ID & Embedding Generation:** For each new URI, generate a unique ID and an embedding vector. This can be a separate Python service called via RPC, using models like Sentence-BERT from the Hugging Face library to create meaningful embeddings. The mapping of URI to ID and embedding is cached in Redis.
 - **Type/State Inference:** Use in-memory Caffeine caches for high-speed aggregation.
 - **Map<String, Set<String>> subjectToPredicates:** This map tracks all attributes for a given subject.
 - A background job periodically analyzes this map. Subjects with a high Jaccard similarity in their predicate sets are grouped into an inferred Type.
 - **FCA (Formal Concept Analysis):** Use the fcalib Java library. Create a formal context where "objects" are the subject URIs and "attributes" are their predicates. The resulting concept lattice directly forms the is-a type hierarchy.
 - **Output:** Produces Statement<ID, ID, ID, ID> messages to the

aggregation-reference-model-v1 topic.

Given a set of raw SPO triples from Datasources Service, performs type inference (common attributes aggregation) and state inference (common attribute values aggregation) and performs type / state hierarchies inference.

Type inference: Subjects with the same Attributes belong to the same type.

State inference: Subjects with Attributes (types) with the same Values are in the same state.

Hierarchies: Attributes / Values subset / superset relationship (less common attributes are “higher” into the type hierarchy, same for values). Entities with the same attributes are considered as of the same type, superset / subset of attributes: type hierarchy. Attributes with the same values, same states. Superset / subset of values / states: state hierarchy.

Types are ordered in respect to their common attributes. Most specific types (more common attributes) are considered to inherit from types with less common attributes included into the more specific types. A more specific type is considered to be “after” a more generic type (Person → Employee). Regarding state values, hierarchies are to be considered regarding attribute values, being resources with common state grouped into hierarchies (Marital status attribute: Single → Married → Divorced).

Order: Inferred via Type / State hierarchies. Types: Married extends from Single, Divorced extends from Married. States: Young extends from Child, Old extends from Young. Cycles in types resolved by state (Unemployed, Employed, Unemployed). Used in Alignment Service Ordering upper ontology.

Data structures:

```
Map<Subject, Set<Predicate>
Map<Set<Predicate>, Type>
Map<Type, Set<Map<Predicate, Value>>>
Map<Set<Map<Predicate, Value>>, State>
```

- **Alignment Service (Java, RDF4J):**

- **Core Logic:** Consumes from the aggregation-reference-model-v1 topic.
- **Ontology Matching:** Use the RDF4J framework's MemoryStore for in-memory graph operations. Load the Reference Model and pre-defined upper ontologies (e.g., Schema.org, custom domain ontologies in OWL format). Use the SPARQL engine with SHACL rules to find and materialize equivalences (owl:sameAs, rdfs:subClassOf).
- **Link Completion:** Implement this with SPARQL CONSTRUCT queries. For example, a query can find paths like (A)-[:hasRole]->(B) and

(B)-[:partOf]->(C) to infer a new link (A)-[:contributesTo]->(C).

- **Output:** Produces enriched Statement<Context, Subject, Predicate, Object> messages to the alignment-graph-model-v1 topic.

Upper ontologies:

a) Domains: Aligned integrated application domains inferred common concepts and relationships. Infer equivalent concepts and relationships between source applications domains and populate Domains upper ontology. Materialize integrated domains concepts and relationships mappings to inferred upper concepts and relationships. Abstract common meaning (semantics) of source applications concepts and relationships to enable inter domain contexts interactions.

b) Order: Dimensional arrangement of entities attributes and values. Align measures (attribute values) into dimensional units. According Aggregation Service types and states hierarchies establish order relationships (before, greater than, contains, etc.) between measures. Materialize measures relationships and map dimensional units measures occurrences into the materialized order relationships. See: [Dimensional Features].

Ontology Matching: Find and map equivalent entities and relationships domains occurrences (Core Model Classes), align core model resources into Domains upper ontology.

Links / Attributes inference: Given an aligned model (mapped to Domains upper ontology) infer possible links / relationships between resources and possible attributes and their values.

Ordering: Order dimensional upper ontology alignment. Materialize inferred Type / State hierarchies order relationships.

Multidimensional features (OLAP like):

Dimensions: Time, Product, Region.

Units: Month / Year, Category / Item, State / City.

Context : (Context, Attribute, Value)

Examples:

(soldDate, aProduct, aDate)

((soldDate, aProduct, aDate), Product, aProduct)

((soldDate, aProduct, aDate), Product, aProduct), Region, aRegion)

TODO: Materialize / Query Cubes Context Statements into graph models.

- **Naming Service (Helper Service - Java, Apache Jena):**

- **Core Logic:** A dedicated service that manages ontologies.

- **Storage:** Use Apache Jena with a TDB2 persistent backend.
- **API:** Expose a full SPARQL 1.1 endpoint using Jena Fuseki. This allows other services to query the ontologies directly. It will also have custom REST endpoints like POST /v1/align/concepts which takes two sets of concepts and returns a mapping of potential matches with confidence scores.

Phase 3: Activation & Use Case Enablement (Months 8-10)

Objective: Infer and enable the execution of business processes by implementing a dynamic, message-driven model based on DCI, DDD, and Dynamic Object Model principles.

3.1. Deep Dive: The Activation Model's Dynamic Object Model (DOM)

The **Activation Service** consumes the semantically rich Graph Model and produces the Activation Model. This is not a static data structure but a **Dynamic Object Model (DOM)**, where an object's capabilities can change at runtime. This is a direct implementation of the ideas found in works like "Dynamic Object Model" and the Actor Role pattern. Its core entities are Class, Instance, Actor, Role, Context, Interaction, Transform, and Dataflow.

3.2. Patterns in Practice: DDD, DCI, and the Actor-Role Model

- **DDD (Domain-Driven Design):** The entire Activation Service is a single **Bounded Context**. Its Ubiquitous Language consists of the entities above (Context, Role, etc.).
- **DCI (Data, Context, and Interaction):** The pattern is the blueprint for the runtime logic. An Interaction (Context) "casts" plain data Instances into Actors by dynamically injecting Roles (behavior) for the duration of the use case. This avoids bloating data objects with all possible behaviors, a core tenet of DCI as described by James Coplien and Trygve Reenskaug.

3.3. Deep Dive: Actor State and Dataflow via Transforms

An Actor's state is its position in the use case flow, modeled as a set of available Transforms (previous, current, next). The dataflow is driven by a reactive stream of declarative Transform messages, which instruct Actors on how to mutate their internal state. This creates a scalable, auditable, and distributed state machine.

Phase 3.5: LLM Integration & Agentic Architecture (MCP, DIDs, COST)

This phase runs in parallel with the latter part of Phase 3 and the start of Phase 4. It elevates the Activation Service from a simple orchestrator to an intelligent, agentic system capable of communicating with LLMs and other ApplicationService instances

using standardized protocols.

3.4. Deep Dive: The ApplicationService as a Model Context Protocol (MCP) Server

The ApplicationService will expose an **MCP Server** endpoint, allowing external clients (like LLM-based agents or other ApplicationService instances) to interact with its capabilities in a standardized way. We will use **Spring AI** as the primary tool to bridge our internal services with the LLM world.

- **MCP Endpoint Implementation (Spring WebFlux):** A single REST endpoint (/mcp) will handle all MCP requests. The request body will specify the desired capability (resource, tool, or prompt_template).
- **Exposing Capabilities via MCP & Spring AI:**
 1. **Resources (Aggregation/Index):** An MCP client can ask for resources. "Find me resources similar to 'a senior Java developer'."
 - **Implementation:** The MCP endpoint routes this to the Index Service. The text query is fed into Spring AI's ReactiveEmbeddingClient to get a vector. This vector is used to perform a similarity search in the vector DB. The results (a Flux of resource IDs) are returned.
 2. **Tools (Activation/Registry):** An MCP client can request to use a tool. "Execute the 'OnboardNewEmployee' tool for resource 'user:JohnDoe'."
 - **Implementation:** This is the core of the agentic behavior. The MCP endpoint maps the tool name "OnboardNewEmployee" to an Activation Context. It then instantiates an Interaction for that Context and assigns user:JohnDoe as an Actor. The Interaction's dataflow is executed. The LLM decides *what* to do; our framework provides the verifiable, stateful Tool to *do it*.
 3. **Prompt Templates (Alignment/Naming):** An MCP client can request a template for complex reasoning. "Give me the 'ConceptAlignment' prompt template to compare 'Customer' and 'Client'."
 - **Implementation:** The endpoint fetches a pre-defined prompt string from the Naming Service. This template has placeholders for context (e.g., attributes of 'Customer' and 'Client'). The MCP client populates these and sends the completed prompt to an LLM using Spring AI's ReactiveChatClient. The LLM's response (e.g., a mapping of equivalent attributes) can then be fed back into the system to augment the Graph Model.

