Implementation Roadmap: Application Service Framework

Version: 1.5 **Date:** July 28, 2025

1. Introduction

1.1. Purpose and Scope

This document provides a comprehensive, implementation-focused roadmap for the Application Service framework. Its goal is to allow the integration of diverse existing and legacy applications or API services by parsing their backend data, inferring semantic models, and representing application behaviors into an activatable, use-case-driven model. It breaks down each development phase into specific technical tasks, architectural decisions, and technology choices, offering a guide for building a system where a resource's capabilities are as important as its data.

1.2. Architectural Vision: The Reactive, Semantic, and Agentic Enterprise

The framework's architecture is founded on a significant evolution towards a fully reactive, functional, and behavior-driven system. The core vision is to transform raw data from disparate sources into an interconnected, semantically rich knowledge graph. This graph doesn't just store data; it models behavior. The system infers and enables the execution of business processes through a dynamic, message-driven model based on principles like Data, Context, and Interaction (DCI) and Domain-Driven Design (DDD).

Ultimately, this allows for the creation of a generic "use case browser" client or API that can browse and execute contexts and interactions within and between the integrated applications. This vision culminates in an intelligent, agentic system capable of communicating with Large Language Models (LLMs) and other framework instances using standardized, modern protocols.

1.3. Core Principles

- **Reactive and Event-Driven**: The entire architecture is built on a non-blocking, event-driven dataflow. It uses reactive streams (leveraging frameworks like Spring WebFlux and Project Reactor) to ensure scalability and resilience from data ingestion to API responses.
- **Semantic Knowledge Representation**: Data is processed through progressively richer conceptual models, moving from raw triples to a formal, mathematically grounded identification system and finally to a semantically rich graph that includes concepts, relationships, and behavioral content types.

- **Behavior-Driven and Agentic**: The framework moves beyond data orchestration to pragmatic inference—reasoning about goals and capabilities. Interactions are inferred from and driven by the "Content Types" of resources, which define their behavioral essence (e.g., `buy-able`, `trackable`).
- **Layered Abstraction**: The framework is organized into distinct services (Datasource, Aggregation, Alignment, Activation) that correspond to layers of data abstraction (Reference, Graph, Activation models). This separation of concerns allows for equivalent data representations to be used in the most appropriate task for that representation.

---

2. Overall Architecture

2.1. Service-Oriented Layout

The framework is composed of a set of collaborating microservices, each with a distinct responsibility. An Augmentation Service orchestrates the flow of data between the core layers, which are supported by several helper services.

- **Core Services**:
  - **Datasource Service**: Responsible for ETL, synchronization with backend datasources, and producing raw data streams.
  - **Aggregation Service**: Consumes raw data and produces the Reference Model, performing type/state inference and assigning formal identities.
  - **Alignment Service**: Consumes the Reference Model and produces the semantically enriched Graph Model, handling ontology matching, link completion, and order inference.
  - **Activation Service**: Consumes the Graph Model to infer and execute business processes (use cases) as part of the Activation Model.
- **Orchestration**:
  - **Augmentation Service**: The reactive backbone that manages the flow of messages between all services using the Saga pattern for distributed transactions.
- **Client-Facing**:
  - **Producer Service**: Provides the public-facing API and a generic frontend for Browse and executing use cases inferred by the Activation Service.
- **Helper Services**:
  - **Registry Service**: A central repository for the unified property graph (implemented on Neo4j), storing the state of resources across all models.
  - **Naming Service**: Manages ontologies, providing alignment and matching capabilities, often through a SPARQL endpoint.
  - **Index Service**: A repository for aligned resources, providing similarity search capabilities, often using vector databases.

2.2. The Layered Model Architecture

Data is processed through three conceptually distinct, progressively richer models. These can be implemented on a single underlying property graph, where a node's state in each model is represented by labels and properties.

