COSMOS

Cultivate resilient smart Objects for Sustainable city applicatiOnS

Grant Agreement Nº 609043

D4.2.2 Information and Data Lifecycle Management: Software prototype (Updated)

WP4 Information and Data Lifecycle Management

Version: 1.0
Due Date: 30/6/2015
Delivery Date: 30/6/2015
Nature: Prototype
Dissemination Level: Public

Lead partner: IBM
Authors: Adnan Akbar (University of Surrey), Guy Hadash (IBM), Achilleas Marinakis (NTUA), Eran Rom (IBM), Juan Sancho (ATOS), Paula Ta-Shma (IBM), Yaron Weinsberg (IBM)
Internal reviewers: Franz Carrez (Uni. Surrey), Abie Cohen (Hildebrand)

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	27/05/2015	Paula Ta-Shma	IBM	Initial version based on last year's deliverable.
0.2	9/06/2015	Paula Ta-Shma, Guy Hadash	IBM	Added the scalable data mapper section.
0.3	9/06/2015	Eran Rom	IBM	Updated the storlets section
0.4	11/06/2015	Paula Ta-Shma	IBM	Updated the metadata search section
0.5	11/06/2015	Yaron Weinsberg	IBM	Filled in the cloud storage – security and privacy section
0.6	12/06/2015	Juan Sancho	ATOS	Reworked CEP section
0.7	18/06/2015	Paula Ta-Shma	IBM	Integration with an Analytics Framework section
0.8	24/06/2015	Juan Sancho	ATOS	Reworked Message Bus section
0.9	24/06/2015	Paula Ta-Shma, Guy Hadash	IBM	Updated Integration with an Analytics Framework section
0.10	25/06/2015	Paula Ta-Shma	IBM	Finalized Integration with an Analytics Framework section. Updated Message Bus section. Updated Cloud Storage Security and Privacy section. Updated conclusions.
0.11	25/06/2015	Juan Sancho	ATOS	Misc. Updates
0.12	30/6/2015	Juan Sancho	ATOS	Addressed review comments
0.13	30/6/2015	Paula Ta-Shma	IBM	Addressed review comments
1.0	30/06/2015	Paula Ta-Shma	IBM	Release version

Table of Contents

List of Figures

1 Introduction

1.1 About this deliverable

This document is the complement to the delivered software as prototype for deliverable D4.2.2 Information and Data Lifecycle Management: Software prototype (Updated). For information on the motivation, architecture and design of the components in this work package, please refer to document D4.1.2 Information and Data Lifecycle Management: Design and open specification (Updated) [1]. This document contains information and technical specifications of the prototype software implemented for this work package.

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. There is also a new section in Year 2 on Integration with an Analytics Framework. 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 (D4.2.2).

2 Complex Event Processing

2.1 Implementation

2.1.1. Functional description

A Complex Event Processing (CEP) engine is used in COSMOS in order to analyse several streams of data and create value-added information (complex events) that will be consumed by application developers. The input *data sources* are diverse, ranging from direct VE sensing data to even third-party Open data services. Due to that, we created a specific *template translation process* that takes care of readjusting heterogeneous ways of data formats to the one needed by the CEP engine in place. Given that in this project the selected engine is the µCEP, said process translates the input data formats to the specific one defined by the DOLCE language. However, it is possible to readjust to any other format in case a different stream analytics platform is used. In addition, the generated outputs are being converted to a more standard message representation like JSON so to facilitate its exploitation by other services.

2.1.1.1. Fitting into overall COSMOS solution

What application developers are able to perform using a CEP engine mainly depends on the defined rules, like filtering, aggregation, transforming, splitting or detecting abnormal behaviour. It is true that some of these features are always possible to be implemented here and there in the code –meaning that application developers can opt to use their own methods into their programs to face certain data analytics operations—. However, COSMOS provides application developers with a CEP engine component:

- for them to get rid of adding analytical libraries to their programs,
- so to keep programs simpler,
- what allows them focusing on the real problems in hand.

Many software gurus began recently to call this with the term *Microservices*, which are *a particular way of designing software applications as suites of independently deployable services*[1].

2.1.2. Technical description

2.1.2.1. Prototype architecture

Regarding the prototype architecture, please see Section 4.2.3 CEP System Architecture of the D4.1.2 Information and Data Lifecycle Management: Design and open specification (Updated) document [1]. Diagrams depicting the architecture can be found there.

2.1.2.2. Components description

Regarding the components description, please see Section 4.2.3 CEP System Architecture of the D4.1.2 Information and Data Lifecycle Management: Design and open specification (Updated) document [1]. Text describing the components in the architecture can be found there.

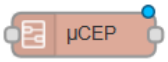
[1] Martin Fowler – Microservices: http://martinfowler.com/articles/microservices.html

2.1.2.3. Technical specifications

The modular µCEP engine built in COSMOS allows its deployment into high performance machines as well as constrained embedded devices. Despite the fact that the initial version of this deliverable described a running implementation of a previous work, during Year 2 we have been performing several fixings in order to realign our work, addressing several comments received by this platform, and making the engine to fit better into an IoT ecosystem. On the one hand, we disassembled the whole engine so to make it modular, thus being deployable in three different manners as explained in section 8.1.3.4 of deliverable D2.3.2 Conceptual Model and Reference Architecture. On the other hand, not only we get rid of unnecessary libraries and dependencies, but we also switched other ones with the goal of decreasing the amount of memory required, reaching a stable version that consumes less than 10 MB of memory.

2.1.2.4. Functionalities to be offered through Node-RED

As Node-RED [3] enforces a Flow Base Programming paradigm, we are able to represent said modularization using *Subflows*, which are "self-defined flows" encapsulated in a node. Subflows can be exported in JSON to other Node-RED workspaces in an easy way, thus making it easy and straightforward to share the engine among the Use Cases and prototypes. The following node represents the µCEP engine defined as a *subflow*:



The code below is a JSON object that represents what is inside the node:


```
[{"id":"50ef7066.cf7268","type":"mqtt-broker","broker":"lab.iot-cosmos.eu","port":"1883","clientid":""},{"id":"ccb7b270.e8b22","type":"udp out","name":"Send to Local
µCEP","addr":"localhost","iface":"","port":"50001","outport":"","base64":false,"multicast":"false","x":874.6943969726562,"y":241.80555725097656,"z":"53eff815.0c48c","wires":[]},
{"id":"6ad8fa07.950a9c","type":"template","name":"Message Translator","field":"payload","template":"1 TrafficInfo string codigo {{codigo}} int TrafficSpeed {{velocidad}} int
TrafficIntensity ts {{ts}} string tf
{{tf}}","x":442.3611297607422,"y":261.80555629730225,"z":"53eff815.0c48c","wires":[["5d8f548c.9b989c","9d7a3be0.cfb87"]]},{"id":"e4c2999e.f7a59","type":"mqtt out","name":"Send
to remote
µCEP","topic":"cosmos/uCEP/event","qos":"0","retain":"false","broker":"50ef7066.cf7268","x":879.6943969726562,"y":161.80555725097656,"z":"53eff815.0c48c","wires":[]},{"id":"37ff
faaf.5e7c3e","type":"udp in","name":"Receive from Local
µCEP","iface":"","port":"50002","multicast":"false","group":"","datatype":"utf8","x":318.66664123535156,"y":635.2778196334839,"z":"53eff815.0c48c","wires":[["da4c8e45.a72e28","b
b4889f2.31928"]]},{"id":"9d868ed9.666b98","type":"mqtt out","name":"Generated Complex
Events","topic":"cosmos/MB/complexEvent","qos":"0","retain":"false","broker":"50ef7066.cf7268","x":1007.6666412353516,"y":475.2778196334839,"z":"53eff815.0c48c","wires":[]},{"id
":"da4c8e45.a72e28","type":"function","name":"CEP to JSON","func":"var cadena = msg.payload;\n\nvar cadenaDividida = cadena.replace(/(\\r\\n|\\n|\\r)/gm,\"\").split(' ');\n\nvar
hijos = {};\n\nfor(i=2; i<cadena.length-1;i+= 3){\n  if(typeof(cadenaDividida[i+1]) !== \"undefined\" && typeof(cadenaDividida[i+2]) !== \"undefined\"){\n
hijos[cadenaDividida[i+1]] = cadenaDividida[i+2];\n  \n}\nvar padre = {};\n\npadre[cadenaDividida[1]] = hijos;\n\nmsg.payload = padre;\n\nreturn
msg;\n","outputs":1,"valid":true,"x":584.6666412353516,"y":575.2778196334839,"z":"53eff815.0c48c","wires":[["d90c2f41.85931","98a3d1d9.4d094"]]},{"id":"dba59d15.1187c","type":"c
omment","name":"------------ uCEP Complex Events to MB ------------
","info":"","x":596.2221374511719,"y":434.3888998031616,"z":"53eff815.0c48c","wires":[]},{"id":"be357400.af9b5","type":"mqtt in","name":"Receive from remote
µCEP","topic":"cosmos/uCEP/complexEvent","broker":"50ef7066.cf7268","x":323.66664123535156,"y":515.2778196334839,"z":"53eff815.0c48c","wires":[["da4c8e45.a72e28"]]},{"id":"e2f2e
033.8d298","type":"comment","name":"------------ Events to µCEP ------------
","info":"","x":598.166748046875,"y":134.72225856781006,"z":"53eff815.0c48c","wires":[]},{"id":"5d8f548c.9b989c","type":"file","name":"","filename":"Event.cep","appendNewline":t
rue,"overwriteFile":"false","x":662.6943969726562,"y":321.80555725097656,"z":"53eff815.0c48c","wires":[]},{"id":"bb4889f2.31928","type":"file","name":"","filename":"complexEvent
.cep","appendNewline":true,"overwriteFile":"false","x":598.6666412353516,"y":695.2778196334839,"z":"53eff815.0c48c","wires":[]},{"id":"d90c2f41.85931","type":"file","name":"","f
ilename":"complexEvent.json","appendNewline":true,"overwriteFile":"false","x":800.1666412353516,"y":635.2778196334839,"z":"53eff815.0c48c","wires":[]},{"id":"9d7a3be0.cfb87","ty
pe":"switch","name":"","property":"payload","rules":[{"t":"eq","v":""}],"checkall":"true","outputs":1,"x":653.1943969726562,"y":201.80555725097656,"z":"53eff815.0c48c","wires":[
["e4c2999e.f7a59","ccb7b270.e8b22"]]},{"id":"98a3d1d9.4d094","type":"switch","name":"","property":"payload","rules":[{"t":"eq","v":""}],"checkall":"true","outputs":1,"x":763.166
6412353516,"y":515.2778196334839,"z":"53eff815.0c48c","wires":[["9d868ed9.666b98"]]},{"id":"a86cb9a0.29c628","type":"file","name":"","filename":"Event.json","appendNewline":true
,"overwriteFile":"false","x":418.1665954589844,"y":336.94445419311523,"z":"53eff815.0c48c","wires":[]},{"id":"10c004f0.5e5a4b","type":"mqtt in","name":"Receive from
MB","topic":"cosmos/uCEP/complexEvent","broker":"50ef7066.cf7268","x":213.33331298828125,"y":235.27778911590576,"z":"53eff815.0c48c","wires":[["6ad8fa07.950a9c"]]}]
```


