Automated Job Submission Management for Grid Computing

Gaëtan Longrée

Master Thesis

Master en Architecture des Systèmes Informatiques

Academic year **2018-2019**

Acknowledgments

There comes a time in a man's life where one may ponder: "how did I get here?" While this paper and the process of redacting it certainly had me contemplate this very fact, this section is not dedicated to how, but rather whom do I have to thank for this journey. After 25 years of this life, five of which spent in the higher education system, and a total of seven years spent studying in the IT field, I have had the privilege of encountering some amazing people that made me who I am today. This thesis, marking the end of a master's degree cursus, seems like the perfect opportunity put my gratitude to paper.

First and foremost, I would like to thank my mother, without whom none of this would be possible; for pushing me forward, supporting me throughout the years and for all that she has done for me and more. Despite all the challenges she faced, she has always dedicated herself to my brothers and myself, and I would not be the man I am today if it were not for her.

I would like to thank all of my many friends spread around the world, wherever they may be, you have in some aspect or another brought joy, laughter and support to my life, even more so when times were hard and life seemed dull. A special thanks goes out to my Dungeons and Dragons group of friends; thank you for all the cheers and laughter through the years, and hopefully to many more years of rolling dice alongside you.

A special thanks to my promoting professor, **Dr Samuel Hiard**, for his guidance, experience, knowledge, wisdom and his continuous follow-up on my progress through this project.

I would like to thank the entire staff of **Henallux's "Département Technique de Marche-en-Famenne"** for putting together such a welcoming environment for working through a master's degree; all **the professors** for their knowledge and expertise shared with us; and to all **the members of the board of jury** for their continuous feedback through the process of creating this master thesis.

Finally, yet importantly, I would like to thank all the people that I did not mention directly but who contributed in my education and experience, whether directly or indirectly, in both my professional and personal life.

Table of Contents

Table of Figures

Introduction

Since the dawn of the digital era, mankind has used computers to aid in performing complex operations. From electromechanical switches and vacuum tubes of ages past, to the silicon transistors of our modern days, these testaments of human ingenuity have grown more and more powerful with each iteration. Through this continuous evolution, computers have been able to solve increasingly complex problems, some of which were the foundation of groundbreaking discoveries that have shaped the world we know today.

Nevertheless, modern problems require modern solutions, and today's computers must face an increasing demand for processing power to solve more and more complex problems. **High-performance clusters** have risen to solve such increasing demands: these mainframes are composed of multiple computers linked together in order to perform as one large entity. Specialized software have been developed to leverage the combined computing power of each individual computer into one large processing unit.

These clustering software, albeit performant, have proved to be aimed towards efficiency as opposed to being user oriented. Despite this approach being motivated by modern standards, this has created a general disparity between existing solutions. As a result, a same disparity has developed between users' interactions with cluster systems. The procedure to submit tasks to these clusters has become tedious to the point where users have to create their own personal solutions for their submission workflows. In some extreme cases, the complexity of the submission procedure itself has deterred some users from making use of these clusters.

This thesis proposes and discusses a new architecture to solve the disparity and complexity in the users' submission workflows by interacting in a more unified and simplified manner with some of the modern high-performance clusters of today. More specifically, this architecture delves into the interaction with the clusters available throughout the universities in Belgium that are members of the CÉCI[1]. This architecture is presented as an **automated job submission and management platform**. The objective is to offer a more user-oriented abstraction layer for existing clustering software, while maintaining a strong set of features to

[1] "Consortium des équipements de Calcul Intensif" - Consortium for Computation Intensive Equipment.

leverage the performance of these clusters by providing flexibility to the users' submission workflows.

This document will start off by introducing some key concepts, before discussing the problem statement, and defining the reasons behind the need for such a platform. Following will be a list of objectives and challenges faced by the project, as well as an analysis of the existing solutions akin to the platform proposed herein. Subsequently, the platform proper will be discussed at lengths, addressing how the solution proposed responds to the problem statement, the objective and challenges discussed earlier. To demonstrate the feasibility of such a solution, a proof of concept has been created and is discussed after the overview of the solution. Finally, a few words on any potential future projects and a conclusion will draw this thesis to a close.

Theoretical Concepts

In the scientific world, it is common for researchers to have to perform highly complex calculations, so complex in fact that a computer is required to compute them. However, despite the growth in processing power in the last decades, commodity hardware alone may not always be sufficient to perform the calculations in a timely manner.

A **high-performance computing (HPC) cluster** is a group of highly performant computers/servers, called nodes or compute nodes, interconnected together to share resources in order to allow performing multiple jobs in parallel across multiple nodes.

To manage the jobs and the nodes within the cluster, an HPC cluster usually requires a combination of two software. A **scheduler**, required to manage the various jobs submitted and to distribute them across the nodes accordingly, and a **resource manager**, used to manage the resource allocations, such as processor or memory, of the nodes across the jobs to run.

In most cases, a user will interact with a head node (or master node depending on the scheduler/resource manager used). This node will receive the jobs submitted by a user, and based on the scheduler and resource manager configuration and/or algorithm, as well as the user-given input (such as desired CPU power or memory size), the job will be allocated to a number of nodes and executed (given that the cluster has free nodes and enough resources to execute the job at the time of submission).

Slurm

Slurm is an open source workload manager for HPC clusters combining both a scheduler and a resource manager. It is also the solution used by the various Belgian universities that are part of the **CÉCI**. Slurm provides a performant and scalable solution that can also be flexible through the use of a plugin mechanism that allows to add new features such as authentication mechanisms or additional schedulers.

Slurm provides various command line utilities to submit, monitor and manage submitted jobs, such as **sbatch** to submit a job, **sinfo** or **squeue** to monitor the cluster resources or the submitted jobs respectively, or **scancel** to cancel/stop a submitted job.

To submit a job through Slurm, a user is required to create a Bash script to act as a wrapper for the program to execute. In the script file the user needs to enter various options such as the resources requirements, number of tasks to run in parallel, the output files, but also which modules to load if the user’s job requires multi-threaded, multi-process or memory sharing capabilities. With the Bash script complete, the user can then proceed to submit the job using the **sbatch** command and monitor his job execution with **squeue**.

Problem Statement

The **CÉCI** (“Consortium des équipements de Calcul Intensif”[2]) is an association of various universities in Belgium that have gathered to aggregate their high-performance computation clusters in order to unify and simplify the availability of computing resources for university members and researchers. These clusters are hosted by the universities part of the association, including the University of Liège, the University of Namur, the University of Brussels and the University of Leuven.[3]

However, while this consortium allows the use of compute intensive hardware, the clusters remain widespread and require the researchers to make the decision of which destination cluster to use at which location. As introduced earlier in Slurm, the researcher has to create the script with appropriate variables for the cluster’s framework, manually connect to the cluster and finally start running his task from the command line. Moreover, there is no explicit mechanism in place to notify the researcher of the job status, meaning that unless the script was written to run for a finite amount of time, the user must manually connect back to the cluster to monitor the status of his job and to collect the job results when complete.

Although this method allows a certain freedom for the researchers, it also proves to be a hindrance to the researchers’ workflow. Furthermore, this process implies that the researcher is computer literate enough to understand the computational needs of his specific job, know the Bash scripting language and the process required to submit a task to the cluster. The latter is not necessarily a given as the researchers that can make use of the clusters include all the faculties’ students and professors that are part of the universities members of the CÉCI. This can range from astrophysicists making calculations on gravitational waves, to chemists working on protein calculations.

[2] Consortium for Computation Intensive Equipment.
[3] More information at www.ceci-hpc.be.

Needs Analysis

Following a first overview of the problem statement, a more in-depth analysis of the requirements that would need to be fulfilled was performed. The first step in that analysis was to examine the present environment in order to determine the components that would directly impact the architecture to design. This first step highlighted the following components as part of the environment:

- Researchers/Users that wish to submit a job on a cluster;
- The job to submit;
- The clusters, and more specifically the destination cluster for the job execution;
- The job's output/result;
- The user's computer;
- The network between the user's computer and the cluster.

Based on the first analysis of the environment, a second analysis was performed to examine more closely the interaction between each external component and the desired final product. This analysis allows to extrapolate various functions required by the architecture to design.

The diagram in Figure 1 below represents the components outlined in the first analysis and the interaction required with the final solution to design (temporarily named "Submission platform"). The various links between the components are differentiated between **FC** and **FT**. **FC** represents a "function by constraint", meaning that they occur directly between two entities in the system. **FT** represents a "function by transfer", meaning that they occur indirectly between more than two entities.

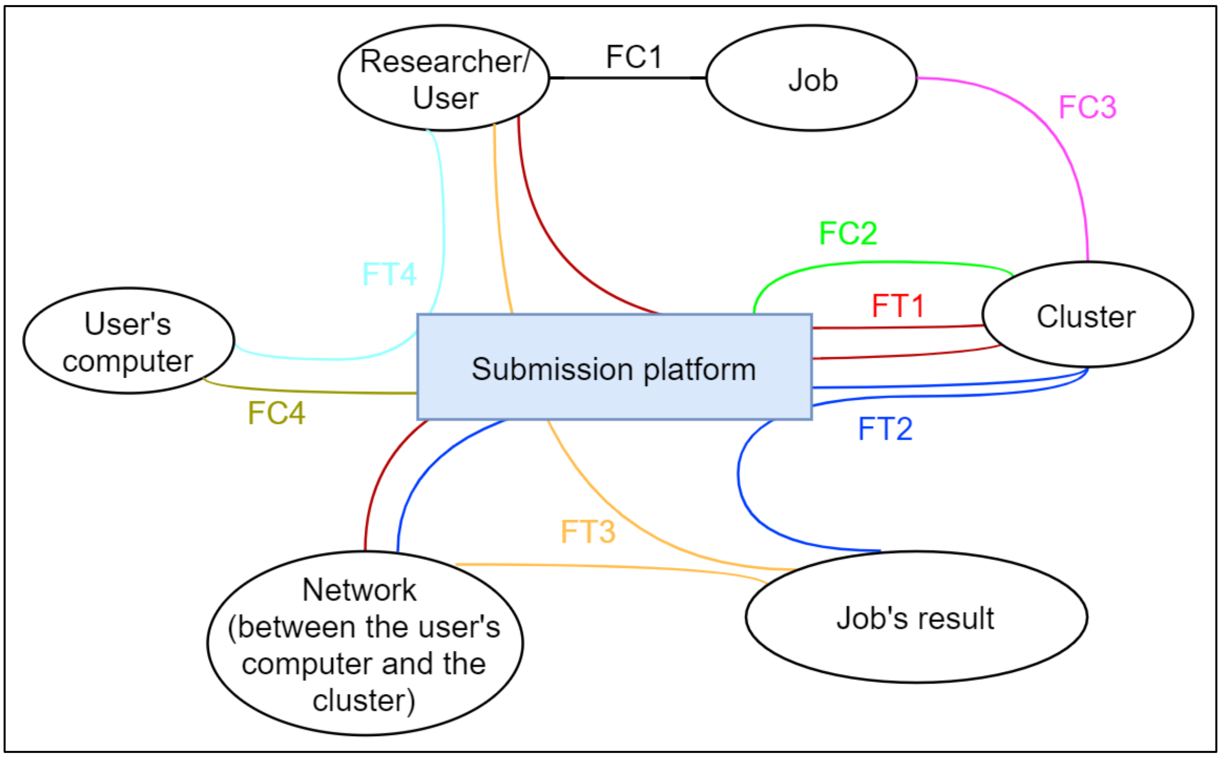


Figure 1 - Environment analysis outlining the interactions between the components/actors.

The following table explains the various functions, their meaning and the criteria that represent the expected behaviors.

