Scalable Understanding of Multilingual MediA (SUMMA)

H2020 Research and Innovation Action
Number: 688139
D6.1 – Platform architecture and API Modelling tools selection

Nature	Report (public)	**Work Package**	WP6
Due Date	31/7/2016	**Submission Date**	29/07/2016
Main authors	Guntis Barzdins, Renars Liepins, Didzis Gosko		
Co-authors			
Reviewers	Andreas Giefer, Steve Renals		
Keywords	Microservices, Swagger, Docker		
Version Control			
v0.1	**Status**	Draft	25/6/2016
v0.2	**Status**	Draft	12/7/2016
v1.0	**Status**	Final	25/7/2016
v1.1	**Status**	Final	28/7/2016

Contents

List of Figures

1 Introduction

This document describes the overall SUMMA Platform architecture, its design principles, and the selected API modelling tools.

The SUMMA Platform implements the complete system pipeline including Automatic Speech Recognition (ASR), Machine Translation (MT), Multilingual Named Entity Linking (NEL), relation extraction, Knowledge Data Base (KDB) construction, event clustering, topic detection, sentiment detection and story-line summarisation. It also provides the User eXperience (UX) interface with the relevant visualisations for news data enriched by the above mentioned Natural Language Processing (NLP) modules.

The SUMMA Platform is expected to have several versions during the lifetime of the project: it will start with the initial Common Demonstrator (M15) from which the use-case specific Internal and External Demonstrators will be derived afterwards. The use-cases will be elaborated on by WP1 and described in the deliverable D1.1: Use Case Description and Requirements (M6) and may lead to some adjustments to the architecture described here. The architecture is expected to be finalised by the MS4: Architecture and APIs of SUMMA platform agreed (M12). Issues relating to data management are discussed in more details in the Data Management Plan (D2.1).

1.1 Architecture goals

The SUMMA Platform architecture is constrained by the three core goals:

1. Integrate NLP tools from WP3, WP4, WP5 into the common pipeline for both batch and stream processing modes,
2. Provide UX interfaces as per requirements and use-cases elaborated in WP1,
3. Ensure BigData scalability to be evaluated in WP7 (ability to process 200–400 live streams).

To ensure that the proposed architecture is viable and meets the above goals, a baseline proof-of-concept SUMMA Platform prototype (Section 4) was implemented prior to writing this document.

2 Data sources

SUMMA Platform relies on data sources described in this section.

2.1 Deutche Welle data access API

DW provides an API (Giefer (2015)) that closely mirrors the content of the Deutche Welle website[1]. Accordingly, the API gives access to articles, video, audio and image galleries, as well as the search function.

The API is a RESTful API that returns JSON formatted data, the JSON format is mostly self-describing and can be compared to DW's website to recognize the corresponding structure (sections, stories etc.). The content is provided in multiple languages.

The configuration feed serves as the main entry point:
http://www.dw.com/api/config/init?product=dwapp&platform=ios&version=2.1.0
It provides a list of all available languages, including the language IDs, URLs to live streams (HLS) for various channels etc. It provides also the baseApiUrl values.

Each language has it's own entry page where the URL is formatted as follows:

```
http://{baseApiUrl}/api/navigation/{locale}
```

For example, the URL for the English entry page is
http://www.dw.com/api/navigation/en_GB
and the content of the returned JSON structure can be easily mapped to the DW webpage
http://www.dw.com/en
to identify the corresponding sections etc.

The search URL template is accessible at the main entry point key-path
"urlConfig.globalSearchUrlPattern" where search arguments must be filled in.

The RESTful API responses contain URLs for media files. The Video on Demand (VoD) files are in mp4 format. The live DW streams are in HLS format.

2.2 BBC data access API

BBC is building the monitoring data access API from scratch specifically for SUMMA as specified in the internal whitepaper (Sheppey (2016)).

BBC will provide live data streams, along with at least 100-hour data dumps, that are required by partners for testing and machine learning. The data in both data dump and live system will consist of:

1. Material essence files (video and audio).
2. JSON documents describing source material (metadata / side-car file) with embedded content in case of text-based data (social networks, websites etc.).