Once copy-pasted and imported, one would obtain the following 2 subflows:

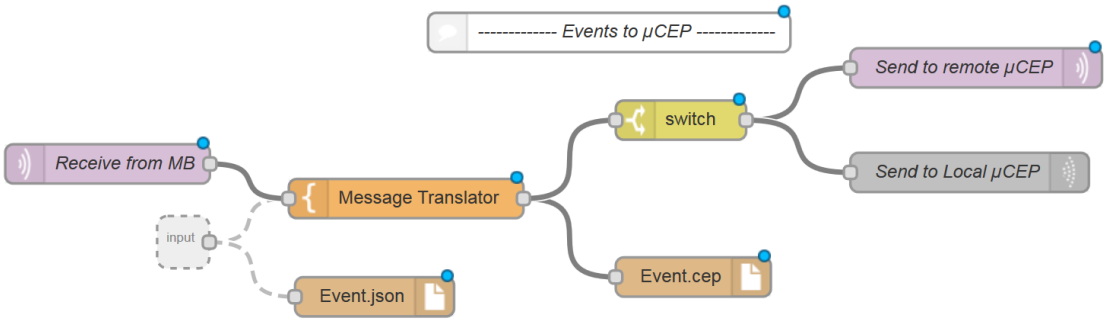


Figure 1: Event collection step

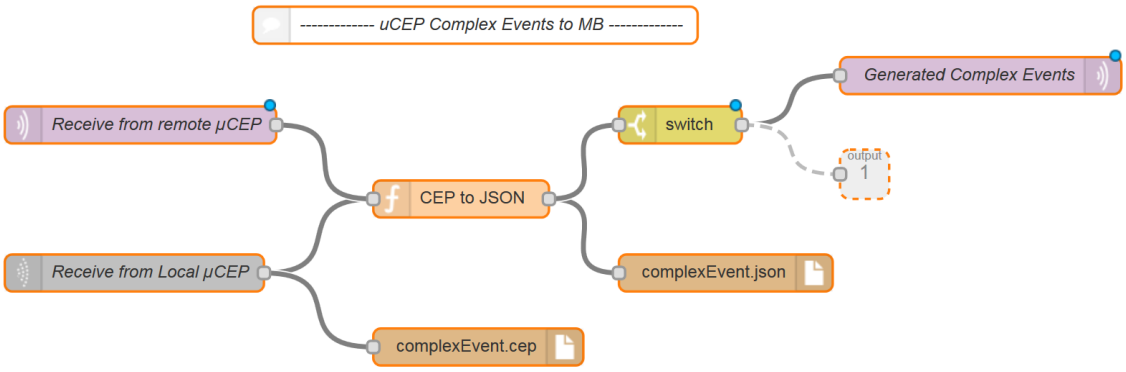


Figure 2: Complex Event publication step

Both Figure 1 and Figure 2 represent the interaction with the µCEP engine. They encapsulate in a single node the actions required to handle the usage of both remote and local engines. These subflows even capture datasources from within the local machine where the engine is deployed or from the Message Bus (MB); the same applies to the generated output complex events.

2.2 Delivery and usage

2.2.1. Package information

The µCEP engine needs 3 files to run:

- ucep: binary executable file that runs the core application
- confFile.ini: collects required parameters for the communication interfaces
- detect.dolce: defines the rules that will be apply to incoming datasources

The *flows* used by Node-RED are stored in the respective Node-RED user's directory. Additional nodes needed (freeboard, splitter, kafka, etc.) are installed in that same folder.

- $HOME/.node-red/flows.json: a JSON file with the latest deployed flows
- $HOME/.node-red/node_modules: legacy additional nodes

2.2.2. Installation instructions

The µCEP engine doesn't need to be installed but be placed in a directory with the right permissions to be executed. What has to be paid attention to is using the engine released for the appropriate running OS: Ubuntu, CentOS and Raspbian are supported.

Node-RED can be installed using Node Package Manager:

```
$ npm install -g node-red
```

Then launched using PM2:

```
$ pm2 start -f -n cosmos_node-red /usr/bin/node-red -- -v --settings $HOME/.node-red /settings.js
```

Which is also easy to install just executing:

```
$ npm install -g pm2
```

2.2.3. User Manual

The µCEP engine behaves autonomously; it is always expecting an input event, evaluates the rules defined in detect.dolce file and generates and output. The rules have to comply with the DOLCE language as explained in D4.1.2 [1]. They can be updated by hand, using a Node-RED flow or even using an HTTP API, as explained in D5.1.2 [4]. In this section we are focusing in running the engine (and by extension, the desired Node-RED instances) the proper way.

Given a situation in which one would like to run an application (µCEP and/or the Node-RED instances) in background, the most common way do so is `nohup ./ucep &`, but this is not suitable if the machine reboots. An improved way could be the creation of the typical `/etc/init.d/my_ucep` script in which we copy-paste the quasi-standard start-stop-status-reload rutines, and then `chkconfig --add my_ucep`, and then `chkconfig my_ucep on` and finally `service my_ucep start`. But still, if the application shutdowns unexpectedly (for whatever reason) we are finished. The solution comes by using PM2 [5]. Tell it to handle the app:

```
$ pm2 start -f -n my_ucep ucep
```

Looking at the applications managed by PM2:

```
$ pm2 list
```

```
[root@lab ~]# pm2 list
```

App name	id	mode	pid	status	restart	uptime	memory	watching
node-red	0	fork	24280	online	0	2D	127.383 MB	disabled
node-red-atos	1	fork	24033	online	32	2D	94.438 MB	disabled
kafka_bridges	2	fork	28328	online	0	5D	1.199 MB	disabled
my_ucep	7	fork	8071	online	3	14h	1.340 MB	disabled

Figure 3: List of processes in PM2

To get more info about a specific one use its name or id:

```
$ pm2 info my_ucep
```

```
[root@lab ~]# pm2 info my_ucep
Describing process with id 7 - name my_ucep
```

status	online
name	my_ucep
id	7
path	/home/juan.sancho/projects/ucep/ucep
args	
exec cwd	/home/juan.sancho/projects/ucep
error log path	/root/.pm2/logs/my-ucep-error-7.log
out log path	/root/.pm2/logs/my-ucep-out-7.log
pid path	/root/.pm2/pids/my-ucep-7.pid
mode	fork_mode
node v8 arguments	
watch & reload	❯
interpreter	none
restarts	3
unstable restarts	0
uptime	14h
created at	2015-06-10T17:49:45.733Z

Figure 4: Details of a process

PM2 keeps track of the console's output of the app. Access to these logs at `./pm2/logs` folder or just have a look at it the `tail -f` way:

```
$ pm2 logs my_ucep
```

