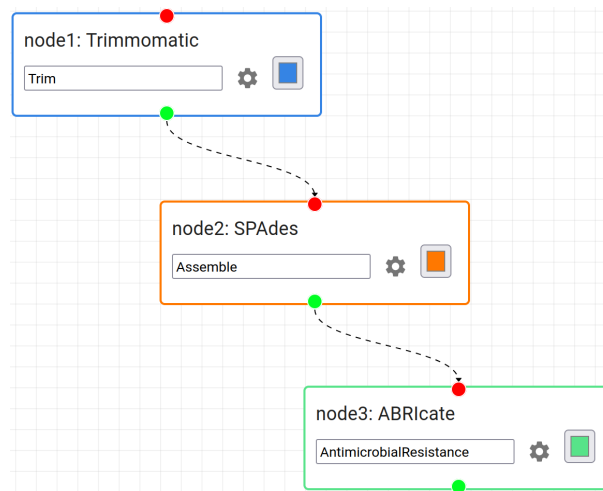




Instituto Superior de Engenharia de Lisboa
Departamento de Engenharia de Electrónica e Telecomunicações
e de Computadores



FLOWViZ: Framework for Phylogenetic Processing

Miguel Filipe Paiva Luís
(Bachelor of Science)

Final project to obtain the Master Degree in Computer Science and Engineering

Supervisor: Doctor Cátia Raquel Jesus Vaz

Committee:

President: Doctor Nuno Miguel Soares Datia

Members: Doctor José Manuel de Campos Lages Garcia Simão
Doctor Cátia Raquel Jesus Vaz

December, 2022

Acknowledgments

À Professora Doutora Cátia Raquel Jesus Vaz, do Instituto Superior de Engenharia de Lisboa, pela sua excelente orientação, disponibilidade e apoio imprescindível que me prestou ao longo da realização da minha tese.

Ao INESC-ID pela atribuição da bolsa, que financiou o projeto e permitiu-me submeter e apresentar um artigo no simpósio INForum 2022, na Guarda.

Aos meus amigos, em especial, aos meus colegas de mestrado e de licenciatura David Albuquerque e Nuno Gomes.

Em especial, um enorme agradecimento aos meus pais e irmã que sempre estiveram comigo, apoiaram-me ao longo da minha vida e sempre acreditaram em mim.

Sem todas estas pessoas nada disto seria possível e dedico este trabalho a todas elas.

Abstract

The increasing risk of epidemics and a fast-growing world population has contributed to a great investment in phylogenetic analysis, in order to track numerous diseases and conceive effective medication and treatments.

Phylogenetic analysis requires large quantities of information to be analyzed and processed for knowledge extraction, using adequate techniques and, nowadays, specific software and algorithms, to deliver results as efficiently and fast as possible. These algorithms and techniques are already provided by a great set of free and available frameworks and tools, such as PHYLOViZ[23].

Most of the applied techniques and algorithms used for phylogenetic inference tend to form work pipelines - procedures formed by steps, which typically have an intrinsic dependency between them. Although it is possible to execute work pipelines manually, as it has been done for decades, nowadays, is not feasible, as genomic datasets are very large, and the respective analysis is time-consuming. The transition between steps also needs human interaction and each step must receive the matching data, correctly, which can introduce human error. Because of this, software were made to ease and reduce manual interaction, so these procedures could be automated. This type of software is typically referred as a workflow system - software which allows users to create workflows, on top of a provided Domain-Specific Language (DSL)[13], where procedures are translated into scripts, through the definition of a group of steps and their specific parameters and dependencies.

There are already many software solutions available, which differ in their Domain-Specific Language and workflow structuring, leading to a great software heterogeneity, but also low workflow shareability - as users work on different workflow systems. Thus, when they share workflows with others, time needs to be spent converting and adapting certain workflows to a specific workflow system, so work pipelines can be executed, making workflow sharing a difficult task.

This lead to the creation of the Common Workflow Language (CWL)[2] - a new standard which provides a way to execute workflows and work pipelines among different workflow systems. However, not every system supports this new standard.

This project aims to build a framework on top of an already existing project - PHYLOViZ, which provides a set of state-of-the-art tools for phylogenetic inference. The developed framework, will link phylogenetic inference web frameworks with workflow systems, giving the user freedom to build its workflows, using the provided web framework's or its remote tools, through a user-friendly web interface. Resulting in workflow automation, task scheduling and a more efficient and faster phylogenetic analysis.

The project was supported by funds, under the context of a student grant of Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020, for a INESC-ID's project - NGPHYLO PTDC/CCI-BIO/29676/2017 and a Polytechnic Institute of Lisbon project - IPL/2021/DIVA_ISEL.

Keywords: Workflow systems, Phylogenetic framework, Software integration.

Resumo

O aumento do risco epidemiológico e o constante crescimento da população mundial contribuiu para que se fizesse um forte investimento na análise filogenética, de modo a monitorizar doenças e a conceber tratamentos e medicação rápidos e eficazes.

A análise filogenética utiliza grandes quantidades de informação, que deve ser analisada e processada para se extrair conhecimento, utilizando técnicas adequadas e, atualmente, *software* especializado e algoritmos, de modo a produzir resultados eficazes e rápidos. Estes algoritmos já são fornecidos por um grande conjunto de *frameworks* e ferramentas disponíveis gratuitamente, um bom exemplo é a *framework* de inferência filogenética PHYLOViZ[23].

A maioria das técnicas de análise utilizadas na inferência filogenética tendem a formar, topologicamente, *pipelines* de trabalho - procedimentos constituídos por passos, cujos fluxos de dados são dependentes entre si. Apesar de ser possível executar *pipelines* de trabalho manualmente, como tem sido feito há várias décadas, atualmente, já não é factível, dado que os *datasets* utilizados são volumosos, tornando a sua análise manual contraproducente. A transição manual entre passos necessita também que haja interação humana para que cada passo receba os dados necessários, o que pode também estar sujeito ao erro humano. Por isso, foi construído *software* que reduzisse a interação humana e que automatizasse estes procedimentos. Este tipo de *software* é designado por sistemas de *workflow* - *software* que permite os utilizadores criarem *workflows*, através de uma *Domain-Specific Language (DSL)*[13], onde estes procedimentos são traduzidos para *scripts*, especificando-se o grupo de tarefas, com os seus parâmetros e dependências de dados.

Existem atualmente várias soluções de sistemas de *workflow*, que diferem na sua linguagem e estruturação de *workflows*, o que leva a que exista uma grande heterogeneidade de *software*, mas que piora também a partilha destes procedimentos. Por isso, quando se partilham *workflows*, é necessário despende-se tempo a traduzir *pipelines* de trabalho para a linguagem específica do sistema de *workflow* que vai executar a *pipeline* partilhada.

Este problema levou a que fosse criada a *Common Workflow Language (CWL)*[2] - um novo *standard* que permite executar *workflows* entre vários sistemas de *workflow*. No entanto, nem todos os sistemas suportam este novo *standard*.

Este projeto pretende construir uma *framework*, recorrendo a um projeto existente - PHYLOViZ e ao seu conjunto de ferramentas de inferência filogenética. Esta *framework*, permitirá ligar *frameworks* de inferência filogenética a sistemas de *workflow*, dando ao utilizador liberdade para construir os seus *workflows* personalizados, recorrendo à *framework* e às ferramentas do utilizador, fornecidas remotamente, que poderão ser geridas através de uma interface intuitiva. Tudo isto, fornecerá automatização de *workflows* e uma análise filogenética mais rápida e eficaz.

Este projeto foi financiado, no contexto de uma bolsa de estudo da Fundação para a Ciência e a Tecnologia (FCT) com referência UIDB/50021/2020, no projeto NGPHYLO PTDC/CCI-BIO/29676/2017 e num projeto do IPL - IPL/2021/DIVA_ISEL.

Palavras-chave: *Workflows*, *Frameworks* filogenéticas, Integração de *software*.

Contents

1	Introduction	1
1.1	Objective	3
1.2	Document Structure	3
2	Case Studies	5
2.1	Pipelines for phylogenetic analysis	6
2.2	Pipelines for genomics processing	7
3	State of the art	9
3.1	Workflow Systems	9
3.1.1	Airflow	10
3.1.2	Nextflow	13
3.1.3	Snakemake	15
3.1.4	Systems comparison	17
3.2	Web Frameworks	19
3.2.1	PHYLOViZ	19
3.2.2	NGPhylogeny	20
3.2.3	Other web frameworks	23
3.2.4	Observations	23
4	Requirements	25
4.1	Functional requirements	26
4.1.1	FLOWViZ Framework	26
4.1.2	Workflow System	26
4.1.3	Database	26
4.1.4	Phylogenetic framework	27

4.2	Non-functional requirements	27
4.2.1	FLOWViZ Framework	27
4.2.2	Workflow System	27
4.2.3	Database	27
4.3	Proposed solution	27
4.4	General use cases	28
5	Solution	31
5.1	System architecture	31
5.2	Domain model	34
5.3	Use case implementation	38
5.4	Server architecture	40
5.5	Client architecture	43
6	Implementation	45
6.1	Server	45
6.1.1	Used technologies	46
6.1.2	Authentication	46
6.2	Client	48
6.2.1	Used technologies	49
6.2.2	Data layer implementation details	49
6.2.3	Application overview	51
6.2.4	Tool integration	54
6.2.5	Workflow building	57
6.2.6	Result production	59
6.3	Workflow system	61
6.4	Deployment	64
6.4.1	HTTP server and React client	64
6.4.2	Phylogenetic framework integration	65
7	Conclusion	67
A	Airflow workflow	75

B	Nextflow workflow	77
C	Snakemake workflow	79
D	ETL pipeline	81
E	HTTP server's REST API endpoints	85

List of Figures

2.1	Representation of a pipeline	6
2.2	Representation of a Directed Acyclic Graph	6
2.3	Hamming distance → UPGMA algorithm workflow, from Phylolib	6
2.4	Hamming distance → goeBURST algorithm → LBR optimization workflow, from Phylolib	7
2.5	Trimmomatic → Spades → ABRicate workflow, from Flowcraft	7
2.6	Trimmomatic → Spades → Pilon → (ABRicate, Prokka) workflow, from Flowcraft	8
3.1	DAG representation of airflow’s example workflow script	13
3.2	PHYLOViZ Online: web application home page. Source: online.phyloviz.net 19	19
3.3	PHYLOViZ Online: phylogenetic tree real-time visualization, with the respective settings. Source: online.phyloviz.net/index/tutorial/	20
3.4	NGPhylogeny web application home page. Source: ngphylogeny.fr	21
3.5	NGPhylogeny: list of executing and schedule tasks. Source: NGPhylogeny’s documentation (ngphylogeny.fr/documentation#title13)	22
3.6	T-Rex web application home page. Source: trex.uqam.ca	23
4.1	System’s general functionality of the proposed solution	28
4.2	User’s general use cases and interaction with the framework	29
4.3	Framework’s general use cases and interaction with library and the workflow system	29
5.1	System architecture - FLOWViZ detail	32
5.2	System architecture - interactions among involved components	33
5.3	Tool domain model	35
5.4	Workflow domain model	37

5.5	User domain model	37
5.6	Tool integration module interaction model	38
5.7	Workflow building module interaction model	39
5.8	Result production module interaction model	40
5.9	Server's subsystem diagram	41
5.10	Client's subsystem diagram	43
5.11	Flowchart of a generic HTTP request	44
6.1	A generic and successful workflow of the JWT usage	47
6.2	User sign up (registration) with JWT strategy	47
6.3	User sign in (login) with JWT strategy	48
6.4	FLOWViZ: home page	51
6.5	FLOWViZ: documentation page	52
6.6	FLOWViZ: specific tool documentation	52
6.7	FLOWViZ: sign in and sign up buttons location (home page top-right corner)	53
6.8	FLOWViZ: sign in page	53
6.9	FLOWViZ: tool integration general fragment	54
6.10	FLOWViZ: tool integration access fragment	55
6.11	FLOWViZ: tool integration rules fragment	56
6.12	FLOWViZ: <i>whiteboard</i>	57
6.13	FLOWViZ: configuration of a task inside the workflow	58
6.14	FLOWViZ: workflow submission form	59
6.15	FLOWViZ: workflow list page	59
6.16	FLOWViZ: workflow execution log	60
6.17	FLOWViZ: workflow source code	61
6.18	ETL pipeline	62
6.19	ETL pipeline detail	62

List of Tables

2.1	Pipeline <i>versus</i> DAG	6
3.1	Workflow system comparison table	17

Listings

3.1	Downloading docker-compose.yaml	11
3.2	Starting Airflow services	11
3.3	Airflow mock workflow's script example	12
3.4	Nextflow mock workflow's script example	14
3.5	Snakemake mock workflow's script example	16
6.1	useFetch custom hook useEffect operation	50
6.2	Request function	50
6.3	Airflow development DockerFile	63
6.4	Docker network creation	63
6.5	Docker network inspect	64
6.6	React client production build command	64
6.7	Server using client static assets (flowviz.js)	64
6.8	Server path prefix	65
6.9	Client path prefix	65
6.10	FLOWViZ npm package installation	65
6.11	FLOWViZ linking to phylogenetic framework	65
6.12	FLOWViZ npm package required dependencies for framework extension	66

Acronyms

API Application Programming Interface. xi, 11, 18, 27, 34, 36, 39, 42, 46, 55, 61, 62, 67, 85

CLI Command Line Interface. 17, 27, 34, 36, 55, 56

CWL Common Workflow Language. v, vii, 2, 10, 15, 17, 18, 25, 26, 27, 68, 69

DAG Directed Acyclic Graph. xiii, xv, 5, 6, 10, 11, 12, 13, 36, 39, 40, 60, 61, 62, 63, 64, 67, 85

DCG Directed Cyclic Graph. 6

DSL Domain-Specific Language. v, vii, 2, 9, 14, 17, 28, 30, 36, 59, 60, 61, 62, 63, 67, 68, 69

FOSS Free and Open-Source Software. 45

GUI Graphical User Interface. 9, 11, 17

HPC High Performance Computing. 25, 26

HTML Hypertext Markup Language. 43

HTTP Hypertext Transfer Protocol. x, xi, xiv, 31, 32, 36, 39, 40, 41, 42, 43, 44, 45, 46, 48, 49, 50, 51, 59, 62, 64, 65, 66, 67, 85

JSX JavaScript XML. 50

REST Representational State Transfer. xi, 11, 18, 39, 46, 55, 61, 62, 67, 85

Glossary

allelic Comes from term *allele* - denotes the variant of a given gene.. 19

blob A file-like object of immutable, raw data. 67

Covid-19 a contagious disease caused by severe acute respiratory syndrome coronavirus 2 (SARS-CoV-2). 1

ETL pipeline An ETL pipeline is a set of processes to *extract* data from one system, *transform* it, and *load* it into a target repository. xi, xiv, 36, 39, 40, 61, 62, 64, 67, 81

genome The genetic information of an organism. 1, 7, 8

interoperability the basic ability of different computerized products or systems to readily connect and exchange information with one another, in either implementation or access, without restriction. 2, 25

loosely coupled Components which are connected via loose coupling - an approach to interconnecting the components in a system or network so that those components, also called elements, depend on each other to the least extent practicable. 2, 3, 22, 24, 25, 26

phylogenetic the systematic study of reconstructing the past evolutionary history of extant species or taxa, based on present-day data, such as morphologies or molecular information (sequence data). v, ix, xiii, 1, 2, 3, 6, 7, 19, 20, 23, 25, 26, 27, 28, 29, 31, 32, 33, 42, 45, 48, 51, 64, 65, 66, 67, 68

prokaryotic genome annotation a multi-level process that includes prediction of protein-coding genes, as well as other functional genome units such as structural RNAs and tRNAs. 8

scale-out adding more nodes to (or removing nodes from) a system, such as adding a new computer to a distributed software application. 14

scale-up adding resources to (or removing resources from) a single node, typically involving the addition of CPUs, memory or storage to a single computer. 14

Chapter 1

Introduction

Biology has been strongly influenced by the digital era we live in. The relationship between it and informatics had proven great humanitarian and scientific advances, mainly in the areas of healthcare and virology, not only contributing to create, but also enrich the science field of Bioinformatics. The need of delivering fast and efficient results and track different viruses and diseases has been fast-increasing, specially in the last two years with the ongoing Covid-19 pandemic and due to the continuously growing world population, which contributes to the appearance of new epidemics. Consequently, scientists rely on phylogenetic and genome analysis for variant tracking, in order to conceive effective medication.

Phylogenetic and genome analysis involve large quantities of data, that needs to be processed in order to be usable. Genome data can sometimes reach terabytes (TB) in size[24], this added to the multiple tasks which compose analysis procedures, can make these processes very demanding, complex and time-consuming. In pursuance of easing these tasks' requirements, software were made to better fit their specificities. Nowadays, these also use distributed and parallel computing to decrease processing times. There are already multiple free and available solutions that let users make this type of analysis, usually providing them with tools to build their phylogenetic trees. Solutions such as PHYLOViZ[23] and NGPhylogeny[17] are good examples of projects that have web frameworks which do this type of work.

These analysis procedures are composed by groups of tasks or steps, which are usually intrinsically dependent with each other - an output of a certain task may serve as an input of a future task. This pattern repeats until the final task's output - where the procedure ends. The procedure's structure and flow resembles a **pipeline** - a channel where a certain product is forwarded, suffering multiple transformations along the way, until reaching a final result. In Bioinformatics, this term was applied to these phylogenetic analysis procedures, along with the term **workflow**. This one can be interpreted as a *flow of work* that has a beginning and an end, which also relates with the firstly approached term. It is common to see these two terms being treated and used as they were the same. However, as it will be later explained in this report, a pipeline is a subset of a workflow, thus a pipeline can also be designated as a workflow. Informally, in Bioinformatics, it is common

to designate most workflows as pipelines.

Workflows are built and executed in **workflow systems** - specialized software, which provides a Domain-Specific Language (DSL), that allows users to script their own workflows and manage complex distributed computation and data in distributed resource environments. Nowadays, there are many available workflow systems and each one provides a different and, sometimes, unique way to build workflows. At first, this software diversity can bring many options to build workflows, however, workflow shareability worsens when users are building their workflows in their specific systems instead of a common one. In order to provide workflow shareability among users that use different workflow systems, the Common Workflow Language (CWL)[2] standard was created. This new standard contributed to an increasing interoperability between different workflow systems, however, as it is still new, a small percentage of them support this standard. It is expected that, given this standard increasing popularity, more implementations will be made in more workflow systems.

Most available phylogenetic web frameworks have a limited set of tools, without possible user integration, and almost none of them possess a workflow building mechanism nor uses a workflow system. From all the studied available frameworks, only one integrated a workflow system - NGPhylogeny[17], which is the only found example of a successful integration between a phylogenetic web framework and a workflow system. However, the implementation was only made to fit NGPhylogeny and does not provide any tool integration mechanisms, restricting users to the available tool set. This web framework will be detailed during chapter 3 - State of the art.

Given these facts, **FLOWViZ** - the framework introduced by this project, aims to provide users an environment that eases workflow building and shareability, by linking the phylogenetic web framework to a workflow system that schedules and executes users' workflows and by granting these characteristics:

- **Automation:** make phylogenetic analysis automatic processes, by providing users ways to, priorly, define work pipelines and not just individual tasks.
- **Flexibility:** allow users to use their own tools, since provided in containers or remote computing instances;
- **Scalability:** support large-scale analysis by relying on a workflow system that enables distributed and parallel computations on large clusters;
- **Result production:** provide complete results and logging, regarding the workflow's execution;
- **Integration:** implement a loosely coupled relationship between the web framework, the state-of-the-art phylogenetic tools and the workflow system. This allows better integration for other web phylogenetic frameworks that want to integrate this framework in the future, as for other workflow systems that might be implemented later;

- **Interoperability:** by allowing easy integration with other web phylogenetic frameworks and workflow systems, through defined contracts or interfaces;
- **Reproducibility:** supplying ready-to-use tools, which smooths the users' experience, by not require them to manage installations or tool dependencies on their personal computers; and workflow extraction which allows process reproducibility of the entire procedure.

At the end of this project, the framework will be tested and integrated with the PHYLOViZ web framework, version 2.0, via a contract, which will create a loosely coupled relationship between them. This will allow to not only better integrate FLOWViZ with the web framework, but will also allow other web frameworks' integrations possible in the future. The integration of FLOWViZ with PHYLOViZ 2.0 version will create the PHYLOViZ 3.0 version, as the first one will deliver tool integration and workflow building to PHYLOViZ.

The project was supported by funds, under the context of a student grant of Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020, for a INESC-ID's project - NGPHYLO PTDC/CCI-BIO/29676/2017 and a Polytechnic Institute of Lisbon project - IPL/2021/DIVA_ISEL. In the context of this project, three articles were also made and submitted to three different sources. The first one was submitted and presented at the INForum 2022 Conference*. The second one was submitted at ISEL Academic Journal of Electronics, Telecommunications and Computers (i-ETC)[†] and the third was submitted at arxiv.org [20].

1.1 Objective

Design an architecture and create a prototype of a framework that provides phylogenetic frameworks with tool integration and workflow building mechanisms, which will ease the process of binding phylogenetic frameworks with workflow systems. This is achieved by implementing contracts - interfaces that users need to comply with in order to integrate a great variety of tools through a single form. Integrated tools are then used by users to build customized workflows, which are scheduled and executed by the FLOWViZ's implemented workflow system. The FLOWViZ framework will be tested with the phylogenetic framework PHYLOViZ version 2.0 and the integration of these two will result in the generation of new PHYLOViZ version - version 3.0.

1.2 Document Structure

This document is divided by the following main chapters:

*https://inforum.org.pt/sites/default/files/2022-09/Actas_INForum.pdf#page=224

[†]<http://journals.isel.pt/index.php/i-ETC>

- **Introduction** - Provides the context of the problem, project's objectives and the document's structure;
- **Case Study** - A dedicated chapter presenting prior knowledge required to the state of the art's full comprehension;
- **State of the art** - Contains the information which was retrieved before the development of the project. Here are mainly stated comparisons between the different workflow systems and tools and is mentioned related work available online. It is also studied and concluded which workflow system is best for the project;
- **Requirements** - Shows the project's functional and non-functional requirements, the general use cases and the proposed solution;
- **Solution** - Displays and explains the system architecture and all associated models;
- **Implementation** - Presents the implementation of the solution and displays details about the developed applications;
- **Conclusion** - States final notes about the implementation and the obtained results; Discusses future improvements that could be implemented in this project.

