Research and Innovation Action (RIA) H2020 – 957017

Stream Learning for Multilingual Knowledge Transfer

https://selma-project.eu/

D4.1 Platform Architecture and API Documentation

Work Package	4
Responsible Partner	IMCS
Author(s)	Guntis Barzdins, Sebastião Miranda
Contributors	Arturs Znotins, Didzis Gosko, Mikus Grasmanis, Afonso Mendes, Paulis Barzdins, Roberts Dargis, Normunds Gruzitis, Christoph Andreas Schmidt
Version	1.0
Contractual Date	31 October 2021
Delivery Date	29 October 2021
Dissemination Level	Public

Version History

Version	Date	Description
0.1	17/08/2021	Initial Table of Contents (ToC)
0.2	20/09/2021	TokenQueue description added
0.3	04/10/2021	Integration with UC1 and UC2 section added
0.4	14/10/2021	Basic testing/configuration UC0 introduced
0.5	20/10/2021	Ready for internal review
0.6	25/10/2021	Internal review
0.7	26/10/2021	Final version for PM
0.8	28/10/2021	Last minute changes by all partners
1.0	29/10/2021	Final version for submission

Executive Summary

The following platform architecture and API documentation provide a detailed description of the software platform to be built within the SELMA project. The SELMA project software stack is split in three parts: (i) The **SELMA platform** which is a highly scalable standalone processing and orchestration platform (Open Source), (ii) **NLP containers** developed in WP2/3 (mixed Open Source / Proprietary) and (iii) **client-applications** developed in WP1/5 specific to the use-cases (Proprietary). The SELMA platform builds on top of the time-proven industry standard technologies such as RabbitMQ, JSON, SQLite and PostgreSQL, Kubernetes to avoid the pitfalls of the legacy approaches, which relied on now phased-out RethinkDB and Docker-swarm technologies.

SELMA's central concept is to build a deep-learning NLP platform that trains unsupervised language models, using a continuous stream of textual and video data from media sources and make them available in a user/topic-oriented form in over 30 languages.

The knowledge learnt in the form of deep contextual models is transferred to a set of NLP tasks and made available to users through a **Media Monitoring Platform** (Use Case 1) to be able to handle up to ten million news items per day. The media monitoring platform will be able to transcribe, translate (on demand), aggregate, write abstractive summaries, classify, and extract knowledge in the form of entities, relations and topics and present all this to the user using new visualizations and analytics over the data. The learnt contextual models will also be applied to a **News Production Tool** (Use Case 2), using enriched models for transcription (ASR) and translation (MT), giving journalists in an operational editorial environment a multilingual tool that will be able to learn over time. An additional **Basic Testing and Configuration Interface** (Use Case 0) is introduced in this document for the sake of design fidelity.

Table of Contents

Table of Figures

Table of Tables

1.Introduction

This document describes the SELMA platform architecture for integrating the natural language processing (NLP) components developed in WP2 and WP3 into the highly scalable (up to 10M documents/day) NLP pipeline serving as the backend for the SELMA end-user applications as defined in the use cases U1/U2 elaborated in WP1 and WP5. Additional requirements of the SELMA platform architecture are collection and storage of editorial corrections and integration of the continuous massive stream learning functionality into the extended NLP pipeline. The SELMA platform will be accessed through the API described in this document.

The SELMA platform architecture is emerging not in an empty place – rather it is building on the substantial experience and codebase acquired by the SELMA consortium partners who have been developing NLP pipelines previously. Most notably the SELMA architecture is influenced by the pioneering SUMMA platform[1] developed in 2016-2019 within the H2020 project "Scalable Understanding of Multilingual MediA" elements of which were successfully reused by the consortium partners in a number of follow-up media-applications such as NewsBridge, PlainX, Insight, LV-pipeline, PiniTree and others.

Striking the balance between the proven microservices based platform architectures (such as SUMMA) and novel cloud DAG (directed acyclic graph) pipeline environments (such as Kubeflow, Airflow, Metaflow, Argo, Prefect, Nvidia Triton, Apache Spark, Pachiderm, Couler, MLFlow, Flite) has been the core challenge during the design of the SELMA platform architecture. The SELMA architecture falls in the gap between the dynamic DAG flows and fixed microservices architectures, as high efficiency (10M documents/day scalability goal) can only be achieved by continually re-using the NLP model containers preloaded in the computer memory (a feature present only in Nvidia Triton). We bridge this gap with the Maestro Orchestrator and TokenQueue techniques – novel approaches developed in the SELMA project. Simplicity has been another design objective, as the legacy SUMMA platform complexity was a major obstacle for its wider adoption and re-use in the open-source community.

[1] http://summa-project.eu/

2. The Scope of the SELMA Platform

In this section we define the boundaries of the term "SELMA Platform", as one might be tempted to interpret "SELMA Platform" as all software developed in the SELMA project. In reality, in the SELMA project, we will develop a number of independent software packages both for the natural language processing components (such as automatic speech recognition or named entity recognition in WP2 and WP3) and the end-user applications for the multilingual media monitoring (UC1) and multilingual media production (UC2) use-cases elaborated in WP1 and evaluated in WP5. By the "SELMA Platform", we only refer to the "glue" layer, which seamlessly glues together all these independent software components, developed by various partners in different software development environments, into a single fully functional system.

2.1 The SELMA Project Software Stack

The following high-level diagram (Fig. 1) illustrates the entire envisioned SELMA project software stack.