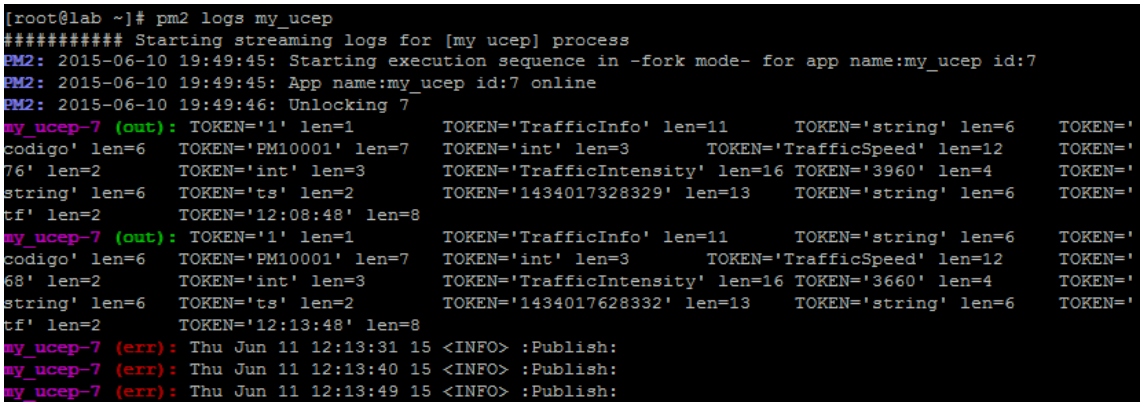


Figure 5: Logging output of a process

It is also possible to monitor the consumption of each app:

```
$ pm2 monit
```

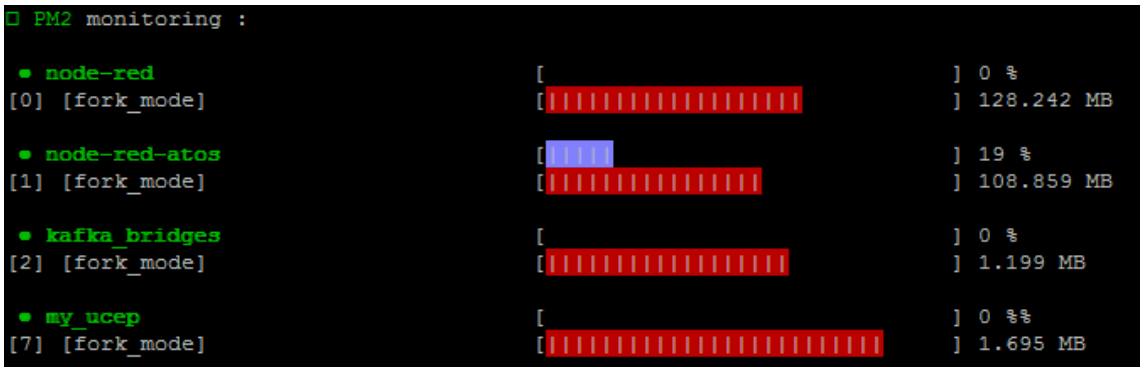


Figure 6: Monitoring processes

Finally, to ensure that PM2 starts after the machine reboots and loads your apps, issue the following command (what creates a `/etc/init.d/script`, indeed):

```
$ pm2 startup [platform] // ubuntu, centos, raspberry-pi
```

2.2.4. Licensing information

Currently, the µCEP engine is distributed as closed source software. However, first discussions have been carried out by the ATOS Internet of Everything Lab in order to release it as Open Source software, most probably distributed under Apache 2.0 license. PM2 is made available under the terms of the GNU Affero General Public License 3.0 (AGPL 3.0). Node-RED is distributed under the Apache 2.0 license.

2.2.5. Download

The µCEP engine, currently release 1.0.14, can be downloaded for each operating system at http://lab.iot-cosmos.eu/ucep/download. Source code of PM2 and Node-RED can be found in Github at https://github.com/Unitech/PM2 and https://github.com/node-red/node-red respectively.

3 Data Mapper

3.1 Implementation

3.1.1. Functional description

Data mapping is used in COSMOS in order to collect raw data published by virtual entities to the message bus and store it as data objects, with their associated metadata, in object storage.

The scalable data mapper is based on the open source Secor tool developed by Pinterest. Secor is a service which allows persistently storing topics from Apache Kafka in Amazon S3. We enhanced Secor by enabling OpenStack Swift targets, so that data can be uploaded by Secor to Swift. In addition we enhanced Secor by enabling data to be stored in the Apache Parquet format, which is supported by Spark SQL. Moreover, we enhanced Secor to generate Swift objects with metadata.

We use the term 'scalable' data mapper to distinguish our year 2 version from the year 1 version of the data mapper. The year 2 version is a completely new implementation based on Secor which has been designed to support scalability. We plan to investigate the performance and scalability aspects in year 3 of the project.

3.1.1.1. Schema and data expectations

- Each data source (Kafka topic) has its own schema
- Schemas are described using Avro
  - We store them in a dedicated container in Swift
- Mandatory columns for each data source :
  - ts (timestamp) : preferably in epoch format
- For each data source, we also store a list of columns to index using metadata search in a schemas container in Swift
  - For these columns we generate the min and max values as metadata for each object
- The data mapper converts a collection of JSON records from a Kafka topic (representing a data source) to a parquet object
- We store the objects corresponding to each data source in a separate Swift container

Additional information and references can be found in Section 4.1.6 Scalable Data Mapper of deliverable D4.1.2 Information and Data Lifecycle Management: Design and open specification (Updated).

3.1.1.2. Fitting into overall COSMOS solution

With our enhancements, Secor can be used to upload use case data from the COSMOS Message Bus (based on Kafka) to the COSMOS object storage (based on Swift). Moreover the data can be stored in Parquet format and therefore be amenable to analysis using Spark SQL.

Generation of metadata allows the use of metadata search, in particular for filtering data close to the storage when analyzed by Spark SQL.

A diagram of how Data Mapping fits into the WP4 architecture can be found in Section 3 High Level Architecture of the D4.1.2 In addition, D2.3.2 [6] 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

Our extensions provide the following features:

- **Swift support** - We added support for persisting Kafka logs to OpenStack Swift. This includes support for keystone and tempAuth (and for the Softlayer object service).
- **Parquet format** - We added the ability to upload the data in Parquet format, which is a column based format with many advantages such as allowing different compression schemes for each column, reducing I/O by allowing retrieval of only the required columns, and many more. Data in Parquet format can be analysed using frameworks such as Apache Spark SQL.
- **Uploading metadata** - Available only for Swift. We added the option to add metadata to the uploaded objects. The metadata can be automatically generated when using Parquet format according to the object's content, currently it can generate minimum and maximum values for columns specified by the user.
- **Uploading each topic to different container** - Available only for Swift.

3.1.2.2. Technical specifications

The solution is an extension of the open source tool Pinterest Secor https://github.com/pinterest/secor. Secor is developed in Java, and our extensions are also written in Java.

3.1.2.3. Functionalities to be offered through Node-RED

We are currently studying node-red to determine whether we can integrate the data mapping process with Node-RED and if so, what is the best way to do this.

3.1.2.4. Relation to UC Scenarios

COSMOS use cases need to publish their data to the COSMOS message bus. Currently the Camden use case publishes its data via mosquitto to the COSMOS message bus. In addition, a Node-RED service was set up to publish Madrid traffic data to the COSMOS message bus.

In both cases we use the extended Secor to collect the data and upload it to Swift as objects with metadata in Parquet format. Currently we configured Secor to upload data every 60 minutes for both cases. (One can use a size based policy). Each data set is uploaded to a

separate Swift container. For each data set we defined a schema using Avro and specified which data fields should be uploaded as metadata.

This data can then be analysed using Spark SQL by writing SQL queries on the data. This will be used by COSMOS machine learning algorithms written in Spark. For example, in the Camden use case, clustering may be used to group apartments into groups according to their energy usage. In the case of Madrid traffic, clustering may be used to identify traffic patterns.

3.1.2.4.1. Camden use case

Example data

{"estate":"Dalehead","hid":"cOCkAsjx5jJE","ts":"1433168388","instant":"249","returnTemp":"76","flowTemp":"78.7","flowRate":"83","cumulative":"58245"}

{"estate":"Dalehead","hid":"cKQXGAvCNJNM","ts":"1433168388","instant":"19637","returnTemp":"40.2","flowTemp":"78.8","flowRate":"465","cumulative":"47148"}

{"estate":"Oxenholme","hid":"csOgqzXoh0sg","ts":"1433168389","instant":"0","returnTemp":"42.3","flowTemp":"65.9","flowRate":"0","cumulative":"33160"}

{"estate":"Gillfoot","hid":"cduVIvFPJTBk","ts":"1433168388","instant":"1907","returnTemp":"62.2","flowTemp":"73.2","flowRate":"152","cumulative":"41155"}

Example schema

```
{"namespace": "cosmos",
  "type": "record",
  "name": "CamdenLog",
  "fields": [
    {"name": "estate", "type": "string"},
    {"name": "hid",  "type": "string"},
    {"name": "ts", "type": "long" },
    {"name": "instant", "type": "int"},
    {"name": "returnTemp",  "type": "float"},
    {"name": "flowTemp", "type": "float" },
    {"name": "flowRate", "type": "int"},
    {"name": "cumulative",  "type": "int"}
  ]
}
```

Camden metadata keys

This is a list of schema fields for which metadata will be collected. This list is stored as an object in Swift. The metadata collected is the minimum and maximum values of the fields for the object.

```
ts
returnTemp
flowTemp
flowRate
cumulative
```

Camden example metadata

The below is example metadata collected for a single Swift object. This metadata is used by metadata search to efficiently find objects with data having certain values.

```
X-Object-Meta-Cumulative-Max-I → 114663
X-Object-Meta-Cumulative-Min-I → 400
X-Object-Meta-Flowrate-Max-I → 1293
X-Object-Meta-Flowrate-Min-I → 0
X-Object-Meta-Flowtemp-Max-F → 83.2
X-Object-Meta-Flowtemp-Min-F → 49.6
X-Object-Meta-Returntemp-Max-F → 82.4
X-Object-Meta-Returntemp-Min-F → 14.4
X-Object-Meta-Ts-Max-D → 2015-04-22T01:46:51
X-Object-Meta-Ts-Min-D → 2015-04-21T13:01:33
```

3.2 Delivery and usage

3.2.1. Package information

Pinterest Secor is available on github (with source code and documentation) here https://github.com/pinterest/secor

Our extensions have been developed as a git branch and include various code additions and changes.

