D4.1 Smart Search Framework

Grant Agreement nr	780470
Project acronym	SAUCE
Project start date (duration)	January 1st 2018 (36 months)
Document due:	31st December 2018
Actual delivery date	31st December 2018
Leader	Peri Friend
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	SAUCE
Document name	SAUCE D4.1
Security (distribution level)	Public
Contractual date of delivery	31st December 2018
Actual date of delivery	31st December 2018
Deliverable name	D4.1 Smart Search Framework
Type	Demonstration
Status & version	submission version
Number of pages	32
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 smart search framework along with a proof of concept graphical user interface
Keywords	search, ontology, vocabulary,taxonomy, framework, gui, descriptor, resuse, similarity, comparison
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**
0.0 26-11-18	template created by Peri Friend
1.0 14-12-18	First draft for peer review by William Greenly
1.1 24-12-18	Final draft for submission

Table of Contents

1 EXECUTIVE SUMMARY

This document contains a detailed, comprehensive and concrete overview of a proof of concept for the Smart Search Framework. Although the scope of the deliverable is proof of concept , it must be emphasised and stated that there are concrete working examples of all the components detailed in this document, that materially and tangibly prove the concept and corroborate this report.

We start by providing some background to the Framework; the key drivers and use cases; relevant research and deliverables from the SAUCE program that inform this deliverable; the main objectives and goals; and the methodology applied that will prove and demonstrate the concept thereafter.

Following on from this, we then proceed to give an overview of the framework, how all the components interact from end-to-end and ultimately compliment and inform a search. Without this overview, individual contributors will not have the necessary context to comprehend more detailed aspects of the system.

Moving on, we provide detail around the data architecture. Building on the concept of a generalised asset descriptor in D2.3, we provide concrete examples of the core concepts, ontologies and vocabularies, with accompanying usage along with means of extension and alignment with other vocabularies, or those which evolve over the course of the project. It must be noted that whilst vocabularies and ontologies are extensible, the underlying model theoretic semantics provided by RDF, is canonical and provides a consistent means by which we can search, filter and unify.

This provides a suitable interlude to the next section, which details the technical architecture of the platform, the components and archetypes along with guidelines for building, testing and deploying additional components into the framework. Since this is a framework, the focus is around interoperability between components, and this is demonstrated by the adherence to a common interface and canonical data model. We provide guidelines for building and contributing classifiers, enrichers, reasonsers and predictive models and describe how data passes through all these components, finally updating the search engine. To demonstrate the completeness of the framework and ingest system, we detail how the search API works and demonstrate how it can be used for a variety of use cases along with advanced search features.

In the final section we present a user experience architecture which visually and intuitively provides a set of extensible patterns, paradigms and modalities for a human to interact with the framework and API's along with a means to search, browse, navigate and filter across all the domains and concepts previously detailed, and any extensions that might be developed over the course of the project.

2 BACKGROUND

This deliverable builds upon work package 2.1 and 2.3 respectively, but its scope and importance within the project should not be underestimated.

2.1 describes some of the key use cases and requirements for the smart search framework and provides a good basis for the problem domain.

This is elaborated further in 2.3, which focuses on the need for a generalised asset descriptor, solely for the purposes of search and discovery, along with technical building blocks to support it.

This deliverable builds on both work packages and also tries to incorporate the needs and aspirations of the consortia as whole. It should be noted that many other deliverables will be informed, or build upon the smart search framework, including work package 5.

The scope of this deliverable was to demonstrate a proof of concept using one asset type, however it should be obvious and easily apparent, that the architecture can support any number of asset types and support the extension of functionality across those types. Additionally it should be noted that the advertisement of transformation capabilities, whilst part of the framework, is not demonstrated in this report, rather the next one.

3 INTRODUCTION

This chapter provides an overview of the framework, re-establishes the goals and objectives of the framework and describes how we are able to demonstrate the capability of the framework, its suitability as a solution for the problem domain, described in D2.1 and D2.3, and the core concepts that accompany it.

3.1 Main objectives and goals

Description of the main objectives and goals of the framework are as follows.

- To provide a uniform method by which 3rd party asset managements systems can publish data to the search framework for ingestion, classification, transformation and search
- To provide a framework and data architecture which is extensible and interoperable with 3rd party VFX ontologies and domains, and supports tagging, classification and reasoning across those domains.
- To provide a framework by which 3rd parties can write smart classifiers and enrichers that execute at ingestion time to enrich data from asset management systems and extend and expand upon the aforementioned domains and ontologies.
- To provide programmatic data API's to update, search and query asset data
- To provide a proof of concept graphical user interface and user experience architecture that demonstrates the entire search and transformation experience from end to end and can be extended over time.

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, high fidelity wireframes and documentation which contains the following

- A proof of concept platform, built on open source components, that runs locally, and functionally demonstrates ingestion and search from end to end.
- A collection of core plugins that demonstrate the capabilities of the framework and its extensibility to 3rd parties and consortia contributors.
- A data API supporting SPARQL query, update, federation and unification of different query logics such as similarity, comparison and compatibility
- A set of high fidelity wireframes and user experience architecture, alluding to different user interface modalities, paradigms, components and patterns, satisfying a variety of use cases.

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 Terminology

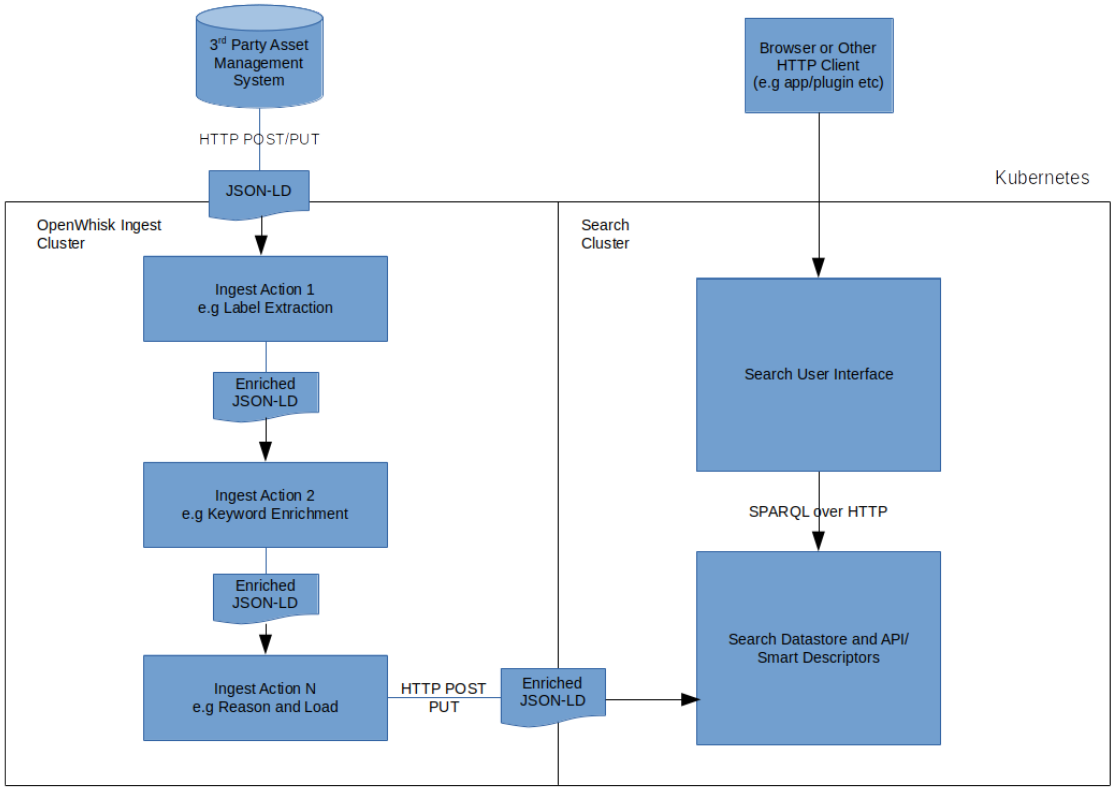
Library: A unit of software, with a runtime, that can be reused across different applications
Action: An OpenWhisk function or micro-application which performs a task in the framework
Ontology/Vocabulary: very specifically a formalised model of knowledge, represented in the framework using Web Ontology Language, RDF Schema or the Simple Knowledge Organisation System.
CURI: A Compact Uniform Resource Identifier
SPARQL: Sparql and RDF query language (a recursive acronym). A graph based query language.

