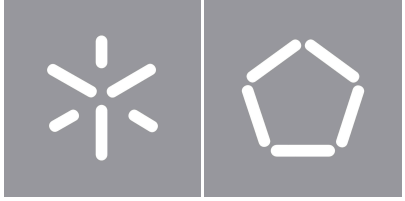


**University of Minho**
School of Engineering

André Gonçalves Vieira

# Study of Python-based Workflow Engines

october 2024



**University of Minho**
School of Engineering

André Gonçalves Vieira

# Study of Python-based Workflow Engines

Masters Dissertation
Master's in Informatics Engineering

Dissertation supervised by
**Ricardo Manuel Pereira Vilaça**

october 2024

# Copyright and Terms of Use for Third Party Work

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

## License granted to users of this work:



**CC BY**

https://creativecommons.org/licenses/by/4.0/

# Acknowledgements

I would like to begin by expressing my gratitude to my supervisor, Ricardo Vilaça. Your guidance, insightful feedback, support and, above all, patience have been crucial in the realisation of this dissertation. Without his assistance and dedication in every step of the process, this dissertation would have never been accomplished.

I would like to thank my family for their support, love and patience. To my parents, Fátima and José, who have always believed in me, your endless encouragement has been a source of strength throughout my academic journey. To my sister, Rita, I owe a special debt of gratitude, not only for teaching me to cook but also for your words of reassurance and belief in my potential were more valuable than I can ever express. To my dog, Juno, I thank all the company she gave me while writing.

I would like to thank all my friends and colleagues, who have made this entire journey more enjoyable and I will always be grateful for the experiences we shared along the way. I would also like to thank Sr. Manuel and D. Rosa, owners of Gota D'Água, where I have spent countless hours with my friends.

Thank you all for playing an essential role in this accomplishment.

# Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, october 2024

André Gonçalves Vieira

# Abstract

Scientists, engineers, and other professionals across various fields of study seek automated, fast, and cost-effective ways to execute their programs. However, in order to achieve this, it would be necessary to increase the complexity of applications and make constant changes to their code. Workflow management systems are tools capable of executing workflows in different environments, ranging from the local machine to HPC clusters and clouds.

Currently, there are several workflow management systems available, which are implemented using different approaches and can be executed on a wide variety of platforms. It is important to let users experiment to compare various tools in different environments, but there is a lack of tools available to conduct these experiments seamlessly.

This dissertation presents *Huginn*, a modular framework designed to automate the process of performance metrics collection of workflow executions using workflow management systems. *Huginn* enables users to create and execute synthetic workflow applications and generates reports with the metrics collected. This framework leverages the WfCommons Project, which has been expanded with two new translators to allow the creation of synthetic CWL and Snakemake workflow applications.

In order to validate our framework, we have created reports for a variety of test cases and conducted a comparative analysis of Toil, Snakemake and Nextflow. The objective of this analysis is to provide an impartial comparison guide to these mainstream workflow management systems in a multi-node environment with different workloads and workflow applications from various scientific domains.

**Keywords** Scientific Workflows, Workflow Management Systems, Common Workflow Language, Toil, Nextflow, Snakemake.

# Resumo

Cientistas, engenheiros e outros profissionais de várias vertentes procuram formas automáticas e rápidas de executar os seus programas. No entanto, para o conseguir, seria necessário aumentar a complexidade das aplicações e efetuar alterações constantes ao seu código. Os sistemas de gestão de fluxo de trabalho são ferramentas capazes de executar fluxos de trabalho em diferentes ambientes, desde a máquina local até clusters HPC e clouds.

Atualmente, existem vários sistemas de gestão de fluxo de trabalho disponíveis, que são implementados através de diferentes abordagens e podem ser executados numa grande variedade de plataformas. É importante permitir que os utilizadores façam experiências para comparar várias ferramentas em diferentes ambientes, mas há uma falta de ferramentas disponíveis para realizar essas experiências de forma simples.

Esta dissertação apresenta o *Huginn*, uma framework modular concebida para automatizar o processo de recolha de métricas de performance de execuções de fluxos de trabalho utilizando sistemas de gestão de fluxo de trabalho. O *Huginn* permite aos utilizadores criar e executar fluxos de trabalho sintéticos e gerar relatórios com as métricas recolhidas. Esta ferramenta recorre ao WfCommons Project, que foi enriquecido com dois novos translators para permitir a criação de fluxos de trabalho sintéticos escritos em CWL e Snakemake.

Para validar a nossa framework, foram criados relatórios para uma variedade de casos de teste e efectuada uma análise comparativa entre Toil, Snakemake e Nextflow. O objetivo desta análise é fornecer um guia de comparação imparcial para estes sistemas de gestão de fluxo de trabalho mais utilizados num ambiente multi-node com diferentes cargas de trabalho e tipos de fluxos de trabalho provenientes de vários campos científicos.

**Palavras-chave** Fluxos de Trabalho Científico, Motores de Fluxo de Trabalho, Common Workflow Language, Toil, Nextflow, Snakemake.

# Contents

**List of Figures** viii

**List of Tables** ix

**List of Acronyms** xi

**1 Introduction** 1

1.1 Context . . . 1
1.2 Motivation . . . 3
1.3 Goals . . . 4
1.4 Contributions . . . 4
1.5 Document Structure . . . 4

**2 State of the Art** 6

2.1 Background . . . 6
2.1.1 Python programming language . . . 6
2.1.2 Workflows Concepts and Scientific Workflows . . . 6
2.1.3 Scientific **Workflow Management System**s Overview . . . 8
2.1.4 **FAIR**ness in Scientific Workflows and **WMS**s . . . 11
2.1.5 Cross-Facility Workflows . . . 12
2.1.6 Workflow Applications . . . 12
2.2 Related Work . . . 20
2.2.1 **Workflow Management System**s . . . 20
2.2.2 **Workflow Management System**s Comparisons and Evaluation . . . 32

**3 Approach** 36

3.1 Scope Definition . . . 36

3.1.1 Requirements . . . 37
3.2 Architecture . . . 39
3.2.1 Huginn Framework . . . 39
3.2.2 Individual Modules . . . 41
3.3 Implementation . . . 43
3.3.1 SpecsGen . . . 43
3.3.2 Runner Scripts . . . 45

**4 Evaluation and Results** **49**
4.1 Tests . . . 49
4.1.1 Workloads and Test Workflows . . . 49
4.2 Results . . . 55
4.2.1 Elapsed Time . . . 55
4.2.2 Data Footprint . . . 58
4.2.3 CPU Utilisation . . . 59
4.2.4 Discussion . . . 60

**5 Conclusions and future work** **64**
5.1 Conclusions . . . 64
5.2 Prospect for future work . . . 64

# List of Figures


1 Example of a simple workflow. . . . 7
2 Basic Workflow structures . . . 8
3 Radar chart of workflow specification types and their scope. . . . 10
4 **FAIR** Principles [1] . . . 11
5 Portion of Galactic plane - Galactic Legacy Infrared Mid-Plane Survey Extraordinaire (GLIMPSE) [2]. . . . 13
6 The Montage workflow example [3]. . . . 14
7 Small **SIPHT** workflow example [3]. . . . 15
8 Small Cybershake workflow example [3]. . . . 16
9 Laser Interferometer Gravitational-wave Observatory workflow example [3]. . . . 17
10 Epigenomics workflow example [3]. . . . 18
11 A branch of a 1000Genome workflow example [4]. . . . 19
12 The DagOnStar architecture [5]. . . . 21
13 Basic FireWork's infrastructure [6]. . . . 22
14 A Galaxy server deployment with a remote execution site [7]. . . . 23
15 Parsl architecture [8]. . . . 24
16 Pegasus architecture [9]. . . . 25
17 PyCOMPSs architecture. The arrows depict the flow of information to launch a task with Python. The pink ball represents a Python object [10]. . . . 26
18 Nextflow architecture [11]. . . . 27
19 Snakemake architecture. . . . 28
20 StreamFlow architecture [12]. . . . 29
21 Toil architecture [13]. . . . 30
22 WfCommons architecture [14]. . . . 33

23 Activity Diagram of *Huginn* for report creation. . . . 39
24 *Huginn* Framework Architecture. . . . 40
25 Overview of *wfSpecsGen* module. . . . 41
26 Overview of *wfGenerator* module. . . . 42
27 Overview of *wfReportGen* module. . . . 43
28 Execution time of *wfbench.py* for multiple *cpu-work* and *percent-cpu* values with an input file of 1 KByte. . . . 50
29 Montage workflow **DAG** with 58 tasks generated with WFBench. . . . 51
30 Seismology workflow **DAG** with 8 tasks. . . . 52
31 1000Genome workflow branch **DAG** with 16 tasks. . . . 54
32 Mean execution time for each workflow application by **WMS**s . . . . 55
33 Mean Start Overhead for each workflow application by **WMS**s . . . . 56
34 Mean elapsed time (execution time + start overhead time) for each workflow application by **WMS**s . . . . 57
35 Mean Data Footprint for each workflow application by **WMS**s . . . . 58
36 Mean CPU utilisation of the orchestrator of each workflow application by **WMS**s . . . . 59

# List of Tables


1 Summary of the presented workflow applications and their characteristics. . . . . 20

2 Summary of the presented **WMS**s and their characteristics . . . . . 31

3 Parameters used in the Specification file . . . . . 52

4 Number of tasks in the Montage workflow . . . . . 52

5 Parameters used in the Specification file . . . . . 53

6 Number of tasks in the workflow . . . . . 53

7 Parameters used in the Specification file . . . . . 54

8 Number of tasks in *1000Genome* . . . . . 54

9 Comparative results of the **WMS**s analysed. . . . . 60

10 Summary of other characteristics that influence the choice of a **WMS**. . . . . 63

# Acronyms

**AI** Artificial Intelligence. 1, 2, 32

**CWL** Common Workflow Language. 1, 4, 10, 11, 28, 29, 33, 45, 56, 60, 61, 63

**DAG** Directed Acyclic Graph. 1, ix, 6, 26, 27, 34, 51, 52, 54, 56

**DSL** Domain Specific Language. 1, 9, 10, 26, 27, 31, 45, 56, 61, 63

**FAIR** Findable, Accessible, Interoperable, Reusable. 1, vi, viii, 2, 4, 11, 22, 37, 38, 47

**FITS** Flexible Image Transport System. 1, 13, 20

**HPC** High-Performance Computing. 1, 3, 12, 21, 22, 24, 27, 28, 29, 32, 33, 45, 55, 64

**IoT** Internet of Things. 1, 12, 21, 31

**LIGO** Laser Interferometer Gravitational-wave Observatory. 1, 16, 19, 20

**ML** Machine Learning. 1, 2

**SGT** Strain Green Tensor. 1, 20

**SIPHT** sRNA identification protocol using high-throughput technologies. 1, viii, 14, 15, 19

**WDL** Workflow Description Language. 1, 10, 11, 29, 33, 61, 63

**WMS** Workflow Management System. 1, vi, ix, x, 1, 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 40, 41, 42, 45, 46, 49, 51, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65

# Chapter 1

# Introduction

## 1.1 Context

Workflows have been used by researchers for almost two decades in order to address problems related to the management of complex distributed pipelines. As Workflow Management Coalition [15] defined, a workflow is the automation of a process in which data is passed regarding a defined set of rules to achieve an overall goal. This definition can be extended to both the business and science communities [16]. Divergent from business workflows, scientific workflows are data-oriented and intended to be shared with the scientific community to be re-utilised and foster other studies. Thus, scientific workflows are widely used in research projects, and modern scientific findings are becoming increasingly dependent on workflows [17, 18]. The significance of workflows in science has been demonstrated by the Nobel Prize-winning discovery of gravitational waves, which used **Laser Interferometer Gravitational-wave Observatory (LIGO)** data analysis workflows run by Pegasus **WMS** [19, 20]. Workflows are also widely employed in other important scientific domains such as drug design, material sciences, and, more recently, COVID-19 spread simulations. [21].

A substantial variety of programming languages is capable of describing and executing workflows. Typically, researchers opt for the Python [22] programming language due to the large number of available library modules and the ease it brings when writing more complex programs [23].

The volume of data available, used and generated by research projects increased exponentially over the years. Researchers started leveraging tasks' parallelism and other computing resources, such as **High-Performance Computing (HPC)**, cloud infrastructures and edge computing, in order to overcome this problem, yet other obstacles emerged [24]. Developing idempotent parallel scripts with dependencies between steps to be executed in multiple environments and binding tasks with their appropriate resources is challenging. Furthermore, parts of the workflow may fail and it is mandatory to provide fault-tolerance mechanisms [16, 25].

The orchestration of workflows became unattainable with the vast amount of data sources, computing environments and the parallelism between tasks involved. Thus, simple scripts were not able to fulfil the requirements needed to manage workflows since it is imperative to coordinate tasks executed in machines over the network [18, 26]. Furthermore, given the volume of data handled it became challenging to follow the **Findable, Accessible, Interoperable, Reusable (FAIR)** principles [27]. These principles serve as guidelines for sharing and publishing data (and other digital objects) enhancing knowledge discovering through computational resources and data reuse [27, 28]. Faced with the ever-growing complexity of the workflows, the usage of *Ad-hoc* scripts does not only compromise the **FAIR** data principles but also compromises the research's reproducibility, a key characteristic needed to validate findings/results [28, 24]. As a result, Workflow Management Systems began to be developed.

A **Workflow Management System (WMS)** is a tool used to define, create and execute workflows assisting users in monitoring and orchestrating scientific pipelines. Using these systems to automate scientific studies allows research professionals to efficiently handle workflows that involve thousands of simultaneous tasks and various data sources, whether in a single or multiple environments. Another essential feature is ensuring the reproducibility of workflows by capturing and documenting the provenance of data, facilitating the validation of research studies. Additionally, **WMS**s may provide other properties such as fault-tolerance, self-tunning and parallelism to existing tools [29, 30].

As scientific computing evolves, so do scientific workflows forcing **WMS** to enable new functionalities to achieve better results or introduce new types of workflows [31].

The ubiquity of **Artificial Intelligence (AI)** and **Machine Learning (ML)** methodologies within contemporary scientific practices is evident in workflow applications [32]. Scientists have increasingly adopted **WMS** to create or employ **AI** for problem-solving purposes. Additionally, research is exploring the usage of **AI**/**ML** techniques to assist in runtime orchestration and resource management, diverging from traditional workflows [31, 33, 34].

The prevalence of workflow engines' popularity has spurred the proliferation of numerous **WMS**s . Rather than fostering enhancements or the integration of pre-existing systems or workflow descriptions, organisations develop their own tools specifically to their use case, posing additional challenges in establishing standards in the workflow community[35]. Consequently, this has led to a surplus of underutilised systems [36].

## 1.2 Motivation

A shared objective across all research projects is to minimise both the costs and the time required to conduct studies. Despite the increasing investment in research, the need for more powerful and expensive computation has also increased. Likewise, the use of pay-as-you-go resources has grown in popularity to minimise costs and provide access to more powerful assets. Thus, it is mandatory to choose the most appropriate tools to achieve the best performance and efficiency possible [37].

Currently, there is a plethora of open source **WMS**s emerging from different scientific domains[1], with many distinct approaches regarding the portability, reproducibility, data provenance and usability [16]. The increasing complexity of scientific workflows calls for increasing computing power, usually provided by **HPC** clusters. There have been efforts to operate **WMS** even with Quantum Computing and to integrate it in **HPC** facilities as additional performance boosting devices [21, 38, 39].

Considering the computational requirements of workflows and the need to select suitable environments for their execution, it is imperative that **WMS** possess the capability to operate seamlessly across diverse and distinct sites, such as **HPC**, cloud or local infrastructures [40]. This heterogeneity has many challenges to the development of **WMS**, that were not taken into account due to their specialisation on a specific computing platform [41, 42]. Despite starting as a domain-specific tool, some **WMS**s evolved and began to incorporate fundamental needs from other research fields, providing researchers with a broader range of options.

Having these many options researchers need studies that compare these tools in different and hybrid environments using programs from various fields that analyse and conduct a comprehensive side-by-side comparison of **WMS**s to enhance the decision-making process in selecting the most suitable one for use.

Although there are a significant number of studies comparing **WMS**s [25, 43, 44, 45, 46, 47, 48], most of them address their characteristics and ease of use, regarding a specific domain. Since many users of **WMS**s have a limited set of skills and computer science knowledge, a substantial amount of exploratory work has been conducted concerning the learning curve of this type of software, disregarding performance metrics. Additionally, professionals make many comparisons in articles while presenting their tools, posing a challenge to a truly impartial analysis between tools [49].

Similar to all software, a **WMS** regularly gets updated and improved, supporting new computation environments and new features, so investigation work may face quick obsolescence, continually calling for more studies to help researchers choose tools that best meet their needs.

[1] Visit `https://github.com/pditommaso/awesome-pipeline` to see examples.

## 1.3 Goals

New unbiased and comprehensive studies are necessary to help researchers seamlessly adopt the **WMS** which better meets the requirements for their projects and hence improves their research methods and costs. Therefore, in this thesis, we encompass two primary objectives that inherently contribute to contemporary scientific research and assist professionals in selecting appropriate software resources.

Primarily, to provide an updated and impartial comparison guide of **WMS**s in multi-node environments, regarding their performance executing workflows with different loads and native to distinct science domains. During this process, we will evaluate and differentiate the features from the analysed **WMS**s, pointing out possible enhancements.

The other goal is to encourage impartial and independent research in the **WMS** domains, making a small framework to collect performance metrics in a reproducible way, fostering cooperation between science-focused developers towards **FAIR** work. This framework also aims to extend the creation of synthetic workflow applications for some **WMS**s specification languages.

From another perspective, this dissertation can also stimulate and encourage developers to improve or incorporate an existing **WMS** instead of creating a new one.

## 1.4 Contributions

From this dissertation results *Huginn*, a framework that automates the metrics collection of **WMS**s orchestrators, generating homogeneous reports with performance metrics. The framework is comprised of individual modules, which allows for seamless adaptation to other contexts and individual expansion. During the development process, *WfBench* was also expanded to generate synthetic workflow code for Snakemake and **CWL**.

Using *Huginn* to generate reports, we perform a comprehensive performance analysis and evaluation comparing Toil, Nextflow and Snakemake multi-node executions of synthetic *Montage*, *Seismology* and *1000Genome* workflow applications, providing insights into the suitability of these **WMS**s for large-scale computational workflows.

## 1.5 Document Structure

The remainder of this document is structured into four chapters.

Chapter 2, **State of the Art**, provides a comprehensive review of the existing literature and technologies related to **WMS**s. Here we describe the basic concepts of workflows and **WMS**s common across all domains. We explore current scientific workflow applications and **WMS**s used in managing these workflows across various computational environments, identifying the strengths and weaknesses of these tools. We also make

Chapter 3, **Approach**, the methodology and design of the proposed solution are discussed in detail. This chapter presents the approach taken in the design of a framework for the evaluation and comparison of **WMS**s, together with a set of requirements for the framework. The architectural and implementation details are also described here.

Chapter 4, **Evaluation and Results**, details the use cases of *Huginn* by conducting experiments to evaluate the performance of the selected **WMS**s with different workflow applications. This chapter provides detailed information about the workflow application test cases generated by the framework, the environments in which they were conducted, and the methodology used in the experiments. It also presents a comparative analysis of all the **WMS**s, with metrics collected using both *Huginn* reports and qualitative features.

Chapter 5, **Conclusions and future work**, summarises the key contributions of our work, limitations encountered and suggests potential future directions in this field of study.

# Chapter 2

# State of the Art

## 2.1 Background

### 2.1.1 Python programming language

Python [22] is an open-source, high-level and cross-platform programming language well known for its shallow learning curve and easy readability, making it novice-friendly and useful for researchers from other domains without computer science knowledge, such as life and health sciences, to learn [23].

Another of the numerous advantages of the language is the extensive collection of libraries available, rendering it a powerful tool in multiple domains with the simplicity provided by Python [50]. There are only a few drawbacks like the fact of being slower than compiled languages since it is an interpreted programming language [51].

This programming language grew in popularity in the past decade, having maintained its position in the first place of the *TIOBE Programming Community Index* [52] and placing second in *RedMonk* rankings [53]. Python is also the language with the highest number of pull requests and pushes in *GitHub*[1] since 2019 [54].

### 2.1.2 Workflows Concepts and Scientific Workflows

Workflows are often represented as a **Directed Acyclic Graph (DAG)** [55]. A graph is an arrangement of two finite sets: a set of edges and a set of vertices. If a graph has edges identified with an ordered pair of vertices (orientation), it is called a directed graph [56]. Thus, in typical a workflow representation, the vertices illustrate the tasks and the edges illustrate the data dependencies between them. In these representations, the input and output data are not represented. In figure 1 it is possible to visualise an example illustrating the representation of a simple workflow where all data produced and consumed by

[1] https://github.com

each task is displayed, thereby offering a more comprehensive understanding of the workflow dynamics. Here, data is represented by blue rectangles and the tasks by yellow circles. It is important to outline that this example is not a standard representation. In this example, the values **A** and **B** are multiplied originating **C** and the values **D** and **E** are multiplied originating **F**. Finally, using the results from both the multiplication tasks, the sum of the values **C** and **F** occurs, originating **E**. The values **A**, **B**, **C** and **D** are called the inputs of the workflows. The value **E** is the output since the workflow's goal is to create it [16, 55].

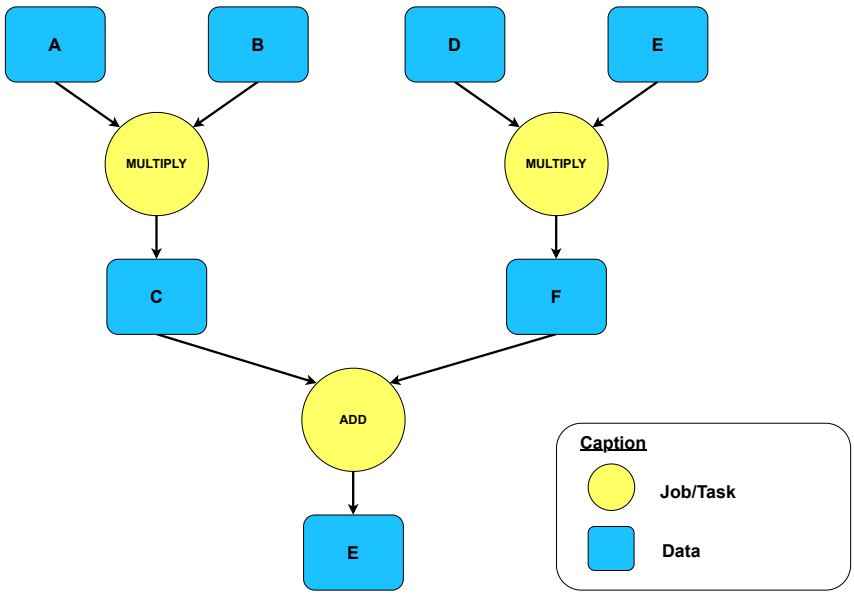



Figure 1: Example of a simple workflow.

The structure of a workflow can be depicted in two perspectives: the control plane and the data plane. Both types represent connections between tasks, although they have different meanings. In a control-driven or control-flow workflow, the connections represent the transfer of control from one task to another, including control structures such as conditionals or iterations. In contrast, data-driven or data-flow workflows exhibit the flow of data between tasks. Some representations try to merge both planes creating hybrid workflows [55]. Figure 1 is a data-driven workflow, as it represents the data flow between its tasks.

The workflow computation paradigm was initially used in business as a way to orchestrate services but rapidly aroused interest in the science community. Professionals in all research fields need to share, normalise, process, and analyse data, hence relying on workflows to ease the completion of their work [57]. These workflows manage the data flow and the desired applications. Each step or task corresponds to a specific process or computation to be carried out at a designated site, all within a potentially distributed environment [17]. They provide a systematic approach to describe the method used and facilitate the re-utilisation of workflows, thereby increasing productivity [58].

Scientific workflows differ from the ones used by the business community since they are data-intensive and are intended to be shared and reused. Scientific workflows often require changes, both in the simple case of changing a data source or in order to introduce/modify a task in the workflow [18].

Scientific workflows can be divided into 5 basic data-flow structures. The *Process* structure is a simple task executed that requires input data and produces an output. The *Pipeline* is a common structure among workflows, consisting in a sequence of *processes*, where the output of one job is the input of the next. *Data Distribution* produces multiple different outputs or is used for partitioning data to enable more parallelism. *Data Aggregation* combines data from multiple individual tasks creating a single output. The *Data Redistribution* takes multiple inputs and generates multiple outputs, often representing a synchronisation point in data processing. These simple patterns can be combined to create larger and more complex workflows [3]. In Figure 2 it is possible to visualise all the structures mentioned.

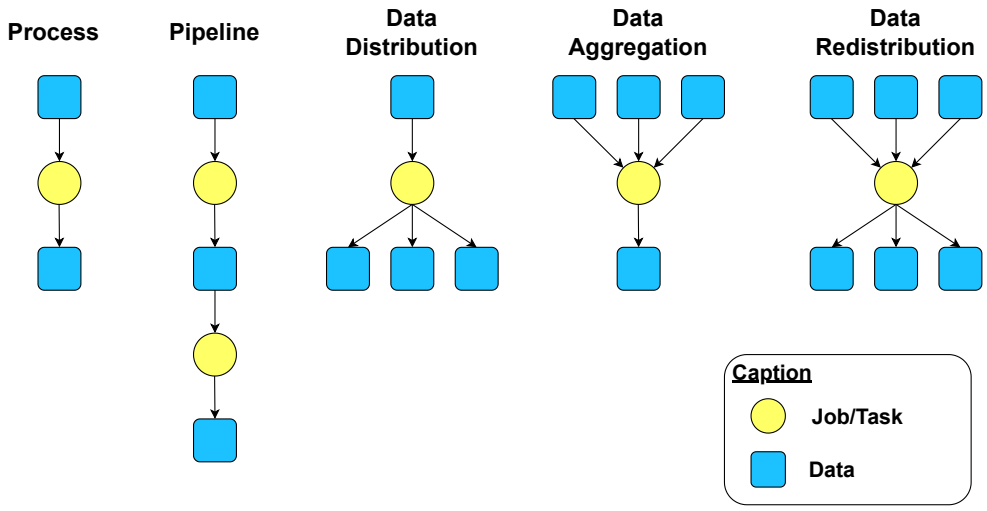



Figure 2: Basic Workflow structures

### 2.1.3 Scientific Workflow Management Systems Overview

The workflow life cycle, as described in the article [59], has four phases:

1. **Composition, Representation and Data Model**: creating the detailed description of tasks and data dependencies between them in a designated format.

2. **Mapping to Resources**: generating an executable workflow from the description and selecting the resources to run the tasks.

3. **Execution**: examination and monitoring of the workflow execution.

4. **Metadata and Provenance**: recording the process and history of creation of a data object.

**WMS**s facilitate the implementation of robust and complex analysis pipelines throughout their life cycle while simplifying and enhancing resource management. Furthermore, **WMS**s provide features such as fault tolerance, real-time monitoring, data provenance recording, portability and scalability of workflow executions [46].

One of the main objectives of using these tools in scientific research is to ensure that the workflows are self-documented and readable, which facilitates and encourages their reuse in other projects [60].

**WMS**s can define workflows in various forms, based on the development approach taken. Based on the way that workflows are represented, **WMS**s can be classified into five primary categories or groups [46, 49]:

- **Graphical WMSs**, e.g. Galaxy [61], are known for their ease of use, allowing users lacking programming knowledge to define and execute workflows through a point-and-click interface [46]. As expected, with the abstraction of the environment comes the limitation of tools available and control of the hardware resources used. The deployment of such tools is also more complex, compared to the other **WMS**s. To address this issue, these projects often provide instantly accessible computing services to execute pipelines [61].

- **Programming-library-based WMSs**, such as PyCOMPSs [10] and Parsl [62], use classes and functions of programming languages to implement their pipeline management systems. These **WMS**s leverage all the features and tools from the programming language and do not have other dependencies [63]. Thus, it allows a more expressive way of describing workflows. On the other side, programming skills are needed to use these systems and do not employ as many features as Graphical and **Domain Specific Language (DSL)**-based **WMS**s.

- **DSL-based WMSs** use **DSL** to describe workflows. These robust and flexible frameworks facilitate the integration of tools and scripts from a broad range of languages. **DSL**-based **WMS**s demand some initial programming knowledge from their users and their learning curve is steeper than Graphical **WMS**s. Snakemake [49] and Nextflow [64] are examples of this kind of **WMS**s.

- **Declarative WMSs**, like Popper [65], configure and describe workflows in a declarative way via configuration files such as YAML [66]. While this approach is simple and readable for users, it lacks flexibility, making it more restrictive than other types of specifications.

- **Standard Specification WMSs** employ system-independent workflow languages, like **CWL** [67] or **Workflow Description Language (WDL)** [68], separating the workflow definition from the executing environment. These systems promote interoperability between workflow engines, granting researchers a higher portability of their pipelines with a wider range of systems to choose from. Although easier to read, these languages are less flexible than **DSL**-based and Programming-library-based **WMS**s. Notable examples include StreamFlow [41] and Cromwell [69].

### Discussion

The workflow specifications utilised by **WMS**s present both advantages and disadvantages, representing trade-offs between ease of use, expressiveness and interoperability. Each project has its use case and decides upon one of these five niches that best aligns with its objectives.

In domains such as bioinformatics, the most popular **WMS**s are Graphical and **DSL**-based **WMS**s because of their shallow learning curve, providing automation of simple workflows with basic or no programming knowledge [46]. With the urge to share and reuse pipelines, standard specification **WMS**s have grown in popularity among scientists.

Users with programming expertise opt for Programming-library-based **WMS**s, as these systems have the full capabilities of their host programming language, enabling them to build more complex workflows [63]. Declarative **WMS**s are the least popular approach, being less used than the others.

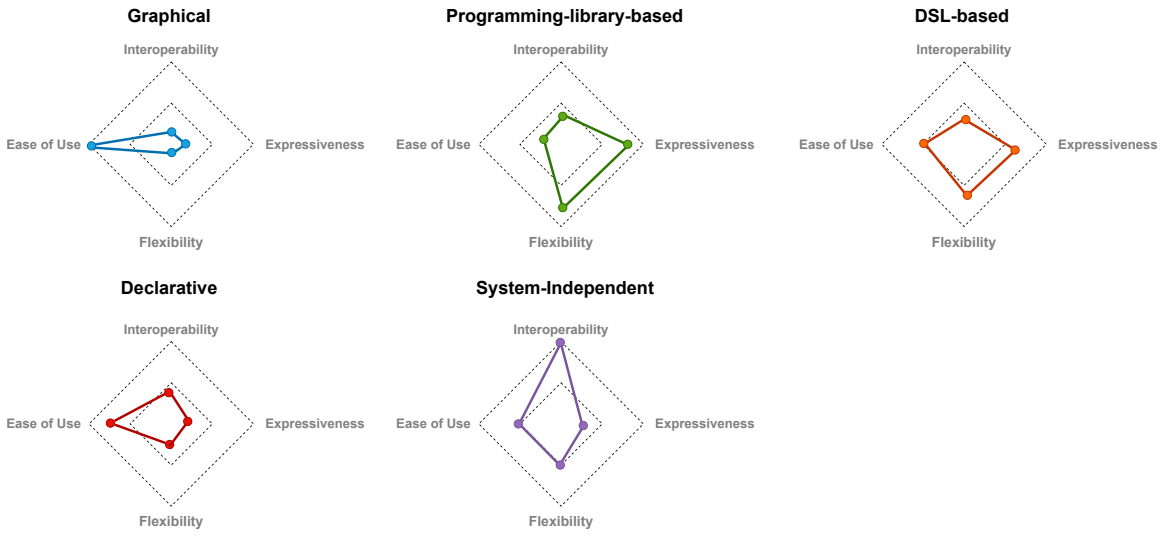



Figure 3: Radar chart of workflow specification types and their scope.

There is no silver bullet to specify a workflow, as all the approaches have advantages and disadvantages. Furthermore, as a **WMS** matures and its community grows, it implements language converters

and/or integrates its specification type with others [61, 70, 71]. This does not only mitigate its drawbacks but also fosters interoperability between **WMS**s [58]. In order to better understand the five types of workflow specification, we have created a Figure 3 that illustrates the scope of each type.

### 2.1.4 FAIRness in Scientific Workflows and WMSs

Firstly introduced to improve the re-use of academic data and later extended to digital objects in general, workflows can assist the **FAIR** Guiding Principles [27], not merely as data and meta-data producers but also as digital objects, enabling the re-use of the workflow itself [72]. Workflows are shared in online repositories and registries such as WorkflowHub [73], Dockstore [74], although the majority is either not properly registered or is available in general-purpose repositories as other software [72]. To address this issue, repositories for the major **WMS**s started to be developed, allowing researchers to seamlessly share pipelines, eliminating the need to redevelop existing workflows, and ensuring their maintenance and documentation [75].

These principles can be applied to **WMS**s as well, shaping their future development in order to conform to the **FAIR** principles, such as previously implemented data-sharing features. There have been attempts to make a standard workflow description language, such as **Common Workflow Language (CWL)** [67] and **Workflow Description Language (WDL)** [68], but different **WMS** have distinct features and capabilities, hence it is difficult to achieve a common general approach without mitigating its capabilities.

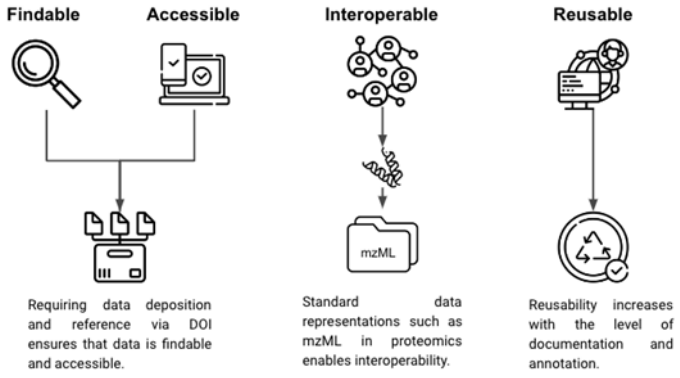



Figure 4: **FAIR** Principles [1]

There are still many challenges to building **FAIR** workflows and **WMS**s, given their distinct domains and recent evolution towards new paradigms, such as the lack of portability of workflows across multiple **WMS**s [31].

The goal of creating workflows that are transparent, reproducible, and accessible to the scientific

community is a shared objective among numerous professionals and originating communities, such as the Workflow Community Initiative [76]. This initiative brings developers, users and trainees together, acting not only as a source of knowledge in the **WMS** world but also as a united effort to understand and solve workflow challenges.

### 2.1.5 Cross-Facility Workflows

Recently, cross-facility workflows have emerged, a paradigm whose definition is not yet final due to its recent breakthrough [31].

Workflows are considered cross-facility when they use more than one computational site, which may include multiple or a combination of local computers, **HPC** facilities, cloud, edge, campus cluster, and sensors at the edge [42].

This paradigm enables each workflow step to be executed in the most suitable environment. It allows a trade-off between the cost of moving the required data between sites and the performance gained with the execution of the step in the new environment.

An approach to address these workflows is for the **WMS** to do matchmaking between task requirements across facilities, raising questions of what metadata is required to describe tasks [31].

Another approach is hybrid workflows, exploiting heterogeneity by adding the explicit definition of execution environments, available resources, and communication channels to the definition of steps and data dependencies. This approach introduces complexity and requires knowledge of distributed execution environments [31].

This workflow paradigm is often connected to **Internet of Things (IoT)** environments, because of the constant need to transform data from multiple sources, sometimes combining numerous computational environments. The constant need to transform data from multiple sources, especially in these complex, multi-environment workflows, can be streamlined using **WMS**s. These systems enable seamless orchestration of data transformation tasks across the computing continuum, from edge to cloud to **HPC**, to ensure optimal resource utilisation, automation and reproducibility in data processing and scientific computing. [77].

### 2.1.6 Workflow Applications

There is a myriad of workflow applications from many different science domains. As previously mentioned, most workflows are not publicly available, resulting in redundant implementations of the same analysis pipelines by different developers [46]. **WMS**s present or enable users to publish their workflows

implementations as guides, e.g. Pegasus [20] **WMS** has a workflow gallery[2] with some implementation examples of different workflows ready to be used.

In this section, we will present some of the most popular workflows in computational and data-intensive science research. These workflows are typically named after the tools they are used with or the projects they originate from. Figure 2 illustrates the data structures visible in the workflows and pointed out in the graphic representation of the presented workflows. Figures 6, 7, 8, 9 and 10 are original from the article [3].

### Montage [78] (Astronomy)

Montage is an open-source application developed by NASA/IPAC Infrared Science Archive used to assemble **Flexible Image Transport System (FITS)** [79] images into custom image mosaics to conduct astronomical research or other displays, thus being a data-intensive workflow [80]. It has been used to generate mosaics from data released by sources such as the Spitzer Space Telescope, and the famous Hubble Space Telescope [78]. The application size is proportional to the number of images used. Figure 5 depicts an example of a Galactic plane measured with the Spitzer Space Telescope and assembled by Montage [78].



Figure 5: Portion of Galactic plane - Galactic Legacy Infrared Mid-Plane Survey Extraordinaire (GLIMPSE) [2].

Figure 6 shows an example of the Montage workflow and a small description of each task.

[2] https://pegasus.isi.edu/workflow_gallery/

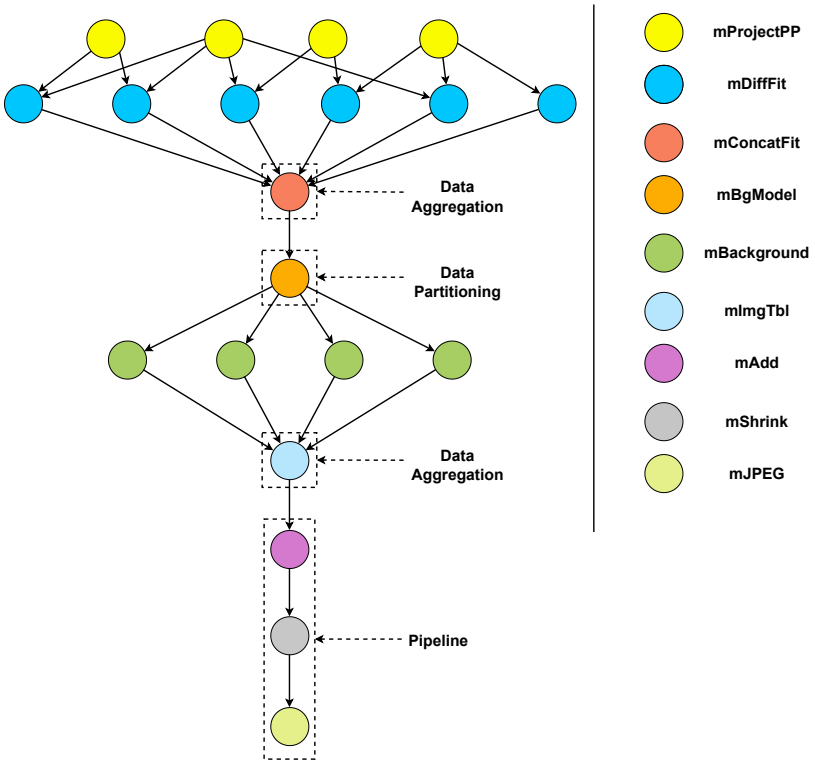



Figure 6: The Montage workflow example [3].

### SIPHT [81] (Bioinformatics)

**sRNA identification protocol using high-throughput technologies (SIPHT)** is a tool developed at Harvard University for conducting kingdom-wide predictions and annotations of intergenic sRNA-encoding genes [81]. This program uses a workflow to automate the search for sRNA-encoding genes for all bacterial replicons in the National Center for Biotechnology Information database [82]. The kingdom-wide prediction and annotation of sRNA encoding genes involves a variety of programs that are executed in the proper order using Condor DAGMan [82, 83]. Most of the tasks in this workflow have a high CPU and low I/O utilisation [3]. **SIPHT** workflows have identical structures, varying only in the number of *Patser* tasks. Of all the jobs in the workflow, the *BLAST* jobs are the most computation-intensive [82]. In Figure 7 it is possible to see a small **SIPHT** workflow depicted.

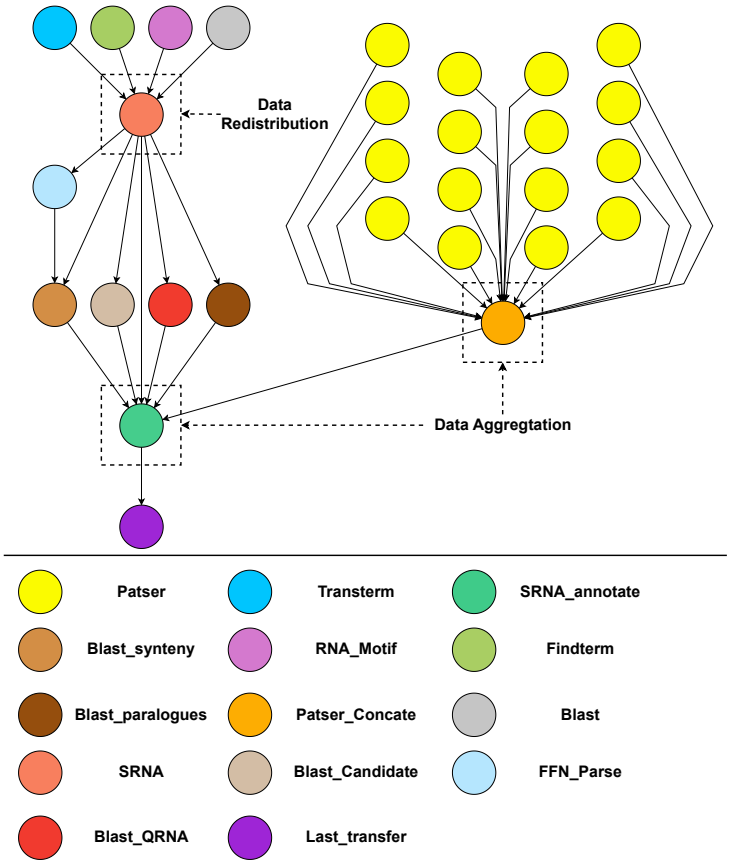



Figure 7: Small **SIPHT** workflow example [3].

### CyberShake [84] (Seismology)

CyberShake is a Southern California Earthquake Center research project that developed a physics-based computational approach to probabilistic seismic hazard analysis to characterise earthquake hazards in a region [85]. In Figure 8 it is possible to see a small Cybershake workflow depicted. This workflow starts with *ExtractSGT* jobs, that extract from a large file, thus being a data partitioning task. The tasks of *ZipSeis* and *ZipPSA* can be considered basic data aggregation jobs. Although simple in structure, this type of workflow may perform significant amounts of computation and generate a substantial amount of data.

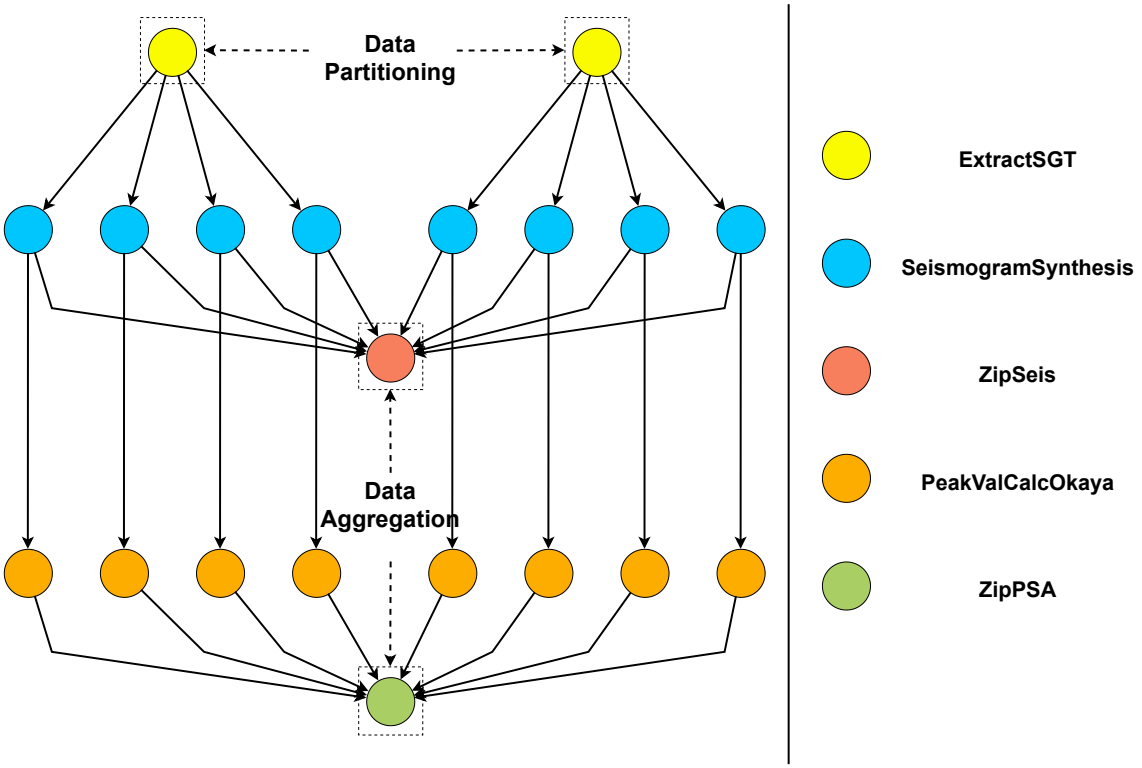



Figure 8: Small Cybershake workflow example [3].

### LIGO Inspiral Analysis [86] (Astrophysics)

The **Laser Interferometer Gravitational-wave Observatory (LIGO)** workflow was designed to open the gravitational-wave astrophysics field by directly detecting gravitational waves predicted by Einstein's General Theory of Relativity [87]. The **LIGO** Inspiral Analysis workflows are used to detect gravitational waves produced by violent events in the universe, such as the collision of two black holes or the explosion of *supernovae* [86]. This is a complex workflow comprised of four distinct workflows: *DataFind*, *Full*, *Injection* and *CAT*. Figure 9 illustrates a simplified version of a **LIGO** workflow and illustrates its sub-workflows. In this workflow, *Inspiral* jobs are the most computationally intensive jobs in the workflow. This type of workflow has large memory and resource requirements [82, 3].

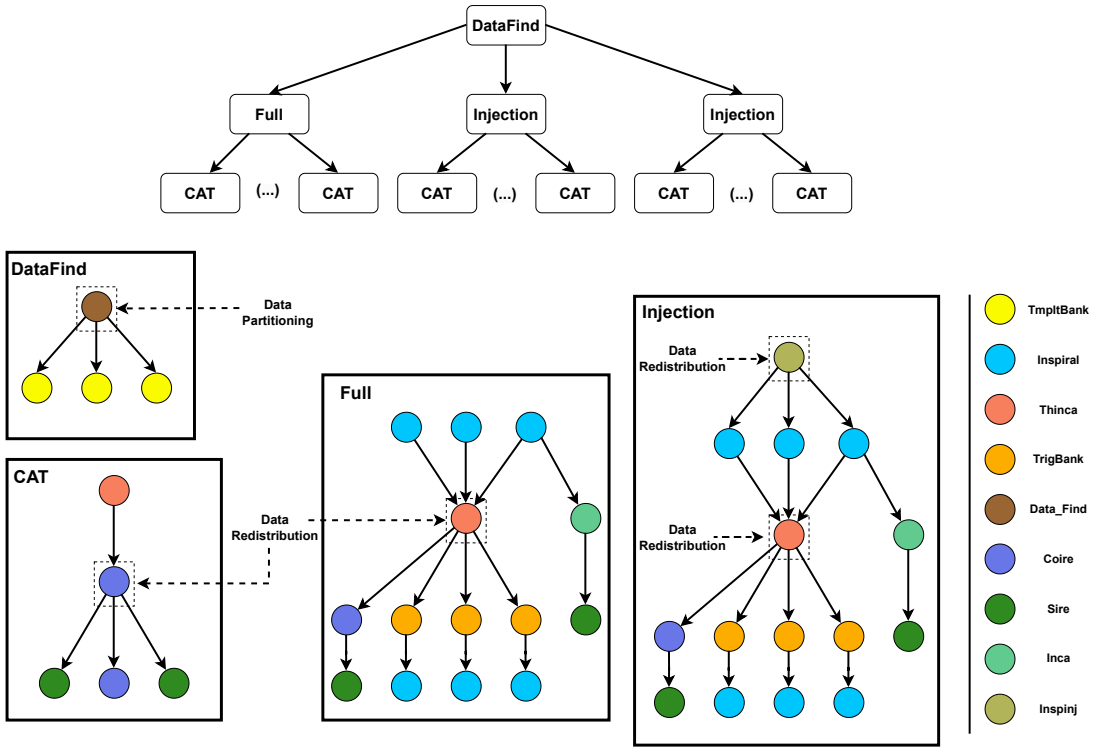



Figure 9: Laser Interferometer Gravitational-wave Observatory workflow example [3].

### Epigenomics (Bioinformatics)

The Epigenomics workflow is used to analyse the epigenetic status of human cells. It contains a sequence of automated operations to collect short DNA segments using high-throughput gene sequencing machines and it uses MAQ [88] software and Illuminia-Solexa [89] Genetic Analyser. It is a highly-pipelined application with several independent data chunks processing simultaneously [82]. The remaining steps of the process involve removing any noisy and contaminating sequences, correctly mapping the remaining sequences onto a reference genome, creating a global map, and finally determining the sequence density at each position in the genome [3]. The workflows of this type involve tasks that require a lot of CPU power but fewer I/O operations. In Figure 10 the structure of the workflow shows the pipelines previously described.

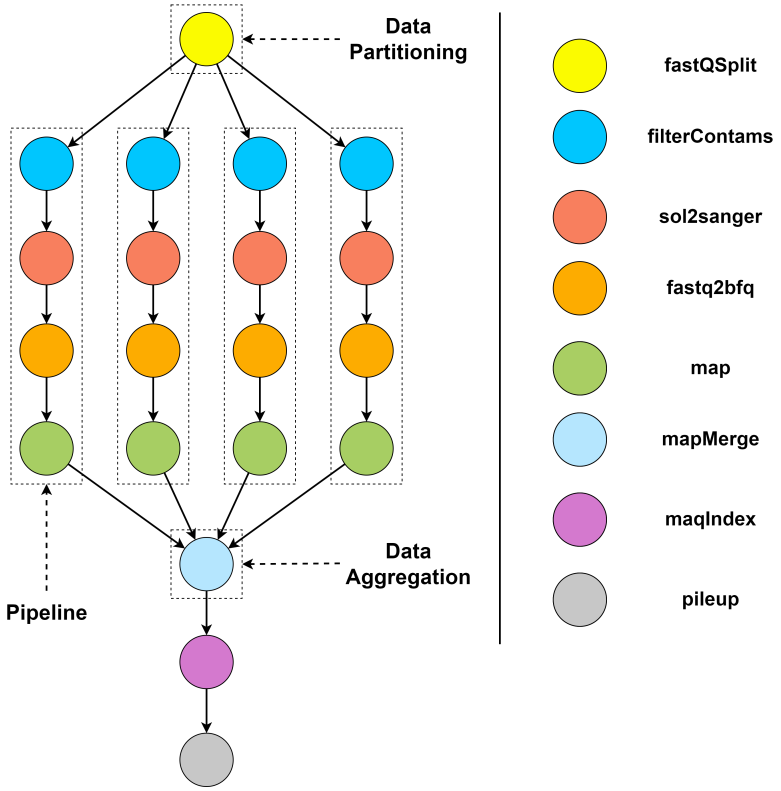



Figure 10: Epigenomics workflow example [3].

### 1000Genome (Bioinformatics)

The 1000 Genomes Project, active from 2008 to 2015, was a project to find common genetic variants with frequencies of at least 1% in the populations studied. With 2,504 individual genomes reconstructed from 26 different populations, this research created the biggest public catalogue of human variation and genotype data [90, 91]. This workflow identifies mutational overlaps using data from the 1000 Genomes Project to provide a null distribution for rigorous statistical evaluation of potential disease-related mutations [92]. This is a Data-intensive workflow, particularly in the *individuals* tasks which have a high disk and memory demand and make 81.85% of the workflow execution time [4, 92, 93]. In Figure 11 is depicted one branch of the 1000Genome workflow.

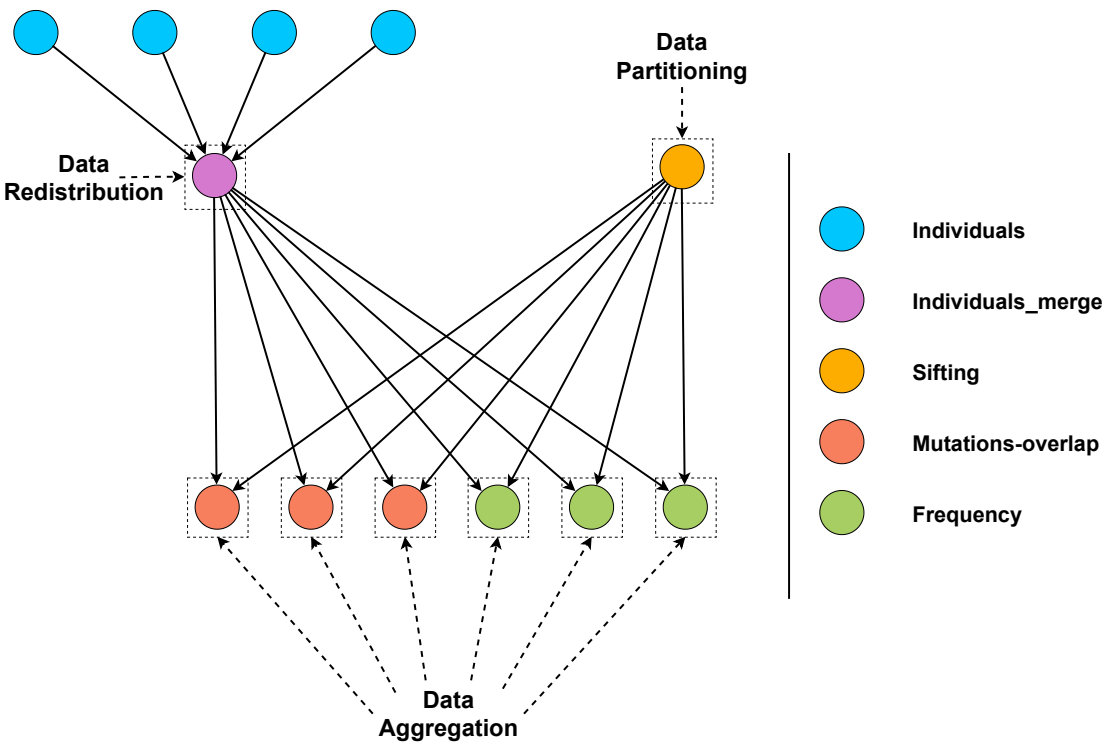



Figure 11: A branch of a 1000Genome workflow example [4].

### Summary

The workflows presented have different characteristics and resource requirements, thus it is possible to assume an overall type of each workflow. It is important to note that this type is just an overview of the workflow, as each task has different requirements.

**SIPHT**, Montage and Epigenomics workflows are mainly CPU-intensive, while the others are data-intensive. Montage is also I/O-bound as it spends a substantial amount of time in I/O operations. **LIGO** and Cybershake are the largest workflows, both with a massive standard number of tasks and large memory and resource requirements, that despite being data-intensive also have tasks with relatively high CPU utilisation. **SIPHT** and 1000Genome typically have a smaller number of tasks compared to the other workflows [82, 93].

Table 1 shows an overview of the main features of each workflow. With these examples, we cover the major workflow usage domains [94].

| Workflow | Number of Tasks* | Domain | Basic Type | Other notable characteristics |
|---|---|---|---|---|
| **Montage** | 300 - 10000[1] | Astronomy | CPU-intensive | I/O-bound |
| **SIPHT** | 30 - 50[2] | Bioinformatics | CPU-intensive | - |
| **CyberShake** | >800000[3] | Seismology | Data-intensive | Large resource and memory requirements |
| **LIGO Inspiral Analysis** | 100000 - 200000[4] | Astrophysics | Data-intensive | Large resource and memory requirements |
| **Epigenomics** | 500 - 2000[5] | Bioinformatics | CPU-intensive | - |
| **1000Genome** | 100 - 1000[6] | Bioinformatics | Data-intensive | - |

* This is the typical number of tasks, it is possible to find workflows with any number of tasks

[1] depends on the number of input **FITS** images.

[2] depends on the number of inputs describing transcription factor binding sites.

[3] depends on the number of **SGT** datasets.

[4] depends on input data size from the **LIGO** detectors.

[5] depends on input size and data partitioning factor.

[6] depends on the number of chromosomes analysed.

Table 1: Summary of the presented workflow applications and their characteristics.

## 2.2 Related Work

### 2.2.1 Workflow Management Systems

**Workflow Management System**s provide a programming environment to ease the development of workflows used by scientists to conduct their research, as previously mentioned. Currently, common features of all workflow engines are automation of tasks and workflow deployment, portability and scalability between different environments, fault tolerance mechanisms and reproducibility of workflows. These tools often started as specialised **WMS** from a particular research domain, having unique architectures and implementation details.

The various architectures offer or enable the incorporation of unique features related to the primary purpose of the **WMS**. Some are tailored for specific environments or a combination of various or specialised workloads from specific scientific domains. We present examples along with a brief introduction

of **WMS**s studied and used for scientific purposes [59].

### DagOnStar [5]

DagOnStar, also known as DagOn*, (Direct acyclic graph On anything) is a general-purpose, production-oriented **WMS** implemented as a lightweight Python library. Its main objectives rely on providing a flexible tool capable of deploying workflows on any combination of different infrastructures with minimal data footprint and avoiding engine-centred data management [95]. Initially, DagOnStar was created specifically for use in weather and marine forecasts, which commonly use **IoT** devices such as small sensors. The strengths of **WMS**s are thus evident in handling continuum computing workflows and its minimal footprint in terms of storage room used for the external software components' execution. The workflows in DagOnStar are written in Python with an object-oriented programming approach having the possibility to have error handling mechanisms and dynamic procedures. The **WMS** only accepts SLURM as a scheduler for **HPC** systems.

Represented by *Runtime* in Figure 12, which deploys the tasks of a workflow and delivers data to each respective task and a *Service* which monitors the execution of the workflow and handles the tasks' state. It includes a Stager that manages data stage-in and stage-out, along with a Garbage Collector that reduces the data footprint of workflow tasks. The architecture of DagOnStar, alongside the type of tasks, is present in Figure 12.

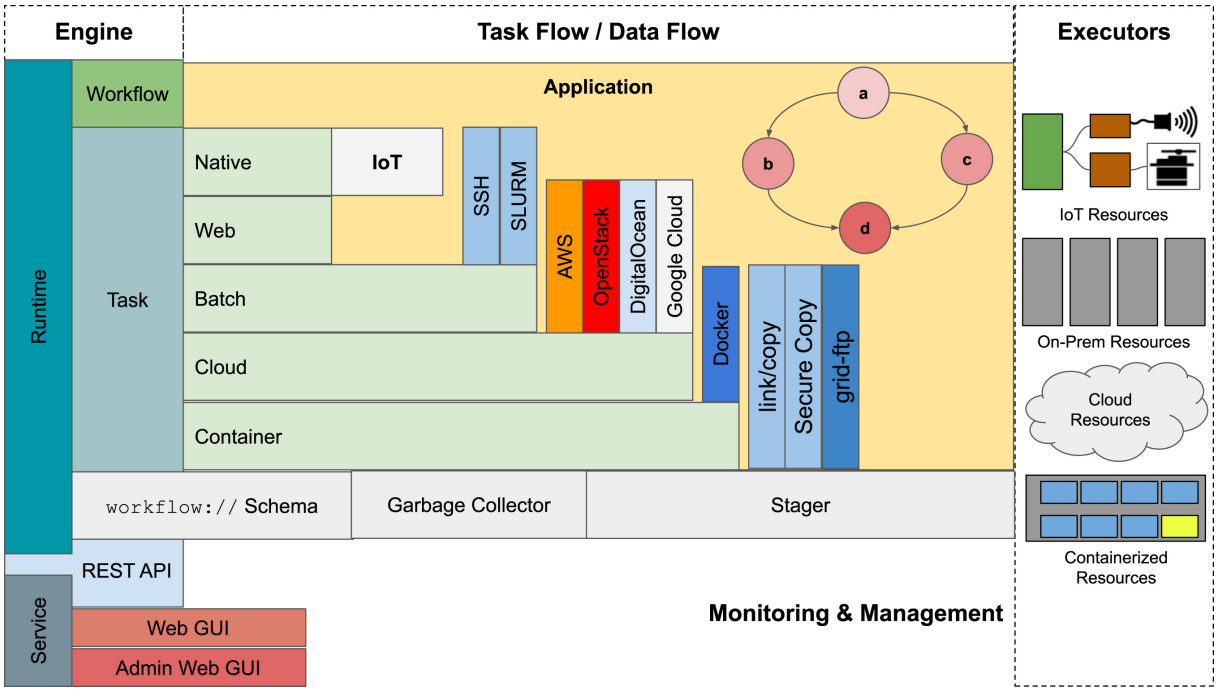



Figure 12: The DagOnStar architecture [5].

### FireWorks [96]

FireWorks is an open-source **WMS** planned to be user-friendly with a shallow learning curve while enabling the usage of more complex workflows. It allows the definition of workflows in Python, JSON, or YAML, task parallelism within and across systems and supports dynamic workflows. FireWorks is database-backed, using a MongoDB database to add, remove and get the status of workflows [96]. This **WMS** has built-in tools and execution modes for running data intensive jobs in large-scale computing centres, like the job packing feature which aggregates numerous small jobs into a larger ensemble, effectively utilising a substantial number of computation nodes/cores. For **HPC** systems, the **WMS** allows users to use schedulers such as PBS/Torque, Sun Grid Engine, SLURM, and IBM LoadLeveler.

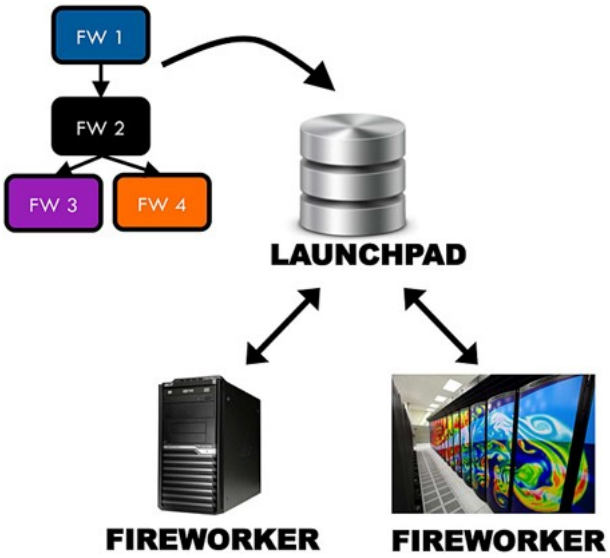



Figure 13: Basic FireWork's infrastructure [6].

FireWorks has a centralised server/worker model. The server is called *LaunchPad* and is capable of launching, querying the state and rerunning workflows. This is where the database is deployed and acts as a centralised queue where tasks are queried for execution. In order to launch a workflow, users add a workflow to the *LaunchPad*. The worker(s) are known as *FireWorker(s)* and run the workflow's tasks in their designated location defined by the user. Figure 13 exemplifies a basic infrastructure with two *FireWorkers*.

### Galaxy [61]

Galaxy is an open-source platform, community-driven and web-based focused on **FAIR** data analysis, allowing and providing ways to manage and share data and analysis seamlessly. The Galaxy Project provides services, public infrastructure to run workflows free of charge and a plethora of training resources, from tutorials to computational resources especially used for educational purposes. It is a **WMS** designed

for users with limited computer science expertise with a graphical interface to design workflows, abstracting them from implementation details. Initially, this platform was used as a bioinformatics **WMS** but later expanded to other domains, offering numerous tools integrated and giving the possibility for users to integrate more [61].

Although mainly used on Galaxy Project large public servers across the world, it is also possible to deploy a Galaxy server on on-prem resources. This option is usually utilised to process large datasets too big for public servers or when sensitive data is used (e.g. clinical) [97]. The **WMS** supports many schedulers, such as Torque, Slurm, HTCondor and Altair PBS Professional. There is also Galaxy Pulsar [98] to function as a job runner for Galaxy servers.

Without delving into intricate details, Figure 14 depicts an example of a Galaxy server installation with a remote execution site.

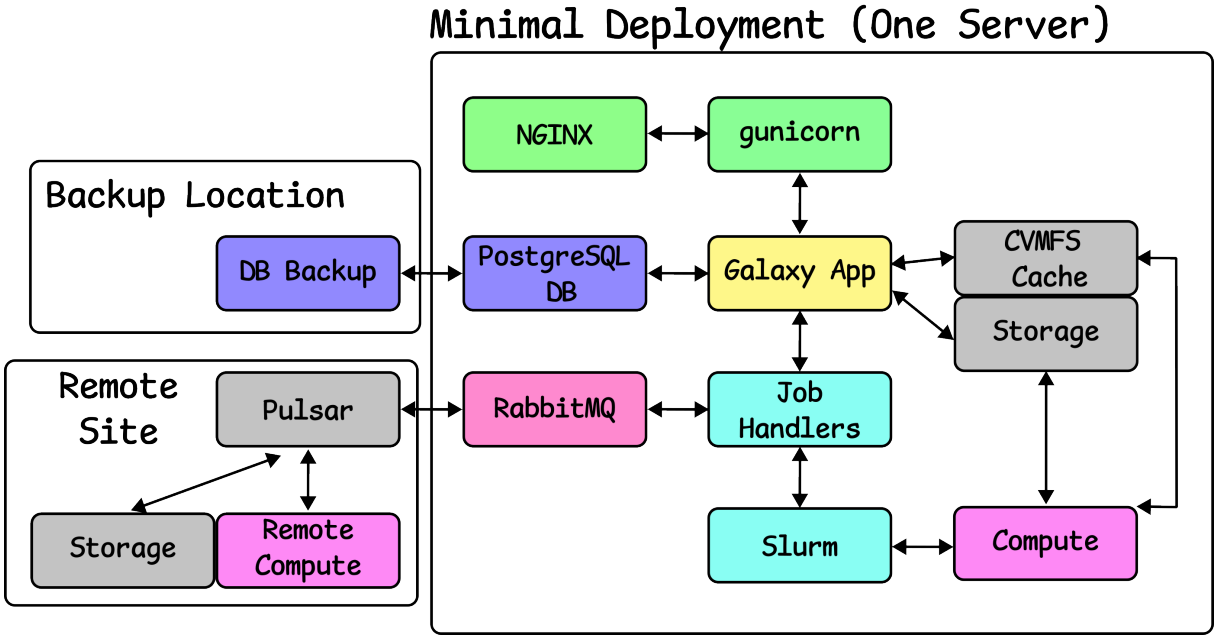



Figure 14: A Galaxy server deployment with a remote execution site [7].

### Parsl [62]

Parsl (Parallel Scripting Language) is a scripting Python library that extends parallelism to a multi-machine environment and supports the execution of asynchronous data-oriented workflows. Workflows in Parsl are defined through Python scripts where functions that can be executed concurrently are annotated with Python Decorators as *Apps*. Thus, the **WMS** creates a dynamic data dependency graph, exploring parallelism where it is possible [62]. Data in this **WMS** is passed between *Apps* using data *futures*. *Futures* block the *app* execution until the data of a *future* is written, enabling asynchronous executions. This

**WMS** has 3 main components that are illustrated in figure 15: the *Providers*, the *Executors* and the *DataFlow Kernel*. Parsl enables heterogeneous environments (e.g. clouds, supercomputers, and local machines) resources through *Providers*. These are expressed in the configuration and have information about the details of the connection channel, allocation size, queues, durations, and data management options [62]. For **HPC** clusters Parsl uses built-in *providers* for distributed resource management, such as Slurm, Torque/PBS and GridEngine. The Executors are responsible for executing the *Apps* in one or more executing sites for distinct workloads, regarding the type of computation needed for the workflow.

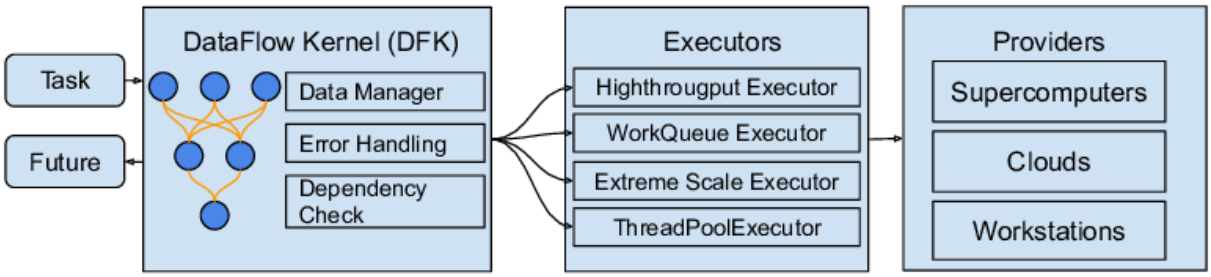



Figure 15: Parsl architecture [8].

The *DataFlow Kernel* is the main component of Parsl, and creates and manages the execution of workflows and tracks data dependencies between *Apps*. Figure 15 depicts the main components of Parsl.

### Pegasus [20]

Created in 2001, Pegasus is one of the oldest **WMS** in use. It is used in many scientific domains such as astronomy, bioinformatics, seismology and astrophysics. This **WMS** is built on top of the HTCondor task management system and DAGMan workflow engine [83], providing users with a higher level of abstraction. A workflow can be used in distinct environments and across heterogeneous resources.

Pegasus has a *Mapper* that generates the workflow from the abstract specification and, as the name suggests, maps resources, tasks and data. The tasks of the new workflow are then submitted and tracked by the *Local Workflow Execution Engine* provided by DAGMan, according to data dependencies. Pegasus also presents a *Remote Workflow Execution Engine* that manages the execution of tasks in remote resources and can structure them as sub-workflows. The *Job scheduler* manages, supervises and provides task-level reliability. The **WMS** monitoring component monitors the progress of the workflow and stores information about data provenance and performance [20]. Figure 16 represents the architecture of Pegasus.

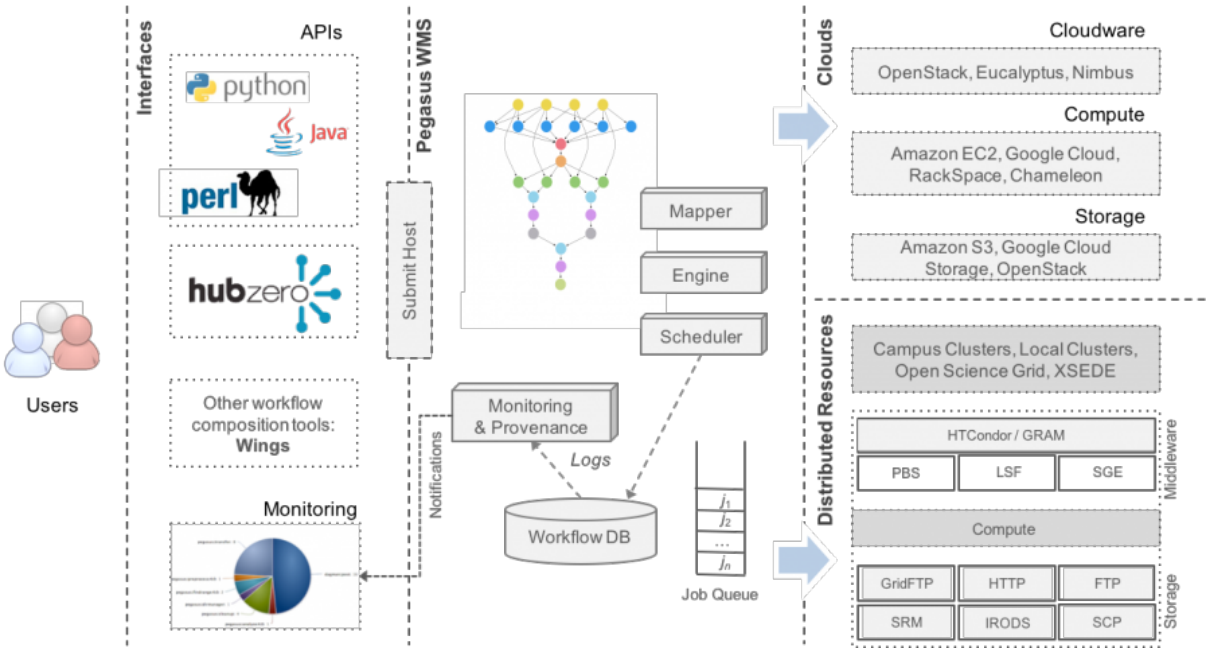



Figure 16: Pegasus architecture [9].

### PyCOMPSs [10]

PyCOMPSs is a parallel programming framework for Python applications built on top of COMPSs. This **WMS** attempts to simulate a traditional computational model with sequential programming, single memory and storage space, abstracting users from distributed and heterogeneous environments and the usage of parallelism. It implements data dependency analysis, task scheduling, data transfer and fault tolerance. PyCOMPSs defines workflows through sequential Python scripts and the user identifies functions to be executed as asynchronous parallel tasks by annotating them with Decorators. In these decorators, users provide information about the data used, if it is modified and the direction of the data flow [99]. In the Java Runtime, the Task analyser receives the tasks and, with the information given by the File Information Provider builds the dependency graph and calculates the tasks without dependencies, passing them to the Task Scheduler. With the information about the available Grid resources and their capabilities, the set of task constraints defined by the user and the location of the data required, this component communicates its decision to the Job Manager. The Job Manager requests all the necessary files from the File Transfer Manager and once they are in the desired host launches the worker [100].

As illustrated in Figure 17, PyCOMPSs serves as a language bidding for COMPSs Java Runtime. The figure also shows how the **WMS** manages Python objects since these objects do not exist in the Java space.

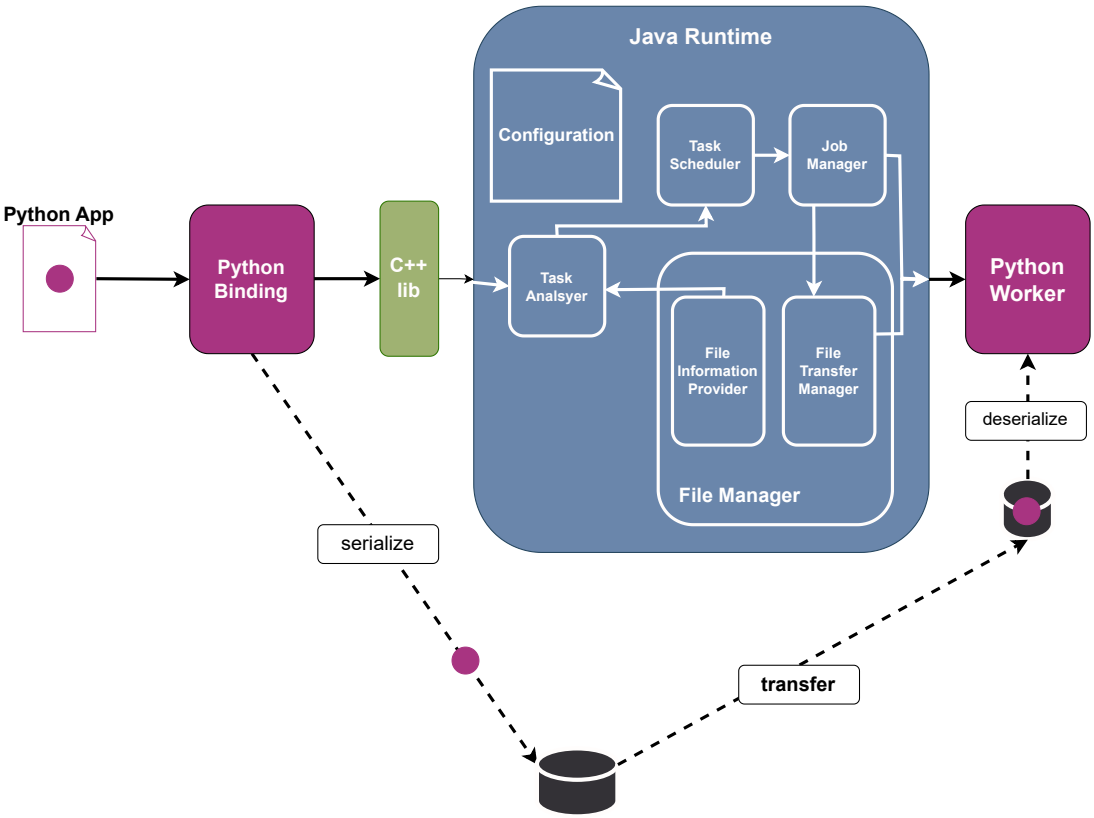



Figure 17: PyCOMPSs architecture. The arrows depict the flow of information to launch a task with Python. The pink ball represents a Python object [10].

### Nextflow [64]

Nextflow is a **DSL**-based **WMS**, that uses an extension of the Groovy programming language, designed for bioinformaticians familiar with programming. A notable feature is being integrated with GitHub to share and reuse pipelines seamlessly.

This **WMS** follows a reactive programming and dataflow model, not needing to pre-compute or store a **DAG**, in which tasks start once data is received [64].

The Nextflow language is based on chaining multiple processes, basic processing primitive of the **DSL**, in order to create complex pipelines. Processes, written in any scripting language that the Linux platform can execute, are executed independently and isolated from each other, communicating asynchronously using first-in-first-out queues, referred to as channels. Describing input and output channels implicitly defines the dependencies between processes, thus creating the execution flow. Nextflow separates the execution environment from the workflow logic, providing an abstraction between the process and the

underlying execution environment. The Executor is the component that determines where the process will run and oversees it [101]. Figure 18 presents a simplified view of the architecture of the **WMS**.

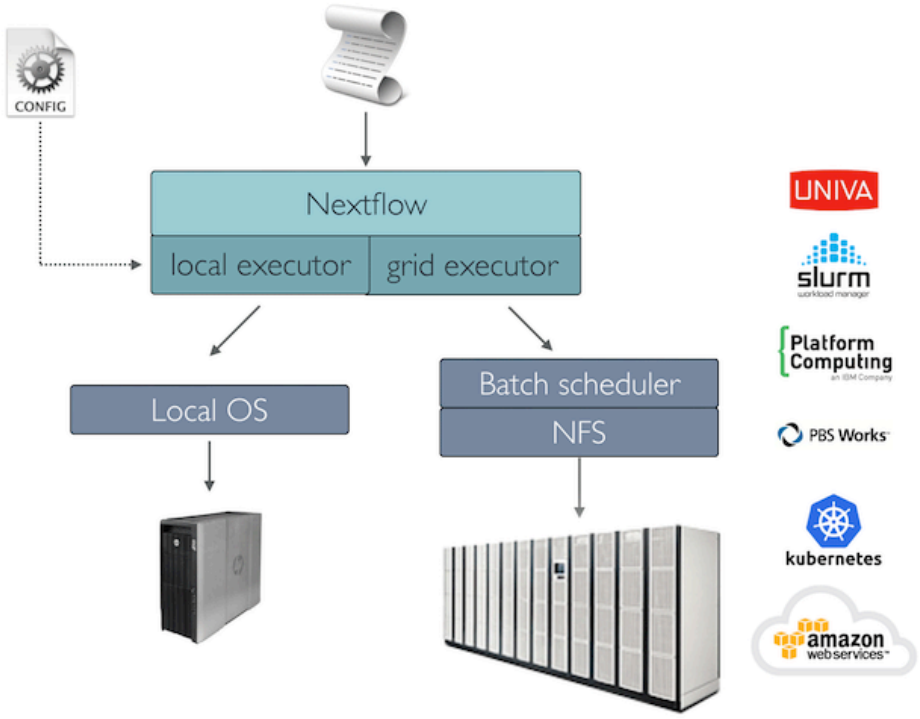



Figure 18: Nextflow architecture [11].

### Snakemake [49]

Snakemake is one of the most popular **WMS**s in science, with more than eleven new citations per week. It uses an easily readable **DSL** that extends Python and defines the workflows in files named *Snakefiles*. The **WMS** infers automatically the data dependencies between tasks and creates a **DAG** of tasks.

Snakemake allows workflows to run in different computational environments, such as a local workstation, **HPC** clusters and Cloud computing, but only one environment in a single run. A Snakemake workflow should be able to run in all these environments without any changes to the workflow definition, only by changing the executor on the command line arguments. Furthermore, it allows users to define the command line setup of Snakemake for any computing platform via configuration *profiles*. With this **WMS**, input and output files may be either stored in remote or local storage [49].

A powerful plugin system [3] which allows Snakemake to expand its features with alternative and independent implementations. Execution backends and remote storage for this **WMS** are implemented as plugins [70]. Snakemake allows the integration of conda and container environments for its tasks and

[3] https://snakemake.github.io/snakemake-plugin-catalog/index.html

offers the ability to automatically generate a containerized version of a workflow. