# Cosmos

Cultivate resilient smart Objects for Sustainable city applicatiOnS

Grant Agreement Nº 609043

# D4.2.1 Information and Data Lifecycle Management: Software prototype (Initial)

## WP4 Information and Data Lifecycle Management

**Version:**
**Due Date:** 30/6/2014
**Delivery Date:** 24/7/2014
**Nature:** P
**Dissemination Level:** PU

**Lead partner:** IBM
**Authors:** Jozef Krempasky (ATOS), Achilleas Marinakis (NTUA), Eran Rom (IBM), Paula Ta-Shma (IBM)
**Internal reviewers:** Adnan Akbar (Univ. Surrey)

www.iot-cosmos.eu

The research leading to these results has received funding from the European Community's Seventh Framework Programme under grant agreement n° 609043

**Version Control:**

| Version | Date | Author | Author's Organization | Changes |
|---|---|---|---|---|
| 0.1 | 17/07/2014 | Paula Ta-Shma and co-authors | IBM | First version for internal review. |
| 0.2 | 23/07/2014 | Paula Ta-Shma and co-authors | IBM | Second version incorporating review comments. |

**Annexes:**

| Nº | File Name | Title |
|---|---|---|
| | | |
| | | |

# Contents

# 1 Introduction

## 1.1 About this deliverable

This document is the complement to the delivered software as prototype for deliverable D4.2.1 Information and Data Lifecycle Management: Software prototype (Initial). For information on the motivation, architecture and design of the components in this work package, please refer to document D4.1.1 Information and Data Lifecycle Management: Design and open specification (Initial).

## 1.2 Document structure

In this document there is a section for each component of WP4. This includes sections on Data Mapping, CEP, Message Bus and 2 sections on Cloud Storage - Metadata Search and Storlets. In addition, there is an additional Cloud Storage section describing Security and Privacy – this describes work belonging to WP3 (End-to-end Security and Privacy) but which is part of the current deliverable (D4.2.1).

# 2 Complex Event Processing

## 2.1 Implementation

### 2.1.1. Functional description

The delivered prototype introduces ability to dynamically (at run-time) change the evaluation of Complex Event Processor rules.

The main motivation for implementation of this prototype is to increase applicability and flexibility of Complex Event Processor for event detection and monitoring features provided by COSMOS. These features are further described in WP6.

#### *2.1.1.1. Fitting into overall COSMOS solution*

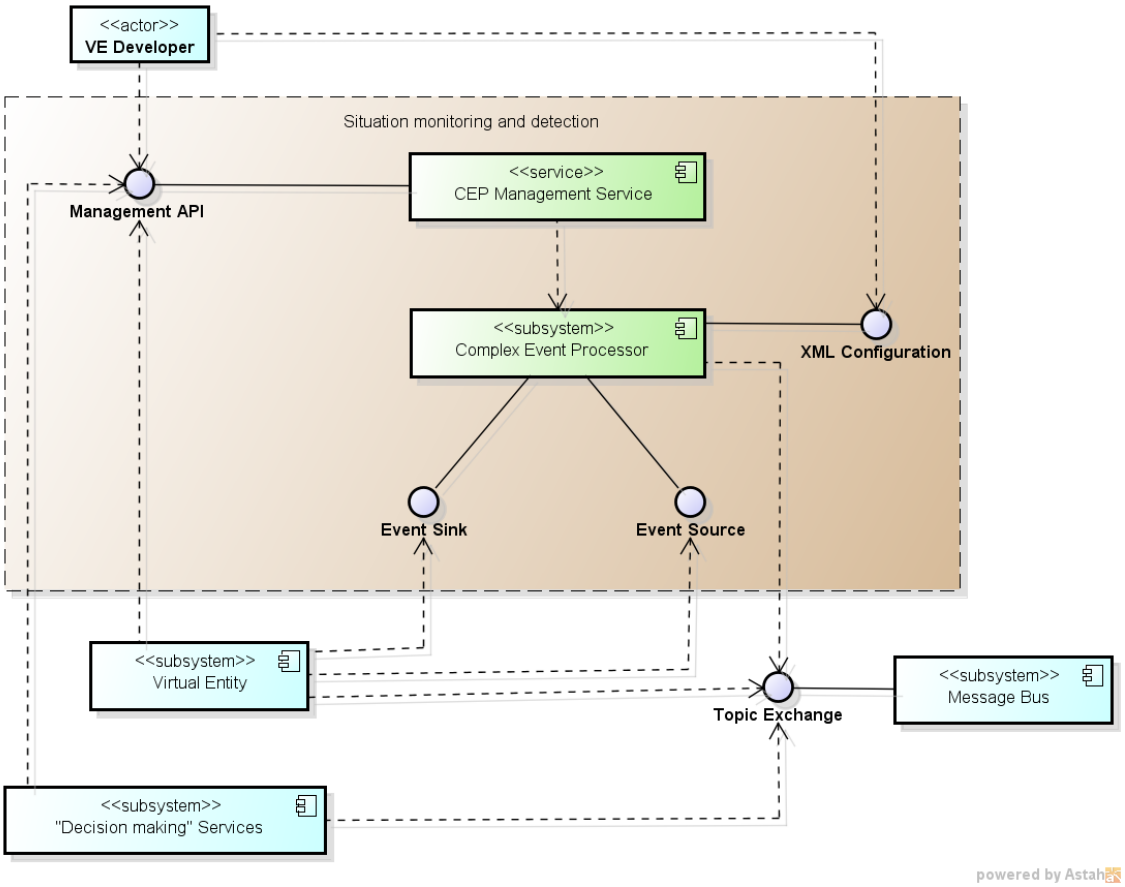



Figure 1: Situation monitoring and detection subsystem

As depicted in figure 1, prototyped solution for situation detection and monitoring functionality collects information mainly from virtual entities either directly or through the message bus and detected situations are consumed by applications or decision making components within COSMOS such as decentralized management described by WP5.

More detailed technical information can be seen in D4.1.1 document.

## 2.1.2. Technical description

The ability to dynamically change the evaluation of Complex Event Processor rules is provided by CEP Management service via REST API.

### *2.1.2.1. Prototype architecture*

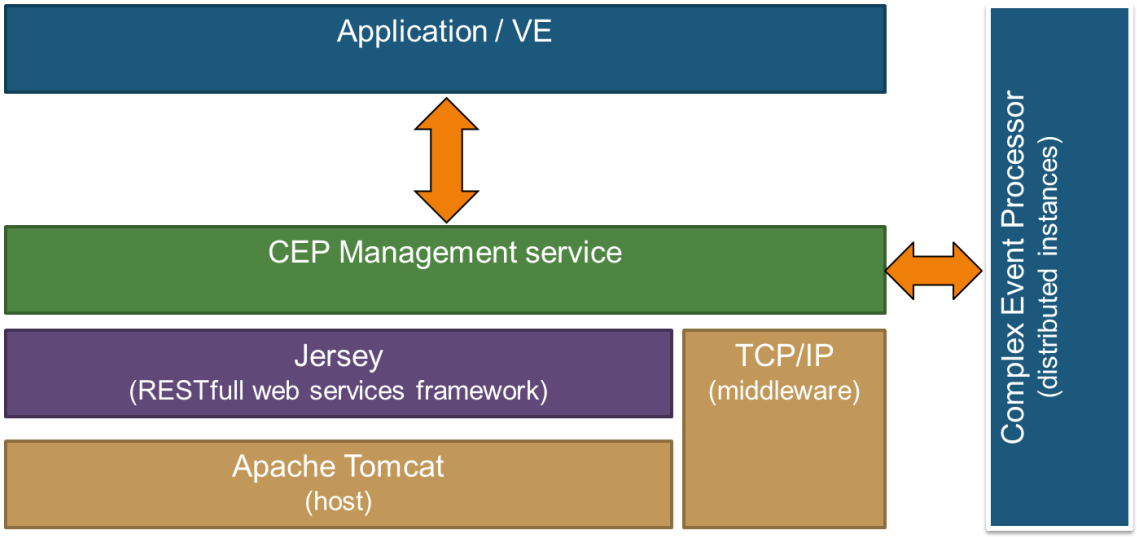



Figure 2: CEP Management service architecture

The figure 2 describes more detailed architecture of the CEP Management service. For high level architecture, please see D4.1.1 document.

### *2.1.2.2. Components description*

For communication with external clients, a CEP management service utilizes Jersey [2] framework which offers support for seamless exposing of data in variety of representation media types without a need to implement low level communication details.

For hosting purposes, we decided to use Apache Tomcat [3] web server which is directly supported by the Jersey framework and 64bit Linux operating system.

In order to support distributed CEP deployment, administration and modification of all running CEP instances is controlled through single CEP management service.

### *2.1.2.3. Technical specifications*

The prototype will be deployed on and executed by single 64bit Linux operating system. Virtual entities and applications will be deployed on their own execution environment and communicate with COSMOS through the message bus using standard network connection.

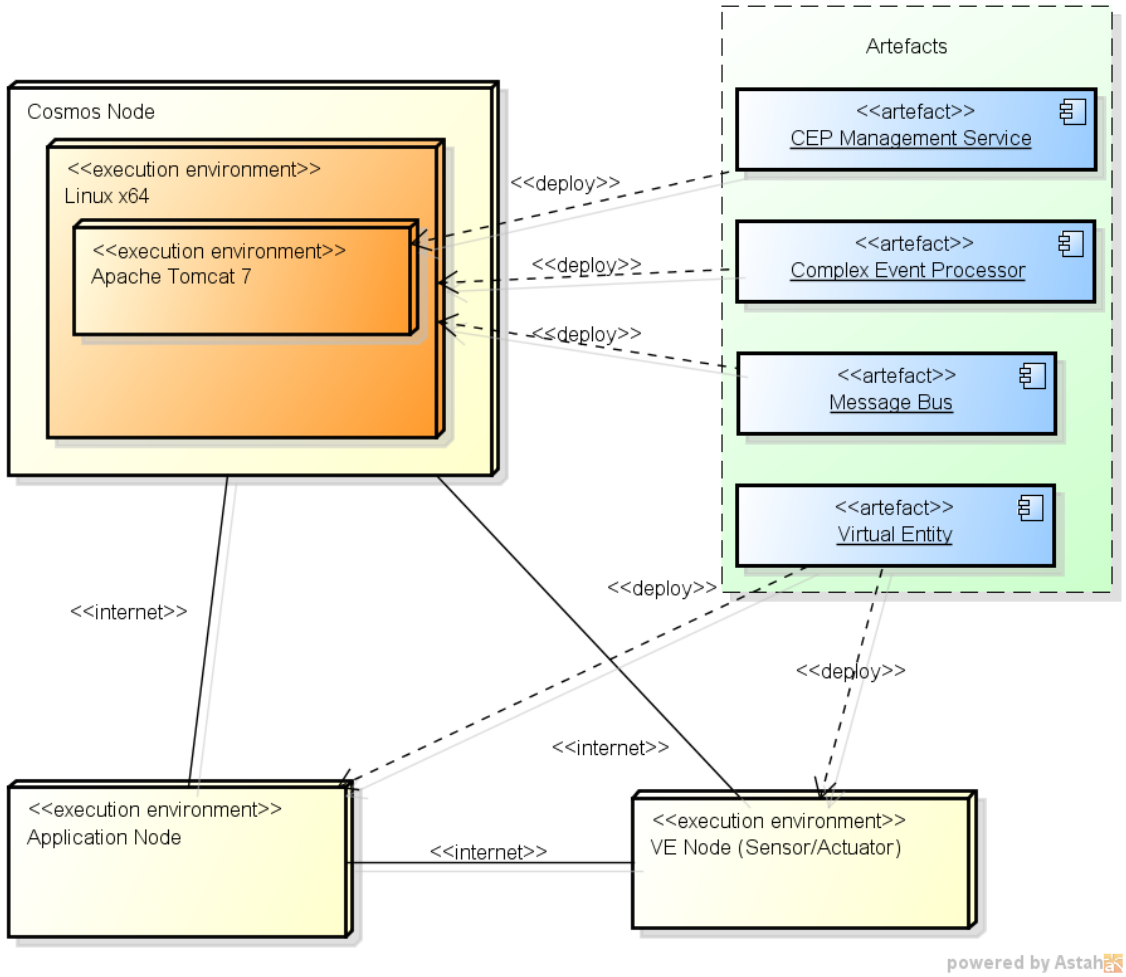



Figure 3: Deployment model

The proposed deployment for prototype is described on figure 3. The Complex Event Processor and the Message bus are deployed on Linux as system services.

### 2.1.2.4. CEP Management Service

A CEP Management Service provides RESTful web service based on HTTP[5] methods and the concept of REST. Service is accessible through central URI and supported MIME type is JSON[6]. Service is implemented in Java by utilizing Jersey[2] reference implementation for the JSR 311[7] (Java Specification Request) specification. The service itself is executed in the Java servlet container and hosted on the Apache Tomcat [3] web server.

### 2.1.2.5. Complex Event Processor

In order to achieve high and stable event evaluation rate as well as low end-to-end latency, the Complex Event Processor is implemented in C++. Distributed CEP components utilize enhanced middleware based on Zero MQ [4] for fast and reliable data transfer.

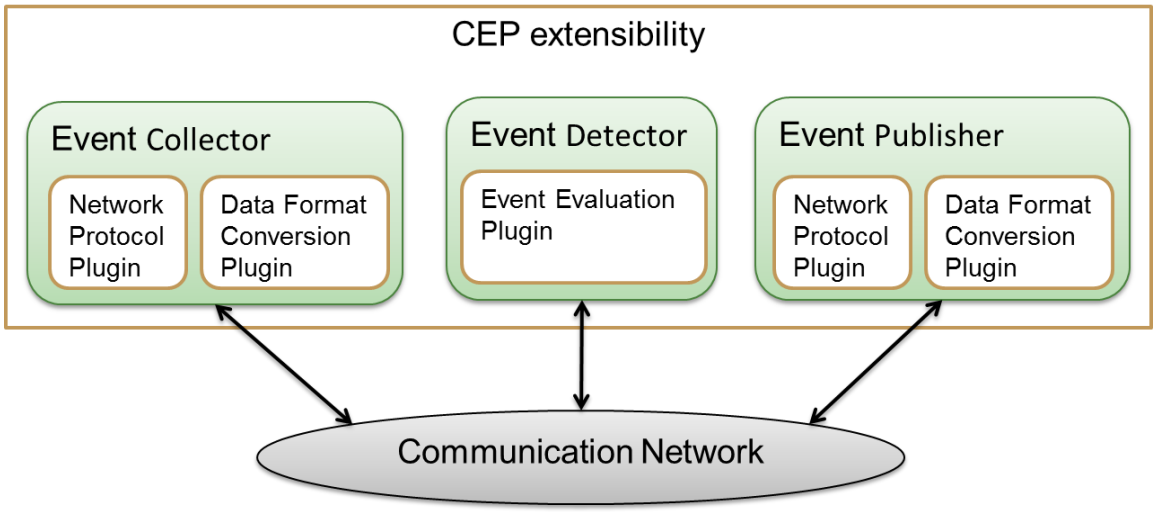



Figure 4: Extensibility via plugins

The figure 4 illustrates extension possibilities of the CEP which is primarily achieved through SPI – plugin mechanism. A new Json data format plugins have been introduced for demo purposes.

## 2.2 Delivery and usage

### 2.2.1. Package information

The delivery contains following files:

The delivery of CEP contains following files:

- General files
  - Manifest.txt
  - License.txt
- Configuration files
  - Config/solcep.conf.xml -- Default Configuration file
  - Config/solcep.detector.xml -- Standalone detector configuration
  - Config/solcep.collector.xml -- Remote event collector configuration
  - Config/solcep.publisher.xml -- Remote event publisher configuration
  - Config/detect.dolce -- Dolce detection specification
- System test files
- Executables
  - solcep_ctrl -- Server control for Debian
  - solcep -- Standalone SOL/CEP binary
  - Plugins – Network protocol and data format plugins

The delivery of the CEP Management service contains following files:

- Libraries – libraries mentioned above.
- Source files – Java sources of this component.
- Configuration files.

### 2.2.2. Installation instructions

This installation manual assumes that following prerequisites are already installed and running:

- Java SE/EE Runtime Environment
- Apache Tomcat

It is recommended to create a new user and home directory before actual installation of the services by executing: *sudo adduser [user name]* and log-in as the new user.

Steps:

1. Unpack the content of provided package: tar –xvzf /custom location/Cosmos_CEP_Services.tar.gz
2. Ensure that the execution bits of services are enabled. If not, execute: chmod +x Solcep.
3. Copy control script solcep_ctrl into /etc/init.d directory.
4. Register service with the operating system infrastructure: sudo update-rc.d add solcep_ctrl defaults
5. Review and update provided configuration files when interoperation with distributed CEP components is required.

### 2.2.3. User Manual

An event detection mechanism within SOL/CEP is variation of rule-based inference engine. Rules are defined using specialized **Dolce** language focusing on IoT domain. For detailed information about how to define custom events and event detection rules, please refer to dolce language specification mentioned in D4.1.1 .

The dynamic changing of rules will be available via REST client.

### 2.2.4. Licensing information

Currently, the SOL/CEP is distributed as closed source software.

CEP Management service is distributed under Apache 2.0 license.

### 2.2.5. Download

The source code is available on the COSMOS SVN, under SourceCode\M10 Prototypes\WP4\CEP

# 3 Data Mapping

## 3.1 Implementation

### 3.1.1. Functional description

Data mapping will be used in COSMOS in order to collect raw data that is published from virtual entities through the message bus and store it as data objects, with their associated metadata, in the cloud storage. Additional information on motivation can be found in section 4.1.1 Functional Overview of Deliverable 4.1.1 Information and Data Lifecycle Management: Design and open specification (Initial).

#### *3.1.1.1. Fitting into overall COSMOS solution*

In COSMOS we would like to be able to store objects with enriching metadata, in order to enable search on them as described in chapter 5.

This metadata could be:

- Timestamps
- Information like the number of a bus line, the number of a bus, etc.

A diagram of how Data Mapping fits into the WP4 architecture can be found in section 3 High Level Architecture of the D4.1.1 In addition, D2.3.1 discusses the COSMOS overall architecture.

### 3.1.2. Technical description

This section describes the technical details of the implemented software.

#### *3.1.2.1. Component description*

Regarding the component description, please see section 4.1.2 of the D4.1.1 Text describing the design decisions and details can be found there.

#### *3.1.2.2. Technical specifications*

The prototype uses the following open source components:

- Rabbit MQ which is used as a message broker. It allows publishers to send messages and subscribers to receive them – please see http://www.rabbitmq.com/

- OpenStack Swift which is used in order to store these messages as data objects in the cloud storage – please see http://docs.openstack.org/api/openstack-object-storage/1.0/content/storage-object-services.html

The source code is developed in Java and uses the following Java ARchive (.jar) files:

- json-simple – used for parsing json files

- rabbitmq-client – a Java client for Rabbit MQ

## 3.2 Delivery and usage

### 3.2.1. Package information

The delivered package contains the following folders:

- dependencies: contains the jar files mentioned above
- input: contains the json files to be published through Rabbit MQ server
- src: contains the JAVA files

### 3.2.2. Installation instructions

Please follow these steps to install and start up the prototype: (Java SE/EE Runtime Environment is a prerequisite)

- Install OpenStack Swift
- Install Rabbit MQ server
- Download the package and install it under the main root of your machine
- Open the package through an IDE, like NetBeans 8.0, Eclipse Kepler 4.3.0 etc.
- Run the Receiver.java continuously
- Publish the input files through the Sender.java

### 3.2.3. User Manual

For detailed information about how to configure the Rabbit MQ publisher and subscriber, please see the section 4.6.1 of the D4.1.1 Please see also http://www.rabbitmq.com/tutorials/tutorial-five-java.html

For detailed information about how to use Openstack Swift, please see http://docs.openstack.org/api/openstack-object-storage/1.0/os-objectstorage-devguide-1.0.pdf

### 3.2.4. Licensing information

**Dependencies**

1. json-simple : Apache 2.0
2. RabbitMQ : Mozilla Public Licence version 1.1
3. OpenStack Swift : Apache 2.0

### 3.2.5. Download

The source code is available on the COSMOS SVN, under SourceCode\M10 Prototypes\WP4\DataMapping

# 4 Message Bus

## 4.1 Implementation

### 4.1.1. Functional description

COSMOS platform needs to interoperate with Virtual Entities provided by different vendors. These entities may run on a variety of platforms and they need to exchange information (such as experience) between them as well as with COSMOS subsystems. The message bus provides solution for connection of independent components through message exchange mechanism.

#### *4.1.1.1. Fitting into overall COSMOS solution*

The purpose of the message bus is to integrate COSMOS components as well as external components such as Virtual entities. For the demo purposes, components will exchange messages according to static message routing configuration between publishing and listening components.

### 4.1.2. Technical description

#### *4.1.2.1. Prototype architecture*

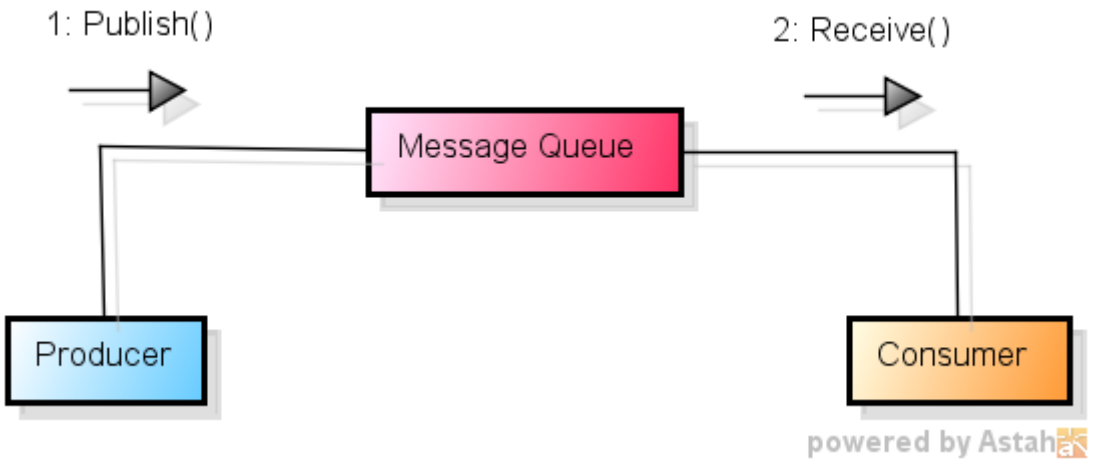



Figure 5: Message bus overview

From the high level perspective, there are two roles interacting with the message bus: a producer and a consumer. Producer sends messages and a consumer receives messages.

The more description of the high level architecture of the message bus is described in chapter 4.6 Message Bus of the D4.1.1 document.

#### *4.1.2.2. Components description*

There are no custom components introduced for the prototype. The information about data format adapters is described in section 4.6.2 of the D4.1.1 document.

#### *4.1.2.3. Technical specifications*

The RabbitMQ is implemented on top of the Erlang virtual runtime.

From the client perspective, RabbitMQ provides official support for all mainstream operating systems and programming languages.

In addition to that, RabbitMQ community has created numerous adapters and tools for specialized tasks such as integration with other existing platforms.

## 4.2 Delivery and usage

### 4.2.1. Package information

The RabbitMQ is not delivered as a standalone package but is installed using OS specific repository.

### 4.2.2. Installation instructions

1. As mentioned in the previous chapter, the RabbitMQ runs on top of the Erlang[1] virtual runtime. Therefore it is necessary to install erlang before actual installation of the RabbitMQ.
2. Install the RabbitMQ server. Packages are available on the http://www.rabbitmq.com/download.html or in OS specific repository. By default, RabbitMQ server is installed as an OS service.
3. Install the RabbitMQ Management Plugin. The purpose of this plugin is to provide Web-based management functionalities.

### 4.2.3. User Manual

The manual how to connect to the RabbitMQ and exchange messages is available online at: http://www.rabbitmq.com/documentation.html.

### 4.2.4. Licensing information

The RabbitMQ is protected by the Mozilla Public License.

### 4.2.5. Download

The source code is available on the COSMOS SVN, under SourceCode\M10 Prototypes\WP4\MessageBus

# 5 Cloud Storage – Metadata Search

## 5.1 Implementation

### 5.1.1. Functional description

Metadata search will be used in COSMOS in order to index objects according to metadata attributes and values and therefore enable search on them. Additional information on motivation can be found in section 4.3.1 Metadata Search of deliverable 4.1.1.

#### *5.1.1.1. Fitting into overall COSMOS solution*

In COSMOS we would like to be able to index objects according to various properties such as

- Timestamps
- Geospatial locations
- Textual information such as a residence street name
- Numerical values such temperature readings

This allows searching and retrieving objects according to their values for these properties.

A diagram of how metadata search fits into the WP4 architecture can be found in section 3 High Level Architecture of the D4.1.1 Information and Data Lifecycle Management: Design and open specification (Initial) document. In addition deliverable D2.3.1 discusses the COSMOS overall architecture.

Storlets (described in the next section) can both read and write metadata. Metadata which is written is indexed and therefore becomes searchable.

### 5.1.2. Technical description

#### *5.1.2.1. Prototype architecture*

Regarding the prototype architecture, please see section 4.3.2 Metadata Search Architecture of the D4.1.1 Information and Data Lifecycle Management: Design and open specification (Initial) document. Diagrams depicting the architecture can be found there.

#### *5.1.2.2. Components description*

Regarding the components description, please see section 4.3.2 Metadata Search Architecture of the D4.1.1 Information and Data Lifecycle Management: Design and open specification (Initial) document. Text describing the components in the architecture can be found there.

#### *5.1.2.3. Technical specifications*

This prototype is based on code developed by IBM SoftLayer and adapted for the needs of COSMOS. We designed and implemented a new search API which supports complex queries. For example one can search for objects meeting multiple constraints. We also implemented data type support which is needed for COSMOS data.

The prototype uses the following open source components

**Elastic Search** – a search engine built using the Lucene search library – see http://www.elasticsearch.org/

**RabbitMQ** – Rabbit MQ is used to queue the metadata indexing requests and submit them in bulk to Elastic Search – see http://www.rabbitmq.com/

**OpenStack Swift** – object storage – see http://docs.openstack.org/developer/swift/

The source code is developed in Python and uses the following open source Python libraries

- pyparsing – used for parsing the search API requests
- pyes – a python client for elastic search
- pika – a python client for Rabbit MQ

## 5.2 Delivery and usage

### 5.2.1. Package information

The **swearch_hrl** package has the following structure

- **setup.py** – python installation script
- **bin** – admin scripts
- **etc** – config files
- **swearch** – metadata index and search source code
  - **middleware** – OpenStack Swift middleware
- **tests** – unit tests

### 5.2.2. Installation instructions

1. Install Elastic Search. An installation script is provided in the swift++deployment module (described in the next section)
2. Install RabbitMQ.
3. Install OpenStack Swift
4. Install metadata search using the following command :
   - sudo python setup.py install
5. Setup the indexes using the following command
   - sudo python bin/swearch-prep

### 5.2.3. User Manual

Once metadata search has been installed, Swift objects which are created are automatically indexed according to their metadata.

Metadata search is accessed using an extension of the OpenStack Swift REST API. The metadata search API was described in Appendix 7.3 Cloud Storage and Metadata search API in the D4.1.1 Information and Data Lifecycle Management: Design and open specification (Initial) document.

