# The Data Observability Signal Factory: A Comprehensive Architectural Specification and Implementation Report

## 1. Executive Strategic Assessment and Contextual Imperative

The contemporary enterprise data landscape has undergone a radical fragmentation. The transition from monolithic warehouses to decentralized data meshes, microservices, and event-driven architectures has exponentially increased the complexity of data delivery. In this hyper-distributed environment, data no longer resides in static silos but flows as a continuous stream across heterogeneous boundaries—from RESTful microservices to Kafka streams, through batch Spark transformations, and into Delta Lakes. This architectural shift has rendered traditional monitoring approaches obsolete. The passive observation of system outputs, characterized by static dashboards and threshold-based alerts, fails to capture the semantic nuance of data reliability.1 It generates noise rather than signal, leading to "dashboard fatigue" where engineering teams are inundated with alerts that lack context, causality, or actionable resolution paths.1

To address this "Observation Gap," this report defines the architecture for a **Data Observability Signal Factory**. This concept represents a fundamental paradigm shift: treating telemetry not as an exhaust product of running systems, but as a first-class manufactured good. Just as a physical factory applies rigorous quality controls, standardization, and process engineering to its outputs, the Signal Factory industrializes the production of observability signals.1 It moves the organization from ad-hoc, artisanal logging—where every developer defines their own format—to a standardized, canonical telemetry stream that powers sophisticated AI-driven reasoning.

The strategic imperative for this architecture is grounded in a hierarchy of needs that prioritizes **Return on Investment (ROI)** and **Mean Time to Resolution (MTTR)** over generic coverage.1 It acknowledges that 100% coverage of low-value assets is economically inefficient if critical incidents on Tier-1 pipelines take hours to diagnose. By implementing a centralized "Signal Factory," the organization creates a control plane capable of enforcing contracts, injecting correlation identifiers, and validating schemas *before* data enters the complex downstream ecosystem. This "choke point" strategy, specifically realized through a **Central Managed Streaming Ingestion Gateway** (Option D), effectively solves the fragmentation problem at its source, ensuring that the "garbage in, garbage out" cycle is broken.1

This comprehensive report serves as the definitive technical blueprint for this ecosystem. It details the architectural topology, the Neptune-based graph data model, the canonical event schemas, the operational process flows, and the capabilities required to build each component. It bridges the gap between high-level strategy and low-level implementation, providing the specific engineering directives necessary to construct a self-healing, trustworthy data platform.

## 2. Architectural Paradigm: The Signal Factory

### 2.1 The Core Philosophy: Active Production vs. Passive Observation

The central thesis of the Signal Factory is that high-quality observability data does not occur naturally; it must be engineered. In traditional architectures, telemetry is often an afterthought—added by developers during debugging sessions or tacked on to meet a compliance checklist. This results in a "Telemetry Sprawl" where order\_id in one system is ord\_id in another, timestamps lack timezone context, and tracing contexts are dropped at asynchronous boundaries like Kafka consumers or Airflow sensors.1

The Signal Factory reverses this dynamic. It treats the generation of telemetry as a managed process. It employs an **Instrumentation Autopilot**—a combination of static analysis agents and Pull Request (PR) bots—to mechanically inject standardized instrumentation into codebases.1 This ensures that every service, job, and pipeline emits telemetry that conforms to a strict, globally governed schema. This "manufactured" telemetry is then processed by a **Signal Factory Core**, running on AWS, which normalizes, correlates, and enriches these signals before storing them in a Knowledge Graph (Amazon Neptune). This graph becomes the semantic "brain" of the system, enabling an AI Copilot to reason about incidents with a complete understanding of the system's topology and state.1

### 2.2 The "Push vs. Pull" Adoption Dynamic

A critical failure mode in observability initiatives is the reliance on administrative mandates to force adoption. This "Push" strategy often encounters resistance from product teams pressured to deliver features. The Signal Factory architecture is designed to leverage a "Pull" dynamic initially, transitioning to "Push" only as the system matures.1