3.5. Deep Dive: COST (COnversational State Transfer) & The HAL Protocol

The communication between the Producer Service (the client) and the Activation Service (the server) will be implemented as **COST**, a stateful, conversational protocol

built on the principles of **HATEOAS** using the **Hypertext Application Language (HAL)** specification.

- **Principle:** Every API response not only contains the state of a resource but also the links (`_links`) to all possible actions (the next Transforms) that can be taken from that state. The client does not need to hardcode application logic; it just needs to know how to follow links.
- **Example HAL Response for an Actor:**

```
{
  "actorId": "user:Alice",
  "role": "Buyer",
  "state": "AwaitingPurchaseConfirmation",
  "instanceData": {
    "shippingAddress": "123 Main St",
    "paymentMethod": "***** * 1234"
  },
  "_links": {
    "self": { "href": "/interactions/123/actors/user:Alice" },
    "interaction": { "href": "/interactions/123" },
    "next": [
      {
        "href": "/interactions/123/transform",
        "method": "POST",
        "name": "ConfirmPurchase",
        "title": "Confirm and Finalize Purchase"
      },
      {
        "href": "/interactions/123/transform",
        "method": "POST",
        "name": "ChangeShippingAddress",
        "title": "Edit Shipping Address"
      }
    ],
    "previous": {
      "href": "/interactions/123/revert",
      "method": "POST",
      "name": "RevertPaymentSelection"
    }
  }
}
```


}

The UI simply renders a button for each object in the `_links.next` array. This makes the frontend incredibly dynamic and resilient to changes in the backend workflow.

3.6. Deep Dive: W3C DIDs for Decentralized & Verifiable Identity

To ensure security, provenance, and interoperability, all canonical resource IDs will be **W3C Decentralized Identifiers (DIDs)**.

- **Implementation:**

1. **Creation:** In the Aggregation Service, when a new resource is first encountered, we will use a library like **did-common-java** to generate a `did:ion` or `did:key`. The ID's `primeID` can even be part of the DID string for deterministic generation.
2. **DID Document:** The generated DID Document (containing cryptographic keys and service endpoints like the resource's MCP endpoint) is stored in the Registry Service's property graph, linked to the resource node.
3. **Usage:** The resource's canonical identifier throughout the system becomes its DID (e.g., `did:ion:Ei...`).

- **Enabled Features:**

- **Verifiable Provenance:** Any Statement created by an `ApplicationService` can be cryptographically signed using the private key associated with the service's own DID. Downstream consumers can verify this signature, guaranteeing data integrity and non-repudiation.
- **Secure Interoperability:** When one `ApplicationService` acts as an MCP Client to another, it can use **DID-Auth** to authenticate. This eliminates the need for API keys or pre-shared secrets, enabling a zero-trust, federated network.
- **Decentralized Discovery:** The `serviceEndpoint` in a resource's DID Document can point directly to its MCP API, allowing different instances to dynamically discover how to interact with each other.

3.7. The `ApplicationService` as an MCP Client

The framework's true power is realized when an `ApplicationService` instance can act as a client to others.

- **Scenario:** A local Interaction for "HireEmployee" requires a "PerformBackgroundCheck" tool, which is provided by a trusted, external HR `ApplicationService`.
- **Implementation:**
 1. The local Activation Service determines the need for the external tool.
 2. It looks up the HR service's DID in its Registry.

3. It resolves the DID to find the HR service's MCP endpoint from its DID Document.
4. It authenticates using DID-Auth.
5. It sends an MCP request: { "capability": "tool", "name": "PerformBackgroundCheck", "params": { ... } }.
6. The remote service executes the tool and returns the result.
7. The local Activation Service receives the result and integrates it into its own Interaction dataflow, advancing the "HireEmployee" process.

Phase 3: Activation & Use Case Enablement (Months 8-10)

Objective: Infer and enable the execution of business processes by implementing a dynamic, message-driven model based on DCI, DDD, and Dynamic Object Model principles.

3.1. Deep Dive: The Activation Model's Dynamic Object Model (DOM)

The **Activation Service** consumes the semantically rich Graph Model and produces the Activation Model. This is not a static data structure but a **Dynamic Object Model (DOM)**, where an object's capabilities can change at runtime. This is a direct implementation of the ideas found in works like "Dynamic Object Model" and the Actor Role pattern.

- **Core Entities:**

- **Class:** The schema for an object. It defines the fields (attributes) an object can have. This is analogous to a Java Class definition but is itself a data object that can be created or modified at runtime.
- **Instance:** An instantiation of a Class. It holds the actual data in its attributes map. This is the "Data" part of DCI.
- **Actor:** An Instance that is actively participating in a use case. It is stateful.
- **Role:** A behavioral contract. It defines the capabilities and dataflows an Actor can have within a specific Context. This is the "Interaction" part of DCI.
- **Context:** The schema for a use case. It defines the Roles that must be filled for the use case to proceed. This is the "Context" part of DCI.
- **Interaction:** A running instance of a Context. It's the stateful orchestrator that binds Actors to Roles.
- **Transform:** A message representing a single, atomic state change operation to be performed by an Actor.
- **Dataflow:** A rule within a Context that defines the sequence of Transforms for a given Role.

- **Parallel Statement Types:**

- **Schema Statements:** Statement<Context, Role, Dataflow>. These define the

"rules of the game." For example: "In the Purchase Context, the Buyer Role follows a Dataflow that involves creating a payment."

- **Data Statements:** Statement<Interaction, Actor, Transform>. These represent the actual "moves in the game." For example: "In Interaction #123, Actor user:Alice executes the Transform to set her payment method."

3.2. Patterns in Practice: DDD, DCI, and the Actor-Role Model

- **DDD (Domain-Driven Design):** The entire Activation Service is a single **Bounded Context**. Its Ubiquitous Language consists of the entities above (Context, Role, Interaction, etc.). It listens for Domain Events from the Alignment service (e.g., NewProductKindDiscovered) and uses them to infer and create new Context schemas.
- **DCI (Data, Context, and Interaction):** This pattern is the blueprint for the service's runtime logic.
 1. **Data:** The Instances (e.g., a specific user, a specific product) are the plain data objects. They have state but no intrinsic business logic.
 2. **Context:** An Interaction is created (e.g., a user clicks "Buy"). This Interaction is the Context.
 3. **Interaction (The "Casting"):** The Interaction "casts" Instances into Actors by assigning them Roles. The user:Alice Instance is now the Buyer Actor. The product:Laptop Instance is now the ItemForSale Actor. The Role (Buyer), which contains the business logic, is dynamically injected into the Actor for the duration of this Interaction. This avoids bloating data objects with all possible behaviors, a core tenet of DCI as described by James Coplien and Trygve Reenskaug.

3.3. Deep Dive: Actor State and Dataflow via Transforms

The core of the model's dynamism lies in how Actors manage state and interact via Transform messages.

- **Actor State (previous, current, next):**
 - An Actor's state is not just its data; it's its position in the use case flow. This is explicitly modeled as a set of available Transforms.
 - **current: Map<Context, Transform>:** The Transform that led the Actor to its current state.
 - **next: Map<Context, Transform[]>:** A map of the **available Transforms** the Actor can execute next within a given Context. This is the system's "API" at runtime. The Producer service reads this map to render the available buttons or actions to the user.
 - **previous: Map<Context, Transform>:** The Transform that can be used to

revert the current state (for undo functionality).

- **Implementing Dataflow with Transform Messages:**

- A Dataflow is a sequence of Transforms. A Transform is a declarative message, not imperative code. It's a data object that instructs an Actor on how to mutate its internal Instance data (its DOM).

