Observations API

This document is stored in a document management system (DMS).

Department:

DMS document owner:

DMS document approver:

Applicable location: **Error! Unknown document property name.**

Summary

NOTE: *The following API documentation is a draft. It was created prior to my departure from the company. I am including it as a writing sample to show a draft document created collaboratively with the developer. The graphics were intended to be replaced, a final review was scheduled to occur, and it would have been released to the internal engineering community.*

Applicable Documents

- Not applicable

Table of Contents

1. Observations API

The Observations API is [redacted] unified platform for capturing, structuring, and distributing timestamped events—called *observations*—generated by vehicles, simulation environments, or [redacted] systems. An observation represents a moment of truth, such as a sensor reading, decision point, or test outcome. By centralizing the ingestion, validation, and dissemination of these events, the API frees developers from building fragmented logging pipelines and instead delivers a governed, searchable, and reproducible data foundation.

Developers define observation types using declarative JSON schemas; produce data through a simple REST API interface; and trust the system to handle validation, versioning, search indexing, and long-term storage. Whether iterating on a new perception metric in simulation or streaming real-time diagnostics from a fleet vehicle, the API ensures consistency across modalities, enforces producer ownership, and makes data immediately discoverable, both in real time via OpenSearch and persistently in the datalake.

1.1 How it Works

Refer to the Observations API workflow for a visual depiction of how data flows using the API.

The Observations API operates as a multi-region system built on the AWS infrastructure, with API Gateway endpoints in both us-east-1 and us-west-2 backed by globally replicated DynamoDB tables. Requests are load-balanced across regions using health-checked routing, ensuring availability even during regional degradation. While client-facing writes can land in either region, all downstream processing—search indexing, datalake streaming, and cross-account delivery—converges in us-east-1, where OpenSearch and core event pipelines reside.

An observation begins with schema definition and flows through validation, storage, event-driven enrichment, parallel distribution, and finally, searchable persistence. This architecture supports high-throughput ingestion (hundreds of observations per minute per producer) while maintaining strict consistency between the API and downstream consumers.

1.1.1 Defining an Observation Type

Before producing data, a developer must register an *observation_type*. Refer to the article “Easy Way to Create Observation Types” [link removed] for details. This is accomplished by sending a POST request to */observation-types* with a JSON payload containing a Pydantic-compatible schema expressed in JSON format.

The schema must be flat (no nested objects are permitted in self-service mode) to align with datalake table constraints and enable automated transformation into Query API structures. The following fields are mandatory and enforced at registration:

- *uid* — A unique string identifier representing the observation.
- *modality* — Specifies which environment the data will be sent ([redacted]).
- *observation_type* — The observation type name.
- *start_time* and *end_time* — Timestamps defining the observation's duration.
- *vehicle_id* — Identifies the vehicle or source.

Additionally, a *Make end_time Required* flag specifies whether *end_time* is required in the data. If *end_time* is marked *required* but omitted during ingestion, the API automatically injects a default value of *0*.

Additional metadata accompanies the schema: the producer (your team or service), a list of allowed modalities, an optional ttl for simulation data retention, and a productionize flag that defaults to *false*. Upon successful validation, the system stores the schema in the global observation-type-registry DynamoDB table, using the *observation_type* as the partition key and an auto-incremented version (e.g., v1, v2) as the sort key. Every change—whether a new field, a type modification, or a toggle of the production flag—triggers a new version and a detailed diff record in the *observation-type-registry-history* table, giving full auditability.

This registration step is the gateway to both sandbox experimentation and production deployment. With a type defined, developers can begin producing test observations.

1.1.2 Producing and Storing Observations

With a registered type in place, producing an observation is a single HTTP POST to */observations*. The request payload must conform exactly to the active schema version. The API retrieves the latest schema from the registry, performs real-time validation, and, if valid, writes the record to the observations-api-data DynamoDB table.

Each observation carries a set of system-managed attributes alongside producer-defined fields. The uid serves as the primary key. The following hierarchical relationships are natively supported:

- *parent_id* links to an immediate predecessor.
- *root_id* points to the top-level ancestor.
- *path* encodes the full lineage as a concatenated string.

Global secondary indexes (GSIs) on *vehicle_id*, *parent_id*, *root_id*, and *observation_type* enable efficient filtering without scans.

For simulation modality observations, an expiration timestamp is computed based on the type-level time to live (TTL), which defaults to 3 days. This ensures automatic cleanup, which ensures that expired records are removed from DynamoDB but persist in the datalake marked with an archived flag.