- **Reference Model (Grammar & Identity)**: Focuses on formal, mathematically grounded identification.
    - **Entities**: `ID` (the canonical concept of an entity, identified by a unique `primeID`) and `IDOccurrence` (an ID appearing in a specific context). Statements in this model take the form `Statement<ID, ID, ID, ID>`.
- **Graph Model (Semantics & Relationships)**: Focuses on relationships, types, and semantics by reifying statements into higher-order concepts called `Kinds`. This model is based on set theory, where Kinds are intersections of Subject, Predicate, and Object sets.
    - **Entities**: `Context`, `Subject`, `Predicate`, `Object` as sets of IDOccurrences. It supports both Data Statements `Statement<Context, Subject, Predicate, Object>` and Schema Statements `Statement<ContextKind, SubjectKind, PredicateKind, ObjectKind>`.
- **Activation Model (Pragmatics & Behavior)**: Focuses on behavior, state, and use case execution. It is a Dynamic Object Model (DOM) where an object's capabilities can change at runtime.
    - **Entities**: Implements the DCI pattern with `Context` (a use case), `Role` (a behavior), and `Interaction` (an instance of a Context). Actors are data `Instances` that are cast into Roles. Dataflows are sequences of declarative `Transform` messages that drive the state machine.

2.3. Communication and Dataflow

- **Message Bus**: Apache Kafka serves as the central message bus, ensuring a decoupled and scalable event-driven architecture. All message types use versioned Avro schemas for compatibility. Key topics include `datasource-raw-triples-v1`, `aggregation-reference-model-v1`, and `alignment-graph-model-v1`.
- **API Protocol**: The primary API protocol is **COST (COnversational State Transfer)**, a stateful, conversational approach built on HATEOAS principles using the **HAL (Hypertext Application Language)** specification. Every API response includes links to all possible next actions, allowing the client to be dynamic and resilient to backend changes.
- **Agentic Protocol**: For communication between framework instances and with LLMs, the **Model Context Protocol (MCP)** is used. This allows external clients to interact with the framework's capabilities (Resources, Tools, Prompt Templates) in a standardized way.
- **Identity Protocol**: **W3C Decentralized Identifiers (DIDs)** are used as the canonical identifier for all resources, providing a foundation for verifiable provenance and

secure, interoperable communication between services without traditional credentials.

---

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

3.1. Objective

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

3.2. Components & Implementation Details

Datasource Service (Java, Spring WebFlux)

- **Core Logic**: Built on a non-blocking stack using Spring WebFlux. It implements a `DataSourceAdapter` interface for various data source types.
- **Reactive Ingestion Adapters**:
  - **JCA Integration**: The primary mechanism for enterprise systems (e.g., ERPs, SCMs) is the Java EE Connector Architecture (JCA) 1.7. A custom `JcaResourceAdapter` handles both inbound event listening (via `MessageEndpointFactory`) and outbound transactional write-backs (via `ConnectionFactory`). This is crucial for synchronizing state back to source systems.
  - **RestApiAdapter**: Uses `spring-webflux`'s non-blocking `WebClient` to consume external APIs, natively returning a `Flux<T>` and handling pagination seamlessly.
  - **R2DBCAdapter**: For supported SQL databases, it uses R2DBC (`spring-boot-starter-data-r2dbc`) to perform non-blocking database queries, returning a `Flux<Row>`.
  - **FileAdapter**: Uses the Jackson library to parse JSON/XML files and watches a designated directory for changes.
- **Transformation & Provenance**: The core logic is a pure function within a reactive pipeline that transforms source entities into SPO triples. A provenance context (e.g., source application name) is added to each triple. For a database row, the output message would be `("product:123", "hasName", "Laptop", "source:db1")`.

Augmentation Service (Java, Spring Cloud Stream)

- **Core Logic**: This service is the reactive backbone, built using Spring Cloud Stream's functional programming model. It orchestrates the message flow by binding `java.util.function.Function` beans to Kafka topics for a fully event-driven dataflow.

- **Saga Pattern**: Implements the Saga pattern for long-running, distributed transactions. It listens for completion events (e.g., `RAW_TRIPLE_INGESTED`) to trigger the next service in the chain, logging state transitions and compensating actions to a dedicated Kafka topic.
- **Resiliency**: Uses Spring Retry for handling transient failures and a dead-letter queue (DLQ) for messages that repeatedly fail.

