Project acronym:	**PrEstoCloud**
Project full name:	**Proactive Cloud Resources Management at the Edge for efficient Real-Time Big Data Processing**
Grant agreement number:	**732339**

D2.4 Format and procedures for plugging in real-time data streams

Deliverable Editor:	**Nenad Stojanovic (Nissatech)**
Other contributors:	Birkan Dag, Birgit Helbig (Software AG), Panagiotis Gkouvas, Giannis Ledakis (Ubitech), Dimitris Apostolou, Yiannis Verginadis, Nikos Papageorgiou (ICCS), Guillaume Urvoy-Keller (CNRS), Iyad Alshabani, Vincent Kherbache (ActiveEON), Nenad Stojanovic, Dusan Kostic, Aleksandar Stojanovic (Nissatech), Quentin Jacquemart (CNRS), Nectarios Efstathiou (ADITESS)
Deliverable Reviewers:	George Kioumourtzis (ADITESS) Yevgeniya Sulema (NAM)
Deliverable due date:	**31/07/2017**
Submission date:	**31/03/2018**
Distribution level:	**Public**
Version:	**1.4**

This document is part of a research project funded by the Horizon 2020 Framework Programme of the European Union

List of Changes as Compared to 1st Submission

Deliverable D2.4 is being resubmitted, in order to address the reviewers' recommendations following the outcome of the 1st interim review of the PrEstoCloud Project. In particular, the revisions aim at the rectification of the following recommendations:

- Recommendation 1

  "One of the challenges of the deliverable is to deal with big data however, the amount of data to deal with is never quantified."

- Recommendation 2

  "There are three ways of deploying the architecture. However, the architecture itself is not described. What is the database for? It seems that all data is stored there. Which kind of database is being used for storing big data streams with millions of events per second? Later several centralized databases are mentioned which cannot provide such a support."

- Recommendation 3

  "Figure 20 in Section 6.4.1 is not explained. What is the role of each component (Hbase, Hadoop…)?"

- Recommendation 4

  "There are several undefined references in the deliverable."

- Recommendation 5

  "The definition of the data of the use cases should include the number of events /time unit for streams and the size of the events."

First, to address the 1st issue and the 5th issues, we have extended the deliverable D7.2. In chapter 7 of this deliverable we have included a KPI table per use case. The use case partners extended the definition of the use case relevant data sets and included the detailed information about data to deal with. This data belongs to Big Data category, as it covers at least one dimension (V) of Big Data.

Our responses to the 2nd recommendation are included at the end of section "3.2.2 Real-time integration architecture".

Regarding the 3rd recommendation, we clarify the role of each component in section 3.6.1 right below Figure 20.

As far as the 4th recommendation is concerned, we have corrected all undefined references.

Change Log

Version	Date	Amended by	Changes
0.1	29.06.2017	Nenad Stojanovic	Definition of the scope (all relevant partners)
0.2	17.07.2017	Nenad Stojanovic	Structure, contributions delegated
0.3	21.07.2017	Nenad Stojanovic	Added content from one use case
0.4	24.07.2017	Nenad Stojanovic	Explained architecture, provided examples
0.5	24.07.2017	Dimitris Apostolou Yiannis Verginadis Nikos Papageorgiou	Provided content for section 5
0.6	25.07.2017	Birgit Helbig	Provided content for section 5 for Software AG
0.8	28.8.2017	Nenad Stojanovic, Dusan Kostic, Aleksandar Stojanovic	Extended content Formatted properly
0.9	28.8.2017	Nenad Stojanovic	Reviewers comments implemented
0.95	11.9.2017	Nenad Stojanovic	Tuning of the whole document to polish the style
1.1	07/02/2018	All partners	Analysis of the reviewers' recommendations following the outcome of the 1st Review
1.2	20.03.2018	Nissatech, end users	Review comments addressed
1.3	28.03.2018	Nissatech	Consolidated version ready for review
1.4	31.032018	Nissatech	Final version, review comments addressed

Table of Contents

List of Figures

Definitions, Acronyms and Abbreviations

Acronym	Title
API	Application Programming Interface
BDVA	Big Data Value Association
CEP	Complex Event Processing
Dx	Deliverable (where x defines the deliverable identification number e.g. D1.1.1)
EDA	Event Driven Architecture
ETL	Extract Transform Load
GPS	Global Positioning System
GTP	General Trip Preferences
HTTP	HyperText Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
I/O	Input / Output
ICT	Information and Communication Technologies
IoT	Internet of Things
JSON	JavaScript Object Notation
PubSub	Publish-Subscribe Middleware
REST	Representational State Transfer
SQL	Structured Query Language
TB	Terabyte
WP	Work Package
XML	Extensible Markup Language

Executive Summary

An intensive data-driven processing begins with an efficient collection and pre-processing of available data. This task requires a valid architecture and a set of methods for efficient processing. It also requires a deeper analysis of the properties of data sources in order to understand properly which methods can be applied. This deliverable elaborates on these topics, put in the specific context of selected use cases. Indeed, in a complex processing structure, as presented in this project, it is important to understand the relations between the available data sources and components that process this data, in order to define the most efficient architecture for gathering and processing data.

There are several challenges related to this task and they are taken as the main requirements in defining the architecture. The most important ones are:

- the amount of data processed requires the application of big data technologies;
- real-time nature of processing requires the usage of the architectures suitable for efficient data collection and processing;
- the complex processing structure and the need for doing a part of the processing close to data sources (edge) require a flexible processing infrastructure (edge / fog computing).

There are three main outcomes from this deliverable, which will be used in developing the conceptual architecture of the system (D2.3):

- the design of the architecture for data collection and processing
- the structure of the data adapters which are required by each component (technical partner) in order to get and understand in the proper functional context the data from available data sources
- the specification of the nature of data source available in use cases, emphasizing their big data nature (which requires specific types of collection and processing)

This deliverable will serve as one of the sources for the development of the conceptual architecture (D2.3).

1. Introduction

1.1 Objectives

The main objective of this deliverable is to elaborate on the methods how the data from external data sources can be gathered and stored in the systems, in order to be processed as planned.

In a complex processing structure, as presented in this project, it is important to understand the relations between the available data sources and components that process this data, in order to define the most efficient architecture for gathering and processing data.

Moreover, in an edge processing architecture, as required by use cases, it is very important to understand what kind of processing is required close to data sources in order to define the most suitable methods/technologies.

1.2 Relation to other deliverables

This deliverable will serve as an important source for the deliverable D2.3 related to the development of the conceptual architecture.

The outcome of this deliverable will also serve as input for D3.1 (Communication broker for real-time data streams) and the pilot evaluation deliverables in WP7.

1.3 Challenges

There are many challenges related to the data collection and preprocessing which should be resolved in order to realize the planned system properly. We discuss them briefly:

- the amount of data processed requires the application of big data technologies;
- real-time nature of processing requires the usage of the corresponding architectures;
- the complex processing structure and the need for doing a part of the processing close to data sources (edge) require a flexible processing infrastructure (edge / fog computing).

1.4 Method

This work is based on three main sources:

- The findings from the relevant (mainly industry) communities,
- The need of the technical components regarding the data (sources),
- The requirements from use cases regarding processing available data sources.

1.5 Structure of the document

This deliverable is structured in the following way:

- In section 4 we present an analysis of the methods for data collection and preprocessing
- In section 5 we discuss the architectures for efficient data collection and preprocessing
- In section 6 we present an overview of the data sources that are available in use cases and the methods for accessing data
- In section 7 we focus on the use case data flow as it is driven by the analytic demands
- In section 8 we provide some concluding remarks.

2. Data collection and processing

In this section we provide an analysis of the challenges for the data collection and processing in the context of the types of data that can be processed in envisioned scenarios.

In addition, we provide information related to the existing work in the classification of data sources and methods for data gathering and processing. As the main source we use the work done in the scope of BDVA[1] (Big Data Value Association, www.bdva.eu), related to the architectures for the big data processing. Although there are different proposals for such an architecture, due to the involvement of several partners in the BDVA and the possibility to create an impact on its work (and the reference architecture), we have decided to follow this model.

2.1 BDVA reference architecture

The work on the architecture for data collection and processing is based on the work done in BDVA, mainly focusing on the reference model for Big Data Processing, as presented in Figure 1. This figure explains a processing pipeline from the extraction of data and its management, through the processing/analytics, till the visualization.

The most important parts of the architecture for this deliverable are:

1. Classification of Data Types, which suggests how big data sources can be classified in order to better understand how they can be connected to the processing system and
2. Data Processing cycle, which explains how the data can be processed (big data processing pipeline).

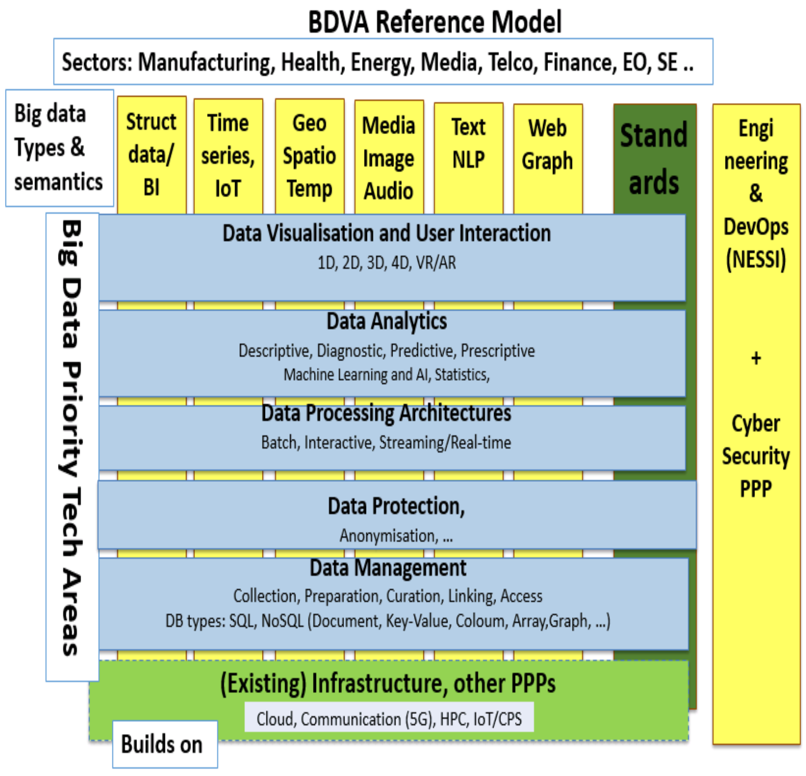

BDVA Reference Model
Sectors: Manufacturing, Health, Energy, Media, Telco, Finance, EO, SE ..
Big data Types & semantics
Struct data/ BI
Time series, IoT
Geo Spatio Temp
Media Image Audio
Text NLP
Web Graph
Standards
Engineering & DevOps (NESSI) + Cyber Security PPP
Big Data Priority Tech Areas
Data Visualisation and User Interaction
1D, 2D, 3D, 4D, VR/AR
Data Analytics
Descriptive, Diagnostic, Predictive, Prescriptive
Machine Learning and AI, Statistics,
Data Processing Architectures
Batch, Interactive, Streaming/Real-time
Data Protection,
Anonymisation, ...
Data Management
Collection, Preparation, Curation, Linking, Access
DB types: SQL, NoSQL (Document, Key-Value, Coloum, Array,Graph, ...)
(Existing) Infrastructure, other PPPs
Cloud, Communication (5G), HPC, IoT/CPS
Builds on

Figure 1: BDVA Reference architecture

[1] The partner responsible for this deliverable is very active in BDVA. Nenad Stojanovic is the co-chair of Data analytics (TF6 SG3)

From the data preparation and processing point of view, the type of data is the most crucial element that can predetermine rest of the processing pipeline. For example, real-time streaming data is usually processed on-the fly mode and stored only on demand. Therefore, we need to clarify the nature of data and the way how it will be collected in order to be able to define an optimal architecture for processing.

As one of the important contributions of this deliverable, in Figure 2 we illustrate the relations between three main phases (Data Preparation, Analytics, Visualization) in order to get a better understanding of how the data can be collected and processed. Consequently, it provides a high level view on the data processing pipeline.

This figure gives an important global context for the work presented in this deliverable. In the rest of this subsection we provide more details about this context (phases).

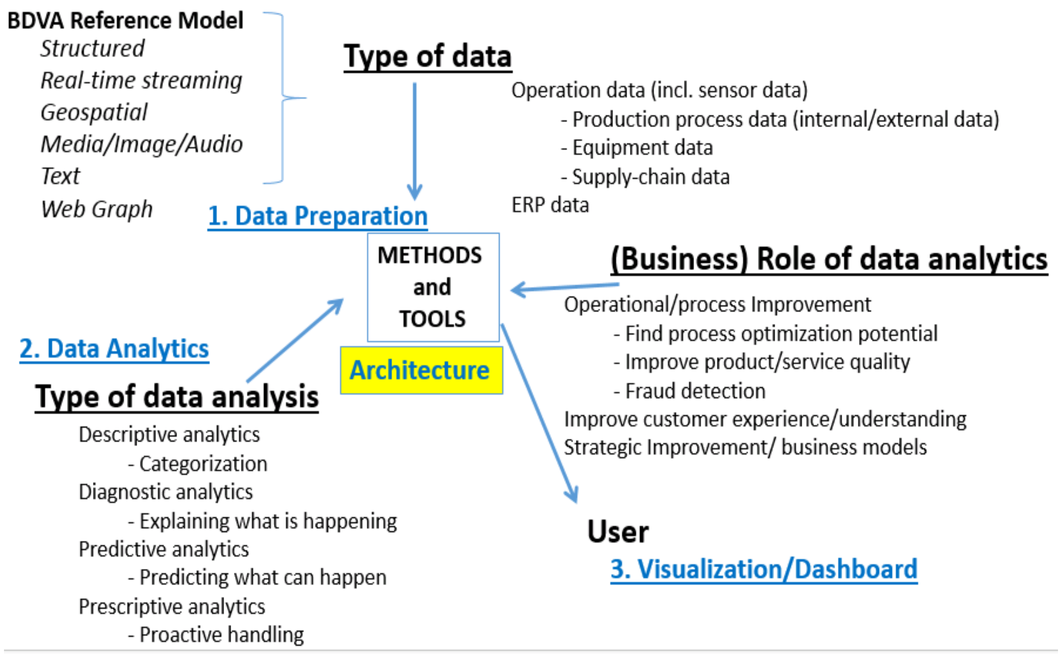

BDVA Reference Model
Structured
Real-time streaming
Geospatial
Media/Image/Audio
Text
Web Graph
Type of data
Operation data (incl. sensor data)
- Production process data (internal/external data)
- Equipment data
- Supply-chain data
ERP data
1. Data Preparation
METHODS and TOOLS
Architecture
(Business) Role of data analytics
Operational/process Improvement
- Find process optimization potential
- Improve product/service quality
- Fraud detection
Improve customer experience/understanding
Strategic Improvement/ business models
2. Data Analytics
Type of data analysis
Descriptive analytics
- Categorization
Diagnostic analytics
- Explaining what is happening
Predictive analytics
- Predicting what can happen
Prescriptive analytics
- Proactive handling
User
3. Visualization/Dashboard

Figure 2: Data Processing Life Cycle

2.2 Data preparation

There are several characteristics we want to know from a data source in order to understand the nature of the data to be processed and to define the most efficient processing methods (for example, normalization of values can be in done in different ways depending on the nature of data). They are summarized as the following features of a data type:

- Type of data
  - Source: e.g. sensor X on machine Y (IoT)
  - Type/storage: e.g. real-time streams/historian
  - Type of streaming data:
    - E.g. smooth, irregular, multimodal, sparse , discrete, (cf. Figure 4.3)
  - Volume (past data): e.g. 3 years, ca. 10TB
  - Velocity (real-time data), e.g. freq.: 1kHz
  - Uncertainty:

    (incl. the main cause/source of uncertainty)

- Dynamicity (changeability), (cf. Figure 4.4)
- Seasonality, (cf. Figure 4.5)
- Complexity, (cf. Figure 4.6)

The goal is to collect all mentioned features in order to alleviate the work of developing adapters (pre-processing pipeline). In the following figures we illustrate some of these features.

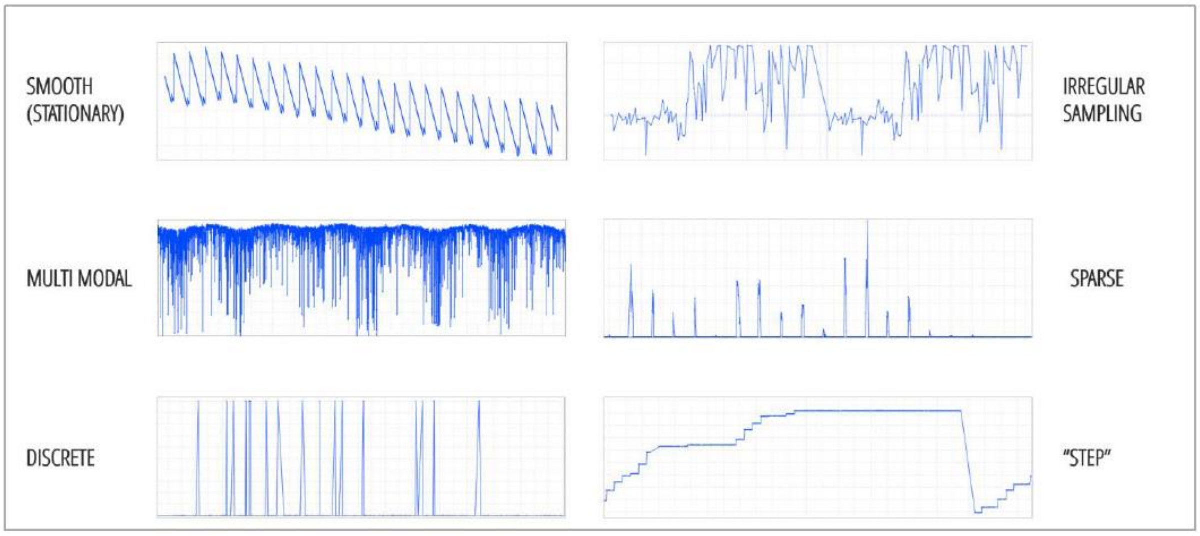


Figure 3: Examples for Types of streaming data

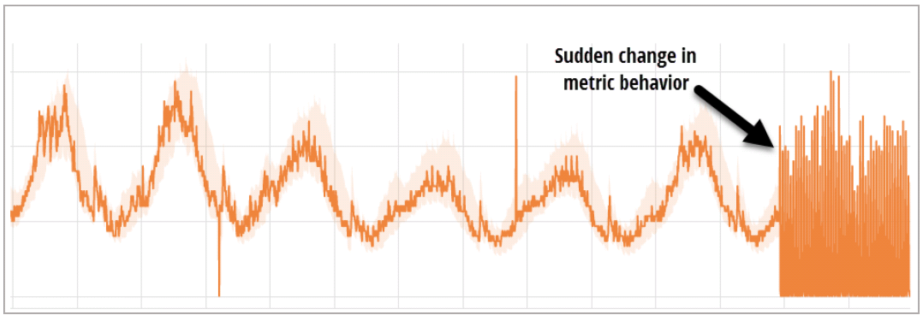


Figure 4: Dynamicity

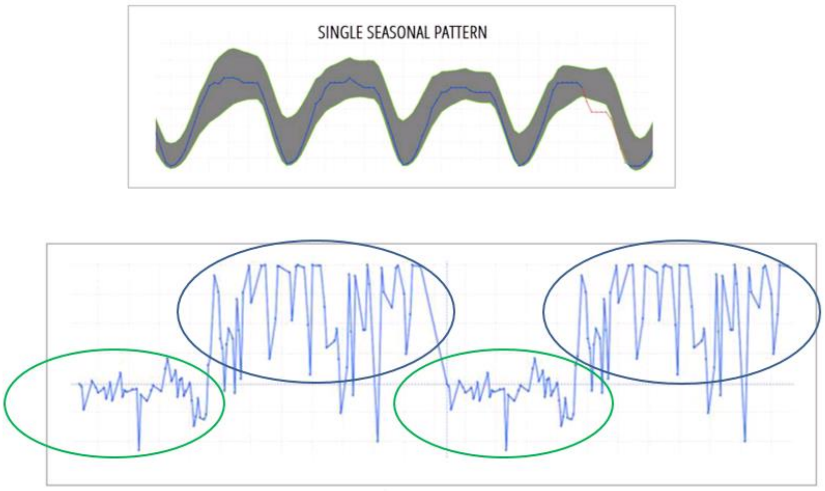


Figure 5: Seasionality

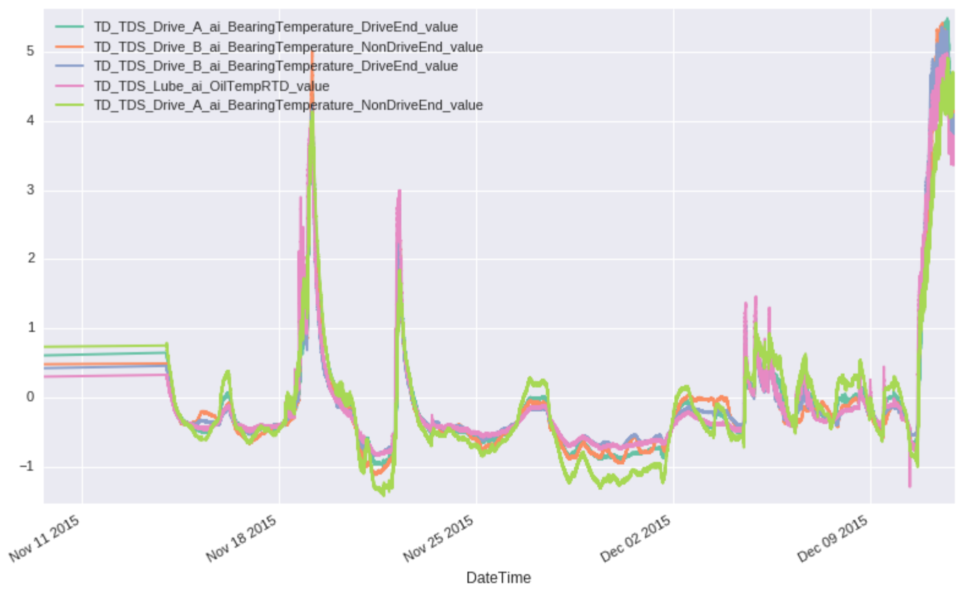


Figure 6: Complex data

2.3 Data analytics

There are four main types of data analysis:

- Descriptive analytics
  - Explaining what is happening
- Diagnostics
  - Explaining why did it happen
- Prognostics
  - Predicting what can happen
- Prescriptive
  - Proactive handling

Their relations between mentioned types of analytics are shown in Figure 7.

