GRANT AGREEMENT No 609035
FP7-SMARTCITIES-2013

Real-Time IoT Stream Processing and Large-scale Data Analytics for Smart City Applications

Collaborative Project

Smart City Demonstrator

Document Ref.	D6.2
Document Type	Prototype
Workpackage	WP6
Lead Contractor	AI
Author(s)	João Fernandes (AI), Dan Puiu (SIE), Thorben Iggena, Daniel Kuemper, Marten Fischer (UASO), Sefki Kolozali, Nazli Farajidavar, Daniel Puschmann (UniS), Feng Gao, Thu-Le Pham (NUIG), Azadeh Bararsani, Aneta Vulgarakis (ERIC)
Contributing Partners	AI, UASO, UniS, ERIC, SIE, NUIG
Planned Delivery Date	M33
Actual Delivery Date	M36 Update
Dissemination Level	Public
Status	Final
Version	V1.2
Reviewed by	Stefan Bischof (SAGO) and Payam Barnaghi (UniS)

Executive Summary

The CityPulse project provides a framework for large-scale data analytics that extract patterns/events in real-time, transforming raw data into actionable information. CityPulse enables the creation of different smart city applications that use the software components and methods that are included in the CityPulse framework. The CityPulse framework components can be divided into three main categories: large scale data stream processing; reliable information processing; and adaptive decision support modules. The components are generic and reusable in different application domains and are provided as open-source via the CityPulse GitHub repository (https://github.com/CityPulse).

This document reports on the Smart City Demonstrators that have been set-up and developed in the scope of Activity 6.2. This includes the integration of all the CityPulse components and tools into a functional system that supports the easy and comprehensive development of smart city applications. The report also describes the integration of different datasets in which the sources of information are fed into the framework. The development of different application prototypes that demonstrate the usefulness and added value of the CityPulse framework are also discussed in the report.

This report will also serve as documentation for developers, providing information on how to use the different software components developed in CityPulse. The document describes how to set-up the components and reuse or implement new innovative services and applications that rely on the CityPulse framework.

Table of contents

List of Figures

List of Tables

Abbreviations List

APIs	Application Programmable Interfaces
WP	Work Package
GDI	Geospatial Data Infrastructure
OSM	Open Street Map
KAT	Knowledge Acquisition Toolkit

CSV	Comma Separated Value
PIR	Passive Infrared Sensor
PIP	Pip Installs Packages
JSON	JavaScript Object Notation
HTTP	Hypertext Transfer Protocol
URI	Uniform Resource Identifier
CPU	Central Processing Unit
ODAA	Open Data Aarhus
IP	Internet Protocol
SAO	Stream Annotation Ontology
ED	Event Detection
ACEIS	Automatic Complex Event Implementation System
RDF	Resource Description Framework
RSP	RDF Stream Processing
CQELS	Continuous Query Evaluation over Linked Streams
SPARQL	Simple Protocol and RDF Query Language
JDK	Java Development Kit
QoI	Quality of Information
REST	Representational State Transfer
RPC	Remote Procedure Call
LSM	Linked Sensor Middleware
CKAN	Comprehensive Knowledge Archive Network
KML	Keyhole Markup Language
UUID	Universally Unique Identifier
GUI	Graphical User Interface
GPS	Global Positioning System
IDE	Integrated Development Environment
URL	Uniform Resource Locator
NLP	Natural Language Processing
PoI	Point of Interest

1. Introduction

The CityPulse project provides a distributed framework for semantic discovery, processing and interpretation of large-scale real-time Internet of Things and relevant social data streams for knowledge extraction in a city environment.

The platform is envisioned to serve and be explored by the different city stakeholders. Their involvement, understanding and use of the platform is very important as it reflects the usefulness and added value of the framework.

One of the objectives of WP6 in the project is to select and develop a set of representative smart city scenarios that cover the functionalities of the innovations developed in the scope of the other technical WPs and that are of interest to cities and their stakeholders. Moreover, and in the scope of Activity 6.2, the objectives are to perform continuous and iterative integration of individual innovations into a system that fulfills the functional requirements for the selected scenarios.

This report provides guidelines to design the functionalities and interfaces of the applications based on the CityPulse components. The development of different smart city applications will serve as demonstration and verification of the added value of the developed applications.

In the scope of this activity the following applications have been developed:

- **Travel Planner** – an Android application for citizens, that can be used for obtaining user centric travel and parking recommendations;
- **City Dashboard** –supports visual analytics for the different relevant datasets registered in the platform;
- **3D Map** –provides a 3D visualisation of cities and different geo-located events provided by the platform;
- **Twitter Data/Event Map** – the users can visualise events extracted from social media streams (Twitter);
- **Tourism Scheduler** – provides citizens relevant information of tourism-related points of interest and optimal routes according to the user's location, transportation information and users' preferences;
- **Pick-up Planner** – provides optimal planning and routing for a fleet management company;
- **Transportation Planner** – provides optimal bus transportation schedule based on user demands and real-time traffic in the city;

This document provides a detailed description of the Smart City Demonstrator that serves as documentation for developers that want to use the CityPulse to develop new innovative applications or use/update the existing ones. It includes an overall view of the CityPulse components and the provided APIs to be used by application developers. It also includes a description of all the CityPulse components and applications including system requirements, dependencies, installation and running details and link to the code which has been extracted from the GitHub repository. Finally, additional tools and datasets that are used by the developed applications are also described.

2. Smart City Demonstrator

The smart city demonstrator provides an integration of components and data sources and describes the development of a number of applications using the CityPulse framework. This does not only demonstrate the usefulness and added value of the CityPulse framework, but also describes how start-ups and developers to develop further applications and services in an easy and seamless way.
The main components of the Smart City Demonstrator are shown in Figure 1 (taken as basis from [CityPulse-D5.3]):

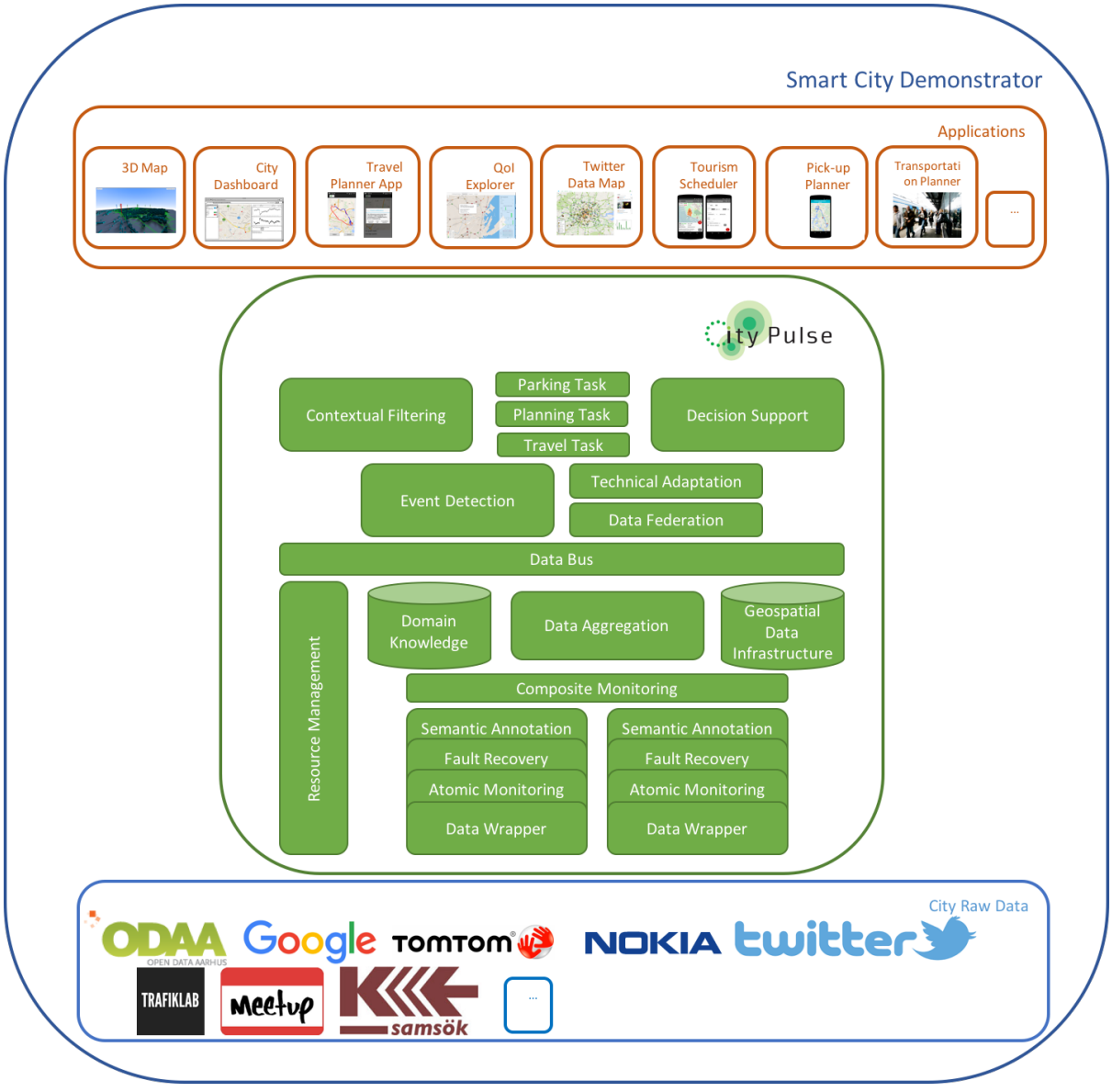


Figure 1: CityPulse Smart City Demonstrator

The CityPulse framework is composed by a number of different components as shown in the above figure. These are integrated into a common system and provide different functionalities that allow application and service developers to obtain necessary information for their solutions. Overall the components of the CityPulse framework are the following:

- **Data Wrappers and Semantic Annotation** – offers an easy and generic way to describe the characteristics of a set of sensors using sensory meta-data. A data wrapper

connects to the sensor, parsing the incoming data extracting the relevant information from the resource, as well as providing access to historical data. After the parsing phase the data is semantically enriched (Stream Annotation Ontology and Quality Ontology) and published to the message bus. Detailed information regarding this component is reported in [CityPulse-D2.2], [CityPulse-D3.1] and [Puiu et al. 2016].

- **Resource Management** – manages Data Wrappers (resources). The Resource Management is used to deploy all Data Wrapper units placed in a predefined folder, it also distributes Data Wrapper's outputs to other framework components. The RM is responsible for publishing semantically annotated observations and aggregated data coming from different parts of the framework – Data Federation and Event Detection modules, or even directly from the applications. [CityPulse-D2.2], [CityPulse-D3.1] and [Puiu et al. 2016] describe in detail the Resource Management component.
- **Data Aggregation** – that deals with large volumes of data using time series analysis and data compression techniques reducing the size of the raw sensory data observations that are delivered by the Data Wrappers. Further information regarding the Data Aggregation component is reported in [CityPulse-D2.2], [CityPulse-D3.2] and [Puiu et al. 2016]
- **Data Federation** – that answers users' queries, e.g. what is the average vehicle speed on my current route to the destination over the past 5 minutes. The Data Federation receives its inputs from the application interface, querying the service metadata stored in the Domain Knowledge to perform stream discovery and composition. It then subscribes to the Data Bus to consume real-time data and its outputs can be delivered to the application interface or the Decision Support. Further information regarding the Data Federation component is reported in [CityPulse-D2.2], [CityPulse-D3.2] and [Puiu et al. 2016]
- **Event Detection** – provides generic tools for processing annotated and aggregated data streams to obtain events occurring in the city. The ED is flexible allowing the user to deploy new event detection mechanisms in an easy way. Its process/execution follows two main steps: real-time data acquisition, interpretation and validation; and event detection mechanism in order to detect the patterns of events. The reports [CityPulse-D2.2], [CityPulse-D3.3] and [Puiu et al. 2016] include detailed information regarding the Event Detection component.
- **Quality Monitoring** – providing a two-layered quality calculation mechanism to annotate data streams with a Quality of Information (QoI) metric. The lower layer – Atomic Monitoring - is a stream based quality calculation, whereas the upper layer – Composite Monitoring – combines different data streams to include several sensor observations into QoI calculation. Detailed information related to the Quality Monitoring component is reported in [CityPulse-D2.2], [CityPulse-D4.1] and [Puiu et al. 2016].
- **Fault Recovery** – ensures continuous and adequate operation of the CityPulse application by generating estimated values for the data streams in case the quality drops or in case of temporary missing information. This component is described in detail in the documents [CityPulse-D2.2], [CityPulse-D4.2] and [Puiu et al. 2016].
- **Geo-spatial Database** – integrated to utilize infrastructure knowledge of the city such as buildings, traffic flows, road networks, ongoing construction work, traffic density, etc. The GDI enables the calculation of different distance measures and allows enhanced information interpolation to increase reliability. Reports [CityPulse-D2.2], [CityPulse-D4.2] and [Puiu et al. 2016] describe in detail the GDI component.
- **Contextual Filtering** – that continuously identifies and filters events that might affect the optimal result of the decision making task performed by the Decision Support component. The CF component reacts to real-time changes by requesting the Decision Support for the computation of a new solution when needed. The reports [CityPulse-D2.2], [CityPulse-D5.1] and [Puiu et al. 2016] describe in detail the Contextual Filtering component.
- **Decision Support** – enables reactive decision support functionalities to be easily deployed, providing the most suitable answers through its reasoning capabilities at the right time. [CityPulse-D2.2], [CityPulse-D5.2] and [Puiu et al. 2016] report in detail the Decision Support component and its functionalities.

- **Technical Adaptation** – that automatically detects critical quality updates for the federated data streams used in the Data Federation component and makes adjustments. The Technical Adaptation component is described in detail in the reports [CityPulse-D2.2], [CityPulse-D4.1] and [Puiu et al. 2016].

In order to ease the reuse of the CityPulse framework components and to provide online documentation these have been released on the CityPulse GitHub repository: https://github.com/CityPulse (Figure 2). The goal is to engage the open source communities to use the CityPulse framework components. This will also allow the continuous development of new functionalities or the update of existing ones.

The release of the components to a broader community of developers involves more developers and users in the process. The software can be customized to users' needs and expectations, as they are the contributors to it. The customization of software reduces the time for development and brings solutions closer to what the user is expecting. The software components are also provided with the technical documentations and code descriptions in the GitHub repository to ease the use and re-use of the components.

In addition to the code, all the software repositories include a README.md with descriptions of the components, how to install them, and run them. This documentation also includes some tutorials and videos that explain thoroughly how the user can make use of each piece of software.
In the following section we present an extraction of each of the repositories' documentation.

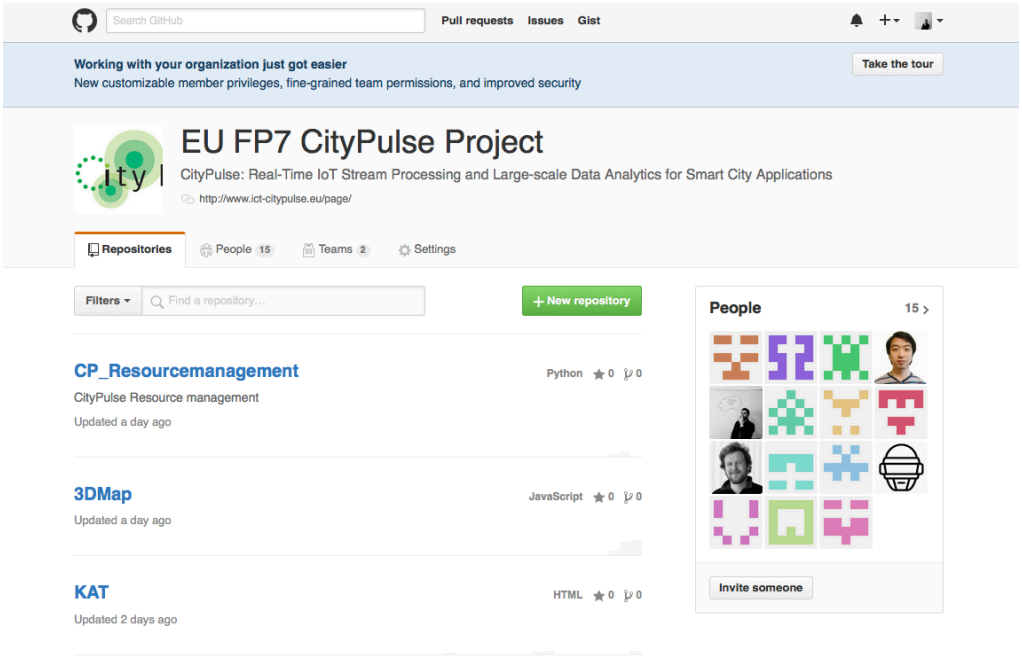


Figure 2: CityPulse GitHub account

2.1. Framework Components

In this section a describe each of the CityPulse framework components which has been extracted from each of the components' repositories on GitHub. This documentation includes a description of the component, a list of requirements and dependencies to other pieces of software, as well as installation and running guides.

2.1.1 GDI – Geospatial Data Infrastructure

The GDI component is used by a number of other CityPulse components to tackle geo-spatial tasks. For example, the Composite Monitoring uses the GDI to find nearby sensors or the Decision Support uses the GDI to get a set of routes across the city, which follow non-functional user requirements. To overcome the limitations of semantic data storage and processing, the GDI is used to support geospatial analysis and implement spatiotemporal reasoning processes. The GDI is used to analyse dependencies between infrastructure, sensors and events. This goal is achieved by utilising an optimised search index for geospatial objects. Since most city sites and elements (e.g., streets or places) in cities cannot just be described as a coordinate (e.g. longitude, latitude in the WGS84 system of GPS) it is necessary to support more complex geometric types like Linestrings and Polygons to represent roads, public places, polluted areas, etc. Processing these kinds of complex geometries needs a qualified geospatial indexing process to find nearby objects, process overlapping areas and generate adapted routing graphs.