Our extensions introduce the following dependencies to Secor:

- hadoop-openstack
- hadoop-client
- parquet-mr
- openstack swift

3.2.2. Installation instructions

Our extensions adopt the installation instructions of the original package and are documented here https://github.com/pinterest/secor.

3.2.3. User Manual

Usage instructions for Secor can be found here https://github.com/pinterest/secor

We now describe the usage instructions for our extensions. The reader needs to be familiar with Secor usage.

3.2.3.1. Configuration

Secor has many configuration files. Some of the instructions below apply to more than one file – we specify this using XXX and YYY. Below XXX, YYY can be anything, it describes the set of configuration you use.

secor.common.properties

- cloud.service - [Swift|S3] - The service you upload to.
- Swift Login Details - if you chose Swift as the cloud service, fill in this area.

secor.XXX.properties

- secor.swift.containers.for.each.topic - [true|false] - determines if each topic will be uploaded to its own container according to the topic name.
- secor.swift.container - If the previous configuration is false, determines which container the logs will be uploaded to.
- secor.info.container - Info container for ParquetFactoryReaderWriter, in this container there should be .schema and .metaKeys files. More details in the Parquet section.

secor.XXX.YYY.properties

- secor.swift.path - Prefix to the uploaded object names.

3.2.3.2. Parquet

In order to use the parquet upload option, the info container (secor.info.container) should contain object named topic.schema (replace topic with the topic name) for each topic which should contain the topic data schema in Avro format.

In the Parquet writer, we also added an option for automatically generating minimum and maximum values of user specified keys (column names) to be written as metadata. In order to use this option, the info container should contain an object named topic.metaKeys which should contain all the keys you want to be used to generate the metadata. Each line in this object should contain one key, and all keys should have type Int, Long, or Float.

3.2.3.3. Uploading Metadata

Secor will upload metadata that is stored as a file in the same directory where the object is stored. The metadata file name should be the same as the object name, with another extension - ".metadata". Secor assumes the file was saved in this way using Java code, where Swift metadata key value pairs are encoded by a Java HashMap:

```
            HashMap<String,Object> savedMetadata = new
HashMap<String,Object>();
            ...
            ObjectOutputStream oos = new
ObjectOutputStream(path);
      oos.writeObject(savedMetadata);
```

3.2.4. Licensing information

- Secor: Apache 2.0

3.2.5. Download

The source code for Secor with our extensions will be available on the COSMOS SVN, under SourceCode\M22 Prototypes\WP4\DataMapping.

We are planning to contribute our extensions to open source.

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 between them as well as with COSMOS platform services. The message bus provides solution for connection of independent components through message exchange mechanism.

4.1.1.1. Fitting into overall COSMOS solution

The message bus technology implemented in COSMOS relies on Apache Kafka [10], whose software suite includes not only the message broker but also the producer and consumer clients (also known as publishers and subscribers in other technologies). However, given that we are dealing with many different functional components and heterogeneous VEs, we are providing the means to connect seamlessly other pub/sub brokers to our solution. To do so, we have implemented two different methods: running pipelined scripts on Linux machines; and using Node-RED flows.

4.1.2. Technical description

4.1.2.1. Prototype architecture

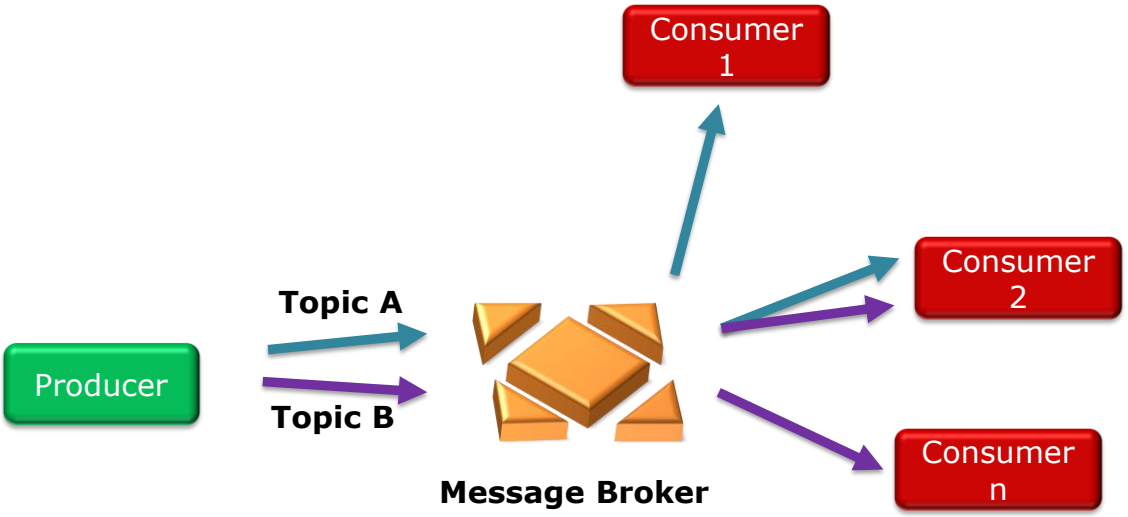


Figure 7: Message bus overview

From the high level perspective, there are three actors interacting with the message bus: producers, consumers and brokers. Producers send messages to the broker, and the broker redirects messages to subscribed consumers. Worth mentioning that in this model publishers are not programmed to send their messages to specific subscribers. Rather, published messages are characterized into channels, without knowledge of what (if any) consumers there may be. Consumers express interest in one or more channels, and only receive messages that are of interest, without knowledge of what (if any) producers there are. This decoupling of

producers and consumers can allow for greater scalability and a more dynamic network topology.

4.1.2.2. Technical specifications

Apache Kafka is a scalable and performant distributed messaging system. We described our motivation for using Kafka in COSMOS in deliverable D4.1.2 [1] including references to its performance/scalability.

4.1.2.3. Functionalities to be offered through Node-RED

Several Node-RED flows created within the COSMOS project make use directly or indirectly of the Message Bus, for instance passing messages between functional components, collecting data from VEs or even connected datasources provided by third-party services. When used directly, one just need to instantiate a Kafka node (producer or consumer). When used indirectly, one can instantiate MQTT nodes, taking into account that there will be a pipelined script bypassing certain topics into the Kafka broker.

4.2 Delivery and usage

4.2.1. Package information

The latest stable Kafka release is being used, which corresponds to version 0.8.2.1. Same applies to the correspondent producer con consumer clients. As there are some components that are using MQTT at the present time, we are also provisioning a Mosquitto MQTT broker [11], version 1.4 with support for websockets.

4.2.2. Installation instructions

Installation instructions are described here:
http://kafka.apache.org/documentation.html#quickstart

4.2.3. User Manual

Documentation of Kafka can be retrieved from http://kafka.apache.org/documentation.html.

4.2.4. Licensing information

The Kafka suite is open source software distributed under the terms of the Apache License 2.0.

4.2.5. Download

The latest stable version as well as previous releases can be downloaded from http://kafka.apache.org/downloads.html.

5 Cloud Storage – Metadata Search

5.1 Implementation

5.1.1. Functional description

Metadata search is 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 4.3.2 Metadata Search of deliverable D4.1.2 [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 D4.1.2 [1]. In addition deliverable D2.3.2 [6] discusses the COSMOS overall architecture.

The Data Mapper (described in a previous section) can upload objects to Swift with metadata to enable metadata search on those objects.

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

Our integration of Spark SQL with Swift uses an external data source driver which utilizes metadata search to push selections down to Swift.

5.1.2. Technical description

5.1.2.1. Prototype architecture

Regarding the prototype architecture, please see Section 4.3.3 Metadata Search Architecture of the D4.1.2 Information and Data Lifecycle Management: Design and open specification (Updated) document [1]. Diagrams depicting the architecture can be found there.

