Co-funded by the Horizon 2020
programme of the European Union

h2020mirror.eu

MIRROR

Migration-Related Risks caused by
misconceptions of Opportunities and Requirements

Grant Agreement No. GA832921

Deliverable D7.3

Work-package	WP7: Architecture and Information Model
Deliverable	D7.3: MIRROR Framework - First Prototype Release
Deliverable Leader	EURIX
Quality Assessor	MoDAT
Dissemination level	Public
Delivery date in Annex I	M14, July 31, 2020
Actual delivery date	July 31, 2020
Revisions	1
Status	Final
Keywords	First Prototype, MIRROR Framework, Continuous Integration

Disclaimer

List of Authors

Partner Acronym	Authors
EURIX	Paolo Castelvetere, Francesco Gallo, Paolo Manca
LUH	Erick Elejalde, Claudia Niederee, Miroslav Shaltev
SAIL	Gerhard Backfried, Erinç Dikici, Miroslav Janosik
CERTH	Vasileios Mezaris, Alexandros Pournaras

Table of Contents

Executive summary

This deliverable describes the implementation of the first prototype of the MIRROR framework. According to the project plan, three major releases will be delivered. The second and third release are expected at M30 and M36, respectively. The main objective for the first release is to provide an early validation of the MIRROR approach, integrating the available components with their current status of development after the first year and also to demonstrate the capability of the system to perform and end-to-end media analysis, crawling data from different data sources, analysing such data and providing the results to the user.

The main objective for the first prototype was the integration of technologies into a coherent framework targeting a main scenario, namely the detection of migration-related (mis)information campaigns. The detection of such campaigns, which is one of the core objectives of the project, was identified as a first high-priority scenario to drive the integration of the components.

The background for the first prototype is provided by several results which have already been produced by the project, in particular all the requirements and scenarios identified by WP2, the conceptual information model and the architecture defined in WP7 and the analysis components developed in WP4, WP5, WP6.

The overall framework can be represented in a top-down approach as a number of layers, namely the client applications, the access layer, the data analysis layer and the data management layer. All such layers are connected by a cross-layer enabling the integration and communication. For each layer the components developed and integrated after the first year have been described.

In addition to the technical results, another WP7 objective for the first year was the creation of a continuous integration environment, fostering the collaboration among technical and non technical partners for collection of feedback from the users and the deployment of new functionalities.

During the first year a joint effort from all partners was performed to identify the main expectations from the users and the high priority functionalities to be provided by the system. Adopting an Agile approach, the collaboration included periodic calls beyond the plenary meetings and the adoption of tools for the management of the development process. Timeboxed iterations were supported by weekly calls where all interested partners could share progress, discuss about impediments and plan the next steps.

One of the core tools enabling the collection of feedback was a mock-up of the user interface, which was available to all the partners and was improved iteratively, collecting the news ideas and requirements along the first year of the project, providing a common vision of the target system and a common ground for the discussion.

Targeting the project pilots, the environment for continuous integration of the results has been prepared, adopting cutting edge technologies for the development and deployment of the prototype. The deployment mechanism leverages DevOps practices, with automation and configuration management tools.

1 Introduction

In this document we describe the first release of the MIRROR integration framework, corresponding to the first of three major releases which will be delivered during the project and will be described in deliverable D7.5 and D7.6, at M30 and M36 respectively.

Since the main objective for the first release of the framework is to provide an early validation of the MIRROR results, in terms of user requirements, scenarios, developed components and their continuous integration, in the following we focus mainly on providing an overview of the current status of the development and integration after the first year, rather than on the technical details of each components, deliberately leaving this information for the deliverable D7.5, where the overall system will be consolidated and in a far mature state. The focus of the first prototype is on data crawling from the different data sources, analysis of different media types and on the results provided by the user interface. Concerning the technical components, the current status has been described in the corresponding M12 deliverables, namely D4.1, D5.1 and D6.1 and the integration level reflects the status of the different components after the first year.

The main objective for the first prototype was the integration of technologies into a coherent framework targeting a main scenario, namely the detection of migration-related (mis)information campaigns. This detection of such campaigns, which is one of the core objectives of the project, was identified as a first high-priority scenario to drive the integration of the components.

The overall framework is presented in a bottom-up approach, starting from data layer up to the user interface. As described in the following, all framework layers are connected by a cross-layer enabling the integration and communication.

In addition to the technical development, WP7 is also in charge for the creation of a continuous integration environment, fostering the collaboration among all partners and the early deployment of a working system where new functionalities are continuously integrated in order to collect feedback from the users along the whole project duration and in particular during the project pilots. In the following we describe the collaborative environment, the tools and the development approach and the use of a system mock-up to stimulate discussion and gather feedback.

This deliverable provides input to WP10 for the preparation of the pilots, in particular for what concerns the technical requirements for the demonstrations hosted by different partners.

1.1 Background

The first MIRROR framework prototype leverages previous results achieved in WP7 and other WPs. In particular it is worth mentioning the requirements analysis and scenarios from WP2, described in deliverable D2.1 and the definition of the MIRROR architecture and integration approach discussed in deliverable D7.1. The information model in deliverable D7.2 defines the core entities and their relationships to be implemented in the system. The framework has been designed in order to integrate the technical components from WP4, WP5 and WP6, described in deliverables D4.1, D5.1 and D6.1, respectively. The input from WP3 has driven the security by design approach for the design of components: even if the actual implementation of the components devoted to data privacy and security is in progress and they will be part of the second prototype, the current architecture is already compliant to the requirements from WP3. Finally, targeting the second year pilots, the setup of the continuous integration environment and the analysis of the requirements for the pilots have been conducted in collaboration with WP10.

1.2 Target Audience

The results presented in this deliverable refer to the implementation of the first prototype and to the integration of the components. Therefore, the target audience of the deliverable are mainly software engineers and other technical roles in the IT department, although the emphasis is often on the integration of components as boxes with connectors, rather than on their internal functioning. From another perspective, this deliverable can also be seen as a proof of concept for validating the MIRROR approach into a coherent system for future adopters, and as such people with management roles evaluating the exploitation of the MIRROR framework could be interested in the results reported here, mainly for what concerns the impact on the IT infrastructure.

1.3 Structure of the Deliverable

The document is organized as in the following: in Section 2 we briefly summarize the relevant requirements and scenarios for the MIRROR project, focusing on the main identified scenario for the first prototype; in Section 8 we describe the development strategy and the continuous integration environment; in Section 4 we provide an overview of the overall architecture in terms of layers and components; the architectural layers are described in Section 6 (Data Management), Section 6 (Data Analysis), Section 7 (Integration and Access), and Section 8 (Client Applications). For each layer we discuss the role in the overall framework and the status of the components; in Section 4.6 we discuss security and privacy issues related to the prototype implementation.

2 Requirements and Scenarios

The analysis of requirements and the identified scenarios has been discussed in D7.2. The original scenarios included in the project proposal, namely *Scenario 1 - Detecting hybrid threats driven by perception manipulation* and *Scenario 2 - Counteracting threats created by misperceptions* have been further analyzed and discussed with the users, resulting in two additional scenarios: *Scenario 3 - Supporting border agents with targeted information* and *Scenario 4 - The cyber intelligence analyst – an ordinary day at work*. Different personas and user stories were also identified (see D2.1 Section 4).

Such scenarios and the associated user stories, combined with the analysis of the migration factors, were used to create a list of functionalities and identify the requirements for the overall MIRROR system. Those functionalities and requirements have been used as starting point for designing the system and have been further analyzed and discussed with the users.

The requirements and scenarios defined in D2.1 advanced the awareness about the project objectives and enabled the discussion among all partners about the high priority functionalities to be implemented by the system. The users shaped their vision of the MIRROR system and stressed the need for a user interface supporting the activities of border agents and analysts in monitoring specific situations, enabling advanced searches on all media sources, including specific topics, locations, languages or more general migration related concepts. The core functionalities of the user interface, discussed with the support of a mock-up (see Section 8, included not only the advanced search, but also a dashboard and an update page where the system should also provide automatic notifications about the observed situations. Moreover, the possibility to bookmark the searches and also to export the results for internal reports were also identified as relevant functionalities.

Based on the requirements and scenarios and on the feedback collected from the users during the first year, a first scenario was identified to drive the development of the prototype. The scenario can be summarized as **detection of migration-related (mis)information campaigns** and is briefly described in the following Section. The user involved in the scenario is a border agent.

2.1 First scenario for prototype implementation

Title Detection of migration-related (mis)information campaigns

Description As an officer involved in border control, the user has heard about new migration relate activities on a specific route. Using the MIRROR system, a Situation can be used to observe these activities. The user can create a new Situation or can switch to one of the situations that have already been defined (e.g. situation at Greek-Turkish boarder). The system provides the results associated to the Situation, showing media perception and activities possibly related to information campaigns. The system provides the user an overview of the Situation (with a summary) and can be used to continuous monitoring, notifying the user about what is new. The user can further refine the search using different types of filters, such as Language, Location, Topics, Migration-Related Semantic Concept (MRSC). Based on the results, the user can create new Situations or change existing ones in order to have customized results from the system. The Observed Situations are used as an alternative access mechanism to free search, for a more long-term perspective, to observe via Filters more than one Situation, e.g. Situation at Greek Turkish Border, Situation in Mediterranean sea, etc. The system prepares and updates a media overview and overview over Information Campaigns for the Situations. The system periodically search for new information and notifies the user about new results concerning the Observed Situations. The user can navigate the results which are also

provided separately for the different media types. The system provides the image, video and text analysis results, the automatic speech transcription and highlights the identified topics.

Goals Obtain media perception and activities possibly related to information campaigns concerning a Situation of interest.

Relevance The scenario demonstrates the use of technologies developed in the project for the detection of (mis)information campaigns, which is one the core concepts behind the MIRROR approach. The scenario involves the technical components developed in the first year, leverages their integration in the framework and the implementation of the workflows for data collection, analysis, storage and allows the assessment of the user interface. For what concerns the information model, the core concept of Situation has a central role in the scenario, since it defines the relevant information the user is looking for and also provides a validation of the model. The border agent is one of the key personas defined in the overall project scenarios. The scenario leverages the support for relevant languages (e.g. in the analysis of the Greek-Turkish situation the relevant languages could be Greek, Turkish, English plus others). The scenario also demonstrates the capability of the system to crawl information from many sources and to perform automated media analysis for different media types and sources and to filter the results.

3 Continuous Integration Environment

In the following Sections we describe the environment for the development of the MIRROR framework, which includes both practices and technologies. One of the objectives for WP7 during the first year was establishing such environment, which will support the development and deployment activities during the second and third year of the project.

3.1 Agile Approach

The adoption of an Agile methodology and the advantages for the MIRROR context have been already discussed in detail in Section 4 of deliverable D7.1, where it was foreseen that the best approach for MIRROR would be to define a custom Agile approach fostering *continuous feedback* and *timeboxed development*.

The continuous feedback was achieved through the involvement of all partners into the design of the system and in particular of the user interface, as already described in Section 2. The creation of a mock-up (see Section 8 was beneficial to stimulate feedback and agree on a common vision for the target system. The mock-up was iteratively improved and provided a common ground for the discussion.

Beyond plenary meetings, periodic WP7 calls were organized during the first year, some were organized as extended calls to discuss at a non technical level, while others were limited to technical partners involved in the software development. Typically the WP7 calls were biweekly, but in specific periods they have been organized weekly.

In order to manage the software development in a collaborative way, a Kanban[1] dashboard was used. A screenshot of the board is shown in Figure 1, where the card containing the tasks are managed in a very simple and intuitive manner, using cards which can be moved through different stages, from the backlog to the To Do list (active tasks), up to the completion (Done).

The MIRROR Kanban board is available on the demo server hosted by LUH, at the following URL (registration required):

https://d-mirror.l3s.uni-hannover.de/wekan

The cards can be assigned, revoked, edited and populated with comments and attachments. The cards have a priority field with different levels (Low, Medium, High, Critical): in this way each member of team who is assigned a number of cards, can identify the ones with higher priority which must be closed earlier. It is worth noticing that for the implementation of the Kanban board we used WeKan[2], an open source implementation of the popular Trello[3] board, which has the advantage that it can be installed locally (e.g. in a Docker container), while Trello is managed by the Atlassian company and data are stored on the cloud.