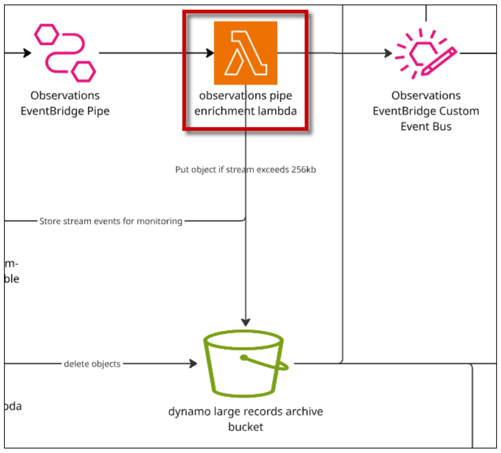
Updates follow a PUT to */observations/{uid}*, refreshing the expiration window and emitting a new stream event. Deletions—available only to the producer—trigger a soft delete; the record is removed from the API but appears in the datalake with *deleted=true*.

1.1.3 Finding Datasets

Dataset metadata is open to all with a valid Okta credential. A user calls the API to retrieve lists of registered datasets and their metadata or to retrieve a version list for a particular dataset. The API also provides an additional metadata keyword search route to enable the user interface search functionality. Using this metadata, it is the dataset creator's responsibility to prevent a logical dataset duplication.

1.1.4 From Storage to Distribution: The Enrichment Pipeline

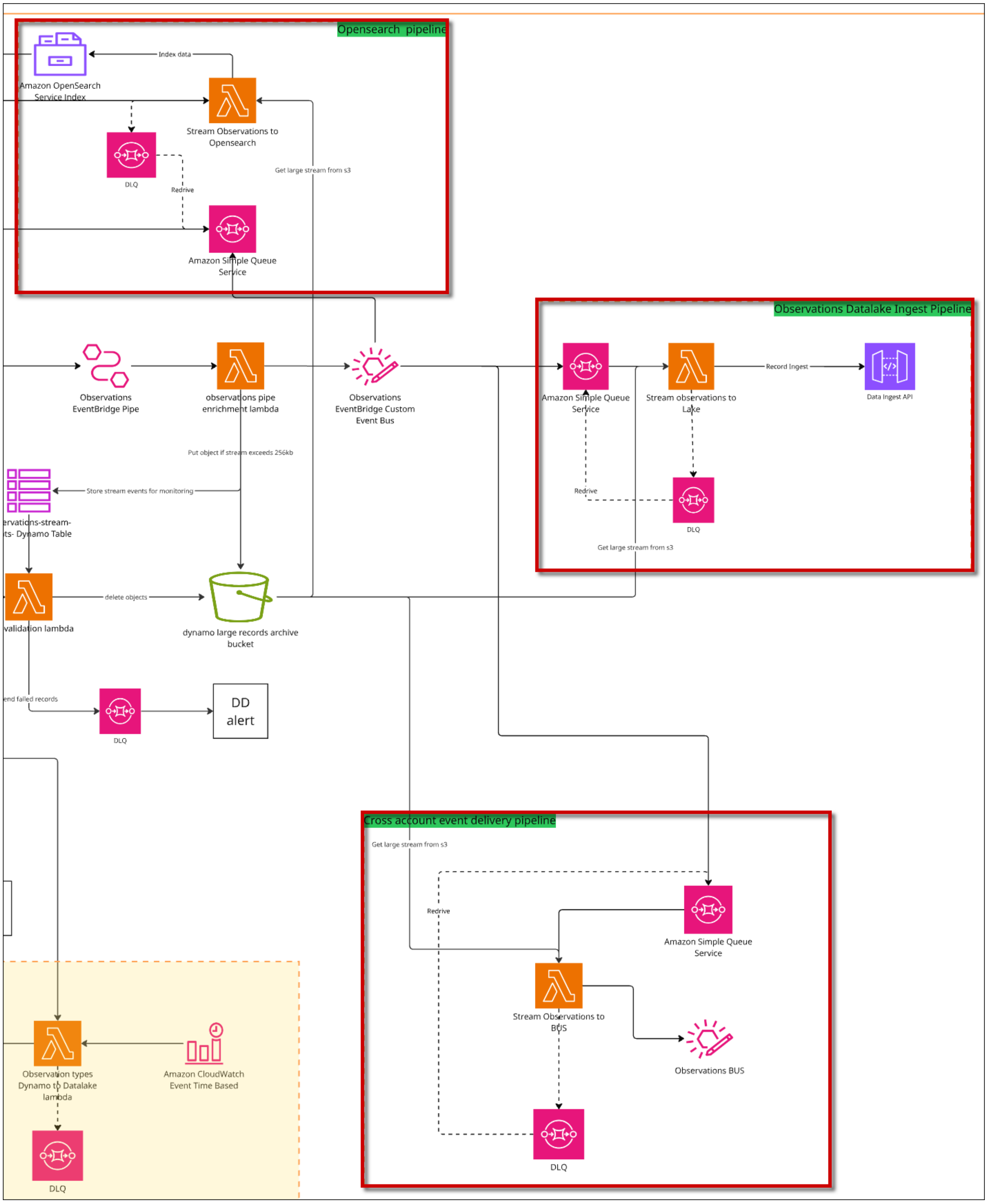
Every insert, update, or delete in either registry table emits a DynamoDB stream event, captured in us-east-1. A central Pipe Enrichment Lambda inspects each event and makes a critical routing decision based on payload size.