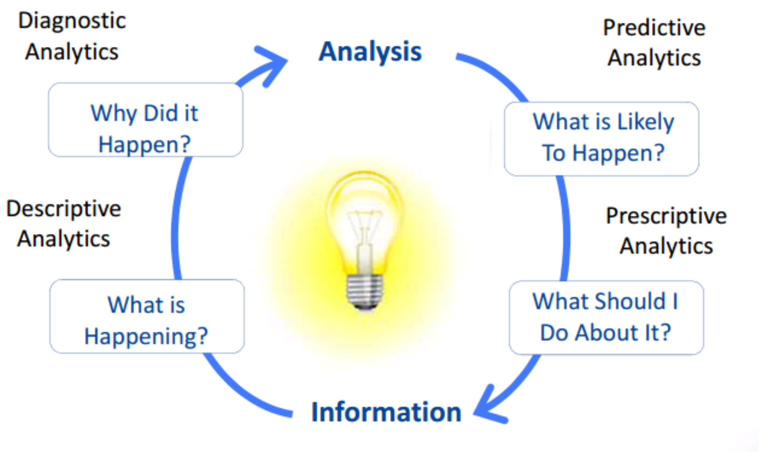


Figure 7: Types of analytics

This classification is important in order to understand the nature of some processing methods, as illustrated in Figure 8.

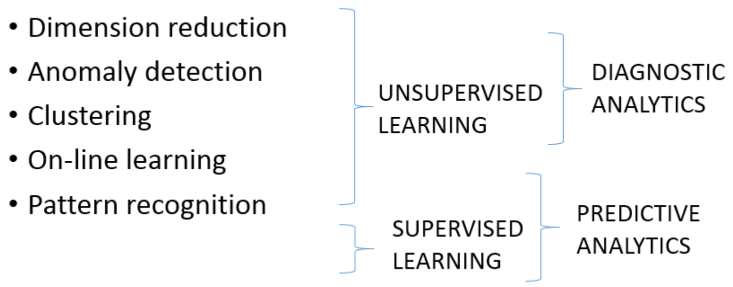


Figure 8: Learning methods as parts of data analytics

Following figures illustrate some of the processing methods, put in the context of the anomaly detection scenario:

- Figure 9: the dimensionality of the input data set should be decreased in order to perform some data analytics methods (e.g. clustering)
- Figure 10: the clusters of the normal and anomalous behaviour should be derived (machine learning step)
- Figure 11: these clusters should be described as patterns that can be detected on the real-time streams
- Figure 12: anomalies should be detected in the real-time streams

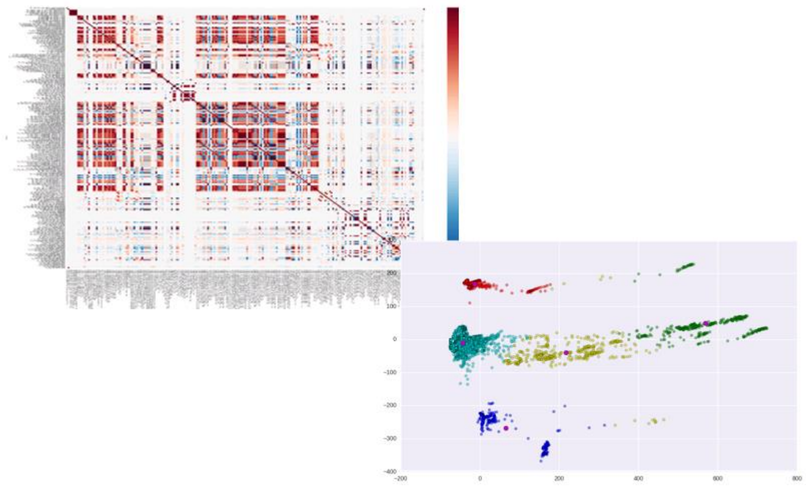


Figure 9: Dimension reduction

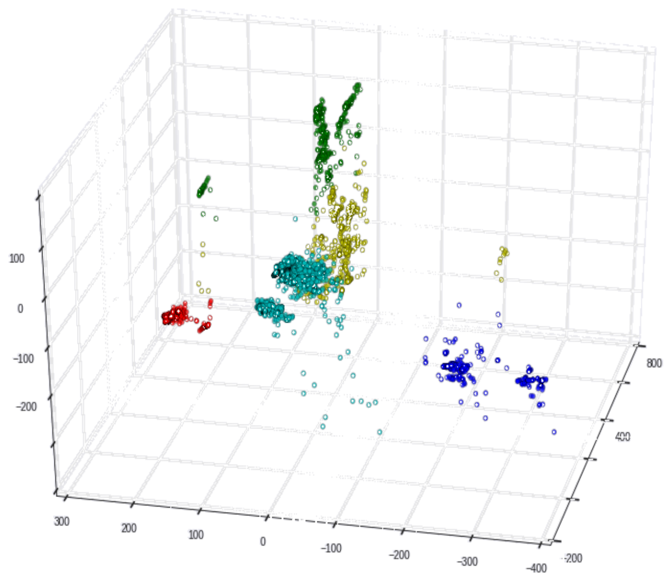


Figure 10: Clustering

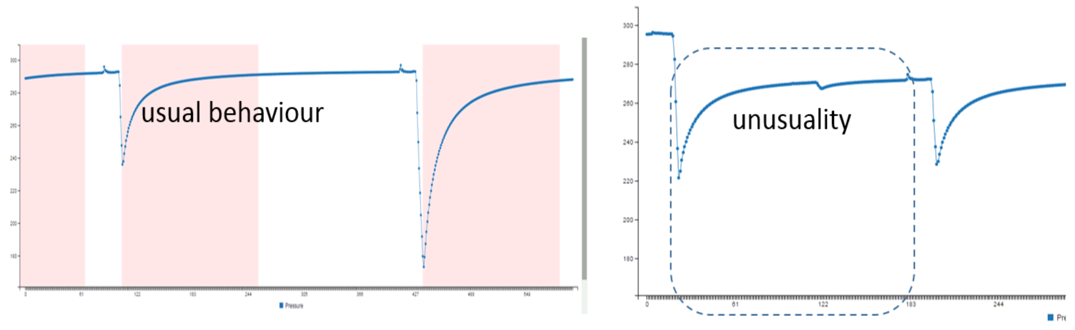


Figure 11: Pattern recognition

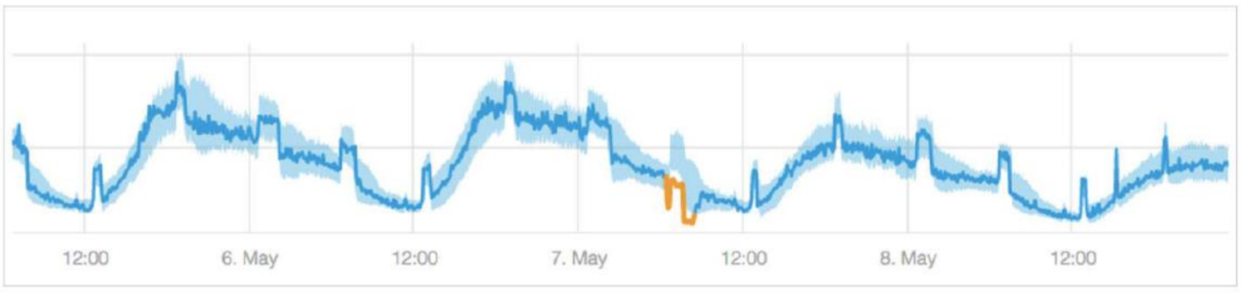


Figure 12: Anomaly detection

It is clear that the data processing is rather complex and requires special methods in order to cope with the challenges originating from the amount and complexity of data (i.e. volume, variety, velocity, veracity of big data). This is one of the main reasons why the real-time data from real-world data sources should be processed in a systematic way, which is part of the descriptions given in the next section.

Just as an illustration of the complexity of the problem: if the data analytics applied on the data collected from the past requires a complex pre-processing (e.g. the application of some dimension reduction methods) in order to drive knowledge (e.g. clusters of normal/anomalous behaviour), then the system that should realize real-time detection of anomalies in the streaming data has to implement all pre-processing steps as the real-time processing pipeline.

3. The architecture for data processing and collection

In this section we provide an analysis of the types of architectures which can be applied for the data collection and pre-processing and detail the concrete architectural contributions from the technical partners.

We start with a discussion of the types of analyses which can be applied in the pre-processing step, implying the need for architectures suitable for intensive processing close to a data source.

3.1 The need for real-time processing on the edge

IDC[2] pointed out that, of the 160 ZB, about a quarter of it will be real-time data in nature (generated, processed and instantly accessible) up from around 5 percent today, and most of that real-time data (95 percent) will come from the world of IoT. However, data storage capacity has not, and will not, be able to keep up. In the words of IDC, "…the quantity of data generation can and will continue to outpace any reasonable expectation of our ability to store all of the data."

It turns out by 2025 only a tiny percentage of data – between 3 percent and 12 percent depending on the source – will be able to be stored[3]. Quite literally, nowhere near enough storage hardware can be manufactured to handle the predicted data volumes. The conclusion is that the data must be collected, processed and analyzed in-memory, in real-time, close to where it is generated.

3.1.1 The role of Edge Computing: basics

Data received from a data source is not information. Data is raw, unfiltered, unprocessed facts. Much of that data can be:

3. repetitive (e.g. high-speed sensor data),
4. generally useless (a lot of developer-produced application log files), or
5. lacking sufficient context (data in the form of binary code).

The primary goal of Edge Computing Layer must then be to transform data into information. It is the role which was traditionally dedicated to the data adapters. Therefore, we see the notion of Edge Computing as an extension of the data adapters, which are focused on complex processing of the input data. This transformation can be achieved through a combination of following processing types:

1. "simple event processing" EPAs - filter and routing,
2. "mediated event processing" EPAs - enrichment, transformation, validation
3. "complex event processing" EPAs - pattern detection.

The list is derived from the types of event processing agents[4] proposed in the CEP (Complex Event Processing Community).

1. Filtering is the possibility to select a part of the input data according to a criteria (filter). It is a technique that enables focusing only on a subset of the input data.

2. Enrichment is the technique for extending the initial data set with some additional information. It is especially useful if the input stream doesn't contain sufficient context to be properly processed, like only IDs, codes or binary data. Real-time data is joined with some context (about devices, parts, customers, etc.) in order to get valuable information.

3. Pattern detection is the process of matching input data stream with a predefined "template" (pattern). It is the common term for condensing or grouping data, usually time-series data, to

[2] http://www.seagate.com/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf

[3] https://www.rtinsights.com/the-rise-of-real-time-data-prepare-for-exponential-growth/

[4] http://epthinking.blogspot.rs/2008/04/on-event-processing-agents.html

reduce its granularity. This can involve basic statistical analysis, sampling, or other means that retain the information content, but reduce the frequency of the data. A related notion is change detection which, as the name suggests, only outputs data when it changes. The most appropriate technique will depend on the source data and use case.

3.1.2 Creating a Smart Data Architecture: requirements

We consider edge processing as part of data architecture to handle the predicted deluge of data. More sophisticated analytics functions can also be applied at the edge, as we indicated in the previous subsection. Many use cases call for very fast reaction times to complex scenarios. In these situations, bringing analytics capabilities such as real-time correlation, complex event processing, and machine learning scoring to the edge has numerous advantages[5].

The edge can be scaled as source devices and data ingestion grows, while still retaining fast response times, and often requires lower cost hardware. Analytics at the edge has the added side-effect that the source data for analytics doesn't have to leave the edge, only the results, further reducing the need for data storage. This is important from several reasons, starting from the privacy till the efficiency in the data transport and costs.

Of course, overall big data processing architecture includes more than edge processing and analytics. There may be on-premises and cloud requirements for further processing, analytics, storage and machine learning. The key is to recognize that data can be collected, processed, analyzed and stored in many different zones of a smart data architecture, and it is essential to perform each of these actions where it most makes sense to gain the greatest value from the exponential growth in data.

3.1.3 The relation to the fog computing: impact

Fog computing – a term originally coined by Cisco—is in many ways synonymous with edge computing and in this subsection we resolve that relation briefly.

OpenFog Consortium (https://www.openfogconsortium.org/) describes fog computing as bridging the continuum from cloud to things. It's a continuum because fog overlaps cloud and things and fills the computing gap in between. Fog provides the missing link in what data needs to be pushed to the cloud, and what should be analyzed locally – at the edge.

Pragmatical explanation of the difference (https://www.rtinsights.com/what-is-fog-computing-open-consortium/): „While edge computing or edge analytics may exclusively refer to performing analytics at devices that are on, or close to, the network's edge, a fog computing architecture would perform analytics on anything from the network centre to the edge."

In contrast to the cloud, fog platforms have been described as dense computational architectures at the network's edge. Characteristics of such platforms reportedly include low latency, location awareness and use of wireless access. Benefits include real-time analytics and improved security.

The following figure[6] illustrates the differences between two concepts. Edge runs specific applications in a fixed logic location and provides a direct transmission service without data analysis. Fog works with edge to run applications in a multi-layer architecture that decouples and meshes the hardware and software functions, allowing for configuring / reconfiguring for different applications while performing intelligent transmission services with computing/storage/communication capabilities along the cloud to things continuum.

[5] https://www.rtinsights.com/the-rise-of-real-time-data-prepare-for-exponential-growth/

[6]https://www.rtinsights.com/10-ways-fog-computing-extends-the-edge/?utm_medium=email&utm_campaign=August%2023%202017&utm_content=August%2023%202017+Version+A+CID_d55462c577002d50cb05f49f4536e454&utm_source=CampaignMonitor&utm_term=Read%20more

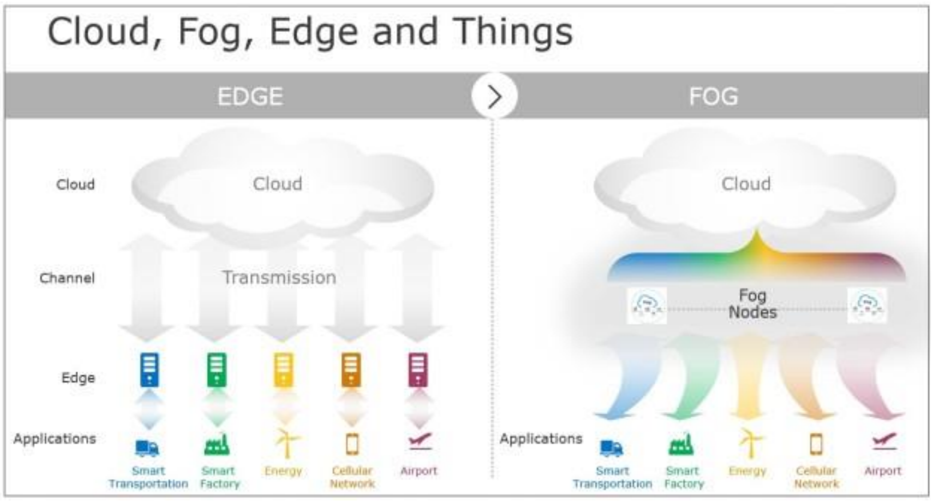


Figure 13: Side by side view of edge and fog architectures

However, in order to simplify the explanation, in this deliverable we will not make a difference between two concepts, which will be resolved in D2.3.

3.2 Data architectures types

We have developed three types of architectures for the integration of data sources into a platform/system for data processing (general one):

1. Local integration architecture (cf. Figure 14)
2. Remote integration (cf. Figure 15 and Figure 16)
3. Real-time integration architecture (cf. Figure 17)

The main reason for this classification is the need for the flexibility for the data processing, since there are different policies and demands for data handling in different organizations/environments. We describe shortly each of them.

3.2.1 Local integration architecture

Local integration architecture assumes that the data processing platform can be deployed in the internal/local use case infrastructure. In that case:

1. Data source is connected to a dedicated database
2. Remote client is communicating with the database
3. Local client is communicating with the external service (installed locally)

Figure 14 differs also the case that the Remote client can be only implicitly connected to the database (triggers are initiated from another component).

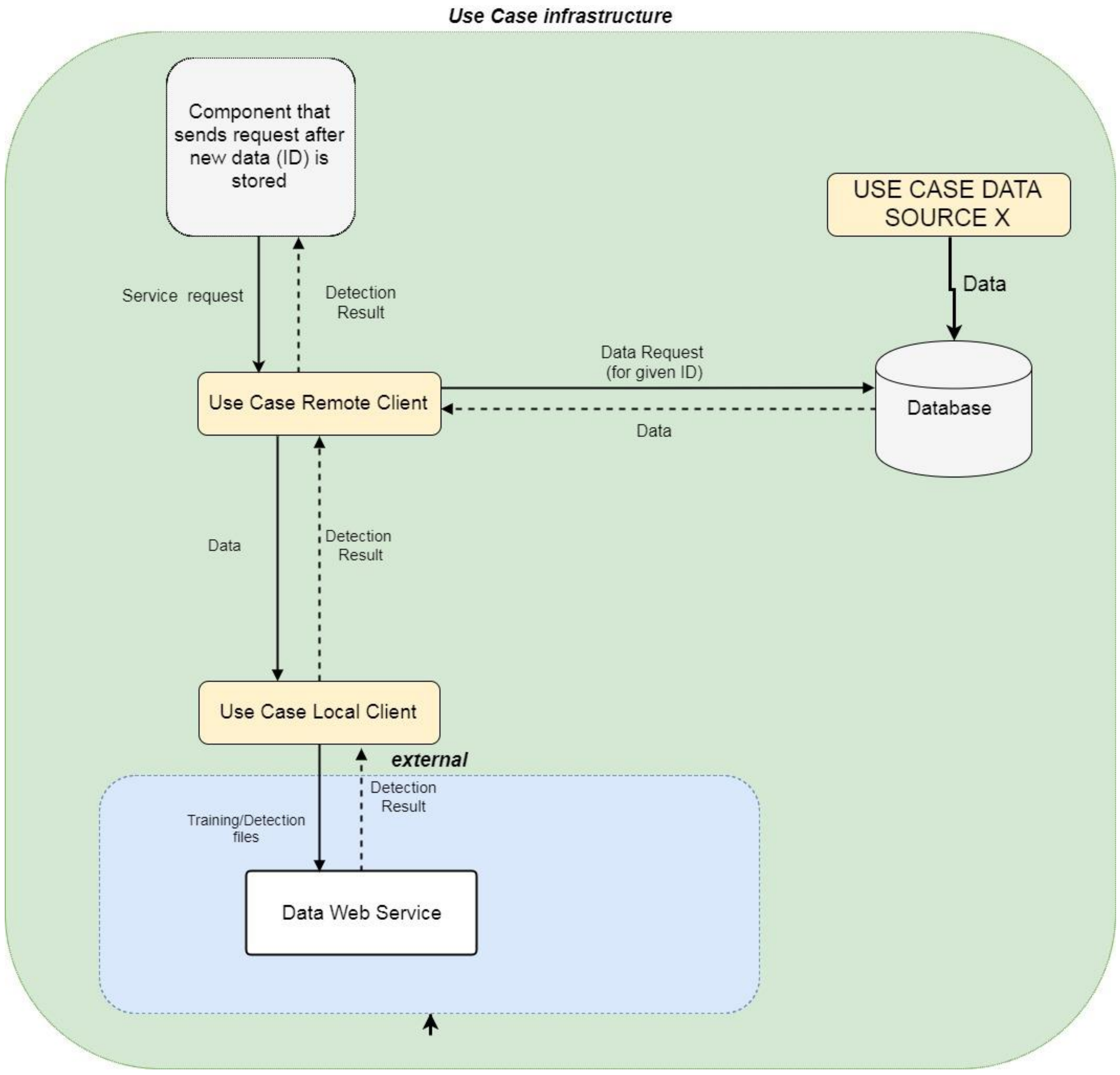


Figure 14: Local integration architecture

3.2.1 Remote integration architecture

Remote integration architecture assumes that the data processing platform cannot be deployed in the internal/local use case infrastructure, but in an external space (e.g. public/private cloud). In that case:

1. Data source is connected to a dedicated database
2. Remote client is communicating with the database
3. Local client is communicating with the external service (installed locally).

Figure 15 shows the case that the Remote client can be only implicitly connected to the database (triggers are initiated from another component). This is the situation where the direct access to the data source is not allowed.