Registry Service (Java, Spring Boot, Neo4j)

- **Core Logic**: The central repository for the unified property graph, implemented on Neo4j. It provides a reactive REST API built with Spring WebFlux functional endpoints.
- **Database Schema**: Nodes have a `:Resource` label and an indexed `uri` property. Relationships represent predicates. The graph stores detailed JCA provenance for each resource, such as `connectorId`, enabling the system to know which adapter to use for write-backs.
- **Reactive API**:
  - `POST /v1/graph/statements`: A batch endpoint that uses an optimized Cypher `UNWIND ... MERGE` query for high-performance writes.
  - To keep the API non-blocking, blocking driver calls are offloaded to a dedicated scheduler (`Schedulers.boundedElastic()`), a core tenet of reactive programming.

---

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

4.1. Objective

Transform raw, ingested data into an interconnected, semantically rich knowledge graph where resources are defined by their data and their behaviors (Content Types).

4.2. Deep Dives: Foundational Concepts

The Reference Model & Prime Number Semantics

This model moves from simple string identifiers to a formal system.

- **ID & IDOccurrence**: An `ID` is the canonical concept of an entity, assigned a unique, immutable `primeID` upon first encounter. An `IDOccurrence` is an `ID` appearing in a specific role or context.
- **Prime Number Semantics**: Leveraging the Fundamental Theorem of Arithmetic, a concept's context can be represented by a set of `primeIDs`. Similarity between two concepts can be calculated with a deterministic Jaccard Index on these sets.

Formal Concept Analysis (FCA) with Prime IDs

FCA is a mathematical method used to find conceptual structures in data, forming a cornerstone of the Aggregation Service for inferring type hierarchies.

- **FCA with Primes**: We use `primeIDs` for objects and attributes in the formal context. A concept's "intent" (its shared attributes) can be uniquely identified by the product of its attribute `primeIDs`.
- **Inference via Prime Products**: This is a key innovation. Subsumption checking (`is-a` relationships) becomes a simple integer division operation: 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 hyper-efficient arithmetic, making large-scale hierarchy inference computationally feasible. We will use a library like **fcalib**.

The Graph Model & Set-Oriented Kinds

The Alignment Service elevates the Reference Model to the Graph Model, which reifies statements into higher-order concepts called Kinds.

- **Visualizing the Model**: The model is based on set theory, where `SubjectKind`, `PredicateKind`, and `ObjectKind` are intersections of the base `Subject`, `Predicate`, and `Object` sets. `ContextKind` is the intersection of all three.
- **Set-Based Inference**: This model enables powerful, type-safe inferences. To validate if a `Customer` (SubjectKind) can perform a `Return` (PredicateKind) on a `Service` (ObjectKind), the system checks if a `ContextKind` exists at the intersection of these three sets. This can be expressed with functional interfaces like `Function<SubjectKind, Set<PredicateKind>>` ("what actions can this type of subject perform?").

Dimensional Features & Order Inference

The Alignment Service is responsible for inferring order, which is crucial for understanding processes and state transitions.

- **Order from Hierarchies**: Order is inferred from type and state hierarchies established by the Aggregation Service. For example, `Person` → `Employee` implies that `Person` is a prerequisite concept, establishing an order. Similarly, state transitions like `Placed` → `Paid` → `Shipped` are materialized as ordered relationships in the graph.
- **Dimensional Model**: This OLAP-like model provides analytics directly on the graph. Measures (like `salesValue`) are captured in `ContextMeasure` nodes, which are then sliced by

dimensions (`Time`, `Region`, `Product`) through a series of `IS_SLICE_OF` relationships, creating queryable paths in the graph.

4.3. Components & Implementation Details

Aggregation Service (Java, Spring AI, Python, fcalib)

- **Core Logic**: Consumes raw triples from `datasource-raw-triples-v1` and performs type and state inference. Subjects with similar predicate sets are grouped into inferred types.
- **ID & Embedding Generation**: For each new URI, it generates a unique `primeID` and a vector embedding. This can be a separate Python service using models like **Sentence-BERT** from the Hugging Face library, called via RPC. Within the reactive stream,
  Spring AI's `ReactiveEmbeddingClient` will be used to generate embeddings without blocking.
- **FCA Implementation**: Uses the **fcalib** Java library to create formal contexts. The resulting concept lattice directly forms the
  `is-a` type hierarchy.

