TROMPA

TROMPA: Towards Richer Online Music Public-domain Archives

Deliverable 5.3

TROMPA Processing Library v2

Grant Agreement nr	770376
Project runtime	May 2018 - April 2021
Document Reference	TR-D5.3-TROMPA Processing Library v2
Work Package	WP5 - TROMPA Contributor Environment
Deliverable Type	Report
Dissemination Level	PU- Public
Document due date	28 February 2021
Date of submission	28 February 2021
Leader	UPF
Contact Person	Aggelos Gkiokas (aggelos.gkiokas@upf.edu)
Authors	Aggelos Gkiokas, Alastair Porter, Helena Cuesta, Juan Sebastian Gomez, Lorenzo Porcaro (UPF), David Weigl (mdw)
Reviewers	Werner Goebl (mdw), Nicolas Gutierrez (UPF)

Executive Summary

This deliverable describes the work carried out under Task 5.3 - Multimodal Integration and is an extension of the 1st version of deliverable D5.3 - TROMPA Processing Library (TPL). As a general description, this deliverable provides functionalities for descriptions and syntheses of music data coming from supported music data repositories. This comprises common access to (combinations of) public music data contained in the Contributor Environment (CE) database regardless of content-type and common access to WP3 algorithm processes to be run against (combinations of) this public music data. The components delivered in this document can be summarized as follow:

- An extension of CE functionalities, offering a generic integration solution for WP3-CE-Application interaction. This is also strongly related with **Task 5.1 - Data Infrastructure** and the corresponding deliverable.
- The Multimodal Component, which offers a graphical interface to search for items in the CE without the need to construct GraphQL queries manually.
- A helper library for accessing and updating data in the CE, named ce-client.
- The TROMPA Processing Component (TPC), which is a software for wrapping, scheduling and executing algorithms, as well handling all the input/output communication of an algorithm with the CE. We can say that executed algorithms are completely "agnostic" of the CE environment. In this sense, any algorithm can be integrated within the TROMPA Processing Component easily and can also be run in any machine; thus no explicit software/hardware requirements are imposed.
- A number of algorithms integrated within the TROMPA Processing Component.

The TPL offers a communication layer between client applications, the CE, and WP3 technologies. Each time a client application requests any type of data or metadata, this is done via TPL functionalities which derive from:

- **The CE infrastructure** including the **CE neo4j database** along with the **GraphQL Interface,** which provides Pilot developers rich access to the TROMPA dataset, stored as public data references in the CE database, as well as the **Extended API Functionalities,** which allows for the real-time definition, creation, completion and consumption of WP3 technology 'jobs' on public music data referenced in the CE database.
- **The Multimodal Component,** which allows a Pilot user to browse (and combine) CE references to public music data, regardless of content type. Moreover the Multimodal Component provides a graphical interface to browse the CE data.
- **A set of algorithms** that run on specific data types.
- Various instances of the **TROMPA Processing Component** that are run on different compute nodes and invoke different algorithms.
- **A client application** that makes algorithm invoking requests, e.g. ask to run a specific algorithm A on an item B. In the TROMPA context this client application is any of the five software prototypes developed to cater for TROMPA's use cases (see deliverables of WP6).

The processing workflow for triggering a WP3 algorithm/task is as follows:

- Initially, an algorithm definition is given to the TPC, which converts this definition into a CE representation, by creating the corresponding CE nodes. This definition contains information about the algorithm such as the number and type of inputs and parameters, the command line invocation, etc.

- Additionally, the TPC client program creates a template query[1] for triggering this algorithm, that can be used by the client application.
- If a given task has already processed the CE node corresponding to a particular piece of data, the client application can retrieve and show the result immediately. Otherwise the application can request the running of the task on a selected piece of data by submitting the query from the previous step. This process creates a new node in the CE that corresponds to this specific task.
- The TPC either polls the CE for new 'jobs' created, or subscribes to 'job' creation triggers via a WebSocket.
- On reception of a new job, the TPC retrieves the necessary parameters and input files, and validates and executes the task. While running, it updates the task status in the CE.
- From the client application's perspective, in order to track the progress of the execution it can either poll the CE regularly, or subscribe to update triggers via WebSocket.
- Once the TPC task is completed, the output is transferred to TPC storage (See section 3.2.5), a reference node for the storage location is created in the CE, a *result* relation is added between the task node and reference node, and the task status is set to 'complete'.
- The client application receives the 'complete' update (by polling or subscription), retrieves the result reference and can create additional relations between the result and other CE nodes for enriching, semantic interlinking and provenance tracking purposes.

The proposed workflow provides the flexibility of running algorithms 'on demand', i.e. running a specific algorithm on a specific data item when requested by the use cases, as well as triggering specific algorithms when new nodes of certain types are added in the CE (as will be described in section 3.3.3).

The CE database contains references to public online musical data in many different formats. The Multimodal Component (MMC) assists developers to retrieve that assorted data and present it to users in a comprehensive way. To this end, the Multimodal Component offers 'shortcut' access to CE functionalities and example UI components that give users access to these functionalities and enable them to consume the rich TROMPA data set. Currently, the Multimodal Component consists of a React component that implements search functionality on CE database content through the GraphQL interface. A client application developer can include this component and use (parts of) its code and UI elements where they fit use-case-specific functionalities. The React component can also serve as a working example for developers to build use-case-specific implementations of the same GraphQL functionalities or to use another technology stack.

In order to allow developers to interact with the CE more easily, we wrote the Python library trompace-client to help generate and make requests. The library contains functionality to generate GraphQL mutations to create and modify objects for the main data types that are used in TROMPA and join objects together using relations, and functionality to generate GraphQL queries to retrieve data from the CE. It also contains helper methods to perform HTTP requests to the CE. A client package provides higher-level methods to perform tasks using a single function call.

The TROMPA Processing Component is a Python open source library that is designed in such a way that can be run in multiple instances on different machines, making it scalable and robust. Each application/algorithm is represented by a configuration file that contains all the relative information about the algorithm (e.g. inputs, outputs, metadata as creator, link to source code etc) accompanied by a Docker image that contains the program hosting the algorithm along with the respective

[1] RequestControlAction mutation. See Deliverable D2.3 - Technical Requirements and Integration

software dependencies. Each TPC instance is associated with a set of algorithms. The main component of the TPL is the Scheduler. The Scheduler has three distinct functionalities:

- It creates all the necessary nodes in the CE in order to represent each algorithm
- It continuously polls the CE for new jobs.
- It launches the individual jobs in different processes and handles job prioritisation and load balancing of the TPC.

Once new tasks are requested, for each polling iteration the scheduler selects which jobs will be executed on this iteration based on their priority , updates the information on the CE that these jobs are already assigned and creates a new process for each job. Once a job is finished, the result is stored in the TPC storage, and a node that refers to this item is created in the CE.

Apart from the mechanism of an “on demand” execution of tasks as described, we provide an automatic triggering mechanism (see section 3.3.3). This mechanism associates each node type of the CE with a set of predefined algorithms. Every time a new node of a specific file type is added in the CE, then TPC creates the corresponding requests. These requests are then handled by the TPC in the manner described above.

For defining an algorithm one has to provide a configuration file. Each algorithm must be packaged as a Docker image. This has the advantage that TPC is agnostic of the software requirements of the algorithms and can be run on any computer / operating system.

There are two ways of running tasks: on demand, and automated triggering. On demand task launching, described in section 2.2.3, is where a client application requests a corresponding job for a specific node in the CE. However there is the need for automatically processing specific item types when they are added in the CE. To do so, TPC implements an automated triggering by acting as a client program of itself. It creates all the job requests as being a client program; then these jobs are handled by the TPL as all other job requests coming from the user pilots.

TPL supports multiprocessing, i.e. running more than one task at a time. Each TPC instance has a maximum number of processes that can be run simultaneously which is related to the hardware capabilities of the computer running this TPC instance. We consider three levels of task priorities, namely low, medium and high. TPC has a queuing system that ensures that all priority level tasks can be run simultaneously and to avoid the case where low priority tasks occupy all processes, thus not allowing the quick execution of high priority tasks.

Regarding data privacy, TPC is able to store data privately within user-controlled Solid Pods TPC also contains capabilities for encrypting data files and URIs that may reveal personal information.

There are two types of algorithms that are run under the context of TPC. The first is a certain collection of algorithms that are run on certain types of the data in the CE, as for example extract the tempo of an audio file or detect its emotion, convert a scanned image to MEI file, or to assess the performance of a choir singer. These algorithms serve as processes to use cases or other application contexts (e.g. other tools). The second consists of processes related more to the curation of the data in the CE, as for example data importers from external repositories. In the following table we summarize the processes of the two categories.

Algorithm / tool	Description	Use case / application context
Algorithms		
Extraction of Baseline Features	It comprises a set of baseline audio descriptor algorithms that are run on audio files.	CE enrichment
Rhythm descriptors	It contains algorithms for the rhythm description of audio files as well as a human-in-the-loop method for annotating beats	CE enrichment, Annotator tool
Active learning for emotion recognition	Active learning for music emotion recognition is to allow the classification models to improve with new annotations from particular users.	Music enthusiasts
Emotion-based Music Recommendation	The recommended tracks are provided analyzing previous users' annotations, where tracks are listenable through a player presenting additionally a short explanation of part of the features used in the MER models.	Music enthusiasts
Singing Performance Assessment	Uses information from the piece's score and pitch descriptors from the actual performance to calculate the average deviation of the singers' pitch from the reference	Choirs use case
Performance to score alignment	It is used to align MIDI piano performances recorded by users of D6.5 (Working prototype for instrumental players) with MEI score encodings.	Piano players
Image to MEI conversion	Detects music measure from a score image as well as other symbols (systems, pauses etc) and creates a MEI file encoding that information.	Orchestras use case
Data Importers		
IMSL Importer	We import metadata for composers,works, score images, and score encodings from imslp.org.	Specialised importer, all use cases
CPDL Importer	We import metadata for composers, works, and score encodings from cpdl.org.	Specialised importer, singers use case
Muziekweb Importer	For each track in the Music Enthusiasts use case, we import an AudioObject which is linked to a	Specialised importer, music enthusiasts use case

	MusicComposition and one or several Persons to the CE	
GitHub MEI importer	Importer for MEI score encodings generated during the TROMPA project, stored in the trompamusc-encodings GitHub organisation. This includes a number of transcriptions of Beethoven and other piano works for seeding an initial repertoire for the working prototype for instrumental players	instrumental players
Humdrum scores importer	Another source of scores encoded in the Humdrum format is used in the working prototype for music scholars	music scholars
Data importer	This importer allows anyone in the consortium to import data into the CE from one of the supported sources. The algorithm takes as input a data source (e.g. MusicBrainz, Wikidata, IMSLP), and a specific identifier for an item in that data source.	General data importer, all use cases
IMSLP MusicXML file	Identifies the MusicXML file inside a zip archive of scores available on IMSLP.	Data importer fix
MusicXML/Humdrum to MEI converter	Converts MusicXML and Humdrum scores that have been imported to the CE to MEI using Verovio.	All use cases