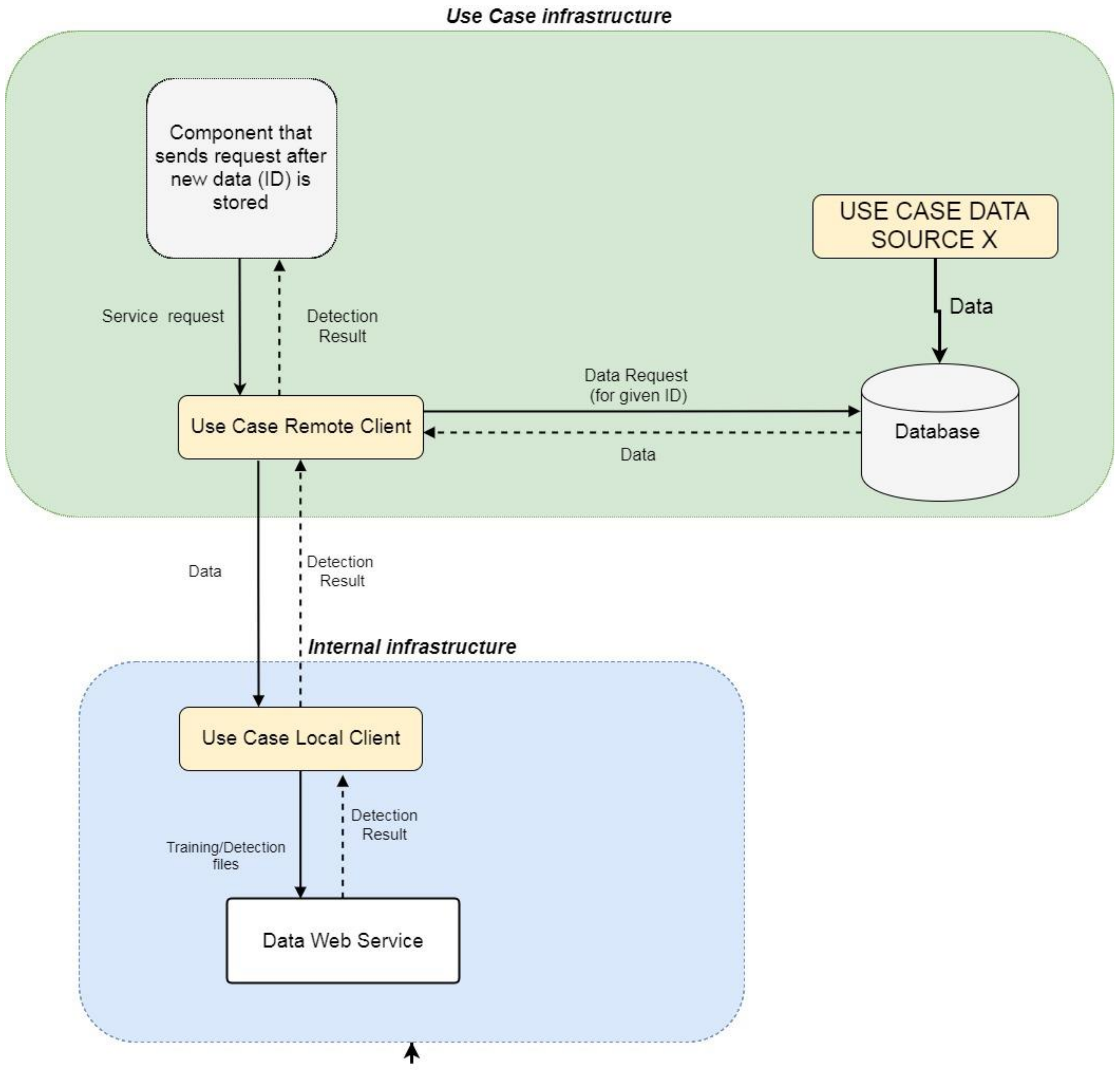


Figure 15: Remote integration architecture with implicit triggers

Figure 16 shows an architecture with a direct access to the database.

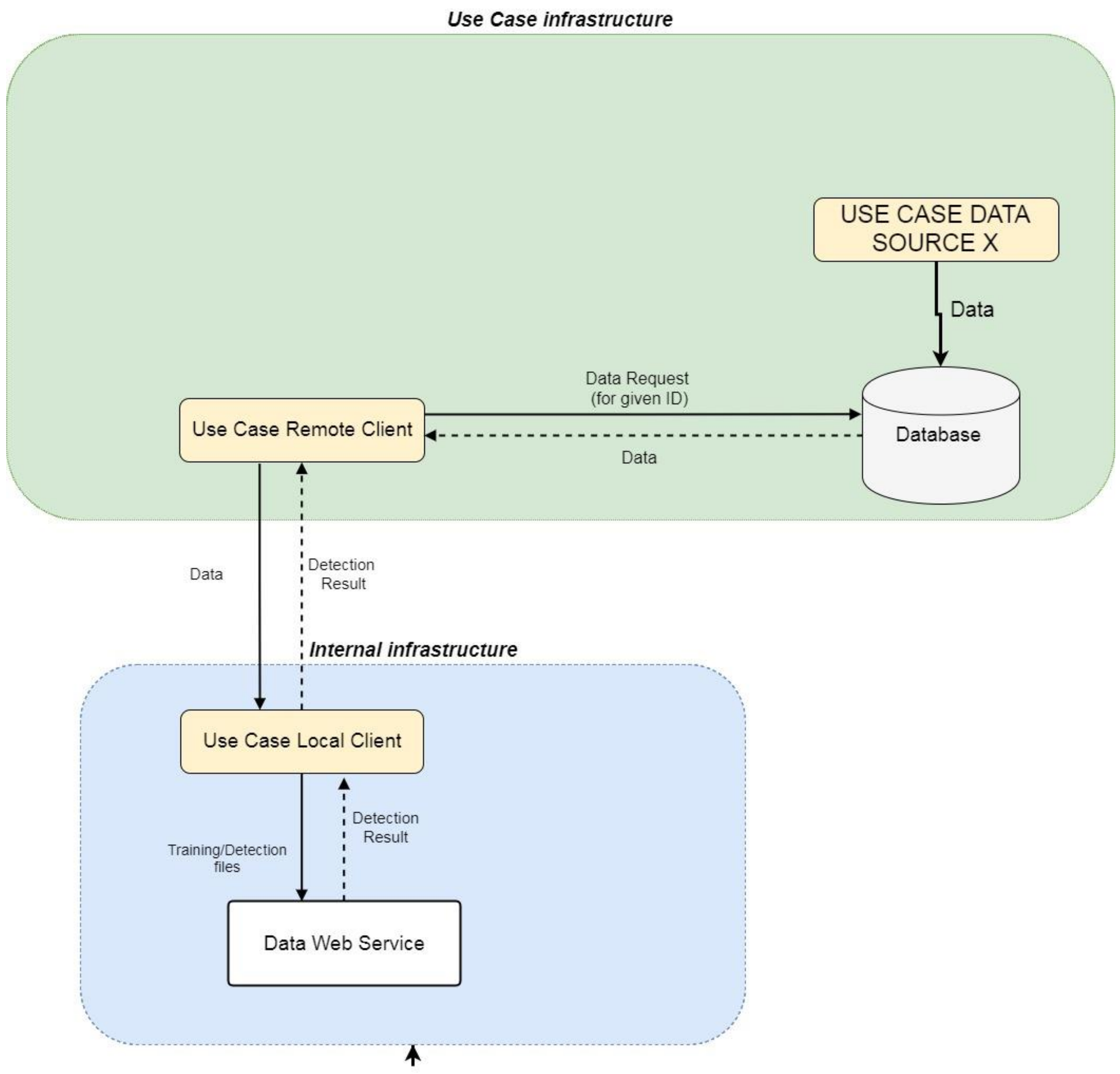


Figure 16: Remote integration architecture with a direct access to the database

3.2.2 Real-time integration architecture

Real-time integration architecture (cf. Figure 17) assumes that the real-time data can be directly transferred to the external service (asynchronously, through a broker). In that case:

4. Data source is connected to a Remote client
5. Remote client is communicating with the Broker (pub-sub)
6. Local client is communicating with the Broker (pub-sub).

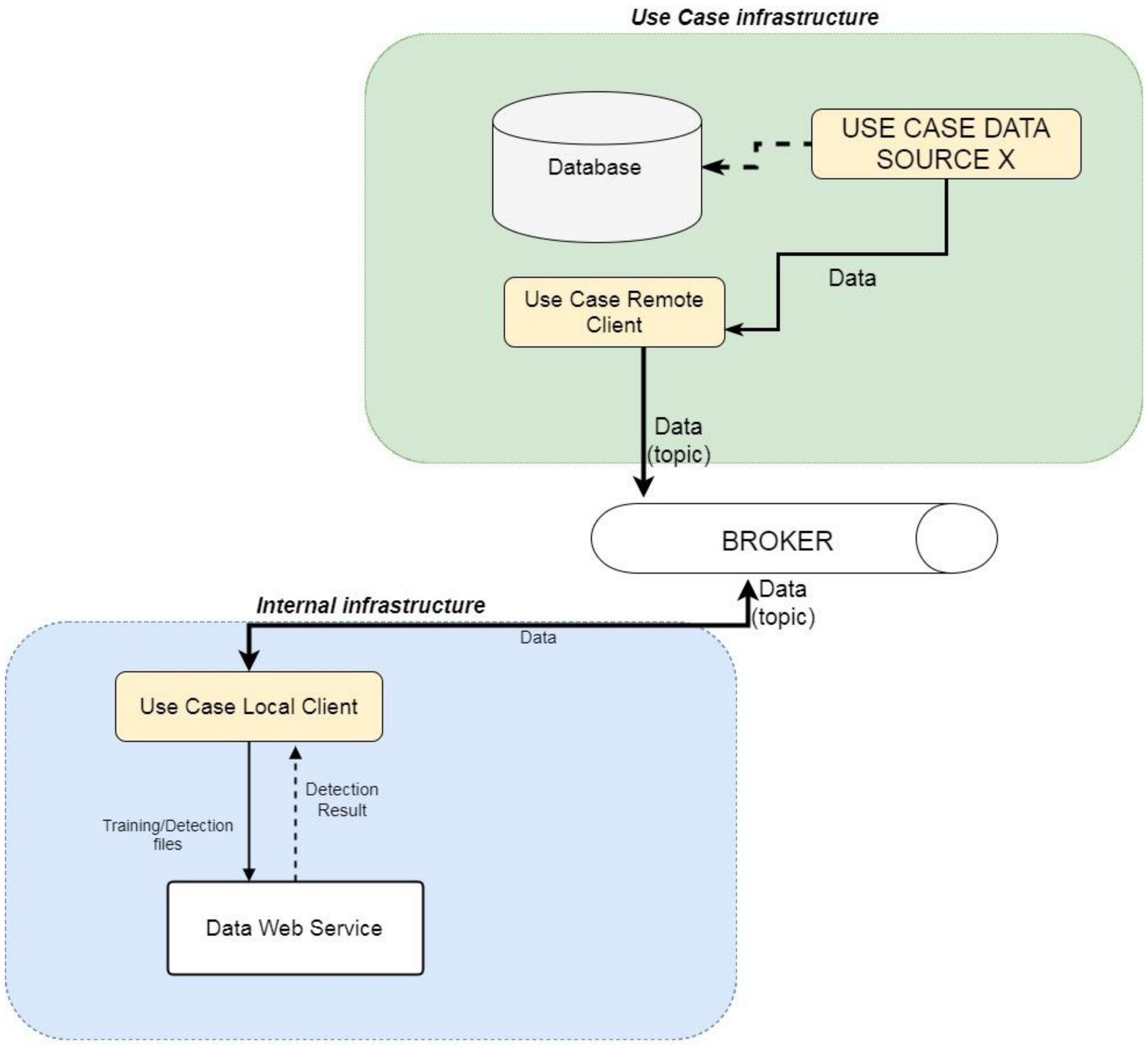


Figure 17: Real-time integration architecture

It is important to emphasize that our preferred type of the data architecture is the real-time integration due to several advantages, originating from the pub-sub architecture, which is in the nutshell of this integration type:

1. Loosely decoupling of data sources and data processing modules

2. Easier integration of new data sources (and processing modules)

3. Scalability of the architecture

Below we include our responses regarding to the following reviewers' recommendation:

> ***Comment:*** *"There are three ways of deploying the architecture. However, the architecture itself is not described. What is the database for? It seems that all data is stored there. Which kind of database is being used for storing big data streams with millions of events per*

second? Later several centralized databases are mentioned which cannot provide such a support."

Figures 14 – 17 present possible architectures for the integration of data sources in the context how and where the data (originating from data sources) will be processed, depending mainly on the constraints related to deployment of the platform (D2.3). For example, if the platform cannot be deployed in the internal/local use case infrastructure, but in an external space (e.g. public/private cloud), we will use Remote integration architecture. Presented situations correspond to different cases that can be found in an industry environment.

As we explicitly stated, our preferred type of the data architecture is the real-time integration due to several advantages mentioned above.

The database presented in figures is the local storage that is used by the end users for persisting the data. This is not a part of our architecture. Such a database can be any operational historian, like OSISoft PI (https://www.osisoft.com/).

In the rest of this section we provide more details about the integration architectures of all components which communicate to any type of external data sources.

3.3 Contribution to the data architecture: ActiveEON

3.3.1 Overview

All three ActiveEon's components (resp. Autonomic resource manager, Autonomic Data-intensive Application Manager and Fog Deployment & Monitoring) rely on a single and central database. They currently support HSQL, MariaDB, MySQL, Oracle, PostgreSQL and SQLServer. The database need to be initially setup with specific usernames and databases but there is no constraint on the location of the database which can be hosted locally or remotely with secure communications. However, the three aforementioned components must be strictly executed in a single machine to be able to communicate together.

Figure 18 shows an overview of the current architecture provided by ActiveEon. All interactions between internal components are represented and especially the communications with the central database. The main components used in PrEstoCloud are highlighted with blue color and the database appears in red.

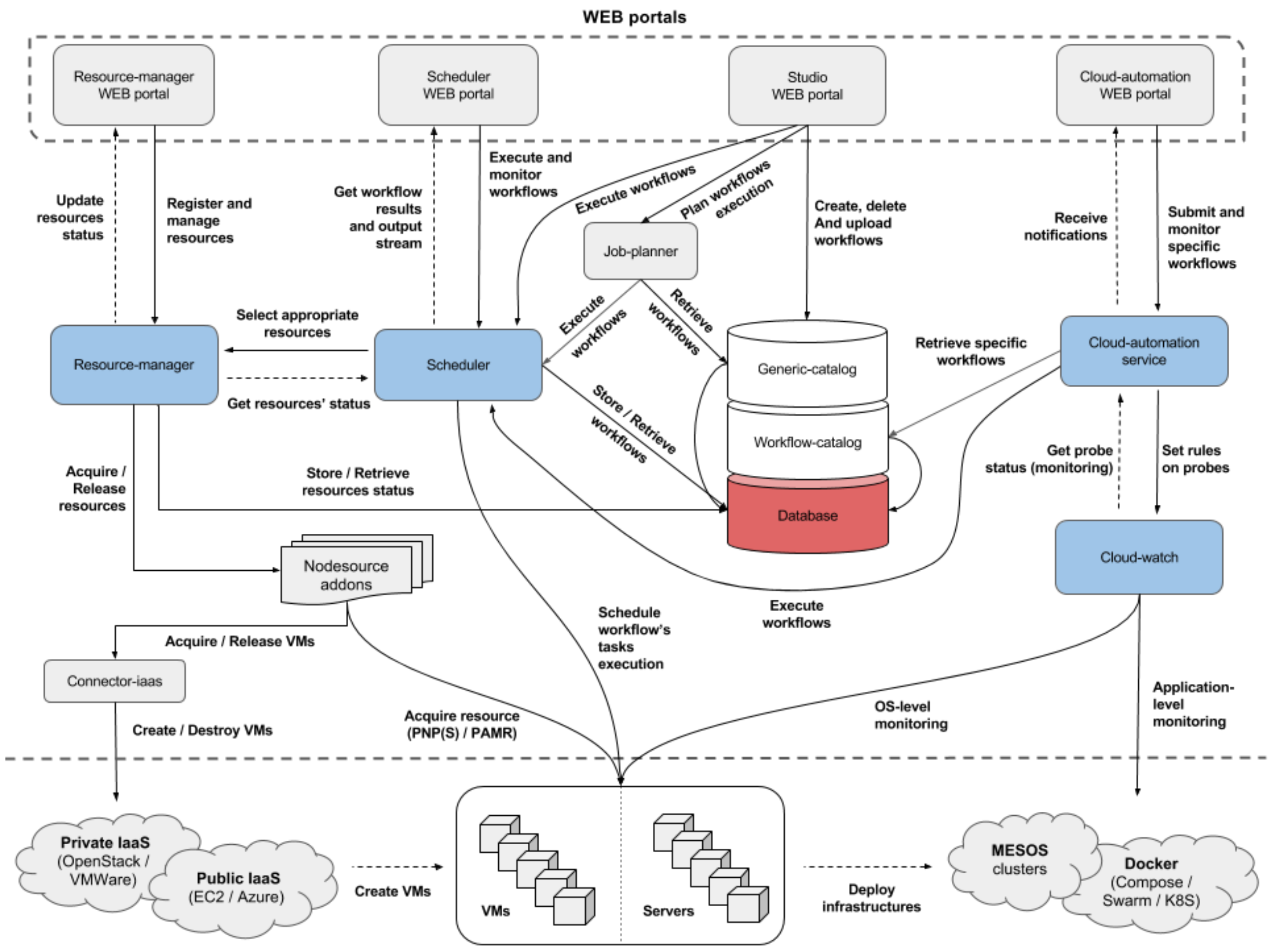

WEB portals
Resource-manager WEB portal
Scheduler WEB portal
Studio WEB portal
Cloud-automation WEB portal
Update resources status
Register and manage resources
Get workflow results and output stream
Execute and monitor workflows
Execute workflows
Plan workflows execution
Create, delete And upload workflows
Receive notifications
Submit and monitor specific workflows
Job-planner
Select appropriate resources
Get resources' status
Resource-manager
Scheduler
Execute workflows
Retrieve workflows
Generic-catalog
Workflow-catalog
Database
Retrieve specific workflows
Cloud-automation service
Store / Retrieve workflows
Acquire / Release resources
Store / Retrieve resources status
Get probe status (monitoring)
Set rules on probes
Cloud-watch
Nodesource addons
Schedule workflow's tasks execution
Execute workflows
Acquire / Release VMs
Connector-iaas
Create / Destroy VMs
Acquire resource (PNP(S) / PAMR)
OS-level monitoring
Application-level monitoring
Private IaaS (OpenStack / VMWare)
Public IaaS (EC2 / Azure)
Create VMs
VMs
Servers
Deploy infrastructures
MESOS clusters
Docker (Compose / Swarm / K8S)

Figure 18: The existing architecture provided by ActiveEon

Following more details about each component including input/output data formats and interactions with other components.

3.3.2 Autonomic resource manager

The Autonomic resource manager component is in charge of deploying, acquiring and releasing resources. The resources can be virtual (Virtual Machines (VMs), containers) or physical (local/remote servers, raspberry/odroid, drone, camera, etc.).

It will interact with public or private cloud infrastructures' (IaaS) REST APIs to deploy, configure, and release Virtual Machines (VMs). The VMs are configured to execute an agent (using the Java Virtual Machine (JVM)) that will interact with the Autonomic resource manager to send status notifications, and with the Autonomic Data-intensive Application Manager component to retrieve and execute tasks and send back outputs.

The communication between deployed 'agents' and the two aforementioned components is based on custom communication protocols, namely PNP and PAMR, that allow to establish secure connections and to transit via an external router in order to allow communications behind Network Address Translation (NAT) configured on both sides.

All these internal communications are already implemented and therefore data format information is not mentioned in the document.

To be able to communicate with private and public IaaS, the Autonomic resource manager must be configured first. To do so, it shall receive a set of required configurations about each IaaS including credentials, metadata and information about the VM to deploy but also the scaling policies to use in order to dynamically scale/up down the amount of resources in each IaaS. The information can be given manually from the provided WEB portal, or from its REST interface. Here is the description of the query to send:

POST http://www.example.com/rm/nodesource/create/

Form parameters

nodeSourceName *string*	name of the node source to create
infrastructureType *string*	fully qualified class name of the infrastructure to create
infrastructureParameters *[string]*	String parameters of the infrastructure, without the parameters containing files or credentials
infrastructureFileParameters *[string]*	File or credential parameters
policyType *string*	fully qualified class name of the policy to create
policyParameters *[string]*	String parameters of the policy, without the parameters containing files or credentials
policyFileParameters *[string]*	File or credential parameters

Header parameters

sessionid *string*	current session id

Returns

true if a node source has been created

3.3.3 Autonomic Data-intensive Application Manager

The Autonomic Data-intensive Application Manager is in charge of executing and scheduling workflows' tasks into the set of available resources (acquired by the Autonomic resource manager).

The resources can be selected according to a 'selection script' which can be defined for each concerned tasks. For example, different tasks from a single workflow may be executed in different types of resources depending of the needs. To do so, the selection script must be defined as part of the workflow tasks.

The workflows to execute can be created and managed from a dedicated WEB portal or through the REST APIs. They are stored into a virtual catalog which rely itself on the central database.

A workflow can be imported/exported into an existing XML format. Here is the description of the query to send in order to upload a new Workflow:

POST http://www.example.com/studio/workflows/

Header parameters

sessionid *string*	identifier of the user session

Body

Accept: *application/json*

```
{
    ←id:number
    ←metadata:string
    ←xml:string
    ←name:string
}
```

Returns

Content-Type: *application/json*

```
{
    ←name:string
    ←xml:string
    ←metadata:string
    ←id:number
}
```

To manage data intensive applications, the Autonomic Data-intensive Application Manager can be used as a meta-scheduler by deploying and managing underlying schedulers (e.g. MESOS clusters). In this case, the component will act as a high level location aware scheduler by deploying, configuring, and migrating sub-scheduling environments in specific locations (i.e. according to the type of resource). Therefore, all data processing tasks will not directly go through this component, but rather to the underlying scheduling environment which actually execute the tasks.

Placement decisions are computed by the Application Placement and Scheduling component (BtrPlace) and provided back to the Autonomic Data-intensive Application Manager.

The initial placement computation must be provided through the workflow itself by implementing selection scripts as described above.