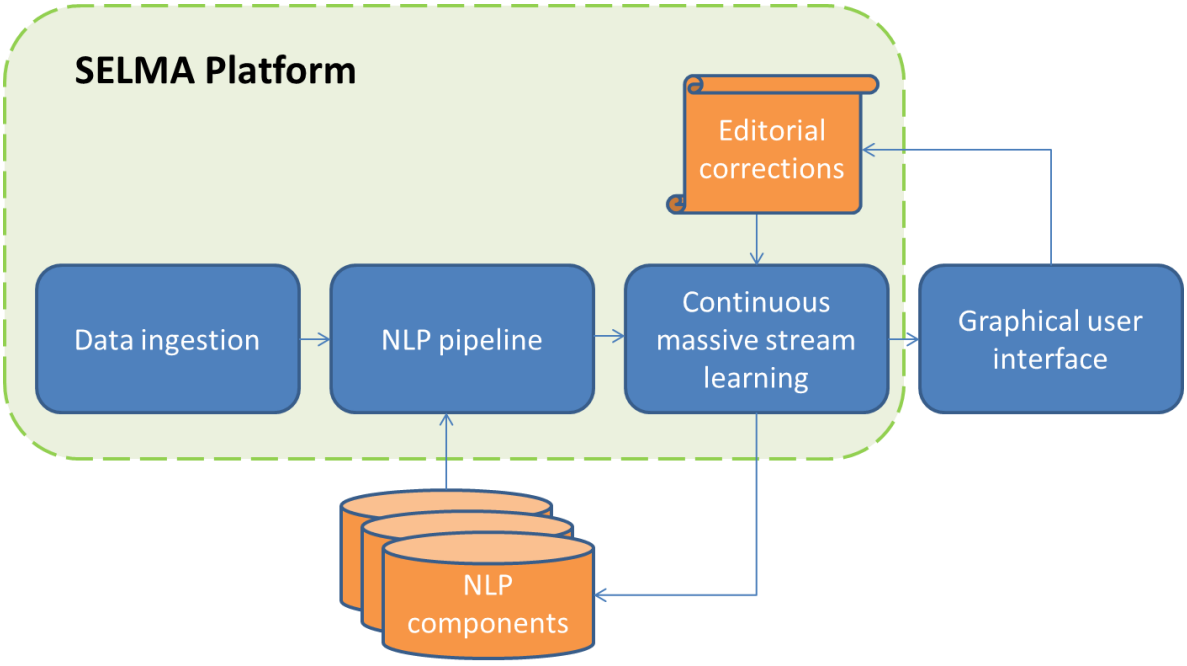

Figure 1 The SELMA project software stack

The above diagram lists the core blocks of the fully functional SELMA project software stack: the multilingual and multimodal news media documents are ingested from various external sources and passed to the NLP pipeline for enrichment by automatic transcription, translation, etc. The enriched media documents are further sent to the end-user application via the continuous massive stream learning module, which records the editorial corrections made by the users and uses them to re-train the NLP pipeline components.

2.2 The SELMA Platform

The dashed rectangle in the Figure 1 shows the scope of the SELMA platform acting as a "glue" between the independently developed NLP components (WP2, WP3) and independently developed end-user applications components (WP1, WP5). Restricting the SELMA platform scope to only the "glue" function is what makes its universal API also applicable beyond the SELMA project as a self-sustainable open-source project: it can be used for other kinds of NLP components and for other kinds of graphical user interfaces or applications.

Restricting the SELMA platform only to the "glue" function also resolves the issue of separating the proprietary NLP components and proprietary end-user applications to be used in the SELMA project – such as PlainX, Insight, streaming news clustering (from Priberam and Deutsche Welle), advanced speech transcription (University of Avignon, Fraunhofer), and the PiniTree knowledge base (University of Latvia, IMCS).

The release of the open-source SELMA platform (at the end of the project) is expected to include also a basic set of open-source NLP components and a basic testing and configuration graphical user interface referred to as Use Case 0 (UC0) providing a template for more advanced UC1 and UC2 implementations.

Functional requirements for the SELMA platform are described in the SELMA deliverable D1.1 and have been an integral part of the SELMA platform design process leading to this document.

3. The Architecture of the SELMA Platform

Although the basic principles of the SELMA platform architecture were laid out already in the SELMA project proposal as evolution on top of the successful SUMMA project (2016-2019), the landscape of the available software development technologies and environments has changed in the meantime. Additionally, significant experience has been accumulated with the legacy SUMMA architecture to see which decisions back then were right (use of JSON as core data storage and exchange format; use of RabbitMQ as core job queue) and which turned out to be less successful (Rethink DB with Live Queries as core storage; reliance on Docker-swarm as core scaling environment).

If there is something to be learnt from the above experiences, it is avoiding as core functionalities promising, yet experimental technologies, which have not yet stabilized as industry standard and are phased out soon afterwards (Rethink DB, Docker-swarm). Not to fall into the same trap again, the SELMA platform will switch to the time-proven industry standard SQL databases (SQLite, PostgreSQL) and Kubernetes (local and cloud based) as the core storage and scaling environments (along with Docker and Docker-compose support for smaller deployments).

3.1 Maestro Orchestrator: Core of The SELMA Platform

The SELMA platform architecture has been in a prototyping and testing stage by the SELMA partners already for several months and will be based on the Maestro Orchestrator[2] software package (Fig. 2) originally designed by the consortium partner Priberam and further enhanced by IMCS. Maestro Orchestrator supports a REST API documented in the Annex "SELMA Platform API".