2.1.1.1. Requirements and Dependencies

The GDI uses OpenStreetMap Data (OSM) and is implemented in R and Java. Therefore it depends on the following components:

- For Websocket Routing Interface: R >= V3.2.3 (required libraries: websockets, RPostgreSQL,rgeos,DBI,jsonlite)
- For Java GDI Implementation: Java 8
- PostgreSQL version >= V9.3
- OpenStreetMap data
- osm2pgsql >= V0.82.0
- osm2pgrouting >= V2.1.0

2.1.1.2. Installation

Before using the GDI, a PostgreSQL instance must be set-up. The following lists the necessary steps:

- Install PostgreSQL (http://www.postgresql.org/)
- Install PostGIS extension for Postgresql (http://postgis.net/)
- Install pgRouting (http://pgrouting.org/)
- Download Openstreetmap(OSM) dataset from required (e.g. from http://download.geofabrik.de/)
- Create and import database
- At shell level:

```
user@server:~$: createdb cp_sweden
user@server:~$: psql cp_sweden
```

- At Database level:

```
cp_sweden=# CREATE EXTENSION postgis;

cp_sweden=# CREATE EXTENSION hstore;

cp_sweden=# CREATE EXTENSION pgrouting;

cp_sweden=# \q
```

- Optional (at shell level) cut down area smaller area by bounding box:

```
osmosis   --read-xml sweden-latest.osm --tee 1 --bounding-box left=17.2410
top=60.2997 bottom=58.5328 right=20.0253 --write-xml stockholm.osm
```

- At shell level (example Import of OSM Layer and Routing process for Sweden's data):

```
user@server:~$: osm2pgsql -C2000 -d cp_sweden -k -l --slim --flat-nodes flat-
nodes.bin --number-processes 8  sweden-latest.osm.pbf

user@server:~$: osm2pgrouting --file ./sweden-latest.osm --conf
/usr/share/osm2pgrouting/mapconfig.xml --dbname cp_sweden -p5432  --user XXX --
passwd XXX
```

- At shell level create some CityPulse specific tables and indexes (sql file in the repository)

```
user@server:~$: psql cp_sweden -f CityPulseGDI_Public/res/Initialise-CP-
specials.sql
```

2.1.1.3. Running the component

After the installation of the GDI, - the websocket routing interface can be started with:

```
user@server:~$: R -f websocketRoutingInterface.R
```

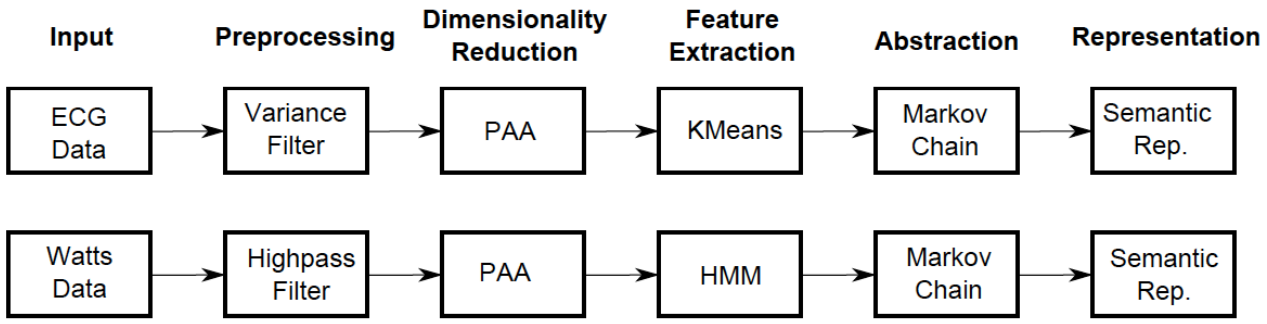
Websocket usage parameters are defined in the README file. - The GDI Java implementation can be found at https://github.com/CityPulse/GDI/tree/master/CityPulseGDI_Public - It provides a full Javadoc Documentation - Example class for first experiments: eu.citypulse.uaso.gdiclient.CpClientExampleStockholm

2.1.1.4. Link

The code of the GDI can be found here: https://github.com/CityPulse/GDI.git

2.1.2 KAT (Knowledge Acquisition Toolkit)

KAT follows a simple processing approach. We use a general workflow that has been extracted by observing several different solutions for information abstraction. The existing solutions either follow the steps shown in the figure below or implement some parts of it. We identified the following main steps: Pre-processing to bring the data into shape for further processing, Dimensionality Reduction to either compress the data or reduce its feature vectors, Feature Extraction to find low-level Abstractions in local sensor data, Abstraction from raw data to higher-level Abstractions and finally semantic representations to make the abstracted data available for the end-user and/or machines that interpret the data.



(Source: Frieder Ganz, Daniel Puschmann, Payam Barnaghi, Francois Carrez, "A Practical Evaluation of Information Processing and Abstraction Techniques for the Internet of Things", IEEE Internet of Things Journal, 2015.)

Figure 3: KAT Processing Flow

2.1.2.1. Pre-Processing

The raw sensory data is pre-processed stage to prepare the data for information acquisition. Pre-processing can be done on sensor nodes to reduce transmission cost or filter unwanted data and can include mathematical/statistical methods to smooth the data by applying moving average windows, or methods from signal processing such as band-, low-, high pass filtering to focus on a certain frequency spectrum.

Transmission cost can be reduced by only sending aggregated information or a selection of data to a base-station or a gateway; e.g. sending minimum and/or maximum values or the mean value of the current window. The pre-processing is not only limited to a single sensor node; some solutions use in-network processing to aggregate the data before further processing it by finding the minimum, mean or maximum value in a set of sensor nodes and before transmitting the data to a base station. In addition to data aggregation, in-network techniques can also be used to improve the accuracy of the data by calculating correlation with data from neighbouring nodes.

2.1.2.2. Dimensionality Reduction

To handle the large amount of data that has to be processed and stored. Dimensionality reduction can decrease the size and length of samples by applying different methods on the data while preserving the important features and patterns. This section presents a simple example based on a dataset gathered in an office environment consisting of several hundred thousand sensor readings. The sensor was deployed near an office desk and measured light level, PIR, sound, and energy consumption of an office workstation. The dataset can be downloaded here: http://kat.ee.surrey.ac.uk/download.html

2.1.2.3. Data Import

The first step is the data import. KAT currently supports CSV and MS Excel formats. To import data click on the Load Data button and select a comma separated file or an MS Excel file. You can download the sample dataset (http://kat.ee.surrey.ac.uk/download.html), save it as a csv file and import it to KAT.

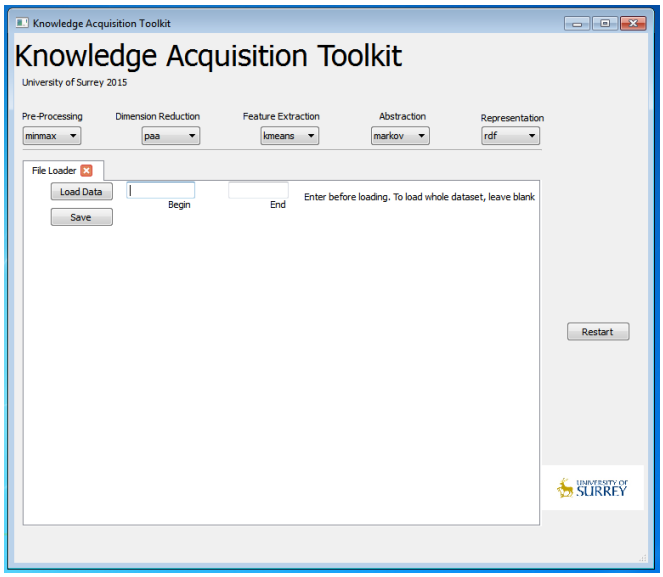


Figure 4:KAT - Import Data

Once the data is imported, the labels will appear on the screen. Select "light" check box from the data labels to show the data in a diagram (as shown below). In case that the data is not shown, select check box and left click on empty diagram.

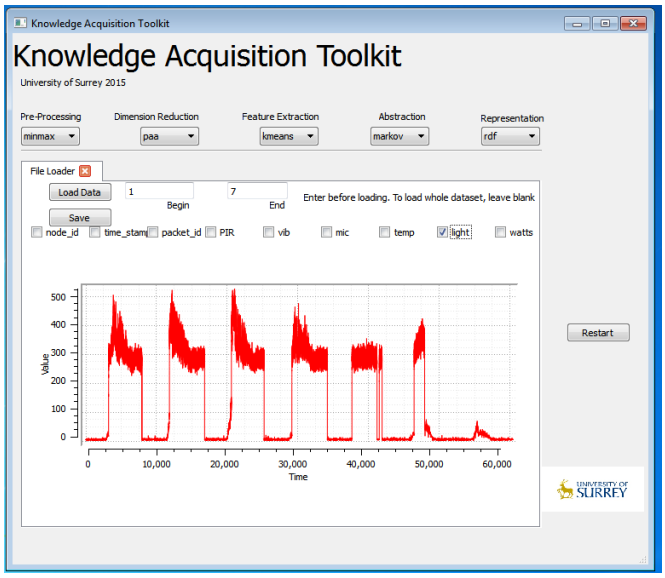


Figure 5: KAT - Data displayed in a graph

2.1.2.4. Data Pre-Processing

To highlight features of the data set, a mean filter is used in this example. The mean filter divides the data into windows and replaces the window with the mean value. This filter can highlight outliers and reduces the noise.

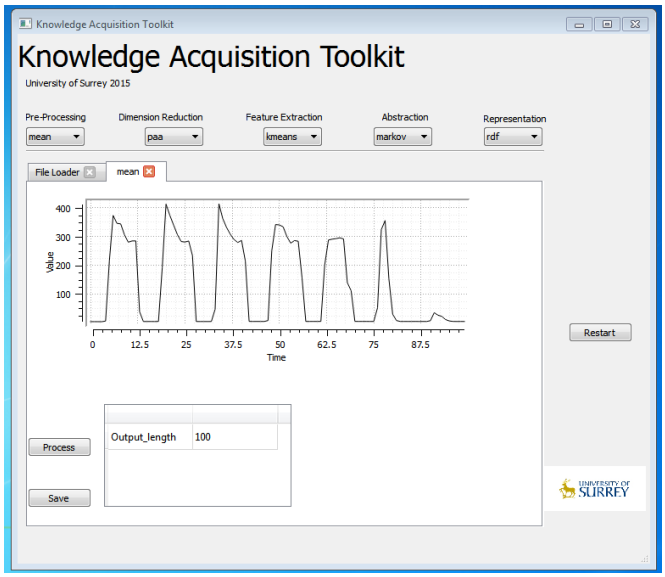


Figure 6: KAT - Data Processing

2.1.2.5. Dimension Reduction

To create a symbolic representation of the data, we use a technique called SAX.

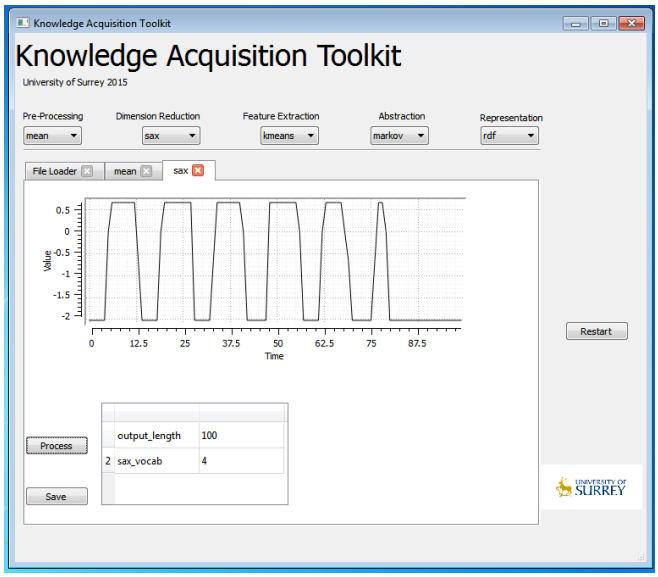
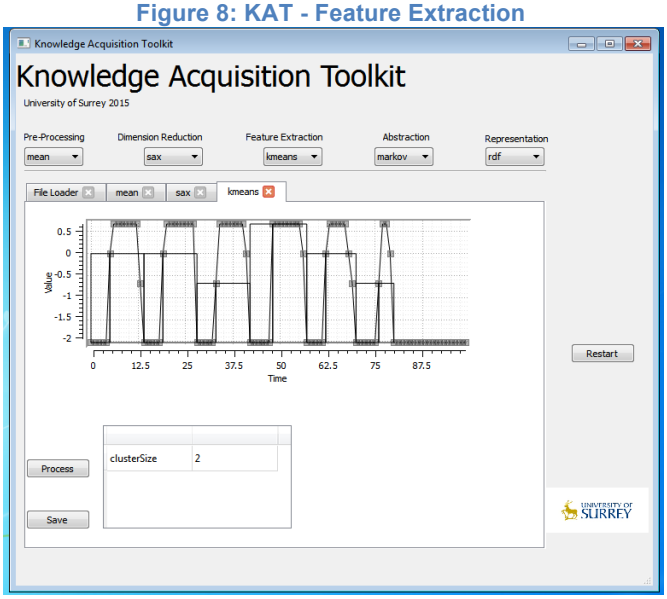


Figure 7: KAT - Create a Symbolic Representation

2.1.2.6. Feature Extraction

To find interesting patterns that are likely to represent an event, phenomena or interesting observations a k-means clustering algorithm is used. In this example, the k-means algorithm is run with a group size of three. The algorithm clusters periods of low activity (low power), medium activity, and peaks (high power usage) into the groups labelling them from 0 to 2.

Figure 8: KAT - Feature Extraction



2.1.2.7. Abstraction

To find relationships between different groups produced in the previous section, a Markov based statistical model is applied to the data. The model returns the likelihood of the temporal presence of the different groups.

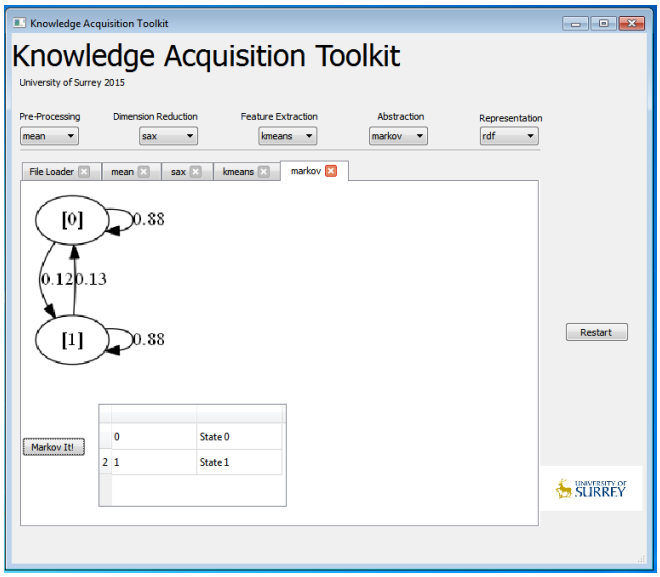


Figure 9: KAT – Data Abstraction

2.1.2.8. Link

The code of the KAT can be found here: https://github.com/CityPulse/KAT.git

2.1.3 Resource Management

The Resource Management component is responsible for managing all Data Wrappers. During runtime an application developer or the CityPulse framework operator can deploy new Data Wrappers to include data from new data streams. The folder "wrapper_dev" contains exemplary Data Wrappers for traffic and parking data of the city of Aarhus, Denmark as well as weather, air quality and incidents of the city of Brasov, Romania. The Resource Management component can be used for the following types of scenarios:

- Fetch live stream data via one or more Data wrappers
- Replay historic data embedded in a Data wrapper

2.1.3.1. Requirements

The Resource management is implemented in the programming language Python. The following Python packages have to be installed before using the component. The packages are available either in the package repository of the Ubuntu operating system or can be installed using PIP.

- Pika
- CherryPy
- Psycopg2
- NumPy
- SciPy
- SKLearn
- RDFlib
- Requests (version > 2.8)
- Requests-oauthlib
- Chardet

In addition, the following Ubuntu packages need to be installed in order to run the Resource management:

- python-pip
- libgeos++-dev
- libgeos-3.4.2
- python-dev
- libpq-dev
- git
- zip

2.1.3.2. Dependencies to other CityPulse components

For the Resource management in order to run properly it needs to have access to the following CityPulse components: Message Bus; Geospatial Data Infrastructure and the Knowledge Base (Triplestore).

2.1.3.3. Installation

As mentioned before, the CityPulse Resource management requires additional libraries, which can be installed using the following command on an Ubuntu Linux installation. The Resource management is not limited to Ubuntu Linux, but no other Linux distribution has been tested so far.

```
sudo apt-get install python-pip libgeos++-dev libgeos-3.4.2 python-dev libpq-dev
python-scipy git automake bison flex libtool gperf unzip python-matplotlib
```

In addition, using the following command required python packages will be installed: sudo pip install pika cherrypy shapely psycopg2 numpy sklearn rdflib chardet requests requests-oauthlib

The Resource management uses the Virtuoso triplestore to store annotated observations. As of February 2016, the virtuoso provided with the apt-repository in Ubuntu 14.04 LTS is outdated and lacks required features. Therefore, an installation from the sources is necessary. This can be achieved with the followings commands:

```
wget --no-check-certificate -q https://github.com/openlink/virtuoso-
opensource/archive/stable/7.zip -O virtuoso-opensource.zip
unzip -q virtuoso-opensource.zip
cd virtuoso-opensource
./autogen.sh
./configure
make
sudo make install
```

After that start the virtuoso:

```
sudo /etc/init.d/virtuoso-opensource-7 start
```

NOTE: the make command may hang after "VAD Sticker vad_dav.xml creation ..." if there is a virtuoso process running. Check with "ps ax|grep virtuoso" and kill if a virtuoso is running.

Afterwards you can download the Resource Management source code from the GitHub repository:

```
git clone https://github.com/CityPulse/CP_Resourcemanagement.git
```

The next step is to edit the configuration file with your favourite editor. An example configuration can be found in virtualisation/config.json. For details about the configuration file see Table 1. When running the Resource management in replay mode the python process may require a lot of file descriptors to read the historical data. Users may be required to increase a limit for file descriptors in the operating system. To change the limit on Mac OS X 10.10 and higher run the following command in a terminal:

```
sudo launchctl limit maxfiles 2560 unlimited
```

This will set the limit to 2560. On Linux

```
ulimit -n 2560
```

should do the trick. Add the line into the .bashrc in your home directory to make it permanent.

2.1.3.4. Configuration

The Resource management uses a configuration file to store the connection details to other components of the framework. The configuration is provided as JSON document named “config.json” within the “virtualisation” folder. The configuration consists of a dictionary object, where each inner element holds the connection details to one component of the framework, also as a dictionary. The following table lists all inner dictionary keys (bold) and their content.