3.2 Development Environment

For the development of the framework and internal components we make use of GitLab[4], hosted at LUH premises at the following URL (registration required):

[1] https://en.wikipedia.org/wiki/Kanban_board, last retrieved 31 July 2020
[2] https://wekan.github.io/, last retrieved 31 July 2020
[3] https://trello.com/, last retrieved 31 July 2020
[4] https://gitlab.com/, last retrieved 31 July 2020

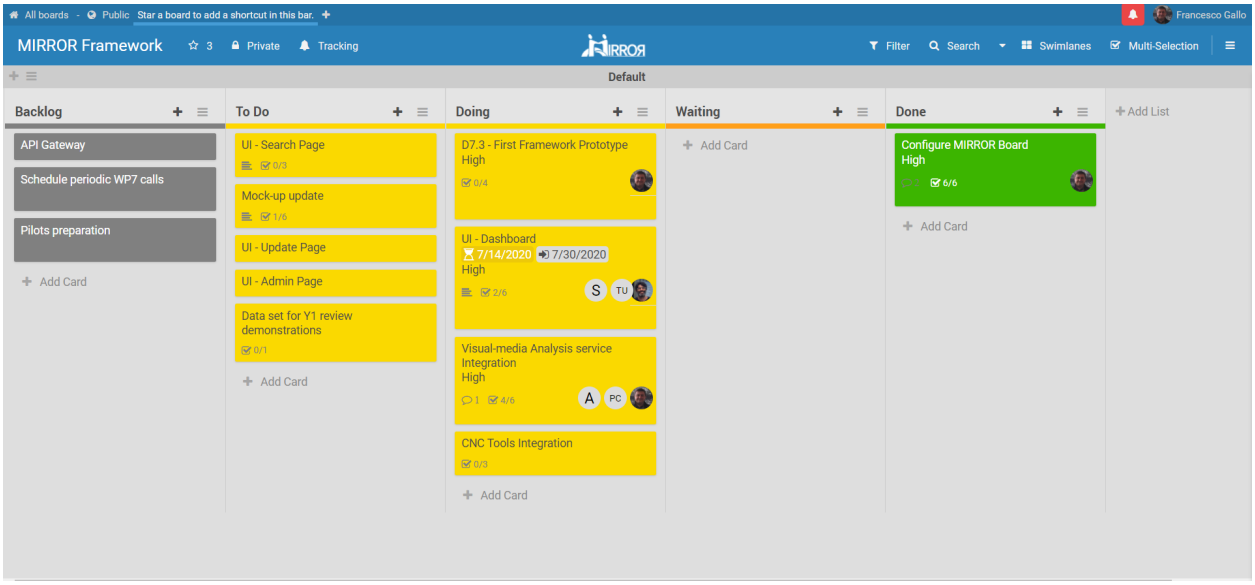

Figure 1: MIRROR Kanban board for framework development

https://git.l3s.uni-hannover.de/

GitLab is used as source code repository. The framework related development tasks are managed in a WP7 group, with several internal repositories for the fronted and backend components. The code is available to all partners and the codebase available on the project GitLab will be used for the deployment of the framework in the different environments, for the demo instance and for the pilots.

It is worth noticing that having adopted an automated approach based on DevOps [Kim et al., 2016] practices, not only the framework source code but also the infrastructure is managed using a repository (*infrastructure as code*).

For the development of the framework, different languages and technologies are used. The glue code used to integrate the components with the middleware is written using Python. The integrated components have their own language and internal technologies, this is enabled by the use of deployment within Docker containers and the communication through REST APIs, which makes the communication among components agnostic of internal implementation details. Concerning the different databases used by the framework, they include MongoDB and TinyDB for the storage part, while other components for indexing and messaging have their own databases. The front-end part is mainly developed using Javascript. The communication is based mainly on JSON and occasionally on XML.

3.3 DevOps practices

The relevance of DevOps [Kim et al., 2016] practices in the MIRROR context has been already discussed in Section 4 of deliverable D7.1. Continuous integration (CI), automated configuration management and continuous delivery (CD) [Humble and Farley, 2010] make use of specific tools to implement a toolchain

supporting the whole software lifecycle. Popular technologies such as Ansible[5], Jenkins[6], and Artifactory[7] have been used for CI/CD. Recently the use of CI/CD facilities provided by GitLab have been investigated and will be taken into account for the future development of the framework. The main advantage of using GitLab would be that the software development and CI/CD are managed using a single tool, rather than splitting the toolchain among different systems. Moreover, GitLab is gaining more and more momentum in the DevOps community, providing several best practice and support for many different technologies, APIs and protocols.

3.4 Deployment Environment

The current deployment environment is hosted at LUH premises and includes a Debian Server 10 64-bit (d-mirror.l3s.uni-hannover.de), which is accessible to all partners for the development. Access to the server is enabled via SSH and HTTP(S). Currently the resources assigned to the demo server include 16 GB RAM and 1 TB disk, which have been estimated as sufficient for the deployment of the prototype. The MIRROR system is currently made up of a number of components running locally on the same machine (inside containers) plus some remote services. The remote services are operated by partners at their premises or on dedicated infrastructure. The local components include the integration layer, the UI and some technical components.

The deployment environment is based on container and in particular on Docker[8]. The different components of the framework are deployed within Docker container. Nowadays, containers are emerging as a reference technology for the deployment of complex IT systems. In the MIRROR context, they provide several advantages, in particular they help reduce the complexity of the configuration of the overall system, simplify the update of each component independently, enabling the development teams to release new functionalities and new versions of their components without interrupting the operations of the overall system or requiring manual intervention for deployment and configuration.

Since the number of containers to be managed is increasing due to new components made available during the project, a management dashboard was configured to manage all container in the Docker instance on the demo server. A screenshot of the Docker management dashboard is shown in Figure 2:

The dashboard provides detailed information about the status of all containers, including access to the logs, as shown in Figure 3:

The dashboard is based on portainer.io, which is considered adequate for MIRROR scopes, since it provides enough administrative and monitoring functionalities without the complexity of other solutions such as Kubernetes.

[5] https://www.ansible.com/, last retrieved 31 July 2020
[6] https://www.jenkins.io/, last retrieved 31 July 2020
[7] https://www.jenkins.io/, last retrieved 31 July 2020
[8] https://www.docker.com/, last retrieved on 31 July 2020

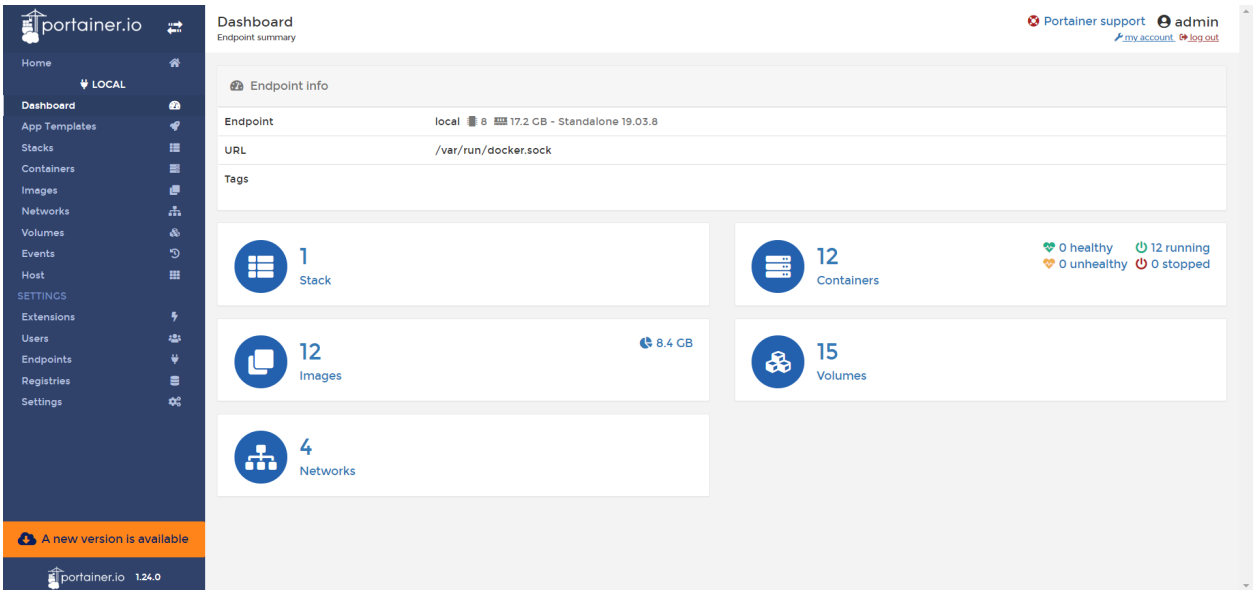

Figure 2: Docker management board for framework components

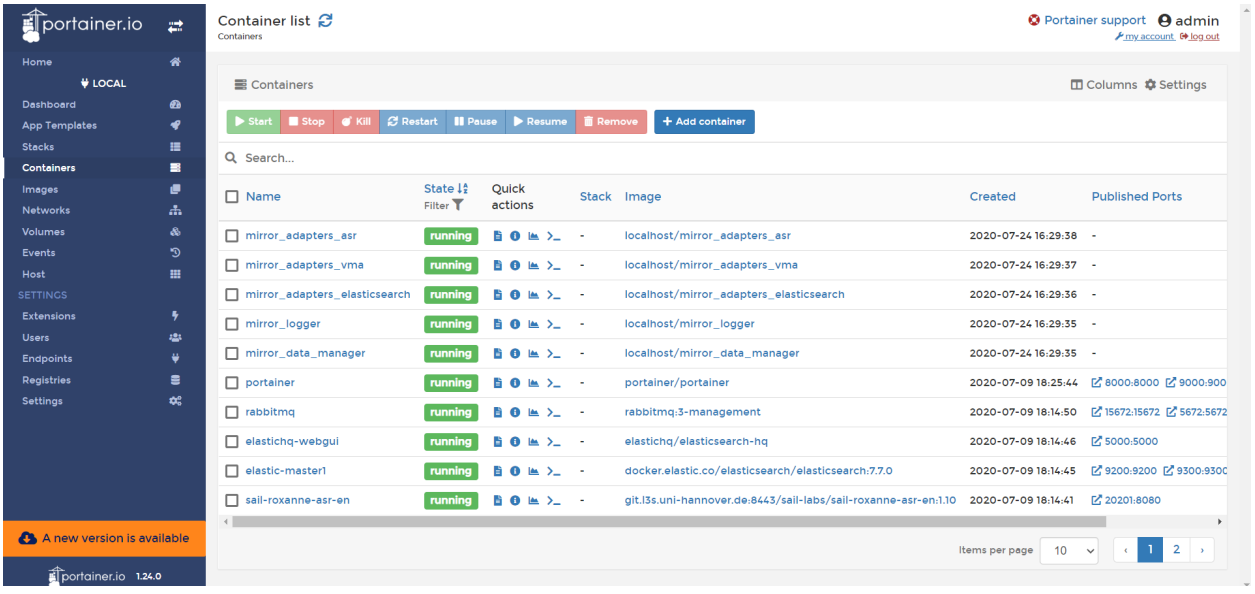

Figure 3: Docker containers monitoring

4 MIRROR Framework Architecture

An overview of the MIRROR architecture is depicted in Figure 4, based on the diagrams included in D7.1. The different layers of the architecture are represented.