**The Value Engine (Pull):** The architecture deploys an AI RCA (Root Cause Analysis) Copilot as the primary interface for users. By accurately diagnosing incidents—e.g., "The dashboard is stale because Schema v42 introduced a breaking change at 10:00 AM"—the system provides immediate, tangible value to on-call engineers. This utility creates a natural "Pull" for adoption: teams *want* to be onboarded because it reduces their operational burden.1

**The Signal Factory (Push):** Once value is established, the organization introduces "Progressive Gates." These are automated checks in the CI/CD pipeline and the Ingestion Gateway that gradually enforce stricter observability standards. What starts as a warning ("Missing Owner Tag") evolves into a blocker ("Schema Violation Rejected"), converting observability from an optional best practice into a mandatory platform capability.1

### 2.3 Architectural Layers

The Signal Factory is composed of five distinct planes, each responsible for a specific stage in the signal lifecycle.

| **Plane** | **Function** | **Key Components** |
| --- | --- | --- |
| **Production Plane** | Generates raw telemetry and data events. | Microservices, Ingestion Gateway, Kafka, Spark/Delta, Airflow. |
| **Ingestion Plane** | Collects, buffers, and routes signals. | OTel Collector, Signal Router, Event Bus (Kinesis/MSK). |
| **Processing Plane** | transform raw signals into state and insights. | Signal Engines (Freshness, Drift, Contract, DQ, Cost). |
| **Knowledge Plane** | Stores topology, state, and lineage. | Amazon Neptune (Graph), DynamoDB (State), S3 (Archive). |
| **Consumption Plane** | Interfaces for humans and agents. | RCA Copilot API, Governance Registry, Alerting. |

## 3. Core Component: The Central Managed Streaming Ingestion Gateway

The pivot to a **Central Managed Streaming Ingestion Gateway** (Option D) is the architectural linchpin that enables high-fidelity observability in a microservices environment. In previous architectural iterations, individual microservices were responsible for configuring Kafka producers, managing schema serialization, and handling connectivity. This "fat client" model led to configuration drift, inconsistent tracing, and "shadow" producers that bypassed governance.1

### 3.1 Architectural Role and Deployment Logic

The Gateway is deployed on **Amazon EKS** as a scalable, stateless service behind an Application Load Balancer (ALB). It exposes a RESTful API (POST /ingest/{endpoint\_id}) that internal services use to publish data. By mediating all writes to the Kafka backbone, the Gateway acts as a "smart pipe" that enforces governance at the edge. It is the single source of truth for correlation injection, ensuring that the "Hybrid Correlation Model" is correctly initialized for every data packet entering the system.1

**Deployment Specifications:**

* **Compute Architecture:** The Gateway runs on EKS Managed Node Groups. It utilizes Horizontal Pod Autoscaling (HPA) triggered by custom metrics: Request Per Second (RPS) and CPU saturation. This ensures elastic scalability to handle bursty ingest loads without degradation.
* **Network Topology:** It resides in private VPC subnets. Connectivity to MSK (Managed Streaming for Kafka) and Neptune is established via VPC Endpoints (PrivateLink) to minimize latency, enhance security, and eliminate data transfer costs associated with NAT gateways.4
* **High-Speed Caching:** To meet the requirement of adding single-digit millisecond latency overhead, the Gateway employs a multi-layer cache strategy. L1 is in-memory (local to the pod), storing hot schema definitions and contracts. L2 is a centralized Redis (ElastiCache) or DynamoDB DAX cluster, ensuring consistency across pods.1

### 3.2 Internal Processing Capabilities

The Gateway is not a passive proxy; it is an active enforcement engine. Its internal processing pipeline executes the following logic for every request:

1. **Authentication & Quota Management:**
   * **Capability:** Verifies caller identity via IAM roles or mTLS.
   * **Mechanism:** Checks strict rate limits (token bucket algorithm) per tenant to prevent "noisy neighbor" issues in the multi-tenant Kafka cluster.
2. **Context Resolution:**
   * **Capability:** Maps the REST resource endpoint\_id to the associated Data Contract, active Schema Version, and target Kafka Topic.
   * **Mechanism:** fast lookups against the cached Registry data.
3. **Correlation Injection (The Hybrid Model):**
   * **Capability:** Initializes the trace context if missing.
   * **Mechanism:** Checks for an incoming traceparent header. If absent, it initiates a new trace. It generates a deterministic ingestion\_request\_id (for idempotency) and injects a suite of x-obs-\* headers into the Kafka message record, creating the "Layer 1" link.1
4. **Schema & Contract Validation:**
   * **Capability:** Ensures payload integrity before persistence.
   * **Mechanism:** Validates the JSON/Avro payload against the resolved Schema. It enforces the "Schema Drift Policy"—for example, allowing forward-compatible changes (adding optional fields) but rejecting breaking ones (removing required fields) based on the Contract definition.1
5. **Publishing & Partitioning:**
   * **Capability:** Writes to the durable log.
   * **Mechanism:** Only if validation passes is the message produced to the underlying MSK topic. The Gateway handles partitioning logic based on the contract's defined partition keys, ensuring semantic ordering.

### 3.3 Idempotency and Deduplication Strategy

To ensure data integrity in a distributed system, the Gateway implements strict idempotency using a **DynamoDB** table. This is critical for reliable "exactly-once" processing semantics from the producer's perspective.1

The composite key (producer\_service\_urn, endpoint\_id, idempotency\_key) allows the Gateway to recognize retried requests:

* **Scenario A (Success):** Request arrives with key K1. Gateway processes, publishes to Kafka, writes K1 to DynamoDB with TTL, and returns 202.
* **Scenario B (Retry):** Network timeout occurs, client retries with K1. Gateway checks DynamoDB, sees K1 was processed, and returns the cached 202 response *without* duplicating the Kafka message.
* **Scenario C (Conflict):** Client sends a *new* payload with reused key K1. Gateway detects a hash mismatch between the new payload and the stored record, returning 409 Conflict.

This mechanism moves the complexity of idempotency out of individual microservices and into the shared infrastructure layer, significantly simplifying the developer experience.

## 4. The Hybrid Correlation Model

A central critique of many observability implementations is the naive assumption that a single trace\_id can successfully propagate across all system boundaries.1 In reality, modern data stacks are riddled with "Semantic Breaks" where runtime context is lost.

### 4.1 The Anatomy of Semantic Breaks

* **The Fan-Out Break (HTTP → Kafka):** A single HTTP request may trigger multiple Kafka messages. A 1:1 trace relationship is insufficient; a 1:N parent-child relationship is required.
* **The Aggregation Break (Kafka Consumer Batching):** Consumers often process messages in batches (e.g., 100 at a time) and produce a single aggregated output. The output cannot inherit 100 different trace IDs without bloating headers.1
* **The Orchestration Break (Airflow):** Airflow DAGs often run on schedules, processing all data available since the last run. They are not triggered by a specific upstream trace, severing the link to the producer.1
* **The State Boundary Break (Spark → Delta):** Spark jobs write to storage (Delta tables). Downstream jobs read from storage. There is no direct runtime link between the writer and reader; the table acts as a firewall for traces.1

### 4.2 The Hybrid Solution Layers

To mitigate these breaks, the Signal Factory employs a **Hybrid Correlation Model** that layers three distinct linking strategies.1

#### Layer 1: Runtime Correlation (Synchronous & Direct Async)

* **Scope:** Microservice → Gateway → Kafka → Landing Service.
* **Mechanism:** Standard W3C Trace Context (traceparent). The Gateway acts as the trusted injector.
* **Implementation:** The Gateway accepts or creates a trace ID and stamps it into the Kafka headers. The Delta Landing Service extracts this ID and logs it alongside the commit metadata.

#### Layer 2: Batch Handoff (Explicit Linking)

* **Scope:** Airflow → Spark.
* **Mechanism:** Explicit parameter passing.
* **Implementation:** When Airflow triggers a Spark job, it passes the dag\_run\_id and task\_instance\_id as configuration parameters to the Spark application. The Spark OpenLineage listener captures these parameters and includes them in the RunEvent, effectively linking the orchestration layer to the compute layer.1