- **Transform Message Structure:**

```
{  
  "transformId": "txf_987",  
  "targetActorId": "user:Alice",  
  "operation": "SET_FIELD", // or GET_FIELD, ADD_TO_LIST, MUTATE_FIELD  
  "payload": {  
    "fieldName": "shippingAddress",  
    "value": { "street": "123 Main St", "city": "Anytown" }  
  }  
}
```

- **Message-Driven Implementation:**

1. The Interaction orchestrator (or another Actor) publishes a `Statement<Interaction, Actor, Transform>` message to a Kafka topic.
2. The target Actor (a stateful microservice instance or an object managed by the Activation Service) consumes this message.
3. The Actor inspects the Transform's operation and payload.
4. It applies the change to its internal Instance object's attributes map. For example, for `SET_FIELD`, it executes `this.instance.attributes.put(fieldName, value)`.
5. After successfully applying the Transform, the Actor updates its own previous, current, and next state maps based on the Dataflow rules defined in its Role.
6. It then emits an `ActorStateUpdated` event, potentially triggering the next Transform in the sequence.

- **Example: "Purchase" Interaction Dataflow**

1. **Initial State:** Buyer Actor's next transforms include `[SelectPaymentMethod]`. Seller Actor's next is `[WaitForPaymentSelection]`.
2. User selects a credit card. The Producer sends a message that the Interaction translates into a Transform statement: `(interaction:123, actor:Buyer, transform:SetPaymentMethod)`.
3. The Buyer Actor consumes this. It updates its internal Instance data with the payment info. Its state changes:
 - current becomes `SetPaymentMethod`.

- next is now [ConfirmPurchase].
- 4. The Buyer emits BuyerPaymentMethodSet.
- 5. The Interaction orchestrator hears this and sends a Transform to the Seller: (interaction:123, actor: Seller, transform: ReceivePaymentNotification).
- 6. The Seller Actor consumes this. Its state changes:
 - current becomes ReceivePaymentNotification.
 - next is now [ShipItem].
- 7. This message-based flow of declarative Transforms continues until the Interaction is complete. This approach is highly scalable, auditable, and allows for complex, long-running, and distributed use cases.

Phase 3: Activation & Use Case Enablement (Months 8-10)

Objective: Infer and enable the execution of business processes and use cases from the knowledge graph.

3.1. Components & Implementation Details:

- **Activation Service (Java, Spring Boot):**
 - **Core Logic:** Consumes from the alignment-graph-model-v1 topic.
 - **DCI (Data, Context, Interaction):** Implement the DCI pattern.
 - **Context Inference:** Use graph traversal algorithms (e.g., Depth First Search) or Cypher queries on the Registry to find recurring patterns that represent potential use cases. For example, a pattern of (Order)-[contains]->(Product)<-[trackedIn]-(Inventory) infers a ReplenishStock Context.
 - **Role & Actor:** Roles are the types of nodes in the pattern (e.g., Product, Inventory). Actors are specific instances (e.g., product:123).
 - **Interaction:** An Interaction is an instantiated Context. It's a stateful object that tracks the assigned Actors and the progress of the use case.
 - **Dataflow & Rules:** Use a rules engine like Drools to define the business logic. A rule might be: WHEN Inventory.level < Inventory.threshold THEN CREATE ReplenishStock.Interaction. The actual data transformations between actors can be defined using XSLT or implemented as simple Java methods.
 - **Output:** Produces Statement<Context, Interaction, Role, Actor> to the activation-dci-model-v1 topic.
- **Index Service (Helper Service - Python, Vector DB):**
 - **Core Logic:** A service for similarity-based retrieval.
 - **Database:** Use a dedicated vector database like Milvus or Pinecone.
 - **API:**
 - POST /v1/index/resources: Adds a resource's embedding to the index.

- POST /v1/search/similar: Takes a vector and context filters (e.g., "find products similar to this one") and returns a list of matching resource URIs. This is used to find suitable Actors for a Role.

Phase 4: API & User Interface (Months 11-12)

Objective: Expose the framework's capabilities through a developer-friendly API and an intuitive user interface.

4.1. Components & Implementation Details:

- **Producer Service (API/Frontend - Java/Spring Boot, React):**
 - **Backend API:** A Spring Boot application that provides the public-facing interface.
 - **REST API:**
 - GET /v1/contexts: Lists available use cases.
 - POST /v1/interactions: Creates a new instance of a use case.
 - GET /v1/interactions/{id}: Retrieves the state of a specific transaction.
 - POST /v1/interactions/{id}/roles/{roleName}/assign: Assigns an actor to a role.
 - **Hypermedia (HATEOAS):** Use spring-boot-starter-hateoas. Each response will contain `_links` that guide the client. An Interaction response will have links like `assign-actor` or `complete-step`.
 - **Frontend (React):**
 - A Single-Page Application (SPA) built with React and TypeScript.
 - Use a component library like Material-UI or Ant Design for a consistent look and feel.
 - Implement a generic form renderer that builds input forms dynamically based on the JSON schema of the Roles provided by the API.
 - Use WebSockets to connect to the Augmentation Service (through an API Gateway) to receive real-time updates on the status of Interactions.
 - **Authentication:** Implement OAuth 2.0 with an identity provider like Keycloak or Auth0. The API Gateway will enforce authentication and authorization policies.

1.1. Components & Reactive Implementation Details:

- **Datasource Service (Java, Spring WebFlux):**
 - **Reactive Core:** The service will be built entirely on a non-blocking stack. Instead of traditional controllers, it will use Spring's functional handler functions.
 - **Reactive Ingestion:**

- **RestApiAdapter:** Will use WebClient to consume external APIs. The WebClient natively returns a Flux<T>, allowing the service to stream paginated results without holding a thread, processing each item as it arrives.
 - *Example:*

```
webClient.get().uri("/items?page=0").retrieve().bodyToFlux(Item.class)
    .expand(lastItem -> fetchNextPage(lastItem))...
```
- **R2DBCAdapter:** For supported SQL databases, it will use R2DBC (spring-boot-starter-data-r2dbc) to perform non-blocking database queries, returning a Flux<Row>.
- **Functional Transformation:** The transformation from source format to SPO triples will be a pure function within a reactive pipeline.
 - *Example (Project Reactor):*

```
Flux<SourceData> sourceStream = adapter.fetchData();
Flux<Statement<String,String,String,String>> tripleStream = sourceStream
    .flatMap(data -> Flux.fromIterable(transformer.toTriples(data))); //
1-to-many transform
```

This approach aligns with functional principles described in resources like "Functional Programming in JavaScript" by treating data transformation as a series of composable, stateless operations on a stream.

- **Augmentation Service (Java, Spring Cloud Stream):**
 - **Reactive Dataflow:** This service is the reactive backbone. It will be implemented using Spring Cloud Stream's functional programming model. Instead of @StreamListener, we define beans of type java.util.function.Function<Flux<T>, Flux<R>>. The framework automatically binds these to Kafka topics.
 - *Example:* A function that routes raw triples to the aggregation service.


```
@Bean
public Function<Flux<RawStatement>, Flux<AggregatableStatement>>
processRawTriples() {
    return flux -> flux
        .map(this::enrichWithMetadata)
        .log(); // Log each event in the stream
}
```

This embodies the principles of event-driven microservices discussed in the "Simple Event-Driven Microservices with Spring Cloud Stream" reference.

- **Saga Pattern (Reactive):** The Saga orchestrator will be implemented using Flux.usingWhen to manage transactional boundaries across services, ensuring that compensating actions are triggered reactively on error signals.
- **Registry Service (Helper Service - Java, Spring WebFlux, Neo4j):**
 - **Reactive API:** The REST API will be built with Spring WebFlux functional endpoints. Endpoints will return Mono<ServerResponse> for writes and Flux<Statement> for reads.
 - **Database Interaction:** While the official Neo4j Java driver is blocking, we can make it non-blocking from the perspective of the event loop by offloading the work to a dedicated scheduler.
 - *Example:*

```
public Mono<Void> saveStatement(Statement stmt) {
    return Mono.fromRunnable(() -> {
        // Blocking driver call
        session.run("MERGE (s:Resource {uri: $s_uri})", parameters("s_uri",
stmt.getSubject()));
    }).subscribeOn(Schedulers.boundedElastic()).then();
}
```

This prevents the blocking call from consuming a precious event-loop thread, a core tenet of reactive programming.

Phase 2: Semantic Core & Knowledge Representation (Months 4-7)

Objective: Transform raw data into an interconnected, semantically rich knowledge graph using reactive streams and AI/ML models.