### 5.2.4. Licensing information

**Dependencies**

1. Elastic Search : Apache 2.0
2. RabbitMQ : Mozilla Public Licence version 1.1
3. OpenStack Swift : Apache 2.0

**Open source Python modules**

1. pyparsing - MIT License
2. pyes – new BSD licence
3. pika – Mozilla Public Licence version 1.1 and GPL v2.0 or newer

## 5.2.5. Download

Selected source code is available on the COSMOS SVN, under SourceCode\M10 Prototypes\WP4\CloudStorage

The metadata search source code should be considered confidential i.e. accessible only by COSMOS partners and reviewers from the EU.

# 6 Cloud Storage - Storlets

## 6.1 Implementation

### 6.1.1. Functional description

Storlets are computational objects that run inside the object store system. Conceptually, they can be thought of as the object store equivalent of database store procedures. Please see D4.1.1 Information and Data Lifecycle Management: Design and open specification (Initial) document section 4.4 regarding motivation and main innovations.

#### *6.1.1.1. Fitting into overall COSMOS solution*

Data in COSMOS will be store as objects in the cloud storage. Examples of such objects are energy and temperature data for a building in Camden for a particular week, or the movements of buses in a Madrid bus line over the course of a particular day. Another example of a data object is images uploaded to the COSMOS system, for example, pictures or video taken by a bus camera or by COSMOS users' mobile phones. These objects are stored in OpenStack Swift cloud storage. We augment this cloud storage with storlets, which enable computation to take place close to the data objects. For example, storlets could perform privacy preserving filtering operations, or could be used to prepare data for visualization or reporting purposes. Storlets can also be used to pre-process data before it is fed into an analytics or machine learning computation. Alternatively the machine learning computation could be run as a storlet. The use of storlets has several advantages in the COSMOS context

- Avoid sending large amounts of data across the network – apply storlets to send only the data which is necessary to send. For example
    - pre-process data thereby reducing its size and perform some needed calculations before sending to machine learning for further processing
    - apply machine learning algorithms as storlets directly to the data and avoid the need to send data across the network altogether
    - prepare data for visualization. Such data may be presented to the user by a browser or new objects may be created for visualization purposes. In the latter case, if these objects were to be created outside the cloud storage, the corresponding data would need to be sent across the network in both directions, assuming it needs to be retained in the cloud storage for future use. This can be avoided using storlets.
- Apply privacy preserving filters so that only privacy filtered data leaves the cloud storage
    - Such filters could transform or hide certain information. Examples of privacy preserving storlets will be described in section

A diagram of how storlets (Analysis Close to the Data) fit into the WP4 architecture can be found in section 3 High Level Architecture of the D4.1.1 Information and Data Lifecycle Management: Design and open specification (Initial) document. In addition deliverable D2.3.1 discusses the COSMOS overall architecture.

## 6.1.2. Technical description

### *6.1.2.1. Prototype architecture*

The prototype is built of the following components (See figure 8 in section 4.4.2.3 in the M8 scientific report document).

1. A Swift cluster, augmented with a middleware plug-in that allows invoking the storlet processing.
2. A Linux container that runs on each of the cluster nodes.
3. A per storlet daemon. A JVM process that runs inside the Linux container. The daemon loads a storlet code on startup and awaits execution requests.
4. An agent running inside the Linux container used to control the per storlet daemons. We refer to it as the 'daemon factory' below.

### *6.1.2.2. Components description*

**The Storlet middleware.** The Storlet middleware is made of two pieces. One piece is plugged into the Swift proxy server, and the other to the Swift object server. The role of the storlet proxy server middleware is twofold:

1. To intercept storlets upload and validate that they carry all the necessary metadata (e.g. the language in which they are written)
2. To authorize storlet execution requests.

The roles of the storlet middleware in the object server are:

1. Fetch a storlet code from the cluster upon first invocation, and copy it into the Linux container.
2. Bring up a daemon that can execute a certain storlet code. Specifically, this daemon loads the storlet code that was copied into the Linux container.
3. Forward storlet execution requests coming from the user to the above daemon.

**The Storlet Daemon.** The Storlet daemon is a generic daemon that can load given storlets and serve invocation requests on given data.

**The Daemon Factory.** A daemon process brought up with the Linux container, used to start and stop the execution of storlet daemons.

**The Storlet API Library.** A library that defines the interface a storlet needs to support, and the API's class definitions. See section 7.4.1 in the M8 scientific report.

**Schannel.** A communication channel between the Storlet middleware in the host side and the daemon factory and storlet daemon on the Linux container side. The channel is based on unix domain sockets.

### *6.1.2.3. Technical specifications*

Our prototype is built over Swift version 1.12. Swift as well as our middleware is written in Python using the WSGI framework. The daemon factory is written in python, the storlet

daemon as well as the Storlet API library are written in Java. Schannel is written in “C”, Python, and Java/JNI.

Most of the code is based on standard Python and Java libraries. The below libraries are used by various parts of the Storlet engine:

- Json-simple Apache 2.0
- logback-classic-1.1.2 Eclipse Public License - v 1.0, GNU Lesser General Public License
- logback-core-1.1.2 Eclipse Public License - v 1.0, GNU Lesser General Public License
- slf4j-api-1.7.7 MIT license

The below libraries are used as part of an example storlet that transform .pdf to .text and extract metadata from .pdf

- commons-logging-1.1.3 Apache 2.0
- fontbox-1.8.4 Apache 2.0
- jempbox-1.8.4 Apache 2.0
- pdfbox-1.8.4 Apache 2.0

## 6.2 Delivery and usage

### 6.2.1. Package information

The code is made of two modules:

1. swift++_deployment module. This module has configuration files and installation scripts required for installing Swift with Keystone as well as scripts for doing cluster wide installation of storlets.
2. nemo_storlet module. The module has the various components described above.

**swift++_deployment module.** The module consists of the following:

- cluster_config directory. A set of json files, each describes a cluster where we install Swift, Storlets and metadata search. Used by the various installation scripts.
- cluster_wide_constants. A json file with installation defaults.
- md_search_install. Installation scripts for the metadata search components
- swift_cluster_install. Installation scripts for Swift and Keystone.
- scp.py-master. An LGPL library used by the installer for scp operations.
- Paramiko. An LGPL library used by the installer for ssh operations.

**nemo_storlet module.** The module consists of the following:

build.xml – ant build files

schannel – The implementation of the communication channel between the host and the Linux container mentioned above.

storlet_daemon_factory - The implementation of the daemon factory mentioned above.

Storlet_Samples – Mainly the .pdf to .text converter storlet mentioned above.

system_tests – A bunch of system tests.

storlet_daemon - The implementation of the storlet daemon mentioned above.

StorletManager – A Java command line tool used for uploading storlets.

## 6.2.2. Installation instructions

Step 1: Preparing the Environment.

1. Make sure you have an eclipse installation with pydev, CDT and java.
2. Checkout the Storlets and swift++_deployment repositories.
3. The storlets repo has an eclipse project definition in its root directory (nemo_storlets), you will need to use it so that the java code will get compiled.