#### Layer 3: Lineage-Based Correlation (State Boundaries)

* **Scope:** Spark Write → Delta Table → Spark Read.
* **Mechanism:** Time-window and asset version heuristics backed by the Graph.
* **Implementation:** Since a trace cannot persist through a table at rest, the RCA engine uses the Knowledge Graph. It queries: "Which Job Run wrote to Table Y (Commit Version N)?" and "Which Job Run read from Table Y (Commit Version N)?" By linking the *writer* and the *reader* through the *asset version*, the system infers causality even without a continuous trace ID.1

## 5. The Knowledge Plane: Amazon Neptune Graph Data Model

The efficacy of the RCA Copilot depends entirely on the fidelity of the underlying data model. The Signal Factory uses **Amazon Neptune** to model the data estate as a directed Labeled Property Graph (LPG). This graph unifies three disparate domains: the physical infrastructure (Topology), the logical governance (Contracts), and the runtime execution (Observability).1

### 5.1 Vertex Definitions (Nodes)

The schema defines the following primary vertices. Each vertex type corresponds to a critical entity in the data ecosystem.

| **Vertex Label** | **Description** | **Required Properties (Keys)** |
| --- | --- | --- |
| **Service** | Logical application/microservice. | urn, name, domain (e.g., "checkout"), tier (Tier-1/2/3), owner\_team. |
| **Endpoint** | Gateway ingestion resource (API). | urn, endpoint\_id, env, auth\_mode, tier, status. |
| **KafkaTopic** | MSK Topic. | urn, cluster, topic, env, tier. |
| **DeltaTable** | Data Lake Table. | urn, layer (bronze/silver), physical\_location (S3 path), owner, tier. |
| **Column** | Schema field (for granular lineage). | urn, name, data\_type, parent\_asset\_urn. |
| **Contract** | Governance document. | id, asset\_urn, policy\_hash, status, sla\_freshness. |
| **SchemaVersion** | Immutable data structure def. | schema\_id, version, fingerprint, compatibility. |
| **Run** | Execution unit (Job/DAG). | urn, system (airflow/spark), run\_id, status, start\_ts. |
| **Incident** | Detected issue. | id, severity, status, detected\_ts, primary\_signal. |
| **Evidence** | Fact supporting RCA. | id, kind, confidence, summary, pointer (URI). |
| **Deployment** | Code change event. | urn, service\_urn, build\_id, deploy\_id, ts. |

### 5.2 Edge Definitions (Relationships)

The edges define the semantic relationships that allow the RCA engine to traverse from symptom to root cause.

* **Topology Edges:**
  + CALLS: Service → Endpoint
  + PUBLISHES\_TO: Endpoint → KafkaTopic
  + LANDS\_IN: KafkaTopic → DeltaTable (representing the landing job)
  + DERIVES\_FROM: DeltaTable → DeltaTable (downstream transformations)
* **Granular Lineage:**
  + HAS\_COLUMN: Table → Column
  + COLUMN\_DERIVES\_FROM: Column → Column (capturing transformation logic like SELECT a AS b) 1
* **Governance Edges:**
  + GOVERNED\_BY: Asset → Contract
  + USES\_SCHEMA: Endpoint/Topic → SchemaVersion
  + COMPATIBLE\_WITH: SchemaVersion → SchemaVersion (tracks evolution history)
* **Incident Edges:**
  + AFFECTS: Incident → Asset (Blast Radius)
  + TRIGGERED\_BY: Incident → SignalState
  + SUPPORTED\_BY: Incident → Evidence

### 5.3 Graph Traversal Logic

The graph structure is optimized for specific recursive queries used by the Copilot.

Blast Radius Calculation:

Starting from a compromised KafkaTopic, the query traverses out(LANDS\_IN, DERIVES\_FROM) recursively to identify all downstream DeltaTables and Dashboards. By filtering on the tier property, the engine can immediately assess business severity (e.g., "3 Tier-1 dashboards impacted").1

Root Cause Back-Tracing:

Starting from a stale DeltaTable, the query traverses in(DERIVES\_FROM, LANDS\_IN) to find upstream sources. It then cross-references Run nodes attached to those sources via PRODUCED\_BY edges to find failed jobs, or SchemaVersion nodes to find recent incompatible changes.1

## 6. The Processing Core: Signal Engines

The "Factory Core" is the processing engine that transforms raw telemetry into the structured data stored in the graph. It consists of specialized engines, each dedicated to a specific class of data problem.