[2] https://github.com/SELMA-project/orchestration-maestro

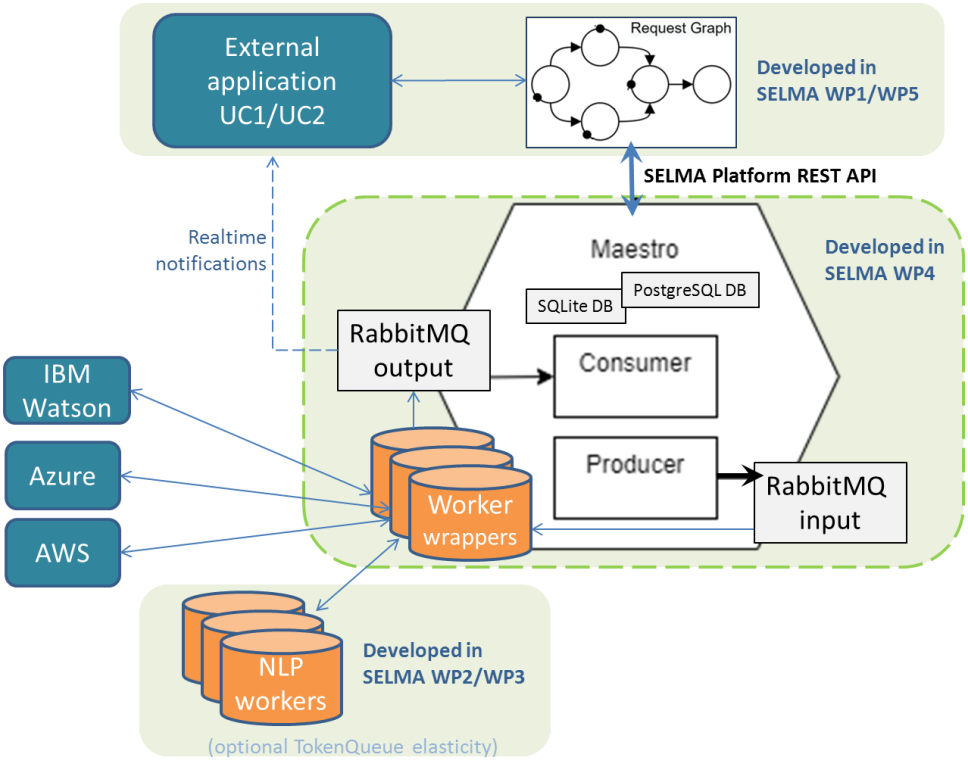

Figure 2 Architecture of the Maestro Orchestrator

Maestro Orchestrator is a task-agnostic DAG (directed acyclic graph) workflow execution engine designed for running NLP pipelines and handling job queues. Given a graph with the jobs as nodes and the dependencies as directed edges (encoded in JSON, see Appendix), Maestro Orchestrator maximizes the number of jobs running in parallel, while ensuring each job has the necessary input. Furthermore, it provides a RabbitMQ communication framework with multithreaded prefetching option for the development of autonomous workers or external worker wrappers compatible with the aforementioned Orchestrator.

Maestro Orchestrator can be viewed as a "glue" between the REST API facing the external application and RabbitMQ facing the active (preloaded) NLP container instances accessible via REST API. To handle the REST API interface and job queues, the Maestro Orchestrator includes a persistent SQL database (SQLite or PostgreSQL – a configurable option) shadowing the transient RabbitMQ contents. The Maestro Orchestrator is implemented in a .NET environment as a set of containers launched either by a Docker-compose script (in a Docker environment) or by a Kubernetes resources .yaml file (in a Kubernetes environment).

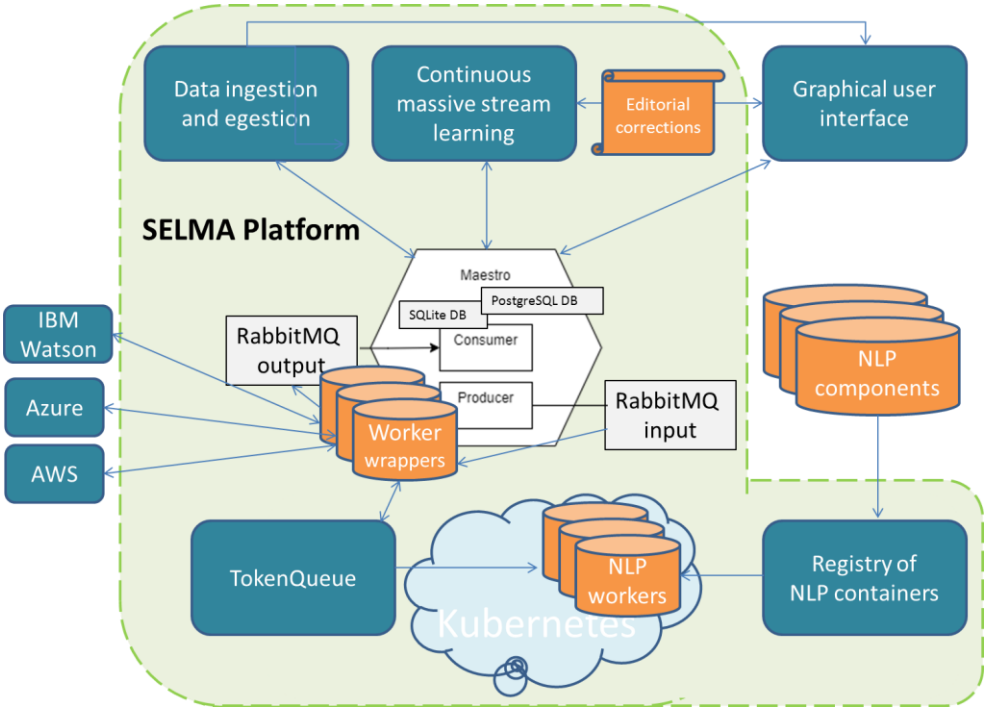

Figure 3 Core SELMA platform internal architecture

With Maestro Orchestrator already implemented and handling the central "glue" functionality of the SELMA platform, the overall SELMA platform architecture (Fig. 3) is reduced to the number of extensions for the Maestro Orchestrator to support the full functionality of the SELMA project software stack. In the following, a list of the Maestro Orchestrator extensions and features along with their envisioned implementation is given:

(A) **TokenQueue** (described in more detail in the next section): functions to dynamically launch and stop multiple NLP worker instances based on the SELMA platform load similarly to industrial NLP services from MS Azure, AWS, IBM Watson and Google Cloud. Because NLP workers developed in WP2 and WP3 usually support a REST API and custom data formats, each NLP worker will be paired with a worker wrapper for communication with the RabbitMQ and custom input/output format conversion. In a production environment, the TokenQueue module might need to interact with the billing system to bill the compute resources to the respective users.

(B) **NLP container registry**: stores locally the NLP container images for fast dynamic loading into the Kubernetes pods by the TokenQueue upon demand. Harbor cloud native repository (https://goharbor.io/) for Kubernetes and Docker is a possible option.

