Data-driven and Dynamic Space and Assets for Physical Internet-led Urban Logistics and Planning

DISCOLLECTION SMART DATA PLATFORM SETUP GUIDE

IMEC
Alper BASAK
April 2025

Funded by the European Union

This project has received funding from the European Union's Horizon Europe research and innovation programme under grant agreement No 101103954. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or CINEA. Neither the European Union nor the granting authority can be held responsible for them.

Table of Contents

Abstract

Data spaces are a great way for cities to connect to logistics players in their cities, and integrate valuable city data directly into the logistics ecosystem. However, making city data available on the data space can come with some technical challenges, both in transforming the data and connecting to the data space.

The Smart Data Platform reduces these obstacles to offering data on the data space by providing city data engineers with the tools they need to prepare their data for data space sharing and connect it to the data space. To achieve this, The Smart Data Platform makes use of a pre-existing data transformation (ETL) tool, packaged with a data space connector, which it connects through an overarching UI. This provides cities with all the capabilities of an ETL tool, while reducing the barrier to connecting to the data space.

The below document serves as a setup guide for the Smart Data Platform and attached urban Freight data space connector. The setup guide follows the case study of the Ghent Living lab where data was gathered from the Gent Open Data Platform to the Smart Data Platform, and made available to the Disco network via the embedded Sovity dataspace connector (and UI). The setup guide starts by presenting the overall architecture and building blocks of the Smart Data Platform. It provides a comprehensive description of each of these building blocks, as well as a technical installation guide, providing access to source codes for easy downloads.

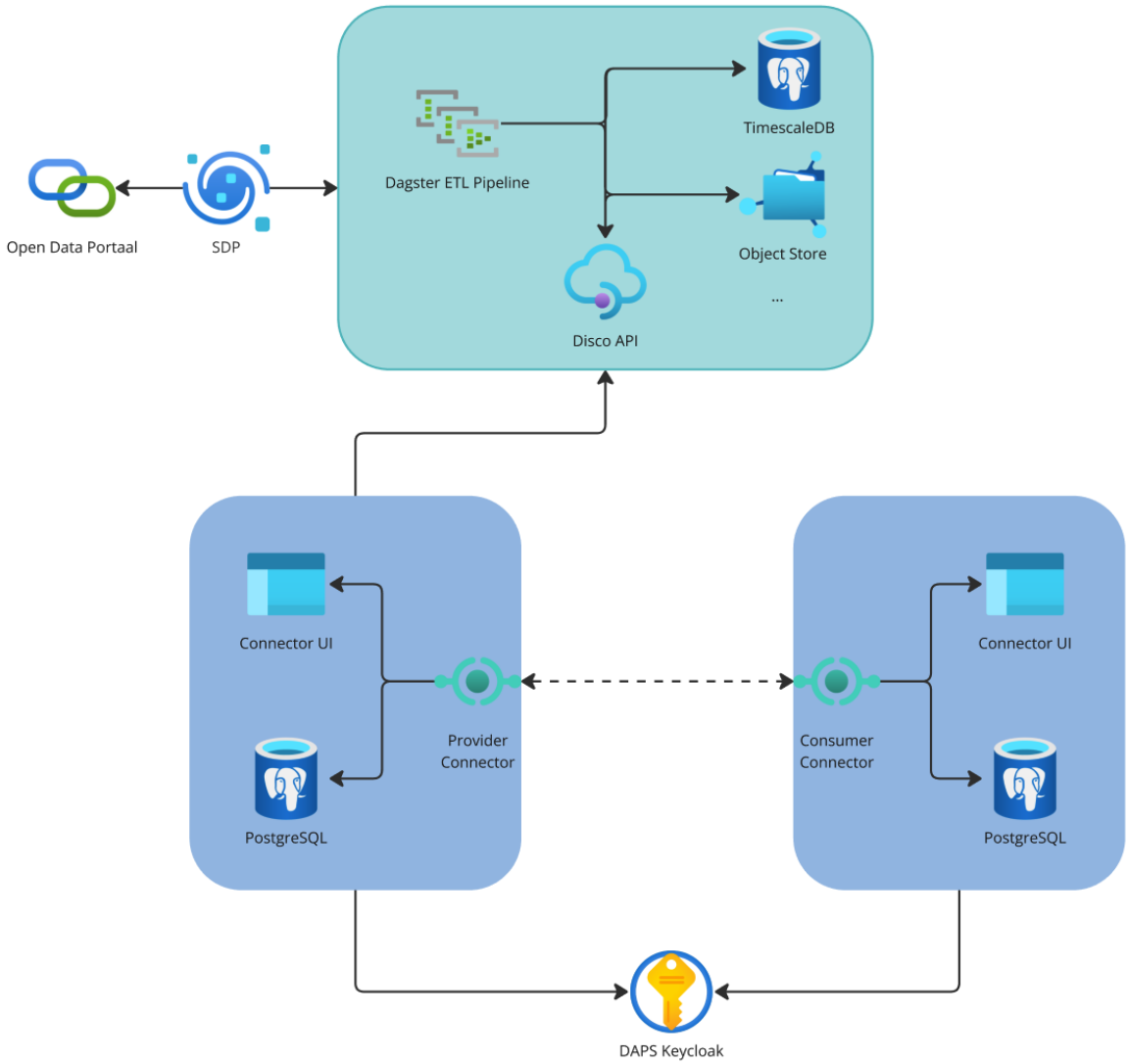
Document history

Version	Date	Organisation	Main area of changes	Comments
0.8	10 April 2025	imec		First version
0.9	18 April 2025	Imec	Added some extra technical docs and refined some text	

List of acronyms

EDS	European Data Space
EDC	Eclipse Dataspace Connector
OAuth2	Open Authorization 2.0
UVAR	Urban Vehicle Access Regulations

1. SDP System Architecture



Full System Architecture overview

The DISCOLLECTION solution consists of key components that interact with various parts of the system to facilitate data flow and integration. This architecture represents the data flow and system integration for delivering standardized, interoperable data to a dataspace environment.

The Smart Data Platform (SDP) serves as the central orchestrator that coordinates various components of the data lifecycle, from retrieval to consumption:

- Data Retrieval:
  DISCOLLECTION interfaces with the Gent Open Data Platform[1] using a REST API to fetch relevant datasets periodically. This platform provides data on public parking spaces, street parking locations, and real-time occupancy. These raw datasets will be processed by an ETL (Extract, Transform, Load) pipeline managed by Dagster[2]. SDP ensures that the data is retrieved in a timely and efficient manner, leveraging its orchestration capabilities to match raw data with the appropriate transformation processes and scheduling.

- Data Transformation:
  The Dagster ETL Pipeline plays a crucial role in transforming raw data into standardized formats such as APDS[3], which is standard in parking spaces or the DATEX II[4], which is a European standard for the exchange of traffic and travel data. This transformation ensures consistency and standardization across different data sources. The pipeline is orchestrated by SDP and may include validation, enrichment, and restructuring operations to meet the schema and semantic requirements of these formats.

- Data Storage:
  Once transformed, data is stored in one or more storage backends. The architecture supports flexibility in storage options, including TimescaleDB[5] for time-series data providing historical data queries and Object Stores[6] for unstructured or bulk data. These stores are part of the same infrastructure layer that includes an API server responsible for making the processed data accessible.

- DISCO Dataspace Connector:
  The DISCOLLECTION API facilitates interaction with the dataspace by exposing standardized data assets to external consumers via the Sovity[7] connectors. This includes Provider Connectors that publish data and Consumer Connectors that retrieve and consume it.

- Identity and Access Management:

---

[1] https://data.stad.gent/explore
[2] https://dagster.io
[3] https://www.allianceforparkingdatastandards.org/
[4] https://datex2.eu/
[5] https://www.timescale.com/
[6] https://en.wikipedia.org/wiki/Object_storage
[7] https://sovity.de/en/sovity-en/

All components involved in data exchange and access via the connectors are integrated with DAPS[8]; a Keycloak[9] implementation for Sovity, for secure authentication and authorization, ensuring trusted interactions within the dataspace.

- Integration with other DISCO-X solutions:
  The data provided by DISCOLLECTION can be integrated with other DISCO-X solutions across various cities and regions. This integration can help in visualizing and analyzing different types of geodata in logistics, such as historical occupation info, parking locations, and logistics real estate.

By clearly defining the interaction between these components, DISCOLLECTION ensures a seamless flow of data from initial collection to final distribution, enhancing the accessibility and utility of city data within the data space.