2.1. Components & Reactive Implementation Details:

- **Aggregation Service (Java, Spring AI, Python):**
 - **Functional Aggregation Pipeline:** The core of this service is a multi-stage reactive pipeline.
 - *Example:*

```
// 1. Consume raw triples
Flux<RawStatement> rawStream = ...;
// 2. Group by subject to collect all predicates
Flux<GroupedFlux<String, RawStatement>> groupedBySubject =
rawStream.groupBy(RawStatement::getSubject);
// 3. Process each group to infer type
Flux<InferredTypeStatement> typeStream = groupedBySubject
.flatMap(group -> group
```

```

        .map(RawStatement::getPredicate)
        .collect(Collectors.toSet())
        .flatMap(this::inferTypeFromPredicates) // Calls FCA logic
    );

```

- **FCA (Formal Concept Analysis):** The inferTypeFromPredicates method will use fcalib (as cited in the references). The set of predicates for a group of subjects is used to build a FormalContext. The resulting ConceptLattice provides the type hierarchy, which is then flattened back into a Flux of type assertion statements. This aligns with the use of FCA for knowledge discovery outlined in papers like "Formal Concept Analysis for Knowledge Discovery and Data Mining".

- **Spring AI (Reactive Embeddings):** Embeddings will be generated within the reactive stream using Spring AI's ReactiveEmbeddingClient.

- *Example:*

```

// Inside the flatMap pipeline
.flatMap(statement ->
    reactiveEmbeddingClient.embed(statement.getObject())
        .map(embedding -> statement.withEmbedding(embedding))
)

```

This ensures that the network call to an embedding model (like one from Hugging Face or a local Ollama instance) is non-blocking.

- **Alignment Service (Java, RDF4J):**

- **Reactive Ontology Matching:** This service consumes the Reference Model stream. For each statement, it performs a lookup against the upper ontologies.

- **RDF4J Integration:** SPARQL queries via RDF4J will be wrapped in Mono.fromCallable and executed on a dedicated scheduler to avoid blocking.

- *Example:*

```

public Flux<Statement> align(Flux<Statement> statements) {
    return statements.flatMap(stmt ->
        Mono.fromCallable(() -> executeSparqlAlignment(stmt)) // Blocking
        call
        .subscribeOn(Schedulers.boundedElastic())
        .flatMapMany(Flux::fromIterable) // Flatten results into the stream
    );
}

```

This approach leverages the power of semantic frameworks like RDF4J within a fully reactive architecture, as envisioned by concepts in "SPARQL-Micro-Services".

- **Naming Service (Helper Service - Java, Apache Jena):**
 - **Reactive SPARQL Endpoint:** While Jena Fuseki is typically servlet-based, it can be proxied by a Spring WebFlux application to provide a fully reactive interface to the rest of the system, ensuring end-to-end non-blocking I/O.

Phase 3: Activation & Use Case Enablement (Months 8-10)

Objective: Infer and enable the execution of business processes using the DCI and DDD patterns within a reactive model.

3.1. Components & Reactive Implementation Details:

- **Activation Service (Java, Spring Boot):**
 - **DDD (Domain-Driven Design):** This service is a classic DDD Bounded Context. The Activation Model is its Ubiquitous Language. It consumes AlignmentModelChanged domain events from Kafka and produces InteractionStateChanged events. This follows the principles from Eric Evans' "Domain-Driven Design: Tackling Complexity in the Heart of Software".
 - **DCI (Data, Context, and Interaction):** This pattern is implemented reactively.
 - **Context:** A Context is a class that defines a use case. It contains the logic to find the required Roles. This logic can be a reactive graph query.
 - **Role:** A Role is a `java.util.function.Function<Flux<ActorState>, Flux<TransformedState>>`. It's a functional interface that defines the behavior an Actor will perform.
 - **Interaction:** An Interaction is a stateful, but non-blocking, orchestrator. When instantiated, it subscribes to the Flux streams representing the state of its assigned Actors. It then applies the Role functions to these streams to drive the use case forward. This dynamic composition of behavior is a core idea from the DCI papers by Trygve Reenskaug and James Coplien.
 - *Example:*

```
// An Interaction orchestrating a 'Buy' use case
Flux<BuyerState> buyerStream =
actorRepository.find(buyerId).getStateStream();
Flux<SellerState> sellerStream =
actorRepository.find(sellerId).getStateStream();
// Apply the Role functions
Flux<Payment> paymentStream = buyerRole.process(buyerStream);
```

```

Flux<Shipment> shipmentStream = sellerRole.process(sellerStream);
// Combine the results
Flux.zip(paymentStream,
shipmentStream).subscribe(this::handleCompletedTransaction);

```

- **Index Service (Helper Service - Python, Vector DB):**

- **Reactive Indexing:** It will subscribe to a Kafka topic of ResourceUpdated events. Using a reactive Kafka consumer (like aiokafka in Python), it will update the vector database (e.g., Milvus) as soon as a resource's embedding changes.

Phase 4: API & User Interface (Months 11-12)

Objective: Expose the framework's capabilities through a fully reactive API and a real-time user interface.

4.1. Components & Reactive Implementation Details:

- **Producer Service (API/Frontend - Java/Spring WebFlux, React):**

- **Fully Reactive API:** The entire API will be built with Spring WebFlux.
- **Server-Sent Events (SSE):** For real-time updates on long-running Interactions, the API will use SSE. A client can subscribe to an endpoint like GET /v1/interactions/{id}/stream, which returns a Flux<InteractionState> with the Content-Type of text/event-stream.

- *Example:*

```

@GetMapping(value = "/interactions/{id}/stream", produces =
MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<InteractionState> streamInteractionUpdates(@PathVariable
String id) {
    return
interactionRepository.findById(id).flatMapMany(Interaction::getStateStream);
}

```

This provides a much more efficient and standard-based alternative to WebSockets for server-to-client data pushes, as advocated in many reactive programming tutorials (e.g., "Building Reactive Microservices with Spring WebFlux").

- **Frontend (React with RxJS):** The React frontend will use a library like RxJS to manage the SSE streams from the backend. The state of a component can be directly bound to an Observable derived from the event stream, causing

the UI to update automatically and efficiently as new data arrives. This aligns with the "Thinking in React" and "Thinking in RxJava" mental models.

1.1. Components & Reactive Implementation Details:

- **Datasource Service (Java, Spring WebFlux):**

- **Reactive Core:** The service will be built entirely on a non-blocking stack using Spring WebFlux's functional handler functions instead of traditional controllers.
- **Reactive Ingestion:**
 - **RestApiAdapter:** Will use WebClient to consume external APIs. It natively returns a Flux<T>, allowing the service to stream paginated results without holding a thread, processing each item as it arrives.
 - *Example:*

```
webClient.get().uri("/items?page=0").retrieve().bodyToFlux(Item.class)
.expand(lastItem -> fetchNextPage(lastItem))...
```
 - **R2DBCAdapter:** For supported SQL databases, it will use R2DBC (spring-boot-starter-data-r2dbc) to perform non-blocking database queries, returning a Flux<Row>.
- **Functional Transformation:** The transformation from source format to SPO triples will be a pure function within a reactive pipeline. This aligns with functional principles described in resources like "Functional Programming in JavaScript" by treating data transformation as a series of composable, stateless operations on a stream.
 - *Example (Project Reactor):*

```
Flux<SourceData> sourceStream = adapter.fetchData();
Flux<Statement<String,String,String,String>> tripleStream = sourceStream
    .flatMap(data -> Flux.fromIterable(transformer.toTriples(data))); //
```

1-to-many transform

- **Augmentation Service (Java, Spring Cloud Stream):**

- **Reactive Dataflow:** This service is the reactive backbone, implemented using Spring Cloud Stream's functional programming model. We define beans of type `java.util.function.Function<Flux<T>, Flux<R>>`, which the framework automatically binds to Kafka topics. This embodies the principles of event-driven microservices discussed in the "Simple Event-Driven Microservices with Spring Cloud Stream" reference.
- **Saga Pattern (Reactive):** The Saga orchestrator will be implemented using `Flux.usingWhen` to manage transactional boundaries across services, ensuring that compensating actions are triggered reactively on error signals.