[1] http://www.dw.com

For delivery purposes the A/V data will be chunked into 60 minute segments for the data dump and 10 second segments for the live system. Text based data is expected to be in logical segments.

For each chunk of media source there will be a low quality 'preview' version of video material and a high quality audio for training, testing, and ASR purposes. Text-based sources will be 'scraped' to reduce the amount of noise present in data and anonymised by replacing all usernames with MD5 hashes.

The audio is de-multiplexed from the video and will be present in the data dump or live system media store as MPEG2 .TS files in AAC-HE 64 kb/s format, chunked into 60 minute segments in the case of data dump and 10 seconds for the live system.

As well as the audio files there will also be preview video files encoded as MPEG2 .TS files with audio in AAC-HE 64 kb/s format and video as H264 VBR at bitrates between approximately 1Mbps and 12Mbps depending on the source. These will be chunked and aligned with the audio described above at 60 minute intervals in the data dump and 10 second intervals in the live stream (intended for HLS transport).

All JSON metadata will be encoded in UTF-8.

The AV sources will initially consist of a maximum of 9 video streams and will be scaled to eventually being around 200 simultaneous sources. Text based material will be provided from various sources: Twitter, Facebook, Blogs, Webpages.

2.3 Intermediate layer: HLS storage

LETA is developing an intermediate layer – HLS storage for storing ongoing live stream data from DW, BBC, and LETA itself. The purpose of this intermediate AV storage is to enable live stream replay in GUI and live stream replay for testing and performance measurements of NLP modules. Intermediate HLS storage will also minimise the possibility of data loss in case of a crash or malfunction of any data processing infrastructure component. HLS storage is intended to be highly failure-proof due to simple and robust design described below. It will be able to query stored data from any point in the past up until the current time of the live stream. This will enable restarting data processing from any point in time, e.g. when the data flow pipeline experiences a failure.

The technical design of the HLS storage is as follows. The system consists of two independent parts joined only by the storage media (filesystem):

1. a "pull" process that downloads data from the given HLS source (identical to a standard HLS playback process supported by DW API) or from the BBC live stream API described above;
2. a "server" process that serves HLS data stored in the storage (filesystem) with extra query capabilities (time conditions).

Both parts are considered to be lightweight. Each media source channel will have its own pull process and the pulled data will be stored in a separate directory.

However, new server processes can be spawned on demand as the streaming (or "listening") requests rise.

For each channel the filesystem is structured in a directory tree with at least two sublevels (date and hour):

<channel base directory>/<date>/<hour>/<segment file>

The decision to use such directory tree structure was made to minimise directory scan time and organise segments into logical units. The date and time will be in UTC for time zone invariance.

The pull process writes segments directly to this directory tree, and appends a single line per each segment to a separate segment-list file that resides in the base directory. Each line contains the following information about the segment file: sequence number, date-time, duration and relative path to the segment file. This is required as each segment can have a different duration.

The serving process generates the HLS index file dynamically either by traversing the directory tree and watching for changes or more optimally by reading and watching for changes of the segment list file.

It will be possible to query the HLS storage for a specified time window in the past or for the current moment to receive a live-stream.

The decision to use a filesystem as a database was made to ensure the simplest and safest in terms of data loss design, as the only failures can arise during downloading from the source or writing to the filesystem.

It is recommended that the live-stream HLS index contains date-time tag EXT-X-PROGRAM-DATE-TIME[2] to avoid time shift because of HLS processing delays.

[2] https://tools.ietf.org/html/draft-pantos-http-live-streaming-13#page-9

3 Design Principles and Modelling Tools

The project use-cases are centred around BBC and DW which are large international broadcasters with strong internal software development and integration departments, and massive deployment and maintenance experience. Therefore the following best-practice SUMMA Platform design principles were suggested by BBC and DW and further elaborated be LETA as the main integration partner:

1. *Microservices* (Newman (2015)) should be used for building a highly modular and distributed software architecture where project partners take responsibility for various services comprising the overall system.
2. *Docker*[3] should be used for virtualisation and replication of the computationally intensive NLP services to achieve BigData scalability.
3. *RESTful APIs* (Mulloy (2012)) should be used as the primary interconnection approach between the various services.
    - *Swagger*[4] should be used as the primary modelling and documentation tool for RESTful APIs.
    - *JSON* (Google (2016)) should be used as the primary data interchange format by RESTful APIs. This facilitates data exchange beteen the services through no-SQL JSON databases, such as RethinkDB[5].
4. *WebSockets API* (IBM Watson (2016)) should be used only for supporting streaming data processing mode.