Step 2: Configure your development / deployment cluster

1. Edit your cluster configuration file. Examples can be found in the swift++_deployment repo under the cluster_config directory. If you are on a dev machine you probably want to look at localhost.json which has a single node.
2. Make sure that each node to be installed with storlets has the role 'storlet'. Also, make sure that the root password is updated.
3. Edit your cluster wide constants file (swift++_deployment/cluster_wide_constants). Leave it as is. Just make sure you know where it is. **Note** the file has an entry called 'lxc_device'. This entry points to a directory where all LXC related persistent data will be kept. Make sure that:
    1. The directory exists.
    2. It has full permissions (777).

To deploy storlets on a node (or any storlet sub module as described below) the node **must have the role 'storlet'**.

**Step 3: Building the code**

1. Auto build the storlet java code.
2. Use **ant** to build the sub module you are working with (or all modules if you are about to deploy everything)

The storlet packaging scripts **assume that the code was automatically built in Eclipse**. More specifically, the scripts will search for the bin directory under the nemo_storlets module. Once built in eclipse the first step is to go to the root dir of the module and do:

*ant all*

Step 4: Deploying the code

1. Make sure you got lxc installed on all nodes (apt-get install lxc).
2. Make sure you have paramiko and scp python libs installed. If prior to this deployment you have deployed Swift and Keystone using swift++_deployment then do not worry about it. Otherwise, do:
    - cd swift++_deployment/swift_cluster_install
    - python install_dependencies.py ssh
3. Run the following from the nemo_storlet/deploy directory:

**python management_install.py install /root/workspace/storlets/nemo_storlet storlets_modules.json cluster_configuration cluster_wide_constants local_install.sh all**

The parameters are:
Install - in the future we will also support "remove"
/root/workspace/storlets/nemo_storlet - the path to the root directory of the nemo_storlet module.
storlets_modules.json - a json file representing a list of all supported storlet modules. Located in the swift++_deployment module.
cluster_configuration - a json file representing the cluster configuration (more information on the cluster configuration file appears below). For examples, take a look at the json files in the swift++_deployment/cluster_config/ directory
cluster_wide_constants - a json file representing the cluster wide constants. For an example, take a look at swift++_deployment/cluster_wide_constants. This constants file replaces the constants previously in package_deployment_constants.py
local_install.sh - the script to be run on each cluster node. It can be found in the deploy directory.

## 6.2.3. User Manual

**Overview**

This section describes how to write, deploy and execute a storlet. The instructions are user oriented and assume you already have a storlet enabled swift cluster deployed. Storlets can be invoked as follows:

1. As part of a GET, where the object's data appearing in the GET request is the storlet's input and the response to the GET request is the storlet's output.
2. As part of a PUT, where the request body is the storlet's input, and the actual object saved in Swift is the storlet's output.

**How to Write a Storlet**
In this paragraph we cover:

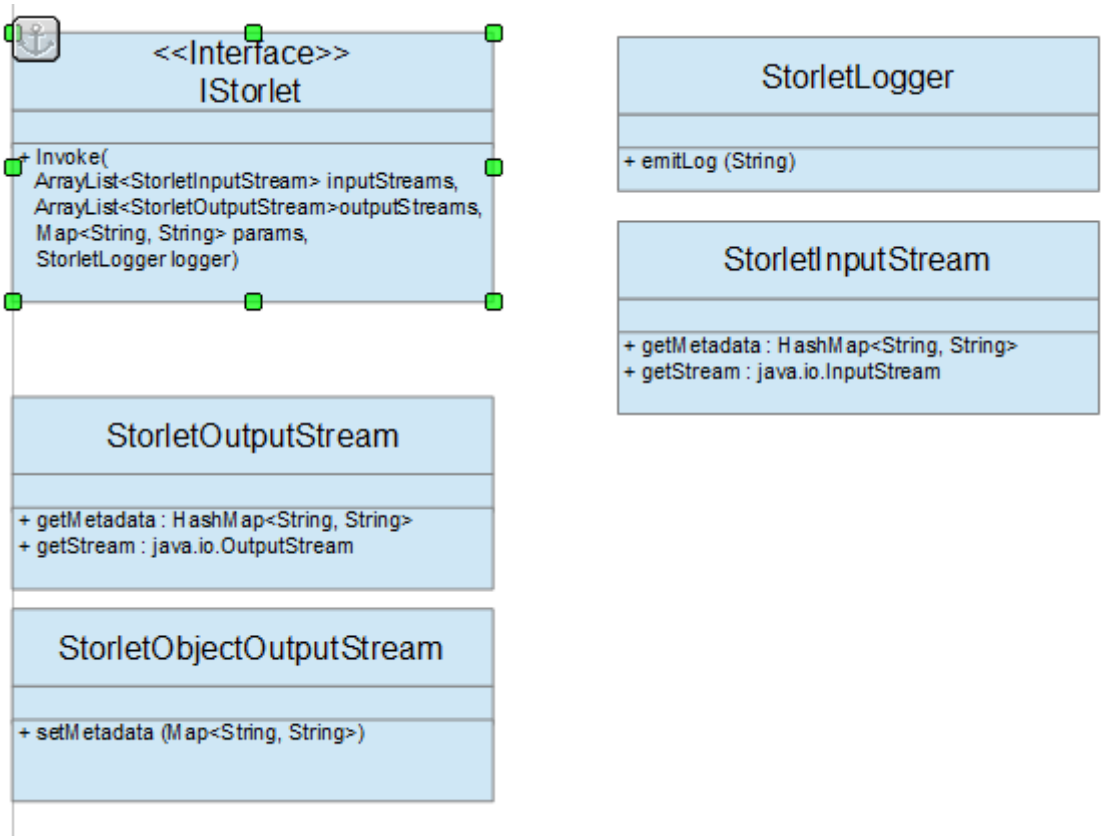
1. How to write a storlet.
2. The best practices of writing a storlet.
3. StorletTranscoder - An example of a storlet.

Writing a Storlet.

Storlets can currently only be written in Java. To write a storlet you will need the *storletcommonapi-10.jar which is built as part of the installation process*. Import the .jar to Java project in Eclipse and implement the com.ibm.storlet.common.IStorlet interface. The interface has a single method that looks like this:

```
public void invoke(ArrayList<StorletInputStream> inStreams,
                   ArrayList<StorletOutputStream> outStreams,
                  Map<String,String> parameters, StorletLogger logger)
throws StorletException;
```

Here is a class diagram illustrating the classes involved in the above API.




The StorleInputStream is used to stream in object's data into the storlet. It is used both in the GET scenario as well as in the PUT scenario to stream in the object's content. In the GET scenario it is the content of the object in the store to be processed and streamed to the user. In the PUT scenario it is the content of the user's uploaded data to be streamed to the store as an object. To consume the data do a getStream() to get a java.io.InputStream on which you can just read().

The StorleOutputStream is a base class for the StorletObjectOutputStream. When the storlet is invoked it will **never** be with the base class.

In the PUT scenario the storlet is called with an instance of StorletObjectOutputStream. You will need to first need to call the setMetadata function to set the appropriate metadata of the to be created object, and then use getStream to get a java.io.OutputStream on which you can call write() with the content of the object. It is important to note that metadata **cannot** be set once you started to stream out data via the java.io.OutputStream. Also, one needs to set the metadata atmost 40seconds from invocation, otherwise, a timeout occurs.