(C) **Ingestion and egestion of the documents:** A single large JSON file (with the structure inherited from the legacy SUMMA platform) will be used to store and export all data (transcripts, translations, etc.) produced for each ingested document to enable arbitrary data dependencies between the NLP processing jobs in the DAG pipeline. Smaller JSON files could be used in the intermediate steps for interactive applications aimed at continuous massive stream learning, e.g. when a user corrects an ASR transcript, only the affected paragraph is re-translated.

(D) **Continuous massive stream learning** based on the editorial corrections collected from the SELMA platform end-users. From the implementation point of view, the continuous massive stream learning is just another NLP workflow periodically executed by the Maestro Orchestrator. The only difference is that the outcome of this job is an updated NLP container, which needs to be stored as a new container version in the NLP container repository (B) and is subsequently labeled as "latest" for use by the TokenQueue (A).

(E) **Long-term storage of the ingested documents and media files:** Initially the ingested documents and media files are stored by the Maestro Orchestrator in its internal SQL database. Meanwhile, for performance reasons the Maestro Orchestrator should not be used as the long-term massive content storage (this was one of the design flaws with Rethink DB in the legacy SUMMA platform). Ideally the Maestro orchestrator SQL database should be purged daily to maintain an acceptable performance at the top load of 10M documents per day. Therefore, the long-term content storage must be handled by the use-case specific frontends / external systems optimized for massive long-term storage.

(F) **Scalability considerations:** The Maestro platform along with NLP workers can be scaled through sharding. Exceptions to simple scaling through sharding are global search/clustering operations over entire long-term storage of the ingested documents (E). As long as this long-term storage and search/clustering is handled by the use-case specific frontend, this does not affect the SUMMA platform design. This design will be

validated and might be reconsidered as part of the SELMA platform scalability tests later in the project.

3.2 TokenQueue: Management of the Elastic Scaling

TokenQueue is a mechanism invented in the SELMA project[3] to convert single-threaded NLP docker containers developed in WP2 and WP3 into elastically scalable NLP API services like those provided by MS Azure, AWS, IBM Watson, Google Cloud. While these traditional industrial solutions rely on complex middleware handling authentication, billing, queueing and scaling of the NLP services, the TokenQueue is a lightweight yet highly scalable side-car approach not intervening in the actual communication between the API client and server. The TokenQueue side-car approach means that the API is not restricted to any specific technology and can use REST , WebSocket or any other protocol for communication as is typical for the above-mentioned industrial solutions. The TokenQueue side-car handles only the elastic NLP container scaling in Docker or Kubernetes compute-cloud based on the NLP job influx rate (demand). A distinct feature of the TokenQueue approach is that multiple NLP job types (such as speech recognition, machine translation, named entity recognition, etc.) can be handled in a single TokenQueue leading to the optimal Docker or Kubernetes compute resource sharing between the NLP service types, which used to be a problem of its own in legacy approaches such as in the SUMMA project.

To achieve the above-mentioned functionality, the TokenQueue mimics the popular customer service optimization technique widely deployed in the physical world, where an arriving customer immediately takes the queue token (ticket) with a unique number and then waits for his number to appear on the queueing screen directing him to one of the service counters. The beauty of this solution is that the number of service counters can be elastically changed based on the customer influx rate (demand) without disrupting the queue.

[3] https://github.com/SELMA-project/token-queue

Barzdins et al., 2021. Metamodel Specialisation Based Tool Extension. Baltic J. Modern Computing (submitted)

Figure 4 SELMA TokenQueue mimics a real-world customer service queueing approach

The TokenQueue approach for NLP service scaling is already implemented by IMCS and demonstrated for the simple stand-alone NLP pipeline[4]. It is also implemented in the highly scalable GUI application with interactive on-demand NLP pipeline[5] for speech recognition, text segmentation and machine translation into 30+ languages of the SELMA project (Fig. 5) serving as the blueprint for the UC0 application.

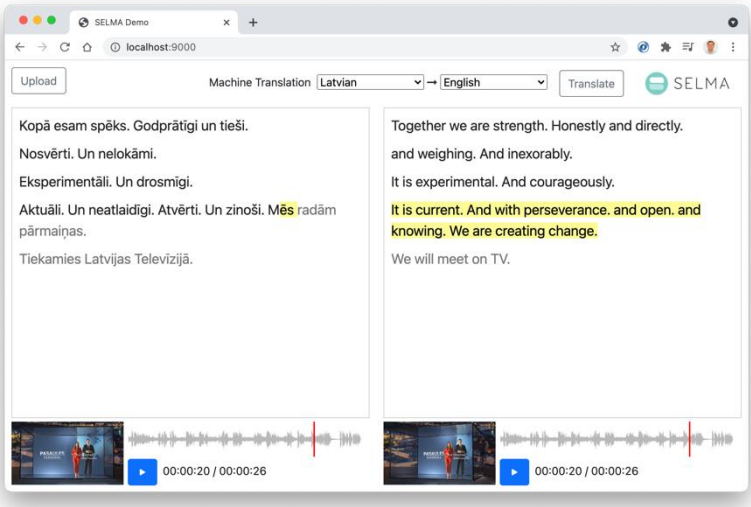

Figure 5 Basic Testing and Configuration application (UC0)

[4] https://github.com/SELMA-project/makefile-orchestrator-demo

[5] https://github.com/SELMA-project/selma-v8

3.3 Use Case 0: Basic Testing and Configuration Application

For the sake of design fidelity, we will call the Basic Testing and Configuration application illustrated in Figure 5 Use Case 0 (UC0) as it will eventually be extended into the SELMA Platform backend configurator (for scaling Kubernetes/Docker compute resources) and testing interface for the NLP modules developed in WP2 and WP3 as they get integrated into the SELMA platform.