Chapter 2

Case Studies

This chapter depicts the pipeline examples that were used for testing purposes in the context of this project.

Before the workflows' examples, it is relevant to clarify differences and details between workflows and pipelines, which were firstly approached during the introduction.

A workflow is a generic term to designate the automation of a process, which data is processed by different logical data, processing activities according to a specified set of rules [19].

Workflows can fall into two main categories: **business workflows** and **scientific workflows**. Although they are similar, they differ in some aspects:

1. **Abstract level:** business workflows take advantage of traditional programming languages, while scientific workflows use higher abstraction level tools to prove a scientific hypothesis[4];
2. **Interaction with participants:** In business workflows, data can be processed by different participants - computing instances or humans. In scientific workflows, data is processed only by computing instances, while the scientists are just required to monitor the workflow execution or control execution, when needed[4];
3. **Data flows and control:** Business workflows focus on procedural rules that generally represent the control flows, while scientific workflows highlight data flows that are depicted by data dependencies[32].

It should be mentioned that the workflows being used in this project are scientific.

During this research, it was also found that workflows only have two subsets: **Pipelines** and **Direct Acyclic Graphs** (DAGs) [19].

Their differences and representations [30, 19] are shown in the following Table 2.1.


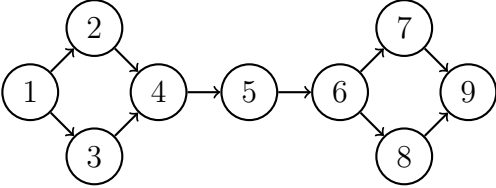
Pipeline	DAG
Linear Process	Non-linear process
Data flow does not branch	Data flow can branch
Data may come from a single source, usually the previous task's output	Data may come from multiple processes
Usually simple, quick and short-lived	Usually complex and long-lived
 <p>Figure 2.1: Representation of a pipeline</p>	 <p>Figure 2.2: Representation of a Directed Acyclic Graph</p>

Table 2.1: Pipeline *versus* DAG

DAGs are graphs where nodes represent tasks and edges represent data dependencies. This also applies to pipelines, however, they are described as more straight forward procedures, which graphical representation resembles a physical pipeline. Meaning that, like it was stated, these can not have multiple inputs or outputs and thus can not branch.

Although rarely approached, there is also a subtype of the DAG, which is a more complex version of it - a Directed Cyclic Graph (DCG)[30]. The only difference between this two is that the data flow can loop inside a DCG, whereas in a DAG that does not happen. The Direct Cyclic Graph is also more difficult to represent, as loops' iterations need to be represented, making its representation more complex. DCGs will not be focused much during this report, as they will not be used during the project's development.

2.1 Pipelines for phylogenetic analysis

This section shows workflows related with phylogenetic analysis, namely phylogenetic trees building. These two workflows are examples provided by a library of efficient algorithms for phylogenetic analysis - **Phylolib**[28]. Figure 2.3 presents the first workflow.

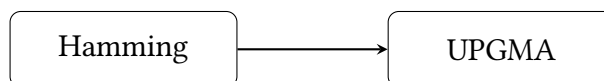


Figure 2.3: Hamming distance → UPGMA algorithm workflow, from Phylolib

The first task of this workflow is to calculate the **Hamming**[8] distance - obtain the number of positions at which two aligned genetic sequences differ. Using the output of

the calculated distance - a matrix, the next task - the **UPGMA**[29] (unweighted pair group method with arithmetic mean) algorithm will infer the phylogenetic tree.

Figure 2.4 displays the second workflow.



Figure 2.4: Hamming distance → goeBURST algorithm → LBR optimization workflow, from PhyloLib

This workflow also starts with the **Hamming** distance calculation, then the output matrix will be used as input of the **goeBURST**[14] algorithm - another algorithm used to build phylogenetic trees, that is an optimized implementation of the eBURST algorithm which identifies alternative patterns of descent for several bacterial species. Finally, the last step - **LBR**[28] (Local Branch Recrafting) optimization, will optimize the phylogenetic tree generated from the previous step.

2.2 Pipelines for genomics processing

This section displays workflows related to genome assembly. These workflows are examples provided by **Flowcraft*** - a pipeline assembler for genomics. This first workflow (Figure 2.5) has the following structure:



Figure 2.5: Trimmomatic → Spades → ABRicate workflow, from Flowcraft

This workflow starts with the tool **Trimmomatic**[7] - a trimming tool for next-generation sequencing data, which receives a pair of .fastq files, containing genomic data, and outputs two pairs of .fastq files with trimmed genomes. The next tool - **Spades**[3], will receive one pair of the resultant pairs and assemble the genome according to a given set of parameters, this will output .fasta files. The final tool - **ABRicate**[10], will use the produced .fasta and find antimicrobial resistance or virulence genes in the provided genomic data.

This document's appendices contains, as example, this pipeline, written for the three main workflow systems' Domain-Specific Languages (appendices A, B and C), which are going to be studied during chapter 3 - State of the art. Only these examples are available in the appendices, the rest is available through this repository*.

*<https://github.com/assemblerflow/flowcraft>

The second workflow uses some tools in common with the last workflow, however, it is more complex, as it forks at the end, as shown in Figure 2.6:

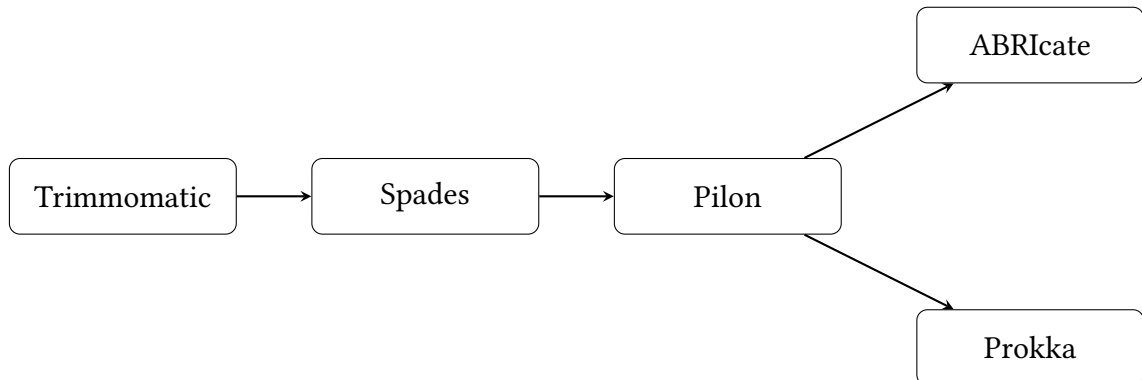


Figure 2.6: Trimmomatic → Spades → Pilon → (ABRicate, Prokka) workflow, from Flowcraft

Here are introduced two new tools: **Pilon**[31] and **Prokka**[27]. Their functionalities are the following:

- **Pilon**: an automated genome assembly improvement and variant detection tool;
- **Prokka**: rapid prokaryotic genome annotation.

For demonstration purposes, a public online repository* was made, where all the tested workflows and their requirements are available, provided with guides so that each one can be executed by each studied workflow system.

*github.com/mig07/Phylviz-Workflow-Examples.

Chapter 3

State of the art

This state of the art is composed by two main study components:

- The study of available workflow systems;
- The study of existing related web frameworks.

The state of the art will focus on three distinct workflow systems: Airflow[12], Nextflow[11] and Snakemake[16].

The previously demonstrated workflows, available in a GitHub's repository* and presented during chapter 2 - Case Studies, will be used here to study each workflow system. The usage of these workflows among this set of systems will provide detailed comparisons, which will be detailed during the study of each system and summarized by the end of this chapter, with some final observations. This provides a better overview, which links the project's objectives with these chapter's studied topics.

Available web frameworks will also be studied, in order to retrieve what solutions are available and how they could be improved. This way, important remarks can be taken, which will be considered in the project's requirements.

3.1 Workflow Systems

Workflow systems are specialized software which manage, not only workflows' schedules and executions, but also associated resources and distributed computation. They provide users ways to build workflows through a Domain-Specific Language (DSL), and some, through a specialized GUI, providing users with more configuration options, better workflow visualization and logging.

* github.com/mig07/Phyloviz-Workflow-Examples

As previously mentioned, there are many workflow solutions available, which offer different functionalities and standards. During this research, a curated list of these workflow systems, elaborated by *pditommaso*[†] - the co-founder of Seqera Labs[‡], was analyzed.

The listed systems were filtered based on their popularity and functionalities of interest. Consequently, the workflow systems which were studied are the following:

- **Airflow**[12][§];
- **Nextflow**[11][¶];
- **Snakemake**[16]^{||};

The following subsections 3.1.1, 3.1.2 and 3.1.3 will detail each enumerated workflow system, respectively.

3.1.1 Airflow

Overview

Airflow[12] is a scientific workflow system, written in Python, that enables scalable and reproducible data analysis. It started being developed by Airbnb^{**} and currently belongs to Apache^{††} as an open-source project.

Here, workflow scripts are designated as DAGs (Directed Acyclic Graph).

Its general features are the following:

- The unit of work here is a **Task**;
- Tasks usually call **Operators**, which are specialized objects that invoke specific functions or perform specific tasks (e.g., a PythonOperator can only invoke a Python function);
- DAGs can be executed by a defined schedule or by an external trigger;
- It can run on the cloud (Google Cloud Platform or Amazon Web Services) or inside a cluster;
- It supports CWL;

[†]<https://github.com/pditommaso/awesome-pipeline>

[‡]<https://seqera.io/about/>

[§]<https://airflow.apache.org/>

[¶]<https://www.nextflow.io/>

^{||}<https://snakemake.readthedocs.io/en/stable/>

^{**}<https://www.airbnb.com/>

^{††}<https://www.apache.org/>

- Offers a GUI through a Web Application, that executes when running the Airflow services;
- Creates a detailed log of the executed DAGs, with the execution timestamps and stack traces;
- Provides a ready-to-use REST API, that can receive external triggers or supply complete logs regarding workflows and their executions.

Installation and prerequisites

Airflow can be installed locally or can be executed inside containers, using the provided `docker-compose.yaml` that provide the minimum required services with a minimal configuration.

Airflow requires Python as its execution environment (version 3.7 up to 3.10). It requires docker and docker-compose if running the containerized version.

Installation observations

Docker-Compose must be installed alongside with Docker, in order to execute the Airflow services.

After this, the `docker-compose.yaml` - the file where each Airflow-related service is defined, can be downloaded and executed, using the following commands, respectively:

Listing 3.1: Downloading docker-compose.yaml

```
1 curl -LfO 'https://airflow.apache.org/docs/apache-airflow/2.2.4/
  docker-compose.yaml'
```

Listing 3.2: Starting Airflow services

```
1 docker compose up
```

In this state of the art, it is used the containerized version of Airflow. With the version being used, there are some precautions that must be taken, regarding the usage of other containers by the container where Airflow services are running. By default, the containerized version of Airflow can not access the host's Docker daemon and thus can not call other containers. This happens because the user created for Airflow does not have root permissions. To solve this, there are two possibilities:

- Change the Airflow user to the root user, less desirable in terms of security;
- Add the Airflow user to the sudoers group, which provides more configurability in some security aspects.

Scripting

In Listing 3.3 there is an example of an Airflow DAG structure.

Listing 3.3: Airflow mock workflow's script example

```
1 # Imports
2 from airflow import DAG
3 from datetime import datetime, timedelta
4 from airflow.providers.docker.operators.docker import DockerOperator
5 from airflow.operators.bash import BashOperator
6
7 from docker.types import Mount
8
9 # Default Arguments
10 default_args = {
11     'owner'           : 'owner',
12     'description'     : 'Example',
13     'depend_on_past'  : False,
14     'email_on_failure': False,
15     'email_on_retry'  : False,
16     'retries'         : 1,
17     'retry_delay'     : timedelta(minutes=5),
18     ...
19 }
20
21 # DAG definition
22 with DAG('Mock-Example-workflow', default_args=default_args,
23         schedule_interval="5 * * * *", catchup=False) as dag:
24
25     dockerOpOne = DockerOperator(
26         task_id='dockerOpOne',
27         image='...',
28         api_version='auto',
29         mounts=[Mount(target='...', source='...', type='bind')],
30         command='...',
31         auto_remove=True,
32         docker_url='unix://var/run/docker.sock',
33         network_mode='bridge'
34     )
35
36     dockerOpTwo = DockerOperator(
37         task_id='dockerOpTwo',
38         image='...',
39         api_version='auto',
40         mounts=[Mount(target='...', source='...', type='bind')],
41         command='...',
42         auto_remove=True,
43         docker_url='unix://var/run/docker.sock',
44         network_mode='bridge'
45     )
46
47     bashOpOne = BashOperator(
48         task_id='bashOpOne',
```

```

48     bash_command='...',
49 )
50
51 bashOpTwo = BashOperator(
52     task_id='bashOpTwo',
53     bash_command='...',
54 )
55
56 # Execution order definition
57 dockerOpOne >> dockerOpTwo >> [bashOpOne, bashOpTwo]

```

As the workflow depicted in the previous Listing 3.3 forks at its end, and it is harder to visualize than the others, the following Figure 3.1 represents it graphically.

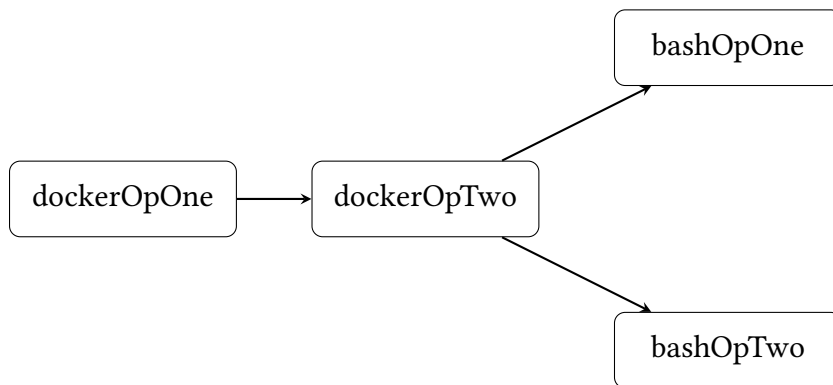


Figure 3.1: DAG representation of airflow's example workflow script

Contrary to other studied workflow system, Airflow allows users to explicitly specify the execution order, which is a great advantage, specially with workflows that have tasks with multiple inputs or outputs.

3.1.2 Nextflow

Overview

Nextflow[11] is a scientific workflow system which enables scalable and reproducible scientific workflow, focused on using software containers. It was developed by Seqera Labs^{‡‡}. The unit of work inside a workflow is labeled as a *process*. Multiple processes can be defined with inputs and outputs, inside a script. The sequential order of processes is implicitly inferred by their inputs and outputs' usage of each adjacent process.

The features that Nextflow provide are the following:

- **Reproducibility:** supports Docker[22], Singularity[15] and integrates with GitHub - workflows and pipelines can be executed from an online repository;

^{‡‡}<https://seqera.io/about/>

- **Portability:** provides an abstraction layer which enables the workflows and pipelines to be executed on multiple platforms;
- **Stream oriented programming:** Nextflow took inspiration in the Unix pipes* model and uses it in its fluent DSL, allowing the user to handle complex stream interactions easily;
- **Parallelism:** developed application are inherently parallel, providing the ability to scale-up or scale-out a system with transparency;
- **Statefulness:** having continuous checkpoints along a workflow's execution, making it possible to resume one if it stopped;
- **Fast Prototyping:** multiple programming languages can be used inside a script to configure each process.

Installation and prerequisites

The system **prerequisites** are:

- **Operating system:** Any posix system (Linux or macOS), Windows is supported through WSL;
- **Execution environment:** Java 8 or later, up to version 16. Java 11 LTS is recommended;
- **Others:** Bash 3.2 or later.

Installation observations

Of all the tested workflow systems, Nextflow has the simplest installation, consisting of a single executable file that can be added to the system path and invoked anywhere when executing `.nf` type files.

Scripting

A Nextflow script, shown in Listing 3.4, has the following structure:

Listing 3.4: Nextflow mock workflow's script example

```

1  #!/usr/bin/env nextflow
2
3  params.saveMode = '...'
4  params.filePattern = '...' // Files directory, can be override by
   program arguments

```

*[en.wikipedia.org/wiki/Pipeline_\(Unix\)](https://en.wikipedia.org/wiki/Pipeline_(Unix))

```

5     params.resultsDir = '...' // Results directory
6
7     // A data channel for program arguments
8     Channel.fromPath(params.filePattern)
9         .set { ch_in }
10
11    process proc1 {
12        // Where the script will output its results
13        publishDir params.resultsDir, mode: params.saveMode
14
15        input:
16            ...
17        output:
18            ...
19        script:
20            """
21            ...
22            """
23    }
24
25    process proc2 {
26        ...
27    }

```

3.1.3 Snakemake

Overview

Snakemake[16] is another scientific workflow system, written in Python, that enables scalable and reproducible data analysis. It was developed by Johannes Köster.

Here are presented its general features:

- The work unit is labeled as a *rule*;
- Inside a script, code can be written in these languages: Python, R, Julia, Jupyter Notebook, Rust and Shell, which eases some configurations;
- Snakemake introduced Wrappers, which provide a ready-to-use environment to execute the most popular tools;
- It supports CWL;
- It has in-built script linting;
- Containerization is supported, however, it has some limitations when using local containers.

Installation and prerequisites

The system **requirements** are:

- **Operating system:** Any posix system (Linux or macOS), Windows is supported through WSL and Vagrant;
- **Execution environment:** Python 3.5 or later;
- **Others:** BWA, SAMtools, Pysam, BCFtools, Graphviz, Jinja2, NetworkX and Matplotlib, **which can be installed through MambaForge.**

Installation observations

The documentation advises to install MambaForge to install Snakemake and its requirements. After this, a Conda environment can be set up and activated through a configured terminal, so that snakemake can be executed.

Scripting

A Snakemake script has the structure shown in Listing 3.5, which resembles the topology of a Nextflow script with syntax differences.

Listing 3.5: Snakemake mock workflow's script example

```
1 rule r1:
2   input:
3     ...
4   output:
5     ...
6   shell:
7     """
8     ...
9     """
10
11 rule r2:
12   ...
```

3.1.4 Systems comparison

Table 3.1 summarizes the main differences between each workflow system.

Characteristics	Workflow System		
	Airflow	Nextflow	Snakemake
Base Language	Python	Java / Groovy	Python
Composition Style (Values: Script, GUI)	Script	Script	Script
Execution Style (Values: CLI, GUI)	GUI / CLI	CLI	CLI
CWL Support (Values: Yes, No)	Yes	No	Yes
Containerization Support (Values: Yes, No)	Yes	Yes	No
Execution Order (Values: Explicit, Implicit)	Explicit	Implicit	Implicit
Dependency Order (Values: Explicit, Implicit)	Explicit	Implicit	Implicit
Workflow Sharing (Values: Yes, Partial, No)	Yes	Partial	Partial

Table 3.1: Workflow system comparison table

The **Base Language** states the language in which the workflow system is based on and, also, the execution environment. It is preferred that the language for the chosen workflow system is widely known and provides good and complete documentation.

The **Composition Style** describes if workflows in a certain workflow system can be built via Script or Graphical User Interface. For the framework, it is preferred a system that offers script composition, so the framework can compose workflows internally with the appropriated DSL. Some systems also allow external dependencies, which can provide more composition style features.

The **Execution Style** describes if workflows are executed via Command Line Interface or by a provided Graphical User Interface. For the framework's implementation, it is preferred the CLI style, as the GUI one might abstract features or even make the framework's workflow execution impossible.

The **CWL Support** indicates if the workflow system supports the Common Workflow Standard. It is preferred that the chosen workflow system has this feature to provide a superior workflow shareability among users that user different systems.

Containerization Support states that a workflow system fully supports containers and containerized executions. For the framework being developed, it is crucial that the used system has this feature, so users can provide external tools to build their workflows.

Execution Order indicates if the execution of the tasks composing a workflow are

implicitly inferred by the system or can be explicitly defined by the users. Although the implicit inference orders the execution by order the tasks are written, it is preferred that a workflow system offers explicit execution order, in order to provide more control to the end-user.

The **Dependency Order** indicates if the workflows' dependencies are implicitly inferred by the system or if the user can explicitly define them. As the last characteristic, it is preferred that the chosen system provide explicit dependency order, so the users have more control over their workflows' data dependencies.

Workflow Sharing describes the ability of a workflow system to maintain and save the execution states of certain workflows, so other users can execute them from a determined point of the procedure, without needing to restart them. For the framework, it is preferred that the workflow system offers this feature, to provide better shareability and superior error tolerance.

Results from Table 3.1 show that only Airflow and Snakemake have CWL support, which is a characteristic of interest. Nextflow also has a CWL parsing tool*, however, the project's maintenance has been on hold for years, and it was left in its experimental phase. Because of this, it was considered that Nextflow does not have a proper Common Workflow Language support.

Airflow also supports another characteristics of interest, such as: explicit execution, explicit dependency order, CWL[†] and containerization which are key features, being these the primary reasons why **Apache Airflow** was chosen to be the workflow system for the FLOWViZ implementation. The supplied ready-to-use Airflow REST API is also a great advantage, as it provides another way to interact with the workflow system externally.

*github.com/nextflow-io/cwl2nxf

[†]During the development of this project, integrating the Airflow's CWL plugin was not possible due to the plugin's outdate and some incompatibilities with the project's objectives and requirements.

3.2 Web Frameworks

This section states, primarily, the related work which was found during this research and that share common objectives with this project.

PHYLOViZ online^{§§} will be the first studied web framework, as it does not provide mechanisms that allow workflow building and execution, and it is the framework which was planned to integrate FLOWViZ with.

