Semantic Layer for Agentic AI using Ontology, Symbolic Reasoning, and Virtual Knowledge Graph

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Semantic Layer for Agentic AI using Ontology, Symbolic Reasoning, and Virtual Knowledge Graph

Table of Contents

Semantic Layer for Agentic AI 1
Summary 1
Attachments 2
Introduction 3
Why Now? 3
Personas 3
State of the Art 4
Neurosymbolic AI 4
Data-Information-Knowledge-Wisdom (DIKW) Hierarchy 5
Semantic Web Stack 7
Prerequisites and limitations 9
Prerequisites 9
Limitations 10
Product versions 12
Architecture 14
High-Level Architecture (HLA) 14
Target technology stack 14
Target architecture 15
Interface Layer 15
Intelligence Layer 45
Knowledge Layer 56
Data Layer 69
Orchestration Layer 90
Governance / Security 102
Technology Tradeoffs & Alternatives 108
Amazon Neptune Database — Deployment Options & Capabilities 108
Deployment Modes 108
Key Capabilities Relevant to the Semantic Layer 109
Integration with Amazon OpenSearch Service and Neptune Analytics 110
Amazon S3 Vectors as an Alternative for Vector Storage 112
Compute Tradeoffs: ECS (EC2 vs Fargate) vs Lambda 114
When to Use Which 115
Virtual Knowledge Graphs (VKG) and R2RML Mapping Services 116
Why VKG Matters for the Semantic Layer 116

VKG Engines and Deployment Considerations 117
R2RML Mapping Lifecycle 118
Open-Source Alternatives: RDF/SPARQL Graph Databases 118
Decision Framework 120
Best practices 121
Best Practices 121
Well-Architected Considerations 123
1. Operational Excellence 123
2. Security 124
3. Reliability 127
4. Performance Efficiency 127
5. Cost Optimization 128
6. Sustainability 129
Related resources 131
References 131
AWS Marketplace links 132
Contributors 133
Document history 134

Semantic Layer for Agentic AI using Ontology, Symbolic Reasoning, and Virtual Knowledge Graph

Don Simpson, Taylor Riggan, and Chun Schiros, Amazon Web Services

Summary

This AWS Prescriptive Guidance establishes a comprehensive **ontology-driven semantic layer architecture** that enables trustworthy, explainable agentic AI systems for enterprise environments. While traditional semantic layers focus on BI metrics and data abstraction, this guide delivers a **knowledge-driven foundation** that transforms raw enterprise data into formal ontologies, automated symbolic reasoning, and virtual knowledge graphs — enabling AI agents to reason over complex business domains with **accuracy, consistency, and explainability** — the ACE principles that underpin trustworthy agentic AI. This aligns with emerging research showing that ontology-based semantic representations significantly improve LLM accuracy for enterprise question answering, where incomplete retrieval cascades into incomplete context for agent decision-making, leading to hallucinations, non-deterministic behavior, and ultimately poorer response quality and relevance (Allemang & Sequeda, 2024).

The guide addresses the critical challenge of autonomous agents requiring structured, verifiable knowledge to make decisions, validate actions, and provide auditable reasoning chains. It accomplishes this by implementing a **composable layered architecture** spanning Data, Knowledge, Intelligence, and Orchestration layers that transform raw enterprise data into ontology-driven knowledge graphs accessible through standardized interfaces (MCP and A2A protocols). The architecture is designed for flexible consumption — organizations can leverage the pre-built Q&A Agent for turnkey semantic question answering, or selectively invoke individual MCP tools (`intent_classification`, `generate_queries`, `execute_queries`, `graphrag_search`, `reasoning`) to compose custom workflows tailored to specific use cases. This composability enables teams to either adopt the full orchestration layer or integrate semantic capabilities directly into existing agents and applications, providing a spectrum of implementation options from fully managed to fully customized.

This guide is designed for data architects and engineers, chief data officers, chief AI officers, AI architects and engineers, and enterprise architects. Each role engages with different layers of the architecture — executive leaders and enterprise architects can focus on the composable

design patterns, governance model, and business outcomes described above, while data and AI practitioners will find detailed implementation guidance in the technology-specific sections that follow. You do not need deep familiarity with every standard or technology referenced below to successfully leverage this architecture; the layered design intentionally abstracts complexity so that each persona can engage at the appropriate level of depth.

The architecture leverages Amazon Neptune for graph databases, Amazon Bedrock for LLM capabilities and agent runtime (AgentCore), Amazon ECS for containerized services, AWS Glue Data Catalog and Amazon SageMaker Catalog for metadata management, and Amazon OpenSearch for vector storage and semantic search. The guide incorporates open-source technologies including Ontop for virtual knowledge graph implementation, ELK and HermiT for OWL reasoning, SHACL validators for constraint enforcement, and unstructured.io for document processing. It adheres to W3C Semantic Web standards including RDF, RDFS, OWL, SPARQL, SHACL, R2RML, and SWRL to ensure interoperability and ecosystem compatibility.

This is a cloud-native architecture guide designed for greenfield semantic layer implementations or incremental modernization of existing data landscapes. The guide enables organizations to virtualize existing data sources through **ontology-based data access (OBDA)** without requiring upfront data migration, materializing only what requires reasoning or cross-domain integration. The **composable design** supports multiple adoption paths: organizations can start with high-level agent orchestration (using the `ask` tool), progressively adopt individual semantic capabilities (query generation, reasoning, GraphRAG retrieval), or build entirely custom agents that leverage the semantic layer's MCP and A2A interfaces. Key outcomes include autonomous agents with explainable reasoning, federated querying across heterogeneous data sources, automated ontology induction from data catalogs, continuous knowledge graph validation through automated reasoning, and human-in-the-loop workflows for semantic governance. The **neurosymbolic approach** combines deterministic symbolic reasoning with probabilistic LLM capabilities to deliver Accuracy, Consistency, and Explainability (ACE) for regulated industries requiring auditable AI systems.

Attachments

To access additional content that is associated with this document, download and unzip the following file:

attachment.zip

Introduction

Why Now?

The convergence of several technological and business forces makes this the ideal moment for semantic data foundations to enable agentic AI at enterprise scale.

Agents with autonomy are challenging without semantic foundations. Autonomous agents require a shared semantic foundation that encodes organizational knowledge, enabling them to understand business context, validate decisions against explicit constraints, and provide auditable reasoning, capabilities that retrieval and probabilistic models alone cannot deliver.

LLMs have made it easier than ever to facilitate the creation and management of ontologies and associated rules/axioms. Foundation models can now accelerate ontology induction from data catalogs, extract entities and relationships from specifications, generate semantic annotations, and propose axioms based on data patterns — tasks that previously required months of manual knowledge engineering. This democratizes semantic layer development, enabling organizations to bootstrap knowledge graphs rapidly while maintaining quality through human-in-the-loop validation workflows.

Customers need an incremental approach to get their data ready for AI. Enterprise data landscapes are complex, heterogeneous, and constantly evolving. Organizations cannot afford "big bang" transformations that require upfront data migration or complete system rewrites. The semantic layer approach enables incremental value delivery — virtualizing existing data sources through ontology-based data access (OBDA), materializing only what requires reasoning or cross-domain integration, and progressively enriching the knowledge graph as new use cases emerge. This reduces time-to-value, minimizes risk, and aligns with modern data mesh and data fabric architectures.

Personas

This guidance addresses three primary personas who collaborate to build and leverage semantic data foundations for agentic AI:

Administrators — Platform engineers, cloud architects, and infrastructure specialists responsible for deploying, securing, and operating the semantic layer infrastructure. They configure AWS

services (Neptune, ECS, Bedrock, Glue), implement governance policies, manage access controls, and ensure observability, scalability, and cost optimization across the semantic stack.

Data-Focused Practitioners — Data engineers, ontology engineers, and knowledge engineers who design ontologies, create R2RML mappings, configure reasoning rules, and manage the lifecycle of semantic artifacts. They bridge technical data structures (schemas, tables, APIs) with business semantics (ontologies, glossaries, constraints), ensuring that the semantic layer accurately represents enterprise knowledge and remains synchronized with evolving data sources.

Business Analysts — Domain experts, data stewards, and business analysts who understand data domains, data products, business rules, and organizational knowledge. They define business glossaries, validate ontology completeness, approve semantic changes, and ensure that the knowledge graph reflects business reality. Their domain expertise grounds the semantic layer in real-world context, enabling agents to reason about business concepts rather than just technical data structures.

State of the Art

Neurosymbolic AI

The semantic data foundations architecture embraces **neurosymbolic AI** — the integration of neural networks (LLMs, embeddings) with symbolic reasoning (ontologies, logic, rules) — to deliver the best of both paradigms.

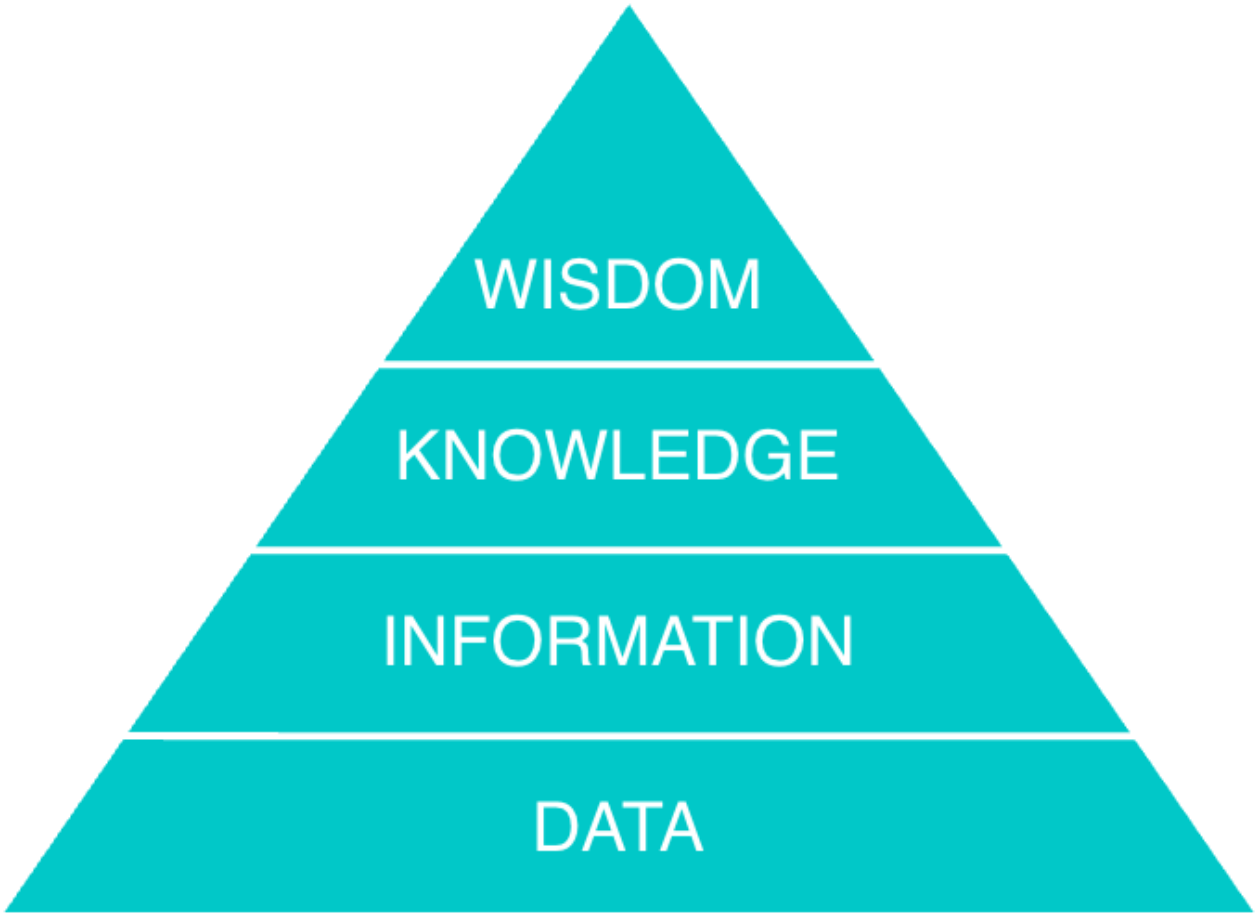
Deterministic vs. Non-Deterministic Reasoning — Symbolic reasoning (OWL, SPARQL, SHACL, SWRL) provides deterministic, explainable inference based on formal logic and ontology axioms. Given the same knowledge graph and rules, reasoning engines produce consistent, reproducible results with full provenance. In contrast, LLMs and neural networks are non-deterministic — they generate probabilistic outputs based on learned patterns, which can vary across invocations and may produce hallucinations or inconsistent answers. By combining both approaches, the architecture leverages LLMs for semantic enrichment, natural language understanding, and flexible pattern recognition, while relying on symbolic reasoning for validation, consistency checking, and explainable inference.

Accuracy and Explainability — Symbolic reasoning ensures accuracy for structured queries and rule-based inference, with every conclusion traceable to specific axioms, constraints, or data assertions. This explainability is critical for regulated industries (healthcare, finance, government)

where decisions must be auditable and defensible. LLMs complement this by handling ambiguous natural language, extracting entities from unstructured text, and synthesizing human-readable explanations — but their outputs are validated against ontology constraints and reasoning rules before being asserted as knowledge.

Complementary Strengths — LLMs excel at semantic similarity, entity disambiguation, ontology induction, and natural language generation — tasks that benefit from learned patterns across vast corpora. Symbolic reasoning excels at consistency validation, transitive inference, constraint enforcement, and logical deduction — tasks that require formal guarantees. The architecture orchestrates both: LLMs propose candidate knowledge, symbolic reasoners validate and refine it, and the combination delivers trustworthy, explainable AI that scales across enterprise use cases.

Data-Information-Knowledge-Wisdom (DIKW) Hierarchy



The **DIKW hierarchy** provides a conceptual framework for understanding how raw data evolves into actionable wisdom - a progression that inspired the layered architecture of the Semantic Layer for Agentic AI. While the architecture aligns closely with DIKW's four tiers, it extends the framework with an Intelligence tier that addresses enterprise requirements for continuous self-assessment and semantic evolution.

Data Layer — Data Tier Raw data from source systems (databases, data lakes, documents, APIs) represents the Data tier — facts without context or semantic meaning.

Data Layer — Information Tier The Data Layer transforms raw data into the Information tier through document ingestion, GraphRAG processing, and information extraction — chunking documents, extracting entities, indexing concepts with vector embeddings, and cataloging metadata. The result is indexed, contextualized information assets stored in the Lexical Graph Database and accessible through data catalogs (AWS Glue Data Catalog, Amazon SageMaker Catalog). Information is data with context — organized, structured, and prepared for semantic interpretation.

Knowledge Layer — The **Knowledge** tier emerges when data is organized into ontology-driven structures with explicit semantics, relationships, and constraints. The Knowledge Layer (Virtual Knowledge Graph Service, Semantic Layer Graph Database, Reasoning Service) transforms data into knowledge by applying ontologies (T-Box) to instance data (A-Box), enabling semantic queries, federated access, and inference. Knowledge is contextualized, validated, and interconnected — entities are linked by meaningful relationships, and reasoning derives implicit facts from explicit assertions.

Intelligence Layer — While DIKW traditionally progresses directly from Knowledge to Wisdom, this architecture introduces an Intelligence tier representing meta-knowledge — knowledge about knowledge. This extension recognizes that enterprise semantic systems require continuous self-assessment and evolution beyond static knowledge representation. The Intelligence Layer (Ontology Induction, Ontology Evaluation, Truth Maintenance, Context Graphs) continuously assesses, refines, and evolves the semantic layer based on data patterns, usage analytics, and domain expertise. It identifies gaps, resolves ambiguities, maintains alignment with source data, and provides episodic memory for agents — enabling the system to learn and adapt over time.

Orchestration Layer — The **Wisdom** tier represents the application of knowledge and intelligence to make decisions, answer questions, and take actions. The Orchestration Layer (Q&A Agent, Business Elicitation Agent, Ontology Maintenance Agent) embodies wisdom by leveraging semantic foundations to deliver business value — synthesizing knowledge from multiple sources, reasoning

over complex scenarios, and orchestrating human-in-the-loop workflows when automated reasoning requires domain expertise.

This alignment demonstrates how the architecture operationalizes the DIKW progression, enabling organizations to move from data management to knowledge-driven decision-making and ultimately to autonomous, intelligent systems.

Semantic Web Stack

The architecture is grounded in **Semantic Web standards** — a mature, open, and interoperable technology stack developed by the W3C for representing, querying, and reasoning over knowledge graphs at web scale.

RDF (Resource Description Framework) — The foundational data model representing knowledge as subject-predicate-object triples, enabling flexible, graph-native knowledge representation without rigid schemas.

RDFS (RDF Schema) and OWL (Web Ontology Language) — Ontology languages for defining classes, properties, hierarchies, and axioms. OWL provides rich expressivity for complex domain modeling, including property characteristics (transitive, functional), cardinality constraints, and class expressions.

SPARQL — The standard query language for RDF, enabling federated queries across distributed knowledge graphs and data sources. SPARQL supports SELECT (retrieve bindings), CONSTRUCT (generate new graphs), ASK (boolean queries), and DESCRIBE (entity descriptions).

SHACL (Shapes Constraint Language) — A constraint validation language for defining data quality rules, cardinality constraints, and business logic as shapes that validate RDF graphs.

R2RML (RDB to RDF Mapping Language) — The standard for defining mappings from relational databases to RDF, enabling ontology-based data access (OBDA) and virtual knowledge graphs without data duplication.

SWRL (Semantic Web Rule Language) — A rule language for expressing inference rules that extend OWL reasoning, enabling domain-specific business logic and complex deductions.

By adhering to Semantic Web standards, the architecture ensures **interoperability** (knowledge graphs can be shared across systems), **longevity** (standards-based implementations are future-proof), and **ecosystem compatibility** (integration with open-source tools like Ontop, Apache Jena,

and commercial platforms like Stardog). This open architecture ensures that semantic capabilities are not locked into proprietary formats or vendor-specific implementations, enabling customers to leverage best-of-breed tools and evolve their semantic layer as the ecosystem matures.

This introduction establishes the strategic context (Why Now?), target audience (Personas), theoretical foundations (Neurosymbolic AI, DIKW), and technical grounding (Semantic Web Stack) for the AWS Prescriptive Guidance on Semantic Data Foundations for Agentic AI — positioning the architecture as a pragmatic, standards-based approach to enabling trustworthy, explainable, and scalable autonomous agents for enterprise use cases.

Prerequisites and limitations

Prerequisites

AWS Services and Access:

- Active AWS account with appropriate IAM permissions for deploying and managing Amazon Neptune, Amazon Bedrock, Amazon ECS, AWS Glue Data Catalog, Amazon SageMaker Catalog, Amazon OpenSearch, Amazon S3, Amazon SNS/SQS, and AWS X-Ray
- Amazon Bedrock model access enabled (e.g., Anthropic Claude Sonnet 4.0) through the Bedrock console
- Amazon Bedrock AgentCore access (requires Python 3.10+ for development with the starter toolkit)
- VPC configuration with appropriate subnets, security groups, and network connectivity for ECS services and Neptune clusters

Note: some AWS services aren't available in all AWS Regions. For Region availability, see the Service endpoints and quotas page in the AWS documentation, and choose the link for the service.

Technical Skills and Knowledge:

- Understanding of W3C Semantic Web standards (RDF, RDFS, OWL, SPARQL, SHACL, R2RML, SWRL)
- Experience with ontology engineering and knowledge graph design
- Familiarity with data catalog systems (AWS Glue Data Catalog, Amazon SageMaker Catalog/ DataZone)
- Knowledge of containerization and Amazon ECS deployment patterns
- Understanding of event-driven architectures (SNS/SQS, Neptune DB Streams)
- Experience with LLM integration and prompt engineering via Amazon Bedrock
- Familiarity with observability tools (OpenTelemetry, AWS X-Ray, CloudWatch)

Data and Infrastructure:

- Existing data sources (relational databases, data warehouses, data lakes, document repositories) with documented schemas
- Business glossaries, data dictionaries, or domain expertise to inform ontology development
- Network connectivity between data sources and the semantic layer infrastructure
- S3 buckets for storing ontology versions, R2RML mappings, and document processing artifacts

Open-Source Tools (Optional but Recommended):

- Ontop 5.5.0 or later for Virtual Knowledge Graph implementation
- ELK, HermiT, or other OWL reasoners for inference and consistency checking
- SHACL validators for constraint enforcement
- unstructured.io for document processing (or equivalent document parsing capabilities)

Organizational Readiness:

- Defined personas and responsibilities (Administrators, Data-Focused Practitioners, Business Analysts)
- Governance processes for ontology approval, versioning, and lifecycle management
- Human-in-the-loop workflows for semantic validation and domain expert review

Limitations

Amazon Neptune:

- Cluster volume size limited to **128 TiB per cluster** (up to 64 TiB in some regions)
- Maximum of 15 read replicas per cluster
- Query throughput scales to 100k+ queries per second but depends on query complexity and instance types

Amazon Bedrock AgentCore:

- Framework support via starter kit CLI limited to prominent frameworks (Strands, LangChain, CrewAI)
- Session duration and memory persistence constraints depend on AgentCore service limits

- Model access and token limits vary by foundation model provider (e.g., Anthropic Claude)

Virtual Knowledge Graph (Ontop):

- Query performance depends on underlying data source capabilities and R2RML mapping complexity
- SPARQL-to-SQL translation limited to OWL 2 QL profile for optimal query rewriting
- Federated queries across heterogeneous sources may experience latency based on network and source system performance

Reasoning Services:

- OWL reasoning complexity (ELK supports OWL 2 EL profile; HermiT supports OWL 2 DL but with higher computational cost)
- Reasoning over large A-Box (instance data) may require materialization strategies or incremental reasoning approaches
- SWRL rule execution performance depends on rule complexity and data volume

Document Processing:

- File size limits for document ingestion (typically 50MB per file for unstructured.io and similar tools)
- OCR accuracy for scanned documents depends on image quality and document structure
- GraphRAG retrieval quality depends on embedding model selection and chunk size optimization

General Constraints:

- Incremental adoption requires careful planning of ontology scope and R2RML mapping coverage to avoid incomplete virtualization
- Cross-domain ontology queries may require multiple execution steps, impacting latency
- Human-in-the-loop workflows introduce approval latency for ontology changes and entity resolution

Product versions

AWS Services:

- Amazon Neptune Database (1.4.7.0+)
- Amazon Bedrock (with AgentCore support, requires Python 3.10+)
- Amazon ECS (Fargate or EC2 launch types)
- AWS Glue Data Catalog
- Amazon SageMaker Catalog / Amazon DataZone
- Amazon OpenSearch Service (3.5+)
- Amazon S3, SNS, SQS, X-Ray, CloudWatch

Open-Source Components:

- **Ontop**: Version 5.5.0 (released February 14, 2026) or later
- **ELK Reasoner**: Latest stable version compatible with OWL 2 EL
- **HermiT Reasoner**: Latest stable version compatible with OWL 2 DL
- **SHACL Validators**: Implementations compatible with W3C SHACL 1.1 or SHACL 1.2 specifications
- **unstructured.io**: Latest stable version for document processing

W3C Standards (Specifications):

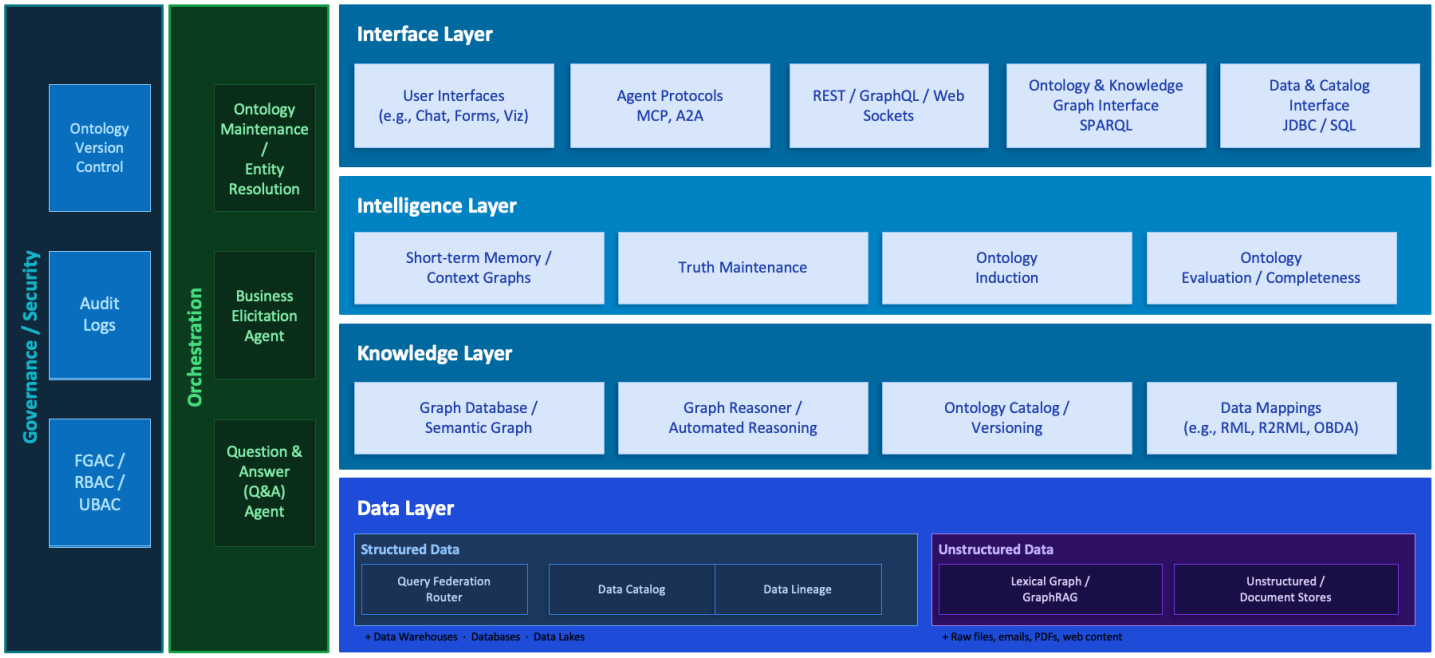
- RDF 1.1 (https://www.w3.org/RDF/)
- RDFS 1.1 (https://www.w3.org/TR/rdf-schema/)
- OWL 2 (https://www.w3.org/OWL/)
- SPARQL 1.2 (https://www.w3.org/TR/sparql12-query/)
- SHACL 1.1 / 1.2 (https://www.w3.org/TR/shacl/)
- R2RML (https://www.w3.org/TR/r2rml/)
- SWRL (community-maintained specification)

Note

This architecture is designed to be version-agnostic where possible, leveraging AWS managed services that automatically update and open standards that ensure long-term compatibility. Specific version requirements apply primarily to development tooling (Python 3.10+ for AgentCore) and open-source components (Ontop 5.5.0+).

Architecture

High-Level Architecture (HLA)



Target technology stack

- **Amazon Neptune** – A fully managed graph database service that makes it easy to build and run applications that work with highly connected datasets. An Amazon Neptune-based knowledge graph finds relationships between projects and provides recommendations.
- **Amazon Bedrock** – A fully managed service providing API access to high-performing foundation models (FMs) from Amazon and leading AI companies. It supports fine-tuning, RAG-based customization, and Automated Reasoning — without managing infrastructure — with built-in security, privacy, and responsible AI capabilities for building and scaling generative AI applications on AWS.
- **Amazon Bedrock AgentCore** – A managed platform for building, deploying, and operating enterprise-grade AI agents at scale. It provides purpose-built infrastructure, session management, memory, and security controls to move agents from proof-of-concept to production — enabling secure, reliable autonomous agent operations at enterprise scale.
- **Ontop** – An open-source Virtual Knowledge Graph (VKG) system that exposes relational databases as virtual RDF knowledge graphs — without moving data. It translates SPARQL queries

over ontology-defined knowledge graphs into SQL executed directly against relational sources, using R2RML mappings and lightweight OWL ontologies. Built on OBDA (Ontology-Based Data Access) principles, Ontop enables semantic querying over existing data infrastructure with no data duplication.

- **Amazon OpenSearch Service** – A managed service for real-time search, monitoring, and analysis. It provides vector storage and semantic similarity search for embedding-based retrieval in RAG and GraphRAG workflows. Available as provisioned clusters or OpenSearch Serverless (auto-scaling, pay-per-use).

Target architecture

- **Interface Layer**
- **Intelligence Layer**
- **Knowledge Layer**
- **Data Layer**
- **Orchestration Layer**
- **Governance / Security**

Interface Layer

Interface Layer

The Interface Layer establishes the formal contract between the Semantic Layer and its consumers — including User Interfaces, Services, and Agents. By supporting a broad set of protocols and query languages, it enables diverse use cases and personas to interact with semantic data in the way that best fits their needs.