Resources adaptation can be then issued by the Resources Adaptation Recommender by directly updating the scaling policy of the desired infrastructures (eg. increase/decrease the amount of deployed resources) for example. When new placement decisions must be taken, the computation must be done by BtrPlace according to existing and new placement constraints of the recommendation. The decision must then be acted either by submitting a new dedicated workflow (with specific parameters) or by updating the scaling policy of the selected infrastructures.

The format in which these data will be transferred between those three components still need to be defined as it is part of upcoming development and collaboration between ICCS, CNRS, and ActiveEon.

3.3.4 Fog Deployment & Monitoring

The Fog Deployment & Monitoring system is composed of two ProActive components (resp. ProActive Cloud Automation and ProActive Cloud Watch).

Cloud Automation

Cloud Automation is an interface that allows to easily submit workflows and retrieve notifications about their executions. It provides a WEB portal and a REST API that allows to submit workflows and retrieve notifications/events. POST and GET methods are provided to submit and retrieve workflows, here is an example of a workflow execution using the REST API (JSON format):

```
POST http://www.example.com/cloud-automation-service/serviceInstances

Header parameters:

SessionId: identifier of the user session
```

```
Content-Type: application/json

{
 "genericInfo":{
  "pca.service.model":"occi.infrastructure.compute",
  "pca.service.type":"infrastructure",
  "pca.service.name":"",
  "pca.service.description":"",
  "pca.action.type":"create",
  "pca.action.name":"",
  "pca.action.description":"",
  "pca.action.origin.states":"",
  "pca.action.icon":""
 },
 "variables":{}
}
```

Specific workflows can be then automatically submitted from custom events configured into the ProActive Cloud Watch component described below.

Cloud Watch

The monitoring system (i.e. Cloud Watch) is able to detect complex events and trigger user-specified actions according to predefined set of rules.

ProActive Cloud Watch contains 3 main parts: the Monitoring part, the Processing part (used for analyzing and planning) and the Action triggering part.

It can monitor all physical metrics, such as:

- System memory, swap, cpu, load average, uptime, logins.
- Per-process memory, cpu credential info, state, arguments, environment, open files.
- File system detection and metrics.
- Network interface detection, configuration information and metrics.
- Network route and connection tables.

The rules must be specified in JSON format. The design of the Fog Deployment & Monitoring system follows the paradigm of MVC (Model View Controller). When the rule is received in the system the parameters in JSON format is parsed to an internal model representation. A Rule Service is responsible for controlling and interacting with all the other internal services.

It allows to add, consult and modify the monitoring rules submitted to the platform (ProActive Cloud Watch) through the dedicated REST API. POST and GET methods are available to submit and retrieve rules respectively. Here is an example of a monitoring rule in JSON format:

```
POST http://www.example.com/cloud-automation-service/rules/

Header parameters:

SessionId: identifier of the user session

Content-Type: application/json

{
 "name": "ruleUrlMetric",
 "ruleContent": {
  "drl": "package org.ow2.proactive.cloud_watch.rules \n import java.util.*;  \n import
org.ow2.proactive.cloud_watch.model.*; \nimport  org.ow2.proactive.cloud_watch.service.*; \nglobal
org.ow2.proactive.cloud_watch.action.ActionExecutor actionExecutorCloudAutomationService;
\ndeclare isNodeAction \n nodeMetrics: NodeMetrics \nend \n \nrule ruleUrlMetric when\n exists
kpi : NodeMetrics( nodeUrl == \"localhost\" )\n then \n      Map<String, String> parameters = new
HashMap(); \n parameters.put(\"username\",\"someusername\"); \n
parameters.put(\"password\",\"somepassword\"); \n
actionExecutorCloudAutomationService.execute(parameters, \"ruleUrlMetric\", \"UrlAction\");
\nend \n"
 },
 "pollConfiguration": {
  "nodeInformations": [
   {
    "nodeURL": "localhost"
   }
  ],
  "kpis": [
   "sigar:Type=Cpu Total"
  ],
  "pollingPeriodInSeconds": "10",
  "calmPeriodInSeconds": "500"
 }
}
```

Here is the description of all the provided parameters:

- name - unique identifier of rule.
- ruleContent - final drl rule with all required imports and call to the special executor object
- pollConfiguration - contains info about monitoring resources
  - nodeInformations: the JMX of end point
  - kpis: metrics to poll
  - pollingPeriodInSeconds: frequency of monitoring in seconds
  - calmPeriodInSeconds: time delay for monitring after the action was triggered

Figure 19 illustrates the interactions with other PrEstoCloud components and actors. The relations between ActiveEon components have been simplified and the REST/WEB interfaces are highlighted for clarity.

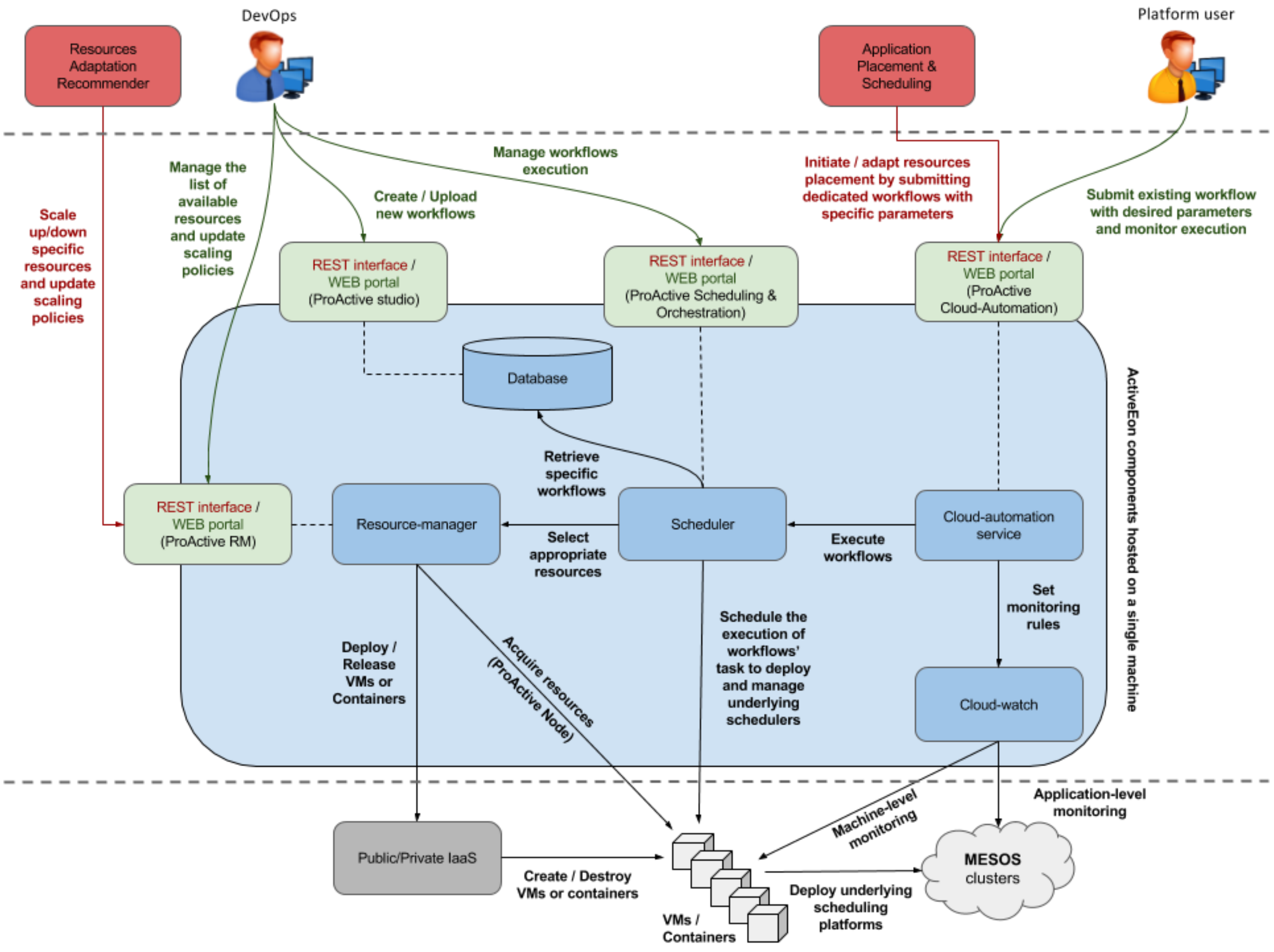

Resources Adaptation Recommender
DevOps
Application Placement & Scheduling
Platform user
Scale up/down specific resources and update scaling policies
Manage the list of available resources and update scaling policies
Create / Upload new workflows
Manage workflows execution
Initiate / adapt resources placement by submitting dedicated workflows with specific parameters
Submit existing workflow with desired parameters and monitor execution
REST interface / WEB portal (ProActive studio)
REST interface / WEB portal (ProActive Scheduling & Orchestration)
REST interface / WEB portal (ProActive Cloud-Automation)
Database
Retrieve specific workflows
REST interface / WEB portal (ProActive RM)
Resource-manager
Select appropriate resources
Scheduler
Execute workflows
Cloud-automation service
Set monitoring rules
Deploy / Release VMs or Containers
Acquire resources (ProActive Node)
Schedule the execution of workflows' task to deploy and manage underlying schedulers
Cloud-watch
ActiveEon components hosted on a single machine
Application-level monitoring
Machine-level monitoring
Public/Private IaaS
Create / Destroy VMs or containers
VMs / Containers
Deploy underlying scheduling platforms
MESOS clusters

Figure 19: The interactions with other PrEstoCloud components and actors

3.3.5 Planned developments

The ActiveEon's internal architecture (in blue) including interactions with external entities (in grey) already exist and don't require additional development.

Regarding the interactions with external components, the following developments are are planned for the PrEstoCloud project:

- Improve IaaS connectors to easily acquire Docker resources on edge devices including configuration of persistent data storage and networking redirections.
- Create custom workflows that will compose, deploy and configure specific MESOS clusters designed to manage intensive data flow processing.
- Create and design custom workflows that will update, migrate, and reconfigure already deployed MESOS clusters.
- Introduce a mapping between the monitoring of ActiveEon's acquired resources and the resources used by sub-schedulers.
- Define flexible scaling polic ies that allow to manually trigger scale up/down actions, most of them would be specifically oriented to specific PrEstoCloud use-cases. This would require to extend the REST API of the Resource manager.
- Additional development need to be done on the Resource manager's REST API to receive and update scaling policies of existing infrastructures. This is necessary to allow the Resources Adaptation Recommender component to periodically update scaling rules based on detected events.

3.4 Contribution to the data architecture: ICCS

3.4.1 Overview

The components of the PrEstoCloud Meta-Mgmt. layer that should "directly" interact with the available incoming events and data streams are the Situation Detection Mechanism, the Mobile Context analyzer and the Workload Predictor. The rest two components of this architectural layer (i.e. Resources Adaptation Recommender and Application Fragmentation & Deployment Recommender) will exploit the outputs of these three components, mentioned above, thus we do not expect (based on the current version of the conceptual architecture) to directly engage incoming data streams from any external sources. We note here that the "direct" interaction to external data sources has the meaning of subscribing to and aggregating either raw data or processed data (e.g. complex event patterns detected) from external data sources (e.g. UAVs, VMs etc.) through a dedicated event-based communication broker (that will be analyzed in terms of the deliverable D2.3).

Starting with the Mobile Context Analyzer which will be developed in WP3 under the responsibility of ICCS, we envision an interaction with data streams originated by any edge device relevant to PrEstoCloud. This corresponds to either health status data (e.g. battery level), use case specific data (e.g. video, audio, location) or metadata on the use case specific data (e.g. video file size, throughput etc.) acquired by edge devices like mobile phones, UAVs, other IoT devices. The latter type of data might be produced/calculated by an internal to PrEstoCloud component, in cases that it is not provided by the data source. The Mobile Context Analyzer component requires real time integration with such relevant input sources, following the architectural pattern presented in section 5.1 (figure 5.4). Based on this, the component should be able to subscribe to relevant edge devices streams in order to acquire through a dedicated communication broker all the necessary values of parameters that are needed for inferring the current context of the devices. For example, a framework such as netdata (https://github.com/firehol/netdata) can be installed in every edge device in order to produce and publish health status data. Independently from the framework used in order for the sources to publish such data to the PrEstoCloud framework, the Mobile Context Analyzer should be able to aggregate the necessary data following a commonly agreed format that is valuable for all the PrEstoCloud components. Such an indicative format to be used is presented in section 5.3.2 below. The procedures for acquiring the appropriate data format for PrEstoCloud, independently on which format was used by the data source will be part of D2.4. One last point for the Mobile Context Analyzer is possibility to store (when it is allowed) some of the raw or processed data in an internal store available to any component of the PrEstoCloud Meta-Mgmt. layer. We will examine the use of Elasticsearch[7] for persisting contextual data.

With respect to the Situation Detection Mechanism which will be developed in WP5 under the responsibility of ICCS, we envision an interaction with event and data streams originated by any cloud resource that is going to be managed by PrEstoCloud. This corresponds to either health status events and data (e.g. CPU usage), use case specific events and data (e.g. face recognition, gunfire detection) or metadata on the use case specific data (e.g. incoming stream throughput, quality etc.) acquired by private or public cloud resources (physical machines, VMs or containers). As above, the use of a framework such as netdata (https://github.com/firehol/netdata) can be installed in every VM node in order to produce and publish valuable events. The Situation Detection Mechanism requires real time integration with such relevant input sources, following the architectural pattern presented in section 5.1 (figure 5.4). Based on this, the component should be able to subscribe to relevant streams originated from live VM instances in order to acquire through a dedicated communication broker all the necessary values of parameters that are needed for detecting interesting or critical situations. The detection of such situations will be used as input by the Resources Adaptation Recommender in order to suggest meaningful reconfigurations of the used topology. The event and data sources in this case might involve either system or application specific metrics. The Situation Detection should be able to exploit the necessary events and data following a commonly agreed format that is valuable for the all the PrEstoCloud components. Such an indicative format to be used is presented in section 5.3.2 below.

[7] https://www.elastic.co/

3.4.2 Data format

Data from either cloud or edge devices that run Linux can be published to PrerstoCloud by monitoring tools like Netdata[8]. Netdata can collect over 1000 metrics and publish them with multiple methods and formats. Two commonly used methods are the use of the Graphite API (which is understood by Graphite, InfluxDB, KairosDB, Blueflood, ElasticSearch via logstash TCP input and the graphite codec, etc) or JSON-formatted documents. Two examples of a JSON-formatted metrics published by Netdata are the following:

```
// A JSON-formated event that describes the percentage of system CPU
available to the system with hostname "myhost1" at timestamp 1500892312
{"prefix":"netdata",
"hostname":"myhost1",
"chart_id":"system.cpu",
"chart_name":"system.cpu",
"chart_family":"cpu",
"chart_context":"system.cpu",
"chart_type":"system",
"units":"percentage",
"id":"guest_nice",
"name":"guest_nice",
"value":40,
"timestamp": 1500892312}
```

Listing 1: Example JSON message for CPU usage

```
// A JSON-formated event that informs about the number of active processes
of the system "myhost1" at timestamp 1500892312

{"prefix":"netdata",
"hostname":"myhost1",
"chart_id":"system.active_processes",
"chart_name":"system.active_processes",
"chart_family":"processes",
"chart_context":"system.active_processes",
"chart_type":"system",
"units":"processes",
"id":"active",
"name":"active",
"value":317,
"timestamp": 1500892312}
```

Listing 2: Example JSON message for active processes

There are also applications that can collect monitoring data from mobile devices. For example, the mobile application Device Analyzer[9] collects monitoring data from devices running Android 2.1 and higher. Device Analyzer can collect over 30 types of data like the battery level and voltage, cellular signal strength, the amount of data transferred over 3G and wifi, etc. Carat[10] runs on both Android and IOS mobile devices and collects metrics like what apps are running, the % battery remaining, memory and CPU utilization, the unique device ID, the battery state (e.g., plugged in), and the OS version and

[8] Netdata , https://github.com/firehol/netdata

[9] Device Analyzer , https://deviceanalyzer.cl.cam.ac.uk/

[10] Carat, http://carat.cs.berkeley.edu/

phone model. An example of an event in JSON format that contains information about the status of a mobile device could be like the following:

```
// A JSON-formated event that contains information about the mobile phone with IP 170.30.21.123 at timestamp 1500845312. CPU usage is 28%, memory usage is 56%, battery level is 51%, etc.

{
"IP":"170.30.21.123",
"cpu":28,
"memory":56,
"battery":51,
"data_in":234098,
"data_out":7783455,
"model":"JXX59005",
"timestamp": 1500845312}
```

Listing 3: Example JSON message for mobile device status

We can perceive, in the case of a mobile device that the application developer, in order to reduce the network load, chooses to pack more than one metrics in one event and to omit including details about the measurement units which may be retrieved from a commonly agreed and stored on a commonly used reference document.

Tools easily extensible with plugins like Logstash[11] and Fluentd[12] that can read, parse and transform different formats of monitoring data in real-time, can be used during the data ingestion and preprocessing stages of Mobile Context Analyzer and the Situation Detector Mechanism.

For example a logstash plugin can derive in real-time contextual information about the location of a mobile device with the use of a GeoIP database from the IP field and add it to the received event before sending it to the Situation Detection Mechanism. The event from the mobile device could be transformed as follows:

```
{
"IP":"170.30.21.123",
"geoip.continent_code":"EU",
"geoip.coordinates":[9,51],
"cpu":28,
"memory":5609213,
"battery":51,
"data_in":234098,
"data_out":7783455,
"model":"JXX59005",
"timestamp": 1500845312}
```

Listing 4: Enhanced JSON message for a mobile device

In order to have a common understanding of each data source it is necessary to describe it with a reference document that will contain information about the data format, the metadata, the semantics of each field and its possible values. Fields like the parameter name, the value, the timestamp that the

11 Logstash, https://www.elastic.co/products/logstash

12 Fluentd, http://www.fluentd.org/

value refers to, and a unique id that distinguishes the source of the data are necessary and cannot be omitted.

An example reference document that describes some important fields of an event may include the following:

Key	Type	Description
IP	A string representing an IP Address	The IP address of the mobile phone
contintent_code	String	The code that corresponds to the continent of
memory	Integer	Free memory of mobile phone (in bytes)
Cpu	Real	CPU utilization in percent (0-100).

Of course this reference document might also include additional aspects per event parameter that will assist on using an even more compact event format. For example the "data_in":234098 in the example event payload of listing 4, should be accompanied by the measurement unit (e.g. MBs). But if there is a commonly agreed reference document that includes measurement units (per parameter) this information doesn't have to be included in the event payload.

3.5 Contribution to the data architecture: CNRS

3.5.1 BtrPlace

BtrPlace shall be used by the meta-management layer to place tasks under constraints (e.g. hardware affinity, colocation of tasks on the same hardware or site, etc.).

In order to be able to find a viable placement for the jobs, BtrPlace relies on the following elements:

- a *model* of the hosting platform. This model contains the number of nodes on which tasks can be placed (e.g. physical servers), and the current status of usage of their primary resources (e.g. CPU use or pinning, physical RAM use);
- a set of *satisfaction constraints* that impose restrictions on the placement. Popular examples of such constraints are colocation of VMs (e.g. on the same physical server), or spreading of VMs (e.g. on different physical servers). When used in combination over multi-tier applications, they achieve a global high-availability of the system. The constraints can also apply on the infrastructure itself, e.g. enforcing a specific number of nodes to be online (reducing energy consumption), or some specific nodes to be offline (e.g. for maintenance), etc.;
- one *optimization constraint*, which is a special constraint that can be minimized or maximized. Popular examples are "use as few nodes as possible", or "minimize the mean time to repair", i.e. the amount of time that VMs will be unavailable during reconfiguration.

Given these elements and a *queue* of jobs to place, BtrPlace will produce a *reconfiguration plan*, i.e. a scheduled set of transition actions that either starts, migrates, or stops VMs, or boot or shutdown physical hardware. Once the reconfiguration plan has been applied (e.g. by the cloud manager software), the tasks placement has been optimized and meets all the specified constraints.

These elements can be given to BtrPlace using either a Java API, a JSON message, or a custom specification language, which is very descriptive and script-like. Documentation for these elements are available on the project website: http://www.btrplace.org/. An example of the specification language is shown in Listing 5: Example of the BtrPlace Specification Language (Hermenier, Lawall, & Muller, 2013)., where a data center with 12 servers is modeled. These servers are divided among 3 racks. A queue of 10 VMs must be placed. The 7 first VMs are based on the "small" VM template (which maps to a specific amount of CPU and RAM), can be cloned (i.e. can restarted without saving their state), and take 5 seconds to boot or stop. Similarly, the two other VMs are large VMs. The three first VMs are assigned to the first task, VMs 4-7 are assigned to the second task, and VMs 8-10 to the third task. Then the loop specifies that each VMs, within each task, must be *spread*, i.e. must be placed on distinct VMs from one another. The among constraint additionally forces the VMs for T3 to be located within the same server rack. Finally, the last part of the script instantiates the BtrPlace VM itself, which is set to be run on the first server, and

```
$servers = @srv[1..12];
$racks = {@srv[1..4], @srv[5..8], @srv[9..12]};

VM[1..7]: small<clone, boot=5, halt=5>;
VM[8..10]: large<clone, boot=60, halt=10>;
$T1 = {VM1, VM2, VM3};
$T2 = VM[4..7];
$T3 = VM[8..10];
for $t in $t[1..3] {
  spread($t);
}
among($T3, $racks);

vmBtrPlace: large;
fence(vmBtrPlace, @srv1);
lonely(vmBtrPlace);
ban($clients, @srv5);
```

Listing 5: Example of the BtrPlace Specification Language (Hermenier, Lawall, & Muller, 2013).