Function	Detail	Criteria
FC1	The researcher has a job to submit for execution.	- Access to the platform is restricted to researchers and authorized users.
FC2	The platform can adapt to a varying number of clusters.	- The platform's clusters inventory is dynamic.
FC3	The cluster is able to interpret and execute the job to submit.	- The job is written in a language supported by the destination cluster; - Alternatively, the user is able to install the language required for the job's execution.
FC4	The platform is accessible from the researcher's computer.	- The platform is hosted in a location that is both reachable by the researcher, and where the platform can successfully reach the clusters.

FT1	The research submits a job to a cluster through the submission platform.	- The choice of destination cluster is done based on the type of job to execute; - The job can seamlessly be submitted to the platform; - The platform is able to transfer and submit the task to the destination cluster.
FT2	The cluster returns the job's result to the submission platform.	- The platform is able to retrieve/receive the job's result/output; - In case of error during execution, the user is notified.
FT3	The researcher is able to retrieve the job's result through the network.	- The research can easily view and retrieve the result of the job's execution.
FT4	The researcher can use his/her computer to access the submission platform.	- The submission platform is accessible by the researcher independently of his/her work environment (computer-wise).

Needs Validation

Following the analysis of the environment and the functions/criteria required by the final architecture, the last analysis performed was a validation of whether the project's features and functions would satisfy the problem statement. This is done by defining both the "**why**" and "**what for**" of the final solution and listing the reason(s) for each.

- **Why** does the need exist?
    - Reason 1: the researcher must determine to which cluster to connect to before submitting a job;
    - Reason 2: the researcher must connect manually and remotely to the desired destination cluster;
    - Reason 3: the researcher must create and adapt a script based on both the job's need and the destination cluster;
    - Reason 4: the researcher must submit a job and retrieve its result manually by connecting to the cluster every time.

- **What for**/To what end does the need exist?
    - To facilitate the procedure of submitting a job and retrieving its result;
    - To provide a central location for submitting/consulting jobs scattered across multiple clusters.

Based on the reasons outlined above, a quick table was redacted to emit initial ideas and to provide a solution to each reason that creates the need for this project. Each objective mentioned is then further improved upon throughout this project's overview.

	Reason	**Proposed Solution**
1	*The researcher must determine to which cluster to connect to before submitting a job.*	The choice of destination cluster is either automated or greatly simplified.
2	*The researcher must connect manually and remotely to the desired destination cluster.*	The researcher is not required to manually connect to each destination cluster.
3	*The researcher must create and adapt a script based on both the job's need and the destination cluster.*	The script is created and adapted on behalf of the researcher based on the desired destination cluster.
4	*The researcher must launch a job and retrieve its result manually by connecting to the cluster every time.*	Job submission and retrieval is done on behalf of the researcher from a central interface independently of the job's original destination cluster.

Objectives & Challenges

The goal of this project is to conceptualize a software architecture that will act as an interface between the researchers and the various HPC clusters part of the CÉCI and spread across the universities in Belgium. The objective being to simplify the workflow required by the researchers to choose a destination cluster, submit their computation job, and later retrieve the results.

The main challenge of this project is the modular aspect requested. This implies that if desired by a user, the architecture can adopt one or more new modules designed for adding or replacing an existing task. As an example, given that the first iteration of the software is to let the researcher manually select the destination cluster, the software should be able to accept a new module that would automatically determine the destination cluster based on a given input.

Another challenge faced by this architecture is the task submission itself. This process will require from the software to be able to remotely connect to the cluster, execute the code on behalf of the researcher and retrieve the results of said execution. All the while also maintaining a secure and stable environment to account for the potentially large amounts of connections that could be performed by multiple researchers simultaneously.

Following the task submission itself, the next challenge is retrieving the results. The architecture should be capable of retrieving results that may be stored in diverse forms, or in the worst-case scenario may not be present at all. This implies that either execution of the task and/or task retrieval must be able to handle exceptions and report the failure to retrieve the results to both the software and, consequently, to the researcher.

Lastly, the architecture must also be able to monitor the cluster's presence. This will allow the software to be aware of the available destination clusters in order to either notify the researcher or any modules tasked with selecting the destination cluster.

One challenge that has been voluntarily omitted through this project is the validity of the tasks submitted. As a result, this architecture is developed with the assumption that the code submitted by the researcher is correct and does not need to be checked for errors. However, thanks to modularity objective of the project, the final architecture will be able to accept future modules that may be tasked with the sole purpose of analyzing and validating the submitted code for any errors.

State of the Art

Prior to performing an analysis of the architecture, a state of the art of the existing solutions was established. This research yielded mixed results, some solutions being purely for academic purposes, while other being commercially distributed.

While the academic projects were openly discussed in scientific papers, the commercial products were not as thoroughly documented, hindering the depth of the analysis that was capable. Nonetheless, studying the overview of the functionalities provided by these commercial solutions helped in inspiring the architecture ensued by this project.

Academic Projects

Pegasus is a framework discussed in the paper "*Pegasus: A framework for mapping complex scientific workflows onto distributed systems*" (Deelman, et al., 2005) which proposes a framework in which scientific workflows for complex calculations can be abstracted to render them independent from the underlying execution platform. The framework offers an extensive number of features to provide flexibility and reliability in mapping the abstract scientific workflows to any destination underlying hardware. The mapping itself is performed by redefining the workflow provided as an input to the resources available on the destination computing host. This mapping requires a specific software agent deployed on each destination node to gather information and execute the workflow on behalf of the user.

The paper "*An Interoperable, Standards-based Grid Resource Broker and Job Submission Service*" (Elmroth & Tordsson, 2005) presents a job submission solution that, unlike the Pegasus framework, attempts to be as independent as possible of the Grid middleware used on the clusters. Despite this fact, some configuration is still required to determine the middleware in use and server-side scripts may also be required (this was the case for the NorduGrid ARC discussed in the paper). The job submission platform does not require abstracting of the workflows but provides a web interface for users to submit a job description using a standardized **JSDL**[4] document. Cluster selection is performed by the

[4] Job Submission Description Language – an XML-based data structure for describing non-interactive computational jobs.

architecture using a resource brokering algorithm based on a priori estimations of the total time it would take to perform the submitted job.

The paper "*Creating Personal Adaptive Clusters for Managing Scientific Jobs in a Distributed Computing Environment*" (Walker, Gardner, Litvin, & Turner, 2006) discusses a job submission solution in the context of multiple simultaneous jobs being launched across multiple geographically separate but logically interconnected HPCs. It does so by using proxies deployed in each clusters' site to which a connection is established upon each virtual login performed on the user's local machine. A master agent is then used from the user's machine to communicate with the remote proxies, who will interact with the local cluster's resources (scheduler, job queue, etc.). This project is highly dependent on the middleware used in the HPC; however, it does offer support for other middleware by using an approach of wrapping the underlying middleware script into an overlay script (e.g.: Condor/SGE[5] job-starter daemon executables are wrapped in a GridShell[6] script).

The paper "*Open Standards-based Interoperability of Job Submission and Management Interfaces across the Grid Middleware Platforms gLite and UNICORE*" (Marzolla, et al., 2007) aims to make use of open standards created by the Open Grid Forum (OGF) to allow two groups of HPC clusters using different Grid middleware to provide seamless interoperability. This is achieved by leveraging the web services offered by both the gLite and the UNICORE middleware that support job submission using JSDL formatted job description. It is important to note that this interoperability for job submission is only possible because the gLite and UNICORE middleware support the open standards, however the clusters remain very independent and the paper does not discuss a single and centralized point of contact for job submission.

Commercial Solutions

While many commercial solutions related to HPC clustering technologies exist (including job schedulers, workload managers and grid computing software), very few were found in close relation to this project's scope: a tool/architecture acting as an interface for

[5] Sun Grid Engine – a grid computer middleware developed by Oracle Corporation.
[6] Scripting language used for managing submission on Condor-based clusters.

submitting jobs to an HPC cluster or group of clusters. Below is a brief introduction of the solutions found.

eQueue is a web-based job submission solution offered by Advanced Clustering Technologies Inc. (ACT). eQueue provides a front-end for their underlying clustering technologies, supporting existing schedulers that have already been deployed. The Job submission forms are highly customizable by any administrator, although they require extensive manual configuration on a head cluster node. Their web-based submission platform aims at simplifying the job submission in order to abstract the complexity of command-line based submission for regular users. This solution also helps in providing administrators with improved overview and monitoring on their cluster's usage.

EnginFrame is another example of HPC cluster portal from Nice Software. Similarly to eQueue, this portal is compatible with multiple underlying job schedulers including Slurm and Torque/MOAB. EnginFrame is an open framework relying on Java and SOAP/XML to present a web interface for submitting, monitoring and retrieving jobs to clusters. EnginFrame also offers easy integration with external application and third-party solutions through WebServices publication.

Slurm Federation

Slurm, the workload manager used in the CÉCI's clusters, also includes a federation feature, which has been available since version 17.11 (late 2017/early 2018). The Slurm Federation feature is a scheduling process specifically designed for multiple HPC clusters interconnected together. As opposed to the standalone Slurm manager, the federation feature allows to submit a job from a single cluster but is then replicated to all configured clusters in the federation. Each cluster will attempt to schedule it to its own local cluster, the first cluster to successfully allocate the resources queues the job with a federation-wise unique job ID.

This feature allows to simplify job submission across group of clusters in a manner that, unless the user specifies it manually, the job can be submitted to any cluster member of the federation, independently from the cluster where the job was originally submitted.

Takeaways

After reviewing and analyzing the various solutions mentioned above, I noted that while the academic project aimed for open source solutions, they also tend to remain rather closed in their designs, not leaving much room for change or adaptability. This is mainly due to the dependency between the various processes and operations performed within each project's architecture.

Another aspect I noticed from the academic projects were the dependency of most of the projects with the underlying grid middleware and/or scheduler used within the clusters. While I understand the benefits this dependency provides in terms of functionality, this renders the project statically linked to a single middleware/scheduler, hindering expansion capabilities between heterogeneous clusters.

The growing use of open standards throughout projects for job submission in grid computing is a major step forward to render heterogeneous grid clusters more and more interoperable. However, one might criticize the use of XML-like data structures in standards such as JSDL, as opposed to more modern and more human readable standards such as JSON or YAML.

Another potential issue might be the age of such academic projects ranging from 2005 to 2008, with some grid middleware mentioned already discontinued such as Condor/GridShell. Due to the close integration with the Grid middleware, such projects are very sensible to any major change in the middleware API, rendering them hardly evolutive. This is further motivated by the evolution of computing hardware architectures and the foreseeable introduction of quantum computing, which most certainly will require the grid middleware to adapt to such a technological paradigms.

On the other end of the spectrum, commercial solutions are less present for job submission solutions. Although maintained and up to date with a 2019 version, based on the documentation provided EnginFrame seems to rely on old technology concepts for its deployment. The requirements cited in the configuration guides mention multiple third-party software solutions required (with a majority being proprietary) to be interconnected with the framework in a bus-like network reminiscent of some of the past CORBA[7]-inspired interactions. Additionally, while EnginFrame is presented as an open framework, access to the source code is still protected by a licensing mechanism.

The alternative solution, eQueue, on the other hand, seems to be very closely related to the solution envisioned for this project. Despite citing compatibility with existing schedulers, only Torque and GridEngine are mentioned in the documentation, with no word on Slurm support for example. The web-based approach is very akin to the ideas behind this project, however the major pitfall for its applicability to the CÉCI clusters is that eQueue is aimed towards supporting a single cluster per submission platform, as opposed to multi-cluster support desired for this project.

A common setback found between commercial solutions, and potentially most of the commercial solutions to come, is the proprietary nature of their development. Due to this fact, any desire to make a platform more modular and flexible, both for the administrators and for the users/researchers, becomes more complex, even impossible in a worst-case scenario. One example is that although eQueue allows an administrator to create custom forms for specific job submissions, a submission must still fall within a range of pre-defined variables specified by an administrator, thus not giving full freedom to the users. Furthermore, a user cannot freely create a custom submission workflow and is limited to the submission workflows created by the administrators.