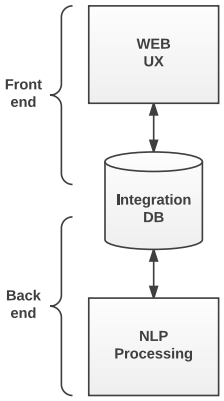
These design principles differ from the ones mentioned in the original SUMMA project proposal where BigData scalability was envisioned through the use of Apache Spark and HDFS architectures. The architecture shift is caused by the massive recent uptake of the more flexible Microservices interconnect and Docker virtualisation technologies which allow to achieve BigData scalability by containerisation of the existing NLP software packages rather than rewriting them for Apache Spark and HDFS environment.

Swagger is a simple and yet powerful modelling and documentation representation for RESTful APIs. As the most popular framework of its kind[6] it has gained immediate acceptance among the SUMMA project partners and has been used successfully to communicate RESTful API specifications among the project partners. Speedy development of the initial SUMMA Platform prototype (M4 instead of the planned M12) confirms appropriateness of Swagger as a RESTful API modelling tool within the SUMMA project.

[3] https://www.docker.com
[4] http://swagger.io
[5] https://www.rethinkdb.com
[6] Closest alternatives are RESTful API Modeling Language (RAML) and API Blueprint

4 Baseline Architecture (Single PC)

On the high level, the SUMMA Platform consists of two parts: the web front-end UX; and the back-end NLP processing pipeline. The two parts intersect through a common DB. The DB stores individual news items together with all extra information about the items that are produced by NLP modules. Each NLP module can process only a specific type of data. E.g. the ASR module produces a transcript from an audio file. The NLP modules are coordinated through the shared DB. When a news item gets a JSON-property that is processable by a given NLP module, then the platform passes the JSON-property to the module for processing and stores the result in a new JSON-property of the news item. The new JSON-property then is passed further to the appropriate NLP modules. The process continues until there are no more applicable NLP modules. Meanwhile, the front-end UX displays the available data about the news item as the data becomes available.



4.1 Overview

The SUMMA Platform (see Figure 1) is a platform for aggregating and analysing various news items (text, audio, video). The Platform consists of multiple NLP modules (ASR, MT, NEL, KDB, etc.) each of them performing a discrete task in the pipeline. Each module is developed independently by a team that is focused on that module. The goal of the Baseline Architecture is to provide a maximum independence so that each team is free to choose whatever technologies are most appropriate. The only constraint is that the module honours the API contract.

All news items are stored in a central DB together with metadata that describes the current state of the news items with respect to what NLP services have been applied to it. Each module has specified JSON-properties that it can process. For example, the ASR module specifies that it can transform audio to text. Thus the integration platform monitors the metadata state of each news item. If the metadata indicates that a news item has audio but does not have the corresponding transcript then the platform enqueues the audio file of the news item for asynchronous processing by the ASR module. The asynchronous queue ensures that there can be multiple ASR modules operating at the same time to satisfy current load, provided there are spare resources. Moreover, each NLP module can be deployed to use the hardware that best matches a modules' resource requirements. For example, the platform can deploy a computation-intensive speech recognition module on an instance with dedicated GPU and deploy a Knowledgebase construction service on a Memory-optimised instance.