Version Log		
#	Date	Description
v0.1	10 February 2021	Initial version submitted for internal review
v0.2	23 February 2021	Revised version after internal review
v0.3	27 February 2021	Minor changes, formatting etc
v1.0	28 February 2021	Final version submitted to EU

Table of Contents

1. Introduction

1.1 Scope

This deliverable describes the work carried out under Task 5.3 - Multimodal Integration and is an extension of the 1st version of deliverable D5.3 - TROMPA Processing Library[2] (TPL). This deliverable provides functionalities for descriptions and syntheses of music data coming from supported music data repositories. This comprises common access to (combinations of) public music data contained in the Contributor Environment (CE) database regardless of content-type and common access to WP3 algorithm processes to be run against (combinations of) this public music data. The components delivered in this document can be summarized as follow:

- An extension of CE functionalities, offering a generic integration solution for WP3-CE-Application interaction. This is also strongly related with Task 5.1 - Data Infrastructure[3] and the corresponding deliverable.
- The Multimodal Component, which offers a graphical interface to search for items in the CE without the need to construct GraphQL queries manually.
- A helper library for accessing and updating data in the CE, named trompace-client.
- The TROMPA Processing Component (TPC), which is a software for wrapping, scheduling and executing algorithms, as well handling all the input/output communication of an algorithm with the CE. We can say that executed algorithms are completely "agnostic" of the CE environment. In this sense, any algorithm can be integrated within the TROMPA Processing Component easily and can also be run in any machine; thus no explicit software/hardware requirements are imposed.
- A number of algorithms integrated within the TROMPA Processing Component.

1.2 Relation to the 1st version and document structure

This deliverable is the second and final version, it is chosen to be self-contained. Thus, it is an extension of the first version and contains overlapping content with it. The difference between the two versions indicate the evolution of the tool from M12 (1st version) and M34 (2nd version) in order to meet the requirements imposed by the client applications as well as the data modelling in the CE. Section 2 overlaps significantly with the 1st version of the deliverable and describes the extended CE functionalities for the support of algorithm invoking, the Multimodal Component as well as the new section (2.4) on the CE client (trompace-client) component. Section 3 is the main contribution of this deliverable, which describes the TROMPA Processing Component in detail: all the mechanisms needed to define, configure, execute algorithms, and handle all the communication with the CE. Section 4 presents the deployment aspects of the TPC; which technologies are integrated within the TPC and which use case's client applications use which of these technologies, and hardware considerations.

[2] https://trompamusic.eu/deliverables/TR-D5.3-TROMPA_Processing_Library_v1.pdf
[3] https://trompamusic.eu/deliverables/TR-D5.1-Data_Infrastructure_v2.pdf

2. Infrastructure and Workflow

2.1 Overview

The TROMPA Processing Library offers a communication layer between client applications, the CE, and WP3 technologies. An overview of the TPL workflow is shown in Figure 2.1. Each time a client application requests any type of data or metadata, this is done via TPL functionalities which derive from:

- **The TROMPA data infrastructure** (detailed in Deliverable 5.1) including the **CE neo4j database** along with the **GraphQL Interface,** which provides Pilot developers rich access to the TROMPA dataset, stored as public data references in the CE database, as well as the **Extended API Functionalities,** which allows for the real-time definition, creation, completion and consumption of WP3 technology 'jobs' on public music data referenced in the CE database.
- A decentralised contribution layer in the form of Solid Pods, intended for the storage of private, user-generated content.
- **The Multimodal Component,** which allows TROMPA users to browse (and combine) CE references to public music data, regardless of content type. The Multimodal Component also provides a graphical interface to browse the CE data.
- **A set of algorithms** that run on specific data types that are facilitated by the TROMPA use cases. Apart from this set of algorithms, more algorithms can be added at any time.
- Various instances of the **TROMPA Processing Component** that are run on different compute nodes and invoke different algorithms.
- **A client application** that makes algorithm invoking requests, e.g. ask to run a specific algorithm A on an item B. In the TROMPA context this client application is any of the **five software prototypes** developed to cater for TROMPA's use cases (see deliverables of WP6).

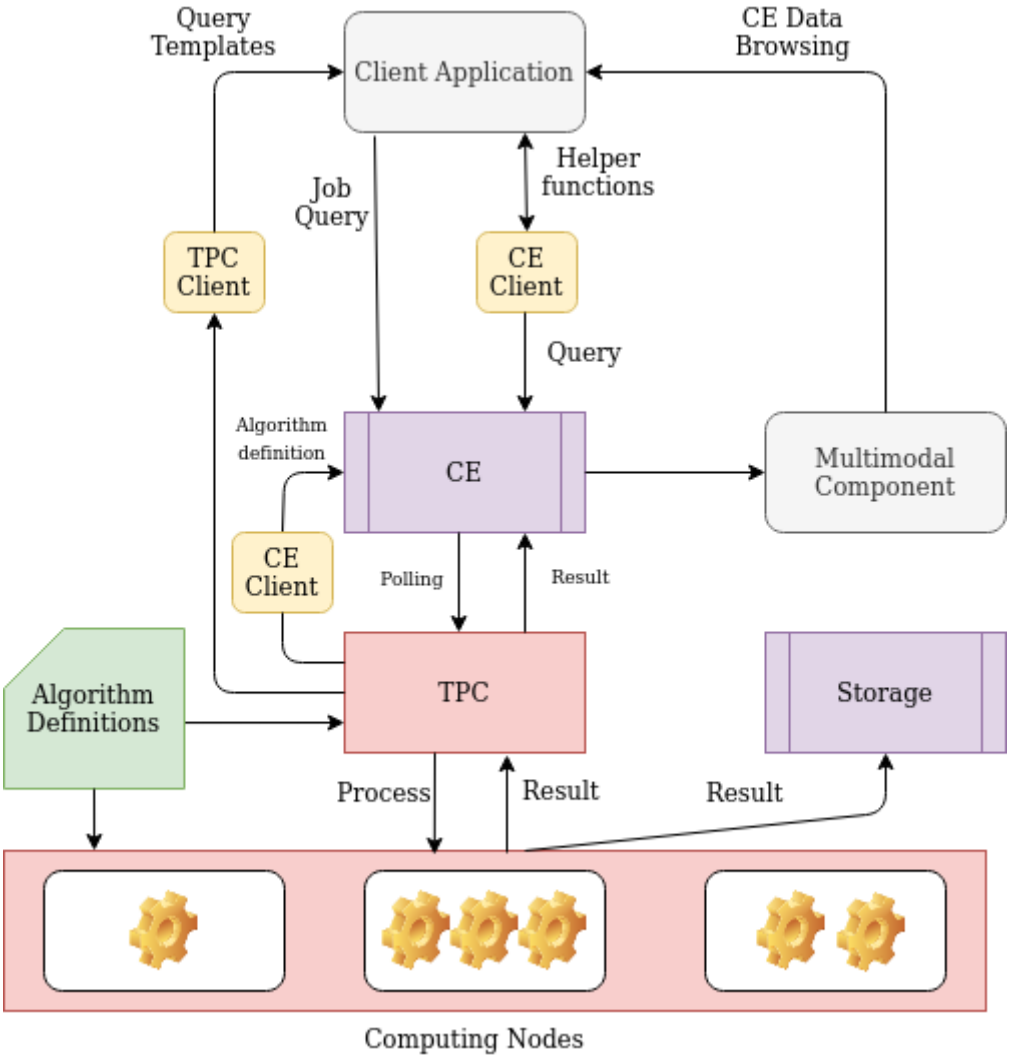

Figure 2.1 TROMPA Processing Library Workflow

The processing workflow for triggering a WP3 algorithm/task is as follows:

- Initially, an algorithm definition is given to the TPC, which converts this definition into a CE representation, by creating the corresponding CE nodes. This definition contains information about the algorithm such as the number and type of inputs and parameters, the command line invocation, etc.
- Additionally, the TPC client program creates a template query[4] for triggering this algorithm, that can be used by the client application.
- If a given task has already processed the CE node corresponding to a particular piece of data, the client application can retrieve and show the result immediately. Otherwise the application can request the running of the task on a selected piece of data by submitting the query from the previous step. This process creates a new node in the CE that corresponds to this specific task.
- The TPC either polls the CE for new 'jobs' created, or subscribes to 'job' creation triggers via a WebSocket.

[4] RequestControlAction mutation. See Deliverable D2.3 - Technical Requirements and Integration

- On reception of a new job, the TPC retrieves the necessary parameters and input files, and validates and executes the task. While running, it updates the task status in the CE.
- From the client application's perspective, in order to track the progress of the execution it can either poll the CE regularly, or subscribe to update triggers via WebSocket.
- Once the TPC task is completed, the output is transferred to TPC storage (See section 3.2.5), a reference node for the storage location is created in the CE, a *result* relation is added between the task node and reference node, and the task status is set to 'complete'.
- The client application receives the 'complete' update (by polling or subscription), retrieves the result reference and can create additional relations between the result and other CE nodes for enriching, semantic interlinking and provenance tracking purposes.

The proposed workflow provides the flexibility of running algorithms 'on demand', i.e. running a specific algorithm on a specific data item when requested by a client application, as well as triggering specific algorithms when new nodes of certain types are added in the CE (as will be described in section 3.3.3).

2.2 Algorithm definition mechanism

This section provides details on the mechanism by which TPC can communicate with the client applications through the Contributor Environment. At its most basic, this integration mechanism offers automation for the following process:

- A user of a client application chooses target content, referenced in the CE database
- The application creates a job to run a process on this content
- TPC Process picks up job
- TPC Process executes job on target content, creating and storing a result
- TPC Process writes reference to result in CE database
- User Pilot picks up result
- User Pilot user consumes result

In the following subsections we will describe the technical details of this mechanism.