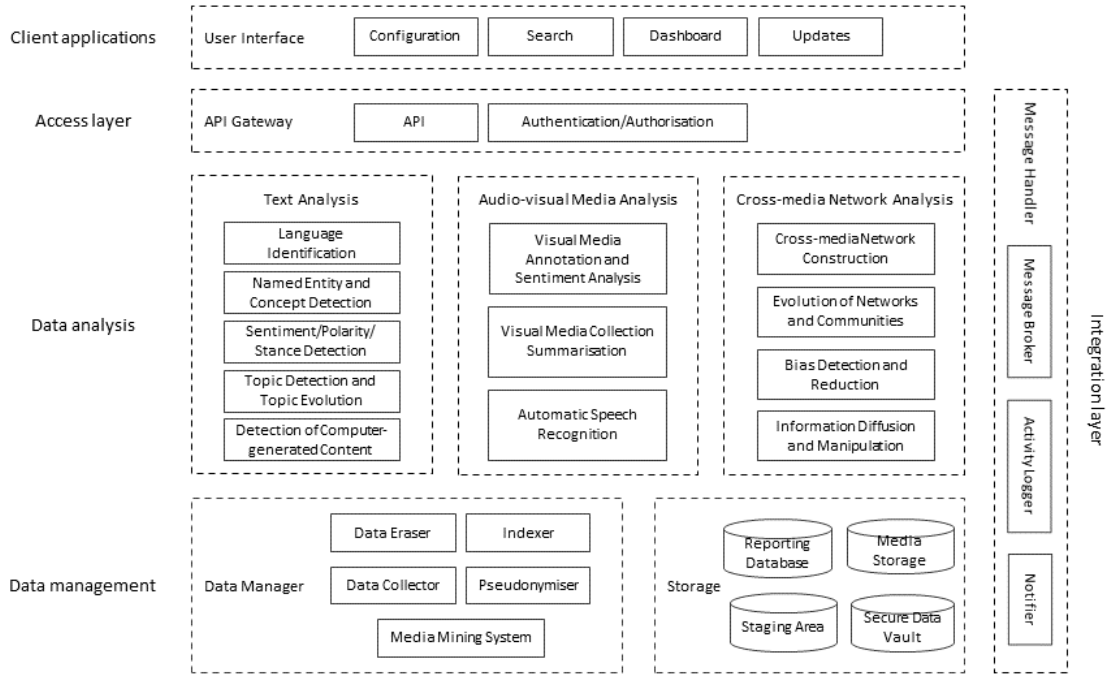

Figure 4: MIRROR architecture overview: the different layers and components are represented.

4.1 Data Management Layer

The Data Management layer includes both the components responsible for data collection and storage, but also those in charge for implementing privacy and security processes, such as pseudonymisation and retention policies. Such components are activated according to some of the processes defined in D7.2, in particular Data Collection, Pseudonymisation, Deletion and Verification. The collected data are represented in the MIRROR Conceptual Information Model (MIRROR-CIM) by the Media Element concept (see D7.2).

4.2 Data Analysis Layer

The Data Analysis layer includes the components responsible for the different types of data analysis, supporting the different source types. The associated processes are Media Analysis, Linking and Summarization. The results of the analysis are mapped to Annotations, Links, different types of Concepts and other related model elements in D7.2.

The Access and Integration layers enable the transition from the data level to the user level, the Client Applications layer.

4.3 Access Layer

The role of the Access Layer consists in translating the technical information available in the framework backend into data accessible and understandable by the user, through high level APIs. In addition to this, the Access layer takes into account the user role and profile and monitors the user activity: this information is useful to provide the user with the correct information and at the right level of detail, where privacy and security are guaranteed.

4.4 Integration Layer

The Integration layer is in charge for the communication among the layers and of orchestration of the processes, hence is supposed to be running behind the scenes during the execution of the processes in D7.2.

4.5 Client Applications Layer

The Client Applications layer, which communicates with the Data Analysis layer through the Access layer, is the user level: on top of the analysis results, adding a higher level of aggregation, the user can access the information provided by the framework. At this level the model elements which are relevant are the Findings, the Observed Contexts and Situations and all the other elements described in D7.2. Based on them, the analysis of the perception and misperception applies: this is represented in the model by Perception and Threats elements in D7.2.

4.6 Security and privacy by design

The architecture of the MIRROR system does not include a security layer, as an additional sub-system to be added later to the system to implement perimeter security to protect the inner unsecured system, but we rather tried to implement a security by design approach, were all issues related to security and privacy are taken into account since the beginning and have an impact on the design and implementation of the components.

The two layers which are most involved in security and privacy by design are the Data Management layer and the Access layer.

4.6.1 Data Management components for security and privacy

The Data Management layer includes components for implementing data retention policies (Data Eraser) and pseudonymization (Pseudonomyzer). Such components interact with a number of Storage components within the Data Management layer, namely the Reporting Database, the Staging Area, the Media Storage and the Secure Data Vault (see Figure 4 and Figure 4 in Section 5).

The Data Eraser and the Pseudonomyzer operate at data management level and affect the way the system stores information in order to prevent access to sensitive data and to guarantee that data is retained for the minimum amount of time necessary for system operations. Moreover, the system is designed to support different policies about data storage and exchange with external systems: one of the integration patterns

used by the system is the implementation of specific adapters for each component in order to exchange only information required for the specific analysis tasks (e.g. video or image URL) and the system itself could be configured to store or not the original raw data.

The Data Collector temporarily stores the retrieved raw data in a Staging Area, then it notifies via the Message Broker the presence of the new data to the Pseudonomyser. The pseudonymization entails the substitution of all occurrences of real names, surnames and nicknames found in the collected media with fictitious counterparts, to hinder the tracking of statements and media to their owner's identity, while preserving a one-to-one correspondence between real and fictitious names. Then the Pseudonymiser writes the pseudonymised data in the Media Storage, notifying their availability through the Message Broker, and stores in an encrypted way the sensitive mappings between real and fictitious names in a separate secure place called Secure Data Vault.

Sensitive content is hence sent by the Data Manager to the Secure Data Vault, a storage which is intended to provide higher security safeguards. The Secure Data Vault is also employed by the Message Handler to log all the exchanged messages for auditing purposes. A blockchain is currently being considered as a possible technology for implementing the Secure Data Vault, in order to prevent any tampering in the mappings. These sensitive data are retrieved and used only upon specific requests. This reverse pseudonymisation operation is again implemented by the Pseudonymiser, as part of a dedicated workflow, with the appropriate authorisation checks and safeguards in place. In order to further minimise the chance of exposure of sensitive information, the pseudonymised data stored in the Media Storage is continuously deleted by the Data Eraser, another component of the Data Manager, after the expiration of a configurable retention time. This progressively reduces the amount of data connected to the fictitious-to-real name mappings stored in the Secure Data Vault, ultimately stripping the oldest mappings of their sensitive nature.

4.6.2 Access components for security and privacy

The access layer is based on an API Gateway component which implements authorization and authentication, guarantees encryption of data exposed by the system and defines the APIs to interact in a secure and protected manner with the system, preventing access to internal data, systems and interfaces.

The user interface and other client applications interact with the system through the API Gateway only. The user interface will support the definition of user profiles, as an additional way to better control access to sensitive data, depending on specific authorizations.

Many analysis components run locally in the container infrastructure, hence data flow can be better monitored and controlled. It is worth mentioning that the integration layer and the access layer leverage technologies implemented according to best practices in enterprise application integration and enterprise security, such as loose coupling and identity and access management.

5 Data Management Layer

The data management layer includes several components supporting data crawling, storage, indexing and protection. The following components have been developed so far: Data Collector, Indexer and Media Mining System. The other components defined in the architecture (Data Eraser and Pseudonomyzer) are still under development and have not been integrated yet. In the following we provide a short description about the status of the components.

5.1 Data Collector

The Data Collector is in charge for data scraping from the different sources and for providing such data to the system. As depicted in Figure 5 taken from D7.1, the Data Collector communicates with other components in the data management layer and interacts with the Message Broker in the integration layer.

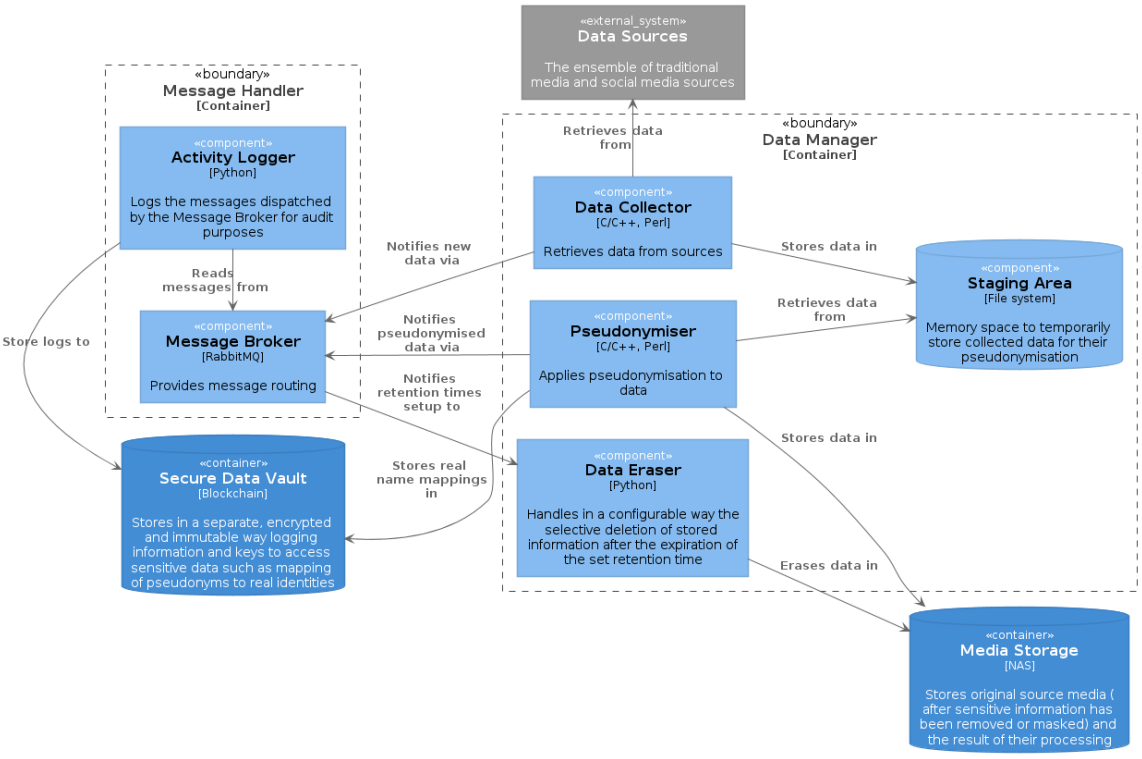

Figure 5: The Data Manager and Message Handler Containers described with a C4 model Component diagram

The communication with the integration layer is provided by an Adapter implemented in Python language, capable of interacting through asynchronous messages on specific queues.

The Data Collector is intended to communicate with external Data Sources supporting different mechanisms. Currently the main data source is provided by the Media Mining System, described in the following. The Data Collector communicates with the Media Mining System through REST APIs. The core module responsible for fetching data is implemented in Python and provides a specific configuration to support specific queries to the Media Mining System and a internal Scheduler which incrementally executes the appropriate queries to get data. The Scheduler also stores time information about the last successful query, in order to

catch up with missing data in case of temporary failures or inaccessibility of the Data Sources. An embedded DB based on TinyDB is used as a supporting DB during the operations.

5.2 Indexer

The Indexer component is responsible for retrieving information about data sources and the results of the analysis, in order to publish this information into a global index to support search and access.

The component currently includes an Adapter to interact with the Message Broker, a search service and a core module to fetch data from the Message Broker and store into the index.

The Adapter and the module to index the results are implemented using the Python language. ElaticSearch is used as search engine: it is based on the Lucene library which provides a distributed, multitenant-capable full-text search engine with a REST interface and schema-free JSON documents. Elasticsearch is developed in Java, clients are available for a huge variety of languages and technologies and currently is the most popular enterprise search engine. Elasticsearch can be used to search all kinds of documents. It provides scalable search, has near real-time search, and supports multitenancy. Indices can be divided into shards and each shard can have zero or more replicas. Each node hosts one or more shards, and acts as a coordinator to delegate operations to the correct shard. Rebalancing and routing are done automatically.