Lastly, midway through the project's progress, I was informed that the workload manager used in the CÉCI's clusters, Slurm, was planned to deploy the federation feature throughout the clusters. While still under testing and configuration at the time of this writing, this feature presents a major impact in regard to this project's utility. As a matter of fact, Slurm federation greatly simplifies the submission throughout the clusters without requiring the user to directly connect to a specific target cluster. This prospect is amongst one of the key factors outlined in the Needs Analysis performed earlier. Nonetheless, as will be explained later on, Slurm Federation is just a variation of the underlying workload manager, and this project's goal is to remain as independent as possible from any specific workload managers.

[7] Common Object Request Broker Architecture.

Overview of the Solution

The solution and its various mechanisms presented below have been designed with several factors in mind. The foremost factor of the architecture is to be able to abstract the complexity of job submission by presenting a simplified web interface to the researchers and having the job submission process automated on behalf of the user.

Another aspect factored during the design process was the willingness to remain as independent as possible from any gridware/middleware, scheduler or workload manager used throughout any given cluster. In other words, the architecture has been designed to be as generic as possible in relations to any cluster that it may come into contact with. This is only possible to certain extent, due to some interactions required with the clusters' scheduler/workload manager, there are limitations to this independency factor, as will be defined later on.

In addition to the independency, a desire to make this platform as universal as possible and compatible out of the box with an heterogeneous array of HPC clusters was a major factor, as is underlined later in the Task Submission & Follow-up section. In a similar mindset, the willingness to make this architecture with as small of a resource footprint as possible was factored in so that deployment in a clustered environment is neither too resource intensive, nor is it disruptive on the existing production environment.

Finally, as introduced in the Objectives & Challenges, the key factor of this solution is the modularity aspect by allowing future integrations of external modules to add additional steps or substitute existing steps with alternative processes. This modularity provides additional freedom and future-proofing qualities to the architecture, while also introducing complexity in the validation of the external module's output results.

Architecture

The solution for a job submission platform proposed by this paper is represented in high level by Figure 2 below. The process workflow represented is as follows:

- A user connects to the platform's web server though a web interface using an Internet browser and submits a job to be run on a specific cluster;
- The web server interacts with a database to read information regarding the clusters and to write various information regarding the newly submitted job;
- The web server interacts with the desired target cluster to submit and monitor the task;
- The job and cluster are monitored during execution and data are sent back to the server and stored in the database for monitoring purposes;
- Once completed, the web server is notified of the job completion, and contacts a mailer server to notify the user;
- The user connects to the web server and downloads the job's output.

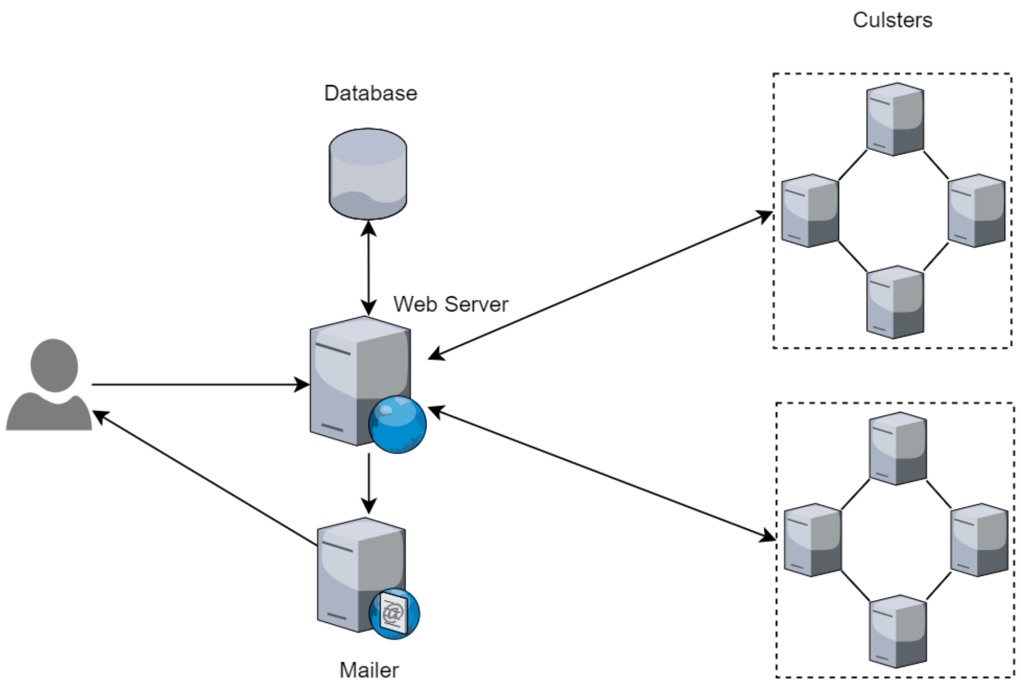


Figure 2 - Overview of the proposed architecture for a job submission platform.

Clusters Inventory

In order for the platform to be aware of the clusters available for job submission, an administrator must enter the cluster's information in an inventory list. This inventory is represented by a database and should keep certain information regarding the clusters. The minimum information to store is as follows:

- **Hostname of the head/master node** – to initiate connections from the platform;
- **Workload manager used** – although our target system for this project only uses Slurm, this value could be useful in the future to allow the platform to support clusters using a different workload manager, and allowing the job submission to be dynamic based on the target cluster's workload manager;
- **List of operation best supported** – this value can be presented to the users when choosing a cluster or could be made available to any external module with automatic decision of the target cluster.

As will be detailed in a later section, the inventory does not require the clusters available resources to be inserted as this will be dynamically monitored remotely.

Authentication

In the context of this project, two authentications must be performed: one by the user when connecting to the platform, and another performed by the platform when connecting to one of the clusters.

Since the connection to the CÉCI clusters are done using CÉCI credentials, themselves linked with the member universities, the authentication performed on the web interface will have to be handled through an external LDAP[8] directory server using the RADIUS[9] protocol. This will allow the platform to avoid storing any passwords, while keeping a Single Sign-On approach for the users.

[8] Lightweight Directory Access Protocol – open standard used for centralized management of distributed directory resources.

[9] Remote Authentication Dial-In User Service – protocol used for providing centralized Authentication, Authorization and Accounting services.

As is, users wishing to connect to a CÉCI cluster through SSH must do so using a private key, sent by one of the cluster's administrator, and a passphrase for the private key (determined upon registration).

For the platform to connect to the cluster, the platform will also require at least a private SSH key to automate the authentication process to the cluster. While this process is the simplest, it does not allow to submit jobs on behalf of a user. To parry this, there are various possible alternatives.

One solution would be to overlook the user who submitted the job through command line and rely solely on the platform's database entry for tracking user submissions. This method would however require some user limits and quotas to be lifted to allow the platform's account on the clusters to submit large quantities of jobs and to potentially store large amounts of data.

Another solution would be to allow the platform's user account on the clusters to perform a user switch and authenticate as another user. In order to avoid storing passwords, this would require the platform's account on the clusters to have the privilege required to switch users without requiring authentication.

Lastly, a third solution would be to have an automated process create a second key pair upon a CÉCI's account creation for a user. This key pair would be dedicated to the submission platform and would allow to ensure a separation between an SSH session initiated by the user and an SSH session initiated by the platform on behalf of another user.

These solutions have a different advantages and disadvantages. The first solution would require a special account to be setup on the clusters, which could represent the least cumbersome solution in terms of management overhead and would additionally represent the least dangerous solution in regard to security.

The second solution is interesting as it would allow to retain a tracking of which user submitted which tasks independently from the submission platform, but if the platform's account were to be compromised, it would represent a major security risk.

The third solution represents the most complex solution to implement and would provide little added benefit compared to the second proposition, other than additional tracking of user submissions through the SSH sessions. The third solution does represent the most dangerous in terms of security risks: if the platform were to be compromised, the secondary private keys stored on behalf of the users would be considered unsafe, requiring a massive purge in public keys on the clusters, which in itself would add an important administrative overhead.

Ultimately, the decision of authentication method to use for such a platform would fall under to the various administrators in the CÉCI's member universities. As such it will not be further investigated in this project.

Database

As introduced in the architecture overview, a database is required to store information to be used through a job submission's process lifecycle. The following section takes a look at a non-exhaustive list of information that should be considered when designing the production database. In an effort to remain technology-agnostic, no database model or technology is dictated and is purely left at the developer's discretion. However, for the sake of simplicity, the various groupings will be referred to as tables.

The first table required and already introduced earlier is the Clusters Inventory. This table is tasked with simply listing the clusters usable as job submission destinations and to maintain the information required for connecting to those cluster. Some of the elements to take into consideration for this table include:

- A user-friendly name (different from the hostname/IP address);
- The cluster's hostname/IP address;
- The port used for SSH connection;
- The workload manager/scheduler used;
- The type of job preferred.

Additionally, depending on the number of differences and requirements of each workload manager/schedulers used, each cluster could also contain information specific to its local workload manager configuration. One such example, in the case of the CÉCI's clusters and the Slurm workload manager, each cluster has a specific "partition" value used for submission which differs from one cluster to another. This information could be stored along with the cluster or in a separate table depending on the model implemented in the production environment.

Another element that should also be taken into consideration are storage specific values. Storing and accessing stored job results is an integral part of the platform and could be variable from cluster to cluster. Having separate storage spaces for job execution and long-term storage is common in cluster deployments; such information should also be stored along with the clusters in database. This information could be stored both as absolute paths, or more practically, as the name of the environment variables used on the clusters to identify these storage spaces. Once again, using CÉCI's clusters as an example, the location for a user's distributed home directory is identified using the **CECIHOME** environment variable. This location differs from the working directory located at **GLOBALSCRATCH** which uses data retention policies much different in terms of long-term storage. These values should be stored in a globally unique field name in order to have a generic access to these values in the underlying code, further abstracting the difference of destination clusters.

A second table that is mandatory for the platform's correct execution is one dedicated to tracking user submissions. This table is potentially the most complex as it should contain the largest amount of information, including:

- The user's username;
- The user's email address;
- The destination cluster;
- The job submitted and whether it is a single local file, a folder or a remote git repository;
- The commands to perform to execute the job;
- During and after execution: the status of a job, the exit code, the time allocated, the resources used, etc.;
- Cluster specific variables, such as resource reservations;
- The location and/or files the job will write its output to for retrieval;
- The external modules that the user may have indicated.

Additionally, this project also promotes a high degree of flexibility, and as will be detailed in later sections, additional information may be stored. This could include information such as compilation commands that are needed to compile the job, or additional packages that the user wishes to install from source, or even additional variables solely destined for the modules later on in the job's process pipeline.

Another table that may be present (and discussed later in section Error Management and Monitoring) is one used for collecting monitoring data regarding the clusters' global resources and availability. This table could contain, for each cluster located in the inventory, information such as:

- The current resource utilization of the cluster (processor, memory, disk, etc.);
- The amount of jobs currently queued and waiting for execution;
- The amount of jobs currently running;
- The total number of nodes available;
- A history of reachability when the cluster was online or offline.

These information could be used for many monitoring purposes, including dashboarding for administrators or users alike. One use case that could be integrated with the aid of such information would be monitoring the job queues prior to submission and estimating the waiting time based on the number of waiting jobs in order to offer alternative solutions to the user for quicker job execution. However, this falls beyond the scope of this project.

Modular Approach

As introduced in the Objectives & Challenges, a strong emphasis is placed on the modular requirements of this job submission architecture. The ability to expand such a submission platform would allow it to remain more future proof by allowing the community to contribute and offer alternative steps and process in the submission workflow.

To tackle this challenge, it was important to define how deep the modularity should go. From a high-level perspective, I conceptualized the submission process to be a sequential list of tasks. This allows the modular aspect to be structured and interoperable while keeping a degree of liberty.

