SDL User Guide

Version latest, 2026-05-01

User Guide

Welcome to the SDL User Guide. This documentation is for end-users, analysts, and operators who need to understand and interact with the SOF Data Layer.

What is SDL?

SDL is a modular, container-native data platform that enables defense and intelligence organizations to collect, process, fuse, query, and disseminate mission-critical data — from cloud data centers to disconnected tactical edge nodes. A single software baseline adapts its runtime behavior to the deployment environment, ensuring operators work with an identical interface whether running at scale in the cloud or on a ruggedized server in a forward operating base.

For a detailed overview of the platform's architecture and capabilities, see the Platform Overview.

Core Capabilities

SDL provides a comprehensive set of capabilities organized around operational needs:

Capability	Description
Federated Query & VKG	"Zero ETL" queries across heterogeneous data sources via virtual knowledge graph
Streaming & Processing	Real-time event streaming and transformation hub supporting 14+ tactical formats
Policy Engine	Data-policy-as-code with row/column-level obfuscation and classification enforcement
Data Tiering & Storage	Hot/warm/cold storage tiers with metadata lake (Bronze, Silver, Gold)
DDIL & Edge Operations	Bandwidth-aware streaming with priority queues and automatic reconnection sync
Semantic Interoperability	BFO/CCO-compliant ontology layer with OBDA virtual mappings and SPARQL query

Getting Started

If you are new to SDL, start here:

1. **Platform Overview** — Understand the platform architecture and deployment model
2. **Data Pipelines** — Learn how to work with data transformation workflows
3. **SDL Examples** — Hands-on demonstrations and demo scripts

Operational Guides

For day-to-day platform operations:

- **Data Pipelines** — Configure and manage data transformation workflows
- **SDK** — Programmatic access via the platform SDK

Video Demonstrations

See SDL capabilities in action with video demonstrations covering mesh synchronization, federated query, streaming pipelines, cross-domain transfer, and more.

Need Developer Documentation?

If you need to **install, configure, or deploy** SDL, refer to the SDL Developer Guide.

Platform Capabilities

Platform Overview

SDL is a modular, container-native data platform that enables defense and intelligence organizations to collect, process, fuse, query, and disseminate mission-critical data — from cloud data centers to disconnected tactical edge nodes. A single software baseline adapts its runtime behavior to the deployment environment, ensuring operators work with an identical interface whether running at scale in the cloud or on a ruggedized server in a forward operating base.

Core Capabilities at a Glance

The capabilities below are available across SDL deployments. The specific capabilities enabled on a given node depend on the deployment form factor, mission requirements, and available resources.

Capability	Description	Learn More
Federated Query & VKG	"Zero ETL" virtual knowledge graph that federates queries across relational databases, streaming topics, object storage, and geospatial services.	Federated Query
Streaming & Processing	Real-time event streaming and transformation hub supporting 14+ tactical data formats with bidirectional conversion.	Streaming
Policy Engine	Data-policy-as-code enforcement with row-level and column-level obfuscation, classification markings, and multi-enclave evaluation.	Policy Engine
Data Tiering & Storage	Hot/warm/cold storage tiers with S3-compatible object storage, metadata lake (Bronze→Silver→Gold), and LZ4 compression.	Data Tiering
DDIL & Edge Operations	Bandwidth-aware streaming with priority queues, automatic caching during disconnection, and backfill sync on reconnection.	DDIL & Edge

Deployment Flexibility

SDL runs wherever the mission requires. The same container images and Helm charts deploy across four operating contexts without modification to application code.

Cloud

Full-scale deployment in classified or unclassified cloud environments. Horizontal scaling, managed storage, and high-bandwidth connectivity between nodes.

On-Premise

Bare-metal or virtualized data-center installations behind organizational firewalls. Supports air-gapped environments with local container registries and offline Helm charts.

Tactical Edge

Ruggedized single-server or small-cluster deployments at forward operating bases, command posts, or mobile platforms. Resource-constrained profiles reduce memory and CPU footprint while preserving core capabilities.

Autonomous Platforms

Embedded deployments on unmanned systems, sensor platforms, and vehicles. Operates fully disconnected with store-and-forward synchronization when connectivity returns.

Single Software Baseline

A defining principle of SDL is the **single software baseline**. Rather than maintaining separate codebases for cloud, on-premise, and edge deployments, one container image adapts its runtime behavior based on the deployment profile.

- **One image, many profiles** — the same OCI container image is promoted from development through staging to production across all deployment contexts.
- **Adaptive resource allocation** — the platform detects available CPU, memory, and GPU resources at startup and scales its internal services accordingly.
- **Feature flags, not forks** — capabilities that require hardware or network resources unavailable at the edge are gracefully disabled rather than compiled out.
- **Consistent operator experience** — the web UI, CLI, and APIs present the same interface regardless of deployment size.

This approach eliminates configuration drift between environments and ensures that every bug fix and feature improvement reaches every deployment tier through a single release pipeline.

Next Steps

- Learn how queries span heterogeneous data sources without ETL: Federated Query & Virtual Knowledge Graph
- Explore real-time format conversion and streaming: Streaming & Processing

Streaming & Processing

SDL provides a real-time event streaming and transformation hub that ingests data from sensors, full-motion video (FMV) metadata, cyber feeds, and operational systems. The hub converts between 14+ tactical data formats bidirectionally, enabling seamless interoperability across coalition partners, echelon boundaries, and heterogeneous C2 systems.

Path for Data Flow

Data entering SDL follows a processing path depending on the mission requirement.

The Native Transformation Path converts directly between external formats without persisting data in a canonical model.

1. **Inbound format received** — data arrives in Format A (e.g., CoT XML from a TAK server).
2. **Direct conversion** — the transformer converts Format A to Format B (e.g., GeoJSON for a mapping application).
3. **Outbound delivery** — the converted payload is delivered to the destination system.

This path is used for coalition interoperability scenarios where data must be relayed between partner systems in their native formats with minimal latency.

Supported Format Transformations

Format	Description
CoT XML	Cursor on Target XML — the standard position-reporting format used by TAK ecosystem applications.
CoT Protobuf	Binary protobuf encoding of Cursor on Target messages for reduced bandwidth and faster parsing.
GeoJSON	Open standard for encoding geographic features. Used by web mapping applications and GIS tools.
Lattice	Data exchange format for mesh networking platforms. Supports entity state, sensor data, and command-and-control messages.
Foundry	Ontology-based actions and object format for analytical platforms.
ISA XML	Intelligence, Surveillance, and Assessment XML format for structured intelligence reporting.
ETF Proto	Entity Tracking Framework protobuf format for high-rate entity tracking.
OMNI/TRAX	Track management format supporting multi-source track correlation and fusion.
TAK Data Package	Bundled data packages containing map overlays, imagery, and mission data for TAK clients.
CATAPULT IIR	Intelligence Information Report format for structured dissemination of intelligence products.
Additional formats	The transformer framework is extensible. New format converters can be deployed as containerized plugins without modifying the core platform.

Connector Ecosystem

Connectors handle the transport-layer integration between SDL and external systems. Each connector manages connection lifecycle, authentication, backpressure, and retry logic for its protocol.

Connector	Description
Event Streaming	Consumes and produces messages on streaming topics. Supports consumer groups, partitioned ordering, and configurable batching.
TAK Server	Connects to TAK servers with protocol auto-negotiation (TCP, TLS, WebSocket). Supports both CoT XML and CoT protobuf wire formats.
Lattice SDK	Integrates with mesh networking platforms via their native SDK for entity exchange and command relay.
Foundry Ontology Actions	Pushes and pulls objects and actions through the analytical platform's ontology API.
Gotham Maps / TWB	Streams geospatial layers and track data to and from operational mapping platforms.
TRAX gRPC Streaming	High-throughput gRPC streaming connector for track management systems.
TCP	Raw TCP socket connector for legacy systems that communicate over plain TCP streams.
PostGIS	Reads from and writes to geospatial databases using SQL with spatial extensions.

Security Markings

Classification markings are preserved and propagated through every stage of the streaming pipeline.

- **Inbound markings** — when a source record includes classification markings, those markings are extracted during ingestion and attached to the output message.
- **Marking modes** — operators configure one of four marking behaviors per connector:
  - **Add** — append markings to existing markings on the entity.
  - **Remove** — strip specific marking categories (for downgrade scenarios with appropriate guard validation).
  - **Overwrite** — replace existing markings with the connector's configured markings.
  - **Ignore** — pass through without modifying markings.
- **Outbound propagation** — when data is transformed to an external format, the markings are translated to the target format's marking schema where supported.

All marking changes are audit-logged with the source, destination, and policy that authorized the change.

Performance & Processing Modes

The streaming hub supports several processing modes to balance throughput, latency, and ordering requirements.

Batch Processing

For high-throughput ingestion scenarios, records are grouped into configurable batches before processing. Batching amortizes per-record overhead and increases throughput for bulk data loads, historical replay, and sensor dump ingestion.

Concurrent Writes

Multiple writer instances operate in parallel, each handling a subset of partitions or data sources. Concurrent writes scale horizontally with the number of available CPU cores and pods.

Partitioned Ordering

When strict ordering is required (for example, maintaining the sequence of updates to a single entity), the platform uses partition-key-based routing to ensure that all updates for a given entity are processed by the same writer instance in order.

Schema Registry

SDL maintains a schema registry that stores versioned schemas for every data format flowing through the platform.

- **Schema validation** — inbound records are validated against their registered schema before processing. Records that fail validation are routed to a dead-letter topic with diagnostic metadata.
- **Schema evolution** — the registry supports backward-compatible schema changes. New fields can be added without breaking existing consumers.
- **Schema discovery** — operators and developers can browse registered schemas through the platform's API to understand available data formats and their fields.

Correlation & Rehydration

Multi-Source Correlation

When multiple data sources report on the same real-world entity (for example, two sensors tracking the same vehicle), the streaming pipeline correlates these reports into a single fused entity. Correlation rules are configurable and can match on geographic proximity, entity identifiers, temporal overlap, or custom attribute comparisons.

Entity Rehydration

Entities that arrive as lightweight position reports (minimal projection from a bandwidth-constrained link) can be rehydrated with full attribute data from local storage or upstream nodes. Rehydration is triggered automatically when an operator requests detailed information about a minimally projected entity, or on a scheduled basis to pre-populate the local entity cache.

Next Steps

- Understand how streamed data is governed by policy: Policy Engine

- Learn about data tiering and storage for processed data: Data Tiering & Storage

Policy Engine

SDL treats data policy as code — policies are defined in code, versioned alongside the platform, tested through CI/CD pipelines, and deployed as part of every release. Rather than relying on external policy administration points, the platform embeds governance directly into the data layer so that every query, every replication event, and every cross-domain transfer is subject to the same enforceable rules.

Row-Level and Column-Level Obfuscation

The policy engine supports fine-grained obfuscation at both the row and column level:

Row-level filtering

Entire records are filtered based on the requesting user's clearance level and need-to-know designations. If a user lacks authorization for a particular record's classification or compartment, that record is excluded from query results entirely.

Column-level obfuscation

Individual fields within a record can be redacted, masked, or generalized based on the data sensitivity of that field. For example, a user with appropriate clearance may see the full entity record, while a user at a lower authorization level sees the same record with sensitive fields replaced by masked values or removed altogether.

All obfuscation is applied at query time. The underlying data remains unchanged in storage, preserving the full-fidelity record for users who hold the required authorization.

Node-Local Policy Enforcement

Policies travel with the deployment. Each SDL node carries the full set of applicable policies and evaluates them locally against its own data and the requesting user's context.

- No reach-back to a central policy server is required for policy evaluation.
- Nodes operate autonomously, enforcing governance even in disconnected or degraded network conditions.
- Policy updates propagate through the mesh alongside data, ensuring that every node converges on the current policy set without requiring a dedicated control channel.

This architecture means that policy enforcement is not a single point of failure and that edge deployments maintain the same governance guarantees as cloud-hosted instances.

Classification Markings

Every entity and every task in SDL carries a complete security marking object. The marking system supports the full range of classification and dissemination controls:

Marking Type	Examples
Classification level	UNCLASSIFIED, CONFIDENTIAL, SECRET, TOP SECRET
Dissemination controls	NOFORN, REL TO, FOUO, and other standard controls
SCI/SAP compartments	Compartmented access designations as required by the data source
Releasability markings	Nationality-based release designations (e.g., REL TO USA, FVEY)

These markings are not metadata annotations applied after the fact. They are integral to the entity data model and are evaluated by the policy engine at every access point.