During this research, a list of other available solutions was also found^{¶¶}. As this list gives a wide range of results and, most applications / frameworks are focused on one tool. Due to this, only the relevant ones with common objectives were studied in this section.

3.2.1 PHYLOViZ

PHYLOViZ project started in 2012, by providing a Java platform that allowed sequence-based typing methods' analysis, which generate allelic profiles and their associated epidemiological data.

Later, in 2016, PHYLOViZ Online[26] was released: the online platform for the PHYLOViZ project (Figure 3.2).

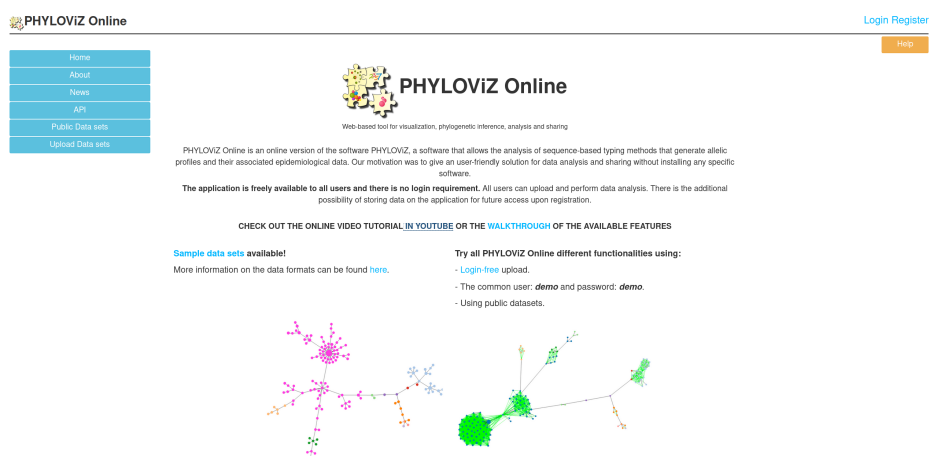


Figure 3.2: PHYLOViZ Online: web application home page. Source: online.phyloviz.net

This web application provides ways to build phylogenetic trees through a user-friendly interface - the user starts to input its data and specific parameters and then a phylogenetic tree is graphically generated, along with a set of settings that allows the tree's manipulation in real-time (Figure 3.3).

^{§§} online.phyloviz.net/

^{¶¶} molbiol-tools.ca/Phylogeny.htm

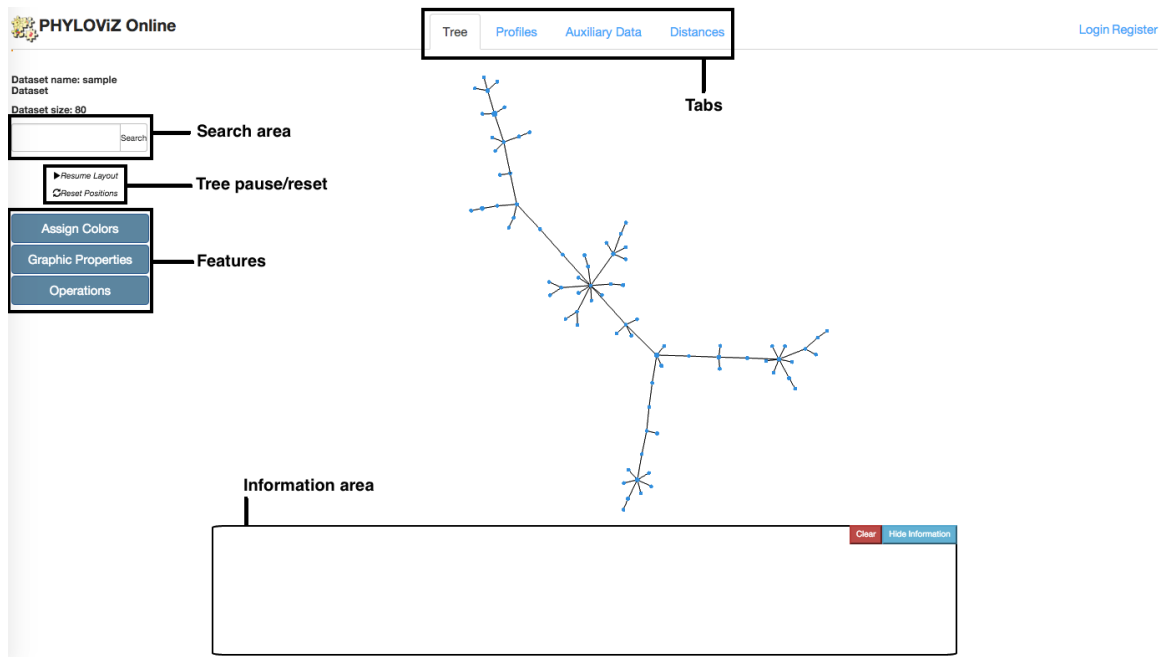


Figure 3.3: PHYLOViZ Online: phylogenetic tree real-time visualization, with the respective settings. Source: online.phyloviz.net/index/tutorial/

Besides tree visualization, PHYLOViZ also provides other functionalities, such as: **distance matrix visualization** and **sequence visualization**.

Although PHYLOViZ provides the necessary tools to successfully elaborate the phylogenetic analysis, it does not provide **workflow building**. Consequently, the user can only prepare and execute one task at a time. Because of this, the user needs to wait for each task's completion, in order to configure and execute the following task.

As previously stated, this project will test and integrate FLOWViZ with this web framework, which will solve this issue. The integration will allow the user to, priorly, prepare all the tasks and execute them by the order specified in the workflow, without needing to manually configure each one at a time.

3.2.2 NGPhylogeny

NGPhylogeny[17] is a web application for phylogenetic analysis, being the second web framework to be studied.

NGPhylogeny started in 2008, previously called Phylogeny, and had its last iteration in 2019, providing us the web application depicted in (Figure 3.4).



Figure 3.4: NGPhylogeny web application home page. Source: ngphylogeny.fr

This web framework is a great case study, as it implemented a **workflow system** to allow workflow building and execution with the available tools. Because of this, it contains many topics of interest that can be used to better substantiate FLOWViZ's requirements and features.

NGPhylogeny offers three main ways to build workflows or pipelines:

- Using pre-made workflows with default values;
- Using pre-made workflows with users' values;
- Building their own workflows with users' values, using the provided tools. This way is labeled as "*à la carte*" workflows, because the users have freedom to choose and configure each tool of the workflow.

This web application uses Galaxy[1] as the workflow system to run the workflows. More specifically, they are executed inside the Pasteur institute's Galaxy Cloud Service instance[21]. The usage of a workflow system provides very relevant advantages:

- **Scheduling:** By using a workflow system, all tasks can be priorly configured and executed at once;
- **Scalability:** Not only computing resources can be scaled-out to serve more users and execute more workflows simultaneously, but also more tools can be added to all users, in order to better fit their needs;
- **Elasticity:** Computing resources can be allocated and better adapted in function of the current workload.

These advantages are possible in this project because, contrary to PHYLOViZ, this one relies on a workflow system for task scheduling and easy computing instance management, not only making the final product practical to the end-user, but also reliable in terms of distributed computing and task automation.

The example depicted in the Figure 3.5 shows a list of scheduled tasks and their execution status.

Tool	Step	File Name	Status
Newick Display	7.	PhyML Newick tree: BMGE Cleaned sequences Phylip.nhx.svg	...
PhyML	6.	PhyML Newick tree: BMGE Cleaned sequences Phylip.nhx	...
	5.	PhyML statistic: BMGE Cleaned sequences Phylip.stats.txt	...
	4.	PhyML log: BMGE Cleaned sequences Phylip.log	...
BMGE	3.	BMGE Cleaned sequences Phylip	...
MAFFT	2.	MAFFT on data 1	...
Upload File	1.	t.phy	✓ .phylip

Figure 3.5: NGPhylogeny: list of executing and schedule tasks.

Source: NGPhylogeny’s documentation (ngphylogeny.fr/documentation#title13)

However, beyond these great advantages, users can only build workflows with the provided tools and parameters, as the application does not allow tool integration.

These are problems that FLOWViZ aims to tackle, by allowing users to specify their own external tools and by building an extensible framework, which provides an easy implementation with other available web frameworks. This can be done by defining contract or interface-based relationships between system components, contributing for loosely coupled relationships between them.

3.2.3 Other web frameworks

Another example is **T-Rex**[6] - a web application focused on phylogenetic tree building and visualization.

T-Rex is a project that started in 2001 with native applications for Windows and macOS. Its latest iteration provided the project with a web application in 2012, which is the target of study.

Contrary to NGPhylogeny, T-Rex and its underlying tools are running on a single web server, and thus does not provide workflow building mechanisms. However, it provides other set of tools that NGPhylogeny does not possess.

In T-Rex, the user can also specify its own values, but it is not clear that this application provides default values, as some tools have pre-generated values, but other tools do not pre-generate any values when invoked.

Below it is shown an overview of the web application (Figure 3.6).

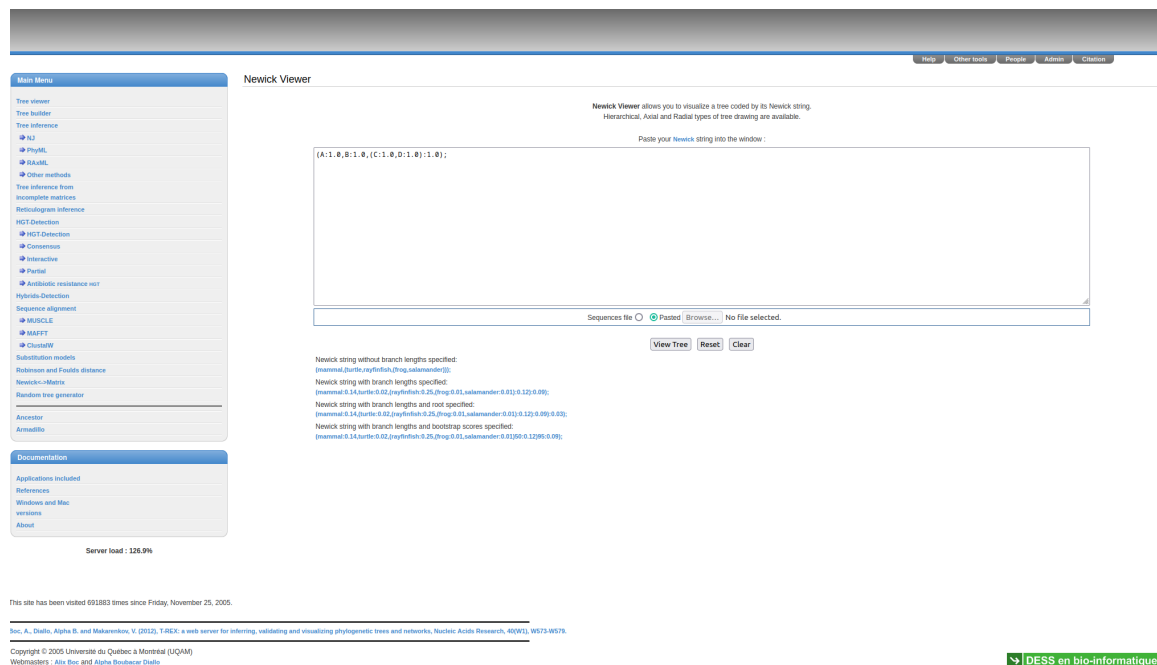


Figure 3.6: T-Rex web application home page. Source: trex.uqam.ca

Other solution which is worth mentioning is **iTOL**[18], as it shares similarities with T-Rex. iTOL stands for *interactive Tree Of Life*, and it is a tool, provided by a web application, for tree visualization and annotation.

3.2.4 Observations

More tools and applications from the encountered list could be explored, however, most of them would fall out of the scope of this research - study web frameworks that share

similar objectives with this project and that can help improve the project requirements and features.

From all the studied applications, NGPhylogeny shared most of its objectives, as this project aims to develop a similar web framework, however, with some improvements:

- Providing the user more ways to build its own workflows, by allowing integration with other libraries and by letting the user call external containers or remote computing instances to execute certain tasks.
- Framework extensibility - enable other web frameworks to integrate FLOWViZ, by creating loosely coupled relationships between system components, based on contracts and interfaces.
- Common workflow language support;

NGPhylogeny can serve as great example for this project's development, as it is the only found web framework which uses a workflow system. FLOWViZ aims to have the same advantages as NGPhylogeny, while providing the mentioned improvements. These will be stated again with more detail in chapter 4, where the project's requirements, general use cases and features will be discussed.

Chapter 4

Requirements

This chapter contains the project's requirements, the general proposed solution and general use cases.

As previously stated in the objective, the end goal of this project is to build a framework that will serve as a component of an already existing project, with state-of-the-art tools - PHYLOViZ. This framework's premise is to provide the user ways to build custom workflows and work pipelines, by providing phylogenetic tool integration through contracts, which will result in a great single point of interoperability among different tools. Users can then use integrated tools to build their customized workflows.

Tools can be integrated with FLOWViZ, since provided in a specific runtime environment that is deployed and accessible online. These runtime environments can be containers or remote computing instances inside a cluster, which are used to host and execute specific tools for a user's workflow. This way, contrary to many studied state-of-the-art frameworks, the user will not be limited to a specific tool set, supplied by the phylogenetic framework, and will have total control over its integrated tools.

This is a task which will be accomplished by the used workflow system, that must have full containerization support and be able to execute on the Cloud / HPC.

Contract based relationships or loosely coupled relationship are not only limited to tool integration. As such, FLOWViZ aims to be an extensible framework, meaning that it can be easily integrated by another phylogenetic framework that suffice the base requirements, it will also expose a contract that needs to be fulfilled, in order to successfully integrate it with the phylogenetic framework.

Another important feature is Common Workflow Language. This is considered another source of interoperability, as it allows users to share workflows, that could be originated by different workflow systems, using different domain-specific languages through a single standard, which provides a superior shareability. As mentioned in the final footnote of the previous chapter 3, the Airflow's CWL plugin was not integrated due to the plugin's outdate and also some incompatibilities with the project's objectives and requirements.

The next sections will list both project's functional requirements and non-

-functional requirements, as a general proposed solution for the system to be developed with the associated general use cases.

4.1 Functional requirements

This subsection lists the project's functional requirements.

4.1.1 FLOWViZ Framework

- Allow the user to integrate customized tools via contracts, by specifying custom runtime environments, such as containers or remote computing instances, that host the phylogenetic tool;
- Allow the user to build customized workflows using tools from the library tool set and integrated ones;
- Present results regarding the workflows executions, where logs, workflow generated source code and task details can be easily accessed;
- Framework extensibility: create a framework that can be easily integrated with another phylogenetic framework, via contracts and loosely coupled relationships.

4.1.2 Workflow System

- Workflow execution with prior scheduling;
- Containerization support;
- Result production.
- Cloud / HPC execution support;
- Allow both implicit and explicit execution order;
- Interfaces and contract-based relationship with other system components, in order to enable implementations from other workflow systems;
- Common Workflow Language compatibility.

4.1.3 Database

- Save tool contracts;
- Save workflows' configurations;
- Save users' data.

4.1.4 Phylogenetic framework

- Expose the phylogenetic tools via Command Line Interface (CLI) or Application Programming Interface (API).

4.2 Non-functional requirements

This subsection lists the project's non-functional requirements.

4.2.1 FLOWViZ Framework

- Export workflows to the supported domain-specific language;
- Export a workflow's script, written with supported workflow's domain-specific language, to the CWL format.

4.2.2 Workflow System

- Automatic container download and execution from a container repository;
- Execute scripts hosted in a code repository, such as GitHub or GitLab.

4.2.3 Database

- Data redundancy;
- High availability, when deployed.

4.3 Proposed solution

This section shows the general diagram of the proposed solution. During the next section, the interactions between this diagram entities will be divided and detailed into general use cases.

The general functionality of the system, depicted in Figure 4.1, starts with the user integrating a customized phylogenetic tool with the FLOWViZ framework. The user integrates the tool by filling the tool contract and, then, submitting the contract, which will go through a validation before being saved into the system. If the validation passes, the tool contract will be saved and the tool will be successfully integrated.

Finished the necessary user's integrations, the user can now build workflows. This is achieved by letting it draw the workflow and the associated data flow with the provided

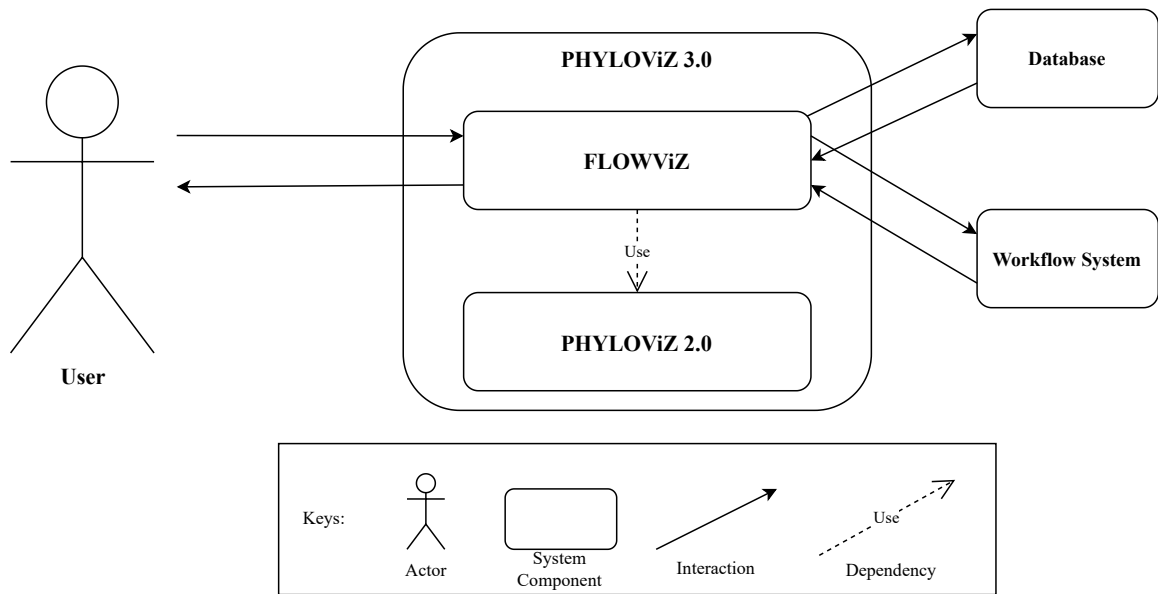


Figure 4.1: System's general functionality of the proposed solution

framework editor and the previously integrated phylogenetic tools. The tool contracts will be useful in this step: as each task in a workflow represents an integrated tool, the user needs to configure each task according to each tool rules or guidelines, which are specified in the previously filled tools contracts. Using the specified rules, the user can configure the workflow and each involved tool with great control, while avoiding configuration errors. When the user finishes the configuration, the workflow can be submitted. After the workflow's submission, the framework will also validate the submitted workflow and notify the workflow system that there's a new workflow that will be executed at a specific time and date, and dynamically generate a workflow DSL script, according to the submitted workflow. The workflow will then execute at the specified date and time.

After the workflow's execution, the user can fetch the results, the associated generated data and retrieve output generated files from its tools or from the framework result page.

Figure 4.1 also depicts the creation of a new PHYLOViZ version - PHYLOViZ 3.0. As it is planned to integrate FLOWViZ with PHYLOViZ, this integration will result in a new version of the latter one, as FLOWViZ is supplying it with workflow building and execution mechanisms.

More details regarding the system solution and architecture will be approached during chapter 5 - Solution.

4.4 General use cases

This section shows the general use cases between the user and the framework and between the framework and the workflow system, in order to better understand the interactions between the system components, introduced in the last section (section 4.3).

(Figures 4.2 and 4.3)

The general use cases that happen between the user and the framework are depicted in Figure 4.2. As explained during the last section (section 4.3), the user is able to (i) integrate its customized phylogenetic tools, since provided inside a container or a remote virtual machine that can be accessed by the FLOWViZ framework; (ii) build and schedule customized workflows using the previously integrated tools; (iii) obtain results, logs and output data originated from the user workflows; (iv) check the user workflows, manage them and retrieve their execution status.

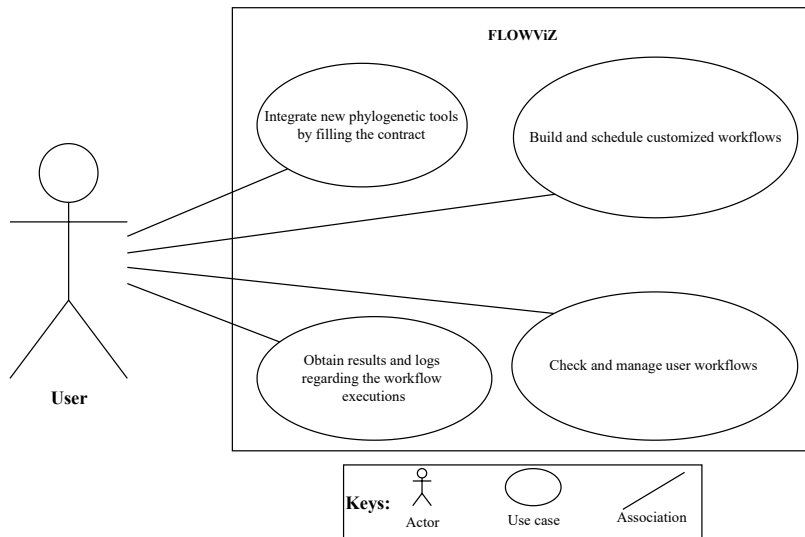


Figure 4.2: User’s general use cases and interaction with the framework

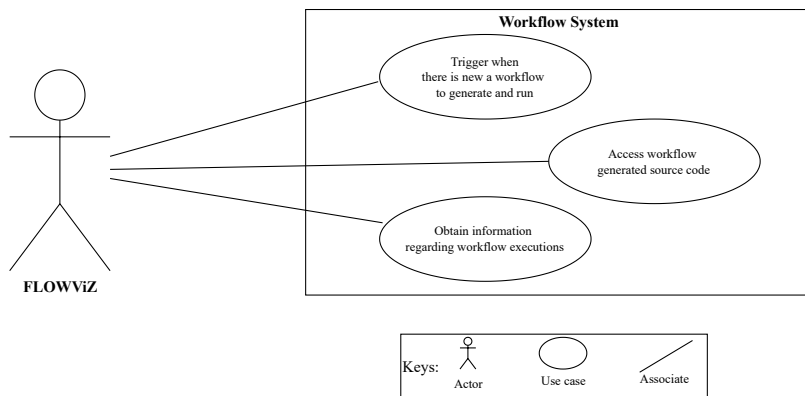


Figure 4.3: Framework’s general use cases and interaction with library and the workflow system

The general use cases between the framework and the workflow system is displayed by Figure 4.3. FLOWViZ framework only interacts with the workflow system to (i) notify the workflow system when there is a new user workflow to be retrieved and parsed at a specified time and date; (ii) access workflow generated source code, as the workflow

system parses the workflow requests to its own DSL; (iii) obtain information and logs about workflow executions.