5.1.2.2. Components description

Regarding the components description, please see Section 4.3.3 Metadata Search Architecture of the D4.1.2 Information and Data Lifecycle Management: Design and open specification (Updated) document [1]. 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 implemented data type support and range searches which are essential 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

5.1.2.4. Relation to UC Scenarios

In Year 1 we demonstrated the use of metadata search on time series and geospatial data generated by EMT Madrid buses, in order to search for bus trip segments in a certain area and within a certain time window. This scenario is documented in detail in Section 2.2.1.2 (Storage and Analytics on Metadata) of Deliverable D7.7.1 [7].

In Year 2 we show the application of Spark machine learning algorithms on historical use case data residing in Swift, for example the Camden use case data and Madrid traffic data. These algorithms can be used, for example, to cluster use case data according to relevant parameters such as energy usage or traffic intensity. These algorithms can access data via Spark SQL and we implemented an external data source driver which can utilize metadata search in order to push SQL selections down to Swift. This reduces the amount of disk I/O as well as the network traffic. For more information see Section 7.

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

Manual instructions are below. In addition we have also developed scripts which can be used for automated installation.

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 D4.1.2 Information and Data Lifecycle Management: Design and open specification (Updated) document [1].

5.2.4. Licensing information

The metadata search code itself is currently proprietary.

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

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) [2] document Section 4.4 regarding motivation and main innovations.

6.1.1.1. Fitting into overall COSMOS solution

Data in COSMOS will be stored 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 videos 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 [2]. In addition deliverable D2.3.2 discusses the COSMOS overall architecture [6].

6.1.2. Technical description

6.1.2.1. Prototype architecture

The storlet prototype used in COSMOS is publicly available in GitHub, and is accompanied with comprehensive documentation. In the below sections we point to GitHub as appropriate

The prototype is built of the following components (See [8]).

1. A Swift cluster, made of Swift nodes. Each node is augmented with a middleware plug-in that allows invoking the storlet processing. The storlet middleware has a pluggable compute engine class which is specific to the way computations are executed. In our case this class is Docker specific. The idea is to allow to plug-in additional engines.
2. A Docker container per tenant that runs on each of the cluster nodes.
3. A 'per storlet daemon'. A JVM process that runs inside the Docker container. The daemon loads a storlet code on start-up and awaits execution requests.
4. An agent running inside the Docker 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
A Swift storlet middleware that can intercept a request for running a storlet over some data, forward the data to the compute engine and stream back the compute engine results.

The Docker Storlet Gateway
The Docker storlet gateway is a class implementing the StorletGateway API. This API defines a generic way in which computations can be invoked by the storlet middleware. This class runs in the context of the Swift storlet middleware inside the proxy and object service pipelines, and is responsible to communicate with the compute engine passing it the data to be processed and getting back the result.

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 Docker 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 https://github.com/Open-I-Beam/swift-storlets/blob/master/doc/source/writing_and_deploying_storlets.rst

SBus
Consists in 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

The Swift middleware and the docker gateway are written in python. The middleware is a standard Swift WSGI middleware. The daemon factory is written in python, the storlet daemon as well as the Storlet API library are written in Java. SBus 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

6.1.2.4. Functionalities to be offered through Node-RED

We will consider if and how storlets should be connected with Node-RED.

6.1.2.5. Relation to UC Scenarios

Storlets can be applied to various scenarios where historical data is needed. For example, when applying machine learning to historical data, storlets can be used to pre-process the data. In our integration with Spark SQL, we demonstrated using metadata search for predicate pushdown and this can be complemented in future by storlets for improved pushdown, which further reduces the amount of data sent from Swift to Spark.

6.2 Delivery and usage

6.2.1. Package information

The code is available in the following GitHub link: https://github.com/Open-I-Beam/swift-storlets. Pre-requisites for building and installing the code are:

- Openjdk-7
- Ant
- Ansible version 1.8.0 and above

6.2.2. Installation instructions

Storlets can be installed over an existing Swift cluster as described here:
https://github.com/Open-I-Beam/swift-storlets/blob/master/doc/source/storlets_installation_guide.rst

Swift can be installed using scripts available in this GitHub repo:
https://github.com/Open-I-Beam/swift-install

There is also a script that would install Swift and Storlets on an 'all in one VM'. Usage instructions can be found here:
https://github.com/Open-I-Beam/swift-storlets/blob/master/doc/source/storlet_all_in_one.rst

6.2.3. User Manual

Instructions on how to write and deploy a storlet can be found here:
https://github.com/Open-I-Beam/swift-storlets/blob/master/doc/source/writing_and_deploying_storlets.rst

Instructions on how to invoke storlets can be found here:
https://github.com/Open-I-Beam/swift-storlets/blob/master/doc/source/invoking_storlets.rst

6.2.4. Licensing information

Swift and Docker are distributed under the Apache License 2.0. Otherwise, we are using Python 2.7 standard libraries, and standard openjdk-7 libraries.

The storlets code in GitHub is under apache license 2.0.

6.2.5. Download

The code can be downloaded from GitHub here: https://github.com/Open-I-Beam/swift-storlets using 'git clone https://github.com/Open-I-Beam/swift-storlets.git'.

7 Integration with an Analytics Framework

7.1 Implementation

We focus here on the implementation work done in year 2 to support projection and predicate pushdown for Spark SQL with OpenStack Swift. This was described briefly in section 4.5.5 of deliverable 4.1.2 [1] and we provide more details here.

7.1.1. Functional description

In Apache Spark SQL, a Data Frames API was added in version 1.3.0, where a Data Frame is an RDD (Resilient Distributed Dataset) having a schema. RDDs are described in section 4.5.2 of Deliverable 4.1.2 [1]. Data Frames can be accessed like database tables using SQL style queries. The Data Frames API supports data stored in Parquet and other formats, where such data can be stored in Swift, Amazon S3, HDFS or other systems implementing the Hadoop FileSystem API. The API also supports the definition of drivers for external data sources – for example drivers have been written for MongoDB, Cloudant and others. The API enables drivers to support projection and predicate pushdown by implementing certain functions defined in the API.

Currently we store data from the Camden use case and data recording Madrid Traffic in object storage in Parquet format. There are 2 ways to access this data using Spark SQL:

Using the Parquet external data source driver

This is the default driver for Spark SQL. Since Parquet is a column based format, the driver can support projection pushdown by retrieving only the columns needed for the query. This driver also supports partitioning schemes which allow a limited form of predicate pushdown. In the case of COSMOS, data is partitioned according to date so only selections on date can be pushed down.

Using the Swift metadata search external data source driver

We implemented an external data source driver which can be applied to Swift data stored in Parquet format and annotated with certain metadata, indexed for metadata search. The metadata required are minimum and maximum values for certain user specified columns – predicates on these columns can then be pushed down to metadata search. Examples of such metadata can be found in Section 3.1.2.4.1 describing the Camden use case. The particular columns to be indexed are stored in a metadata keys object in the Object Storage which is also described in the same section. In addition, our driver can push down projections by exploiting the Parquet format of the data.

The advantage of our driver is an enhanced ability to push down predicates. For example, machine learning on the Madrid Traffic dataset needs to access and analyse data for a certain time period for example morning peak hour traffic which could be defined as 9:00-10:00. An example query using Spark SQL is shown below

```
val myresults = sqlContext.sql("SELECT codigo, intensidad, velocidad FROM madridtrafficMDS WHERE tf >= '09:00:00' AND tf <= '10:00:00'")
```

This results in a metadata search for objects with the following conditions

```
tf-max >= 9:00 AND tf-min <= 10:00
```

In COSMOS, we currently upload objects every hour, therefore the vast majority of objects (in this case typically 22 out of 24) do not overlap with this time interval. Therefore, our method significantly reduces the amount of data read from storage and transferred across the network to Spark.

7.1.1.1. Fitting into overall COSMOS solution

In COSMOS we store historical data for VEs in object storage (based on OpenStack Swift), in a format that is amenable to access by Spark SQL for the purpose of analytics on the data. Our driver allows optimized access to this data. All COSMOS services which make use of historical data can benefit – including applications which use machine learning on historical data and applications which analyse or report on past behaviour or activity of VEs.

7.1.2. Technical description

7.1.2.1. Prototype architecture

Our driver code adopts the following flow

1. Receive the predicates and projections for the given SQL query from Spark SQL
2. Build a metadata search query based on the predicates above which match the list of indexed columns
3. Submit the metadata search query and retrieve the resulting list of object names
4. Retrieve the object contents while processing them according to parquet format and applying projection to only the required columns
5. Return the data to Spark SQL

7.1.2.2. Components description

Our driver implements the Spark SQL APIs for defining an external data source. These APIs allow pushing predicates and projections down to the data source and we exploit this facility.

7.1.2.3. Technical specifications

Our code is developed in Scala and includes the following packages

1. Apache Spark SQL APIs
2. JOSS client for accessing OpenStack Swift
3. Avro
4. Parquet
5. Parsing utilities for JSON (net.liftweb.json)
6. Http client for scala (scalaj.http.Http)

7.1.2.4. Relation to UC Scenarios

We use this driver when accessing data we collected for the Madrid transportation system, where traffic data from over 3000 fixed monitoring locations is available. We analyse the historical data using k-means clustering and provide parameters for a CEP engine to infer complex events such as congestion or bad traffic in near real-time. We also provide a proactive approach for intelligent traffic management by predicting traffic parameters using regression mechanisms.