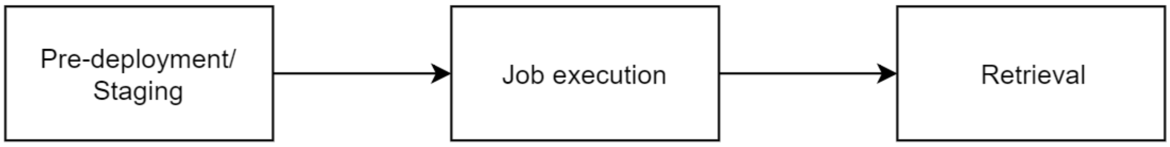


Figure 3 - High-level overview of the modular approach.

The process is staged in three different steps:

- **Pre-deployment/Staging**

  This first stage is responsible for accepting as input the job that a user wishes to execute as well as the entirety of the parameters that are needed for the later stages. It represents the front end of the solution. It is from this stage that all the pre-processing of the task will occur and information will be gathered: the type of job (serial, parallel, etc.), the destination cluster, the resources required, the data files used, the runtime desired, and so on. These various parameters can be user submitted or defined by another method such as through user created modules (detailed later on).
- **Job execution**

  In this stage, the solution will connect to the desired cluster on behalf of the user, configure the required parameters as defined by the previous process, and execute the job. In order to provide monitoring, the job will be executed from a "**wrapper**" that will act as an overlay to maintain communication with the central submission platform.
- **Retrieval**

  This stage will take care of notifying the intended user of the end of the job execution and of the retrieval of the job result by providing a space to view and/or retrieve the results.

Module Interaction

To render the modular approach as interoperable as possible, these stages require a precise list of inputs from the previous stage. This allows to maintain a clear structure in the process while allowing user submitted modules to be leveraged. A more technical representation of the steps envisioned with the modular aspect and the interaction between the modules is represented in Figure 4 below.

This approach allows user created modules to be used at various points in the submission pipeline. The **Staging** area allows external modules to be used for various tasks relating to determining the parameters required for the execution. This is precisely where a module leveraging Machine Learning/Artificial Intelligence to determine the resources required for a job could be implemented. Another module that could be integrated would be a module tasked with parsing the user's code to determine the type of operation, whether it is prominently serial or parallel based, or to look for fatal errors in the code and potential memory leaks (see Future Works).

The **Job Execution** phase includes the possibility to implement modules to be executed before and after the submitted task is ran. This **preprocessing** step could be leveraged to include operations that should be ran on the destination cluster prior to run a task. This could include operations such as downloading and building custom libraries for execution or gathering and formatting data used by the job itself. The **postprocessing** step on the other hand could be used for further processing a job's output, to execute a custom clean-up procedure, or to upload the results to a third-party external server. Both modular stages' presence allow for a greater deal of flexibility to the customizability of the submission workflow executed by a user.

The **Retrieval** phase includes the most basic process of gathering the information from the user's job completion and notifying the user of said completion. Depending on the production environment, this phase could be further augmented with result migration functions if the production environment makes use of external storage, independent from the cluster's storage, for storing the job results.

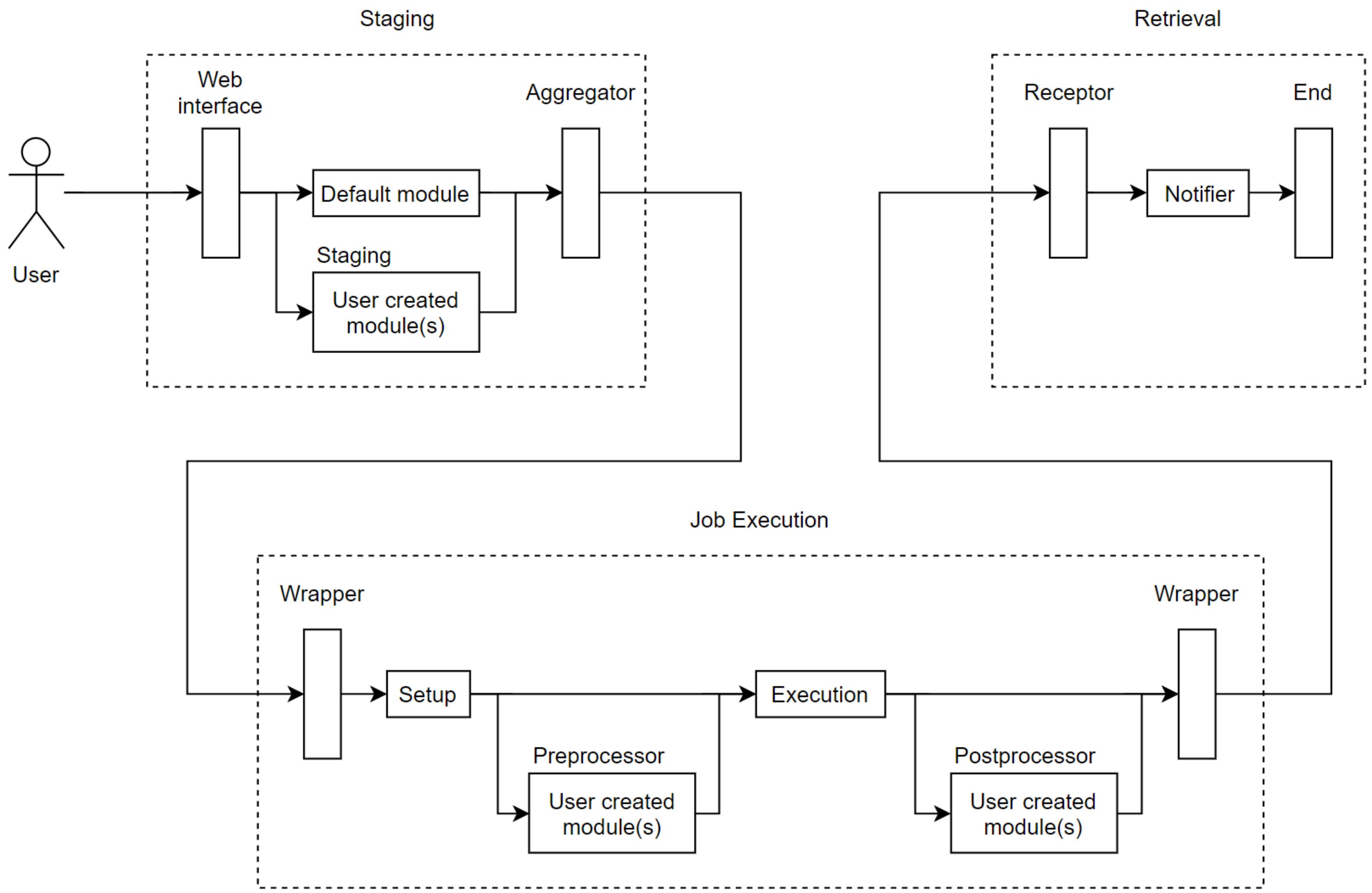


Figure 4 - Detailed overview of the modular approach.

In order to ensure the interoperability between user created modules and the remainder of the infrastructure, it is imperative to define and impose a set of input and output standards. Through this process, users wishing to define their own modules for integration with the final platform would be required to comply to this imposed structure in order for the process to follow through.

First of all, user modules would have to be submitted through a web platform. Along with uploading their modules code, the submitter should also specify the input to be expected and the output it provides.

To optimize modularity further, the platform would employ a dynamic dashboard from which the user could select the various modules available and organize them in the order they wish for them to be executed. Based on the user submitted input/output, the platform could perform a verification to ensure the input/output relations are respected and validate the submission for next step.

Through this dashboard-style organization, a user could choose which modules should be executed sequentially. For example, one could imagine a set of modules for determining the type of job (serial or parallel) and an Artificial Intelligence module to calculate resources dependent on the type of job. An additional module could be available to analyze the code for potential memory issues. The user would then be able to determine which modules are executed and in what order through the dynamic dashboard to represent a sequence as shown in Figure 5. The platform would then perform verifications to ensure those modules are compatible based on the input/output provided by the user who created and submitted the modules to the platform.

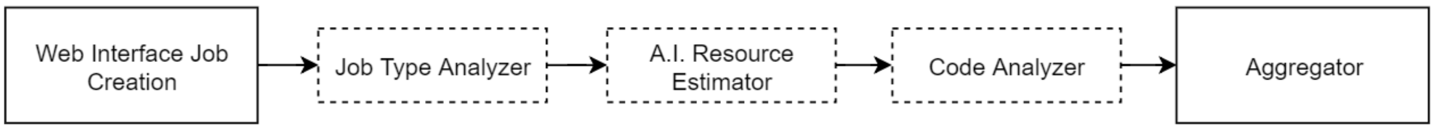


Figure 5 - Example of modules' sequence of execution as determined by a user.

The advantage of such a design is that multiple modules could be used at once, while still ensuring compatibility between them. On the other hand, it also means that the input of the last modules must be imposed and verified. The role of the Aggregator would be, as the name implies, to aggregate the various outputs from the modules but most importantly to validate that it has all the data and information needed to correctly submit the job for execution.

As for the input data coming into the modules, it is not as crucial to ensure conformity as for the output data at the end of the process. This is mainly due to the fact that the input data will be passed as a parameter from one module to another (or from the beginning of the process and into the first modules). Hence this data can be either used or ignored by the modules themselves without having an impact on the functionality or the interoperability of the process.

Inter Module Communication and Input/Output Parameters

The **Default** module (represented in Figure 4) would present a simple web interface with a form for the user to submit the information required by the Aggregator to submit the job for execution. This information should include:

- User submitting the job;
- Destination cluster to use;
- Resources required;
- The job file or the URL to a Git repository;
- Optionally, the compilation command to perform on the submitted file or in the Git repository;
- Any external data used by the job;
    - *Could also be a single file, a URL, a TAR/ZIP archive;*
- Any additional software or packages required.

These arguments described above would be the minimum required by the Aggregator module to allow the Execution module to be able to run the submitted job. Some arguments may be optional, such as the additional software or packages, but overall this is what the Aggregator module must validate prior to pursuing the process.

In the presence of custom user submitted modules, additional parameters must be present in order for the subsequent steps and sub-processes to be aware of the presence of additional modules to be executed in the pipeline.

The first additional argument is an array to represent the additional modules to execute and in which order of execution. To allow each module to have its own set of argument and parameters, the array or list should contain within each entry a dictionary object with following values:

- An index for the order of execution;
    - *To ensure compatibility with data structures using unordered arrays by default;*
- The module's name;
- Compilation command and arguments;
    - *If the module is written in a compiled language as opposed to interpreted;*
- Execution command;
    - *Again, if the module has been written in C++ for example, the execution will differ than from a Python script;*
- Arguments to be passed at command execution;
- Modules resulting output;
    - *Filled after module execution.*

These extra array parameters should be present at most three times in the data structure passed between steps, one for each potential sequence of external modules present in the workflow: staging, preprocessing and postprocessing.

While these structures could be dropped after each major step in the workflow (i.e.: the Execution step does not require the Staging's sequence of custom modules), it remains of potential interest to keep such information for future-proofing reasons. Keeping this information between steps allows for future modules to perform operations based on the knowledge of previous operations performed in the workflow's sequence.

Note that for this to be possible, the modules must also be written in order to accept a "keyword arguments" style as last parameter (such as the *kwargs* in Python or the *argv* in C++). With such a support, the modules could be consistently executed and passed as last parameter the data structure maintained by the modules' supervisor.

The concept of passing as last argument the entirety of the platform-wide data structure would empower the developers and their created modules with the knowledge of the entire process's information, including global parameters, which custom modules are yet to be executed and how, and which modules were already executed and what result did they yield.

User Created Modules

As has been hinted in the previous section, one of the desired strengths of this platform is not only to allow the inclusion of user created modules in the workflow of the job execution, but also to allow users to write modules in different languages.