5.3 Media Mining System

The Media Mining System (MM-System) provides a framework for the collection, processing, storage and retrieval of a variety of data from traditional and social media sources. The overall architecture is depicted below in Figure 6.

The overall system consists of multiple types of components:

- *Media Mining Feeders: to automatically record and harvest multimedia, audio and textual data from a series of traditional and social media platforms*
- *Media Mining Indexer (MMI): to process and enrich data and produce annotated outputs*
- *Media Mining Server: to store indexed and multimedia content*
- *Media Mining Clients: a set of clients for querying and visualization of information*

All of the above are components. The MMI itself integrates a series of sub-components, such as audio-segmentation and -classification, Automatic Speech Recognition (ASR), speaker identification (SID), Named Entity Recognition (NER), Topic Detection (TD) and Sentiment Analysis (SA). Not all of these technologies may be available depending on the particular language of data being processed. The MMI – and hence all the individual technologies – can also be made available as dockerized images, thus allowing the include MMI functionality into a micro-service based architecture.

Once set up, the system automatically and continuously records and harvests data from multiple sources in different languages. The raw data then runs through a series of processing steps: the speech content is transcribed, indexed and enriched with information regarding language, speaker, named entities, topics, and sentiment. It is subsequently stored in the back-end server for later search and retrieval, and can also be archived. Visualisation and analysis of trends, relations, global hot spots, profiles, as well as ontologies

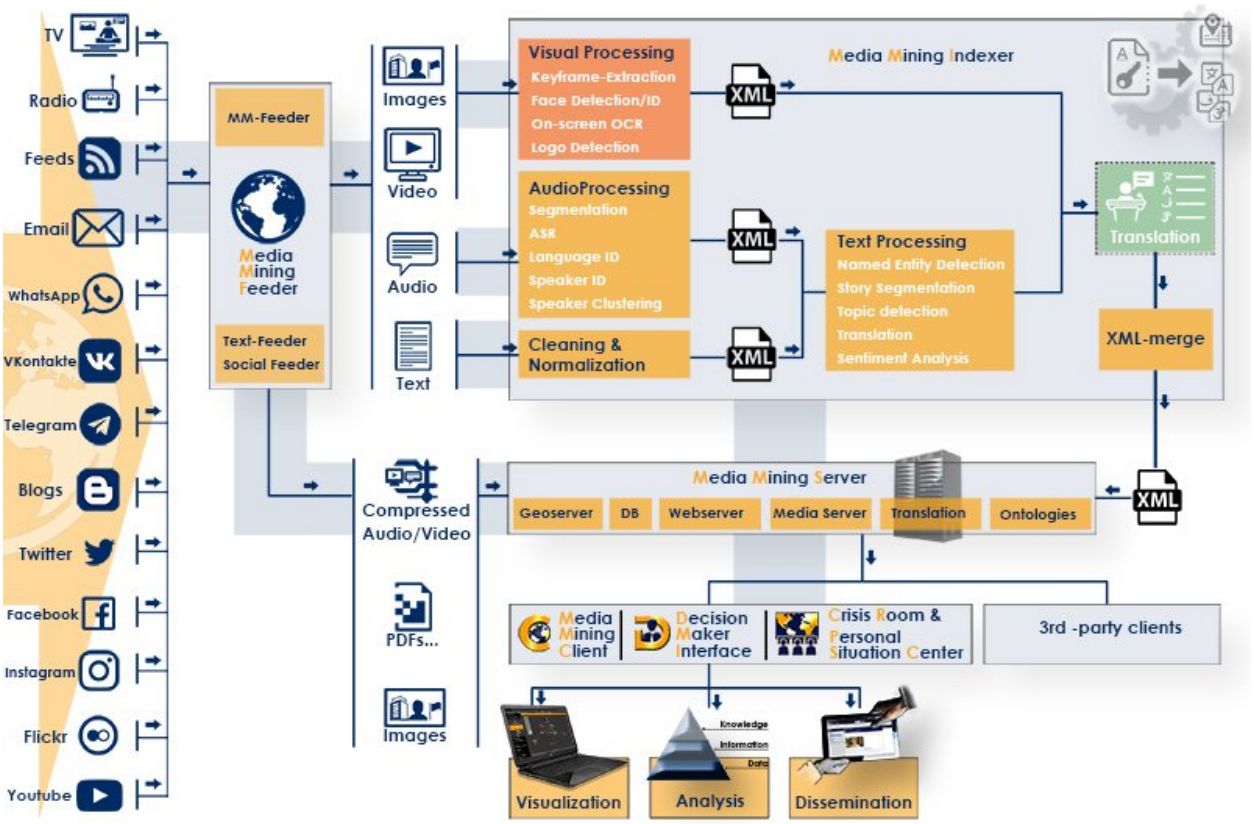

Figure 6: Media Mining System overview.

and Social Media Analytics pave the way for turning raw, unstructured data into actionable knowledge and intelligence.

The MM-System can also serve as a data-provider in a DaaS-manner, in which case all ingestion and processing is carried out as described above, but instead of accessing the enriched content and data via the clients, this access takes place via a REST-API. The same breadth and variety of data can be access this way and be funneled to further processing (outside the MM-System). Figure 7 below provides an overview of the MM-System for DaaS operation.

The overall MM-System is a modular system aiming to cover the complete OSINT workflow cycle from the requirements phase to the dissemination and feedback phases. It enables OSINT professionals to quickly extract meaningful analyses from unstructured data in a variety of formats across multiple languages and sources. Analysts are supported in their work by tools to visually explore and search large volumes of data according to their mission and fields of intelligence. The MM-System consists of a set of technologies packaged into components and models, combined into a single system for end-to-end deployment. A number of toolkits allow end-users to update, extend and refine models in order to respond flexibly to a highly dynamic environment. Not all components and technologies need to be present initially and can be added flexibly over time. Several Feeders, Indexers and Servers may be combined to form a complete system.

Within MIRROR, the MM-System will be used in the DaaS-manner described above, serving as a data-backend for the collection and enrichment of input. Enrichment capabilities will also become available

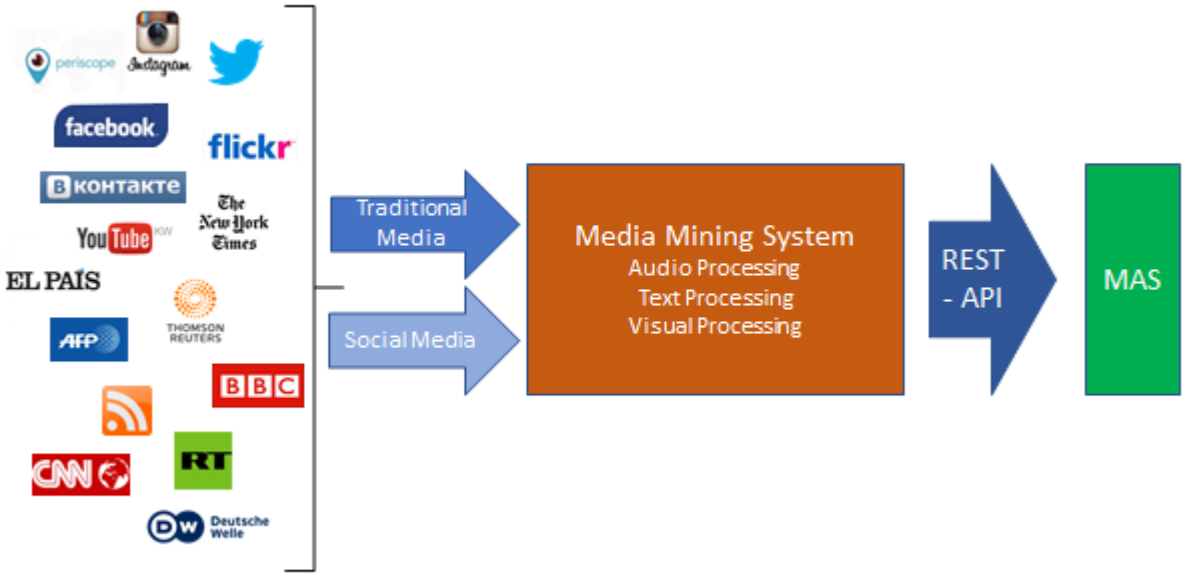

Figure 7: Media Mining System DaaS operation.

within the MIRROR platform during the project effectively reducing (and finally eliminating) the need for enrichment within the MM-System. Eventually, this will render the MM-System to be only a provider of raw-data.

5.4 Components for privacy and security

As already mentioned in Section 4.6, the data management layer includes two components related to privacy and security, namely the Pseudonomyzer and the Data Eraser. The Pseudonomyzer is intended to implement procedures to prevent access to sensitive data, part of its functionalities could be embedded in other components. The Pseudonomyzer is not included in the first prototype since it is still under development, but is planned for the integration in the second release. The Data Eraser is in charge for implementing data retention policies and mainly operates at data storage level. Also this component is still under development.

For what concerns the Storage components, the Staging Area and the Media Storage are currently based on a NFS mounted by the server hosting the framework instance and on TinyDB, an embedded DB, just for metadata related to the data scraping process, used for further processing. The Secure Data Vault is still under development, along with the Pseudonomyzer, in collaboration among different partners, evaluating the use of the blockchain for the implementation of the key mappings generated during the pseudonymization process and the reverse one. The Reporting Database is provided by the implementation of the Cross-media Analysis components, described in Section 6

6 Data Analysis Layer

In this Section we describe the Data Analysis layer, which includes the components developed within technical Work Packages (WP4, WP5, WP6), covering the different aspects of media analysis implemented in the project. According to the architecture diagram in Figure 4, the components are associated to the different types of analysis for Text, Audio-visual Media and Cross-media Network.

We focus here on the list of components which are actively involved in the main scenario and that (are ready for testing?) at the current stage of the project. For the others, we briefly summarize the status and the future plan targeting the second prototype implementation which will be described in deliverable D7.5.

For each component we briefly describe the role in the overall framework and in the implementation of the main scenario for the first prototype, described in Section 2, the development status with available functionalities and the deployment and integration mechanism. For technical details about the components please refer to the corresponding WP technical deliverables.

6.1 Text Analysis

6.1.1 Named Entity and Concept Recognition

Role of the component The Named Entity Recognition (NER) component takes as its input a sequence of words and annotates certain classes of named-entities and concepts in the input text. This input text may come from textual sources or be the output of a prior processing-step such as Automatic Speech Recognion (ASR). The component supports different models for tagging: pattern-based matching or statistical matching.

Development status As of the submission of this deliverable, the NER component is available for the English language. Further languages which are relevant to MIRROR are currently under development. Several (pattern-based) models for the languages most relevant to MIRROR have been developed and will be packaged within the upcoming months. In addition, a statistics-based model is being developed and currently in beta-testing for English.

Deployment and integration The NER component has been packaged into a docker-component, exposing a REST-API. Currently, the NER models are packaged together with the NER-engine, resulting in one docker-component per language. The component receives a text-document as input and returns an enriched document as JSON upon successful completion of the annotation process.

6.1.2 Sentiment Analysis

Role of the component The Sentiment Analysis (SA) component attempts detect positive and negative polarity as well as the intensity of these aspects uttered in a text. Different levels of analysis, from phrases to complete documents as well as the possibility to combine positive and negative aspects into a mixed class (with a 4th class, neutral in case neither of these are detected) characterize the Sentiment Analysis component. The component is based on different lexica in combination with processing mechanisms for phenomena such as negation, boosting or the use of idioms and slang. Furthermore, the component allows for adaptation to particular domains as well as for content from social media.

Development status As of the submission of this deliverable, the SA component is available for the English language. Further languages which are relevant to MIRROR are currently under development. As the component allows for domain-dependent extension, migration-related terminology will be accomodated within the models.

Deployment and integration The SA component has been packaged into a docker-component, exposing a REST-API. Currently, the SA models are packaged together with the SA-engine, resulting in one docker-component per language. The component receives a text-document as input and returns an enriched document as JSON upon successful completion of the annotation process. The annotated sentiment is available on different levels of granularity from the sentence level to the document level.