The same approach can be used for smart energy management which infers office occupancy state from electricity consumption data in the III use case, or in order to infer energy usage patterns for apartments in the Camden council.

7.2 Delivery and usage

7.2.1. Package information

Our code is developed in Scala, and we built a jar file based on it. A different jar file is built for different versions of Spark.

7.2.2. Installation instructions

1. We assume that Apache Spark has been installed and configured for use with OpenStack Swift.
2. Download the jar file sparkExternal-assembly-1.0-spark-<version>.jar (we have supplied one for spark 1.3.1 and one for 1.4.0 - choose what you need)
3. Go to the spark directory
4. Copy the jar to current dir

We have installed Spark on the WP6 machine in the testbed and have installed the relevant jar files there for use by COSMOS.

7.2.3. User Manual

1. Go to the spark directory
2. Run the following command
    - `./bin/spark-shell --jars sparkExternal-assembly-1.0-spark-<version>.jar` (replace <version> with the one you downloaded)

Usage Example

- After entering the spark shell as above:
  sqlContext.setConf("fs.swift.service.GENERICPROJECT.auth.url","http://127.0.0.1:5000/v2.0/tokens")
    - The IP address should refer to the keystone service running with Swift
- `sqlContext.setConf("fs.swift.service.GENERICPROJECT.username","your_username")`
- `sqlContext.setConf("fs.swift.service.GENERICPROJECT.password","your_password")`
- `sqlContext.setConf("fs.swift.service.GE`NERICPROJECT.http.port", "8080")`
- `sqlContext.setConf("fs.swift.service.GENERICPROJECT.tenant", "your_tenant")`
- `sqlContext.setConf("fs.swift.service.GENERICPROJECT.public", "true")`
- `sqlContext.setConf("fs.swift.service.GENERICPROJECT.region", "1")`

Example for the Camden data source

```
sqlContext.sql(
      s"""
        |CREATE TEMPORARY TABLE camdenMDS
        |USING sql.CustomPFRP
        |OPTIONS (partitions '9', container 'camden', topic 'camden', schemaContainer 'secorSchema')
      """.stripMargin)
```

Options explained:

- partitions - how many requests could go in parallel
- container - the container which contains the data
- topic - the data's topic (from secor)
- schemaContainer - the container which contains the schema (.schema object) and the keys (.metaKeys object)

Run queries:

```
val data =
  sqlContext.sql(
    s"""
      |SELECT ts, flowTemp
      |FROM camdenMDS
      |WHERE ts>=1431436226 AND ts<=1431522026
      |ORDER BY `ts`
    """.stripMargin)
```

Example for the Madrid Traffic data source

```
sqlContext.sql(
    s"""
      |CREATE TEMPORARY TABLE madridtrafficMDS
      |USING sql.CustomPFRP
      |OPTIONS (partitions '9', container 'TrafficFlowMadridPM', topic 'TrafficFlowMadridPM', schemaContainer 'secorSchema')
    """.stripMargin)
```

Run queries:

```
val myres = sqlContext.sql("SELECT codigo, intensidad, velocidad FROM madridtrafficMDS WHERE tf >= '12:00:00' AND tf <= '13:00:00' AND ts <= 1434409562770 AND ts >= 1434326679262")
```

Count results :

```
myres.count()
```

7.2.4. Licensing information

The code for our external data source driver based is currently proprietary.

7.2.5. Download

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

8 Cloud Storage – Security and Privacy

8.1 Implementation

8.1.1. 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.

Year 1 Work

There are many important security and privacy aspects related to cloud storage. We mention here two security and privacy aspects of the cloud storage components developed for COSMOS in Year 1.

1. Privacy preserving storlets
    - We implemented and demonstrated a facial blurring storlet which operates on images stored in the COSMOS object storage. It detects human faces and blurs the details so that the person cannot be identified.
    - We also implemented a mechanism whereby storlets are integrated into the access control mechanism for COSMOS object storage. This mechanism allows enabling 3 different levels of data access
        i. Access to raw data
        ii. Access to data only after filtering by a storlet (such as facial blurring)
        iii. No access
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.

Year 2 Work

In addition to security and privacy aspects of the cloud storage components developed for COSMOS in Year 1, we have also chosen to address activity monitoring and security policy management, which are important areas for IoT. This has been handled by an area known as Database Activity Monitoring (DAM), and we have extended this to object storage and currently research how this technology can be used for IoT workloads. The following describes the developed components in this context:

1. Activity monitoring for Swift – we have prototyped a middleware for OpenStack Swift that extract all data accesses via the Swift proxy layer and reports it to an audit server currently IBM InfoSphere Guardium.
2. Data access policy enforcement – we have extended the audit agent above to also support policy processing. The policy is defined by Guardium UI and deployed to the agent. Once deployed, the agent can block data access according to the policy rules.

8.1.1.1. Fitting into overall COSMOS solution

Year 1 work

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 important to allow arbitrary users to write storlet code for the COSMOS platform.

Year 2 work

OpenStack Swift is used at the COSMOS object store, therefore extending DAM to deal with Swift is relevant to COSMOS. Activity monitoring for Swift enables auditing access to object storage and enable reporting and analysis on which entities are accessing which data. Moreover, policy enforcement for Swift can allow blocking access to certain data residing on Swift according to a user defined policy. This can allow richer access control policies than are otherwise allowed by Swift which controls access at the container level, not at the object level.

8.1.2. Technical description

In this section we focus on Year 2 work.

8.1.2.1. Prototype architecture

The following figure describes the overall architecture of the prototype. Swift object storage provides its users with a REST API for issuing I/O requests. The proxy interacts with the Swift backend, comprised of the account, container and object servers, to store the data on the disks. The figure presents users accessing the Swift cluster via the Swift Proxy interface.

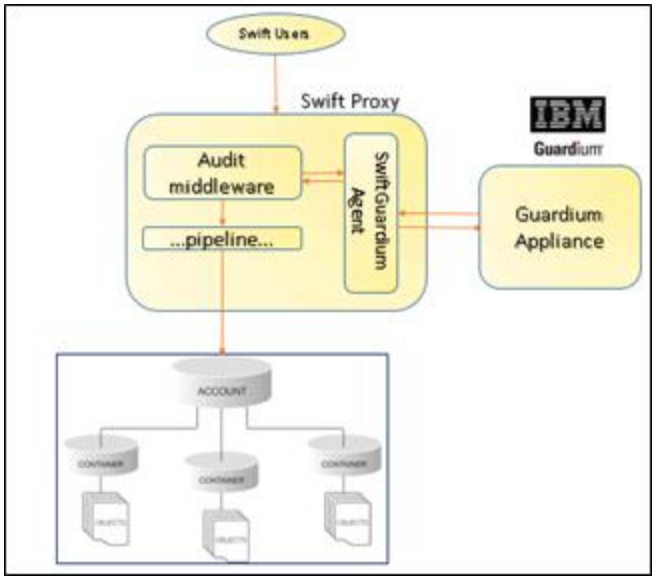


Figure 8: Overall architecture of OpenStack Swift – Guardium integration prototype

Swift is implemented using WSGI technology that allows plugging functionality into the request processor. Each request hitting a WSGI based server goes through a pipeline of such plug-ins, called middleware. The figure presents the ***Audit middleware*** that has been developed as a WSGI middleware. The middleware intercepts all I/O requests arriving at the proxy. Reporting such events requires an audit server (e.g. DAM server). In our prototype we have integrated

with an IBM product called IBM InfoSphere Guardium. The interface to the specific DAM server is done via the ***SwiftGuardium Agent*** that communicates with Guardium via a secure protocol. We also plan to explore a possible integration with open source DAM projects such as the relatively new Apache incubation called Ranger [13].

8.1.2.2. Components description

The prototype includes the following components:

8.1.2.2.1. Audit Middleware

The Swift audit middleware is implemented using WSGI technology. It is a python component that receives every request arriving at the proxy. The middleware is responsible to collect the required information related to the request. For example, the middleware extracts user information (who is issuing the request, from where), the timestamp, the target Swift container object, the target Swift object, the authentication token used for the I/O request, etc. Once the related information is collected that middleware passed the information to the SwiftGuardium agent (below) for further processing.

8.1.2.2.2. SwiftGuardium Agent (S-TAP)

The agent is the proxy to a specific audit-trail implementation (in the prototype this agent is a Guardium agent). On one hand, the agent, receives a data access event coming from the Swift audit middleware, on the other hand it interacts with a specific audit-trail service that receives events (e.g. audit events) and provides policy rules to be enforced on specific events. Specifically, each event received by this agent is checked against the set of rules previously installed. The rules may decide to send an event to the appliance for auditing, to block the request or to ignore it. Note that not all events will require an audit event, typically the security officer that defined the rules will determine which storage related events will be of any interest and will require logging or special auditing.

8.1.2.2.3. InfoSphere Guardium

IBM InfoSphere Guardium is a solution that addresses the entire database security and compliance life cycle with a unified web console, back-end data store and workflow automation system. Guardium enables to find and classify sensitive data in corporate databases (for which the database owner provided access for mining), assess database vulnerabilities and configuration flaws, capture and examine all database transactions, including local access by privileged users, monitor and enforce policies for specific data access patterns (such as sensitive data, privileged user actions, etc.) and to centralize the compliance auditing process for enterprise compliance reporting, performance optimization, investigations and forensics.