Observations EventBridge Pipe
observations pipe enrichment lambda
Observations EventBridge Custom Event Bus
Put object if stream exceeds 256kb
Store stream events for monitoring
m-
ble
delete objects
bda
dynamo large records archive bucket

If the record is 256 KB or smaller, it is embedded directly into an EventBridge custom event. Larger records are first archived to an S3 bucket (dynamo-large-records-archive), and only a small JSON payload containing the S3 key and metadata is forwarded. This bypasses EventBridge's size limitation while preserving end-to-end traceability.

From EventBridge, the event expands to three dedicated SQS queues, each supporting an independent pipeline:

- **OpenSearch Pipeline**: A lambda pulls the record (from the event or S3), transforms it, and updates the OpenSearch domain. This delivers sub-two-minute indexed search availability across all productionized types.
- **Observations Datalake Ingestion Pipeline**: Another lambda streams the record into the observations-db database. For productionized types, it lands in a dedicated table. Deletion and archival flags are injected as needed.
- **Cross-Account Event Delivery Pipeline**: A third lambda publishes the event to a separate EventBridge bus, enabling other [redacted] accounts or services to subscribe and react in real time.


Opensearch pipeline
Amazon OpenSearch Service Index
Index data
Stream Observations to Opensearch
DLQ
Redrive
Get large stream from s3
Amazon Simple Queue Service
Observations Datalake Ingest Pipeline
Amazon Simple Queue Service
Stream observations to Lake
Record Ingest
Data Ingest API
Redrive
DLQ
Get large stream from s3
Observations EventBridge Pipe
observations pipe enrichment lambda
Observations EventBridge Custom Event Bus
Put object if stream exceeds 256kb
Store stream events for monitoring
ervations-stream-
ts- Dynamo Table
validation lambda
delete objects
dynamo large records archive bucket
end failed records
DLQ
DD alert
Cross account event delivery pipeline
Get large stream from s3
Redrive
Amazon Simple Queue Service
Stream Observations to BUS
Observations BUS
DLQ
Observation types Dynamo to Datalake lambda
Amazon CloudWatch Event Time Based
DLQ

A dedicated monitoring table—*observations-stream-events*—tracks the completion status of each pipeline per uid. A periodic validation lambda queries OpenSearch and the datalake to confirm presence; any discrepancy routes the uid to a dead-letter queue, triggering Datadog alerts for operational monitoring.