Protocol Support

- **Agent Protocols:** Native support for Model Context Protocol (MCP) and Agent-to-Agent (A2A), enabling autonomous agents to discover, invoke, and compose semantic capabilities
- **Web & Service APIs:** REST, GraphQL, and WebSockets provide flexible, standards-based integration for applications and services

Query Language Support

- **SPARQL:** Full CRUD operations over Ontologies and Knowledge Graphs, enabling precise semantic querying and knowledge engineering workflows
- **SQL over JDBC:** Data and Catalog CRUD for familiar, structured access to semantic assets by data engineers and analysts

Supported Use Cases & Personas

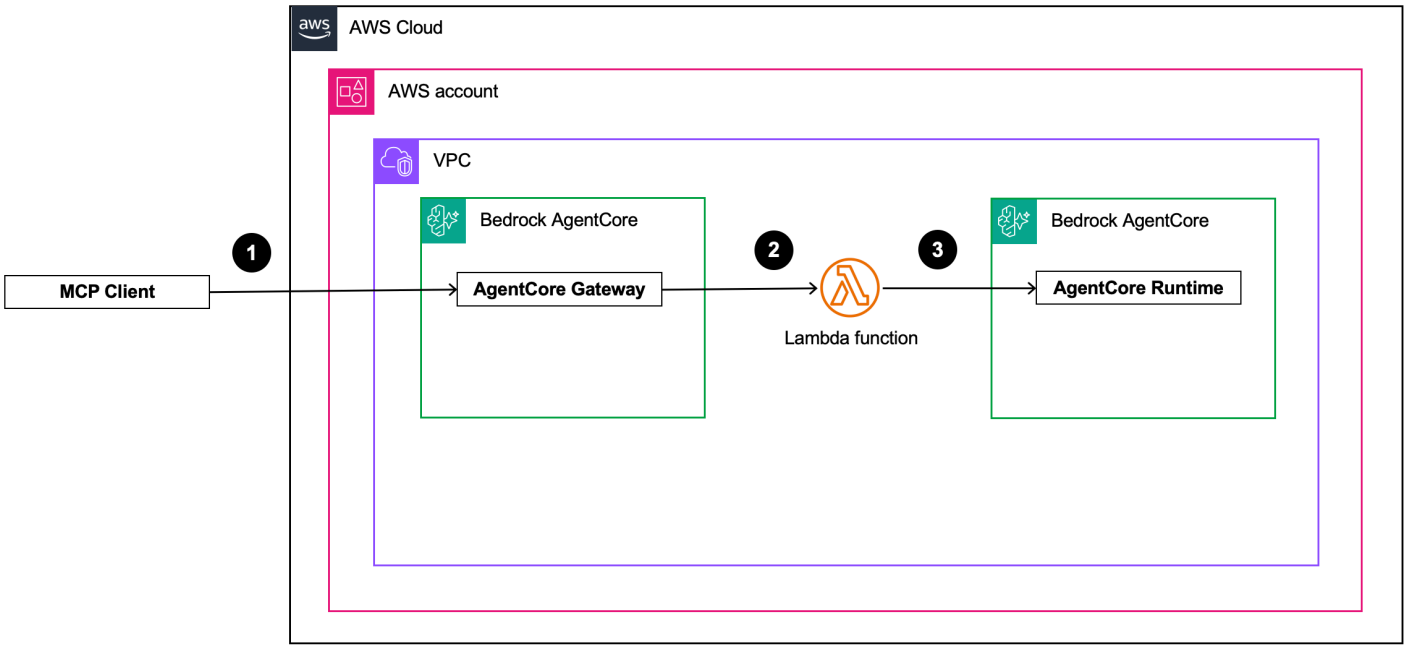
- Ontology Management & Knowledge Engineering UIs
- Chatbots and conversational interfaces
- Autonomous Agents operating within agentic AI pipelines

By abstracting the complexity of the Semantic Layer behind well-defined interfaces, the Interface Layer ensures interoperability, extensibility, and broad adoption across the enterprise.

- Model Context Protocol (MCP)
- Agent-to-Agent (A2A)
- REST API

Model Context Protocol (MCP)

MCP Server



Data Flow

1. MCP Client → AgentCore Gateway The MCP Client initiates a connection to the MCP Server endpoint exposed via **Amazon Bedrock AgentCore Gateway**. AgentCore Gateway supports two inbound authorization mechanisms: **IAM SigV4** (the default, requiring no additional configuration) and **JWT/OIDC bearer tokens** from trusted identity providers — including Amazon Cognito, Microsoft Entra ID, Okta, Auth0, Keycloak, and other OIDC-compliant providers. For JWT authorization, the Gateway is configured with a discovery URL, allowed audiences, and optionally allowed client IDs. The Gateway validates the token, enforces access policies, and routes the MCP tool invocation to the configured backend target.

2. AgentCore Gateway → AWS Lambda (MCP-Enabled Tool) AgentCore Gateway MCP-enables an **AWS Lambda** function by mapping MCP tool definitions to Lambda invocations — without requiring changes to the Lambda function's core logic. The Gateway manages session context, input schema validation, and response formatting, returning structured MCP-compliant tool

responses to the client. Lambda executes within a VPC or public subnet depending on the target data source or service integration.

3. (Optional) Lambda → AgentCore Runtime (Agent-as-Tool Pattern) For use cases requiring multi-step reasoning or orchestration, the Lambda function invokes **Amazon Bedrock AgentCore Runtime**, implementing the *Agent-as-Tool* pattern. AgentCore Runtime hosts the agent process in an isolated, serverless execution environment with managed memory and session state. Any supported agent framework — such as **AWS Strands** — can run within AgentCore Runtime, enabling the MCP tool to transparently delegate to a full agentic workflow.

4. Observability — OpenTelemetry Integration All components emit traces, metrics, and logs via **OpenTelemetry (OTel)**. AgentCore Runtime natively supports OTel instrumentation, capturing agent reasoning steps, tool invocations, latency, and token usage. Telemetry is exported to **AWS X-Ray** for distributed tracing and **Amazon CloudWatch** for metrics and log aggregation — providing end-to-end visibility across the MCP client, Gateway, Lambda, and agent execution layers.

5. Security & Governance AgentCore Gateway enforces **IAM-based authentication** and supports resource-based policies for fine-grained tool access control. Lambda execution roles follow least-privilege principles. Optionally, **Amazon Bedrock Guardrails** — including Automated Reasoning checks — can be applied within the agent pipeline to validate outputs before returning responses to the MCP client.

Tool Catalog for Answer-Getting

Tool Catalog for Question & Answer

Tool Catalog for Question & Answer

1. ask — Natural Language Query with Clarification Support

Purpose: Primary entry point for natural language questions. Accepts a question and returns an answer grounded in the knowledge graph, with support for multi-turn clarification when additional information is needed.

Input:

- `question: string` — Natural language question (required)
- `ontology_iri?: IRI` — Optional ontology scope to limit query to specific domain ontologies
- `conversation_id?: string` — Optional conversation context for multi-turn interactions

Output:

- `status: "needs_clarification" | "complete" | "error"` — Response status
- `answer: string` — Natural language answer
- `clarification_request?: { message: string, options?: [], required_fields?: [] }` — Structured clarification request
- `evidence: [{ subject, predicate, object, graph }]` — Supporting evidence triples
- `confidence: float` — Overall confidence score (0.0-1.0)
- `conversation_id: string` — Conversation ID for follow-up queries
- `metrics_used: [{ metric_id, metric_name, version, expression, publish_date }]` — lists which Metric Store definitions were injected during query generation

Example Input:

```
{
  "question": "What is the average claim amount for diabetes patients in Q4 2025?",
  "ontology_iri": "urn:ontology:healthcare:claims:v2"
}
```

Example Output:

```
{
  "status": "complete",
  "answer": "The average claim amount for diabetes patients in Q4 2025 was $3,847.23,
 based on 1,247 claims across 892 unique patients.",
  "evidence": [
    {
      "subject": "urn:claim:12345",
      "predicate": "claimAmount",
      "object": "4200.00",
      "graph": "urn:graph:claims:2025-q4"
    }
  ],
  "confidence": 0.94,
  "conversation_id": "conv-12345"
}
```

2. search — Semantic Search

Purpose: Explore ontology classes, properties, and individuals using keyword, semantic similarity, or hybrid search modes.

Input:

- `query: string` — Search query (required)
- `mode?: "keyword" | "semantic" | "hybrid"` — Search mode (default: "hybrid")
- `ontology_iri?: IRI` — Optional ontology scope
- `top_k?: int` — Maximum number of results to return

Output:

- `results: [{ iri, label, type, score, rdfs_comment }]` — Ranked search results with similarity scores

Example Input:

```
{
  "query": "patient claims",
  "mode": "hybrid",
  "ontology_iri": "urn:ontology:healthcare:claims:v2",
  "top_k": 5
}
```

Example Output:

```
{
  "results": [
    {
      "iri": "urn:ontology:healthcare:claims:Claim",
      "label": "Claim",
      "type": "owl:Class",
      "score": 0.92,
      "rdfs_comment": "A request for payment submitted to an insurance provider for
 medical services rendered. Synonyms: insurance claim, medical claim, reimbursement
 request"
    },
```

```
    {
      "iri": "urn:ontology:healthcare:claims:Patient",
      "label": "Patient",
      "type": "owl:Class",
      "score": 0.88,
      "rdfs_comment": "An individual receiving or registered to receive medical care.
 Synonyms: healthcare recipient, medical patient"
    }
  ]
}
```

3. intent_classification — Question Intent Classification

Purpose: Analyze question intent, extract entities, and determine processing requirements.

Input:

- `question: string` — Natural language question (required)
- `ontology_iri?: IRI` — Optional ontology scope
- `conversation_id?: string` — Optional conversation context

Output:

- `intent_category: string` — Intent type
- `extracted_entities: [{ entity_iri, label, type, ontology_iri }]` — Entities grounded in ontology
- `extracted_relationships: [{ subject, predicate, object }]` — Relationships identified
- `suggested_strategy: string` — Recommended query strategy
- `requires_graphrag: boolean` — Whether document retrieval is needed
- `confidence: float` — Classification confidence score

Example Input:

```
{
  "question": "What is the average claim amount for diabetes patients in Q4 2025?",
  "ontology_iri": "urn:ontology:healthcare:claims:v2"
```

```
}
```

Example Output:

```
{
  "intent_category": "AGGREGATION",
  "extracted_entities": [
    {
      "entity_iri": "urn:ontology:healthcare:claims:Claim",
      "label": "Claim",
      "type": "owl:Class",
      "ontology_iri": "urn:ontology:healthcare:claims:v2"
    },
    {
      "entity_iri": "urn:ontology:healthcare:claims:Patient",
      "label": "Patient",
      "type": "owl:Class",
      "ontology_iri": "urn:ontology:healthcare:claims:v2"
    }
  ],
  "extracted_relationships": [
    {
      "subject": "Claim",
      "predicate": "submittedBy",
      "object": "Patient"
    }
  ],
  "suggested_strategy": "sparql",
  "requires_graphrag": false,
  "confidence": 0.91
}
```

4. generate_queries — Query Generation & Optimization

Purpose: Generate optimized SPARQL queries and execution plans based on question intent.

Input:

- `question: string` — Natural language question (required)
- `ontology_iri?: IRI` — Optional ontology scope
- `conversation_id?: string` — Optional conversation context

- `intent_classification: object` — Output from `intent_classification` tool (required)

Output:

- `queries: [{ query: string, named_graph?: IRI, description: string }]` — Generated SPARQL queries
- `execution_plan: { strategy, estimated_cost, estimated_latency_ms }` — Execution strategy
- `graphrag_plan?: { similarity_threshold, top_k, traversal_depth }` — GraphRAG parameters
- `explanation: string` — Natural language explanation

Example Input:

```
{
  "question": "What is the average claim amount for diabetes patients in Q4 2025?",
  "ontology_iri": "urn:ontology:healthcare:claims:v2",
  "intent_classification": {
    "intent_category": "AGGREGATION",
    "extracted_entities": [...],
    "suggested_strategy": "sparql"
  }
}
```

Example Output:

```
{
  "queries": [
    {
      "query": "PREFIX : <urn:ontology:healthcare:claims:> SELECT (AVG(?amount) AS ?
avgAmount) WHERE { ?claim a :Claim ; :claimAmount ?amount ; :submittedBy ?patient. ?
patient :hasCondition :Diabetes. FILTER(?claimDate >= '2025-10-01' && ?claimDate <=
 '2025-12-31') }",
      "named_graph": "urn:graph:claims:2025-q4",
      "description": "Calculate average claim amount for diabetes patients in Q4 2025"
    }
  ],
```

```
  "execution_plan": {
    "strategy": "parallel",
    "estimated_cost": 0.15,
    "estimated_latency_ms": 250
  },
  "explanation": "This query filters claims by diabetes patients in Q4 2025 and
 calculates the average claim amount using SPARQL aggregation."
}
```

5. execute_queries — Federated Query Execution

Purpose: Execute SPARQL queries across federated knowledge graphs.

Input:

- `queries: [{ query: string, named_graph?: IRI }]` — SPARQL queries to execute

Output:

- `results: [{ query_index, bindings, graph?, boolean? }]` — Query results
- `provenance: [{ source_type, source_id, confidence }]` — Source metadata
- `execution_stats: { total_latency_ms, queries_executed, sources_queried }` — Performance metrics

Example Input:

```
{
  "queries": [
    {
      "query": "PREFIX : <urn:ontology:healthcare:claims:> SELECT (AVG(?amount) AS ?
avgAmount) WHERE { ?claim a :Claim ; :claimAmount ?amount ; :submittedBy ?patient. ?
patient :hasCondition :Diabetes. FILTER(?claimDate >= '2025-10-01' && ?claimDate <=
 '2025-12-31') }",
      "named_graph": "urn:graph:claims:2025-q4"
    }
  ]
}
```

Example Output:

```
{
  "results": [
    {
      "query_index": 0,
      "bindings": [
        {
          "avgAmount": "3847.23"
        }
      ]
    }
  ],
  "provenance": [
    {
      "source_type": "neptune_graph",
      "source_id": "urn:graph:claims:2025-q4",
      "confidence": 0.98
    }
  ],
  "execution_stats": {
    "total_latency_ms": 245,
    "queries_executed": 1,
    "sources_queried": ["neptune_graph"]
  }
}
```


6. graphrag_search — GraphRAG Retrieval

Purpose: Retrieve relevant document chunks and concepts using hybrid retrieval.

Input:

- `question: string` — Natural language question (required)
- `ontology_iri?: IRI` — Optional ontology scope
- `conversation_id?: string` — Optional conversation context
- `similarity_threshold?: float` — Minimum cosine similarity (default: 0.7)
- `top_k?: int` — Maximum results (default: 10)
- `traversal_depth?: int` — Graph traversal depth (default: 2)

Output:

- `document_chunks: [{ chunk_id, document_id, text, page_number, similarity_score }]` — Retrieved chunks
- `concepts: [{ concept_iri, label, definition, similarity_score, related_concepts }]` — Extracted concepts
- `graph_context: [{ subject, predicate, object }]` — Graph relationships
- `provenance: [{ document_id, chunk_id, s3_uri }]` — Source document links

Example Input:

```
{
  "question": "What are the common complications of diabetes?",
  "similarity_threshold": 0.75,
  "top_k": 5
}
```

Example Output:

```
{
  "document_chunks": [
    {
      "chunk_id": "chunk-42",
      "document_id": "diabetes-guidelines-2025",
      "text": "Common complications of diabetes include cardiovascular disease,
 neuropathy, nephropathy, and retinopathy...",
      "page_number": 15,
      "similarity_score": 0.89
    }
  ],
  "concepts": [
    {
      "concept_iri": "urn:concept:diabetes-complications",
      "label": "Diabetes Complications",
      "definition": "Medical conditions arising from diabetes",
      "similarity_score": 0.91,
      "related_concepts": ["cardiovascular-disease", "neuropathy"]
    }
  ],
  "graph_context": [
    {
      "subject": "urn:concept:diabetes",
```

```
      "predicate": "hasComplication",
      "object": "urn:concept:cardiovascular-disease"
    }
  ],
  "provenance": [
    {
      "document_id": "diabetes-guidelines-2025",
      "chunk_id": "chunk-42",
      "s3_uri": "s3://medical-docs/diabetes-guidelines-2025.pdf"
    }
  ]
}
```

7. reasoning — Semantic Reasoning & Validation

Purpose: Perform inference, constraint validation, and consistency checking on query results.

Input:

- `vkg_results?: object` — Output from `execute_queries` (optional)
- `graphrag_results?: object` — Output from `graphrag_search` (optional)
- `search_results?: object` — Output from `search` (optional)
- `ontology_iri?: IRI` — Optional ontology scope for reasoning rules

Output:

- `inferred_triples: [{ subject, predicate, object, rule, confidence }]` — New triples derived
- `validation_results: { conforms, violations }` — SHACL validation results
- `consistency_check: { is_consistent, conflicts }` — Consistency validation
- `reasoning_steps: [{ step, axiom, rule }]` — Reasoning chain explanation
- `provenance: [{ source_type, source_id, rule, confidence }]` — Links to axioms/rules

Example Input:

```
{
  "vkg_results": {
```

```
    "results": [
      {
        "bindings": [{ "avgAmount": "3847.23" }]
      }
    ]
  },
  "ontology_iri": "urn:ontology:healthcare:claims:v2"
}
```

Example Output:

```
{
  "inferred_triples": [
    {
      "subject": "urn:claim:12345",
      "predicate": "requiresReview",
      "object": "true",
      "rule": "HighValueClaimRule",
      "confidence": 0.87
    }
  ],
  "validation_results": {
    "conforms": true,
    "violations": []
  },
  "consistency_check": {
    "is_consistent": true,
    "conflicts": []
  },
  "reasoning_steps": [
    {
      "step": "Applied HighValueClaimRule",
      "axiom": "Claims > $5000 require review",
      "rule": "SWRL: Claim(?c) ^ claimAmount(?c, ?amt) ^ greaterThan(?amt, 5000) ->
 requiresReview(?c, true)"
    }
  ],
  "provenance": [
    {
      "source_type": "reasoning_rule",
      "source_id": "HighValueClaimRule",
      "rule": "SWRL",
      "confidence": 0.87
```

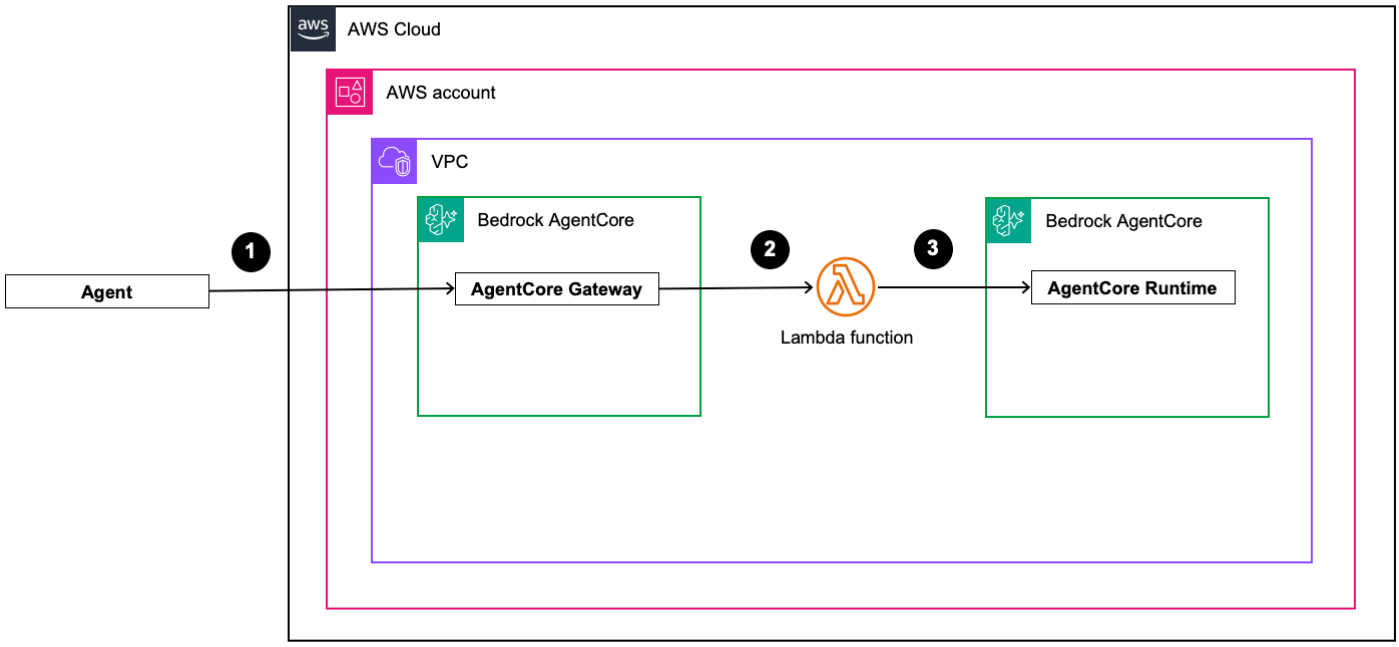
```
    }
  ]
}
```

Tool Design Principles

- `ask` is the primary entry point for chatbots and non-technical users — it abstracts all complexity and orchestrates the other tools internally
- `intent_classification`, `generate_queries`, `execute_queries`, `graphrag_search`, `reasoning` expose the Q&A Agent's internal orchestration steps as discrete tools, enabling power users and agents to compose custom workflows
- `search` provides lightweight semantic search without full Q&A orchestration
- All tools propagate OpenTelemetry trace context for end-to-end observability
- All tools support `ontology_iri` for scoping to specific domain ontologies
- All tools return provenance metadata to support trustworthy AI and explainability

Agent-to-Agent (A2A)

The **Agent-to-Agent (A2A) Protocol** enables external agents to discover and delegate tasks to Orchestration Layer agents (Q&A Agent, Business Elicitation Agent, Ontology Maintenance Agent) using **HTTP + JSON-RPC 2.0 + SSE**.



Agent Discovery

Each agent publishes an **Agent Card** (JSON document) advertising capabilities, schemas, and endpoints. External agents discover cards via Agent Registry, DNS-SD, or `./well-known/agent-card.json`.

Q&A Agent Card

```
{
  "agent_id": "urn:aws:bedrock:...:agent/qa-agent",
  "name": "Question & Answer Agent",
  "capabilities": [{
    "capability_id": "semantic_qa",
    "input_schema": { "question": "string", "ontology_id": "string?" },
    "output_schema": { "answer": "string", "evidence": "array", "citations": "array" }
  }],
  "endpoints": { "invoke": "/invoke" },
  "authentication": {
    "methods": ["IAM_SigV4", "JWT_OIDC"],
    "required_claims": ["sub", "groups", "act"],
    "scope": "semantic-layer:access"
```

```
  }
}
```

Task Delegation Workflow

1. Discovery → External agent retrieves Agent Card

2. Invocation → Sends JSON-RPC request:

```
{
  "jsonrpc": "2.0",
  "method": "semantic_qa",
  "params": { "question": "What is the average claim amount for diabetes patients in Q4
 2025?" }
}
```

3. Authentication → JWT/OIDC validation (preferred) or IAM SigV4 for AWS-native callers

4. Execution → Agent orchestrates internal services (Intent Classification, VKG, GraphRAG, Reasoning)

5. Response → Returns result with answer, evidence, citations, confidence

6. Async (Optional) → Long-running tasks return `task_id` and stream progress via SSE

Orchestration Layer Agents

All agents can support A2A delegation:

- **Q&A Agent** — Semantic question answering with federated knowledge graphs
- **Business Elicitation Agent** — Business rule extraction, requirement validation
- **Ontology Maintenance Agent** — Entity resolution, ontology alignment, truth maintenance

A2A + MCP Integration

A2A (agent-to-agent) and **MCP** (agent-to-service) are complementary:

```
External Agent # A2A # Q&A Agent # MCP # Semantic Layer Services
```

Security & Observability

Auth — Auth - JWT/OIDC (preferred), IAM SigV4 (AWS-native callers) | Authz - Group-based authorization via JWT `groups` claim with `semantic-layer:access` scope

Observability — OpenTelemetry traces, CloudWatch Logs, AWS X-Ray

Provenance — All responses include source data, reasoning steps, ontology axioms

Benefits

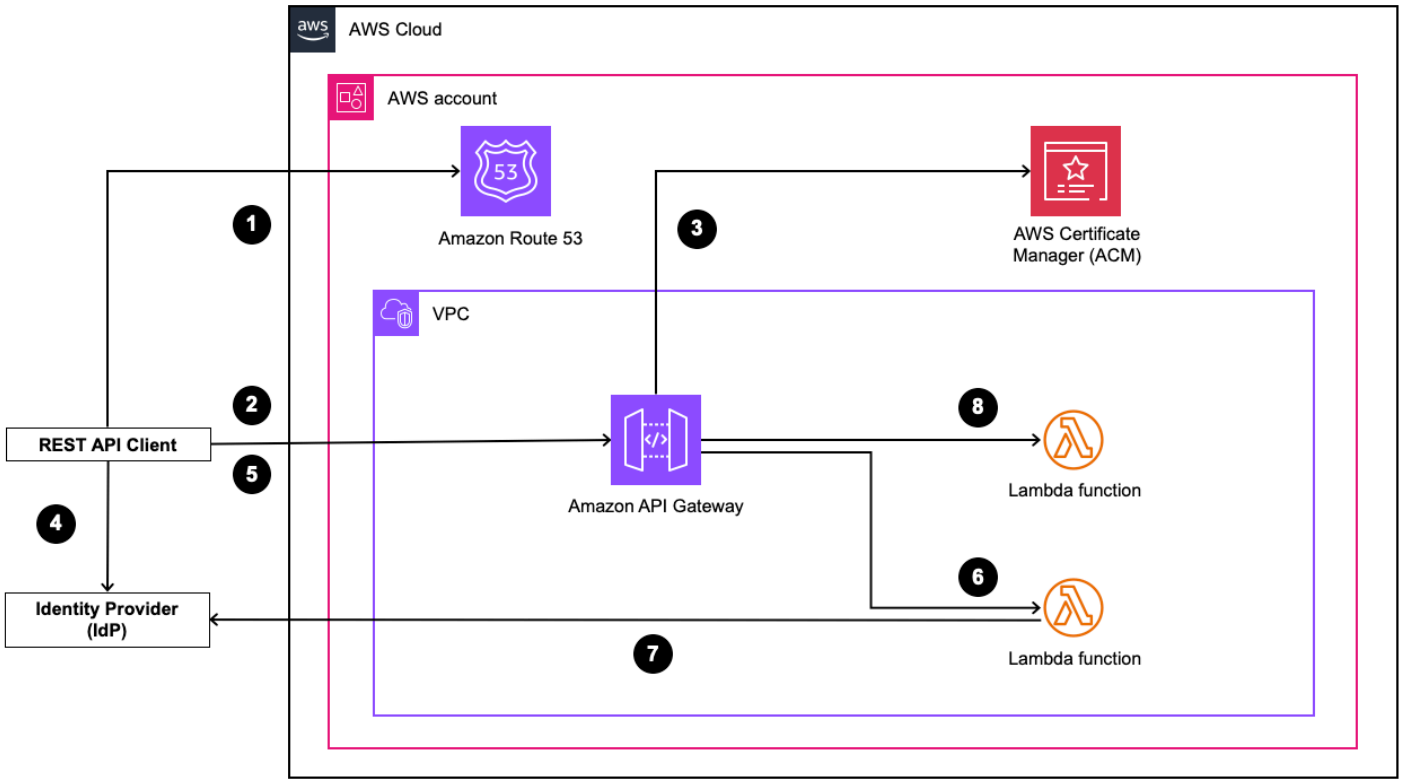
Interoperability — Agents from different vendors collaborate without custom integration

Scalability — Enterprise-wide agent discovery and delegation

Open Standards — Linux Foundation governance ensures long-term compatibility.

REST API

REST API



Data Flow - REST API

Step 1 — DNS Resolution (REST API Client → Amazon Route 53): The REST API Client resolves the API's custom domain name via Amazon Route 53, which serves as the DNS routing service. Route 53 returns the IP address associated with the API Gateway custom domain endpoint.

Step 2 — HTTPS Connection (REST API Client → Amazon API Gateway): The REST API Client establishes an HTTPS connection to Amazon API Gateway. The connection is secured using the SSL/TLS certificate managed by AWS Certificate Manager (ACM).

Step 3 — Certificate Association (Amazon Route 53 → AWS Certificate Manager): AWS Certificate Manager (ACM) provides the SSL/TLS certificate for the API Gateway's custom domain. This is a certificate association for HTTPS termination rather than a runtime API call — ACM ensures that all traffic between the client and API Gateway is encrypted in transit.

Step 4 — Authentication (REST API Client → Identity Provider): The REST API Client authenticates with the external Identity Provider (IdP) using an OIDC-compliant flow. For browser-based applications, this may involve a redirect-based authorization code flow. For machine-to-machine communication, this typically uses the client credentials grant. The IdP issues a JWT (JSON Web Token) / Bearer token upon successful authentication.