It is for this specific reason that the data structure that is carried between various steps includes information such as compilation and execution specifications. This would allow modules written in Python or Java to be simply executed using the Python interpreter or the Java VM, while languages written in C or C++ to be compiled for the specific cluster and then executed.

The major pitfall of this implementation is the sensibility to errors as well as the potential difficulty in troubleshooting module errors in the workflow. While module error management is out of the scope of this project per se, this issue could be mitigated at least in part by a couple of mechanisms.

Module contribution guidelines should be redacted and very strict on input parameters and output data format. This would minimize the risks associated with bad interpretation of input and output data. Other contribution guidelines should also include indications on the compilation and/or execution information required for each module.

The wrapper in charge of executing the modules should be capable of not only capturing the output from the module’s standard output, but also be capable of capturing and storing the output from the module’s standard error output. This would be able to provide a minimum of information as to any module’s execution errors, which could be reported both to the user who submitted the task, as well as to the user who created the module.

Task Submission & Follow-up

As introduced in the Modular Approach section, the stage of Job Execution is taken care of by a "**wrapper**". This wrapper will act as an overlay as a means of managing and monitoring the underlying process tasked with the job execution. This process is in part inspired by the Ansible project[10], which uses Python and SSH to remotely manage and orchestrate large number of machines.

While this thesis is mainly programming language agnostic, this is a rare case where the Python language is highly motivated for this part of the project. The reason being that the goal of this project is to remain largely independent from the underlying middleware used throughout the clusters, as well as to be compatible with a wide array of destination clusters and heterogeneous environments. As a direct result of this independency, no external software should be required to be deployed in the clusters.

To this end, Python becomes a primary choice for the wrapper development language for many reasons. The Python interpreter is available on a large majority of UNIX and Linux-based systems and is installed out-of-the-box and ready to use. Another advantage of Python is that, unlike compiled languages, Python is portable and does not require to be recompiled for the destination system. This makes a Python-developed wrapper compatible with the majority of HPC clusters out-of-the-box, making it relatively effortless to expand the list of destination clusters.

Development language aside, as introduced earlier, the role of the wrapper is to provide an overlay for management and monitoring. The process envisioned is to leverage SSH tunnels (once again a feature readily available on the majority of HPC clusters) to connect to the head/master node and deploy the wrapper software along with the job files and all the parameters described in the Inter Module Communication and Input/Output Parameters section.

As represented in Figure 4 under Module Interaction, a **Setup** step is part of the wrapper's first task. This Setup step is itself divided into a two-phase part. The first part of the setup task is to gather information about the destination system. This is sub-step used to

[10] More information at www.ansible.com. Source code available at github.com/ansible/ansible.

gather information such as the operating system, the overall resources available and other information related to the shell environment. On subsequent connections, it is used to update such information to maintain the remote platform's inventory up to date.

The second phase of the setup task is responsible for maintaining and configuring the environment. The first part will be to ensure that the required Python packages are present and up to date. The second part is will be to gather the job files according to the user input (such as cloning the job's Git repository from the URL or unpack the job files from the submitted file/folder), install any additional software required and/or run any compilation command indicated by the user upon submission (if applicable).

With the setup complete, the wrapper will then execute any **Preprocessor** modules indicated by the user, catch the output result of each module and pass them to the next module, if multiple modules are present, or pass the module(s)'s output to the Job Execution step.

For the **Job Execution** phase, the wrapper will execute the job as configured for the destination cluster's workload manager. In this paper's scenario, the workload manager used by the target cluster is restricted to Slurm. This implies that the wrapper will create the bash script on-the-fly, using the parameters passed by the previous steps, and add the job to the queue. The wrapper will then monitor the progress of the job, sending periodic updates to the remote platform for the user to monitor as well (view Error Management and Monitoring for more details).

Upon job completion, the wrapper will then move on to the **Postprocessor** phase and execute any modules indicated and catch the output, using the same process as for the Preprocessor phase. Once the Postprocessor is finished, the wrapper then completes its execution by sending the result back to the platform for the last step in the workflow.

Independency from the Workload Manager

As mentioned at numerous occasions, amongst the objectives of this project is to ensure the generic approach to interacting with the underlying workload managers/schedulers of any HPC clusters. This method ensures that submission of a job is possible and successful, regardless of the heterogenous nature of the destination clusters.

This independency has a major limitation: in order to have this abstraction layer, there must be at least one component aware of the workload manager and what interactions are needed to correctly abstract it in a manner that seems transparent to the remainder of the execution.

Remaining on the Python approach envisioned for the wrapper, one method to implement this abstraction would be to design a module/package that would return a different object depending on the current cluster's workload manager, while still using consistently named methods and variables regardless of the underlying differences.

Knowing the underlying workload manager is possible thanks to the Clusters Inventory database containing the needed information. The abstraction could then easily be implemented though an **abstract factory design pattern**. The abstract factory would allow for the entirety of the wrapper to rely on generic method names and static variables, with the abstracting module/package providing the abstract constructors containing all of the abstraction layer.

Error Management and Monitoring

Due to the complexity of the workflow presented in this thesis, the modular approach and the number of components requiring interaction between one another is bound to present errors at some point in a job's execution.

While the wrapper's function acting as overlay would certainly allow it to catch errors and report them, it is important to note that this type of error management is not within the original scope of this paper. As such it is assumed that any module's execution and the job submitted is devoid of any errors.

The primary concern in error management for this paper's scope is the reachability of the HPC clusters upon job submission. Additionally, it felt adequate to incorporate some kind of monitoring mechanisms in order for users to not only be aware of whether the desired target cluster was online or not, but to also be warned of the resources already in use on the destination cluster.

Error Management

Due to the more extensive monitoring mechanism described in the next sub-section, the verification of a cluster's reachability becomes somewhat of a trivial task. To periodically monitor whether the clusters in the platform's inventory are up and running, the use of ICMP[11] Echo-Request messages[12] at regular intervals, and the subsequent update of the information in a database, should suffice in providing both current and historical record on clusters' reachability. This information could be displayed in both a front page and a monitoring dashboard, providing useful data for both users and administrators.

Moreover, an additional verification mechanism could be implemented so that whenever a user accesses the submission form, or more specifically the destination cluster page, an ICMP Echo-Request message could be sent to either all clusters, or the selected cluster. This would allow to provide the user with the most up to date information on a cluster's reachability.

This kind of monitoring could also be leveraged by an additional step in the workflow; one could envision a form that would request only the type of operation a job requires, such as parallel, serial or single threaded. Based on this information, along with the cluster's supported operations, current reachability and resource monitoring information (detailed below) all stored in a database, the platform could perform the choice of destination cluster to maximize job efficiency, minimize the job's queue time on the cluster, and ensure the destination cluster is live and running.

Cluster Monitoring

As introduced earlier, the wrapper's overlay mechanism provides a number of advantages, one of them being the ability to act as a "chaperone" for the job being executed on the cluster.

In order to provide information to the users through the platform, the wrapper could be leveraged to include a "phone home" feature. This feature would periodically send information back to platform for monitoring purposes. This information could include global cluster resource utilizations such as processor, memory and disk usage, the number of cluster

[11] Internet Control Message Protocol.
[12] Otherwise more commonly referred to as a "ping".

nodes currently available, as well as the number of jobs currently running and the number of jobs currently waiting in queue. The same feature could also send information about the job currently submitted and chaperoned by the specific wrapper.

As with the reachability data, this information could be collected and stored in a database and subsequently displayed in a dashboard for users and administrators to monitor the status of the various clusters in the platform's inventory.

Once more, while this thesis aims to be language and technology agnostic, one technology that could easily be leveraged for this single purpose is a RESTful API. Given that the target platform is mostly aimed at being a web platform with a graphical user interface, a REST API is a rather simple technology to implement in parallel. This would provide the wrapper with an efficient communication channel for the "phone home" feature, allowing to periodically perform POST/PUT operations with monitoring data regarding both the cluster and the current job's status.

Result Retrieval

As indicated by Figure 4 detailing the various steps in the job submission's workflow, the last step of the **Execution** process is for the wrapper to hand over the operation to the **Retrieval** process. This last step is tasked with notifying the user of the end of the job execution upon receiving a notification from the wrapper ending its execution on the cluster.

As already discussed in Error Management and Monitoring, a REST API service is already of prime interest for a monitoring function of the jobs' execution and clusters' resource utilization. As a result, the REST API service can also be reused for notification mechanism. This solution also aligns with modern standards and solutions in addition to the JSON approach of RESTful APIs being fully compatible with our inter-module communication's data structure employing a key-value pair approach.

The wrapper's last task in the Execution process would be to make an API call to the platform's endpoint, submitting the data structure discussed in Inter Module Communication and Input/Output Parameters and all its additional values to the platform's notification endpoint. This would trigger the Retrieval step to execute the **Notifier** process that would send a mail to the job submitter.

Thanks to the data structure discussed, it should also be possible to notify the user if the job was unable to finish due to an error, in addition to providing the error message that was caught on the standard error output by the wrapper. This would require in the API endpoint's POST payload a flag indicating whether the execution was successful or not and in the case of the latter, what error message to send to the user.

Job Result Storage

In order to maintain the platform's footprint as small as possible, the actual result of job's output will not be stored on the same server as the one hosting the platform. Prior to making an API call, the Wrapper will store the result in a dedicated space for later retrieval by the user.

As per the configuration of the CÉCI clusters, the job's result will be stored on a more long-term location, to allow for a larger time frame for users to retrieve their job result. In order to support other cluster configuration than CÉCI's, the long-term location storage for finished job should be stored in the inventory database (as mentioned in Database). Additionally, the result should also be placed into an archive to make the download more optimized as opposed to requiring the transfer of multiple small files.

Since the result will not be shared on the platform proper, and to keep within the scope of simplifying the user's workflow without requiring him/her from connecting to the cluster, when a user connecting to the platform wishes to download their job's result, the platform could act as a proxy between the client and the cluster where the results are stored.

Proxying of the download is performed by the platform initiating an SCP[13] connection to the cluster to download the job's result in memory. Once entirely in memory, the transfer between the user and the platform can be done through a regular HTTP channel.

This method requires a "buffering" process as seen from the client side, as well as requiring the platform to have sufficient memory at its disposal for short-term storage. This approach allows to reduce the requirements of the platform by leveraging the existing storage of the clusters, reducing the overall resource footprint without compromising user experience in any significant way.

[13] Secure Copy Protocol – file transfer protocol between network hosts using SSH.

Proof of Concept

In order to concretely apply the concepts presented above as well as to demonstrate the feasibility of this project, a Proof of Concept has been developed. The following sections will describe the architecture and mechanisms of the Proof of Concept as well as motivate some of the choices and the reasons why they differ from the concepts introduced earlier.

Features Implemented

Out of the features proposed in the Overview of the Solution, a subset of features has been selected. These features have been selected based on three major aspects and objectives of the project:

- **Modular approach:** the ability to integrate various external user-created modules on the fly;
- **Compatible with heterogeneous environments:** the ability to operate with different HPC clusters out of the box;
- **Independency from the underlying workload managers:** the ability for the source code to be as generic as possible despite any difference in a cluster's underlying workload manager.

Additionally, some accessory features have also been implemented, such as an API server and a database for centralized access to data related to existing clusters and submitted jobs. The **Notifier** role has also been implemented for a simple follow-up on submitted jobs.

Design Choices

Due to the relatively trivial aspect of the web graphical interface, the proof of concept has been developed as a standalone command line interface tool. In order to replace the user input that would be performed by the web interface, the user specified parameters are given to the program through the use of YAML files. As will be defined later, user input includes both information specific to a job as well as any potential external modules to run.

The YAML file format has been chosen for its simple human readability, while remaining complex enough to support some basic data structures such are lists/arrays and dictionaries.