- **Registry Service (Helper Service - Java, Spring WebFlux, Neo4j):**
 - **Reactive API:** The REST API will be built with Spring WebFlux functional endpoints, returning `Mono<ServerResponse>` for writes and `Flux<Statement>` for reads.
 - **Database Interaction:** To keep the event loop non-blocking, the blocking Neo4j Java driver calls will be offloaded to a dedicated scheduler.
 - *Example:*

```
public Mono<Void> saveStatement(Statement stmt) {
    return Mono.fromRunnable(() -> {
        // Blocking driver call
        session.run("MERGE (s:Resource {uri: $s_uri})", parameters("s_uri",
            stmt.getSubject()));
    }).subscribeOn(Schedulers.boundedElastic()).then();
}
```

Phase 2: Semantic Core & Knowledge Representation (Months 4-7)

Objective: Transform raw data into an interconnected, semantically rich knowledge graph using reactive streams, Formal Concept Analysis, and a set-oriented model.

2.1. Deep Dive: The Reference Model and Prime Number Semantics

The **Reference Model**, produced by the **Aggregation Service**, moves from string-based URIs to a formal, mathematically grounded identification system.

- **ID & IDOccurrence:** An ID is the canonical concept of an entity, identified by a unique `primeID`. An IDOccurrence is an ID appearing in a specific role within a specific context (e.g., as the subject of a statement).
- **Prime Number Semantics:** We leverage the **Fundamental Theorem of Arithmetic**. An IDOccurrence's "embedding" is a set of prime IDs defining its full context. Similarity is a deterministic Jaccard Index on these sets.
- **FCA with Prime IDs:** In Formal Concept Analysis, we use `primeIDs` for objects and attributes. A concept's "intent" (its set of shared attributes) can be uniquely identified by the **product of its attribute primeIDs**. This allows for hyper-efficient subsumption checking: **a concept C1 is a sub-concept of C2 if C1's intent-product is cleanly divisible by C2's intent-product**. This transforms expensive set logic into simple integer arithmetic, a technique vital for large-scale inference as explored in works like "Formal Concept Analysis for Knowledge Discovery and Data Mining".

2.1. Deep Dive: The Reference Model and Prime Number Semantics

The **Reference Model** is the first layer of abstraction over raw data, produced by the **Aggregation Service**. Its purpose is to move from string-based URIs to a formal, mathematically grounded identification system that facilitates powerful inferences. It revolves around two key entities: ID and IDOccurrence.

- **ID Entity:**

- **Definition:** An ID represents the canonical, context-free identity of a resource. It is the "idea" of an entity. For example, the URI `http://example.com/users/alice` and the database row (users, PK=123) both resolve to the same single ID for the concept of "Alice".
- **primeID: long:** The core of the ID. Upon first encountering any new resource URI, the Aggregation Service assigns it a unique prime number. This is its immutable identifier.
- **Implementation:** A centralized, atomic "Prime Number Service" (e.g., using a Redis INCR command against a pre-computed list of primes) will be used to dispense unique primes, guaranteeing no collisions across the distributed system.

- **IDOccurrence Entity:**

- **Definition:** An IDOccurrence represents an ID appearing in a specific role within a specific context. It is the "instance" of an idea in action. For example, in the statement (Alice, worksFor, Google), "Alice" is not just her canonical ID; she is an IDOccurrence playing the *subject* role.
- **Structure:** It contains the ID of the entity itself (`occurringId`), and a reference to the context in which it appears (`context`, which is itself an IDOccurrence representing the statement).

- **Prime Number Embeddings & Similarity:**

The document mentions "embeddings," but in this model, it refers to a set of prime numbers, not a dense vector from an LLM. This leverages the Fundamental Theorem of Arithmetic, as alluded to in John Sowa's work referenced in the source document.

- **Composition:** An IDOccurrence's embedding is a set of primeIDs that define its complete context: {`primeID_of_self`, `primeID_of_predicate`, `primeID_of_object`, `primeID_of_statement_context`}.
- **Similarity Calculation:** Similarity between two IDOccurrences is calculated using a **Jaccard Index** on their prime embedding sets. A high score signifies a high degree of shared context, implying semantic similarity. This is computationally cheaper and more deterministic than vector cosine similarity.

- **Model Statements:**

- **Data Statements:** A raw triple (Subject, Predicate, Object) is transformed into

Statement<IDOccurrence, ID, IDOccurrence>. This captures that the subject and object are specific occurrences, while the predicate is the canonical relationship ID.

- **Schema Statements:** A schema statement like (Person, hasName, String) is represented as Statement<ID, ID, ID>, as it describes relationships between canonical concepts, not specific instances.

2.1. Deep Dive: Formal Concept Analysis (FCA) in the Aggregation Service

FCA is a mathematical method used to find conceptual structures in data. It is a cornerstone of the **Aggregation Service** for inferring types, hierarchies, and hidden relationships. We will use the fcalib library (as cited in the references) within our reactive pipeline. The service will construct three different kinds of formal contexts from the incoming stream of Statement<ID, ID, ID, ID>.

A formal context is a triplet (G, M, I) where G is a set of objects, M is a set of attributes, and I is a binary relation $I \subseteq G \times M$.

1. Predicate-as-Context Analysis:

- **Context:** For a given predicate P (e.g., worksFor), the formal context is (Subjects, Objects, I), where I contains a pair (s, o) if the statement (s, P, o) exists.
- **Example:** Given statements (Alice, worksFor, Google), (Bob, worksFor, Google), (Alice, worksFor, StartupX).
 - G (Objects/Subjects): {Alice, Bob}
 - M (Attributes/Objects): {Google, StartupX}
 - I (Relation): {(Alice, Google), (Bob, Google), (Alice, StartupX)}
- **Inference:** The resulting concept lattice will group employees by their employers. It allows for **attribute implication**. For example, the lattice might reveal that "every person who worksFor both Google and StartupX also has the attribute isSeniorDeveloper". This discovers implicit rules in the data. This aligns with FCA's use in ontology alignment as described in "Aligning Ontologies through Formal Concept Analysis".

2. Subject-as-Context Analysis:

- **Context:** For a given subject S, the formal context is (Predicates, Objects, I).
- **Example:** Given (Alice, title, "Engineer"), (Alice, uses, Java).
 - G (Objects/Predicates): {title, uses}
 - M (Attributes/Objects): {"Engineer", Java}
 - I (Relation): {(title, "Engineer"), (uses, Java)}
- **Inference:** This helps define what a subject *is*. By comparing the concept lattices of different subjects (e.g., Alice vs. Bob), we can find similarities in

their attributes and thus establish a "type" hierarchy. Subjects with similar lattices belong to the same inferred type.

3. Object-as-Context Analysis:

- **Context:** For a given object O, the formal context is (Subjects, Predicates, I).
- **Example:** Given (Alice, uses, Java), (ProjectX, builtWith, Java).
 - G (Objects/Subjects): {Alice, ProjectX}
 - M (Attributes/Predicates): {uses, builtWith}
 - I (Relation): {(Alice, uses), (ProjectX, builtWith)}
- **Inference:** This helps understand the different roles an entity plays. The lattice for "Java" reveals all the subjects that interact with it and the ways (predicates) they do so, defining its role in the ecosystem.

2.2. Deep Dive: The Graph Model (CSPO) in the Alignment Service

The **Alignment Service** elevates the Reference Model to the Graph Model, which reifies statements into higher-order concepts called Kinds. This is where the system's understanding of the domain truly takes shape.

- **CSPO Entities (Context, Subject, Predicate, Object):**

- These are not mere identifiers; they are rich, first-class objects in the model. In a Java implementation, they would be records or classes.
- **Data vs. Schema Statements:** The model supports two parallel universes:
 1. **Data (Instance) Statements:** Statement<Context, Subject, Predicate, Object>. Here, each element is an *instance*. Context is a specific transaction ID (e.g., tx:9987), Subject is user:123, Predicate is the specific purchase event, and Object is product:456.
 2. **Schema (Type) Statements:** Statement<ContextKind, SubjectKind, PredicateKind, ObjectKind>. Here, each element is a *type*. ContextKind is PurchaseEvent, SubjectKind is Customer, PredicateKind is Purchases, and ObjectKind is Product.

- **Domains Modeling & Traversal:**

- A domain (e.g., "E-commerce," "Human Resources") is modeled as a collection of related Kinds. The schema statements form a "type graph" that defines the rules of the domain.
- **Traversal** becomes type-based. Instead of just "find connected nodes," we can ask, "Find all ContextKinds that the Customer SubjectKind can participate in." This is a query on the schema graph, providing powerful domain discovery capabilities.
- **Implementation:** This is best implemented in a property graph database like **Neo4j**. We would use a dual-schema approach:
 - Instance nodes: (u:Instance:User {id: 'user:123'})

- Type nodes: (c:Kind:Customer {name: 'Customer'})
- Connection: (u)-[:INSTANCE_OF]->(c)
This allows a single Cypher query to traverse both instance and type information seamlessly.