Chapter 5

Solution

This chapter presents the implemented solution of this project, namely, the implemented system architecture, the domain model and use cases' implementation.

5.1 System architecture

The final goal of this project is to develop an integration framework, that can also be easily integrated with other frameworks or phylogenetic tools. FLOWViZ main goal is to act as a middleware between the phylogenetic tool or framework and the workflow system and provide workflow scheduling and execution for phylogenetic frameworks or tools that lack this mechanism. A previous work called NGSPipes[9] already implemented part of this project's logic, when it comes to tool annotation and integration, however, the annotations were more detailed and it was not made to be an extensible framework.

Figure 5.1 details the FLOWViZ component of the global architecture. FLOWViZ is mainly composed by a React Web client and an HTTP Express Server. Both client and server use and provide the three displayed functional modules: the *(i) tool integration*, *(ii) workflow building* and *(iii) result production* modules, which allow users to integrate new phylogenetic tools, build workflows with them and retrieve results from workflow executions, respectively.

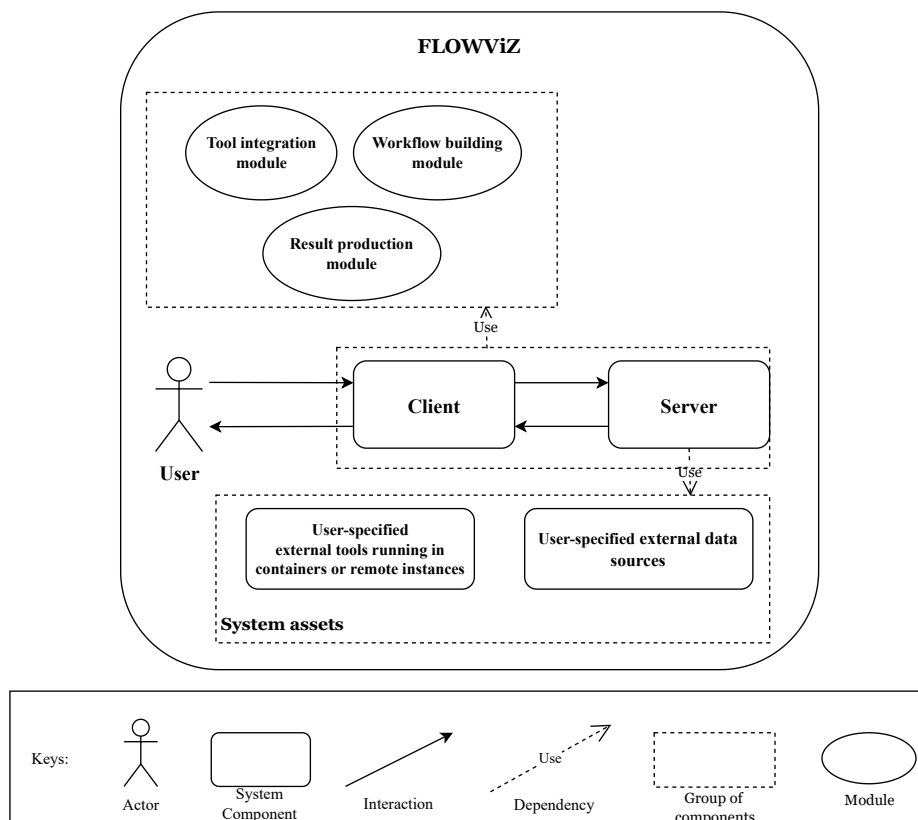


Figure 5.1: System architecture - FLOWViZ detail

The external database, depicted in the previous Figure 4.1, stores tool contracts, submitted workflows and user credentials. Tool contracts and workflows contain information which can point out to external tools (hosted by containers or remote computing instances) and data sources, shown inside the *System assets* system component. This is a core requirement of the FLOWViZ architecture, as tool integration requires users to expose their phylogenetic tools, by using docker containers or virtual machines, and specify in the tool contract how the framework can invoke them. In short, the database stores collections of pointers that point to external tools or data, in order to allow the framework and the workflow system to access them for workflow building and execution.

The workflow system interacts both with the database and the HTTP server. Airflow, the previously selected workflow system in subsection 3.1.4, retrieve information from the database, in order to dynamically create and execute users' workflows, and it provides the server with information regarding workflows' executions, which latterly the server will deliver to the client.

For a better overview, Figure 5.2 shows the interactions among all the components of the system.

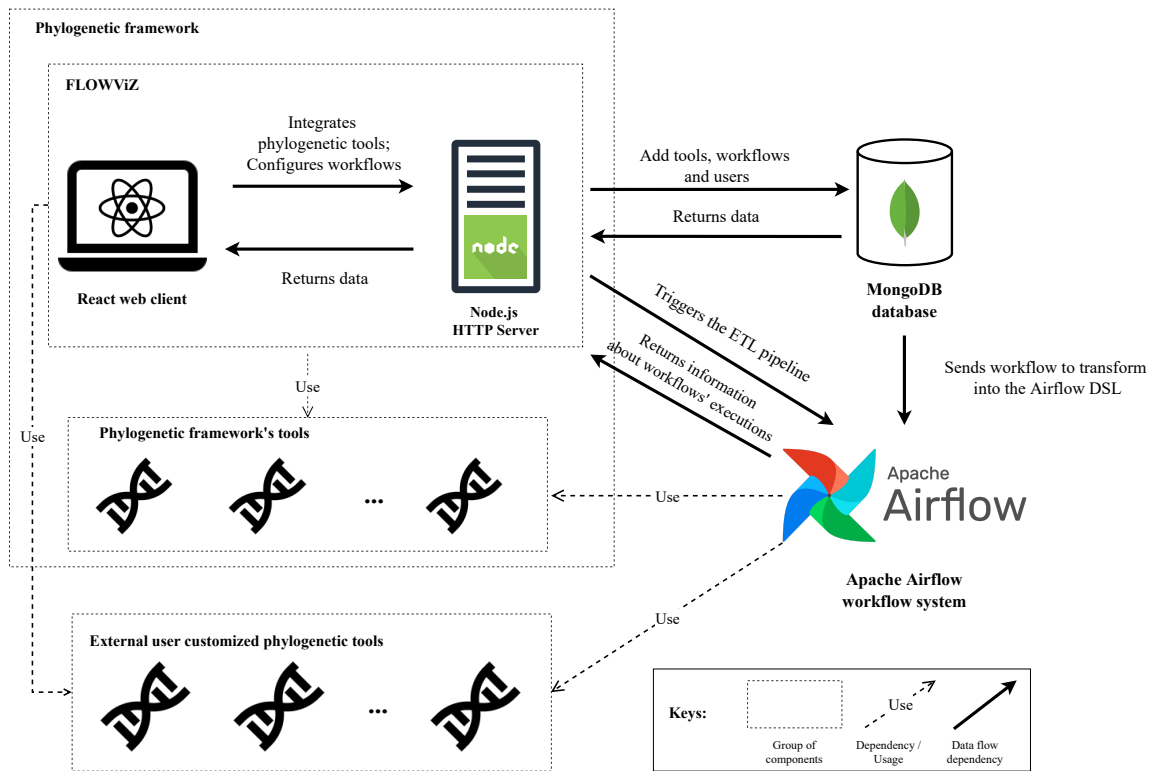


Figure 5.2: System architecture - interactions among involved components

In the depicted Figure 5.2, FLOWViZ is shown as an internal module of the phylogenetic framework. FLOWViZ is intended to be integrated with the phylogenetic framework via an established contract. If the latter one has common core dependencies, such as the express server dependency, the express instance can be shared with FLOWViZ, if specified in the contract. This way, FLOWViZ will behave as an *extension* or a *plugin* of the phylogenetic framework. There are essentially **two** types of contracts in this solution: the contract in which the Phylogenetic framework shares common dependencies and that allows its extension with FLOWViZ; the contracts that allow phylogenetic tools, either from the phylogenetic framework or external ones, to be specified and integrated with FLOWViZ.

FLOWViZ will also be able to be integrated by frameworks that do not possess common dependencies or technologies, however, the deployment of FLOWViZ will have to be standalone, meaning that it can not be integrated as an internal module and share object instances, as depicted in Figure 5.2, but it will be an external module that communicates with the phylogenetic framework and its tools over the network.

More details regarding the solution's deployment and the contract between FLOWViZ and the phylogenetic tool will be described during section 6.4.2.

5.2 Domain model

There are three main domain models: tool, workflow, user.

Tool model

The tool model represents the contract of the integrated tool.

This model is composed by three main properties:

- **general**: general information about the tool, such as name and description;
- **access**: information that specifies how the tool can be accessed by FLOWViZ and the workflow system;
- **rules**: the rules and guidelines to correctly invoke and use the tool.

As there are different types of tools, the tool model has two subtypes:

- **api**: a tool which functions are available through API endpoints;
- **container**: a tool that is available inside a specified container and provides a CLI.

These subtypes were considered the most generic to integrate users' tools, as most tools provide a Command Line Interface (CLI) or are APIs that provide endpoints to perform different operations. It should be noted, that this domain model and its properties can be extended in the future, in order to accommodate more tools and their respective data. Figure 5.3 represents the tool domain model diagram.

The diagram in Figure 5.3 shows the General and the Access main properties, however, the Rules property is represented by both CommandGroup and Endpoint, as it was not needed to create an intermediary model.

When the tool is a library, it must be hosted by a container to be integrable with FLOWViZ. This is why there are properties which are only for containers - all the properties in LibraryAccess that have the "docker" prefix. Although Docker was used to test and integrate tools, more container engines can be used for tools' integration, since they suffice the minimum requirements to fill this contract main property.

When a library tool is supplied, the user must only supply CommandGroup models, which contains Command models, this way the user can easily specify the tool's command *tree* and establish the command hierarchy. If the user supplies an API tool it can only configure it with Endpoint models.

The CommandGroup sub-properties and their functions are the following:

- The invocation property allows the user to specify an alias to invoke a specific command group or command.

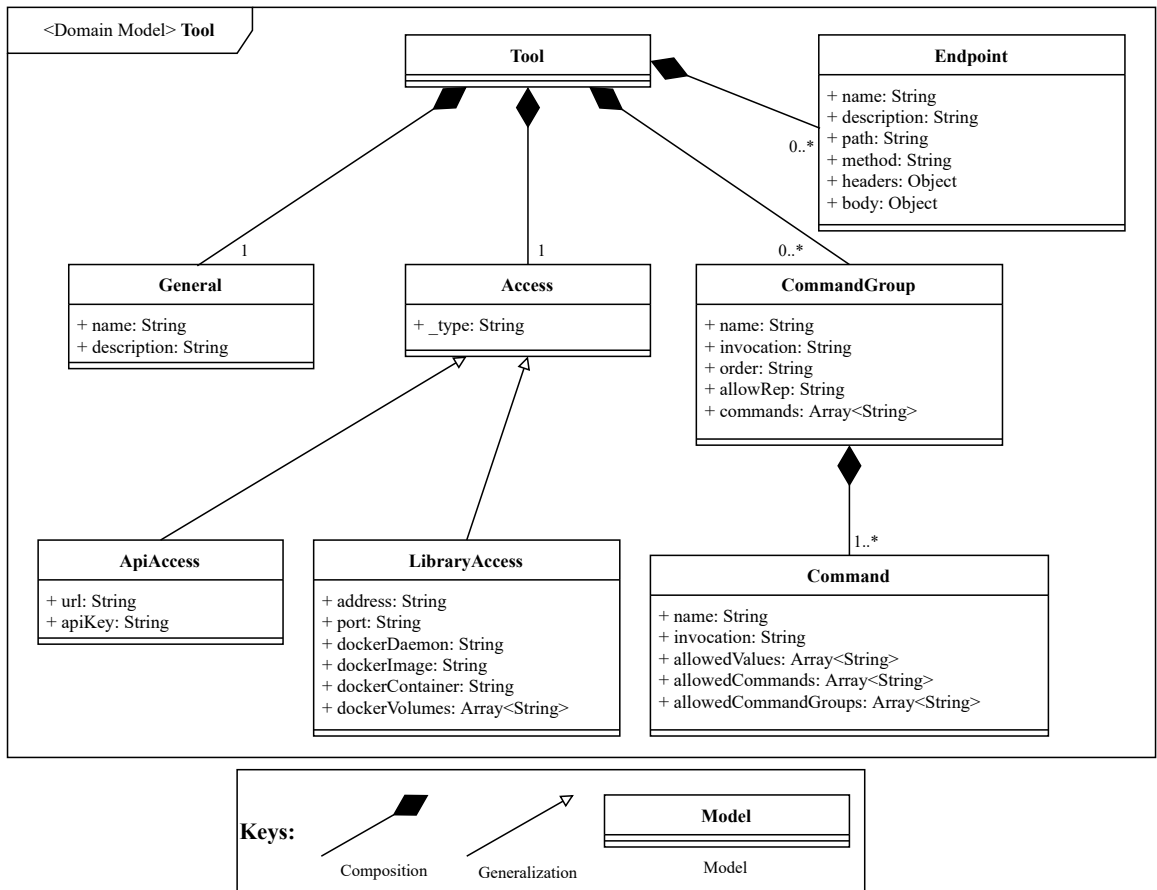


Figure 5.3: Tool domain model

- The `order` property defines the priority of a specific command or command group: a command with a higher order value (e.g., 1) can not be invoked before command with a lower order value (e.g., 0).
- The `allowRep` property defines if a command or command group can be invoked again, after a previous invocation.

Each `CommandGroup` is composed by one or more `Command` models, which represent a single command, inside the tool's CLI. The sub-properties which compose a `Command` model and their functions are following:

- `name`: the name of the command;
- `invocation`: the multiple command invocation aliases, specify how the command can be invoked;
- `allowedValues`: which are the values that the specific command accept;
- `allowedCommands`: the list of commands which can be invoked after the current command invocation;
- `allowedCommandGroups`: the list of command groups and the included commands that can be invoked after the current command invocation;

Inside the `Endpoint` model, the `method` property specifies the HTTP method of the endpoint. Properties `headers` and `body` allow users to specify which are the allowed HTTP headers and body that are accepted by the API's endpoint. For this sub-model, there are no groups as for the library model variation, because it is not usual to create the same hierarchy that is created with commands. Endpoints can have hierarchy among them, however, this happens in the context of the request path, which is out of the model's scope and control. In the application's prototype, although this model exists, it is purely conceptual, because the client did not fully implement the API tool integration, as it was chosen to invest and test more the containerized tool integration instead of the first one.

Workflow model

The workflow model, depicted in Figure 5.4 represents a workflow to be executed by Apache Airflow.

This model is responsible to bind a workflow to the user who submitted it, as Apache Airflow does not have a way to separate workflows for each user. Later in section 6.3, it will be shown how Apache Airflow will extract the user created workflows, in order to dynamically generate the Airflow DAGs to execute the workflow, using the ETL pipeline, which will transform the user workflow, stored in the database, into an Airflow DSL script, in order to be executable by the workflow system.

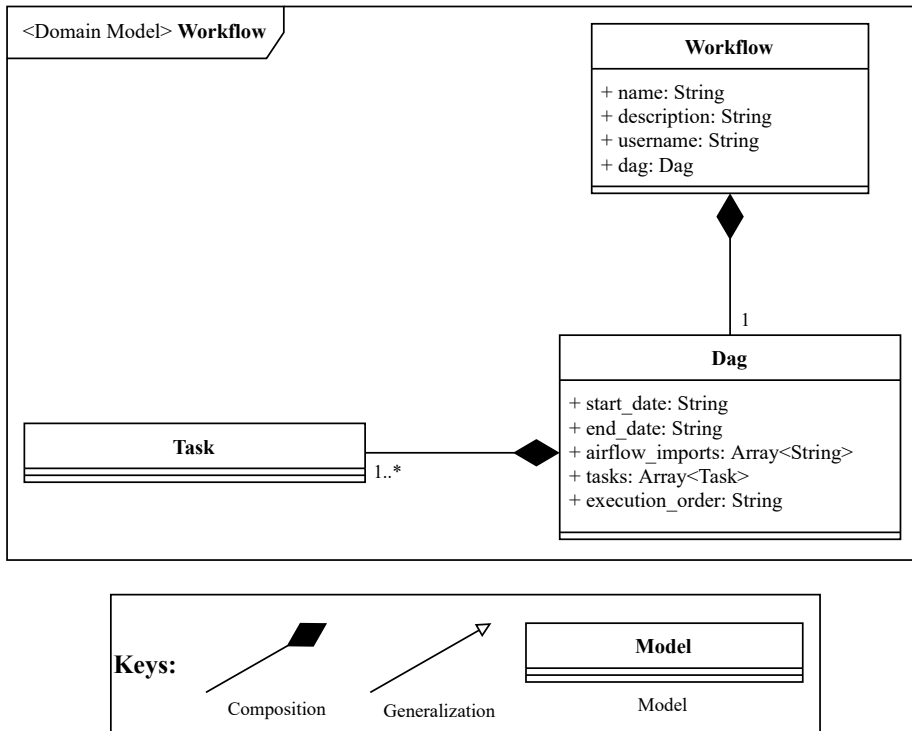


Figure 5.4: Workflow domain model

User model

The user model, depicted by figure 5.5, just represents the FLOWViZ user, and it is mainly used during the registration and login procedures.

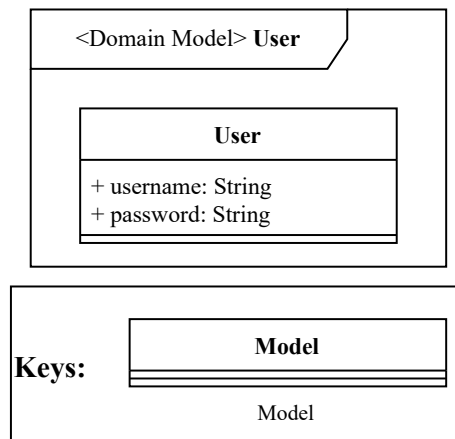


Figure 5.5: User domain model

In the model implementation, the password field is always stored in the hashed form, in order to be unusable in case of a data leak.

5.3 Use case implementation

Given the project main requirements and use cases, this section presents all the use case implementations. As previously shown, this project has three functional modules, each one can have one or more use cases: (i) *tool integration*, (ii) *workflow building* and (iii) *result production* modules.

The (i) *tool integration module* allows users to integrate new tools with the framework; the (ii) *workflow building* module allows them to build workflows with the previously integrated tools; finally, (iii) *the result production* module delivers results associated with workflows' executions. The rest of this section will now detail each functional module.

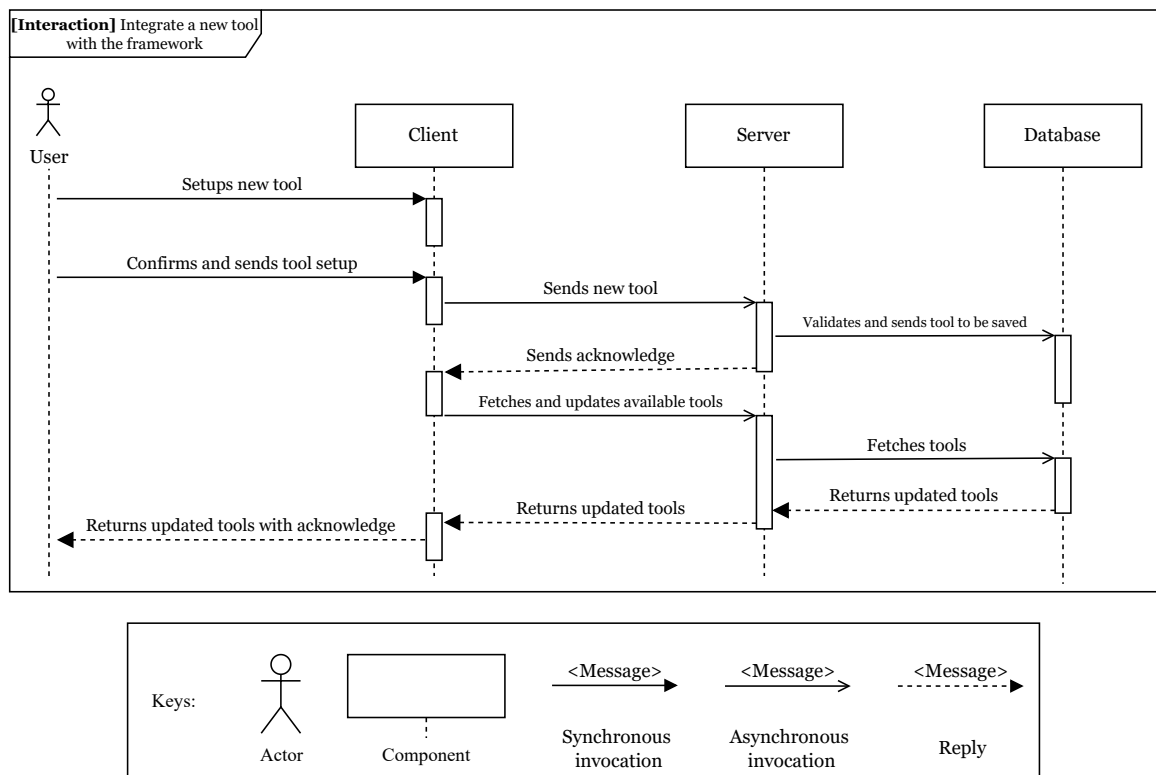


Figure 5.6: Tool integration module interaction model

Figure 5.6 shows the tool integration use case. During this use case, the user fills the tool contract form, which corresponds to the previously presented tool contract model. This is achieved by using the web client, that supplies a form divided into three steps: general, access and rules, which correspond to the three main properties inside the tool contract model. The user must fill all the required fields in order to successfully integrate a new tool with the framework and, also, needs to deploy its tool inside a docker container or in a remote computing instance, that can be accessible by the FLOWViZ framework.

