D5.3 Basic capability to enable asset transform

Grant Agreement nr	780470
Project acronym	SAUCE
Project start date (duration)	January 1st 2018 (36 months)
Document due:	June 30th 2019
Actual delivery date	June 28th 2019
Leader	DNEG
Reply to	William Greenly - wmg@dneg.com
Document status	Submission Version

Project funded by H2020 from the European Commission

Project ref. no.	780470
Project acronym	SAUCE
Project full title	**S**mart **A**sset re-**U**se in **C**reative **E**nvironments
Document name	D5.3 Basic capability to enable asset transform
Security (distribution level)	Public
Contractual date of delivery	June 30th 2019
Actual date of delivery	June 28th 2019
Deliverable name	Basic capability to enable asset transform
Type	Demonstration
Status & version	Submission Version
Number of pages	18
WP / Task responsible	DNEG
Other contributors	-
Author(s)	William Greenly - DNEG
EC Project Officer	Ms. Cristina Maier, Cristina.MAIER@ec.europa.eu
Abstract	A report that demonstrates and describes the architecture and capabilities of the transformation framework along with examples and documentation
Keywords	Transformation, pipeline, representation, ontology, vocabulary, reuse, distribution
Sent to peer reviewer	Yes
Peer review completed	Yes
Circulated to partners	No
Read by partners	No
Mgt. Board approval	No

Document History

Version and date	**Reason for Change**
1.0 01-06-19	document created by William Greenly
1.1 21-06-19	Version for peer review
1.2 28-06-19	Revisions in response to review: final version submitted to Commission

Table of Contents

1 EXECUTIVE SUMMARY

This document provides an overview of the basic transformation framework and its capabilities, a foundational part of work package 5.

We begin by providing some background to the work package and in particular draw the reader's attention to a number of other work packages and documents that provide a prelude to this document, along with documents that are in turn informed by this. Without a strong understanding of previous work packages, the ability to assess this work package is sorely diminished.

We then proceed to highlight the key objectives and methodology that we apply in order to demonstrate the framework, along with an overview of the solution, placing special emphasis on the need for real world working software and examples.

In the same vein as work package 4.1, we then proceed to provide an overview of the data architecture, drawing upon and extending core concepts from the aforementioned work package, providing the conceptual and ontological foundations by which we can achieve interoperability between our different components and contributions.

Finally we provide a detailed overview of the technical architecture and technical componentry, supplying examples by which we are able to demonstrate the capabilities qualities of the framework as a whole. By means of detailed guidance, instruction and examples, we prove that contributors can create transformation plugins and transformation pipelines that can harness and chain together transformation capabilities, exponentially increase the value and reusability of our assets. This also compliments and completes the work and foundations laid out in work package 5.2, by providing a platform to actually deliver and execute transformation capabilities.

2 BACKGROUND

This document is an integral part of WP5 and provides a framework for a number of other contributions. In particular it builds upon D2.3 and D4.1 and reuses many of the same components and building blocks in the search framework. To fully understand and appreciate the transformation framework, we must ensure we have an understanding of the generalised assets descriptor covered in D2.3 and the search framework covered in D4.1.

It also compliments and extends D5.2. D5.2 provided examples of how to advertise and chaining transformation capabilities, without specifying how they are executed. This work package provides a comprehensive framework for building, testing and deploying individual transformations and chained transformation pipelines.

This work package also provides a prelude to D4.2, whose delivery date coincides precisely with the delivery date of this work package. Proviso of this document has been made to the relevant partners in order to peer review D4.2

Finally, this work package has interdependencies with the storage framework in WP7. Whilst the details of this are not defined in the work package, an interface for a storage system is provided independent of any concrete implementation. In succeeding work packages we will investigate concrete integration with the storage provided in WP7.

3 INTRODUCTION

This chapter provides an overview of the transformation framework, re-establishes the goals and objectives of the framework and describes how we are able to demonstrate the capability of the framework and its suitability as a solution for the problem domain.

3.1 Main objectives and goals

The main objectives of this work package are to:

- Provide a framework by which contributors can build, test, share and deploy transformation software and tools.
- Provide a way in which transformation pipelines can be created and transformations can be chained and combined.
- Provide a means for integration with the search framework and storage framework

3.2 Methodology

To demonstrate that the framework satisfies the objectives and goals, we have created a proof-of-concept consisting of source code, binaries, and documentation which contain the following:

- A proof of concept platform, which extends the platform in D4.1, which in turn allows transformations to be created and chained.
- Several transformation actions which demonstrate and prove the platform's transformation and pipelining capabilities, along with integration with an abstract storage system
- A simple command line interface which demonstrates how transformations can be advertised to the search framework
- Comprehensive documentation covering the data and technical architecture.