BtrPlace as part of the Meta-management layer shall be periodically requested to issue optimal placement solutions for jobs over the appropriate cloud or edge resources. In both cases of initial placement and placement re-configuration, the Meta-management layer will drive the decision process that will also engage BtrPlace. In the first case any placement constraints or preferences will be considered while in case of re-configurations (functional requirement #FR-39 in D2.2), the time and the reason for the required adaptations will be detected by the components of the same layer before invoking BtrPlace with the appropriate placement issue parameters (e.g. how many jobs to be re-configured, how many VM instances available etc.)

3.5.2 Inter-Site Network Virtualization

The Inter-Site Network Virtualization component will be responsible for creating the necessary network overlay between the multiple sites elected to execute tasks within the workload. Using this overlay, any node will be able to talk (i.e. exchange data) in an optimized and secure way with any other node that is part of the workflow.

The overlay network will be originally deployed alongside the first instantiation of the workflow by the Autonomic Resource Manager (see Section 5.3). The necessary steps for the creation of the overlay will be integrated into the Autonomic Resource Manager as a workflow of its own. The elements of this workflow will contain:

- A blueprint, or a template, for a virtual machine that will be instantiated alongside the workload infrastructure on each site. This virtual machine will serve as a network gateway device for all nodes that will deployed within the sites. These gateways will connect to each other, so as to provide the overlay.
- A function to receive an IP numbering plan. Because the IP network will be shared among multiple sites, it is necessary to ensure that there will not be any duplicate IP address over any node within any site. Consequently, the component shall be responsible for assigning either the direct (internal) IP address of the nodes, or establish a numbering plan that the Autonomic Resource Manager can rely on to avoid having duplicates. To provide maximum adaptability

this numbering plan should take into consideration constraints, such as IP subnets or addresses to avoid using, or range restriction.

On top of the initial deployment, the Inter-Site Network Virtualization component shall be required to establish new links, or destroy existing ones, depending on the need, when the infrastructure is adjusted during the execution of the task. At this point, if a new site is required, the gateway virtual machine shall be deployed, and the new site should be added to the numbering plan.

During the execution of the workload, the Inter-Site Network Virtualization component will be able to provide data related to the local environment of each site back to the Meta-Management Layer. This data will reflect information about the networking environment and usage:

- Available bandwidth: each site will be able to estimate the available bandwidth for communication with each other site. This estimation is based on active measurements campaigns launched between each site's gateway virtual machine. Because this estimation can be quite costly in terms of network transfers, which is problematic in the context of public clouds because users are charged per unit of data transferred, two measurement testbeds will be available. One will measure the available bandwidth using packet dispersion technique. This technique is advantageous because it uses statistical inference to estimate the available bandwidth, while transferring a low amount of data. However, the algorithms are quite sensitive to perturbations, which may lead to imprecise estimation. This kind of algorithm can be run in a loop, e.g., every 10 minutes, and return the last up-to-date information to the meta management layer. The other technique is well-known to end users, as it is the basis behind the popular SpeedTest website. Basically, it amounts to transferring data between two points, faster and faster, until reaching the maximum speed. While the resulting estimation is quite good (although usually over estimated, but with an accuracy of 90%), the amount of data transferred between the two points is at least in the order of hundreds of megabytes, reaching gigabytes on faster links. This testbed will be available for trigger between two sites (e.g. by the devops) from the platform so as to provide better insight in the deployment/redeployment phases, but could lead to high monetary operational costs if used without caution.
- Available capacity: each site will be able to estimate the available capacity. While the bandwidth reflects the available (instantaneous) speed between two points, the capacity estimates the maximal achievable bandwidth, without cross traffic, or waiting queues, etc. The value of the capacity is crucial when considering the performances (i.e. the bandwidth) of a network. A low bandwidth may be a transient event (e.g. due to cross traffic), or due to a low-capacity link. Given both the bandwidth and the capacity, it is easier to detect possible bottlenecks. The estimation of the capacity is possible with statistic estimation based on packet dispersion measurements. The measurement campaigns tend to require longer run times (tens of minutes), but generate low traffic (tens of megabytes).
- Instantaneous and cumulative inbound and outbound traffic speed and volume: these values return the instantaneous (i.e. current, or over a short period of time) transfer rate and volumes (both inbound and outbound, to and from each other site and the public Internet), as well as the corresponding average transfer speed and total transferred volumes since deployment at the local site. These values are collected with passive measurements, on the gateway virtual machines.

Armed with these metrics, the meta-management layer will be able to take better placement/deployment decisions.

3.6 Contribution to the data architecture: Nissatech

3.6.1 Overview

Figure 20 shows a high-level view on the communication between the data sources and the relevant processing pipeline. The architecture follows the Remote integration architecture as described in Section 5.2. In the case that the data source sends the data directly to the broker, our system can be adapted to the Real-time integration architecture.

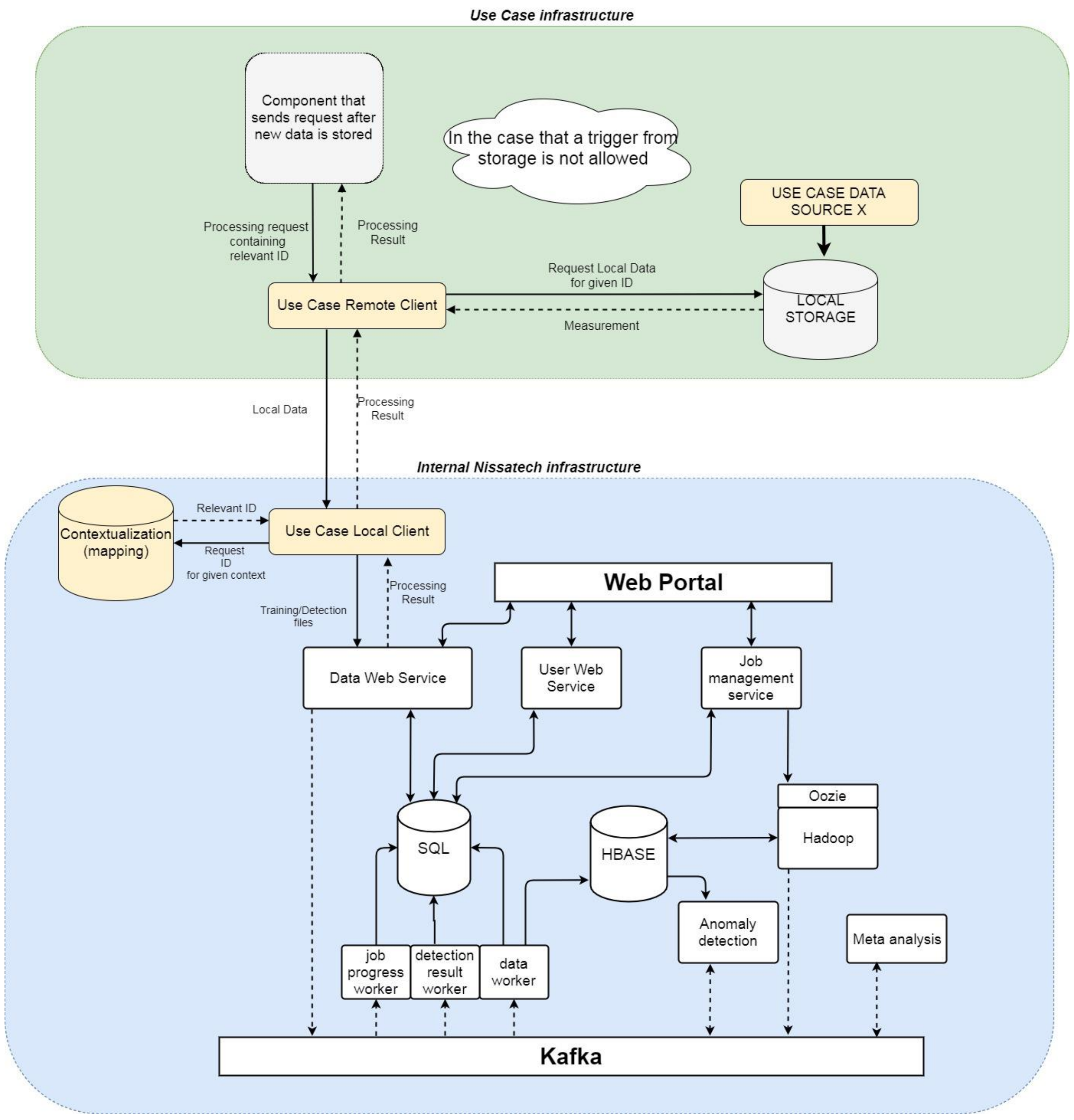


Figure 20: Data integration architecture

Below we include our responses regarding to the following reviewers' recommendation:

Comment: *"Figure 20 in Section 6.4.1 is not explained. What is the role of each component (Hbase, Hadoop…)?"*

Hadoop: Apache Hadoop is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. The most important components of Apache Hadoop are Hadoop MapReduce, which enables parallel processing of large data sets and HDFS (Hadoop Distributed File System), which is a distributed file system that provides high-throughput access to application data. We make a difference between Hadoop and Hadoop 2, better known as Hadoop YARN, which is a framework for job scheduling and cluster resource management, allowing applications other than Hadoop to execute on the same cluster. Hadoop has a master/slave architecture for both storage and processing. In a Hadoop cluster we identify:

- Name Node – HDFS master which maintains the name system (directories and files) and manages the blocks present on the Data Nodes;
- Secondary Name Node – responsible for performing periodic checkpoints and is a replacement for Name Node in case of its failure;
- Data Node – HDFS slaves which are deployed on each machine and provide the actual storage. They are responsible for serving read and write requests;
- Job Tracker – MapReduce master which manages the jobs and resources in the cluster (Task Trackers). Job Tracker tries to schedule each map as close to the actual data being processed;
- Task Tracker – MapReduce slaves deployed on each machine of the cluster. They are responsible for running map and reduce tasks as instructed by the Job Tracker.

HBase: Apache HBase is the Hadoop database, a distributed, scalable, big data store, modeled after Google's Bigtable, allowing random read/write on top of HDFS. It provides linear and modular scalability and strictly consistent reads and writes. Process of table sharding is automatic and configurable, as well as failover mechanism of RegionServers. It is very flexible, as each row might have different structure (different number of columns) and no schema is needed. HDFS is used for durable and reliable storage, distributed coordination is implemented using Apache Zookeeper and a built-in MapReduce support exists for running MapReduce jobs. HBase cluster consists of a Master and a number of Region servers. HBase master coordinates the HBase cluster and handles administrative operations (such as creation, deletion and modification of tables). Region server contains a number of Regions, where a Region represents a part of table's data. When a region is *full* (exceeds a limit), a split is triggered, and two regions are created. Being a NoSQL database, other solutions were developed to allow usage of SQL for querying data stored in HBase. One such example is Apache Pheonix, which compiles SQL query into a series of HBase scans and orchestrates running of those scans to produce regular JDBC result sets. HBase clients exist in different languages, such as Java, Python and Scala.

Oozie: Oozie is a workflow scheduler for Apache Hadoop jobs. Oozie workflows are defined as Directed Acyclical Graphs (DAGs) of actions, which means that there should be no loops in the workflow. Oozie Coordinator allows to schedule execution of Oozie Workflows based on time (frequency) or data availability. Client interacts with Oozie HTTP service which allows him to submit workflows which can be executed immediately or later. Workflows are defined using XML. Nodes in Oozie workflow can consist of action nodes (such as map-reduce jobs, java, pig jobs and shell scripts) and control nodes (such as start, end, decision, fork and join).

Kafka: Apache Kafka is a distributed stream processing platform capable of handling trillions of events a day. Kafka is used for building real-time data pipelines and streaming applications. It is horizontally scalable, fault-tolerant and incredibly fast. Its storage layer is essentially a massively scalable pub/sub message queue designed as a distributed transaction log. Those messages are persisted on disk and replicated within the cluster to prevent data loss. A single Kafka cluster can server as the central data backbone for a large organization because each broken can handle terabytes of messages without performance impact. It maintains feeds of messages in categories called topics. Processes which push messages to Kafka are called producers, while processes that subscribe to a topic and process the feed of messages are called consumers. Kafka is run as a cluster containing one or more servers, each called a *broker*. Topics are broken up into ordered commit logs called partitions. Each message in a partition has a sequential called an offset. Ordering of messages within a partition is guaranteed. Topics are replicated, where a unit of replication is the partition. Kafka is written in Scala and Java.

In the following text, we describe briefly the most important elements of the architecture.

1. Local Storage

This is local database where the (new) data from the data source is stored.
In the case that it is not allowed to get a trigger when new data is stored, another component is doing such a notification (the box colored in gray).

2. Use Case Remote Client

This client is implemented as service with endpoint for requesting processing of new data. When it receives processing request, *Remote Client* queries *Local Storage* using serial number in order to retrieve measurement data. After data is retrieved, it adjusts data format before delegating the request to *Local Client*. After getting an answer from *Local Client,* it returns processing result to it's client.

3. Use Case Local Client

This client is also implemented as a service with an endpoint for processing requests. As mentioned, it receives requests from a *Remote Client.* The request contains new data and some contextual data (like the model number). This context can be used for receiving more relevant information for the processing task. *Local Client* also contains endpoints for adding and removing mappings from this database.

4. Contextualization

Contains pairs of contextID-dataID. It can be altered using *Local Client* endpoints.

There are two basic steps in the analytics: training and detection.

- Training procedure - Training procedure had to be extended to support analysis which includes contextual parameters. Specific steps were needed to determine the influence of contextual parameters and to emphasize it.
- Real time anomaly detection - Anomaly detection is the most common type of processing, but any other learning method can be used.

3.6.2 Data format

We provide an example (JSON) in order to illustrate how data should be described:

```
"header": {
                "when": "1/1/2001"
                "where": "country",
                "who": "author",
                "desc": "A part"
        },
        "parameters": [{
                "name": "Power",
                "values": [{
                        "timestamp": "12/12/2012",
                        "value": 1.0
                }, {
                        "timestamp": "12/12/2013",
                        "value": 2.0
                }, {
                        "timestamp": "12/12/2014",
                        "value": 3.0
                }]
        }, {
                "name": "Current",
                "values": [{
                        "timestamp": "12/12/2015",
                        "value": 4.0
                }, {
                        "timestamp": "12/12/2016",
                        "value": 5.0
                }]
        }]
}
```

We assume that Local client will produce this format of data (to be sent to the Remote Client).

There can be different formats (e.g. more compact) for describing input data, as following:

```
"header": {
        "when": "1/1/2001"
        "where": "country",
        "who": "author",
        "desc": "A part"
        },
        "parameters": [{
                "name": "Current",
                "values": [4.0, 5.0, 6.0]
        }, {
                "name": "Power",
                "values": [1.0, 2.0]
        }]
```

3.7 Contribution to the data architecture: Ubitech

For providing a security enforcement overlay layer that can be used in cloud and edge setups, PrEstoCloud introduces a mechanism that handles protection on the network infrastructure level. The *Security Enforcement Mechanism* provides the appropriate business logic that is required in order to configure programmable resources such as SDN switches, Firewall, Intrusion Detection Systems(IDS), Intrusion Prevention Systems (IPS) following the Network Function Virtualization (NFV) model. In order to achieve this, the Security Enforcement Mechanism should be aware of the actual configuration and status of the network setup of the use cases.

As with the PrEstoCloud Meta-Management layer, for the components of the *Security Enforcement Mechanism* we do not expect to directly consume incoming data streams from any external sources provided by the use cases, in the sense of subscribing to and aggregating data (raw or processed) from external data sources (e.g. cameras, UAVs, VMs etc.). Based on the current, version of the conceptual architecture, the Security Enforcement Mechanism is expected to interact directly with components like the *Situation Detection Mechanism* of the Meta-Management layer and the *Autonomic Resource Manager* component.

The business logic of security configuration per each programmable resource cannot be generalized since each component entails a diverse 'northbound' API. However, the *Security Enforcement Mechanism* will interact with all the programmable components through an adaptation layer. However, already in this document the vision of how the data from external (use case) data sources will be collected, analyzed and stored has been provided, so we can envision how the Security Enforcement Mechanism will be able to access the required data from other components.

The integration regarding the use case infrastructure is depending on the way that the aforementioned components are communicating with the use case infrastructure in order to collect the needed input data. In the scope of integration of Security Enforcement with the components of the platform, the preferred way is to follow the local integration architecture paradigm by deploying the mechanism artefacts in the same infrastructure as the components, and when this is not possible to use the real integration paradigm in order to collect the required information using pub-sub mechanism. However, these details will be decided when the conceptual architecture will be finalized.

Based on this, the Security Enforcement Mechanism that will be able to access the required data of detecting interesting or critical situations as input in order to process or suggest to the user to make changes using SDN configurations and enhancements using NFV. The data format that will be used should be commonly agreed format with the *Situation Detection Mechanism* and the *Autonomic Resource Manager*.

For the *Autonomic Resource Manager,* there is already REST API described along with JSON examples (see Section 5.3). The Autonomic Resource manager component will be used by the Security Enforcement Mechanism in order to enhance the network security by deploying on demand VMs that provide firewalling, IDS and IPS functionalities, on the public or private cloud infrastructures’ of the use cases. For this reason, the Security Enforcement Mechanism shall communicate with Autonomic Resource manager in order to provide the required configurations about the VM to be deployed, using the REST interface that has been briefly presented in table in Section 5.3.2. The configuration parameters relate to the security goals that are desired.

The data input required by the security enforcement mechanism can be retrieved from *Situation Detector Mechanism* using a REST endpoint or a pub-sub mechanism. This information is used mostly to inform the user about interesting situations and if possible, this information will be processed in order to suggest possible changes on the network configuration using SDN and NFV models. The information that will be retrieved by both of these components will be stored in a relational database preserved for the Security Enforcement Mechanism needs.

It has to be stated that as the architecture is concretized and the actual PrEstoCloud is constructed, the information of this component may be subjected to changes. Furthermore, it will be further examined if the Cloud Automation and Cloud Watch components of Fog Deployment & Monitoring system could also provide valuable information that can be accessed by The Security Enforcement mechanism.

3.8 Contribution to the data architecture: JSI

The "Mobile offload processing" microservice will combine real-time local, remote edge node, and cloud resource and power usage metrics information (as described in the "Mobile context analyser" microservice description) and a semantic model of the computational process of the application, in order to provide the Autonomic Resources Manager with options for offloading portions of application processing from RTPN to the extreme edge of the network.

The microservice will not analyze the raw user plane data stream, and as such does not need generic data adaptation. Any application data that will be either acquired locally or transferred from a neighbouring edge node or the RTPN will be encapsulated in an application specific format, with no modification.

In order to seamlessly handle the imperfect availability of edge processing resources, all communication between the microservice and edge node monitoring probes will be asynchronous and specific to the type of the processing node (e.g. a user interface device in a truck, a drone, etc.). To support the communication, a message queue will be needed to connect the microservice and edge nodes, but disconnections and lost messages should be tolerated.

The CVS Mobile specific component for driving behaviour and situation monitoring will run partially on the edge nodes and partially in the cloud. For the on-demand processing of data in the RTPN, data will be transferred as a set of application specific packet data units through the message queue, and will not need adaptation.

Depending on realized overall system latencies, a HTTP REST transport mechanism, initiated by the edge node, might be employed to further reduce the requirements on transport reliability. Storage of intermediate results, such as models, is envisioned to be performed locally at the location of the RTPN portion of the microservice, where additional analytics will also occur.

3.9 Contribution to the data architecture: Software AG

3.9.1 Supported integration scenario and data sources

Real-time integration of data sources as described in the architecture introduction of this document are supported by Software AG's event processing offering "Apama". While real-time event streaming is the core competence of Apama, external static data stores can also be connected. Connectivity plugins provide a simple abstraction with high performance for in-process connectivity to external data sources.
A runtime engine called the Apama Correlator consumes the data sources, executes the desired application by looking for patterns and delivers insights and potential actions in real-time.
The stream processor which is the processing heart within the Correlator organizes windows of events and orchestrates the execution of real-time analytics over these event windows. Once an event pattern is identified, the initiating Apama application is notified. Applications can be provided in Standard Java or via the Apama Event Programming Language (EPL), a powerful and easy-to-use description language which has been designed around the requirements of defining and acting on event patterns.

Scalability
Correlators execute applications as modular compositions that are segmented into event monitoring, analysis and action stages, logically related but independent parts. This segmentation is particularly valuable in addressing the requirement to allow thousands of instances operating simultaneously. In order to achieve this, Apama supports the execution of multiple parallel executing contexts, where each context can be considered as a lightweight execution container. Each container can process any number of Apama applications which allows Apama to scale easily.