Multi-Enclave Policy Evaluation

In multi-enclave deployments, the policy engine evaluates security context across classification boundaries:

- **FVEY, NOFORN, and REL TO evaluation** — Policies determine whether a given entity's releasability markings permit transfer to the requesting enclave or user.
- **Source and destination enclave security levels** — The engine evaluates the classification ceiling of both the originating and receiving enclaves before permitting data flow.
- **Cross-domain transfer decisions** — Each entity's marking is compared against the destination enclave's authorization. Transfer is permitted only when the entity marking falls within the destination's allowed classification range.
- **Automatic downgrade and redaction** — Where policy permits, entities can be automatically downgraded or have sensitive fields redacted to meet the destination enclave's classification ceiling.

Integration with Platform Capabilities

The policy engine is not a standalone component. It integrates directly with the core platform capabilities:

Query

Policies enforce row-level and column-level filtering at query execution time. Users see only the data they are authorized to access, with no additional application logic required.

Cross-domain

Policies govern what data flows between classification boundaries. Every cross-domain transfer decision is evaluated against the entity's marking and the destination enclave's authorization.

Streaming

Security markings are propagated through transformation pipelines. As data moves through enrichment and fusion stages, the resulting entities inherit or receive updated markings based on the source data and the transformation rules.

Data Tiering & Storage

SDL implements a multi-tier storage architecture optimized for different access patterns and data freshness requirements. Data moves through well-defined tiers — from raw ingestion through normalization and fusion — with each tier providing the appropriate balance of latency, throughput, and storage efficiency.

Storage Tiers

The platform organizes operational data into three access tiers based on age and query frequency:

Tier	Latency	Description	Capacity Profile
Hot	<2ms	Last 10 minutes of operational data. Optimized for real-time queries and the highest throughput requirements.	Highest throughput
Warm	<2 sec	Last 2 weeks of data. Balances query speed with increased storage capacity for recent historical access.	Balanced speed and capacity
Cold	<5 sec	Older data beyond the warm window. Optimized for storage efficiency while maintaining query latency under 5 seconds.	Optimized for storage efficiency

Tier boundaries are configurable per deployment. Data transitions between tiers automatically based on age, with no manual intervention required.

Metadata Store

Beyond the operational access tiers, the platform maintains a structured metadata pipeline that tracks data quality and provenance:

Bronze (RAW)

Ingested data in its original format. This tier serves as the immutable audit copy, preserving the exact representation received from the source system. No transformations are applied at this stage.

Silver (Standard Schema)

Data normalized to the canonical data model. Records are validated against the schema, enriched with standard metadata (timestamps, source identifiers, classification markings), and made available for downstream processing.

Gold (Fused)

Conflict-resolved, multi-source fused entities with full provenance chains. When multiple sources report on the same real-world entity, the fusion process reconciles conflicts, merges attributes, and produces a single authoritative record that retains links to every contributing source.

Object Storage

SDL provides S3-compatible object storage for bulk data, attachments, and artifacts that fall outside the structured entity model:

- **S3-compatible APIs** for programmatic access using standard tooling and client libraries.
- **LZ4 compression** applied to stored objects for storage efficiency without significant CPU overhead.
- **Lifecycle policies** for automatic tier migration, moving infrequently accessed objects to lower-cost storage classes on a configurable schedule.
- **Multi-region replication** where the deployment topology and network connectivity support it, ensuring object availability across geographically distributed nodes.

ETL & Enrichment Pipelines

Data movement through the tier hierarchy is managed by configurable pipelines. Operators build, monitor, and manage pipelines through the pipeline catalog, which provides a centralized view of all active, inactive, and degraded pipelines across the deployment.

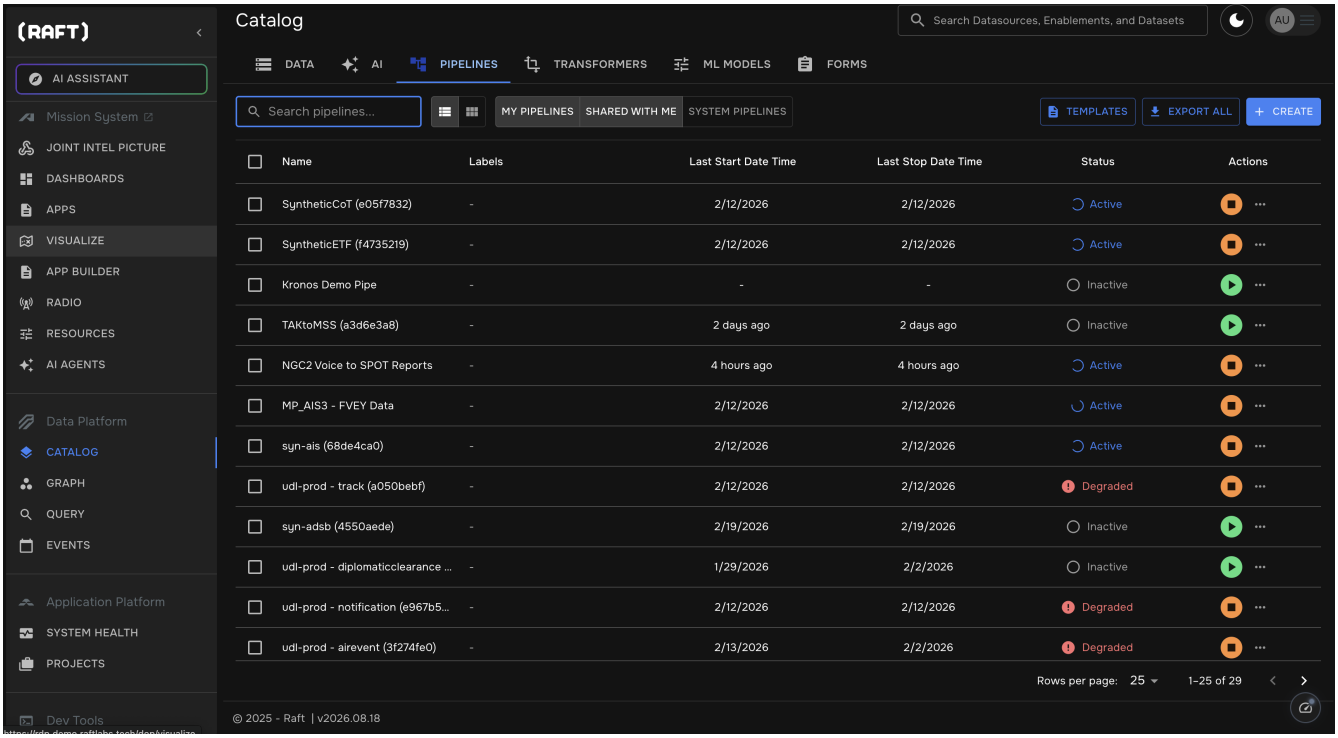


Figure 1. Pipeline Catalog — centralized view of all configured pipelines with status, scheduling, and lifecycle controls

Each pipeline is assembled visually as a directed graph of stages. Operators chain together data generators, format transformers, storage connectors, and enrichment steps using a drag-and-drop builder. The built-in data preview shows live streaming records flowing through the pipeline, enabling operators to verify transformations in real time before promoting a pipeline to production.

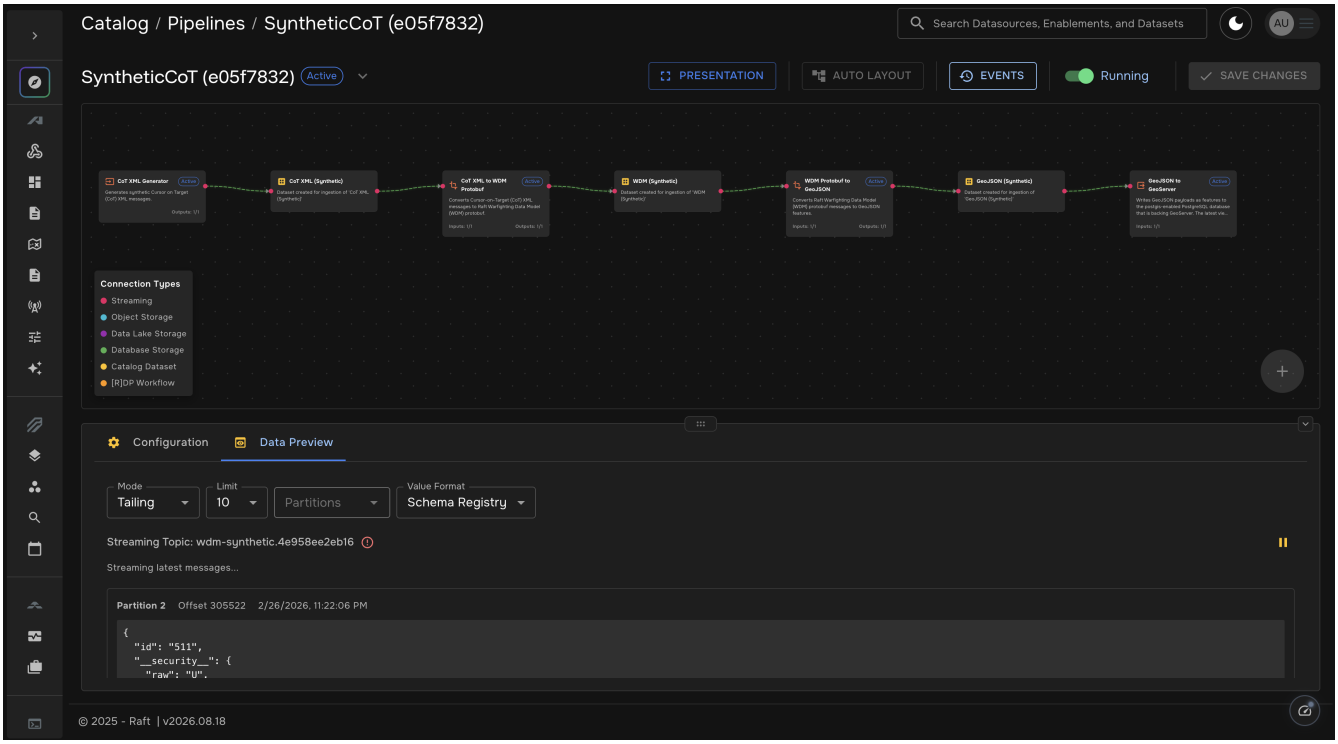


Figure 2. Pipeline Detail — visual stage builder with live data preview showing a CoT-to-GeoServer transformation chain

Low/No-Code ETL configuration

Pipeline definitions use declarative configuration that allows operators to define extraction, transformation, and loading workflows without writing custom code.

Enrichment pipelines

Dedicated enrichment stages add context to records as they progress through the tiers. Enrichments include geospatial correlation, temporal alignment, and classification marking assignment based on source and content analysis.

Schema validation at tier boundaries

Each transition between tiers — Bronze to Silver, Silver to Gold — includes schema validation to ensure data integrity. Records that fail validation are flagged and routed to exception handling rather than silently propagated.

Lineage tracking

The platform maintains a complete lineage record from raw ingestion through fusion. Every transformation, enrichment, and merge operation is logged, enabling operators to trace any fused entity back to its original source records.

DDIL & Edge Operations

Disconnected, Disrupted, Intermittent, and Limited bandwidth (DDIL) operations are first-class capabilities in SDL, not afterthoughts or degraded modes. The platform is designed from the ground up to operate reliably when network connectivity is constrained, unreliable, or completely absent.

Bandwidth-Aware Streaming

SDL implements message priority tiers with bandwidth-aware streaming to ensure that the most critical data reaches its destination first:

- Messages are assigned to priority tiers based on data type and operational importance.
- Configurable bandwidth thresholds define how much available bandwidth each priority tier receives.
- Higher-priority data (e.g., threat warnings, time-sensitive operational updates) is transmitted first, consuming available bandwidth before lower-priority traffic.
- Lower-priority data is queued until sufficient bandwidth is available, ensuring that critical messages are never delayed by bulk transfers.

Disconnection Handling

The platform handles connectivity loss gracefully, with no operator intervention required:

- **Automatic local caching** — When bandwidth drops below a configurable threshold, the node caches outbound data locally and continues normal operations against its local data store.
- **No data loss** — Connectivity interruptions do not result in lost messages or dropped updates. All data is persisted locally until it can be transmitted.
- **Automatic sync on reconnection** — When connectivity is restored, the node pulls missed updates from the mesh and pushes locally accumulated changes, restoring consistency without manual intervention.
- **Backfill snapshots** — Nodes that have been disconnected for an extended period receive a targeted snapshot to bring them up to the current state efficiently, rather than replaying the full change history.