Alignment Service (Java, RDF4J)

- **Core Logic**: Consumes from `aggregation-reference-model-v1` and performs deep semantic inference.
- **Ontology Matching**: Uses the **RDF4J** framework's `MemoryStore` for in-memory graph operations. It loads pre-defined upper ontologies (e.g., Schema.org) and uses the SPARQL engine with SHACL rules to find and materialize equivalences (`owl:sameAs`, `rdfs:subClassOf`).
- **Inference & Materialization**:
  - **Link Completion**: Uses SPARQL `CONSTRUCT` queries to infer and create new links based on graph patterns and logical rules (e.g., transitivity).
  - **Attribute Closure**: Discovers domain-specific entailments. For example, the pattern

    `(:Developer)-[:WORKS_ON]->(:Project)-[:USES_LANGUAGE]->(:Language)` entails a new link: `(:Developer)-[:KNOWS_LANGUAGE]->(:Language)`.
  - **Content-Type Inference**: Infers a resource's behavioral essence (e.g., `buy-able`) by analyzing graph patterns and the capabilities of its underlying JCA resource adapter. These are stored as a list property on the resource's node.

Naming Service (Java, Apache Jena)

- **Core Logic**: A dedicated helper service for managing ontologies.
- **Storage & API**: Uses **Apache Jena** with a TDB2 persistent backend and exposes a full SPARQL 1.1 endpoint via **Jena Fuseki**. To ensure end-to-end reactivity, this can

be proxied by a Spring WebFlux application. It also provides custom REST endpoints for alignment tasks.

---

5. Phase 3: Activation & Behavior-Driven Interactions (Months 8-10)

5.1. Objective

Implement a dynamic, behavior-driven activation layer where interactions are inferred from and driven by the Content Types of the participating resources, enabling the execution of business processes.

5.2. Deep Dives: Foundational Concepts

The Activation Model: DCI, DOM, and Stateful Dataflows

The runtime logic is a direct implementation of modern software design patterns.

- **DCI (Data, Context, and Interaction)**: This is the blueprint for the runtime logic. An `Interaction` (the Context) "casts" plain data `Instances` into `Actors` by dynamically injecting `Roles` (behavior) for the duration of a use case. This avoids bloating data objects with all possible behaviors.
- **Dynamic Object Model (DOM)**: The Activation Model is a DOM where an object's capabilities can change at runtime, built on the Actor-Role pattern. An Actor's state is its position in the use case flow, modeled as available `Transforms` (previous, current, next).
- **Stateful Dataflow**: The entire process is driven by a reactive stream of declarative `Transform` messages. These are data objects that instruct an Actor on how to mutate its internal state, creating a highly scalable and auditable distributed state machine.

JAF-Inspired Semantic Engine

The Activation Service operates like a distributed, semantic JavaBeans Activation Framework (JAF).

- **ContentTypeDataHandler**: For each defined `Content-Type` (e.g., `buy-able`), a corresponding Spring bean implementing this interface is registered. This pluggable handler defines the available `Verbs` (e.g., `BUY`) and the `Dataflow` for each verb.
- **JCA for Transactional Write-Back**: When a dataflow completes, the final `Transform` is processed by the relevant `ContentTypeDataHandler`. The handler retrieves the resource's JCA provenance from the Registry, obtains a JCA

`Connection` from the Datasource Service, and invokes the outbound transaction on the backend system (e.g., an ERP), guaranteeing data consistency.

5.3. Components & Implementation Details

Activation Service (Java, Spring Boot)

- **Core Logic**: Implements the DCI and DDD patterns within a reactive model. It is a single Bounded Context where the Activation Model is its Ubiquitous Language. It consumes `AlignmentModelChanged` events and produces `InteractionStateChanged` events.
- **Context Inference**: Uses graph traversal algorithms or Cypher queries to find recurring patterns that represent potential use cases (Contexts). For example, a pattern of `(Order)-[contains]->(Product)<-[trackedIn]-(Inventory)` infers a `ReplenishStock` Context.
- **Role Implementation**: A Role is implemented as a reactive function: `Function<Flux<ActorState>, Flux<TransformedState>>`. This allows for the dynamic composition of behavior onto data objects (Actors).
- **Interaction Implementation**: An Interaction is a stateful, non-blocking orchestrator that subscribes to the `Flux` streams of its Actors' states and applies Role functions to drive the dataflow forward.