2.2.1 Data Model for WP3 Processes

Process jobs are maintained as nodes in the CE. A subscription[5] or a polling mechanism enables the TPC to be actively updated on job creation and status updates in real time. This way, the CE becomes the communication layer between the TPC and the client applications. This offers a standardized solution for integration and ensures that WP3-produced data through the TPC is referenced and gets interlinked with the larger TROMPA dataset. The generic TPC-use case interaction solution is based on a schema.org[6] compatible data model that can be broken down into three parts:

- **Template Nodes:** Maintained by TPC and used by client applications. The Template Nodes represent generic algorithmic processes, e.g. "the extraction of Pitch Class Profiles (PCP) features from an audio recording".
- **Instance Nodes**: Created by client applications and maintained by the CE. Instance Nodes correspond to specific tasks requested by User Pilots, e.g. "the extraction of PCP features from the audio recording X".

[5] https://graphql.org/blog/subscriptions-in-graphql-and-relay/
[6] https://schema.org/

- **Public Nodes**: - They represent the (public) content on which the TPC is run (e.g. the audio recording X) and the corresponding results (e.g. the PCP features).

The job-running infrastructure uses the following CE node types:

- **SoftwareApplication:** Every software tool is represented by a **SoftwareApplication**[7] node in the database.
- **EntryPoint**: Each of the available algorithmic processes run by a software application is represented as an **EntryPoint**[8] node. This entry point enables the user pilot to request, monitor and control the running of a process. The **EntryPoint** needs to be related to the **SoftwareApplication** through the *actionApplication*[9] property.
- **ControlAction:** The **ControlAction**[10] is the `template` for a user's request for a certain process to be run and is related to the **EntryPoint** through the *potentialAction* property. It is like a template for a potential job that needs to be carried out by the process represented by the **EntryPoint**.

In order to pass the required parameters and input to an algorithm, TPC uses the following CE node types:

- **Property**: any content available in the CE, like an audio file the user needs the process to act on, can be specified by adding a **Property**[11] and relating this to the **ControlAction** through the *object*[12] property.
- **PropertyValueSpecification:** These nodes are used to represent the parameters of an algorithm. Thus any number of required or non-required scalar arguments (numbers, strings etc.) can be set up by adding **PropertyValueSpecification**[13] nodes and relating them to the **ControlAction** through the same *object* property.

Together, these **EntryPoint**, **ControlAction**, **PropertyValueSpecification** and **Property** nodes determine what the client application will interact with when requesting and controlling a process. This model provides enough information to dynamically generate a process-specific user interface. A user requesting a job through this interface will instantiate the model as a job request which can then be picked up, followed and controlled by the user and by the TPC.

2.2.2 Algorithm Registration

Following the data model presented in the previous section, each algorithm must register itself to the CE. The procedure to do so is summarized as:

- Create a **SoftwareApplication** node in the CE: This node corresponds to a software package that hosts a specific algorithm. A SoftwareApplication can have many algorithms related to it.
- Create an **EntryPoint** node in the CE: The entry point corresponds to a specific algorithm/method to be run.

[7] https://schema.org/SoftwareApplication
[8] https://schema.org/EntryPoint
[9] https://schema.org/actionApplication
[10] https://schema.org/ControlAction
[11] https://meta.schema.org/Property
[12] https://schema.org/object
[13] https://schema.org/PropertyValueSpecification

- Create an *actionApplication* relation between **SoftwareApplication** and **EntryPoint**: This actions defines that this EntryPoint is a part of the **SoftwareApplication** node**.**
- Create a **ControlAction** template node: This **ControlAction** node will be the model for the 'job' created for a specific algorithm process request. Each request will result in a copy of this **ControlAction** node to be created (instantiated) which will then represent the 'job instance' that can be acted on and followed.
- Create *potentialAction* relation between **EntryPoint** and (template) **ControlAction:** Relates the **ControlAction** template to the specific **EntryPoint**.
- Create **Property** template node: Corresponds to existing nodes in the CE database, e.g., to reference a content file stored in a public repository. These will be the inputs to the WP3 algorithms.
- Create **PropertyValueSpecification**: Corresponds to a scalar parameter (string, number, boolean flag) that needs to be given by the user to configure the algorithm process.
- Create *object* relations between **ControlAction** and **Property**/**PropertyValueSpecification** template nodes, respectively.

2.2.3 Algorithm Invoking Mechanism

Following the data model presented previously, each algorithm is invoked by the client applications using the following pipeline:

- The client application requests a new job by using the `trompace-client` or TPC-client to execute the **RequestControlAction** graphql mutation. With this mutation a new **ControlAction** node is created in the CE. This mutation provides all the necessary information about the task to be run: the CE translates this request by creating a set of nodes on the basis of the **EntryPoint/ControlAction** template and subsequently responds with the thus created **ControlAction**, including its unique *identifier.* It also defines the values for the **Property** and **PropertyValueSpecification** which are the input and parameters of the task to be run.The client application can then subscribe to the CE using the unique identifier and receive a notification each time the corresponding **ControlAction** is updated.
- The new **ControlAction** node (which is an instance node as defined in Section 2.2.1) is linked with a *derivedFrom* relation to the template node.
- The TPC can frequently poll the CE by getting all nodes with a *derivedFrom* relation to the template **ControlAction** node or by a subscription mechanism that opens a WebSocket to the CE that receives a message every time a new instance of the template node is created.
    - Subscribe to the CE on RequestControlAction requests on a specific EntryPoint, through the WebSocket: This functionality offers the possibility to handle user requests for algorithms in real-time.
    - Frequently check for tasks in the CE: It is possible to query for **ControlAction**s created on the basis of a certain **EntryPoint**. In this way, the software developed under WP3 can check if new tasks have to be run at convenient times or intervals. This functionality offers WP3 software to handle user requests for algorithms in batches.
- Once the TPC receives a new node (job), it reads all the inputs and parameters from the **Property** and **PropertyValueSpecification** fields of the **ControlAction** instance and executes the algorithm.

2.3 The Multimodal Component

2.3.1 Overview

The CE database contains references to public online musical data in many different formats. The Multimodal Component (MMC) assists developers to retrieve that assorted data and present it to users in a comprehensive way.

To this end, the Multimodal Component[14] offers 'shortcut' access to CE GraphQL functionalities and example UI components that give users access to these functionalities and enable them to consume the rich TROMPA data set.

2.3.2 Communication with the Use-Case Client Applications

The Multimodal Component consists of a React component that implements search functionality on CE database content through the GraphQL interface. An application developer can include this React component and use (parts of) its code and UI elements where they fit use-case-specific functionalities. The React component can also serve as a working example for developers to build use-case-specific implementations on the same GraphQL functionalities and as a reference for future implementations in different programming languages.

2.3.3 A Graphical Interface for Accessing Data

The Multimodal Component offers a GUI to search the TROMPA dataset as contained in the CE database, implementing the mockups as researched and produced in **Task 5.3 - Multimodal Integration of Music Data**. Figure 2.1 presents the output of the query of 'Gustav Mahler' to the CE database. On the left side we can see the different categories of the items matching the query. For each result of the query additional information is provided, as for example the source of the item (e.g. wikidata, musicbrainz), and links to related items. For instance a composer is linked to her/his compositions, and the compositions are linked to scores and performances. The end user can get deeper into the structure of the data, for example by filtering only by Person entity type, as shown in Figure 2.2. Figure 2.3 shows a screenshot of a demo that is available online[15]. The demo application will be updated in the run-up to the final evaluations in M36 to reflect the latest developments of the Multimodal Component search functionalities. The GitHub repository of the MMC demo application provides code examples for the most common user stories plus inlined documentation on query construction.

[14] https://github.com/trompamusic/trompa-multimodal-component
[15] https://multimodal.trompamusic.eu/