### 6.1 Freshness Engine

* **Capabilities:** Tracks data arrival times against defined SLOs.
* **Flow:** Consumes LandingOutcomeEvent and RunEvent. Updates the freshness property on DeltaTable nodes in Neptune. If the time since the last update exceeds the sla\_freshness defined in the linked Contract node, it generates a FreshnessBreach signal.

### 6.2 Schema Drift Engine

* **Capabilities:** Detects unauthorized or incompatible schema changes.
* **Flow:** Consumes IngestionDecisionEvent. It compares the incoming schema\_fingerprint against the SchemaVersion node currently active for the asset. If they differ, it checks the Contract policy. If the change is backward-incompatible (e.g., field deletion), it generates a SchemaIncompatibility signal and potentially triggers a Gateway rejection (if G3 gating is active).

### 6.3 Volume Engine

* **Capabilities:** Identifies anomalous data throughput.
* **Flow:** Aggregates row counts from LandingOutcomeEvent and RunEvent. It uses a sliding window algorithm (e.g., T-Digest or simple moving average) to establish a baseline for the specific time of day/week. Deviations beyond a threshold (e.g., -50% drop) trigger a VolumeAnomaly signal.

### 6.4 Data Quality (DQ) Engine

* **Capabilities:** Detects content-level anomalies.
* **Flow:** Ingests DQResultEvent (generated by Deequ/Great Expectations). It maps these results to the DeltaTable node. It tracks trends in pass/fail rates. A spike in failures for a specific constraint (e.g., NULL check on order\_id) generates a DQFailure signal.1

### 6.5 Contract Engine

* **Capabilities:** The governance enforcement layer.
* **Flow:** It acts as the meta-evaluator. It subscribes to signals from the other engines. If a FreshnessBreach occurs on an asset with a strict Contract, the Contract Engine escalates this to an Incident and notifies the owning team, enforcing the accountability model defined in the organization's governance framework.1

## 7. Canonical Event Specifications

To operationalize the "Signal Factory," the telemetry emitted must be standardized. We define a set of **Canonical JSON Schemas** that all components (Gateway, Airflow, Spark) must adhere to. This decoupling ensures that the central processing core is agnostic to the upstream implementation details. These schemas are compliant with the **CloudEvents** specification to ensure interoperability.5

### 7.1 SignalEvent (Base Envelope)

Used for all events to ensure consistent routing, metadata, and correlation.

JSON

{  
 "$id": "https://company.internal/schemas/signal-event.json",  
 "specversion": "1.0",  
 "type": "com.company.observability.SignalEvent",  
 "source": "urn:svc:prod:ingestion-gateway",  
 "id": "A234-1234-1234",  
 "time": "2023-10-27T10:00:00Z",  
 "datacontenttype": "application/json",  
 "data": {  
 "env": "prod",  
 "asset": {  
 "urn": "urn:kafka:prod:us-east-1:orders\_topic",  
 "type": "Topic"  
 },  
 "correlation": {  
 "trace": { "trace\_id": "hex", "span\_id": "hex" },  
 "handoff": {  
 "kafka": { "topic": "orders", "offset": 100 },  
 "delta": { "commit": 99 }  
 }  
 },  
 "producer": { "service\_urn": "urn:svc:order-service", "deploy\_id": "git:sha123" }  
 }  
}

### 7.2 IngestionDecisionEvent (Gateway)

Critical for detecting producer-side issues (Schema drift, Bad contracts).

JSON

{  
 "type": "com.company.observability.IngestionDecisionEvent",  
 "source": "urn:svc:prod:ingestion-gateway",  
 "data": {  
 "endpoint\_id": "orders\_v1",  
 "ingestion\_request\_id": "uuid",  
 "decision": {  
 "status": "REJECTED",  
 "reason\_code": "SCHEMA\_INCOMPATIBLE",  
 "details": "Field 'total\_amt' type mismatch: expected double, got string"  
 },  
 "schema\_context": { "id": "s1", "version": 1 },  
 "latency\_ms": 15  
 }  
}

### 7.3 LandingOutcomeEvent (Consumer)

Bridges the gap between the stream and the lake, implementing Layer 3 of the Correlation Model.