3.4 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 highlighting of something important.

4 Framework Overview

Before we detail aspects of the data and technical architecture, it is first necessary to understand how the solution as a whole operates and hangs together. Below is a diagram which provides an overview of the different components and archetypes and how they interact, followed by an explanation:



Data is ingested into the framework over HTTP via POST, PUT and PATCH operations as JSON-LD resources (documents).

The framework provides a way to define URLs for different types of asset and different types of asset facet, and dependent on the type of asset and the url, the JSON-LD resource gets decorated via a series of actions. These actions are micro-applications which perform a variety of tasks ranging from metadata extraction, classification and enrichment. Some of these actions are provided as core components of the framework (general purpose reasoning, keyword enrichment, data management) and some are provided by other members of the consortia (smart classifiers for specific asset types).

The final operation that the system performs on the asset metadata store is dependent on versioning conventions, URI and 3rd party/independent surrogate identifier guidelines, in conjunction with the original HTTP method.

Asset metadata is partitioned in the data store and exposed via a standard SPARQL interface, supporting query, federation, graphstore protocols, update and additional extensions for full text and Lucene based searches.

More complex search filters for comparison, similarity or other type specific predicates, can be incorporated into the SPARQL interface as SPARQL function extensions implementing a standard interface.

An HTML based search user interface interacts with the SPARQL using the HTTP based SPARQL protocol to search for matching assets. This user interface allows navigating and browsing assets in an intuitive, consistent and extensible way.

To this effect, we have provided an overview of the data architecture of the framework, the technical architecture and the user experience architecture. The data architecture describes the core ontologies and data model used through the ingestion and search, along with the means to extend and align with other ontologies. The technical architecture illustrates the core features and archetypes of the framework, how to contribute features in the form of classification, enrichment and search plugins, and finally how the framework can be deployed and run. The user experience architecture provides extensible componentry and patterns for interrogating the data and search framework.

5 Data Architecture

The deliverable D2.3 highlighted the requirement for a generalised asset descriptor for search purposes. In this specification, a recommendation was put forward for a canonical data format and common data model. The basis of this model was RDF (Resource Description Framework), with domain vocabularies represented using RDFS (Resource Description Framework Schema) and OWL (Web Ontology Language), and serialised as JSON-LD. It is this serialised document that is passed through the ingestion process, with different micro applications (classifiers, transformers, etc) decorating and enriching it accordingly, all the way through to the point that is persisted in the search engine.

JSON-LD is the only supported serialisation, since it is both OpenWhisk conformant and RDF conformant. However, all JSON-LD sub-serialisations are supported (compact, embedded, flattened, etc).

An Ingest document can contain information about one or more assets, identified using combinations of URIs originating from one or more systems. In addition to this, T-Box data can also be included, representing either logical rules, which should be applied during ingestion, or additional properties derived from enrichment. To enable a better understanding of the Generalised descriptor, the framework includes:

- A set of core vocabularies.
- A SHACL (Shapes Constraint Language) document explaining the shape of an ingest document.
- A breakdown of the anatomy of an ingest document.

5.1 Core Vocabularies and Ontologies

The framework includes a set of core vocabularies that contain terms to describe the core concepts of an ingest document. This includes terms for keywords, depictions, taxonomic information, actions, provenance, etc. This is represented in a TURTLE resource, source controlled in a GitHub repository and published to a Maven binary. Additions and addendums to this can be built, tested and published.

The prefixes used in these projects are important as they will also determine prefixes to form CURI's at query time and during ingestion. Below is a snapshot of the prefixes and ontology names:

```
@prefix wdt: <http://www.wikidata.org/prop/direct/>.
@prefix wd: <http://www.wikidata.org/entity/>.
```

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix google: <http://schema.org/>.
@prefix dct: <http://purl.org/dc/terms/>.
@prefix dc: <http://purl.org/dc/elements/1.1/>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix skos: <http://www.w3.org/2004/02/skos/core#>.
@prefix dcat: <http://www.w3.org/ns/dcat#>.
@prefix prov: <http://www.w3.org/ns/prov#>.
@prefix three: <http://3dmodelontology.org/>.
@prefix exif: <http://www.cipa.jp/>.
@prefix jpeg: <https://www.jpeg.org/jpeg/>.
@prefix nikon: <https://www.nikon.com/>.
@prefix wns: <http://www.w3.org/2006/03/wn/wn20/schema/>.
@prefix wni: <http://www.w3.org/2006/03/wn/wn20/instances/>.
@prefix ivy: <http://ivy.dneg.com/>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
```

5.2 Sauce Ontology and Contributing to the Core

In addition to the third party vocabularies, there is a core vocabulary for SAUCE with terms for VFX and SAUCE specific concepts. In addition to this, there is also a vocabulary of SAUCE data instances and SAUCE custom / dynamic vocabularies.

```
@prefix sce: <https://vocabularies.sauce.dneg.com/core/>.
@prefix scei: <https://sauce.dneg.com/entities/>.
@prefix scec: <https://vocabularies.sauce.dneg.com/custom/>.
```

To add terms to the SAUCE vocabulary, simply update the resource inside the following project, version accordingly, and re-publish. Section 6 describes the common software development lifecycle tasks.

To create a new ontology or vocabulary, the process is the same, but the new vocabulary should also be referenced / prefixed in the core vocabulary for it to be advertised across applications. e.g:

```
@prefix sce: <https://vocabularies.sauce.dneg.com/core/>.
@prefix scei: <https://sauce.dneg.com/entities/>.
@prefix scec: <https://vocabularies.sauce.dneg.com/custom/>.
@prefix scea: <https://vocabularies.sauce.dneg.com/animation/>.
```

5.3 Anatomy of the Generalised Descriptor and Ingest Document

As previously mentioned, the core vocabulary project contains a SHACL document describing the data shape of an ingest document. However, it is useful to explain and draw out some of the concepts of the ingest document, so that contributors and consortia members can correctly decorate the document with results from classifiers and enrichers.

Appendix 1 shows a complete asset with all the terms described below, including with prefixes. For clarity, each of these is explained below:

5.3.1 Context

All prefixes and namespaces are declared in the context fragment of the JSON-LD document. All the examples below use the Compact IRI document form for JSON-LD.

```
{
"@context": {
       "sce": "https://vocabularies.sauce.dneg.org/core/",
       "scei": "https://sauce.dneg.org/entities/",
       "scec": "https://vocabularies.sauce.dneg.org/custom/",
       "wdt": "http://www.wikidata.org/prop/direct/",
       "wd" : "http://www.wikidata.org/entity/",
       "rdfs" : "http://www.w3.org/2000/01/rdf-schema#",
       "rdf" : "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
       "xsd" : "http://www.w3.org/2001/XMLSchema#",
       "google" : "http://schema.org/",
       "dct":"http://purl.org/dc/terms/",
       "dc":"http://purl.org/dc/elements/1.1/",
       "foaf": "http://xmlns.com/foaf/0.1/",
       "skos": "http://www.w3.org/2004/02/skos/core#",
       "dcat": "http://www.w3.org/ns/dcat#",
       "prov" : "http://www.w3.org/ns/prov#",
       "3d" : "http://3dmodelontology.org/",
       "exif" : "http://www.cipa.jp/",
       "jpeg" : "https://www.jpeg.org/jpeg/",
       "nikon" : "https://www.nikon.com/",
       "wns" : "http://www.w3.org/2006/03/wn/wn20/schema/",
       "wni" : "http://www.w3.org/2006/03/wn/wn20/instances/"
      }
}
```

5.3.2 Identifiers, Asset Types and Asset Relationships

An asset ingested into the system can include a number of identifiers, which either identifies the resource in the search system, or references the resource in the external asset management system. The presence of an identifier, in the context of the search system, is important during ingest since it helps determine the operation that is performed at write time. The omission of an search identifier results in a new one being issued.