The programming language chosen for developing this proof of concept is Python. The reasons behind this choice are partially for its low learning curve, but mainly for the arguments already presented in Task Submission & Follow-up; Python is a very widespread and highly portable programming language across Linux and UNIX based distributions, thus working towards the **heterogenous support** objective of the proof of concept. Additionally, Python is a rather explicit language with good human readability, making it ideal for an open source project that can be forked and extended by anyone interested in expanding the project.

Architecture

To keep the proof of concept in a similar design as the project's conceptualized architecture, the proof of concept's architecture has been developed as follows:

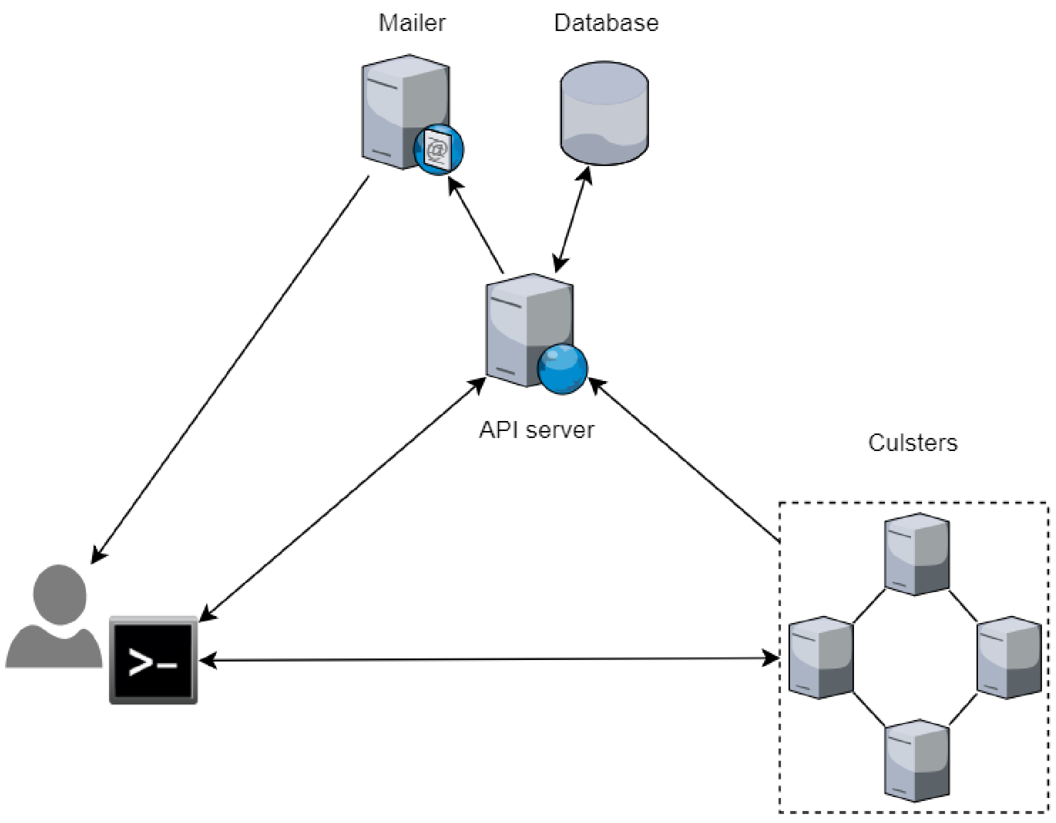


Figure 6 - Overview of the Proof of Concept's architecture.

The command line tool is executed on the user's local computer, and will interact with an external API server, which itself will interact with a dedicated database. The command line tool will connect to a cluster and deploy a wrapper, which will submit a given job and will also interact with the same API server. Additionally, as an added feature, the wrapper will also interact with an SMTP server indirectly through the API server.

Proof of Concept Workflow

As mentioned previously, the web interface has been replaced by a command line tool, which is executed locally on the user's computer. The user must redact one or two YAML files (one for the job's specific information, the other for the modules definition) and submit them to the command line tool. The following paragraphs present an outline of the proof of concept's execution workflow, also represented in Figure 7 below.

Upon submission the tool will validate the information provided in input, and if indicated, will start executing the **Staging** external modules. If no external modules have been given, or if the modules have executed successfully, the tool will locally validate the data one last time to ensure that all the required information is present in the user input.

Before attempting to connect to a cluster, the tool will make sure the cluster is reachable using a single ICMP Echo-Request/Reply. If the original destination cluster is not reachable, the tool will query the database for any other cluster that is reachable **and** presents a similar affinity to the preferred job types as the original destination cluster. This allows to manage against any unavailable cluster while still giving the user the ability to select an alternative cluster based on the knowledge of the potential type of job the user wishes to perform.

Once a cluster has been selected, the tool will connect to the destination cluster via SSH using the private key provided by the user. The tool will transfer the **wrapper module** and the files necessary for the remote operation: the user input data as well as user's modules specification and the necessary module files (if applicable). For performance reasons, the wrapper and modules are first compressed into a TAR archive to optimize data transfers.

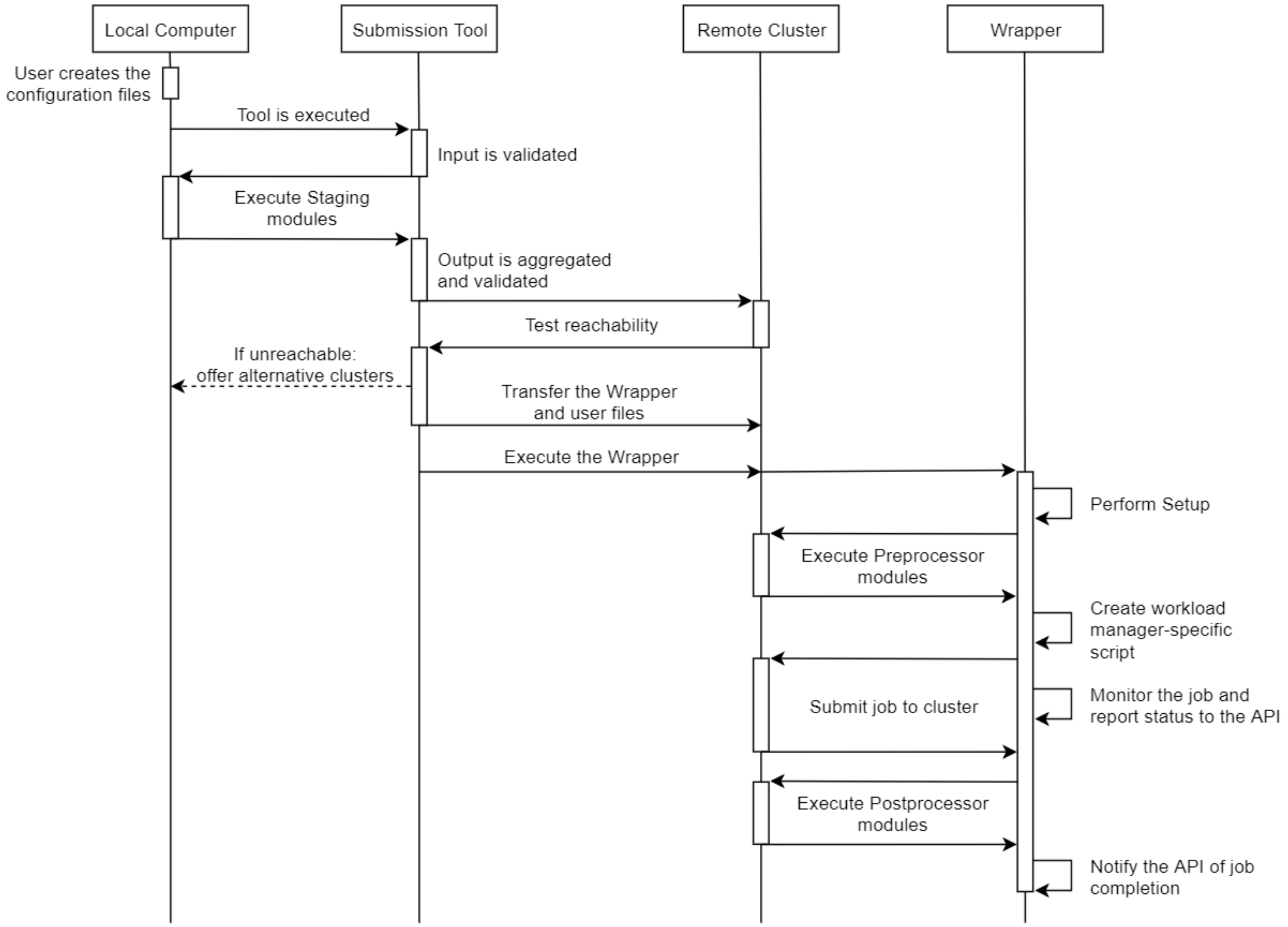

Local Computer
Submission Tool
Remote Cluster
Wrapper
User creates the configuration files
Tool is executed
Input is validated
Execute Staging modules
Output is aggregated and validated
Test reachability
If unreachable: offer alternative clusters
Transfer the Wrapper and user files
Execute the Wrapper
Perform Setup
Execute Preprocessor modules
Create workload manager-specific script
Submit job to cluster
Monitor the job and report status to the API
Execute Postprocessor modules
Notify the API of job completion

Figure 7 - Overview of the Proof of Concept tool's workflow.

Once the required files have been correctly transferred and unpacked, the tool remotely executes the wrapper in background before closing the connection and returning a UUID to the user to identify the submitted job.

Once the wrapper has been launched, a series of four major steps will take place, mainly reflecting the detailed overview presented in Figure 4 earlier (section Module Interaction):

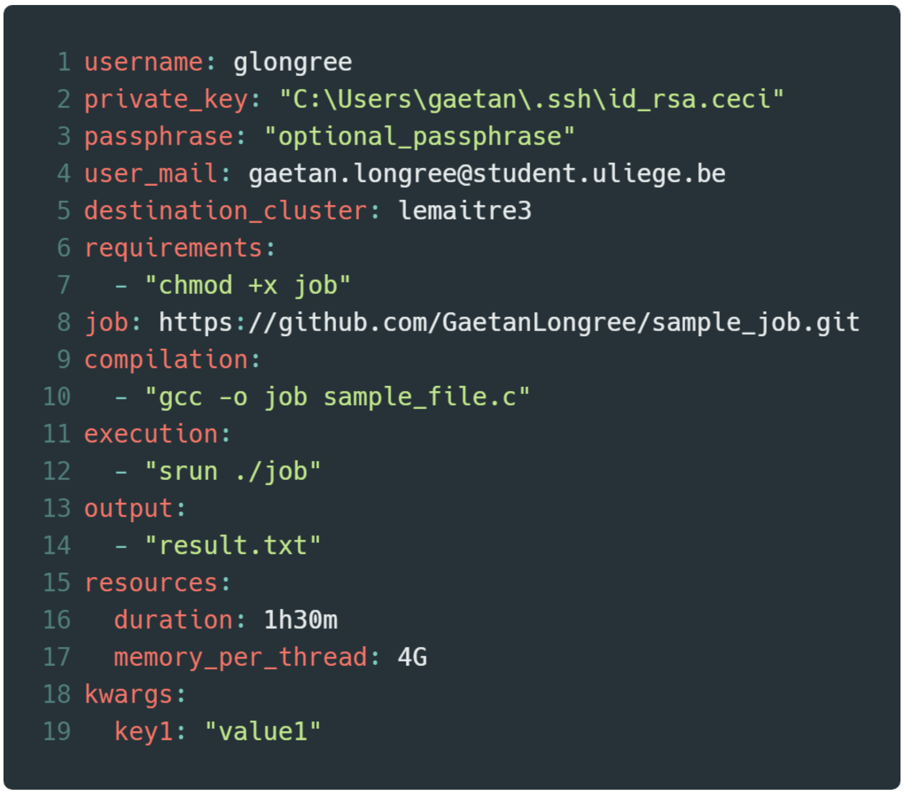
- The **setup phase** will prepare the environment by executing any user defined pre-required commands, downloading or unpacking the job files, and executing the compilation commands (if necessary);
- The **module handler** will compile and/or run any pre-processing modules defined by the user;
- A **workload manager** module will create the script according to the destination cluster's workload manager, the resources requested by the user and the job execution specified, before submitting the job and waiting for its completion;
- Once completed, the **module handler** will once again compile and/or run any post-processing modules defined by the user before handing over the completion.