After the contract's completion, the client sends it to the server, which will validate and, if successful, will save the contract into the database. At this point, the tool is now integrated with the framework and its documentation can be accessed in the web client.

When the necessary tools are integrated, the user can proceed to build workflows with them. The following Figure 5.7 explains how that procedure occurs.

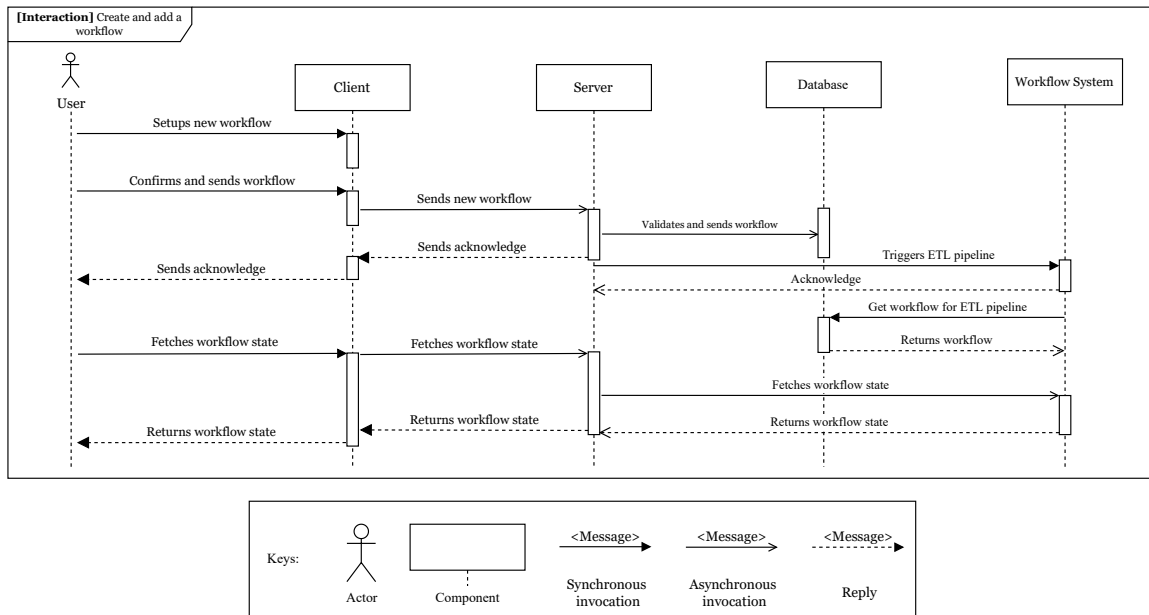


Figure 5.7: Workflow building module interaction model

Figure 5.7 shows the workflow building use case. This use case assumes that there are tools already integrated, which is achieved by fulfilling the previously presented use case.

During the workflow building, the user draws the workflow using the graphical editor, labeled as the *whiteboard*, provided in the web client. By using a side drawer, each integrated tool can be dragged and dropped into the editor. Here, each tool will be represented as a *node* inside the editor and can be configured as a task or step of the workflow. The user must also connect the nodes, in order to let the framework infer the data flow between each involved tool of the workflow being built.

When the user finishes the setup, the client will send the workflow to the server. When it reaches the HTTP server, the workflow will go through a validation and, if successful, it will be saved into the database associated to the user that created it.

If the database save procedure executed successfully, the server will then trigger the Airflow ETL pipeline, by sending an HTTP request to its REST API, which notifies that a new user workflow was created. The ETL pipeline, as it will be detailed in section 6.3, will transform the workflow contract into the Airflow DAG, allowing the workflow system to recognize the workflow and execute it.

If all of these steps succeed, the workflow system will execute the workflow at the date and time configured by the user and the execution's results can be obtained later, after the execution of the workflow.

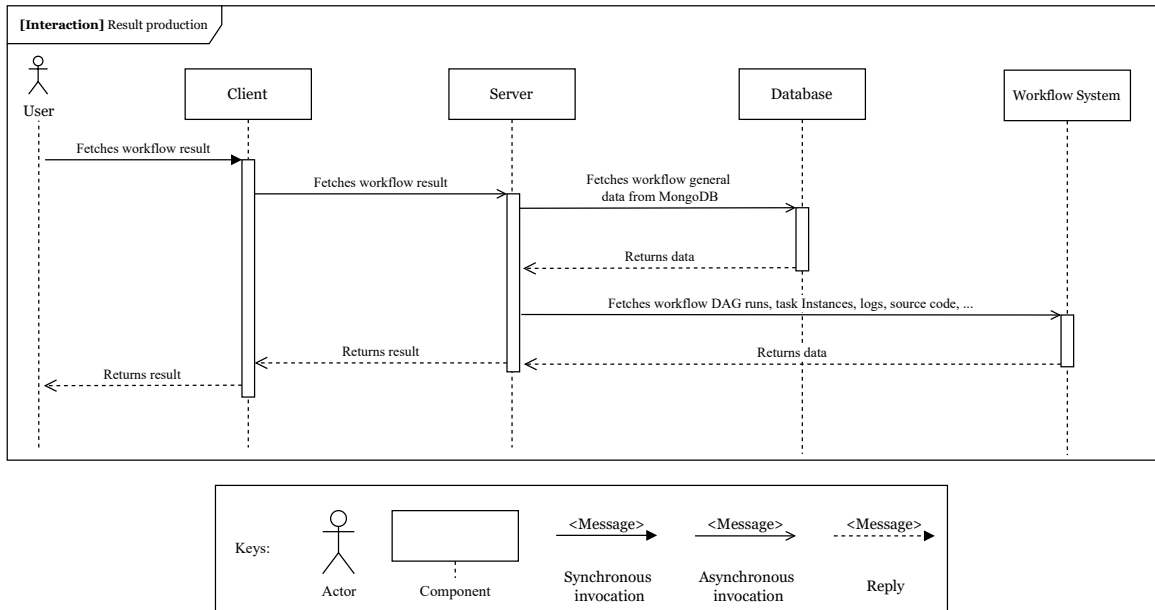


Figure 5.8: Result production module interaction model

Figure 5.8 shows the result production module that contains the last use case, which assumes that the previous two were properly executed. This use case allows the user to retrieve results from workflow executions, by using the web client.

By accessing the workflow list, the user can select one of its workflows and check its execution logs and the dynamically generated DAG's source code, created by the ETL pipeline. It can also access output files or data generated in each step.

This is the final use case of FLOWViZ: after the user integrates the necessary tools and builds a workflow with them, its execution will then produce results that can be obtained via the web client.

5.4 Server architecture

This section presents and explains the software architecture of the HTTP server. The HTTP server is the component that serves as a middleware between web client, and both database and the workflow system. It provides the client with the necessary endpoints that make the execution of the three functional modules possible, while providing user authentication and account management. This system component was designed following a three-layer design as depicted in Figure 5.9.

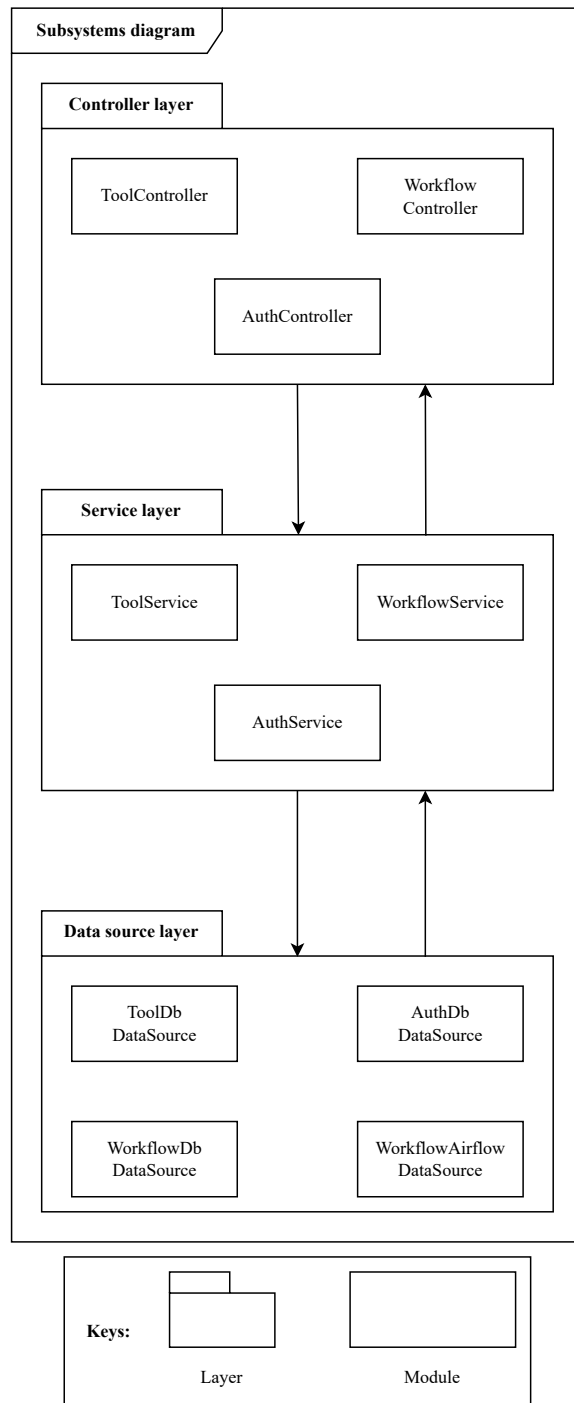


Figure 5.9: Server's subsystem diagram

Each presented layer or subsystem has a specific task, as enumerated below:

- **Controller layer:** matches the HTTP routes with the correspondent express middleware functions, which, consequently, call the necessary services;
- **Service layer:** contains the model business logic and calls the correspondent data

sources;

- **Data source layer:** fetches data from the data sources, namely databases or APIs;

Controllers contain express middleware functions, which will then be bind with their respective API routes. Controllers that have similar operations are also grouped in **Modules**, so they can be selectively attached to Node.js express's instance. This also provides *modularity* to the HTTP server, which is a key factor, when implementing this framework with other framework or application that shares common dependencies. For example, implementing this framework with an application that is a npm package and uses common dependencies.

This framework intends to extend the functionality of already existing phylogenetic frameworks, by only providing workflow building, execution and logging. Many available phylogenetic web frameworks are also built using the npm environment and dependencies, meaning that some executing dependencies can be shared with the FLOWViZ's server, such as: the express app containing all the included middleware functions and configurations and the authentication module. This avoids the FLOWViZ framework from creating new instances, by allowing it to reuse the existent ones provided by the phylogenetic framework. This way, the developer that integrates FLOWViZ can also choose which dependencies can be shared from the phylogenetic framework.

More details about the integration between FLOWViZ and phylogenetic framework can be found in section 6.4.2.

5.5 Client architecture

This section presents and explains the software architecture of the React web client. Like the HTTP server, the web client was also designed following a three-layer design as shown in Figure 5.10.

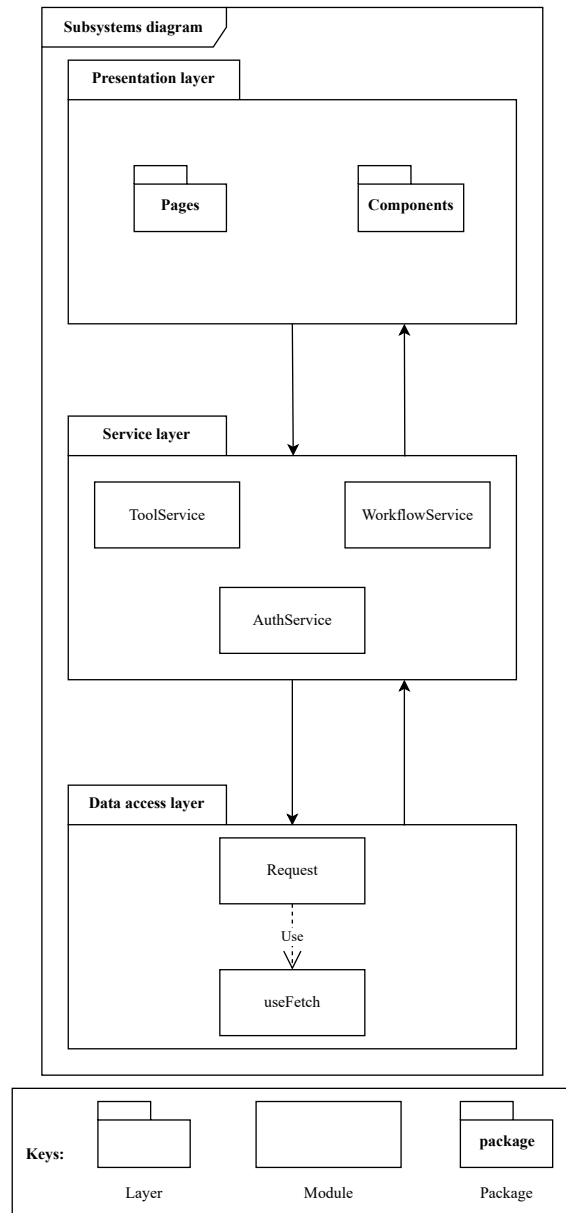


Figure 5.10: Client's subsystem diagram

Each layer performs its own specific task, as follows:

- **Presentation layer:** is composed by React Components, each are available among the Pages and Components packages. These components return HTML and create

the web client's user interface;

- **Service layer:** contains classes that retrieve the required information, as example, the ToolService only retrieves information from the HTTP server endpoints that are related to tools.
- **Data access layer:** All services use the Request class, which in cooperation with the useFetch custom React hook, creates the HTTP request to interact with the server's endpoints.

In particular, useFetch, from the Data access layer, is a custom hook meant to simplify HTTP requests, which use the JavaScript Fetch function. An HTTP request lifecycle can be primarily divided into three phases: Starting, Loading and a Conclusion. As such, useFetch uses an enumeration to easily identifies each phase of an HTTP request. For this implementation, four values were used: Starting, Loading, Success and Error. Figure 5.11 display a flowchart that illustrates the lifecycle of a simple and generic HTTP request and how the client's data access layer handles it.

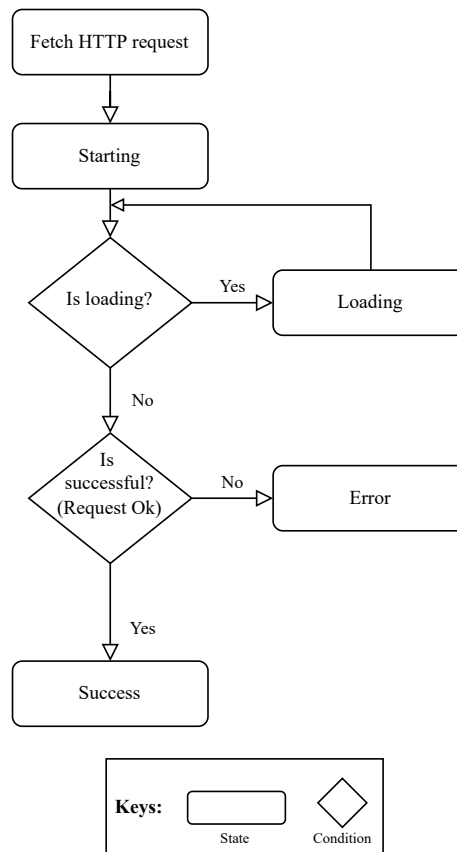


Figure 5.11: Flowchart of a generic HTTP request

Chapter 6

Implementation

This chapter displays the final results of the developed applications, according to the previously presented architecture and requirements. Two applications were developed to accomplish the project's objectives: a web client, built with React and an HTTP express server, built with Node.js. Both written in JavaScript.

The implementation was made along with the chosen workflow system - Apache Airflow and a MongoDB database - a *schemaless* database. A *schemaless* database was chosen over a *schema* database, as it was the simplest and most adequate database model for this project. The database was not the focus of this project and, because of this, it was chosen to integrate a database that complied with the project's objectives and minimum requirements.

Their specifications and details are shown during the sections 6.2 and 6.1, respectively. Section 6.3 also states configurations and developed scripts needed to integrate the workflow system with the developed applications. Finally, section 6.4 presents details regarding the solution's deployment.

The implementation was developed under a Free and Open-Source Software (FOSS) context and its source code is available in a public code repository*.

6.1 Server

The HTTP server is a middleware application between: the client, the Mongo database and the workflow system. It is mainly responsible to: (1) receive, validate and save tool contracts into the database tool model collection; (2) receive, validate and save user workflows into the database workflow model collection; (3) authenticate the user; (4) supply the client with integrated phylogenetic tools, users' created workflows and workflow execution logs.

The HTTP server was also built to be extensible: it can be integrated as a dependency

* github.com/mig07/FLOWViZ

along with another Node.js application that suffice the minimum requirements.

The current section presents all the used technologies and dependencies (subsection 6.1.1), used during development and a subsection regarding user authentication and security details (subsection 6.1.2).

Appendix E shows the table presenting all the implemented server's REST API endpoints.

6.1.1 Used technologies

This subsection mentions the technologies and dependencies used by the HTTP server. Only the main ones are mentioned, as most dependencies used in the application only complement these.

The first one is **Node.js**^{*}, it is an asynchronous event-driven JavaScript runtime to build scalable network applications and HTTP servers and the primary used dependency. **Express.js**[†] comes next, being a Node.js web application framework which provide a robust set of features to build web servers, it is the framework to establish and manage the HTTP endpoints and their middleware functions. As the HTTP server has to establish a connection with an external MongoDB[‡] database, **Mongoose**[§] is an ODM (Object Document Mapping) for Node.js, that is used to modulate data objects and establish data validation for the MongoDB database. As user authentication is also a server's requirement, **Passport.js**[¶] is an authentication middleware, providing a set of strategies to implement user authentication and authorization in any Express-based web application.

6.1.2 Authentication

When the user is authenticated, workflows are associated with its username, so only the user can manage its own workflows. This requires security measurements, as user authentication and authorization are involved. Due to this, user credentials need to be stored as secretly and safely as possible.

The adopted authentication strategy is passport-jwt, which requires the user to authenticate using a signed and valid JSON Web Token (JWT)^{||}, priorly generated by the server and sent inside the HTTP header. Figure 6.1 shows a generic and successful use case of the JWT authentication and Figures 6.2 and 6.3 show the user sign up and sign in server's implementations.