Step 5 — Authenticated Request (REST API Client → Amazon API Gateway): The REST API Client sends the API request to Amazon API Gateway with the JWT/Bearer token included in the `Authorization` header (e.g., `Authorization: Bearer <token>`). This is the primary API invocation carrying the business payload.

Step 6 — Token Validation (Amazon API Gateway → JWT/OIDC Authorizer Lambda): Amazon API Gateway invokes the Lambda authorizer (also known as a custom authorizer) to validate the JWT token. The Lambda authorizer is configured as a request-based or token-based authorizer that inspects the `Authorization` header.

Step 7 — JWKS Key Retrieval (JWT/OIDC Authorizer Lambda → Identity Provider): The JWT/OIDC Authorizer Lambda function calls the Identity Provider's `.well-known/openid-configuration` endpoint to retrieve the JSON Web Key Set (JWKS) containing the public keys. The authorizer uses these keys to verify the JWT token's signature, validate claims (issuer, audience, expiration), and generate an IAM policy that grants or denies access to the requested API resource.

Step 8 — Business Logic Execution (Amazon API Gateway → Backend Lambda): After the Lambda authorizer returns an "Allow" policy, Amazon API Gateway routes the request to the

backend Lambda function. This Lambda function executes the REST API business logic serverlessly, processing the request and returning the response through API Gateway back to the client.

Infrastructure Organization: All AWS resources are deployed within a single AWS account. The Amazon API Gateway and both Lambda functions operate within a Virtual Private Cloud (VPC) for network isolation and security. The VPC boundary (blue border with padlock icon) encompasses the core compute components. Amazon Route 53 and AWS Certificate Manager operate at the account level. The Identity Provider (IdP) is external to the AWS Cloud boundary.

Architecture Highlights:

- **Serverless Design:** Uses Lambda functions for scalable, event-driven compute with no infrastructure management
- **Zero-Trust Security:** Every request is authenticated (IdP) and authorized (Lambda authorizer) before reaching business logic
- **Certificate Management:** ACM automates SSL/TLS certificate provisioning and renewal for the custom domain
- **Standards-Based Authentication:** Leverages OIDC/OAuth 2.0 and JWT for interoperable, industry-standard authentication
- **Separation of Concerns:** Authentication (IdP), authorization (Lambda authorizer), and business logic (backend Lambda) are cleanly separated

Ontology Engineering REST API

Ontology Version Management REST API

Ontology Mapping Management (Source Schema → Ontology)

Rules Management (SHACL + SPARQL)

Semantic Metadata Management (Classes & Properties)

Semantic Search (Classes & Properties)

Metric Definition REST API

Supported MIME Types — Reference

Format	MIME Type	Use Case

Turtle	`text/turtle`	Default; human-readable, compact; preferred for ontology authoring and import/export
N-Quads	`application/n-quads`	Named graph support; recommended for versioned and provenance-aware exports
N-Triples	`application/n-triples`	Line-based, streaming-friendly; suited for large bulk exports
TriG	`application/trig`	Turtle + named graphs; natural fit for T-Box/A-Box separation in a single document
RDF/XML	`application/rdf+xml`	Legacy interoperability
JSON-LD	`application/ld+json`	Web/API-friendly; recommended for developer tooling and downstream API consumers
N3	`text/n3`	Superset of Turtle; includes rules support

Cross-Cutting Concerns

- **Content Negotiation:** All serialization endpoints support `Accept` (response format) and `Content-Type` (request body format) headers per the MIME type table above. Default response format is `text/turtle` for RDF resources and `application/json` for metadata/list endpoints. Returns `406 Not Acceptable` with a body listing supported types when an unsupported format is requested.

- **Named Graph Recommendations:** Use `application/n-quads` or `application/trig` for exports requiring named graph provenance (e.g., versioned ontology snapshots, T-Box/A-Box separation).
- **Authentication:** JWT/OIDC bearer token validated by API Gateway Lambda authorizer. Supports any OIDC-compliant identity provider including Amazon Cognito, Microsoft Entra ID, Okta, Auth0, Keycloak, and others. Required claims: `sub`, `groups`, `act`. Scope: `semantic-layer:access`.
- **API Versioning:** All endpoints versioned via path prefix `/v1/`.
- **Audit Logging:** All write operations emit structured events to CloudWatch Logs with `ontologyId`, `operation`, `principalId`, `contentType`, and `timestamp`.
- **Observability:** OpenTelemetry traces propagated across all API calls; spans include ontology IRI, operation type, serialization format, and latency.

Ontology Engineering

Ontology Engineering REST API

The following API surface defines the contract for Ontology Management & Knowledge Engineering UIs, services, and agents to interact with the Semantic Layer.

Operation	Method	Endpoint	Request Content-Type	Response Accept	Notes
Import Ontology	POST	`/ontologies`	`text/turtle`, `application/n-quads`, `application/n-triples`, `application/trig`, `application/rdf+xml`,	—	Multipart or body; format auto-detected from Content-Type

			`application/ld+json`		
Export Ontology	GET	`/ontologies/{id}/export`	—	`text/turtle` (default), `application/n-quads`, `application/n-triples`, `application/trig`, `application/rdf+xml`, `application/ld+json`	Returns `406 Not Acceptable` for unsupported types
Get Ontology	GET	`/ontologies/{id}`	—	`text/turtle`, `application/ld+json`	Returns ontology metadata + IRI
List Ontologies	GET	`/ontologies`	—	`application/json`	Supports pagination, filter by namespace/domain
Delete Ontology	DELETE	`/ontologies/{id}`	—	—	Soft-delete with version retention

Ontology Version Management

Ontology Version Management REST API

Operation	Method	Endpoint	Request Content-Type	Response Accept	Notes
List Versions	GET	`/ontologies/{id}/versions`	—	`application/json`	Returns version history with timestamps and authors
Get Version	GET	`/ontologies/{id}/versions/{versionId}`	—	`text/turtle` , `application/n-quads` , `application/trig`	Retrieve a specific version snapshot; N-Quads recommended for provenance-aware retrieval
Publish Version	POST	`/ontologies/{id}/versions/{versionId}/publish`	—	`application/json`	Publish ontology version
Diff Versions	GET	`/ontologies/{id}/versions/diff?from={v1}&to={v2}`	—	`application/json` , `text/turtle`	Returns added/removed/changed axioms

Rollback	POST	`/ontologies/{id}/versions/{versionId}/rollback`	—	`application/json`	Restore a prior version as active

Ontology Mapping Management (Source Schema → Ontology)

Ontology Mapping Management (Source Schema → Ontology)

Operation	Method	Endpoint	Request Content-Type	Response Accept	Notes
Create Mapping	POST	`/ontologies/{id}/mappings`	`text/turtle` (R2RML), `application/json`	`application/json`	Accepts R2RML (Turtle serialization) or custom mapping format
List Mappings	GET	`/ontologies/{id}/mappings`	—	`application/json`	Filter by source schema, target class
Get Mapping	GET	`/ontologies/{id}/mappings/{mappingId}`	—	`text/turtle`, `application/json`	
Update Mapping	PUT	`/ontologies/{id}/mappings/{`	`text/turtle`, `application/json`	`application/json`	

		`mappingId}`			
Delete Mapping	DELETE	`/ontologies/{id}/mappings/{mappingId}`	—	—	
Validate Mapping	POST	`/ontologies/{id}/mappings/{mappingId}/validate`	—	`application/json`	Dry-run against source schema

Rules Management

Rules Management (SHACL + SPARQL)

Operation	Method	Endpoint	Request Content-Type	Response Accept	Notes
Import Rules	POST	`/ontologies/{id}/rules`	`text/turtle` (SHACL Shapes Graph), `application/n-quads`	`application/json`	Accepts SHACL Shapes Graph (Turtle) or SPARQL CONSTRUCT/ SPIN
Export Rules	GET	`/ontologies/{id}/rules/export`	—	`text/turtle` (default), `application/n-quads`	Turtle for SHACL Shapes Graph; N-Quads

					for rule provenance with named graph context
List Rules	GET	`/ontologies/{id}/rules`	—	`application/json`	Filter by type: `shacl`, `sparql`, `swrl`
Get Rule	GET	`/ontologies/{id}/rules/{ruleId}`	—	`text/turtle`, `application/json`	
Update Rule	PUT	`/ontologies/{id}/rules/{ruleId}`	`text/turtle`, `application/json`	`application/json`	
Delete Rule	DELETE	`/ontologies/{id}/rules/{ruleId}`	—	—	
Validate Rules	POST	`/ontologies/{id}/rules/validate`	—	`application/json`	Run SHACL validation against a named graph; returns violation report

Semantic Metadata Management (Classes & Properties)

Semantic Metadata Management (Classes & Properties)

Operation	Method	Endpoint	Request Content-Type	Response Accept	Notes
Update Class Metadata	PATCH	`/ontologies/{id}/classes/{classIRI}`	`application/json` , `text/turtle`	`application/json`	Set `rdfs:comment` , `rdfs:label` , `skos:altLabel` , `skos:definition`
Update Property Metadata	PATCH	`/ontologies/{id}/properties/{propertyIRI}`	`application/json` , `text/turtle`	`application/json`	Same metadata vocabulary as class metadata
Get Class	GET	`/ontologies/{id}/classes/{classIRI}`	—	`text/turtle` , `application/ld+json` , `application/json`	Returns full axiom set + metadata
List Classes	GET	`/ontologies/{id}/classes`	—	`application/json`	Filter by namespace, superclass
List Properties	GET	`/ontologies/{id}/p`	—	`application/json`	Filter by domain, range, type

		roperties			(owl:ObjectProperty , owl:DatatypeProperty)

Semantic Search (Classes & Properties)

Semantic Search (Classes & Properties)

Operation	Method	Endpoint	Request Content-Type	Response Accept	Notes
Keyword Search	GET	/ontologies/{id}/search?q={term}	—	application/json	Matches rdfs:label , skos:altLabel , rdfs:comment
Semantic Similarity Search	POST	/ontologies/{id}/search/semantic	application/json	application/json	Vector-based search using embedding of class/property descriptions
Subclass Traversal Search	GET	/ontologies/{id}/classes/{classIRI}/subclasses	—	application/json , text/turtle	Returns full subclass hierarchy ; ?depth= param for bounded traversal

Combined Search	POST	`/ontologies/{id}/search/hybrid`	`application/json`	`application/json`	Combines subclass structure + semantic similarity; returns ranked results with `rdfs:subClassOf` path

Metric Definition REST API

Operation	Method	Endpoint	Request Content-Type	Response Accept	Notes
List Metrics	GET	/ontologies/{id}/versions/{versionId}/metrics	-	application/json	List all metric definitions for an ontology version
Get Metric	GET	/ontologies/{id}/versions/{versionId}/metrics/{metricId}	-	text/turtle	
Create Metric	POST	/ontologies/{id}/versions/{versionId}/metrics	text/turtle	-	application/json

Update Metric	PUT	/ontologies/{id}/versions/{versionId}/metrics/{metricId}	text/turtle	-	application/json
Delete Metric	DELETE	/ontologies/{id}/versions/{versionId}/metrics/{metricId}	-	application/json	Soft-delete; governed by approval workflow
Validate Metric	POST	/ontologies/{id}/versions/{versionId}/metrics/{metricId}/validate	-	application/json	Test pre-compiled expression against sample data via VKG
Publish Metric	POST	/ontologies/{id}/versions/{versionId}/metrics/{metricId}/publish	-	application/json	Promote from draft to published

Intelligence Layer

Intelligence Layer Overview

The **Intelligence Layer** serves as the cognitive foundation of the architecture, providing automated capabilities for ontology lifecycle management, knowledge quality assurance, and contextual

reasoning that bridge the gap between raw data and actionable semantic knowledge. This layer orchestrates the continuous evolution and validation of the semantic fabric, ensuring that ontologies remain aligned with business reality, knowledge graphs reflect source data truth, and agents operate with contextually grounded, episodic memory.

Ontology Induction enables the automated construction and enrichment of ontologies through multiple complementary approaches. **Bottom-up induction** analyzes business and technical data catalogs (AWS Glue Data Catalog, Amazon SageMaker Catalog / DataZone, DataHub, Collibra), structured and semi-structured data sources (relational schemas, JSON/XML structures, Parquet metadata), controlled vocabularies and taxonomies extracted from documents (via the Data Layer's information extraction workflows), and LLM-guided inputs from specifications and domain expert interviews to discover entity types, relationships, and constraints present in existing data. **Top-down integration** incorporates established open-source upper ontologies (e.g., BFO, DOLCE) and domain-specific ontologies (e.g., SNOMED CT for healthcare, FIBO for financial services) to provide foundational semantic structures and industry-standard vocabularies. This hybrid approach — combining data-driven discovery with expert-curated knowledge — produces ontologies that are both grounded in organizational reality and aligned with industry best practices, accelerating time-to-value while ensuring semantic interoperability.

Ontology Evaluation and Completeness Assessment continuously analyzes ontologies against available data sources and broader business requirements to identify gaps, inconsistencies, and opportunities for enrichment. The evaluation process compares the ontology's coverage (classes, properties, constraints) against actual data distributions, entity types present in source systems, and business concepts referenced in documentation or specifications. Industry ontologies like FIBO (Financial Industry Business Ontology) or HL7 FHIR (healthcare) serve as benchmarks, highlighting missing concepts or relationships that would enhance semantic expressiveness and enable broader use cases. Automated metrics track ontology completeness, consistency (via ELK/HermiT reasoners), and alignment with data catalogs, surfacing recommendations for ontology engineers and domain experts.

Truth Maintenance ensures that knowledge materialized in the Semantic Layer Graph Database remains synchronized with source data, preventing semantic drift and maintaining trust in the knowledge graph. Leveraging provenance metadata stored with every triple (source system, extraction timestamp, confidence score), the Truth Maintenance Service monitors source data changes via event streams (Neptune DB Streams, Kafka, AWS EventBridge) and triggers validation workflows when discrepancies are detected. For virtualized knowledge accessed through the VKG Service, truth is maintained by design — queries always retrieve current data from source systems.

For materialized knowledge in Neptune, the service reconciles assertions against source data, flags stale or contradictory facts, and triggers re-extraction or reasoning workflows to restore consistency. This closed-loop validation ensures that agents and applications operate on verified, current knowledge.

Short-Term Memory and Context Graphs provide episodic, session-scoped knowledge that grounds agent reasoning in specific conversational contexts, user interactions, or workflow executions. These context graphs — stored as named graphs in Neptune or in-memory graph structures — capture entities, relationships, and events relevant to a specific agent session (e.g., a Q&A conversation, a business elicitation interview, an ontology maintenance task). Context graphs are **ontology-grounded** where appropriate, linking session-specific entities to the broader Semantic Layer ontology to enable reasoning and knowledge reuse. For example, a Q&A Agent's context graph might include the user's query history, retrieved document chunks, and inferred user intent — all linked to ontology classes like `Query`, `Document`, `Intent`. These context graphs inform agents executing in the Orchestration Layer, including **Question & Answer (Q&A) Agents** (retrieving and synthesizing knowledge), **Business Elicitation Agents** (capturing requirements and domain knowledge from stakeholders), and **Ontology Maintenance Agents** (performing entity resolution, disambiguation, and ontology refinement based on usage patterns).

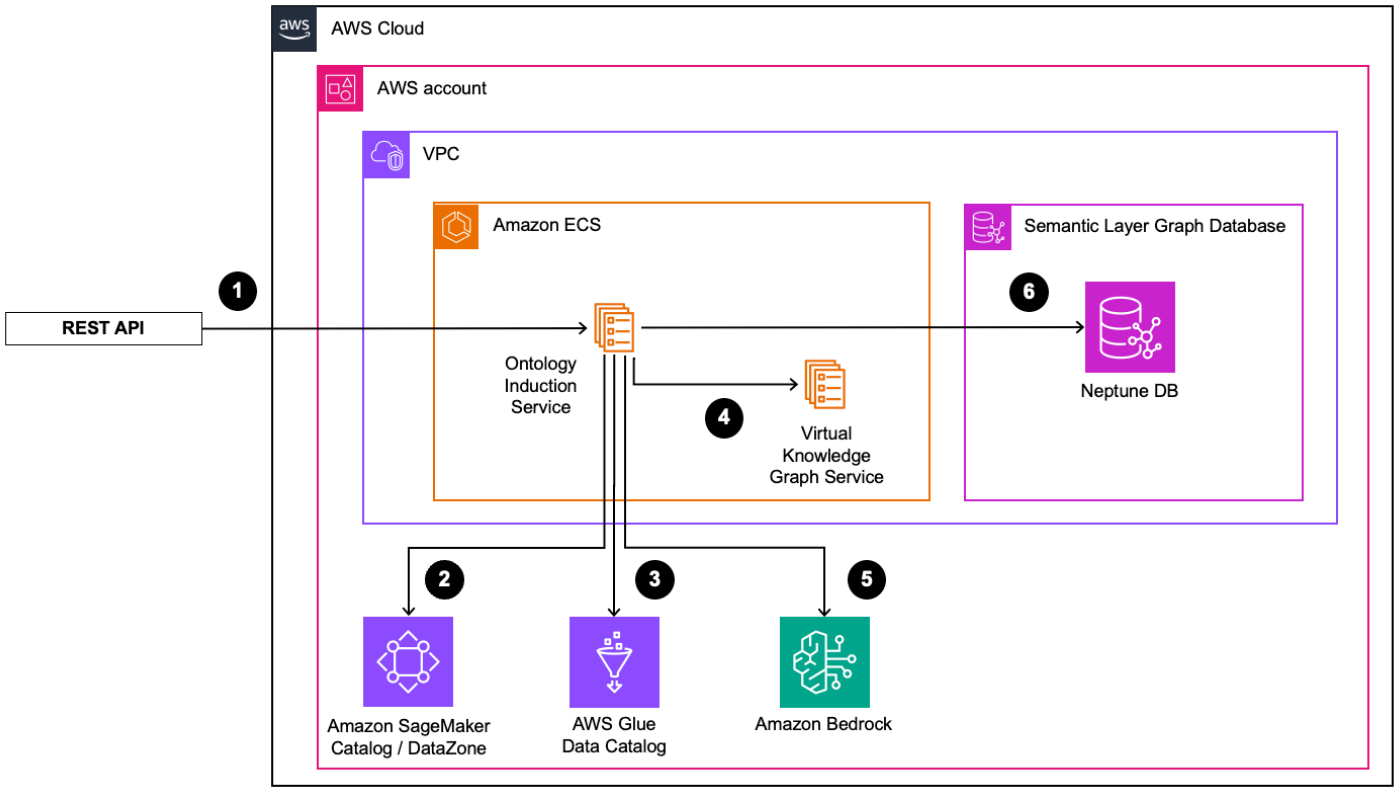
Intelligence Layer Capabilities

Capability	Purpose	Key Techniques
Ontology Induction	Automated ontology construction from data and domain knowledge	Bottom-up (schema analysis, LLM-guided extraction), Top-down (upper/domain ontologies)
Ontology Evaluation	Assess ontology completeness and alignment with data/business needs	Coverage metrics, industry ontology benchmarking (FIBO, SNOMED), gap analysis
Truth Maintenance	Ensure materialized knowledge reflects source data	Provenance tracking, change data capture, reconciliation workflows

Context Graphs	Episodic, session-scoped memory for agent reasoning	Named graphs, ontology-grounded entities, session lifecycle management

This Intelligence Layer enables **self-improving semantic systems** that continuously refine ontologies based on data evolution, maintain knowledge graph fidelity through automated validation, and provide agents with contextual, episodic memory — supporting autonomous, knowledge-driven decision-making across the enterprise.

Ontology Induction



Data Flow — Ontology Induction

1. REST API → Ontology Induction Service (Amazon ECS) A client initiates the ontology induction workflow via a **REST API** endpoint, invoking the **Ontology Induction Service** deployed as containerized tasks on **Amazon ECS** (with **Fargate** or **EC2** instances) within the VPC. The API accepts parameters specifying:

- Induction approach (bottom-up from catalogs, top-down from reference ontologies, LLM-guided from specifications, or hybrid)
- Target data sources (SageMaker Catalog, Glue Data Catalog, specific databases or schemas)
- Ontology scope and namespace
- Quality thresholds and validation requirements

The request is authenticated via JWT/OIDC bearer tokens validated by the AgentCore Gateway, and the ECS service scales horizontally based on induction workload complexity. The Ontology Induction Service implements **various techniques** including **LLM-based semantic enrichment** via Amazon Bedrock and **vector-based similarity techniques** such as **OWL2Vec*** (generates vector embeddings from OWL ontologies for similarity computation) and **RDF2Vec** (generates vector embeddings from RDF graphs using random walks, enabling semantic similarity matching between discovered entities and reference ontologies).

2. Ontology Induction Service → Amazon SageMaker Catalog / DataZone Based on input parameters and availability of business data catalog information, the service inspects and analyzes **business glossary terms and metadata** from **Amazon SageMaker Catalog** and **Amazon DataZone**. The service retrieves:

- **Business glossary terms** — Curated business definitions, synonyms, and entity relationships from DataZone's business glossary
- **Data assets** — Datasets, features, model artifacts registered in SageMaker Catalog with business context
- **Domain context** — DataZone domain memberships, data product classifications, and semantic tags applied by data stewards
- **Data quality metrics** — Completeness, uniqueness, validity scores, and data profiling statistics from SageMaker Catalog
- **Lineage information** — Upstream and downstream data flows, transformations, and dependencies

This metadata provides the **business-oriented view** of data assets, capturing how data is understood and used within the organization. The service analyzes naming conventions, glossary term associations, and lineage patterns to identify candidate ontology classes (e.g., `Customer`, `Claim`, `Transaction`) and relationships.

3. Ontology Induction Service → AWS Glue Data Catalog The service retrieves **technical data catalog information** from **AWS Glue Data Catalog**, analyzing database objects, tables, views, and associated columns. The service queries:

- **Database and table schemas** — Column names, data types, constraints (primary keys, foreign keys), nullability, and cardinality
- **Crawled metadata** — Statistics, discovered patterns, data classifications, and partitioning schemes
- **Schema evolution history** — Version history of table schemas to understand how data structures have evolved
- **Glue Schema Registry** — Avro, Protobuf, or JSON schemas for streaming data sources
- **Source system mappings** — Connection metadata linking Glue tables to source databases (RDS, Redshift, Snowflake, S3)

The service applies **Ontology-Based Data Access (OBDA) mapping patterns** to generate candidate R2RML mappings:

- **Tables → Ontology Classes** (e.g., `claims` table → `ex:Claim` class via R2RML `rr:TriplesMap`)
- **Columns → Ontology Properties** (e.g., `claim_amount` column → `ex:claimAmount` datatype property via `rr:PredicateObjectMap`)
- **Foreign Keys → Object Properties** (e.g., `customer_id` FK → `ex:submittedBy` object property linking `Claim` to `Customer` via `rr:RefObjectMap`)
- **Primary Keys → Entity IRIs** (e.g., `claim_id` → `ex:Claim/{claim_id}` via `rr:SubjectMap` with IRI template)

These R2RML mappings define how relational data is virtualized as RDF, enabling the VKG Service to translate SPARQL queries into optimized SQL without data duplication. The mappings are generated based on schema analysis and will be refined in subsequent steps using sample data and LLM enrichment.

Optionally, technical data catalog information is linked to business data catalog information (if available from Step 2), reconciling technical schemas with business glossary terms to create a unified view. For example, a Glue table column `cust_id` might be linked to a DataZone business term "Customer Identifier" with a formal definition and business owner, and this business context is incorporated into the R2RML mapping annotations.

4. Ontology Induction Service → Virtual Knowledge Graph (VKG) Service The Ontology Induction Service calls the **Virtual Knowledge Graph (VKG) Service** to retrieve **existing ontologies** and **sample data** from both the **Semantic Layer Graph Database** (Neptune DB) and **Data Layer sources**. This step enables the induction process to:

- **Retrieve existing ontologies** — Query Neptune for T-Box definitions (classes, properties, hierarchies, axioms) from customer-provided ontologies, previously induced ontologies, or integrated reference ontologies (SNOMED CT, FIBO, BFO, DOLCE)
- **Retrieve existing R2RML mappings** — Query Neptune for deployed R2RML mappings to understand how existing ontology classes and properties are currently mapped to Data Layer sources, ensuring consistency with established OBDA patterns
- **Sample materialized data** — Execute SPARQL queries against Neptune to retrieve sample instances (A-Box) of existing classes, understanding how ontology concepts are currently used in the knowledge graph
- **Sample virtualized data** — Query the VKG Service to retrieve sample data from Data Layer sources (via existing R2RML mappings) without moving data, observing actual data distributions, value ranges, and relationship patterns in relational databases, data warehouses, and data lakes
- **Validate catalog-to-data alignment** — Compare catalog metadata (from Steps 2 and 3) with actual sampled data to detect schema drift, verify foreign key relationships, validate data type mappings, and refine candidate R2RML mappings

The VKG Service federates these queries across Neptune (materialized graphs in Semantic Layer Graph Database) and Data Layer sources (Snowflake, Redshift, Athena, S3 Tables/Iceberg), returning unified RDF results. This provides the Ontology Induction Service with a **comprehensive, data-grounded view** of the current semantic landscape, enabling it to:

- Avoid duplicating existing ontology concepts
- Align newly induced classes and properties with established patterns
- Validate that catalog metadata accurately reflects actual data
- Generate realistic R2RML mappings based on observed data structures and existing OBDA patterns
- Ensure consistency with the VKG Service's query translation and federation capabilities

5. Ontology Induction Service → **Amazon Bedrock** Business and technical data catalog information (Steps 2-3), combined with existing ontologies, R2RML mappings, and sample data retrieved from the VKG Service (Step 4), are analyzed and **linked to existing ontologies** (e.g., customer-provided ontologies, open-source upper ontologies like BFO or DOLCE, industry domain ontologies like SNOMED CT for healthcare or FIBO for financial services) using **Amazon Bedrock** foundation models. The Ontology Induction Service invokes Bedrock to:

- **Semantic enrichment** — Generate `rdfs:label`, `rdfs:comment`, `skos:definition`, and `skos:altLabel` annotations for classes and properties based on catalog metadata, column names, business glossary terms, and sample data values
- **Entity disambiguation** — Resolve ambiguous terms (e.g., "status" could be order status, claim status, or account status) using context from lineage, domain metadata, business definitions, and existing ontology patterns
- **Ontology alignment** — Map discovered entity types to concepts in existing ontologies (retrieved in Step 4) using **vector-based similarity techniques** (OWL2Vec*, RDF2Vec) to compute semantic similarity between candidate classes and reference ontology concepts
- **Axiom generation** — Propose domain/range constraints, cardinality restrictions, disjointness axioms, and property characteristics based on data patterns, foreign key relationships, sample data distributions, and business rules
- **R2RML mapping refinement** — Enhance candidate R2RML mappings (from Step 3) with semantic context, optimizing IRI templates, adding conditional mappings based on data values, and incorporating business logic from glossary terms
- **Specification extraction** — Parse technical specifications, data dictionaries, requirements documents, or domain expert interviews (text/document inputs) to extract entity types, relationships, and constraints
- **Consistency validation** — Use LLM reasoning to validate that newly induced ontology elements and R2RML mappings are consistent with existing ontology patterns, OBDA best practices, and domain conventions

This hybrid approach combines **bottom-up discovery** (from catalogs and schemas), **data-grounded validation** (from VKG-sampled data), and **top-down integration** (from existing and reference ontologies), producing ontologies and R2RML mappings that are both aligned with organizational data reality and consistent with established semantic patterns and industry best practices.

6. Ontology Induction Service → Neptune DB (Semantic Layer Graph Database) Induced ontologies are added to the **Semantic Layer Graph Database** (Neptune DB). The service writes:

- **T-Box (Ontology/Schema)** — OWL classes, object properties, datatype properties, class hierarchies (`rdfs:subClassOf`), property domains and ranges, and axioms
- **Annotations** — `rdfs:label`, `rdfs:comment`, `skos:definition`, `skos:altLabel` for all classes and properties
- **Provenance metadata** — Links back to source catalogs (SageMaker Catalog, Glue Catalog), source tables/columns, existing ontologies referenced, induction method (catalog-driven, LLM-assisted, vector similarity), confidence scores, and timestamps
- **SHACL constraints** — Data quality and validation rules derived from schema metadata, data statistics, and LLM-generated business rules
- **R2RML mappings** — R2RML mappings linking the induced ontology classes and properties to source tables and columns (informed by sample data retrieved via VKG Service in Step 4 and refined by Bedrock in Step 5), enabling immediate virtualization via the VKG Service. These mappings follow OBDA best practices and are consistent with the Knowledge Layer's VKG architecture.