Triplestore	
Driver	Driver The Resource Management supports the use of either Virtuoso or Apache Fuseki as triplestore. The value “sparql” tells the Resource Management to use Virtuoso. “fuseki” for Fuseki.
Host	The host name/IP of the triplestore as String.
Port	The port of the triplestore as integer.
Path	The path to the sparql-endpoint.
base_uri	The base URI is used to create the graph name.
Rabbitmq	
Host	The hostname/IP of the message bus as String.
Port	The port of the message bus as integer.
Interface	
The configuration of the HTTP based API interface. The API is realised using the CherryPy framework. The configuration here is directly passed to CherryPy’s ‘quickstart’ method. Therefore, all configuration options CherryPy provides are available. For more details see https://cherrypy.readthedocs.org/en/3.2.6/concepts/config.html#configuration.	
gdi_db	
Host	The hostname/IP of the geo-spatial database as String.
Port	The port of the geo-spatial database as integer.
Username	The username for the database as String.
Password	The user’s password for the database as String.
Database	The name of the database to use as String.

Table 1: Resource Management Configuration Parameters

2.1.3.5. Running the component

The Resource management is started via command line terminal. There are a series of command line arguments available to control the behaviour of the Resource management. In the following all command line arguments and their purpose.

Argument	Purpose
Replay	Start in replay mode. In replay mode historic sensor observations between the time frame START and END are used instead of live data. Also the replay speed can be influenced by the speed argument. Requires that the Resource Management has been started at least once before.
from START	In replay mode determines the start date. The format is "%Y-%m-%dT%H:%M:%S".
to END	In replay mode determines the end date. The format is "%Y-%m-%dT%H:%M:%S".
Messagebus	Enable the message bus feature. The Resource Management will connect to the message bus and publish new observation as soon as they are made.
Triplestore	Enable the triplestore feature.
Aggregate	Use the aggregation method, as specified in the SensorDescription, to aggregate new observations.
speed SPEED	In replay mode determines the speed of the artificial clock. The value range is [0-1000]. An artificial second within the replay will take 1000 – SPEED milliseconds.
Gdi	Geospatial Database Injection. Newly registered Data wrappers are reported to the Geospatial Database.
Gentle	Reduces the CPU load in replay mode, but slower.
Cleartriplestore	Deletes all graphs in the triplestore (may take up to 300s per wrapper!)
Restart	Restarts the Resource Management with the same arguments as last time.
Eventannotation	The Resource Management will listen on the message bus for new events to semantically annotate them and store them into the triplestore. Last feature requires the triplestore argument.

Table 2: Resource Management Execution Parameters

2.1.3.6. Link

The code of the Resource Management can be found here:
https://github.com/CityPulse/CP_Resourcemanagement

2.1.3.7. License of historical data

Historical data sets for the Aarhus Parking[1] and the Aarhus Traffic[2] data wrapper are collected from the Open Data Aarhus portal ODAA[3]. The data is published under the Creative Commons-license CC0 or CC-BY[4]. The format has been changed from JSON to CSV and the data has been sorted by the 'REPORT ID' or the 'GARAGECODE' respectively.

2.1.4 Decision Support and Contextual Filtering

The Contextual Filtering component and the Decision Support component are implemented in the same jar package as two different web-socket endpoints of the same server (we call it User-Centric Decision Support server. Therefore, the Decision Support component is automatically deployed when the Contextual Filtering component is deployed and vice versa.

[1] http://www.odaa.dk/dataset/parkeringshuse-i-aarhus
[2] http://www.odaa.dk/dataset/realtids-trafikdata
[3] http://www.odaa.dk/
[4] https://creativecommons.org/licenses/by/4.0/legalcode

2.1.4.1. Decision Support

The main role of the Decision Support component is to enable reactive decision support functionalities to be easily deployed, providing the most suitable answers at the right time by utilizing contextual information available as background knowledge, user patterns and preferences from the application as well as real-time events. The reasoning capabilities needed to support users in making better decisions require handling incomplete, diverse input, as well as constraints and preferences in the deduction process. This expressivity in the Decision Support component is achieved by using a declarative non-monotonic logic reasoning approach based on Answer Set Programming. The Decision Support component produces a set of answers to the reasoning request that satisfy all user's requirements and preferences in the best possible way. These solutions are computed by applying sets of rules deployed as scenario-driven decision support modules. We currently support three different types of decision support modules, covering a broad range of application scenarios:

- Scenario 1 (Routing APIs): It provides the best solution(s) for a routing task. The users not only identify starting point and ending point but are also able to provide various selection criteria (including constraints and preferences) in order to select the optimal routes.
- Scenario 2 (Parking space APIs): It provides the best selection among a set of available parking places based on optimisation criteria, constraints and preferences of the users.

2.1.4.1.1. Component prerequisite

The following applications have to be installed before using the component:

- Clingo4 (available at: http://potassco.sourceforge.net)
- Python 2.7 and package interruptingcow (Installation: $ pip install interruptingcow)

2.1.4.1.2. CityPulse framework dependencies

The Decision Support component in order to run properly needs to have access to the following CityPulse components:

- Routing component
- Triple store
- GDI
- Data Federation

2.1.4.1.3. Start Decision Support component

The Decision Support component is implemented as a websocket server endpoint (/reasoning_request) at port 8005. It requires a Reasoning Request in JSON format as an input. The following steps have to be achieved in order to deploy the component:

- Step 1: download the following resources provided at https://github.com/CityPulse/DecisionSupport-ContextualFiltering and place them in same folder:
  - jar package in folder /target with title CityPulseWP5-jar-with-dependencies.jar
  - folder res
- Step 2: edit the configuration file config.properties in folder res which includes the following fields:
  - hostname: the IP address of the server hosting the User-Centric Decision Support server.
  - port: the main port used by the User-Centric Decision Support server
  - routing_uri: the URI to access the Routing component (for Routing APIs only)

  - data_federation_uri: the URI to access the Data Federation component
  - knowledge_base_uri: the URI to access the triple store
  - GDI_URI: the URI of GDI component
  - clingo: the path of Clingo
- Step 3: run CityPulseWP5-jar-with-dependencies.jar to start the server.

2.1.4.1.4. Reasoning Request

The Reasoning Request is the request that an application/user should send to the CityPulse framework in order to perform a task that involves the Decision Support component. A Reasoning Request consists of:

- User Reference: uniquely identifies the user that made the request. Such reference is related to user credentials that will be used in the final integration activities in order to manage user logins and instances of the CityPulse framework in different cities.
- Type: indicates the reasoning task required by the application. This is used directly by the Decision support component to select which set of rules to apply, and needs to be identified among a set of available options at design time by the application developer.
- Functional Details: represent the qualitative criteria used in the reasoning task to produce a solution that best fits the user needs. The Functional Details are composed by:
  - Functional Parameters, defining mandatory information for the Reasoning Request (such as start and end location in a travel planner scenario);
  - Functional Constraints, defining a numerical threshold for specific functional aspects of the Reasoning Request (such as cost of a trip, distance or travel time in a travel planner scenario). These restrictions are evaluated as hard constraints, which needs to be fulfilled by each of the alternative solutions offered to the user;
  - Functional Preferences, which encode two types of soft constraints: a qualitative optimisation statement defined on the same functional aspects used in the Functional Constraint (such as minimisation of the travel time); or a qualitative partial order over such optimization statements (such as preference on the minimisation of the distance over minimization of the travel time). Preferences are used by the Decision support component to provide to the user the optimal solution among those verifying the functional constraints.

In what follows, we detail each element of the Reasoning Request for two mentioned scenarios Routing and Parking space.

2.1.4.1.5. Reasoning Request for Routing module

The Reasoning Request of the Routing scenario includes the following parameters:

- Type: TRAVEL-PLANNER
- Functional parameters:
  - STARTING_POINT: the coordinate of starting point
  - ENDING_POINT: the coordinate of ending point
  - STARTING_DATETIME: the desired date time to start traveling
  - TRANSPORTATION_TYPE: the vehicle that the user uses to travel. This parameter includes CAR, WALK, BICYCLE
- Functional constraints:
  - TRAVEL_TIME LESS_THAN X: indicates that the travel time should be less than X
  - DISTANCE LESS_THAN X: indicates that the distance should be less than X
  - POLLUTION LESS_THAN X: indicates that the pollution amount should be less than X
- Functional preferences:

- MINIMIZE TRAVEL_TIME: indicates that the calculated route should prefer to minimize the time of travel
- MINIMIZE DISTANCE: indicates that the calculated route should prefer to minimize the distance of travel
- MINIMIZE POLLUTION: indicates that the calculated route should prefer to minimize the pollution amount on the route

2.1.4.1.6. Sample Reasoning Request for Routing module

Bellow an example of the reasoning request of a user asking routes from STARTING_POINT to ENDING_POINT when he travels by car at particular datetime. As his constraint, the routes must have pollution level is less than 135. Moreover, he wants to have fastest routes (minimize in time).

```
{
   "arType":"TRAVEL_PLANNER",
   "functionalDetails":{
      "functionalConstraints":{
         "functionalConstraints":[
            {
               "name":"POLLUTION",
               "operator":"LESS_THAN",
               "value":"135"
            }
         ]
      },
      "functionalParameters":{
         "functionalParameters":[
            {
               "name":"STARTING_DATETIME",
               "value":"1434110314540"
            },
            {
               "name":"STARTING_POINT",
               "value":"10.103644989430904 56.232567308059835"
            },
            {
               "name":"ENDING_POINT",
               "value":"10.203921 56.162939"
            },
            {
               "name":"TRANSPORTATION_TYPE",
               "value":"car"
            }
         ]
      },
      "functionalPreferences":{
         "functionalPreferences":[
            {
               "operation":"MINIMIZE",
               "value":"TIME",
               "order":1
            }
         ]
      }
   },
   "user":{

   }
}
```

2.1.4.1.7. Reasoning Request for Parking Space module

The Reasoning Request of the Parking Space scenario includes the following parameters:

- Type: PARKING_SPACES
- Functional parameters:
  - STARTING_POINT: the coordinate of starting point
  - POINT_OF_INTEREST: the coordinate of point of interest
  - STARTING_DATETIME: the desired date time to start traveling
  - DISTANCE_RANGE: the desired distance from the POINT_OF_INTEREST to any available parking slot.
  - TIME_OF_STAY: the time for parking the car
- Functional constraints:
  - COST LESS_THAN X: indicates that the parking cost should be less than X
  - WALK LESS_THAN X: indicates that the distance to walk from the parking slot to the POINT_OF_INTEREST should be less than X
- Functional preferences:
  - MINIMIZE COST: indicates that the calculated parking slot should prefer to minimize the cost
  - MINIMIZE WALK: indicates that the calculated parking slot should prefer to minimize the distance to the POINT_OF_INTEREST

2.1.4.1.8. Sample Reasoning Request for Parking Space module

Bellow an example of the reasoning request of a user asking available parking slot near his POINT_OF_INTEREST at a particular datetime. The parking slots should be in 1000m far from his point of interest (DISTANCE_RANGE) and he will leave his car there for 100 minutes. As his constraint, he wants the parking cost is less than 100 euros.

```
{
        "arType":"PARKING_SPACES",
        "functionalDetails":{
          "functionalConstraints":{
            "functionalConstraints":[
              {
                "name":"COST",
                "operator":"LESS_THAN",
                "value":"100"
              }
            ]
        },
          "functionalParameters":{
            "functionalParameters":[
              {
                "name":"STARTING_DATETIME",
                "value":"1455798332761"
              },
              {
                "name":"POINT_OF_INTEREST",
                "value":"10.1827464 56.16169679999999"
              },
              {
                "name":"DISTANCE_RANGE",
                "value":"1000"
              },
              {
                "name":"TIME_OF_STAY",
                "value":"100"
              },
              {
                "name":"STARTING_POINT",
                "value":"10.103644989430904 56.232567308059835"
              }
            ]
          },
          "functionalPreferences":{
            "functionalPreferences":[

            ]
          }
        },
        "user":{

        }
}
```


2.1.4.2. Contextual Filtering

The main role of the Contextual Filtering component is to continuously identify and filter events that might affect the optimal results of the decision making task (performed by the Decision Support component). The users need to input their current context such as place of interest, filtering factors, and ranking factors. The Contextual Filtering will subscribe to events only in the place of interest and determine which event is critical to users based on filtering and ranking factors.

Known limitations:

- User Context Ontology: the current ontology about user's context includes only the activity of the user. The developer can extend this ontology for their specific scenarios.

2.1.4.2.1. Component prerequisite

The following applications have to be installed before using the component:

- Clingo4 (available at: http://potassco.sourceforge.net)

2.1.4.2.2. CityPulse framework dependencies

The Contextual Filtering component needs to have access to the following CityPulse components:

- Geospatial Data Infrastructure (GDI)
- Event Detection

2.1.4.2.3. Start Contextual Filtering component

The Contextual Filtering component is implemented as a websocket server endpoint (/contextual_events_request) at port 8005. It requires a Contexual Event Request in JSON format as an input.

The following steps have to be achieved in order to deploy the component

- Step 1: download the following resources provided at https://github.com/CityPulse/DecisionSupport-ContextualFiltering and place them in the same folder:
  - jar package in folder /target with title CityPulseWP5-jar-with-dependencies.jar
  - folder res
- Step 2: edit the configuration file config.properties in the folder res which includes the following fields:
  - hostname: the IP address of the server hosting the User-Centric Decision Support server.
  - port: the main port used by the User-Centric Decision Support server
  - GDI_URI: the URI of GDI component
  - eventRabbitURI: the URI of rabbitMQ to subscribe events from the Event Detection component
- Step 3: run CityPulseWP5-jar-with-dependencies.jar to start the server.

2.1.4.2.4. Contextual Event Request

The Contextual Event Request is the request that an application/user should send to the CityPulse framework in order to perform a task that involves the Contextual Filtering component. It consists of:

- Place of interest: identifies place to subscribe critical events. Currently, we support three types of place
  - Point: one coordinate
  - Route: a list of coordinates in which starting coordinate different to ending coordinate.
  - Area: a list of coordinates which represent to a polygon

- Filtering Factors: used to filter unrelated (unwanted) events and includes event's source, event's category, and user's activity
- Ranking Factor: identifies which metric is preferred by the user for ranking the criticality of incoming events. We have currently implemented the Ranking Factor based on two metrics: distance, and gravity of the event. In order to combine these two metrics, we use the linear combination approach, where the user can identify weights (or importance) for each metric.

2.1.4.2.5. Sample Contextual Event Request

Bellow an example of the Contextual Event Request of a user asking about critical events happening in his "place" when he is traveling by car. The criticality of event is identified by EVENT_LEVEL and DISTANCE between event's location and the user's current location.

```
{
    "filteringFactors": [
        {
            "name": "ACTIVITY",
            "values": [
                {
                    "value": "CarCommute"
                }
            ]
        }
    ],
    "messageType": "ContextualEventRequest",
    "place":
"{\"route\":[{\"latitude\":56.156399,\"longitude\":10.2138401},…,\"placeId\":\"0
\",\"type\":\"ROUTE\"}",
    "rankingFactor": {
        "rankingElements": [
            {
                "name": "EVENT_LEVEL",
                "value": {
                    "value": 30
                }
            },
            {
                "name": "DISTANCE",
                "value": {
                    "value": 70
                }
            }
        ],
        "type": "LINEAR"
    }
}
```

NOTE: Only partial message. The full message can be found here: https://github.com/CityPulse/DecisionSupport-ContextualFiltering/blob/master/README-ContextualFiltering.md#sample-contextual-event-request

2.1.5 SAOPY

How to use saopy (source: http://iot.ee.surrey.ac.uk/citypulse/ontologies/sao/saopy.html)

saopy depends on:

1. rdflib (http://rdflib.org/) v2.4.0 (easy_install)

saopy can be downloaded from the following link: SAOPY (v1.1.4). Since it is an ongoing work, please make sure that you are using the latest version. In order to install SAOPY library, you have to use the following command in the directory that you have downloaded the SAOPY python wheels.

```
$ pip install ./saopy-1.1.4-py2.py3-none-any.whl
```

then start python using

```
$ python
```

Now let's open the python interpreter and give it a go...

```
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.56)] on darwin. Type "help",
"copyright", "credits" or "license" for more information.
```

Start by importing the saopy package :

```
import saopy
```

To view all "saopy" classes:

```
dir(saopy)

['DUL', 'PropertySet', 'RDFInterface', 'SaoInfo', 'builtins', 'doc', 'file', 'name',
'package', 'path', 'ces', 'ct', 'exportRDFFile', 'exportRDFGraph', 'foaf', 'geo',
'importRDFFile', 'importRDFGraph', 'model', 'muo', 'owl', 'owlsg', 'owlss', 'owlssc',
'owlssp', 'owlssrp', 'prov', 'qoi', 'rdfs', 'sao', 'ssn', 'tl', 'tm', 'tzont']
```

To view all classes of sao ontology from saopy library:

```
dir(saopy.sao)

['DiscreteCosineTransform', 'DiscreteFourierTransform', 'Mean', 'Median',
'PiecewiseAggregateApproximation', 'Point', 'Segment', 'StreamAnalysis', 'StreamData',
'StreamEvent', 'SymbolicAggregateApproximation', 'builtins', 'doc', 'file', 'name',
'package', 'path', 'saopy']
```

Creating a sensor object:

```
sensor124 = saopy.ssn.Sensor("http://example.org/x")
```

Why shall we use saopy for annotation of IoT data? Saopy enables a user to avoid various common problems including use of undefined properties and classes, poorly formed namespaces, problematic prefixes, literal syntax and other optional heuristics.

To mash up multiple sensory objects for serialisation, you can use the ``SaoInfo" class. The following

```
sensor124 = saopy.ssn.Senzor("http://example.org/x")

Traceback (most recent call last): File "", line 1, in AttributeError: 'module' object
has no attribute 'Senzor'

cityofaarhus = saopy.foaf.Organisation("http://example.org/cityofaarhus")

Traceback (most recent call last): File "", line 1, in AttributeError: 'module' object
has no attribute 'Organisation'
```

example shows how to create and export a sensor observation object with a value:

```
trafficData124 = saopy.sao.StreamData("http://example.org/data1")

trafficData124.value = "1234"

saoOut = saopy.SaoInfo()

saoOut.add(trafficData124)

saopy.RDFInterface.exportRDFFile(saoOut,”example1.rdf”,"n3")
```

Now that we know how to describe a simple observation, let's start annotating more detailed sensory observation. First create provenance information for sensory data:

```
cityofaarhus = saopy.foaf.Organization("http://example.org/cityofaarhus")

cityofaarhus = saopy.prov.Agent("http://example.org/cityofaarhus")

trafficsensor158324 = saopy.ssn.Sensor("http://example.org/data158324")

trafficsensor158324.actedOnBehalfOf = cityofaarhus

creating properties of sensory data:

measuredTime = saopy.ssn.Property("http://unis/ics/property003")

measuredTime.description = "Measured Time"

estimatedTime = saopy.ssn.Property("http://unis/ics/property004")

estimatedTime.description = "Estimated Time"

avgSpeed = saopy.ssn.Property("http://unis/ics/property001")

avgSpeed.description = "Average Speed"

vcCount = saopy.ssn.Property("http://unis/ics/property002")

vcCount.description = "Vehicle Count"

trafficsensor158324.observes.add(vcCount)

trafficsensor158324.observes.add(avgSpeed)

trafficsensor158324.observes.add(estimatedTime)

trafficsensor158324.observes.add(measuredTime)
```

SAOPY allows to describe time instants and intervals. Time instants should only be used with tl:at to describe the time instance that data has been collected, whereas time interval should specify both the beginning time of the interval and duration. Here we provide both of the examples.