^{*}nodejs.org

[†]expressjs.com

[‡]mongodb.com

[§]mongoosejs.com

[¶]passportjs.org

^{||}rfc-editor.org/rfc/rfc7519

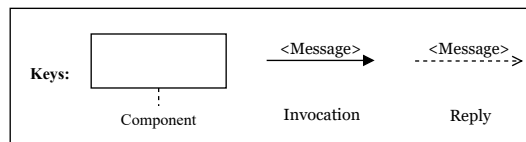
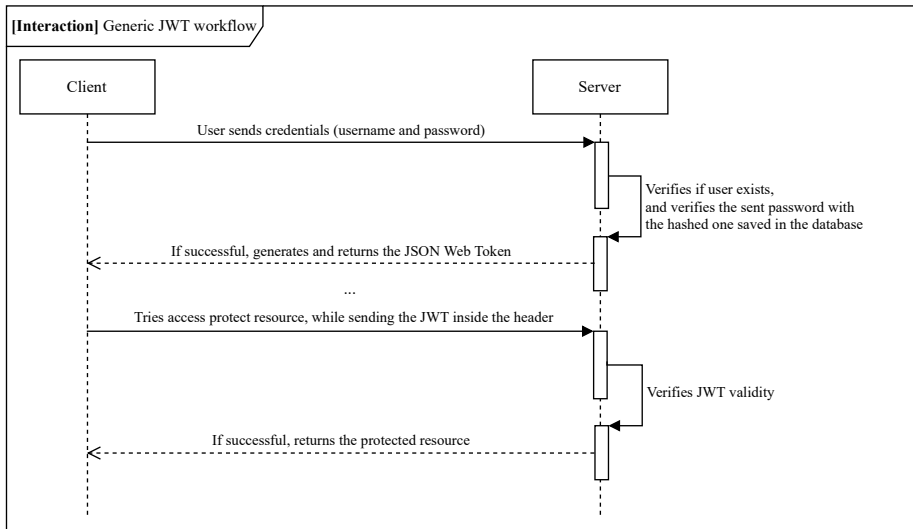


Figure 6.1: A generic and successful workflow of the JWT usage

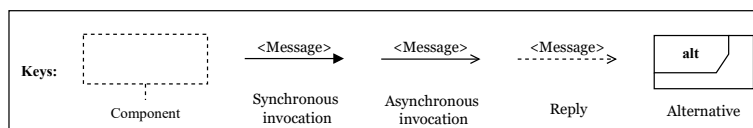
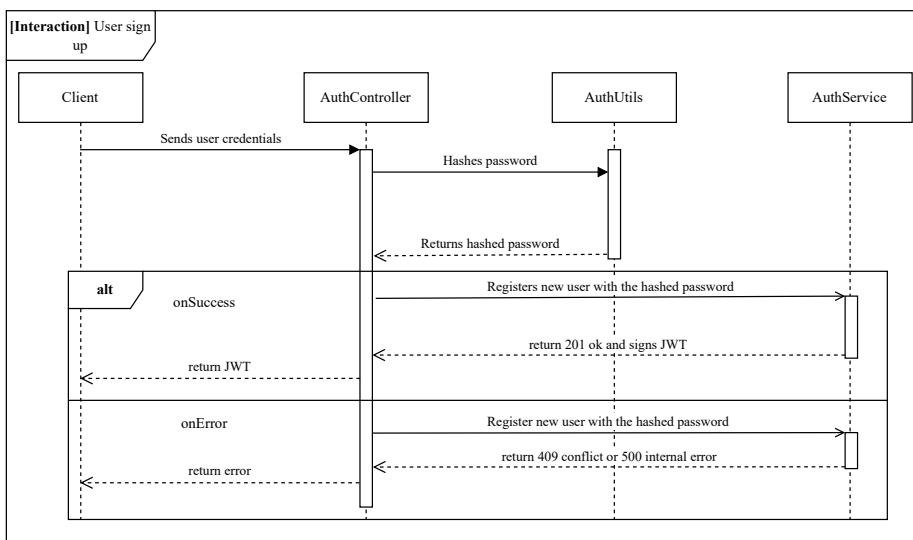


Figure 6.2: User sign up (registration) with JWT strategy

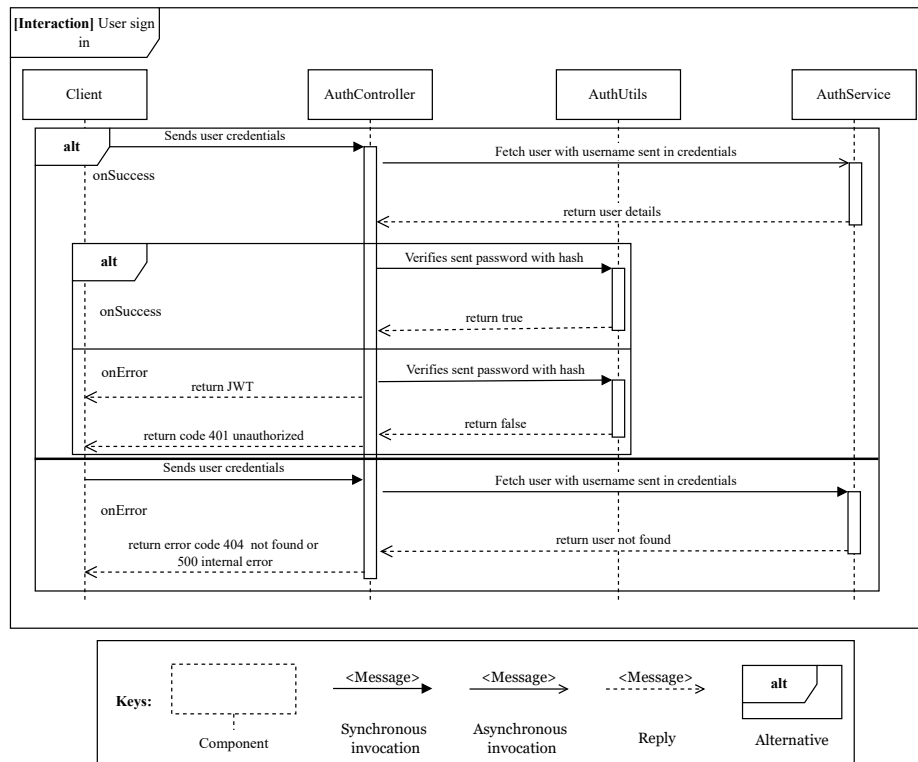


Figure 6.3: User sign in (login) with JWT strategy

Sensitive information, such as passwords, are safely stored into the database. This is achieved by **hashing** users' passwords. Contrary to encryption, hashing is an one-way function, meaning that it only transforms plaintext into a hash and the reverse operation is impossible to achieve. This is beneficial, because even if users passwords get leaked, an attacker can not use the information to infiltrate users' accounts.

The most popular hashing password dependencies available are: BCrypt[25] and Argon2[5]. The first one has been available for quite a long time, delivers excellent results, and it is still widely used on legacy systems, however, Argon2 is more recent and delivers the same as results as BCrypt, while using less computational resources, being this the reason why Argon2 was chosen for this project. This library was also recommended by OWASP*.

6.2 Client

The web client is the middleware application between the user and the HTTP server, which allows the user to: (i) integrate new phylogenetic tools; (ii) build, schedule and manage workflows; (iii) check workflow executions' results. This is achieved by allowing the user to create contracts, where tools' guidelines and rules are specified and sent to the server; and by supplying an editor or *whiteboard* where the user can cast each previously

*https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

integrated tool, configure it and connect with other integrated tools, using edges. This way, the user can graphically draw its customized workflows, while configuring each tool as desired.

The current section presents all the used technologies and dependencies (subsection 6.2.1), used during development, provides more details about the data layer implementation in subsection 6.2.2 and, finally, displays the developed application in subsections 6.2.3, 6.2.4, 6.2.5 and 6.2.6, while matching the implementations shown in the last three subsections with the previously shown use case implementation diagrams 5.6, 5.7 and 5.8, respectively.

6.2.1 Used technologies

As previously mentioned, the web application was built with **React** and written in JavaScript. React was chosen due to its current abilities, performance and popularity. A great feature provided by this framework is data state management, which is a frequently used in the web application, from composing web pages to building HTTP requests. **Material-UI** was used for the client's user interface. It is a CSS framework which provides React with out-of-the-box components that follow the Google's Material Design, in order to seamlessly build the application's graphical user interface, without needing to write CSS from the ground up. One of the requirements of this application is to allow the user to graphically build workflows. For this to be attainable, the application must have a page that supplies an editor, where the user can input nodes and draw edges between them. In this implementation, nodes represent different integrated tools and edges between nodes represent data dependencies and the *flow of work*. **React Flow** provides a highly customizable React component for building node-based editors and interactive diagrams, allowing the developer to build editors where users can graphically manage nodes and their respective data dependencies.

6.2.2 Data layer implementation details

This subsection provides more details about the data layer implementation, namely the `useFetch` custom hook and the request function. The first one was built according to the previously presented flowchart in Figure 5.11. The `useFetch` returns an array that contains three objects: data, request's state and error. The request's state shows in which phase the HTTP request is, there are only four established phases: starting, fetching, error and success. Listing 6.1 shows the code implementation of the previously shown flowchart, with the custom hook return array.

*reactjs.org
†mui.com
‡reactflow.dev

Listing 6.1: useFetch custom hook useEffect operation

```
1 const [data, setData] = useState(null);
2 const [reqState, setReqState] = useState(null);
3 const [error, setError] = useState(null);
4
5 useEffect(async () => {
6     setReqState(RequestState.starting);
7
8     try {
9         setReqState(RequestState.fetching);
10
11         const response = await fetch(url, options);
12         const res = await response.json();
13
14         if (!response.ok) {
15             setError(res);
16             setReqState(RequestState.error);
17             return;
18         }
19
20         // Saving response data into state
21         setData(res);
22         setReqState(RequestState.success);
23     } catch (error) {
24         setError(error);
25         setReqState(RequestState.error);
26     }
27 }, [url]);
28
29 return [data, reqState, error];
```

Listing 6.2: Request function

```
1 function Request(url, options, onError, onSuccess, onLoading) {
2     const [data, requestState, error] = useFetch(url, options);
3
4     switch (requestState) {
5         case RequestState.fetching:
6             return onLoading;
7         case RequestState.error:
8             return onError(error);
9         case RequestState.success:
10            return onSuccess(data);
11    }
12    //...
13 }
```

The function depicted by Listing 6.2 allows React components from the client's presentation layer to pass JSX code inside the callback arguments: the last three function's arguments. Depending on which state the HTTP request is, the request function will execute and return the passed callback functions that matches with the ongoing HTTP request state.

The service layer also abstracts the presentation layer from the request function, as it already pre-configures the url and options arguments, which are only related to the HTTP request. This way, the presentation layer only needs to call the desired service to fetch specific information for a certain visual component, while only passing a function for success, one for error and one to execute when the HTTP request is loading (fetching state), that can be, as an example, a loading animation.

6.2.3 Application overview

This subsection provides an overview of the developed client application, by providing images of the graphical user interface and explaining the application's functionalities.

When FLOWViZ is integrated with another phylogenetic framework, the image depicted in Figure 6.4 will be the first page that the user will be redirected to.

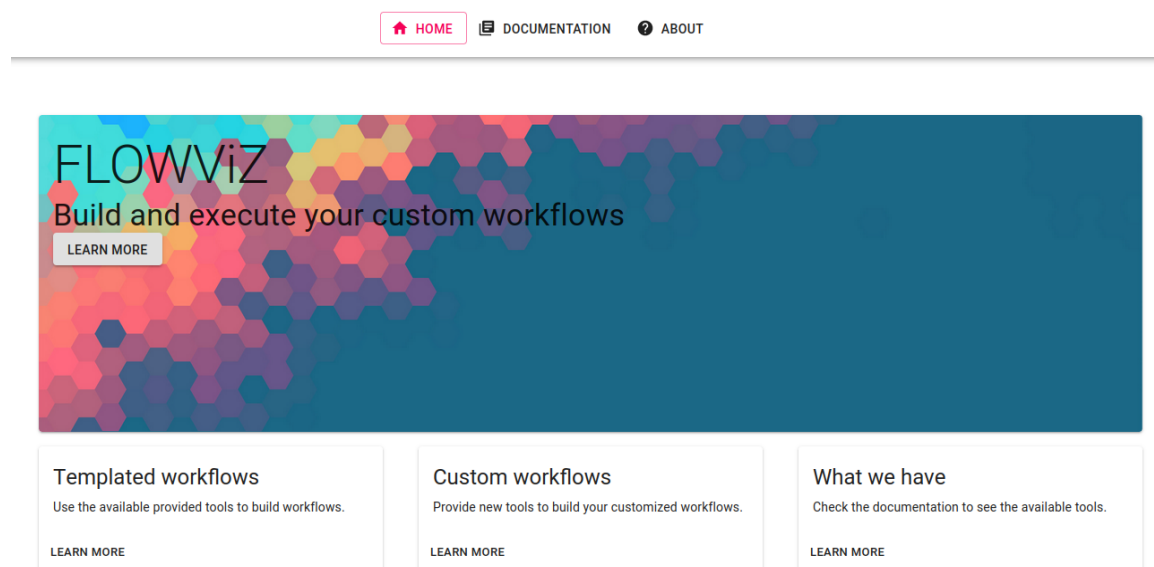


Figure 6.4: FLOWViZ: home page

From the home page, the user can check the documentation, where all integrated tools are displayed (Figure 6.5). It can also consult the about page, where all the technical information can be consulted, such as the code repository, project issues can be consulted and also some utilization tutorials, regarding the FLOWViZ framework.

Available tools

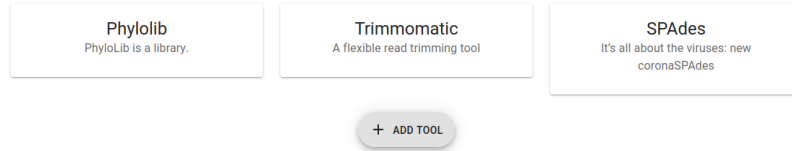


Figure 6.5: FLOWViZ: documentation page

By clicking on a tool item, the user can access all the documentation about a tool, which is automatically generated when integrating the tool with FLOWViZ (Figure 6.6).

Phylolib

Type: library

Description: PhyloLib is a library.

Library

Usage

Phylolib [Arguments] [Options]

Arguments

help : <help>
distance : <distance> (hamming,grapetree,kimura) [Options]
correction : <correction> (jukescantor) [Options]
algorithm : <algorithm> (goeburst,edmonds,sl,cl,upgma,upgmc,wpdma,wpdmc,saitounei,studierkepler,unj) [Options]
optimization : <optimization> (lbr) [Options]

Options

File Output : <-o,-out> (file) -> Output file as <format>:<location> with format being (asymmetric|symmetric|newick|nexus)
Dataset Input : <-d,-dataset> (file) -> Input dataset file as <format>:<location> with format being (fasta|ml|snp)
Distance Matrix Input : <-m,-matrix> (file) -> Input distance matrix file as <format>:<location> with format being (asymmetric|symmetric)
Phylogenetic Tree Input : <-t,-tree> (file) -> Input phylogenetic tree file as <format>:<location> with format being (newick|nexus)
Limit of focus variants : <-l,-lvs> (file) -> Limit of locus variants to consider using goeBURST algorithm [default: 3]

Figure 6.6: FLOWViZ: specific tool documentation

From the home page, the user can also sign up or sign in (Figure 6.7). Being authenticated is a requirement when consulting or adding workflows.

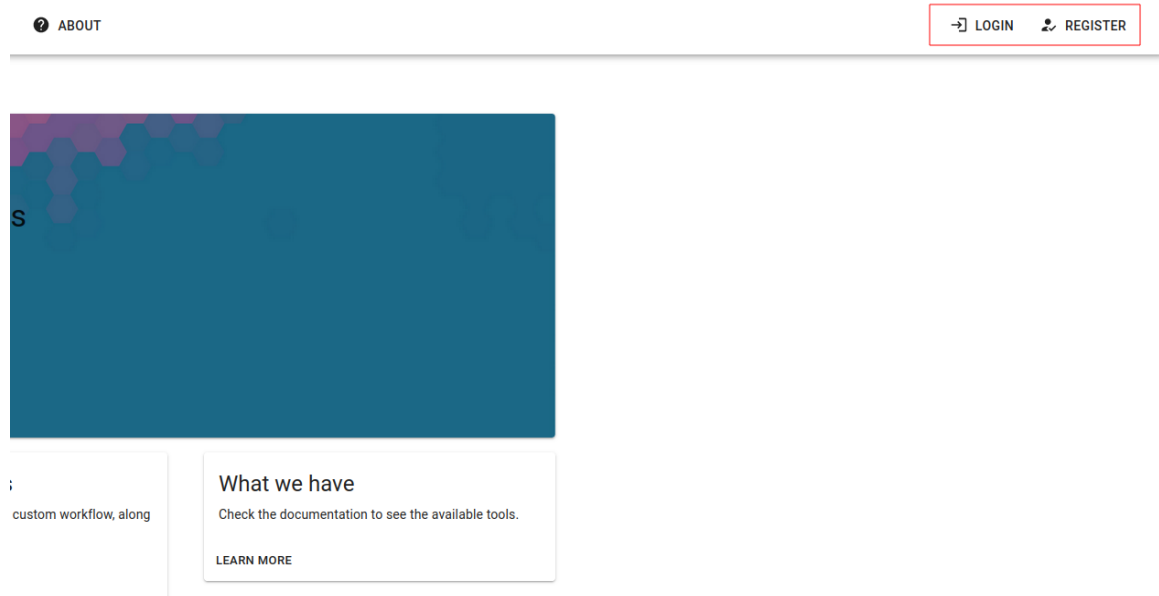


Figure 6.7: FLOWViZ: sign in and sign up buttons location (home page top-right corner)

The image shows a sign-in form. At the top center is an orange circular icon with a white right-pointing arrow. Below the icon is the text 'Sign in'. The form consists of two input fields: 'Username *' with the text 'mig07' and 'Password *' with a masked password represented by dots. Below the password field is a checked checkbox labeled 'Remember me'. At the bottom of the form is a blue button with the text 'SIGN IN'.

Figure 6.8: FLOWViZ: sign in page

The following subsections 6.2.4, 6.2.5 and 6.2.6 detail how the use case implementations (section 5.3) are implemented in the client application.

6.2.4 Tool integration

As previously shown by the interaction diagram 5.6, the tool integration use case is divided in three main steps: general (Figure 6.9), access (Figure 6.10) and rules (Figure 6.11).

The screenshot shows a web form titled "Add tool". At the top, there are three tabs: "General" (selected), "Access", and "Rules". Below the tabs, the "General" section contains two text input fields. The first field is labeled "Tool name *" and contains the text "Phylolib". The second field is labeled "Tool description *" and contains the text "PhyloLib is a library.". At the bottom of the form, there are two buttons: "PREVIOUS" on the left and "NEXT" on the right.

Figure 6.9: FLOWViZ: tool integration general fragment

In the general fragment, depicted in Figure 6.9, the user only needs to specify the name and description of the tool. The name is used as the tool's *primary key*, meaning that no other tool must have the name. Both attributes are just used for documentation purposes.

Add tool

General Access Rules

Choose your configuration method

API Library

Access

Tool address *
localhost

Tool port

Container

Docker image *
luanab/phyloblib

Docker URL *
unix:///var/run/docker.sock

Docker container
phyloblib

Volume source: /opt/.phyloblibVol Volume target: /phyloblib +

[< PREVIOUS](#) [NEXT >](#)

Figure 6.10: FLOWViZ: tool integration access fragment

It is assumed that the user must deploy its customized tool in a remote virtual machine or in a container, that provides the conditions to allow the framework to remotely access it.

In the access fragment, depicted in Figure 6.10, the user must specify the tool's host access parameters. There are only two possible ways to specify the tool's access, either the tool provides a CLI, where the commands can be specified, or a REST API, where the endpoints rules can be specified. These specifications occur in the next and final fragment: the rules fragment.

Figure 6.10 shows the access specification of a library hosted by remote docker container.

Add tool

General Access Rules

Number of commands groups

Name

Invocation +

order

Allow command repetition

Number of commands

Commands

Figure 6.11: FLOWViZ: tool integration rules fragment

Finally, Figure 6.11 displays the rules fragment of the tool integration. The depicted figure, exemplifies the rules' specification of a library that exposes a CLI. In this case, the user must specify the group of commands and each command inside each created group. This allows the user to implicitly create the command's hierarchy, defining which commands can be called after one's invocation. This way, tasks' configurations during the workflow building will be easier to configure and less prone to human errors, as the user implementing the tool has to strictly define the commands' hierarchy and allowed values.

6.2.5 Workflow building

This subsection represents the previously mentioned implemented use case displayed by Figure 5.7, which assumes that the user has already completed the previous step and integrated its desired tools. Figure 6.12 shows the application’s graphical editor where users can build the customized workflows - the *whiteboard*.

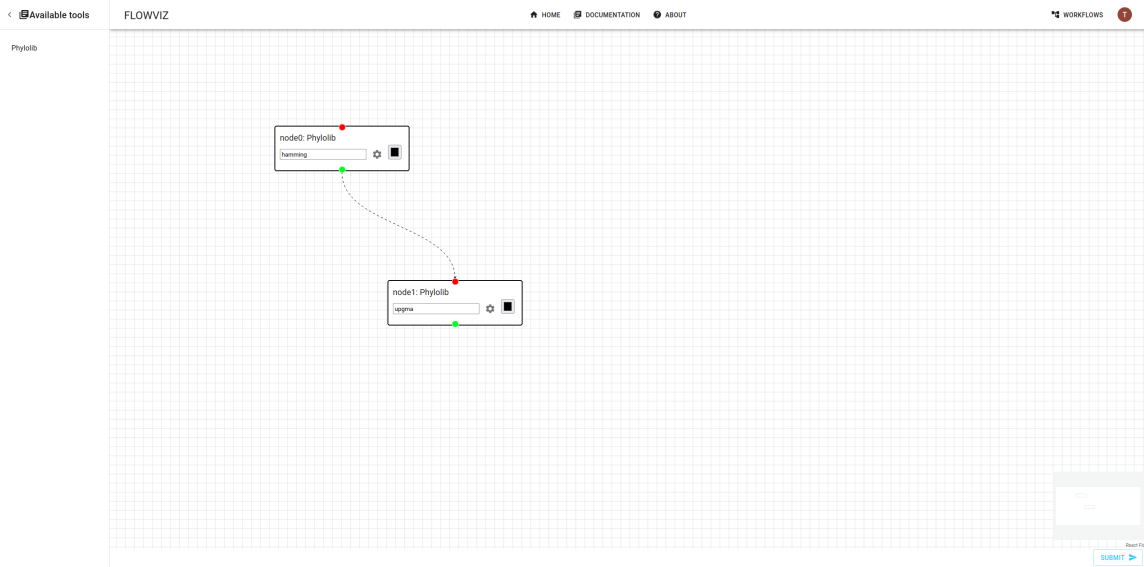


Figure 6.12: FLOWViZ: *whiteboard*

Inside *whiteboard* page the user will be supplied with a menu drawer, that contains the previously integrated tools. From the drawer, the user can *drag and drop* a pretended tool into the editor and the invoked tool will be part of the workflow, represented as task and graphically shown as a node. The user can drag multiple tools and wire the nodes with edges, which will implicitly infer the data workflow’s data dependencies. The user can configure each involved task present inside the workflow, as presented by Figure 6.13.

Task Setup

Tool information

<p>name: Phylolib address: localhost dockerImage: luanab/phyloblib dockerAutoRemove: dockerApiVersion: auto</p>	<p>description: Phylolib is a library. dockerUrl: unix:///var/run/docker.sock dockerContainer: phylolib dockerNetworkMode: bridge</p>
--	--

I/O variables

<p>Input</p> <p>Input key: <input style="width: 90%;" type="text" value="dataset"/> ?</p> <p>Input value: <input style="width: 90%;" type="text" value="datasets/10.txt"/> ?</p> <p>dataset ✕</p>	<p>Output</p> <p>Output key: <input style="width: 90%;" type="text" value="out"/> ?</p> <p>Output value: <input style="width: 90%;" type="text" value="phyloblib/out.txt"/> ?</p> <p>out ✕</p>
---	--

Setup

Command preview

```
phylolib distance grapetree -d dataset -o out
```

✕	Arguments	▼
	distance	▼
	Options	▼
✕	Dataset Input	▼
	dataset	▼
	Options	▼
✕	File Output	▼
	out	▼

ADD COMMAND

CANCEL APPLY

Figure 6.13: FLOWViZ: configuration of a task inside the workflow

Figure 6.13 shows the configuration of task which uses the *Phylolib* library. For this specific tool, the user can configure multiple commands for each task. The shown *drop-down menus* are filled with values that come from the established tool contract, which avoids the user from manually inputting information and, implicitly, reducing possible human errors when configuring the task.

After the configuration of all tasks, the user can submit the workflow. By clicking the bottom-right submission button, which will show the following form displayed by Figure 6.14.