For all of the components and building blocks listed in the course of this document, there is working software that demonstrates the description and narrative. There is nothing hypothetical or suppositional. It is all provable with working software.

3.3 Convention

This document contains many snippets of code, which are pre-formatted accordingly to the most suitable code style with a black background. In some cases bold italics of code snippets has been done for the purpose of highlighting of something important.

4 Solution Overview

The transformation framework builds upon and leverages much of the same technology building blocks as the Smart Search Framework. Transformation actions are responsible for executing transformations against asset representations and persisting results in both the storage and search. These are built, tested and published in exactly the same way as actions referenced in section 6 in WP4.1 and can be chained together into sequences, providing a way to explicitly create transformation pipelines.

By extending the core vocabulary, we introduce terms for describing different available asset representations and provide the means to determine what representations are suitable for which transformations. The vocabulary contains the means to describe the aforementioned transformation actions along with classes and instances of transformations that use these actions to produce outputs from various inputs, which meet the criteria defined in the new vocabulary. This in turn provides the means for a reasoner to infer transformations, implicitly chaining together transformations in different combinations, to achieve a graphs of transformation pipelines. By applying different classes and constraints of representation, we can determine the suitability of actions to perform transformations at different levels of quality.

5 Data Architecture

The transformation framework includes a vocabulary of terms and classes along with instances of available actions and transformations. This extends and builds upon both the core vocabulary included in D4.1 and the terms described in D5.2

5.1 Core Transformation Vocabulary

The core transformation vocabulary is created and packaged in exactly the same way as the core search vocabulary described in sections 5.1 and 5.2 in D4.1. To enable decoupling, versioning and greater reuse, it is provided as a separate library and prefixed as follows:

```
@prefix scet: <https://sauce.dneg.org/transformation/>.
<https://sauce.dneg.org/transformation//> a owl:Ontology;
    owl:versionIRI <https://sauce.dneg.org/transformation/0.0.1> ;
      dc:title "SAUCE Transformation Ontology" ;
      dc:abstract "Core set of terms for SAUCE Transformation
Framework";
      rdfs:comment "Core set of terms for SAUCE Transformation
Framework." .
```

Over the course of this document and their accompanying examples, we will add new available representations, actions and transformations. It is recommended that for all of these, we provide them with their own prefixes, especially in the case of transformation action parameters.

5.2 Available Asset Representations

One of the core concepts in the transformation vocabulary is the ‘Available Asset Representation’. This provides a way to describe and advertise to the search and transformation framework, the representations that an asset can attain, either implicitly or explicitly, by means of transformation. We use this as a basis to describe the necessary and sufficient conditions for transformation inputs, along with associated outputs, which can subsequently act as inputs in other transformations. The superclass for all the different types

of available asset representations is AvailableAssetRepresentation. Below is an example of specialisations of this class for different purposes:

```
scet:AvailableAlembicRepresentation a owl:Class;
      rdfs:label "AvailableAlembicRepresentation";
      rdfs:comment "Available Alembic Representation";
      rdfs:subClassOf scet:AvailableRepresentation.

scet:AlembicDistribution a owl:Class;
      rdfs:label "AlembicDistribution";
      rdfs:comment "A AlembicDistribution Distribution";
      rdfs:subClassOf  [
      a owl:Restriction;
      owl:onProperty dcat:mediaType;
      owl:hasValue "application/alembic";
      rdfs:subClassOf scet:AvailableAlembicRepresentation
      ].
```

In the example above we create an AvailableAlembicRepresentaton which is the class of things which can be represented as a file format Alembic. We also state that existing distributions which have a mediaType of “application/alembic” qualify as a member of this set of representations.