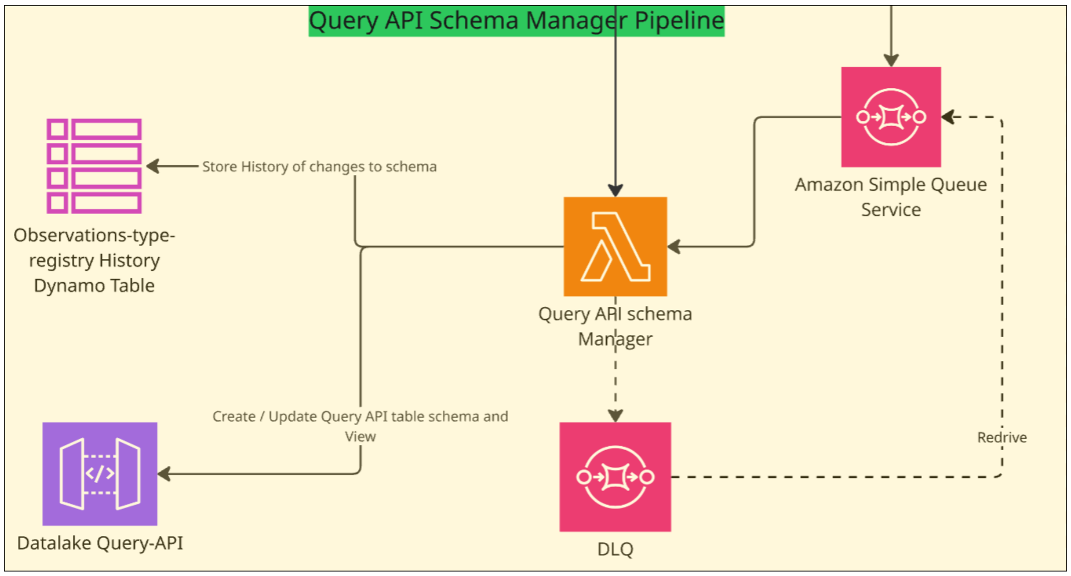
1.1.5 Sandbox Mode: Safe Iteration

When productionize is false, the system operates in sandbox mode. Observations are searchable, retrievable, and versioned within the API but short-lived (24 hours) and never leave the API boundary. No datalake tables are created, no OpenSearch documents are indexed for production, and no cross-account events are emitted.

This environment is ideal for schema prototyping. Developers can generate hundreds of test observations, inspect them via the UI or API, refine field definitions, and repeat without polluting production data stores. Only when the schema is stable and testing complete should productionize be enabled.

1.1.6 Productionization: From Sandbox to Scalable Data

Setting productionize to true, through the web UI or a PUT request, unlocks the full data lifecycle. The change triggers the query API schema manager pipeline.


Query API Schema Manager Pipeline
Store History of changes to schema
Observations-type-registry History Dynamo Table
Amazon Simple Queue Service
Query API schema Manager
Create / Update Query API table schema and View
Redrive
Datalake Query-API
DLQ

The current JSON schema is transformed into Query API format, and tables and latest views are automatically provisioned in the datalake under *observations_db.<observation_type>*. All future observations of this type flow through all three distribution pipelines.

Importantly, enabling productionization is a one-way transition. All existing sandbox observations of that type are immediately purged from the API to prevent data leakage into production streams. From this point forward, the type is subject to full retention policies, search indexing, and cross-account availability.

1.1.7 Searching and Consuming Observations

Real-time discovery is powered by OpenSearch. Developers can query via the */observations/search* API endpoint, the internal web UI, or direct OpenSearch client access (with appropriate IAM permissions). Filters include time ranges, vehicle IDs, observation types, modalities, and full-text content.

For analytical workloads, the datalake is the source of truth. Productionized types appear as structured Athena tables. Use uid to retrieve the full version history or query the latest_<type> view for the current state. Cross-account consumers subscribe to the EventBridge bus and receive enriched events within seconds of ingestion.

1.1.8 Managing Lifecycle: Retention and Deprecation

Simulation observations are ephemeral by design. Unless overridden via ttl, they expire after three days, vanishing from DynamoDB but persisting in the datalake with *archived=true*. Vehicle and HIL observations have no default expiration.

When an observation type is no longer needed, set *deprecated=true*. This blocks all new writes while preserving historical data for analysis. The type remains visible in the registry and UI but attempts to produce new observations return an error.

1.1.9 Developer Experience and Best Practices

The Observations API is designed for efficiency and accuracy. Start in sandbox mode to iterate rapidly on schema design. Use the web front end to browse versions, view change diffs, and monitor pipeline health. Leverage hierarchical fields to model complex event sequences, such as a planning cycle with child trajectory points. Subscribe to the cross-account EventBridge bus to power live dashboards or anomaly detectors.

The system guarantees that every observation successfully stored in the API will appear in OpenSearch within two minutes and in the datalake within two hours, with >99% reliability. Datadog monitors replication latency, pipeline backlogs, and data consistency, ensuring operational transparency.

By abstracting away the complexity of schema management, regional replication, large-payload handling, and multidestination streaming, the Observations API lets [redacted] developers focus on building robust, data-driven systems with confidence that their observations are valid, discoverable, and enduring.