- **Functional Operations & Inference:**

- The model enables powerful, type-safe functional operations on reactive streams.
 - Function<Subject, Flux<Context>>: "Given a specific user instance, stream all transaction contexts they participated in."
 - Function<SubjectKind, Flux<PredicateKind>>: "Given the Customer type, stream all possible action types they can perform."
- **Inference:** The primary inference is **validation**. "Can user:123 (who is a Customer) perform a deleteAccount action?" The system validates this by querying the schema graph: MATCH (:Customer)-[:CAN_PERFORM]->(:AccountDeletionAction). If a path exists, the operation is valid according to the learned domain rules.

2.3. Deep Dive: Dimensional Features & Order Inference

The **Alignment Service** is also responsible for inferring order, which is crucial for understanding processes, trends, and state transitions. Order is not explicit; it's inferred from hierarchies in type and state.

- **The Dimensional Model:** Order is understood along specific Dimensions (e.g., Time, Price, Complexity). A measurement is a tuple: (Dimension, Unit, Value). Multidimensional features (OLAP like):
Dimensions: Time, Product, Region.
Units: Month / Year, Category / Item, State / City.
Context : (Context, Attribute, Value).

Examples:

(soldDate, aProduct, aDate)

((soldDate, aProduct, aDate), Product, aProduct)

((soldDate, aProduct, aDate), Product, aProduct), Region, aRegion)

TODO: Materialize / Query Cubes Context Statements into graph models.

- **Order from Type/Schema Hierarchies:**

- **Principle:** Generality defines order. More general concepts come "before" more specific ones.
- **Mechanism:** The type hierarchy is derived from the FCA lattice. A type with more attributes (a more specific concept) is considered "after" a type with fewer attributes.

- **Example:**
 - Person Type: Attributes {hasName, hasAge}.
 - Employee Type: Attributes {hasName, hasAge, hasEmployeeID, hasSalary}.
 - **Inference:** The attribute set of Employee is a superset of Person's. The system establishes a conceptual ordering: Person < Employee. This means Person is a prerequisite concept to Employee.
- **Order from State/Data Hierarchies:**
 - **Principle:** States in a process follow a defined sequence.
 - **Mechanism:** For certain attributes (e.g., "OrderStatus"), a state transition graph can be defined or learned.
 - **Example:** A Flux<OrderEvent> stream is processed. The system sees an order transition through states: Placed -> Paid -> Shipped -> Delivered. This sequence is materialized as an ordered relationship in the graph, allowing the system to infer that Paid comes "after" Placed.

2.4. Deep Dive: Property Graphs as a Unifying Implementation

While the Reference, Graph, and Activation models are conceptually distinct layers of abstraction, they can be elegantly implemented on a **single underlying property graph database (Neo4j)**. This avoids data duplication and enables powerful, cross-layer queries.

- **Unified Node Strategy:** A single node in the graph can represent an entity across all models, distinguished by labels and properties.
 - **Example Node (n):**
 - id: 'user:123'
 - labels: ['Resource', 'Instance', 'User', 'Actor']
 - **Reference Model Properties:** primelid: 8675309, uri: 'db://users/123'
 - **Graph Model Relationships:** (n)-[:INSTANCE_OF]->(:Kind:Customer)
 - **Activation Model Relationships:** (interaction:Interaction)-[:PLAYS_ROLE {roleName: "Buyer"}]->(n)
- **Benefits for Traversal, Inference, and Functional Operations:**
 - **Traversal:** This unified graph allows for seamless traversal across abstraction layers in a single, powerful Cypher query.
 - *Example Query:* "Find all Interactions of type HighValuePurchase where the Buyer Role was played by an Actor of Kind PremiumCustomer, and then find other Actors of the same Kind who have a high Reference Model prime-set similarity to the original buyer."
 - **Inference:** Inferences become multi-layered and incredibly rich. The system can reason about an Activation Interaction based on the Graph Kind of its actors and their Reference primelID similarity to other entities.

- **Functional Operations:** The unified graph enables powerful functional compositions.
 - `Function<Interaction, Flux<ReferenceModelSimilarityScore>>`: "Given a specific transaction instance, calculate the Reference Model similarity scores for all its participating actors." This function would trigger a Cypher query that traverses from the `:Interaction` node to its `:Actor` nodes and then computes the Jaccard index on their `primeId` context sets.

2.2. Deep Dive: Formal Concept Analysis (FCA) with Prime IDs

FCA is a cornerstone of the **Aggregation Service** for inferring types and hierarchies. Using `primeIDs` as the identifiers for objects and attributes in the FCA context provides unique mathematical properties for inference.

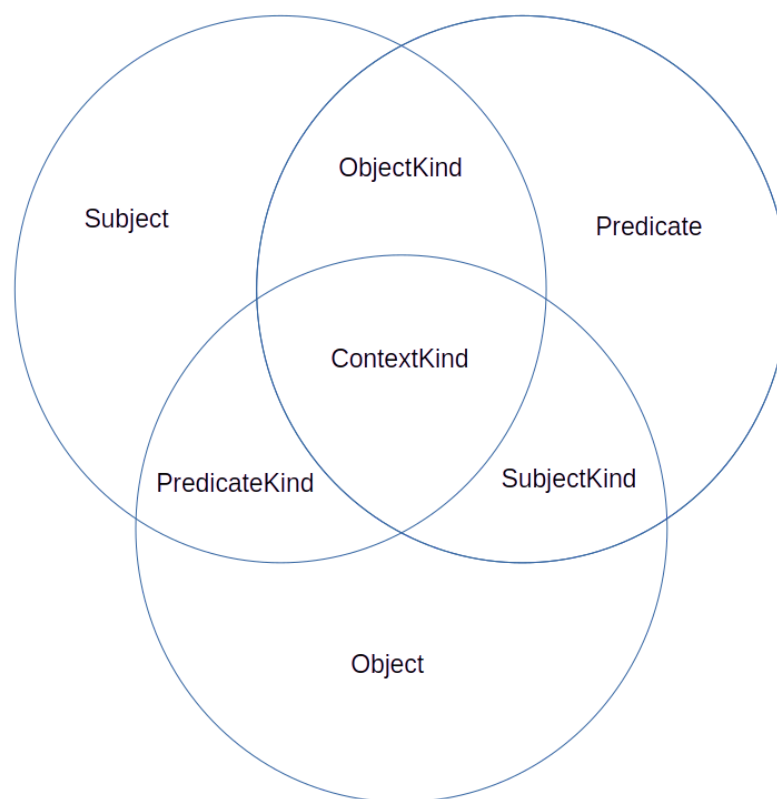
- **Context Construction with Primes:** The formal context (G, M, I) is built as follows:
 - G (Objects): A set of `primeIDs` representing the subjects of a set of statements.
 - M (Attributes): A set of `primeIDs` representing the objects of those statements.
 - I (Relation): A binary relation connecting a subject's `primeID` to an object's `primeID`.
- **Inference via Prime Products:** This is the model's key innovation. A "formal concept" in the resulting lattice is a pair (A, B), where A is a set of subject `primeIDs` (the *extent*) and B is a set of object `primeIDs` they all share (the *intent*).
 1. **Concept Intent Identifier:** For each concept, we can compute a unique identifier for its intent B by **multiplying all the prime IDs** in B. Let's call this the `IntentProduct`. Due to the Fundamental Theorem of Arithmetic, this product is unique to that specific set of attributes.
 2. **Subsumption Inference via Division:** This allows for incredibly efficient hierarchy checking. If we have two concepts, C1 with `IntentProduct1` and C2 with `IntentProduct2`, we can determine if C1 is a sub-concept of C2 (i.e., if all objects in C1's extent are also in C2's) by a simple integer division check. **If `IntentProduct1` is cleanly divisible by `IntentProduct2`, then C2 is a more general concept than C1.**
- **Example:**
 - Concept Vehicle: Intent {hasWheels, canMove} -> Primes {5, 7} -> `IntentProduct` = 35.
 - Concept Car: Intent {hasWheels, canMove, hasEngine} -> Primes {5, 7, 11} -> `IntentProduct` = 385.
 - **Inference:** $385 \% 35 == 0$. The divisibility mathematically proves that Car is a sub-concept of Vehicle without performing any expensive set

operations. This technique, referenced in papers like "Formal Concept Analysis for Knowledge Discovery and Data Mining," makes large-scale hierarchy inference computationally feasible.