Upon completion of its execution, the wrapper will contact the API to update the final status of the job, which in turn will launch the **notifier** module which will send an email to the job owner to notify of the job's completion status.

User Input

As mentioned, the user input is passed to the command line tool via a YAML file. The structure outlined in Figure 8 below is separated into a series of key/pair values. Some of these values are mandatory and are validated by the tool before execution. Other values are optional due to the potential use of external modules to define them further down the workflow (one example being the resources values).

The **job** key is used to specify the job file to be used. This value can be a single file, a folder containing the job files or a URL to a git repository. This allows the flexibility to either transfer local files or to clone a git directory to the cluster for execution.



```yaml
username: glongree
private_key: "C:\Users\gaetan\.ssh\id_rsa.ceci"
passphrase: "optional_passphrase"
user_mail: gaetan.longree@student.uliege.be
destination_cluster: lemaitre3
requirements:
  - "chmod +x job"
job: https://github.com/GaetanLongree/sample_job.git
compilation:
  - "gcc -o job sample_file.c"
execution:
  - "srun ./job"
output:
  - "result.txt"
resources:
  duration: 1h30m
  memory_per_thread: 4G
kwargs:
  key1: "value1"
```

Figure 8 - Sample output of a YAML user input file.

Some keys are flexible, such as **requirements**, **compilation** and **execution** which can accept either a single value or an array of values. The proof of concept has been developed to accept both types of variable to allow greater freedom to the user, and greater readability in the case of single values.

Requirements represents operations that should be performed prior to the job submission, notably to download or install additional packages from source. The **compilation** is also performed prior to submission should the job files require compilation. Both those steps are executed during the **setup** phase described earlier. Finally, the **execution** key are commands to be placed in the workload manager's script. In the case of this proof of concept, these values would be placed in a Slurm script.

In the other entries, the **username**, **private_key** and **passphrase** values are used to authenticate to the clusters. The **private_key** is the path to the private key file (both absolute and relative paths are supported, both on Linux and Windows based operating systems) while the **passphrase** value is used for ease of use. Due to security concerns, if the user is not willing to store his passphrase in cleartext, it is possible to omit this information, in which case the passphrase will be queried upon the execution of the tool.

The **resources** key accepts another key/pair dictionary with various keys depending on the resources the user wishes to reserve for his or her job. The values were mostly determined based on the CÉCI's documentation as well as the Script Generation tool also provided by the CÉCI[14]. These keys are later interpreted by the **workload_manager** module detailed later.[15]

The **results** key is used to list the file or files where the job will write the result of its operation. This field and its use are detailed later in section Result Storage and Retrieval.

Lastly the **kwargs** is a fully optional entry that is independent from the tool's normal operation. As will be developed later on, this entry is used to store other key/value pairs solely used to communicate variables to or between custom modules further down the operation workflow.

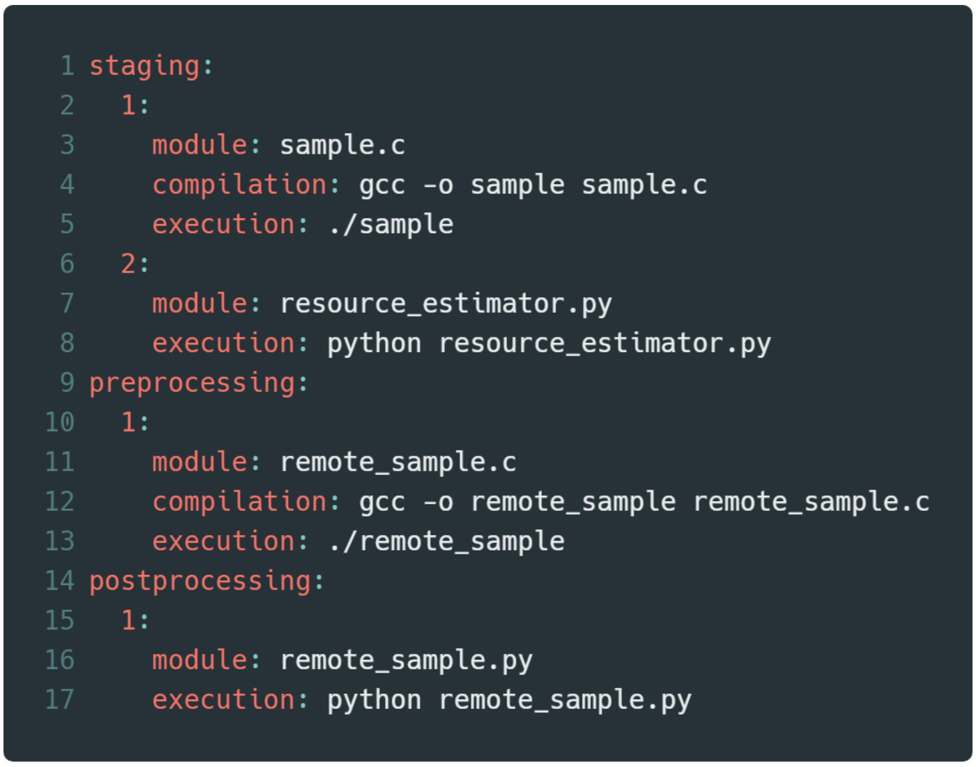
External Modules Support

In a similar manner as for the user input, the external modules process order and the user defined details were substituted for an external configuration file. Once again, a YAML file is used with a specific structure to define which modules to execute, how to execute them and in what order, as presented with the sample output in Figure 9 below.

As introduced in the Modular Approach section of the overview, the proof of concept has also implemented the three main stages of the external modules' execution: **Staging**, **Preprocessing** and **Postprocessing**. This allows greater flexibility and freedom to the end user in the overall workflow of execution.

Additionally, each stage is divided into a key/value dictionary, each key indicates the order of execution in the stage's process, while the value holds the information linked with the module's execution in another key/value object. The keys related to the module's execution are tasked with defining how to execute the module according to the module creator's guidelines.

[14] Script generation tool available at www.ceci-hpc.be/scriptgen.html.
[15] For a list of keys possible, please refer to the README file in the project's GitHub repository.



```yaml
staging:
  1:
    module: sample.c
    compilation: gcc -o sample sample.c
    execution: ./sample
  2:
    module: resource_estimator.py
    execution: python resource_estimator.py
preprocessing:
  1:
    module: remote_sample.c
    compilation: gcc -o remote_sample remote_sample.c
    execution: ./remote_sample
postprocessing:
  1:
    module: remote_sample.py
    execution: python remote_sample.py
```

Figure 9 - Sample output of a YAML user modules specification file.

The first key, named **module**, serves two purposes. First and foremost, it identifies the name of the module itself. Secondly, and most importantly, it is the name of the module's file, which must be stored in a specific location. All of the **module** values in the file will be aggregated by the tool upon first execution in order to create a list of modules that need to be transferred along the wrapper. This is done for optimization: by compiling the list once and packaging all the modules in a single TAR archive file, a single file needs to be transferred over SSH instead of transferring each file in a granular fashion. This improves the overall efficiency of the initial transfer of the wrapper and the modules.

The **compilation** key is a fully optional key. Given a module written in a compiled language, this key is used to specify the command to execute in order to compile the module on the destination cluster. This key allows the modular approach framework to support modules written in languages both compiled and interpreted. Furthermore, by allowing the user to install external packages through the **requirements** directive of the User Input, users are able to install other languages that may be used by external modules.

The **execution** key represents the command to execute for the module to be run. While a variety of modules could be executed using a variety of languages and interpreter, a common factor with all the modules during execution is that an additional argument is appended at the end the of the indicated command. This last argument is a string representing a JSON dump of the entire runtime variables used by the tool and wrapper right before the module's execution.

By passing the runtime variables as parameter, this allows the modules to be aware of the entire context of the job submission process. The runtime information contains mainly the entire user input provided (as described in User Input), which includes the **kwargs** directive with user defined variables, in addition to the information about the current cluster. A **modules** directive is also present, containing the user defined information about the modules, both that have been or will be executed, but also includes the output of each module already executed.

This method of passing the runtime information to the module and storing its output presents the opportunity for the modules to make use of the information regarding the job execution. It also allows the modules to make use of any data concerning all the other modules in the workflow (both those that have already ran or those that have yet to be executed) in addition to allowing them to make use of the output created by any other modules previously executed.

This process does come with a major pitfall in the proof of concept: the execution of the modules is done through the **subprocess** Python package, using the **Popen()** method, which allows to monitor and directly communicate with the process. However, the communication is done through the standard input/output and error outputs. As a result, in order to capture the output produced by a modules, it must be created on the process's **standard output**. While functional, it implies that the last directive of any module before exiting must be to print its output to the STDOUT channel such as through a **printf()** in C and C++ or a **print()** in Python.

Workload Manager Independency

In order to maintain the code fully independent from the underlying workload manager, the proof of concept contains a package named **workload_manager.py**. This package provides a level of abstraction to act as an interface for the underlying manager in use on the destination cluster.

The approach used is similar to an **abstract factory** design pattern. The **workload_manager.py** provides a **get()** function accepting as single parameter a string denoting the workload manager in use on the current cluster. This same value is obtained from a database containing the information regarding the cluster (in our case, from the API server).

The **get()** function returns a class object containing various methods used to interact with the underlying manager. The methods contained within the class are generically named, such as **submit_job()** and **get_job_status()**, for basic interaction with the manager.

The class also contains some static keyword lists such as **TERMINATED_STATUSES**, **TERMINATED_SUCCESFULLY_STATUSES**, **WAITING_STATUSES**, and **RUNNING_STATUSES**. These lists are used to monitor the current job state during its execution and notify the client of any change. Note that two separate lists exist for processes that have **terminated** and that have **terminated successfully**. It is essential to know not only when a submitted job has terminated, for notifying the user first and foremost, but in order to continue the process correctly the wrapper must ensure that the status returned confirms that the job has terminated successfully as opposed to being stopped due to an error.

While the proof of concept includes a single class for the **Slurm** workload manager, the goal of the project would be to have the **workload_manager** module extended to support a wider range of workload managers **using the same interface** as the one currently in place and with the identical return format.

By using this approach, the majority of the code is allowed to remain fully independent from the workload manager used by the underlying cluster thanks to the use of generic class methods and variable names. This allows the tool to make decisions based on information about a running job obtained from the abstract factory and comparing these values with a subset of static variables containing the expected values based on the workload manager class returned by the same abstract factory method.

Result Storage and Retrieval

As mentioned in the User Input section, upon submission, the user is asked to specify the name, or the list of names, of the file(s) where the job will write its result(s). This information is initially used solely for reference as it is not used by the tool until the retrieval phase, later initiated by the user after the job's completion. This value is still stored and transferred to the wrapper running on the cluster, so as to remain a variable usable by any subsequent modules.

For the sake of simplicity, the proof of concept assumes that any job stores its result in the same folder where it was executed. All submissions performed by the tool are executed in the user's home folder on the cluster, in a directory named after the job's UUID (the same UUID returned to the user upon submission). Depending on whether the job file is a single file, a folder or a git repository, the execution may vary slightly. Therefore, when executing a job, the wrapper saves the precise location, in its absolute path format, and saves it with the runtime information, later uploaded to the database through the API.