Governance is integral — versioning and workflow management ensure quality and control:

- **Draft Mode** — Initially induced ontologies and updates to existing ontologies are written to **draft named graphs** in Neptune (e.g., `ex:ontology/v2.0-draft`) and flagged with `ex:status "DRAFT"` annotations. Draft ontologies and R2RML mappings are not used by VKG Service or Reasoning Service workflows.
- **Review and Approval** — Draft ontologies must be reviewed and promoted using a **customer-defined approval process**. Approval workflows may include:
  - **Human approvals** — Requiring a specified number of ontology engineers, domain experts, or data stewards to review and approve changes via Ontology Engineering UIs, Git pull requests, or workflow management systems (ServiceNow)
  - **Automated evaluation techniques** — **LLM as a Judge** patterns where foundation models assess ontology quality, completeness, consistency, and alignment with business requirements before approval
  - **Reasoning validation** — ELK or HermiT reasoners validate consistency and detect unsatisfiable classes before promotion

  - **Coverage metrics** — Automated checks ensure the ontology covers a minimum percentage of catalog entities and relationships
  - **R2RML validation** — Test queries validate that R2RML mappings produce correct RDF output and that the VKG Service can successfully translate SPARQL queries to SQL
- **Versioning** — Each approved ontology is assigned a semantic version (e.g., `v2.0`) and stored in a versioned named graph. S3 Versioned Buckets or Git repositories maintain version history with rollback capabilities. Neptune named graphs enable side-by-side deployment of ontology versions for A/B testing or gradual migration.
- **Promotion to Published** — Once approved, the ontology is promoted from draft to published status, triggering updates to the VKG Service (R2RML mappings), Reasoning Service (ontology reloads), and downstream consumers. Neptune DB Streams emit events for the newly promoted ontology, notifying subscribers.

Ontology Induction Strategies

Strategy	Data Sources	Key Techniques
Bottom-Up (Catalog-Driven)	SageMaker Catalog, Glue Data Catalog, DataZone	Schema analysis, foreign key inference, lineage analysis, statistical profiling, R2RML mapping generation
Data-Grounded Validation	VKG Service (Neptune + Data Layer sample data)	Sample data analysis, schema drift detection, relations hip validation, value range inspection, R2RML mapping validation
Top-Down (Reference Ontologies)	SNOMED CT, FIBO, BFO, DOLCE, existing ontologies in Neptune	Ontology alignment , semantic similarity (OWL2Vec*, RDF2Vec), concept mapping
LLM-Guided (Specifications)	Requirements docs, interview s, business glossaries	NER, relationship extractio n, constraint generation via

		Bedrock, R2RML mapping refinement
Hybrid	All of the above	Reconciliation, consensus -based entity resolution, provenance tracking, OBDA pattern consistency

Cross-Cutting Concerns

- **VKG Service Integration** — By calling the VKG Service (Step 4), the Ontology Induction Service gains access to both existing ontologies and live/sample data from across the enterprise without needing direct connections to Neptune or Data Layer sources. This abstraction simplifies the induction architecture and ensures ontology induction uses the same federated query capabilities as agents and applications. The service also retrieves existing R2RML mappings to ensure consistency with established OBDA patterns.
- **Scalability** — ECS tasks auto-scale based on the number of data sources and ontology complexity. Bedrock API calls and vector similarity computations are parallelized across multiple catalog entities for efficient processing. VKG Service queries are optimized to sample representative data without overwhelming source systems.
- **Quality Assurance** — Induced ontologies are validated against consistency checks (ELK reasoner), completeness metrics (coverage vs. catalog), alignment with existing ontologies, and sample data patterns before being promoted to published. R2RML mappings are tested to ensure correct RDF output and successful SPARQL-to-SQL translation.
- **Human-in-the-Loop** — Ontology engineers review induced ontologies via Ontology Engineering UIs, accepting, rejecting, or refining suggested classes, properties, and R2RML mappings before final approval. Draft mode ensures systems are not impacted by unvalidated ontologies.
- **Versioning & Governance** — Each induction run produces a versioned ontology stored in a named graph with draft/published status flags. Approval workflows (human + automated) gate promotion to published. S3 Versioned Buckets or Git repositories provide version control and rollback capabilities.
- **Observability** — OpenTelemetry traces propagate from REST API → ECS → Catalog queries → VKG Service queries → Bedrock invocations → Neptune write, providing end-to-end visibility via AWS X-Ray. CloudWatch Logs capture entity counts, LLM prompts/responses, vector similarity

scores, sample data statistics, R2RML mapping generation details, and induction confidence scores.

- **Cost Optimization** — ECS Fargate Spot reduces compute costs. Bedrock pay-per-token pricing is optimized by batching catalog metadata into structured prompts. Glue Catalog and SageMaker Catalog API calls are cached to minimize redundant queries. VKG Service sampling queries are limited to representative subsets of data. Vector similarity computations are performed only when aligning with reference ontologies.

This Ontology Induction architecture enables **rapid, automated, data-grounded ontology bootstrapping** from existing data infrastructure — reducing the time and effort required to build semantic foundations while ensuring alignment with organizational data assets, existing semantic patterns, actual data distributions, industry standards, OBDA best practices, and governance requirements. The draft-review-promote workflow ensures quality and control while accelerating time-to-value.

Knowledge Layer

Knowledge Layer Overview

The **Knowledge Layer** serves as the semantic intelligence foundation of the architecture, enabling unified querying across heterogeneous data sources through a Virtual Knowledge Graph (VKG) approach. At its core, the layer exposes SPARQL query capabilities that allow applications, agents, and users to interact with knowledge as a unified graph — regardless of whether that knowledge resides in a dedicated graph database, relational databases, data warehouses, or data lakes.
The VKG Service, deployed on **Amazon ECS** with **Fargate** or **EC2** instances, orchestrates query translation and federation, enabling seamless access to both materialized semantic knowledge and virtualized data layer sources within a single SPARQL query. **Ontop** is a strong choice for implementing the VKG Service — it's an open-source platform purpose-built for OBDA (Ontology-Based Data Access) that translates SPARQL queries over ontology-defined knowledge graphs into optimized SQL executed directly against relational sources, using R2RML mappings and lightweight OWL ontologies, with no data duplication required.

Amazon Neptune Database serves as the persistent graph store for knowledge that cannot or should not remain in the data layer, including entities, relationships, and propositions extracted from unstructured documents (PDFs, images, reports) and mapped to ontology classes and properties via the Information Extraction pipeline, deduction results that infer new relationships or properties through symbolic reasoning, cross-domain associations linking entities across disparate

data sources, and enriched semantic metadata. Together, these represent the materialized A-Box (instance data) of the knowledge graph, capturing both directly extracted and logically derived facts as first-class graph citizens. Neptune's native RDF and SPARQL support makes it the natural home for the T-Box (ontology/schema) and this materialized A-Box, which require graph-native reasoning and traversal. Neptune DB Streams provide near real-time change data capture for all create, update, and delete operations, enabling event-driven architectures where downstream processes, such as the Symbolic Reasoning Service, can react immediately to knowledge graph mutations, maintaining consistency and triggering inference workflows without polling.

The VKG architecture's key strength lies in its ability to query the **Semantic Layer Graph Database** (Neptune), **relational databases and data warehouses** (via JDBC-connected sources like Snowflake, Redshift, Athena), and **data lakes** (Iceberg/Parquet on S3) — either independently or federated together in a single query. This eliminates data duplication while preserving the semantic richness of the knowledge graph, allowing the Knowledge Layer to serve as a true abstraction over the enterprise's distributed data landscape. Ontology-driven mappings (R2RML) define how relational and tabular data sources are virtualized as RDF, enabling SPARQL queries to transparently span both materialized and virtual knowledge without requiring applications to understand the underlying physical storage topology.

The **Symbolic Reasoning Service** maintains knowledge graph consistency, validates data quality, and derives new knowledge automatically through event-driven workflows triggered by Neptune DB Streams. Deployed on Amazon ECS, the service implements a modular ensemble architecture supporting multiple reasoning engines including **ELK** (lightweight OWL reasoning), **HermiT** (full OWL 2 DL expressivity), **SHACL** validators (constraint enforcement), and **SWRL** engines (rule-based inference). When graph mutations occur, the Symbolic Reasoning Service identifies applicable rules, retrieves necessary sub-graphs from the VKG Service (federating across Neptune and Data Layer sources), performs reasoning, and persists inferred triples, validation results, and consistency flags back to Neptune with full provenance metadata. This closed-loop architecture ensures that ontology changes, new assertions, and constraint violations are automatically validated and that inferred knowledge is materialized in near real-time, supporting autonomous agents with verified, explainable knowledge.

The Knowledge Layer includes a **Metric Store** — a governed registry of reusable, ontology-grounded metric definitions persisted alongside the T-Box, R2RML mappings, and SHACL constraints in the Semantic Layer Graph Database.

Metric Definition: A reusable, pre-defined calculation bound to an ontology class and its datatype properties. Each definition specifies an aggregation function, the properties involved, optional

filters, and a pre-compiled SPARQL/SQL expression resolved deterministically at query time — no LLM required.

```
ex:ProcessingTime a ex:MetricDefinition ;
    rdfs:label "Processing Time" ;
    ex:metricForClass ex:Claim ;
    ex:aggregationFunction "AVG" ;
    ex:metricExpression "AVG(DATEDIFF('day', ?submissionDate, ?approvalDate))" ;
    ex:metricUnit "days" .
```

Key properties:

- **Ontology-grounded** — metrics reference classes and properties, not physical tables, enabling resolution across federated sources via the VKG
- **Deterministic** — pre-compiled expressions are injected at query time without LLM generation, supporting ACE principles (Accuracy, Consistency, Explainability)
- **Governed** — metric definitions follow the same draft → review → publish lifecycle as ontology versions, validated by SHACL shapes
- **Agent-discoverable** — the Q&A Agent resolves recognized metric names directly from the Metric Store during query generation, bypassing LLM-based calculation synthesis

Virtual Knowledge Graph (VKG) Query

Data Flow - Virtual Knowledge Graph (VKG) Query

1. SPARQL Client → Virtual Knowledge Graph Service A SPARQL Client (application, agent, or user interface) submits a SPARQL query to the **Virtual Knowledge Graph (VKG) Service** deployed on **Amazon ECS** within the VPC. The VKG Service endpoint accepts the query via HTTPS, receiving the JWT/OIDC bearer token propagated by the AgentCore Gateway. The VKG Service extracts user identity (`sub`), authorization groups (`groups`), and agent identity (`act`) from the token for row-level security enforcement and audit logging. The VKG Service (implemented using Ontop or similar OBDA platform) parses the SPARQL query and consults the loaded ontology (T-Box) and R2RML mappings to determine the query execution plan.

2. VKG Service Query Planning & Federation The VKG Service analyzes the SPARQL query structure and determines which data sources are required to satisfy the query — materialized knowledge in **Neptune DB** (Step 3), virtualized data from relational/data warehouse sources (Step 4), or a federated combination of both. For triple patterns that correspond to ontology-mapped

relational tables, the VKG Service translates the SPARQL query into optimized SQL using the R2RML mappings. For triple patterns representing deduced knowledge, cross-domain relationships, or ontology axioms, the query is routed to Neptune DB via SPARQL endpoint. The VKG Service applies query optimization techniques including **predicate pushdown**, **join reordering**, and **filter propagation** to achieve locality optimization and minimize data transfer from sources.

3. VKG Service → Neptune DB (Semantic Layer Graph Database) For SPARQL query fragments targeting materialized knowledge, the VKG Service issues a SPARQL query to **Amazon Neptune DB** via the Neptune SPARQL endpoint. Neptune executes the query against the stored RDF graph, which includes the T-Box (ontology/schema), materialized A-Box (instance data), deduction results from reasoning workflows, and cross-domain entity linkages. Neptune enforces **access control policies** at query execution time, ensuring users only retrieve authorized data. Neptune returns RDF bindings (for SELECT queries) or RDF graphs (for CONSTRUCT/DESCRIBE queries) to the VKG Service. Neptune's native support for SPARQL 1.1, named graphs, and OWL inference ensures semantic fidelity in query results.

4. VKG Service → Data Layer (Virtualized Query Execution via JDBC) For SPARQL query fragments mapped to relational or data warehouse sources, the VKG Service translates the SPARQL into native SQL and dispatches queries to the appropriate **Data Layer** sources via **JDBC connections**. The VKG Service retrieves tabular result sets from these sources, converts rows and columns to RDF bindings using the R2RML mapping definitions, and integrates them with results from Neptune (Step 3) if the query is federated. SQL queries are executed directly against the source systems — no data is duplicated or moved into Neptune unless explicitly materialized by a separate ETL or reasoning process.

5. Data Layer Sources Execute SQL Queries Each data source in the **Data Layer** executes the translated SQL query against its native storage. Supported sources include:

- **Amazon Athena** — queries Iceberg or Parquet tables on **S3 Tables** using Presto/Trino query engine, with federated access to **DynamoDB** and other AWS databases
- **Amazon Redshift** — executes queries against its columnar data warehouse
- **Snowflake** — executes queries within its cloud platform
- **Databricks** — executes queries within its lakehouse platform

VPC peering or AWS PrivateLink ensures secure connectivity back to the VKG Service. Query results are returned as standard JDBC result sets, which the VKG Service consumes, transforms to RDF using R2RML mappings, and merges with Neptune-sourced results for the final SPARQL response.

6. VKG Service Returns Unified SPARQL Results to Client The VKG Service merges results from Neptune (Step 3) and Data Layer sources (Steps 4–5), applies any final SPARQL query operators (e.g., UNION, OPTIONAL, FILTER), and returns a unified SPARQL result set to the SPARQL Client. Results are serialized in the requested format — SPARQL JSON, XML, or Turtle — based on the client's `Accept` header. The VKG Service caches query plans and frequently accessed R2RML mappings to improve performance for repeated queries. OpenTelemetry traces are emitted across all steps, capturing query latency, source attribution, and row counts for observability.

Cross-Cutting Concerns

- **Authentication & Authorization:**

  Inbound Authentication - JWT/OIDC bearer tokens authenticate inbound SPARQL queries. The AgentCore Gateway validates tokens before forwarding to the VKG Service. Required JWT claims: `sub` (user identity), `groups` (authorization), `act` (agent identity).

  Service-Level Authorization - IAM execution roles for the ECS tasks grant least-privilege access to Neptune, Athena, Redshift, and Secrets Manager (for JDBC credentials).

- **Row-Level & Column-Level Security** — The VKG Service enforces row-level security (RLS) and column-level security (CLS) across all federated data sources, adapting the enforcement mechanism to the capabilities of each source:
  - **Neptune DB (Semantic Layer Graph Database)** — The VKG Service enforces RLS through SPARQL query rewriting, injecting FILTER clauses or named graph restrictions based on the user's `groups` claim and authorization policies (OPA, Lake Formation, or ontology-defined access rules). Data masking and redaction of sensitive attributes (PII, PHI) are applied based on user roles.
  - **AWS Data Layer Sources (Amazon Athena, Amazon Redshift)** — The VKG Service leverages AWS Trusted Identity Propagation (TIP) via IAM Identity Center, propagating the end-user's IdP identity to enable Lake Formation fine-grained access controls and native RLS/CLS at the source. This approach delegates security enforcement to the data platform, ensuring that access policies are evaluated natively and consistently with the organization's Lake Formation governance model.
  - **Non-AWS Data Layer Sources with Native Security (e.g., Snowflake Secure Views)** — For data platforms that support session-based or user-context security, the VKG Service establishes per-user sessions using OAuth token propagation or session-level user context variables. The data platform enforces RLS/CLS natively — for example, Snowflake Secure Views evaluate row-filtering and column-masking policies based on the session's user context,

ensuring that security enforcement occurs within the source system without exposing underlying table structures.

- **Data Layer Sources without Native Identity Propagation** — For sources that do not support identity propagation or session-based security, the VKG Service falls back to query rewriting — adding one or more predicates to restrict row selection and/or limiting column projection in the translated SQL query based on the user's authorization context. This approach ensures consistent RLS/CLS enforcement regardless of the source system's native security capabilities.

Authorization policies are defined centrally — in the ontology, AWS Lake Formation, OPA, or a combination — and the VKG Service selects the appropriate enforcement mechanism at query planning time based on the target data source's capabilities. This layered approach ensures that security is enforced as close to the data as possible, leveraging native platform controls where available and falling back to query-level enforcement where necessary.

- **Network Security:** VKG Service runs within a **VPC** with private subnets. Neptune is VPC-native. Data Layer sources are accessed via VPC endpoints (Athena, Redshift), VPC peering, or AWS PrivateLink (Snowflake, Databricks).
- **Observability:** OpenTelemetry instrumentation emits traces, metrics, and logs to **AWS X-Ray** and **Amazon CloudWatch**. Traces include query parse time, Neptune query latency, SQL execution time per source, and total federation overhead.
- **Scalability:** ECS with **Fargate** auto-scales based on query load. Neptune supports read replicas for query scalability. Data Layer sources (Athena, Redshift, Snowflake, Databricks) scale independently.
- **Event-Driven Updates: Neptune DB Streams** emit change data capture events (create, update, delete) to downstream consumers such as the **Symbolic Reasoning Service**, enabling near real-time inference and knowledge graph consistency without polling.

Event-Driven Architecture

Data Flow - Event-Driven Architecture

1. SPARQL Client → Virtual Knowledge Graph Service (Data Manipulation Query) A SPARQL Client submits a SPARQL data manipulation query to the **Virtual Knowledge Graph (VKG) Service** running on **Amazon ECS** within the VPC. This could be an INSERT DATA, DELETE DATA, or CONSTRUCT query — including construct queries generated by reasoning processes that result in new knowledge or deductions. For example, a reasoning workflow may infer new relationships between entities based on ontology axioms and assert them as new triples. The VKG Service

authenticates the request using JWT/OIDC bearer tokens propagated by the AgentCore Gateway and validates the SPARQL UPDATE syntax.

2. VKG Service → Neptune DB (Semantic Layer Graph Database) The VKG Service routes the SPARQL data manipulation query to **Amazon Neptune DB** via the Neptune SPARQL Update endpoint. Neptune persists the changes to the RDF graph — inserting new triples (CREATE), modifying existing statements (UPDATE), or removing triples (DELETE). Neptune enforces **access control policies** at write time, ensuring the requesting principal has authorization to modify the targeted named graphs or triples. The update is committed to Neptune's ACID-compliant transaction log, ensuring durability and consistency. Changes may target the **T-Box** (ontology/schema modifications) or **A-Box** (instance data updates, deduced relationships, cross-domain entity linkages).

3. Neptune DB → Neptune DB Streams → AWS Lambda (Change Data Capture) Upon successful commit of the SPARQL UPDATE, **Neptune DB Streams** automatically emit change data capture (CDC) events in near real-time. Each stream event captures the mutation type (`CREATE`, `UPDATE`, `DELETE`), the affected triples (subject, predicate, object), the named graph IRI, timestamp, and transaction metadata. Neptune DB Streams provide an ordered, durable event log with configurable retention (up to 7 days). An **AWS Lambda function** subscribes to the Neptune DB Streams and is invoked automatically as new change events arrive. The Lambda function acts as the stream event handler, deserializing the event payload and routing it to downstream subscribers based on event type and affected knowledge domain.

4. AWS Lambda → Amazon SNS (Event Publication to Subscribers) The Lambda function publishes the processed Neptune DB Streams event to **Amazon SNS** topics. SNS acts as a pub/sub messaging layer, enabling fan-out to multiple downstream subscribers. Subscribers receive CUD events and handle them appropriately based on their role:

- **Symbolic Reasoning Service** — Processes updates to the **T-Box** (ontology changes) and triggers respective reasoners to validate changes. For example, **ELK** (EL++ reasoner) validates ontology consistency, computes class hierarchies, and detects unsatisfiable classes. Changes to the **A-Box** (instance data) trigger data validation and reasoning workflows using **SHACL** (constraint validation), **SWRL** (rule-based inference), and **SHACL+SPARQL** (complex validation rules). The Symbolic Reasoning Service may generate new CONSTRUCT queries that loop back to Step 1, creating a feedback cycle for iterative knowledge enrichment.
- **Search Index Service** — Updates vector embeddings or full-text search indexes in **Amazon OpenSearch Service** to reflect knowledge graph changes, ensuring semantic search remains current.

- **Ontology Engineering UI** — Notifies knowledge engineers of schema or mapping changes, enabling collaborative ontology management.
- **Agent Orchestration Platform (AgentCore)** — Signals agents to refresh context, invalidate cached knowledge, or re-evaluate reasoning chains affected by the knowledge graph mutation.
- **Audit & Governance Services** — Logs knowledge graph mutations for compliance, lineage tracking, and explainability requirements.

SNS supports both in-account subscribers (Lambda, SQS, EventBridge) and cross-account or external subscribers (HTTPS webhooks, email) for enterprise-wide event distribution. Message filtering policies can route events based on metadata attributes (e.g., `eventType`, `namedGraph`, `ontologyId`).

Event Types & Symbolic Reasoning Triggers

Event Type	Description	T-Box Reasoning Actions	A-Box Reasoning Actions
CREATE	New triples inserted into the knowledge graph	Validate ontology consistency (ELK); recompute class hierarchies	Run SHACL validation on new instances; trigger SWRL rules for new inferences
UPDATE	Existing triples modified (delete + insert)	Detect breaking changes to ontology axioms; revalidate dependent mappings	Revalidate affected instances against SHACL constraints; recompute SWRL-based deductions
DELETE	Triples removed from the knowledge graph	Check for orphaned subclasses or properties; validate ontology integrity	Cascade delete dependent inferences; remove invalidated SHACL validation results

Reasoning Outcome Handling - All reasoning actions produce one or more of the following outcomes, persisted to Neptune with full provenance metadata: (1) *new inferred triples* stored in

dedicated named graphs, distinguishing inferred from asserted knowledge; (2) *SHACL validation reports* with severity levels (violation, warning, info), focus nodes, and remediation suggestions; (3) *consistency flags* annotating axioms or instances with their validation status; (4) *invalidation signals* sent via SNS to AgentCore (triggering cache refresh and reasoning chain re-evaluation), Search Index Service (updating embeddings), and Ontology Engineering UI (notifying knowledge engineers). All outcomes emit new CDC events via Neptune DB Streams, enabling cascading reasoning workflows and maintaining a complete audit trail for compliance and explainability.

Cross-Cutting Concerns

- **Event Ordering:** Neptune DB Streams guarantee ordered delivery within a shard. Lambda processes events in order to maintain consistency across reasoning workflows.
- **Idempotency:** Lambda functions implement idempotency keys (using transaction ID from Neptune streams) to safely handle retries without duplicate processing.
- **Error Handling:** Lambda dead-letter queues (DLQ) capture failed events for later reprocessing. CloudWatch Alarms monitor DLQ depth and Lambda error rates.
- **Security:** Neptune DB Streams are VPC-native. Lambda functions run within the VPC with private subnet access to Neptune. SNS topics enforce IAM-based publish/subscribe authorization.
- **Observability:** OpenTelemetry traces propagate from Neptune write → Stream event → Lambda processing → SNS publish → Symbolic Reasoning Service invocation, providing end-to-end visibility via AWS X-Ray. CloudWatch Logs capture detailed event payloads, reasoning outcomes, and validation results.
- **Scalability:** Lambda auto-scales based on Neptune DB Streams throughput. SNS provides virtually unlimited fan-out to subscribers without backpressure on the event source. Symbolic Reasoning Service can scale horizontally to process multiple reasoning workflows in parallel.

This architecture enables **near real-time, event-driven knowledge graph workflows** — ensuring that reasoning processes, validation engines, search indexes, and downstream agents react immediately to knowledge mutations without polling Neptune. The pattern is particularly valuable for maintaining consistency in federated knowledge graphs where materialized knowledge, deduced relationships, and ontology changes must trigger automated reasoning and validation workflows to preserve semantic integrity.

Symbolic Reasoning

Data Flow — Knowledge Layer Symbolic Reasoning

1. Amazon SNS → Amazon SQS (Graph Mutation Event Trigger) The **Semantic Layer Graph Database** (Neptune DB) emits change data capture events via **Neptune DB Streams** for all knowledge graph mutations — create, update, and delete operations on ontologies (T-Box), instance data (A-Box), or relationships. These mutation events are published to an **Amazon SNS topic** configured for reasoning triggers. The SNS topic implements a fan-out pattern, publishing to an **Amazon SQS queue** dedicated to the Symbolic Reasoning Service. This event-driven architecture ensures that ontology changes, new assertions, or data deletions automatically trigger reasoning workflows without manual intervention. The SQS message contains the mutation type (CREATE, UPDATE, DELETE), affected triples (subject, predicate, object), named graph IRI, transaction metadata, and references to the modified entities.

2. Amazon SQS → AWS Lambda (Reasoning Orchestrator) An **AWS Lambda function** polls the SQS queue and is triggered when graph mutation events arrive. The Lambda function acts as the reasoning orchestration layer, analyzing the event payload to determine which reasoning workflows to invoke based on the type and scope of change:

- **T-Box changes** (ontology modifications) → trigger consistency checking and class hierarchy recomputation
- **A-Box changes** (instance data updates) → trigger constraint validation and rule-based inference
- **Bulk updates** → batch multiple events together for efficient reasoning

The Lambda function prepares the invocation payload for the Symbolic Reasoning Service, including references to affected named graphs, ontology versions, and reasoning policies to apply. The orchestrator invokes the Reasoning Service asynchronously, enabling parallel processing of independent reasoning tasks.

3. AWS Lambda → Symbolic Reasoning Service (Amazon ECS) The Lambda function invokes the **Symbolic Reasoning Service** deployed as containerized tasks on **Amazon ECS** (with **Fargate** or **EC2** instances). The Reasoning Service implements a **modular, ensemble architecture** supporting multiple reasoning engines and techniques:

- **Semantic Reasoners — ELK** (EL++ reasoner for lightweight OWL reasoning), **HermiT** (OWL 2 DL reasoner for full expressivity), or other open-source implementations for ontology consistency checking, class hierarchy computation, and detecting unsatisfiable classes

- **Constraint Validators** — **SHACL** validators for data quality and constraint enforcement
- **Rule Engines** — **SWRL** engines for rule-based inference (e.g., "if X submittedBy Y and Y employedBy Z, then X affiliatedWith Z")
- **Hybrid Validators** — **SHACL+SPARQL** for complex validation rules combining constraints with federated queries

Ontology and Rules Management — The Symbolic Reasoning Service maintains **near real-time synchronization** with ontologies and rules stored in S3 Versioned Buckets or Git repositories. The service subscribes to ontology change events (via SNS/SQS) and automatically reloads updated ontologies and rule sets. This mechanism ensures that reasoning always operates against the current ontology version without manual intervention.

Rule Set Loading and Planning — Based on the graph mutation event details, the Symbolic Reasoning Service identifies the **related rules that need to be evaluated**. For example, if a new claim instance is asserted, only rules applicable to claims (e.g., eligibility validation, fraud detection rules) are loaded. The service then **assembles a reasoning plan** to determine what data is necessary to satisfy the rule set:

- Identifies required sub-graphs (e.g., claim, patient, provider, policy entities and their relationships)
- Determines whether data is materialized in Neptune or virtualized in the Data Layer
- Optimizes query execution order to minimize data retrieval

4. Symbolic Reasoning Service → Virtual Knowledge Graph (VKG) Service The Symbolic Reasoning Service sends **requests to retrieve necessary sub-graphs** to the **Virtual Knowledge Graph (VKG) Service** to satisfy the rule set. The VKG Service federates queries across:

- **Materialized knowledge** in the Semantic Layer Graph Database (Neptune DB)
- **Virtualized data** from the Data Layer (relational databases, data warehouses) via R2RML mappings

The VKG Service translates SPARQL queries into optimized SQL (for virtualized sources) and native SPARQL (for Neptune), returning unified RDF bindings to the Symbolic Reasoning Service. This enables reasoning over both the Semantic Layer and live data from source systems without requiring full materialization.

Once all sub-graphs have been retrieved, the Symbolic Reasoning Service performs reasoning using the loaded rule set and selected reasoning engines:

- **Consistency checking** — Validates that new assertions do not violate ontology axioms
- **Inference** — Derives new triples based on OWL axioms, SWRL rules, or property chains
- **Constraint validation** — Checks SHACL shapes to detect violations
- **Conflict detection** — Identifies contradictory assertions or constraint violations

5. Symbolic Reasoning Service → Neptune DB (Semantic Layer Graph Database) Any reasoning that **results in updates** (e.g., new knowledge via deductions, inferred relationships, validation results) to the Semantic Layer Graph Database are **persisted and reflected** in **Neptune DB**. The Symbolic Reasoning Service writes:

- **Inferred triples** — New relationships or properties derived through reasoning (e.g., transitive closure, property chains, SWRL rule conclusions), stored in dedicated named graphs to distinguish inferred knowledge from asserted knowledge
- **Validation results** — SHACL validation reports indicating constraint violations, with severity levels (violation, warning, info), focus nodes, constraint paths, and remediation suggestions
- **Consistency flags** — Annotations on ontology axioms or instances indicating consistency status, unsatisfiable classes detected by semantic reasoners, or conflicts requiring resolution
- **Reasoning provenance** — Metadata linking inferred triples to the reasoning engines used, source axioms or rules that triggered the inference, timestamps, and confidence scores

Neptune DB Streams then emit events for these newly asserted triples, potentially triggering additional downstream processing (e.g., notifying agents, updating search indexes, triggering cascading reasoning workflows). This creates a **closed-loop, event-driven reasoning architecture** where reasoning results can trigger further reasoning or validation cycles.