SAO ontology subsumes the measurement unit descriptions from Measurement Unit Ontology (muo). Therefore, it enables to describe measurement unit of an observation as follows:

```
unitseconds = saopy.muo.UnitOfMeasurement("http://unis/ics/unit1:seconds")

unitkilometer = saopy.muo.UnitOfMeasurement("http://unis/ics/unit2:km-per-hour")
```

Now, we can annotate sensor observations for two sensor features, namely average speed and

```
trafficData001 = saopy.sao.StreamData("http://unis/ics/trafficdataavgspeed001")

trafficData001.value = "60"

trafficData001.hasUnitOfMeasurement=unitkilometer

trafficData001.observedProperty = avgSpeed

trafficData001.observedBy = trafficsensor158324

trafficData001.time = instant

trafficData003 = saopy.sao.StreamData("http://unis/ics/trafficdataMeasuredTime001")

trafficData003.value = "30"

trafficData003.hasUnitOfMeasurement=unitseconds

trafficData003.observedProperty = measuredTime

trafficData003.observedBy = trafficsensor158324

trafficData003.time = interval
```

measure time:

```
universaltimeline =
saopy.tl.PhysicalTimeLine("http://purl.org/NET/c4dm/timeline.owl#universaltimeline")

instant = saopy.tl.Instant("http://unis/ics/timeinstant")

interval = saopy.tl.Interval("http://unis/ics/timeinterval")

instant.at = "2014-09-30T06:00:00"

instant.onTimeLine = universaltimeline

interval.beginsAtDateTime = "2014-09-30T06:00:00"

interval.durationXSD = "PT5M"
```

exporting sensor data in N3 format:

```
saoOut.add(trafficData001)

saoOut.add(trafficData003)

saoOut.add(cityofaarhus)

saoOut.add(trafficsensor158324)

saoOut.add(estimatedTime)

saoOut.add(measuredTime)

saoOut.add(avgSpeed)

saoOut.add(vcCount)

saoOut.add(unitkilometer)

saoOut.add(unitseconds)

saoOut.add(universaltimeline)

saoOut.add(instant)

saoOut.add(interval)

saopy.RDFInterface.exportRDFFile(saoOut, "example2.rdf","n3")
```

SAOPY also allows to annotate quality values, such as correctness, frequency, age and completeness. The following example illustrates how to annotate quality features for a data segment of the traffic data stream. The correctness has been obtained based on an algorithm by examining the difference between two successful submitted traffic streams to validate the correctness quality value.

```
age = saopy.qoi.Age("age-sample-1")

segmentSample1 = saopy.sao.Segment("segment-sample-1")

age.hasAge = "10"

completeness = saopy.qoi.Completeness("completenesssample-1")

completeness.hasCompleteness = "1"

correctness = saopy.qoi.Correctness("correctnesssample-1")

correctness.hasCorrectness = "1"

frequency = saopy.qoi.Frequency("frequency-sample-1")

frequency.hasFrequency = "10"

segmentSample1.hasQuality.add(frequency)

segmentSample1.hasQuality.add(correctness)

segmentSample1.hasQuality.add(completeness)

segmentSample1.hasQuality.add(age)

owner = saopy.prov.Person("MisterX")

segmentSample1.hasProvenance = owner

saoOut = saopy.SaoInfo()

saoOut.add(segmentSample1)

saoOut.add(age)

saoOut.add(completeness)

saoOut.add(correctness)

saoOut.add(frequency)

saoOut.add(owner)

saopy.RDFInterface.exportRDFFile(saoOut, "example3.rdf","n3")
```

2.1.6 Event Detection

The Event detection component provides the generic tools for processing the annotated as well as aggregated data streams to obtain events occurring into the city. This component is highly flexible in deploying new event detection mechanisms, since different smart city applications require different events to be detected from the same data sources.

Figure 10 depicts the architecture of the proposed framework which allows event detection for sensory data streams. It provides a set of generic tools which can be used for detecting an event. In order to detect a certain event, the domain expert, has to deploy a piece of code called event detection node. These nodes contain the event detection logic, which represent the pattern that has to be identified during raw data processing.

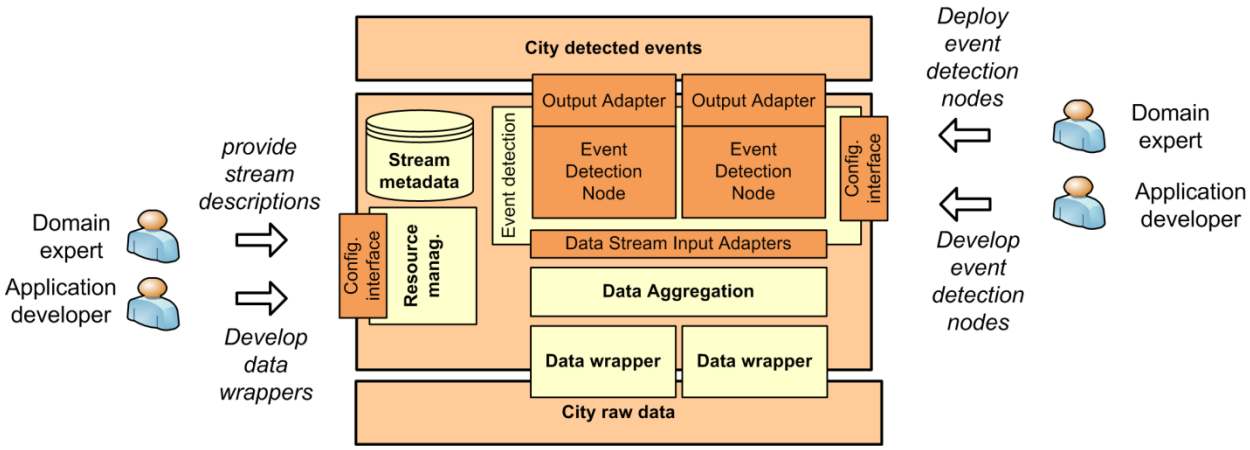


Figure 10 Event detection component main blocks and the dependent components.

A domain expert, with some help from an application developer, can use two configuration interfaces displayed by the framework in order to perform the following activities:

- Describe how to access the streams endpoints and how to interpret the received messages on a design time phase;
- Register the data streams from the city by providing their descriptions;
- Develop the event detection nodes on a design time phase;
- Deploy the event detection nodes for various stream combinations.

Within the CityPulse project life time a suit of event detection nodes have been developed (e.g. traffic jam detection, parking garages status changed detection). The application developer can, at any time, trigger the deployment of these mechanisms in order to analyse the data coming from the considered city. In addition to that these, the application developer, by implementing a java interface, can develop custom made event detection logic.

These instructions will get you a copy of the project up and running on your local machine for development and testing purposes. See deployment for notes on how to deploy the project on a live system.

2.1.6.1. Other CityPulse framework dependencies

- The Event Detection (ED) component requires a connection to Geo-spatial data base (GDI) in order to send it events that it has detected in its deployed nodes.

- ED component also connects to the Data bus from where it gets the annotated and aggregated data.
- Resource Manager Component is another dependency for the ED. ED connects to it to get the sensors description, information that is later used in the interpretation on aggregated and annotated data.

2.1.6.2. Component deployment steps and Configuration file

The Event Detection component is packed in a .jar file. To be used it has to be included in your project. The component has a configuration file named config.properties that holds the configuration parameters required for connecting to several other components from the CityPulse framework such as:

- Resource Manager Configuration parameters: IP and Port.
- The Data bus Configuration parameters: IP and Port.
- Geo-spatial data base Configuration parameters: IP and Port.
- Testing mode parameter: testParameter (Boolean type).
- The URI for GDI parameter: GDI_AMQP_URI.

The configuration file can be found in /resources within the .jar file and can be modified easily.

2.1.6.3. API method description, parameters and response

In the main method, the first thing to do is create e new EventDetection object. By doing this you will initialize Esper, GDI, connect to the message bus from where you get the aggregates/annotated messages and to the message bus used to send the newly detected events to GDI.

Then we need to create a new class that extends EventDetectionNode (let us say it is named ParkingEventDetectionNode) and initialize it in the main method you just created earlier, with the required parameters (plus the thresholds). Then, again in the main method, create a Coordinate object where you set the coordinates of the sensor. Then use the method setEventCoordinate on the ParkingEventDetectionNode object. You will also create a HashMap containing as key the UUID of the stream and as value the stream's name (eg: parkingGarageData) the same name you used when creating your custom node (in the method getListOfInputStreamNames()).

Then simply add the new custom node object to EventDetection using the method addEventDetectionNode().
And that's about it. In your custom node class ParkingEventDetectionNode in the method getEventDetectionLogic(EPServiceProvider epService) write all your Esper queries. When you get your desired event, simply create a ContextualEvent object with the required parameters and use the method sendEvent to send your ContextualEvent to GDI.

2.1.6.4. Methods

Can be seen in the Javadoc[5] file.

[5] https://github.com/CityPulse/EventDetection/tree/master/javadoc

2.1.7 ACEIS

The Automatic Complex Event Implementation System (ACEIS) is a middleware for complex event services. It is implemented to fulfill large-scale data analysis requirements in the CityPulse project and is responsible for event service discovery, composition, deployment, execution and adaptation. In the CityPulse project, data streams are annotated as event services, using the Complex Event Service Ontology. Event service discovery and composition refer to finding a data stream or a composite set of data streams to address the user-defined event request. Event service deployment refers to transforming the event service composition results (a.k.a., composition plans) into RDF Stream Processing (RSP) queries and registering these queries to relevant RSP engines, e.g., CQELS and C-SPARQL. Event service execution refers to establishing the input and output connections for the RSP engines to allow them consuming real-time data from lower-level data providers and delivering query results to upper-level data consumers. Event services adaptation ensures the quality constraints over event services are ensured at run-time, by detecting quality changes and make adjustments automatically. Figure 11 illustrates the architecture of ACEIS.

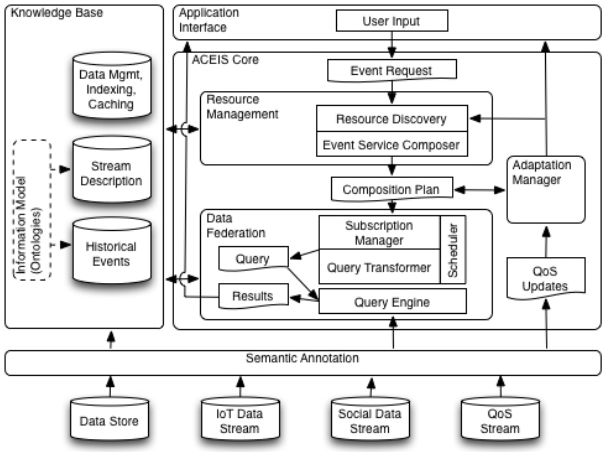


Figure 11: ACEIS Architecture

2.1.7.1. Prerequisite

The following applications have to be installed before using the component:

- JDK 1.7
- Web Server (e.g., Jboss 7.1.2)
- CQELS engine (available at:)
- CSPARQL engine (available at:)
- Other Java libraries located at https://github.com/CityPulse/ACEIS

2.1.7.2. CityPulse framework dependencies

- Resource management, including the knowledge base, to access the stream meta-data.
- Data aggregation, to consume aggregated observations.
- Data bus, to access real-time observations.
- Data quality analysis, to receive static and dynamic data quality information.

2.1.7.3. Installation

Download resources and executables at https://github.com/CityPulse/ACEIS . No further installation required.

2.1.7.4. Configuration

The file "aceis.properties" is the static configuration file, it consists the following configuration parameters:

- **hostIp**: the IP address of the server hosting data federation server,
- **port**: the main port used by the data federation server,
- **ontology**: the folder storing the ontology files (used only if local dataset files are used),
- **streams**: the folder storing the stream data files for simulation (used only if operating on simulated streams),
- **dataset**: the location of stream meta-data, could be a local file or an URI for the virtuoso endpoint provided by the resource management component, and
- **request**: the location of sample test queries.

2.1.7.5. Deploying and Running the component

- Step 1: download resources provided at https://github.com/CityPulse/ACEIS .
- Step 2: check and edit configuration file when necessary.
- Step 3: run ACEIS.jar or compile from Main.java to start ACEIS server, with the following program parameters: **cqelsCnt**: number of CQELS engine instances, **csparqlCnt**: number of CSPARQL engine instances, **smode**: load balancing strategy, could be "elastic", "balancedLatency", "balancedQueries", "rotation" or "elastic", **qCnt**: number of concurrent queries (used only in simulation mode), **step**: interval between registering new queries (used only in simulation mode), **query**: path to the query file (used only in simulation mode), **duration**: life-time of the ACEIS server instance, 0 means indefinite.
- Alternatively to Step 3: run $sh run.sh to skip step 3 and start ACEIS with default settings.

2.1.7.6. API specification

See javadoc hosted at: http://fenggao86.github.io/ACEIS/doc/index.html

2.1.8 Fault Recovery

The Fault recovery component ensures the continuous and proper operation of the CityPulse enabled application by generating estimated values for the data stream when the quality drops or it has temporally missing observations. When the quality of the data stream is low for a longer time period, an alternative data source has to be selected. The selection can be performed automatically by the Technical adaptation component. In other words, the technical adaptation process does not have to be triggered if the QoI of a stream is low only for a short period of time because the Fault recovery component provides estimated values.

The Fault recovery component is integrated into the data wrapper. The fault recovery mechanism is triggered to generate an estimated value when the atomic monitoring component has determined that the current observation is invalid or missing.

The building blocks of the Fault recovery component are presented in the image below. The component has a buffer which temporary stores the latest observations generated by the data stream, and a reference dataset which contains sequences of valid consecutive observations from the data stream. When an estimated value is requested, the k-nearest neighbor algorithm is used to select a few sequences of observations from the references dataset, which are similar to the current situation. At the end the estimated value is computed from the selected sequences of observations.

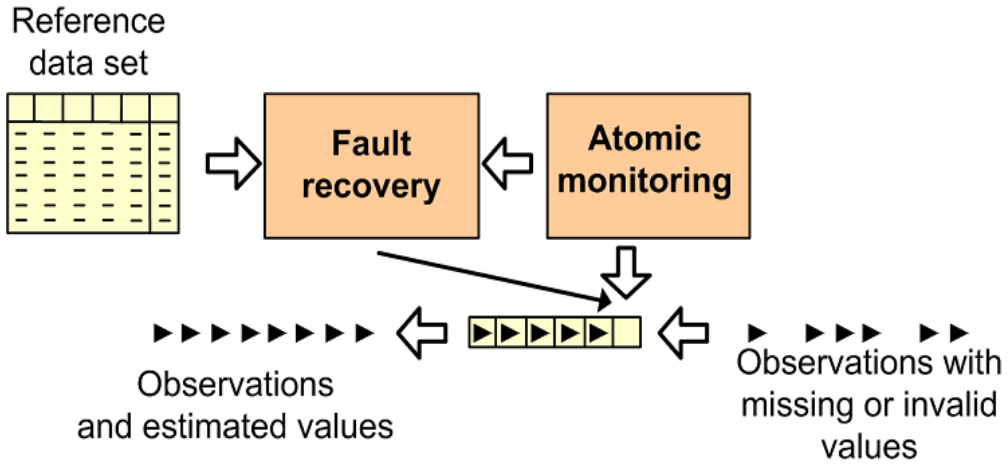


Figure 12: Fault Recovery process flow

During the normal operation, when the QoI of the stream is high, the fault recovery component extends the reference data set with the sequences of observations from the buffer if a similar signal pattern was not included before.

Initially, when the Data wrapper is deployed, the reference data set is empty and based on the normal operation (from stream quality point of view) it is extended. As a result of that, the work of the 3rd party application developer is reduced, because he does not have to collect historic data from the stream, to clean and to validate it in order to create the reference dataset. Using the API exposed by the resource management, the 3rd party application developer can turn on and off this component based on the CityPulse enabled application requirements.

2.1.8.1. Component pre-requisite

The following applications have to be installed before using the component:

- Python 2.7: https://www.python.org/download/releases/2.7
- SciLab library for Python

2.1.8.2. CityPulse framework dependencies

The Fault Recovery component in order to run properly needs to have access to the following CityPulse components:

- Data wrapper

2.1.8.3. Component deployment procedure

It is already integrated in the data wrapper.

2.1.8.4. Component location on GitHub

This component can be found at the following GitHub location:
https://github.com/CityPulse/FaultRecovery

2.1.9 Composite Monitoring

The Composite Monitoring is triggered by events from the Event Detection, by the Atomic Monitoring (in case of QoI metric drops) or for a manual event or stream evaluation by the visual Quality Explorer interface. In addition to the Atomic Monitoring the Composite Monitoring utilises historic time series of various dependent sensor streams. Therefore, a direct access to sensor information in the Resource Management is necessary.

Figure 13 shows a sequence diagram for the plausibility evaluation of a stream. The Composite Monitoring receives a request to check the correctness of a sensor stream for a specific timespan. Based on the propagation and distance model of [CityPulse-D4.2] a set of relevant sensors is identified to acquire information for comparison. Therefore, depending on the sensor's nature, distance models e.g., shortest path distance on the roads, Euclidean distance for radial propagation or infrastructure based metrics (e.g., change of road types) are used in the method *analyseInfrastructure* of the Geospatial Data Infrastructure (see Table 3). The distances between the selected streams and the *streamIDs* are returned to the Composite Monitoring component. In the next step the historic sensor datasets for those streams are fetched from the Resource Management.