The UC0 GUI application also illustrates the elastic scalability of the TokenQueue approach as it runs entirely in RAM and thus scales beyond 10M documents/day (web client connections or API calls) on an adequate hardware. The initial scalability tests were conducted on intel64/amd64 processor hardware and arm64 Apple M1 processor hardware – the latter being the most energy-efficient. Implemented in GO language it launches a concurrent go-routine executing an NLP pipeline for each connected client independently, except for shared TokenQueue. At the heart of the TokenQueue implementation is the YAML configuration file listing available NLP worker containers along with their type and URL (see below).

workers.yaml

```yaml
- type: asr
  engine: lv
  version: 0.1
  url: http://194.8.1.234:9001/post

- type: segmenter
  engine: default
  version: 0.1
  url: http://194.8.1.234:7811/

- type: mt
  engine: lv-en
  version: 0.1
  url: http://194.8.1.234:8000/translate
```

Figure 6 Central to TokenQueue configuration is the list of available NLP workers

The TokenQueue handles the elastic NLP container scaling in Docker or Kubernetes compute-cloud by starting or shutting down extra NLP workers of specific types based on the NLP job influx rate (demand) and dynamically updating the *workers.yaml* file shown above. A dedicated token server is set up (this is what we call the Token Queue); the requesting API client queries the TokenQueue for a URL of a free worker, and then just waits on this query until a worker is free, and its URL is returned. Then the requesting API client directly executes the job on that

free worker and after it is done returns the URL back to the TokenQueue, so that the worker can be offered to the next in line. The side-car implementation of TokenQueue assigns the workers to the requesting API clients in the three-step approach illustrated below by a simple bash script (a more fail-safe code will be used in the actual implementation):

```
curl -o url.tmp -s 'http://TokeQueue.ailab.lv:9000/api/workers/acquire?type=mt'
curl "`cat url.tmp`"  -d "@segm.json"  > translationEN.json
curl -s 'http:// TokeQueue.ailab.lv:9000/api/workers/release' --data \@url.tmp
```

In the above example, the actual client API request to the NLP worker (second line) is wrapped between two side-car TokenQueue requests – the first one requests a URL for the worker of the specified type "mt" (Machine translation) while the last one releases that URL back to the TokenQueue. If all NLP workers of the specified type are busy, the waiting occurs in the first side-car TokenQueue request until one of two events happens:

(a) Another TokenQueue API client releases a URL for the specified type of NLP worker,
(b) TokenQueue starts an additional instance of the NLP worker with the specified type.

Due to its three-step side-car implementation the only true scalability limitation of the TokenQueue approach is 65536 outgoing TCP ports per IP address for connections to the NLP workers, the problem known as "Ephemeral Port Exhaustion". This would become a limitation for the 10M/day performance envisioned in the SELMA project only if a single document is processed for longer than 10 min on average or documents are ingested unevenly throughout the day (e.g., only during the working hours of the company). In case this ephemeral port exhaustion becomes a bottleneck, a possible workaround would be sharding or binding a percentage of outgoing connections to a list of static local IP addresses through NGINX integration; this could also offer additional scalability, security and proxy functions.

3.4 SELMA Use Cases in the Integrated Platform

In the previous sections we introduced the core SELMA Platform components Maestro Orchestrator and TokenQueue. This chapter presents how they work together in the integrated SELMA platform. The integrated architecture is generic enough to accommodate many other use-cases which have streaming or on-request requirements, thus being a valuable Open-Source

contribution instead of just the backend for the UC0/UC1/UC2 interfaces. Figure 7 shows the integrated SELMA architecture, consisting of 5 main components: (i) A GUI for platform administration, the (ii) The SELMA Maestro graph orchestrator (iii) an existing message queue component - RabbitMQ, a (iv) The SELMA TokenQueue component to scale workers elastically according to workload, and a (v) microservice worker management framework (Kubernetes or Docker).

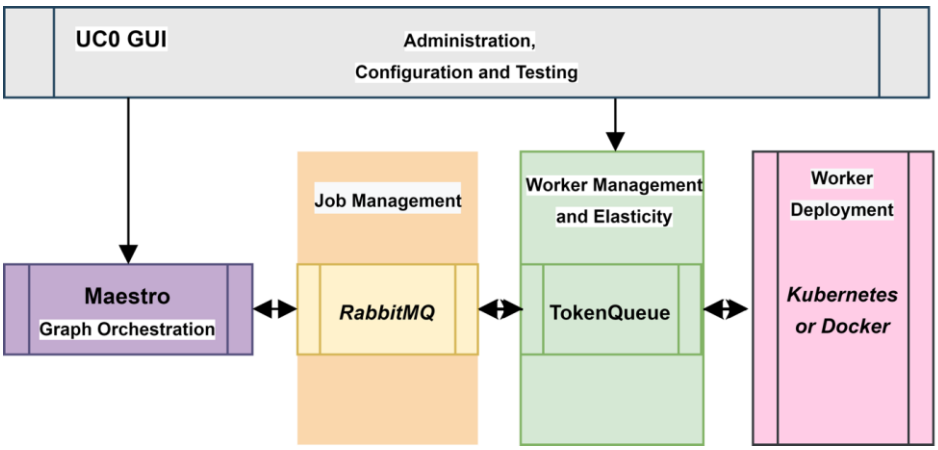

Figure 7 Integrated SELMA platform architecture

Each component handles a part of the processing pipeline, and the UC0 GUI allows administrating the platform and testing the components. The processing results can be integrated in a client application (such as UC0/1/2) in different ways to facilitate the extensibility of the platform and its usage as an Open-Source project:

1) Actively register a listener to receive job updates. This is useful for attaching other pipelines into SELMA, or to show live updates on a GUI. Technically, this is possible e.g. by registering a websocket listener after the results queue.
2) Query the result of a specific job. Technically this is possible if a large history is maintained with the document results – depending on the use-case, history purging can be configured.
3) Creating a Job Graph Workflow in which the last node is an export job (e.g., a worker which sends the final result to an external system).