Performance Validation

SDL has been validated across a wide range of bandwidth and latency conditions:

Condition	Validated Performance
Full bandwidth (200 Mbps / 25ms)	Nominal throughput with full mesh consistency
Degraded bandwidth (down to 0.01 Mbps / 600ms)	Priority-based data delivery maintained, lower-priority data queued
40-node mesh at degraded bandwidth	Sustained 20,000+ entities across the mesh
Partition recovery	Full consistency restored in 10—15 seconds after partition heals

Single Software Baseline

A single software baseline runs across all deployment environments:

- The same container image runs in cloud, on-premise, tactical edge, and autonomous platform

deployments.

- Runtime behavior adapts to available resources and connectivity — a cloud instance with abundant bandwidth and compute uses the same software as a ruggedized edge device with constrained resources.
- Operators use an identical interface regardless of deployment scale, eliminating the need for environment-specific training or procedures.

This approach ensures that capabilities validated in one environment are available in every environment, and that software updates are tested and deployed through a single pipeline.

Semantic Interoperability

Semantic interoperability ensures that data exchanged between systems retains its meaning regardless of the source format or originating system. SDL achieves this through standards-based ontology alignment, virtual semantic mappings, and a knowledge graph projection layer that lets users query heterogeneous data as though it resided in a single, unified model.

BFO/CCO Compliance

SDL's semantic layer is built on Basic Formal Ontology (BFO) and Common Core Ontologies (CCO), in compliance with the DNI memorandum mandating their adoption across the defense and intelligence communities.

- **Shared top-level framework** — BFO provides a domain-neutral upper ontology that defines the most general categories of entities and processes. CCO extends BFO with mid-level concepts for information, agents, events, and artifacts.
- **Cross-organizational reasoning** — because BFO and CCO are shared standards, data annotated against them can be understood and reasoned over by any system that implements the same ontological framework, regardless of organizational boundaries.
- **Formal semantics** — the ontology carries machine-readable axioms that enable automated consistency checking, classification, and inference across datasets from different producers.

Ontology Support

The platform ships with support for defense-specific ontology extensions and provides an extensibility framework for mission-tailored vocabularies.

Defense Information Core Ontology (DICO)

DICO extends the BFO/CCO foundation with concepts specific to defense information management — intelligence reports, operational orders, mission plans, and associated metadata. It provides the vocabulary needed to represent and query defense-domain data in a semantically consistent way.

Joint Information Knowledge Ontology (JIKO)

JIKO adds joint interoperability concepts that span service branches and coalition partners. It

defines shared semantics for entities, activities, and relationships that are common across joint operations.

Extensible Ontology Framework

Operators can extend the ontology with mission-specific vocabularies:

- New classes and properties are defined using standard OWL/RDF syntax.
- Extensions are validated against the BFO/CCO upper ontology to ensure consistency.
- Deployed extensions are immediately available to the virtual knowledge graph and federated query engine.
- Multiple extensions can coexist, enabling different missions to define domain-specific concepts without conflicting with one another.

OBDA Virtual Mappings

Ontology-Based Data Access (OBDA) creates virtual mappings from native data structures to ontology projections. These mappings are the bridge between the physical world of databases, streaming topics, and object stores and the semantic world of ontology classes and properties.

- **No data movement required** — mappings are evaluated at query time. Data remains in its source system; no ETL pipelines, no materialized triples, no synchronization jobs.
- **Native data sources supported** — relational databases, streaming topics, and S3-compatible object stores are all projected into RDF triples through their respective connectors.
- **Declarative mapping definitions** — each mapping specifies the source table or topic, the target ontology class or property, and any column-to-property transformations. Mappings are defined once and updated only when the source schema changes.
- **SPARQL query interface** — users query the virtualized ontology views using standard SPARQL. The OBDA engine translates SPARQL queries into optimized native queries against each source system.

Knowledge Graph Projection

The virtual knowledge graph (VKG) constructs semantic views over operational data, enabling graph-based reasoning and traversal without materializing a separate graph database.

RDF and SPARQL

The platform's semantic layer is built on W3C standard RDF (Resource Description Framework) and SPARQL (SPARQL Protocol and RDF Query Language).

- RDF provides the data model: every data element is represented as a subject-predicate-object triple.
- SPARQL provides the query language: users write declarative graph queries that traverse relationships, filter by properties, and aggregate results.

Graph Traversal and Reasoning

The VKG supports multi-hop graph traversal, enabling queries that follow chains of relationships across entities and data sources. Reasoning capabilities allow the system to infer new relationships from existing data based on the ontology's axioms — for example, inferring that an entity participating in a specific activity is also associated with the activity's parent operation.

Natural-Language Query Interface

Non-technical users interact with the knowledge graph through a natural-language interface. Users describe their information needs in plain language, and the VKG translates the request into a structured SPARQL query against the ontology. This enables operators without query-language expertise to access the full depth of the semantic layer.

Next Steps

- Explore data-policy enforcement applied to semantic query results: Policy Engine

Video Demos

This section will contain video demonstrations of SDL capabilities. Videos are currently in production.

Platform Overview Demo

An end-to-end walkthrough of the platform's user interface, navigation, and core features. See how operators interact with data across the full platform.

Video coming soon.

Federated Query Demo

Query data across multiple heterogeneous sources without moving or duplicating data. See SPARQL and natural-language queries in action.

Video coming soon.

Streaming Pipeline Demo

Observe live data flowing through transformation pipelines, converting between tactical formats in real time. See security markings propagated through transformations.

Video coming soon.

Policy Engine Demo

See data-policy-as-code in action with row-level and column-level obfuscation. Watch how classification markings govern data visibility.

Video coming soon.

DDIL Operations Demo

Observe platform behavior during bandwidth degradation and disconnection. See priority-based data streaming and automatic sync on reconnection.

Video coming soon.

Edge Deployment Demo

See the platform running on ruggedized edge hardware with the same interface as cloud deployments. Watch local operations during complete disconnection.

Video coming soon.

Data Pipelines

SDL Data Pipelines allow users to develop, create, and manage complex data transformations at scale. These transformations can work on batch or streaming data in a variety of formats and protocols.

See the Available Transformers for a listing of available transformers.

Transformer Guide contains a step-by-step guide to each part of a Transformer as it's represented in **SDL**.

Available Transformers

This reference documents all transformers available in SDL.

Transformers are processing units in a pipeline that do one of:

- ingress data from a source
- transform or filter data
- egress data to a sink

For information on how to create and register custom transformers, see Transformer Guide.

For information on dynamic transformers (JavaScript, Python, etc), see Dynamic Transformers.

Overview

Transformers are categorized by their primary function:

- **Format Conversion Transformers**: Convert data between different formats (XML, Protobuf, JSON, GeoJSON)
- **Sink Transformers**: Write data to external systems (databases, APIs)
- **Source Transformers**: Ingest data from external systems into SDL
- **Data Processing Transformers**: Filter, enrich, or otherwise process data streams
- **Dynamic Transformers**: User-defined transformation logic (see Dynamic Transformers)

Format Conversion Transformers

These transformers convert data between different formats, enabling interoperability between systems that use different data representations.

CoT XML to GeoJSON

Converts Cursor-on-Target (CoT) XML messages to GeoJSON features.

Property	Value
UID	`urn:rdp:transformer:cotxml-to-geojson`
Input	Kafka topic (CoT XML messages)
Output	Kafka topic (GeoJSON features)
Labels	`cot`, `xml`, `geojson`

Configuration Parameters:

Parameter	Description	Default	Required
`LOG_LEVEL`	Log level (trace, debug, info, warn, error, fatal, panic, disabled)	`info`	No

Use Cases:

- Converting CoT tracking data for visualization in GIS applications
- Enabling web-based mapping applications to display CoT entities

CoT XML to CoT Protobuf

Converts Cursor-on-Target (CoT) XML messages to their protobuf equivalent.

Property	Value
UID	`urn:rdp:transformer:cotxml-to-cotproto`
Input	Kafka topic (CoT XML messages)
Output	Kafka topic (CoT Protobuf messages)
Labels	`cot`, `xml`, `protobuf`

Configuration Parameters:

Parameter	Description	Default	Required
`LOG_LEVEL`	Log level (trace, debug, info, warn, error, fatal, panic, disabled)	`info`	No

Use Cases:

- Reducing message size for bandwidth-constrained networks
- Improving serialization/deserialization performance
- Preparing CoT data for systems that require protobuf format

CoT XML to ETF Protobuf

Converts Cursor-on-Target (CoT) XML messages to Entity Tracking Format (ETF) protobuf.

Property	Value
UID	`urn:rdp:transformer:cotxml-to-etfproto`
Input	Kafka topic (CoT XML messages)
Output	Kafka topic (ETF Protobuf messages)
Labels	`cot`, `xml`, `etf`, `protobuf`

Configuration Parameters:

Parameter	Description	Default	Required
`LOG_LEVEL`	Log level (trace, debug, info, warn, error, fatal, panic, disabled)	`info`	No

Use Cases:

- Converting legacy CoT XML data to modern ETF format
- Integrating CoT sources with ETF-based systems

CoT XML to Lattice Entities

Converts Cursor-on-Target (CoT) XML messages to Anduril Lattice entity JSON.

Property	Value
UID	`urn:rdp:transformer:cotxml-to-lattice`
Input	Kafka topic (CoT XML messages)
Output	Kafka topic (Lattice JSON entities)
Labels	`cot`, `xml`, `lattice`

Configuration Parameters:

Parameter	Description	Default	Required
`LATTICE_INTEGRATION_NAME`	Integration name to set in Lattice entity provenance	`SOF Data Layer`	No
`LATTICE_SIMULATED`	Mark entities as simulated (sets indicators.simulated flag)	`false`	No
`LOG_LEVEL`	Log level (trace, debug, info, warn, error, fatal, panic, disabled)	`info`	No

Use Cases:

- Publishing CoT tracks to Anduril Lattice for C2 visualization
- Converting CoT data for consumption by Lattice-based applications

CoT Protobuf to CoT XML

Converts Cursor-on-Target (CoT) protobuf messages to their XML equivalent.

Property	Value
UID	`urn:rdp:transformer:cotproto-to-cotxml`
Input	Kafka topic (CoT Protobuf messages)
Output	Kafka topic (CoT XML messages)
Labels	`cot`, `protobuf`, `xml`

Configuration Parameters:

Parameter	Description	Default	Required
`LOG_LEVEL`	Log level (trace, debug, info, warn, error, fatal, panic, disabled)	`info`	No

Use Cases:

- Converting protobuf data for legacy systems that require XML
- Human-readable debugging of CoT protobuf messages

CoT Protobuf to Lattice Entities

Converts Cursor-on-Target (CoT) Protobuf messages to Anduril Lattice entity JSON.

Property	Value
UID	`urn:rdp:transformer:cotproto-to-lattice`
Input	Kafka topic (CoT Protobuf messages)
Output	Kafka topic (Lattice JSON entities)
Labels	`cot`, `protobuf`, `lattice`

Configuration Parameters:

Parameter	Description	Default	Required
`LATTICE_INTEGRATION_NAME`	Integration name to set in Lattice entity provenance	`SOF Data Layer`	No
`LATTICE_SIMULATED`	Mark entities as simulated (sets indicators.simulated flag)	`false`	No
`LOG_LEVEL`	Log level (trace, debug, info, warn, error, fatal, panic, disabled)	`info`	No

Use Cases:

- Publishing CoT protobuf tracks to Anduril Lattice

- Integrating protobuf-based CoT sources with Lattice

ETF Protobuf to Lattice Entities

Converts Entity Tracking Format (ETF) protobuf messages to Anduril Lattice entity JSON.

Property	Value
UID	`urn:rdp:transformer:etfproto-to-lattice`
Input	Kafka topic (ETF Protobuf messages)
Output	Kafka topic (Lattice JSON entities)
Labels	`etf`, `protobuf`, `lattice`

Configuration Parameters:

Parameter	Description	Default	Required
`LATTICE_INTEGRATION_NAME`	Integration name to set in Lattice entity provenance	`SOF Data Layer`	No
`LATTICE_SIMULATED`	Mark entities as simulated (sets indicators.simulated flag)	`false`	No
`LOG_LEVEL`	Log level (trace, debug, info, warn, error, fatal, panic, disabled)	`info`	No

Use Cases:

- Publishing ETF tracks to Anduril Lattice
- Converting GCCS-J or other ETF sources for Lattice consumption

ETF Protobuf to CoT Protobuf

Converts Entity Tracking Format (ETF) protobuf messages to Cursor-on-Target (CoT) protobuf.

Property	Value
UID	`urn:rdp:transformer:etfproto-to-cotproto`
Input	Kafka topic (ETF Protobuf messages)
Output	Kafka topic (CoT Protobuf messages)
Labels	`etf`, `protobuf`, `cot`

Configuration Parameters:

Parameter	Description	Default	Required
`LOG_LEVEL`	Log level (trace, debug, info, warn, error, fatal, panic, disabled)	`info`	No

Use Cases:

- Converting modern ETF data to CoT format for legacy systems
- Bridging ETF-based and CoT-based tracking systems

ETF Protobuf to CoT XML

Converts Entity Tracking Format (ETF) protobuf messages to Cursor-on-Target (CoT) XML.