Reasoning Ensemble — Modular Architecture

Reasoning Engine	Purpose	Use Cases
ELK (EL++ reasoner)	Lightweight OWL reasoning, class hierarchy computation	Ontology consistency checking, subsumption inference

HermiT (OWL 2 DL)	Full OWL 2 expressivity, complex reasoning	Advanced ontology validatio n, complex class expressions
SHACL Validator	Constraint validation against shapes	Data quality enforcement, business rule validation
SWRL Engine	Rule-based inference	Domain-specific rules (eligibility, fraud detection, compliance)
SHACL+SPARQL	Hybrid constraint validation	Complex validation rules spanning multiple entities

Note: These are open-source implementation options. The modular architecture allows selection and combination of reasoning engines based on use case requirements.

Cross-Cutting Concerns

- **Event-Driven Architecture** — Neptune DB Streams and SNS/SQS fan-out enable near real-time reasoning triggered by knowledge graph mutations, eliminating the need for batch processing or polling.
- **Scalability** — SQS and Lambda enable elastic scaling of reasoning orchestration. ECS tasks auto-scale based on reasoning workload complexity and queue depth. Reasoning can be parallelized across independent sub-graphs.
- **Near Real-Time Ontology Updates** — The Symbolic Reasoning Service subscribes to ontology change events and automatically reloads updated ontologies and rule sets, ensuring reasoning always operates against the current version.
- **Federated Reasoning** — VKG Service integration enables reasoning over both materialized (Neptune) and virtualized (Data Layer) knowledge without data duplication, supporting validation rules that reference live source system data.
- **Reasoning Isolation** — The Symbolic Reasoning Service runs with read-only access to Neptune for input and write-only access for output, preventing unvalidated modifications during reasoning.
- **Observability** — OpenTelemetry traces propagate from SNS → SQS → Lambda → Symbolic Reasoning Service → VKG Service → Neptune write, providing end-to-end visibility via **AWS X-Ray**. CloudWatch Logs capture reasoning results, inference counts, validation violations, and

processing errors. CloudWatch Metrics track reasoning latency, rule evaluation time, and sub-graph retrieval performance.

- **Cost Optimization** — ECS Fargate Spot reduces compute costs for reasoning workloads. Reasoning plan optimization minimizes data retrieval from VKG Service. Incremental reasoning (processing only changed sub-graphs) reduces computational overhead.
- **Provenance & Explainability** — All inferred knowledge includes metadata about the reasoning process (engine used, source rules, axioms), enabling explainability and debugging for agentic AI applications.

This Knowledge Layer Symbolic Reasoning architecture enables **continuous, event-driven reasoning** that maintains knowledge graph consistency, validates data quality, and derives new knowledge automatically as the Semantic Layer evolves — supporting autonomous agents with verified, explainable knowledge.

Data Layer

Data Layer Overview

The **Data Layer** serves as the foundational data fabric of the architecture, encompassing both structured and unstructured data sources across AWS, hybrid, and multicloud environments. For structured data, the layer provides unified access to a broad spectrum of sources including **Amazon Athena** (with connectors for AWS, hybrid, and multicloud data sources), **Amazon Redshift**, **Amazon RDS/Aurora**, **S3 Tables with Apache Iceberg**, **Snowflake**, **Databricks**, and any JDBC-compliant data source. Access control and governance are enforced through **AWS IAM** and **AWS Lake Formation** where supported, with comprehensive data catalog and lineage capabilities provided by **Amazon SageMaker Catalog** and **AWS Glue Data Catalog**. These catalogs also enable ontology induction — the automated discovery and generation of ontology classes, properties, and relationships from existing data schemas, table structures, and column metadata. This structured data foundation enables the Knowledge Layer's Virtual Knowledge Graph (VKG) Service to virtualize relational and data warehouse sources as RDF without data duplication, using ontology-driven R2RML mappings to translate SPARQL queries into optimized SQL.

For unstructured data — including PDFs, images, and documents — the Data Layer implements **Document Ingest Pipelines** that process, chunk, and extract semantic content from raw files. AWS services such as **Amazon Textract** (Optical Character Recognition (OCR) and document extraction), **Amazon Comprehend** (Named Entity Recognition (NER) and entity recognition), and **AWS Lambda** (orchestration) can be combined with open-source alternatives like **unstructured.io**

for flexible document processing workflows. These pipelines prepare unstructured content for semantic indexing and knowledge extraction, generating controlled vocabularies and lightweight taxonomies that bridge the gap between raw documents and formal ontologies.

Processed documents and content are indexed using **GraphRAG** techniques — with the **Amazon Neptune GraphRAG toolkit** serving as a strong implementation choice. GraphRAG creates graphs of concepts, topics, and their vector embeddings, along with associations to document chunks and source documents, stored in the **Lexical Graph Database**. The Lexical Graph Database consists of **Amazon Neptune Database** (for the concept graph and document-chunk relationships) and **Amazon OpenSearch Serverless** (for vector embeddings and semantic search), or alternatively **Amazon Neptune Analytics** which provides integrated vector storage without requiring a separate vector store. This architecture enables hybrid retrieval — combining graph traversal (finding related concepts and documents) with vector similarity search (semantic matching) — to power retrieval-augmented generation (RAG) workflows for agentic AI applications.

Why GraphRAG over Vector-Only RAG? Traditional vector RAG retrieves document chunks based solely on embedding similarity, effective for direct lookups but limited when queries require cross-document reasoning, multi-hop relationships, or corpus-level understanding. GraphRAG augments vector search with a knowledge graph of concepts and relationships, enabling discovery of contextually relevant information that is structurally connected but not semantically similar to the query. Benchmarks demonstrate that GraphRAG improves retrieval accuracy by 7-35% over vector-only approaches (sources: AWS Blog, arXiv:2507.03608). This hybrid pattern delivers the retrieval quality required for agentic AI, where incomplete retrieval cascades into incomplete context for agent decision-making.

GraphRAG Retrieval provides hybrid semantic search capabilities that combine vector similarity with graph traversal to deliver context-aware document and concept retrieval. When a query is submitted via REST API, the GraphRAG Service applies Amazon Bedrock Guardrails to detect and redact PII or toxic content, then generates a query embedding using Amazon Titan Embeddings. The service executes a semantic similarity search against Amazon OpenSearch Service (commonly Serverless), retrieving document chunk identifiers that meet the specified similarity threshold. Using these identifiers, the service queries the Lexical Graph Database (Neptune DB) to perform graph traversal — discovering additional related topics, concepts, and document chunks through multi-hop relationship following. Finally, the service retrieves the full content for relevant document chunks from S3 (where they were cost-effectively stored by the Document Ingest Pipeline), returning ranked results with concept metadata, document chunks, provenance links, and graph context. This hybrid retrieval approach — combining vector embeddings for

semantic matching with graph structures for contextual relevance — powers retrieval-augmented generation (RAG) workflows that deliver accurate, citation-backed answers for agentic AI applications.

In parallel with GraphRAG, **Information Extraction** workflows extract structured, ontology-grounded knowledge from normalized document chunks. Using modular extraction techniques including **spaCy**, **LLM models via Amazon Bedrock**, **Amazon Comprehend**, and **Amazon Comprehend Medical**, the Information Extraction Service identifies entities and relationships guided by domain ontologies. Extracted knowledge is validated and written to the **Semantic Layer Graph Database** — a curated, ontology-aligned knowledge graph containing only validated facts — while comprehensive extraction results (including candidate entities and low-confidence extractions) are stored in the Lexical Graph Database with full provenance links to source documents. This **two-tier knowledge architecture** balances production-ready, validated knowledge with exploratory, comprehensive extraction results, enabling both semantic queries over structured domain knowledge and citation-based retrieval back to source documents.

Document Ingest Pipeline

Data Flow — Document Ingest Pipeline

1. S3 Bucket (Raw Documents) → AWS Lambda (Ingest Orchestrator) Unstructured documents (PDFs, images, Word documents, spreadsheets, etc.) are uploaded to a source **Amazon S3 bucket** configured as the raw document landing zone. An S3 event notification triggers an **AWS Lambda function** automatically upon object creation (PUT or POST). The Lambda function acts as the ingest orchestrator, validating the uploaded object metadata (file type, size, source path) and enforcing guardrails such as a **100 MB file size limit**. The function assigns a unique processing job ID (UUID), determines the appropriate processing workflow based on document type, and enriches the event payload with document metadata including source system, upload timestamp, and content classification tags.

2. AWS Lambda → Amazon SQS (Enqueue Processing Job) The Lambda function publishes a document processing message to an **Amazon SQS queue** for asynchronous, decoupled processing. The SQS message payload includes the S3 object key, bucket name, document type, processing job UUID, and metadata extracted from S3 object tags (e.g., source system, document classification, tenant ID). SQS provides durability, fault tolerance, and the ability to scale the Document Processing Service independently from the ingest rate. The queue is configured with appropriate visibility timeout (aligned with expected processing duration), message retention (up to 14 days), and dead-letter queue (DLQ) settings to handle processing failures and retries.

3. Amazon SQS → Document Processing Service (Amazon ECS) The **Document Processing Service**, deployed as containerized tasks on **Amazon ECS** (with **Fargate** or **EC2** instances), continuously polls the SQS queue for new document processing messages. **unstructured.io is a good choice** for implementing this service — its official Docker image provides native support for PDFs, Office file formats (DOCX, XLSX, PPTX), images (PNG, JPEG), HTML, and other document types, with built-in table extraction capabilities. Alternative implementations could use AWS-native services like **Amazon Textract** and **Amazon Comprehend**, or other open-source document processing frameworks. ECS tasks scale horizontally based on queue depth using CloudWatch metrics and Application Auto Scaling policies.

Each ECS task retrieves a batch of messages, downloads the corresponding source documents from the raw S3 bucket, and executes the document processing pipeline:

- **Format Detection & Element Extraction** — The processing service automatically detects document format and extracts structured elements including text blocks, tables, images, headers, footers, and metadata. When using unstructured.io, these capabilities are built-in. For AWS-native implementations, **Amazon Textract** provides OCR and table extraction, while **Amazon Rekognition** handles image analysis.
- **Plain Text Normalization** — Extracted elements are combined to create a plain text representation of the document, preserving semantic structure while removing formatting artifacts. Tables are converted to structured text representations.
- **Text Chunking** — The plain text representation is segmented into chunks optimized for downstream embedding and retrieval. Each chunk is written as a separate file following the naming convention: `<UUID>-chunk-1.txt`, `<UUID>-chunk-2.txt`, ..., `<UUID>-chunk-n.txt`, where the UUID matches the processing job ID assigned in Step 1.
- **Metadata Generation** — A metadata file is created with the same UUID used for the chunk filenames (e.g., `<UUID>-metadata.json`). The metadata file includes:
  - Processing job UUID
  - Source document S3 key and bucket
  - Total number of chunks generated
  - Per-chunk details: page numbers included, token count, character count, chunk sequence number
  - Document-level metadata: file type, processing timestamp, extraction confidence, detected language
  - Chunk file references/URLs (S3 keys for each chunk file)

The Document Processing Service writes all output — chunk files (`<UUID>-chunk-*.txt`) and metadata file (`<UUID>-metadata.json`) — to the **NORMALIZED S3 Bucket** designated for processed documents. Upon successful S3 write, the service deletes the original message from the SQS queue to prevent reprocessing.

4. NORMALIZED S3 Bucket → AWS Lambda (Completion Handler) When new normalized documents (metadata files) are written to the NORMALIZED S3 Bucket, an S3 event notification triggers an **AWS Lambda function** that acts as the completion handler. The Lambda function reads the metadata file to extract processing details and prepares the event payload for downstream subscribers.

5. AWS Lambda → Amazon SNS (Normalized Document Processing Topic) The completion handler Lambda function publishes a document processing completion event to the **Normalized Document Processing SNS Topic**. The SNS message payload includes:

- Processing job UUID
- Source document S3 key
- Metadata file S3 key/URL
- Chunk file S3 keys/URLs (array of references to `<UUID>-chunk-*.txt` files)
- Number of chunks generated
- Total token count across all chunks
- Processing status (`SUCCESS`, `PARTIAL_SUCCESS`, `FAILED`)
- Document metadata (file type, language, page count)

SNS Topic with SQS Fan-out Pattern — The Normalized Document Processing SNS Topic is configured with an **SQS fan-out pattern** for extensibility and parallel processing. Each downstream processing service subscribes to the SNS topic via its own dedicated SQS queue, enabling:

- **Independent scaling** — Each subscriber (GraphRAG, Information Extraction, Search Indexing) scales independently based on its own queue depth
- **Fault isolation** — Failures in one subscriber do not impact others
- **Parallel execution** — Multiple downstream processes consume the same normalized document event simultaneously without blocking each other
- **Replay capability** — SQS message retention allows subscribers to replay events if processing fails

Downstream subscribers include:

- **GraphRAG Service** — Subscribes to build concept graphs and vector embeddings using the **Amazon Neptune GraphRAG toolkit** (covered in next architecture diagram). Processes chunks independently and in parallel with other subscribers.
- **Information Extraction Service** — Extracts structured entities, relationships, and domain-specific knowledge from normalized chunks (covered in subsequent architecture diagram). Operates independently of GraphRAG ingest.
- **Search Indexing Service** — Updates full-text and semantic search indexes in **Amazon OpenSearch Serverless**
- **Audit & Governance Services** — Logs document processing events for compliance and lineage tracking

SNS message filtering policies can route events based on document type, classification, or processing outcome, ensuring subscribers receive only relevant events.

Processing Capabilities by Document Type (unstructured.io)

Document Type	Direct Processing	Table Extraction	Additional AWS Services (Optional)
PDF (text-based)	# Yes	# Yes	Amazon Textract (enhanced table extraction)
PDF (scanned/image)	# Yes with OCR	# Yes	Amazon Textract (higher accuracy OCR)
DOCX, XLSX, PPTX	# Yes	# Yes	—
Images (PNG, JPG)	# Yes with OCR	# No	Amazon Rekogniti on (object detection, scene analysis)
HTML, Markdown	# Yes	# Yes	—
Email (EML, MSG)	# Yes	# No	—

Note: This table reflects unstructured.io capabilities. Alternative implementations using AWS-native services (Textract, Comprehend) or other frameworks will have different capability profiles.

Cross-Cutting Concerns

- **Scalability:** S3 and SQS provide virtually unlimited throughput. ECS tasks auto-scale based on SQS queue depth. SNS with SQS fan-out enables independent scaling of downstream subscribers. Lambda functions scale automatically to handle S3 and SNS event volumes.
- **Error Handling:** SQS visibility timeout prevents duplicate processing during retries. Dead-letter queues (DLQ) capture failed messages after maximum retry attempts. CloudWatch Alarms monitor DLQ depth and ECS task failure rates. Failed jobs are logged with detailed error context for manual remediation.
- **Security:** S3 buckets enforce server-side encryption (SSE-S3 or SSE-KMS). IAM execution roles for Lambda and ECS tasks follow least-privilege access to S3, SQS, SNS, and Secrets Manager. VPC endpoints ensure private connectivity to AWS services without traversing the public internet.
- **Guardrails:** 100 MB file size limit enforced at ingest (Step 1). Additional guardrails can include maximum page count, supported file types whitelist, and virus scanning (Amazon GuardDuty Malware Protection or ClamAV).
- **Idempotency:** Document Processing Service checks for previously processed UUIDs before initiating extraction, preventing duplicate processing of SQS message retries.
- **Observability:** OpenTelemetry traces propagate from S3 upload → Lambda → SQS → ECS processing → S3 output → Lambda → SNS → SQS fan-out, providing end-to-end visibility via **AWS X-Ray**. **Amazon CloudWatch Logs** capture chunk counts, token counts, processing latency per document type, and error details. CloudWatch Metrics track ingest rate, processing throughput, queue depth, and fan-out latency.
- **Cost Optimization:** S3 Intelligent-Tiering automatically moves infrequently accessed normalized documents to lower-cost storage classes. ECS Fargate Spot can reduce compute costs for fault-tolerant processing workloads. SQS fan-out pattern eliminates the need for custom routing logic, reducing Lambda invocations.

This document ingest pipeline provides a **scalable, fault-tolerant, and event-driven architecture** for normalizing unstructured content at scale. The **SNS with SQS fan-out pattern** enables parallel, independent downstream processing — ensuring that GraphRAG indexing, Information Extraction, and other workflows can scale and fail independently without impacting each other.

GraphRAG Ingest

Data Flow — GraphRAG Ingest

1. Amazon SNS → Amazon SQS (Normalized Document Processing Event) The **Normalized Document Processing SNS Topic** (from the Document Ingest Pipeline) publishes document completion events to an **Amazon SQS queue** dedicated to the GraphRAG Service. This implements the SQS fan-out pattern, allowing the GraphRAG service to scale independently from other downstream subscribers (Information Extraction, Search Indexing, etc.). The SQS message contains the processing job UUID, metadata file S3 key/URL, chunk file S3 keys/URLs, and document metadata from the normalization pipeline.

2. Amazon SQS → AWS Lambda (GraphRAG Orchestrator) An **AWS Lambda function** polls the SQS queue and is triggered when new messages arrive. The Lambda function acts as the GraphRAG orchestration layer, deserializing the SQS message payload, validating the event structure, and coordinating the GraphRAG ingest workflow (concept extraction, graph construction, and vector embedding indexing). The Lambda function prepares the invocation payload for the GraphRAG Service, including references to the normalized document chunks and metadata in S3. For high-volume scenarios, the Lambda can batch multiple document events together to optimize ECS task utilization.

3. AWS Lambda → GraphRAG Service (Amazon ECS) The Lambda function invokes the **GraphRAG Service** deployed as containerized tasks on **Amazon ECS** (with **Fargate** or **EC2** instances). The **Amazon Neptune GraphRAG toolkit is a good choice** for implementing this service — it provides a framework for automating the construction of a graph with vector embeddings from unstructured data, and composing question-answering strategies that query this graph to retrieve structurally relevant information when answering user questions. The toolkit includes purpose-built capabilities for concept extraction, graph construction, and vector embedding generation optimized for Neptune. Alternative implementations could use other graph construction frameworks or custom LLM-based extraction pipelines. The ECS service scales horizontally based on queue depth or direct Lambda invocations, enabling parallel processing of multiple documents simultaneously.

4. GraphRAG Service → S3 Bucket (Retrieve Normalized Chunks) The GraphRAG Service retrieves the normalized document chunks and metadata file from the **NORMALIZED S3 Bucket** using the S3 keys provided in the event payload. The service reads:

- Chunk text files (`<UUID>-chunk-1.txt`, `<UUID>-chunk-2.txt`, etc.)

- Metadata file (`<UUID>-metadata.json`) containing token counts, page references, chunk sequence numbers, and document structure

The service processes each chunk through the GraphRAG ingest pipeline:

- **Concept & Topic Extraction** — Uses Amazon Bedrock foundation models or other LLMs to extract key concepts, topics, entities, and themes from each chunk. Concepts are identified with semantic labels and definitions.
- **Relationship Detection** — Identifies semantic relationships between extracted concepts (e.g., "relates to", "is a type of", "mentioned in context with", "co-occurs with").
- **Vector Embedding Generation** — Generates dense vector embeddings for each concept/topic and chunk using Amazon Bedrock embeddings models (e.g., Amazon Titan Embeddings, Cohere Embed).
- **Graph Construction** — Builds an RDF graph structure connecting concepts to each other, to their source document chunks, and to parent documents.

5. GraphRAG Service → Neptune DB (Lexical Graph Database) The GraphRAG Service writes the constructed concept graph to **Amazon Neptune DB** within the **Lexical Graph Database**. The graph includes:

- **Concept/Topic nodes** — Each extracted concept as an RDF resource with properties (`rdfs:label`, `skos:definition`, concept type, extraction confidence)
- **Document chunk nodes** — References to the source chunks from which concepts were extracted, with metadata (page numbers, token counts, chunk sequence)
- **Document nodes** — Top-level document metadata and references to source files in S3
- **Concept-to-concept relationships** — Semantic associations discovered between concepts (stored as RDF triples)
- **Concept-to-chunk relationships** — Links from concepts to the specific document chunks where they appear, enabling provenance and citation
- **Chunk-to-document relationships** — Links from chunks to their parent documents

Neptune's native RDF and SPARQL support enables efficient graph traversal and querying across the lexical graph. The service uses Neptune's bulk load API or SPARQL INSERT queries to persist the

graph data. Named graphs can be used to organize concepts by document, domain, or processing batch for easier management and querying.

6. GraphRAG Service → OpenSearch (Vector Embeddings) In parallel with writing to Neptune (Step 5), the GraphRAG Service indexes the vector embeddings into **Amazon OpenSearch Serverless** (also part of the **Lexical Graph Database**). OpenSearch stores:

- **Concept/Topic embeddings** — Dense vectors representing the semantic meaning of each concept
- **Chunk embeddings** — Dense vectors representing the semantic content of each document chunk
- **Metadata** — Concept labels, chunk IDs, document IDs, and references back to Neptune graph nodes (via concept IRI or chunk ID)

OpenSearch's k-NN (k-nearest neighbors) plugin enables fast semantic similarity search across concepts and chunks. Each embedding document in OpenSearch includes a reference to the corresponding Neptune graph node, enabling hybrid retrieval that combines:

- **Vector similarity search** in OpenSearch (semantic matching based on embeddings)
- **Graph traversal** in Neptune (relationship following, multi-hop reasoning, and context expansion)

Alternative: Amazon Neptune Analytics — Instead of using separate Neptune DB and OpenSearch services, **Amazon Neptune Analytics** can serve as the unified Lexical Graph Database, providing integrated vector storage without requiring a separate vector store. Neptune Analytics supports both graph queries and vector similarity search natively, simplifying the architecture.

GraphRAG Output — Lexical Graph Structure

Graph Component	Description	Storage
Concept/Topic Nodes	Extracted topics, themes, entities with labels and definitions	Neptune (RDF resources)
Concept Embeddings	Vector representations of concepts for semantic search	OpenSearch (k-NN vectors) or Neptune Analytics

Document Chunk Nodes	Normalized text chunks with metadata (page, tokens, sequence)	Neptune (RDF resources)
Chunk Embeddings	Vector representations of chunk content	OpenSearch (k-NN vectors) or Neptune Analytics
Document Nodes	Top-level document metadata and source references	Neptune (RDF resources)
Concept-Concept Edges	Semantic relationships between concepts (co-occur rence, similarity)	Neptune (RDF triples)
Concept-Chunk Edges	Links from concepts to chunks where they appear	Neptune (RDF triples)
Chunk-Document Edges	Links from chunks to parent documents	Neptune (RDF triples)

Cross-Cutting Concerns

- **Scalability:** SQS and Lambda enable elastic scaling of GraphRAG ingest pipleine. ECS tasks auto-scale based on queue depth or processing load. Neptune supports read replicas for query scalability. OpenSearch Serverless automatically scales compute and storage based on workload.
- **Error Handling:** SQS visibility timeout and retry policies handle transient failures. Dead-letter queue (DLQ) captures failed GraphRAG ingest jobs after maximum retry attempts. CloudWatch Alarms monitor DLQ depth and ECS task failures.
- **Security:** Lambda execution roles grant least-privilege access to SQS, S3, ECS, Neptune, OpenSearch, and Bedrock. Neptune and OpenSearch are VPC-native with private subnet access. S3 buckets enforce encryption at rest (SSE-KMS). VPC endpoints ensure private connectivity to AWS services.
- **Idempotency:** GraphRAG Service checks for previously processed document UUIDs in Neptune before initiating extraction, preventing duplicate graph construction from SQS message retries.
- **Observability:** OpenTelemetry traces propagate from SNS → SQS → Lambda → ECS → Neptune/OpenSearch writes, providing end-to-end visibility via **AWS X-Ray**. CloudWatch Logs capture

concept extraction results, embedding generation latency, graph write statistics, and processing errors. CloudWatch Metrics track processing throughput, concept count per document, vector indexing latency, and queue depth.

- **Cost Optimization:** ECS Fargate Spot reduces compute costs for GraphRAG ingest. S3 Intelligent-Tiering optimizes storage costs for normalized chunks. OpenSearch Serverless charges only for actual usage. Neptune's serverless option (if using Neptune Analytics) eliminates idle capacity costs. Bedrock foundation models are invoked on-demand with pay-per-token pricing.
- **Hybrid Retrieval Pattern:** The combination of Neptune (graph) and OpenSearch (vector) enables advanced RAG workflows:
  1. Start with vector similarity search in OpenSearch to find semantically related concepts/chunks
  2. Expand results via graph traversal in Neptune to discover related concepts and additional context
  3. Return enriched results with both semantic relevance (vector similarity) and structural context (graph relationships)

This GraphRAG architecture enables **semantic understanding of unstructured documents at scale**, building a queryable graph of concepts, topics, and their relationships that powers advanced retrieval-augmented generation (RAG) for agentic AI applications. The Lexical Graph Database serves as the semantic index layer between raw documents and agent reasoning workflows, enabling both semantic search and graph-based reasoning over document content.

GraphRAG Retrieval

Data Flow — GraphRAG Retrieval

1. REST API → GraphRAG Service (Amazon ECS) A client initiates a retrieval request via a **REST API** endpoint, invoking the **GraphRAG Service** deployed as containerized tasks on **Amazon ECS** (with Fargate or EC2 instances) within the VPC. The API request includes:

- **Query input** — Natural language question or search query
- **Similarity threshold** — Minimum cosine similarity score for vector search results (e.g., 0.7)
- **Directives/hints** — Optional parameters such as document type filters, date ranges, ontology scope, result limit (`top_k`), or traversal depth

The request is authenticated via JWT/OIDC bearer tokens propagated by the AgentCore Gateway. The GraphRAG Service acts as the orchestration layer for hybrid retrieval, coordinating vector similarity search and graph traversal to retrieve relevant document chunks and concepts.

2. GraphRAG Service → Amazon Bedrock (Guardrails & Embedding Generation) The GraphRAG Service applies **guardrails to enforce policy** before processing the query. AWS provides **Amazon Bedrock Guardrails** as a guardrail capability, which may satisfy policy requirements such as:

- **Detecting and redacting PII** — Identifying and removing personally identifiable information from the query
- **Removing toxic content** — Filtering offensive, harmful, or inappropriate language
- **Blocking sensitive topics** — Preventing queries related to restricted domains

After applying guardrails, the service **creates a vector embedding of the query** using an embedding model. One option is **Amazon Titan Embeddings** (via Amazon Bedrock), which generates dense vector representations of the query text. The embedding model used must match the model used during GraphRAG indexing (Document Ingest Pipeline) to ensure semantic consistency.

3. GraphRAG Service → Amazon OpenSearch Service (Vector Similarity Search) The GraphRAG Service **submits the vector embedding of the query** and executes a **semantic similarity search** against **Amazon OpenSearch Service** (commonly **OpenSearch Serverless** for auto-scaling and cost efficiency). The service performs a k-NN (k-nearest neighbors) search against the indexed vector embeddings of concepts, topics, and document chunks. OpenSearch returns:

- **Document chunk identifiers** that are semantically similar (cosine similarity **greater than or equal to the threshold**)
- **The most similar results** based on a limit (`top_k`)
- **Similarity scores** for each result
- **Metadata** including concept labels, chunk text snippets, and references to Neptune graph nodes (concept IRIs, chunk IDs)

This vector search provides the initial set of semantically relevant results based on embedding similarity.

4. GraphRAG Service → Lexical Graph Database (Neptune DB) — Graph Traversal The GraphRAG Service **issues a query to the Lexical Graph Database (Neptune DB)** with a **graph traversal based on the identifiers of the returned document chunks** from Step 3. The service executes SPARQL queries to:

- Retrieve full concept/topic nodes with properties (`rdfs:label`, `skos:definition`, extraction confidence, provenance)
- **Traverse the extracted topics and concepts** to find additional topics/concepts that are related to the documents
- Follow concept-to-concept relationships (co-occurrence, semantic similarity, domain relationships)
- Follow concept-to-chunk relationships to discover additional document chunks that mention related concepts
- Retrieve document chunk metadata (page numbers, token counts, chunk sequence, source document references)
- Expand context by following multi-hop relationships (e.g., "retrieve all concepts related to the matched concept within 2 hops")

Neptune returns the graph of topics/concepts, document chunks, and documents, providing structural context that enriches the vector search results. This graph traversal **discovers additional document chunks** that may not have been semantically similar to the query but are contextually relevant through their relationships to matched concepts.

5. GraphRAG Service → S3 Bucket (Document Chunk Content Retrieval) For the relevant document chunks identified through Steps 3-4, the GraphRAG Service **retrieves the content for document chunks and documents** that are **stored cost-effectively in S3** (where they were processed by the Document Ingest Pipeline). Using the S3 keys stored in Neptune's chunk metadata, the service fetches:

- **Chunk text files** (`<UUID>-chunk-1.txt`, `<UUID>-chunk-2.txt`, etc.) containing the normalized text content
- **Associated metadata files** (`<UUID>-metadata.json`) with page numbers, token counts, and source document references

This step ensures that the service can return the actual text content to the user or agent, not just chunk IDs or references. The retrieved chunks include the original normalized text, enabling citation with exact passages and page numbers.