2.3. Deep Dive: The Graph Model & Set-Oriented Kinds

The **Alignment Service** elevates the Reference Model to a Graph Model based on set theory, reifying statements into higher-order concepts called Kinds.

- **Visualizing the Model:**



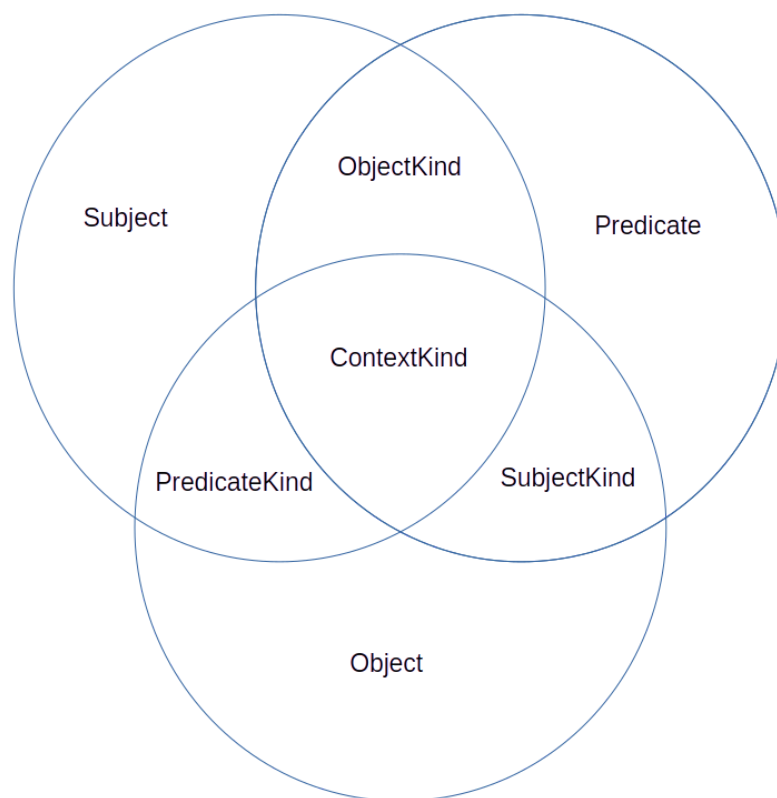
- **Reification & Inference:**
This model enables powerful, type-safe inferences using functional interfaces.
 - **Inference:** Can a Customer (SubjectKind) perform a Return (PredicateKind) on a Service (ObjectKind)? We check if a ContextKind exists at the intersection of these three sets. This validates interactions based on the system's learned knowledge.
 - **Functional Interface:** The logic can be expressed functionally:
 - $\text{Function} \langle \text{SubjectKind}, \text{Set} \langle \text{PredicateKind} \rangle \rangle$: "Given a type of subject, what are all the types of actions it can perform?"

- $\text{Function}\langle \text{PredicateKind}, \text{Tuple}\langle \text{Set}\langle \text{SubjectKind} \rangle, \text{Set}\langle \text{ObjectKind} \rangle \rangle \rangle$:
"Given a type of action, what are the valid types of subjects and objects for it?"

2.2. Deep Dive: The Graph Model & Set-Oriented Kinds in the Alignment Service

The **Alignment Service** consumes the Reference Model and elevates it to a Graph Model based on set theory, as depicted in the source document. This reifies the raw statements into higher-order concepts called Kinds.

- **Visualizing the Model:**



- **Reification Process:**
The service consumes a $\text{Flux}\langle \text{Statement}\langle \text{ID}, \text{ID}, \text{ID}, \text{ID} \rangle \rangle$ and groups statements to build these Kind sets.
 1. **SubjectKind:** A SubjectKind is formed by grouping all Subjects that interact with a similar set of Predicates and Objects. It represents a "type" of subject. For example, all subjects that interact with Predicates like hasOrder and Objects like Product would be reified into the Customer SubjectKind.
 2. **PredicateKind:** A PredicateKind groups Predicates that connect similar

SubjectKinds and ObjectKinds. It represents a "type" of relationship, like Transaction.

3. **ObjectKind:** An ObjectKind groups Objects that are acted upon by similar SubjectKinds via similar PredicateKinds. It represents a "type" of object, like PurchasableItem.
 4. **ContextKind:** This is the intersection of all three, representing a complete, reified event or use case type, like PurchaseEvent.
- **Set-Based Inferences and Functional Interfaces:**
This model enables powerful, type-safe inferences using functional interfaces.
 - **Inference:** We can check for valid interactions. Can a Customer (SubjectKind) perform a Return (PredicateKind) on a Service (ObjectKind)? By checking the set intersections, the system can determine if this is a valid operation.
`IsInteractionValid(s: SubjectKind, p: PredicateKind, o: ObjectKind): boolean.`
 - **Functional Interface:** The core of the alignment logic can be expressed functionally:
 - `Function<SubjectKind, Set<PredicateKind>>`: "Given a type of subject, what are all the types of actions it can perform?"
 - `Function<PredicateKind, Tuple<Set<SubjectKind>, Set<ObjectKind>>>`: "Given a type of action, what are the valid types of subjects and objects for it?"

2.3. Other Components & Reactive Implementation Details:

- **Aggregation Service (Continued):**
 - **Functional Interface:** `Function<Flux<Statement>, Flux<ConceptLattice>>`.
 - **Spring AI (Reactive Embeddings):** Embeddings are generated within the reactive stream using Spring AI's `ReactiveEmbeddingClient`, ensuring network calls to models (e.g., from Hugging Face or a local Ollama instance) are non-blocking.
- **Alignment Service (Continued):**
 - **Functional Interface:** `Function<Flux<ReferenceStatement>, Flux<GraphStatement>>`.
 - **RDF4J Integration:** SPARQL queries via RDF4J will be wrapped in `Mono.fromCallable` and executed on a dedicated scheduler to avoid blocking, as envisioned by concepts in "SPARQL-Micro-Services".
- **Naming Service (Helper Service - Java, Apache Jena):**
 - Provides a reactive SPARQL endpoint by proxying Jena Fuseki with Spring WebFlux, ensuring end-to-end non-blocking I/O.

Phase 3 & 4: Activation, API, and UI

These phases build upon the rich, interconnected knowledge graph established in Phase 2.

- **Activation Service (Java, Spring Boot):** Implements **DDD** and **DCI** patterns. A Role is a reactive Function<Flux<ActorState>, Flux<TransformedState>>, allowing for the dynamic composition of behavior onto data objects (Actors). The key inference is Function<DesiredOutcome, Flux<InteractionPlan>>, which uses reactive graph traversal to find a sequence of Role functions to achieve a goal.
- **Producer Service (API/Frontend - Java/Spring WebFlux, React):** Exposes the framework's capabilities via a fully reactive API using **Server-Sent Events (SSE)** for real-time updates. The React frontend uses **RxJS** to create a responsive UI that is directly bound to these event streams.

Phase 3: Activation & Use Case Enablement (Months 8-10)

Objective: Infer and enable the execution of business processes using the DCI and DDD patterns within a reactive model.

3.1. Components & Reactive Implementation Details:

- **Activation Service (Java, Spring Boot):**
 - **DDD (Domain-Driven Design):** This service is a classic DDD Bounded Context. The Activation Model is its Ubiquitous Language. It consumes AlignmentModelChanged domain events from Kafka and produces InteractionStateChanged events, following principles from Eric Evans' "Domain-Driven Design".
 - **DCI (Data, Context, and Interaction):** This pattern is implemented reactively.
 - **Role (Functional Interface):** A Role is a Function<Flux<ActorState>, Flux<TransformedState>>. It's a functional interface defining the behavior an Actor will perform, a direct implementation of the DCI pattern where Roles are injected into Data objects at runtime, as described in the papers by Trygve Reenskaug and James Coplien.
 - **Interaction:** A stateful, non-blocking orchestrator that subscribes to Actor state streams and applies Role functions to drive the use case forward.
 - **Activation Model Inferences:** Inferences here are pragmatic. The key functional interface is: Function<DesiredOutcome, Flux<InteractionPlan>>. "Given a desired outcome, what sequence of Role functions must be applied to which Actors?"
- **Index Service (Helper Service - Python, Vector DB):**
 - **Reactive Indexing:** Subscribes to a Kafka topic of ResourceUpdated events

and updates a vector database (e.g., Milvus) as embeddings change.