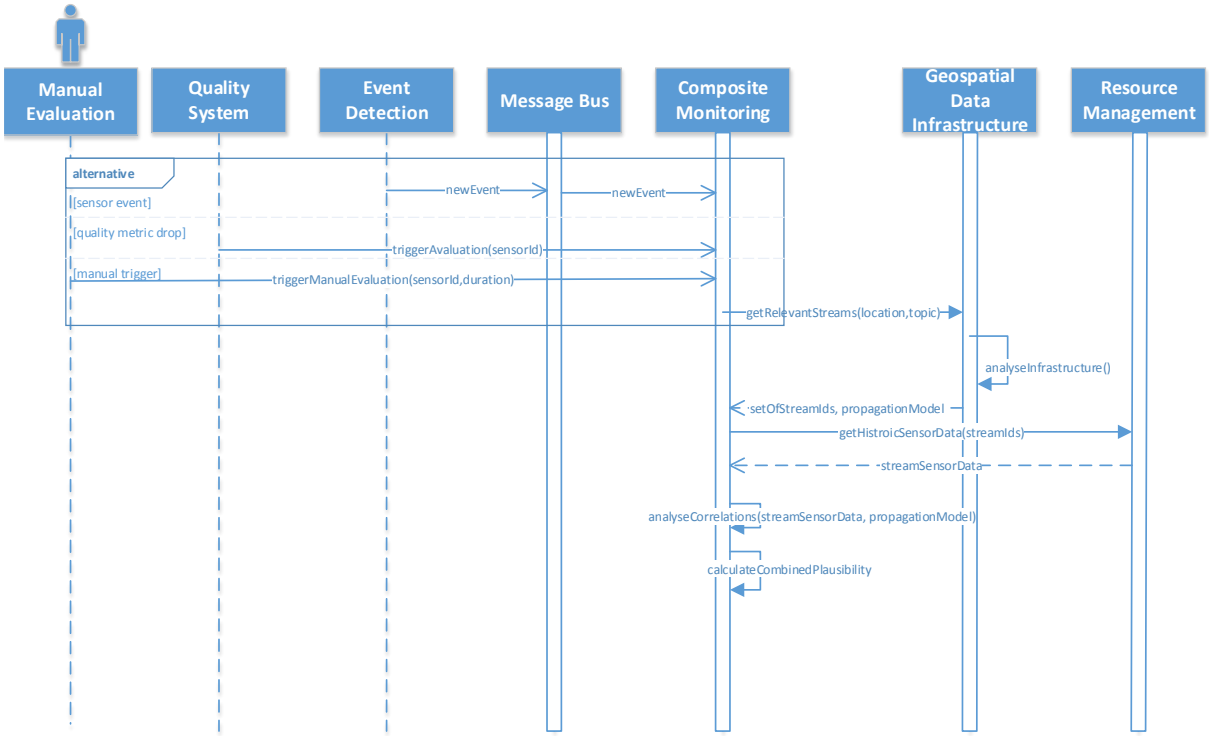


Figure 13: Sequence Diagram of the Composite Monitoring Process

Metric Parameter	Processing
euclidean_distance	Direct Euclidean distance between two sensors in meters regarding earth curvature.
route_distance	Route distance between two sensors in meters .
route_duration	Travel time by car of the route between two sensors.
route_steps	Number of steps (road crossover, turns, etc.) of the route between two sensors.
directed_duration	Additive composition of the duration and the angle between two sensors.
sensorHops	Number of sensors to jump over to reach from one sensor to another. Equal to number of sensors if it is not possible to reach another sensor.

Table 3: Implemented Distance Metrics in the Geospatial Data Infrastructure Module

The method *analyseCorrelations(streamSensorData, propagationModel)* offers a set of pre-processing options for the data series (see Table 4).

Identifier	Modification of data
raw	Unmodified input data. Limited significance due to seasonal time series patterns.
dct	Data with high frequency filter discrete cosine transform: DCT (data)
random	Data minus seasonal amount over given period: data – seasonal (data)
random_dct	Dct filtered random data: DCT (data – seasonal (data))
dct_random	Random of dct filtered data: DCT (data) – seasonal (DCT (data))
dct_random_dct	Dct filtered random of dct filtered data: DCT (dct_random)
offset_ ...	The time series of the second stream is shifted according to the estimated duration that is calculated with the propagation model.

Table 4: Time Series Pre-Processing Options

2.1.9.1. API method description, parameters and response

In standard mode the Composite monitoring is acting as a standalone component that monitors Resource Management activities and evaluates events and streams. To get a demonstrational benefit he Correlation monitoring offers a RESTbased Web Service for evaluation purpose. It is triggered via the following commands:

- GET: http://<host>/triggerEvaluation/<sensorId>
- GET: http://<host>/triggerEvaluation/<eventId>
- POST: http://<host>/triggerManualEvaluation/<List of Stream IDs>
  Content: Correlation Model File

Full method overview and parameter explanations can be found in the GitHub Repository.

2.1.9.2. Requirements and Dependencies

- R >= V3.2.3

required libraries:

- Jsonlite

- Rshiny
- Rgeos
- FastRWeb
- plyr
- data.table
- ggplot2
- parallel
- RPostgreSQL

An installation script for R libraries can be found in the installLibraries.R scriptfile.

2.1.9.3. Component deployment steps and Configuration file

The GIT repository can be cloned from https://github.com/CityPulse/CompositeMonitoring

The Composite monitoring is a standalone R-Project. The component has a configuration file config.json stores the configuration parameters required for connecting to several other components from the CityPulse framework and configuring its own interface:

- Resource Manager Configuration parameters: IP and Port.
- Geo-spatial data base Configuration parameters: IP and Port.
- Webserver port, bind address
- Correlation Model input file

The project can be started by executing:

```
R –f compositeMonitoring.R
```

2.2. Datasets

In this section we list the different datasets used by the CityPulse components and applications. The information has been extracted from the following link: http://iot.ee.surrey.ac.uk:8080/datasets.html

Vehicle Traffic, Provided by City of Aarhus in Denmark	
Description	A collection of datasets of vehicle traffic, observed between two points for a set duration of time over a period of 6 months (449 observation points in total). The data is available in raw (CSV) and semantically annotated format using the CityPulse information model.
Further information	http://iot.ee.surrey.ac.uk:8080/datasets.html#traffic

Table 5: Vehicle Traffic in Aarhus Dataset

Pollution Measurements (generated data)	
Description	A collection of pollution measurements designed to complement the vehicle traffic dataset above. For this pollution mockup stream we decided to simulate one sensor for each of the traffic sensor at the exact location of this traffic sensor. The data is measured using Air Quality Index[6] metric (449 observation points in total). The data is available in raw (CSV) and semantically annotated format using the CityPulse information model.
Further information	http://iot.ee.surrey.ac.uk:8080/datasets.html#pollution

Table 6: Pollution Measurement in Aarhus Dataset

Pollution Measurements for the City of Brasov in Romania (generated data)	
Description	A collection of pollution measurements designed to complement the vehicle traffic dataset above. For this pollution mockup stream we decided to simulate one sensor for each of the traffic sensor at the exact location of this traffic sensor. The data is measured using Air Quality Index metric (449 observation points in total). The data is available in raw (CSV) and semantically annotated format using the CityPulse information model.
Further information	http://iot.ee.surrey.ac.uk:8080/datasets.html#pollution

Table 7: Pollution Measurement in Brasov Dataset

Parking Data Stream, Provided by City of Aarhus in Denmark	
Description	A datastream with parking data provided from the city of Aarhus. There are a total of 8 parking lots providing information over a period of 6 months (55.264 data points in total).
Further information	http://iot.ee.surrey.ac.uk:8080/datasets.html#parking

Table 8: Parking Data Stream in Aarhus

Weather Data for the City of Aarhus in Denmark	
Description	A collection of datasets of weather observations from the city of Aarhus.
Further information	http://iot.ee.surrey.ac.uk:8080/datasets.html#weather

Table 9: Weather Data for Aarhus

[6] https://en.wikipedia.org/wiki/Air_quality_index

Weather Data for the City of Brasov in Romania	
Description	A collection of datasets of weather observations from the city of Brasov.
Further information	http://iot.ee.surrey.ac.uk:8080/datasets.html#weather

Table 10: Weather Data for Brasov

Webcasted Events Dataset, Provided by City of Surrey in the United Kingdom	
Description	A set of webcasted social events from the municipality of Surrey, provided here as a datastream from the city of Surrey, the website is www.surreycc.public-i.tv/core/portal.
Further information	http://iot.ee.surrey.ac.uk:8080/datasets.html#eventss

Table 11: Webcasted Events Dataset for Surrey

Cultural Events Dataset, Provided by City of Aarhus in Denmark	
Description	A set of cultural event announcements provided as a datastream from the municipality of Aarhus.
Further information	http://iot.ee.surrey.ac.uk:8080/datasets.html#eventsc

Table 12: Cultural Events for Aarhus Dataset

Library events, Provided by City of Aarhus in Denmark	
Description	A set of events hosted by libraries in Denmark (1548 events in total).
Further information	http://iot.ee.surrey.ac.uk:8080/datasets.html#library

Table 13: Library Events in Aarhus Dataset

Public Transport Scheduling Data in Sweden	
Description	An API that provides timetables for all trains, buses, planes, subways and trams in Sweden.
Further information	https://www.trafiklab.se/api/resrobot-reseplanerare

Table 14: Public Transport Scheduling Data for Sweden

Real-time Traffic Situation in Sweden	
Description	Provides information regarding the current status of traffic and real-time information regarding bus, subway, commuter trains, local trains and boats.
Further information	https://www.trafiklab.se/api/sl-trafiklaget-2 https://www.trafiklab.se/api/sl-realtidsinformation-3

Table 15: Real-time traffic data in Sweden

Points of Interest/Events in Sweden – Museums, Sports and Meetings	
Description	Different information regarding museums and historical notes of monuments in Sweden; sports events happening and general meetings.
Further information	http://www.ksamsok.se/in-english/ https://github.com/menmo/everysport-api-documentation www.meetup.com/meetup_api

Table 16: Points of Interest and Events in Sweden

2.3. Tools

In this section we briefly describe the different tools that have been developed in CityPulse. Tools include different ontologies and other IoT data related tools. The information has been extracted from: http://www.ict-citypulse.eu/page/content/tools-and-data-sets

2.3.1. Quality Ontology

The Quality Ontology is used to represent the quality of information for data streams in smart cities. This quality can then be used to select streams by the quality of the information they provide. The annotated data streams are represented with the Stream Annotation Ontology. To provide information about the provenance of the streams parts of the PROV-O[7] ontology are used.

Figure 14 depicts the Quality Ontology in detail. It shows the combination with the SAO and the PROV-O. It allows to annotate StreamData of the SAO with Quality. The Quality itself has five categories with subcategories, which are not shown in the figure, to describe the attributes of the annotated data stream regarding its quality. In addition, it provides a concept of trustworthiness for data sources. This concept is realised by two additional object properties hasProvenance and hasReputation to link the StreamData with an Agent as the owner and its Reputation.

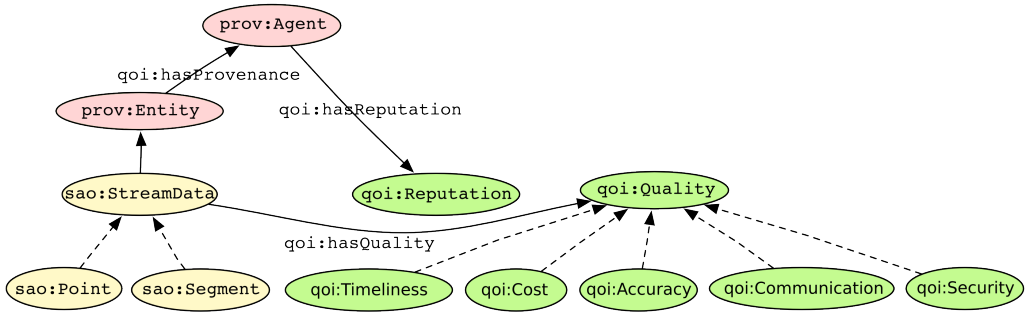


Figure 14: Quality Ontology

Further information about the Quality Ontology can be found here: http://purl.oclc.org/NET/UASO/qoi .

2.3.2. Stream Annotation Ontology

Representing IoT data streams is an important requirement in semantic stream data applications, as well as in knowledge-based environments for Smart Cities. This study aims to semantically represent the features of a data stream defining the specifications of an information model on top of Semantic Sensor Networks (SSN), PROV-O and TimeLine Ontologies, and involves connections with the Complex Event Processing Ontology and Quality Ontology[8].

SAO is used to express temporal features, e.g. segments and window size, as well as data analysis features, e.g. FFT, DFT and SAX of a data stream. It provides concepts such as sao:StreamData, sao:Segment, sao:StreamAnalysis on top of the TimeLine ontology and the IoTest model, allowing the publication of content-derived data about IoT streams. SAO extends the ssn:Observation concept with sao:StreamData to annotate sensory observations and provides a link to sao:Segment to describe temporal features in a granular detail for each data segment. Each data point and segment are also linked to the sao:StreamAnalysis concept that includes the name and parameters of the method that has been used to analyse the data stream.

[7] https://www.w3.org/TR/prov-o/
[8] https://mobcom.ecs.hs-osnabrueck.de/cp_quality/

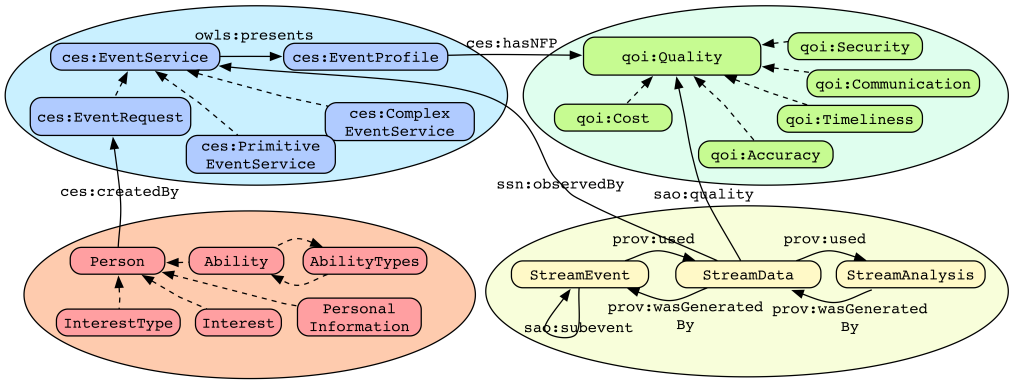


Figure 15: CityPulse Information Model

Further information about SAO can be found here:
http://iot.ee.surrey.ac.uk/citypulse/ontologies/sao/sao

2.3.3. Extracting City Traffic Events from Social Streams

This tool has two major components: (1) Annotator and (2) Extractor. *Annotator*: Sequence labeling model trained with declarative knowledge from location and event knowledge base is utilized for annotation of raw tweets. Open Street Maps is used as a location based knowledge specific to a city and 511.org schema of events is used as a knowledge of traffic related events.

The proposed Twitter Annotator component is composed of a dedicated data wrapper unit (in PhP) which connects to the Twitter stream API and Google translate API and simultaneously collects data in the form of tweets and automatically detects the source language and translates the tweets to English, facilitating the following data processing step; Natural Language Processing (NLP) unit (in Python). This unit is composed of three sub-modules: a Conditional Random Field (CRF) Name Entity Recognition, a Convolutional Neural Network (CNN) Part of Speech tagging, and a multi-view event annotation which combines the information extracted from the previous sub- modules to extract a sentence-level inference for event classification. Given a tweet and its translation, the NLP processing unit assigns it to one of the event classes from the pre-defined class set: Transportation and Traffic, Weather, Cultural Event, Social Event, Sport and Activity, Health and Safety, Crime and Law, Food and Drink.

Extractor: Extraction algorithms use space, time and theme characteristic of city events to aggregate all the tags for emitting events. Additionally, a terminology catalogue is defined to specify the traffic event levels. The acquired event level facilitates the integration of social stream events with traffic sensor extracted ones.

Further information about these two components can be found here: https://osf.io/b4q2t/wiki/home/

2.3.4. Linked Sensor Middleware (LSM)

Sensing devices are becoming the source of a large portion of the Web data. To facilitate the integration of sensed data with data from other sources, both sensor stream sources and data are being enriched with semantic descriptions, creating Linked Stream Data. Despite its enormous potential, little has been done to explore Linked Stream Data. One of the main characteristics of such data is its “live” nature, which prohibits existing Linked Data technologies to be applied directly.

Moreover, there is currently a lack of tools to facilitate publishing Linked Stream Data and making it available to other applications.
To address these issues we have developed the Linked Sensor Middleware (LSM), a platform that brings together the live real world sensed data and the Semantic Web. It provides many functionalities such as: i) wrappers for real time data collection and publishing; ii) a web interface for data annotation and visualisation; and iii) a SPARQL endpoint for querying unified Linked Stream Data and Linked Data. The architecture of LSM is depicted in the following Figure 16:

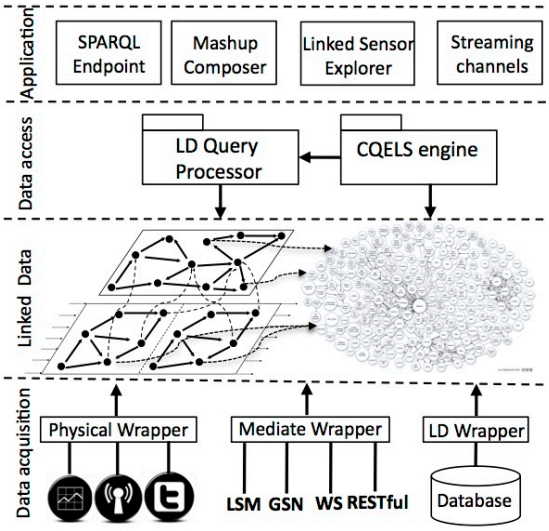


Figure 16: LSM Architecture

Further information and code related to LSM can be found here:
https://code.google.com/archive/p/deri-lsm/

2.3.5. ODAA – Open Data Aarhus

The overall objective of http://www.odaa.dk/ is to make data openly and freely available to support productivity and innovation in the development of greater use of data. Developers, entrepreneurs, companies, institutions, citizens and others will be able to easily access the open data and turn them into new services/applications.

ODAA currently provides access to 135 different datasets related to the city of Aarhus in the areas of Transportation and Infrastructure, Environment and Nature, Geography, Culture, Administration, etc.

CityPulse uses the traffic, parking, weather and cultural events datasets from ODAA as data sources that provide relevant information for different use case scenarios/applications that are described in the section Applications.

2.3.6. Ckanext-realtime

ckanext-realtime is the latest contributions to Free Software from Gatesense team. It is an extension for CKAN open data platform which enables app developers to make realtime applications with CKAN. More specifically, you can subscribe to resources (datastores) and thus monitor their changes in realtime. gatesense.com is a realtime-enabled CKAN portal, among other things. This means that you can build realtime apps with datasets found at http://gatesense.com/data/dataset .A tutorial on how to use ckanext-realtime can be found here: http://alexandrainst.github.io/ckanext-realtime/tutorial.html

2.3.7. The SSN Ontology Validation Service

This tools allows to validate an ontology against the W3C SSN Ontology[9] and also provides a tag cloud of the most common concepts from the SSN ontology that are used in the validated ontologies.