To retrieve a job, the user must run the same tool with different arguments, mainly the UUID of the job the user wishes to retrieve. Before attempting to connect, the tool contacts the API to get the latest status of the job execution. Until the status is within a range of accepted values defined by the destination cluster's workload manager (through the same abstract factory mechanism as described above in Workload Manager Independency), the tool will not attempt to connect to the cluster, and notifies the user that the job is still being executed. If the job has terminated successfully, the tool will contact the API to get the original user input submitted by the user (in order to use the **results** key), as well as to get the job's working directory (saved by the wrapper upon execution of the job). Additionally, the destination cluster's information is also fetched to connect to the cluster.

For security reasons, this proof of concept does not store neither the private key nor the passphrase of the user, hence before connecting to the cluster, the user is also asked to provide the private key and enter the passphrase to connect to the cluster.

Once connection is established, using the **results** and the **job's working directory** values gathered beforehand, the tool transfers the file, or files, to the current directory.

Improvements

The proof of concept is fully functional as is with the code being freely available in a public Git repository available at github.com/GaetanLongree/MASI-TFE. However, as introduced in the beginning, this proof of concept presents a reduced scope of the features discussed in the complete Overview of the Solution, some of which were omitted due to their complexity in implementation in respect to what they brought to the proof of concept's demonstration of feasibility.

Amongst some of the most important improvements that were not implemented is the validation of output/input between modules. The main reason for this feature not being present is the overhead in the data needed to be stored separately in its own database. The concept of validation is based on a centralized platform where users can submit their modules for other users to use. Implementing the validation between modules would imply developing a similar submission solution, hence increasing the size and complexity of the proof of concept. Since the proof of concept is intended to be used with specific modules with a full knowledge of what each modules do, this part seemed overly complex for the relative benefit it would add to the proof of concept as a whole.

Similarly, the proof of concept includes some parsing of user values. The parsing performed is to ensure correct operation of some of the components later down the process's pipeline. While in a finished version, a lot more input parsing and validation would be required, once again this proof of concept is not intended for final users, but for demonstration purposes, and uses a subset of predefined values within a range of known parameters. Thus, working on making the proof of concept as resilient as possible to bad user input was deemed counter productive in relation to what it would add to the final proof of concept.

One feature that could have been implemented was the permanent storage location based on the destination cluster. The goal of this feature, as partially explained in Result Storage and Retrieval, would be to move the job's result to a more permanent storage for later retrieval as opposed to leaving the results stored in the user's home directory. This would allow the result to be stored for potentially longer period of time, and in a space dedicated to long-term storage. This was not implemented for the simple reason that through the development of the proof of concept, the API was an accessory implementation. The API itself ended up being developed much later in the development process, with the core tool functionalities having already been written earlier in the development lifecycle. Thus, implementing the storage location functionality would have required a major refactor of the tool, increasing on the development, and most importantly testing, debugging and analyzing times required in this project. Furthermore, such a feature does not impact the functionality nor the demonstration capability of the proof of concept, hence representing only a minor feature.

Next Steps

The goal of this proof of concept was to demonstrate the feasibility of the project's concepts presented throughout the Overview of the Solution. While limited in its scope and features, it still integrates the key aspects which motivate the need for such a solution in a multi-cluster environment.

This proof of concept's tool is limited to a command line utility, but it represents the equivalent of the back-end aspect of the submission platform envisioned. The next phase of development would be to migrate this demonstration proof of concept to a more full-fledge prototype aimed at direct user interaction and evaluation. By involving users early in the project development, this would allow to gain even more valuable feedback that could guide the development of the project and would also help to more closely align the project with the user's day to day needs.

Ideally, the **workload_manager** module mentioned earlier in Workload Manager Independency could also be the subject of a wider development, by including abstract factories for additional workload managers, increasing the list of supported workload managers. Such an initiative would allow for the project to increase the compatibility with more cluster implementations. This approach could be further motivated by an open source initiative towards expanding the current proof of concept.

Future Works

This thesis, and the ensuing proof of concept, have been redacted in such a manner as to leave the door open for future improvements or derived work. The following paragraphs outline potential and non-exhaustive avenues for further study and improvements on this project, or avenues that could be the subject of their own research and paper.

Additional External Modules

One of the key objectives of this project is to give users the freedom and flexibility to integrate external modules in their submission process. This modularity offers the possibility to integrate various external modules with varying functions (some of which have already been mentioned in this paper).

One such module could be a "resource estimator" module tasked with parsing the submitted code and provide an estimation of the resources required to compute the task. This module could also use information such as the destination cluster and the knowledge of the cluster's preferred type of operation in order to better calibrate the resources required. Such a module could perform an estimation based on a simplistic "sandbox execution" approach or make use of more advanced technologies such as Machine Learning/Artificial Intelligence to study the resources required.

Another module idea that could be explored would be one tasked with parsing the user's submitted code and analyze it for any potential programming errors. The module could search for errors as simple as syntax mistakes, or as complex as potential memory leaks, race-conditions and loop optimization. The parsing process could also be used to analyze the code and determine the type of task required: whether it is more oriented toward serial or parallel execution (information that could be further leveraged by a resource estimation module described above). One could argue that such a module could also directly integrated into the platform's submission workflow out-of-the-box for direct error checking upon submission.

Dedicated Packet Manager

As introduced in the Modular Approach and later developed in the Proof of Concept, the **Setup** stage of the execution process can be used to install additional packages based on the commands provided by the user. Despite giving the user freedom over what and how he wishes to install additional packages, this process can be cumbersome and impede the simplistic approach of a job submission.

A solution that could work towards simplifying installation of additional packages in the submission workflow would be a dedicated package manager. This could allow users to select additional packages to deploy by submitting a list of packages required. Such a solution would require interpreting the packages required and make use of an abstraction layer to perform the commands to install the packages on behalf of the user. This abstraction would also require access to a database of known packages and known deployment methods based on the destination system, including differentiation the operating system and architecture used by the clusters.

Conclusion

This thesis aimed at presenting a new architecture to offer as an automated job submission platform for researchers wishing to submit tasks to high-performance clusters. As has been presented, despite the existing solutions, both academic and commercial, very few offer both an independency from the underlying clusters' middleware and a high degree of flexibility for the users.

Throughout this thesis, I have presented a proposal for a new architecture that would offer the best of both worlds. Through the use of an abstraction layer with the abstract factory design pattern, the platform can be fully independent from any underlying workload manager or scheduler used by a cluster. The modular approach also stands as a great flexibility feature for any researcher wishing to submit a task. This very approach allows a user to redefine the workflow of a job submission process, by allowing him to integrate various external processes in the form of modules.

Additionally, the architecture also presents some forward-thinking features to allow it to work with a variety of heterogenous environments. Through a dynamic inventory system, the platform can easily integrate additional clusters for the researchers to use, and an automated fact-gathering process on these same destination systems allows the platform to be aware of the systems' capacity and reachability. The result retrieval process, leveraging existing storage on the destination clusters, also makes the architecture relatively low-profile in regard to deployment.

As demonstrated through a proof of concept, this automated job submission architecture is not only feasible, but also presents itself as a versatile solution as a command line utility tool. The modular approach has also proved to be capable of a high degree of adaptability when offering support for multiple programming languages, both compiled and interpreted, thus reinforcing the flexibility and strength of this approach as a submission workflow.

Despite the success of the proof of concept in demonstrating the feasibility of the architecture's concepts, due to the scope of the project and the destination systems limited to clusters part of the CÉCI, the abstraction layer has only been developed for a single workload manager. As a result, the effectiveness of the independency from heterogeneous workload managers/schedulers used by different clusters is only as strong the abstraction module itself. Such a project could stand to benefit the most from a community driven open-source development model in order to expand the list of supported workload managers through contributions by the community.

Finally, this thesis is merely the presentation of a feasible concept for an automated job submission platform. Based on the proof of concept, the project could be fully expanded to represent the architecture presented here: combining the command line tool with a full-fledged web interface for a more user-oriented interaction. Furthermore, as stated multiple times, this project and its modular approach could be expanded in a continuous manner through user-created modules, adding more and more features to the submission workflows. This approach makes this architecture a great opportunity to act as a launch pad for future projects and development.

Bibliography

Abstract Factory. (n.d.). Retrieved March 2019, from Refactoring Guru: https://refactoring.guru/design-patterns/abstract-factory

Adaptive Computing. (n.d.). *Moab HPC Suite*. Retrieved February 2019, from Adaptive Computing: http://www.adaptivecomputing.com/moab-hpc-basic-edition/

Advanced Clustering Technologies, inc. (n.d.). *eQUEUE*. Retrieved February 2019, from Advanced Clustering Technologies: https://www.advancedclustering.com/products/software/equeue/

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., . . . Zaharia, M. (2009). Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS, 28*(13). Retrieved February 2019

CECI. (n.d.). *CECI Documentation*. Retrieved February-May 2019, from CECI Documentation: https://support.ceci-hpc.be/doc/

CECI. (n.d.). *Consortium des Équipements de Calcul Intensif*. Retrieved February-May 2019, from CECI: http://www.ceci-hpc.be/

Colignon, D., Lozano, A., Cabrera, J., Wautelet, F., Skozlowskij, S., Leplae, R., . . . Keutgen, T. (2018). CÉCI News by the Sysadmins. *CÉCI Scientifc Day*, (p. 8). Namur. Retrieved April 2019

Colignon, D., Lozano, A., Cabrera, J., Wautelet, F., Skozlowskij, S., Leplae, R., . . . Keutgen, T. (2019). CÉCI News by the Sysadmins. *CÉCI Scientific Day*, (pp. 8-15). Brussels. Retrieved April 2019

Deelman, E., Singh, G., Su, M.-H., Blythe, J., Gil, Y., Kesselman, C., . . . Katz, D. S. (2005). Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming, 13*(3), 219-237. Retrieved February 2019

Donders Centre for Cognitive Neuroimaging (DCCN). (n.d.). *Running computations on the Torque Cluster*. Retrieved February 2019, from The HPC Wiki: https://dccn-hpc-wiki.readthedocs.io/en/latest/docs/cluster_howto/compute_torque.html

Elmroth, E., & Tordsson, J. (2005). An interoperable, standards-based Grid resource broker and job submission service. *First International Conference on e-Science and Grid Computing (e-Science'05)* (p. 9). IEEE. Retrieved February 2019

Lee, W., McGough, A., & Darlington, J. (2005). Performance evaluation of the GridSAM job submission and monitoring system. *UK e-Science All Hands Meeting*, (pp. 915-922). Retrieved February 2019

Marzolla, M., Andreetto, P., Venturi, V., Ferraro, A., Memon, S., Memon, S., . . . Hedman, F. (2007). Open standards-based interoperability of job submission and management interfaces across the grid middleware platforms glite and unicore. *Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007)* (pp. 592-601). IEEE. Retrieved February 2019

NICE Software. (n.d.). *EnginFrame - HPC Portal*. Retrieved February 2019, from NICE Software: https://www.nice-software.com/products/enginframe

Pegasus WMS - Automate, recover, and debug scientific computations. (n.d.). Retrieved February 2019, from Pegasus Workflow Management System: https://pegasus.isi.edu/

Slurm Federated Scheduling Guide. (n.d.). Retrieved April 2019, from Slurm Workload Manager: https://slurm.schedmd.com/federation.html

Varrette, S., Bouvry, P., Cartiaux, H., & Georgatos, F. (2014). Management of an academic HPC cluster: The UL experience. *2014 International Conference on High Performance Computing & Simulation (HPCS)* (pp. 959-967). IEEE. Retrieved February 2019

Walker, E., Gardner, J. P., Litvin, V., & Turner, E. L. (2006). Creating personal adaptive clusters for managing scientific jobs in a distributed computing environment. *2006 IEEE Challenges of Large Applications in Distributed Environments* (pp. 95-103). IEEE. Retrieved February 2019

What is GridShell/Condor, and where is it installed on XSEDE? (2018, January 18). Retrieved April 2019, from Indiana University: https://kb.iu.edu/d/axdd