6.2 Audio-visual Media Analysis

6.2.1 Visual Media Annotation and Sentiment Analysis

Role of the component This component performs visual analysis on images and videos and annotates them with MRSCs, other generic visual concepts, captions and sentiment value. The annotations are used for the media summarization and to provide a deeper understanding of the media content, for search and retrieval.

Development status For all visual annotation approaches and sentiment analysis, initial methods have been developed. For MRSC detection, driven by the lack of ground-truth MRSC-annotated image/video corpora, we formulated the problem as a zero-shot learning / Ad-hoc Video Search (AVS) problem, and adapted to this end a SoA method that we previously developed for AVS. We tested this method in MIRROR on a couple of generic-concept benchmark datasets so as to quantitatively assess and document the sensibility of re-purposing an AVS method for concept detection, and experimented with retrieving videos that are associated with specified MRSCs that were defined in MIRROR. For generic visual concept detection in video, we started from a SoA deep-learning method (published in early 2020) that we had previously developed, and we extended it by introducing a NetVlad layer in the input of the network (so that multiple video frames, instead of one frame at a time, can be effectively represented), and a Mixture of Experts layer in the output of the network. Experiments in the large-scale YT8M video dataset showed that these extensions increased the performance. For video captioning, we experimented with the HACA model, published in 2018, and following several experiments we optimized the initial implementation that was eventually adopted in MIRROR. For sentiment analysis, we started with a literature review, and by adopting, for training our algorithms and for experimentation, the large scale SentiBank Flickr dataset. For representing the image/video content, we employed the “inception” neural network architecture, pretrained on the YT8M video dataset. On top of these representations, we trained two networks to perform classification to polarity (a one-dimensional sentiment indicator ranging from positive to negative), and adjective noun pairs (ANPs). The initial results were promising.

Deployment and Integration A REST service has been implemented to serve as the API of the Visual Media Annotation and Sentiment Analysis component. All the visual analysis methods, with the exception of sentiment analysis, have been integrated in the service. The service is running remotely in CERTH in an adequately equipped Linux server with GPU support. The programming language used for the REST service and the visual analysis methods is Python. Full documentation for the use of the API has been provided.

6.2.2 Visual Media Collection Summarization

Role of the component This component will create a concise, yet descriptive, summary of media items retrieved by the media mining system of the MIRROR framework. This condensed representation will facilitate search and navigation in large media collections.

Development status We reviewed the latest literature in image/video collection summarization. We established a baseline approach that uses the YT8M extractor to represent the images in a vector space; this is a SoA representation that is also used by the Visual Media Annotation and Sentiment Analysis component, for generating the generic visual concept annotations and the sentiment analysis results, respectively, thus minimizing the computational overhead of summarizing collections of media items that have gone through these earlier analysis steps. The vector representations within a media collection are clustered using a k-means algorithm, which had been successfully used in the past in combination with older image/video representations and was shown to have advantages over other clustering algorithms such as DBSCAN / HDBSCAN. Representative images/videos are selected from each cluster, based on minimal distance from the cluster's center. Early qualitative results of this baseline look promising; the more formal assessment of them is work in progress.

Deployment and Integration This component has not yet been integrated in the MIRROR framework. It is planned to perform the summarization locally in the MIRROR framework by exploiting the visual representations extracted by the Visual Media Annotation and Sentiment Analysis component.

6.2.3 Automatic Speech Recognition

Role of the component The Automatic Speech Recognition (ASR) component aims to provide the user with a written transcription of the spoken content in audio- or multimedia files. It encompasses several stages of processing, such as signal processing, segmentation into speech and non-speech sections of the input and the actual recognition and transcription process. The output of this component is a JSON structure which contains an automatic segmentation of the audio, a list of words in each segment together with start-end timestamps and confidence scores. In addition it is possible to output alternative transcripts in an *nbest-manner*. The ASR component thus aims at making audio and multimedia contents available for further enrichment and retrieval, allowing to detect relevant content also in these kinds of input.

Development status As of the submission of this deliverable, the ASR component is available for the English language. Further languages which are relevant to MIRROR are currently under development. Several models for ASR for the languages most relevant to MIRROR have been developed and will be packaged within the upcoming months.

Deployment and integration The ASR component itself provides a C/C++/C# API and has been packaged into a docker-component, exposing a REST-API. Currently, the ASR models are packaged together with the ASR-engine, resulting in one docker-component per language. The component receives an audio-file (WAV or MP3) as input and returns the transcript as JSON upon successful completion of the transcription process.

6.3 Cross-media Network Analysis

These components aim at improving the framework's capability to explore and examine migration-related social networks through analytical methods. The information for these networks is collected from available

sources and enriched with the output of the components in the Text and Audio-visual Media Analysis (see sections 6.1 and 6.2). In turn, the output of the components in the Cross-media Network Analysis will provide to the users (such as border control agencies) the tools for exploring and understanding the possible sources of misperceptions that can result in threats. Future components will also provide additional means for tracking and explaining misperception evolution across the geographical areas of interest as well as along the migration paths. A specific focus for this first release has been in the Cross-media Network design and the identification of information bias.

6.3.1 Cross-media Network Construction

The objective of these components is the construction of networks in accordance with the use cases identified in WP2 during the end-users' requirements analysis. The cross-media network construction component is implemented as a web service with four main subsystems, namely CNC-parser, CNC-gate, CNC-analyser and CNC-gui. The CNC-parser and the CNC-gate components are written around Janusgraph[9] , an open source distributed graph database using the Gremlin[10] graph traversal language. The temporal evolution of the data is tracked through the datetime property of the edges, which allow us to compute frames of the graph. The CNC-analyser makes use of the NetworkX[11] library.

Role of the component Through the CNC-parser the component translates the input coming from the Multimedia System (MMS), as well as Text Analysis (see Section 6.1) and Audio-visual media Analysis components, into data structures suitable to be written into the unified graph model. The CNC-gate is responsible for serving queries to the CNC database. It has two basic modes of operation, an exploratory free search and an advanced search based on filters. Finally, the CNC-analyser performs aggregation on the subgraph returned by the CNC-gate and provides analysis results from the subgraph input.

Development status Initial versions of all CNC components have been created as described in Deliverable 6.1. The implemented basic functionality is sufficient to allow integration into the MIRROR system and further development and addition of features as the project advances. In particular the CNC-parser is able to process input from the multimedia and text analysis systems and create records in the graph database. The CNC-gate component is able to select subgraphs based on user filters, including temporal information. The output of the CNC-gate can be passed to the CNC-analyser for aggregation and basic analyses. This is certainly the component which will be most often subject of further development.

Deployment and integration Following the microservice architecture of the MIRROR system, the CNC-parser, CNC-gate and CNC-analyser components are isolated and attached to the whole system via Docker technology. The communication within the system is done via the REST API. For a more detailed description of the API and the component please refer to deliverable D6.1 First Release of Cross-media SN Analysis Technologies.

6.3.2 Bias Detection and Reduction

Our perception of the situation in a country or a region is strongly influenced by the reflection of this situation in mass and social media channels. To avoid information overload, news outlets typically filter the available news from foreign countries based on the expected interest of the target audiences. The objective

[9] https://janusgraph.org/
[10] https://tinkerpop.apache.org/gremlin.html
[11] https://networkx.github.io/

of this component is to identify some possible kinds of biases. In the first version of the component, we help identify the bias created in EU-related news reports due to editorial policies.

Role of the component The input for the component will be collected from the CNC-gate and CNC-analyser. Given that the CNC-gate generate a subgraph based on the specified filters, the result from this component will be also based on the same set of filters. The analysis produced will feed the dashboard in the user interface. The results can be represented in multiple ways. For example, it can show difference in coverage by topic for different regions. A disproportionate attention to a subject or an unusual polarization of the reporting may indicate an information campaign in support of a specific agenda.

Development status The initial version of the component allows integration into the MIRROR system and facilitates further development of additional analysis methods as the project progress. An initial functionality provide a filter-aware comparison of counties of origin against countries of destination with respect to mass media coverage. Different method will be provided through specific paths (e.g., $/\mathrm{process}/\mathrm{mbd}$).

Deployment and integration The component has been integrated via Docker technology and accessed via REST API. For practical purposes, the initial version shares a container with the CNC-analyser.

7 Integration and Access Layers

In the following we briefly describe the Integration layer and the Access layer, which are responsible for internal and external communication respectively. The Integration components enable the communication among the Data Management and Data Analysis components and provides information to the Access layer. The Access layer is responsible for the communication and interaction from external clients (including the User Interface). Both layers are described in more detail in Section 4

7.1 Message Broker

The Message Broker component (see also D7.1) implements a Message Oriented Middleware, enabling loose coupling among components and message-based asynchronous communication. The advantages of such messaging systems have been successfully proved in the past decades in scenarios involving small to big size enterprise applications. The message-based middleware solutions enable the implementation of several Enterprise Application Patterns in order to achieve better flexibility, scalability and security. For a definition of such patterns see [Hohpe and Woolf, 2003].

Since many open source enterprise grade solutions are available, we made a quick analyses of the most important solutions based on the requirements of the MIRROR system. In particular, the search was focused on established and mature open source solutions maintained by an active community and on possibly lightweight and flexible systems, to avoid complications due to heavy complex middleware solutions.

Two candidate solutions were initially selected: Apache Kafka[12] and RabbitMQ[13]. In the end, RabbitMQ was chosen due to its flexibility, simplicity in terms of configuration and also to the simple deployment mechanism. Moreover, RabbitMQ has been considered a better solution in the context of MIRROR framework since the data streams are actually managed by other components (Media Mining System and Data Collector), so part of the core functionalities provided by Apache Kafka would not be used.

In terms of data retention policies, in the context of privacy by design, RabbitMQ has been considered easier to manage: unlike most messaging systems, the message queue in Kafka is persistent, the message stays in the queue until a given retention period/size limit is exceeded, meaning the message is not removed once it's consumed, but in principle data could be kept forever, it can be replayed or consumed multiple times; in RabbitMQ, messages are stored until a receiving application connects and receives a message off the queue, once the message is consumed, it's removed from the queue.

RabbitMQ has been integrated in the MIRROR framework using a Docker container. It also offers a simple yet powerful administrative interface to manage all queues.

AMPQ, the message protocol implemented by RabbitMQ, is flexible and supported by a variety of client applications written in several languages. The integration with the queues has been implemented in Python.

An example of RabbitMQ administrative interface is shown in Figure 8, taken from the RabbitMQ instance deployed on the MIRROR demo server. Using such interface, it is possible to manage users, queues, resources and to monitor the flow of messages exchanged on the message broker.

[12] https://kafka.apache.org/, last retrieved 31 July 2020
[13] https://www.rabbitmq.com/, last retrieved 31 July 2020

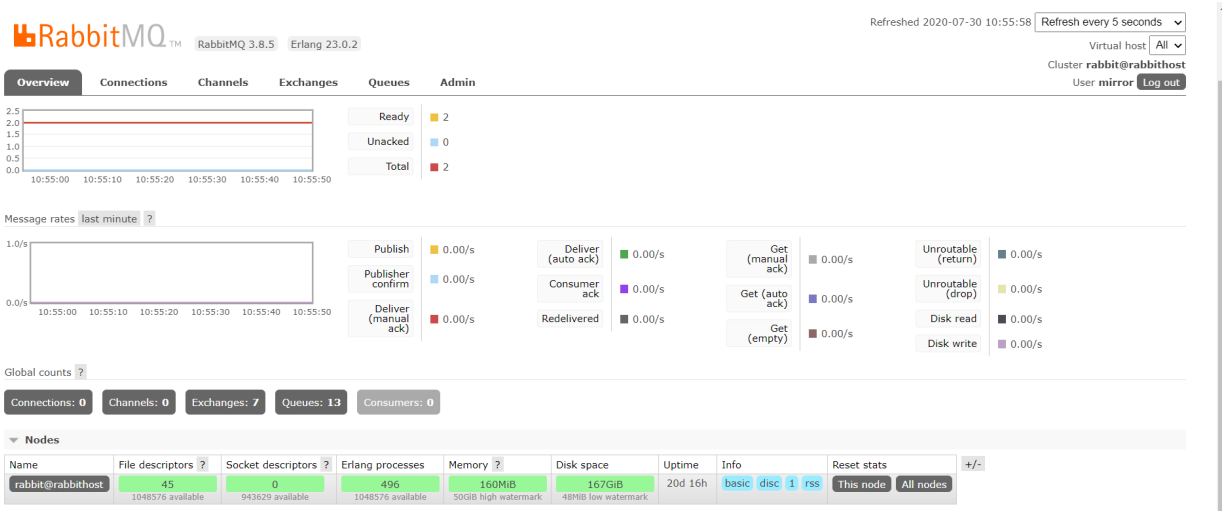

Figure 8: Admin interface for RabbitMQ

7.2 Activity Logger

The Activity Logger component is responsible for logging all relevant activities in the system, focusing on technical internal information (access to the system, processes, use of resources, etc). The component is implemented in Python and communicates with a dedicate queue on the message broker. The deployment mechanism is through a Docker container, as part of the core components of the system.