```
{
  "@id": "scei:asset-12345-67890",
  "@type": "sce:Mesh || sce:Rig ||sce:ImageSequence || sce:Animation || etc.",
  "dct:references": {
    "@id": "ivy:asset-id-in-proprietary-system-1234"
   },

}
```

The generalised descriptor facets assets into 3 types, namely the type of thing its depicting e.g cars, planes, guns, crowds, people, skeletons etc., its type in the context of the Visual FX pipeline (e.g animation, rigging, image sequence etc.) and its type in the context of the available file representations of that asset (e.g maya, alembic, fbx etc.). There its possible to describe an asset as a mesh, depicting a car, available as a maya file. The root node is therefore the type in the Visual FX pipeline and is illustrated in the example above.

5.3.3 Provenance, Derivation and Revision

Provenance data can be attached to any node in the asset graph, to help identify revisions of any entity in the graph, along with those in 3rd party systems. Provenance data can also be used to trace data decoration through ingestion. In the example below, data about the original record in the 3rd party asset management system is used to understand the revision history, derivatives and alternatives of this resource . This data will form a larger provenance graph as more assets are ingested into the framework.

```
{
  "@id": "scei:asset-12345-67890",
  "@type": "sce:Mesh || sce:Rig ||sce:ImageSequence || sce:Animation || etc.",
      "dct:references": {
         "@id": "ivy:asset-id-in-proprietary-system-1234",
         "prov:wasDerivedFrom" : {
           "@id" : "ivy:1234",
          },
         "prov:wasRevisionOf" : {
           "@id" : "ivy:45678"
          },
         "prov:alternateOf" : {
           "@id" : "ivy:9101112"
          }
     }
}
```

In addition to the above revision and derivation history, any node can also include date/time stamps of its creation and modification using terms from dublin core.

```
{
   "dct:issued": "datetime created",
   "dct:modified": "datetime modified"
}
```

5.3.4 Copyright and Licensing

Like provenance data, copyright and licensing information can also be attached to any node in the asset graph. Dublin Core already holds enough terms to describe copyright and license. The actual license and its implications for reuse should be managed either in the core vocabularies or as part of the search logic.

```
{
  "dct:license": {
     "@type" : "dct:LicenseDocument",
```

```
    "@id" : "scei:Proprietary"
  },
  "dc:rights": {
    "@type" : "dc:RightsDocument",
    "@id" : "scei:ResuableAcrossFranchise",
    "dc:title" : "Reusable across franchise"
  }
}
```

5.3.5 Available File Representations

As mentioned previously, another important facet of an asset is the information about different available physical file representations of an asset, along with how to download any available samples.

```
{
"@id": "scei:asset-12345-67890",
"@type": "sce:Asset",
"dcat:distribution": [{
            "@id" : "sce:dist-uuid-1234",
            "@type" : "dcat:Distribution",
            "dct:title" : "JPEG Compressed Collection of Images",
            "dct:description": "The asset as a collection of JPEG images",
            "dcat:accessUrl" : {
              "@id": "http://third-party-system/12345"
            },
            "dcat:downloadUrl": {
              "@id" : "http://third-party-system/12345/jpeg.zip"
            },
            "sce:sample" : {
              "@id" : "http://sample-url-for-introspection",
               "dcat:mediaType": "image/jpeg",
              "dct:format" : "image/jpeg"
            },
            "dct:format" : "application/gzip",
            "dcat:mediaType": "image/jpeg",
            "dcat:byteSize" : 12345678,
            "jpeg:Compression-Type" : "Baseline"

        },
        {
            "@id" : "sce:dist-uuid-4567",
            "@type" : "dcat:Distribution",
            "dct:title" : "Nikon Electronic Format",
            "dct:description": "The original Nikon Electronic Format",
            "dct:format" : "application/gzip",
            "dcat:mediaType": "image/x-nikon-nef",
            "dcat:byteSize" : 12345678,
            "nikon:Color-Mode" : "COLOR",
            "nikon:Sharpening" : "AUTO"
        }]
```

```
}
```

media types and file sizes can also be represented. The access url specifies a url where information about the file can be read (i.e an HTML web page in the 3rd party asset management system), whilst the download url is a dereferenceable URL for the actual resource. Most importantly, there is also support for sample URLs to provide data for classification engines and enrichers. These are important since in many instances classifiers only require a select subset of data for introspection, not entire distributions. Finally, chapter 5 describes how metadata can be attached to individual distributions using enrichment plugins. The example above demonstrates this using data extracted and decorated using the OpenImageIO action.

5.3.6 Keywords and Tags

Keywords and tags should be represented using controlled, but extensible vocabularies, and can be applied, as with provenance data, to any node of the graph using 'subject' predicates. The framework leverages Wordnet synsets and associations to help facilitate a semantic search and provides a superficial classification system.

```
{
    "@id" : "scei:some-asset",
    "@type" : "sce:Asset",
    "dc:subject" : {
        "@id" : "scei:123",
        "rdf:value" : "car"
    },
    "dc:subject" : {
        "@id": "scei:567",
        "rdf:value" : {"@id": "wni:wordsense-car-noun-1"}
    }
}
```

In addition to this, the SAUCE vocabulary contains simplified terms for materialising synonyms and hyponyms, in a more flattened way, to improve the performance and efficiency of search. The keyword action described in section 5 describes how keywords are enriched with synonyms and hyponyms from the Wordnet dataset.

Finally, additional dictionaries and synsets can be added to the core synsets. This provides a means to add industry specific terms, which may not exist in Wordnet. This can be achieved by adding new synsets to the search distribution, using the Wordnet terms.

Once added, these will automatically be available during enrichment and search.

5.3.7 Depictions and Actions

Another important facet of an asset is the thing (real or conceptual) that it is depicting. Whilst keywords and tags can be used for a majority of use cases, in some circumstances, more detailed relationships need to be expressed. In some scenes, there are multiple depictions of things with different properties and different emphasis and being able to succinctly describe these could be important for reusability. In addition, for assets that have a temporal aspect and involve many depictions over time with actions, objects and results, then more advanced modelling can be applied. Each depiction can contain the following:

- Instance information relating to a real world thing (e.g a crowd containing a number of people). Each person may also be a depiction as well. Depictions can contain other depictions, forming a depiction graph.

- Reference information relating to things in YAGO, WikiData or the Google Knowledge Graph. Properties from these data sources can be used to decorate the depiction.
- Actions in the asset. Actions involve agents, objects, participants and results (along with other data for different types). Results are either other actions or depictions thus completing a depiction/action scenegraph

```
{
"@id": "scei:asset-1234",
"@type": "sce:Asset",
"dc:about": [{
            "@id": "scei:depiction-uuid-1",
            "@type" : ["google:Bridge"],
            "dct:title":"Golden Gate Bridge" ,
            "google:geo" : {
              "@type" : "google:GeoShape",
              "google:box" : "coords",
              "google:elevation": "WGS 84",
              "google:line" : "coords",
              "google:polygon" : "coords"
            },
            "google:containedInPlace" : {
              "@type" : "wdt:Q34442",
              "@id" : "wd:Q4968916",
              "dc:title" : "San Francisco"
            }

        },
        {
            "@id": "scei:depiction-uuid-2",
            "@type" : ["google:Person","wdt:Superhero", "foaf:Person"],
            "dct:title":"Iron Man" ,
            "google:height" : "height",

            "sce:action" : [{
              "@id" :"sce:action-uuid-abcd",
              "@type" : "google:FlyAction",
              "dc:title" : "Flying towards bridge",
              "google:object" : {"@id":"sce:depiction-uuid-3"},
              "google:agent" : {"@id":"sce:depiction-uuid-2"},
              "google:participant" : {"@id":"sce:depiction-uuid-4"},
              "google:result" : {"@id":"sce:depiction-uuid-5"},
              "google:startTime" : "00:00:00",
              "google:endTime" : "00:01:45"
              ""
                    },
                    {
              "@id" :"sce:action-uuid-abcd",
              "@type" : "google:WalkAction",
              "dc:title" : "walking along bridge",
              "google:object" : {"@id":"sce:depiction-uuid-3"},
              "google:agent" : {"@id":"sce:depiction-uuid-2"},
```

```
              "google:participant" : {"@id":"sce:depiction-uuid-4"},
              "google:result" : {"@id":"sce:depiction-uuid-5"},
              "google:startTime" : "00:01:47",
              "google:endTime" : "00:01:52"

            }],
        }
   ]
}
```

In addition, as mentioned in the next section, these depictions can be enriched with taxonomic data.

5.3.8 Taxonomic Information and Data from 3rd Party Knowledge Bases

Taxonomic and encyclopedic information can be attached to depictions from knowledge bases such as YAGO, WikiData, DBPedia and the Google Knowledge Graph. Whilst any level of semantics up to OWL DL is supported, for many things a slightly weaker, less granular semantics, such as SKOS is preferable. Many existing knowledge bases contain taxonomies that can be leveraged to enrich assets, especially for browsing and searching. Below is an example of enrichment from WikiData:

```
{
       "@id": "scei:depiction-uuid-1",
       "@type" : [“wdt:Bridge", "wdt:SuspensionBridge", "wdt:SteelBridge"],
       "dct:refers" : {
              "@id" : "wd:Q946924",
              "@type" : "wdt:Bridge",
              "wdt:height" : "746 foot",
              "wdt:width" : "90 foot",
              "wdt:length" : "8314 foot",
              "dc:title" : "Golden Gate Bridge"
      }
}
```

By expanding on the WikiData nodes above, we can discover all the facts about the Golden Gate Bridge (its height and length) along with it type and sub-types.

5.3.9 T-Box Data, Custom Properties and Rules

T-Box Data used as custom properties and during ingestion can be included as a separate graph in the ingest document, along with any number of assets. These terms are stored in a dynamic graph in the graphstore with the following URI:

https://vocabularies.sauce.dneg.org/custom/

T-Box terms included during ingest get merged with existing T-Box terms in the dynamic property graph and are used along with core terms during reasoning and query.

```
{
   "@id" : "scec:custom",
   "@type" : "owl:DatatypeProperty",
   "rdfs:range" : {
     "@id" : "xsd:integer",
```

```
    },
   "rdfs:label" : "custom",
   "rdfs:comment" : "A custom property"
}
```

6 Technical Architecture

This section provides an overview of the foundation building blocks of the framework, provides patterns for 3rd parties who want to contribute classifiers, enrichers and advanced search filters, and provides some guidelines for common integration types.

6.1 Building Blocks and Archetypes

The framework adopts a 'Libraries, Actions, Solutions' software architecture, which is recommended as a pattern for 3rd party integrations as well. Below, we describe each of these archetypes in the most abstract way possible and then provide an overview of how they fit together, illustrating this by decomposing the lifecycle of a core component of the framework, which is consistent with the software archetype.

6.1.1 Libraries

Libraries are reusable units of software that provide some core piece of business logic (in this case, enrichment or classification), which can be built, tested and published in isolation. Libraries should be independent of an application or system runtime, and should be distributable in accordance with the package management tools compatible with their programming language (e.g PIP, Gradle, Maven etc). It is important that libraries conform to the Semantic Versioning Standard. The framework encourages the use of libraries and the core components listed below all rely heavily on libraries as part of their composition. The image metadata enricher library is an example of a java library below.

The project contains a project build file, which in turn defines its own version along with dependent software. These dependencies are obviously important because when it comes to building an action, all runtime dependencies are required. Finailly, there are some common tasks which all libraries should include, which lexically or syntactically may differ, but semantically should be equivalent, namely:

- build - builds a binary
- test - runs tests
- publish - publishes to a artifact repository
- tasks - lists all the tasks for the project

6.1.2 Actions

Actions are the default runtime for the framework and are based entirely on the underlying Openwhisk architecture and runtime. Actions operate in a completely uniform way, by taking in a JSON-LD document as input and producing a JSON-LD document as output. This can be a completely different JSON-LD document, although the ingestion system works on the basis that the output is a decorated/enriched version of the input.

Actions can be built, tested and deployed entirely in isolation, or as part of a sequence of actions in a solution (see below). The framework supports actions written in the following software runtimes:

- Java
- Python

- Ruby
- PHP
- Javascript
- Bash
- Docker

In all instances, actions should packaged and published with all the runtime dependencies that they require in order to execute. Again, actions should be consistent with the same lifecycle as a library and the same tasks, namely:

- build - builds an action with all required runtime dependencies
- test - runs tests
- publish - publishes to a artifact repository
- tasks - lists all the tasks for the project

6.1.3 Solutions

A solution is a project which orchestrates / coordinates and combines actions, applications and services to provide a complete end-to-end experience or product, or significant part of an experience or product. In the case of the framework, there is one solution, which contains a number of components:

The platform directory contains all the scripts and configuration to create the infrastructure for the framework. This includes kubernetes initialisation scripts and configuration, helm packages and docker compose files for OpenWhisk and Fuseki (the SPARQL Graph Store). This enables the framework to be deployed and run to any Kubernetes cluster, or run locally using Minikube or Docker Compose, maintaining complete parity and predictability across local or production environments.

The solution directory contains openwhisk configuration and routing for all the actions, triggers and events. This package, published and deployed as a Serverless framework file, describes how actions are chained together, the events that trigger certain actions and any HTTP endpoints associated with those events.

The solution can be packaged, versioned and published as per any other archetype, and deployed independently in isolation to any openwhisk cluster.

6.2 Core Framework Components

In order to demonstrate the capabilities of the framework, a number of core actions have been developed and deployed. These actions also provide general purpose utility and value to additional plugins and components developed by the consortia, and can be chained or sequenced accordingly

6.2.1 The Keyword Enricher

This action iterates through keywords in the JSON-LD ingest document and enrichs them with synonyms and references to the the Wordnet dictionary.

This behaviour is largely dependent on the specificity of the keyword provided in the subject, as highlight in section 5.3.5. If a string literal is provided, then all the synonyms for that literal are provided, irrespective of the word sense or context. If a wordsense URI is provided, then only the synonyms for that word sense are provided. Below is an example of the resulting enrichment after running the plugin against an ingest document with keywords described as per 5.3.5:

```
{
```

```
        "@id" : "scei:some-asset",
        "@type" : "sce:Asset",
        "dc:subject" : {
            "@id" : "scei:123",
             "rdf:value" : "car",
             "sce:keyword" : {
                 "@id" : "scei:uuid-1234",
                 "@type" : "sce:keyword",
                 "sce:keywordValue" : "car",
                 "sce:synonym" : {
                     "@id" : "scei:uuid-5678",
                     "@type" : "sce:Keyword",
                     "sce:keywordValue" : "automobile"
                     }
                }
        }
}
```

(for the PoC, and in this document, synonyms are injected into the JSON-LD document for clarity and simplicity).

A number of string literal synonyms are materialised against individual keyword instances which can be used in a full text search

6.2.2 Image MetaData Enricher

The image MetaData enricher extracts image metadata from a wide variety of image formats to decorate the ingest document. These include the following industry standard metadata models:

- Exif
- IPTC
- XMP
- JFIF / JFXX
- ICC Profiles
- Photoshop fields
- WebP properties
- WAV properties
- AVI properties
- PNG properties
- BMP properties
- GIF properties
- ICO properties
- PCX properties
- QuickTime properties
- MP4 properties

For the following file formats:

- JPEG
- TIFF
- WebP
- WAV
- AVI
- PSD
- PNG

- BMP
- GIF
- ICO
- PCX
- QuickTime
- MP4
- Camera Raw
    - NEF (Nikon)
    - CR2 (Canon)
    - ORF (Olympus)
    - ARW (Sony)
    - RW2 (Panasonic)
    - RWL (Leica)
    - SRW (Samsung)

The enricher does a number of things. Firstly it appends extracted metadata to corresponding file representations (see section 5.3.5), so if there are assets with file representations in different image formats, for each file representations it will append properties relevant to that format. Secondly it will also add these terms to the dynamic ontology, along with corresponding data types, with the corresponding namespace / prefix and descriptions, so that they can be reused. Below is an example of enriched file representations:

```
{
"@id": "scei:asset-12345-67890",
"@type": "sce:Asset",
"dcat:distribution": [{
         "@id" : "sce:dist-uuid-1234",
         "@type" : "dcat:Distribution",
         "dct:title" : "JPEG Compressed Collection of Images",
         "dct:description": "The asset as a collection of JPEG images",
         "sce:sample" : {
             "@id" : "http://sample-url-for-introspection",
             "dcat:mediaType": "image/jpeg",
             "dct:format" : "image/jpeg"
             },
         "dct:format" : "application/gzip",
         "dcat:mediaType": "image/jpeg",
         "dcat:byteSize" : 12345678,
         "jpeg:Compression-Type" : "Baseline",
         "jpeg:Data-Precision" : 8,
         "jpeg:Image-Height" :  1536,
         "jpeg:Image-Width" : 2048,
         "jpeg:Number-of-Components" : 2,
         "jpeg:Component-1" : "horiz/2 vert",
         "jpeg:Component-2" : "horiz/1 vert"
       }
}
```

6.2.3 The Photo Label Enricher

The photo label enricher iterates through all the available labels from photos and adds them as subject keywords to the asset. It also gets URIs relating to those labels from the Google Knowledge Graph. Obviously it is recommend that immediately after running a label enricher, that the keyword enricher is also run.

6.2.4 The General Purpose OWL Reasoner

The general purpose OWL reasoner simply combines the core, custom and dynamic ontologies, along with any T-Box data in the ingest document and reasons over it using the standard Jena OWL DL reasoner. Additional T-Box and A-Box data is appended to the ingest document, which is also persisted on load as detailed below.

6.2.5 The Search Data Loader

This action is responsible for updating the search engine / datastore with information after ingestion. The actual operation that this action performs is dependent on the asset's URIs (if it is an existing asset or a new one). If an id exists, and the incoming method was a PATCH or a POST, then the data is merged. If the operation was a PUT to the ordinate URI, then the entire resource is replaced. If no URI is present for the asset, then a new resource is created and a new URI issued. The data load creates a named graph in the RDF store for each asset. T-Box data, which has not been created as part of the OWL reasoner, is stored in the dynamic vocabulary graph, whilst materialised T-Box and A-Box data is part of the asset graph.

6.3 How to Plugin a Classifier, Metadata Extractor or Enricher

Plugins can be integrated into the framework as OpenWhisk actions, following the action archetype detailed in 6.1.2 and using any library dependencies available to the project. As stated, actions can be developed using a variety of runtimes, as stated in 6.1.2 and should be packaged, versioned and published with all their runtime dependencies. Once published, a new action can be included into the framework, by adding it to the solution, detailed in 6.1.3 and 6.1.4. Below is an example based on the image metadata enricher, built as a java action:

Firstly we will create a java library which can take a ingest document and stream of images, extracts all the metadata from the images, and appends it to the relevant file nodes in the ingest document. We do this by creating a library for an archetype, adding it to source control, writing the code that does the extraction and enrichment and publishing to an artifact repository:

```
mkdir sauce-lib-imagemetadata && cd sauce-lib-imagemetadata
gradle init --type=groovy-library
gradle wrapper
mkdir -p src/main/groovy/dneg/sauce/imagemetadata
mkdir -p src/test/groovy/dneg/sauce/imagemetadata
```

we can now write the code, in the above packages, along with accompanying tests and ensuring the project is correctly semantically versioned

```
rootProject.name = 'sauce-lib-imagemetadata'
group = 'dneg.sauce'
version = '1.0.0'
```

and then publish the library

```
./gradlew build test publish
```

this creates a binary artifact that can be bundled into an action.

Creating an action is similar to the above:

```
mkdir sauce-action-imagemetadata && cd sauce-action-imagemetadata
gradle init --type=java-library
gradle wrapper
mkdir -p src/main/groovy/dneg/sauce/actions/imagemetadata
mkdir -p src/test/groovy/dneg/sauce/actions/imagemetadata
```

We can now include the above library in our dependencies to be called in our action, along with the necessary OpenWhisk libraries and the core vocabularies:

```
dependencies {

 compile 'dneg.sauce:sauce-lib-imagemetadata:1.0.0'
 compile 'dneg.sauce:sauce-lib-vocabulary:1.0.0'
 compile 'com.google.code.gson:gson:2.3.1'

 //helper library for working with JSON-LD
 compile 'dneg.sauce:sauce-lib-linkeddata:1.0.0'

}
```

To use this library in our action, we need to implement a method with the correct OpenWhisk Signature:

```
import com.google.gson.*;
import dneg.sauce.linkeddata.LdModel;
import dneg.sauce.vocabularies.SauceVocabulary;
import dneg.sauce.imagemetadata.ImageMetadataEnricher;

public class ImageMetaDataAction {

    public static JsonObject main(JsonObject args) {

       // create an asset from the input json object
        LdModel asset = new LdModel(SauceVocabulary.prefixes());
        asset.add(args.getAsString(), "JSON-LD");

       //enrich using the library
       ImageMetadataEnricher imageMetadataEnricher = new
ImageMetadataEnricher(asset);


        JsonObject response = new JsonParser().parse(
imageMetadataEnricher.enrich().json() ).getAsJsonObject();

        return response;


    }
```

```
}
```

within this method we can create an instance of a JSON-LD ingest document from the raw input, create an instance of the enricher, enrich the document, and then provide it as a response for a subsequent action. The method signature is static, which means tests can be written without access to an OpenWhisk runtime. To ensure we can build a binary with all dependencies (a fat jar) we include the following task in our build script:

```
jar {
	from {
		configurations.compile.collect { it.isDirectory() ? it :
zipTree(it) }
	}
}
```

We can now build and publish our action rather like a library:

```
./gradlew build test publish
```

Finally we can include the action in our solution, by firstly adding the dependency to the solution dependency graph, and then adding it to the OpenWhisk deployment package, defining where it executes in relation to other actions:

```
service: ingest
  provider:
	name: openwhisk
  functions:
	imagehttprequesthandler:
	 runtime: java
	 handler:
binaries/sauce-action-imagehttprequesthandler-1.0.jar:HandleRequest
	 memory: 128
	imagelabelenricher:
	 runtime: java
	 handler: binaries/sauce-action-imagelabelenricher-1.0.jar:Enrich
	 memory: 256
	imagemetadataenricher:
	 runtime: java
	 handler: binaries/sauce-action-imagemetadataenricher-1.0.jar:Enrich
	 memory: 256
	keywordenricher:
	 runtime: java
	 handler: binaries/sauce-action-keywordenricher-1.0.jar:Enrich
	 memory: 256
	owlenricher:
	 runtime: java
	 handler: binaries/sauce-action-owlenricher-1.0.jar:Enrich
	 memory: 512
```

```
    loader:
     runtime: java
     handler: binaries/sauce-action-loader-1.0.jar:Load
     memory: 128
    ingestimages:
     sequence:
     - imagehttprequesthandler
     - imagelabelenricher
     - imagemetadataenricher
     - keywordenricher
     - owlenricher
     - loader
     events:
     - http: POST /assets/images
  plugins:
     - serverless-openwhisk
```

In the example above, we have added it to the chained action sequence for Image ingestion.

For a different type of asset, we can create a different URL and action sequence, which is suitable for that type of asset. Certain actions, such as keyword enrichment and loading, are applicable to all.

We can publish the solution without deploying it by running the following:

```
serverless package --package sauce-solution-ingestion-1.0.0.zip
```

This provides a convenient way to package complete solutions, with all dependencies, for deployment to different environments.

To deploy to a OpenWhisk environment we can run the following:

```
serverless deploy --package sauce-solution-ingestion-1.0.0.zip
```

6.4 How to Plugin a Google ML Model or Other Training Model

In this section, we will describe how to plugin a machine learning model for classification or prediction. The framework supports ML models deployed in 3 different ways:

- As part of a an action, deployed as an action into the OpenWhisk runtime
- As a separate HTTP service, deployed into the search cluster as a standalone microservice
- As a completely remote HTTP service, consumed as software as a service

The easiest way to interact with a Google machine learning model is through the gcloud command line interface. This can be executed as part of an action as a docker:

```
functions:
  animationlabelenricher:
    handler: tcd/animation
    runtime: docker
```

The docker image, when run, will execute whatever exists in the /actions/exec directory contained within the Docker image taking in ***stdin*** and responding with ***stdout***

The service can be deployed into the search cluster by packaging the the service as a Helm chart and deploying into the search kubernetes namespace configured in the platform directory specified in sauce-solution-core, also mentioned in section 6.1.3

Finally, a complete remote HTTP service can be consumed as an action in exactly the same way as any action defined in 6.1.2. The imagelabelextractor action does this itself by remotely calling the Google Vision API in order to extract labels from images.

6.5 How to Search Using the SPARQL API

Data ingested into the search engine can be accessed, interrogated and queried using a fully compliant SPARQL 1.1 interface. The framework exposes the following endpoints for various SPARQL operations:

```
https://[hostname]/sauce/query
```

in accordance with SPARQL 1.1 query protocol.

Below is a simple query, which returns all meshes (and sub-classes of asset types thereof). Please note that for the sake of brevity, prefix declarations required for CURI's are omitted, but all those in section 5.1 will suffice.

```
SELECT ?s ?title ?description WHERE {
    ?s rdf:type/rdfs:subClassOf* sce:Mesh.
    ?s dc:title ?title.
    ?s dc:description ?description.
}
LIMIT 10
ORDER BY ?title
```

Obviously, this is a relatively simplistic search. What if we want to search by using all the concepts defined in the data architecture section in a single query e.g:

1. find 10 Reference Photographs,
2. depicting automobiles and all subclasses of things that are automobiles,
3. available in a JPEG file format with compression of type of Baseline,
4. with keyword "sunset",
5. or things that "sunset" is a synonym for,
6. sorted alphabetically

```
SELECT ?s ?title ?description WHERE {
    ?s rdf:type/rdfs:subClassOf* sce:ReferencePhotography.
    ?s sce:depicts ?depiction.
    ?depiction rdf:type/rdfs:subClassOf* yago:Automobile.
    ?s dc:title ?title.
    ?s dcat:distribution ?distribution.
    ?distribution dcat:mediaType "image/jpeg".
    ?distribution jpeg:Compession-Type "Baseline".
```

```
    ?s dc:subject ?subject.
    ?subject sce:keyword ?keyword.
    ?keyword sce:keywordValue|sce:synonym/sce:keywordValue "sunset".
}
LIMIT 10
ORDER BY ?title
```

We can now start to navigate across the graph. If we want to broaden our search to things one level broader to Automobiles using SKOS terms:

```
SELECT ?s ?title ?description WHERE {
    ?s rdf:type/rdfs:subClassOf* sce:ReferencePhotography.
    ?s sce:depicts ?depiction.
    ?depiction rdf:type/skos:broader/rdfs:subClassOf* yago:Automobile.
    ?s dc:title ?title.
    ?s dcat:distribution ?distribution.
    ?distribution dcat:mediaType "image/jpeg".
    ?distribution jpeg:Compession-Type "Baseline".
    ?s dc:subject ?subject.
    ?subject sce:keyword ?keyword.
    ?keyword sce:keywordValue|sce:synonym/sce:keywordValue "sunset".
 }
LIMIT 10
ORDER BY ?title
```

6.5.1 Full Text Searching

The search API also supports full text search across string literals using either a complete set of Lucene search terms or Elastic search terms. In the example above, we demonstrate additional full text searching to keywords using wildcards:

```
SELECT ?s ?title ?description WHERE {
       ?s rdf:type/rdfs:subClassOf* sce:ReferencePhotography.
       ?s sce:depicts ?depiction.
       ?depiction rdf:type/skos:broader/rdfs:subClassOf* yago:Automobile.
       ?s dc:title ?title.
       ?s dcat:distribution ?distribution.
       ?distribution dcat:mediaType "image/jpeg".
       ?distribution jpeg:Compession-Type "Baseline".
       ?s dc:subject ?subject.
       ?subject sce:keyword ?keyword.
        ?keyword text:query (sce:keywordValue|sce:synonym/sce:keywordValue
"sun*set").
}
LIMIT 10
```

The complete list of modifiers, operators and groupings are listed below:

- Term Modifiers
    - Wildcard Searches
    - Regular expression Searches
    - Fuzzy Searches
    - Proximity Searches
    - Range Searches
    - Boosting a Term
- Boolean Operators
    - OR
    - AND
    - +
    - NOT
    - -
- Grouping
    - Field Grouping
    - Escaping Special Characters

6.6 How to Create Advanced Search for Similarity, Compatibility, Comparison or Other Custom Predicates, Unifying Different Orders of Logic

For more complex use cases such a similarity, comparison or compatibility, the RDF graph and corresponding SPARQL query, either won't contain the terms and data representations, or the complexity of the comparison and algorithms necessary to run them require a more sophisticated sub-system. Fortunately some of the advanced features of SPARQL accommodate for this with the use of extension and custom predicates. To facilitate the development of extension predicates, the framework includes a library with a set of interfaces that a contributor can implement to enable integration into a search.

To incorporate a custom predicate, an appropriate prefix for the predicate should be appended to the vocabulary prefixes listed in section 5, which correctly refers to the implementation of the custom predicate in the prefix URI.

In the example or a similarity or comparison search, a URI of an asset that should be used as reference for comparison (either pre-existing or ingested at search time) is use as the object of the predicate in the custom predicate, Below is an example of a comparison by a lightfield comparison of two assets

```
SELECT ?s ?title ?description WHERE {
       ?s rdf:type/rdfs:subClassOf* sce:ReferencePhotography.
       ?s sce:depicts ?depiction.
       ?depiction rdf:type/rdfs:subClassOf* yago:Automobile.
       ?s dc:title ?title.
       ?s dcat:distribution ?distribution.
       ?distribution dcat:mediaType "image/jpeg".
       ?distribution jpeg:Compession-Type "Baseline".
       ?s dc:subject ?subject.
       ?subject sce:keyword ?keyword.
       ?keyword sce:keywordValue|sce:synonym/sce:keywordValue "sunset".
       ?s lighfields:similarTo scei:asset-uploaded-at-searchtime.
}
```

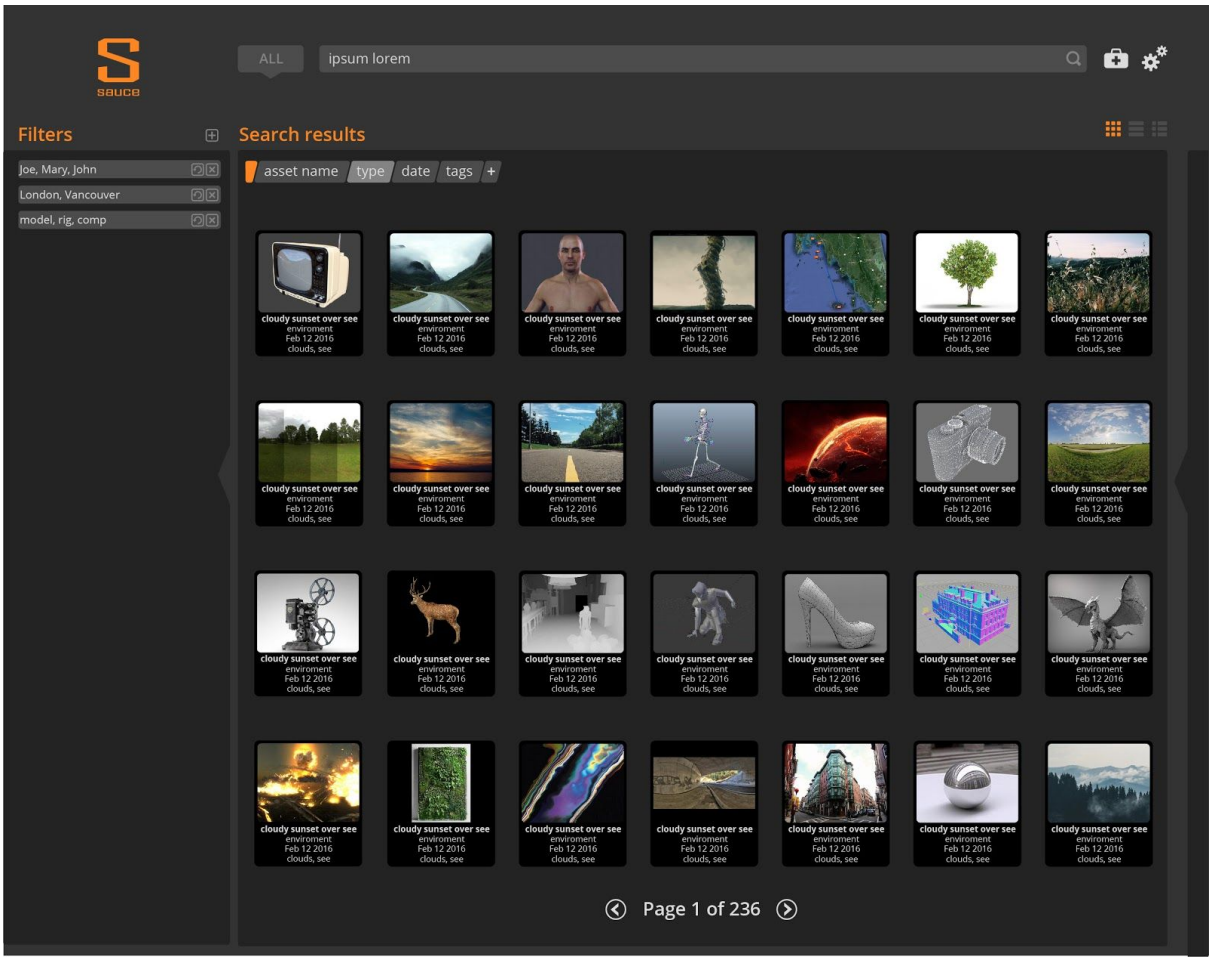
```
LIMIT 10
ORDER BY ?title
```

Of course the actual logic of the comparison is part of the implementation, but the outcome of the function is unified and consistent with the underlying Datalog model of a SPARQL query.

7 User Experience Architecture

To further demonstrate the framework in a more tangible and visual way, a proof of concept graphical user interface has been developed along with an underlying architecture and set of interaction patterns that can accomodate extensions to the framework. Highlights and screenshots are provided below:

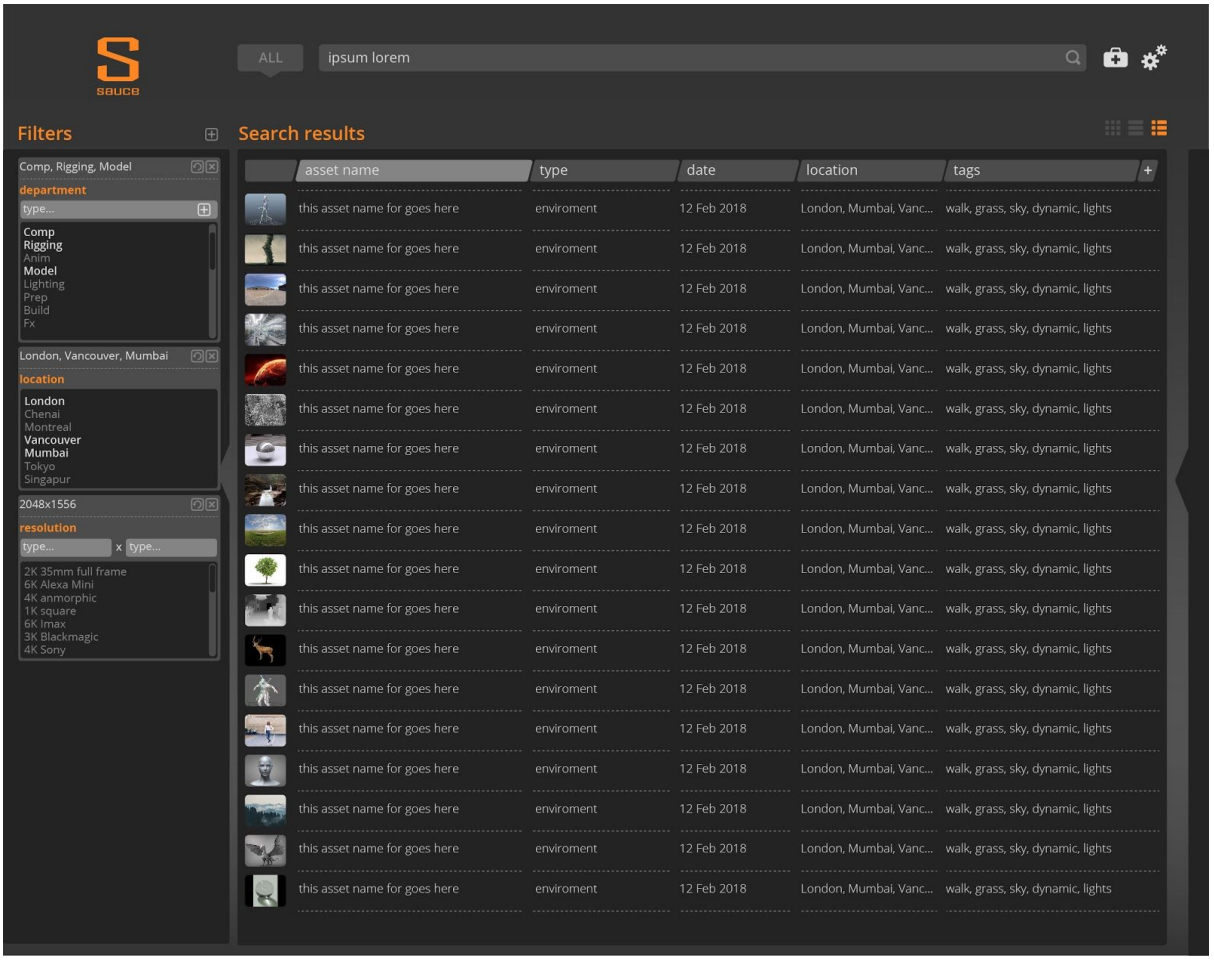
7.1 Search Results



- User is presented with the results of the search based on her/his preferences or last used view.
- Filters tab can be automatically extended to help refine search. They are presented collapsed.
- User can extend them to modify the parameters. Adding and removing the filters can be added by the small icon buttons in filter area.
- User can select one of views (details, large preview, details with small preview)

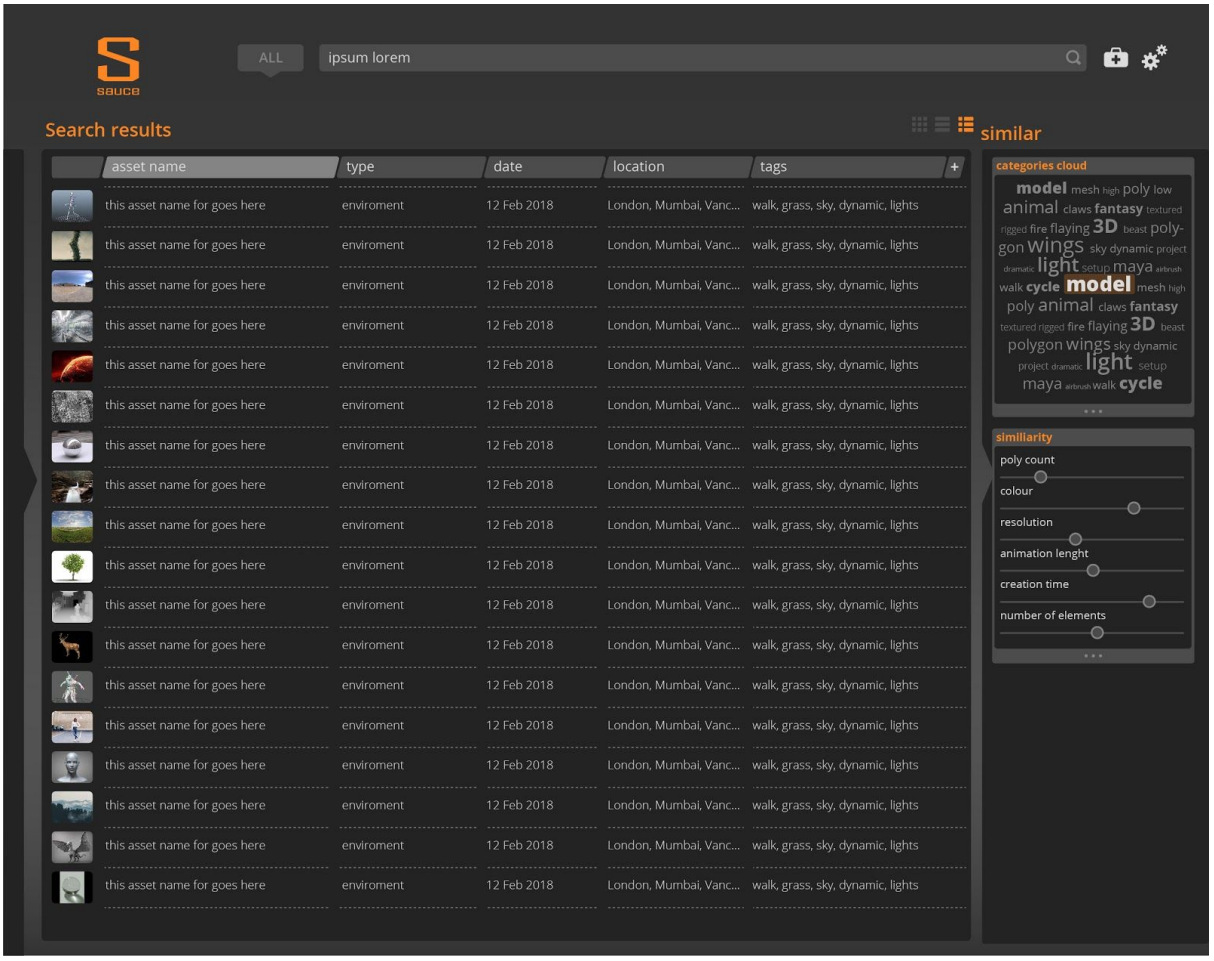
- Category selector (left of search bar) can be used to quickly limit the results to just one type of asset

7.2 Filters Expanded and Table View



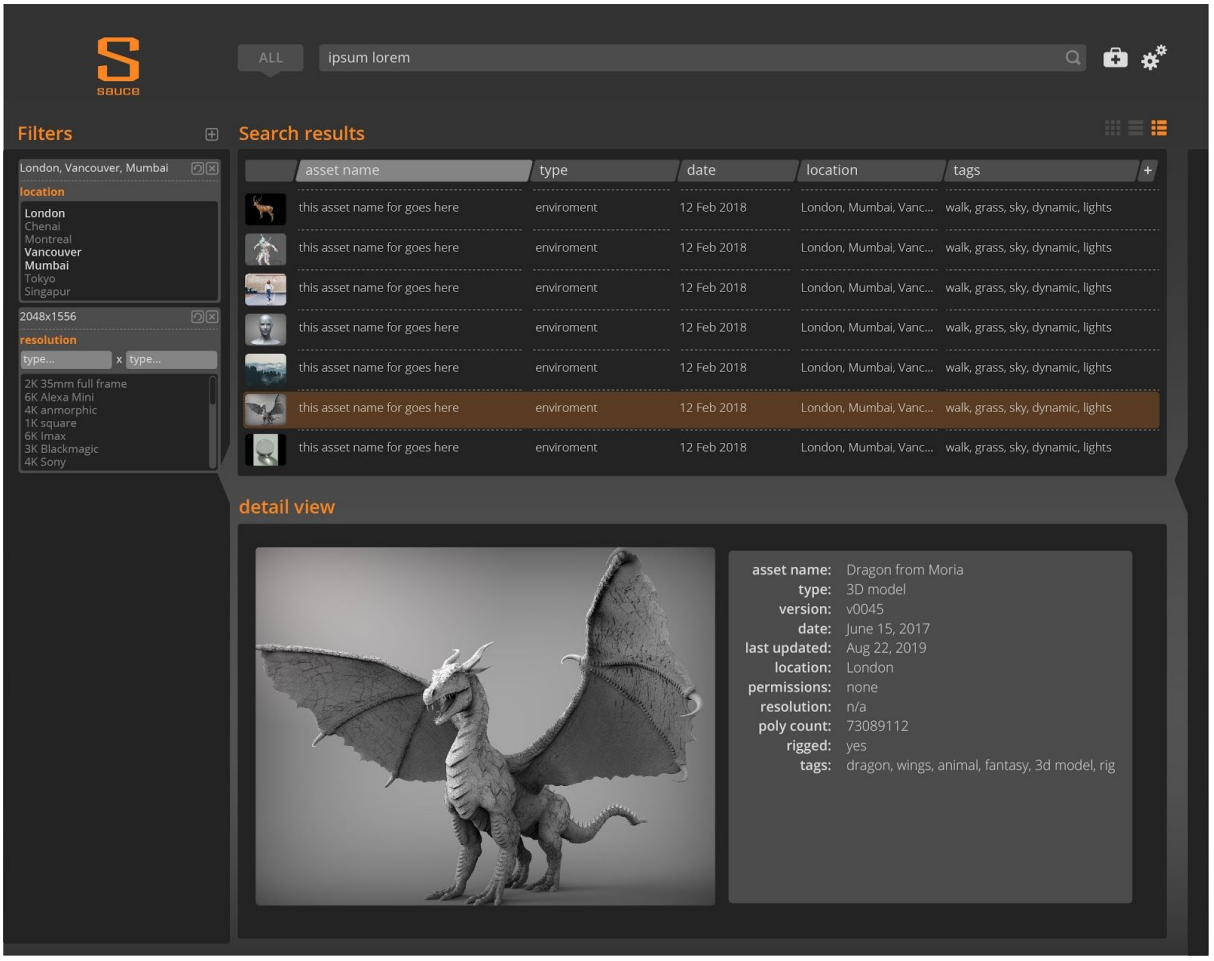
- Table view is sortable by the column
- Columns are expandable to show full set of information or just a snippet
- User can modify and rearrange the columns
- Tags (icon representation) can be used to quickly choose desired type.
- Preset filters will be available in addition to the option of specifying the custom one.

7.3 Advanced Search Features



- A categories cloud alllows users to navigate across categories, broadening and narrowing searches.
- More advanced search controls and filters for similarity, comparison and compatibility are presented below

7.4 Details View



- when selecting an asset user can display detailed info about the asset
- Large preview with playback or gallery
- Type tags are overlaid to groups that asset is part of
- Revisions and versions of the asset are also displayed

8 Conclusion

It should hopefully be clear from this report that the smart search framework is extensible and customisable for a variety of use cases, and will accommodate and provide interoperability for a range of classifiers and search descriptors which will evolve over the course of the programme and well into the future.

While we have only demonstrated with one asset type, it should be obvious that the framework can be extended across any number of asset types without constraint, since many of the principles, concepts and building blocks are consistent.

The user interface is a starting point for human interaction with the framework, but it should be noted that feedback and iteration will ensure the optimal user experience.

Additionally the framework uses the most suitable technology building blocks available to achieve its objectives, which may also be improved, extended or optimised for greater scale and coverage over the course of the project.

9 Written references

Sikos, L. F. (2017) Description Logics in Multimedia Reasoning. Cham, Switzerland: Springer. DOI: 10.1007/978-3-319-54066-5

10 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/

SKOS

https://www.w3.org/2004/02/skos/

SHACL

https://www.w3.org/TR/shacl/

Wordnet

https://wordnet.princeton.edu/

HTTP 1.1 (rfc 7230-7235)

https://tools.ietf.org/html/rfc7230

https://tools.ietf.org/html/rfc7231

https://tools.ietf.org/html/rfc7232

https://tools.ietf.org/html/rfc7233

https://tools.ietf.org/html/rfc7234

https://tools.ietf.org/html/rfc7235

SPARQL 1.1

https://www.w3.org/TR/sparql11-overview/

Lucene

https://lucene.apache.org

OpenWhisk

https://openwhisk.apache.org/

Kubernetes

https://kubernetes.io/

Blazegraph

https://www.blazegraph.com/