[8] https://github.com/sovity/sovity-daps
[9] https://www.keycloak.org/

2. Smart Data Platform

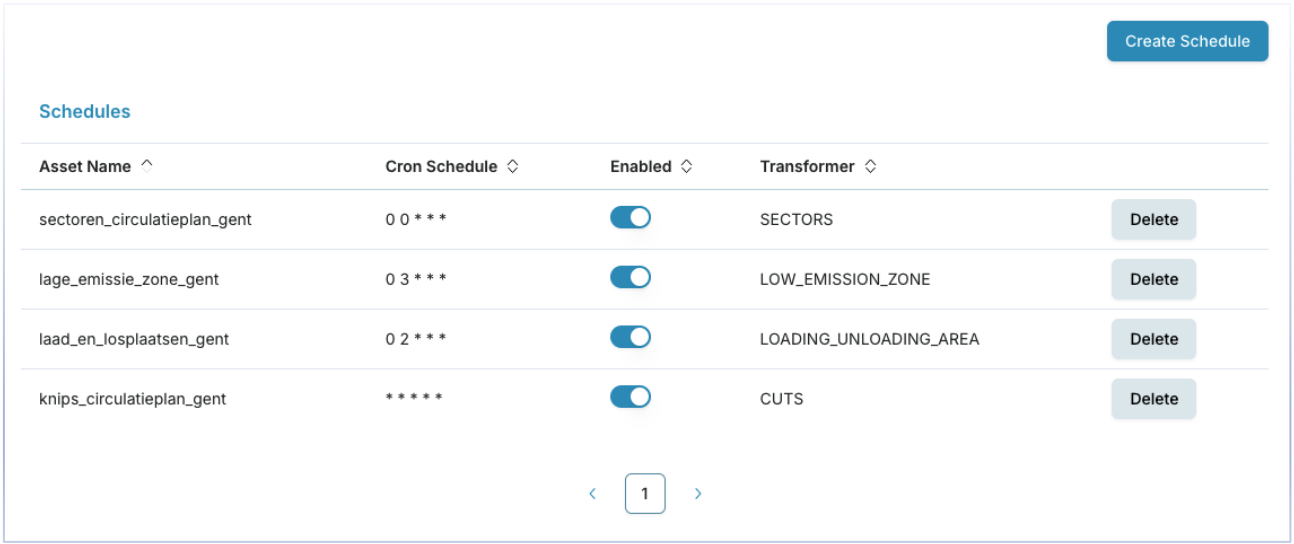
2.1. Component Description

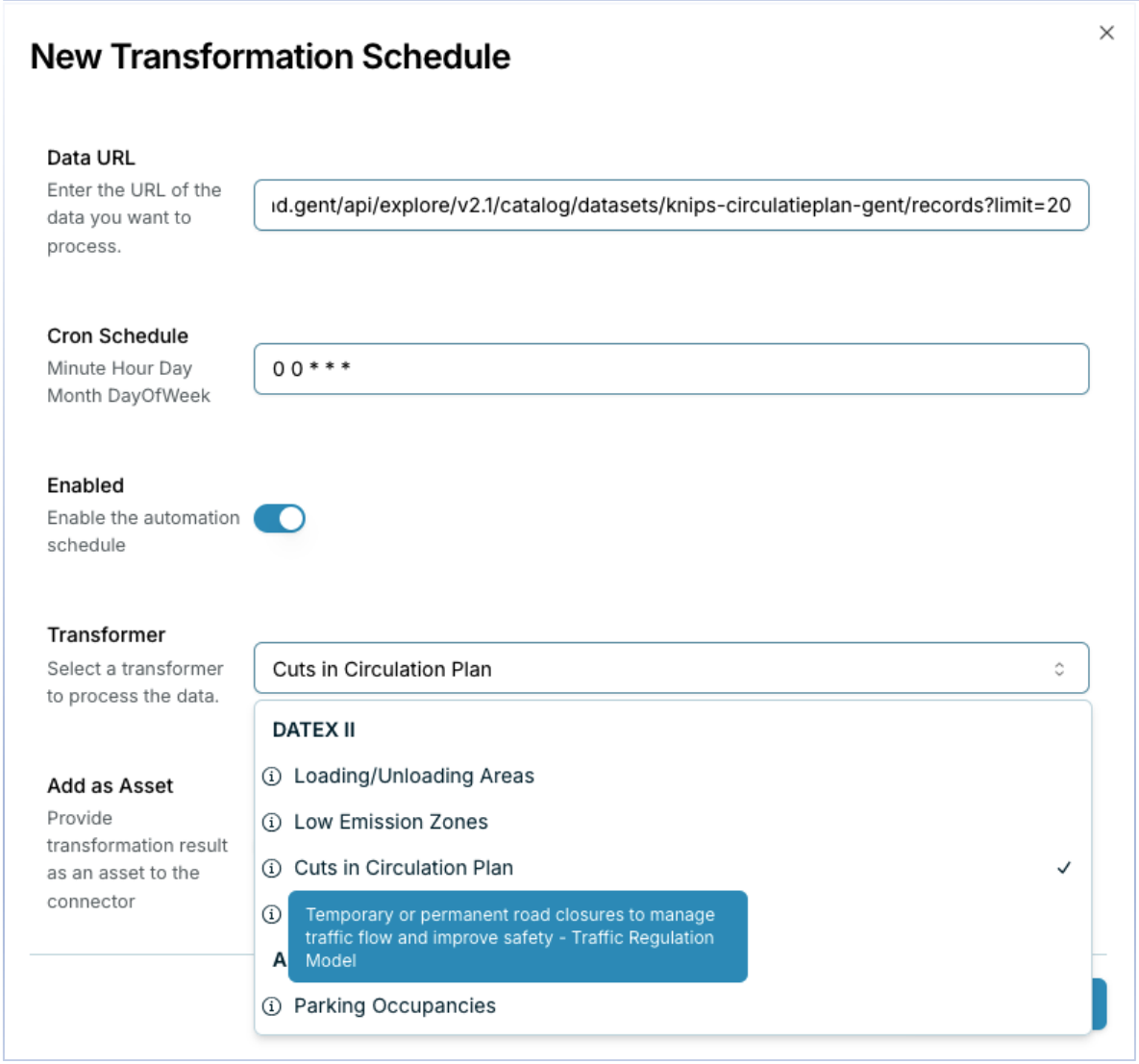
The Smart Data Platform is a data management and analytics platform that provides a unified environment for data ingestion, processing, storage, and analysis. It is designed to handle standardization and normalization of data from various sources. The platform acts as a central hub for data integration, allowing users to setup a pipeline for connecting the data sources, performing data transformations, and storing the data in a structured format. Scheduling capabilities are also included to automate the data ingestion and processing tasks.

2.2. Technical Specifications

Smart Data Platform is a Next.js[10] application that runs on Node.js. It is designed to be deployed in a cloud environment, and can be accessed via a web interface. The platform supports pipeline management for data ingestion and processing, allowing users to define the flow of data from source to destination. Platform uses the PostgreSQL[11] database for storing pipeline metadata and configuration information. User manages pipelines through a web interface, where they can define the data source url, data transformation rules and schedules for data ingestion.

[10] https://nextjs.org/
[11] https://www.postgresql.org/





Amongst the available standard transformers, suitable ones can be selected and will initiate the pipeline in the Dagster via the Schedule Monitor (see 1.3).

For convenience it is possible to automatically create an asset in the connector for the processed data.

2.3. Configuration Details

3 main configuration sets are required for the Smart Data Platform:

- Database configuration: This includes the database connection string, username, password, and other database settings.
- Auth configuration: This includes the Keycloak server URL, client ID, and other authentication settings for the platform.
- Connector configuration: This includes the configuration for the data space connectors, such as the management URL and secret key.

```
# DB
DB_HOST=
DB_PORT=
DB_USER=
DB_PASSWORD=
DB_NAME=
DB_SSL=

# NextAuth
AUTH_SECRET=
AUTH_KEYCLOAK_ID=
AUTH_KEYCLOAK_SECRET=
AUTH_KEYCLOAK_ISSUER=

# EDC CONNECTOR
EDC_HOST=
EDC_API_KEY=
```

2.4. Deployment Information

A Docker and a Helm chart is provided for the Smart Data Platform, which can be deployed in a cloud environment. Helm chart includes services, configmaps and deployment specifications for the platform. The manifest looks like this:

```
apiVersion: helm.toolkit.fluxcd.io/v2beta1
kind: HelmRelease
metadata:
  name: disco-schedule-manager
spec:
  releaseName: disco-schedule-manager
  interval: 5m
  chart:
    spec:
      chart: disco-schedule-manager
      version: 0.1.4
      sourceRef:
        kind: HelmRepository
        name: acrdisco-charts
        namespace: flux-system
  values:
    environment:
      DB_HOST: 'timescaledb.etl'
      AUTH_TRUST_HOST: true
      NEXTAUTH_URL: 'https://sdp.${domain}'
      DB_SSL: false
      EDC_HOST: 'https://provider.${domain}'
    secrets:
      - name: DB_PASSWORD
        secret:
disco.timescaledb.credentials.postgresql.acid.zalan.do
        key: password
      - name: AUTH_SECRET
        secret: schedule-manager-secrets
        key: auth-secret
      - name: AUTH_KEYCLOAK_SECRET
        secret: schedule-manager-secrets
        key: auth-keycloak-secret
      - name: EDC_API_KEY
        secret: schedule-manager-secrets
        key: edc-api-key
```

An ingress[12] is also included to expose the platform.

[12] https://sdp.disco.ai-data.imec.be

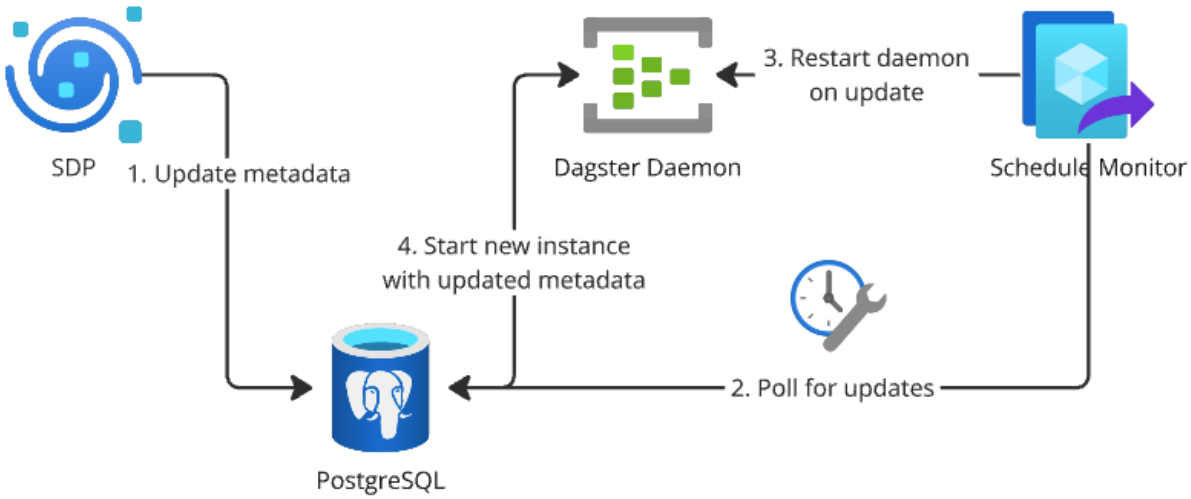
3. Schedule Monitor

3.1. Component Description

The Dagster Kubernetes deployment does not allow for runtime asset or schedule updates. On startup, the Dagster service will load all the definitions from the user code image, which is built and pushed to a container registry. The Dagster service will then use this image to run the user code in a Kubernetes cluster. If the User Code already has schedules and assets defined, they cannot be changed at runtime. This scenario is not really user friendly, as it requires a new image to be built and pushed to the container registry every time a schedule or asset is changed. This is not ideal for a production environment, where schedules and assets may need to be updated frequently.
An alternative method is being used here to allow for runtime updates of schedules and assets. As Dagster bootstraps the user code, it is possible to inject the asset and schedule definitions from external resources, such as the pipeline metadata database used by the Smart Data Platform.
The Dagster Kubernetes deployment uses this custom Kubernetes container image, namely Schedule Monitor, as a sidecar to the Dagster service. This sidecar is responsible for monitoring the schedule and asset changes in the PostgreSQL database that Smart Data Platform uses to store the pipeline metadata. Schedule Monitor polls the Postgres database to check for any changes to the schedules or assets. If a change is detected, the Schedule Monitor will trigger a restart of the Dagster service. This allows for runtime updates of schedules and assets without requiring a full redeployment of the Dagster service.
This means that any changes to the schedule or asset definitions require a restart of the Dagster service, triggered by this component. In the end, the Dagster service will be running with the latest schedule and asset definitions, and the user code image need not be updated.