[9] https://www.w3.org/2005/Incubator/ssn/XGR-ssn-20110628/

2.4. Applications

This section describes in detail each of the applications that have been developed for demonstration, including the integration with the CityPulse framework, brief tutorials on how to use the application, system requirements and finally how to run the application. The information has been extracted from the applications' repositories on GitHub.

2.4.1. 3D Map

This application has been developed with the core goal to provide a 3D visualisation and experience to the users. By using it the users can "fly" around this 3D model of a city and visualise the effect of real-time data in the model. The following Figure 17 shows an example of the visualisations the user can experience:

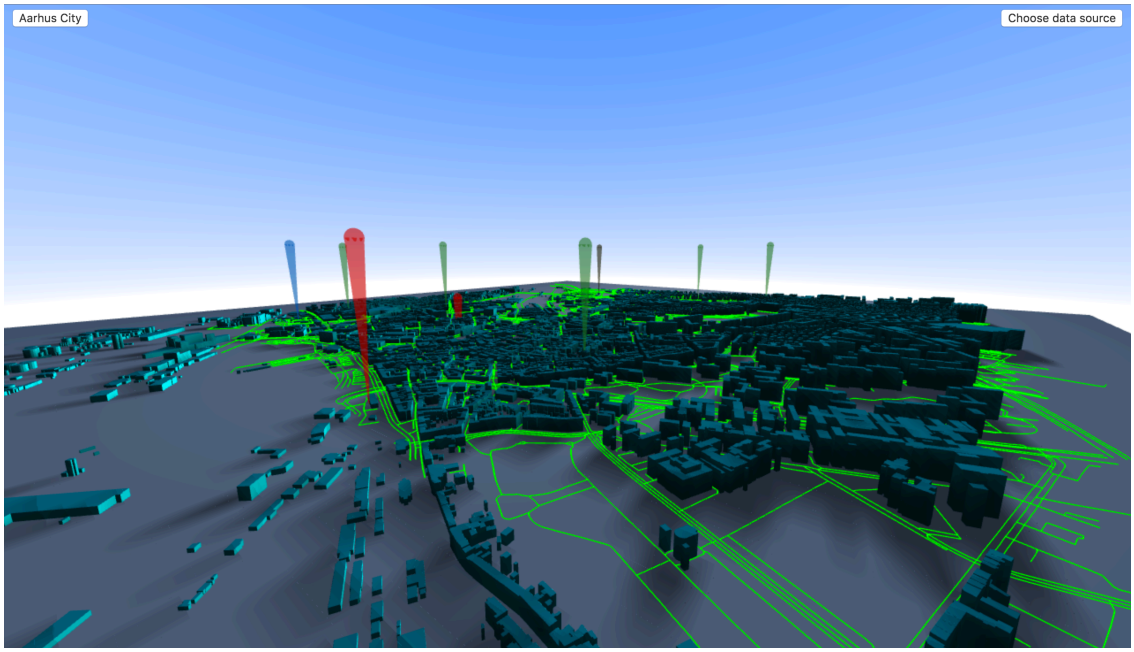


Figure 17: 3D Map GUI

When starting the application the user can select on the top-left corner from different cities to be visualised (currently Danish cities of Aarhus, Copenhagen, Odense and Ry). After loading the model the user can on the top-right corner select the different data sources to be visualised in the map. The current version of the map is supporting visualisation of events of type parking, traffic, noise and pollution for the city of Aarhus. These events are detected by the CityPulse middleware in the background and then further displayed in the map in form of geo-localised pins.

The map is also supporting visualisation of energy consumption in buildings in the city, where buildings' heights' are changing according to the current energy consumption levels. However this visualisation is currently only a demo, since we do not have yet access to the real consumption data in buildings for the city of Aarhus.

This application has been conceptualised and developed to be modular, meaning that the user can add new cities and/or new data sources in a very simple way.

The current elements the map renders are buildings, roads, waterfronts and trees. When initialising the visualisation of a city, the application takes a KML file for each of the map elements as input for visualisation. This means that if the user wants to add new models of cities, he/she needs to provide those data in form of KML file (you can browse under the folder “data/kml/” for the existing models).

2.4.1.1. Integration with CityPulse framework

In order to have access to the real-time events information for the city of Aarhus the 3D map is connecting to the CityPulse message bus, to which it subscribes to messages of type “event”. This is done in the amqpclientcallback.js file under the “CityPulseIntegration” folder. After subscribing to this type of events the message bus client will be listening to incoming messages from the message bus. Moreover, and in order to be modular this component is providing a websocket connection to where the 3D map is connecting to in order to receive the updated information and render it. The following Figure 18 depicts the overall architecture:

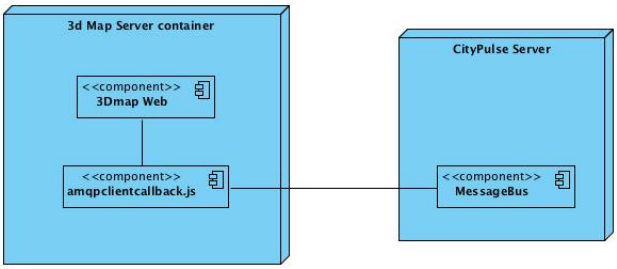


Figure 18: 3D Map application Deployment Diagram

The flow of interactions between the involved components is the following:

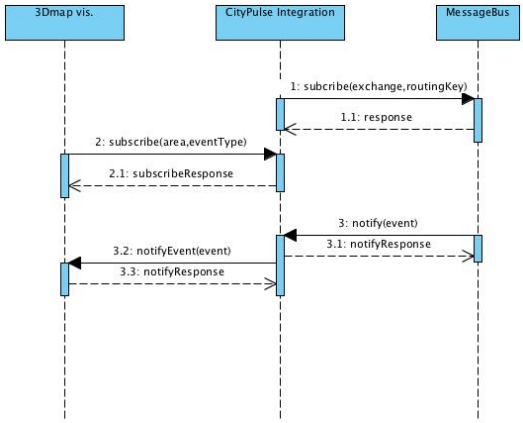


Figure 19: 3D Map - Sequence Diagram

2.4.1.2. System requirements

In order to be able to run the application the user will need to have installed the following software:

- Web server container e.g. Apache, JBoss, etc
- Nodejs installed with the following APIs:
  - Websocket: https://www.npmjs.com/package/websocket
  - amqplib: https://www.npmjs.com/package/amqplib
  - n3: https://www.npmjs.com/package/n3

2.4.1.3. Running the application

To run the application the user needs to follow the following steps:

- Deploy the 3D map into a web server container
- Open a terminal and browse the folder “CityPulseIntegration”. Run the amqpclientcallback.js by using the command

```
$ node amqpclientcallback.js
```

- Open your browser and browse e.g. http://localhost/3dmap-master/3dmap-master/index.html

2.4.1.4. Link

The code for the 3D Map can be found at: https://github.com/CityPulse/3DMap

2.4.2. City Dashboard

The CityPulse framework provides immediate and intuitive visual access to the results of its intelligent processing and manipulation of data and events. The ability to record and store historical (cleaned and summarized) data for post-processing makes it possible to analyse the status of the city not only on the go but also at any point in time, enabling diagnosing and “post mortem” analysis of any incidents or relevant situation that might have occurred. To facilitate that, a dashboard for visualising the dynamic data of the smart cities is provided on top of the CityPulse framework. Based on this dashboard, the user has the possibility to visualise a holistic and summarised view of data across multiple contexts or a detailed view of data of interest, as well as to monitor the city life as it evolves and as things happens. The investigation of past city events or incidents can be conducted from different perspectives, e.g. by observing the correlations between various streams, since the streaming data is stored in the framework for a period of time which can be configured, and it can be retrieved for visualisation and analysis at any moment. Figure 20 depicts a screenshot of the CityPulse dashboard application. Here we can see that the user has decided to display with red the average speed and with green and blue the number of cars, respective the total number of parking slots, from a garage. On the map the diameter of circles is changing in real time and it is proportional with the measured value. The graphs present the evolution of these data sources.

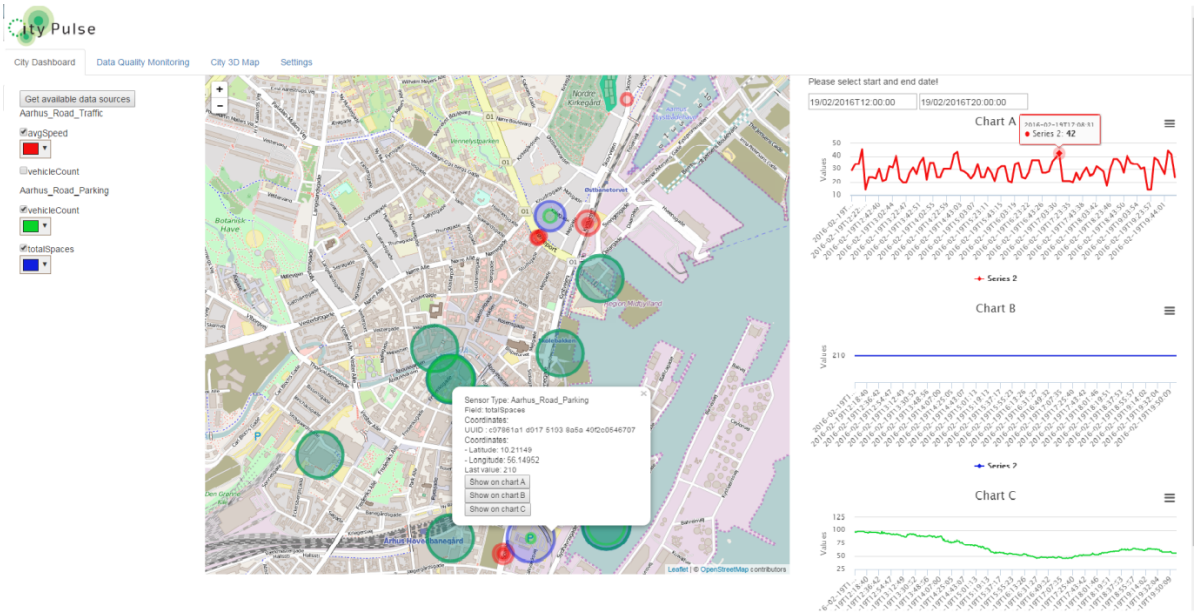


Figure 20: CityDashboard GUI

In order to display the status of the city, the dashboard application connects directly to the resource management or to the data bus for fetching the description of the available streams or the real-time/historic observations. The dashboard application can be used out of the box and there are no configuration or development steps that have to be done by the application developer.

Figure 21 depicts the workflow executed by the CityPulse components when the user wants to visualize the real-time or historic data.

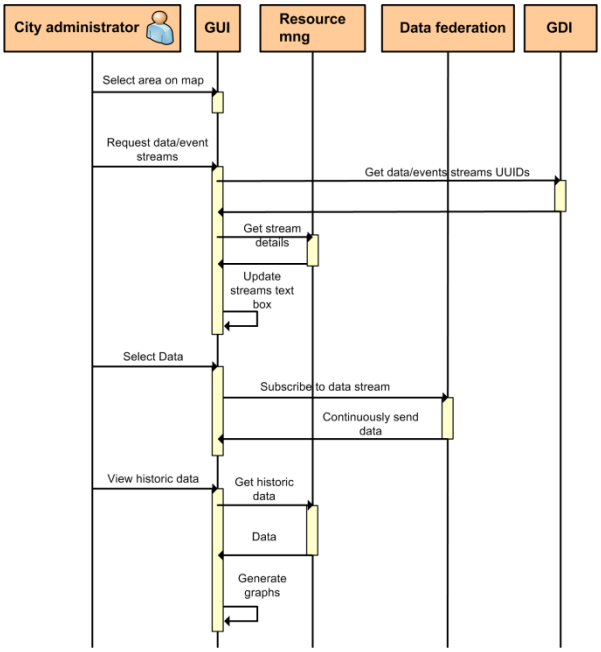


Figure 21: CityPulse dashboard workflow

The CityPulse dashboard workflow steps are briefly described below:

- First, the user has to select an area of interest on the map;

- When the user clicks the *Get available data sources* button, a request is made to the geospatial database, which contains the GPS coordinates of the area selected by the user (a list containing all the available sensors and their UUIDs is received);
- The user selects the data sources which he wants to display on the map (and also pick a colour for their representation). Now a new request is generated to the data federation component in order to receive in real-time the selected data sources observations;
- After the data federation sends back the responses, the map will be populated with markers for each sensor. The size of the marker is proportional to the observation value and the colour is the one selected by the user previously;
- If a sensor on the map is selected, a popup will appear that will contain the description of the sensor (its type, field name, coordinates and last value);
- The user then has three options of displaying the historical data: Show on chart A, B or C from the left side of the dashboard;
- If one of these 3 options is selected, a request to the resource management will be made for obtaining the historical data for the time period specified in the upper region of the left section of the dashboard;
- At the end, a graph is going to be created and displayed on the right part of the GUI.

2.4.2.1. Other CityPulse framework dependencies

The CityPulse dashboard relies on the following framework components:

- Resource management for acquiring the stream description.
- Geospatial data infrastructure for determining the id of the streams located within a certain area.
- Data federation retrieving real time observations from the streams.

2.4.2.2. Component deployment steps and Configuration file

In order to be able to run the application the user will need to have installed an application server like Tomcat or JBoss. The user has only to copy the war file of the backend application and the content of the frontend one in the deployment folder of the server. The backend and the front end application can be found here: https://github.com/CityPulse/CityDashboard .

2.4.3. Travel Planner application

In order to demonstrate how the CityPulse framework can be used to develop applications for smart cities and citizens, we have implemented a context-aware real time Travel Planner using the live data from the city of Aarhus, Denmark. This scenario aims to provide travel-planning solutions, which go beyond the state of the art solutions by allowing users to provide multi-dimensional requirements and preferences such as air quality, traffic conditions and parking availability. In this way the users receive parking and route recommendations based on the current context of the city. In addition to this, Travel Planner continuously monitors the user context and events detected on the planned route. User will be prompted to opt for a detour if the real time conditions on the planned journey do not meet the user specified criteria anymore. All the CityPulse framework components are deployed on a back-end server and are accessible via a set of APIs. As a result of that the application developer has only to develop a user-friendly front-end application, which calls the framework APIs. In our case we have developed an Android application.

2.4.3.1. Application UI

Figure 22 depicts the user interfaces used by the end user to set the travel preferences and the destination point.

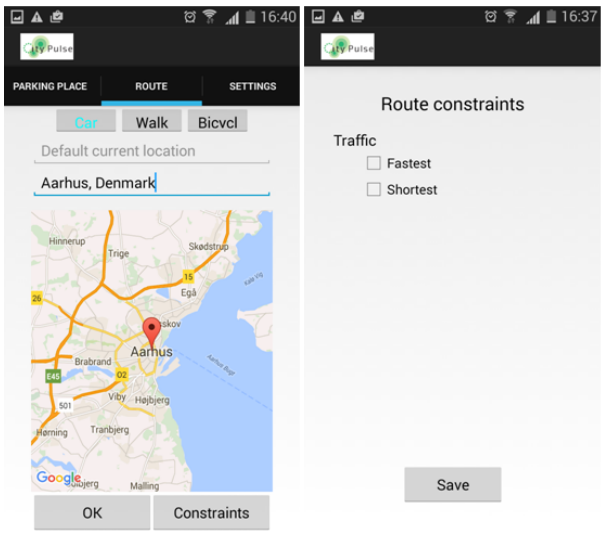


Figure 22: Travel Planner GUI

After the user has filled in the details and made the request using the user interface, the mobile application generates the appropriate request for the Decision support component which has the following main fields:

- Type: indicating what decision support module is to be used for this application ("TRAVEL-PLANNER" in this case);
- Functional details: specifying possible values of user's requirements, including:
  - Functional parameters: mandatory information that the user provides such as starting and ending locations, starting date and time, and transportation type (car, bicycle, or walk).
  - Functional constraints: numerical thresholds for cost of a trip, distance, or travel time.
  - Functional preferences: the user can specify his preferences along selected routes, which hold the functional constraints. These preferences can be the minimization or the maximisation of travel time or distance.

The functional constraints and preferences specify different thresholds and minimization criteria for electing the route. During the development of the mobile application the domain expert has computed a default set of values for these thresholds. As a result of that, the route constraints user interface from Figure 23 b) allows the user to select between the fastest/shortest routes. If needed, more fields can be added in this user interface in order to allow more fine-grained constraints specification, but the usability of the application may suffer.

Figure 23 a) depicts the user interface where the routes computed by the Decision support are displayed to the end user.

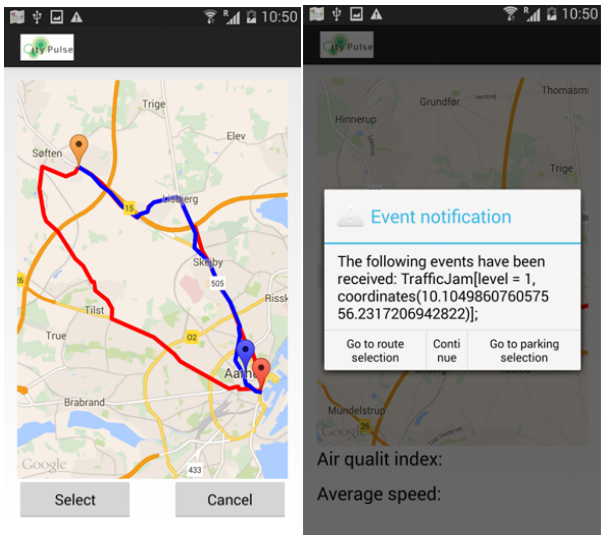


Figure 23: Travel Planner - Route suggestion

After the user selects the preferred route, a request is generated to the Contextual filtering component in order to identify the relevant events for the use while he/she is traveling. The request includes the following properties:

- Route of interest: the current route of the user
- Filtering Factors: used to filter unrelated (unwanted) events out, and include event's source, event's category, and user's activity, as specified in a user-context ontology. For this particular scenario, the activities included in our ontology include CarCommute (user is traveling by a car), or Walk, or BikeCommute (user is traveling by a bike).
- Ranking Factor: identifies which metric is preferred by the user for ranking the criticality of incoming events. We have currently implemented the Ranking Factor based on two metrics: distance, and gravity of the event. In order to combine these two metrics, we use the linear combination approach, where the user can identify weights (or importance) for each metric.