The StorletLogger class supports a single method called emitLog, and accepts only String type. Each invocation of the storlet would result in a newly created object that contains the emitted logs. This object is located in a designated container called *storletlog* and will carry the name *<storlet_name>.log*. Creating an object containing the logs per request has its overhead. Thus, the actual creation of the logs object is controlled by a header supplied during storlet invocation. More information is given in the storlet execution section below.

When invoked via the Swift GET REST API (exact details below), the invoke method will be called as follows:

1. The inStreams array would include a single element representing the object to read.
2. The outStreams would include a single element representing the response returned to the user. Anything written to the output stream is effectively written to the response body returned to the user's GET request.
3. The parameters map includes execution parameters sent. These parameters can be specified in the storlet execution request as described in the execution section below. IMPORTANT: Do not use parameters that start with 'storlet_' these are kept for system parameters that the storlet can use. Currently we have: 'storlet_execution_path' which carries the full path (as seen by the code running in the container) where the storlet code runs. This is also where all dependencies reside.
4. A StorletLogger instance.

When invoked via the Swift PUT REST API , the invoke method will be called as follows:

1. The inStreams array would include a single element representing the object to read.
2. The outStreams would include a single element which is an instance of StorletObjectOutputStream.
3. The parameters, and StorletLogger as in the GET call.

The compiled class that implements the storlet needs to be wrapped in a .jar. This jar **must not** include the storletcommonapi-1.0.jar. Any jars that the class implementation is dependent on should be uploaded as separate jars as shown in the deployment section below.

**Best Practices of Storlet Writing**

- Storlets are tailored for stream processing, that is, process the input as it is read and produce output while still reading. In other words a 'merge sort' of the content of an object is not a good example for a storlet as it requires to read all the content into memory (random reads are not an option as the input is provided as a stream). While we currently do not employ any restrictions on the CPU usage or memory consumption of the storlet, reading large object into memory or doing very intensive computations would have impact on the overall system performance.
- Once the storlet has finished writing the response, it is **important to close** the output stream. Failing to do so will result in a timeout.
- With the current implementation, a storlet must start to respond within 40 seconds of invocation. Otherwise, Swift would timeout.
- The call to setMetadata **must** happen before the storlet starts streaming out the output data. Note the applicability of the 40 seconds timeout here.
- While this might be obvious it is advisable to test the storlet prior to its deployment.
- The storlets are executed in an **open-jdk 7 environment.** Thus, any dependencies that the storlet code requires which are outside of open-jdk 7 should be stated as storlet dependencies and uploaded with the storlet. Exact details are found in the deployment section below.

**Storlet Deployment**

Storlet Deployment Principles

Storlet deployment is essentially uploading the storlet and its dependencies to designated containers in the account we are working with. While a storlet and a dependency are regular Swift objects, they must carry some metadata used by the storlet engine. When a storlet is first

executed, the engine fetches the necessary objects from Swift and 'installs' them in the Linux container. Note that the dependencies are meant to be small. Having a large list of dependencies or a very large dependency may result in a timeout on the first attempt to execute a storlet. If this happens, just re-send the request again.

Following is an example for uploading a storlet that transforms .pdf to .text. It is called TranscoderStorlet and has 4 dependencies:

1. The storlet packaged in a .jar. In our case the jar was named: storlettranscoder-10.jar The jar needs to be uploaded to a container named **storlet**. The name of the uploaded storlet **must** be of the form <name>-<version>.The metadata that **must** accompany a storlet is as follows:
    - X-Object-Meta-Storlet-Language - currently must be **'java'**
    - X-Object-Meta-Storlet-Interface-Version - currenltly we have a single version **'1.0'**
    - X-Object-Meta-Storlet-Dependency - A comma seperated list of dependent jars. In our case: **'commons-logging-1.1.3.jar,fontbox-1.8.4.jar,jempbox-1.8.4.jar,pdfbox-app-1.8.4.jar'**
    - X-Object-Meta-Storlet-Object-Metadata - Currently, not in use, but must appear. Use the value **'no'**
    - X-Object-Meta-Storlet-Main - The name of the class that implements the IStorlet API. In our case: '**com.ibm.storlet.transcoder.TranscoderStorlet**'
2. The .jar files that the storlet code is dependent on. The below jars are the storlettranscoder dependencies. These should be uploaded to a container named dependency. The metadata that **must** accompany a dependency is its version as follows:
    - X-Object-Meta-Storlet-Dependency-Version - While the engine currently does not parse this header, it must appear.
        - commons-logging-1.1.3.jar
        - jempbox-1.8.4.jar
        - fontbox-1.8.4.jar
        - pdfbox-app-1.8.4.jar

If one wishes to update the storlet just upload again, the engine would recognize the update and bring the updated code.

**Important**: Currently, dependency updates are not recognized, only the Storlet code itself can be updated.

Deploying a Storlet with Python

Here is a code snippet that uploads both the storlet as well as the dependencies. The code was tested against a Swift cluster with:

1. Keystone configured with a 'service' account, having a user 'swift' whose password is 'password'
2. Under the service account there are already 'storlet', 'dependency', and 'storletlog' containers.

```
from swiftclient import client as c
def put_storlet_object(url, token, storlet_name,
local_path_to_storlet, main_class_name, dependencies):
    # Delete previous storlet
    resp = dict()

    metadata = {'X-Object-Meta-Storlet-Language':'Java',
                'X-Object-Meta-Storlet-Interface-Version':'1.0',
                'X-Object-Meta-Storlet-Dependency': dependencies,
                'X-Object-Meta-Storlet-Object-Metadata':'no',
                'X-Object-Meta-Storlet-Main': main_class_name}
    f = open('%s/%s' % (local_path_to_storlet, storlet_name), 'r')
    content_length = None
    response = dict()
    c.put_object(url, token, 'storlet', storlet_name, f,
                 content_length, None, None, "application/octet-
stream", metadata, None, None, None, response)
    print response
    f.close()
    status = response.get('status')
    assert (status == 200 or status == 201)

def put_storlet_dependency(url, token, dependency_name,
local_path_to_dependency):
    metadata = {'X-Object-Meta-Storlet-Dependency-Version': '1'}
    f = open('%s/%s'% (local_path_to_dependency, dependency_name),
'r')
    content_length = None
    response = dict()
    c.put_object(url, token, 'dependency', dependency_name, f,
                 content_length, None, None, "application/octet-
stream", metadata, None, None, None, response)
    print response
    f.close()
    status = response.get('status')
    assert (status == 200 or status == 201)
AUTH_IP = '127.0.0.1'
AUTH_PORT = '5000'
ACCOUNT = 'service'
```

```
USER_NAME = 'swift'

PASSWORD = 'password'

os_options = {'tenant_name': ACCOUNT}

url, token = c.get_auth("http://" + AUTH_IP + ":" + AUTH_PORT +
"/v2.0", ACCOUNT +":"+USER_NAME, PASSWORD, os_options = os_options,
auth_version="2.0")

put_storlet_object(url, token,'storlettranscoder-
10.jar','/tmp' ,'com.ibm.storlet.transcoder.TranscoderStorlet',
'commons-logging-1.1.3.jar,fontbox-1.8.4.jar,jempbox-1.8.4.jar,pdfbox-
app-1.8.4.jar')

put_storlet_dependency(url, token,'commons-logging-1.1.3.jar','/tmp')

put_storlet_dependency(url, token,'fontbox-1.8.4.jar','/tmp')

put_storlet_dependency(url, token,'jempbox-1.8.4.jar','/tmp')

put_storlet_dependency(url, token,'pdfbox-app-1.8.4.jar','/tmp')
```