Property	Value
UID	`urn:rdp:transformer:etfproto-to-cotxml`
Input	Kafka topic (ETF Protobuf messages)
Output	Kafka topic (CoT XML messages)
Labels	`etf`, `protobuf`, `cot`, `xml`

Configuration Parameters:

Parameter	Description	Default	Required
`LOG_LEVEL`	Log level (trace, debug, info, warn, error, fatal, panic, disabled)	`info`	No

Use Cases:

- Converting ETF data to CoT XML for legacy systems
- Human-readable debugging of ETF protobuf messages

ETF Protobuf to GeoJSON

Converts Entity Tracking Format (ETF) protobuf messages to GeoJSON features.

Property	Value
UID	`urn:rdp:transformer:etfproto-to-geojson`
Input	Kafka topic (ETF Protobuf messages)
Output	Kafka topic (GeoJSON features)
Labels	`etf`, `protobuf`, `geojson`

Configuration Parameters:

Parameter	Description	Default	Required
`LOG_LEVEL`	Log level (trace, debug, info, warn, error, fatal, panic, disabled)	`info`	No

Use Cases:

- Visualizing ETF tracks in web-based mapping applications
- Converting ETF data for GIS tools

ETF Protobuf to JSON

Converts Entity Tracking Format (ETF) protobuf messages to human-readable JSON for debugging.

Property	Value
UID	`urn:rdp:transformer:etfproto-to-json`
Input	Kafka topic (ETF Protobuf messages)
Output	Kafka topic (JSON messages)
Labels	`etf`, `protobuf`, `json`

Configuration Parameters:

Parameter	Description	Default	Required
`LOG_LEVEL`	Log level (trace, debug, info, warn, error, fatal, panic, disabled)	`info`	No

Use Cases:

- Debugging ETF protobuf messages
- Inspecting ETF data structure during development
- Creating human-readable logs of ETF data streams

This transformer is intended for debugging and should not be used in production pipelines due to the large message sizes generated.

Kafka Passthrough

Passes all messages from the inbound topic to the outbound topic without modification.

Property	Value
UID	`urn:rdp:transformer:passthrough`
Input	Kafka topic (any format)
Output	Kafka topic (same format as input)
Labels	`passthrough`, `kafka`

Configuration Parameters:

Parameter	Description	Default	Required
`LOG_LEVEL`	Log level (trace, debug, info, warn, error, fatal, panic, disabled)	`info`	No

Use Cases:

- Testing data pipeline connectivity

- Copying data between topics
- Creating topic mirrors for separate consumption paths
- Adding monitoring points in data pipelines

Source Transformers

Source transformers ingest data from external systems into SDL. They do not consume from Kafka topics but produce output topics.

UDL Ingester

Ingests data from Unified Data Library (UDL).

Property	Value
UID	`urn:rdp:source:udl`
Input	None (polls UDL API)
Output	Kafka topic
Labels	`sdl.catalog.datasource=f39f9db2-21cb-4f0a-a0a6-ca0befbafd1b`
Type	Source

Configuration Parameters:

Parameter	Description	Default	Required
`CATALOG_DATASOURCE_PATH`	Path to the datasource in the catalog	*(empty)*	Yes
`CATALOG_DATASET_PATH`	Path to the dataset in the catalog	*(empty)*	Yes
`CLASSIFICATION_LEVEL`	Network classification	*(empty)*	Yes
`OAUTH_CLIENT_ID`	Client ID for internal OAuth2 communications	*(empty)*	Yes
`OAUTH_CLIENT_SECRET`	Client secret for internal OAuth2 communications	*(empty)*	Yes
`UDL_URL`	URL of the Unified Data Library instance	*(Internet-facing instance)*	No
`UDL_USER`	Username for authentication	*(empty)*	Yes
`UDL_PASS`	Password for authentication	*(empty)*	Yes
`CATALOG_URL`	URL of the SDL Catalog API	*(empty)*	No
`DF_CLASSIFICATION_URL`	URL of the SDL Classification service	*(empty)*	No

Use Cases:

- Ingesting data from DoD's Unified Data Library
- Pulling reference data into SDL

- Synchronizing external datasets

REST JSON API Ingester

Ingests data from a generic JSON REST API.

Property	Value
UID	`urn:rdp:source:rest`
Input	None (polls JSON API)
Output	Kafka topic (JSON messages)
Labels	`sdl.catalog.datasource=REST`
Type	Source

Configuration Parameters:

Parameter	Description	Default	Required
`CATALOG_DATASOURCE_PATH`	Path to the datasource in the catalog	*(empty)*	Yes
`CATALOG_DATASET_PATH`	Path to the dataset in the catalog	*(empty)*	Yes
`CLASSIFICATION_LEVEL`	Network classification	*(empty)*	Yes
`OAUTH_CLIENT_ID`	Client ID for internal OAuth2 communications	*(empty)*	Yes
`OAUTH_CLIENT_SECRET`	Client secret for internal OAuth2 communications	*(empty)*	Yes
`HTTP_JSON_CONFIG`	JSON configuration for the HTTP client	*(empty)*	No
`CATALOG_URL`	URL of the SDL Catalog API	*(empty)*	No
`DF_CLASSIFICATION_URL`	URL of the SDL Classification service	*(empty)*	No

Use Cases:

- Ingesting data from custom REST APIs
- Polling external services for data
- Integrating third-party data sources

GCCS-J ETF Generator

Generates realistic mock ETF (Entity Tracking Format) protobuf messages using ShadowTraffic for testing and development.

Property	Value
UID	`urn:rdp:source:gccsj-etf-mock`
Input	None (generates mock data)
Output	Kafka topic (ETF Protobuf messages)

Property	Value
Labels	`gccs-j`, `etf`, `mock`
Type	Source (mock data generator)

Configuration Parameters:

Parameter	Description	Default	Required
`LOG_LEVEL`	ShadowTraffic log level (TRACE, DEBUG, INFO, WARN, ERROR)	`INFO`	No
`THROTTLE_MS`	Milliseconds between message generation cycles	`5000`	No

Generated Data:

- Military tracking data with MIL-2525 codes
- AIS (Automatic Identification System) data
- Realistic callsigns and unit identifiers
- Geospatial positioning data

Use Cases:

- Testing ETF processing pipelines
- Development and debugging without live data sources
- Load testing and performance evaluation
- Training and demonstration environments

This transformer requires ShadowTraffic to be enabled in the platform configuration and is conditionally included based on the `global.shadowtraffic.enabled` setting.

Sink Transformers

Sink transformers write data to external systems, databases, or APIs. They consume from Kafka topics but do not produce output topics.

Publish to Lattice

Publishes entities to Anduril Lattice via the Lattice API.

Property	Value
UID	`urn:rdp:sink:lattice`
Input	Kafka topic (Lattice JSON entities)
Output	None (writes to Lattice API)

Property	Value
Labels	`lattice`
Type	Sink

Configuration Parameters:

Parameter	Description	Default	Required
`LATTICE_ENDPOINT`	Lattice API endpoint URL	*(empty)*	Yes
`LATTICE_TOKEN`	Lattice API authentication token	*(empty)*	Yes
`LATTICE_SANDBOX_TOKEN`	Lattice sandbox authorization token (sent as anduril-sandbox-authorization header)	*(empty)*	No
`LATTICE_MAX_RETRIES`	Maximum number of retries for transient failures	`3`	No
`LATTICE_TIMEOUT_MS`	Request timeout in milliseconds	`30000`	No
`LATTICE_RESET_INVALID_EXPIRY`	Reset any invalid expiryTime values to LATTICE_DEFAULT_EXPIRY_SECONDS	`false`	No
`LATTICE_DEFAULT_EXPIRY_SECONDS`	Default expiry period (from now) in seconds when resetting invalid expiryTime	`86400` (24 hours)	No
`LATTICE_OVERWRITE_SOURCE_UPDATE_TIME`	Overwrite provenance.sourceUpdateTime to current time before publishing	`false`	No
`LATTICE_INTEGRATION_NAME`	Integration name to set in Lattice entity provenance	`SOF Data Layer`	No
`LOG_LEVEL`	Log level (trace, debug, info, warn, error, fatal, panic, disabled)	`info`	No

Use Cases:

- Publishing real-time tracks to Lattice C2 system
- Integrating SDL data with Anduril's ecosystem
- Feeding processed sensor data to Lattice

GeoJSON to GeoServer

Writes GeoJSON payloads as features to a PostGIS-enabled PostgreSQL database backing GeoServer.

Property	Value
UID	`urn:rdp:sink:geojson-to-postgis`
Input	Kafka topic (GeoJSON features)

Property	Value
Output	None (writes to PostgreSQL/PostGIS)
Labels	`gis`, `postgres`, `geojson`
Type	Sink

Configuration Parameters:

Parameter	Description	Default	Required
`POSTGRES_TABLE_NAME`	Table name (view name, history table will be {table}_history)	`geojson_features`	No
`POSTGRES_SCHEMA`	Schema name	`public`	No
`POSTGRES_BATCH_SIZE`	Buffer this many messages before writing	`100`	No
`POSTGRES_FLUSH_INTERVAL`	Max milliseconds to buffer messages before flush	`50`	No
`GEOJSON_FEATURE_ID_FIELD`	Property field to use as feature ID if GeoJSON id is missing	`uid`	No
`LOG_LEVEL`	Log level (trace, debug, info, warn, error, fatal, panic, disabled)	`info`	No

Database Schema:

- **View**: `geojson_features` (or configured table name) - Latest view of all unique features
- **History Table**: `geojson_features_history` - Full history of all feature updates

Use Cases:

- Serving GeoJSON data via WFS/WMS from GeoServer
- Maintaining historical track data in PostGIS
- Enabling spatial queries on streaming geospatial data
- Visualizing real-time tracks in GIS applications

PostgreSQL connection details are automatically configured from the `sdl-geoserver-db-app` secret and are not user-configurable.

Iceberg Sink

Sinks data into an Apache Iceberg table for advanced analytics.

Property	Value
UID	`urn:rdp:sink:iceberg`
Input	Kafka topic (JSON messages)
Output	Iceberg table in object storage

Property	Value
Labels	`DataOnboardingSink`, `IcebergEnabled`
Type	Sink

Configuration Parameters:

Parameter	Description	Default	Required
`CATALOG_DATASET_PATH`	Path to the dataset in the catalog	*(empty)*	Yes
`OAUTH_CLIENT_ID`	Client ID of the OAuth2 client to use for internal communications	*(empty)*	Yes
`OAUTH_CLIENT_SECRET`	Client secret of the OAuth2 client to use for internal communications	*(empty)*	Yes
`LOG_LEVEL`	Log level (DEBUG, INFO, WARN, ERROR)	`INFO`	No
`ICEBERG_CATALOG_REST_HOST`	Iceberg catalog REST host	`http://df-lakekeeper`	No

Use Cases:

- Building a data lakehouse for historical analytics
- Enabling time-travel queries on streaming data
- Creating queryable archives of data pipeline outputs
- Supporting BI tools and data science workflows

Data Processing Transformers

These transformers filter, enrich, or otherwise process data streams.

Kafka JSON KSQL Filter

Filters JSON messages based on a KSQL query for data quality and routing.

Property	Value
UID	`urn:rdp:transformer:json-ksql-filter`
Input	Kafka topic (JSON messages)
Output	Kafka topic (filtered JSON messages)
Labels	`DataProcessing`, `StreamProcessing`

Configuration Parameters:

Parameter	Description	Default	Required
`PROCESSOR_CONFIG_SQL`	The KSQL query to use for filtering messages	`SELECT * FROM d WHERE d.title LIKE '%Demo%'`	Yes

Use Cases:

- Filtering messages based on field values
- Routing messages to different topics based on content
- Data quality checks and validation
- Removing unwanted or malformed messages

Example Queries:

- `SELECT * FROM d WHERE d.priority > 5` - Filter by priority
- `SELECT * FROM d WHERE d.type = 'alert'` - Filter by type
- `SELECT d.id, d.timestamp FROM d WHERE d.status = 'active'` - Project specific fields

Dynamic Transformers

Dynamic transformers allow users to define custom transformation logic using JavaScript, Python, or Rhai scripting languages.

Custom JavaScript Function

Runs a dynamic JavaScript snippet on each message from Kafka.