---

6. Phase 3.5: LLM Integration & Agentic Architecture (Parallel)

6.1. Objective

Elevate the Activation Service from a simple orchestrator to an intelligent, agentic system capable of communicating with LLMs and other ApplicationService instances using standardized protocols.

6.2. Deep Dives: Foundational Concepts

- **Model Context Protocol (MCP)**: A protocol that allows clients (like LLM agents) to discover and interact with a server's capabilities (its "context") in a standardized way. The ApplicationService will act as an MCP Server.
- **COST (COnversational State Transfer) / HAL**: A stateful API paradigm where each response guides the client by providing links to all possible next actions. This is achieved by embedding placeholders within HAL `_links`, creating a template for the next `Transform` message the client should send. This makes the client dynamic and decoupled from the server's business logic.

- **W3C Decentralized Identifiers (DIDs)**: Universal, decentralized identifiers that give resources a secure, verifiable identity independent of any central registry. They are used for verifiable provenance (signing statements) and secure, password-less authentication between services (DID-Auth).

6.3. Components & Implementation Details

MCP Server Implementation (in Activation Service via Spring AI)

- **Core Logic**: A single REST endpoint (`/mcp`) will be implemented using Spring WebFlux to handle all MCP requests.
  Spring AI will be the primary tool to bridge internal services with the LLM world.
- **Exposed Capabilities**:
  1. **Resources**: An MCP client can request resources (e.g., "find me senior Java developers"). The request is routed to the Index Service, using `Spring AI's ReactiveEmbeddingClient` to convert text to a vector for similarity search.
  2. **Tools**: An MCP client can request to use a tool (e.g., "execute OnboardNewEmployee tool"). This maps the tool name to an Activation Context, instantiates an Interaction, and executes its dataflow. The LLM decides
     what to do; the framework provides the verifiable, stateful Tool to do it.
  3. **Prompt Templates**: An MCP client can request a structured prompt template from the Naming Service. The client populates it and uses `Spring AI's ReactiveChatClient` to query an LLM. The response can be fed back into the system to augment the graph.

DID Integration (did-common-java)

- **Implementation**:
  1. **Creation**: When a new resource is discovered, the Aggregation Service uses a library like **did-common-java** to generate a DID (e.g., `did:ion` or `did:key`).
  2. **Storage**: The generated DID Document (containing cryptographic keys and service endpoints) is stored in the Registry Service's property graph, linked to the resource node.
  3. **Usage**: The resource's DID becomes its canonical identifier throughout the system.

---

7. Phase 4: The Behavior-Driven API & UI (Months 11-12)

7.1. Objective

Expose the framework's dynamic capabilities through a standardized agentic protocol and a user interface that is entirely driven by the available behaviors of its resources.

7.2. Components & Implementation Details

Producer Service (Java/Spring WebFlux, React/RxJS)

- **Backend API**: A fully reactive public-facing interface built with **Spring WebFlux**. It implements the COST/HAL protocol, with HATEOAS responses guiding the client through interactions.
- **Real-time Updates with SSE**: For long-running interactions, the API will use **Server-Sent Events (SSE)**. A client can subscribe to an endpoint like `GET /v1/interactions/{id}/stream` which returns a `Flux<InteractionState>` with a `Content-Type` of `text/event-stream`, providing real-time updates.
- **Frontend Architecture**: A Single-Page Application (SPA) built with **React** and **TypeScript**. It will use a component library like Material-UI for consistency. Crucially, it will use **RxJS** to manage the SSE streams, binding UI component state directly to an `Observable` so the UI updates automatically and reactively as new events arrive from the server.

Index Service (Python, Vector DB)

- **Core Logic**: A helper service that provides fast similarity search and dimensional query capabilities. It will likely be implemented in Python to leverage its rich ecosystem of data science and vector database libraries.
- **Reactive Indexing**: Subscribes to a Kafka topic of `ResourceUpdated` events using a reactive consumer (like `aiokafka` in Python). As soon as a resource's embedding changes, it updates the vector database (e.g., **Milvus**, **Pinecone**, or **Weaviate**).
- **API**: Provides an API for two main functions:
    1. `POST /v1/similarity/search`: Accepts a vector and returns the DIDs of similar resources.
    2. `POST /v1/dimensional/query`: Accepts a declarative OLAP-style request and translates it into a complex traversal query against the graph to provide real-time BI analytics.