JSON

{  
 "type": "com.company.observability.LandingOutcomeEvent",  
 "source": "urn:svc:prod:delta-landing-service",  
 "data": {  
 "target": { "delta\_table\_urn": "urn:delta:prod:orders\_bronze", "layer": "bronze" },  
 "outcome": {  
 "status": "SUCCEEDED",  
 "rows\_written": 1000,  
 "bytes\_written": 50000,  
 "delta\_commit\_version": 120  
 },  
 "partition\_info": { "date": "2024-01-01" },  
 "trace\_ids\_processed": // Sampling of traces in this batch  
 }  
}

## 8. The Root Cause Analysis (RCA) Copilot

The RCA Copilot is the user-facing intelligence of the platform. Unlike generic LLM chatbots, it operates on a strict **Evidence-First** architecture. It never speculates; it only synthesizes facts present in the Knowledge Graph.1

### 8.1 Architecture: The RAG Context Builder

The Copilot does not query raw logs directly, as this is prohibitively slow and expensive. Instead, it utilizes a **Retrieval-Augmented Generation (RAG)** pattern backed by the Neptune Graph and the DynamoDB IncidentContextCache.

**The Context Builder Flow:**

1. **Scope Definition:** Identifies the asset in distress (e.g., Orders\_Silver) and the relevant time window ($T\_{detect}$ back to $T\_{baseline}$).
2. **Graph Retrieval:** Queries Neptune to retrieve the topological subgraph: upstream dependencies (Lineage), downstream impact (Blast Radius), and related entities (Deployments, Contracts).
3. **Signal Aggregation:** Fetches the state of all nodes in the subgraph (e.g., "Schema Version changed at T-10m", "Gateway Rejections spiked at T-5m").
4. **Prompt Construction:** Serializes this structured graph data into a coherent textual prompt for the LLM (e.g., Claude).
5. **Synthesis:** The LLM generates a natural language explanation, citing the specific Evidence IDs provided in the prompt.

### 8.2 Blast Radius Computation

The Copilot enables proactive incident management by calculating the "Blast Radius"—the full set of downstream assets impacted by an upstream failure.

* **Mechanism:** It executes a recursive graph traversal starting from the failing node, following LANDS\_IN and DERIVES\_FROM edges.
* **Output:** A list of impacted Dashboards, ML Models, and Executive Reports, grouped by Tier. This allows incident commanders to prioritize communication (e.g., "Notify the Finance team immediately because the Board Report dashboard is compromised").1

### 8.3 Evidence-First Reasoning

To mitigate hallucination, the Copilot's system prompt enforces a strict rule: "Every claim must be supported by an Evidence ID."

* **Bad Output:** "It looks like the schema changed."
* **Good Output:** "The schema changed to Version 42 at 10:00 UTC, which correlates with a 500% spike in Gateway rejections."

This constraint ensures that the AI acts as a sophisticated interface to deterministic data, rather than a creative writer.1

## 9. The Instrumentation Autopilot

To populate the Signal Factory without imposing an unmanageable burden on developers, the architecture employs an **Instrumentation Autopilot**. This component automates the adoption process, converting "manual toil" into "managed code".1

### 9.1 The Repo Scout and PR Bot

The Autopilot consists of a "Repo Scout" agent that scans repositories to identify their archetype (e.g., "Java Microservice," "Python Airflow DAG"). Based on the archetype, it determines the missing instrumentation.

**Capabilities:**

* **Dependency Injection:** Adds the OpenTelemetry SDK, OpenLineage libs, or the internal Gateway Client SDK to the build files (pom.xml, requirements.txt).
* **Code Instrumentation:** Injects standard boilerplate code. For a microservice, it adds the request interceptor for trace propagation. For Airflow, it adds the lineage extraction listeners.
* **Configuration:** specific environment variables (e.g., OTEL\_EXPORTER\_OTLP\_ENDPOINT) and standard tags (service.name, owner.team).