3.2. Technical Specifications

Schedule Monitor is simple a Python script with sleep loop that checks the PostgreSQL database for any changes to the schedules or assets. If a change is detected, it will trigger a restart of the Dagster service. The script uses 'psycopg2[13]' to connect to the Postgres database and 'kubernetes[14]' to trigger the restart of the Dagster service. The script is run as a sidecar container in the same Kubernetes pod as the Dagster service. The script is configured to run periodically, and the period can be changed by modifying the sleep time in the script.



3.3. Configuration Details

[13] https://pypi.org/project/psycopg2/
[14] https://pypi.org/project/kubernetes/

The Schedule Monitor component is configured using environment variables. The following environment variables are required:

```
# DB
DATABASE_IP=
DATABASE_PORT=
DATABASE_USER=
DATABASE_NAME=
DATABASE_PASSWORD=

# Polling period
MONITOR_PERIOD=30

# Namespace of Dagster services in cluster
NAMESPACE=
```

3.4. Deployment Information

The Schedule Monitor is deployed as a sidecar to the Dagster Daemon pod as indicated in the helm chart values file [15]of the Dagster. It is possible to deploy as a standalone resource but as daemon and this service share similar resources, it was chosen to deploy as a sidecar.

```
      extraContainers:
        - name: dagster-schedule-monitor
          image: "acrdisco.azurecr.io/dagster-schedule-
monitor:0.2.15"
          imagePullPolicy: Always
          envFrom:
            - configMapRef:
                name: disco-user-code-config
          env:
            - name: DATABASE_PASSWORD
              valueFrom:
                secretKeyRef:
                  name:
disco.timescaledb.credentials.postgresql.acid.zalan.do
                  key: password
```

[15] https://github.com/dagster-io/dagster/blob/master/helm/dagster/values.yaml#L1268

4. Dagster ETL Component

4.1. Component Description

Dagster[16] is a data orchestrator that makes it easy to build scalable data pipelines. It allows you to define the data pipelines as a series of tasks that are executed in a specific order. This makes it easy to build complex data pipelines that can be easily scaled and maintained.
It is useful for building ETL (Extract, Transform, Load) pipelines that are used to extract data from various sources, transform it into a format that is suitable for analysis, and load it into a data warehouse or other storage system .
It has been considered as an alternative for Apache Airflow[17] or Prefect[18], which are other popular data orchestrators. Dagster has a more modern architecture and has a lower learning curve compared to others.

It uses Python to define the data pipelines. You can define the tasks as Python functions and then use the Dagster API to define the dependencies between the tasks.
It has a rich set of built-in libraries that make it easy to interact with various data sources and storage systems. It also has a web-based UI that allows you to monitor the progress of the data pipelines and view the logs of the tasks.

4.2. Pipeline Architecture

Pipelines in Dagster are defined as a series of operations that are executed in a specific order. Each task is defined as a python function that takes inputs and produces outputs. The tasks can be connected to form a data pipeline. These operations can be defined using the @asset[19] decorator. It is possible to connect one operation to multiple other operations, allowing for complex data pipelines to be built. For additional information, please refer to the official documentation[16].

[16] https://docs.dagster.io/
[17] https://airflow.apache.org/
[18] https://www.prefect.io/
[19] https://docs.dagster.io/guides/build/assets/

```
@asset
def knips_circulatieplan_gent(context: AssetExecutionContext) ->
list[Cut]:
    url =
"https://data.stad.gent/api/explore/v2.1/catalog/datasets/knips-
circulatieplan-gent"

    try:
        context.log.info(f"Fetching {url}")
        json_data = requests.get(url).json()

        response = CutsResponse(cuts=[Cut(**data) for data in
json_data])

        context.log.info(
            "Queried %s results for cuts spots in gent",
            len(response.cuts),
        )

        return response.cuts

    except Exception as e:
        context.log.error(e)
        raise e
```

In Dagster, assets can be chained together to form a directed acyclic graph (DAG) of dependencies[20].

```
# Example of defining assets with dependencies
@asset
def asset_a():
    return "Data from asset A"

@asset
def asset_b(asset_a):
    return f"Data from asset B, dependent on {asset_a}"

@asset
def asset_c(asset_b):
    return f"Data from asset C, dependent on {asset_b}"
```

[20] https://docs.dagster.io/guides/build/assets/defining-assets-with-asset-dependencies

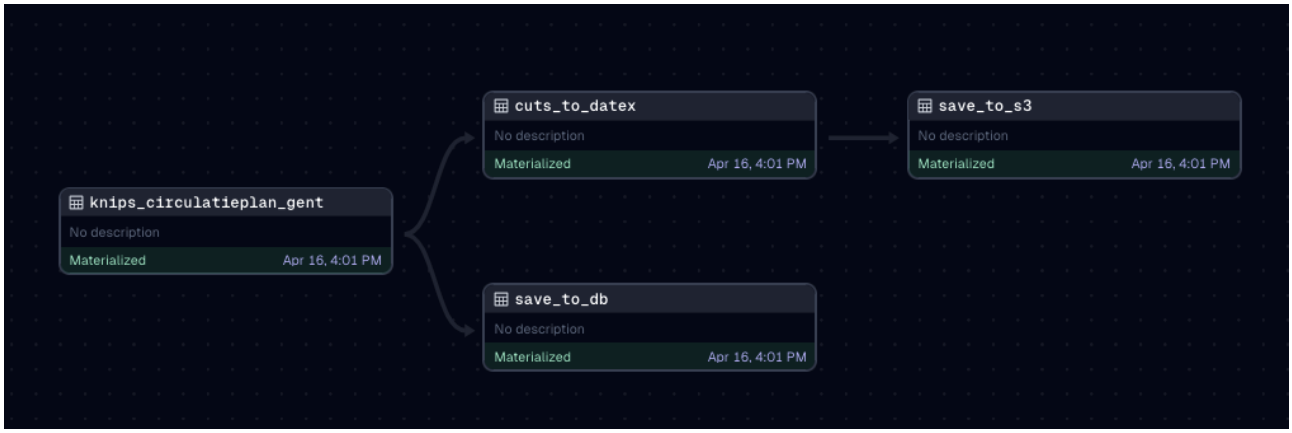
Assets can be considered as the functions that are executed in the pipeline. When a schedule is attached to an asset, it becomes a job[21]. Jobs can be scheduled to run at specific times or intervals[22].

```
cuts_schedule = ScheduleDefinition(
    job=define_asset_job(
        name="knips_circulatieplan_gent_job",
        selection="knips_circulatieplan_gent*",
    ),
    cron_schedule="0 2 * * *",  # every day at 2am
    default_status=DefaultScheduleStatus.RUNNING,
)

defs = Definitions(
    resources={
        "timescale_resource": TimescaleResource(
            database_ip=EnvVar("DATABASE_IP"),
            database_port=EnvVar("DATABASE_PORT"),
            database_user=EnvVar("DATABASE_USER"),
            database_password=EnvVar("DATABASE_PASSWORD"),
            database_name=EnvVar("DATABASE_NAME")
        ),
    },
    assets=load_assets_from_modules(cuts),
    schedules=[cuts_schedule]
)
```

Assets can be connected to other resources such as databases, storage systems, or other data sources[23]. This allows for the data to be extracted, transformed, and loaded into the desired storage system.

[21] https://docs.dagster.io/guides/build/jobs/
[22] https://docs.dagster.io/guides/automate/schedules/defining-schedules
[23] https://docs.dagster.io/guides/build/external-resources/



The data pipelines can be monitored using the web-based UI[24] that is provided by Dagster. This allows you to view the progress of the data pipelines, view the logs of the tasks, and monitor the performance of the data pipelines. All these sources add up to be a definition.

4.3. Deployment

Dagster deployment consists mainly of three components[25]: the Dagster daemon, Dagster web server and the user code that defines the data pipelines.

[24] https://docs.dagster.io/guides/operate/webserver
[25] https://docs.dagster.io/deployment/dagster-instance#k8srunlauncher

Minimally 2 images are enough for a standard setup: one image for daemon and web server, another for the user code. Daemon will be using gRPC to run the user code that is served from the container[26]. It is possible to create more user code images, suitable for the use case.

```
# workspace.yaml
load_from:
  # Each entry here corresponds to a service that exposes user code.
  - grpc_server:
      host: disco_user_code
      port: 4000
      location_name: "app/etl"
```

It is possible to tell the Dagster daemon where to find the user code from the 'workspace.yaml' file[27]. If more user codes are required, they need to be referenced here.

The Dagster daemon is responsible for executing the tasks in the data pipelines, while the dagster web server provides a web-based UI to monitor the progress of the data pipelines. The user code is responsible for defining the data pipelines and the tasks that are executed in the data pipelines.