Figure 8 displays a more detailed diagram of the architecture presented in Figure 7. It shows a few extensions added to the Core of the Maestro graph orchestrator (e.g. MaestroCore as

presented before) which are required to make job nodes compatible with each other – this is done by applying a Json Patch[6] which transforms the JSON document into the format needed for each job node / service API format. The request graph is constructed and sent to Maestro Core, which orchestrates the graphs through the message queues, and eventually all partial results are communicated back to the client application. Behind the message queues we either have external cloud APIs (which handle their own elasticity) or TokenQueue, as discussed in the previous chapter, which handles the elasticity for the SELMA workers or any other self-deployed workers that a user might wish to manage.

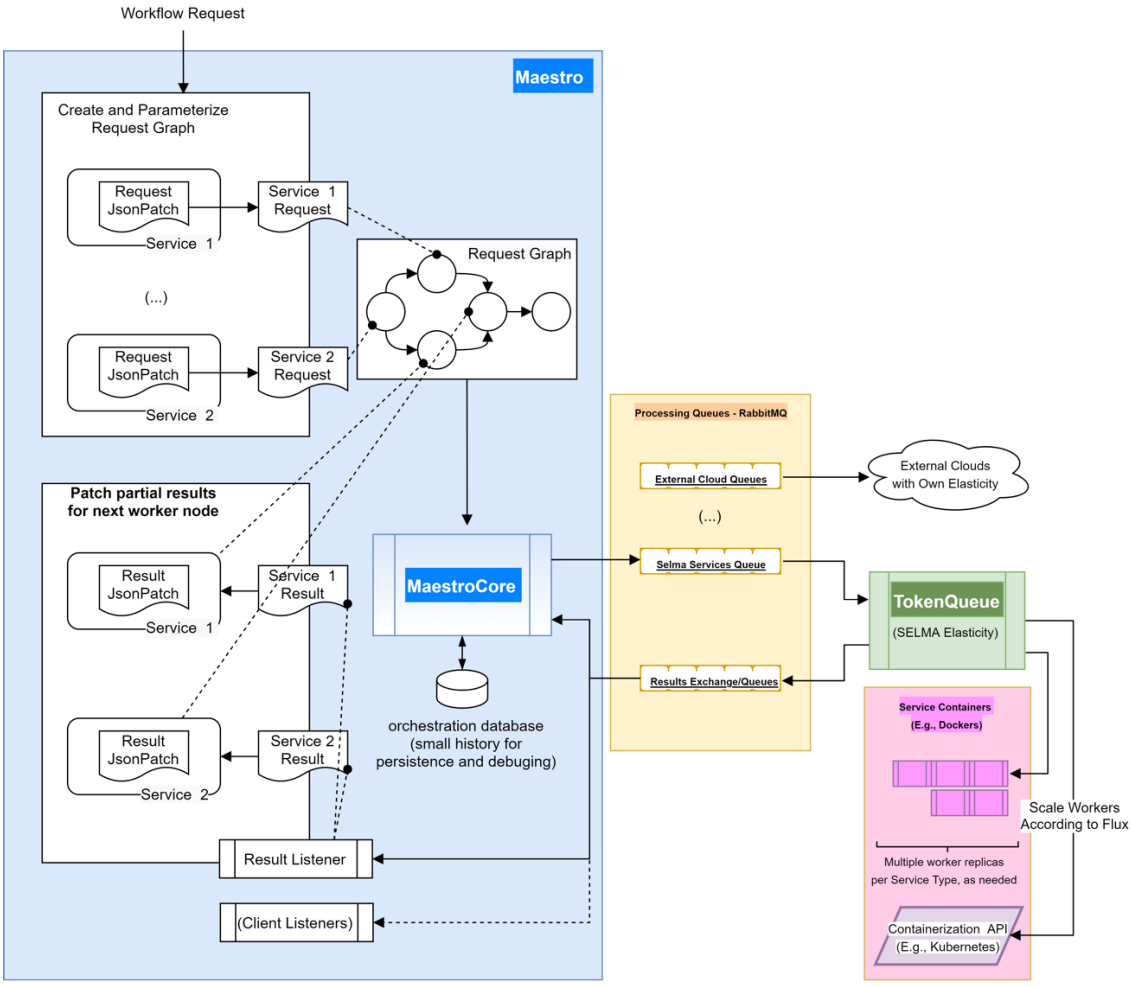

Figure 8 The integrated SELMA platform expanded in more detail

[6] https://datatracker.ietf.org/doc/html/rfc6902

3.5 Kubernetes and Scalability

Before taking the decisions to switch the SELMA platform core technologies to Kubernetes and SQL databases (SQLite, PostgreSQL), these were thoroughly tested in the small-scale pilot installations.

Kubernetes testing was the most difficult part as it involved a notoriously steep learning curve compared to the legacy Docker-swarm approach used in the legacy SELMA project (there is a reason why Kubernetes setup and administration is considered a high-skilled full-time job). There are at least three pitfalls when switching from the Docker-swarm to Kubernetes:

- Kubernetes cluster nodes do not use the native IP network but rather rely on external packages for Container Network Interface (CNI) plugins to handle networking
- Access to CUDA GPUs for neural network acceleration is not handled via regular Linux drivers, but rather a bare-metal GPU that has to be dedicated to the Kubernetes cluster
- Docker-compose integration scripts are replaced by very different Kubernetes resources .yaml definitions of Kubernetes pods.

Despite all these difficulties, SELMA project partners IMCS and Priberam successfully installed three different Kubernetes cluster variations: Minikube, DockerDesktop with Kubernetes and Canonical MicroK8s. The Kubernetes installations were tested with several applications – a neural machine translation NLP container and with the Maestro Orchestrator, ported from the docker-compose to the Kubernetes pod.

Although moving away from the Docker-swarm to the industry standard Kubernetes is difficult, besides local scalability it enables unprecedented scalability in the large cloud infrastructures such as AWS, Microsoft Azure, DigitalOcean and others.