### 9.2 The PR Lifecycle

1. **Scan:** The Scout identifies a repo lacking x-obs-\* header propagation.
2. **Generate:** It generates a Pull Request (PR) with the necessary code changes.
3. **Validate:** The PR includes a "Telemetry Validator" CI step that spins up the service in a sandbox, sends a synthetic request, and verifies that the expected signals arrive at the OTel Collector.
4. **Merge:** The service owner reviews and merges the PR. The system now emits standardized telemetry without the developer writing a single line of observability code.1

## 10. Operational Process Flows: Timelines and Narratives

To visualize the system in action, we detail two distinct process flows: the "Happy Path" of normal data production and the "Incident Path" of failure diagnosis.

### 10.1 Timeline: The Life of a Data Packet (Happy Path)

This narrative follows a single data packet from a microservice to a dashboard.

* **T+00ms (Production):** The OrderService handles a checkout. It calls the internal Gateway endpoint POST /ingest/orders\_v1. The SDK attaches traceparent: T1.
* **T+05ms (Gateway Processing):** The Gateway receives the request. It resolves the contract, validates the schema (v42), generates an ingestion\_request\_id, and injects x-obs-\* headers. It produces the message to Kafka topic orders\_topic and emits an IngestionDecisionEvent (ACCEPTED).
* **T+15ms (Transport):** Kafka persists the message.
* **T+50ms (Landing):** The DeltaLandingService consumes the message. It extracts T1 and writes the data to the orders\_bronze table (Commit 105). It emits a LandingOutcomeEvent linking T1 to Commit 105.
* **T+100ms (Graph Update):** The Signal Router receives the events. The Signal Engine updates the Neptune graph, creating a Run node for the landing job and linking it to the SchemaVersion and DeltaTable nodes.
* **T+5min (Transformation):** An Airflow DAG runs. It triggers a Spark job reading orders\_bronze (Commit 105). OpenLineage captures this read and the subsequent write to orders\_silver.
* **T+6min (Completion):** The data is available in the Silver table. The graph now reflects the full lineage from the HTTP request T1 to the orders\_silver table.

### 10.2 Timeline: The Anatomy of an Incident (Failure Path)

This narrative describes how the system handles a schema violation.

* **T+00ms (The Change):** A developer deploys a new version of OrderService that changes the total\_amt field from double to string. This violates the Data Contract.
* **T+05ms (The Block):** The service calls the Gateway. The Gateway validates the payload against the Schema. It detects the type mismatch.
* **T+06ms (The Rejection):** The Gateway rejects the request with 400 Bad Request and reason\_code: SCHEMA\_INCOMPATIBLE. It emits an IngestionDecisionEvent (REJECTED).
* **T+10ms (Signal Processing):** The Signal Router processes the rejection event. The Schema Drift Engine creates an Evidence node in Neptune: Kind: SchemaRejection, Source: OrderService.
* **T+1m (The Alert):** The Volume Engine detects a drop in successful writes to orders\_bronze (since requests are being rejected). It triggers an Incident.
* **T+2m (The Diagnosis):** The RCA Copilot is triggered. It queries the graph for the Orders\_Topic. It sees the VolumeAnomaly linked to the topic. It traverses upstream to the Gateway Endpoint. It finds the spike in SchemaRejection evidence.
* **T+2.1m (The Output):** The Copilot alerts the on-call engineer: "Incident: Volume drop on Orders. Root Cause: Schema Incompatibility. Evidence: Gateway is rejecting 100% of requests from OrderService due to field total\_amt type mismatch."
* **T+5m (Resolution):** The engineer rolls back the OrderService deployment. The incident is resolved.

## 11. Organizational Implementation & Governance

The technical architecture must be supported by a robust organizational framework.