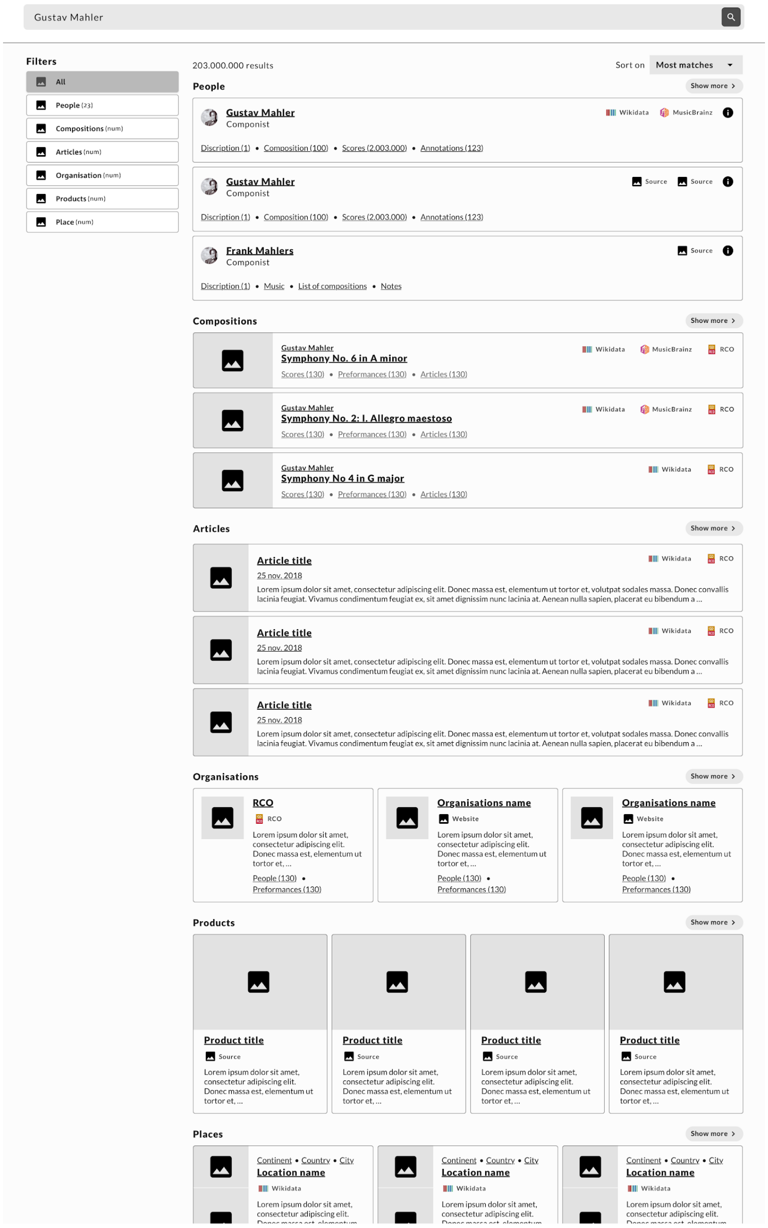

Figure 2.1 MMC Search mockup - initial search concept

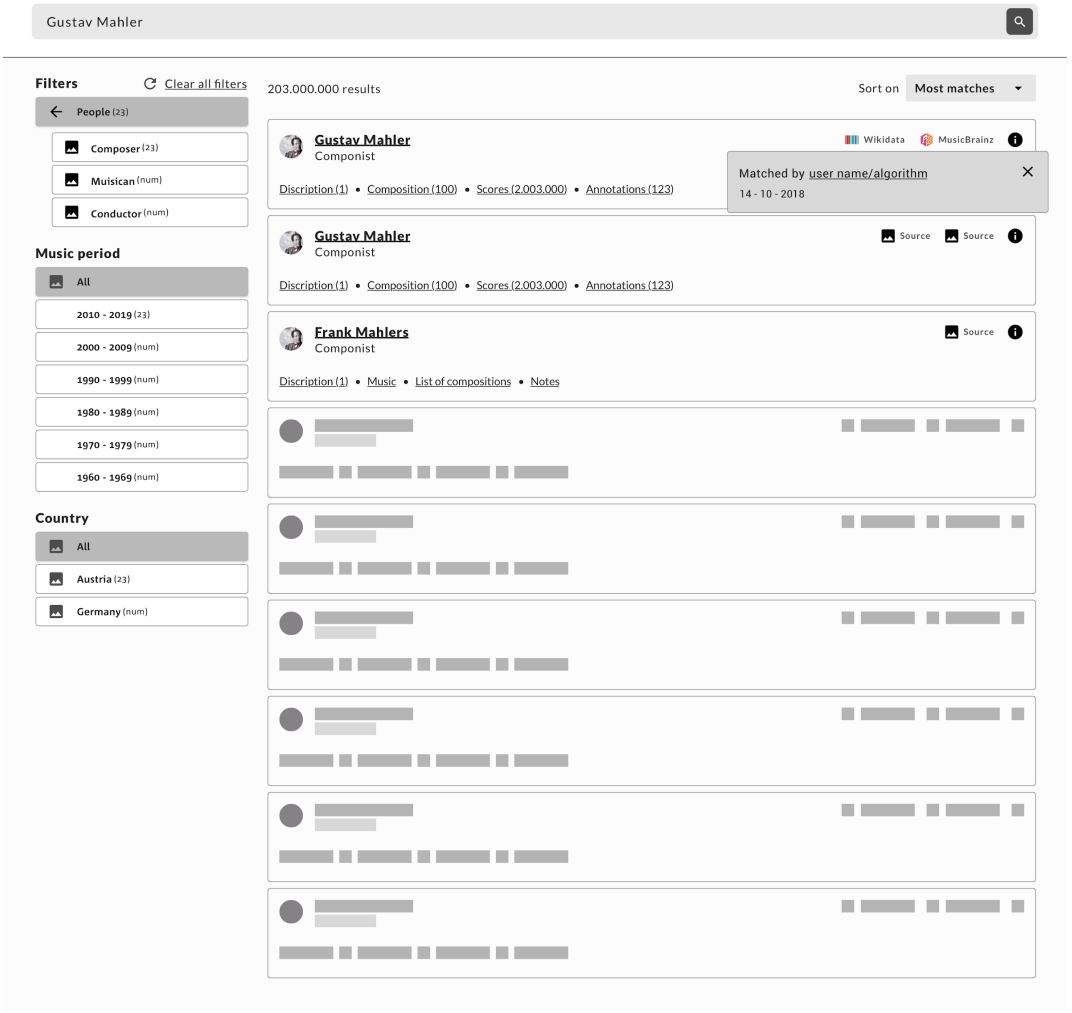

Figure 2.2 MMC Search mockup - refined search on People

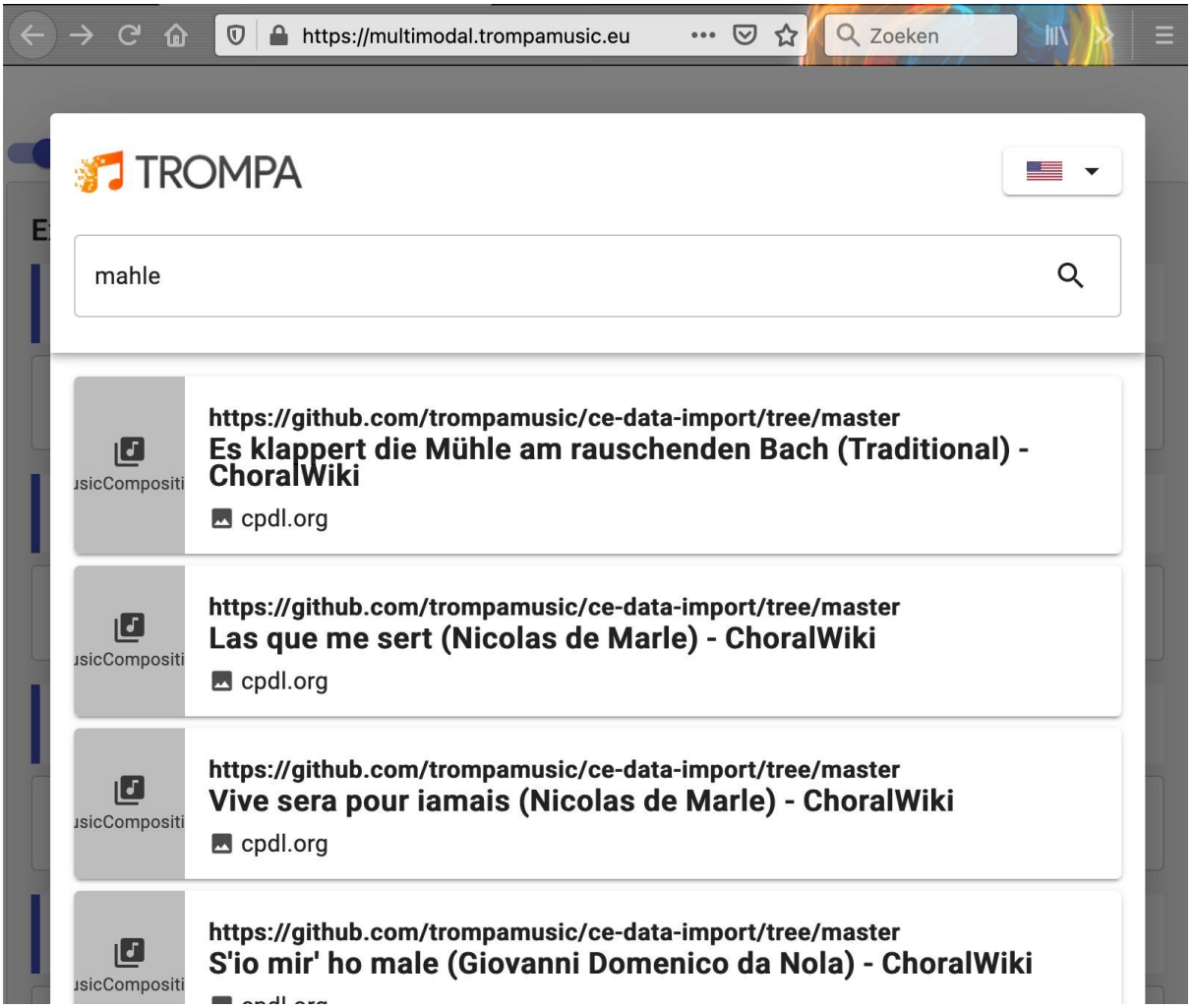

Figure 2.3 MMC Search implementation - unrefined search on music compositions

2.4 The Contributor Environment Client

In order to allow developers to interact with the CE more easily, we implemented a Python library to help generate and make requests[16]. The *trompace-client* is available on the Python Package Index (pypi)[17], making it installable by anyone using the standard Python dependency management tools. The library is licensed under the Apache 2.0 software license, allowing it to be widely used in both Open Source and commercial software packages.

The library contains functionality to generate GraphQL mutations to create and modify objects for the main data types that are used in TROMPA and join objects together using relations (the `trompace.mutations` package), and functionality to generate GraphQL queries to retrieve data from the CE (the `trompace.queries` package). It also contains helper methods to perform HTTP requests to the CE. Authentication credentials can be set in a configuration file, and the library will automatically use these credentials where necessary in the background when making mutations, without the developer needing to manage them manually. A client package (`trompace.client`) provides higher-level methods to perform tasks using a single function call. For example, when creating a list of items, it is first necessary to create an ItemList object, then a ListItem for each element. Each ListItem must be linked to the ItemList with another mutation, and optionally linked to an existing object in the CE with yet another mutation. The client interface provides a single Python method which generates all necessary mutations and performs the necessary HTTP requests to the CE.

Documentation for the trompace-client package is published on readthedocs[18], an industry-standard resource for publishing documentation for open source projects. The documentation includes standard Python documentation for the available methods, and also includes a description of each type of object (entity) that can be created in the CE, along with a list of the properties of each object and a short description of the expected values associated with each property. A graphical representation of entities and their semantic relationships is included, generated using standard semantic web tools.

3.The TROMPA Processing Component

3.1. Overview

The TROMPA Processing Component is a Python open source library[19]. It is designed to be run in multiple instances on different machines, making it scalable and robust. Figure 3.1 presents the overview of a specific instance of a TPC, i.e. a TPC that runs on a single machine. Each application/algorithm is represented by a configuration file that contains all the relative information about the algorithm (i.e. inputs, outputs, metadata as creator, link to source code, etc.) accompanied by a Docker image that contains the program hosting the algorithm along with the respective software dependencies. Each TPC instance is associated with a set of algorithms. Note that an

[16] https://github.com/trompamusic/trompace-client
[17] https://pypi.org/project/trompace-client
[18] https://trompace-client.readthedocs.io
[19] https://github.com/trompamusic/tpc

algorithm can be run on multiple TPC instances. The main component of the TPC is the **Scheduler**. The Scheduler has three distinct functionalities:

- It creates the CE nodes necessary to represent each algorithm by following the procedure described in Section 2.2
- It continuously polls the CE for new jobs.
- It launches the individual jobs in different processes and handles job prioritisation and load balancing of the TPC.

Once a new job is requested (as for example a request by a client application) using the mechanism described in 2.2.3, a new node is created in the CE that corresponds to this job. The scheduler frequently polls the CE for new tasks, given the algorithms assigned to this TPC instance. The reason for using polling instead of the subscription mechanism described in Section 2.2.3, is that we ensure that:

- We minimize the chance of a "lost" task. In the worst case where TPC is not functioning when the job request is made, this job will be executed once the TPC is up.
- All the jobs will be processed irrespective of available computational resources.
- Using the CE in order to store which jobs are assigned and which are still pending makes it easier to distribute the tasks to more than one computer.