7.3 Notifier

The Notifier component is also responsible for collecting events within the system, but at a business level, such as the availability of new results related to the observed situations or any other notification which should be provided on the User Interface in the Update page. This component is also implemented in Python, deployed in a Docker container and is tied to a dedicated queue on the broker.

7.4 API Gateway

The API Gateway component provides the single point of access to the MIRROR framework by external systems and client application through the APIs. The REST style is used for the implementation of the framework APIs.Using a gateway to manage and monitor the interaction with the back-end components is a well known pattern, which provides several advantages, as reported in the following.

The access to the system is controlled and managed, the gateway should manage both authentication and authorization, possibly through standard services. If the APIs exposed by the system are well defined, they do not change over time (or do not change very frequently), enabling the parallel and independent development of the back-end and front-end component without breaking their interaction: in particular the back-end system can evolve and change over time, but the client applications do not have to change accordingly, provided the APIs and the exchange protocol and data are the same.

Moreover, the gateway hides the technical APIs exposed by the back-end services, preventing operations which could break or corrupt the system; the APIs also provide an high level description of the system and can be shared among all teams developing the different components, since they represent, in some way, the functionalities provided by the system. The APIs are exposed by the gateway using a single or a limited number of endpoints, even if behind the scenes each API could be related to a specific back-end component (e.g. the Indexer or the Notifier).

As done for the Message Broker, also for the API Gateway we made a quick market analysis in order to identity a solution which was suitable for the MIRROR framework requirements. In particular, we focused on established open source solutions with relevant applications, maintained by an active community.

The technical requirements included the capability to easily design, implement and maintain the APIs, the support for current standards (e.g. Swagger[14]) and the possibility to deploy the solution in a Docker container.

We quickly evaluated three candidate solutions: WS02, StrongLoop, KongHQ.

KongHQ[15] is a leading solution used for a variety of commercial solutions, with good support and documentation, natively deployed using Docker, with an extensive ecosystem and supports the OpenAPI[16] standard. The main drawback with this solution, from a MIRROR point of view, was the lack of some useful plugins and tools in the open source version. Although not a relevant limitation, it also currently supports only PostgreSQL and Cassandra DBs.

WS02[17] is one of the most popular API management solutions, with a complete set of plugins and tools for the API development, including graphical tools for API design and management, with an extensive documentation and support for OpenAPI. The main drawbacks from a MIRROR perspective are that WS02 is resource angry in terms of CPU and RAM, compared to other solutions and that the Docker installation requires registration to the WS02 repository; moreover, the installation is not straightforward, but due to the complexity of the system in terms of features (which makes it one of the market leaders), it requires additional effort.

StrongLoop[18] is part of the Loopback solution provided by IBM, is based on NodeJS, supports OpenAPI and provides a flexible solution for the API development. Compared to other solutions, the administrative interface is limited but is pretty stable and easy to use. One of the drawbacks of the solution is that compared to other solutions it requires some manual steps for the configuration, but on the other hand this gives also flexibility. The graphical tools for the API design are quite essential but are stable and complete. Moreover, the installation is pretty straightforward since it is based on the popular npm tool from NodeJS ecosystem and the deployment in a Docker container, although not native, is easy to manage.

Taking into account the provided features for API management, the documentation and support and also the previous expertise available in the consortium, we have chosen **StrongLoop**. Moreover, since NodeJS is one of the backbone technologies used by the MIRROR framework, with an extensive ecosystem of free plugins and a large community, the maintenance of the solution is expected to be easier.

Compared to other candidate solutions, StrongLoop is quite lightweight and provides just the required components for the implementation of the API Gateway, without additional unnecessary components which

[14] https://swagger.io/, last retrieved 31 July 2020
[15] https://konghq.com/, last retrieved 31 July 2020
[16] https://www.openapis.org/, last retrieved 31 July 2020
[17] https://wso2.com/, last retrieved 31 July 2020
[18] https://strongloop.com/, last retrieved 31 July 2020

fall outside the scope of the MIRROR framework. Moreover, from a security by design perspective, the NodeJS ecosystem and community provide tools, practices and support for the development of the API in a secure way.

The API Gateway is deployed in a Docker container on the MIRROR demo server. The main endpoint of the system is available at the following URL:

https://d-mirror.l3s.uni-hannover.de/api

During the second year of the project, the complete set of REST APIs exposed by the system will be developed and tested. The MIRROR framework APIs will be documented in deliverable D7.4.

8 Client Applications

In this Section we describe the Client Applications layer, which includes the User Interface and any client applications which could integrate with the MIRROR system through the APIs exposed by the API Gateway.

The architecture of the MIRROR system is based on a clear separation between front-end and back-end applications. The front-end application (the client application) communicates with the REST APIs exposed by a dedicated back-end. This approach is based on the so called *back-end for front-end* pattern[19].

As a consequence, the front-end application lays completely within the client (the browser), according to the Single Page Application (SPA) model. Although this model produces a shift of the business logic towards the client, in our implementation we will move as much of the business logic as possible towards the APIs in the back-end. In this case, the back-end will specialize the APIs provided by internal services. Another pattern which will be used is the *Façade*: the back-end APIs will hide the complexity of the APIs provided by other services (see Section 7).

In the following we describe the mock-up of the User Interface that was designed iteratively based on feedback collected from users, and then we provide information the technologies and the approach used for the implementation of the actual User Interface.

8.1 User Interface Design: Mock-Up

In the following we briefly describe the design of the User Interface making use of a mock-up. The iterative process and the collection of feedback from users has been described in Section 3. The User Interface mock-up is available at the following URL.

https://d-mirror.l3s.uni-hannover.de/mockup

It is worth noticing that having adopted an Agile iterative approach, also the mock-up will continue evolving during the project, so at the moment of reading the mock-up could appear slightly different with respect to the screenshots provided in the following, due to improvements and evolutions.

The User Interface has been designed with four main sections: Search, Dashboard, Update and Admin. The user can access each part using the top menu, although links among the sections are expected when navigating the results. The implementation approach is described in the next Section, in particular we will adopt a Single Page Application (SPA) paradigm.

The Search section is intended for advanced search using Situations and a number of filters collected from user feedback, as shown in Figure 9. The concept of Situations has been defined in the MIRROR-CIM in D7.2.

The user can perform the search using previously defined Situations (the Observed Situations), as shown in Figure 10, in order to have direct access to previous search results, which can be bookmarked and shared with others. The user could also be provided with a summary of the results from previous pages, the actual implementation of this functionality will be checked with the users.

The user can then perform the search using one of the Observed Situations or can change them or can create a new set of filters (which can in turn be saved as a new Situation or not). The filters include Time, Country

[19] https://samnewman.io/patterns/architectural/bff/, last retrieved 31 July 2020

Figure 9: User Interface mock-up: landing page

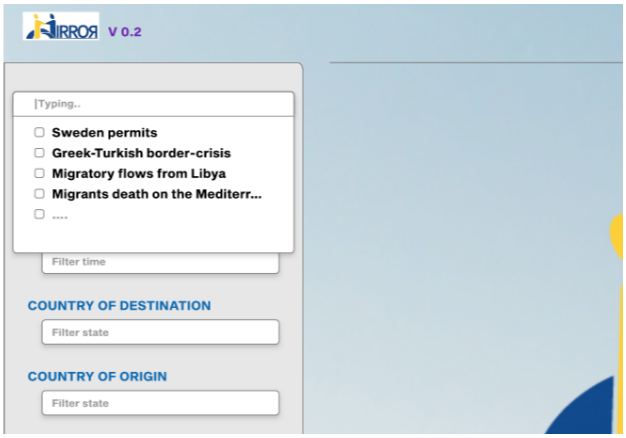

Figure 10: User Interface mock-up: observed situations

of Destination (COD), Country Of Origin (COO), Source Type, Language, MRSC, Named Entities, Topics and Sentiment. The information required by such filters is provided by the analysis tools running in the Data Analysis layer and natively from the Media Mining System.

The representation of the search results is shown in Figure 11. For each result a preview is provided, with additional information. The key words used in the search for the different filters are highlighted.

The user could be more interested in the sources rather than in the visualization of the results, so it is possible to change the visualization of the search results to show only the original sources, as shown in Figure 12.