We can declare more specific sets of this representation which describe additional characteristics, using metadata or labels that were extracted during ingestion (see D4.1) or from information relating to the things they are depicting.

```
scet:AvailableAlembicLowScaleRepresentation a owl:Class;
      rdfs:label "AvailableAlembicLowScaleRepresentation";
      rdfs:comment "Available Alembic Low Scale Representation";
      rdfs:subClassOf [
      a owl:Restriction;
      rdfs:subClassOf scet:AvailableAlembicRepresentation;
      owl:onProperty sce:scale;
      owl:allValuesFrom [
            a rdfs:Datatype;
            owl:onDatatype xsd:integer;
            owl:withRestrictions ([xsd:maxInclusive 0.1])
      ]

      ].
```

In the example above we use data about scale to describe representations that qualify as being low scale Alembic representations. This is particularly important since some

transformations require certain inputs with certain characteristics other than the file format, in order to perform a transformation. Furthermore many transformations also result in some kind of lossiness or change, and so have a way to describe what level of loss has occurred in a transformation is particularly important, especially in the case of chained transformations and convoluted transformation pipelines. Any aspect or characteristic of the asset can be described here, including its depiction, provenance, keywords etc. along with those characteristics specific to the representation, file or data type.

5.3 Transformation Actions

Transformation actions are classes of actions which can actually operate on data and provide transformation capabilities. They are synonymous with actions defined in D4.1 and are packaged and deployed as OpenWhisk actions.

```
usdcat:USDCat a owl:Class;
      rdfs:subClassOf scet:Action;
      rdfs:label "USDCat";
      rdfs:comment "usdcat actions".

usdcat:alembic2usd a usdcat:USDCat;
      dc:title "alembic2usd";
      dc:description "call usd cat action";
      rdfs:label "usdcat".
```

In the example above we specify a class of action called usdcat actions and create and instance of this action. This action will later be referenced as part of a transformation. For different classes of action we can also specify associated parameters along with their range, using standard OWL properties with domain and range attributes.

```
dneg:scale a owl:DatatypeProperty;
      rdfs:label "scale";
      rdfs:domain dneg:ScaleReduction;
      rdfs:range xsd:decimal.
```

The example above demonstrates a parameter that can be used as part of the ScaleReduce action. The domain can include a union of actions and the range provides a means to describe the data type range for that parameter, which in the example above is a decimal, but could also be any range of decimal or literal types.

5.4 Transformations

Transformations are the processes by which we can derive new representations and new assets from existing representations and assets. Like actions, we create classes of

transformations and instances of transformations with specific characteristics. Below is an example of a transformation:

```
scet:AlembicToUSDTransformation a owl:Class;
      rdfs:subClassOf scet:Transformation;
      rdfs:label "AlembicToUSDTransformation";
      rdfs:comment "Alembic to USD transformations".

scet:defaultAlembicToUSDTransformation a
scet:AlembicToUSDTransformation;
      dc:title "Transformation to convert an Alembic to a USD";
      scet:action usdcat:alembic2usd;
      scet:input [
      a scet:AvailableAlembicLowScaleRepresentation
      ];
      scet:output [
      a scet:AvailableUSDRepresentation
      ].
```

In the example above we create a transformation which uses the usdcat action, and requires an asset in an AvailableLowScaleAlembic representation and produces a AvailableUSDRepresentation. The inputs and outputs are both described as instances of anonymous assets, but we could provide specific identifiers if specific assets are required, along with any other criteria relevant, outside the context of the AvailableRepresentation.

Declaring transformations and associated actions gives the framework a way of advertising them to the search framework, along with then providing the means for executing them, which we will cover in the next section.

6 Technical Architecture

The section provides by way of example, guidelines and best practices for creating transformations and transformation pipelines in the framework. Much of this inherits and borrows the same patterns from D4.1 (Smart Search Framework), and builds on D5.2, so having a good understanding of both is mandatory.

6.1 Creating the Transformation Framework

The underlying transformation framework runs on exactly the same platform as the search and classification framework, namely openwhisk. To ensure isolation of transformations from ingestion and classification actions and resources, we have created namespaces for search and transformation respectively and associated the plugins demonstrated in D4.1 with the search namespace, and latterly, transformations with the transformation namespace.

As per the search framework, transformations are packaged and delivered to the framework as openwhisk actions. In the examples in D4.1, we demonstrated how to create Java actions and deliver them into the framework. Since many transformations use a variety of programming runtimes (python, C), we will demonstrate how to create a Docker action with multiple runtime and deliver it into the framework, taking advantage of several of these runtimes in a single action.

To facilitate this process, the framework supplies a two base images that can be extended, which enable transformations to be executed in the OpenWhisk ecosystem:

- openwhisk/dockerskeleton
- sauceproject/baseaction

The former is a small Alpine based image for low dependency, lean actions, whilst the latter is an Ubuntu based image, with a number of additional dependencies for more common command line transformations. Any Docker actions we create should extend either of these two images, since both images include components that manage the interaction with OpenWhisk framework and without these, the action cannot be executed.