Using polling instead of subscription, makes it a lot easier to distribute tasks in more than one computer. Consider the following in the case of using the subscription mechanism on two different computers for the same task. Upon a new task request, the two TPC instances will receive the same message almost at the same time. Handling this would require a mechanism to make the two computers communicate, otherwise they would both run the same task. Polling partially solves this, since it's less probable that two computers receive the same node at the same time, since the control action status is instantly updated to "active", and thus this job won't be received by another computer. The fact that we can run more than one TPC instances for the same task, has two major advantages:

- The fact that any job can be run on any TPC instance, makes it linearly scalable.
- Moreover it allows to have potentially "specialized" TPC instances, with respect to hardware, e.g. have a computer with GPUs for specific tasks.

When new tasks are requested, for each polling iteration the scheduler selects which jobs will be executed on this iteration based on their priority (see section 3.3.4), updates the information on the CE that these jobs are now assigned, and creates a new process for each job. Once a job is finished, the result is stored in the TPC storage, and a node that refers to this item is created in the CE.

Apart from the mechanism of an "on-demand" execution of tasks as described, we provide an automatic triggering mechanism (see section 3.3.3). This mechanism associates each node type of the CE with a set of predefined algorithms. Every time a new node of a specific file type is added in the CE, then TPC creates requests to trigger the corresponding algorithms. These requests are then handled by the TPC as described above.

Each TPC instance is set up using a configuration file[20]. This configuration file contains information about the algorithms to be run, connection information to the CE and to the TPC Storage (see section 3.2.5), information about local paths to be used, etc.

[20] Example available at https://github.com/trompamusic/tpc/blob/dev/config/tpc.ini

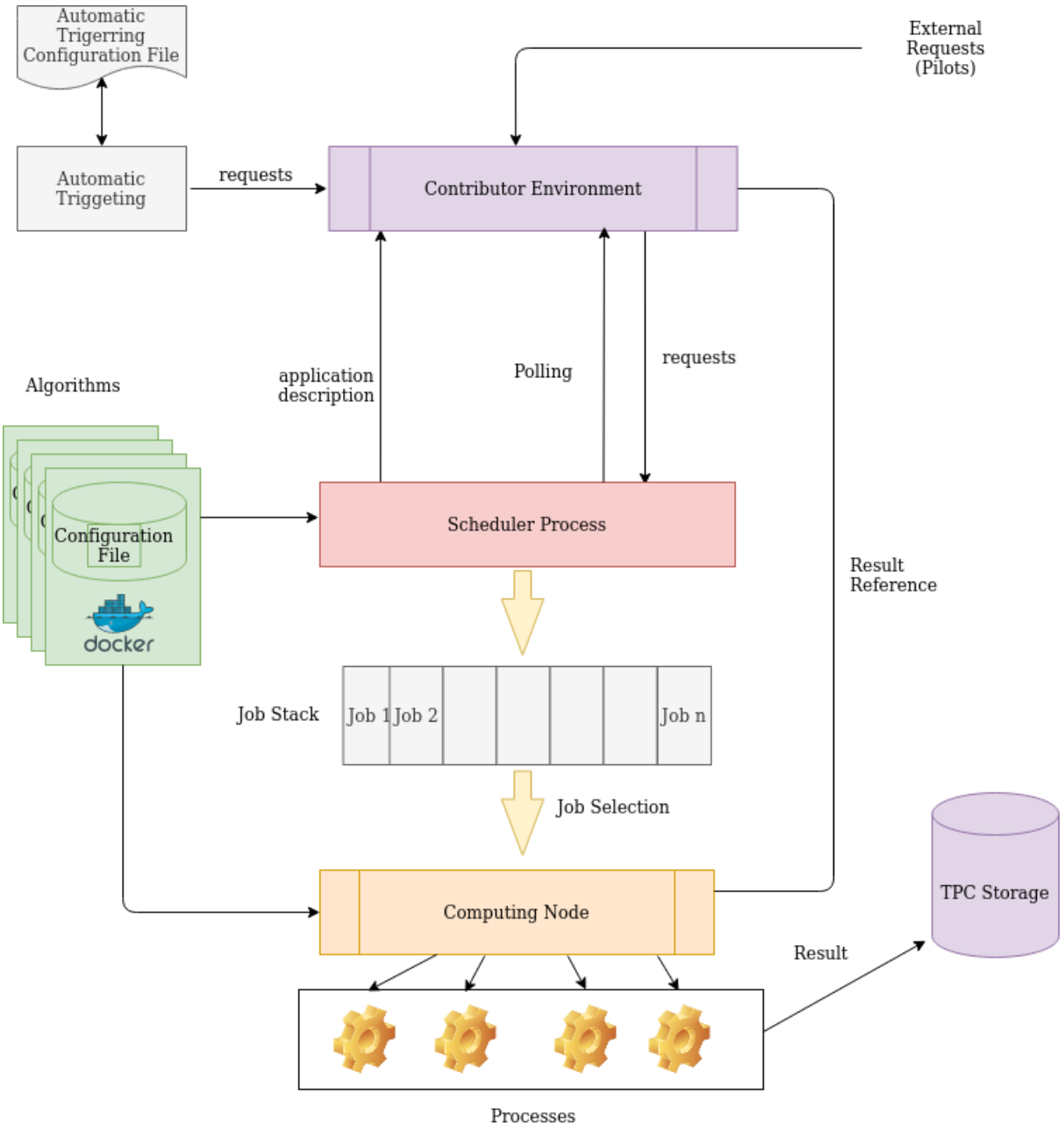

Figure 3.1. The TROMPA Processing Component organization

3.2 Representing an Algorithm

3.2.1 Configuration file

For defining an algorithm one has to provide a configuration file in an ini format[21] containing the following fields. This configuration file defines the nodes that need to be created in order to represent the algorithm in the CE (section 2.2)

- An **Application** section, which contains general information about the application such as its name, its creator, and a link to the source code.

[21] https://github.com/trompamusic/tpl/blob/main/config/application_example.ini

- A **ControlAction** section which apart from general description and naming information as in the application field, contains the number of inputs, parameters, and outputs of the algorithm.
- An **EntryPoint** section which apart from general description and naming information as in the application field, contains information about the Docker image containing the algorithm executable and the command line invocation required to run the algorithm.

  One section for each **input**, **parameter** and **output** of the software to be run (see section 3.3.3). This field provides a general description of the parameter (i.e. name, description) as well as how this parameter is defined in the corresponding command line invocation for the EntryPoint.

3.2.2 Docker image

Each algorithm must be packaged as a Docker image. This has the advantage that TPC is agnostic of the software requirements of the algorithms, which can be run in any computer / operating system.

3.2.3 Passing parameters and variables

TPL has three different types of parameters to be passed to an algorithm: inputs, parameters and outputs. Any algorithm can operate on an arbitrary number of inputs, parameters and outputs.

- **TPC Inputs:** TPC inputs represent objects (e.g. an audio file, a music score) that are to be processed by the algorithm. These inputs correspond to data that is already referenced by nodes stored in the CE. Consequently, the value of the input parameter is the identifier of the corresponding node in the CE. The CE types that correspond to TPC inputs are **DigitalDocument**, **AudioObject** or **MediaObject**. For these types of objects, an algorithm can take as input the source file of the node (e.g. the actual audio file in the case of AudioObject) as well as other fields of this node (e.g. the identifier of this node).
- **TPC Outputs:** TPC outputs correspond to the result of the specific algorithm. This result is usually a file with some numerical information, that can be stored in either binary format or in text format. All TPC outputs represent nodes to the CE, with a link to the location of the result. These nodes are created by the TPL after the execution of the algorithm. As in the case of TPC inputs, the outputs can be **DigitalDocument**, **AudioObject** or **MediaObject** nodes.
- **TPC Parameters:** TPC parameters correspond to any input to the algorithm that does not correspond to a node to the CE (either input or output). These parameters are usually some scalar values that represent properties/parameters to an algorithm, such as number of iterations to compute something, threshold values, flags etc. Apart from this usage however, the parameters offer a flexible way to add to an algorithm inputs that are not in the CE. For example, if one wants an algorithm to process a file outside the CE, the URI of the file can be passed as a parameter (see section 3.2.5).
- **TPC Storage:** In many cases it is necessary for an algorithm to refer to a data point that is not yet created. For example, the performance-to-MEI alignment workflow needs to know the URI of the performed MEI file to be created. To support this functionality, it is possible to refer to the TPC storage by a symbolic variable.

3.2.5 The TPC storage

In the TPC instance running at MTG, we provide a minio[22] server to store generated content. Minio is a data store server which provides an Amazon S3-compatible API for storing files. By default, the TPC stores generated files in this server. Algorithms can also provide S3 credentials to upload files to a storage service specific to that algorithm. Additionally, TPC has the possibility to store data in a Solid Pod in the case of private data (see section 3.4).

3.2.6 The TPC client program

Given a registered application to the TPC, we provide an additional program called the TPCclient[23]. This program is a helper software to be used by client application developers that provides template queries needed to execute a specific algorithm. Specifically, the TPC client provides the following:

- Creates the control action request given specific parameters/inputs of the algorithm
- Creates the query needed to obtain the result(s) node of a specific task.
- Optionally executes the above queries

3.3 Running Tasks

3.3.1 On demand running and automated triggering

There are two ways of running tasks, **on demand** and **automated triggering**. On demand task launching refers to the way to launch tasks as described in section 2.2.3, where a client application requests a corresponding job for a specific node in the CE.

However there is the need for automatically processing specific item types when they are added in the CE. For example we might want to extract some certain audio features each time a new audio object is inserted in the CE. Using all the CE and TPC capabilities that we have described, this is quite straightforward:

- First, we must define which algorithms should be run for each file/node type. This is set up easily using a configuration file.
- TPC subscribes to these node types; this means that for each node type TPC establishes a WebSocket connection with the CE. Each time a new node is created of that specific node type, TPC receives a message that a new node is created along with all the necessary information (e.g. identifier).
- Whenever a new message is created, TPC makes job requests as depicted in the configuration file for this node with a normal priority (see section 3.3.2.1).

In this way TPC acts as a client program of itself. It creates all the job requests as being a client program; then these jobs are handled by the TPC as all other job requests coming from the user pilots.

3.3.2 Job Queuing, multi-processing and error handling

In the initial version of the TPC, a websocket connection with the CE was opened for each subscribed application. Each time a pilot was demanding a specific action to be run by creating a control action

[22] https://min.io/

[23] TPCclient is contained in the TPC repository as a submode of the TPC.

request, the TPC instantly received the request, translated it into a job object, and placed it on the processing stack. As mentioned in Section 3.1, this approach had several drawbacks:

- In the case where the TPC was crashed or not running, all these requests would be lost.
- It would require a threading/processing mechanism able to handle arbitrary many requests at a time. Such an approach would be more unstable: in the extreme case for an arbitrary large number of concurrent requests, the TPC should have a mechanism to control them, as well as create a restoring mechanism.
- It would make the deployment on more than one computer difficult. Although this could be partially achieved by assigning specific tasks to specific machines, we might still require to have a single task running on more than a computer.