Connecting to Apama

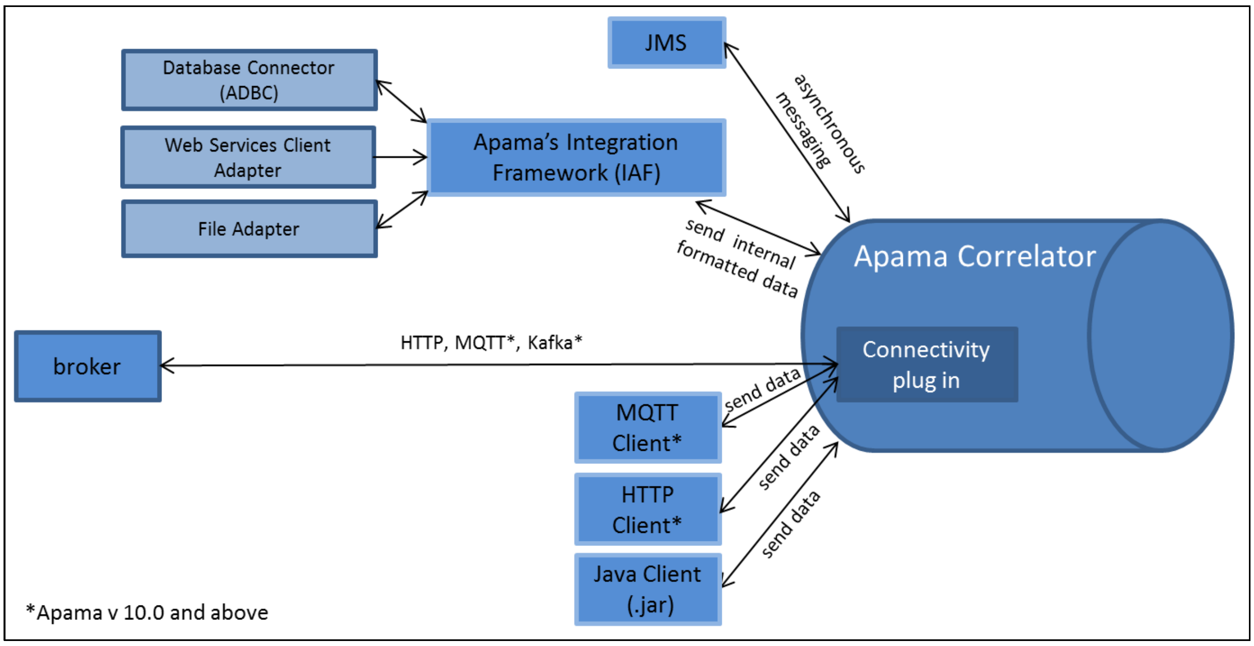


Figure 21: Apama connectivity overview

Apama connectivity via JMS

The Java Message Service (JMS) provides a common programming model for asynchronously sending events and data across enterprise messaging systems. JMS supports two models, "publish-and-subscribe" for one-to-many message delivery and "point-to-point" for one-to-one message delivery. Apama's correlator-integrated messaging for JMS supports both these models. When configured to use correlator-integrated messaging for JMS, Apama applications map incoming JMS messages to Apama events and map outgoing Apama events to JMS messages.

The connectivity examples 1-3 given in Appendix 1 - Apama connectivity examples show how to parse a JMS message whose body contains an XML document and map it to an Apama event called MyEvent:

- Example 1 shows a JMS message whose body contains an XML document.
- Example 2 shows the definition of an Apama MyEvent.
- Example 3 shows how the MyEvent would be mapped to an Apama event string.

3.9.2 Apama's Integration Framework (IAF)

The integration framework is a middleware-independent and protocol-neutral adapter tailoring framework designed to provide for the easy and straightforward creation of software adapters to interface Apama with middleware buses and other message sources. It provides facilities to generate adapters that can communicate with third-party messaging systems, extract and decode self-describing or schema-formatted messages, and flexibly transform them into Apama events. Vice-versa, Apama events can be transformed into the proprietary representations required by third-party messaging systems. It provides highly configurable and maintainable interfaces and semantic data transformations. An adapter generated with the IAF can be re-configured and upgraded at will, and in many cases, without having to restart it. Its dynamic plug-in loading mechanism allows a user to customize it to communicate with proprietary middleware buses and decode message formats.
All events, regardless of source, are converted to the internal format, thus enabling Apama to natively support correlations that span disparate external data formats.
The main Apama adapters of the integration framework are:

1.1 Database Connector

With the Apama Database Connector (ADBC), Apama applications can store and retrieve data in standard database formats. Data can be retrieved using the ADBCHelper API or the ADBC Event API to execute general database queries or data can be retrieved for playback purposes using the Apama Data Player.
The ADBCHelper API contains the basic features you need for most common use cases, such as opening and closing databases and executing SQL commands and queries.
The ADBC Event API contains features for more complex use cases. For example, in addition to opening and closing databases, it contains actions for discovering what data sources and databases are available.
ADBC adapters are available for three different data sources:

(1) JDBC Adapter (Apama Database Connector for JDBC)
    - Apama provides JDBC database drivers for DB2, Microsoft SQL Server an Oracle.

(2) ODBC Adapter (Apama Database Connector for ODBC)
    - Own ODBC drivers are requested in order to use ODBC. The use of JDBC rather than ODBC is recommended.

(3) Sim File Adapter (Apama Database Connector for Sim Files)
    - An Apama Sim data source is a file with data stored in a comma-delimited format with a .sim file extension. The Apama ADBC adapter can read .sim files but it does not store data in that format.

An ADBC example is given in Appendix 1 - Apama connectivity examples - see connectivity example 4.

1.2 Web Services Client Adapter

The Apama Web Services Client adapter is a SOAP-based adapter that allows Apama applications to invoke Web services. You can add multiple instances of the Web Services Client adapter to an Apama project.
Note that Apama applications only invoke Web Service operations in the consume use case; it is not possible to expose actions in Apama applications as Web Services operations.

1.3 File Adapter

The Apama File adapter uses the Apama Integration Adapter Framework (IAF) to read information from text files and write information to text files by means of Apama events. This lets you read files line-by-line from external applications or write formatted data as required by external applications.
The File adapter can read from multiple files at the same time. Send an OpenFileForReading event for each file you want the File adapter to read. This involves emitting an event to the channel specified in the adapter's configuration file, typically FILE, for example (see connectivity example 5 in Appendix 1 - Apama connectivity examples).

3.9.3 Connectivity Plug-Ins

Connectivity plug-ins allows adapters to be run in the same process as the correlator, providing a tightly coupled mechanism to send or receive events from external systems. A simple configuration file specifies the plug-ins used and provides powerful configuration data for plug-ins to use.
A number of connectivity plug-ins can be used together to form a chain of plug-ins in order to separate transports from mapping or decoding functionality.
Sample plug-ins provided include:

- JSON™ codec to convert Apama events to/ from JSON form
- A simple HTTP client
- A simple HTTP server
- Classifiers for identifying the type of events via configuration
- Mapper to provide rule-based transformation of event fields

An example configuration can be found in connectivity example 6 in Appendix 1 - Apama connectivity examples.

The JSON codec does bidirectional translation between JSON documents in the payload of a message on the transport side and an object in the payload of a message on the host side. The JSON document will be stored as a string in the message payload, and the corresponding object will be of a type that the apama.eventMap host plug-in can understand.

See connectivity example 7 in Appendix 1 - Apama connectivity examples for a JSON document.

This java.util.Map (Java) or map_t (C++) maps a string to an integer for one map entry and a string to a list of strings for the other entry. When the apama.eventMap host plug-in sees an object like this, it will be able to map it to/from an EPL (The Apama Event Processing Language (EPL) is an event-based scripting language that is an interface to the correlator) event type as shown in connectivity example 8 in Appendix 1 - Apama connectivity examples.

Connectivity plug-ins can be written in Java or C++, and run inside the correlator process to allow messages to be sent and received to/from external systems. Individual plug-ins are combined together to form *chains* that define the path of a message, with the correlator host process at one end and an external system or library at the other, and with an optional sequence of message mapping transformations between them.
A chain is a combination of plug-ins. Every chain consists of the following:

- **Codec plug-in**. Optionally, one or more codec plug-ins are responsible for applying transformations to the messages (for example, JSONCodec in the above example) to prepare them for the next plug-in in the chain.
- **Transport plug-in**. One transport plug-in is responsible for sending/receiving messages to/from an external system (for example, httpClient in the above example).
- **Host plug-in.** One built-in host plug-in is responsible for sending/receiving messages to/from the correlator process that is hosting the chain. These are built-in plug-ins (which do not need to be specified in the connectivityPlugins) which the correlator supports. Host plug-ins determine in which format events are passed in and out of the correlator. Thus, a chain should specify a host plug-in that is compatible with the next codec or transport element in the chain. Host plug-ins can also specify on which channel the chain receives events from the correlator, and can specify a default channel to send events in to the correlator (for example, apama.eventMap in the above example).

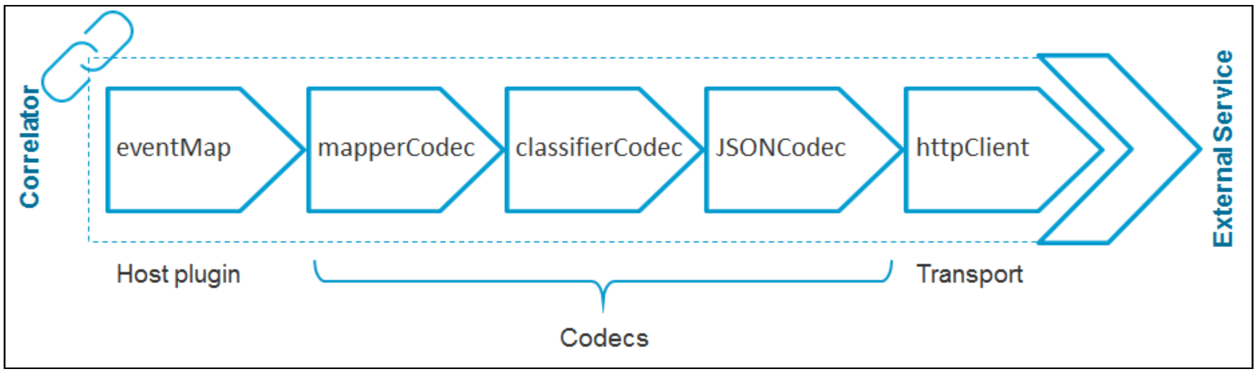


Figure 22: Example of the HTTP Chain Invoking External Services

Apama connectivity via MQTT Client

MQTT is a publish/subscribe-based "lightweight" message protocol designed for communication between constrained devices, for example, devices with limited network bandwidth or unreliable networks. Apama provides connectivity plug-in, the MQTT transport, which can be used to communicate between the correlator and an MQTT broker, where the MQTT broker uses topics to filter the messages. MQTT messages can be transformed to and from Apama events by listening for and sending events to channels such as mqtt:topic. The MQTT transport automatically reconnects in case of a connection failure. The transport will retry sending any messages sent after the connection has been lost when reconnection has succeeded. You configure the MQTT connectivity plug-in by editing the files that come with the MQTT bundle. The properties file defines the substitution variables that are used in the YAML configuration file which also comes with the bundle.

Using MQTT connectivity from EPL

The MQTT transport can either subscribe to or send to a particular topic, depending on whether your EPL is subscribing to or sending to a particular channel.
In EPL, in order to receive an MQTT message, you just need to subscribe to an MQTT topic with the appropriate prefix. For an example see Connectivity example 9 in Appendix 1 - Apama connectivity examples.

To send an Apama event to the MQTT broker, you just need to use the send...to statement to deliver the event to the MQTT topic. For an example see Connectivity example 10 in Appendix 1 - Apama connectivity examples.

Apama connectivity via HTTP Client

The HTTP client is a transport for use in connectivity plug-ins which can connect to external services over HTTP/REST, perform requests on them and return the response as an event. For each service (host and port combination) that you want to connect to, you must create a new instance of a connectivity chain in your configuration file. To use the service, you send events to that chain, where the events are correctly mapped. The response is sent back by the same chain instance, with the configured mapping rules. Persistent connections to the server are used for multiple requests if this is supported by the service. Connection details to the service are part of the configuration of the transport in the configuration file. Details of the individual requests are configured through the events sent to the chain.

The HTTP client accepts requests with metadata fields indicating how to make the request and a binary payload to be submitted as the body of the request. A response contains a binary payload which is the body of the response and further metadata fields describing the response. For the responses to be useful to EPL, they must be converted into the format expected by Apama. This is done using the codecs.

The HTTP client is designed to talk to REST services and supports GET, POST, PUT and DELETE operations.

See connectivity example 11 and 12 in Appendix 1 - Apama connectivity examples for a simple REST service with a single URL. The connectivity example 11 shows the events in EPL and the connectivity example 12 shows the service.

Each PUT request contains a request string which performs an action on the server and returns another string in the response.

Apama connectivity via Java Client

If your environment needs to interface programmatically with the correlator, Apama provides a suite of Client Software Development Kits. These allow you to write custom software applications that interface existing enterprise applications, event sources and event clients to the correlator. These custom applications can be written in Java or .NET. The following interface layers are available:

EngineClient API

The Apama EngineClient API is available in the following languages: Java and .NET. The EngineClient API provides capabilities to interact with an Apama engine, typically a correlator. It provides the ability to send and receive events, inject and delete EPL entities, connect Apama engines together, inspect what was injected into the correlator and monitor the status of the correlator. The Connectivity example 13 in Appendix 1 - Apama connectivity examples shows a Simple Java program to demonstrate interfacing with the Apama Correlator through the Java engine client API.

EventService API

The Apama EventService API is layered on top of the EngineClient API. The EventService API allows client applications to focus on events and channels.

The EventService API also provides an advanced mechanism to emulate request-response functionality. The EventServiceChannel object provides both synchronous and asynchronous mechanisms for request-response. With the synchronous mechanism, the client sends an event to the correlator and waits for the matching response event. With the asynchronous mechanism, the client sends an event to the correlator, and callback is invoked when a matching response event is received.

Apama connectivity via Broker

Apama version 9.12 and above can be used in combination with any broker which supports the following protocols and APIs:

- HTTP 1.1
- XML 1.0 ,XSLT, XML Namespaces, XPATH
- JAAS
- JDBC
- JMS
- REST
- SOAP 1.1
- WSDL 1.1
- PMML 4.2
- Fast 1.1-1.2
- WS-Secuirty 1.0
- WS Adressing FIX 4.1-4.4;5.0.

Apama version 10.0 in addition to the above supports:

- MQTT
- Kafka

- (HTTP (Client))
- TLS 1.0 and above.

All connectivity examples are given in Appendix 1 - Apama connectivity examples.

4. Data in use cases

4.1 Introduction

In this section we provide detailed information about the existing methods for accessing and storing data in each of the use cases.

The goal is to understand how the data, which is available in use cases, can be gathered and stored for the further processing.

There are two main issues clarified:

a) how this data can be accessed in real-time
   a. interface to access the data (method)
   b. format of representing data (e.g. JSON)

b) how this data will be stored (DB schema)

4.2 Wide Area Video Surveillance Use Case – ADITESS

4.2.1 Available data

This subsection contains basic information about the data which is available in this use case, partially described in D7.1 and D7.10.

Available datasets include audio files as part of a previous research. Video datasets will be produced during the implementation of the pilot testing. At the same time, we expect additional datasets based on OpenStack related log files. More information can be found in the relevant deliverable D7.10.

Under this scenario the pilot will produce a large number of video and audio files, as part of the testing data. We foresee that a subset of these video and audio files can be used by other researchers as data feeds, especially in the research area of video and audio analytics. Other useful data could be network and computing related datasets, as part of OpenStack implementation in support of the scenario.

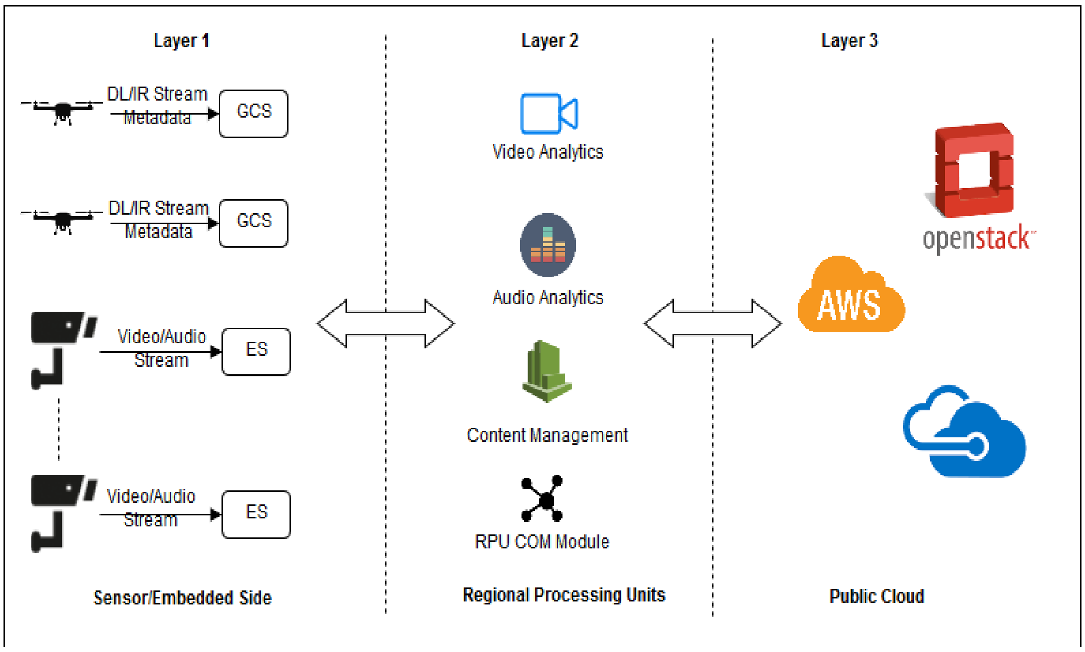


Figure 23: Types of data sources and processing (ADITESS)

In the video surveillance use-case, data will be collected from the sensors, and provided to the Embedded System (ES) or Ground Control Station (GCS) for the initial processing (Layer 1). Results of the processing, as well as raw data will be shared to the Regional Processing Units (RPU) (Layer 2) of the infrastructure. This layer will provide more computational and storage capabilities. Inter-exchanging of data between multiple RPUs is also available in the system. Furthermore, Public Cloud (e.g. OpenStack) is part of the infrastructure and can provide resources for further processing and storage.

Details about the data, including the source, the sharing level, as well as the availability are included below.

Type/Name of Data/parameter	**Data Description**	**Sharing level** [P – Public] [C – Consortium] [R – Restricted] **/Availability** [H – Historical data Exist] [F – Future]
Videos Files (CCTV)	Video files that will be produced as part of the pilot implementation.	C/F
Audio files (CCTV)	Audio files that will be produced as part of the pilot implementation.	P/F
OpenStack Monitoring files	Monitoring files provided by the REST APIs and/or logging mechanisms of the various components/services of OpenStack and produced during the implementation of the pilot scenarios.	C/F
Video Files (UAV)	Video files that will be produced by the usage of Mini-UAV systems. Also, existing video footage from previous flightscan be used as historical data	C/H,F
Mini-UAV Sensor Metadata (Video Embedded)	Video stream includes a set of metadata about the position of Mini-UAV, target location, etc. These data are available (embedded, KLV protocol) for each frame of the stream.	C/H,F
Mini-UAV Sensor Metadata	Metadata about the sensor, Mini-UAV position, target corners coordinates, field of view, etc. These data are available during the flight from a different (than video) stream of data. The frequency of data is about 100ms.	C/H,F
Thermal Image (and/or hyperspectral)	Thermal (and/or hyperspectral) video streams are available.	C/F
Mini-UAV Monitoring data	Status of the Mini-UAV (battery, telemetry, etc.) can be delivered in real-time from the aerial platform to the ground.	C/F

Table 1: Types of data, sharing level and availability (ADITESS)

In the next subsections we describe the ways how the data from external data sources can be accessed and stored.

4.2.2 Data access

In this section we provide details for accessing the data relevant for this use case. We distinguish between the possibilities:

(1) accessing the data in real-time (streaming interface)

(2) accessing data from the storage (DB interface)

There are two main sources:

1. CCTV
2. UAV

Relevant data-streams are:

- Videostream
- Audiostream
- Videostream (incl. metadata)
- Metadata
- Thermal videostream

Monitoring data:

- OpenStack Monitoring files
- ini-UAV Monitoring data

Following table summarizes the main characteristics of the available data sources.

Table 2: Details of the data access (ADITESS)

Type/Name of Data/parameter	**Data Description**	**Interfaces/ Format**	**Type stream / batch, frequency**	**Storage (if / how the data is / will be stored)**
Embedded System related datasets	CPU utilisation RAM utilisation Network (bytes sent) Disc storage	REST API (JSON, provided in Table 1)	Stream	Yes if needed
Videos Files (CCTV)	Video files that will be produced as part of the pilot implementation.	MKV Container (Embedded will send the file through API)	Stream (streaming is activated upon the detection of a security threat)	mongoDB or (File storage supported by OpenStack/Swift)
Audio files (CCTV)	Audio files that will be produced as part of the pilot implementation.	MKV Container (Embedded will send the file through API)	Not streamed. Local processing at the embedded system. (It is not allowed to record and store audio files according to EC data privacy regulations)	
OpenStack Monitoring files	Monitoring files provided by the REST APIs and/or logging mechanisms of the various components/services	Telemetry information collected by Ceilometer and provided through the REST API of	REST API calls can be used in decided frequency (e.g.: 30s)	Data stored by OpenStack should be sufficient.

	of OpenStack and produced during the implementation of the pilot scenarios.	Ceilometer or a service that pre-processes these data (e.g.: Gnocchi, Aodh or Monasca)		
Video Files (UAV)	Video files that will be produced by the usage of Mini-UAV systems. Also, existing video footage from previous flightscan be used as historical data	RTSP Video Server link (live stream during the flight)	Streaming /RTSP protocol	Local storage at ground Control Station (file system). They can also be stored (moved) to any database (e.g.mongoDB).
Mini-UAV Sensor Metadata (Video Embedded)	Video stream includes a set of metadata about the position of Mini-UAV, target location, etc. These data are available (embedded, KLV protocol) for each frame of the stream.	Embedded in the Video (RTSP link)	Streaming /RTSP protocol	Local storage at ground Control Station (file system). They can also be stored (moved) to any database (e.g.mongoDB).
Mini-UAV Sensor Metadata	Metadata about the sensor, Mini-UAV position, target corners coordinates, field of view, etc. These data are available during the flight from a different (than video) stream of data. The frequency of data is about 100ms.	JSON Format	Streaming data (tens of bytes) every 100ms	The source is SQL db (not accessible outside the GCS). We can send the data via Message Bus (i.e. Active MQ, DDS) or other end points.
Thermal Image (and/or hyperspectral)	Thermal (and/or hyperspectral) video streams are available.	Similar with Video Files (UAV)	Streaming /RTSP protocol	Local storage at ground Control Station (file system). They can also be stored (moved) to any database (e.g.mongoDB).
Mini-UAV Monitoring data	Status of the Mini-UAV (battery, telemetry, etc.) can be delivered in real-time from the aerial platform to the ground.	JSON Format	Streaming data (tens of bytes) every 100ms	We can send the data via Message Bus (i.e. Active MQ, DDS) or other end points.