It is possible to use Kubernetes Helm charts[28] to deploy the Dagster daemon and the Dagster web server. This allows for the deployment to be easily scaled and managed.

[26] https://docs.dagster.io/guides/deploy/code-locations/workspace-yaml#running-your-own-grpc-server
[27] https://docs.dagster.io/guides/deploy/code-locations/workspace-yaml
[28] https://docs.dagster.io/guides/deploy/deployment-options/kubernetes/deploying-to-kubernetes

```yaml
apiVersion: helm.toolkit.fluxcd.io/v2beta2
kind: HelmRelease
metadata:
  name: dagster
spec:
  releaseName: dagster
  chart:
    spec:
      chart: dagster
      version: 1.10.5
  values:

    dagsterWebserver:
      image:
        repository: "acrdisco.azurecr.io/dagster"
        tag: 0.2.13
      envConfigMaps:
        - name: dagster-config

    dagsterDaemon:
      image:
        repository: "acrdisco.azurecr.io/dagster"
        tag: 0.2.13
      envConfigMaps:
        - name: dagster-config
        - name: disco-user-code-config
      extraContainers:
        - name: dagster-schedule-monitor
          image: "acrdisco.azurecr.io/dagster-schedule-
monitor:0.2.15"

    dagster-user-deployments:
      deployments:
        - name: disco-user-code
          image:
            repository: "acrdisco.azurecr.io/dagster-user-code"
            tag: 0.2.13
          dagsterApiGrpcArgs:
            - "-m"
            - "disco_user_code"
          port: 4000
          envConfigMaps:
            - name: dagster-config
            - name: disco-user-code-config
```

The user code being a separate image allows it to be referenced from the Dagster instance and can be easily updated without affecting the deployment of the Dagster daemon and the web server.

Dagster uses a PostgreSQL included in the Helm chart[29] for jobs and assets history data persistence.

[29] https://github.com/dagster-io/dagster/blob/fb16ea1f8cfb2a7c910d688bcdd138393588dde3/helm/dagster/values.yaml#L799

5. DISCOLLECTION API Server

5.1. Component Description

The DISCOLLECTION API server is a RESTful API that provides access to the DISCOLLECTION data. It is designed to be lightweight and efficient. The API server allows users to query the DISCOLLECTION data using various parameters, including date range, location, and other filters. It also supports pagination and sorting of results, making it easy to retrieve large datasets in manageable chunks.
The API server is designed to be easy to use and integrate with Sovity Data Space Connectors. It provides a HTTP interface for accessing the DISCOLLECTION data.

5.2. Technical Specifications

It is built using the FastAPI[30] framework and Python programming language. It integrates with the PostgreSQL database and with Minio Object Store[31] that is also used to store the DISCOLLECTION data, which was processed and transformed by the ETL pipeline.
Follows OpenAPI[32] standards for API documentation and provides a Swagger UI[33] for easy exploration of the API endpoints. The API server is designed to be stateless and can be easily scaled horizontally to handle increased load.

[30] https://fastapi.tiangolo.com/
[31] https://min.io/
[32] https://www.openapis.org/
[33] https://swagger.io/

apds

GET `/v1/apds/places` List and search for elements in the Place hierarchy

tuld

GET `/v1/tuld/shops/` Get Shops

GET `/v1/tuld/warehouses/` Get Warehouses

GET `/v1/tuld/lockers/` Get Lockers

datex2

GET `/v1/datex2/cuts` Get Cuts

GET `/v1/datex2/loading-unloading` Get Loading Unloading

GET `/v1/datex2/sectors` Get Sectors

GET `/v1/datex2/low-emission-zones` Get Low Emission Zones

s3

GET `/v1/s3/` Get Object

From the given query params, retrieve an object from S3. This is generic endpoint to retrieve any object from S3.

Parameters Try it out

Name	Description
bucket * required string *(query)*	The S3 bucket name bucket
object_name * required string *(query)*	The S3 object name object_name

5.3. Configuration Details

Environment variables are used to configure the API server.

```
# Comment out to enable CORS for all origins
# BACKEND_CORS_ORIGINS="http://localhost:3000"

# Postgres
PG_SERVER=localhost
PG_PORT=5432
PG_DB=disco
PG_USER=disco
PG_PASSWORD=password

# Minio
S3_SERVER=http://localhost:9000
S3_ACCESS_KEY=
S3_SECRET_KEY=
```

5.4. Deployment Information

For deployment, the API server can be run using Docker. A chart is provided for deploying the API server on Kubernetes. The chart includes configmap, deployment, and service resources for the API server. API server does not necessarily have to be publicly accessible, but it needs to be accessed through the Sovity Data Space Connector. The API server can be deployed in a private network and accessed securely through the connector.

```yaml
apiVersion: helm.toolkit.fluxcd.io/v2beta1
kind: HelmRelease
metadata:
  name: disco-api
spec:
  releaseName: disco-api
  interval: 5m
  chart:
    spec:
      chart: disco-api
      version: 0.1.25
      sourceRef:
        kind: HelmRepository
        name: acrdisco-charts
        namespace: flux-system
  values:
    environment:
      PG_SERVER: "timescaledb.etl"
      S3_SERVER: "http://minio"
    secrets:
      - name: PG_PASSWORD
        secret:
disco.timescaledb.credentials.postgresql.acid.zalan.do
        key: password
      - name: S3_ACCESS_KEY
        secret: minio-client
        key: access-key
      - name: S3_SECRET_KEY
        secret: minio-client
        key: secret-key
```

6. Sovity Connectors

6.1. Component Description

The Sovity Connector is chosen to be the Data Space connector implementation for DISCO, to provide a standardized way to share data assets securely and efficiently. It implements the Eclipse Dataspace Connector (EDC)[34] framework, enabling organizations to publish, share, and consume data across different platforms while maintaining data sovereignty.

6.2. Technical Specifications

It is designed to support various data models, and provides REST API endpoints for managing data assets, establishing data contracts, and initiating data transfers. The Sovity Connector extends EDC connector that uses Java, with support for containerization using Docker and orchestration with Kubernetes. The dependencies include PostgreSQL for metadata storage and Keycloak for secure access control.

6.3. Configuration Details

It is highly configurable through environment variables, allowing customization of database connections, API keys, security tokens, and other settings. The Sovity Connector provides API endpoints for managing data assets, establishing data contracts, and initiating data transfers. Protected by OAuth2, it ensures secure access to APIs and data assets, with support for TLS encryption for data in transit. Protocol support includes IDS for secure data exchange[35], with fine-grained access control policies and comprehensive logging for auditing purposes.

A standard connector configuration looks like this:

[34] https://projects.eclipse.org/projects/technology.edc
[35] https://docs.internationaldataspaces.org/ids-knowledgebase/dataspace-protocol

```
MY_EDC_PARTICIPANT_ID=${connector}
MY_EDC_TITLE=${connector} Connector
MY_EDC_DESCRIPTION=Extended Sovity Community Edition EDC Connector
MY_EDC_CURATOR_URL=https://www.imec-int.com/en
MY_EDC_CURATOR_NAME=imec
MY_EDC_MAINTAINER_URL=https://www.imec-int.com/en
MY_EDC_MAINTAINER_NAME=imec
MY_EDC_FQDN=${connector}.${domain}
MY_EDC_JDBC_URL=jdbc:postgresql://postgres.${connector}:5432/edc
MY_EDC_JDBC_USER=edc
MY_EDC_PROTOCOL=https://
EDC_DSP_CALLBACK_ADDRESS=https://${connector}.${domain}/api/dsp
EDC_DATAPLANE_TOKEN_VALIDATION_ENDPOINT=https://${connector}.${domai
n}/api/control/token
EDC_WEB_REST_CORS_ENABLED=true
EDC_WEB_REST_CORS_HEADERS=origin,content-
type,accept,authorization,x-api-key
EDC_WEB_REST_CORS_ORIGINS=*
EDC_AGENT_IDENTITY_KEY=client_id
EDC_OAUTH_TOKEN_URL=${daps}/realms/DAPS/protocol/openid-
connect/token
EDC_OAUTH_PROVIDER_AUDIENCE=${daps}/realms/DAPS
EDC_OAUTH_PROVIDER_JWKS_URL=${dataspace-
manager}/api/v1/daps/certs/jwks
EDC_OAUTH_TOKEN_EXPIRATION=15
EDC_OAUTH_CLIENT_ID=${connector}
EDC_OAUTH_CERTIFICATE_ALIAS=${connector}
EDC_OAUTH_PRIVATE_KEY_ALIAS=${connector}
WEB_HTTP_PUBLIC_PATH=/public
WEB_HTTP_PUBLIC_PORT=12001
```

Each connector requires a keystore file where the private key is stored. The key will be used for identification purposes, via DAPS[36], which is a customized Keycloak implementation as an identity governance.

DAPS follows similar configuration as a Keycloak instance:

```
KC_HOSTNAME=daps.${domain}
KEYCLOAK_ADMIN=admin
QUARKUS_HTTP_ACCESS_LOG_ENABLED=true
KC_HTTP_ENABLED=true
KC_DB=postgres
KC_DB_URL_HOST=postgres
KC_DB_URL_DATABASE=keycloak
KC_DB_SCHEMA=public
KC_DB_USERNAME=keycloak
KC_PROXY=edge
```

[36] https://github.com/sovity/sovity-daps

And a user interface is also provided for easily interact with the connector. Required configurations are:

```
EDC_UI_ACTIVE_PROFILE=sovity-open-source
EDC_UI_CONFIG_URL=edc-ui-config
EDC_UI_MANAGEMENT_API_URL=https://${connector}.${domain}/api/managem
ent
EDC_UI_CATALOG_URLS=https://${connector}.${domain}/api/dsp
NGINX_ACCESS_LOG=off
```

Most important are the 'management url' and (if set – management api key), to be able to use the connector management API.

6.4. Deployment Information

Packages are available for Docker images for Kubernetes deployment, simplifying the setup and scaling of the Sovity Connector. The deployment process involves either fetching prebuilt Docker images or cloning the community edition GitHub repository[37] and building your own images and configuring environment variables for secure operation.
Community editions are available for testing and development purposes, with enterprise editions offering additional features and support options.
Releases[38] can be used for community edition releases. DISCOLLECTION uses v10.4.2[39] with extended features[40].