Thus we chose to use frequent polling of the CE to get all the pending jobs. This allows to keep track easier which jobs are already started to run and which are queued, since this information is explicitly contained in the CE.

Before describing the job queuing process of the TPC, we define the task priorities and the processing units assigned to each priority level:

3.3.2.1 Task Priorities

We consider three levels of task priorities, namely low, medium and high. Whenever a task is triggered the client application can select the priority for this specific job. However it can be also defined manually in the application configuration file (see section 3.2).

- **High priority tasks** are required to be run almost real-time and mainly correspond to on-demand tasks by the client applications. These tasks are the first to be run when a batch of tasks is processed.
- **Medium priority tasks**: these tasks are by default the ones created by the automated triggering described in section 3.3.1. The intuition behind this choice is that when new data is added on the CE, this is usually not done during the use of the client application, but more as a batch process (for example the importer tools, section 4.2). Thus there is no need for executing certain algorithms on this data with a high priority. However, in the case that a new object is added in the CE by an application that needs the immediate processing of this object by an algorithm, a high priority can be set to this task by the pilot.
- **Low priority tasks**: these are tasks that are related more to the update of the CE (e.g. data importer tools) as well as curation processes of the CE. Such tasks could be methods that verify links in the CE, check for duplicates, etc.

3.3.2.2 Multiprocessing and queuing

TPC supports multiprocessing, i.e. running more than one task at a time. Each TPC instance has a maximum number of processes that can be run simultaneously which is related to the hardware capabilities of the computer running this TPC instance. Each priority task level has a dedicated number of processes. This is done in order to ensure that all priority level tasks can be run simultaneously and to avoid the case where low priority tasks occupy all processes, thus not allowing the quick execution of high priority tasks. Thus, the queuing system of the TPL for one polling iteration can be summarized as follows:

- TPC polls the CE and gets all the available tasks to be run.
- It uses all high priority available processes to run high priority tasks.

- It uses all medium priority available processes to run medium priority tasks.
- It uses all low priority available processes to run low priority tasks.
- If there are remaining high priority tasks and available computing slots for medium/low priority tasks, these are used to run these high priority tasks, leaving only one computing slot empty for each of the medium/low priority tasks. For example if there are N and K available processes for medium/low priority tasks, the *N*-1 and *K*-1 will be used to run high priority tasks (for this polling iteration)
- The same procedure is repeated for the medium priority tasks: if there are available computing slots for low priority tasks, these are used to run these high medium tasks, leaving only one computing slot empty for each of the medium/low priority tasks.
- Each time a task is started to run, TPC updates its status in the CE (**ActiveActionStatus**), so that in the next poll iteration these won't be selected.
- Once a task is finished, TPC updates its status in the CE to **CompletedActionStatus.**

With this procedure we ensure a balanced use of the computational resources by the TPC. We define how many resources will be dedicated to each priority class, while at the same time we avoid that the system resources assigned to lower priority tasks are not used.

3.3.2.3 Error Handling and Recovery

In the case that an algorithm fails to complete the execution, we need a mechanism to recover from these errors. There are two categories of errors that can be captured; the errors caused by the TPC and are irrespective of the running algorithm and the errors occurred during the execution of the algorithms. For the first category, TPC handles its own errors and produces the corresponding error message. For errors that occur in algorithms, TPC captures the exit status code and standard error output of the algorithms. If an error code is reported by the algorithm, this is considered as an error by the TPΨ, and the corresponding output is used as the error message.

Once an error is produced either by the TPC or the algorithm, TPΨ handles this error by immediately updating the status of the corresponding CE node to **FailedActionStatus** and by adding to this node additional information by providing the error message along with the origin (i.e. TPC or algorithm), thus all the information about the error can be retrieved by the TPC or algorithm maintainer. Once the algorithm or TPL maintainer has fixed the error, the updated TPL instance can query the job nodes with a **FailedActionStatus** and rerun the failed jobs.

3.3.3 Using TPC out-of-the-box

TPC algorithm definition in section 3.2 imposes a certain format of the algorithms that can be run. Each algorithm must have a certain number of inputs, outputs that correspond to CE nodes, as well as the parameters. Although this is the case for the majority of the algorithms, TPC allows a more flexible way to integrate an algorithm without the explicit use of inputs and outputs. In this case, the algorithm maintainer developer can use TPC parameters in order to pass the necessary information to the algorithm. For example, the unique identifier of a CE node can be passed as a parameter. In this case, it is the responsibility of the algorithm to query the CE for this node, download the corresponding data that this node refers to, do the processing, store the data etc.

3.4 Data Privacy

3.4.1 Access of data stored in Solid Pods

In some situations, we want to be able to process data with the TPC that is not publically available. For example, this could include personal rehearsals performed by piano players in D6.5. In this case, the pilot allows users to store their rehearsals in a private Solid Pod (See Deliverable 5.1 for a brief description of the use of Solid Pods in TROMPA). In this case we need a process to deliver private files to the TPC for processing and to return the results of the algorithm back to the user's Solid Pod. Solid is based on open authentication technologies, including OpenID Connect[24]. This standard allows users to delegate permissions to external applications to have access to read from and write to private resources in a Solid pod. We implemented an authentication workflow which allows the TPC to request permission from a user to access such private resources[25]. When a user grants permission to the TPC, authentication tokens are stored in local storage. If an algorithm requires access to private resources the TPC will generate a temporary token granting the algorithm access to only this specific user's Solid Pod.

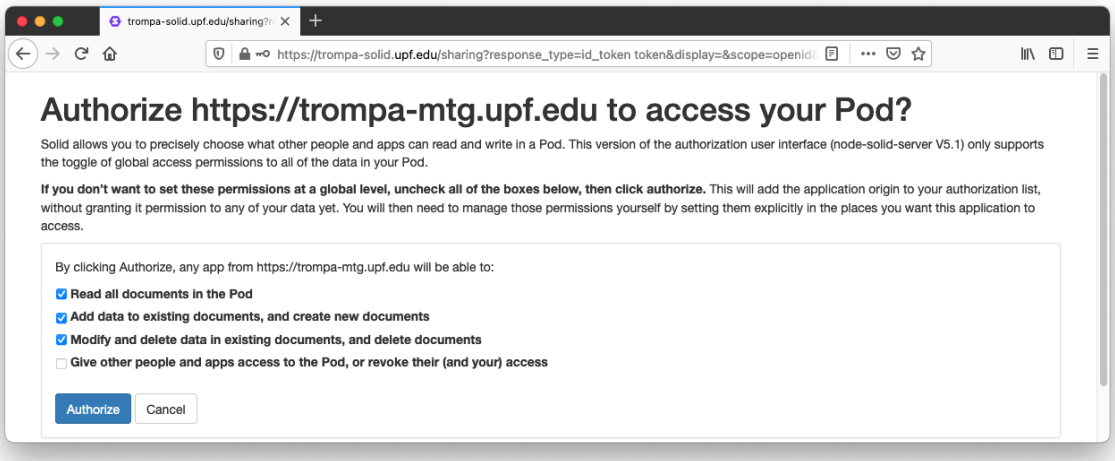

Figure 3.2 OpenID Connect workflow asking a user to grant permission to the TPC to private data in their Solid Pod

3.4.2 Encryption

We support two types of encryptions in order to handle private data: encryption of data, and encryption of URIs. Both encryption/decryption we use the symmetric (also known as "secret key") authenticated cryptography, using the Fernet python library[26].

- **Encryption of data**: We provide the capability of encryption of the input data. The main problem was that we wanted to store private data in the CE. However all nodes in the CE are visible, and in this case the URI of the private data would be visible through the CE. As a solution we incorporated the encryption of this private data, so that only the TPC and the use case could access the data of this user. This was done before the integration of Solid Pods so the functionality is not currently used. However we kept it in order to encrypt Solid Pod URIs (see below).

[24] https://openid.net/connect
[25] https://github.com/trompamusic/solid-oidc-app-permission
[26] https://cryptography.io/en/latest/fernet.html

- **Encryption of URIs:** Although the integration of Solid Pods described above satisfies the need of securely storing and processing private data, there is still some minor leak of personal information; since most of the Solid Pod URIs contain implicit information about the user (e.g., Pod URI contains the username), we used the encryption method described above in order to encrypt the Solid Pod URIs, providing a 2nd security layer of data privacy.

4. Implementation

There are two families of algorithms that are run under the context of TPC. The first is a certain collection of algorithms that are run on certain types of the data in the CE, as for example extract the tempo of an audio file or detect its emotion, convert a scanned image to MEI file, or to assess the performance of a choir singer. These algorithms serve as processes to use cases or other application contexts (e.g. other tools). The second consists of processes related more to the curation of the data in the CE, as for example data importers from external repositories. In the following table we summarize the processes of the two categories. An overview is presented in Table 4.1, along with a description and relation to the use cases.

Algorithm / tool	Description	Use case / application context
Algorithms		
Extraction of Baseline Features	Comprises a set of baseline audio descriptor algorithms that are run on audio files.	CE enrichment
Rhythm descriptors	Contains algorithms for the rhythm description of audio files as well as a human-in-the-loop method for annotating beats	Annotator tool
Active learning for emotion recognition	Active learning for music emotion recognition is to allow the classification models to improve with new annotations from particular users.	Music enthusiasts
Emotion-based Music Recommendation	The recommended tracks are provided analyzing previous users' annotations, where tracks are listenable through a player presenting additionally a short explanation of part of the features used in the MER models.	Music enthusiasts
Singing Performance Assessment	Uses information from the piece's score and pitch descriptors from the actual performance to calculate the average deviation of the singers' pitch from the reference	Choirs use case

Performance to score alignment	Used to align MIDI piano performances recorded by users of D6.5 (Working prototype for instrumental players) with MEI score encodings.	Piano players
Image to MEI conversion	Detects music measure from a score image as well as other symbols (systems, pauses etc) and creates a MEI file encoding that information.	Orchestras use case
Data Importers		
IMSL Importer	We import metadata for composers,works, score images, and score encodings from imslp.org.	Specialised importer, all use cases
CPDL Importer	We import metadata for composers, works, and score encodings from cpdl.org.	Specialised importer, singers use case
Muziekweb Importer	For each track in the Music Enthusiasts use case[27], we import an AudioObject which is linked to a MusicComposition and one or several Persons to the CE	Specialised importer, music enthusiasts use case
GitHub MEI scores importer	Importer for MEI score encodings generated during the TROMPA project, stored in the trompamusc-encodings GitHub organisation.[28] This includes a number of transcriptions of Beethoven and other piano works for seeding an initial repertoire for the working prototype for instrumental players	instrumental players, scholars
Humdrum scores importer	Another source of scores encoded in the Humdrum format is used in the working prototype for music scholars	music scholars
Data importer	This importer allows anyone in the consortium to import data into the CE from one of the supported sources. The algorithm takes as input a data source (e.g. MusicBrainz, Wikidata, IMSLP), and a specific identifier for an item in that data source.	General data importer, all use cases
IMSLP MusicXML file	Identifies the MusicXML file inside a zip archive of scores available on IMSLP.	Data importer fix

[27] https://ilde.upf.edu/trompa/
[28] https://github.com/trompamusic-encodings

MusicXML/Humdrum to MEI converter	Converts MusicXML and Humdrum scores that have been imported to the CE to MEI using Verovio.	All use cases

Table 4.1. A list of the algorithms encapsulated in the TPC along with a short description and relation to the use cases / application context.