6. GraphRAG Service → REST API Response The GraphRAG Service aggregates and ranks the results from vector search (Step 3), graph traversal (Step 4), and chunk retrieval (Step 5), then returns a unified response to the REST API client:

- **Ranked results** — Concepts and document chunks ordered by relevance (combining vector similarity scores and graph-based relevance metrics)
- **Concept metadata** — Labels, definitions, related concepts, extraction confidence, provenance
- **Document chunks** — Full text content with page numbers, chunk sequence, and source document references
- **Provenance** — Links from concepts to source chunks and documents, enabling citation and traceability
- **Graph context** — Related concepts and relationships discovered through graph traversal

The response enables downstream consumers (Q&A Agent, chatbots, user interfaces) to present semantically grounded, citation-backed answers with full context.

Hybrid Retrieval — Combining Vector Search and Graph Traversal

The power of GraphRAG retrieval lies in combining:

1. **Vector similarity search (OpenSearch)** — Fast, semantic matching based on embeddings to find relevant concepts and chunks
2. **Graph traversal (Neptune)** — Structural context, relationship following, and multi-hop reasoning to discover related concepts and supporting evidence that may not be semantically similar but are contextually relevant

This hybrid approach outperforms vector-only or graph-only retrieval by balancing semantic relevance with structural context, enabling more comprehensive and accurate question answering.

Cross-Cutting Concerns

- **Guardrails Integration** — Amazon Bedrock Guardrails are applied at the query entry point to enforce policy before processing, ensuring that sensitive or inappropriate queries are filtered before consuming downstream resources.
- **Embedding Consistency** — The embedding model used for query vectorization (e.g., Amazon Titan Embeddings) must match the model used during GraphRAG indexing to ensure semantic consistency and accurate similarity scoring.
- **Performance** — Vector search in OpenSearch is highly parallelizable and returns results in milliseconds. Graph traversal in Neptune adds latency proportional to traversal depth but is optimized using SPARQL query planning and Neptune's graph-native storage.
- **Scalability** — ECS auto-scales based on query volume. OpenSearch Serverless scales compute and storage automatically. Neptune supports read replicas for query scaling.
- **Observability** — OpenTelemetry traces propagate from REST API → GraphRAG Service → Bedrock (guardrails/embeddings) → OpenSearch query → Neptune query → S3 retrieval, providing end-to-end visibility via AWS X-Ray. CloudWatch Metrics track vector search latency, graph traversal depth, guardrail invocations, and result quality.
- **Cost Optimization** — OpenSearch Serverless charges only for actual usage. S3 Intelligent-Tiering optimizes storage costs for infrequently accessed chunks. Bedrock Guardrails and Titan Embeddings are pay-per-use, minimizing idle capacity costs.

This GraphRAG Retrieval architecture enables **semantic, context-aware document and concept retrieval** that powers RAG workflows for agentic AI applications, combining the strengths of vector embeddings, guardrails, and knowledge graphs to deliver accurate, explainable, and citation-backed answers.

Information Extraction

Data Flow — Information Extraction Processing

1. Amazon SNS → Amazon SQS (Normalized Document Processing Event) The **Normalized Document Processing SNS Topic** (from the Document Ingest Pipeline) publishes document completion events to an **Amazon SQS queue** dedicated to the Information Extraction Service. This implements the SQS fan-out pattern, allowing the Information Extraction processor to scale independently from other downstream subscribers (GraphRAG, Search Indexing, etc.). The SQS message contains the processing job UUID, metadata file S3 key/URL, chunk file S3 keys/URLs,

and document metadata from the normalization pipeline. This queue operates in parallel with the GraphRAG ingest queue, enabling simultaneous, independent processing of the same normalized documents.

2. Amazon SQS → AWS Lambda (Information Extraction Orchestrator) An **AWS Lambda function** polls the SQS queue and is triggered when new messages arrive. The Lambda function acts as the Information Extraction orchestration layer, deserializing the SQS message payload, validating the event structure, and determining the appropriate extraction workflow based on document type and metadata. The Lambda function prepares the invocation payload for the Information Extraction Service, including references to the normalized document chunks and metadata in S3. For high-volume scenarios, the Lambda can batch multiple document events together to optimize ECS task utilization.

3. AWS Lambda → Information Extraction Service (Amazon ECS) The Lambda function invokes the **Information Extraction Service** deployed as containerized tasks on **Amazon ECS** (with **Fargate** or **EC2** instances). The service implements a **modular architecture** that allows for different extraction techniques, including **spaCy** (rule-based and statistical NLP), **LLM models** via Amazon Bedrock, **Amazon Comprehend** (for general entity recognition), and **Amazon Comprehend Medical** (for healthcare-specific extraction). The ECS service scales horizontally based on queue depth or direct Lambda invocations, enabling parallel processing of multiple documents simultaneously.

4. Information Extraction Service → S3 Bucket (Retrieve Normalized Chunks) The Information Extraction Service retrieves the normalized document chunks and metadata file from the **NORMALIZED S3 Bucket** (same bucket used by GraphRAG ingest) using the S3 keys provided in the event payload. The service reads:

- Chunk text files (`<UUID>-chunk-1.txt`, `<UUID>-chunk-2.txt`, etc.)
- Metadata file (`<UUID>-metadata.json`) containing token counts, page references, chunk sequence numbers, and document structure

The service processes each chunk through the information extraction pipeline:

- **Ontology-Guided Entity Recognition** — Extracts entities using the provided ontologies to guide the types of information to extract. The ontology defines target entity classes (e.g., `Person`, `Organization`, `Claim`, `Diagnosis`, `Procedure`) and their expected properties. Extraction techniques (spaCy, LLM models, Amazon Comprehend) are applied based on entity type and document domain.

- **Ontology-Guided Relationship Extraction** — Identifies semantic relationships between extracted entities using ontology-defined object properties (e.g., `employedBy`, `diagnoses`, `submittedBy`, `covers`). The ontology provides relationship constraints (domain, range) to guide extraction and validation.
- **Codification & Normalization** — Extracted entities and relationships are codified according to the ontology vocabulary, ensuring consistency with the Knowledge Layer's T-Box. Entity identifiers are normalized (e.g., standardizing organization names, mapping codes to ontology classes).
- **Validation & Reasoning** — Additional validation and reasoning may be performed to ensure consistency and quality of extracted entities and relationships. This includes:
  - **SHACL validation** against ontology constraints (e.g., cardinality, datatype, value ranges)
  - **Consistency checking** to detect conflicting facts or constraint violations
  - **Inference** to derive implied relationships based on ontology axioms
  - **Confidence scoring** to flag low-confidence extractions for review
- **Provenance Tracking** — Records the source chunks, extraction technique used (LLM model, spaCy, Amazon Comprehend), confidence scores, and extraction timestamps for each extracted fact. Entities and relationships are annotated with properties indicating the extraction method.

5. Information Extraction Service → Neptune DB (Semantic Layer Graph Database) The Information Extraction Service writes the **ontology-based, grounded** extracted knowledge to **Amazon Neptune DB** within the **Semantic Layer Graph Database (SLGD)**. This is the curated, domain-specific knowledge graph that represents validated facts about the business domain. The extracted data includes:

- **Entity instances** — Individuals conforming to ontology classes (e.g., `ex:Person/JohnDoe`, `ex:Organization/AcmeCorp`, `ex:Claim/CLM123456`) with typed properties
- **Relationships** — RDF triples connecting entities via ontology-defined object properties (e.g., `ex:Claim/CLM123456 ex:submittedBy ex:Person/JohnDoe`)
- **Attributes** — Datatype properties on entities (e.g., claim amount, submission date, diagnosis code)
- **Provenance metadata** — Links from extracted facts back to source document chunks in the Lexical Graph Database, extraction confidence scores, extraction timestamps, and extraction methods

- **Validation status** — Flags indicating whether facts passed SHACL validation and reasoning checks

The Semantic Layer Graph Database is **more curated** than the Lexical Graph Database — it contains only validated, ontology-aligned knowledge that has passed quality checks. Named graphs can be used to organize extracted knowledge by confidence level (e.g., high-confidence vs. candidate extractions requiring validation), source document, or extraction batch.

6. Information Extraction Service → Neptune DB (Lexical Graph Database) — Provenance & Enrichment In parallel with writing to the Semantic Layer Graph Database (Step 5), the Information Extraction Service writes **provenance links and enrichment metadata** to the **Lexical Graph Database (LGD)** (Neptune DB instance used by GraphRAG). The Lexical Graph Database will have **more entities and relationships** than the Semantic Layer Graph Database, as it includes:

- **All extracted entities and relationships** — Including low-confidence extractions, candidate facts, and entities that did not pass validation for the Semantic Layer
- **Extraction technique annotations** — Properties on entities and relationships indicating the extraction method (e.g., `ex:extractedBy "LLM Model"`, `ex:extractedBy "spaCy"`, `ex:extractedBy "Amazon Comprehend"`)
- **Document chunk associations** — Links from extracted entities to the specific chunks in the Lexical Graph where they were mentioned, enabling provenance and citation
- **Bidirectional references** — Enriches Lexical Graph chunk nodes with references to the structured entities extracted from them, connecting unstructured content to structured knowledge

This creates a **two-tier knowledge architecture**:

- **Lexical Graph Database (LGD)** — Comprehensive, document-centric, includes all extractions regardless of confidence or validation status
- **Semantic Layer Graph Database (SLGD)** — Curated, domain-centric, ontology-aligned, contains only validated knowledge

The integration between the two databases enables powerful hybrid queries:

- **Semantic queries** — "Find all claims submitted by organization X" (queries SLGD)
- **Provenance queries** — "Show me the document chunks where this claim appears" (traverses to LGD)

- **Exploratory queries** — "What candidate entities were extracted but not validated?" (queries LGD for entities not in SLGD)

Information Extraction Output — Two-Tier Knowledge Graph

Graph Component	Lexical Graph Database (LGD)	Semantic Layer Graph Database (SLGD)
Entity Instances	All extracted entities (validated + candidates)	Only validated, ontology-aligned entities
Relationships	All extracted relationships	Only validated, ontology-defined relationships
Extraction Metadata	Technique, confidence, timestamp for all extractions	Validation status, reasoning results
Provenance Links	Links to source chunks and documents	Links to source chunks via LGD
Curation Level	Comprehensive, exploratory	Curated, production-ready
Use Cases	Document retrieval, exploratory analysis, debugging	Semantic queries, reasoning, agent workflows

Cross-Cutting Concerns

- **Scalability:** SQS and Lambda enable elastic scaling of information extraction processing. ECS tasks auto-scale based on queue depth or processing load. Neptune supports read replicas for query scalability across both Graph Databases.
- **Error Handling:** SQS visibility timeout and retry policies handle transient failures. Dead-letter queue (DLQ) captures failed extraction jobs after maximum retry attempts. CloudWatch Alarms monitor DLQ depth and ECS task failures. Failed extractions can be flagged for manual review or sent to human-in-the-loop validation workflows.
- **Security:** Lambda execution roles grant least-privilege access to SQS, S3, ECS, Neptune (both instances), and Bedrock. Neptune instances are VPC-native with private subnet access. S3 buckets enforce encryption at rest (SSE-KMS). VPC endpoints ensure private connectivity to AWS services.

- **Idempotency:** Information Extraction Service checks for previously processed document UUIDs in Neptune before initiating extraction, preventing duplicate entity creation from SQS message retries. Extracted entities are assigned deterministic IRIs based on source document and extracted identifiers to enable safe re-processing.
- **Data Quality & Validation:** Extracted facts are validated against SHACL constraints defined in the ontology before being committed to the Semantic Layer Graph Database. Low-confidence extractions remain in the Lexical Graph Database for exploratory analysis.
- **Observability:** OpenTelemetry traces propagate from SNS → SQS → Lambda → ECS → Neptune writes (both databases), providing end-to-end visibility via **AWS X-Ray**. CloudWatch Logs capture entity extraction results, relationship counts, confidence scores, validation outcomes, and processing errors. CloudWatch Metrics track processing throughput, entity count per document, extraction latency, validation pass rate, and queue depth.
- **Cost Optimization:** ECS Fargate Spot reduces compute costs for information extraction processing. Bedrock foundation models are invoked on-demand with pay-per-token pricing. Neptune's serverless option eliminates idle capacity costs. Modular extraction architecture allows cost-effective technique selection (e.g., spaCy for simple entities, LLMs for complex relationships).
- **Modular Extraction Architecture:** The service supports pluggable extraction techniques:
  - **spaCy** — Fast, rule-based extraction for well-defined entity types (names, dates, identifiers)
  - **LLM Models (Bedrock)** — Flexible, context-aware extraction for complex entities and relationships
  - **Amazon Comprehend** — General-purpose entity recognition and sentiment analysis
  - **Amazon Comprehend Medical** — Healthcare-specific entity extraction (diagnoses, procedures, medications, anatomy)
  - **Custom Models** — Fine-tuned models for domain-specific extraction tasks

This Information Extraction processing architecture enables **structured, ontology-grounded knowledge extraction from unstructured documents at scale**, building a curated, queryable knowledge graph (SLGD) that powers semantic queries, reasoning workflows, and agent-based question answering — while maintaining comprehensive provenance and exploratory capabilities in the Lexical Graph Database (LGD). The two-tier architecture balances production-ready, validated knowledge with exploratory, comprehensive extraction results.

Orchestration Layer

Orchestration Layer Overview

The **Orchestration Layer** serves as the agentic intelligence hub of the architecture, coordinating autonomous agents that leverage the semantic foundations provided by the Knowledge and Intelligence Layers to deliver business value through automated reasoning, knowledge discovery, and human-in-the-loop workflows. While these agents are the primary consumers of the Orchestration Layer, the same information and capabilities they utilize are accessible to customers and other agents through the Semantic Layer's unified interfaces — ensuring that semantic knowledge, reasoning capabilities, and contextual insights are available enterprise-wide, not siloed within individual agent implementations.

The **Question & Answer (Q&A) Agent** responds to queries from the Interface Layer (primarily via MCP Server, with REST API support as well) by processing `ask` requests that require semantic understanding and knowledge retrieval. The Q&A Agent leverages the VKG Service to execute federated SPARQL queries across the Semantic Layer Graph Database and virtualized Data Layer sources, retrieves relevant document chunks and concepts from the Lexical Graph Database via GraphRAG techniques, and synthesizes responses grounded in ontology-defined knowledge with full provenance. The agent maintains short-term memory via context graphs (managed by the Intelligence Layer) that capture query history, retrieved entities, and inferred user intent, enabling multi-turn conversations with semantic continuity. Responses include citations back to source documents, knowledge graph entities, and reasoning steps, ensuring explainability and trust.

The **Business Elicitation Agent** operates as an event-driven knowledge discovery system that continuously monitors the semantic landscape to identify gaps, inconsistencies, and opportunities for human expert input. The agent subscribes to **Graph Mutation Events** (via Neptune DB Streams), **Document Ingest Pipeline** events (via SNS/SQS), and **Ontology Induction** workflows to analyze newly asserted knowledge, extracted entities, and induced ontology elements. Using LLM-based analysis via Amazon Bedrock, the agent assesses whether new knowledge aligns with existing ontologies, identifies ambiguous or conflicting assertions, and detects missing business context (e.g., undefined relationships, incomplete constraints, or entities lacking business glossary mappings). When **Human-in-the-Loop (HITL)** intervention is needed — such as resolving semantic ambiguities, validating high-impact ontology changes, or capturing domain expert knowledge for underspecified concepts — the agent triggers approval workflows via internal systems or third-party tools, notifying ontology engineers, data stewards, or domain experts with contextual

information and recommended actions. This proactive elicitation ensures that the semantic layer evolves in alignment with business reality and domain expertise.

The **Ontology Maintenance / Entity Resolution Agent** ensures semantic consistency and quality across the Semantic Layer by identifying and resolving overlaps, discrepancies, and duplications in ontology structures and materialized knowledge. The agent analyzes **ontology classes, properties, constraints, and rules** to detect redundant or conflicting definitions (e.g., two classes representing the same concept with different names, properties with overlapping semantics, or contradictory cardinality constraints). Using **vector-based similarity techniques** (OWL2Vec*, RDF2Vec) and **LLM-based semantic analysis** via Amazon Bedrock, the agent computes similarity scores between ontology elements and proposes consolidation or alignment strategies. For **entity resolution**, the agent identifies similar or duplicate entities in the **Semantic Layer Graph Database** (Neptune DB) — such as multiple patient records representing the same individual, or claim instances with overlapping identifiers — and applies disambiguation rules, probabilistic matching, or LLM-based reasoning to merge or link entities with appropriate provenance. The agent has access to **Data Layer sources via the VKG Service**, enabling it to validate entity resolution decisions against source system data, detect schema drift, and ensure that ontology maintenance reflects the current state of enterprise data. Proposed changes are written to draft named graphs and subject to governance approval workflows before promotion to production, ensuring quality control and auditability.

Orchestration Layer Agents

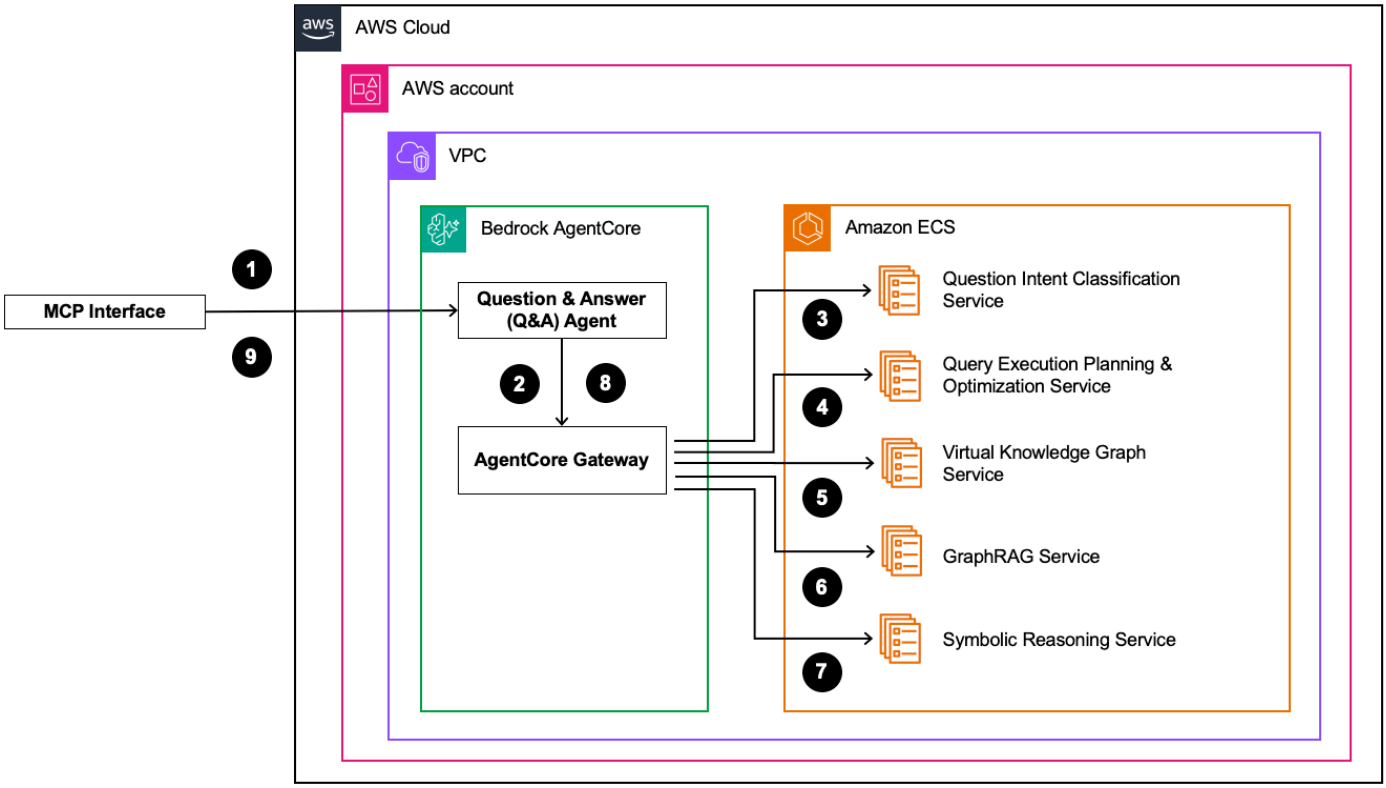
Agent	Purpose	Key Capabilities
Question & Answer (Q&A) Agent	Respond to semantic queries from Interface Layer	Federated SPARQL queries, GraphRAG retrieval, context graph management, provenance-based citations
Business Elicitation Agent	Proactive knowledge discovery and HITL workflow orchestration	Event-driven monitoring, gap analysis, ambiguity detection, workflow triggering
Ontology Maintenance / Entity Resolution Agent	Ensure semantic consistency and quality	Ontology overlap detection , entity disambiguation, similarity-based matching, VKG-based validation

Cross-Cutting Concerns

- **Unified Access** — All capabilities utilized by Orchestration Layer agents (VKG Service queries, Symbolic Reasoning Service invocations, Intelligence Layer context graphs, Lexical Graph Database retrieval) are accessible to customers and other agents through the Semantic Layer's MCP Server and REST APIs, ensuring that semantic knowledge is not siloed within agent implementations.
- **Event-Driven Architecture** — Agents subscribe to Neptune DB Streams, SNS/SQS topics, and workflow events to react in near real-time to knowledge graph mutations, document ingestion, and ontology changes, enabling proactive rather than reactive orchestration.
- **Context-Aware Reasoning** — Agents leverage short-term memory and context graphs (Intelligence Layer) to maintain session state, track multi-turn interactions, and ground reasoning in specific conversational or workflow contexts.
- **Human-in-the-Loop Integration** — Agents trigger approval workflows, notify domain experts, and surface recommendations via Ontology Engineering UIs, workflow management systems, or collaboration platforms when automated reasoning requires human validation or domain expertise.
- **Provenance & Explainability** — All agent actions, reasoning steps, and knowledge assertions include provenance metadata linking back to source data, ontology axioms, and reasoning engines, enabling auditability and trust.
- **Scalability** — Agents are deployed as containerized services on Amazon ECS (with Fargate or EC2 instances), auto-scaling based on workload complexity and event volume.
- **Observability** — OpenTelemetry traces propagate across agent invocations, VKG Service queries, Symbolic Reasoning Service calls, and Intelligence Layer operations, providing end-to-end visibility via AWS X-Ray. CloudWatch Logs capture agent decisions, HITL triggers, entity resolution outcomes, and query results.

This Orchestration Layer enables **autonomous, knowledge-driven agents** that leverage the semantic foundations of the architecture to deliver business value through automated reasoning, proactive knowledge discovery, and human-expert collaboration — while ensuring that the same semantic capabilities are accessible enterprise-wide through unified interfaces.

Question & Answer (Q&A) Agent



Data Flow — Question & Answer (Q&A) Agent

1. MCP Interface → Question & Answer (Q&A) Agent (Bedrock AgentCore) The **MCP Interface** provided by the Semantic Layer receives an `ask` tool invocation with the question in the request body. Additional metadata or directives (TBD) may be provided in the request to guide query processing, specify target ontologies, or set confidence thresholds. The **Question & Answer (Q&A) Agent** runs in **Amazon Bedrock AgentCore Runtime** and may be implemented in any AgentCore Runtime supported framework (e.g., **Strands**). The agent leverages **AgentCore identity** where appropriate for inbound and outbound authentication, receiving and validating the request. The agent applies **guardrails to enforce policy** — AWS provides **Amazon Bedrock Guardrails** as a guardrail capability, which may satisfy policy requirements such as identifying and redacting PII, detecting toxic content, or blocking sensitive topics. The request is authenticated via JWT/OIDC bearer tokens (or IAM SigV4 for AWS-native A2A callers at the Interface Layer). AgentCore validates the token, extracts the user identity (`sub`), authorization groups (`groups`), and agent identity (`act`) from the JWT claims, and provides session management, memory persistence (for context graphs), and secure execution isolation.

2. Q&A Agent → AgentCore Gateway The Q&A Agent calls the **AgentCore Gateway**, an **MCP Gateway that exposes REST APIs from services running in ECS as MCP tools**. The AgentCore Gateway acts as the service mesh and orchestration layer, providing:

- **Service discovery and routing** — Routes requests to downstream services (Question Intent Classification, Query Execution Planning & Optimization, VKG Service, GraphRAG Service, Symbolic Reasoning Service)
- **Authentication and authorization** — Validates and propagates JWT/OIDC tokens to downstream services, enforcing group-based authorization via the `groups` claim. IAM execution roles handle outbound calls to AWS services
- **Protocol translation** — Converts agent-internal requests to service-specific protocols (REST, SPARQL, gRPC)
- **Observability** — Emits OpenTelemetry traces for end-to-end request tracking

The Gateway receives the user's question and metadata from the Q&A Agent and prepares it for distribution to specialized processing services. **Steps 3-7 are MCP Tool invocations** orchestrated by the agent through the AgentCore Gateway.

3. AgentCore Gateway → Question Intent Classification Service (Amazon ECS). The AgentCore Gateway invokes the **Question Intent Classification Service** deployed on **Amazon ECS** (with Fargate or EC2 instances). The **Intent Classification component periodically retrieves the latest version of ontologies** from the Semantic Layer Graph Database (Neptune DB). The assumption is there may be **multiple ontologies** (e.g., different domains, cross-domain ontology enabled with upper ontology), and for some customers, **multiple versions of every ontology** may need to be supported. This request/response pattern may be replaced or augmented with an **event-driven pattern** (e.g., "Updates to Ontology version X, go retrieve") to ensure near real-time synchronization with ontology changes.

The Intent Classification Service determines:

- **Type of question** — Retrieval (factual lookup), aggregation (count, sum, average), comparison, explanation, or validation
- **Additional processing requirements** — Ranking, filtering, temporal constraints, reasoning, or document retrieval

- **Ontology classes and relationships** — Identifies relevant classes (e.g., `Claim`, `Patient`, `Provider`) and relationships (data properties like `claimAmount`, object properties like `submittedBy`)
- **Entity and relationship extraction** — Extracts and identifies entities and relationships **grounded by the provided ontologies**, linking natural language terms to ontology concepts
- **Semantic similarity search** — Semantic similarity search of nodes and relationships may be appropriate here, using vector embeddings (OWL2Vec*, RDF2Vec) to match question terms to ontology concepts when exact matches are not found
- The service also recognizes metric names (e.g., "processing time") and maps them to `ex:MetricDefinition` instances in the Metric Store, returning intent category `METRIC_QUERY` with the resolved metric IRI and pre-compiled expression — bypassing LLM-generated calculation synthesis.

The service uses LLM-based classification via Amazon Bedrock (e.g., prompt engineering with few-shot examples, fine-tuned models) and returns:

- Intent category (e.g., `ENTITY_LOOKUP`, `RELATIONSHIP_QUERY`, `AGGREGATION`, `VALIDATION`, `DOCUMENT_RETRIEVAL`)
- Extracted entities and properties grounded in ontology
- Suggested query strategy (SPARQL, vector search, hybrid, GraphRAG retrieval)
- Confidence score

The intent classification output drives all downstream orchestration decisions. Based on the intent category and suggested query strategy, the Q&A Agent determines which subsequent steps to invoke: Query Execution Planning and VKG (for structured knowledge retrieval), GraphRAG (for document retrieval), Symbolic Reasoning (for inference and validation), or combinations thereof.

4. AgentCore Gateway → Query Execution Planning & Optimization Service (Amazon ECS) — Conditional, Invoked for Structured/Federated Queries. When the intent classification (Step 3) indicates structured knowledge retrieval, aggregation, relationship queries, or federated data access is required, the AgentCore Gateway invokes the Query Execution Planning & Optimization Service deployed on Amazon ECS. The **Execution Planning & Optimization component periodically retrieves the latest schema mapping for Ontologies** from the Semantic Layer. **Schema Mapping provides optimization-oriented information**, such as:

- **Location and completeness of information by source** — Which data sources (Neptune, Snowflake, Redshift, S3) contain instances of specific ontology classes
- **Data quality metrics** — Completeness, accuracy, and freshness scores for each source
- **Cardinality and statistics** — Row counts, relationship cardinality, and data distributions, providing information regarding **how much data will need to be transmitted and processed**

For `METRIC_QUERY` intents, the service injects the metric's pre-compiled `ex:metricExpression` directly into the generated query, replacing LLM-generated aggregation logic. Standard optimizations apply around the injected fragment.

Planning should consider what rules may apply so dependent facts can be retrieved. For example, if a reasoning rule requires additional entities or relationships to evaluate, the planner ensures those sub-graphs are included in the query.

The **Execution Planning and Optimization component determines whether multiple steps are required**. It **creates an execution plan that executes in parallel or serially** based on an **optimizer algorithm that may incorporate statistics** (e.g., cardinality for ontology classes and relationships). **Cross-domain ontology questions most often require multiple steps to execute** — for example, a question spanning healthcare claims and provider networks may require separate queries to different data sources, followed by a join or federation step.