Over time, new images derived from the images above can also be used that extend these. These are location in the ‘platform’ directory in the solution and in addition to the base action, currently include:

- sauceproject/usd (an image with usdcat installed)

6.2 Creating Transformation Actions

In this section we will demonstrate how to create, package, publish and register transformation actions. We will also discuss the relationship between transformations and and the asset storage and transformations and the search framework.

6.2.1 Creating a transformation

The transformation we will demonstrate uses the usdcat command line tool provided as from the USD Tool Suite provided by the USD team, along with the Alembic Plugin. This a command line tool which can apply a variety of file format transformations. The toolset itself has no available binary distributions and must be compiled and installed from scratch on the host runtime. To this effect we have created a Docker extension to our sauceproject/baseaction called sauceproject/usd, which has all the necessary components installed into the docker image.

We will now create our new action in the repository

- sauce-action-transformation-alembic2usd

A docker action consists of 2 parts; the first is the docker image which is used as the container to execute the action, the second is the bash script executable.

```
FROM sauceproject/usd
COPY runtime/ /usr/share/groovy/lib
```

The example above contains a Dockerfile which includes the dependencies specifically to run this action, along with a gradle file and build scripts to get the dependencies and build and publish a docker image.

The ‘exec’ file contains everything that is executed when openwhisk runs the action.

```
#!/bin/bash

# Extract parameters
echo "$1" > "params.json"
uuid=`jq '.scet:uuid' params.json | sed -e 's/^"//' -e 's/"$//'`
asset=`jq '.scet:asset' params.json | sed -e 's/^"//' -e 's/"$//'`
distribution=`jq '.scet:distribution' params.json | sed -e 's/^"//' -e
's/"$//'`
transformation=`jq '.scet:transformation' params.json | sed -e 's/^"//'
-e 's/"$//'`

# Download Data
mkdir /data
wget -O /data/input.abc "${distribution}?token=f9403fc5f537b4ab332d"

# run transformation
usdcat /data/input.abc --out /data/output.usda
sleep 4

#upload data
hst=`host storage`; if [ ${#hst} -gt 0 ] ; then store="storage"; else
store="localhost"; fi
upload=`curl -X PUT -Ffile=@/data/output.usda
"http://${store}:25478/files/${transformation}-${uuid}.usda?token=f9403f
c5f537b4ab332d"`

#update search
./update.groovy $1

#return result
if [ -f "/data/output.usda" ]; then echo "{ \"result\": \"transform\"
}"; else echo "{ \"result\": \"error\" }"; fi
```

In the example above, the script parses parameters sent to the action, downloads the relevant files for transformation, runs the actual transformation using the usdcat command

line tool, uploads the output to the storage and then updates the information about the new distribution and available representation in the search.

6.2.2 Transformation Parameters

In the above transformation, the only parameters passed in are core params, common for every transformation, however we can specify more, ensuring that the appropriate prefix is appended to the parameter relative to the action. Section 5.3 describes how to associate parameters with actions.

```
{
"scet:distribution":"http://storage:25478/files/knife.abc",
"scet:asset":"https://sauce.dneg.org/entities/testAsset",
"scet:transformation" : "defaultAlembic2usd",
"scet:uuid" : "0de07e4e-cbfe-40e9-a047-8711b3008e92"
 }
```

Prefixing parameters this way is very important, especially when transformations are chained together (as discussed in section 6.3), so that we can distinguish between different parameters for different transformation actions.

6.2.3 The relationship between a transformation and the asset storage

In the above example, we are using a simple open source storage server as the physical asset storage, however it is anticipated that any number of different storage mechanisms can be used, and later in this programme, we will integrate this with the storage platform deliver. To ensure decoupling the underlying storage from the transformation framework, we have provided a number of different interfaces (Groovy, HTTP, etc.) through which the framework interacts with the storage platform. This allows us to swap out storage technologies without having to update all the transformation actions.

6.2.4 The relationship between a transformation and the search framework

The transformation is also responsible for updating the asset, describing the newly created available representation and advertising to the search and transformation framework for use or further transformation.

```
#!/usr/bin/groovy
import com.jiolabs.ld.*
import java.text.SimpleDateFormat

def slurper = new groovy.json.JsonSlurper()
def params = slurper.parseText(args[0])

def mdl = new LdModel()
LdDatastore ds = new LdDatastore("http://search:3030/sauce")
```

```
mdl.datastore = ds
mdl.uri = params["scet:asset"]
mdl.add ("""
@prefix sce: <https://vocabularies.sauce.dneg.org/core/>.
@prefix scei: <https://sauce.dneg.org/entities/>.
@prefix scet: <https://sauce.dneg.org/transformation/>.
@prefix dcat: <http://www.w3.org/ns/dcat#>.
@prefix prov: <http://www.w3.org/ns/prov#>.
""")

mdl.get()
def date = new Date()
def df = new SimpleDateFormat("yyyy-MM-dd")
def tf = new SimpleDateFormat("HH:mm:ss")

mdl.add ("""

<${params['scet:asset']}> dcat:distribution scei:${params['scet:uuid']}.
      scei:${params['scet:uuid']} a dcat:Distribution.
      scei:${params['scet:uuid']} a scet:AvailableUsdRepresentation.
      scei:${params['scet:uuid']} dcat:downloadUrl
<http://storage:25478/files/${params['scet:uuid']}.usda>;

      prov:wasGeneratedBy ${params['scet:transformation']};
      prov:wasDerivedFrom <${params['scet:distribution']}>.

""")

mdl.put()
```

In the above example we create a new distribution and file representation referencing the resource which we created in the asset storage.

6.2.4.1 Provenance

Along with creating information about new representations it is also vitally important to ensure we persist provenance information (its origin) about any newly created representation or asset. In the above example we define the transformation that the asset was derived from along with the transformation that derived in. Obviously multiple assets or transforms can be used to generate new data.

6.2.4.2 Updating an asset vs creating a new one

In the example above we created a new available representation and distribution and associated it with the original asset. However there are times when one or more assets are used to create a new asset which is semantically different and on these occasions, a new asset should be created altogether, with provenance associated with the new asset, referencing all the associated derivations. Whether this creation is handled by the

transformation action, or is delegated to the ingestion and classification framework, is entirely down to the implementation.

6.2.5 Testing and publishing the transformation action

We can test the action by building it and deploying it to the framework in isolation. We do this by first zipping up the action with all its necessary dependencies and then deploy it.

```
#!/bin/bash
. ./version.sh

# mock data
./ingest.groovy
curl -X PUT -Ffile=@knife.abc
"http://localhost:25478/files/knife.abc?token=f9403fc5f537b4ab332d"
./build.sh

#run
wsk -i action update usdcat ${PACKAGE}-${VERSION}.zip --memory 512
--docker sauceproject/${PACKAGE}:${VERSION}
wsk -i action invoke usdcat -P testData.json --blocking

#test
curl -I
http://localhost:25478/files/4321.usda?token=f9403fc5f537b4ab332d
```

To test it, we first create some test data, then we invoke the action and then we test that a new resource has been created.

6.2.6 Publishing, registering and deploying the transformation action

Publishing the above transformation action follows exactly the same conventions as those described in D4.1 for publishing actions. It is important that in this scenario both the zip file containing the action executables and the docker image are correctly versioned and published to their respective repositories.

To register the action into the framework, we need to ensure that the vocabulary described in section 5, includes the action and the transformation, so that the transformation capabilities are advertised to the search, and made available to framework.

The final step is to now deploy the actions into the openwhisk framework, being especially careful to ensure that the action names correlate with those found in the transformation vocabulary.

6.3 Transformation Pipelines

In many VFX studios, a series of transformations are applied to assets at different stages in the VFX production. These are known as transformation pipelines and use a series of explicit (or implicit) transformations to achieve results. In D5.2, we demonstrated how to advertise chaining capabilities to the search framework, in this section we will demonstrate how to create transformation pipelines and register them into the transformation framework, so that those capabilities can be realised.

6.3.1 Creating a transformation pipeline

In order to demonstrate a pipeline we will create a new action and transformation, in exactly the same way as we did in section 6.2. This transformation takes an Alembic and reduces its scale to 10% of the original and then outputs the result to a new Alembic. This is achieved using the Python PyAlembic library, however we will build and publish the action in exactly the same way as the usdcat action in the previous section, building and publishing an exec.zip and docker image to the respective repositories.

We will register the action in openwhisk in exactly the same way as the previous example, save that in this instance we register two transformation actions and transformations, one to describe transforming from Alembic to low scale Alembics and the second to describe the pipeline of going from Alembic to low scale Alembic to USD.

```
dneg:ScaleReduction a owl:Class;
      rdfs:subclassOf scet:Action;
      rdfs:label "ScaleReduction";
      rdfs:comment "ScaleReduction actions".

dneg:scale a owl:DatatypeProperty;
      rdfs:label "scale";
      rdfs:domain [
       owl:unionOf (dneg:ScaleReduction dneg:ScaleReductionUSDCat)
      ];
      rdfs:range xsd:decimal.

dneg:largeScaleReduction a dneg:ScaleReduction;
      dc:title "largeScaleReduction";
      dc:description "A large scale reduction";
      rdfs:label "largeScaleReduction";
      dneg:scale 0.1.

dneg:ScaleReductionUSDCat a owl:Class;
    rdfs:subClassOf scet:Action;
    rdfs:label "ScaleReductionUSDCat";
```

```
    rdfs:comment "An Alembic Scale Reduction USD Cat pipeline".

dneg:alembic2lowScaleAlembic2USD a dneg:ScaleReductionUSDCat;
    dc:title "alembic2lowScaleAlembic2USD";
    dc:description "An alembic to low scale Alembic to USD"

scet:defaultAlembicToLowScaleAlembicToUSDTransformation a
scet:AlembicToLowScaleAlembicToUSDTransformation;
       dc:title "defaultAlembicToLowScaleAlembicToUSDTransformation";
    scet:action dneg:alembic2lowScaleAlembic2USD;
    scet:input [
      a scet:AvailableAlembicRepresentation
    ];
    scet:output [
      a scet:AvailableUSDRepresentation
    ].
```

Finally we now deploy the pipeline to the openwhisk framework as a sequence of actions, similar to the ingestion sequence defined in D4.1. We can specify default parameters or have one transformation action adjust the parameters before passing on to the next, creating a form of 'transformation currying'.

```
wsk action update -i alembic2lowScaleAlembic2USD --sequence
largeScaleReduction, alembic2usd
```

Executing this sequence will run both transformations saving respective representations to the storage and updating the search.

6.3.2 Implicit vs explicit transformation pipelines

In the example above we created a two step transformation pipeline using an openwhisk sequence, explicitly defining the sequence of transformations to apply. However since we have registered each transformation into the framework independently of the pipeline, it should be possible to use these with other transformations and create implicit pipelines. The sauce command line interface shows how we can interrogate the search service to materialise transformation options for an asset:

```
./sauce transformation [asset-id]
```

It should be noted that there can often be more than one way to get from one representation to another and the framework will materialise all the different ways, so long as the inputs and outputs qualify according to the necessary and sufficient conditions outlined in section 5. Quality and lossiness considerations should be manifested here in order to guide the correct outcomes.

6.3.3 Deterministic versus nondeterministic pipelines

This document contains the means by which transformations and transformation pipelines can be predetermined, either explicitly or implicity, but it is equally likely that many unforeseen transformation options will only materialise as by products of other transformations not directly chained or inferred. Furthermore, we have placed a strong emphasis on the need to record and store provenance information about transformations which in the long run provide a rich and meaningful datasets by which training models can be created, allowing us to infer optimised and more suitable transformation options using machine learning.

7 Conclusion

It should be clear from this document and examples supplied that the transformation framework provides an extensible and scalable platform that supports a variety of runtimes and allows us to chain and combine transformation software and tools to create a range of outputs which can further provided inputs, exponentially increase the value of our assets and our software. We also introduced the notion of asset storage and provided a basic interface to a storage mechanism. In later work packages and during the course of the project we will start to provide concrete implementations with the work being done in work package 7. Finally by showing how materialisations in the transformation framework can affect materialisations in the search framework, we can demonstrate that both frameworks can complement each other to enrich our assets and provide better results.

8 Web references

RDF
https://www.w3.org/standards/techs/rdf

RDFS
https://www.w3.org/TR/2014/REC-rdf-schema-20140225/

OWL2
https://www.w3.org/TR/owl2-overview/

SPARQL1.1
https://www.w3.org/TR/sparql11-query/

OpenWhisk
https://openwhisk.apache.org/

USD Toolset
https://graphics.pixar.com/usd/docs/USD-Toolset.html

Alembic
https://www.alembic.io/

PyAlembic
https://docs.alembic.io/#pyalembic