4.1 Algorithms encapsulated and relation to the use cases

In this section we provide details of the algorithms run as TPC processes.

4.1.1 - Extraction of Audio Descriptors

We extract some baseline features from audio files, including frame based features and harmonic descriptors. A detailed description of these features can be found in deliverable **D3.2 - Music Description**[29], Sections 2.2.1 and 2.2.2.

Inputs, outputs, interactions with the CE

The algorithm takes as input a specific audio file, and outputs a JSON file with all the extracted features. Apart from the conventional input/output interaction with the CE, there is no additional interaction with it.

Implementation details

For the extraction of these features we deploy Essentia. Essentia [1] is a C++ library with Python bindings for audio analysis, description, and synthesis. The library contains a collection of various algorithms which implement standard digital signal processing blocks, statistical characterization of data, and a large set of spectral, temporal, tonal and high-level music descriptors. For the TPL we use the provided docker images[30].

4.1.2 - Rhythm Descriptors

We extract rhythm descriptors from either audio or MIDI data. These descriptors include beat locations, tempo, and time signature. Moreover we provide a human-in-the-loop mechanism for estimating the beats. The user can provide some manual annotation of beats; consequently the beat estimation algorithm will consider these manual annotated beats for a better automatic estimation of beats for the rest of a piece. This is done in conjunction with the annotation tools described in **Deliverable D5.5 - Annotation Tools**[31].

Inputs, outputs, interactions with the CE

The algorithms take as input either an AudioObject (in the case of audio files) or a DigitalDocument (in the case of MIDI files). In the case of providing partial human annotations for estimating the beat,

[29] https://trompamusic.eu/deliverables/TR-D3.2-Music_Description_v2.pdf
[30] https://hub.docker.com/r/mtgupf/essentia
[31] https://trompamusic.eu/deliverables/TR-D5.5-Annotation_Tools_v2.pdf

the input to the algorithm is both an AudioObject and a DigitalDocument (containing the manual annotations).

Implementation details

We deployed several algorithms from the state-of-the-art as well as algorithms developed during TROMPA as described in D3.2. All these algorithms are packaged in a single Docker image.

4.1.3 - Active learning for emotion recognition

Active learning for music emotion recognition allows classification models to improve with new annotations from particular users. In general, we implemented the strategy of *query by committee* in which 5 classification models are used to produce prediction probabilities of data instances which have not been annotated. In short, *uncertainty sampling using entropy* is used over the prediction probabilities of all classifiers, in order to measure the uncertainty produced by particular predictions: instances with low entropy are assumed to be the *most informative*, while low entropy highlights the *least informative* instances that should be annotated by our users.

Inputs, outputs, interactions with the CE

The TROMPA-MER[32] algorithm offers 5 EntryPoints to the CE: to create and copy initialized models for a newly registered user, to extract emotionally-relevant acoustic features from audio, to perform classwise predictions from the features extracted from an audio file, to get the *hard* tracks to be annotated (i.e., least informative data instances), and to retrain the model for each particular user. We clarify each function as follows:

- Create user model: this method uses the unique user identifier from the TROMPA Music Enthusiasts Pilot and creates an internal directory with pre-trained models.
- Extract features from audio: this method inputs audio (as MP3 or WAV) in order to use the OpenSmile C++ software and extract emotionally-relevant acoustic features. The IS13 feature set has been widely used for speech, music and sound emotion recognition. The output of the method is a CSV file with 65 acoustic features (which are later averaged over 1 second with a 50% overlap, mean and standard deviation of each acoustic feature and their first derivatives are extracted in order to produce 260 features).
- Predict emotion: this method input the CSV feature file extracted in the previous method and loads a particular model (in PKL format) to produce framewise predictions for class and probabilities (JSON file). The obtained prediction file contains: framewise output probabilities for each class (“output_probs”), the mean over time of all probabilities (“mean_probs”), the prediction with highest probability (“highest_prob_quad”), the framewise class predictions (“output_predictions”), the frequency of each class over all frames (“freq_pred”) and the mode of the classes predictions (“mode_quad”).
- Get hard tracks: this method requires that all the features from the unannotated dataset have been previously extracted. In our case, we use the music in the Music Enthusiasts Pilot. Since new annotations from a particular user will be used to retrain our models, the user unique identifier is needed to select each user’s model. Additionally, the number of tracks to annotate is an input parameter. The output of this algorithm is a JSON file which contains the

[32] https://github.com/juansgomez87/active-learning

number of the iteration - how many times has this method been called ("iteration"), a list with the least informative songs ("queried"), and the list of remaining songs to be tested in the next iteration ("next_pool").

- Retrain model: this method uses the new annotations presented by a particular user to retrain their committee of classifiers. The annotations are presented using the Muziekweb's song identifier and the particular class that the user has annotated (see sample JSON file). In short, the new annotations are used to retrain all models in the committee.

Implementation details

The TROMPA-MER algorithm is currently written in Python but incorporates the OpenSmile software, which is in C++.

4.1.4 - Emotion-based Music Recommendation

Music Recommendations have a twofold nature in the context of the TROMPA Music Enthusiast use-case: 1) as method of incentivisation for retaining users' when interacting with the pilot; 2) as explanation of how track features are used while building music emotion recognition (MER) algorithms (see Section 4.1.3). The recommended tracks are provided analyzing previous users' annotations, where tracks are listenable through a player presenting additionally a short explanation of part of the features used in the MER models. Consequently, every user receives a different recommendation based on the previous annotations she provided.

Inputs, outputs, interactions with the CE

The TROMPA-ME Music Recommender System[33] is implemented according to the following procedure:

- **Get input.** Three inputs are needed to compute the recommendation:
    1. User unique identifier
    2. Previous annotated tracks (extracted from the CE)
    3. Previous recommended tracks (extracted from the CE)
- **Analyze previous annotations.** First**,** the quadrant-based emotion value is extracted for the last five annotations. Second, the frequency of the emotion values is analyzed, and it is returned the value most frequent within the user's annotations.
- **Retrieve recommendation.** Within the pool of tracks included for the recommendations, the ones corresponding to the emotion value extracted at the previous step are filtered. Within those tracks, the previously recommended are filtered out. Lasty, they are ordered ascendingly by a popularity indicator, and the resulting first track of the ordered list is returned as output recommendation for the users.

Implementation details

The TROMPA recommendation algorithm is currently written in Python. The dependencies required to run this algorithm are found in the requirements file available in the GitHub repository.

[33] https://github.com/trompamusic/music-enthusiast-rs

4.1.5 - Singing Performance Assessment

Choir singers often rehearse their parts individually at home. While this is an excellent way to practice challenging parts in more depth, the conductor's figure is missing. Therefore, the singer does not get any feedback about their performance. The intonation assessment algorithm[34] uses information from the piece's score and pitch descriptors from the actual performance to calculate the average deviation of the singers' pitch from the reference (i.e., the score).

Inputs, outputs, interaction with the CE

The intonation assessment algorithm's usage is as follows:

```
python assessment.py list-of-input-params
```

In the following we provide the main details about the inputs, outputs, and implementation.

- **Algorithm inputs**[35]
  - Start and end bars of the performance
  - Estimated recording latency in seconds
  - Performance's pitch curve (JSON file from VoDesc)
  - Score
  - Score format (opt)
  - Voice identifier
  - Intonation deviation threshold (opt)
- **I/O-related inputs**
  - Path to the output JSON file with the results
  - Path to the INI config file
- **Outputs**
  - The main function generates two output files (real filenames are given as input parameters): `output_filename.json` and `tpl_output.ini`.
  - The JSON file contains the actual assessment results: a Python dictionary with two fields: *pitchAssessment* and *error*.
  - The first one, 'pitchAssessment', contains a list of arrays with the assessment results for each note in the form: `[note_start_time, intonation_rating]`. If the process fails, the list will be empty.
  - The field 'error' will contain a string with an error message if the process fails, and will be *None* if it's successful.
- **Algorithm functioning**: In short, the singing performance (represented by the pitch contour) is segmented into notes using the score information. The algorithm calculates the pitch deviation between the performance and the score in *cents*, in a frame-wise manner. Finally, an intonation rate for each note is calculated by averaging all deviation values within each note.

Implementation details

[34] https://github.com/helenacuesta/intonationAssessment
[35] https://github.com/helenacuesta/intonationAssessment#input-parameters

The TROMPA intonation assessment algorithm is written in Python. The required Python dependencies are specified in the requirements file[36].

4.1.6 - Score Alignment

The performance-to-score alignment algorithm (detailed in D3.5 - Multimodal music information alignment) is used to align MIDI piano performances recorded by users of D6.5 (Working prototype for instrumental players) with MEI score encodings. The workflow takes as input a reference (URI) to an MEI file, a collection of MIDI events obtained by the D6.5 Web interface using the Web-MIDI API, and an authentication token, and produces as output an RDF representation aligning instants in the performance timeline with score elements in the MEI. This representation, alongside a MIDI file recording the submitted MIDI events and a simple audio representation synthesised from the MIDI information, are stored privately in the user's Solid Pod.