The service generates:

- **SPARQL queries** — One or more SPARQL queries (SELECT, CONSTRUCT, ASK, DESCRIBE) aligned with the target ontology
- **Federated query plan** — Identifies which data sources are required (Semantic Layer Graph Database in Neptune, Data Layer sources via VKG) and plans federated query execution
- **GraphRAG retrieval plan** — If the intent classification (Step 3) indicates document retrieval is needed, specifies GraphRAG Service invocation parameters (similarity threshold, result limit, traversal depth)
- **Query optimization** — Applies predicate pushdown, join reordering, filter propagation, and ontology-driven query rewriting (expanding with subclass hierarchies, property paths, inference rules)
- **Execution strategy** — Parallel or serial execution plan with estimated cost and latency

5. AgentCore Gateway → Virtual Knowledge Graph (VKG) Service (Amazon ECS) — Conditional, Invoked When Execution Plan Requires Structured/Federated Queries. When the execution plan (Step 4) includes SPARQL queries or federated query plans, the AgentCore Gateway invokes the Virtual Knowledge Graph (VKG) Service to execute those queries. The VKG Service federates queries across:

- **Semantic Layer Graph Database (Neptune DB)** — Queries materialized ontologies (T-Box), instance data (A-Box), and inferred knowledge
- **Data Layer sources** — Translates SPARQL queries into optimized SQL (via R2RML mappings) and executes against relational databases, data warehouses (Snowflake, Redshift, Athena), and data lakes (S3 Tables/Iceberg)

The VKG Service returns unified RDF bindings with provenance metadata (source systems, confidence scores, extraction timestamps, data quality indicators).

6. AgentCore Gateway → GraphRAG Service (Amazon ECS) — Conditional, Dependent on Query Type If the query intent (Step 3) indicates that document retrieval is required, the AgentCore Gateway invokes the **GraphRAG Service** to retrieve relevant document chunks and concepts using hybrid retrieval. The GraphRAG Service:

- Applies guardrails (via Amazon Bedrock Guardrails) to detect and redact PII or toxic content from the query
- Generates a query embedding using Amazon Titan Embeddings (or another embedding model)
- Executes vector similarity search against Amazon OpenSearch Service (commonly Serverless) to retrieve document chunk identifiers that meet the specified similarity threshold
- Performs graph traversal in the Lexical Graph Database (Neptune DB) to discover related topics, concepts, and additional document chunks through multi-hop relationship following
- Retrieves full chunk content from S3 (where chunks were cost-effectively stored by the Document Ingest Pipeline)
- Returns ranked results with concept metadata, document chunks, provenance links, and graph context

This step enables the Q&A Agent to incorporate unstructured document content and GraphRAG-indexed concepts into its answers, complementing the structured knowledge retrieved from the VKG Service (Step 5). **This step is optional** — it is invoked only when the question requires

document-grounded evidence or when the VKG Service alone cannot provide sufficient information. [170]

7. AgentCore Gateway → Symbolic Reasoning Service (Amazon ECS) — Conditional, Invoked When Inference or Validation Is Required. TWhen the intent classification (Step 3) or execution plan (Step 4) indicates that inference, constraint validation, or consistency checking is required, the AgentCore Gateway invokes the Symbolic Reasoning Service with the available results from the VKG Service (Step 5) and/or the GraphRAG Service (Step 6). Responses from Query Federation and MCP Server / Tool & Data Integration are reasoned over and processed. In addition to processing the responses, **semantic reasoning may be leveraged for responses that combine data from our Data Layer**.

Since we have Continuous Reasoning for our Semantic Layer Graph Database / Knowledge Layer, Semantic Reasoning may not need to be performed for queries that retrieve only materialized knowledge from Neptune (which is already validated and inferred by the continuous reasoning workflows). **If data is federated and potentially combined with knowledge graphs from our Semantic Layer Graph DB, we may have to perform Semantic Reasoning** to:

- **Validate consistency** — Ensure that virtualized data from the Data Layer is consistent with ontology axioms and constraints
- **Infer new relationships** — Derive additional triples based on OWL axioms, SWRL rules, or property chains when combining materialized and virtualized knowledge
- **Apply backward chaining** — **Backward-Chaining** reasoning may be used to derive facts needed to answer the question by working backward from the query goal to the available data

The Symbolic Reasoning Service applies:

- **Inference** — Derives new triples based on OWL axioms, SWRL rules, or property chains
- **Constraint validation** — Checks if query results satisfy SHACL constraints or business rules
- **Consistency checking** — Validates that returned entities and relationships are consistent with ontology axioms
- **Explanation generation** — Provides reasoning chains explaining why certain facts were inferred or how constraints were validated

The Symbolic Reasoning Service may retrieve additional sub-graphs from the VKG Service if needed for reasoning, creating a potential recursive flow between Steps 5 and 7. Reasoning results

(inferred triples, validation outcomes, explanations) are returned to the AgentCore Gateway with provenance linking back to the axioms or rules that triggered the inference.

8. AgentCore Gateway → Q&A Agent (Response Synthesis) The AgentCore Gateway aggregates results from the invoked services (VKG Service, GraphRAG Service, Symbolic Reasoning Service, or any combination thereof as determined by Steps 3 and 4) and returns them to the Q&A Agent. The agent synthesizes the final response:

- **Natural language generation** — Uses Amazon Bedrock foundation models to generate a coherent, contextual answer from the retrieved RDF bindings, document chunks, and reasoning results
- **Citation generation** — Includes provenance metadata in the response, citing source documents (with chunk references), knowledge graph entities (with IRIs), and reasoning steps (with axiom/rule references)
- **Context graph update** — Updates the short-term memory/context graph (managed by the Intelligence Layer) with the current query, retrieved entities, and inferred user intent for use in subsequent multi-turn interactions
- **Metric provenance** — cites metric definition name, version, and publish date when a metric is used

9. Q&A Agent → MCP Interface (Response Delivery) The Q&A Agent returns the synthesized answer to the MCP Interface in the format specified by the `ask` tool response schema:

- `answer: string` — The natural language answer
- `evidence: [{ subject, predicate, object, graph }]` — Supporting evidence triples from the knowledge graph
- `citations: [{ document_id, chunk_id, page_number, url }]` — Citations back to source documents
- `confidence: float` — Overall confidence score for the answer
- `reasoning_steps: []` — Optional explanation of reasoning and inference steps

The MCP Interface delivers the response to the calling application or user, completing the question-answering workflow.

Q&A Agent Architecture — Key Services

Service	Purpose	Invocation	Key Capabilities
Question Intent Classification Service	Analyze question intent and extract entities	Always	Intent classification, entity extractio n grounded in ontology, semantic similarity search, document retrieval detection, metric recognition and resolution
Query Execution Planning & Optimization Service	Generate optimized execution plans	Conditional	(structured/federated queries)
Virtual Knowledge Graph (VKG) Service	Execute federated queries	Conditional	SPARQL-to-SQL translation, federatio n across Neptune and Data Layer sources
GraphRAG Service	Retrieve document chunks and concepts	Conditional	Hybrid retrieval (vector similarity + graph traversal), guardrails, embedding generatio n, S3 chunk retrieval
Symbolic Reasoning Service	Perform inference and validation	Conditional	OWL/SWRL reasoning, SHACL validation, backward chaining, explanation generation

Cross-Cutting Concerns

- **Intent-Driven Orchestration** — The Question Intent Classification Service (Step 3) is the sole required downstream invocation and serves as the routing decision point for all subsequent steps. Based on the classified intent, suggested query strategy, and extracted entities, the Q&A Agent selectively invokes Query Execution Planning (Step 4), VKG (Step 5), GraphRAG (Step 6), and Symbolic Reasoning (Step 7) as needed. This intent-driven pattern avoids unnecessary service invocations for questions that can be answered through a single retrieval path (e.g., pure document retrieval via GraphRAG without VKG federation).
- **Multi-Modal Retrieval** — The Q&A Agent orchestrates multiple retrieval strategies based on the intent classification (Step 3): structured knowledge retrieval via VKG Service (Step 5) for querying the Semantic Layer Graph Database and virtualized Data Layer sources, unstructured content retrieval via GraphRAG Service (Step 6) for document chunks and concepts from the Lexical Graph Database, or both for hybrid questions requiring structured and unstructured evidence. Both results are reasoned over by the Symbolic Reasoning Service (Step 7) before final synthesis, enabling the agent to answer questions requiring both structured domain knowledge and unstructured document evidence.
- **Open Standards and Open Architecture — All the services provided to the Q&A Agent must be provided to external agents, systems, and users**. The AgentCore Gateway exposes these services as MCP tools and REST APIs, ensuring that semantic capabilities (intent classification, query planning, VKG querying, GraphRAG retrieval, symbolic reasoning) are accessible enterprise-wide, not siloed within the Q&A Agent implementation.
- **Event-Driven Ontology Updates** — Intent Classification and Query Planning services periodically retrieve ontologies and schema mappings, with potential migration to event-driven patterns (Neptune DB Streams) for near real-time synchronization with ontology changes.
- **Multi-Step Execution** — Cross-domain ontology questions often require multiple steps (parallel or serial) to execute, with the Query Execution Planning & Optimization Service orchestrating complex federated queries and coordinating VKG and GraphRAG retrieval.
- **Continuous Reasoning Integration** — The Symbolic Reasoning Service leverages continuous reasoning workflows in the Knowledge Layer, avoiding redundant inference for materialized knowledge while applying reasoning to federated queries combining Data Layer and Semantic Layer Graph Database sources.
- **Observability** — OpenTelemetry traces propagate from MCP Interface → Q&A Agent → AgentCore Gateway → downstream services (Steps 3-7), providing end-to-end visibility via AWS

X-Ray. CloudWatch Logs capture intent classification, query plans, execution statistics, GraphRAG retrieval results, reasoning results, and response confidence scores.

- **Context Awareness** — The Q&A Agent maintains short-term memory via context graphs (Intelligence Layer), enabling multi-turn conversations where follow-up questions can reference previous queries and retrieved entities.
- **Provenance & Explainability** — Every answer includes provenance metadata linking back to source data, source documents, ontology axioms, and reasoning steps, ensuring trust and auditability.
- **Deterministic Metric Resolution** — Recognized metrics resolve from pre-compiled Metric Store expressions, bypassing LLM calculation synthesis and ensuring ACE compliance.

This Q&A Agent architecture enables **semantic, explainable question answering** that leverages federated knowledge graphs, hybrid document retrieval, reasoning engines, and LLM-based synthesis to deliver accurate, grounded answers with full provenance — while ensuring that all semantic capabilities are accessible to external agents, systems, and users through open standards and open architecture.

Governance / Security

Governance / Security

The Semantic Layer implements comprehensive governance and security controls across all components to ensure data protection, access control, compliance, and auditability for enterprise knowledge graph deployments.

Access Control & Authentication

- **Identity & Access Management** - AWS IAM roles and policies enforce least-privilege access for service-to-service communication (ECS tasks calling Neptune, Athena, Redshift, S3, Bedrock, Secrets Manager). Fine-grained permissions control who can read, write, or modify ontologies, mappings, and instance data, enforced through group-based authorization derived from JWT claims.
- **Authentication Model** - The architecture implements a two-tier authentication model:

  Interface Layer (external-facing) - The AgentCore Gateway, MCP Server, and A2A endpoints accept JWT/OIDC bearer tokens from any OIDC-compliant identity provider, including Amazon Cognito, Microsoft Entra ID, Okta, Auth0, Keycloak, and others. For A2A interoperability, IAM

SigV4 is also supported at the Interface Layer to accommodate AWS-native agent callers. The AgentCore Gateway validates inbound tokens and forwards authenticated requests to downstream services.

Internal services (VKG, Symbolic Reasoning, GraphRAG, Intent Classification, Query Planning, Ontology Induction, Information Extraction) - JWT/OIDC only. The AgentCore Gateway propagates the validated JWT to each service, which extracts user identity and authorization context directly from the token. IAM execution roles for ECS tasks handle outbound calls to AWS services (Neptune, Athena, Redshift, S3, Bedrock) via SigV4, managed automatically by the AWS SDK.

Required JWT Claims - All JWT tokens must include:

- `sub` - unique user identifier (for row-level security, audit logging, and provenance tracking)
- `groups` - authorization groups (for named graph restrictions, data masking, and role-based access control). Maps to roles such as `ontology-admin`, `knowledge-engineer`, `data-reader`, `agent-operator`
- `act` - agent identity claim (RFC 8693), identifying the agent acting on behalf of the user (for audit logging, delegation chain tracking, and distinguishing which agent performed an action). Supports nested delegation for multi-agent workflows

Authorization Scope - A single OAuth scope (`semantic-layer:access`) grants entry to the Semantic Layer. Fine-grained authorization is enforced through group-based policies derived from the `groups` JWT claim, not through multiple per-service scopes.

Row-Level Security - The architecture implements row-level security through three complementary techniques based on the target data source:

Neptune (SPARQL Query Rewriting) - For queries targeting the Semantic Layer Graph Database, the VKG Service extracts the authenticated user's identity and groups from the JWT, resolves authorization policies (via OPA, Lake Formation, or ontology-defined access rules), and injects FILTER clauses or named graph restrictions into the SPARQL query before execution. This approach also enables data masking and redaction of sensitive attributes (PII, PHI) based on user roles.

AWS Data Layer Sources (Trusted Identity Propagation) - For queries targeting virtualized Data Layer sources (Athena, Redshift), the VKG Service leverages AWS Trusted Identity Propagation (TIP) via IAM Identity Center. The end-user's IdP identity is propagated through IAM Identity

Center when establishing JDBC connections, enabling Lake Formation fine-grained access controls (for Athena) and native row-level security/column-level security (for Redshift) to enforce per-user data access policies at the source.

Non-AWS Data Layer Sources (OAuth/Session-Based) - For non-AWS sources (Snowflake, Databricks), the VKG Service uses OAuth token propagation or per-user session establishment to enable native row access policies (Snowflake) and Unity Catalog permissions (Databricks). For sources that do not support identity propagation, the VKG Service falls back to SQL query rewriting, injecting WHERE clauses into the translated SQL based on the user's authorization context.

Agent Identity - The `act` claim in the JWT identifies which agent is acting on behalf of the user. This is critical for provenance, audit logging, and distinguishing agent actions in multi-agent workflows. For A2A delegation chains, the `act` claim supports nesting (RFC 8693) to track the full delegation path. The emerging Agentic JWT (A-JWT) IETF draft extends this pattern with cryptographic agent fingerprinting for organizations requiring stronger agent identity guarantees.

Data Protection & Encryption

- **Encryption at Rest** — Neptune DB, OpenSearch, and S3 buckets enforce server-side encryption using AWS KMS with customer-managed keys (CMK) for sensitive ontologies and knowledge graphs.
- **Encryption in Transit** — All communication between Semantic Layer components uses TLS 1.2+ encryption. VPC endpoints ensure private connectivity without traversing the public internet.
- **Data Masking & Redaction** — Sensitive attributes (PII, PHI) can be masked or redacted at query time based on user roles, using SPARQL query rewriting or post-processing filters in the VKG Service.

Network Security & Isolation

- **VPC-Native Architecture** — Neptune DB, VKG Service (ECS), Symbolic Reasoning Service, and Lambda functions run within private subnets with no direct internet access. Security groups and NACLs restrict traffic to authorized sources.
- **VPC Endpoints** — Private connectivity to AWS services (S3, Secrets Manager, CloudWatch) via VPC endpoints eliminates exposure to the public internet.

- **API Gateway & WAF** — AgentCore Gateway fronts the VKG Service with AWS WAF rules to protect against SQL injection, SPARQL injection, and DDoS attacks.

Ontology & Mapping Governance

- **Version Control** — Ontologies (T-Box) and R2RML mappings are versioned using **S3 Versioned Buckets** or **Git repositories** (recommended implementations). S3 versioning provides automatic version history with rollback capabilities, while Git repositories enable collaborative development with branch-based workflows, pull requests, and approval gates. Neptune named graphs enable side-by-side deployment of ontology versions for A/B testing or gradual migration.
- **Change Management** — Ontology modifications trigger automated validation workflows (ELK reasoner for consistency checking, SHACL for constraint validation) before deployment. Neptune DB Streams emit change events for audit logging and downstream notification. Changes are tested in non-production Neptune instances before promotion to production.
- **Approval Workflows** — Critical ontology changes (class hierarchy modifications, property domain/range changes, breaking changes to R2RML mappings) require multi-party approval via internal workflow systems or third-party tools like ServiceNow. Approval policies are enforced through Git branch protection rules or S3 bucket policies that require signed approvals before deployment.
- **Catalog Integration** — **AWS Glue Data Catalog** and **Amazon SageMaker Catalog** track lineage from source data (Data Layer) through R2RML mappings to Neptune entities, enabling impact analysis and compliance reporting. Open-source and third-party data catalog platforms including **DataHub** (LinkedIn's open-source metadata platform) and **Collibra** (enterprise data governance platform) can be integrated via APIs or OpenLineage-compatible connectors to provide unified metadata management, business glossary alignment, and cross-platform lineage tracking across hybrid and multicloud environments.

Data Lineage & Provenance

- **Catalog Integration** — AWS Glue Data Catalog and Amazon SageMaker Catalog track lineage from source data (Data Layer) through R2RML mappings to Neptune entities, enabling impact analysis and compliance reporting.
- **Provenance Metadata** — Every triple in Neptune includes provenance annotations (source system, extraction timestamp, confidence score, validation status) stored as reified statements or named graph metadata.

- **Audit Logging** — All SPARQL queries, ontology modifications, and reasoning operations are logged to CloudWatch Logs with user identity, timestamp, and query details for compliance audits.

Compliance & Regulatory Controls

- **Data Residency** — Neptune DB and S3 buckets are deployed in specific AWS regions to meet data residency requirements (e.g., GDPR, HIPAA).
- **Retention Policies** — S3 lifecycle policies and Neptune backup retention settings enforce data retention requirements. Automated deletion of expired knowledge graph data based on ontology-defined retention rules.
- **Compliance Frameworks** — Architecture supports HIPAA, GDPR, SOC 2, and FedRAMP compliance through encryption, access controls, audit logging, and data residency controls.

Reasoning & Validation Security

- **Trusted Reasoning Engines** — Symbolic Reasoning Service (ELK, SHACL, SWRL) runs in isolated ECS tasks with read-only access to Neptune. Reasoning results are validated before being asserted into the knowledge graph.
- **Constraint Enforcement** — SHACL shapes define data quality and security constraints (e.g., "PII fields must be encrypted", "Claims must have valid submitter"). Violations are flagged and prevented from entering the Semantic Layer Graph Database.
- **Automated Reasoning Guardrails** — Amazon Bedrock Automated Reasoning validates critical inferences to prevent incorrect or malicious knowledge from being asserted.

Monitoring & Incident Response

- **Observability** — OpenTelemetry traces, CloudWatch Logs, and X-Ray provide end-to-end visibility into query execution, reasoning workflows, and access patterns. Anomaly detection alerts on unusual query patterns or access attempts.
- **Security Monitoring** — AWS GuardDuty and Security Hub monitor for threats. CloudTrail logs all API calls to Neptune, S3, and IAM for forensic analysis.
- **Incident Response** — Automated runbooks (AWS Systems Manager) respond to security events (e.g., unauthorized access attempts, SPARQL injection detection) by isolating affected resources and notifying security teams.

This governance and security framework ensures that the Semantic Layer meets enterprise requirements for data protection, access control, compliance, and auditability — enabling trusted, production-grade knowledge graph deployments for agentic AI applications.

Technology Tradeoffs & Alternatives

> **Note**
>
> : AWS service features, pricing, instance families, and capabilities evolve over time. The technical details, comparisons, and recommendations in this section are accurate as of **May 2026**. Always verify current service capabilities, limits, and pricing against the latest AWS documentation before making architectural decisions.

Amazon Neptune Database — Deployment Options & Capabilities

Amazon Neptune is the recommended managed graph database for this architecture. It supports both **RDF/SPARQL** (W3C standards for semantic web) and **Apache TinkerPop/Gremlin** (labeled property graph), making it uniquely suited for semantic layer workloads that require formal ontology representation alongside flexible graph traversal.

Deployment Modes

Deployment Mode	Description	Best For
Neptune DB (Provisioned)	Fixed-capacity instances with up to 15 read replicas. Current-generation instance families include: **R8g** (Graviton4), **R7g** (Graviton3), **R7i**, **R6g**, **R6i**, **R5/R5d**, **X2iedn** (memory-optimized), and burstable **T4g/T3**	Predictable, steady-state workloads with consistent throughput requirements
Neptune Serverless	Auto-scaling compute from 1.0 to 128 NCUs; pay-per-use	Variable/bursty workloads, development environments, or workloads with unpredict able query patterns

Neptune Global Database	**Note** Cross-region replication with <1s replication lag; up to 5 secondary regions; secondary clusters support up to 16 read replicas each. : Global Database setup documentation currently references `r5` and `r6g` instance classes — verify current support for newer instance families (R7g, R8g) before deploying	Disaster recovery, low-latency reads across geographies, regulatory data residency requirements
Neptune Analytics	High-performance, in-memory OLAP engine for graph analytics; supports openCypher queries	Large-scale graph analytics, PageRank, community detection, shortest path, and pattern matching at billion-edge scale

Key Capabilities Relevant to the Semantic Layer

- **RDF & SPARQL Support**: Neptune natively stores RDF triples and supports SPARQL 1.1, enabling direct representation of OWL ontologies, SKOS taxonomies, and SHACL constraints — foundational to the semantic layer's knowledge representation tier.
- **Read Replicas**: Up to 15 replicas for read scaling; critical when multiple AI agents concurrently query the ontology graph for reasoning and context retrieval.
- **Neptune Streams**: Change-data-capture (CDC) for graph mutations, enabling downstream synchronization with vector stores and search indexes.

- **Neptune ML**: Built-in graph neural network (GNN) integration for link prediction and node classification — useful for ontology enrichment and entity resolution.
- **High Availability**: Multi-AZ deployments with automated failover; storage replication across 3 AZs.

Integration with Amazon OpenSearch Service and Neptune Analytics

For the semantic layer architecture, **Neptune DB handles structural graph queries** (SPARQL queries, graph traversal, pattern matching), **Neptune Analytics provides combined graph + vector search** (openCypher only), and **OpenSearch provides vector-based semantic similarity search** and hybrid retrieval. Note that Neptune does not include a built-in OWL reasoner — ontology inference (e.g., class subsumption, property inheritance) must be performed externally using a reasoning engine (e.g., HermiT, Pellet, ELK) or pre-materialized as inferred triples before loading into Neptune.

Capability	Neptune DB	Neptune Analytics	OpenSearch	Combined Value
Graph traversal	# Native (SPARQL, Gremlin, openCypher)	# Native (openCypher only)	#	Navigate ontology hierarchies, class relationships
SPARQL/RDF	# Native	# (supports RDF data import, but queries are openCypher only)	#	Query and traverse OWL/RDFS ontologies stored as RDF triples (inference must be pre-materialized or performed externally)

openCypher	# Native	# Native	#	Graph pattern matching and analytics
Vector similarity	#	# Built-in vector search (k-NN on graph nodes)	# k-NN, HNSW	Semantic similarity for entity resolution, concept matching
Graph + vector in single query	#	# Combine graph traversal with vector similarity in one openCypher query	#	Find semantically similar nodes within a graph neighborhood — e.g., "entities related to X that are also similar to embedding Y"
Graph analytics	Limited	# PageRank, community detection, shortest path, centrality — in-memory OLAP	#	Ontology structure analysis, influence propagation, entity importance ranking
Hybrid search (BM25 + vector)	#	#	# BM25 + k-NN	Combine keyword and semantic relevance for agent context retrieval
Full-text search	# (via OpenSearch integration)	#	# Native	Keyword-based entity lookup, label matching

Faceted analytics	Limited	Limited	# Aggregations	Dashboard-style analytics over entity attributes

Integration patterns:

- **Neptune DB + OpenSearch**: Neptune's full-text search feature can be backed by an OpenSearch cluster, enabling SPARQL queries to include `neptune-fts:search` predicates that leverage OpenSearch's text indexes. For vector similarity, a separate OpenSearch index stores embeddings of ontology entities and their descriptions, enabling agents to find semantically similar concepts even when exact terminology differs.
- **Neptune Analytics for graph + vector**: Neptune Analytics can load RDF data from Neptune DB or S3 and expose it via openCypher queries with built-in vector search — enabling queries that combine graph traversal *and* vector similarity in a single operation (e.g., "find the top-5 semantically similar concepts within 2 hops of `finance:RevenueMetric`"). **Key tradeoff**: Neptune Analytics supports RDF as an import format but **does not support SPARQL queries** — all queries must use openCypher. This means workloads that depend on SPARQL for querying OWL/RDF ontologies and SHACL validation must remain on Neptune DB, while Neptune Analytics serves as a complementary high-performance analytics and vector search tier.
- **Tiered pattern**: Use Neptune DB as the authoritative SPARQL/RDF store for ontology management and querying (with pre-materialized inferred triples), Neptune Analytics for graph analytics and combined graph+vector queries, and OpenSearch for hybrid BM25+vector search and full-text retrieval.

Amazon S3 Vectors as an Alternative for Vector Storage

Amazon S3 Vectors (GA since December 2025) is a cost-effective alternative to OpenSearch for vector storage when:

Consideration	S3 Vectors	OpenSearch Serverless
Cost	Up to 90% lower for large-scale storage	Higher (compute + storage)
Latency (cached)	Subsecond (~100–500ms)	Single-digit millisecond

Latency (non-cached)	Higher (~1–3s cold)	Consistent low latency
Hybrid search	# No BM25 + vector	# Native hybrid
Advanced filtering	Basic metadata filtering	Rich DSL, geo, nested
Scale	Petabyte-scale, pay-per-storage	Compute-bound scaling
Management	Zero ops (serverless, no clusters)	Serverless option available, but more config

When to choose S3 Vectors over OpenSearch:

- The semantic layer's vector search workload is primarily **batch or near-real-time** (not sub-second interactive)
- **Hybrid search is not required** — queries are either pure vector similarity or handled by Neptune's SPARQL engine
- The vector corpus is very large (millions to billions of embeddings) and **cost optimization** is a priority
- The architecture already uses Neptune for structural queries, and vector similarity is a secondary retrieval mechanism

When to stay with OpenSearch:

- Agents require **sub-millisecond** vector retrieval for real-time reasoning
- **Hybrid search** (BM25 + k-NN) is needed for combining keyword relevance with semantic similarity
- Complex **filtering, geo queries, or nested aggregations** are needed alongside vector search

> **Note**
>
> : S3 Vectors and OpenSearch are complementary. S3 Vectors can serve as a **source-of-truth vector store** with one-click export to OpenSearch Serverless collections for

high-performance query workloads. This tiered approach optimizes for both cost and performance.

Compute Tradeoffs: ECS (EC2 vs Fargate) vs Lambda

The semantic layer architecture includes several components and services — ontology ingestion pipelines, SPARQL query mediators, R2RML mapping engines, embedding generators, and agent orchestration endpoints — each with different compute characteristics. Choosing the right compute model affects cost, latency, operational complexity, and scaling behavior.

Compute Option	Description	Best For	Considerations
Lambda	Serverless functions ; event-driven; up to 15 min execution, 10 GB memory; **10 GB container image size limit**	Short-lived, event-driven tasks: SPARQL query dispatch, webhook handlers, Neptune Streams event processing, embedding generatio n for individual entities, SHACL validation triggers	Cold starts (100ms–2s depending on runtime/VPC); 15 min max execution; no persistent connectio ns (e.g., cannot maintain long-lived WebSocket connectio ns to Neptune); **container image size limit of 10 GB** — VKG engines like Ontop with their full dependency stack (JVM, JDBC drivers, ontology/mapping files) may exceed this limit, making Lambda unsuitabl e for hosting VKG services
ECS on Fargate	Serverless container s; no instance	Persistent services: VKG/R2RML mapping	Higher per-unit cost than EC2; minimum

	management; supports long-running tasks	engines (e.g., Ontop), SPARQL federation endpoints, ontology API servers, agent orchestration services requiring warm connection pools to Neptune	task size 0.25 vCPU / 512 MiB; per-second billing with 1-minute minimum; startup latency typically 5–15s for cached images (up to 20–30s for large images on first pull or with VPC ENI attachment)
ECS on EC2	Container orchestration on self-managed EC2 instances	High-throughput, cost-sensitive workloads: large-scale ontology induction batch jobs, bulk RDF loading, embedding batch generation; GPU workloads (Neptune ML GNN training)	Requires capacity planning and instance management; patch responsibility; use Spot Instances for batch workloads (up to 90% savings); Savings Plans for steady-state

When to Use Which

Lambda is the default choice for:

- Event-driven ontology pipelines: S3 object creation → trigger SHACL validation → load to Neptune
- Neptune Streams consumers: process CDC events and update vector stores
- Lightweight SPARQL query proxies for agent tool invocations (if query + response fits within 15 min and doesn't require persistent connections)
- One-off embedding generation for individual entities via Bedrock

Fargate is recommended for:

- **VKG engines** (e.g., Ontop) that require bootstrapping with R2RML mappings and ontology configurations, maintaining persistent JDBC connections to relational data sources, and serving SPARQL queries over virtualized RDF — these are long-running services with startup initialization
- **SPARQL federation endpoints** that maintain connection pools to multiple Neptune clusters or external SPARQL endpoints
- **Agent orchestration services** that need warm connections and low-latency response for real-time agentic reasoning
- **Ontology management APIs** serving import/validation/versioning workflows

ECS on EC2 is justified when:

- **GPU instances** are needed (e.g., p3, g5 for Neptune ML GNN training, large-scale embedding generation)
- **Batch processing** at scale where Spot Instances significantly reduce cost (ontology induction from large data catalogs, bulk RDF transformations)
- **High-memory workloads** that exceed Fargate's 120 GB limit (e.g., in-memory ontology reasoning with very large knowledge bases)
- **Cost optimization** at sustained high utilization (Fargate carries ~20-30% premium over equivalent EC2 capacity)

Virtual Knowledge Graphs (VKG) and R2RML Mapping Services

A key component of the semantic layer is the **Virtual Knowledge Graph (VKG)** pattern, which exposes relational and tabular data as RDF through ontology-based data access (OBDA) — without requiring physical data migration into Neptune. VKG engines translate SPARQL queries into SQL at query time, using **R2RML mappings** (W3C standard for RDB-to-RDF mapping) that define how relational schemas map to ontology classes and properties.