Once the user has selected one of the routes computed by the Decision support component, the Contextual Filtering component sends a request to the Geospatial Database Infrastructure component to obtain the description of the event streams available on the selected route, as registered at design time by the Event detection component. The Contextual Filtering component uses these descriptions to subscribe to detected events via a Data Bus. In addition, the Contextual Filtering also receives the contextual information of the user (currently including location) from the user/application as a stream. Whenever there is a new event detected by the Event detection component, the Contextual Filtering component filters and assigns the most appropriate criticality (0 if not critical, from 1 to 5 if it is critical) to the new event. If the new event is marked as critical, the user receives a notification and he/she has the option to change the current solution and request a new one or ignore the event.

Figure 23 b) depicts the notification received by the end user, while s/he is traveling and a traffic event is detected on his/hers route.

2.4.3.2. Dependencies

In order to allow the application developer to use Google Maps services, he/she will need to follow the steps:

- Import the project in his/hers preferred IDE.
- Edit the AndroidManifest.xml file al line 53(the key to be more exact)

- In order to obtain the new keys, we will need to get the SHA1 key from his IDE and go to https://console.developers.google.com/home/dashboard to get a new key. This will allow him to run the application while still connected via USB cable to the computer.

The application developer also has to edit the DefaultValues.java . He will need to adapt to his scenario the Decision Support, Contextual Filtering and Data Federation websocket URLs.(lines 19,23,30 in DefaultValues.java)

2.4.3.3. Link

The code of the Real-time Travel Planner can be found at: https://github.com/CityPulse/CityPulse-enabled-application-CONTEXT-AWARE-REAL-TIME-TRAVEL-PLANNER

2.4.4. Twitter Data Map

A web interface as shown in Figure 24 is developed which facilitates the visualisation the social streaming extracted city events on a Google map in near real-time. The interface is composed of (a) Google map canvas layer on which the processed and annotated Tweets are displayed with their class-identical icons (b) a live London traffic layer from google traffic API - code coloured paths on the map (c) a bar chart panel which presents the class distribution histogram of daily Tweets and (d) a panel for displaying Twitter timeline. (note: The map interface code is not included in this package)

The map data is updating in 60s time windows by adding the past minute's Tweets to existing ones up to a 60-minutes time window. In practice, the whole data will be updated on hourly bases. Clicking on each event a dialogue box is shown on the map which reveals the underlying Tweet content along with its time-stamp. The twitter user id and the name are anonymised for privacy purpose.

The web interface utilises JavaScript and html coding and reads the annotated data from a CSV rest file of the live Social Stream processing component.

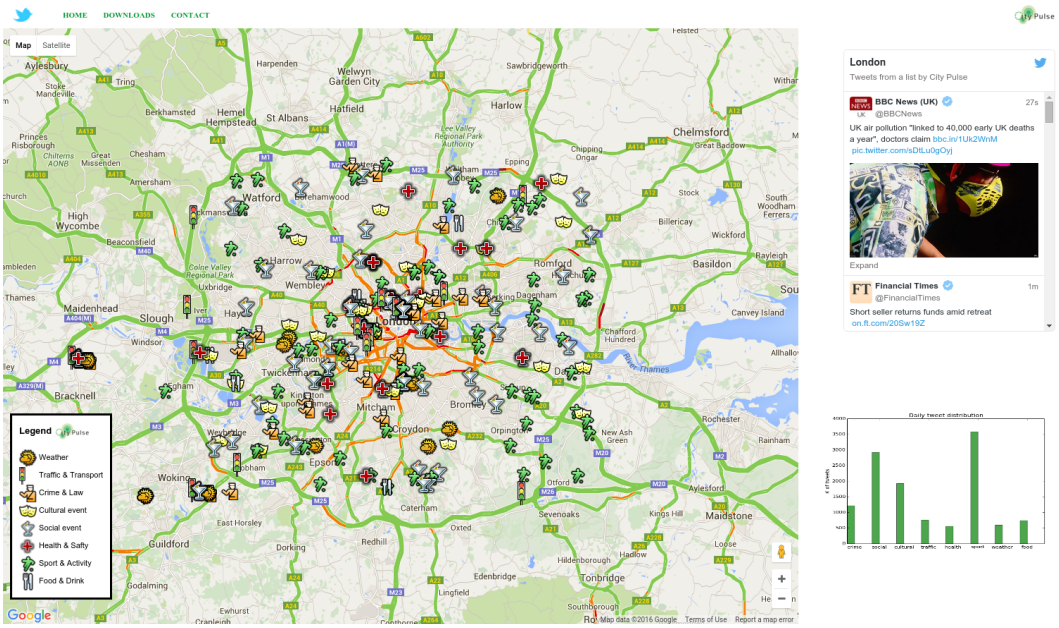


Figure 24: Twitter Data Web interface

2.4.4.1. System requirements

- Java compiler
- C++ compiler
- Python 2.7 or higher
- mysql
- php
- mysql php connector

2.4.4.2. Dependencies

Data collection unit requires a mysql database of the following schema to be constructed prior to collection:

```
'''CREATE TABLE `AarhusTweet` (
            `twitterstream` VARCHAR(100) NOT NULL,
            `userid` VARCHAR(100) NOT NULL,
            `text` VARCHAR(500) NOT NULL,
            `time` VARCHAR(100) NOT NULL,
            `lat` VARCHAR(100) NOT NULL,
            `long` VARCHAR(100) NOT NULL,
            `boundingbox` VARCHAR(400) NOT NULL,
           ) ENGINE=MyISAM DEFAULT CHARSET=latin1;
           '''
```

The package needs the senna C++ Convolutional Neural Network package to be installed in the main directory from: http://ronan.collobert.com/senna/

It also requires Conditional random field NER tagger which can be downloaded from: http://personal.ee.surrey.ac.uk/Personal/N.Farajidavar/Downloads.html

Python package dependencies: Pandas, mysql, matplotlib, pylab, geopy, goslate, pika, rabbitmq

2.4.4.3. Running/Usage

Once the dependencies mentioned above are installed, need to run the main code in Python:

```
$ python Aarhus_v4.py
```

2.4.4.4. Link

The code of the Twitter analysis package can be found at: https://github.com/CityPulse/Twitter

2.4.5. Quality Explorer

The QualityExplorer is a web-based tool to get detailed insight about the quality of information the deploy sensors provide. Each sensor is depicted on a map as a dot. The color of the dot indicates the quality of information, ranging from green (good) to red (bad). Additionally bar charts show the number of visible sensors and the corresponding quality of information value.

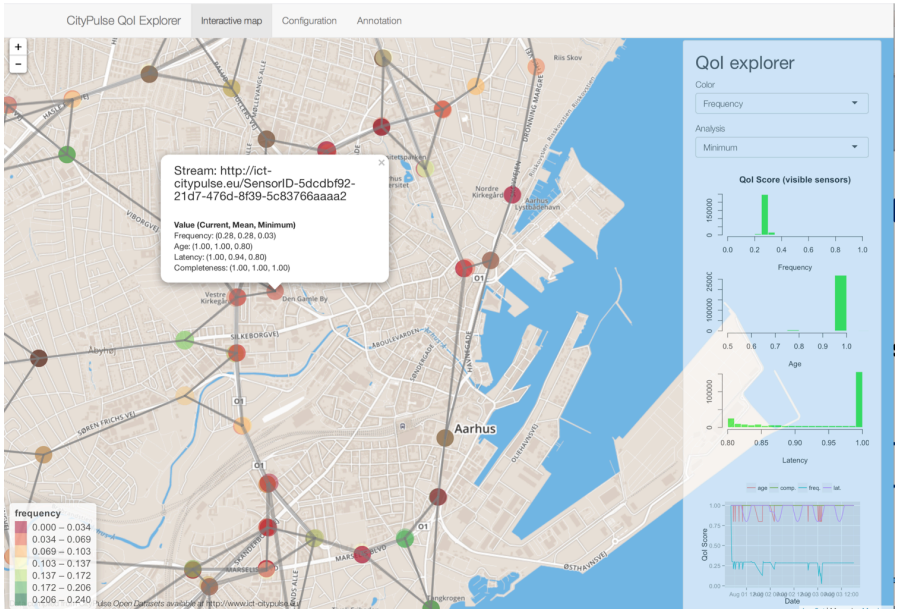


Figure 25: QoI Explorer GUI

2.4.5.1. Dependencies

The following applications/libraries have to be installed (Name/License/Link): R (Version>=3.1)

Shiny	GPL-3	https://cran.r-project.org/web/packages/shiny/
Leaflet	GPL-3	https://cran.r-project.org/web/packages/leaflet/
rgeos	GPL-3	https://cran.r-project.org/web/packages/rgeos/
sp	GPL-3	https://cran.r-project.org/web/packages/sp/
maptools	GPL-3	https://cran.r-project.org/web/packages/maptools/
shinyBS	GPL-3	https://cran.r-project.org/web/packages/shinyBS/
ggmap	GPL-2	https://cran.r-project.org/web/packages/ggmap/
rgdal	GPL-3	https://cran.r-project.org/web/packages/rgdal/
cleangeo	GPL-3	https://cran.r-project.org/web/packages/cleangeo/
rjson	GPL-2	https://cran.r-project.org/web/packages/rjson/
ggplot2	GPL-2	https://cran.r-project.org/web/packages/ggplot2/
lattice	GPL-3	https://cran.r-project.org/web/packages/lattice/
scales	MIT	https://cran.r-project.org/web/packages/scales/
RColorBrewer	Apache 2.0	https://cran.r-project.org/web/packages/RColorBrewer/
xtable	GPL-2	https://cran.r-project.org/web/packages/xtable/
knitr	GPL-2	https://cran.r-project.org/web/packages/knitr/
gdata	GPL-2	https://cran.r-project.org/web/packages/gdata/
RCurl	BSD	https://cran.r-project.org/web/packages/RCurl/
dplyr	MIT	https://cran.r-project.org/web/packages/dplyr/
plyr	MIT	https://cran.r-project.org/web/packages/plyr/
GDAL	X/MIT	http://www.gdal.org
GEOS	LGPL	http://tsusiatsoftware.net/jts/main.html

Table 17: QoI Dependencies

2.4.5.2. Installation

Apart from R and the required libraries above, an installation of the QualityExplorer is not required. Simply clone the Github repository.

```
git clone https://github.com/CityPulse/QualityExplorer.git
```

To start the component change into the 'R-Shiny-Application' folder and execute the following start command:

```
R -e "shiny::runApp(host='127.0.0.1', port=10111)"
```

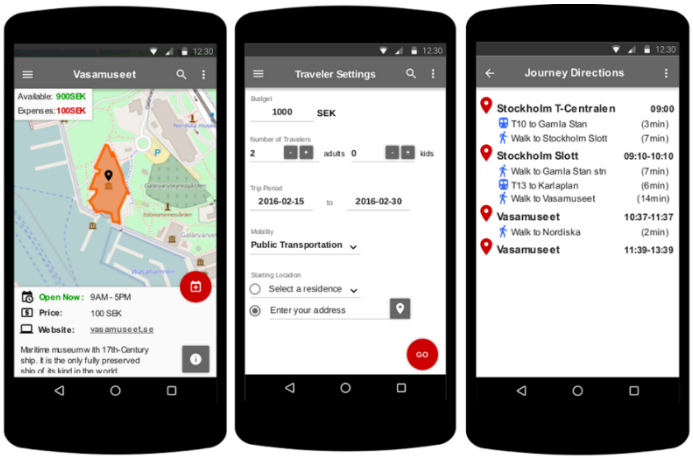
This starts the server listening on localhost (127.0.0.1) at port 10111. Change the 'host' and 'port' parameter in the start command according to your needs.

2.4.5.3. Link

The code of the Quality Explorer can be found here: https://github.com/CityPulse/QualityExplorer.git

2.4.6. Tourism Scheduler

This project combines different existing sources of data related to events and points of interest (POIs) in the city of Stockholm. The objective is to generate a schedule to explore the POIs that the users select based on their visit times, budget constraints, and transportation. The front-end thus consists of a mobile application that shows a map of Stockholm (Figure 26.a) and allows the users to enter information about their respective trips (Figure 26.b). These parameters are later used to generate a schedule (Figure 26.c) based on the opening times of the PIs, the time and distance between every pair of POIs, and other criteria. Note that this project is related to the scenario 18 from the 101 scenarios of CityPulse (available at http://www.ict-citypulse.eu/scenarios/).



(a) **(b)** **(c)**

Figure 26: Design prototype of the Tourism Scheduler application

2.4.6.1. System Architecture & Integration with CityPulse

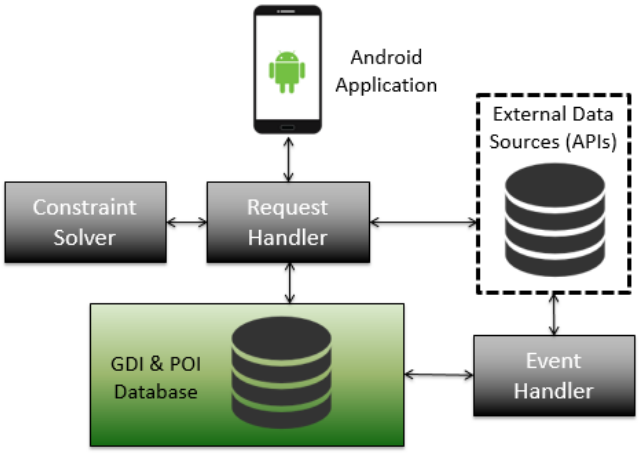


Figure 27: System architecture of the Tourism Scheduler

Figure 27 summarizes the structure of the system. The Android application communicates with the backend through the Request Handler, which gathers all the information needed from the rest of the components and sends the response back to the application. The CityPulse components provide the best route segments between every two POIs by accessing the GDI component. The data is then used by the constraint programming solver to generate the schedule (i.e. determine the visit time for each POI) based on the constraints that were specified by the user in the search request. Figure 28 summarizes this communication in a sequence diagram.

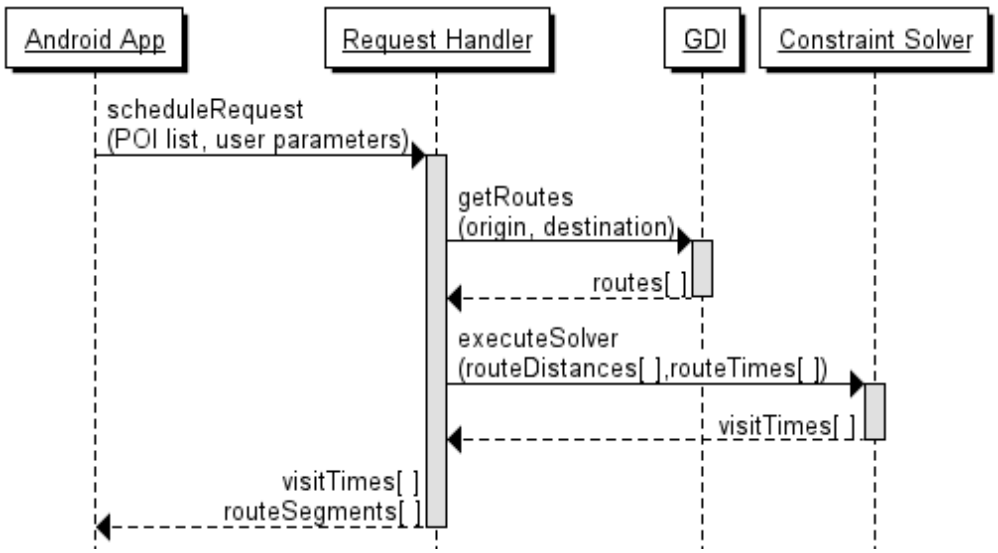


Figure 28: Sequence diagram to generate a schedule

2.4.6.2. System requirements

The following steps and prerequisites are necessary for the system to be operational:

- Mobile application:
  - Android 5.0 Lollipop (API21) or higher
  - Install & enable Google Play Services
  - Enable location services
- Host the Request Handler on a Linux machine
- Install a C++ compiler and JRE
- Compile and install the Gecode library (http://www.gecode.org/download.html)

2.4.6.3. Link

This application is currently being integrated with the CityPulse framework and the code is available on CityPulse's GitHub: https://github.com/CityPulse/TouristScheduler

2.4.7. Pick-up Planner

This use case aims to develop a pick-up planner that can be used by fleet companies to provide travel services to the elderly, who subscribe to the services through a smartphone application. The system presents travel recommendations based on user profiles and travel requests, containing trips that users can join. As depicted in the Pickup Planner vehicle application in Figure 29, a traveling route is presented as a trajectory of user pickup locations that the vehicle should visit in order to transport the passengers to their intended destination(s).

Two components (Decision Support and GDI) from the CityPulse framework are used to illustrate how the framework can be used to develop applications that provide useful service to smart city dwellers.

Note that this use case directly addresses the scenario 42 and is related to the scenarios 19 and 32 from the 101 scenarios of CityPulse (available at http://www.ict-citypulse.eu/scenarios/).