Figure 6.14: FLOWViZ: workflow submission form

In this form, the user must register the workflow name, the description and the date and time that it wants to be executed. The workflow will be sent to the HTTP server, which will validate it and, if successful, save it into the database.

6.2.6 Result production

After the workflow’s submission, a new entry will appear in the client’s workflow list page (Figure 6.15). The user can access each created workflow and its details by clicking on the respective list item. The details are then retrieved from two data sources: (i) the database and (ii) the workflow system. At first, when the workflow is being created, only the database’s information will be available, as the workflow system is parsing the workflow into the Airflow DSL or the workflow is being or will be executed and no logs were generated. In this time window, only the list entry and basic detail information, such as workflow’s name and description, will appear as the information that comes from the workflow system does not exist yet. Figure 6.16 shows an example of workflow execution’s log.

Name	Description
PHYLOLIBWORKFLOW	A test workflow for Phylolib

Figure 6.15: FLOWViZ: workflow list page

PhylolibWorkflow

Description: A test workflow for Phylolib

Number of runs: 1

Run details: manual__2022-09-02T15:48:26.056359+00:00

Run date: 2022-09-02T15:48:26.056359+00:00

Run state: success

Task ID: hamming

Run date: 2022-09-02T15:48:26.056359+00:00

Run state: success

Log number: 1

```
[('a214ee09c6c6', '**** Reading local file: /opt/airflow/logs/PhylolibWorkflow/hamming/2022-09-02T15:48:26.056359+00:00/1.log\n[2022-09-02 15:48:26,767] {taskinstance.py:1035} INFO - Dependencies all met for <TaskInstance: PhylolibWorkflow.hamming manual__2022-09-02T15:48:26.056359+00:00 [queued]>\n[2022-09-02 15:48:26,780] {taskinstance.py:1035} INFO - Dependencies all met for <TaskInstance: PhylolibWorkflow.hamming manual__2022-09-02T15:48:26.056359+00:00 [queued]>\n[2022-09-02 15:48:26,780] {taskinstance.py:1241} INFO - \n-----\n[2022-09-02 15:48:26,780] {taskinstance.py:1242} INFO - Starting attempt 1 of 1\n[2022-09-02 15:48:26,781] {taskinstance.py:1243} INFO - \n-----\n[2022-09-02 15:48:26,789] {taskinstance.py:1262} INFO - Executing <Task(DockerOperator): hamming> on 2022-09-02 15:48:26.056359+00:00\n[2022-09-02 15:48:26,792] {standard_task_runner.py:52} INFO - Started process 660 to run task\n[2022-09-02 15:48:26,794] {standard_task_runner.py:76} INFO - Running: ['****', 'tasks', 'run', 'PhylolibWorkflow', 'hamming', 'manual__2022-09-02T15:48:26.056359+00:00', '--job-id', '1508', '--raw', '--subdir', 'DAGS_FOLDER/PhylolibWorkflow.py', '--cfg-path', '/tmp/tmpgmgqc115', '--error-file', '/tmp/tmpstlut_l_']\n[2022-09-02 15:48:26,795] {standard_task_runner.py:77} INFO - Job 1508: Subtask hamming\n[2022-09-02 15:48:26,823] {logging_mixin.py:109} INFO - Running <TaskInstance: PhylolibWorkflow.hamming manual__2022-09-02T15:48:26.056359+00:00 [running]> on host a214ee09c6c6\n[2022-09-02 15:48:26,862] {taskinstance.py:1429} INFO - Exporting the following env vars:\nAIRFLOW_CTX_DAG_OWNER=****\nAIRFLOW_CTX_DAG_ID=PhylolibWorkflow\nAIRFLOW_CTX_TASK_ID=hamming\nAIRFLOW_CTX_EXECUTION_DATE=2022-09-02T15:48:26.056359+00:00\nAIRFLOW_CTX_DAG_RUN_ID=manual__2022-09-02T15:48:26.056359+00:00\n[2022-09-02 15:48:26,886] {docker.py:258} INFO - Starting docker container from image luanab/phyloblib\n[2022-09-02 15:48:26,891] {docker.py:269} WARNING - Using remote engine or docker-in-docker and mounting temporary volume from host is not supported. Falling back to 'mount_tmp_dir=False' mode. You can set 'mount_tmp_dir' parameter to False to disable mounting and remove the warning\n[2022-09-02 15:48:27,923] {docker.py:320} INFO - INFO: Started running command 'distance' with type 'hamming'\n[2022-09-02 15:48:27,936] {docker.py:320} INFO - INFO: Started reading file '/phyloblib/data/datasets/10.txt'\n[2022-09-02 15:48:27,990] {docker.py:320} INFO - INFO: Finished reading file '/phyloblib/data/datasets/10.txt'\n[2022-09-02 15:48:28,003] {docker.py:320} INFO - INFO: Started writing file '/phyloblib/out.txt'\n[2022-09-02 15:48:28,010] {docker.py:320} INFO - INFO: Finished writing file '/phyloblib/out.txt'\n[2022-09-02 15:48:28,010] {docker.py:320} INFO
```

Figure 6.16: FLOWViZ: workflow execution log

In this workflow detail page, the user can filter the information by (i) DAG run, which represents an execution instance of a workflow inside the Airflow environment, by (ii) task and by (iii) log of each involved task. This page also displays the source code, which represents the Airflow DSL code, parsed from the workflow which was saved in the database (Figure 6.17).

```

Airflow script
-----
from airflow import DAG
from datetime import datetime
from airflow.providers.docker.operators.docker import DockerOperator
from docker.types import Mount

default_args = {
    'owner'          : 'airflow',
    'description'    : 'A test workflow for Phylolib',
    'start_date'     : datetime.today(),
}

with DAG('PhylolibWorkflow', schedule_interval=None, default_args=default_args) as dag:

    hamming = DockerOperator(task_id = 'hamming',
                             image = 'luanab/phyloblib',
                             api_version = 'auto',
                             mounts = [Mount(target='/phyloblib', source='/opt/.phyloblibVol', type='bind')],
                             command = 'distance hamming --dataset=ml:/phyloblib/data/datasets/10.txt --out=symmetric:/phyloblib/out.txt',
                             auto_remove = 'true',
                             docker_url = 'unix://var/run/docker.sock',
                             network_mode = 'bridge',)

    upgma = DockerOperator(task_id = 'upgma',
                           image = 'luanab/phyloblib',
                           api_version = 'auto',
                           mounts = [Mount(target='/phyloblib', source='/opt/.phyloblibVol', type='bind')],
                           command = 'algorithm upgma --out=newick:/phyloblib/tree.txt --matrix=symmetric:/phyloblib/out.txt',
                           auto_remove = 'true',
                           docker_url = 'unix://var/run/docker.sock',
                           network_mode = 'bridge',)

hamming >> upgma

```

Figure 6.17: FLOWViZ: workflow source code

All the information regarding the workflow, such as the DAG Runs, tasks, logs and source code, comes from Airflow via its REST API, which makes this information available.

6.3 Workflow system

This section presents the features that were developed and implemented to integrate the workflow system - Apache Airflow, with FLOWViZ.

The main goal of Apache Airflow is to schedule and execute users' workflows, that are sent from the server and saved into MongoDB. To this end, an ETL pipeline was created, in order to retrieve users' workflows from the database and transform them into Airflow DAGs - Airflow DSL scripts, to be executable by the workflow system. An ETL pipeline is a work pipeline concept that *extracts* information from a data source, namely, from the MongoDB's workflow collection, *transforms* the extracted information into another format and, finally, *loads* the transformed information into a new DAG to be executed by the workflow system. Figure 6.18 shows the use case of the developed ETL pipeline inside the system architecture.

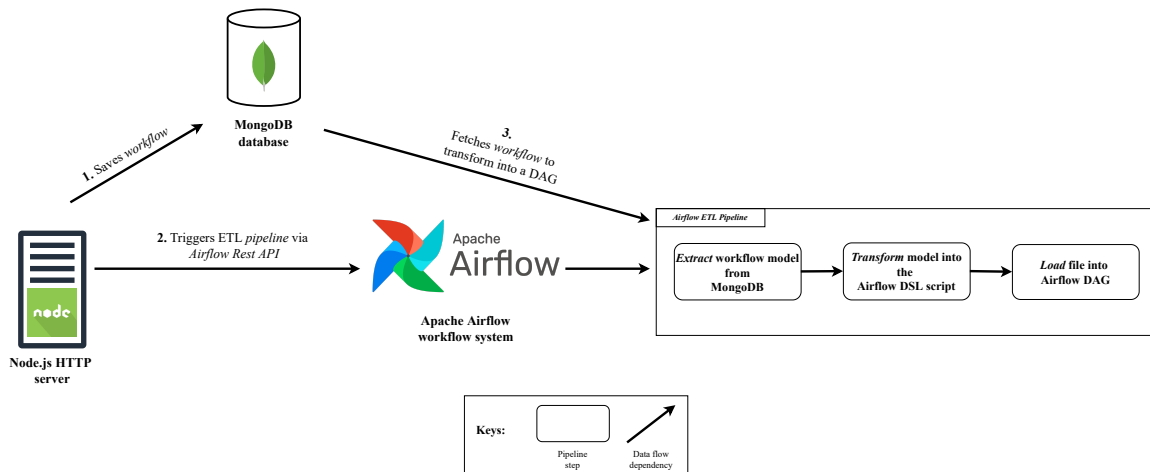


Figure 6.18: ETL pipeline

Observing Figure 6.18, when a client sends a workflow to the server, it will first go through a validation and, if it succeeds, it will be saved into database (1.). If the workflow was successfully saved, the server will then notify the workflow system that there is a new user workflow to be parsed and executed at a certain date and time, that was previously configured by the user (2.). This notification will trigger the Airflow’s ETL pipeline via the workflow system’s REST API, which will retrieve the workflow from the database and parse it to an Airflow DAG, in order to be executable by the workflow system (3.).

The ETL pipeline is a DAG executing inside Airflow, with only the three core functions: *extract*, *transform* and *load*. The last two are performed in one function inside the DAG, because it uses an auxiliary python file, which is an Airflow DSL script template, that is used to replace the templated fields with the parsed information, which happens at the same time, when dynamically generating the workflow’s DAG. By merging users’ workflow information from the Mongo database with the DAG template, it is possible to dynamically generate Airflow DAGs with great flexibility. Its code is shown in appendix D. The following Figure 6.19 details each step of the pipeline.

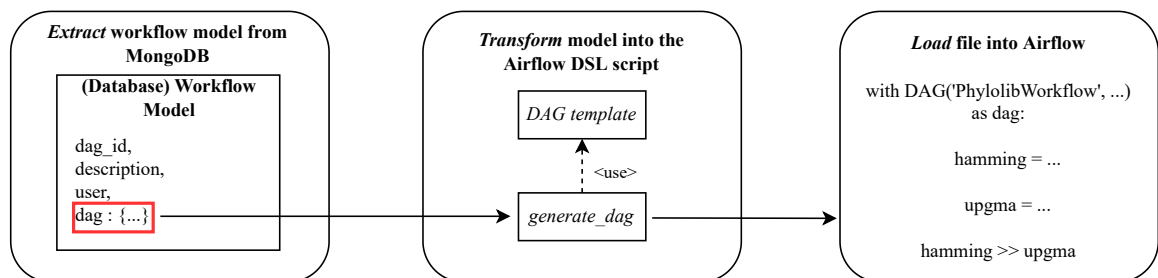


Figure 6.19: ETL pipeline detail

The workflow system starts by fetching the database, using the name of the workflow and its user, which is metadata that is included with the HTTP request that performs the ETL pipeline’s trigger. After the retrieval of the workflow, the *extract* method will extract

the dag property from the database's workflow model and send it to next function. The second pipeline's function will *transform* the database workflow using the dag property and the `generate_dag` function in cooperation with the DAG template will generate a new Airflow DAG - an Airflow DSL script. Finally, Airflow will *load* the script into the Airflow DAG collection, which will be recognized by the workflow system and executed at the configured date and time. This method is also advised by Astronomer.io*, which is the first one that is presented in the *Multiple-file methods*' section. None of the *Single-file methods* were used, as the solution was not as versatile and would also compromise security as generating DAGs from variables is not a good option, because these can be globally available to any DAG.

The interaction between the workflow system and database is possible due to a specific Airflow provider. A provider† is a package that contains classes, such as operators, hooks or sensors, to allow integration with another external systems and extend the Airflow base functionalities. Although Airflow comes with most providers and operators, the MongoDB provider‡ is not included in the Airflow's base providers by default and has to be explicitly installed.

During development, Airflow was executed in containers and, because of this, a DockerFile was made to explicitly install the MongoDB provider. The DockerFile is displayed by Listing 6.3.

Listing 6.3: Airflow development DockerFile

```
1 FROM apache/airflow:2.2.1
2 RUN pip install apache-airflow-providers-mongo
```

In a local deployment solution, to install this provider it is only needed to execute the last line on the displayed Listing 6.3 (install the provider via pip).

During development, Airflow and all its dependent modules were containerized, along with the MongoDB database, which was separated container. To allow communication between the database and the workflow system, not only it was required to install the MongoDB provider, but it was also needed to create a docker network, to enable network communications among the executing containers. The network's creation and configuration is displayed by Listing 6.4.

Listing 6.4: Docker network creation

```
1 docker network create flowviz-docker-network
2 # Connect every involved container to the created network (including
   Airflow workers)
3 docker network connect flowviz-docker-network [name of container]
```

All Airflow containers involved in workflow executions must be included in this network, along with the MongoDB container. When all Airflow and MongoDB containers are finally included in the network, the MongoDB container network IP can be found by

*astronomer.io/guides/dynamically-generating-dags/

†airflow.apache.org/docs/apache-airflow-providers/

‡airflow.apache.org/docs/apache-airflow-providers-mongo/stable/index.html

executing the command displayed by Listing 6.5. The IP address is available through a field called `IPv4Address`, inside the respective mongo container JSON object.

Listing 6.5: Docker network inspect

```
1 docker network inspect flowviz-docker-network
```

With this IP address, a MongoDB connection inside Airflow can be established and, therefore, the ETL pipeline can be executed, which will dynamically create new DAGs, correspondent to the user workflows.

6.4 Deployment

This section shows the deployment details of the developed applications (subsection 6.4.1) and also provides details about the FLOWViZ integration with a phylogenetic framework (subsection 6.4.2).

6.4.1 HTTP server and React client

The HTTP server and the React client are deployed along together in the same platform. To achieve this some requirements need to be done: the first one is to create a production build of the client, which produces a static version of the developed client, minifying the code, optimizing the assets and creating a bundle which size is drastically reduced compared to the development version. Listing 6.6 shows the command that creates the production build.

Listing 6.6: React client production build command

```
1 npm run build
```

After the command execution, a folder with name `build/` will be created inside the client's main code directory, containing the production bundle.

To use this bundle, the HTTP server needs to link the static pages and assets. This can be achieved by using the Express `static` function, which is a middleware function that allows to serve static files from specified paths or folders. Listing 6.7 shows how that is implemented in the FLOWViZ server's entry point module (`flowviz.js`).

Listing 6.7: Server using client static assets (`flowviz.js`)

```
1 // Uses client build version if in production
2 if (production) {
3   const buildDirectory = "../../client/build";
4   app.use(express.static(path.join(__dirname, buildDirectory)));
5   app.get("*", (req, res) => {
6     res.sendFile(path.join(__dirname, `${buildDirectory}/index.
7       html`));
8   });
9 }
```

To effectively use the HTTP server and the client simultaneously client pages' addresses and server's URIs must not collide. To do this, both server and client use path prefixes.

The server needs to create an Express Router, which is a subset of the Express App instance, and associate it with the path prefix, as shown in Listing 6.8.

Listing 6.8: Server path prefix

```
1 // Defining API route prefix
2 app.use("/flowapi", router);
```

This procedure was also performed inside the client, where the prefix was associated to the React-Router instance, as presented in Listing 6.9.

Listing 6.9: Client path prefix

```
1 <Router basename="/flowviz">
2 ...
3 </Router>
```

All client services were also updated, taking into account the path prefix of the server.

After the completion of all these steps, FLOWViZ could be finally published as a npm package* and can be implemented PHYLOViZ and other phylogenetic frameworks alike.

6.4.2 Phylogenetic framework integration

There are three ways of importing FLOWViZ into the phylogenetic framework: two of them require npm installed on the framework's host machine and the third is a standalone deployment.

If the phylogenetic tool is also a npm package, the developer can integrate FLOWViZ by either installing its npm package or cloning the source code and linking it to phylogenetic framework's package, as shown in Listings 6.10 and 6.11, respectively.

Listing 6.10: FLOWViZ npm package installation

```
1 npm i flowviz
```

Listing 6.11: FLOWViZ linking to phylogenetic framework

```
1 # In FLOWViZ source code main directory
2 npm link
3 # In the phylogenetic tool source code main directory
4 npm link flowviz
```

Either when installing the package or linking it, some errors related to dependency versions might occur. The developer must fix it in order successfully integrate the FLOWViZ package.

*npmjs.com/package/flowviz

If the integration is successful, FLOWViZ is now a dependency that can be required by the phylogenetic framework. FLOWViZ package's entry point (`flowviz.js`) requires a contract to be specified in the parameters when the dependency is required. This contract allows the developer to pass object instances of common dependencies, such as: `express`, `passport` or `mongoose`, already initialized by the phylogenetic framework. This way, FLOWViZ reuses the dependencies already being used by the phylogenetic framework. If some object instances are not specified in the contract, FLOWViZ starts them itself.

This can happen for most required dependencies, however, it can not happen for the `express` dependency. If this one is not provided by the phylogenetic framework, the deployment of FLOWViZ must be **standalone**, as it will initialize a new server instance. The same applies to `passport`: if this is not passed, FLOWViZ's deployment does not need to be standalone, but it will not reuse the phylogenetic framework's authentication system, requiring the users to create new accounts to use all the functionalities of FLOWViZ. The authentication must also be **stateless**, meaning that the server must supply authentication tokens to its authenticated clients, to allow them to access protected content.

If the phylogenetic framework is not a npm package and/or does not have the core dependencies used by FLOWViZ, the latter one has to be deployed detached from the phylogenetic framework's execution environment (standalone).

Either way, FLOWViZ requires some environment variables, containing sensitive content, such as secrets, IP addresses and passwords related with (i) the required database credentials, (ii) the Apache Airflow workflow system credentials and (iii) another meta-data related to the HTTP server. The required environment variables' configuration can be found inside the source code repository's *readme**.

If the FLOWViZ integration is done via npm package or npm link, a contract that passes the phylogenetic framework's dependencies instance objects must be specified in order to successfully extend it with FLOWViZ. The contract is presented by listing 6.12.

Listing 6.12: FLOWViZ npm package required dependencies for framework extension

```
1 require("flowviz")({
2   express: express,
3   app: app,
4   ...
5 });
```

The dependencies shown in listing 6.12 are the required ones. The `mongoose` and `passport` dependencies can also be passed, provided that the phylogenetic framework uses a MongoDB database and the `passport` dependency for authentication. The database's credentials must also be added as environment variables. Other dependencies can also be selectively passed. All the FLOWViZ dependencies and the full extension of the listing 6.12, can be found in the source code repository's *readme**.

*github.com/mig07/FLOWViZ/blob/main/README.md

Chapter 7

Conclusion

This chapter closes this thesis report, by pointing out fulfilled objectives and surpassed obstacles during the making of this project. The conclusion also states future work implementations that could take place in a future extension of this project.

In the context of this project it was defined a system architecture, which primary objectives were to allow seamless phylogenetic tool integration and provide workflow building to existent phylogenetic frameworks. This architecture was materialized by the creation of a system prototype called FLOWViZ, an integration framework composed by a web client and an HTTP server, that serves as middleware between the phylogenetic framework and the workflow system. The utilized workflow system was chosen through a selection of workflow systems, which contained favorable characteristics to the developing framework. In the end, the workflow system of choice was Apache Airflow due to its very complete set of features.

However, some difficulties arose during the project's implementation, primarily due to the chosen workflow system. The first difficulty was related to the REST API. Although it provided a very completed and well-documented documentation, it lacked an endpoint for DAG creation. This problem was solved with the implementation of the ETL pipeline for dynamic DAG creation, which is an Airflow DSL script triggered by an HTTP request that creates an Airflow DAG by crossing workflow information from the database with a DSL script template. The second problem was related with tasks' inputs and outputs. Tasks in Airflow are isolated from each other inside workflows, meaning that the data flow needs to be explicitly described, usually by the explicit execution order. Airflow provided a mechanism that allowed communication among the involved tasks, called *XComs*^{*} (short for "*cross-communications*"), which is implicitly used by selected operators and can also be explicitly used if the user desires. However, this mechanism only allows small amounts of metadata, meaning that if some tools generate files or large quantities of text information as output, they only can be sent with another solution of Airflow, that involves changing the metadata's database to other, such as: Google Cloud Storage[†], to allow storing large *XComs* blobs. Another way to circumvent this, is to deploy the necessary tools on the same

^{*} airflow.apache.org/docs/apache-airflow/stable/concepts/xcoms.html

[†] cloud.google.com/storage

container and only output the string paths of the generated files to next task(s) via *XComs*, however, this can drastically limit some workflows. The third problem was related with the CWL plugin that, unfortunately, was not implemented due to the plugin's outdate and also objectives' incompatibility, as this plugin is only an extension to support the CWL format and it is not meant to parse Airflow DSL scripts to the CWL format, but just the inverse. Because of this, the FLOWViZ's CWL support was considered a non-functional requirement (section 4.2) and was not implemented in this iteration.

The project objectives were fulfilled, FLOWViZ is now a ready-to-use integration framework that can be integrated with most phylogenetic frameworks, either by installing the dependency via npm or source code, or by deploying a standalone version of it. However, there are also some features that could be implemented in the future, which would improve either the current project's features, but also the source code's quality. The next paragraphs enumerate and explain some of the features that could be part of a future project's iteration.