Why VKG Matters for the Semantic Layer

- **Data stays in place**: Enterprise data in Amazon RDS, Aurora, Redshift, or Athena is accessed *in situ* through SPARQL, avoiding expensive and fragile ETL pipelines to materialize all data as RDF triples in Neptune.

- **Ontology-driven access**: The R2RML mappings encode the relationship between the physical schema (tables, columns) and the logical ontology (OWL classes, properties), ensuring that SPARQL queries return semantically consistent results regardless of the underlying data model.
- **Composability with materialized graphs**: VKG results can be federated with materialized RDF in Neptune using SPARQL `SERVICE` clauses, combining curated ontology knowledge (in Neptune) with live operational data (via VKG) in a single query.

VKG Engines and Deployment Considerations

Engine	Description	Deployment Model	Configuration Dependencies
Ontop	Leading open-source VKG engine; translates SPARQL → SQL using R2RML/OBDA mappings; supports OWL 2 QL ontologies for query rewriting	**Fargate (recommended)**: Ontop requires bootstrapping with an ontology file (.owl/.ttl), R2RML mapping file (.obda/.ttl), and JDBC connection configuration. These are loaded at startup and kept in memory for query rewriting. Restart-on-config-change pattern via ECS task definition updates.	OWL ontology, R2RML mappings, JDBC driver + connection string to data source (RDS, Aurora, Redshift)
Custom SPARQL-to-SQL mediator	Lightweight, application-specific mapping layer for simple schema-to-ontology projections	**Lambda** (if stateless, low-latency) or **Fargate** (if connection pooling needed)	Mapping configuration (can be stored in S3 or DynamoDB), ontology reference

R2RML Mapping Lifecycle

R2RML mappings are a critical configuration artifact tied directly to the ontology — when the ontology evolves, mappings must be updated in lockstep. Recommended practices:

- **Version R2RML mappings alongside ontology artifacts** in the same source-controlled repository or S3 versioned bucket, with the same approval workflow (reasoner validation → SHACL check → peer review → promotion).
- **Validate mappings against the ontology**: Before deploying updated mappings, verify that:
  - All mapped RDF predicates and classes exist in the current ontology version
  - Mapped SQL queries still return valid results against the current data source schema (integration tests)
  - SHACL constraints on the ontology are satisfiable by the data returned through the mappings
- **Configuration injection for VKG services**: Store ontology files, R2RML mappings, and JDBC connection metadata in **AWS Systems Manager Parameter Store** or **Secrets Manager** (for credentials), and inject them into Fargate tasks via environment variables or mounted EFS volumes. Use ECS task definition revisions to trigger rolling updates when mappings change.
- **Bootstrapping pattern**: VKG engines like Ontop load the full ontology + mapping configuration at startup. Design the Fargate service with:
  - **Health checks** that verify the SPARQL endpoint is responding after bootstrap
  - **Rolling deployments** (ECS service deployment circuit breaker) to ensure zero-downtime mapping updates
  - **Entrypoint scripts** in the container image to pull the latest mapping artifacts from S3 before starting the VKG engine

Open-Source Alternatives: RDF/SPARQL Graph Databases

While Neptune is the recommended managed option, the semantic layer architecture's reliance on W3C standards (RDF, SPARQL, OWL, SHACL) means it is portable to other compliant graph databases. Organizations may consider open-source alternatives for:

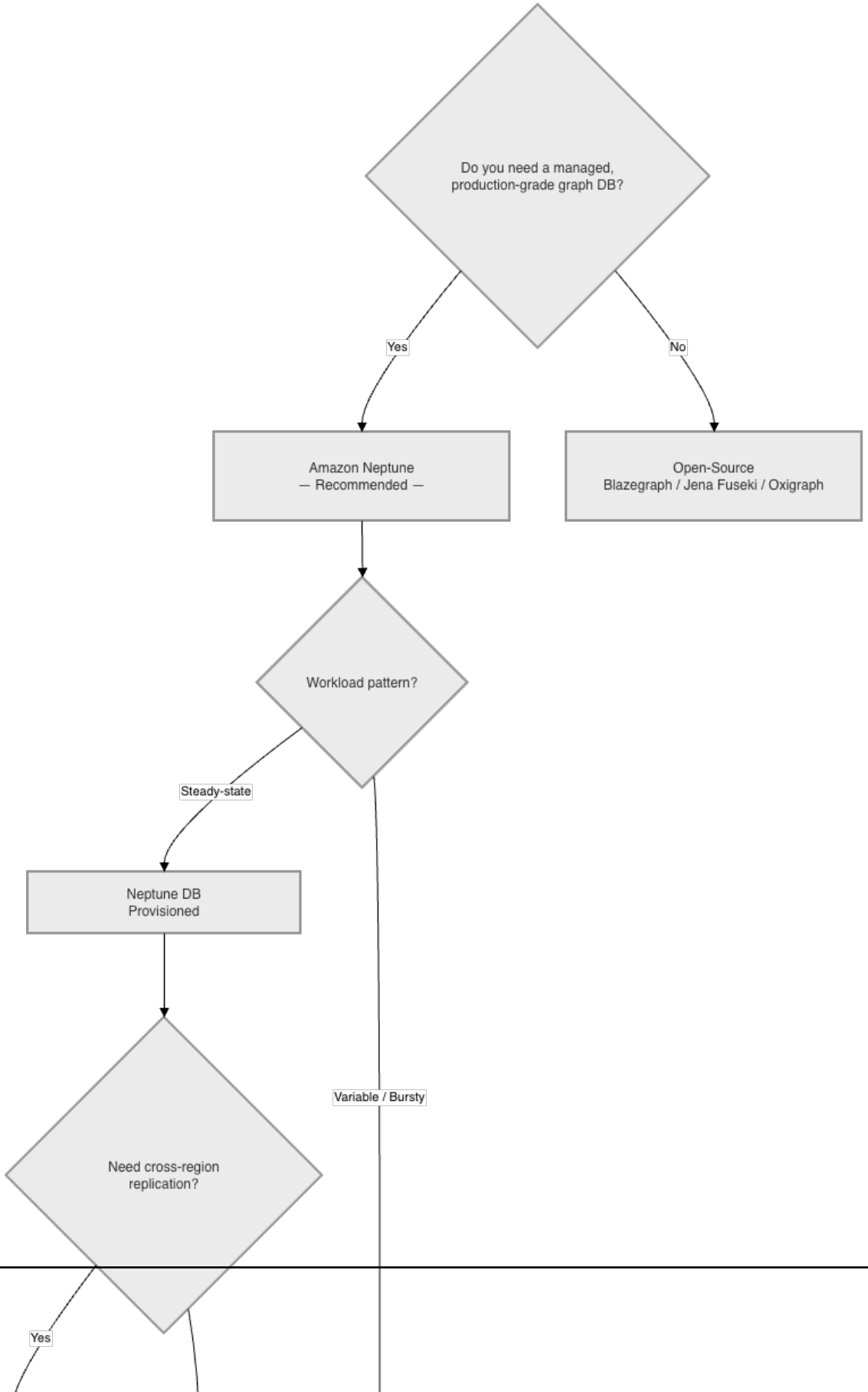
- **Self-managed/on-premises deployments** where managed services are not available
- **Cost sensitivity** in development and testing environments
- **Specific feature requirements** (e.g., advanced reasoning engines, SPARQL federation)

Database	License	RDF/SPARQL	Key Strengths	Considerations
Blazegraph	GPLv2	# Full SPARQL 1.1	High-performance OLAP/OLTP; used by Wikidata	No active development since 2020; community-maintained; no managed AWS offering
Apache Jena Fuseki	Apache 2.0	# Full SPARQL 1.1	Mature, well-documented; TDB2 storage; SHACL validation built-in	Requires self-managed infrastructure; limited horizontal scaling
Oxigraph	Apache 2.0 / MIT	# SPARQL 1.1	Rust-based, embeddable, fast; suitable for edge/embedded scenarios	Younger project; smaller community; limited enterprise features

Migration considerations: Because the semantic layer uses standard RDF serializations (Turtle, N-Triples, JSON-LD) and SPARQL endpoints, workloads can be migrated between Neptune and open-source alternatives with minimal application changes. The primary integration points to validate are:

1. SPARQL query compatibility (Neptune supports SPARQL 1.1 with some extensions)
2. Authentication and network security (Neptune uses IAM Sig4; open-source typically uses HTTP basic auth or custom)
3. Operational aspects (backup, monitoring, scaling) that Neptune handles automatically

Decision Framework



Best practices

Best Practices

Start Small — A Little Semantics Goes a Long Way Begin with a focused domain or high-value use case rather than attempting enterprise-wide semantic coverage from day one. A well-designed ontology for a single business domain (e.g., claims processing, customer relationships, product catalog) delivers immediate value and establishes patterns that can be replicated across the organization. Early wins build organizational confidence, demonstrate ROI, and create momentum for broader adoption.

Develop Showcase Questions and Measure Performance Over Time Define a curated set of representative questions that reflect real business needs and user workflows. These showcase questions serve as both acceptance criteria and regression tests — track query performance (latency, accuracy, completeness), reasoning quality (inference correctness, explanation clarity), and user satisfaction over time. As the semantic layer evolves, these questions validate that improvements enhance rather than degrade the user experience, and they provide concrete evidence of value to stakeholders.

Integrate Data Sources Incrementally Virtualize existing data sources through R2RML mappings before materializing knowledge graphs, enabling rapid integration without upfront data migration. Prioritize data sources based on business value, data quality, and integration complexity. As new sources are added, leverage ontology induction to accelerate mapping creation, but always validate with domain experts. This incremental approach reduces risk, delivers continuous value, and allows the semantic layer to grow organically with organizational needs.

Improve Data Products to Reduce Semantic Gaps Where appropriate, enhance the data products that are fundamental to the Data Layer by exposing views or semantic views that align with your ontology structure. By creating database views, materialized views, or data product interfaces that reflect ontology classes and relationships, you close the gap between raw data structures and semantic concepts — reducing the complexity of R2RML mappings and improving query performance. This approach enables the VKG Service to retrieve the right data more efficiently, as source systems provide pre-structured outputs that match ontology patterns rather than requiring complex SQL transformations at query time.

Embrace Human-in-the-Loop Workflows Automated ontology induction, entity resolution, and reasoning are powerful accelerators, but domain expertise remains essential. Implement approval

workflows for ontology changes, validation checkpoints for entity disambiguation, and review processes for high-impact inferences. Human-in-the-loop patterns ensure semantic quality, build trust with business stakeholders, and capture organizational knowledge that cannot be derived from data alone.

Prioritize Governance from Day One Establish versioning, provenance tracking, and approval workflows before deploying to production. Use draft/production named graphs to isolate experimental changes from production queries. Maintain ontology versions in S3 or Git repositories with rollback capabilities. Document R2RML mappings, reasoning rules, and data quality constraints as semantic artifacts with full lineage. Strong governance enables confident evolution of the semantic layer without disrupting downstream consumers.

Leverage Standards for Interoperability and Longevity Adhere to W3C Semantic Web standards (RDF, OWL, SPARQL, SHACL, R2RML) to ensure that your semantic layer integrates with best-of-breed tools, avoids vendor lock-in, and benefits from decades of research and production deployments. Standards-based implementations are future-proof, enabling you to adopt new reasoning engines, query optimizers, or visualization tools as the ecosystem evolves.

Optimize for Query Performance Early Monitor query latency, federated query execution plans, and reasoning overhead from the start. Use cardinality statistics and data quality metrics to inform query optimization. Materialize frequently accessed sub-graphs or reasoning results when virtualization introduces unacceptable latency. Balance the flexibility of virtual knowledge graphs with the performance of materialized knowledge to meet user expectations.

Invest in Observability and Explainability Implement OpenTelemetry tracing across the semantic stack to diagnose performance bottlenecks and understand query execution paths. Capture provenance metadata for all assertions, inferences, and entity resolutions. Provide citations and reasoning chains in agent responses to build user trust. Observability and explainability are not optional — they are foundational to operating trustworthy semantic AI in production.

These best practices reflect lessons learned from semantic layer deployments across industries and ensure that your semantic data foundation delivers sustainable value while scaling with organizational maturity.

Well-Architected Considerations

The semantic layer architecture should be evaluated against the six pillars of the AWS Well-Architected Framework. The following guidance maps architectural decisions in this guide to each pillar.

1. Operational Excellence

Design Principles Applied:

- **Perform operations as code**: Infrastructure components (Neptune clusters, OpenSearch domains, VPCs, Lambda functions, Step Functions workflows) are defined using AWS CloudFormation or CDK, enabling version-controlled, repeatable deployments.
- **Make frequent, small, reversible changes**: Ontology evolution follows semantic versioning; changes to class hierarchies or property definitions are deployed incrementally with backward-compatible SPARQL queries.
- **Anticipate failure**: The architecture uses Neptune's Multi-AZ deployment with automated failover and OpenSearch's managed availability to minimize single points of failure.

Ontology Lifecycle Management:

- **Import and validation**: Support structured import of ontology artifacts (OWL, SHACL, SKOS) with automated validation using OWL reasoners (e.g., HermiT, Pellet, or ELK) to check consistency, satisfiability, and classification correctness before promotion to production graphs.
- **Versioning and approval**: Maintain ontology versions in a source-controlled repository (e.g., CodeCommit, GitHub) and/or **S3 versioned buckets** (enabling object-level version history, rollback, and lifecycle policies for ontology artifacts serialized as Turtle, N-Triples, or JSON-LD) with a formal approval workflow — ontology changes should pass reasoner validation, SHACL constraint checks, and peer review before being loaded into Neptune.
- **Ontology Induction from data catalogs**: Support automated or semi-automated ontology induction from existing business data catalogs (e.g., **Amazon SageMaker Catalog**) and technical data catalogs (e.g., **AWS Glue Data Catalog**). Induced ontologies extract entity types, relationships, and attribute schemas from catalog metadata (table schemas, column lineage, business glossaries) and propose them as candidate OWL classes and properties. These candidates must go through the same validation pipeline — reasoner checks for logical

consistency, SHACL validation for constraint compliance, and a **Human-in-the-Loop (HITL)** review process where domain experts approve, refine, or reject induced ontology fragments before they are merged into the production semantic layer.

Operational Recommendations:

- Use **Amazon CloudWatch** for monitoring Neptune query latency, SPARQL endpoint availability, and OpenSearch cluster health.
- Implement **Neptune Streams** to capture graph mutations and trigger downstream processing (e.g., re-embedding changed entities into the vector store).
- Use **AWS X-Ray** for distributed tracing across the agent orchestration layer (AgentCore Runtime → Lambda → Neptune → OpenSearch) to identify bottlenecks.
- Establish **runbooks** for common operational tasks: ontology reload, graph snapshot/restore, embedding re-indexing.

2. Security

Design Principles Applied:

- **Implement a strong identity foundation**: Neptune access is controlled through **IAM authentication** (Signature Version 4) with least-privilege policies. AgentCore Runtime agents assume scoped IAM roles for graph queries.
- **Enable traceability**: All SPARQL queries and graph mutations are logged via **Neptune Audit Logs** to CloudWatch Logs. API Gateway access logs capture agent-to-endpoint interactions.
- **Apply security at all layers**: The architecture uses **VPC isolation** for Neptune (private subnets, no public endpoints), **VPC endpoints** for OpenSearch Serverless, and **AWS PrivateLink** for Bedrock and AgentCore.
- **Protect data in transit and at rest**: Neptune encrypts storage at rest using AWS KMS; SPARQL endpoints use TLS. OpenSearch encrypts indexes at rest and enforces HTTPS.

Security Recommendations:

- **Cedar policy language for authorization**: Use the **Cedar policy language** to define fine-grained, attribute-based access control (ABAC) policies for the semantic layer. Cedar can be deployed through **Amazon Verified Permissions** (fully managed) or as a **custom authorizer**

using the open-source Cedar SDK embedded in a Lambda function, Fargate service, or agent orchestration layer — the latter gives full control over policy storage, evaluation context, and integration patterns. Cedar policies can express rules such as: *"Agent X can query ontology classes in namespace* `finance:` *only if the requesting user belongs to the* `FinanceAnalysts` *group."* This provides a centralized, auditable authorization layer that is decoupled from application code and can enforce policies across Neptune SPARQL queries, OpenSearch indexes, and AgentCore tool invocations.

- **RDF namespaces for multi-tenancy and access control**: Use **RDF named graphs and namespace prefixes** to logically partition ontology data by domain, team, business unit, or group (e.g., `finance:`, `engineering:`, `marketing:`). Each namespace maps to a distinct named graph in Neptune, enabling:
  - **Namespace-scoped access control**: Cedar policies and IAM conditions reference the namespace/named graph to grant or deny access — e.g., users in the `FinanceAnalysts` IdP group can query `finance:` graphs, while `EngineeringLeads` can query `engineering:` graphs.
  - **Ownership and delegation**: Namespace ownership is assigned to users or groups sourced from the organization's **Identity Provider (IdP)** via **OIDC** or **SAML** federation through IAM Identity Center. Group memberships from the IdP (e.g., Active Directory groups, Okta groups, Azure AD groups) are propagated as session tags or SAML attributes, which Cedar policies and IAM session policies evaluate at query time.
  - **Tenant isolation**: SPARQL queries are scoped to specific named graphs using `FROM` or `FROM NAMED` clauses, and the agent orchestration layer enforces that agents can only access graphs authorized for the delegating user's IdP groups. This prevents cross-tenant data leakage while allowing shared ontology base classes (e.g., a common `core:` namespace) to be accessible across all tenants.
  - **Hierarchical namespace structure**: Support nested namespace patterns (e.g., `org:finance:risk:`, `org:finance:compliance:`) for fine-grained scoping within large organizations, with inheritance rules defined in Cedar policies.
- **Service accounts for agent identities**: AI agents (AgentCore Runtime agents, Lambda-based orchestrators) should operate using **dedicated IAM service accounts** (IAM roles) with tightly scoped permissions — not shared or overly broad roles. Each agent role should have:
  - Least-privilege access to specific Neptune graph endpoints and named graphs
  - Scoped OpenSearch index permissions (read-only vs. read-write)
  - Explicit deny policies preventing cross-tenant or cross-domain access

  - Session tags or context keys for Cedar policy evaluation
- **User credential delegation for data access**: When agents act on behalf of human users, the architecture must ensure that **the user's own credentials and entitlements** govern data access — not the agent's service account alone. This is critical for maintaining data governance and audit compliance. **AgentCore Identity** provides the identity propagation framework for this pattern. Implementation patterns include:
  - **Identity propagation with IAM Identity Center and AgentCore Identity**: Pass the authenticated user's identity through the agent invocation chain using trusted identity propagation and AgentCore's identity resolution, so downstream services (Neptune, OpenSearch, S3) enforce access based on the user's IdP group memberships and attribute-based policies. AgentCore Identity resolves the calling user's identity context and makes it available to tool implementations and Cedar policy evaluation.
  - **Session policies**: Attach inline session policies when agents assume roles on behalf of users, constraining the agent's permissions to the intersection of the service account role and the user's entitlements.
  - **Cedar + user context**: Pass user attributes (department, role, clearance level) as Cedar context in authorization requests, so that even when the agent's service account makes the API call, the Cedar policy evaluates against the *user's* attributes to determine which ontology classes, named graphs, or search indexes are accessible.
  - **Audit trail**: Log both the agent service account identity *and* the delegating user's identity in CloudTrail and Neptune audit logs, ensuring full traceability of "who asked the agent to do what."
- Use **SHACL constraints** in the ontology layer as a semantic-level validation mechanism — validate that agent-submitted graph mutations conform to the ontology schema before they are committed.
- Implement **fine-grained access control** in OpenSearch for multi-tenant semantic layers (document-level security by ontology namespace), integrated with the Cedar authorization layer.
- Encrypt sensitive entity attributes (PII in knowledge graphs) using **client-side encryption** before storing in Neptune.
- Use custom classifiers or data scanning tools to audit ontology data for unintended PII exposure during bulk loads.

3. Reliability

Design Principles Applied:

- **Automatically recover from failure**: Neptune Multi-AZ with automatic failover (typically 1–2 minutes for standard clusters); OpenSearch Serverless with built-in replication.
- **Scale horizontally**: Neptune read replicas for query fan-out; OpenSearch auto-scaling for vector search throughput.
- **Manage change in automation**: Ontology deployments through CI/CD pipelines with SHACL validation gates.

Reliability Recommendations:

- Use **Neptune Global Database** for disaster recovery when the semantic layer is mission-critical (RPO < 1 second, RTO < 1 minute for read workloads).
- Implement **circuit breakers** in the agent orchestration layer: if Neptune or OpenSearch becomes unavailable, agents should gracefully degrade (e.g., fall back to cached ontology subsets or simplified reasoning).
- Design **idempotent** graph mutation operations — agents may retry failed writes, and the ontology update pipeline should handle duplicate triples gracefully.
- Neptune cluster volume storage **scales automatically** in 10 GB segments (up to 128 TiB) as data grows — no configuration required. Monitor `VolumeBytesUsed` via CloudWatch to track ontology growth trends and plan for cost implications of expanding storage.
- Test **failure modes** regularly: simulate AZ failure, Neptune failover, OpenSearch shard migration.

4. Performance Efficiency

Design Principles Applied:

- **Use purpose-built databases**: Neptune for graph traversal and SPARQL querying, OpenSearch for vector similarity and hybrid search — each service optimized for its query pattern.
- **Go global in minutes**: Neptune Global Database and OpenSearch cross-region replication enable low-latency access from any region.
- **Experiment more often**: Neptune Serverless allows rapid prototyping of new ontology patterns without capacity provisioning.

Performance Recommendations:

- **SPARQL query optimization**: Use `OPTIONAL` patterns sparingly; prefer bound variable patterns. Leverage Neptune's query plan explanations (`EXPLAIN`/`PROFILE`) to identify expensive operations.
- **Caching layer**: Deploy **Amazon ElastiCache (Redis)** to cache frequently-accessed ontology subgraphs (e.g., class hierarchies, property domains/ranges) that rarely change — reducing Neptune query load during high-concurrency agent operations.
- **Neptune Analytics** for bulk graph computations: offload expensive analytics (community detection, influence propagation, shortest path) to Neptune Analytics rather than running them on the transactional Neptune DB cluster.
- **Vector index tuning**: In OpenSearch, configure HNSW parameters (`ef_construction`, `m`) based on the tradeoff between recall accuracy and query latency for the specific embedding dimensions used.
- **Consider S3 Vectors** for cold/archival vector storage tiers where query latency tolerance is >500ms but cost optimization is paramount (up to 90% savings).

5. Cost Optimization

Design Principles Applied:

- **Adopt a consumption model**: Neptune Serverless scales down to minimum configured NCUs (as low as 1.0 NCU) during idle periods; OpenSearch Serverless scales based on actual query volume.
- **Measure overall efficiency**: Monitor cost-per-query metrics across the Neptune and OpenSearch tiers to right-size capacity.
- **Stop spending money on undifferentiated heavy lifting**: Using managed services (Neptune, OpenSearch, Bedrock) eliminates operational overhead of running open-source graph databases.

Cost Optimization Recommendations:

- **Right-size Neptune**: For development/staging, use Neptune Serverless with minimum NCUs; for production steady-state, evaluate provisioned instances (which can be 30-40% cheaper with Reserved Instances or Database Savings Plans).

- **Database Savings Plans**: Neptune Analytics and OpenSearch Service are both eligible for Database Savings Plans (up to 35% savings for serverless deployments, up to 20% for provisioned instances, with 1-year commitment).
- **Tiered vector storage**: Store the full embedding corpus in **S3 Vectors** (low cost) and maintain a hot subset in **OpenSearch** (low latency) — use one-click export from S3 Vectors to OpenSearch for seamless tiering.
- **Neptune storage optimization**: Use `DROP GRAPH` or named graph isolation to manage ontology versions; archive deprecated versions to S3 as RDF exports rather than keeping them in live storage.
- **Query cost tracking**: Use Neptune's `Query/Requests` and `Query/Time` CloudWatch metrics to identify and optimize expensive queries that drive compute costs.

6. Sustainability

Design Principles Applied:

- **Understand your impact**: Quantify the compute and storage footprint of the semantic layer components across Neptune, OpenSearch, and supporting services.
- **Maximize utilization**: Neptune Serverless and OpenSearch Serverless automatically scale down during low-demand periods, reducing energy consumption.
- **Use managed services**: AWS managed services run on shared, optimized infrastructure with better energy efficiency than self-managed deployments.

Sustainability Recommendations:

- **Choose Graviton-based instances** (db.r8g Graviton4, db.r7g Graviton3, db.r6g Graviton2) for Neptune provisioned deployments — AWS Graviton-based instances use up to 60% less energy than comparable x86 instances (general Graviton claim across generations). R8g (Graviton4) is the latest generation available and offers significant price-performance improvements (up to 4.7x better write, 3.7x better read throughput over prior generations).
- **Consolidate ontology namespaces**: Reduce redundant triples and ontology fragments to minimize storage and query processing overhead.
- **Batch embedding operations**: When updating the vector store, batch entity embedding generation through Amazon Bedrock rather than making individual API calls — reduces compute cycles and network overhead.

- **Data lifecycle policies**: Implement TTL or archival rules for temporal graph data (event entities, transient relationships) to prevent unbounded storage growth.

Related resources

This AWS Prescriptive Guidance leverages **W3C Semantic Web standards** — including RDF, RDFS, OWL, SPARQL, SHACL, R2RML, and SWRL — to ensure interoperability, longevity, and ecosystem compatibility. These open standards enable organizations to build semantic data foundations that integrate seamlessly with best-of-breed tools and platforms, avoiding vendor lock-in while benefiting from decades of research and production deployments across industries.

AWS Partner solutions complement, implement portions of, or deliver the reference architecture in its entirety, accelerating time-to-value and providing specialized capabilities tailored to specific use cases. Partners bring deep domain expertise in semantic technologies, ontology engineering, knowledge graph platforms, and agentic AI — enabling customers to leverage proven implementations, pre-built ontologies, and managed services that reduce complexity and operational overhead. Whether you're seeking a fully managed knowledge graph platform, specialized reasoning engines, ontology development tools, or end-to-end semantic layer implementations, AWS Partners offer flexible deployment options that align with your organization's maturity, scale, and strategic objectives. We encourage customers to explore partner solutions that best fit their requirements, and we remain committed to supporting a vibrant ecosystem that drives innovation in semantic AI.

References

- **AWS Workshop**
  - Ontology as a Semantic Layer for Agentic AI Applications (Access to artifacts to self-host)
- **W3C Semantic Web Standards:**
  - **RDF (Resource Description Framework)**
  - **RDFS (RDF Schema)**
  - **OWL (Web Ontology Language)**
    - OWL 2 Document Overview
    - OWL 2 Profiles
  - **SPARQL**
  - **SHACL (Shapes Constraint Language)**
    - SHACL 1.2 Core
    - SHACL JavaScript Extensions

  - SHACL 1.2 Rules
- **R2RML (RDB to RDF Mapping Language)**
  - R2RML Schema
- **Research Papers:**
  - Allemang, D. & Sequeda, J. (2024). *Increasing the LLM Accuracy for Question Answering: Ontologies to the Rescue!* arXiv:2405.11706. https://arxiv.org/abs/2405.11706

AWS Marketplace links

- **Stardog Enterprise Knowledge Graph Platform**
- **PuppyGraph Professional**

Contributors

Taylor Riggan, AWS Principal Graph Architect, Amazon Neptune

Chun Schiros, AWS Enterprise Technologist

Don Simpson, AWS Principal Technologist

Document history

The following table describes significant changes to this guide.

Change	Description	Date
Initial publication	—	July 24, 2026