An example (JSON) of the format used for accessing the data:

```
"header": {
            "when": "1/1/2001"
            "where": "country",
            "who": "author",
            "desc": "Some part"
      },
      "parameters": [{
            "name": "Power",
            "values": [{
                  "timestamp": "12/12/2012",
                  "value": 1.0
            }, {
                  "timestamp": "12/12/2013",
                  "value": 2.0
            }, {
                  "timestamp": "12/12/2014",
                        "value": 3.0
                }]
        }, {
                "name": "Current",
                "values": [{
                        "timestamp": "12/12/2015",
                        "value": 4.0
                }, {
                        "timestamp": "12/12/2016",
                        "value": 5.0
                }]
        }]
}
```

Table 3: Data source description, ADITESS use case

4.3 Media Use Cases – LiveU

4.3.1 Available data

This subsection contains basic information about the data which is available in this use case, partially described in D7.1 and D7.10.

The data available for the media use case is based on LiveU historical data that can be extracted to be used by the project. The following figures are examples that illustrate the need for dynamic resource management at the cloud.

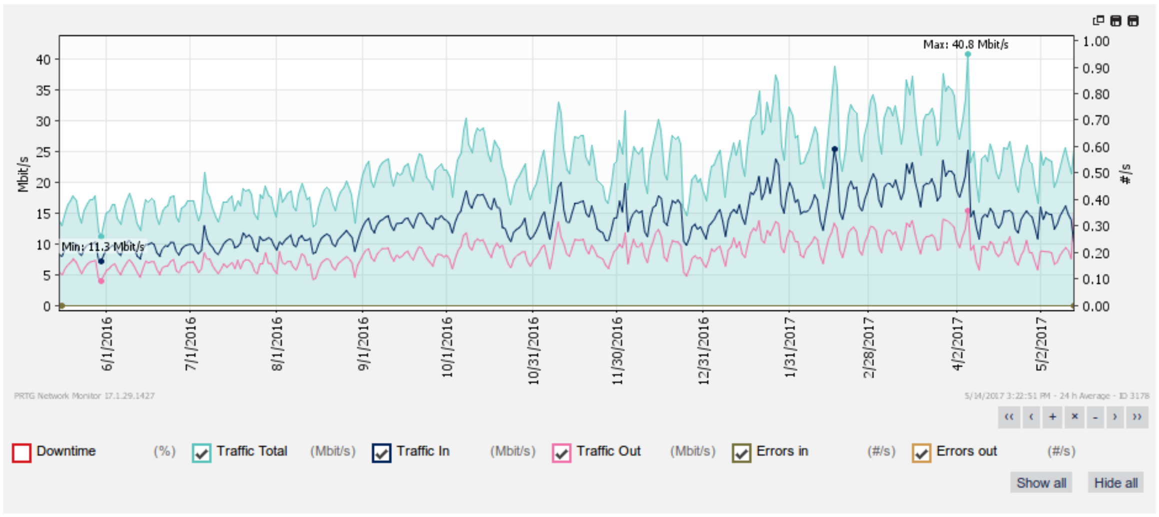


Figure 24: Data volume over one year

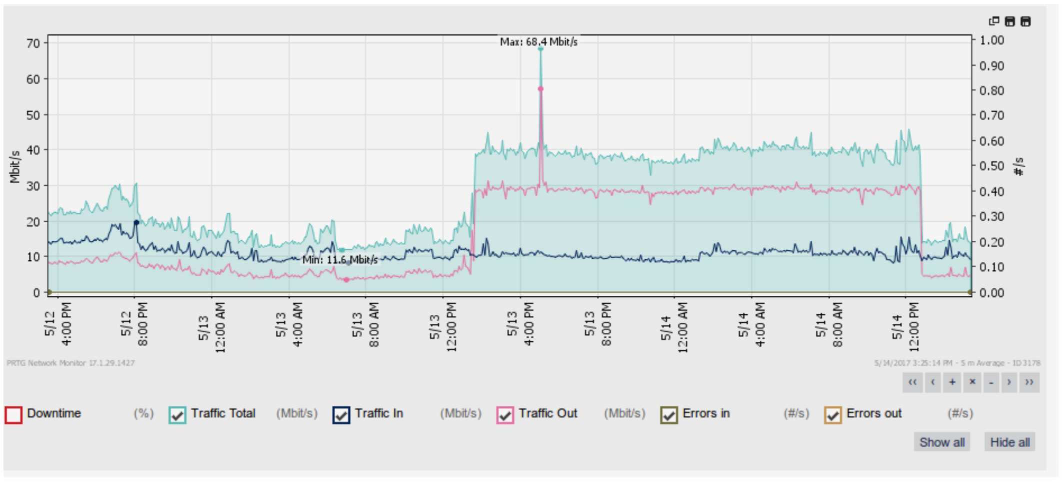


Figure 25: Data volume over two days

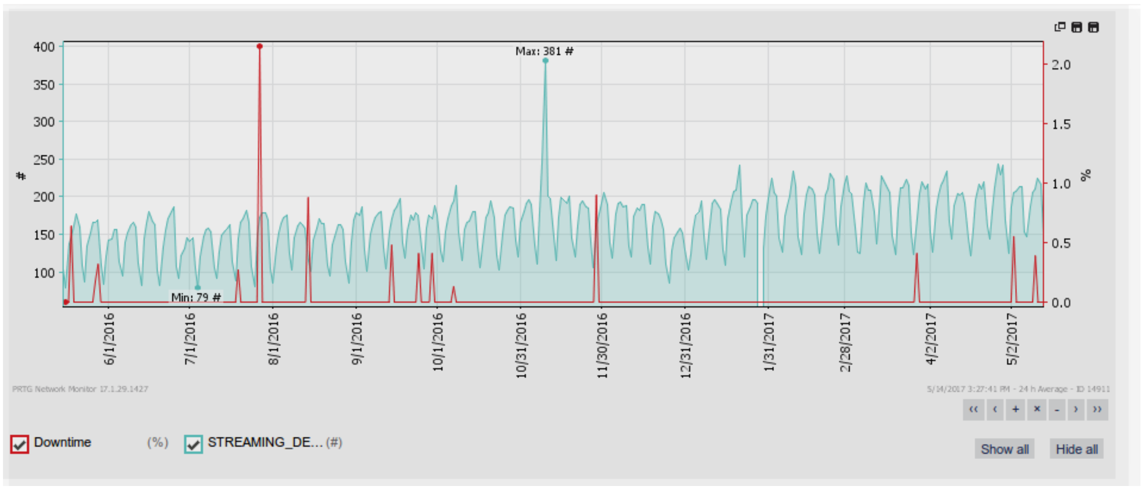


Figure 26: Number of Streaming devices over one year

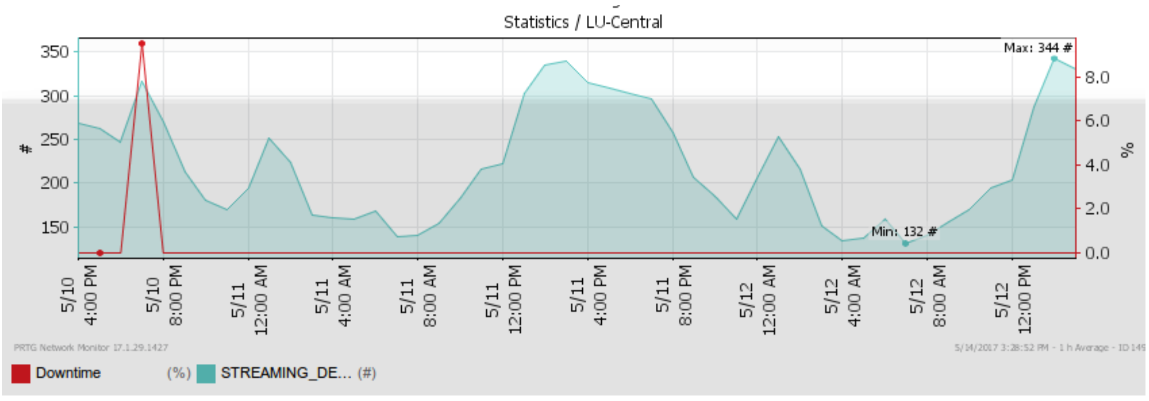


Figure 27: Number of Streaming devices over 2 days

The Media use case involves thousands of unites that are distributed all over the world, the units are referred as „LU“, in the following years to come with WebRTC technologies we expect larger amount of users using their browser and contributing data to the Media use case.

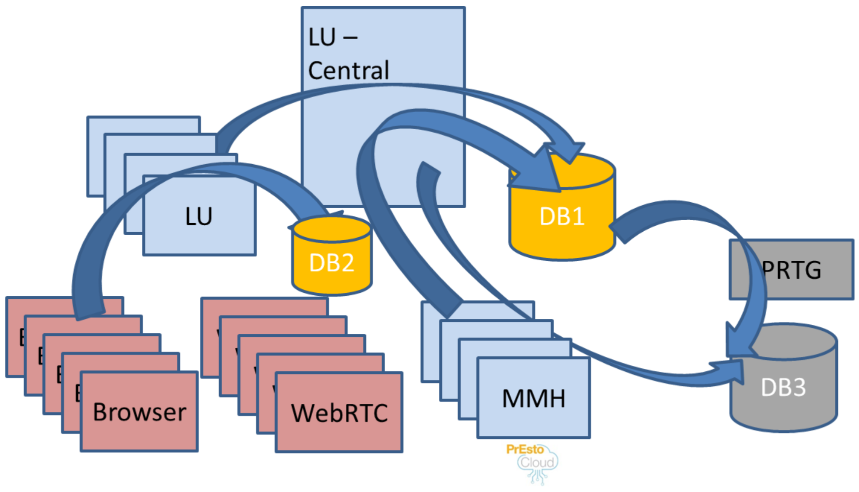
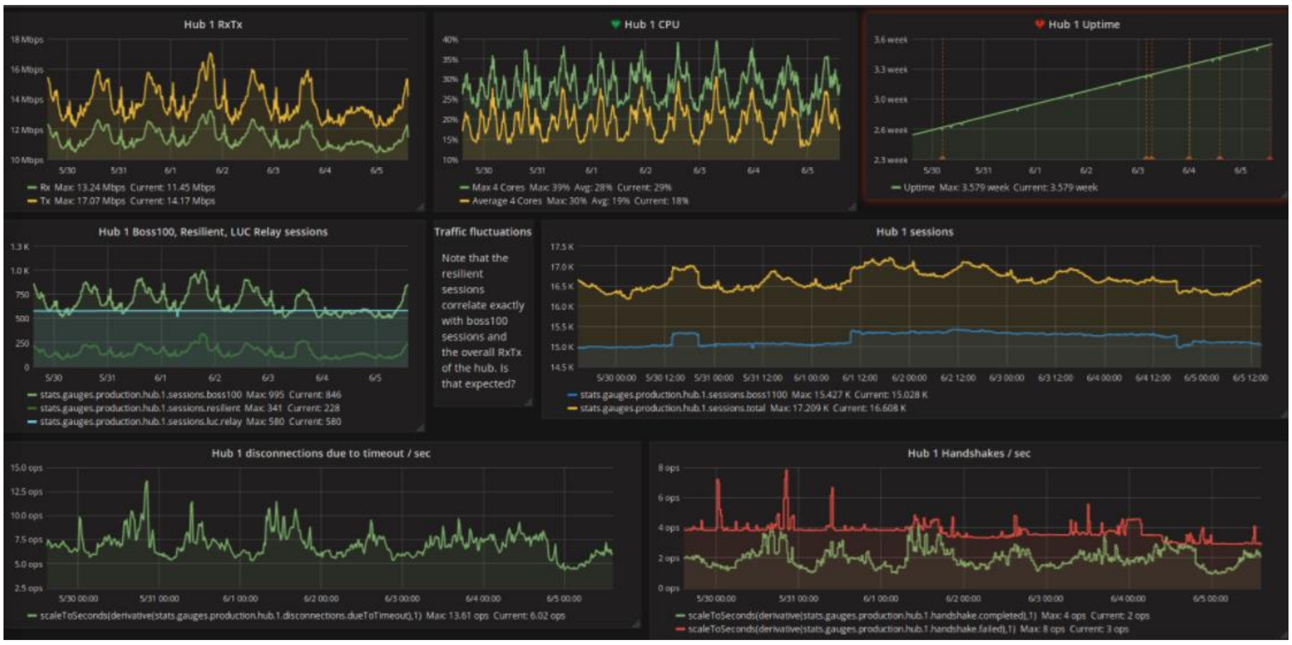


Figure 28: Media Use Case Data Sources

The data is all collected into 2 databases, where additional monitoring tools are used, these tools are also storing the data in different format and might be extracted from them as well (like the PRTG and DB3)
A short explanation on the sources of data we have today, the tools we use, and what type of data is gathered:

- LUC data base, data on devices and their sessions: LUC is the central hub of information for all media connection. Every device and services such as SOLO (do it yourself) provided by LiveU http://gosolo.tv/, community; video connection is done using the LUC. Statistics such as Traffic (in and out), Errors (in and out), streaming and downtime.
- PRTG, taking info via SNMP from central: PRTG is a service product of Paessler company (www.paessler.com), for network monitoring. PRTG can monitor all system and devices, traffic and applications of IT infrastructure. LiveU uses the PRTG to monitor the LUC statistics. PRTG can connect to the LUC via SNMP and SSH protocols.
- Grafana/Graphite, taking info from central via StatD: Grafana is an open source platform for visual analytics and monitoring. Graphite is an advanced query editor that allow quick navigation through metric spaces and functions.
- Google analytics, web sessions to the central, and more: Google Analytics is a freemium web analytics service offered by Google that tracks and reports website traffic. Google Analytics is now the most widely used web analytics service on the Internet.
- Amazon statistics: LiveU infrastructure is based on amazon Elastic Compute Cloud (EC2) services instances. Monitoring is an important part of maintaining the reliability, availability, and performance. Information such as CPU utilization, Memory utilization, Memory used, Network utilization and more are available.

The different tools and monitoring today are mainly used for problems identifications and for health monitoring of the servers. Example of samples information can be found here:



Note that the resilient sessions correlate exactly with boss100 sessions and the overall RxTx of the hub. Is that expected?

Figure 29: Media Use Case Data Samples

Data Type/Name of Data/parameter	Data Description	Sharing level [P – Public] [C – Consortium] [R – Restricted] /Availability [H – Historical data Exist] [F – Future]
NumberofSessions	The number of concurrent sessions the LiveU Central have open over time.	C/H
session_id	Session id number	C/H
session_start_date_timestamp	Time stamp of session start	C/H
unit_sn	Unit serial number	C/H
unit_type	Unit type	C/H
unit_name	Unit name	C/H
unit_sw_version	Version of software version used by the unit	C/H
transmission_type	Type of transmission the session established	C/H
service_type	Service type – media type	C/H
Resolution	Video resolution	C/H
Delay	Video delay	C/H
KbpsUpstreamAvg	Kbps for up stream	C/H
KbpsDownloadAvg	Kbps for down stream	C/H
total_upstream_bandwidth	Upstream bandwidth	C/H
last_session_fragment	C/H	
Interfaces: operator	Operator	C/H
Interfaces: type	Type of connection	C/H
Interfaces: id	Id number	C/H
Interfaces: roaming	If roaming is used	C/H
location	GPS coordinates	C/H

Table 4: Types of data, sharing level and availability (LiveU)

4.3.2 Data access

The data will be accessed via JSON files with the following format:

```
"session_fragments": [
   {
     "session_id": "3bac219e-66a1-445c-8748-bb892dfe0dca",
     "session_start_date_timestamp": 1494405307391,
     "session_start_date": "2017-05-10T08:35:07.391Z",
     "session_fragment_id": "671649",
     "last_session_fragment": false,
     "unit_BOSSID": "Boss100_0fe000091a3a5a01000000000100407a",
     "unit_sn": "501348-7472",
     "unit_type": "LU-500",
     "unit_name": "LUC_QA_LU500_7472",
     "unit_sw_version": "5.1(C10520.G2f6a21d)",
     "unit_primary_inventory_name": null,
     "unit_primary_inventory_id": null,
     "transmission_type": "streaming",
     "service_type": "video",
     "session_fragment_start_date_timestamp": 1494406803375,
     "session_fragment_end_date_timestamp": 1494410402951,
     "session_fragment_start_date": "2017-05-10T09:00:03.375Z",
     "session_fragment_end_date": "2017-05-10T10:00:02.951Z",
     "session_fragment_duration": 3600,
     "resolution": "720p50",
     "delay": "1600",
     "kbpsUpstreamAvg": 1316,
     "kbpsDownloadAvg": 0,
     "total_upstream_bandwidth": 4737090,
     "total_downstream_bandwidth": 0,
     "interfaces": [
      {
       "operator": "Ethernet",
       "type": "eth",
       "id": "1583286",
       "upstream": 4737090,
       "downstream": 0,
       "imsi": null,
       "iccid": null,
       "plmn": null,
       "roaming": null,
       "ssid": null,
       "vidPid": null
      },
      {
       "operator": "Vodafone",
       "type": "modem",
       "id": "1583287",
       "upstream": 0,
       "downstream": 0,
       "imsi": null,
       "iccid": null,
       "plmn": null,
       "roaming": null,
       "ssid": null,
       "vidPid": null
      }
     ],
     "location": null,
     "country": null,
     "latitude": 0,
  "longitude": 0,
  "samples": 637
 },
 {
  "session_id": "ed88fce4-9cc6-4466-aae9-c1c4e9cc9351",
  "session_start_date_timestamp": 1494410412334,
  "session_start_date": "2017-05-10T10:00:12.334Z",
  "session_fragment_id": "671650",
  "last_session_fragment": true,
  "unit_BOSSID": "Boss100_ios_5C1B81E8D6424CA39B0921BB35914A8C",
  "unit_sn": "",
  "unit_type": "LU-Smart",
  "unit_name": "akdrddrg",
  "unit_sw_version": "5.1(6.2.0-14(ios))",
  "unit_primary_inventory_name": "smartdev",
  "unit_primary_inventory_id": "smartdev",
  "transmission_type": "streaming",
  "service_type": "video",
  "session_fragment_start_date_timestamp": 1494410412334,
 "session_fragment_end_date_timestamp": 1494410412334,
  "session_fragment_start_date": "2017-05-10T10:00:12.334Z",
  "session_fragment_end_date": "2017-05-10T10:00:12.334Z",
  "session_fragment_duration": 0,
  "resolution": "720p (NTSC)",
  "delay": "2000",
  "kbpsUpstreamAvg": 274,
  "kbpsDownloadAvg": 0,
  "total_upstream_bandwidth": 274,
  "total_downstream_bandwidth": 0,
  "interfaces": [
   {
    "operator": "QA_MobileApps",
    "type": "wlan",
    "id": "1583288",
    "upstream": 274,
    "downstream": 0,
    "imsi": null,
    "iccid": null,
    "plmn": null,
    "roaming": null,
    "ssid": null,
    "vidPid": null
   }
  ],
  "location": null,
  "country": null,
  "latitude": 0,
  "longitude": 0,
  "samples": 1
 },
```

4.4. Logistics Use case – CVS

4.4.1 Available data

This subsection contains basic information about the data which is available in this use case, partially described in D7.1 and D7.10.

Core data available to the project is a stream of vehicle positions and the associated sensor data. Sensor data is read from the vehicle's CAN bus and processed on the primary vehicle interface unit, which has limited processing power and storage capabilities. Measurements are streamed to the cloud periodically, but with limited frequency, depending on driving conditions and behavior. Currently, sensor data is aggregated into a per-sensor histogram, and uploaded to the central database along with current position information.

The position information is already being streamed to a pair of databases prepared for the purposes of the PrEstoCloud project.

The histogram data is archived in an MS SQL server database at CVS Mobile, and is available as a database snapshot on request.

Higher resolution position and sensor data can be stored either to an on-board storage device, or streamed to the cloud on demand, but this is not enabled for any of the production systems.

We envision that majority of processing will occur on the vehicles user interface device, which can communicate with the vehicle interface unit, and will function as the edge device. In cases of higher computational load, processing will automatically migrate to the cloud.

For the extended use case, we intend to employ the vehicle's user interface unit's integrated cameras, to monitor the state in the cabin and on the road, perform preliminary analysis of data on vehicle, and offload the processing to the cloud in cases of emergency or uncertainty.

The following table lists the datasets available currently or in the future:

Type/name of data/parameter	Data description	Sharing level [Public/Consortium/Restricted], Availability [Historic / Future]
Vehicle sensor stream	This realtime stream contains measurements of vehicle sensors and the current position, direction and velocity.	C/H,F
Vehicle sensor archive	This database contains vehicle sensor histogram information, in high resolution	C/H,F
Vehicle video streams	Video that can be captured from the on-board user interface device	R/F

The following table lists details of each datasets:

Type/name of data/parameter	Interfaces/format	Type stream/batch, frequency	Storage (if/how data is to be stored)
Vehicle sensor stream	Binary stream from the vehicle to the central	Stream	Data is recorded in the primary MS SQL database, secondary PostgreSQL database at JSI, and a dedicated QMiner

	database, JSON over REST API from the central database towards other consumers		database instance
Vehicle sensor archive	MS SQL server archive	Batch	Data is already stored in the MS SQL instance at CVS Mobile, and exported as MS SQL server archive dumps
Vehicle video streams	H.264 stream from the built-in camera to the edge device CPU, H.264 stream embedded in MKV container from edge to cloud possible	Stream and batch	A restricted and time-limited form of storage will be implemented on the cloud side to enable machine learning; long term storage not needed

As some of the data from edge devices is already being archived and processed, we describe the detailed format of the data and methods for access here.