The highly distributed architecture provides many benefits. However, there are some costs too. The inter-module communication could lead to tight coupling between modules, and the developers of the NLP modules would need to write code to handle partial failure in case the destination of a request is slow or unavailable. To ease the development of the NLP modules the integration Platform handles all communication between the modules. For example, if an Arabic audio broadcast news item becomes available and the end-user wants to see a transcript in English then multiple steps must be taken to achieve that. Namely, first, the audio needs to be transcribed to Arabic text (ASR), then the Arabic text needs to be translated into English (MT). Thus it would seem that the MT module needs to call the ASR module if the news item contains only Arabic audio. This would lead to tight coupling between the MT and ASR modules. Furthermore, the developers of

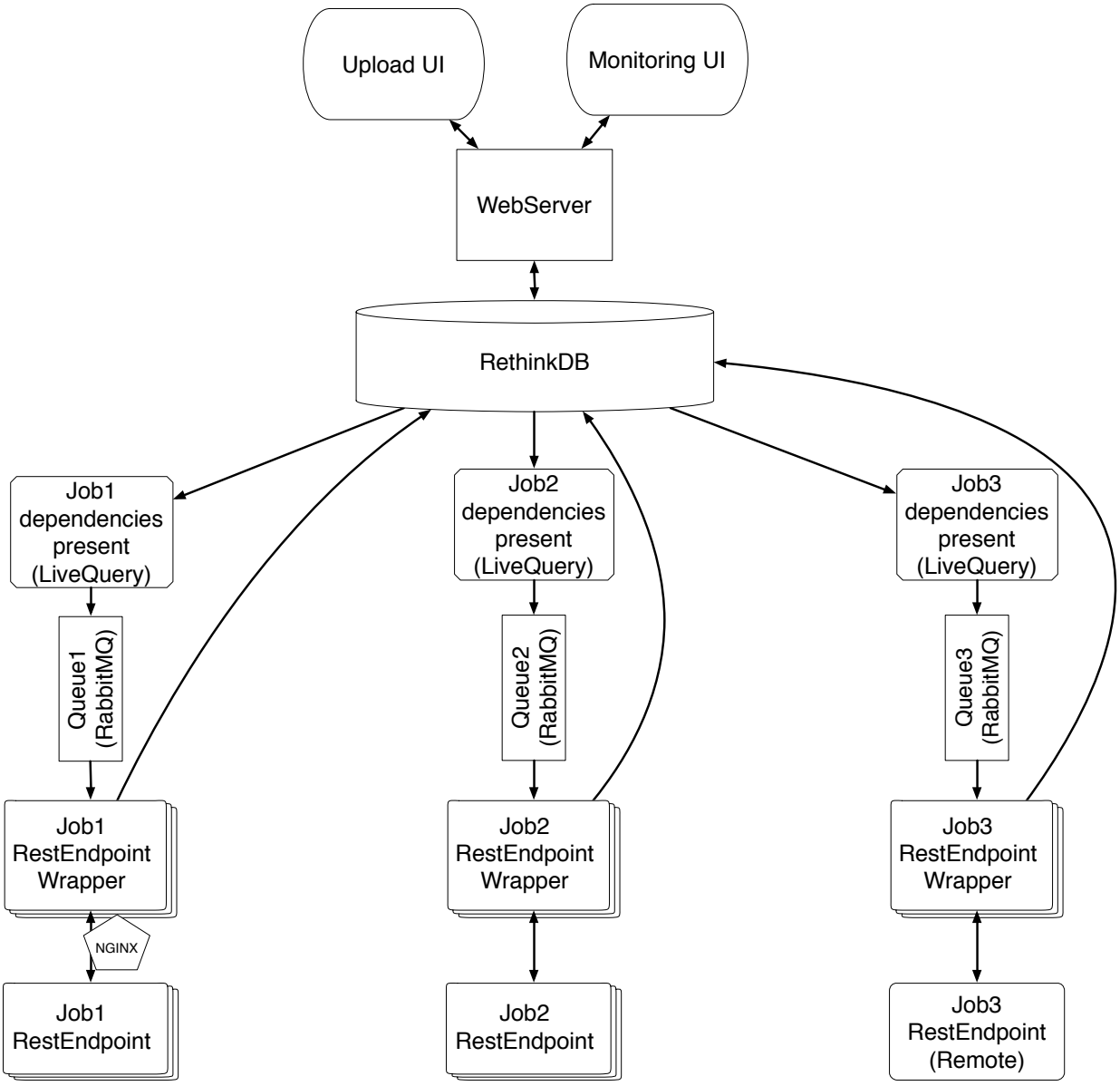


Figure 1: Integration architecture v1.0

MT would have to write code to handle partial failure since the ASR module might be slow or unavailable at the current moment.