### 11.1 Progressive Gating Strategy

To manage the cultural shift, we employ a "Progressive Gating" strategy that gradually ramps up enforcement.1

| **Gate Level** | **Description** | **Action** | **Timeline** |
| --- | --- | --- | --- |
| **Gate 0 (Visibility)** | Telemetry is collected. Dashboarding only. | No blocking. | Weeks 1-4 |
| **Gate 1 (Warn)** | CI checks for contracts/URNs. | CI Warnings. | Weeks 5-8 |
| **Gate 2 (Soft-Fail)** | Validation of Tier-1 assets. | Rejects in Staging; Alerts in Prod. | Weeks 9-12 |
| **Gate 3 (Hard-Fail)** | Full enforcement. | Rejects in Prod. Blocks Deploys. | Week 13+ |

### 11.2 Change Management

The implementation includes a specific workstream for **Change Enablement**.1

* **Champions:** Recruit "Observability Champions" in key domain teams to pilot the Autopilot and provide feedback.
* **Certification:** Create a "Certified Data Publisher" badge for teams that reach Gate 3, offering them "Fast Lane" deployments (bypassing manual CAB reviews) as an incentive.
* **Visibility:** Display "Observability Readiness" scores on team dashboards, gamifying the adoption process.

### 11.3 ROI Realization

The program tracks ROI via distinct metrics 1:

* **Data Downtime Avoided:** Calculated by comparing the duration of incidents caught by the Gateway (pre-consumption) vs. historical incidents caught by downstream users.
* **MTTR Reduction:** Tracking the time from Alert to Resolution for Copilot-assisted incidents vs. manual debugging.
* **Engineering Hours Saved:** Estimated hours saved by the Autopilot's automated instrumentation vs. manual coding.

## 12. Implementation Roadmap

The rollout is structured into three phases to de-risk the deployment and demonstrate early value.1

### Phase 0: Foundations & Steel Thread (Weeks 1-4)

* **Goal:** Deploy the core platform and prove end-to-end correlation on a single pipeline.
* **Deliverables:**
  + Deploy EKS Ingestion Gateway (MVP).
  + Provision Neptune and apply Schema V1.
  + Migrate one "Steel Thread" pipeline (e.g., "Orders") to the Gateway.
* **Exit Criteria:** Successful end-to-end trace from Service to Delta Table visible in the Graph.

### Phase 1: Intelligence & The Copilot (Weeks 5-8)

* **Goal:** Enable the RCA engine and generate the first AI-driven insights.
* **Deliverables:**
  + Implement Signal Engines (Freshness, Drift).
  + Deploy RCA Context Builder (RAG).
  + Launch Copilot for the Steel Thread team.
* **Exit Criteria:** Copilot correctly diagnoses a simulated schema incident in < 2 minutes.

### Phase 2: Governance & Scale (Weeks 9-12)

* **Goal:** Enforce standards and expand to Tier-1 assets.
* **Deliverables:**
  + Activate "Gate 2" (Soft-Fail) for Tier-1 schema violations.
  + Onboard 5 additional Tier-1 services.
  + Launch the Instrumentation Autopilot for general use.
* **Exit Criteria:** 100% of Tier-1 ingestion via Gateway; Zero unmanaged schema changes.

## 13. Conclusion

The Data Observability Signal Factory represents the future of enterprise data reliability. By combining the rigid control of a **Central Ingestion Gateway** with the flexible intelligence of a **Neptune Knowledge Graph**, the architecture solves the twin problems of data fragmentation and monitoring noise. It transforms observability from a passive cost center into an active asset that creates trust, accelerates resolution, and enables the safe scaling of the data estate. This report provides the comprehensive blueprint required to build this capability, positioning the organization to lead in the era of data-driven decision-making.

#### Works cited

1. Data Observability Spec Review.docx
2. Data Observability: A Complete Guide to the Modern Data Stack - Blog de Bismart, accessed January 4, 2026, <https://blog.bismart.com/en/data-observability>
3. Managed Graph Database – Amazon Neptune - AWS, accessed January 4, 2026, <https://aws.amazon.com/neptune/>
4. G.V() graph database client - Neptune Analytics - AWS Documentation, accessed January 4, 2026, <https://docs.aws.amazon.com/neptune-analytics/latest/userguide/gv-tool.html>
5. CloudEventV1 - CloudEvents, accessed January 4, 2026, <https://cloudevents.github.io/sdk-javascript/interfaces/CloudEventV1.html>
6. CloudEvents |, accessed January 4, 2026, <https://cloudevents.io/>
7. CloudEvents - EventSourcingDB, accessed January 4, 2026, <https://docs.eventsourcingdb.io/fundamentals/cloud-events/>