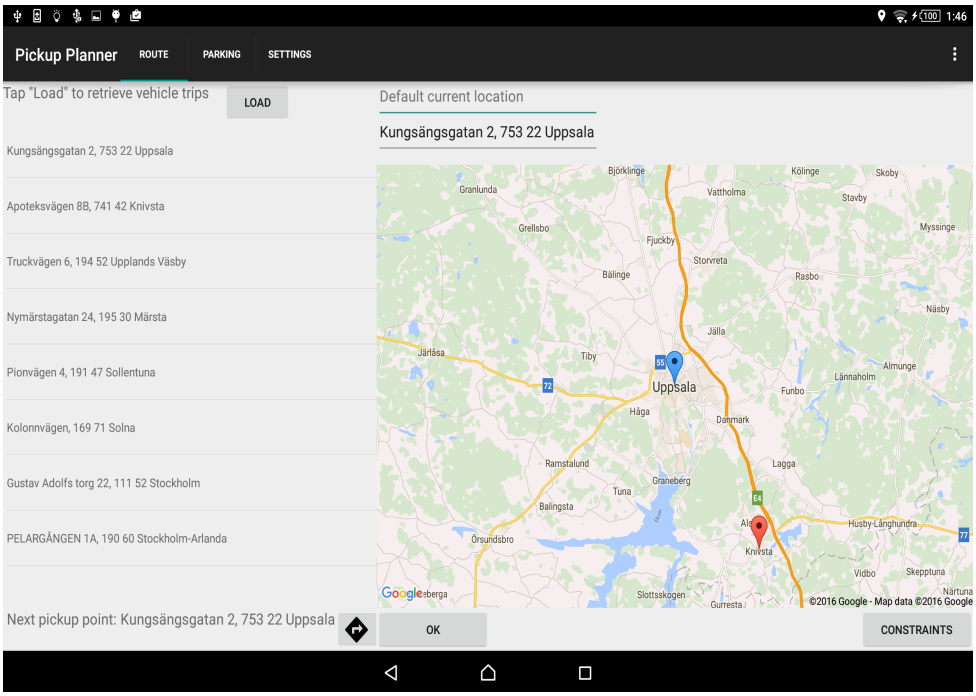


Figure 29: A screenshot of the Pickup Planner vehicle application

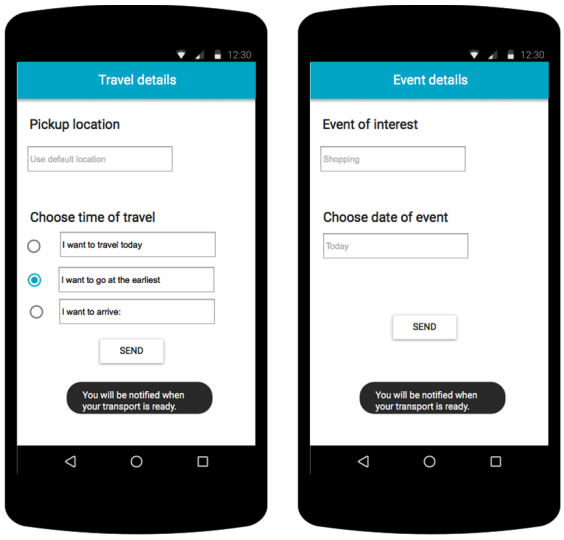


Figure 30: A screenshot of some of the functionalities of the PickUp Planner client application

2.4.7.1. System Architecture & Integration with CityPulse

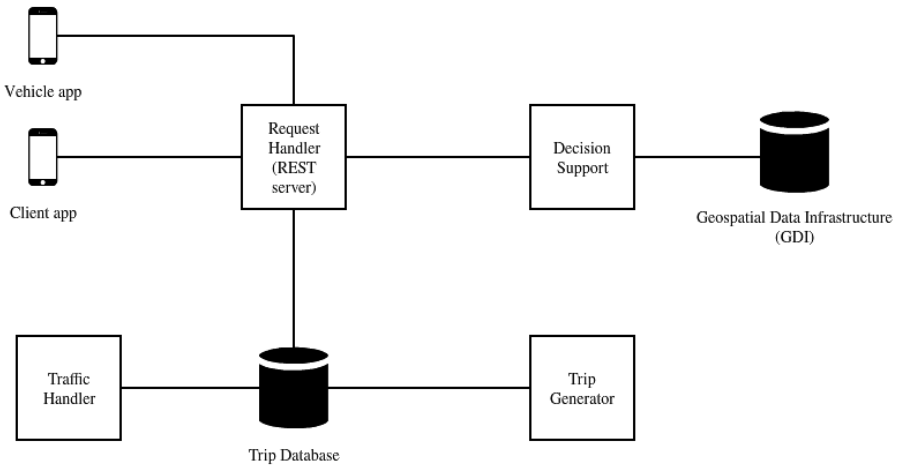


Figure 31: System architecture of the PickUp Planner.

As shown in Figure 31, the system consists of a Client application (see Figure 30) and a Vehicle application. The Client application provides an interface through which users can register for the pickup service and fill in their profiles. A Trip Generator runs in the background to group/cluster travel requests and store user trips in the Trip database. The Vehicle application requests, through CityPulse's Decision Support, the best route to service the user trips (given as an ordered list of user locations). The travel path is visualized on the Vehicle application including additional indicators for the next traveler to pick up and the current vehicle location. The Traffic Handler component is a server that receives traffic data coming from the road infrastructure and from the vehicles.

2.4.7.2. System requirements

The following steps and prerequisites are necessary for the system to be operational:

- Mobile application:
  - Android 5.0 Lollipop (API21) or higher
  - Google Play Services
  - Android Location services enabled
- Flask
- Clingo4
- Psycogp2 and PostgreSQL database (any database to persist user requests and profiles)
- Networkx
- Sklearn, numpy

2.4.7.3. Link

This application is currently being integrated with the CityPulse framework and the code is available on CityPulse's GitHub: https://github.com/CityPulse/pickup-planner

2.4.8. Dynamic Bus Scheduler

The Dynamic Bus Scheduler is an application focused on generating dynamic bus schedules, while making predictions regarding the needs of passengers. It takes into consideration parameters that affect the normal schedule, such as the levels of traffic density on the bus routes.

Note that this application is related to the scenarios 19, 32, 51 and 58 from the 101 scenarios of CityPulse (available at http://www.ict-citypulse.eu/scenarios/).

2.4.8.1. System Architecture & Integration with CityPulse

As illustrated in Figure 33, the application is consisted of three major components, the Route Generator, the Look Ahead, and the Traffic Data Parser that is taking data from the CityPulse Data Bus. The Dynamic Bus scheduler also includes a Data Simulator, a MongoDB database as well as an OSM Parser. All the components are explained in the following text.

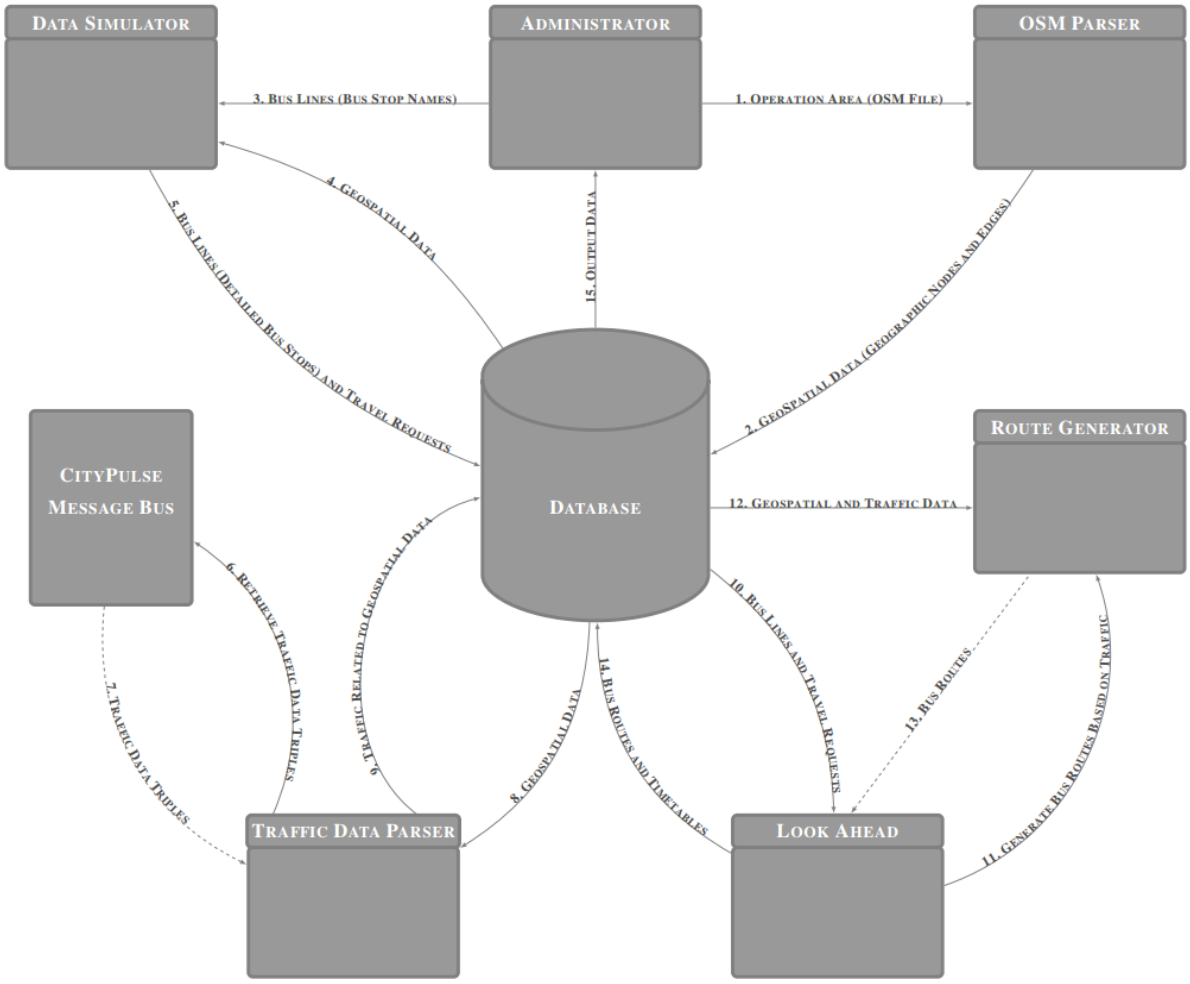


Figure 32: System architecture of the Dynamic Bus Scheduler

The System Administrator provides the OpenStreetMap file, which includes information about the road network and the infrastructure of the selected area, to the OSM Parser. The OSM Parser processes the OpenStreetMap file and parses the geographic nodes of the selected area, such as bus stops, buildings, or geographic points of roads, as well as the edges, which connect them. The extracted geospatial data is stored at the corresponding collections of the Mongo Database. The System Administrator provides the bus lines of the system, as lists of bus stop names, to the Data Simulator. The Data Simulator corresponds the provided bus lines to the retrieved bus stops and generates travel requests for the bus lines.

The Traffic Data Parser connects to the Data Bus in order to retrieve the triples, which include traffic data of the operation area. It also identifies the relation between the traffic data triples and the geospatial data (e.g., estimates the traffic density of the edges, which connect the geographic points between the bus stops of the operation area). The result is stored in the Database.

The Look Ahead connects to the Database to retrieve the bus lines and the travel requests that were produced by the Data Simulator. The Look Ahead sends a request to the Route Generator in order to generate routes for the bus lines, while taking into consideration the current levels of traffic density. The Route generator implements an A* algorithm that identifies the less time-consuming routes (using the geospatial and traffic data) connecting the bus lines of the system, and forwards them to the Look Ahead. The Look Ahead component implements a genetic algorithm for generating timetables for selected routes. In more details, it evaluates the travel requests, while taking into consideration parameters of the bus routes such as the required traveling time, and generates the timetables of the system. The bus routes and the timetables are stored in the Database.

The Dynamic Bus Scheduler is integrated with the CityPulse framework through the CityPulse Data Bus, which provides traffic data triples. Since the pre-scheduled bus routes could be affected by the levels of traffic in the city, the Dynamic Bus scheduler updates the timetables and the routes accordingly.

In conclusion, the Dynamic Bus Scheduler is intended to be used by bus operators in order to dynamically generate timetables that decrease the waiting times for passengers and include adjustments produced by traffic incidents.

2.4.8.2. Link

This application is currently being integrated with the CityPulse framework and the code is available on CityPulse's GitHub: https://github.com/CityPulse/dynamic-bus-scheduling

2.4.9. Transportation Planner

The Transportation Planner is an application focused on generating dynamic bus schedules, while making predictions regarding the needs of passengers which are expected to use the application, and taking into consideration parameters that affect the normal schedule, such as the levels of traffic density.

2.4.9.1. System Architecture & Integration with CityPulse

As illustrated in Figure 33, the application is consisted of three major components, the Route Generator, the Look Ahead, and the Decision Support component of the CityPulse framework, while also including a simulator for generating test traffic data and travel requests, and a MongoDB database.

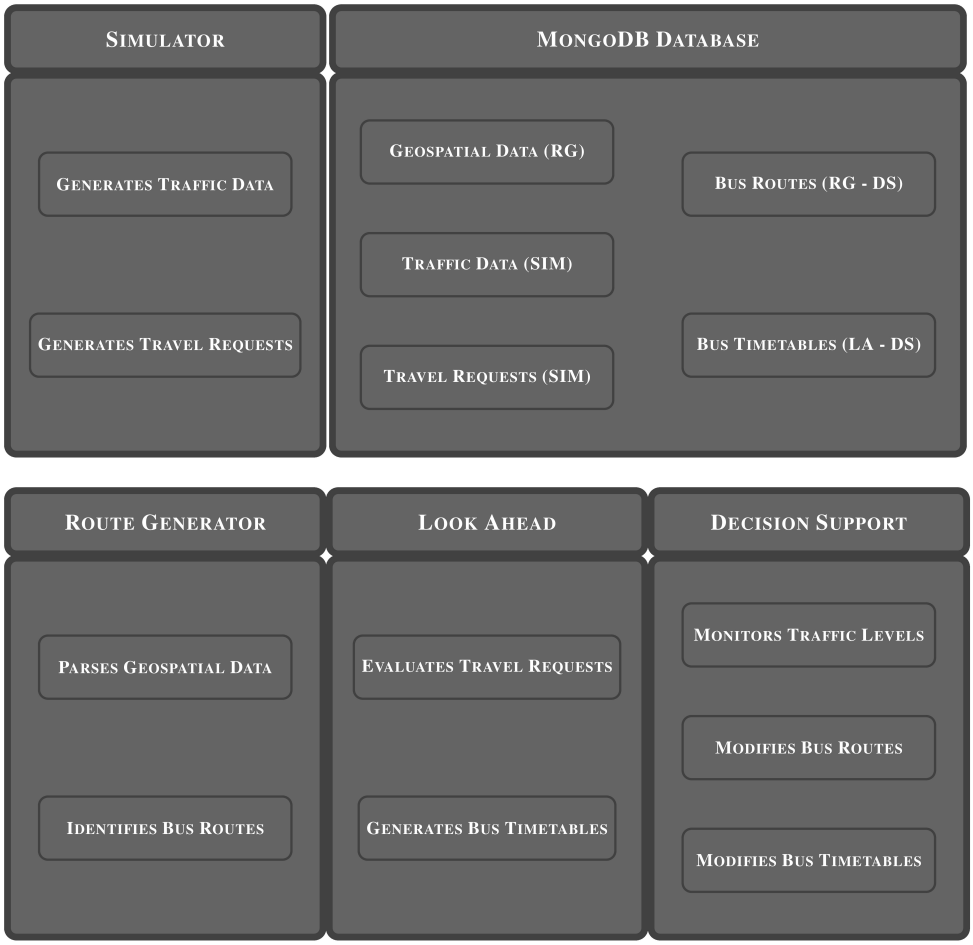


Figure 33: System architecture of the Transportation Planner

The Route Generator is a web-based server responsible for providing responses to requests related to the road network of selected areas (e.g., cities or countries). More precisely, it utilizes the OpenStreetMap API in order to extract geospatial data from OSM files, and represent the road network of provided areas. Taking advantage of the Route Generator, a user has the opportunity to receive information concerning possible routes which connect the bus stops of selected areas, since the Route Generator is capable of identifying them while considering factors like the distance of intermediate waypoints, required travel time, type of covered roads, and traffic density. Apart from the OpenStreetMap parser and the web server functionality, the Route Generator is also integrated with a MongoDB database in order to store updates which are generated by dynamic events.

The Look Ahead component implements a reasoning mechanism capable of generating timetables for selected routes. Being more specific, the implementation includes a Genetic Algorithm which evaluates travel requests of past dates, makes predictions regarding future demand, and modifies the timetables so as to reduce the average waiting time of passengers and the number of required resources (i.e., number of used vehicles). The users of the Look Ahead component, which are considered to be the administrators of the transportation system, have also the opportunity to use a simulator for generating travel requests in order to evaluate the provided timetables.

The Transportation Planner is integrated with the CityPulse framework through the Decision Support component, which is responsible for real-time route and timetable adjustments. Without a doubt, pre-

scheduled bus routes could be affected by the levels of traffic density. For this reason, the Decision Support component takes advantage of the Route Generator in order to identify less time-consuming routes which connect the same bus stops, while taking into consideration the corresponding delays which are triggered by traffic levels and unexpected road incidents.

To conclude, using the Transportation Planner a user has the opportunity to simulate a bus transportation system of a selected area, which provides timetables with minimized waiting times for passengers and includes adjustments produced by traffic incidents.

2.4.9.2. Link

This application is currently being integrated with the CityPulse framework and the code will soon be available on CityPulse's GitHub.

3. Conclusions

This document presents the Smart City Demonstrator and different applications developed in the scope of Activity 6.2 in the CityPulse project. In order to maximize the impact and uptake of the developed software components, they have been released as open source on GitHub. This way the interest and involvement of the development communities in CityPulse will be possible even beyond the project's lifetime. It is key that not only software code is shared, but also other relevant materials such as tutorials (installation and execution guides, etc.) are made available. The report contains a collection of all the produced materials that have been extracted from the project's GitHub repository, as well as the project's website. The Smart City Demonstrator is described through:

- Applications: a description of the different scenarios that have been developed and its integration with the underlying CityPulse framework;
- Framework Components: a complete description of the different components developed in the other technical workpackages, their integrations or dependencies with other components, as well as installation and running guides.
- Datasets and Tools: that are the sources of information for all the applications – these were described and made available on the project's website.

By producing and releasing all these materials the task of reusing the demonstrator for the existing or new innovative services becomes much more comprehensive and less time consuming for developers.

The integrated system fulfills all the initial elicited requirements both technical and non-technical and the implemented applications demonstrate and showcase the potential of the CityPulse platform. The uptake and feedback from the developer communities will be closely tracked and reported as part of Activity 6.4.

References

CityPulse-D5.3	Puiu, D. et al. (2016, February) CityPulse D5.3 – Smart City Environment User Interfaces
CityPulse-D2.2	Tsiatsis, V. et al. (2015, August) CityPulse D2.2 - Smart City Framework
CityPulse-D3.1	Kolozali, S. et al. (2014, September) CityPulse D3.1 - Semantic Data Stream Annotation for Automated Processing
CityPulse-D3.2	Gao, F. et al. (2015, August) CityPulse D3.2 - Data Federation and Aggregation in Large-Scale Urban Data Environments
CityPulse-D3.3	Puiu, D. et al. (2016, February) CityPulse D3.3 - Knowledge-based Event Detection in Real World Streams
CityPulse-D4.1	Kuemper, D. et al. (2015, February) CityPulse D4.1 - Measures and Methods for Reliable Information Processing
CityPulse-D4.2	Kuemper, D. et al. (2016, February) CityPulse D4.2 - Testing and Fault Recovery for Reliable Information Processing
CityPulse-D5.1	Mileo, A. et al. (2014, July) CityPulse D5.1 - Real-Time Adaptive Urban Reasoning
CityPulse-D5.2	Mileo, A. et al. (2015, August) CityPulse D5.2 - User-Centric Decision Support in Dynamic Environments
Puiu et al. 2016	D. Puiu, P. Barnaghi, R. Tönjes, D. Kümper, M. I. Ali, A. Mileo, J. X. Parreira, M. Fischer, S. Kolozali, N. Farajidavar, F. Gao, T. Iggena, T. Pham, C. Nechifor, D. Puschmann, J. Fernandes, “CityPulse: Large Scale Data Analytics Framework for Smart Cities”, IEEE Access (published)