To relieve the NLP module developers, the Platform architecture instead takes the responsibility of calling each module when necessary providing it only with the data that the NLP module needs. The Platform then aggregates all the responses of the modules inside the metadata database. If after the aggregation some additional NLP modules can be applied to the newly created JSON-properties, then the platform calls them and repeats the process until all possible NLP modules have been applied.

The only exception to the above approach is when several NLP modules or their replicated instances need to access a shared persistent database. In this case the NLP modules should access the database module directly. The database module must be provided as a Docker image. The platform will provide a database network address as an environment variable to each NLP instance at startup.

The persistent storage for NLP module instances is provided by mapping a docker instance folder to the Platform host file system. The persistent folder can be preloaded with NLP module-specific content such as language models.

The monitoring UX then can visualise all currently available NLP results for the news items of interest to the end user.

To ensure scalability, the integration Platform does not call the NLP modules directly. Instead the call is split into three parts. The first part is a monitoring component that monitors the DB for changes to the applicable JSON-properties of the news items. When a change is detected then the monitoring component pushes the value of the JSON-property into the second component – a message queue – which stores the values that need to be processed by the given type of NLP module. Finally, the third component – an NLP module wrapper – subscribes to the message queue, takes a single JSON property from the queue, passes it to the NLP module through the REST API, and updates the DB with the result. For NLP modules with the single execution thread the wrapper also ensures that individual NLP module instance is fed with the next task only after it has finished processing the previous task; for NLP modules with multiple execution threads the wrapper can distribute the load to multiple instances evenly. This architecture ensures that the NLP modules and wrappers can have multiple instances and thus can be scaled depending on demand.

The above architecture describes the BigData batch processing mode. The SUMMA Platform additionally supports a live stream processing mode for ASR, MT, and clustering NLP modules. Live stream processing is similar to batch processing, except that communication between the modules is handled over a WebSocket API rather than over a RESTful API to ensure ”always open” pipes between the modules. The pipes will be implemented by the integration platform acting as intermediary which establishes WebSocket API connections to the involved NLP modules; this arrangement allows the integration platform to monitor the date flowing through the pipes and feed this data to the UX via the central DB similar to the batch mode. For the stream processing mode all modules are expected to handle live streams in the real-time and therefore there is no need for specific scaling solution apart from piping the modules together via the WebSocket API. Timestamps and keepalive mechanism ensures that stream processing resumes automatically in case of the the NLP module crashes.