Guardium auditing is provided by software S-TAP agents that are installed on the DB servers and send a copy of the observed traffic to a Guardium collector (also called a Guardium Appliance). Guardium appliance may install a policy that is built via Guardium web interface to the S-TAP. By doing that, the S-TAP can act as a firewall (an authorization mechanism) that can block specific database requests as defined in the installed policy. The local S-TAP can be configured to operate in a disconnected mode in case of intermittent network disconnections to the appliance.

The SwiftGuardium Agent (See Section 8.1.2.2.2) is such an S-TAP developed specifically for OpenStack Swift during Year 2.

8.1.2.3. Technical specifications

8.1.2.3.1. Open Stack Swift Middlewares

The following section describes OpenStack WSGI interface used to develop OpenStack Swift Audit middleware. The full python WSGI implementation can be found in: https://www.python.org/dev/peps/pep-0333/.

The WSGI interface has two sides: the "server" or "gateway" side, and the "application" or "framework" side. The server side invokes a callable object that is provided by the application side. Like many other OpenStack projects, Swift uses paste to build his HTTP architecture. Paste uses WSGI and provides an architecture based on a pipeline. The pipeline is composed of a succession of middleware, ending with one application. Each middleware has the chance to look at the request or at the response, can modify it, and then pass it to the following middleware. The latest component of the pipeline is the real application, and in this case, the Swift proxy server. WSGI middleware should consist of a callable object. Usually this is done with a class implementing the __call__ method as demonstrated in the following figure.

```
class Guardium(object):
    """
    Guardium middleware agent for Swift.

    A mini web UI is provided at guardium_web_path for a simple audit-trail interface.
    """

    def __init__(self, app, conf):
        self.app = app
        self.conf = conf

    def __call__(self, env, start_response):
        return self.app(env, start_response)

def filter_factory(global_conf, **local_conf):
    conf = global_conf.copy()
    conf.update(local_conf)

    def guardium_filter(app):
        return Guardium(app, conf)
    return guardium_filter
```

Figure 9: WSGI middleware implementing the __call__ method

This middleware will just do nothing as it is. It's going to simply pass all requests it receives to the final application, and returns the result.

A specific middleware implementation will simply need to override the call method to provide its specific functionality to the pipeline.

8.1.2.3.2. InfoSphere Guardium S-TAP

The following section describes Guardium interface to its S-TAP agents. The SwiftGuardium agent interact with Guardium based on this specification.

With InfoSphere Guardium, event data is sent to a secure appliance, known as a collector, and stored in an internal database there for reporting and alerting. With the Universal Feed capability included with InfoSphere Guardium, one can integrate related audit data (perhaps for databases or other activity not supported by Guardium) into the current Data Activity Monitoring (DAM) environment by sending it to the collector and storing it in the internal tables there.

The Universal Feed has the following options for supporting different types of activity monitoring. The first one is targeted for activity that can easily integrate into the existing

internal InfoSphere Guardium tables. This would typically mean some kind of database source, since InfoSphere Guardium specializes in support for database activity. The other option enables to integrate any arbitrary data source activity by enabling the creation of custom table structure in the Guardium database to store the messages sent over by the agent.

Guardium provides the following benefits when using the Universal Feed to store the audit data off of the actual device that is monitored:

1. Audit and log information cannot be erased to cover nefarious breaches to the device.
2. Separation of duties can be maintained to ensure correct audit information is captured.
3. Privileged users don't have access to the audit logs if they decided to tamper or alter this information.

The agent must use the Guardium message format to send and receive information to be processed into the Guardium internal tables. The message format shown in Listing 1 is for a database-type Universal Feed agent as indicated by the "0" in the vendor field.

```
struct sqlguard_msg {
unsigned char msg_type;     // Must be 'G'
    unsigned char pad;      // Must be 0
    unsigned short data_len;// Length of  data in the "data" fields, network order
    uint32_t mark;                //
    uint32_t timestamp;     // Time in UNIX format (retval of time() syscall)
    uint32_t protocol_version;  // Must be 7
    uint32_t vendor;        // Must be 0
    char identification[40];    // Must be all 0
    char data[MAX_DATA_LEN];    // Put the serialized protobuf message here
};
```

Figure 10: Guardium message format for a database source. Taken from [12]

There are two directions flowing for these messages: From the agent to the collector and From the collector to the agent. The agent must adhere to the Guardium communication protocol based on TCP, and it must send a Guardium handshake message that does the following:

1. This handshake message allows for the collector to register the name of the UFA.
2. Turns the agent green in the GUI so that you know it is operational.
3. It must send a Guardium Ping message every 30 to 60 seconds.
4. It must read everything the Guardium appliance sends to the agent.

The following figure shows a diagram of the message flow:

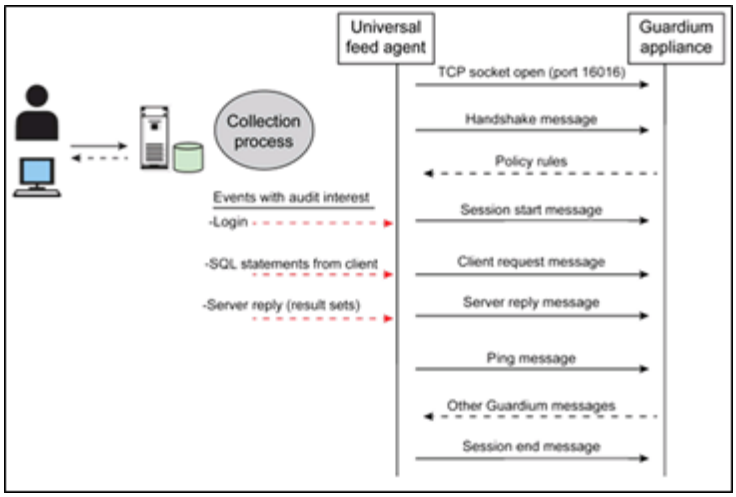


Figure 11: Guardium Universal feed message processing flow. Taken from [12]

8.1.2.4. Relation to UC Scenarios

The activity monitoring and auditing can be used for IoT use cases in general and to COSMOS use cases in particular. For example, COSMOS scenarios allows VEs to share historical data with other VEs for the purposes of experience sharing etc. The audit trail of information sharing patterns could be useful in order learn about the reputation and trust of particular VEs. For example, certain VEs may be information 'parasites' which only request information from other VEs and do not share their own information. Another example is if there is suspicion that data generated by a VE has been leaked, one can examine the audit trail to study how this might have happened and which other VEs consumed the data. The policy management and enforcement aspects of this integration are also relevant to the use cases since they enable the definition of policy rules which can be used to enforce which users can access which data.

8.2 Delivery and usage

The prototype is comprised of two components that will be provided as described below. The audit server (e.g. InfoSphere Guardium) must be pre-installed and configured which is not part of this document.

8.2.1. Audit Middleware

The Audit middleware will be provided as a simple python file. The file should be installed to Swift middleware directory.

8.2.2. SwiftGuardium Agent (S-TAP)

The Guardium agent will be provided as an executable Java JAR file. The JAR will include the libraries that implement Guardium universal feed and the interface to the python audit middleware. For the prototype this interface uses the Apache Thrift software framework [9].

8.2.3. Package information

The two packages will contain Swift middleware and an executable JAR:

- middleware.zip: contains guardium.py file
- guardium-swift-feed.jat: contains the guardium S-TAP agent with its dependencies

8.2.4. Installation instructions

In this section we assume that OpenStack Swift has been installed and is properly configured. In addition, IBM InfoSphere Guardium that is used to collect audit events has been installed at some designated location.

Installation requires the following steps:

8.2.4.1. Install Swift Audit Middleware

Step 1: Install the guardium audit middleware to OpenStack Swift. At this stage we simply copy the middleware to its respective directory. In the near future we will provide the audit middleware via a Python Egg.

Step 2: Update the swift proxy configuration file. First, we need to add the guardium middleware to the pipeline. Second, we need to configure the middleware with the address of the guardium S-TAP agent (noted as *guardium_server*). The *guardium_web_path* property can

be used for debug purposes to observe the events reported to the agent by the middleware via a tiny web interface started on the URL: http://*guardium_server/guardium_web_path* . The following figure demonstrated the steps explained in this section.