A standard deployment of a connector includes the connector, connector UI and a PostgreSQL database. Images are referenced from deployment manifests.

[37] https://github.com/sovity/edc-ce/
[38] https://github.com/sovity/edc-ce/releases
[39] https://github.com/sovity/edc-ce/releases/tag/v10.4.2
[40] https://github.com/imec-int/edc-extensions/pkgs/container/edc-extensions%2Fsovity-xce/354863169?tag=10.4.2

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: connector-ui
spec:
  selector:
    matchLabels:
      app: connector-ui
  template:
    metadata:
      labels:
        app: connector-ui
    spec:
      containers:
        - name: connector-ui
          image: ghcr.io/sovity/edc-ui:4.1.8
          envFrom:
            - configMapRef:
                name: connector-ui-config
          env:
            - name: EDC_UI_MANAGEMENT_API_KEY
              valueFrom:
                secretKeyRef:
                  key: api-auth-key
                  name: connector-secrets
          ports:
            - containerPort: 8080
              name: http
```

For the connector, it is important to have the DSP and the public port accessible from external resources, as they will be used during the negotiation and data transfer between connectors. Management port can be made private, if UI connection is not required. Keystore files should be mounted as volumes.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: connector
spec:
  selector:
    matchLabels:
      app: connector
    spec:
      containers:
        - name: connector
          image: acrdisco.azurecr.io/sovity-xce:latest
          imagePullPolicy: Always
          envFrom:
            - configMapRef:
                name: connector-config
          env:
            - name: EDC_KEYSTORE
              value: keystore.p12
            - name: EDC_KEYSTORE_PASSWORD
              valueFrom:
                secretKeyRef:
                  key: keystore-password
                  name: connector-secrets
                  optional: true
            - name: EDC_API_AUTH_KEY
              valueFrom:
                secretKeyRef:
                  key: api-auth-key
                  name: connector-secrets
            - name: MY_EDC_JDBC_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: edc.postgres.credentials.postgresql.acid.zalan.do
                  key: password
          ports:
            - containerPort: 12001
              name: http-public
            - containerPort: 11001
              name: http
            - containerPort: 11002
              name: management
            - containerPort: 11003
              name: protocol
            - containerPort: 11004
              name: control
          volumeMounts:
            - name: connector-keystore
              mountPath: /app/keystore.p12
              subPath: keystore.p12
              readOnly: true
      volumes:
        - name: connector-keystore
          secret:
            secretName: certs
            items:
              - key: keystore.p12
                path: keystore.p12
```

And the for the DAPS:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: daps
spec:
  selector:
    matchLabels:
      app: daps
  template:
    metadata:
      labels:
        app: daps
    spec:
      containers:
        - name: daps
          image: ghcr.io/sovity/sovity-daps:sha-32a3033
          envFrom:
            - configMapRef:
                name: daps-config
          env:
            - name: KEYCLOAK_ADMIN_PASSWORD
              valueFrom:
                secretKeyRef:
                  key: admin-password
                  name: daps-secrets
            - name: KC_DB_PASSWORD
              valueFrom:
                secretKeyRef:
                  name:
keycloak.postgres.credentials.postgresql.acid.zalan.do
                  key: password
          ports:
            - containerPort: 8080
              name: http
```

6.5. Security Considerations

To utilize authentication and authorization mechanisms, the Sovity Connector integrates with OAuth2 for secure access control. One custom component is the Sovity DAPS which is Sovity's implementation of the Dynamic Attribute Provisioning Service based on the Keycloak IAM platform. It acts as a central identity provider and attribute authority for the Sovity Connector, providing fine-grained access control policies and user management capabilities.
Connectors obtain certificates from Sovity DAPS for authentication and authorization with other Sovity Connectors. These certificates add custom claims to JWT tokens used in this process. Invalid tokens will cause protocol requests between connectors to fail for security reasons.

6.6. Setup Instructions

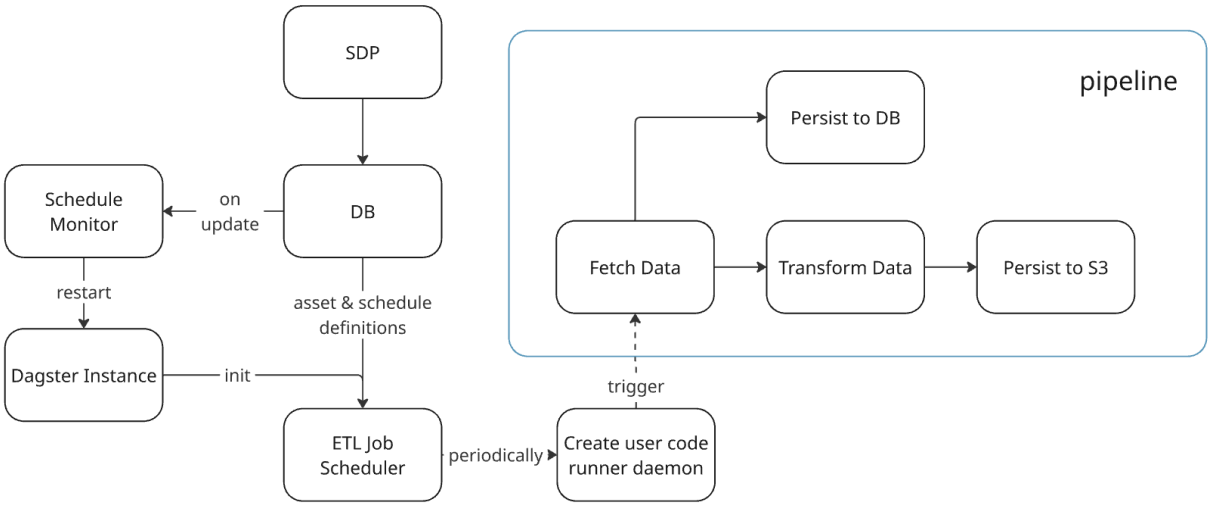
The Sovity Connector can be set up by following these steps:

- Deploy 'sovity-daps'[41]: This will automatically create a 'DAPS' realm on first initialization.
- Login with admin, and create clients and users and generate certificates in the Sovity DAPS for authentication and authorization[42]: Create clients with matching names with the connector IDs. Select 'Signed JWT' from Credentials and from Keys, 'Generate new keys' and save them later to be used as Keystore in the connector.
- Connector docker image for deployment[43] is a customized version referencing `sovity/edc-ce:10.4.2`, the difference being the addition of HTTP Proxy transfer support in the build. This allows the users to be able to use the extra transfer method besides the standard HTTP Push (see transfer methods[44]).
- Configure environment variables for database connections, API keys, and security tokens[45]
- Deploy the Sovity Connector ensuring proper network and storage configurations.
- Register data assets, establish data contracts, and initiate data transfers using the API endpoints provided by the Sovity Connector.

For a more detailed started guide, please check official documentation[46].

[41] https://github.com/sovity/sovity-daps/pkgs/container/sovity-daps/228065319?tag=sha-32a3033
[42] https://github.com/sovity/sovity-daps?tab=readme-ov-file#realm-configuration
[43] https://github.com/imec-int/edc-extensions/tree/feat/sovity-ce-http-proxy
[44] https://github.com/imec-int/edc-extensions/blob/feat/sovity-ce-http-proxy/docs/getting-started/documentation/data-transfer-methods.md
[45] https://github.com/sovity/edc-ce/tree/main/docs/deployment-guide/goals/production
[46] https://github.com/sovity/edc-ce/tree/main/docs/getting-started

7. Integration and Data Flow



Data Flow Pipeline

Preprocessing: Whenever a user code is added, the Dagster instance will schedule a new pipeline run based on configuration. It will trigger the creation of a new Dagster daemon that will run the operation per scheduled process. This allows for parallel processing of multiple pipelines separately. Essentially this is equivalent to a `Job` in Kubernetes.

There are some datasets that are important for DISCO such as parking availabilities and city access rules.

7.1 Straatparkeerplaatsen Gent:

This dataset[47] shows the on-street parking spaces in Ghent. Job schedule is configurable and for now it is configured to run every midnight.

Data Ingestion:
Dataset is not updated regularly, so it was deemed unnecessary to persist the data in a way that allows for historical analysis. Any query will update the data in the database.

[47] https://data.stad.gent/explore/dataset/straatparkeerplaatsen-gent

One example record looks like this:


```
{
  "geometry":{
    "type":"Feature",
    "geometry":{
      "coordinates":[...],
      "type":"Polygon"
    },
    "properties":{
    }
  },
  "pregime":"4",
  "opmerkingen":null,
  "capaciteit":2,
  "datum":"2024-05-06T22:00:00Z",
  "straatcode":71840,
  "pcapcode":4286,
  "wijkcode":null,
  "ptype":400,
  "zone":"TARGROEN",
  "geo_point_2d":{
    "lon":3.7450704829301436,
    "lat":51.03632127319923
  }
}
```


There is a limitation of the records returned for Open Data Portaal API, which is 100. There are 19842 records in total, so either the data was to be fetched in chunks with query API, or the whole dataset was to be exported as JSON[48]. The latter was chosen for simplicity.

Data Transformation:
Transformation is not required for this dataset.

Data Storage:
The data is stored in a Timescale DB instance, with a similar schema to the original dataset.

[48] https://data.stad.gent/api/explore/v2.1/catalog/datasets/straatparkeerplaatsen-gent/exports/json

7.2 Real time bezetting parkeergarages Gent:

This dataset[49] shows the real-time occupancy of parking garages in Ghent. Job schedule is configurable and for now it is configured to run every hour from the SDP.

Data Ingestion:

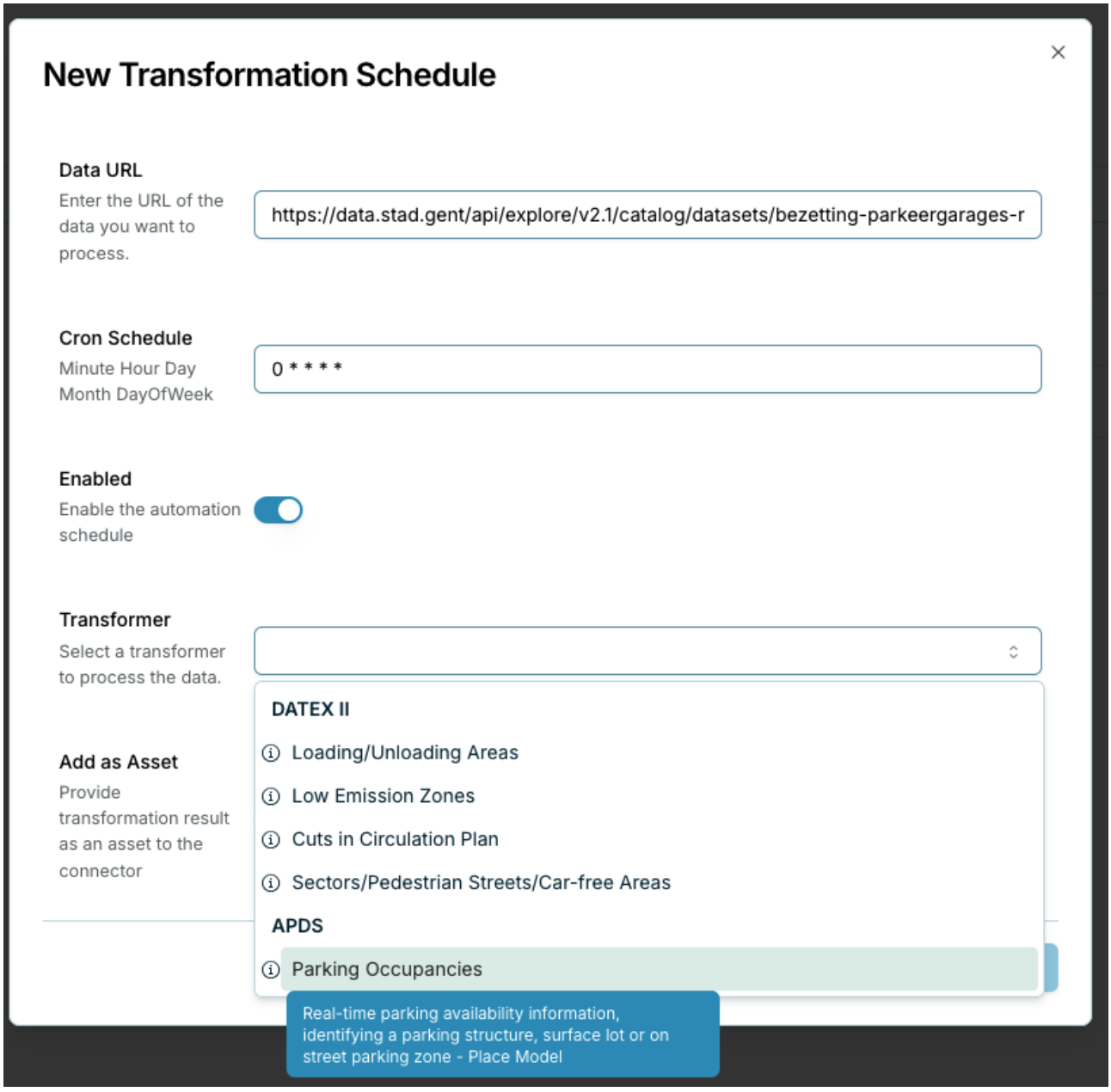
Visit the Open Data Portaal's API details page[50]. There should be the URL of the API call available:

URL of the API call

/api/explore/v2.1/catalog/datasets/bezetting-parkeergarages-real-time/records?limit=20

We will be using this URL as the input in the SDP platform[51]. Create a new transformation schedule with given details. Be sure to select the suitable transformer. As this is a Parking type dataset, APDS format is suitable. SDP will orchestrate the creation of the ETL pipeline where it will schedule a periodic job that fetches the data from the given URL, transforms it and persists the formatted results for queries. It is also possible to view the pipeline details from Dagster web UI[52].

[49] https://data.stad.gent/explore/dataset/bezetting-parkeergarages-real-time/information/
[50] https://data.stad.gent/explore/dataset/bezetting-parkeergarages-real-time/api/
[51] https://sdp.disco.ai-data.imec.be
[52] https://etl.disco.ai-data.imec.be/



Dataset is updated in every minute, so it is important to persist the data in a way that allows for historical analysis. Any query will append the data with query timestamp in the database.

One example record looks like this:


```
{
  "name":"Vrijdagmarkt",
  "lastupdate":"2024-07-30T10:00:31+02:00",
  "totalcapacity":593,
  "availablecapacity":452,
  "occupation":23,
  "type":"carPark",
  "description":"Ondergrondse parkeergarage Vrijdagmarkt in Gent",
  "id":"https://stad.gent/nl/mobiliteit-openbare-
werken/parkeren/parkings-gent/parking-vrijdagmarkt",
  "openingtimesdescription":"24/7",
  "isopennow":1,
  "temporaryclosed":0,
  "operatorinformation":"Mobiliteitsbedrijf Gent",
  "freeparking":0,
  "urllinkaddress":"https://stad.gent/nl/mobiliteit-openbare-
werken/parkeren/parkings-gent/parking-vrijdagmarkt",
  "occupancytrend":"unknown",
  "locationanddimension":"{"specificAccessInformation": ["inrit"],
"level": "0", "roadNumber": "?", "roadName": "Vrijdagmarkt 1\n9000
Gent", "contactDetailsTelephoneNumber": "Tel.: 09 266 29
00\n(permanentie)\nTel.: 09 266 29 01\n(tijdens kantooruren)",
"coordinatesForDisplay": {"latitude": 51.05713405953412,
"longitude": 3.726071777876147}}",
  "location":{
    "lon":3.726071777876147,
    "lat":51.05713405953412
  },
  "text":null,
  "categorie":"parking in LEZ",
  "dashboard":"True"
}
```


Same limitation for the API queries exists for this dataset, however as the total count is generally 10-13 records, it is acceptable to use the API[53] instead of exports.

Data Transformation:
Conversion to APDS standards will be applied, as this dataset is a parking information type[54]. For more detailed information on transformations please refer to the **DISCO Data Conversion Documentation.**

[53] https://data.stad.gent/api/explore/v2.1/catalog/datasets/bezetting-parkeergarages-real-time/records
[54] https://gitlab.com/disco-horizon-europe/open-software-repository/discollection-etl/-/blob/main/app/etl/assets/apds/transform.py?ref_type=heads

Data Storage:
As this dataset is updated very frequently, and as the Open Data Portaal does not allow for historical queries; as an alternative, a TimescaleDB instance is used to store this time series data allowing for efficient querying and storage of large amounts of time series data. The dataset is first stored in a table resembling the original payload.

```
CREATE EXTENSION IF NOT EXISTS timescaledb;
CREATE TABLE IF NOT EXISTS parking_occupancies_gent
(
    id                      TEXT        NOT NULL,
    name                    TEXT        NOT NULL,
    lastupdate              TIMESTAMPTZ NOT NULL,
    totalcapacity           INT,
    availablecapacity       INT,
    occupation              INT,
    type                    TEXT,
    description             TEXT,
    openingtimesdescription TEXT,
    isopennow               INT,
    temporaryclosed         INT,
    operatorinformation     TEXT,
    freeparking             INT,
    urllinkaddress          TEXT,
    occupancytrend          TEXT,
    locationanddimension    JSONB,
    location                JSONB,
    text                    TEXT,
    categorie               TEXT,
    dashboard               TEXT
);

SELECT create_hypertable('parking_occupancies_gent', 'lastupdate',
if_not_exists => TRUE);
CREATE INDEX IF NOT EXISTS idx_occupancy_id_lastupdate ON
parking_occupancies_gent (id, lastupdate DESC);
```

The transformed data also is stored in another table, which is APDS compliant. This table is used to query the ADPS data.

```
CREATE TABLE IF NOT EXISTS apds_parking_occupancies_gent
(
    id                                  TEXT        NOT NULL,
    version                             INT         NOT NULL,
    timestamp                           TIMESTAMPTZ NOT NULL,
    name                                JSONB       NOT NULL,
    description                         JSONB,
    layer                               INT,
    type                                TEXT,
    aliases                             JSONB,
    parent_id                           JSONB,
    child_ids                           JSONB,
    operator_defined_reference          JSONB,
    hierarchy_element_record            JSONB,
    right_specifications                JSONB,
    hierarchy_element_reference         JSONB,
    occupancy_level                     JSONB,
    time_zone                           TEXT,
    place_street_address                JSONB,
    place_bounded_zone                  JSONB,
    indicative_place_point_location JSONB,
    characteristics                     JSONB,
    contacts                            JSONB,
    marketing                           JSONB,
    operating_restrictions              JSONB,
    rgb_colours                         JSONB,
    times                               JSONB,
    payment_methods                     JSONB,
    safety_standard_classifications JSONB
);