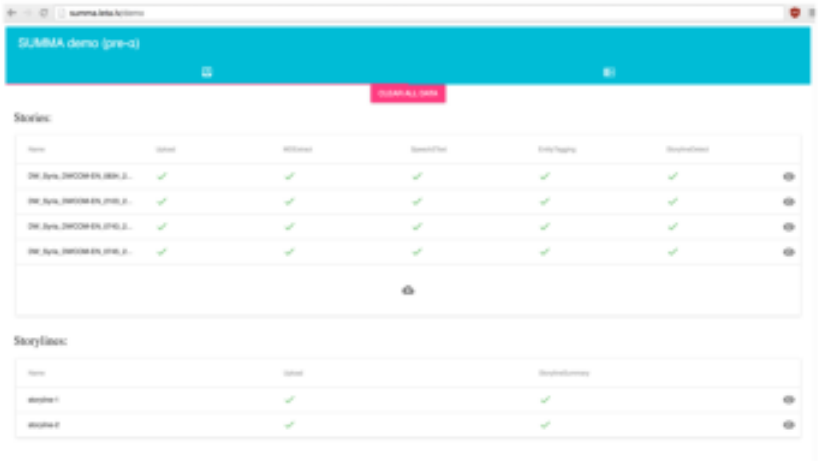


Figure 2: Content Ingestion Interface

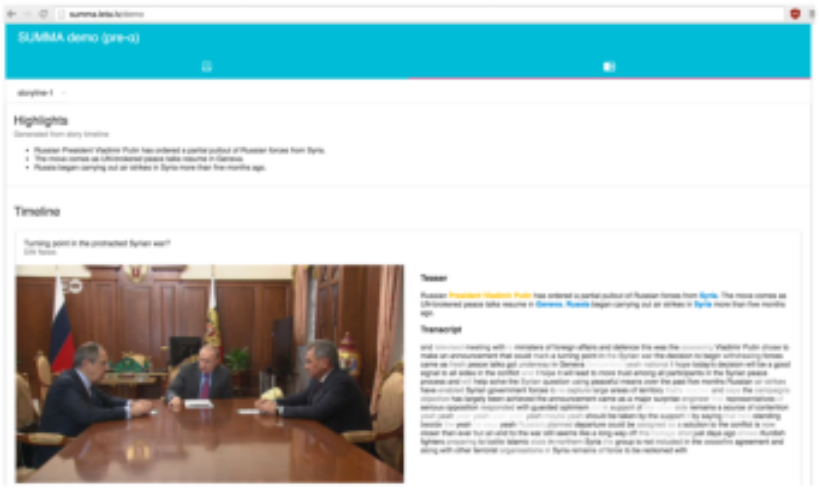


Figure 3: Monitoring Interface

4.2 User interface

The User Interface consists of two parts:

- the content ingestion control panel
- the monitoring interface

The baseline implementation of the Content Ingestion control panel is shown in Figure 2. It will evolve as the actual BBC and DW content provision endpoints (Section 2) become available in WP2.

The monitoring interface will be specified by WP1. A current baseline interface is shown in Figure 3.

5 BigData Scalability (Cloud)

The decision to build the infrastructure from a collection of small logical units (aka microservices) enables easy scaling as any of the logical units can be scaled separately – new instances can be created on demand. Independent news stream nature of the SUMMA Platform input reduces the scaling complexity, as the most demanding ASR and MT steps for each news stream can be processed independently.

A key part of this design is to provide a flexible logical unit discovery and communication: for efficient data flow distribution and processing. It has been decided to use message queues for data distribution and communication between data processing units.

The use of Docker containers as the engine behind each logical unit makes this above design possible and highly efficient, as the containers are self-contained, isolated from the host system at the user space level, but sharing the kernel with the host system. Thus containers use computation resources at the maximum efficiency.

Computing power for BigData scaling tests can be provided by the AWS[7] cloud services, which already provides some of the required services: the ability to run and manage containers, message queues, file storage, auto scaling (spinning up new instances) etc.

Alternatively, a Kubernetes (Vohra (2016)) setup on AWS EC2 instances or plain hardware of project partners will be used. Kubernetes is a complete solution for orchestrating Docker container systems on a variety of platforms.

[7] Amazon Web Services: https://aws.amazon.com/

6 Conclusions

The described SUMMA Platform architecture, its design principles, and the selected API modelling tools have been verified through the baseline proof-of-concept SUMMA Platform prototype. The prototype will be further evolved into the actual SUMMA Platform.

The SUMMA Platform will be based on Microservices with Docker and RESTful APIs as the core architecture. Swagger will be used for RESTful API modelling, representation and documentation. The WebSockets API will be used for stream processing. HLS and JSON formats will be used for transporting and storing AV and text news media.

References

Andy Giefer. *Access to the Deutsche Welle API*. "Internal DW documentation", 2015.

David Sheppey. *Draft Technical Reference Relating to SUMMA Work Package 2*. "Internal documentation from BBC Technology Development & Delivery (TeDD)", 2016.

Sam Newman. *Building Microservices*. " O'Reilly Media, Inc.", 2015.

Brian Mulloy. Web API Design - Crafting Interfaces that Developers Love. *Apigee*, 2012. URL https://pages.apigee.com/rs/apigee/images/api-design-ebook-2012-03.pdf.

Google. Google JSON Style Guide. *Google*, 2016. URL https://google.github.io/styleguide/jsoncstyleguide.xml.

Developer Cloud IBM Watson. Using the WebSocket interface. *IBM*, 2016. URL https://www.ibm.com/smarterplanet/us/en/ibmwatson/developercloud/doc/speech-to-text/websockets.shtml.

Deepak Vohra. *Kubernetes Microservices with Docker*. "Apress", 2016.

ENDPAGE

SUMMA

H2020-ICT-2015 688139

D6.1 Platform architecture and API Modelling tools selection