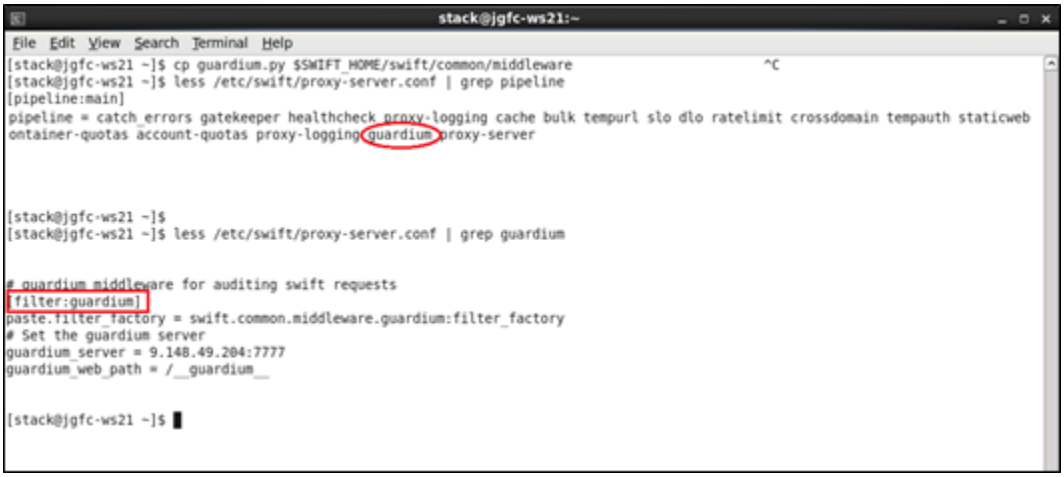


Figure 12: Installing Swift audit middleware

8.2.4.2. Install SwiftGuardium Agent (S-TAP)

No special installation is required for the SwiftGuardium agent. The agent must be started once the Swift proxy machine has been authenticated to the Guardium server. This is done during a typical deployment of Guardium.

To start the agent one should simply invoke:

```
java -jar guardium-swift-
```

And the expected output is presented below.

Figure 13: Expected output after starting the SwiftGuardium Agent

8.2.4.3. Test Installation

Once everything is installed we expect to be able to issue Swift command and observer an audit trail at the Guardium UI. It is beyond the scope of this document to provide an operational manual for Guardium so we provide snapshots of the audit reports.

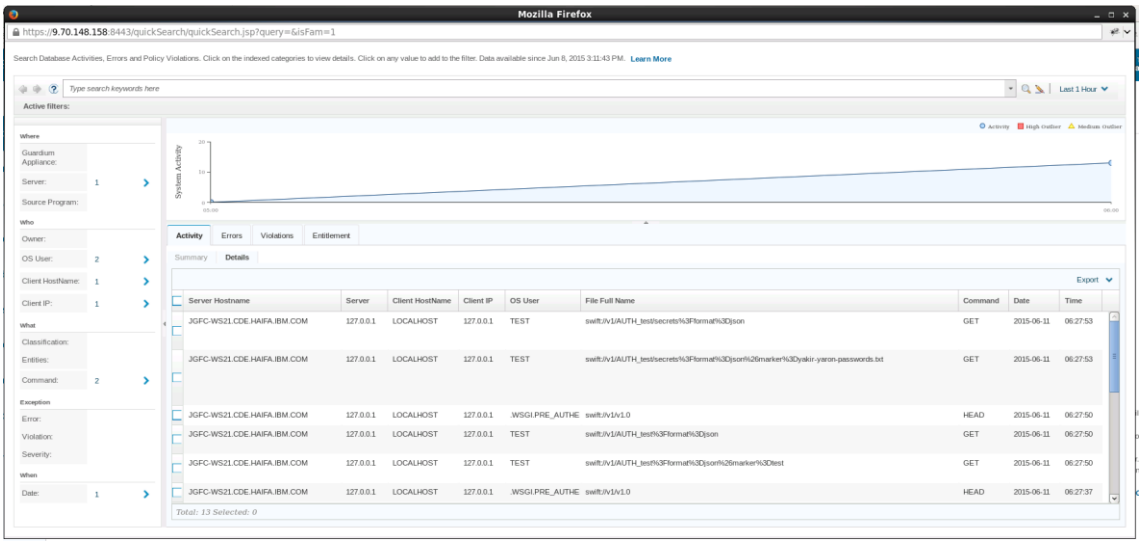


Figure 14: Snapshot of Guardium audit reports for OpenStack Swift

Each event corresponds to a user accessing some object in Swift. The figure presents the access time, the user and the command all are needed to provide an audit trail required by regulations.

Server Hostname	Server	Client HostName	Client IP	OS User	File Full Name	Command	Date	Time
JGFC-WS21.CDE.HAIFA.IBM.COM	127.0.0.1	LOCALHOST	127.0.0.1	TEST	swift://v1/AUTH_test /secrets%3Fformat%3Djs(	GET	2015-06-11	06:27:53
JGFC-WS21.CDE.HAIFA.IBM.COM	127.0.0.1	LOCALHOST	127.0.0.1	TEST	swift://v1/AUTH_test /secrets%3Fformat%3Djs(yaron-passwords.txt	GET	2015-06-11	06:27:53

Figure 15: Information provided by Guardium audit reports for OpenStack Swift

As described in the previous section, the audit middleware provides a tiny web interface that can be used to present the events reported by the middleware. The following figure presents the interface.

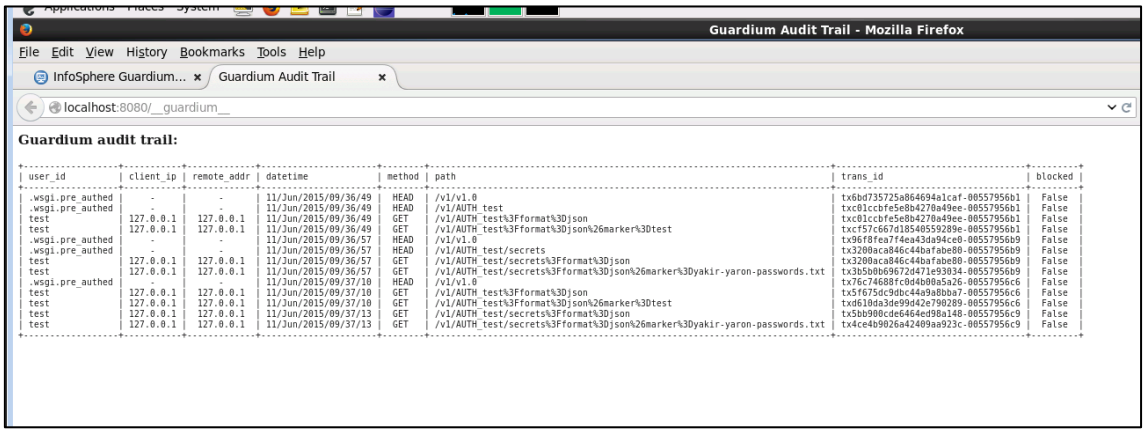


Figure 16: Web interface which presents events reported by OpenStack Swift middleware

8.2.5. User Manual

Once the system is properly installed, one can use the Guardium interface to define policies and generate audit reports. Please refer to the manual of InfoSphere Guardium for details (beyond the scope of this document).

8.2.6. 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 library licenses appear in the technical specification section above.

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

8.2.7. Download

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

9 Conclusions

This document describes the prototypes for the Information and Data Lifecycle Management Work Package. Each component has been implemented independently, and most components have also been integrated within WP4.

We have demonstrated how the information and data lifecycle management for IoT data can be implemented using our prototypes. Data is ingested from devices using a scalable message bus such as Apache Kafka. A data mapper aggregates messages from the message bus for storage in object storage. This data is analyzed using the Apache Spark analytics framework and access is optimized using metadata search and potentially storlets as well in future. Machine learning is applied within the Spark analytics framework to the historical data stored in object storage and generates thresholds or other information to be used by CEP. CEP receives events in real time from the message bus and can respond to them intelligently using the insights provided by Machine Learning. Using this paradigm, COSMOS supports intelligent response to real time events in smart cities by learning the behavior of IoT virtual entities over time. Moreover in the context of COSMOS, we also demonstrated aspects of security and privacy preservation for object storage in this work package, and this work forms part of the bigger security and privacy picture for COSMOS as a whole.

This work will form the basis of our COSMOS Year 2 integrations and demonstrations. This is the updated prototype for our work in COSMOS, which will be revised in Year 3 of the project.

10 References

1 COSMOS Deliverable 4.1.2 - Information and Data Lifecycle Management: Design and open specification (Updated).

2 COSMOS Deliverable 4.1.1 - Information and Data Lifecycle Management: Design and open specification (Initial).

3 Node-Red: A visual tool for wiring the Internet of Things: http://nodered.org/

4 COSMOS Deliverable 5.1.2 - Decentralized and Autonomous Things Management: Design and Open Specification (Updated).

5 Process Manager 2 – PM2: https://github.com/Unitech/pm2

6 COSMOS Deliverable 2.3.2 - Conceptual Model and Reference Architecture (Updated).

7 COSMOS Deliverable 7.7.1 - Integration of Results (Year 1).

8 https://github.com/Open-I-Beam/swift-storlets/blob/master/doc/source/docker_compute_engine.rst

9 Apache Thrift software framework: https://thrift.apache.org/

10 Apache Kafka - A high-throughput distributed messaging system: http://kafka.apache.org/

11 Mosquitto - An Open Source MQTT v3.1/v3.1.1 Broker: http://mosquitto.org/

12 Use InfoSphere Guardium Universal Feed to create a customized data activity monitoring solution, Part 1: Create a feed for a database source http://www.ibm.com/developerworks/data/library/techarticle/dm-1210universalfeed/

13 Apache Ranger - http://ranger.incubator.apache.org/