SQLite and PostgreSQL scalability testing was rather straightforward, because the project partners (IMCS, Priberam) already had substantial experience with these environments in production use. Nevertheless, a basic SQLite + GO + REST API integration test was conducted by IMCS in the PiniTree platform (it is similar to the envisioned UC0) to ingest a large number of news media documents and process them via REST API calls to the existing Latvian NLP pipeline in a fashion similar to the envisioned SELMA platform (see results Table 1). The core

question for this test was how many parallel shards as Kubernetes pods would be needed to achieve the SELMA platform big data scalability goal of 10M documents per day.

Test description	**Processed documents**	**Processing time real**	**Comments**
Actually tested	75,794	2h42m49s	NLP pipeline: tokenizer, POS, inflection generation
Extrapolated to 24h	670,345	24h	Same NLP pipeline
Extrapolated to 10M documents in 24h	10,000,000	24h x 14.9 shards	14.9 copies of the NLP pipeline would be needed

Table 1 Scalability of SQLite + GO + REST API architecture

The results in Table 1 indicate that with this architecture the 10M documents per day target can be achieved with 15 SELMA platform shards running in parallel, which seems a reasonable number even without any further optimization. No specific tests were run with PostgreSQL, because this platform is highly scalable on its own, but would be difficult to maintain for the large number of shards.

4. Conclusions

The core part of the SELMA platform architecture described in this document is already implemented in the Maestro Orchestrator software developed and tested by the SELMA consortium partners. Availability of the operational core platform will simplify the provision of the remaining extensions necessary for the full implementation of the SELMA platform software stack, as described in this document.

5. Annex: SELMA Platform API

The Selma platform API is defined by the Maestro Orchestrator REST API illustrated on the live example below.

Maestro Orchestrator REST API input example:

curl -X POST "http://localhost:10000/Orchestration/Graph" -H "accept: text/plain" -H "Content-Type: application/json-patch+json" -d "{belowJSON}"


```
{
 "workflowId": "f3bd989f-bbdb-4851-857c-549b884e3641",
 "jobNodes": [
  {
   "id": "abba189f-bbdb-4851-857c-549b884e3641",
   "dependencies": [],
   "jobData": {
    "Worker": "ASR-LV",
    "Text": "http://selma.ailab.lv:2020/files/4963f238-9b83-4b37-9553-dc8ae608d719"
   },
   "jobType": "EasyPython",
   "jobProvider": "Selma"
  },
  {
   "id": "abba289f-bbdb-4851-857c-549b884e3641",
   "dependencies": [
    "abba189f-bbdb-4851-857c-549b884e3641"
   ],
   "jobData": {
    "Worker": "ASR-Punctuation"
   },
   "jobType": "EasyPython",
   "jobProvider": "Selma"
  },
  {
   "id": "abba389f-bbdb-4851-857c-549b884e3641",
   "dependencies": [
    "abba289f-bbdb-4851-857c-549b884e3641"
   ],
   "jobData": {
    "Worker": "EasyNMT",
```

```
    "source_lang": "lv",
    "target_lang": "en"
   },
   "jobType": "EasyPython",
   "jobProvider": "Selma"
  },
  {
   "id": "abba489f-bbdb-4851-857c-549b884e3641",
   "dependencies": [
    "abba289f-bbdb-4851-857c-549b884e3641"
   ],
   "jobData": {
    "Worker": "EasyNMT",
    "source_lang": "lv",
    "target_lang": "de"
   },
   "jobType": "EasyPython",
   "jobProvider": "Selma"
  },
  {
   "id": "abba589f-bbdb-4851-857c-549b884e3641",
   "dependencies": [
    "abba289f-bbdb-4851-857c-549b884e3641"
   ],
   "jobData": {
    "Worker": "EasyNMT",
    "source_lang": "lv",
    "target_lang": "pt"
   },
   "jobType": "EasyPython",
   "jobProvider": "Selma"
  },
  {
   "id": "abba689f-bbdb-4851-857c-549b884e3641",
   "dependencies": [
    "abba289f-bbdb-4851-857c-549b884e3641"
   ],
   "jobData": {
    "Worker": "EasyNMT",
    "source_lang": "lv",
    "target_lang": "fr"
   },
   "jobType": "EasyPython",
   "jobProvider": "Selma"
```

```
   }
  ],
}
```


Maestro Orchestrator REST API output example:

curl -X GET "http://localhost:10000/Orchestration/Graph?guid=3fa85f64-5717-4562-b3fc-2c963f66afa6" -H "accept: text/plain"


```
{
  "workflowId": "f3bd989f-bbdb-4851-857c-549b884e3641",
  "jobNodes": [
    {
      "jobResult": {
        "Timestamp": "2021-05-11T12:22:27.7933719+00:00",
        "Result": {
          "status": "ok",
          "words": [
            {
              "confidence": 1.0,
              "duration": 0.1699999962002039,
              "time": 1.0399999767541885,
              "word": "no"
            },
            {
              "confidence": 1.0,
              "duration": 0.30999999307096004,
              "time": 1.2099999729543924,
              "word": "darba"
            },
            {
              "confidence": 1.0,
              "duration": 0.07999999821186066,
              "time": 1.5199999660253525,
              "word": "uz"
            },
            {
              "confidence": 1.0,
              "duration": 0.4899999890476465,
              "time": 1.5999999642372131,
              "word": "mājām"
            },
            {
```

```
  "confidence": 0.823470413684845,
  "duration": 0.1599999964237213,
  "time": 2.0899999532848597,
  "word": "mēs"
 },
 {
  "confidence": 5.133163338832958E-10,
  "duration": 0.5099999886006117,
  "time": 2.249999949708581,
  "word": "braucām"
 },
 {
  "confidence": 2.8930416666217732E-15,
  "duration": 0.2899999935179949,
  "time": 2.7599999383091927,
  "word": "vienā"
 },
 {
  "confidence": 2.8675430102941034E-15,
  "duration": 0.05999999865889549,
  "time": 3.0499999318271875,
  "word": "un"
 },
 {
  "confidence": 1.0,
  "duration": 0.12999999709427357,
  "time": 3.109999930486083,
  "word": "tai"
 },
 {
  "confidence": 1.0,
  "duration": 0.36999999172985554,
  "time": 3.2399999275803566,
  "word": "pašā"
 },
 {
  "confidence": 1.0,
  "duration": 0.4899999890476465,
  "time": 3.609999919310212,
  "word": "laikā"
 },
 {
  "confidence": 1.0,
  "duration": 0.40999999083578587,
```

```
          "time": 4.379999902099371,
          "word": "visu"
        },
        {
          "confidence": 1.0,
          "duration": 0.40999999083578587,
          "time": 4.789999892935157,
          "word": "cauru"
        },
        {
          "confidence": 1.0,
          "duration": 0.29999999329447746,
          "time": 5.199999883770943,
          "word": "gadu"
        }
       ]
      }
    },
    "status": "Done",
    "id": "abba189f-bbdb-4851-857c-549b884e3641",
    "dependencies": [],
    "jobData": {
     "Worker": "ASR-LV",
     "Text": "http://selma.ailab.lv:2020/files/4963f238-9b83-4b37-9553-dc8ae608d719"
    },
    "jobType": "EasyPython",
    "jobProvider": "Selma"
   },
   {
    "jobResult": {
     "Timestamp": "2021-05-11T12:23:00.3818322+00:00",
     "Result": {
      "source_lang": "lv",
      "target_lang": "de",
      "text": [
       {
        "alignment": [
         {
          "text": "No darba uz mājām mēs braucām vienā un tai pašā laikā visu cauru gadu.",
          "translation": "Wir fuhren das ganze Jahr über zur gleichen Zeit von der Arbeit
nach Hause."
         }
        ],
        "text": "No darba uz mājām mēs braucām vienā un tai pašā laikā visu cauru gadu. ",
```

```
          "translation": "Wir fuhren das ganze Jahr über zur gleichen Zeit von der Arbeit nach Hause. "
         }
        ]
       }
     },
     "status": "Done",
     "id": "abba489f-bbdb-4851-857c-549b884e3641",
     "dependencies": [
      "abba289f-bbdb-4851-857c-549b884e3641"
     ],
     "jobData": {
      "Worker": "EasyNMT",
      "source_lang": "lv",
      "target_lang": "de"
     },
     "jobType": "EasyPython",
     "jobProvider": "Selma"
    },
    {
     "jobResult": {
      "Timestamp": "2021-05-11T12:22:28.0084822+00:00",
      "Result": {
       "text": "No darba uz mājām mēs braucām vienā un tai pašā laikā visu cauru gadu. "
      }
     },
     "status": "Done",
     "id": "abba289f-bbdb-4851-857c-549b884e3641",
     "dependencies": [
      "abba189f-bbdb-4851-857c-549b884e3641"
     ],
     "jobData": {
      "Worker": "ASR-Punctuation"
     },
     "jobType": "EasyPython",
     "jobProvider": "Selma"
    },
    {
     "jobResult": {
      "Timestamp": "2021-05-11T12:23:15.8831525+00:00",
      "Result": {
       "source_lang": "lv",
       "target_lang": "pt",
       "text": [
```

```
        {
          "alignment": [
            {
              "text": "No darba uz mājām mēs braucām vienā un tai pašā laikā visu cauru gadu.",
              "translation": "Do trabalho para a casa, fomos ao mesmo tempo durante todo o
ano."
            }
          ],
          "text": "No darba uz mājām mēs braucām vienā un tai pašā laikā visu cauru gadu. ",
          "translation": "Do trabalho para a casa, fomos ao mesmo tempo durante todo o ano.
"
        }
       ]
      }
    },
    "status": "Done",
    "id": "abba589f-bbdb-4851-857c-549b884e3641",
    "dependencies": [
     "abba289f-bbdb-4851-857c-549b884e3641"
    ],
    "jobData": {
     "Worker": "EasyNMT",
     "source_lang": "lv",
     "target_lang": "pt"
    },
    "jobType": "EasyPython",
    "jobProvider": "Selma"
   },
   {
    "jobResult": {
     "Timestamp": "2021-05-11T12:23:36.4086019+00:00",
     "Result": {
      "source_lang": "lv",
      "target_lang": "fr",
      "text": [
       {
        "alignment": [
         {
          "text": "No darba uz mājām mēs braucām vienā un tai pašā laikā visu cauru gadu.",
          "translation": "De la maison au travail, nous sommes allés à la même heure tout
au long de l'année."
         }
        ],
        "text": "No darba uz mājām mēs braucām vienā un tai pašā laikā visu cauru gadu. ",
```

```
          "translation": "De la maison au travail, nous sommes allés à la même heure tout au
long de l'année. "
         }
        ]
       }
     },
     "status": "Done",
     "id": "abba689f-bbdb-4851-857c-549b884e3641",
     "dependencies": [
      "abba289f-bbdb-4851-857c-549b884e3641"
     ],
     "jobData": {
      "Worker": "EasyNMT",
      "source_lang": "lv",
      "target_lang": "fr"
     },
     "jobType": "EasyPython",
     "jobProvider": "Selma"
    },
    {
     "jobResult": {
      "Timestamp": "2021-05-11T12:22:43.5592538+00:00",
      "Result": {
       "source_lang": "lv",
       "target_lang": "en",
       "text": [
        {
         "alignment": [
          {
           "text": "No darba uz mājām mēs braucām vienā un tai pašā laikā visu cauru gadu.",
           "translation": "From work to home we were driving at the same time all year
round."
          }
         ],
         "text": "No darba uz mājām mēs braucām vienā un tai pašā laikā visu cauru gadu. ",
         "translation": "From work to home we were driving at the same time all year round.
"
        }
       ]
      }
     },
     "status": "Done",
     "id": "abba389f-bbdb-4851-857c-549b884e3641",
     "dependencies": [
```

```
      "abba289f-bbdb-4851-857c-549b884e3641"
     ],
     "jobData": {
      "Worker": "EasyNMT",
      "source_lang": "lv",
      "target_lang": "en"
     },
     "jobType": "EasyPython",
     "jobProvider": "Selma"
    }
  ],
  "status": "Done"
}
```