Format of the data stream coming from the current CVS Mobile services include three main data packets, described below.

Header

Field	Field Name	Type	Bytes	Value	Notes	Presence
1	Start of Message	Char	1	F7	Marks the start of the message	Mandatory
2	Provider ID	Integer	2	[1,32767]	98	Mandatory
3	Number of body records	Integer	2	[1,10000]	A higher number of records lowers the overhead and thus increases the performance.	Mandatory
4	Future Use	Byte	1	0x00	Reserved for future use	Mandatory
			Total bytes:	6		

Body

Field	Field Name	Type	Bytes	Value	Notes	Presence
1	VehicleID	Char	20	Unique identifier of a vehicle. The string of characters MUST be null terminated!		Mandatory
2	GPS Date Time	Integer	4	GMT Timestamp of the GPS reading in Seconds from 1/1/1970 00:00:00		Mandatory
3	Latitude	Integer	4	Event latitude in WGS84, decimal format given as an integer value with implied 6 decimal places		Mandatory
4	Longitude	Integer	4	Event Longitude in WGS84, decimal format given as an integer value with implied 6 decimal places		Mandatory
5	Speed	Integer	2	[0,250]	The speed of the vehicle in Km/h	Optional Specify 0 if not supported
6	Heading	Integer	2	[0,359]	Heading (North = 0)	Optional Specify 0 if not supported
7	Reserved	byte	3	0	000	
		Total bytes:	39			

Footer

Field	Field Name	Type	Bytes	Value	Notes	Presence
1	Message Terminator	Char	2	"Z\n"	ASCII Code for Capital 'Z' = 0x5A and control char for <LF> = 0X0A	Mandatory
		Total bytes:	2			

The next section describes how to access this data in a structured form.

4.4.2 Data access and storage

In this section we provide details for accessing the data relevant for this use case. We distinguish between the possibilities:

(1) accessing the data in real-time (streaming interface)

(2) accessing data from the storage (DB interface)

Real-time position stream

The binary data stream that is described in the above table is received and converted to a JSON based format using a simple data adapter. The format of the resulting JSON object is shown on the following listing:

```
{
        'locations': [
                {
                'latitude': <LATTITUDE IN WGS84 COORDINATES>,
                'longitude': <LONGITUDE IN WGS84 COORDINATES,
                'time': <UNIX TIMESTAMP IN MILISECONDS>,
                'activities': { 'in_truck': 100 }
                }
        ]
}
```

The "in_truck" parameter specifies that the vehicle in question is a truck, and is currently a fixed constant. This convention comes from the existing API constraint of the traffic.ijs.si service, which is the current primary consumer of this data stream on the project side.

The data stream is pushed to registered REST consumers using an HTTP GET request for each new JSON object. Additional payload of the REST query can be configured dynamically.

There are between 1000 and 2000 active vehicles in the test data stream, each transmitting position and associated sensor data every 2 minutes or more frequently in case of non-uniform driving.

The data is stored in a PostgreSQL database, from where it can be retrieved in the same JSON format using ODBC or direct PostgreSQL libraries. It is also streamed to the http://traffic.ijs.si/ service, which is the primary distribution hub.

Traffic.ijs.si service provides a REST endpoint for submitting and querying data. Current API documentation is available at https://jozefstefaninstitute.github.io/nextPin/server/spd/documentation/apidoc/index.html.

Archived sensor data

Sensor data that is stored in an MSSQL server database is reduced to periodic histograms from the original time-series format. There are currently 4 database tables available, listed in the following table:

Table name	Table description
Position data	Time-series data representing the vehicle position. These data points are not identical to the ones received from the real-time stream.
EcoDriver	Driving behavior metadata
FMSData	Sensor data retrieved from the vehicle's CAN bus
FuelSonda	Fuel gauge time-series data

All the tables are available as a periodic static dump file.

5. Use case Information Flow

This section focuses on the *use case data* flow as it is driven by the analytic demands. For each use case scenario the data flow is visualized and described in the context of its processing logic.

5.1 Surveillance use case

The surveillance use case covers the monitoring and steering of several camera clusters. The process is started by the security personnel who start the surveillance application.

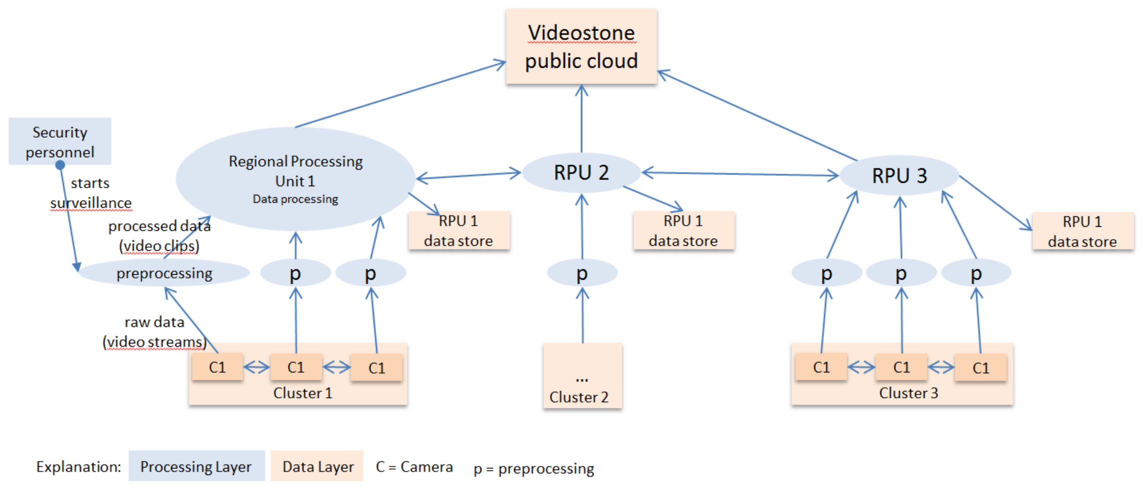


Figure 30: Information flow ADITESS use case

Each camera performs a preprocessing step on its real-time video streams and sends relevant clips to the Regional Processing Unit for further analysis and storage. Each Regional Processing Unit performs further analysis on its associated camera cluster. In case an exceptional issue has been detected, the Regional Processing Units share their data and processing power immediately in order to combine information from the various camera clusters. In case the data volume exceeds the capabilities of the Regional Processing Units, the use case data is transmitted to the cloud. Once the exceptional issue has been further specified and located, the cameras can be re-adjusted in real-time for further specific data collection.

5.2 Media use case

The media use case covers the collection and combination of several edge devices (e.g. professional back pack transmitters, registered social media feeds). The process is started by a media control center collecting feeds for a specific reporting event.

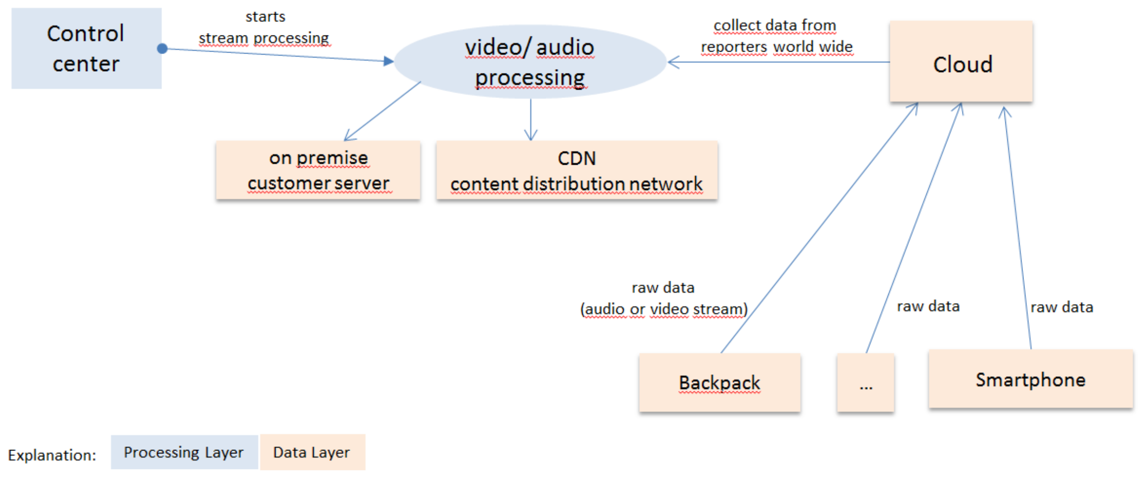


Figure 31: Information flow LiveU use case

Professional field reporters as well as private mobile device contributors provide their video/audio streams to media brokers. This raw data is directly streamed into a cloud. The video and audio processing logic at the media broker's site collects incoming streams and stores them either into an on premise customer server or feeds them into a content distribution network for further broadcasting processing.

5.3 Logistics use case

The logistics use case covers the monitoring and steering of a truck fleet. The truck driver starts the analytical process by starting the truck engine.

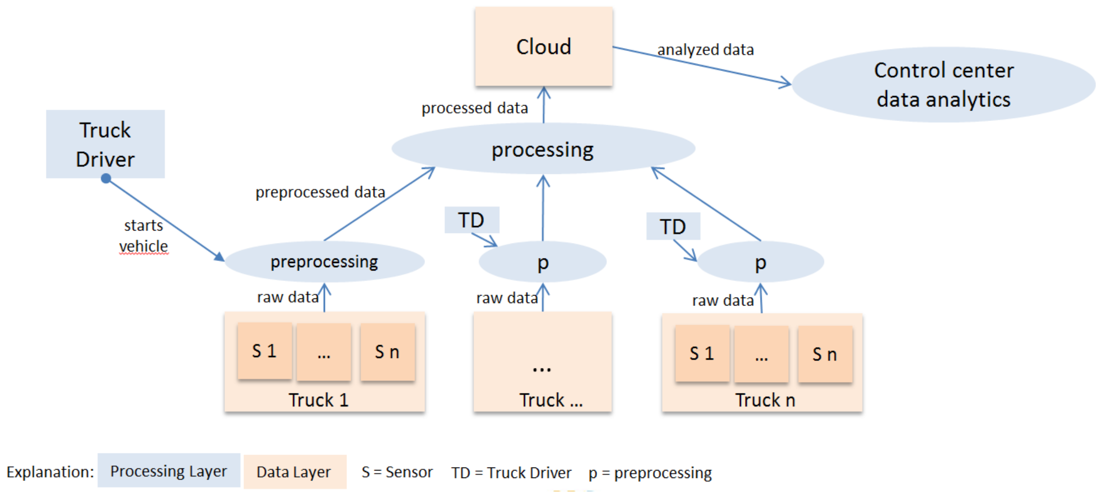


Figure 32: Information flow CVS use case

Each truck holds several sensors. The sensor data ("raw data") is collected and evaluated in a small preprocessing unit within the truck in a first step. In case of an exceptional issue (e.g. truck involved in an accident) the data is transmitted to the cloud. The cloud holds all preprocessed exceptional issue data for all trucks on the road. The data analytics center analyzes the cloud data for real-time fleet optimization options. In case an optimizing result has been identified (e.g. nearby truck can support), this can be automatically transmitted back to the trucks.

5.4 Use case data flow abstraction

In each use case we collect data from edge devices. They are being preprocessed either in the edge device itself or in a nearby small processing unit. The preprocessing result is then transmitted into the cloud for further real-time analysis and process steering. One of the possible outcomes of this real-time analysis is the trigger of a real-time action on the edge device.

6. Conclusion

In this deliverable we presented the outcomes from the work related to the analysis of the challenges for developing an efficient architecture for the data collection and preprocessing.

There are three main outcomes from this deliverable, which will be used in developing the conceptual architecture of the system:

1) the design of the architecture for data collection and processing

2) the requirements for the data adapters which are required by each component (technical partner) in order to get and understand in the proper functional context the data from available data sources

3) the specification of the nature of data source available in use cases, emphasizing their big data nature (which requires specific types of collection and processing).

There are several conclusions which will explicitly influence the development of the architecture:

1) edge processing is requested as a basic mechanism for processing data as close as possible to data sources (local processing)

2) fog computing architecture is desired in order to ensure the flexibility of the local processing

3) real-time data integration architecture is a preferable solution, but not supported by all components

4) data adapter will cope with the heterogeneity of the requirements, provided by components

5) deployment of the data architecture in use cases is challenging, but feasible.

Appendix

7. Appendix 1 - Apama connectivity examples

```
<?xml version='1.0' encoding='UTF-8'?>
<myroot xmlns:p='http://www.myco.com/dummy-namespace'>
<myelement1>An element value</myelement1>
<myelement2 myattribute='123' myboolattribute='true'>456</myelement2>
<ignoredElement>XML content that is not included in the event definition
is ignored</ignoredElement>
<e1>Hello</e1>
<e1>there</e1>
<e-2 e2att='value1'><subElement>e2-sub-value1</subElement></e-2>
<e-2 e2att='value2'><subElement>e2-sub-value2</subElement></e-2>
<e1>world</e1>
<namespacedElement xmlns='urn:xmlns:foobar'>My namespaced
text</namespacedElement>
<p:namespacedElement>My namespaced text 2</p:namespacedElement>
<namespacedElement>My non-namespaced text 3</namespacedElement>
<return>Element whose name is an EPL keyword</return>
</myroot>
```

Connectivity example 1: JMS message whose body contains the XML document

```
event MyElement2
{
string _myattribute;
boolean _myboolattribute;
string xmlTextNode;
}
event E2
{
string _e2att;
string subElement;
}
event MyRoot
{
string myelement1;
MyElement2 myelement2;
sequence<string> e1;
sequence<string> namespacedElement;
string #return;
sequence<E2> e$002d2;
}
event MyEvent
{
string destination;
MyRoot myroot;
}
```

Connectivity example 2: Apama event MyEvent

```
MyEvent("queue:MyQueue",
 MyRoot("An element value",
  MyElement2("123",true,"456"),
  ["Hello","there","world"],
  ["My namespaced text","My namespaced text 2","My non-namespaced text 3"],
  "Element whose name is an EPL keyword",
  [E2("value1","e2-sub-value1"),E2("value2","e2-sub-value2")]
))
```

Connectivity example 3: parsed to the Apama event string

```
Discovery adbc          := new Discovery;
        Connection conn         := new Connection;
        Query query             := new Query;

        float TIME_TO_WAIT      := 5.0; // seconds to wait for responses
        string COMMAND_STRING   := "DECLARE @myvar char(20)";
        string QUERY_STRING     := "SELECT * FROM sys.tables";
        integer BATCH_SIZE      := 5;
        string USER             := "mysql"; // Change to your database username
        string PASSWORD         := "mysql"; // Change to your database password

        // Change to true for parallel execution
        boolean parallelExecution := false;

        // Service ID for database to open.
        // Initialized by handleAvailableServers callback action
        string dbServiceId;

        // Name of database to open.
        // Initialized by handleAvailableDatabases callback action
        string dbName;

        // The current context when the monitor is loaded.
        // Initialized by onload action
        context preSpawnContext;

        action onload()
        {
                preSpawnContext := context.current(); // current context

                // Start the example by finding all available data sources.
                // The callback actions will perform the following:
                //
                //   The findAvailableDataSources callback action will call
                //   getDatabases to retrieve all databases for each data source found.
                //
                //   The getDatabases callback action will open the first database.
                //
                //   The openDatabase callback will run a command on the database.
                //
                //   The runCommand callback action will create a query.
                //   Set the query parameters, batchDoneListener and schemaListener.
                //   Run the query.
                //
                //   The runQuery callback action will close all queries on the database connection.
                //
                //   The closeAllQueries callback action will close the database connection.

                // The adapter is now running
                on AdapterUp() {
                        if parallelExecution {
                                adbc.initPreSpawnContext(preSpawnContext);
                                context discoveryContext := context("ADBC_Example");
                                spawn runExample() to discoveryContext;
                        }
                        else {
                                runExample();
                        }
                }
        }

        action runExample() {
                log "Finding Data Sources ..." at INFO;
                adbc.findAvailableDataSources(TIME_TO_WAIT, handleAvailableServers);

        }
```

Connectivity example 4: ADBC example

```
emit OpenFileForReading(...) to "FILE"
```

The OpenFileForReading event definition is as follows:

```
event OpenFileForReading
{  string   transportName;
   integer  requestId;
   string   codec;
   string   filename;
   integer  linesInHeader;
   string   acceptedLinePattern;
   dictionary<string, string> payload;
}
```

Connectivity example 5: File Adapter example

```
connectivityPlugins:
  JSONCodec:
    directory: ${APAMA_HOME}/lib/
    classpath:
        - json-codec.jar
    class: com.softwareag.connectivity.plugins.JSONCodec
  httpClient:
    libraryName: HTTPClientSample
    class: HTTPClient
  mapperCodec:
    libraryName: MapperCodec
    class: MapperCodec
startChains:
  weatherService:
    - apama.eventMap:
       defaultEventType: com.apamax.Weather
    - mapperCodec:
        "*":
          towardsTransport:
            defaultValue:
              metadata.http.path: /data/2.5/weather?q=Cambridge,uk
              metadata.http.method: GET
    - JSONCodec
    - httpClient:
        host: api.openweathermap.org
```

Connectivity example 6: Connectivity plugin example

```
{"a":2,"b":["x","y","z"]}
```

Connectivity example 7: JSON example

```
event E {
integer a;
sequence<string> b;
}
```

Connectivity example 8: event example from JSON

```
monitor.subscribe("mqtt:topic_a");
on all A() as a {
print a.toString();
}
```

Connectivity example 9: subscribe to an MQTT topic

```
send A("hello world") to "mqtt:topic_a";
```

Connectivity example 10: to send an Apama event to the MQTT broker

```
event PutData {
integer requestId;
string requestString;
}
event PutDataResponse {
integer requestId;
string responseString;
}
```

Connectivity example 11: Apama events

```
tartChains:
  SimpleRestService:
    - apama.eventMap:
        # Channel that responses are delivered on
        defaultChannel: SRS-response
    - mapperCodec:
        PutData: # requests
          towardsTransport:
            mapFrom:
```

```
          - metadata.requestId: payload.requestId
          - payload: payload.requestString
        defaultValue:
          - metadata.http.method: PUT
          - metadata.http.path: /path/to/service
        PutDataResponse:
          towardsHost:
            mapFrom:
               - payload.responseString: payload
               - payload.requestId: metadata.requestId

   - classifierCodec:
        rules:
          - PutDataResponse:
  - stringCodec
  - httpClient:
       host: foo.com
```

Connectivity example 12: Rest service

```
private static final Logger LOGGER = Logger.getLogger(EngineClientSample.class);
 static final Field<String> FIELD_TEXT = FieldTypes.STRING.newField("text");
 static final EventType EVENT_TEST_EVENT = new EventType("TestEvent", FIELD_TEXT);
 static final EventParser EVENT_PARSER = new EventParser(EVENT_TEST_EVENT);
    private void runSample(String host, int port) throws Exception{
      try(
           // Create the engine client (does not connect until first method requiring remove
connection is invoked)
           // EngineClientInterface.close() will automatically get called upon exiting the try-with-
resources block
          EngineClientInterface engineClient = EngineClientFactory.createEngineClient(host, port,
"my-sample-   process");
         ) {
            // Listen for events sent to the "samplechannel" channel
              ConsumerOperationsInterface myConsumer =
engineClient.addConsumer("myConsumer", "samplechannel");
              myConsumer.addEventListener(new EventListenerAdapter() {
               @Override
               public void handleEvent(Event evt) {
                 evt = EVENT_PARSER.parse(evt.getText());
                 System.out.println("Event listener received event with message:
'"+evt.getField(FIELD_TEXT)+"'");                            } });
           // Inject some MonitorScript
             MonitorScript epl = new MonitorScript(


               "event TestEvent {" +"\n" +
              "string text;" +"\n" +
               "}" +"\n" +
              "" +"\n" +
               "monitor Echo {" +"\n" +
               "" +"\n" +
              "TestEvent test;" +"\n" +
            "" +"\n" +
            "action onload {" +"\n" +
            "on all TestEvent(*):test {" +"\n" +
            "send TestEvent(test.text) to \"samplechannel\";" +"\n" +
            "}" +"\n" +
             "}" +"\n" +
             "}");
           engineClient.injectMonitorScript(epl);

        // Wait a few seconds to be sure the injection event has been processed
        Thread.sleep(3000);

        // Get and display status statistics
        EngineStatus status = engineClient.getRemoteStatus();
        System.out.println(status);

        // Send some events (in a real app we might use the EventType class
        // to construct these programatically)
        engineClient.sendEvents(
          new Event(EVENT_TEST_EVENT).setField(FIELD_TEXT, "Hello, World"),
          new Event(EVENT_TEST_EVENT).setField(FIELD_TEXT, "Welcome to Apama")
        );

        Thread.sleep(3000);

        // Delete the event type and monitor we added
        engineClient.deleteName("Echo", false);
        engineClient.deleteName("TestEvent", false);

        // Wait a few seconds for the output event to be received and the deletions processed
        Thread.sleep(3000);

        // Display status again
        status = engineClient.getRemoteStatus();
        System.out.println();
        System.out.println(status);
      }
}
public static void main(String[] argv) {

    if ((argv.length != 2) || (Integer.parseInt(argv[1]) <= 0)) {
      // Bad command line given
      System.out.println("Usage: java EngineClientSample <host> <port>");
      return;
     }

    String host = argv[0];
    int port = Integer.parseInt(argv[1]);
    try {
        new EngineClientSample().runSample(host, port);
      } catch (Exception ex) {
        LOGGER.error("Sample failed: ", ex);
      }
    }
```

Connectivity example 13: interfacing with the Apama Correlator through the Java engine client API