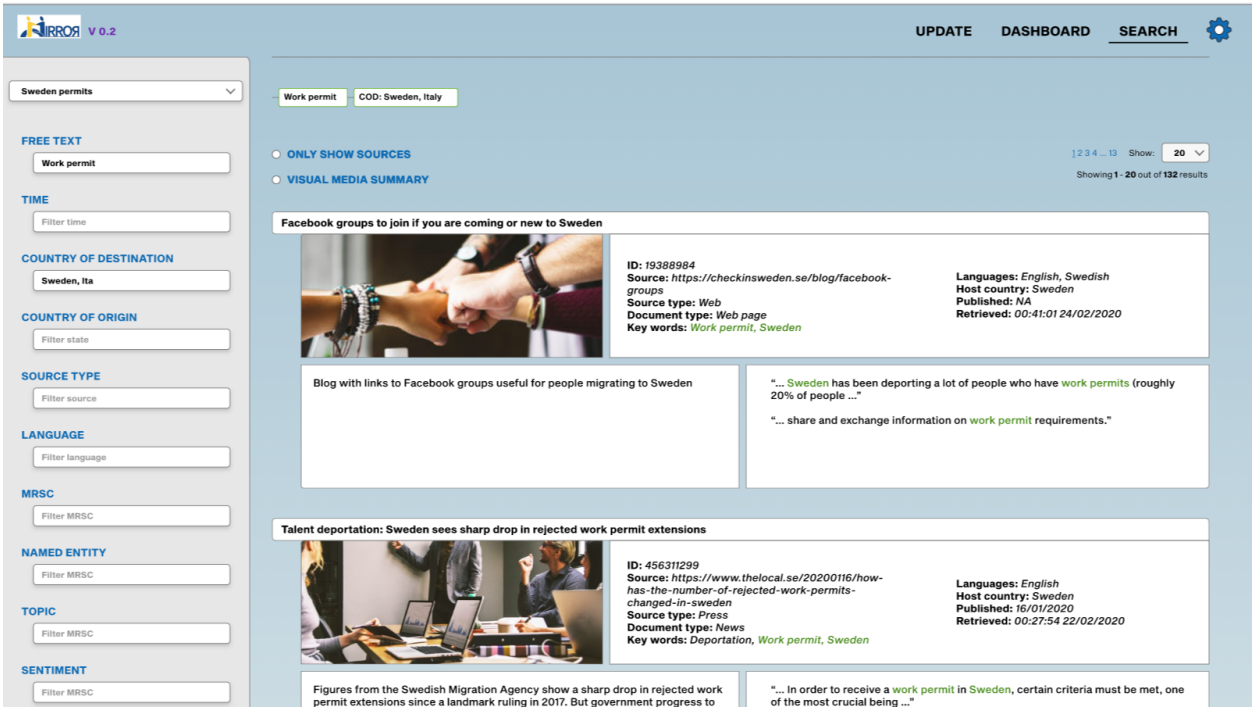

Figure 11: User Interface mock-up: Search page

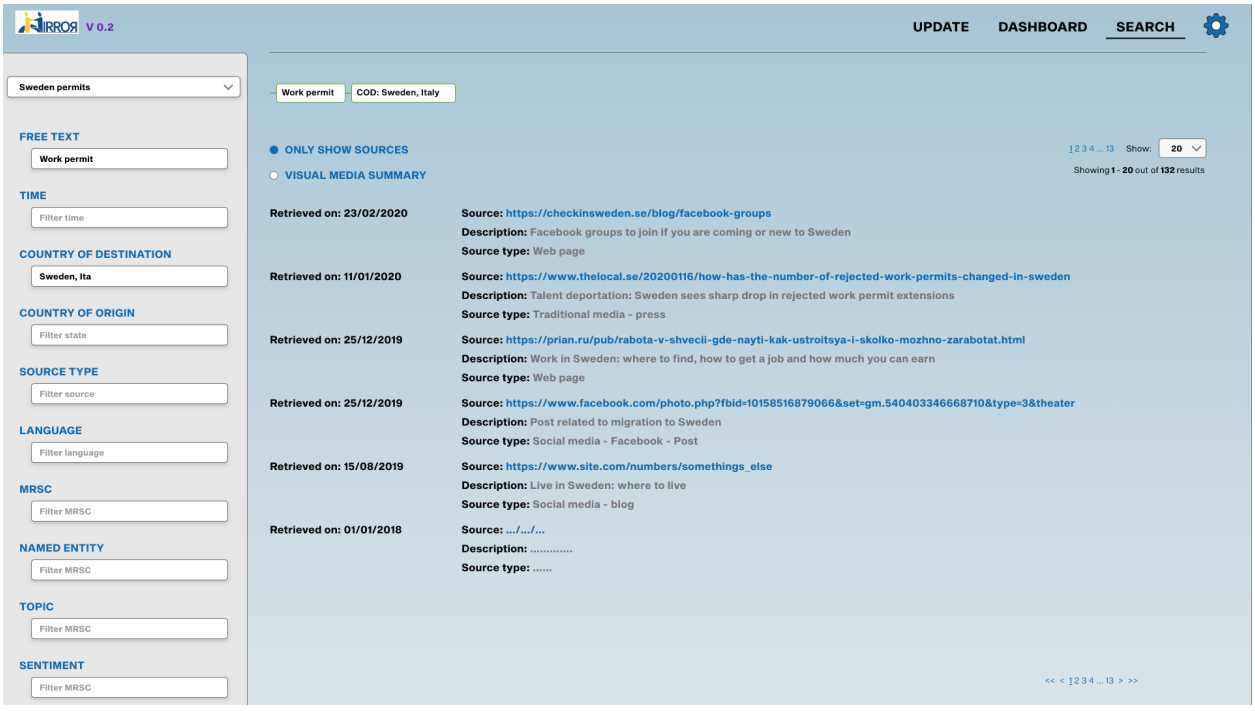

Figure 12: User Interface mock-up: search sources

For each result the user can access specific information for each media type (text, images, videos), as shown in Figures 13, 14 and 15, respectively. It is worth noticing that the provided information is generated by the

specific analysis tools and is different for each media type. For example, for what concerns the videos, the user can play it or access specific segments (shots), which could be more relevant. The transcription, provided by the Automatic Speech Recognition tool, is associated to the whole video or to a specific segments. For each media type specific indicators are provided and the topics identified in a specific excerpt of the text, image or video segment are highlighted.

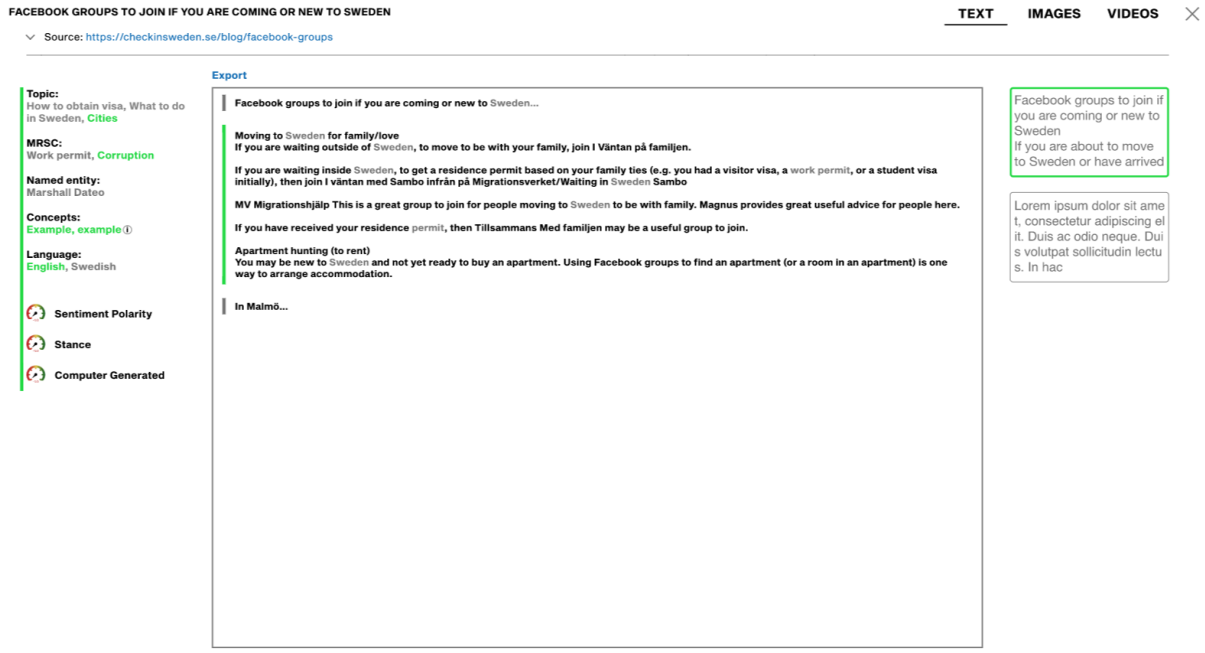

Figure 13: User Interface mock-up: text analysis results

Figure 14: User Interface mock-up: image analysis results

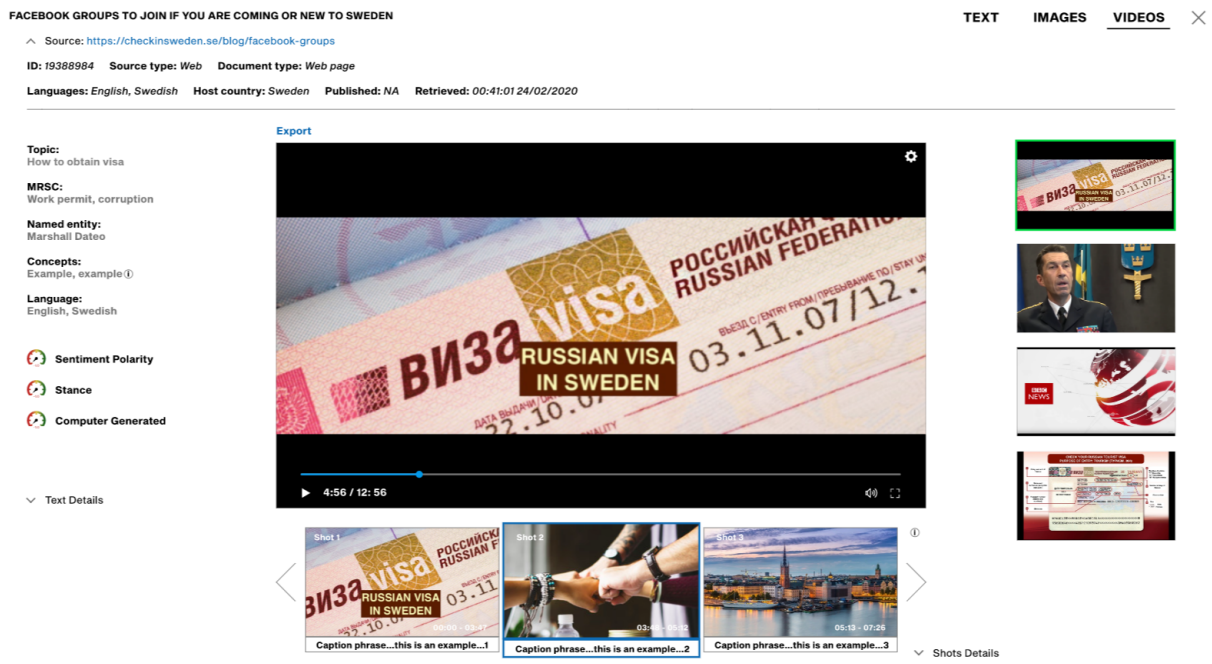

Figure 15: User Interface mock-up: video analysis results

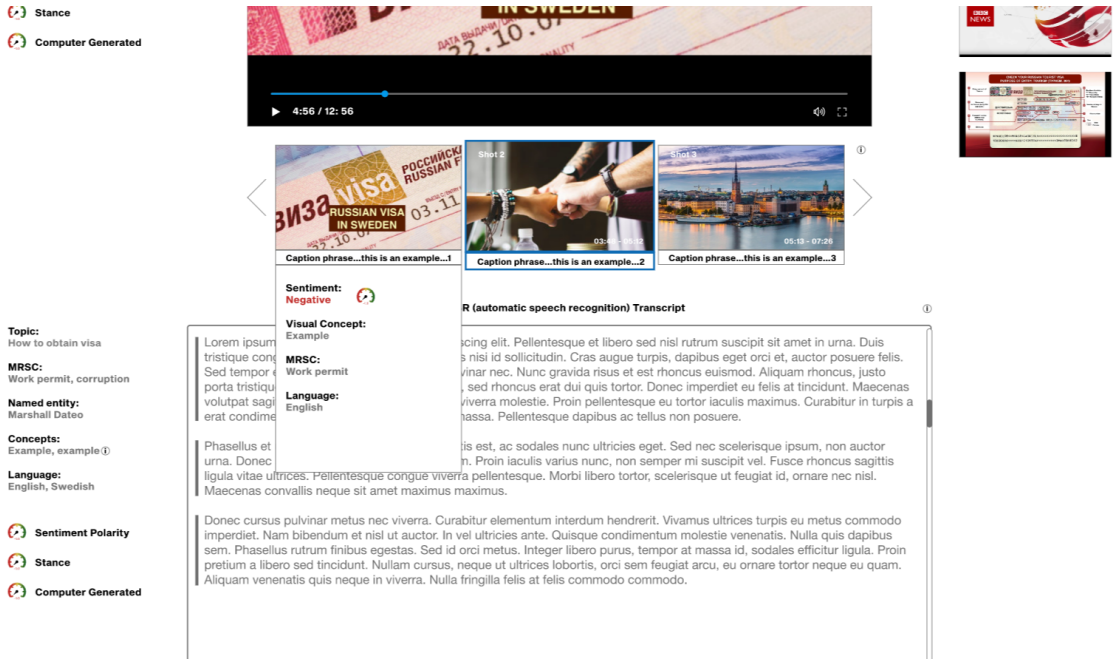

Figure 16: User Interface mock-up: video summarization and automatic speech recognition

The User Interface includes also a Dashboard, with a number of widgets providing information from cross-media analysis and enabling specific searches, either by Situations or e.g. by selecting specific points of interest on a map. This part of the User Interface is intended to show trending topics and other results on a specific time range.