Property	Value
UID	`urn:rdp:transformer:dynamic-js`
Input	Kafka topic (JSON messages)
Output	Kafka topic (transformed JSON messages)
Labels	`Dynamic`

Configuration Parameters:

Parameter	Description	Default	Required
`script` (file)	JavaScript file containing the handler function	Default passthrough script	Yes

Default Script:

```
const handler = (obj) => {
    out = {};
    out.input_msg = obj;
    out.version = '1.0';
    return out
}
```

Use Cases:

- Custom data transformations
- Field renaming and restructuring
- Enrichment with computed values
- Lightweight processing logic

For more information on dynamic transformers, including Python and Rhai variants, see Dynamic Transformers.

See Also

- Transformer Guide - How to create and register transformers
- Dynamic Transformers - Dynamic transformer details
- Transformer `configuration` - Transformer configuration guide
- Inputs and Outputs - Input/output connection types

Dynamic Transformers

Given that a Transformer can accept user configuration at runtime, the possibility exists for truly dynamic transformations whose logic is determined at runtime. Currently, **SDL** supports the following dynamic Transformers:

1. JavaScript
2. Python 3

These Transformers may or may not be present on your environment at startup (depending on the set of Transformers approved for your environment), but if approved they can be loaded easily.

How to use them

The specific usage differs slightly for each dynamic Transformer, but in general the user is responsible for providing the transformation logic, while the platform handles transport logic (i.e. consuming and producing data via the various storage media).

Dynamic JavaScript Transformer

The user is responsible for providing a function of the form:

```
const handler = (obj) => {
    return out
}
```

Where `obj` represents the input object. Note that this Transformer only runs on JSON records passing through Kafka.

Where `input` represents an incoming JSON record, and `output` will be unmarshalled as JSON and

sent to Kafka.

Dynamic Python Transformer

This Transformer is slightly more complex; the user is responsible for providing a runnable Python 3 file (with `main()`). The `df-daft-py` client library is embedded to simplify setting up the transport logic. See the `df-daft-py` documentation for more information on how to use this library. Below is an example of a transformation script:

```
# Transforms messages from one Kafka topic to another

import os

from df_daft_py.kafka.kafka import start_transform_loop


def transform(data) -> dict:
    # Make transformations here, for example:
    # data["latitude"] = 0.12345
    return data


def main():
    src_topic = os.getenv("SOURCE_KAFKA")
    dest_topic = os.getenv("DEST_KAFKA")
    start_transform_loop(src_topic, dest_topic, transform)
```

Additionally, users may specify a `requirements.txt` file configuration, which will be installed before the script is run. Note that on airgapped environments, this functionality may not work correctly unless a suitable mirror repository is available.

Transformer Guide

This section contains documentation on creating a JSON spec for a Transformer so that it can be registered with **SDL** Data Pipelines.

There are a few options for creating new Transformers.

Using Dynamic Transformers

Some Transformers that are shipped out-of-the-box with **SDL** allow users to configure them when creating a Pipeline. You can use the Pipeline UI to edit the transformation code directly, instead of building and pushing a new Transformer yourself. For many use cases, this is the easiest option and gets users started quickly. See Dynamic Transformers for more information.

Creating a New Transformer

Generally, writing a new Transformer will consist of the following steps:

1. Write the Transformer code, containerize it, and push the image.
2. Set environment variables to match those given below.
3. Write the Transformer JSON configuration, and add it to the pipeline engine.

The following pages will explain guidelines behind how to create Transformers, as well as how to create a Transformer JSON to register it.

Transformer Basics

Each transformer is a function of:

- Code (container, Flink SQL, etc.)
- Inputs and Outputs (Kafka Topics, etc.)
- Configuration supplied by the pipeline engine
- Configuration supplied by clients when instantiating it as part of a pipeline

A transformer can be defined to take any number of inputs or outputs, where each input/output can be a Kafka Topic path on MinIO, or direct connection to another Transformer. For Kubernetes-based transformers (i.e. containers), environment variables are the main way dynamic configuration is provided. Input and output dataset references (i.e. Kafka Topic to pub/sub, MinIO file path) will be provided by the pipeline engine as environment variables, though their names are controlled by the transformer's JSON configuration.

Writing the Transformer Template JSON is how the pipeline engine maps a client's request to something running. A breakdown of the various fields is as follows:

```
{
    "uid": "649152b4-ac10-4e2d-9a58-f3ec00d6d1c1",
    "name": "My Transformer",
    "description": "Transforms data",
    "status": "available",
    "security_markings": "UNCLASSIFIED",
    "types": ["sink"],
    "inputs": {
        "SOURCE_TOPIC": {
            "display_name": "Source Topic",
            "conn_type": "INTERNAL_KAFKA",
            "arity": {
                "min": 1,
                "max": 1
            }
        }
    },
    "outputs": {
        "DEST_TOPIC": {
            "display_name": "Destination Topic",
            "conn_type": "INTERNAL_KAFKA",
            "arity": {
```

```
                "min": 0,
                "max": 1
            }
        }
    },
    "configuration": {
        "environment": [
            {
                "name": "MY_VAR_1",
                "description": "Something useful for UIs",
                "default_value": "83412"
            },
            {
                "name": "MY_VAR_2",
                "description": "A required variable",
                "required": true
            },
            {
                "name": "MY_SECRET",
                "description": "A sensitive variable",
                "sensitive": true
            }
        ],
        "static_environment": [
            {"name": "STATIC_VAR_1", "value": "sdl-backend"},
            {"name": "KEYCLOAK_CLIENT_SECRET", "valueFrom": {"secretKeyRef": {"key":
"rdpPlatformClientSecret", "name": "keycloak-realm-init"}}}
        ],
        "engine_provided_environment": [
            "KEYCLOAK_REALM_URL",
            "KEYCLOAK_URL",
            "KEYCLOAK_REALM"
        ],
        "environment_mapping": {
            "KEYCLOAK_REALM_URL": "MY_CUSTOM_REALM_URL"
        }
    },
    "instantiation": {
        "job_image": {
            "image": "ghcr.io/raft-tech/my-transformer:v1.0",
            "pull_policy": "IfNotPresent",
            "image_pull_secret": "regcred-default",
            "default_replicas": 1,
            "args": ["--arg", "val"]
        }
    }
}
```

The key sections are:

uid

Unique identifier for the transformer. If one is not specified, a random UUID will be generated and assigned.

name, description, status

Basic metadata shown in the UI.

security_markings

Classification level applied to this template (e.g. `UNCLASSIFIED`, `SECRET`).

types

Array of transformer types. Options: `sink`, `source`, `ai_agent`, `ai_svc`, `rdp_workflow`, `geoserver_client`.

inputs / outputs

Named data connections. Each key becomes an environment variable name for container-based transformers. `conn_type` specifies the storage medium (e.g. `INTERNAL_KAFKA`). `arity` controls how many connections are valid (`min`/`max`).

configuration.environment

Array of environment variables that clients fill in when creating a pipeline. Each can have a `default_value`, be marked `required`, or flagged `sensitive`.

configuration.static_environment

Environment set as-is on the container. Supports both literal values and Kubernetes secret references (corev1.EnvVar format).

configuration.engine_provided_environment

Environment variable names populated automatically by the pipeline engine (e.g. Keycloak URLs).

configuration.environment_mapping

Renames engine-provided variables to custom names expected by your code.

instantiation.job_image

Container image configuration — image name, pull policy, pull secret, default replica count, and optional args.

Environment Variable Contract

Kubernetes transformers need to be written adhering to the contract dictated by the pipeline engine w.r.t environment variables. There are several categories of environment variables provided to transformers, which is as follows:

[Kafka Transformers] Kafka Environment

For transformers with at least **one** `input` or `output` whose `conn_type` is `INTERNAL_KAFKA`, the following environment will be provided:

- `KAFKA_BROKER_HOST`: The Kafka endpoint to use.
- `KAFKA_SASL_MECHANISM`: SASL Mechanism to use. Usually `SCRAM-SHA-512`
- `KAFKA_SASL_USERNAME`: SASL Username
- `KAFKA_SASL_PASSWORD`: SASL Password
- `KAFKA_GROUP_ID`: Consumer Group ID to be used. This allows for horizontal scaling of Kubernetes Jobs if a transformer's `replicas` is set greater than 1.
- `KAFKA_SECURITY_PROTOCOL`: Security protocol used. Usually `SASL_PLAINTEXT`.

[MinIO Transformers] MinIO Environment

For transformers with at least **one** `input` or `output` whose `conn_type` is `INTERNAL_MINIO`, the following environment will be provided:

- `MINIO_ENDPOINT`: MinIO Endpoint
- `MINIO_ACCESS_KEY`: MinIO username
- `MINIO_SECRET_KEY`: MinIO password

[Iceberg Transformers] Iceberg Environment

For transformers with at least **one** `input` or `output` whose `conn_type` is `INTERNAL_ICEBERG`, the following environment will be provided (in addition to the MinIO environment above):

- `ICEBERG_CATALOG_REST_HOST`: The Iceberg REST catalog endpoint (e.g. Lakekeeper)
- `ICEBERG_CATALOG_REST_CATALOG_PATH`: The catalog path on the REST endpoint

[Dataset Transformers] Dataset Environment

For transformers with `conn_type` of `DATASET`, the engine resolves the dataset reference from the catalog and downcasts it to the underlying resource type (e.g. `INTERNAL_KAFKA`, `INTERNAL_MINIO`). The environment variables provided will match those of the resolved type.

[All Transformers] Prometheus Environment

These variables are provided to enable transformers to emit prometheus metrics. A `PodMonitor` is deployed in **SDL** that will automatically scrape the port and path specified by these variables:

- `PROM_METRICS_NS`: Prometheus metric namespace to use. This isn't strictly checked but should be used to help separate transformers' metric names. See [here](https://prometheus.io/docs/practices/naming/) for more details.
- `PROM_METRICS_PORT`: The port at which to start the Prometheus server at. The pipeline engine will appropriately expose this port on the Job/Pod.
- `PROM_METRICS_ROUTE`: The route at which the Prometheus server should serve metrics on. This is what Prometheus is configured to look for via a `PodMonitor`.

[All Transformers] Contextual Environment

These variables provide transformers with some context about the environment they're running in when instantiated.

- `PIPELINE_UID`: The UID of the pipeline instance that this Transformer is part of
- `PIPELINE_NAME`: The name of the pipeline instance that this Transformer is part of
- `PIPELINE_TRANSFORMER_UID`: The UID corresponding to the Transformer template that was used to instantiate this Transformer.
- `PIPELINE_TRANSFORMER_NAME`: The instantiated name of this Transformer (i.e. the Kubernetes Job name).

[Direct Transformer <→ Transformer] Direct Environment

Used specifically for connections of type `TRANSFORMER`. The configuration contract is defined specifically between two transformers; no default configuration is provided by the data pipeline engine. See Transformer Connections for more information.

Transformer `configuration`

In addition to `inputs` and `outputs`, a Transformer can also have additional `configuration` options assigned to it. They break down into a few categories, which will be explained further:

- `environment`: Configuration options that can be set by users when submitting a Pipeline that uses this Transformer.
- `static_environment`: Environment variables that are set to a specific value; unmodifiable at submission time.
- `environment_mapping`: A special block that allows environment variables to be renamed - this is helpful when bringing an existing application in as a Transformer.
- `engine_provided_environment`: Special environment variables that are only known by the Pipeline Engine when instantiating a Pipeline.
- `file_configuration`: Templatized configurations for files, as opposed to environment variables.
- `properties`: Configuration properties for `rdp_workflow` type transformers that may be static or supplied by the user at runtime.

`environment`

Any transformer can have arbitrary environment variables provided by the client when a pipeline is executed.

An `environment` configuration has the following fields:

- `name`: The name of the environment variable
- `description`: A useful description that aids in documentation.
- `required`: A boolean value indicating whether or not this environment variable has to be included in a Pipeline submission to be valid.

- `default_value`: If unset in a Pipeline submission, what value should be used instead.
- `sensitive`: A boolean value indicating whether this variable contains sensitive information (e.g. secrets or credentials).

NOTE:

Example

```
    "environment": [
        {
          "name": "SSH_SERVER",
          "description": "The SSH server to communicate with",
          "required": true
        },
        {
          "name": "SSH_USE_PASSWORD_AUTH",
          "description": "Boolean value to allow username/password ssh
authentication",
          "default_value": "false"
        },
        {
          "name": "FILTERS",
          "description": "List of filters to use"
        }
    ]
```

When a Pipeline is submitted using this Transformer, the following environment variables will be set:

- `SSH_SERVER` must be provided by the user who submits the Pipeline, or else it'll return an error.
- `SSH_USE_PASSWORD_AUTH` is optional, but if not set by the user who submits the Pipeline then `"false"` will be passed.
- `FILTERS` is optional, but if not set by the user who submits the Pipeline then no value will be set (and `FILTERS` will not be provided in the Transformer's environment).