3.1. Components & Reactive Implementation Details:

- **Activation Service (Java, Spring Boot):**
 - **DDD (Domain-Driven Design):** This service is a classic DDD Bounded Context. The Activation Model is its Ubiquitous Language. It consumes AlignmentModelChanged domain events from Kafka and produces InteractionStateChanged events. This follows the principles from Eric Evans' "Domain-Driven Design: Tackling Complexity in the Heart of Software".
 - **DCI (Data, Context, and Interaction):** This pattern is implemented reactively.
 - **Context:** A Context class defines a use case. It contains logic to find required Roles, often via a reactive graph query.
 - **Role (Functional Interface):** A Role is a `Function<Flux<ActorState>, Flux<TransformedState>>`. It's a functional interface defining the behavior an Actor will perform. This is a direct implementation of the DCI pattern where Roles are injected into Data objects at runtime.
 - **Interaction:** An Interaction is a stateful, non-blocking orchestrator. It subscribes to the Flux streams representing its Actors' states and applies the Role functions to drive the use case forward. This dynamic composition is a core idea from the DCI papers by Trygve Reenskaug and James Coplien.
 - **Activation Model Inferences:** Inferences here are pragmatic and goal-oriented. The key functional interface is: `Function<DesiredOutcome, Flux<InteractionPlan>>`. "Given a desired outcome, what sequence of Role functions must be applied to which Actors?" This is solved using reactive graph traversal and constraint satisfaction.
- **Index Service (Helper Service - Python, Vector DB):**
 - **Reactive Indexing:** It will subscribe to a Kafka topic of ResourceUpdated events. Using a reactive Kafka consumer (aiokafka in Python), it will update the vector database (e.g., Milvus) as soon as a resource's embedding changes.

Phase 4: API & User Interface (Months 11-12)

Objective: Expose the framework's capabilities through a fully reactive API and a real-time user interface.

4.1. Components & Reactive Implementation Details:

- **Producer Service (API/Frontend - Java/Spring WebFlux, React):**
 - **Fully Reactive API:** Built with Spring WebFlux.

- **Server-Sent Events (SSE):** For real-time updates on Interactions, the API will use SSE. A client subscribes to an endpoint like GET /v1/interactions/{id}/stream, which returns a Flux<InteractionState>. This is more efficient than WebSockets for server-to-client data pushes, as advocated in "Building Reactive Microservices with Spring WebFlux".
- **Frontend (React with RxJS):** The React frontend will use RxJS to manage the SSE streams, binding component state directly to an Observable so the UI updates automatically. This aligns with the "Thinking in React" and "Thinking in RxJava" mental models.

4.1. Components & Reactive Implementation Details:

- **Producer Service (API/Frontend - Java/Spring WebFlux, React):**
 - **Fully Reactive API:** The entire API will be built with Spring WebFlux.
 - **Server-Sent Events (SSE):** For real-time updates on long-running Interactions, the API will use SSE. A client subscribes to an endpoint like GET /v1/interactions/{id}/stream, which returns a Flux<InteractionState> with the Content-Type of text/event-stream. This is more efficient than WebSockets for server-to-client data pushes, as advocated in "Building Reactive Microservices with Spring WebFlux".
 - **Frontend (React with RxJS):** The React frontend will use a library like RxJS to manage the SSE streams. The state of a component can be directly bound to an Observable derived from the event stream, causing the UI to update automatically as new data arrives. This aligns with the "Thinking in React" and "Thinking in RxJava" mental models.

Phase 4: API & User Interface

- **Producer Service (API/Frontend - Java/Spring WebFlux, React):** The Producer's role is to render the Actor's next set of Transforms as UI elements (e.g., buttons, forms). When a user interacts, the Producer constructs the appropriate Transform message and sends it to the Interaction to drive the state machine forward. It uses **Server-Sent Events (SSE)** to reactively listen for ActorStateUpdated events and update the UI in real-time.

Phase 4: API & User Interface

- **Producer Service (API/Frontend - Java/Spring WebFlux, React):** The Producer's role is to be a fully compliant **COST/HAL client**. It renders the UI based *entirely* on the `_links` section of the responses from the Activation Service. When a user interacts, the Producer constructs the appropriate request based on the link's href, method, and name, driving the state machine forward. It uses **Server-Sent Events (SSE)** to reactively listen for state change notifications and

trigger a re-fetch of the resource to get the new state and available actions.

Appendix 1: Models Architecture

The idea is to enable model representations being equivalent (containing the same data) in various layers to be switched back and forth between each layer representation to be used in the most appropriate task for a given representation.

Reification: Statements could be about any type of URI (URIOccurrence(s)) in which Statements subjects, predicates and objects occurrences plays determinate role (Kind: Type / State) regarding this Statement occurrence context. Statements themselves are URIOccurrence(s) with their URIOccurrence uri being their subject URI, their statement being the statement itself (this) and their URIOccurrence Kind uri being their subject uri, their Kind type its predicate Kind Type and its Kind state being its object Kind State.

Those entities are to be able to be retrieved and their representations should enable functional programming techniques to be applied to streams of their representations to perform Aggregation, Alignment and Activation.

The nodes and arcs of the graph triples are URIs and should have a "retrievable" internal representation with metadata that each service / layer populates through the "helper" services: Registry, Naming (NLP) and Index service shared by each layer. Describe core model classes serialization in JSON.

Materialize. Reification of RDFS / OWL. Ontology Schema Statements. Same as. Schema (alignment) statements materialization.

Reference Model

(Aggregation / Grammar)

ID

- primeID : long
- urn : string
- occurrences : IDOccurrence[]
- embedding : double[]

IDOccurrence : ID

- occurringId : ID
- context : IDOccurrence
- embedding : double[]

Statement : IDOccurrence (Property Graphs)

- context : ID
- subject : ID

- predicate : ID
- object : ID

Statements:

Data: (IDOccurrence(ID), IDOccurrence(ID), IDOccurrence(ID))

Schema: (ID(IDOccurrence), ID(IDOccurrence), ID(IDOccurrence))

Graph Model

(Alignment, Semantics, Sets / Kinds)

Context : IDOccurrence (Set)

Subject : IDOccurrence (Set)

Predicate : IDOccurrence (Set)

Object : IDOccurrence (Set)

Interface Kind<AttributeType, ValueType>

- superKind : Kind
- attributeValues : Tuple<AttributeType, ValueType>[]

Reification: Kind implementations extends / plays Subject, Predicate and Object roles in statement.

SubjectKind : extends Subject, implements Kind<Predicate, Object> (Predicates intersection Objects)

- occurrences : Subject[]

PredicateKind : extends Predicate, implements Kind<Subject, Object> (Subjects intersection Objects)

- occurrences : Predicate[]

ObjectKind : extends Object, implements Kind<Predicate, Subject> (Predicates intersection Subjects)

- occurrences : Object[]

The underlying model Statements can be represented as sets being Subjects, Predicates and Objects three sets where the intersection of Predicates and Objects sets conforms the "Subject Kinds" set, the intersection of the Subjects and Objects sets conforms the "Predicate Kinds" set, the intersection of the Subjects and Predicates sets conforms the "Object Kinds" set and the intersection of the three sets conforms the "Statements" set.

Sets based inference and functional algorithms should leverage this form of

representation of the model graph.

Statements:

Data: Context(Subject, Predicate, Object)

Schema: Context(SubjectKind, PredicateKind, ObjectKind)

Activation Model

(Activation, DOM / DCI / Actor, Role. Pragmatics)

DOM (Dynamic Object Model):

Instance : IDOccurrence

- id : ID
- label : string
- class : Class
- attributes : Map<string, Instance>

Class : Instance

- id : ID
- label : string
- fields : Map<string, Class>

DCI (Data, Context, Interaction):

Context

- roles : Role[]

Role : Class

- previous : Map<Context, Dataflow>
- current : Map<Context, Dataflow>
- next : Map<Context, Dataflow>

Dataflow : Context

- role : Role
- rule : Rule (TODO)

Interaction

- actors : Actor[]

Actor / Role Pattern:

Actor : Instance

- previous : Map<Context, Transform>
- current : Map<Context, Transform>
- next : Map<Context, Transform>

Transform

- actor : Actor
- production : Production (TODO)

Statements:

Data: (Interaction, Actor, Transform)

Schema: (Context, Role, Dataflow)