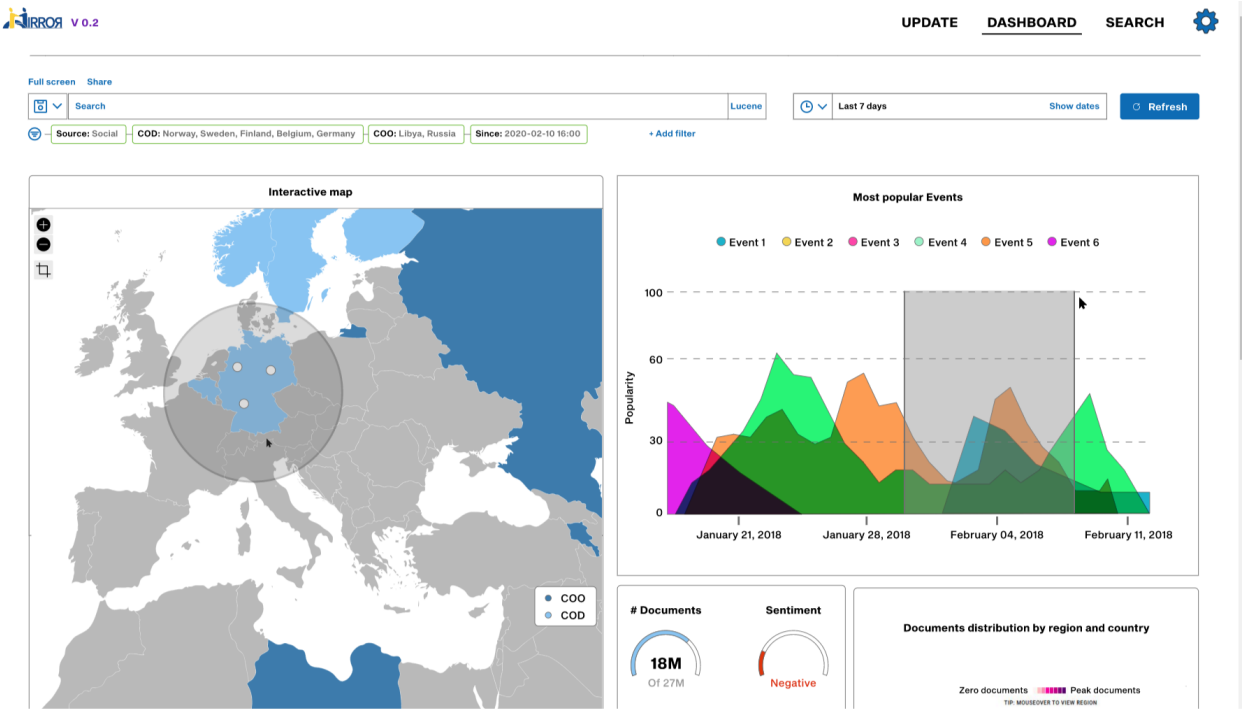

Figure 17: User Interface mock-up: dashboard page

8.2 User Interface Implementation

For the implementation of the front-end application we adopted the Angular 10 platform. There are several advantages provided by Angular 10, compared to other candidate technologies, although in the field of Javascript frameworks several valuable alternatives are available, such as AngularJS, VueJS and React, just to name a few.

Angular is open source, inspired by web standards, enhanced by modern capabilities, and includes development tools and is highly customizable. Angular provides a powerful and extensive ecosystem of free extensions and plugins, supported by a large community: it is sponsored by Google and the development is driven by the open source community. Angular is a quite popular solution for front-end applications and has proven to be successfully in small to large size applications for a huge variety of domains.

The most important factors behind the choice of Angular for the front-end development are briefly summarized in the following.

Angular is platform, not a framework. A framework is usually just the code library used to build an application, whereas a platform is more holistic and includes tooling and support beyond a framework. Angular comes with a leaner core library and makes additional features available as separate packages that can be used as needed. Angular also provides:

- a dedicated CLI for application development, testing, and deployment;
- offline rendering capabilities on many back-end server platforms;
- desktop-, mobile-, and browser-based application execution environments;

- a comprehensive UI component libraries, such as Material Design.

Component architecture The intention is to design each piece of the application in a standalone manner that limits the amount of coupling and duplication across various parts of the program[20].

Modern Javascript Angular is designed to take advantage of many features that are fairly recent to the web platform. Most of these became part of the JavaScript specification in 2015 with the release of ES2015 (also known as ES6)

TypeScript Angular itself is written with TypeScript, which is a superset of JavaScript that introduces the ability to enforce typing information. It can be used with any version of JavaScript, so you can use it with anything ES3 or newer. The basic value proposition of TypeScript is it can force restrictions on what types of values variables hold. It was a way to a get Javascript a more robust programming language.

Among the drawbacks of Angular in the context of MIRROR, it is worth mentioning that Angular has a steep learning curve, the implementation of specific features is often achieved at the expense of simplicity and is quite pervasive. In addition to the aforementioned advantages, Angular is stable, widespread, very well documented, used to build complex applications occasionally, as already mentioned, and offers for free all required tools for the application development. Other front-end frameworks such as React provide a lightweight core but require several external modules and plugins, while Angular, being a platform, includes all requirement dependencies for the development (the so called Angular ecosystem). Finally, Angular was considered a valuable choice also taking into account the expertise in the consortium.

Although Angular has been selected as the reference technology, nowadays another approach is gaining popularity for the development of front-end applications, the so called *microfrontends* (see for example [Newman, 2019]), which can be considered as the transposition of the microservices [Newman, 2015] to the front-end: the front-end application is implemented as a composition of separate micro front-end applications, each developed with its own framework (e.g. Angular, React, VueJS, etc.). This approach provides additional flexibility and the concurrent development of the different parts of the front-end, as done with the microservices for the back-end. This approach will be evaluated and eventually experimented, in order to verify the extensibility of the system.

The MIRROR framework User Interface is available at the following URL on the demo server (registration required):

https://d-mirror.l3s.uni-hannover.de/

The main entry point of the User Interface is a login page, as shown in Figure 18:

8.3 Other Client Applications

Although the MIRROR framework includes a User Interface, in special circumstances it could be necessary to integrate the platform with existing or third party systems or simply there could be specific requirements preventing the access to the User Interface. In such situations, the integration with the MIRROR system could be only via APIs (directly with the API Gateway). Although the User Interface includes relevant parts of the system functionalities, the integration of external systems with the APIs is not prevented and can provide valuable information to the users.

[20] https://github.com/w3c/webcomponents, last retrieved 31 July 2020

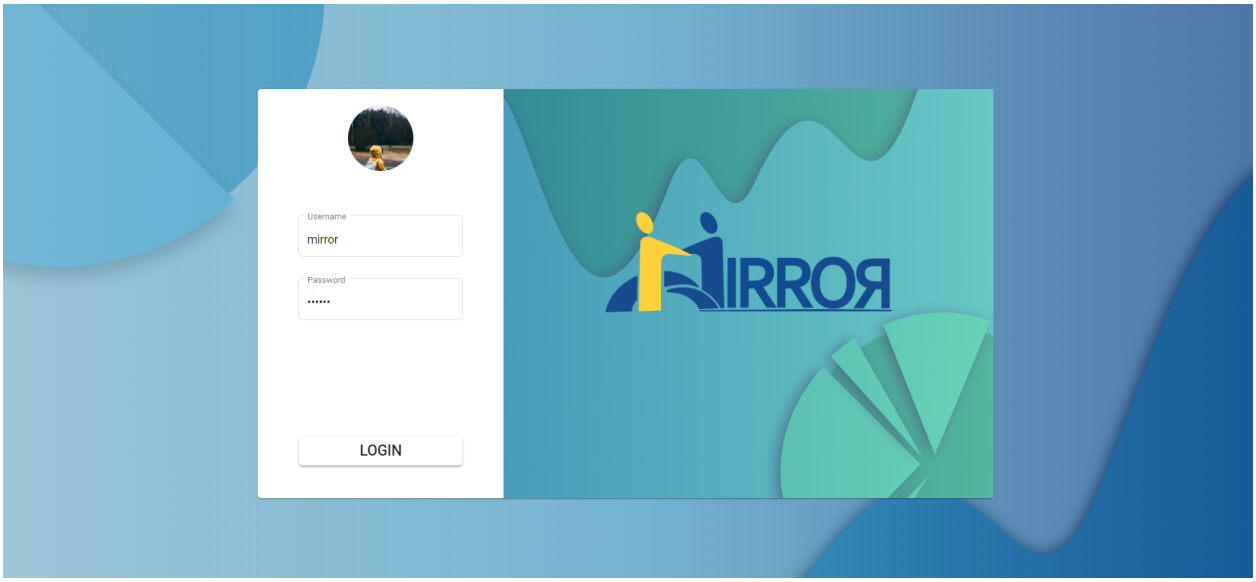

Figure 18: Login page of the MIRROR User Interface

9 Conclusion

In this deliverable we described the first prototype of the MIRROR framework. Starting from the scenarios identified in WP2 and from the architecture and the model designed in WP7, the prototype integrates the technical components from WP4, WP5 and WP6 into a coherent framework which comprises several layers: data management, data analysis, integration, access and client applications.

The framework prototype provides a validation of the MIRROR approach for the automated media analysis and the integration of technologies into a coherent framework targeting the main scenario identified in the project, namely the detection of migration-related (mis)information campaigns.

For each layer the role and the integration mechanism of the developed components has been described. The development of the framework is the result of the joint effort of all project partners, both technical and non technical, based on an Agile approach supporting the collection of feedback from users and frequent iterations among technical partners. The partners representing potential future adopters and users of the framework have provided their feedback and validated the requirements, mainly for what concerns the expected functionalities and user experience with the framework user interface.

A mock-up of the user interface has been iteratively created, in order to collect the feedback and create a common ground for the discussion. For what concerns the development activities, a collaborative environment has been established based on two Agile values: the collection of feedback from users and the timeboxed iterations during the development, organizing weekly calls and adopting tools to manage the tasks and priorities.

Finally, the environment for continuous integration of the results has been prepared, adopting cutting edge technologies for the development and deployment of the prototype. Targeting the project pilots, the deployment mechanism leverages DevOps practices, with automation and configuration management tools.

9.1 Assessment of Performance Indicators

The framework is one of the outcomes expected by the MIRROR project, in particular it is associated to Objective 5 (*Delivery of tools, practices and methods that enable awareness for impact*).

For what concerns the performance indicators, being an integration platform, it enables the assessment of the KPIs of the integrated components not only as individual tools but also in the context of the overall MIRROR system. For the assessment of the performance indicators of the technical components please refer to the corresponding deliverables from WP4, WP5 and WP6.

For what concerns the MIRROR system as a whole, the two related indicators are KPI12 (*Percentage of collected requirements fulfilled (mid project + full project)*) and KPI15 (*Level of user satisfaction with developed MIRROR system*). Both indicators are not supposed to be assessed directly in the first prototype deliverable, but it is worth mentioning that the design of the framework and in particular of the user interface has been based on the scenarios defined in WP2 and on iterations with the project users, who had the opportunity to provide their feedback not only during plenary meetings, but also in periodic WP7 calls dedicated to the discussion of the system and of the user interface. Targeting the project pilots, an assessment of the level of user satisfactions and a review of the collected requirements will be performed.

9.2 Next Steps

The project will deliver three main releases of the framework. After the first prototype implementation described here, the second year of the project will focus mainly on development and engineering of the system, on improving the automated deployment mechanism for the continuous integration of the new functionalities and on testing. In the meanwhile the information model will be refined and completed towards the release of the final version in D7.4 at M27. The second release of the MIRROR framework will be described in D7.5 at M30.

10 References

[Hohpe and Woolf, 2003] Hohpe, G. and Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., USA.

[Humble and Farley, 2010] Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition.

[Kim et al., 2016] Kim, G., Debois, P., Willis, J., and Humble, J. (2016). *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press.

[Newman, 2015] Newman, S. (2015). *Building Microservices*. O'Reilly Media, Inc., 1st edition.

[Newman, 2019] Newman, S. (2019). *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Incorporated.

Glossary

COD Country of Destination. 31

COO Country Of Origin. 32

MIRROR-CIM MIRROR Conceptual Information Model. 15, 31

MRSC Migration-Related Semantic Concept. 9, 32