`static_environment`

Any transformer can have arbitrary environment variables set by the pipeline engine when a pipeline is executed. To add these, refer to Note that these environment variables are **NOT** settable by users. Each item in `static_environment` is a Kubernetes `corev1.EnvVar`. In essence, that means it can either be:

```
{ "name": "", "value": ""}
```

Or take a more complex form to pull from existing `Secret`s or `ConfigMap`s:

```
{ "name": "", "valueFrom": {"secretKeyRef": {"name": "", "key": ""}}}
```

Example

```
"static_environment": [
    {
      "name": "FLINK_JOB_STATUS_RETRIES",
      "value": "3"
    },
    {
      "name": "FLINK_JOB_STATUS_POLL_MS",
      "value": "30000"
    }
]
```

In the example above, each time this Transformer is instantiated as part of a Pipeline, `FLINK_JOB_STATUS_RETRIES` and `FLINK_JOB_STATUS_POLL_MS` will both be set to the static values.

`engine_provided_environment`

Some configurations are only known by the pipeline engine when instantiating a pipeline, which presents a problem when configuring a Transformer generically. Any transformer can optionally specify one of several environment variables that will be dynamically set by the pipeline engine. These are as follows:

- `KEYCLOAK_REALM_URL`: URL to the Keycloak realm, of the form `http(s)://<keycloak base url>/realms/<realm>`. Read from the backend container's environment.
- `KEYCLOAK_URL`: The base URL of Keycloak, of the form `http(s)://<keycloak base url>`. Read from the backend container's environment.
- `KEYCLOAK_REALM`: The realm used by **SDL** services. Read from the backend container's environment.
- `KEYCLOAK_CERTS_URI`: URL to the platform's Keycloak `openid-connect/certs` route.
- `HOSTNAME`: The hostname of the Transformer as it is instantiated by the pipeline engine.

Example

```
"engine_provided_environment": [
    "HOSTNAME",
    "KEYCLOAK_REALM"
]
```

In the example above, when this Transformer is instantiated as part of a Pipeline, the Pipeline Engine will dynamically set the `HOSTNAME` and `KEYCLOAK_REALM` environment variables, which only it knows.

`environment_mapping`

Sometimes, Transformers use libraries or frameworks that assume environment that configuration will be a particular way. `boto3` in Python and Spring Boot in Java are good examples. In order to reduce the number of code changes required to bring an external application into the **SDL** Pipeline Engine, such as re-mapping environment variables to match a naming scheme, an optional `environment_mapping` configuration can be specified in a Transformer's contract.

Example

```
// ...
// Each mapping is <source name> -> <destination name>
"configuration": {
  "environment": [],
  "engine_provided_environment": [],
  "environment_mapping": {
    "MINIO_ACCESS_KEY": "AWS_ACCESS_KEY_ID",
    "MINIO_SECRET_KEY": "AWS_SECRET_ACCESS_KEY",
    "MINIO_ENDPOINT": "AWS_ENDPOINT_URL"
  }
}
// ...
```

The standard environment variables provided by a MinIO connection will be mapped to new names when the Transformer is instantiated. Note that these mappings can use *any* environment variables that are provided to the Transformer. The mapping is performed `source → destination`, meaning at most an environment variable is mapped a single time. The decision to go `source → destination` instead of the reverse was made to reduce environment variable bloat, i.e. multiple environment variables in a running Transformer with the exact same value. If doing so is desired (for example, if using multiple libraries who each assume different names for an environment variable containing a common configuration), then it is the Transformer author's responsibility to perform any additional mapping upon startup.

When an environment variable is mapped into a new name, the **previous variable will not be supplied to the Transformer**. In the example above, the Transformer will **not** see `MINIO_ACCESS_KEY`, `MINIO_SECRET_KEY`, or `MINIO_ENDPOINT` in its environment.

`file_configuration`

While the default way most transformers are configured is via environment (especially in the case of Kubernetes-based apps), environment variables aren't suitable for every use case. The pipeline engine also supports configuration via files whose contents are passed to the Transformer at runtime. In the case of a Kubernetes-based Transformer, file contents are stored in a `Secret` and volume-mounted into the Transformer when it is created. The model for providing file configuration is meant to be as flexible as possible while not allowing users to mount files in arbitrary locations in a Transformer, primarily for security reasons. For that reason, all file configuration must be specified in a Transformer's template when it is created by a Transformer author, similar to how environment configuration is done. Both the values that users provide for

file configuration and their default values as provided in a Transformer template are base64-encoded strings, which allows users to pass newlines without breaking JSON syntax.

A Transformer template with file configuration would add a `file_configuration` block under the main `configuration` block.

Example

```
"configuration": {
  // ...
  "file_configuration": [
    {
      "name": "config-1",
      "path": "/config/config-1.yaml", // mounted at /config
      "default_value_base64": "SGVsbG8gZnJvbSBjb25maWctMQo="
    },
    {
      "name": "config-2",
      "path": "/config/config-2.yaml", // mounted at /config (sharing Volume with
config-1)
      "default_value_base64": "SGVsbG8gZnJvbSBjb25maWctMgo="
    },
    {
      "name": "config-3",
      "path": "config-3.yaml", // mounted at /opt/transformer-configuration
      "default_value_base64": "SGVsbG8gZnJvbSBjb25maWctMwo="
    },
    {
      "name": "config #4",
      "path": "config-4.yaml", // mounted at /opt/transformer-configuration (sharing
Volume with config-3)
      "required": true // The user MUST provide a value (similarly to environment)
    }
  ]
  // ...
}
```

`file_configuration` is a list of objects, each containing a single file to be configurable by a user when submitting a pipeline. Each item of `file_configuration` has the following fields:

- `name`: A human-readable name for the configuration. Does not need to be a filename and can include spaces. When submitting pipelines, users will use this field to tell the engine which configuration they're supplying.
- `description`: A description of the file configuration.
- `path`: The path in the Transformer where this configuration will be mounted. If `path` is absolute, it functions as expected. In Kubernetes Transformers, if `path` is relative, the configuration is stored at `/opt/transformer-configuration`.
- `default_value_base64`: If the user doesn't supply a value, this will be provided.

- `required`: Whether or not a pipeline submission should fail if the user does not provide a value. Note that this field has no meaning if `default_value_base64` is also supplied.

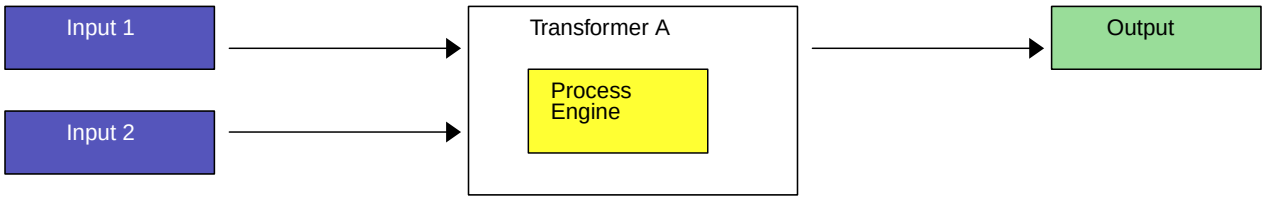
The pipeline engine does not allow user-defined mount paths, as that would impact security. It also does not allow variadic arguments, i.e. defining a directory where the user can provide as many configuration files as they want. This use case is best served by storing those files on MinIO/S3 and loading them into the Transformer as a dataset.

When submitting a pipeline, a user would provide file configurations in a similar manner to `environment`, as a JSON map:

```
"configuration": {
  // ...
  "file_configuration": {
    "config #4": { // Corresponds to 'name'
      "base64_encoded_text": "T3ZlcnJpZGVuIGJ5IHVzZXIgYXQgc3VibWlzc2lvbiB0aW1lIQo="
    }
  }
  // ...
}
```

Inputs and Outputs

The `inputs` and `outputs` sections define connections the transformer will require. In the case of Kubernetes Jobs, each input or output is mapped to an environment variable whose value is provided by the client. Think of these as the arrows pointing in and out of the transformer in a drag-and-drop UI:



Given that the data pipeline engine is dynamic by design, the exact references to inputs/outputs (i.e. Kafka Topics, locations on S3, etc.) shouldn't be known ahead of time (otherwise, what would connecting them to other components mean?).

Thus, inputs/outputs are *always* dynamically configured at runtime. Statically setting configuration can be achieved by (in the case of a K8s app) adding to `configuration.static_environment`.

Inputs and outputs are defined as a map where each key is the connection name (which becomes an environment variable in the transformer):

```
"inputs": {
  "KAFKA_TOPIC_NAME": {
    "display_name": "Destination Topic",
    "conn_type": "INTERNAL_KAFKA",
    "arity": { "min": 1, "max": 1 }
  }
}
```

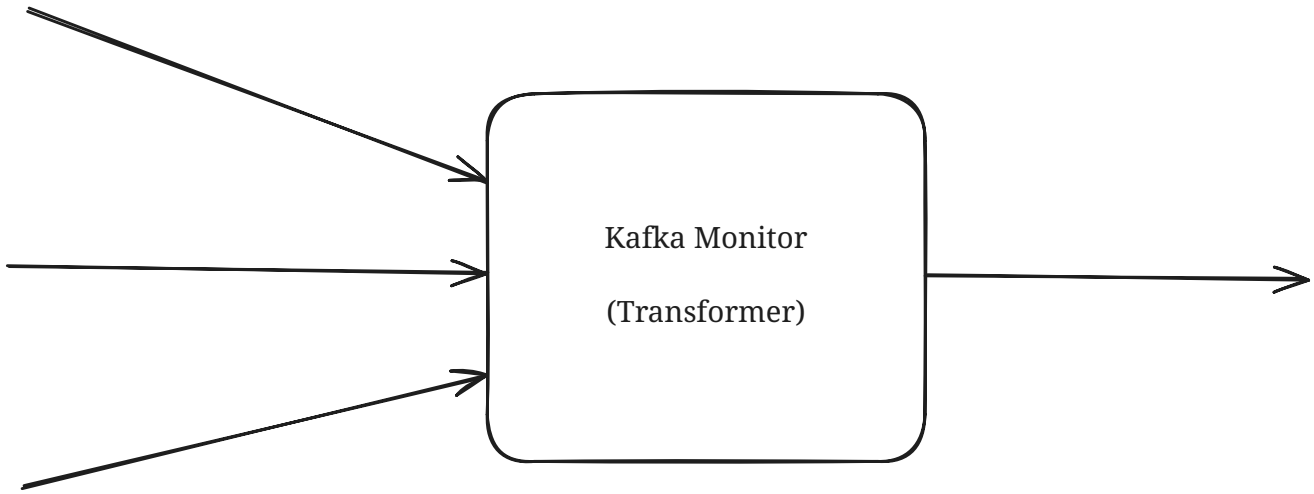
`conn_type` can be one of these values:

- `INTERNAL_KAFKA` - inputs/outputs will map to a Kafka Topic
- `INTERNAL_MINIO` - inputs/outputs will map to a path on MinIO, i.e `s3://abc`
- `INTERNAL_ICEBERG` - inputs/outputs will map to an Iceberg table
- `DATASET` - inputs/outputs reference a dataset registered in the catalog
- `TRANSFORMER` - inputs/outputs are **not** mapped to an underlying data storage driver; instead, communication between two transformers is handled **directly** by them
- `RDP_WORKFLOW` - inputs/outputs are handled by the SDL workflow runtime. Connections of this type must define `workflow_accepted_types`, which specifies the data types the connection can accept (e.g. `string`, `number`, `double`, `boolean`, `map`, `array`)

Connection Arity

Think about a Transformer whose purpose is to monitor a series of Kafka topics for anomalous messages and send an alert somewhere. With the Transformer specification as discussed thus far, a `Kafka Monitor` Transformer would only be able to monitor a set number of Kafka Topics, as each input and output connection must be known when registering the Transformer. The only way to dynamically adjust a Pipeline to new inputs and outputs is to deploy additional replicas.

Visually, this is what the Transformer might look like:



Modeling Transformers as functions that operate on connections, then we'd want to have something akin to this in Python:

```
def monitor_kafka(*args):
  pass
```

Or this in Go:

```
func monitorKafka(inputConnections ...string) {}
```

These are referred to as *variadic* arguments, where the exact number is not specified in the function signature. The number of arguments specified in a function signature is referred to as its *arity*. That same concept can be applied to Data Pipelines.