Inputs, outputs, interaction with the CE

- **Inputs**
  - MIDI events (JSON Object)
  - URI to MEI file
  - Authentication token
- **Outputs**
  - Alignment (timeline) RDF in JSON-LD format
  - Performance MIDI file
  - Audio synthesis (MP3)

The workflow operates in the following steps, which are executed inside a Docker container - refer to D3.5 for further details:

1. A Python script employing the `mido` module is used to write the MIDI events to a local (performance) MIDI file, and to generate an audio synthesis.
2. The MEI score is downloaded from the supplied URI, and rendered as a (synthesised reference) MIDI file using the Verovio renderer[37].
3. The two MIDI files are aligned using the Symbolic Music Alignment Tool [1].
4. The generated "corresp" file is reconciled with the downloaded MEI file using an R script, resulting in an alignment between the performance-MIDI and the MEI score (via the synthesised reference MIDI) in MAPS (Matcher for Alignment of Performance and Score) JSON format.
5. The MAPS JSON object is converted to an RDF (JSON-LD) representation using a Python script employing the `rdflib` module.

A small separate algorithm can be invoked on user request to publish performances (i.e., the outcomes of the performance-to-score alignment algorithm) to the CE. This algorithm is triggered by the D6.5 Web client after modifying the corresponding Solid Pod data to be publicly readable. The

[36] https://github.com/helenacuesta/intonationAssessment/blob/master/requirements.txt
[37] https://verovio.org

algorithm takes a reference (URI) to the alignment data in the Pod as input, and creates a new Digital Document node referencing the items in the Solid Pod within a List Item corresponding to the performed MEI file. If such a List Item does not yet exist, it is instantiated prior to the creation of the Digital Document.

4.1.7 - Visual Analysis of Scores

The algorithm for extracting a MEI file from a scanned score image is one of the main elements of the Music Orchestras Use Case in order to facilitate the crowd-source OMR improvement. The method is explained in detail in Deliverable D3.4 - Visual Analysis of Scores; it takes as input a scanned image, it detects the measure of the scores along with other music notation symbols (systems, pauses etc) and creates an MEI file encoding that information.

Inputs, outputs, interaction with the CE

The algorithm simply takes a single input that is the score image file and outputs a single output file. The algorithm is CE agnostic, no interactions are made with the CE.

Implementation details

Due to mainly exploitation reasons, the OMR software is not released as an open source software. Thus, the algorithm is run on a dedicated server maintained by PN. PN has a web service that one can upload a scanned score image and retrieve the MEI file. The role of the TPC in this case does not serve as a wrapper of this web service. It gets the input file contained in the CE, it uploads it to the service, it downloads the result and stores the result to the TPC storage, creating at the same time the corresponding CE nodes.

4.3 Automated metadata import

The CE provides a database that allows us to link to existing open data repositories. It's important for us to have a reference in the CE for the content that we wish to work with in the project. This allows us to easily search data from multiple resources at the same time using the Multimedia Component, and also allows us to easily identify resources that are processed using algorithms in the TPC. In order to populate the CE, we developed a data importer which can copy references to data from many repositories and load it in the CE[38]. This data importer uses the functionality provided by the CE client (described in Section 2.3). The importer can load data from the following sources:

- Basic Person information from VIAF.org
- Basic Person information from Library of Congress (LoC)
- Basic Person information from ISNI
- Basic Person information from Worldcat
- Person information from Wikipedia and Wikidata, including biographical information if present, photos if present, and a brief description of the person from the english Wikipedia article about the person
- Person information from Musicbrainz.org, including biographical information, dates and places of birth/death if present

[38] https://github.com/trompamusic/ce-data-import

- ❖ MusicComposition information from Musicbrainz.org, including relationships to the composer and relationships to sub-parts (for example separate movements of a symphony)

Objects that are created in the CE refer to the web resource that they were imported from. This allows us to periodically check back to the resource to see if it has been updated since the last time we imported it into the CE. Many of these sites include references to others in the list. For example, MusicBrainz Artist objects have a rich set of relationships including links to VIAF, LoC, ISNI, WorldCat, Wikidata, and IMSLP. In the case that we identify a relationship between two resources that represent the same item they are joined in the CE using an exactMatch[39] relationship, allowing us to identify that the nodes refer to the same natural person or composition. As a goal of the TROMPA project is to contribute back to these open repositories, in the case that we further identify relationships that don't exist on these external sites, we can subsequently contribute this information back to these repositories using the relevant data submission process for each site.

4.3.1 - Specialised importers

In order to support the specific needs of TROMPA's different use-case client applications, we also developed some importers for more specialised websites:

IMSLP: We import metadata for composers, works, score images, and score encodings from imslp.org. For use in the Singers prototype (see Deliverable 6.6), we imported compositions from IMSLP's "For unaccompanied chorus" category[40] when they included transcriptions in MusicXML format. For these compositions we imported metadata about the composer and work, a reference to the score (a MediaObject), and a reference to PDF files that were generated from the score file. IMSLP hides files behind a copyright disclaimer, as copyright law throughout the world determines that some files are only available in some jurisdictions. The disclaimer advises people to ensure that the copyright status of the file is known in their country. In order to not circumvent this warning, we don't include a direct link to scores from IMSLP in the CE. Rather, we store a permanent link that references the copyright disclaimer page for the score. Users of the CE can automatically accept the disclaimer in order to download the contents of each file.

CPDL: We import metadata for composers and works from cpdl.org. For the singers prototype we imported works from the 4-part choral music category[41] which also have MusicXML transcriptions, as well as the PDF files generated from these scores. MediaObject nodes are created in the CE to represent these scores, and they are linked to the relevant MusicComposition objects.

Muziekweb: For each track in the Music Enthusiasts use case[42], we import an AudioObject which is linked to a MusicComposition and one or several Persons to the CE[43]. A Person is generated with the information from Muziekweb and additional Person objects depending on the external links found in the Muziekweb API: ISNI, VIAF, MusicBrainz, Wikidata, and Wikipedia.

MEI encodings: A number of transcriptions of Beethoven and other piano works were transcribed as part of Deliverable 5.2 (section 2.2) for the purpose of seeding an initial repertoire for the working prototype for instrumental players (D6.5). These transcriptions were made from scores available on IMSLP. We imported information about the composition, the original PDF source, and information

[39] https://www.w3.org/2009/08/skos-reference/skos.html#exactMatch
[40] https://imslp.org/wiki/Category:For_unaccompanied_chorus
[41] https://www.cpdl.org/wiki/index.php/Category:4-part_choral_music
[42] https://ilde.upf.edu/trompa/
[43] Stored at https://github.com/trompamusic/ce-import-muziekweb

about the MEI transcription[44]. The PDF and MEI nodes were linked to reflect the relationship that indicated that the MEI file was transcribed from the PDF.

Humdrum scores: Another source of scores encoded in the Humdrum[2] format is used in the working prototype for music scholars (Deliverable 6.3). We import these scores as MediaObject nodes, and link them to MusicComposition and Person nodes imported from MusicBrainz.

4.3.2 - TPC algorithms supporting data import

WIthin the scope of automated metadata import, we have implemented some TPC algorithms to streamline the process of importing and converting data. The source code for these algorithms makes up a part of the `ce-data-importer` source code.

Data importer: This importer allows anyone in the consortium to import data into the CE from one of the supported sources. The algorithm takes as input a data source (e.g. MusicBrainz, Wikidata, IMSLP), and a specific identifier for a resource in that data source. The algorithm performs the import for the given item, loading it into the CE along with references to any other data sources that are linked from the specified item.

MusicXML file path for IMSLP: On IMSLP, MusicXML scores are distributed in a zip file containing the score in a number of different formats. In order to store which file in the zip archive is the MusicXML file, we use the Archive and Package (arcp) URI scheme[45] to identify the relevant file, and store this URI in the related MediaObject's contentURL field. Subsequent tasks that need a MusicXML file can read the file path from this field. We automatically run this algorithm on any MediaObject nodes that are created with MusicXML scores from IMSLP during the IMSLP import.

MusicXML/Humdrum to MEI converter: For MusicXML and Humdrum files imported from external sources (e.g. IMSLP and CPDL), we used the Verovio toolkit[3] to convert these to MEI. The resulting MEI files are stored in the TPL's minio instance, and are publicly available online. The converter creates a new MediaObject node representing the MEI file, and links it to the MusicXML node with the wasDerivedFrom relationship, and to the original MusicComposition node. We use the TPL functionality to automatically run this conversion on any new MusicXML MediaObject nodes that are added to the CE.

4.4 Implementation considerations

Currently TPL is run on a UPF-MTG server. We have assigned 9 processors to the TPC, 4, 2, 2 and 1 processes respectively for high/medium/low/ priority and failed processes. As mentioned, we provide a minio server to store generated content. The TPC, as well as all algorithms, are encapsulated in Docker containers.

5. Conclusion

In this deliverable we described in detail the final version TROMPA Processing Library. Rather than being a single piece of software, TROMPA Processing Library is a collection of tools, software components and CE functionalities that allow the interaction of WP3 components and other tools such as data importers with the CE data. We presented the details of the mechanism for defining

[44] Stored at https://github.com/trompamusic-encodings
[45] https://s11.no/2018/arcp.html#hash-based

tasks on the CE schema, the multimodal component, the TROMPA Processing Component that includes the scheduling and queuing mechanism, the storage of result data as well as the automatic triggering of algorithms on new nodes in the CE, as well as implementation details of the algorithms that have been integrated in the CE.

The proposed architecture of the TPC is flexible, is easy to scale since it can be run in multiple machines simultaneously, and it is almost independent of software / hardware requirements since all algorithms and the TPC iself are distributed as Docker images. The Multimodal Component on the other hand, it offers a user-friendly interface to explore the CE data that is integrated in some of the pilots.

All the components of the TPL are open source and their final versions corresponding to this deliverable are deposited in the Github repository of TROMPA. However since there are two more months remaining for testing the pilots till the end of the project, these components are subject to possible changes for bug fixes that might appear, minor improvements etc.

5. References

5.1 Written references

[1] Eita Nakamura, Kazuyoshi Yoshii, Haruhiro Katayose. (2017).Performance Error Detection and Post-Processing for Fast and Accurate Symbolic Music Alignment. In *Proceedings of the 18th International Society for Music Information Retrieval Conference*, pp. 347-353, 2017

[2] Huron, D. (2002). Music information processing using the Humdrum toolkit: Concepts, examples, and lessons. Computer Music Journal, 26(2), pp. 11-26.

[3] Pugin, Laurent, Rodolfo Zitellini, and Perry Roland. (2014). "Verovio: A library for Engraving MEI Music Notation into SVG." In *Proceedings of the 15th International Society for Music Information Retrieval Conference*, pp. 107-112.

5.2 List of abbreviations

Abbreviation	Description
UPF	University Pompeu Fabra
TPL	TROMPA Processing Library
CE	Contributor Environment
WP	Work Package
TPC	TROMPA Processing Component
PCP	Pitch Class Profiles
MMC	Multimodal Component