**How to Execute a Storlet**

Once the storlet and its dependencies are deployed the storlet is ready for execution, and can be invoked. Invocation via PUT and GET involves adding an extra header to the Swift original PUT/GET requests. Below we invoke the TranscoderStorlet in both PUT and GET. Let us assume that we have uploaded a pdf document called **example.pdf** to a container called **my_container** as appearing in the following Swift URL (again, using the pre-configured account)

http://sde.softlayer.com/AUTH_2dc1440a41e94fc696bced36c6e3c249/my_container/example.pdf

Here is how we can invoke the storlet using Curl, where auth_header is the X-Auth-Header used with Swift.

```
curl -i -X GET
http://sde.softlayer.com/v1/AUTH_2dc1440a41e94fc696bced36c6e3c249/my_c
ontainer/example.pdf -H$auth_header

-H'X-Run-Storlet:storlettranscoder-10.jar'
```

Note the extra header 'X-Run-Storlet' specifying the name of the storelt to execute. When this header is specified, the storlet engine wsgi middleware intercepts the request, activates the storlet and returns the computation result as a repsonse.

To invoke a storlet whose logs will be available as an object, use the below. Note that a container named 'storletlog' needs to be created under the account prior to this.

```
curl -i -X GET
http://sde.softlayer.com/v1/AUTH_2dc1440a41e94fc696bced36c6e3c249/my_c
ontainer/example.pdf -H$auth_header

-H'X-Run-Storlet:storlettranscoder-10.jar' -H'X-Storlet-Generate-
Log:True'
```

Once executed with the generate log header set to true, one can download the resulting object as follows. Note that object name is derived from the storlet name (truncating the version number suffix and adding a .log suffix):

```
curl -i -X GET
http://sde.softlayer.com/v1/AUTH_2dc1440a41e94fc696bced36c6e3c249/stor
letlog/storlettranscoder.log
```

Passing parameters to the storlets is done using the query string, e.g.

```
curl -i -X GET
http://sde.softlayer.com/v1/AUTH_2dc1440a41e94fc696bced36c6e3c249/my_c
ontainer/example.pdf?arg1=value1&arg2=value2 -H$auth_header

-H'X-Run-Storlet:storlettranscoder-10.jar'
```

Now lets assume that we have a local file called example.pdf and we want to keep it as text only. Here is a regular PUT request:

```
curl -i -X PUT
http://sde.softlayer.com/v1/AUTH_2dc1440a41e94fc696bced36c6e3c249/my_c
ontainer/example.txt -H$auth_header

-F filedata=/tmp/example.pdf
```

Here is how to PUT it invoking a storlet:

```
curl -i -X PUT
http://sde.softlayer.com/v1/AUTH_2dc1440a41e94fc696bced36c6e3c249/my_c
ontainer/example.txt -H$auth_header

-H'X-Run-Storlet:storlettranscoder-10.jar' -F
filedata=/tmp/example.pdf
```

### 6.2.4. Licensing information

Swift is under apache license 2.0. LXC user space tools are under LGPL. Otherwise, we are using Python 2.7 standard libraries, and standard openjdk-7 libraries. The additional libraries licenses appear in the technical specification section above.

The storlets source code should be considered confidential i.e. accessible only by COSMOS partners and reviewers from the EU.

### 6.2.5. Download

Selected source code is available on the COSMOS SVN, under SourceCode\M10 Prototypes\WP4\CloudStorage

# 7 Cloud Storage – Security and Privacy

## 7.1 Implementation

### 7.1.1. Functional description

Note that this section describes work belonging to the WP3 work package. It belongs in this document (also according to the DoW) because its prototype source code is closely tied to the prototype source code of the Cloud Storage components which belong to WP4.

There are many important security and privacy aspects related to cloud storage. We mention here 3 security and privacy aspects of the cloud storage components developed for COSMOS

1. Privacy preserving storlets – examples are
    - A facial blurring storlet which operates on images stored in the cloud storage. It detects human faces and blurs the details so that the person cannot be identified.
    - A storlet which masks exact street addresses and reveals only the neighbourhood or postcode
    - A storlet which masks the exact GPS location and reveals only an approximate location
2. Sandboxing of storlets
    - Storlets are sandboxed using linux containers and are only given access to the storage objects they are authorized to access. They are not given permissions to access the network or the file system of the underlying container. This allows running possibly buggy or potentially malicious code written by a wide range of users on the cloud storage system while still protecting the system as a whole as well as the rest of the cloud storage data.
3. Metadata search whose results contain only resources that the user is authorized to access
    - The functionality was described in document D3.1.1 End-to-End Security M8 deliverable.

#### *7.1.1.1. Fitting into overall COSMOS solution*

1. Privacy preserving storlets can be applied when objects are retrieved, before returning data to the user. In this way, complete raw data can be stored within the cloud storage but only privacy filtered data is returned to the user.
2. Sandboxing of storlets is especially important in the future if we want to allow arbitrary users to write storlet code for the COSMOS platform.
3. Metadata search whose results contain only resources that the user is authorized to access – this is important in COSMOS in order to ensure that metadata search does not enable users to have access to more data or metadata than they should be.

### 7.1.2. Technical description

This work is not a separate component but rather is part of the storlets and metadata search components. Therefore please see the relevant sections of this document describing storlets and metadata search.

#### *7.1.2.1. Prototype architecture*

This work is not a separate component but rather is part of the storlets and metadata search components.

#### *7.1.2.2. Components description*

This work is not a separate component but rather is part of the storlets and metadata search components.

#### *7.1.2.3. Technical specifications*

The facial blurring privacy preserving storlet uses the OpenCV open source computer vision and machine learning software library. See http://opencv.org/

Storlet sandboxing has been implemented as part of the storlets implementation using LXC containers.

Metadata search security has not yet been implemented and will be implemented in a later stage of the project.

## 7.2 Delivery and usage

### 7.2.1. Package information

Please see the relevant sections for storlets and metadata search.

### 7.2.2. Installation instructions

Please see the relevant sections for storlets and metadata search.

### 7.2.3. User Manual

Please see the relevant sections for storlets and metadata search.

### 7.2.4. Licensing information

Please see the relevant sections for storlets and metadata search. In addition OpenCV is released using a BSD licence.

### 7.2.5. Download

Please see the relevant sections for storlets and metadata search.

# 8 Conclusions

This document describes the prototypes for the Information and Data Lifecycle Management Work Package. Each component has been implemented independently, and now the various components need to be integrated. This is the initial prototype for our work in COSMOS, which will be revised in years 2 and 3 of the project.