The pipeline engine supports arity for all inputs and outputs. It takes the form of an optional field declared for each input and output connection in a Transformer template. It's presence tells the pipeline engine how to validate and resolve one or more matching instances of that connection in a pipeline submission. This allows a Transformer to accept optional, discrete arity, or variadic connections.

Arity is added inside a connection:

```
"inputs": {
  "KAFKA_TOPIC_NAME": {
    "conn_type": "INTERNAL_KAFKA",
    "arity": {
      "min": 0,
      "max": 1
    }
  }
}
```

`arity` is a range-inclusive `min`/`max` processed by the Pipeline Engine to allow matching of multiple connections in a Pipeline to a single input or output in a Transformer template. Importantly, `arity` enables three key improvements to Transformer templates:

1. The ability to declare an input or output as optional; if a matching connection isn't provided in a Pipeline submission, the Pipeline is still valid.
2. The ability to map a discrete number of connections to an input or output.
3. The ability to use variadic connections, where the number of connections that may map to a single connection template are not specified by the Transformer template.

The various types of arity are discussed below.

Default Arity

By default, any Transformer template with connections that **do not specify** `arity` will retain the default, backwards-compatible behavior: the given connection must be provided by any Pipeline submission, and it must be a single value. Additionally, the following `arity` blocks are interpreted

by the pipeline engine as default cases:

```
"arity":  {"min": 1, "max": 1}
```

```
"arity":  {"min": 0, "max": 0}
```

Optional Arity

With the default arity, all connection templates must be provided a value for a Pipeline submission to be considered valid. With an optional `arity`, if no matching connection is found the pipeline engine moves on:

```
"arity":  {"min": 0, "max": 1}
```

If an optional connection is not provided in a Pipeline submission, the pipeline engine **does not set that configuration (ex. that environment variable)**. Transformer writers should be careful to not depend on environment variables being present if using optional connections.

Discrete Arity

Discrete `arity` allows for a range of possible connections to be matched to a connection template. For any connection template whose `arity.max` is greater than `1` or less than `0` (more about that below), the pipeline engine **does not** match the connection name directly when configuring the Transformer, as it does for the default behavior. Instead, the pipeline looks for connections in the Pipeline submission of the form

```
<connection_name>_{0-9}+
```

i.e. the source connection name followed by an underscore (`_`) and a number. This is flexible enough for any number of matching connections to be passed in a Pipeline submission. To illustrate this, take the following connection template from a Transformer template:

```
"outputs": {
  "DEST": {
    "conn_type": "INTERNAL_KAFKA",
    "arity": {
      "min": 1,
      "max": 3
    }
  }
}
```

This tells the pipeline engine to expect from 1 to 3 matching connections of the form `DEST_{0-9}+`,

and that they should all resolve to the underlying `conn_type` of `INTERNAL_KAFKA`. A valid Pipeline submission might look like:

```
"outputs": {
  "DEST_1": {"conn_type": "DATASET"},
  "DEST_2": {"conn_type": "INTERNAL_KAFKA"},
  "DEST_4": {"conn_type": "DATASET"}
}
```

While numeric suffixes are validated, they don't need to be contiguous *or* start at `1`, as demonstrated above by `DEST_4`. Transformer writers should make sure their app searches for environment according to the regular expression instead of depending on specific numeric indices.

The connection types each may have a different `conn_type`, as long as they all resolve to the correct underlying `conn_type` from the Transformer template. If an invalid number of connections match the connection template, then an error is returned by the pipeline engine prior to instantiating the Pipeline.

For any connection template with an arity greater than 1, the pipeline engine validates that its name does not conflict with any other connection names. For example, a Transformer template with the above output connection template with discrete arity **cannot** also have a connection template like:

```
"DEST_12345": {
  "conn_type": "INTERNAL_KAFKA"
}
```

The reason for this is that the pipeline engine would be unable to validate whether a connection named `DEST_12345` in a Pipeline submission should be counted as part of `DEST` or `DEST_12345`.

Variadic Arity

In this case, variadic arity refers to an unbounded maximum arity:

```
"arity":  {"min": 1, "max": -1}
```

Truly unbounded variadic arity would look like this, where *any number* of matching connections are permissible:

```
"arity":  {"min": 0, "max": -1}
```

The same naming conventions for connections passed in Pipeline submissions apply to variadic connections as for those with discrete arity.

INTERNAL_KAFKA Connections

These connections map to an underlying Kafka Topic; if it does not already exist, then a new topic will be created. Transformers connected via an input of this type will be provided configuration on how to connect and authenticate with Kafka. Users are able to set topic configuration by adding a `properties` block to the connection JSON when submitting a pipeline:

```
"INPUT_TOPIC": {
  "conn_type": "INTERNAL_KAFKA",
  "ref": "some-input-topic",
  "properties": {
    "partitions": "2", // reserved key
    "min.insync.replicas": "1", // passed directly as a Kafka config
    "replicas": "1" // reserved key
  }
}
```

There are currently two reserved keys in the `properties` block:

- `partitions`: If set, override the default number of partitions.
- `replicas`: If set, override the default replication factor.

All other `properties` will be passed through directly as Kafka configuration. See Apache Kafka Configuration for a list of topic configuration options.

By default, created topics have:

- 12 partitions
- 3 replicas

INTERNAL_MINIO Connections

These connections map to a location on S3. Technically, a bucket path (`s3://df-bucket`), key prefix (`s3://df-bucket/folder1/`) or file path (`s3://df-bucket/folder1/file1.txt`) can be provided. Validation for these variations may occur in the future; however, currently it is up to transformers to define handling for each case.

INTERNAL_ICEBERG Connections

These connections map to an Iceberg table managed by the Lakekeeper catalog. The reference format is `catalog.schema.table`. Transformers connected via an input or output of this type will be provided configuration for both the Iceberg REST catalog and MinIO (since Iceberg tables are backed by object storage).

When submitting a pipeline, the `ref` must be a three-part dotted string:

```
"ICEBERG_INPUT": {
  "conn_type": "INTERNAL_ICEBERG",
  "ref": "my_catalog.my_schema.my_table"
```

```
}
```

Each part of the reference (`catalog`, `schema`, `table`) must be non-empty. For example, `my_catalog.my_schema.my_table` is valid, but `my_catalog..my_table` is not.

Environment Provided

Transformers with at least one `INTERNAL_ICEBERG` connection will receive the following environment variables:

- `ICEBERG_CATALOG_REST_HOST` - The Iceberg REST catalog endpoint (e.g. Lakekeeper)
- `ICEBERG_CATALOG_REST_CATALOG_PATH` - The catalog path on the REST endpoint

In addition, the MinIO environment variables (`MINIO_ENDPOINT`, `MINIO_ACCESS_KEY`, `MINIO_SECRET_KEY`) are also provided, since Iceberg tables are backed by object storage.

TRANSFORMER Connections

The concept of an input/output essentially defines two things:

1. What kind of data a transformer expects to consume or produce
2. How to access that data

For many data pipelines, especially those with high volume and and low latency requirements, underlying storage methods are used to provide reliability, fault tolerance, and scalability. However, many potential transformers either don't need to or are unable to connect to these underlying drivers, instead exchanging data point-to-point. A contrived example would be a Kafka HTTP Tap, a hypothetical application that consumes from a Kafka topic/partition and allows clients to pull the latest message via an HTTP request. For these connections, it doesn't make sense to jam a 'dataset' in between two applications just to be consistent. `TRANSFORMER` type connections are one solution, allowing two transformers to exchange information about how to connect each other.

In this example, the Kafka HTTP Tap consumes from kafka and exposes an HTTP route to pull from, accepts a single input of type `INTERNAL_KAFKA`, and produces a single output of type `TRANSFORMER`:

```
// ...
"outputs": {
  "ROUTE": {
    "display_name": "Route to access data from",
    "conn_type": "TRANSFORMER",
    "conn_config": {
      "static_environment": [
        {
          "name": "ROUTE",
          "value": "/latest"
        }
      ]
    }
  }
```

```
  },
  // ...
```

The corresponding sink app periodically pulls from the HTTP route. It places constraints on the upstream `TRANSFORMER` connection in the form of what configuration it **must** provide for it to be accepted:

```
{
  // ...
  "inputs": {
    "SOURCE": {
      "conn_type": "TRANSFORMER",
      "required_conn_config": [
        "ROUTE",
        "HOSTNAME"
      ]
    }
  },
  // ...
}
```

As seen in the `outputs` block of the source app, both vars are provided: `ROUTE` from the upstream `conn_config` and `HOSTNAME` automatically-not specified in the upstream `conn_config`, rather coming from the engine itself (see the sections below for more details on what variables can be auto-supplied.).

When a client makes a data pipeline consisting of this and a (in this case) downstream transformer, they would provide a pipeline with these parts (some parts omitted for clarity):

```
{
    "pipeline": [
        {
            "id": 12345,
            "name": "Kafka HTTP Tap",
            "outputs": {
                "ROUTE": { // Nothing in this block is change-able by clients
                    ...
                }
            }
            ...
        },
        {
            "id": 67890,
            "name": "Downstream HTTP Client",
            "inputs": {
                "SOURCE": {
                    "conn_type": "TRANSFORMER",
                    "stage": 0,
```

```
                    "key": "ROUTE"
                }
            }
        }
    ]
}
```

Note the `stage` and `key` fields in the `inputs` block, in lieu of a `ref` as would normally be present. This tells the data pipeline engine to look for an object in `outputs` in the zeroth stage of the pipeline list named `ROUTE`.

The pipeline engine would then supply `Downstream HTTP Client` with the following environment:

```
SOURCE_HOSTNAME=<hostname of 'Kafka HTTP Tap' Service in Kubernetes> ①
SOURCE_ROUTE=/latest ②
```

① Engine-provided configuration consisting of the hostname of `Kafka HTTP Tap`, which is **only** known by the engine when instantiating the pipeline.

② **All** configuration contained in the `conn_config` block of the `ROUTE` output, as provided by the `Kafka HTTP Tap` template.

The *actual* configuration values are created by concatenating the input name and the configuration provided by the connection being used, separated by `(). Thus, SOURCE (input) +` + `HOSTNAME` (from `engine_provided_environment`). This is intended to prevent naming collisions when multiple inputs and/or outputs are specified.

Push- and Pull-based Transformers

`TRANSFORMER` connections can provide configuration **either** upstream or downstream; that is, an input of type TRANSFORMER that has `conn_config` will provide it to any Transformers with the corresponding output, and vice-versa.

In data-engineering terms, a Transformer that has an output of type `TRANSFORMER` that provides connection information via `conn_config` can be thought of as *pull-based*. A Transformer that has an input of type `TRANSFORMER` that provides connection information via `conn_config` can be thought of as *push-based*. 'Normal' Transformers that provide configuration for an underlying data storage solution like Kafka are *technically* both, as producers produce **and** consumers call `poll()` or `next()`.

Types of `TRANSFORMER conn_config`

Similarly to the `configuration` block of a Transformer, `conn_config` exposes a set of configurations available to provide to any Transformers that connect to this as one of their inputs or outputs.

environment

Currently, the data pipeline engine **DOES NOT** support providing user-settable environment for a connection of type `TRANSFORMER.

static_environment

These work as expected; after they're defined in the transformer's template, any Transformers that connect to that connection will be provided each of these.

engine_provided_environment

Configurations like `HOSTNAME` aren't known until the pipeline decides on what to name the particular stage when it's created (in Kubernetes, for example). Thus, they're specified here.

The full list of supported `engine_provided_environment` is:

- `HOSTNAME`: Specify this to direct the data pipeline engine to populate with the hostname of the Kubernetes `Service` pointing to this Transformer.

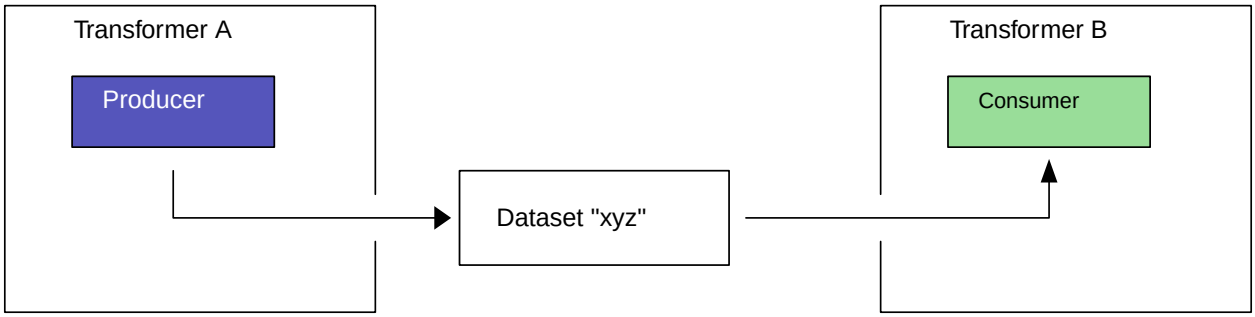
> Transformer writers exposing inputs/outputs of type `TRANSFORMER` are encouraged to make fields in `conn_config` as generically-named as possible.