Appendix A: Business Intelligence & Analytics Layer

The true value of the framework is realized when the unified, contextualized data is leveraged for analytics.

- **Architecture**: While the property graph is excellent for operational queries, traditional BI tools work best with star schemas. A BI layer can be implemented via a periodic

ETL process that runs Cypher queries against the Neo4j graph to extract and flatten the aligned data into a classic dimensional data warehouse (e.g., **PostgreSQL**, **BigQuery**, or **Snowflake**).

- **Example KPIs**:
  - **"Cross-System Customer LTV"**: Since all interactions are unified around a canonical customer DID, combining purchasing, marketing, and support data to calculate a true Lifetime Value becomes trivial.
  - **"Process Flow Analysis"**: By analyzing the materialized state transition graphs (e.g., `Placed` -> `Paid` -> `Shipped`), analysts can create reports on process bottlenecks.
  - **"Product Concept Affinity"**: By analyzing the FCA-derived concept lattices, analysts can discover non-obvious relationships, such as an affinity between `ProductCategory:OutdoorGear` and `CustomerAttribute:OwnsDog`, suggesting new marketing strategies.

Appendix B: End-to-End Integration Use Case: Federated Supply Chain

This use case demonstrates how three independent entities (a Retailer, a Manufacturer, and a Raw Materials Supplier) can form a seamless, automated supply chain using the framework.

1. **Low Inventory Trigger**: The Retailer's `Datasource` service ingests low inventory levels from its ERP. The `Activation` service's `MonitorInventory` context detects this and triggers a `ReplenishStock` interaction.
2. **Automated Order Placement**: The Retailer's service acts as an MCP Client, discovering the Manufacturer's "Purchase" tool via its MCP Server. It sends a standardized `ToolCall` message to place a new order.
3. **Material Check & Procurement**: The Manufacturer's `FulfillOrder` interaction queries its own graph and finds that raw material stock is insufficient. This triggers an internal `ProcureMaterials` interaction, which identifies the Raw Materials Supplier and prepares to act as an MCP Client itself.
4. **Federated BI**: Each participant can perform internal BI. Furthermore, if they agree to share certain `ContextMeasure` data, they can build a federated view of the entire supply chain's health, analyzing end-to-end efficiency without exposing sensitive internal data.

Appendix C: Tools, Frameworks, and Further Reading

This section consolidates the extensive list of technologies and concepts referenced in the source documents.

- **Core Frameworks**:
    - **Spring Ecosystem**: Spring Boot, Spring WebFlux, Spring Cloud Stream, Spring Data (JDBC, Neo4j, R2DBC), Spring Security, Spring AI.
    - **Reactive Programming**: Project Reactor, RxJava.
- **Semantic Technologies**:
    - **Graph Database**: Neo4j.
    - **RDF & Ontology**: RDF4J, Apache Jena (TDB2, Fuseki), SHACL, OWL.
    - **Formal Concept Analysis**: fcalib.
- **Data & Communication**:
    - **Message Bus**: Apache Kafka.
    - **Identity**: W3C Decentralized Identifiers (DIDs), did-common-java.
    - **API Protocols**: HAL (Hypertext Application Language), Model Context Protocol (MCP).
    - **Connectors**: Java EE Connector Architecture (JCA).
- **AI/ML**:
    - **LLM Integration**: Spring AI.
    - **Vector Databases**: Milvus, Pinecone, Weaviate.
    - **NLP/Embedding Libraries**: Hugging Face, Sentence-BERT.
- **Patterns and Methodologies**:
    - **Design Patterns**: DCI (Data, Context, Interaction), DDD (Domain-Driven Design), Dynamic Object Model, Actor-Role, Saga.
    - **Architectural Styles**: Microservices, Event-Driven Architecture, REST, HATEOAS.
- **Frontend**:
    - **Framework/Library**: React, TypeScript.
    - **Reactive State**: RxJS.