During the development of this project, most data requirements could be fulfilled by using a *schemaless* database, which it is considered to still be a very valid approach. However, by the end of the project, some functionalities brought model *generalization* in the project's domain model, introducing data hierarchy, which requires a relational database to support this. Using a relational database would also improve data saving and some queries' efficiency through database normalization. It should be taken in consideration that, as previously stated, the database was not the project's main feature, so it was chosen to keep developing the project with a *schemaless* database.

The React web client also fulfills all the planned objectives and requirements, however, some client's improvements could be made, when it comes to the state and data management of the application. React already provides the necessary basic mechanisms to manage component state through the `useState` and `useContext` hooks, however, the provided mechanisms do not scale well when the state gets increasingly complex over time. When this happens it is recommended to use the `React-Redux`* dependency, which is a dependency that creates an application "global" state, that can be accessed by any component.

During phylogenetic tools integrations, it is required to deploy the tool in a container and specify how it can be accessed by the framework (access phase in tool contract), where the user specifies the docker url for the tool's container docker engine. This is good because the developed system does not need to download and execute the tool containers in its own container engine and allow users to have total control over their tools, however, it is not practical. For future work it was also considered associating a docker engine to Airflow, to allow users to only specify docker images that could be downloaded and executed as containers by our engine.

Airflow also has mechanisms to notify users about workflows executions' states via email. This could be easily implemented, has the user model from the server can also receive an email field to store users' emails.

*react-redux.js.org/

As the developed application is still a prototype, only a basic JWT authentication mechanism was implemented, where users can create accounts by only specifying their usernames and passwords. This does not provide enough safety to the system, as it can be easily targeted by bots, that will create spam accounts and overload the system's database and workflow system. Email verification would help mitigate this problem and even open up other possibilities and improvements for user security, such as two-factor authentication* (2FA).

It is also planned to implement the CWL plugin, if it suffers a new update, that does not imply users to downgrade their Apache Airflow version to a supported one, and that supports Airflow DSL scripts parsing to CWL.

*authy.com/what-is-2fa/

Bibliography

- [1] E. Afgan, D. Baker, B. Batut, M. van den Beek, D. Bouvier, M. Čech, J. Chilton, D. Clements, N. Coraor, B. A. Grüning, A. Guerler, J. Hillman-Jackson, S. Hiltemann, V. Jalili, H. Rasche, N. Soranzo, J. Goecks, J. Taylor, A. Nekrutenko, and D. Blankenberg. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update. *Nucleic Acids Research*, 46(W1):W537–W544, 05 2018. ISSN 0305-1048. doi: 10.1093/nar/gky379. URL <https://doi.org/10.1093/nar/gky379>.
- [2] P. Amstutz, M. R. Crusoe, N. Tijanić, B. Chapman, J. Chilton, M. Heuer, A. Kartashov, D. Lehr, H. Ménager, M. Nedeljkovich, et al. Common workflow language, v1. 0. 2016.
- [3] A. Bankevich, S. Nurk, D. Antipov, A. A. Gurevich, M. Dvorkin, A. S. Kulikov, V. M. Lesin, S. I. Nikolenko, S. Pham, A. D. Prjibelski, et al. Spades: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of computational biology*, 19(5):455–477, 2012.
- [4] A. Barker and J. v. Hemert. Scientific workflow: a survey and research directions. In *International Conference on Parallel Processing and Applied Mathematics*, pages 746–753. Springer, 2007.
- [5] A. Biryukov, D. Dinu, and D. Khovratovich. Argon2: new generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 292–302. IEEE, 2016.
- [6] A. Boc, A. B. Diallo, and V. Makarenkov. T-REX: a web server for inferring, validating and visualizing phylogenetic trees and networks. *Nucleic Acids Research*, 40(W1):W573–W579, 06 2012. ISSN 0305-1048. doi: 10.1093/nar/gks485. URL <https://doi.org/10.1093/nar/gks485>.
- [7] A. M. Bolger, M. Lohse, and B. Usadel. Trimmomatic: a flexible trimmer for illumina sequence data. *Bioinformatics*, 30(15):2114–2120, 2014.
- [8] J. A. Carriço, M. Crochemore, A. P. Francisco, S. P. Pissis, B. Ribeiro-Gonçalves, and C. Vaz. Fast phylogenetic inference from typing data. *Algorithms for Molecular Biology*, 13(1):1–2, 2018.
- [9] B. Dantas, C. Fleitas, A. P. Francisco, J. Simão, and C. Vaz. Beyond ngs data sharing and towards open science, 2017. URL <https://arxiv.org/abs/1701.03507>.

- [10] T. J. B. de Man, B. M. Limbago, and P. Dunman. Sstar, a stand-alone easy-to-use antimicrobial resistance gene predictor. *mSphere*, 1(1):e00050–15, 2016. doi: 10.1128/mSphere.00050-15. URL <https://journals.asm.org/doi/abs/10.1128/mSphere.00050-15>.
- [11] P. Di Tommaso, E. W. Floden, C. Magis, E. Palumbo, and C. Notredame. Nextflow: un outil efficace pour l’amélioration de la stabilité numérique des calculs en analyse génomique. *Biologie Aujourd’hui*, 211(3):233–237, 2017.
- [12] L. Finnigan and E. Toner. Building and maintaining metadata aggregation workflows using apache airflow. *Temple University Libraries*, 2021.
- [13] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.
- [14] A. P. Francisco, C. Vaz, J. Melo-Cristino, M. Ramirez, and J. A. Carriço. Phyloviz: Visualizing epidemiological information on phylogenetic relationships inferred by goeburst algorithm.
- [15] G. M. Kurtzer, V. Sochat, and M. W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20, 05 2017. doi: 10.1371/journal.pone.0177459. URL <https://doi.org/10.1371/journal.pone.0177459>.
- [16] J. Köster and S. Rahmann. Snakemake — a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 08 2012. ISSN 1367-4803. doi: 10.1093/bioinformatics/bts480. URL <https://doi.org/10.1093/bioinformatics/bts480>.
- [17] F. Lemoine, D. Correia, V. Lefort, O. Doppelt-Azeroual, F. Mareuil, S. Cohen-Boulakia, and O. Gascuel. NGPhylogeny.fr: new generation phylogenetic services for non-specialists. *Nucleic Acids Research*, 47(W1):W260–W265, 04 2019. ISSN 0305-1048. doi: 10.1093/nar/gkz303. URL <https://doi.org/10.1093/nar/gkz303>.
- [18] I. Letunic and P. Bork. Interactive Tree Of Life (iTOL) v5: an online tool for phylogenetic tree display and annotation. *Nucleic Acids Research*, 49(W1):W293–W296, 04 2021. ISSN 0305-1048. doi: 10.1093/nar/gkab301. URL <https://doi.org/10.1093/nar/gkab301>.
- [19] J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso. A survey of data-intensive scientific workflow management. *Journal of Grid Computing*, 13(4):457–493, 2015.
- [20] M. Luis and C. Vaz. Flowviz: Framework for phylogenetic processing, 2022. URL <https://arxiv.org/abs/2211.15282>.
- [21] D.-A. O. Mareuil F and M. H. A public galaxy platform at pasteur used as an execution engine for web services. 2017. doi: <https://doi.org/10.7490/f1000research.1114334.1>.
- [22] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

- [23] M. Nascimento, A. Sousa, M. Ramirez, A. P. Francisco, J. A. Carriço, and C. Vaz. PHYLOViZ 2.0: providing scalable data integration and visualization for multiple phylogenetic inference methods. *Bioinformatics*, 33(1):128–129, 09 2016. ISSN 1367-4803. doi: 10.1093/bioinformatics/btw582. URL <https://doi.org/10.1093/bioinformatics/btw582>.
- [24] C. Pan, G. McInnes, N. Deflaux, M. Snyder, J. Bingham, S. Datta, and P. S. Tsao. Cloud-based interactive analytics for terabytes of genomic variants data. *Bioinformatics*, 33(23):3709–3715, 07 2017. ISSN 1367-4803. doi: 10.1093/bioinformatics/btx468. URL <https://doi.org/10.1093/bioinformatics/btx468>.
- [25] N. Provos and D. Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.
- [26] B. Ribeiro-Gonçalves, A. P. Francisco, C. Vaz, M. Ramirez, and J. A. Carriço. Phyloviz online: web-based tool for visualization, phylogenetic inference, analysis and sharing of minimum spanning trees. *Nucleic acids research*, 44(W1):W246–W251, 2016.
- [27] T. Seemann. Prokka: rapid prokaryotic genome annotation. *Bioinformatics*, 30(14):2068–2069, 2014.
- [28] L. Silva. Library of efficient algorithms for phylogenetic analysis. *CoRR*, abs/2012.12697, 2020. URL <https://arxiv.org/abs/2012.12697>.
- [29] T. Stefan Van Dongen and B. Winnepenninckx. Multiple upgma and neighbor-joining trees and the performance of some computer packages. *Mol. Biol. Evol*, 13(2):309–313, 1996.
- [30] D. Talia. Workflow systems for science: Concepts and tools. *ISRN Software Engineering*, 2013, 01 2013. doi: 10.1155/2013/404525.
- [31] B. J. Walker, T. Abeel, T. Shea, M. Priest, A. Abouelliel, S. Sakthikumar, C. A. Cuomo, Q. Zeng, J. Wortman, S. K. Young, et al. Pilon: an integrated tool for comprehensive microbial variant detection and genome assembly improvement. *PloS one*, 9(11):e112963, 2014.
- [32] U. Yildiz, A. Guabtni, and A. H. Ngu. Business versus scientific workflows: A comparative study. In *2009 Congress on Services-I*, pages 340–343. IEEE, 2009.

Appendix A

Airflow workflow

```
1 from airflow import DAG
2 from datetime import datetime, timedelta
3 from airflow.providers.docker.operators.docker import DockerOperator
4
5 from docker.types import Mount
6
7 default_args = {
8     'owner'           : 'airflow',
9     'description'    : 'Trimmomatic -> Spades -> Abricate
    example pipeline',
10    'depend_on_past'  : False,
11    'start_date'      : datetime(2022, 2, 21),
12    'email_on_failure' : False,
13    'email_on_retry'  : False,
14    'retries'         : 1,
15    'retry_delay'     : timedelta(minutes=5)
16 }
17
18 with DAG('Trimmomatic-Spades-Abricate-DockerOperators', default_args=
    default_args, schedule_interval="5 * * * *", catchup=False) as dag:
19
20     trimmomatic = DockerOperator(
21         task_id='trimmomatic',
22         image='trimmomatic:latest',
23         api_version='auto',
24         mounts=[Mount(target='/fastq', source='/opt/.fastqTest', type
            ='bind')],
25         command='java -jar /NGStools/Trimmomatic-0.39/trimmomatic.jar
            \
26             PE -phred33 \
27             /fastq/sample1_1.fastq \
28             /fastq/sample1_2.fastq \
29             /fastq/sample1_R1_trimmed.fastq \
30             /fastq/sample1_R1_untrimmed.fastq \
31             /fastq/sample1_R2_trimmed.fastq \
32             /fastq/sample1_R2_untrimmed.fastq \
```

```

33             ILLUMINACLIP:/NGStools/Trimmomatic-0.39/adapters/
34                 TruSeq3-SE.fa:2:30:10',
35         auto_remove=True,
36         docker_url='unix:///var/run/docker.sock',
37         network_mode='bridge'
38     )
39     spades = DockerOperator(
40         task_id='spades',
41         image='spades:latest',
42         api_version='auto',
43         mounts=[Mount(target='/fastq', source='/opt/.fastqTest', type
44                 = 'bind')],
45         command='/NGStools/SPAdes-3.14.0-Linux/bin/spades.py -k
46                 21,33,55,77 --careful --only-assembler \
47                 --pe1-1 /fastq/sample1_R1_trimmed.fastq \
48                 --pe1-2 /fastq/sample1_R2_trimmed.fastq \
49                 -o /fastq/fq_spades.fasta',
50         auto_remove=True,
51         docker_url='unix:///var/run/docker.sock',
52         network_mode='bridge'
53     )
54     abricate = DockerOperator(
55         task_id='abricate',
56         image='abricate:latest',
57         api_version='auto',
58         mounts=[Mount(target='/fastq', source='/opt/.fastqTest', type
59                 = 'bind')],
60         command='abricate /fastq/fq_spades.fasta/contigs.fasta --db
61                 card --csv',
62         auto_remove=True,
63         docker_url='unix:///var/run/docker.sock',
64         network_mode='bridge'
65     )
66     trimmomatic >> spades >> abricate

```

Appendix B

Nextflow workflow

```
1 #!/usr/bin/env nextflow
2
3 params.saveMode = 'copy'
4 params.filePattern = '../.../res/trimmomatic_spades_abricate/fastq
   /*_{1,2}.fastq'
5 params.resultsDir = '.results'
6
7 Channel.fromPath(params.filePattern)
8     .set { ch_in_trimmomatic }
9
10 process trimmomatic {
11     container 'trimmomatic:latest'
12
13     input:
14     file 'inFastq' from ch_in_trimmomatic
15
16     output:
17     tuple path(fq_1_trimmed), path(fq_2_trimmed) into
        ch_out_trimmomatic
18
19     script:
20
21     fq_1_trimmed = 'sample1_R1_trimmed.fastq'
22     fq_1_untrimmed = 'sample1_R1_untrimmed.fastq'
23     fq_2_trimmed = 'sample1_R2_trimmed.fastq'
24     fq_2_untrimmed = 'sample1_R2_untrimmed.fastq'
25
26     '''
27     java -jar /NGStools/Trimmomatic-0.39/trimmomatic.jar \
28     PE -phred33 \
29     inFastq \
30     inFastq \
31     sample1_R1_trimmed.fastq \
32     sample1_R1_untrimmed.fastq \
33     sample1_R2_trimmed.fastq \
34     sample1_R2_untrimmed.fastq \
```

```

35     ILLUMINACLIP:/NGStools/Trimmomatic-0.39/adapters/TruSeq3-SE.fa
      :2:30:10
36     '''
37 }
38
39 process spades {
40     container 'spades:latest'
41
42     input:
43     tuple 'fq_1_paired.fastq', 'fq_2_paired.fastq' from
      ch_out_trimmomatic
44
45     output:
46     path 'fq_spades.fasta' into ch_out_spades
47
48     script:
49     '''
50     /NGStools/SPAdes-3.14.0-Linux/bin/spades.py -k 21,33,55,77 \
51     --careful --only-assembler \
52     --pe1-1 fq_1_paired.fastq \
53     --pe1-2 fq_2_paired.fastq \
54     -o fq_spades.fasta \
55     '''
56 }
57
58 process abricate {
59     container 'abricate:latest'
60     publishDir params.resultsDir, mode: params.saveMode
61
62     input:
63     path 'fq_spades.fasta' from ch_out_spades
64
65     output:
66     path 'abricate_result.csv' into ch_out_abricate
67
68     script:
69     '''
70     abricate fq_spades.fasta/contigs.fasta --db card --csv >
      abricate_result.csv
71     '''
72 }

```

Appendix C

Snakemake workflow

```
1 rule Trimmomatic:
2 shell:
3     """
4     docker run -v $HOME/.fastqTest:/fastq --workdir /fastq
5         trimmomatic:latest \
6     java -jar /NGStools/Trimmomatic-0.39/trimmomatic.jar \
7     PE -phred33 \
8     /fastq/sample1_1.fastq \
9     /fastq/sample1_2.fastq \
10    sample1_R1_trimmed.fastq \
11    sample1_R1_untrimmed.fastq \
12    sample1_R2_trimmed.fastq \
13    sample1_R2_untrimmed.fastq \
14    ILLUMINACLIP:/NGStools/Trimmomatic-0.39/adapters/TruSeq3-SE.
15    fa:2:30:10
16    """
17 rule Spades:
18 shell:
19     """
20     docker run -v $HOME/.fastqTest:/fastq --workdir /fastq spades
21         :latest \
22     /NGStools/SPAdes-3.14.0-Linux/bin/spades.py -k 21,33,55,77 \
23     --careful --only-assembler \
24     --pe1-1 /fastq/sample1_R1_trimmed.fastq \
25     --pe1-2 /fastq/sample1_R2_trimmed.fastq \
26     -o fq_spades.fasta \
27     """
28 rule Abricate:
29 output:
30     abricate_result='abricate_result.csv'
31 shell:
32     """
33     docker run -v $HOME/.fastqTest:/fastq --workdir /fastq
34         abricate:latest \
```

```
33     abricate /fastq/fq_spades.fasta/contigs.fasta --db card --csv  
34         > abricate_result.csv  
    ""
```

Appendix D

ETL pipeline

```
1 from airflow import DAG
2 from airflow.operators.python_operator import PythonOperator
3 from airflow.providers.mongo.hooks.mongo import MongoHook
4 from airflow.providers.mongo.sensors.mongo import MongoSensor
5 from datetime import datetime
6 import json
7 import shutil
8 import fileinput
9 import os
10
11 from docker.types import Mount
12
13 DAG_PATH = 'dags/'
14 PYTHON_EXT = '.py'
15 MONGO_CONN_ID = 'mongodb_flowviz'
16 DAG_UTILS = 'include/dag_utils.py'
17 DAG_TEMPLATE_FILENAME = 'include/dag_template.py'
18 MONGO_DB = 'test'
19
20 default_args = {
21     'owner' : 'airflow',
22     'description' : 'An ETL pipeline for DAG generation',
23     'start_date' : datetime.today(),
24 }
25
26 mongo_hook = MongoHook(conn_id = MONGO_CONN_ID)
27
28 # Extracts dag from mongodb, provided a dag_id,
29 # sent via HTTP to the REST API
30 def extract_dag_from_mongo(ti=None, **kwargs):
31     conf=kwargs['params']
32     dag_id=conf['dag_id']
33     username=conf['username']
34
35     workflow = mongo_hook.find(
36         mongo_collection = "workflows",
37         mongo_db = MONGO_DB,
```

```

38     query = {'dag_id': dag_id, 'username': username},
39     find_one = True
40 )
41
42 wflow = {
43     'dag_id': workflow['dag_id'],
44     'description': workflow['description'],
45     'username': workflow['username'],
46     'dag': workflow['dag']
47 }
48
49 ti.xcom_push(key='workflow', value=wflow)
50
51 # Transform the stored DAG into an Airflow DAG
52 def transform_and_load_data_into_airflow_dag(ti=None, **kwargs):
53     workflow = ti.xcom_pull(task_ids="extract_dag_from_mongo", key="
54         workflow")
55     dag_id = workflow['dag_id']
56     description = workflow['description']
57     dag = workflow['dag']
58
59     # Create file from template in include/ directory
60     new_filename = str(DAG_PATH + dag_id + PYTHON_EXT)
61     shutil.copyfile(DAG_TEMPLATE_FILENAME, new_filename)
62     generate_dag(dag_id, description, dag, new_filename)
63
64 dag_id = 'dag_generator'
65 schedule = '@once'
66 with DAG(dag_id, schedule_interval=None, default_args=default_args)
67     as dag:
68
69     extract_dag_from_mongo = PythonOperator(
70         task_id='extract_dag_from_mongo',
71         python_callable=extract_dag_from_mongo,
72         provide_context=True)
73
74     transform_and_load_data_into_airflow_dag = PythonOperator(
75         task_id='transform_and_load_data_into_airflow_dag',
76         python_callable=transform_and_load_data_into_airflow_dag,
77         provide_context=True)
78
79     extract_dag_from_mongo >>
80         transform_and_load_data_into_airflow_dag
81
82 # Generate the imports for the involved operators
83 def generate_imports(imports):
84    imps = ""
85     for imp in imports:
86        imps += str(imp + "\n")
87     return imps
88
89 # Generates the operator with the respective task
90 def generate_task(task):
91     t=""

```



```

89     operator_type = task['operator_type']
90     operator = str(str(operator_type) + "(paramsToReplace)")
91     operator_params = task['operator_params']
92
93     t += "task_id" + " = " + "'" + str(task['task_id']) + "',"
94
95     for key, value in operator_params.items():
96         if isinstance(value, str):
97             t += str("\n\t\t" + str(key) + " = " + str(value)+",")
98         else:
99             t += str("\n\t\t" + str(key) + " = " + str(value['
                operator_params']).replace("\'", "'")+",")
100
101     operator = operator.replace("paramsToReplace", t)
102     return operator
103
104 # Generate the DAG's tasks
105 def generate_tasks(tasks):
106     tks=""
107     for task in tasks:
108         tks += str("\t" + str(task['task_id']) + " = " + str(
            generate_task(task)) + "\n\n")
109
110     return tks
111
112 # Generates the DAG
113 def generate_dag(dag_id, description, daggraph, filename):
114     tasks = daggraph['tasks']
115     execution_order = daggraph['execution_order']
116
117     # Replace necessary fields with the passed configurations
118     for line in fileinput.input(filename, inplace=True):
119         line = line.replace("dagIdToReplace", "'" + dag_id + "'")
120         line = line.replace("descriptionToReplace", "'" + description +
            "'")
121         line = line.replace("startDateToReplace", daggraph['start_date
            '])
122         line = line.replace("importsToReplace", generate_imports(
            daggraph['airflow_imports']))
123         line = line.replace("operatorsToReplace", generate_tasks(
            tasks))
124         line = line.replace("executionOrderToReplace",
            execution_order)
125         print(line, end="")

```

Appendix E

HTTP server's REST API endpoints

Method	Path	Description
GET	/tool	Gets all the available integrated tools.
GET	/tool/:name	Gets all the details about an integrated tool.
POST	/tool/	Integrates a new tool.
GET	/workflow	Gets all workflows for a specific user.
GET	/workflow/:name	Gets all workflow's details.
GET	/workflow/:name/:dagRunId	Gets a DAG Run for a specific workflow.
GET	/workflow/:name/:dagRunId/ tasks/:taskInstanceId	Gets details about an executed task inside a DAG Run.
GET	/workflow/:name/:dagRunId /tasks/:taskInstanceId /logs/:logNumber	Gets a detailed log for a specific executed task inside a workflow's DAG Run.
POST	/workflow	Creates a workflow.
GET	/profile	Gets user's profile information.
POST	/register	Creates a new user.
POST	/login	Signs in an user.
POST	/logout	Signs out an user.