DATASET Connections

If you're looking through the pipeline engine code or at the pipeline UI, you may see connections of type `DATASET`. They aren't mentioned in this README as this is meant for transformer authors, and no transformer will ever need to accept connections of type `DATASET`.

`DATASET` connections are an abstraction provided by the pipeline between itself and clients submitting pipelines. It allows clients to not fully specify the interconnecting drivers in between each pair of transformers, instead letting the pipeline engine infer them from the templates provided.

For example, if a pipeline like this would be submitted:



The pipeline engine would look at the transformer templates for `Transformer A` and `Transformer B` to determine if the input/output pair matches. If so, the input and output would be resolved down to the underlying type, and validation/instantiation of the pipeline would continue as normal.

For example, take the following Transformer template:

```
{
  "name": "My Transformer",
  "uid": "1234",
```

```
  "inputs": {
    "SOURCE_TOPIC": {
      "display_name": "Source topic",
      "conn_type": "INTERNAL_KAFKA",
    },
  },
  // ...
}
```

If a pipeline were created using this Transformer like so:

```
{
  "name": "My Pipeline",
  "latest": [
    {
      "uid": "1234",
      "name": "My Transformer",
      "inputs": {
        "SOURCE_TOPIC": {
          "conn_type": "DATASET", // <--------- Note the conn_type
          "ref": "src-topic", // <--------- Note the raw ref
        },
      },
      "outputs": {},
      "configuration": {},
    },
  ],
}
```

The pipeline engine will see the `conn_type` of `DATASET` and attempt to reconcile it with the Transformer template. In this case, the Transformer template specifies that the `SOURCE_TOPIC` input must be of type `INTERNAL_KAFKA`. The pipeline engine will then replace the `conn_type` and proceed. In the case of an `INTERNAL_KAFKA` connection, the pipeline engine will resolve the source `ref` by appending a UUID to it. In the case above, the resulting pipeline would look like this:

```
{
  "name": "My Pipeline",
  "latest": [
    {
      "uid": "1234",
      "name": "My Transformer",
      "inputs": {
        "SOURCE_TOPIC": {
          "conn_type": "DATASET", // <--------- Note the conn_type is INTERNAL_KAFKA
          "ref": "src-topic-7fb2d3a6-57cc-48b2-b7dc-cb06f628b71a", // <--------- Note
the ref has been salted
        },
      },
```

```
      "outputs": {},
      "configuration": {},
    },
  ],
}
```

In the case of an underlying type of `INTERNAL_MINIO`, the pipeline engine **will not salt the `ref`**. Instead, it will prepend an S3 storage prefix. An incoming `DataConn` looking like:

```
"SOURCE_S3_PATH": {
  "conn_type": "DATASET",     // <--------- Note the conn_type
  "ref": "src-bucket/prefix1" // <--------- Note the raw ref
}
```

Would become:

```
"SOURCE_S3_PATH": {
  "conn_type": "INTERNAL_MINIO",   // <--------- Note the conn_type
  "ref": "s3://src-bucket/prefix1" // <--------- Note the expanded ref
}
```

A `conn_type` of `DATASET` is **transient**; it will never be persisted by the pipeline engine, as it will **always** be resolved into a concrete `conn_type` first.

RDP_WORKFLOW Connections

These connections are used by transformers with type `rdp_workflow` and are instantiated with the SDL workflow runtime instead of a Kubernetes Job (like all other transformer types — see Instantiation). Unlike `TRANSFORMER` connections (which use environment variables for direct communication between containers), `RDP_WORKFLOW` connections pass typed data between workflow nodes.

workflow_accepted_types

Every `RDP_WORKFLOW` connection **must** define `workflow_accepted_types`, which specifies what data types the connection can accept. Valid types are `string`, `number`, `double`, `boolean`, `map`, and `array`. The `map` and `array` types can be nested (e.g. an array of maps).

```
"inputs": {
  "WORKFLOW_INPUT": {
    "conn_type": "RDP_WORKFLOW",
    "display_name": "Workflow Input",
    "workflow_accepted_types": [
      { "type": "string" },
      { "type": "array", "items": { "type": "map" } }
    ]
  }
```

```
}
```

Connection Validation

When two `RDP_WORKFLOW` transformers are connected, the engine validates that there is at least one overlapping type between the source output's `workflow_accepted_types` and the target input's `workflow_accepted_types`. If there is no overlap, the pipeline will fail validation.

`RDP_WORKFLOW` connections cannot be mixed with `TRANSFORMER` connections. Both ends of a transformer-to-transformer connection must be the same type.

Instantiation

The `instantiation` block describes how a Transformer is created by the Pipeline Engine. Two options are available: `job_image` (instantiate as a Kubernetes Job) and `rdp_workflow` (instantiate via the SDL workflow runtime).

`job_image`

`job_image` tells the Pipeline Engine to instantiate the Transformer as a Kubernetes `Job`. Available fields are as follows:

```
"job_image": {
    "image": "asdf",
    "pull_policy": "IfNotPresent", // Alternatives: Always, Never
    "image_pull_secret": "pat-name",
    "default_replicas": 2,
    "ports": [
        {
            "name": "http-port",
            "containerPort": 8080
        }
    ],
    "volumes": [ // A list of Kubernetes Volumes, just in JSON format instead of YAML
        {
            "name": "volume-1",
            "configMap": {
                "name": "volume-configmap",
                "items": {
                    "key": "config",
                    "path": "my-config.conf"
                }
            }
        }
    ],
    "volume_mounts": [ // A list of Kubernetes Volume Mounts, just in JSON format
instead of YAML
        {
            "name": "volume-1",
```

```
            "mountPath": "/etc/config"
        }
    ],
    "args": [ // Alternative arguments passed to the main container at startup
        "tail", "-f", "/dev/null"
    ]
}
```

`rdp_workflow`

`rdp_workflow` tells the Pipeline Engine to instantiate the Transformer via the SDL workflow runtime in a one-shot configuration. This is used for transformers with the `rdp_workflow` type.

```
"rdp_workflow": {
    "definition_id": "my-custom-definition-id"
}
```

`definition_id`

Optional. If present, overrides the Transformer template's UID when searching for the appropriate node definition in the workflow runtime. If omitted, the template's own UID is used.

Examples

This section contains a few example Transformer configurations to make things a little clearer. Note that source code for the Transformers are not provided; the application's functionality is described in relation to the available configuration options.

Basic Example

This is a very basic Transformer that does a single thing: forwards messages from one Kafka topic to another. It needs a few configuration options to run:

1. Configuration to connect to Kafka
2. The input Kafka topic
3. The output Kafka topic
4. An optional setting for Kafka compression

Also assume that a container has been built and pushed to a registry the pipeline engine can read from.

This is what a configuration JSON for this Transformer might look like:

```
{
  "name": "Kafka Topic Forwarder",
  "description": "Forwards one Kafka Topic to another",
  "status": "available",
  "types": [],
```

```
  "inputs": {
    "SOURCE_TOPIC": {
      "display_name": "Source topic",
      "conn_type": "INTERNAL_KAFKA"
    }
  },
  "outputs": {
    "DEST_TOPIC": {
      "display_name": "Destination topic",
      "conn_type": "INTERNAL_KAFKA"
    }
  },
  "configuration": {
    "environment": [
      {
        "name": "KAFKA_COMPRESSION_TYPE",
        "description": "Kafka compression to use when publishing",
        "default_value": "none"
      }
    ],
    "static_environment": []
  },
  "instantiation": {
    "job_image": {
      "image": "topic-forwarder-image",
      "pull_policy": "Always",
      "image_pull_secret": "my-registry-secret",
      "default_replicas": 1
    }
  }
}
```

Explanation

Taking the configuration block by block:

```
{
  "name": "Kafka Topic Forwarder",
  "description": "Forwards one Kafka Topic to another",
  "status": "available",
  "types": [],
}
```

These options are fairly self explanatory. `name` and `description` help to describe the Transformer once it's registered in the Pipeline Engine. Another optional field `uid` can be specified and serves as a globally unique identifier for the Transformer; this allows you to have multiple Transformers with the same `name`. `status` should usually be set to `available` so the Transformer can be used in Pipelines. The `types` field holds special tags that represent the transformer type (e.g. `sink`, `source`, `rdp_workflow`, etc). None of these types apply to this transformer, so the `types` field is empty.

```
{
  "inputs": {
    "SOURCE_TOPIC": {
      "display_name": "Source topic",
      "conn_type": "INTERNAL_KAFKA"
    }
  },
}
```

This block defines a single input connection. Its name as it will show up in the Transformer's environment is `SOURCE_TOPIC`. `display_name` is helpful if using the Pipeline UI. We know from `conn_type` that it needs to be of type `INTERNAL_KAFKA`. For more information on inputs and outputs, see Inputs and Outputs

```
{
  "outputs": {
    "DEST_TOPIC": {
      "display_name": "Destination topic",
      "conn_type": "INTERNAL_KAFKA"
    }
  },
}
```

Very similar to `inputs` - we declare a single output named `DEST_TOPIC`. Note that since no `arity` is defined, we assume the default (this output is **required** and must be provided in a Pipeline for it to be valid). For more information on `arity`, see Connection Arity.

```
{
  "configuration": {
    "environment": [
      {
        "name": "KAFKA_COMPRESSION_TYPE",
        "description": "Kafka compression to use when publishing",
        "default_value": "none"
      }
    ],
    "static_environment": []
  },
}
```

This section defines any additional configuration options that should be supplied to this Transformer.

```
{
  "instantiation": {
    "job_image": {
```

```
      "image": "topic-forwarder-image",
      "pull_policy": "Always",
      "image_pull_secret": "my-registry-secret",
      "default_replicas": 1
    }
  }
}
```

The last block identifies **how** this Transformer should be instantiated by the Pipeline Engine. As additional means of instantiating Transformers become available, new options under `instantiation` will become available. For more information, see Instantiation. In this example, a container has been pushed that the Pipeline Engine can refer to. It will be deployed as a Kubernetes `Job` and will have a default number of replicas of 1.

SDL SDK

SDL can be interfaced with in multiple ways: service integration, via REST APIs, or SDKs. SDKs allow users to interact with an instance of SDL without needing to write an HTTP client themselves.

Currently, SDL generates SDKs for the following languages:

- `Python 3`
- `Java`
- `Go v1.23` or later

How to get the SDK

From your SDL deployment (recommended)

Navigate to the "APIS" tab of your SDL UI. There is a button in the bottom-right corner that can be used to download the SDKs along with their documentation.

From Github

SDL SDKs are hosted at https://github.com/raft-tech/sdl-sdk.

The `Releases` page contains released artifacts with copies of the SDKs that can be downloaded and installed locally. Each release of the SDL SDK matches a version of the SDL platform, which ensures that the SDK doesn't drift when updates occur. Note that SDK documentation is also available in the same place.

Java SDK

Installation

The Java SDK is distributed as a JAR file. Install it to your local Maven repository:

```
mvn install:install-file \
  -Dfile=sdl-sdk-0.0.1.jar \
  -DgroupId=com.teamraft \
  -DartifactId=sdl-sdk \
  -Dversion=0.0.1 \
  -Dpackaging=jar
```

Then add the dependency to your project.

Maven

Add to your `pom.xml`:

```
<dependency>
  <groupId>com.teamraft</groupId>
  <artifactId>sdl-sdk</artifactId>
  <version>0.0.1</version>
  <scope>compile</scope>
</dependency>
```

Gradle

Add to your `build.gradle`:

```
repositories {
    mavenLocal()
}

dependencies {
    implementation "com.teamraft:sdl-sdk:0.0.1"
}
```

Requirements

- Java 1.8+
- Maven 3.8.3+ or Gradle 7.2+

Go SDK

Installation

The Go SDK is distributed as a zip file. Extract and install it locally:

```
# Extract the SDK
tar -xzf sdl-sdk-go.tar.gz

# Add to your go.mod as a local module
go mod edit -replace github.com/raft-tech/rdp_sdk=./rdp_sdk
```

Install required dependencies:

```
go get ./...
```

Python SDK

Installation

The Python SDK is available as a `whl` file. Download that file and install it in your environment.

`pip`

```
pip install ./rdp_sdk-1.0.0-py3-none-any.whl
```

`poetry`

```
poetry add ./rdp_sdk-1.0.0-py3-none-any.whl
```

`uv`

```
uv add ./rdp_sdk-1.0.0-py3-none-any.whl
```