SELECT create_hypertable('apds_parking_occupancies_gent',
'timestamp', if_not_exists => TRUE);
CREATE INDEX IF NOT EXISTS idx_occupancy_id_timestamp ON
apds_parking_occupancies_gent (id, timestamp DESC);
```

With two tables in place, both the historical original data and regularly transformed data are always available. Both have hyper tables created for efficient querying and storage based on timestamp. For a more automated approach, it is possible to create Dagster operations to initialize the TimescaleDB schemas as well.

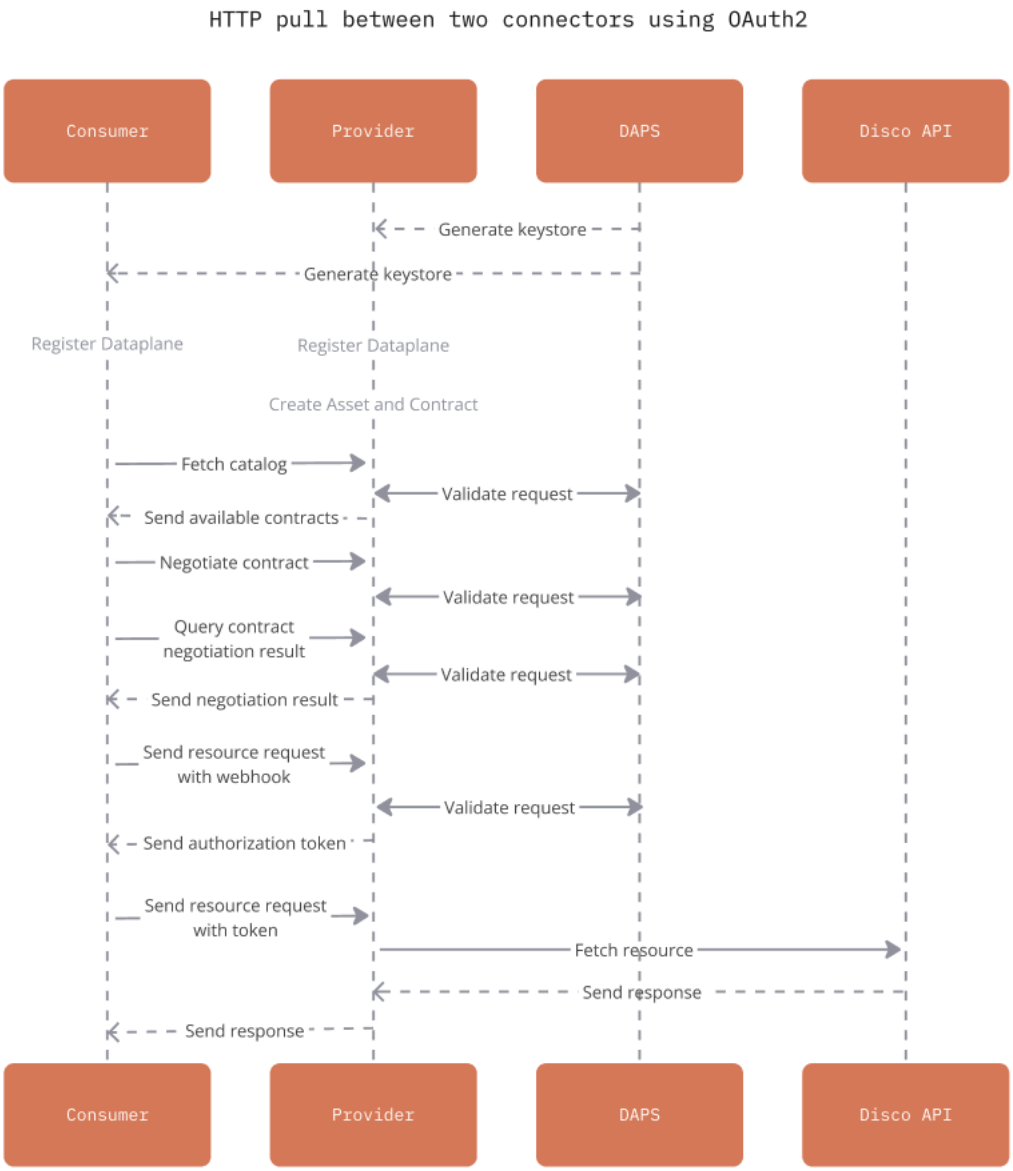
7.3 Connectors

For DISCO data space, we'll be using an extended version (XCE)[55] of the Sovity Community Edition (SCE) connector, forked from their original repository. The SCE extends the base connector with

[55] https://github.com/imec-int/edc-extensions/tree/feat/sovity-ce-http-proxy

some default policies, a user interface, and more. For more information, see the available extensions[56]. The XCE adds support for the Consumer Pull transfer mode[57]. It is important to note Sovity's Connectors (v10.x) are dependent on an older version of the EDC Connector (v0.2.1.4)[58].
When using and EDC Connector, one can choose between two flows when performing a data transfer[59]: Consumer Pull or Provider Push. The Consumer Pull flow allows the Consumer to query the asset's data source through a proxy on the Provider side. This allows for synchronous HTTP-based data transfers that return the data immediately to the Consumer.
Alternatively, the Provider Push flow provides an asynchronous way to transfer data whereby the Provider "pushes" data from the asset's data source to the Consumer's data sink. This method supports more types of data sinks, such as cloud storage services, FTP servers, and more.
The Consumer Pull flow is a simpler approach, allowing the Consumer to send regular HTTP requests through the Provider connector as if it was querying the data source directly.

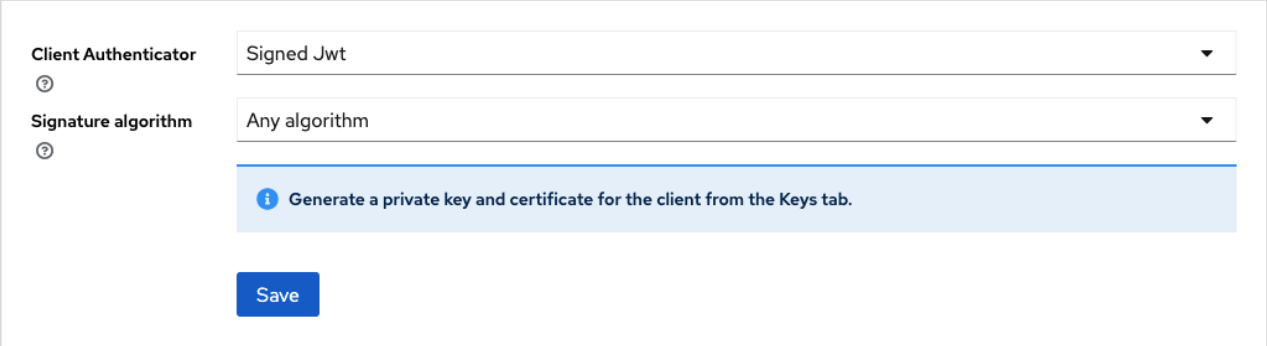
[56] https://github.com/sovity/edc-ce/tree/v10.5.1/extensions
[57] https://github.com/imec-int/edc-extensions/blob/feat/sovity-ce-http-proxy/launchers/common/base/build.gradle.kts#L38-L41
[58] https://github.com/imec-int/edc-extensions/blob/31de454e21c757e10d31f20de8e2d93f6d16103b/gradle/libs.versions.toml#L17
[59] https://github.com/imec-int/edc-extensions/blob/feat/sovity-ce-http-proxy/docs/getting-started/documentation/data-transfer-methods.md



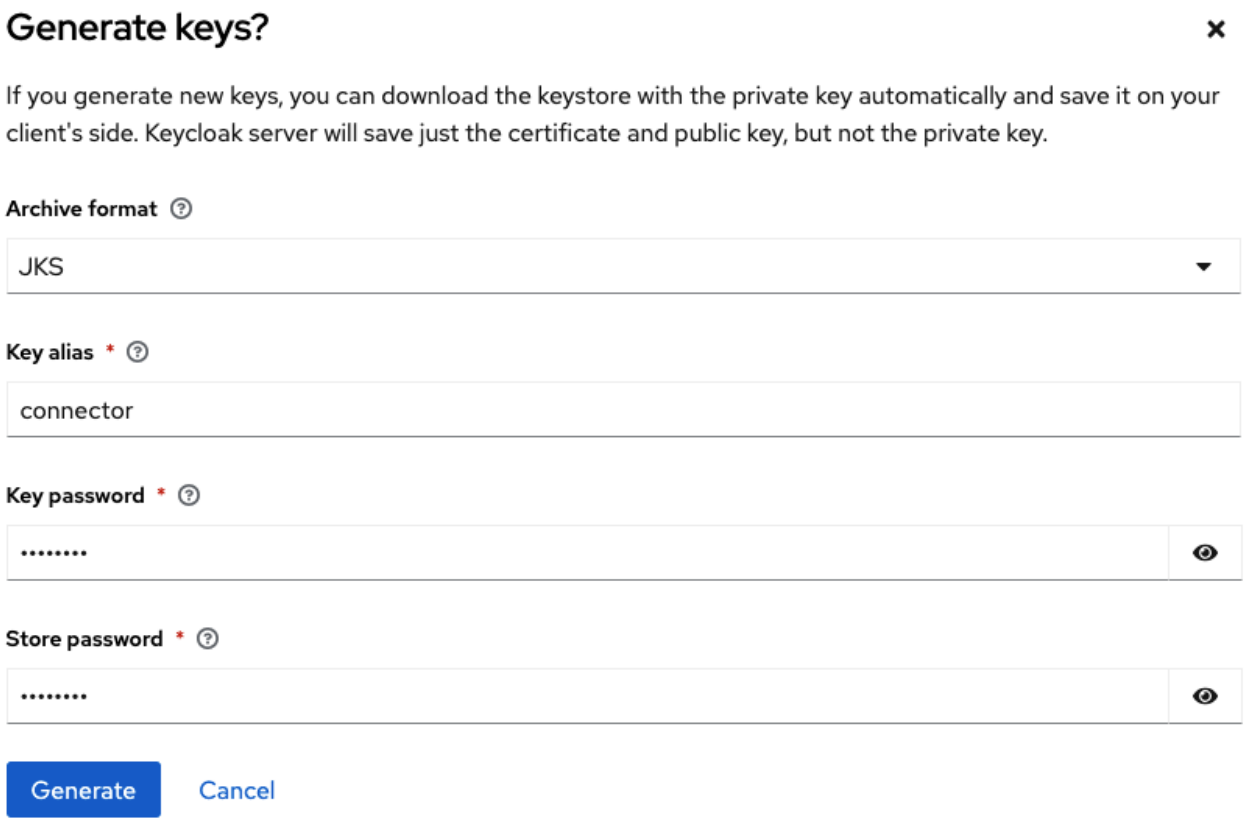
OAuth2 flow

- Create clients from DAPS instance: Be sure to enable Client authentication.

- From Credentials select Signed JWT



- Generate New private and public keys for each connector from DAPS, to be used as keystore for the connectors. Save the passwords as they will be used for connector configuration.



- Register the data planes.

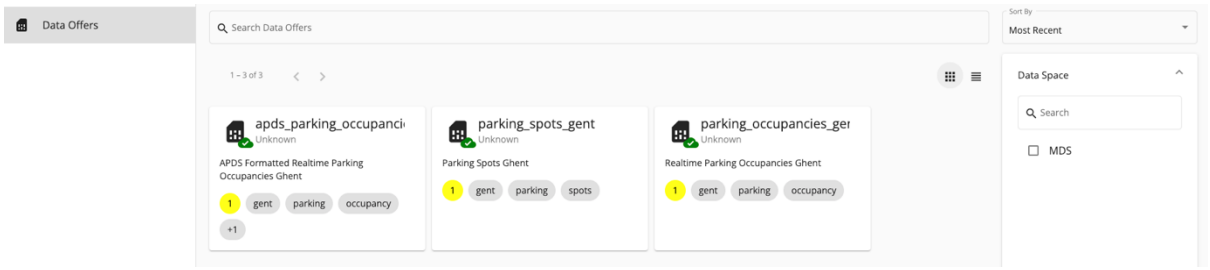
```
curl --location 'https://${connector}/api/management/v2/dataplanes'
\
--header 'Content-Type: application/json' \
--header 'x-api-key: <api-key>' \
--data-raw '{
    "@context": {
        "edc": "https://w3id.org/edc/v0.0.1/ns/"
    },
    "edctype": "dataspaceconnector:dataplaneinstance",
    "id": "http-pull-dataplane",
    "url": "https://${connector}/api/management/transfer",
    "allowedSourceTypes": [
        "HttpData"
    ],
    "allowedDestTypes": [
        "HttpData",
        "HttpProxy"
    ],
    "properties": {
        "edc:publicApiUrl": "https://${connector}/public"
    }
}'
```

- Create Assets which will point to Disco API. Disco API should not be publicly accessible so that it is only available via Provider Connector. You can either not have a public URL for the API or require Authorization for the API. It is possible to add OAuth2 client and secrets in the asset definition, so that the provider will use these while fetching the resource from a public endpoint[60] As this is not available in the community edition of the Sovity connector, the DISCO API is not made publicly available.

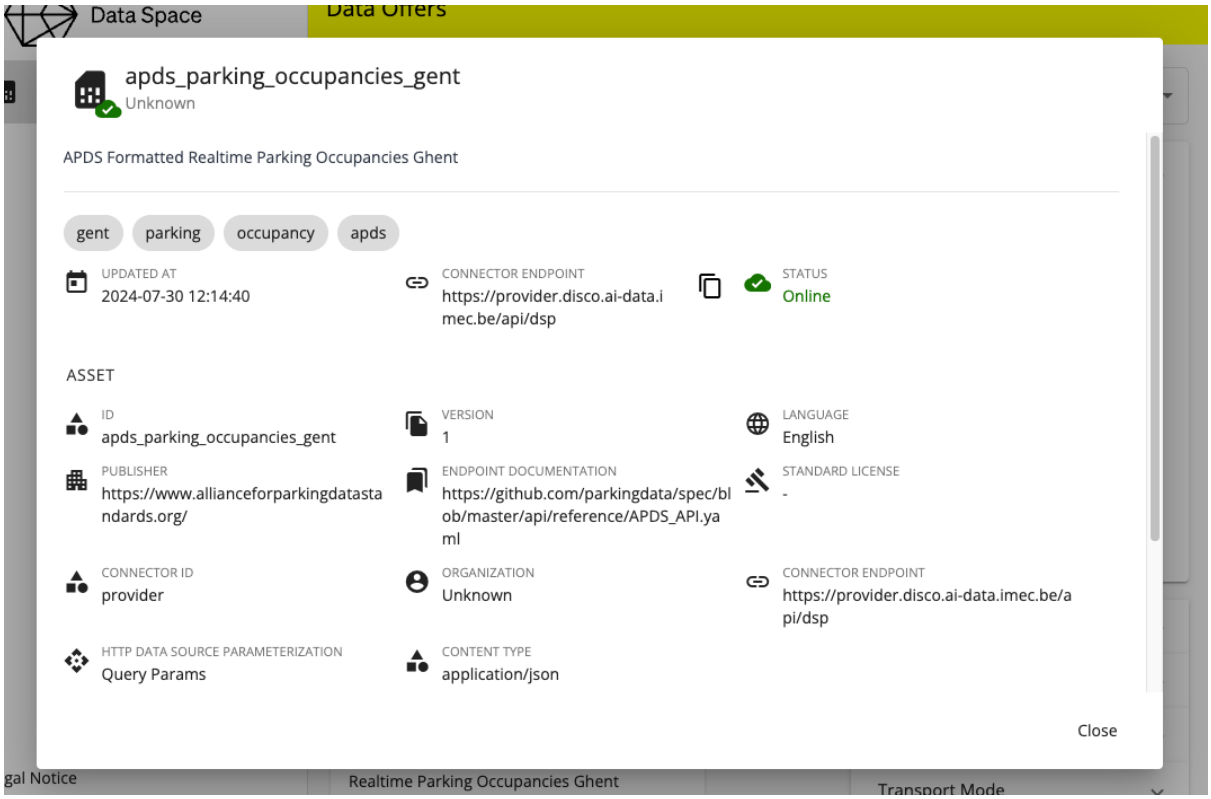
- Create Contracts for these Assets. Connectors provide Management API for these features[61] or also possible via the Connector UI[62]

- Fetch the catalog of the provider, this should return three previously created assets.

[60] https://github.com/sovity/edc-ce/blob/main/docs/getting-started/documentation/oauth-data-address.md
[61] https://github.com/sovity/edc-ce/blob/v10.4.2/docs/getting-started/documentation/parameterized_assets.md
[62] https://github.com/sovity/edc-ce/blob/v10.4.2/docs/getting-started/documentation/parameterized_assets_via_ui.md



Screenshot of Datasets



Screenshot of Datasets Catalog

- As a consumer, negotiate the contract. Once it is approved, it will be possible to start a transfer[63].
- The provider will send a token to the given webhook in the transfer request details. With that token, now it is possible to fetch the resource.

63 https://github.com/sovity/edc-ce/blob/v10.4.2/docs/getting-started/documentation/pull-data-transfer.md

Assets are configured to have dynamic query parameters, which allows for queries to be passed to Disco API.

For example:
https://provider.disco.ai-data.imec.be/public?name=Vrijdagmarkt of the APDS dataset asset would fetch all historic records for parking spots with given name. To fetch all, any query parameters can be omitted. Even though there is only one public endpoint per connector, correct asset resolution is based on the Authorization token provided.

8. Conclusion

In conclusion, DISCOLLECTION effectively streamlines the process for cities to contribute valuable data to the data space, enhancing accessibility and integration across urban data systems. Using an architecture that includes data retrieval from the cities' open data portals, transformation into a standardized model, storage in various platforms, and dissemination through the Urban Freight Data Space connector, DISCOLLECTION not only standardizes data but also facilitates its broader use and integration with other DISCO-X solutions and existing data platforms. This approach enables cities to participate in a unified data ecosystem, promoting better urban management and planning through the efficient sharing and utilization of critical geospatial and logistical information.

9. References

Gent Open Data Portaal:
https://data.stad.gent/explore/dataset/straatparkeerplaatsen-gent/information/
https://data.stad.gent/explore/dataset/bezetting-parkeergarages-real-time/information/

Dagster:
https://dagster.io/

Sovity Community Edition EDC:
https://github.com/sovity/edc-ce

Keycloak Identity and Access Management:
https://www.keycloak.org/

APDS:
https://www.allianceforparkingdatastandards.org/
https://github.com/parkingdata/spec

imec DISCOLLECTION Repository Deliverables:
DISCOLLECTION API: https://gitlab.com/disco-horizon-europe/open-software-repository/discollection-api
DISCOLLECTION ETL: https://gitlab.com/disco-horizon-europe/open-software-repository/discollection-etl
DISCOLLECTION K8S: https://gitlab.com/disco-horizon-europe/open-software-repository/discollection-k8s
DISCOLLECTION SDP: https://gitlab.com/disco-horizon-europe/open-software-repository/discollection-sdp

imec DISCOLLECTION Deployment URLs:
Dagster User Code: https://etl.disco.ai-data.imec.be/
DAPS: https://daps.disco.ai-data.imec.be/
Broker: https://broker.disco.ai-data.imec.be/
Provider: https://provider.disco.ai-data.imec.be/
Consumer: https://consumer.disco.ai-data.imec.be/
SDP: https://sdp.disco.ai-data.imec.be/

DISCOLLECTION APIs (private network – only accessible via provider connector):

http://disco-api/v1/parking-occupancies/
Retrieve parking occupancies of a given time window.

- `gte`: filter by timestamp greater than or equals
- `lte`: filter by timestamp less than or equals

http://disco-api/v1/parking-spots/
Retrieve parking spots

http://disco-api/v1/apds/places
Lists `Places` and allows searching/filtering by:

- `latitude`, `longitude`, `radius`: to check for elements in the `Place` hierarchy within a geographic area
- a modification instant (`modified_since`): to only display changed resources from that point onwards
- `layer`: maximum hierarchy element layer desired
- `type`: types of hierarchy elements * `right_type`: types of `RightSpecifications` associated with Place hierarchy elements
- `structure_type`: types of structure
- `structure_grade`: grades of structures
- `name`: name of facilities

Additionally, supports providing a comma-separated list of optional object attributes associated to a `Place` hierarchy element that should be included in the result set. (see `expand` attribute). If unused or left blank, `none` will be assumed.

Funded by the European Union

CIVITAS Sustainable and smart mobility for all

THE CIVITAS INITIATIVE IS CO-FUNDED BY THE EUROPEAN UNION

DISCO is a project under the CIVITAS Initiative.
Read more - civitas.eu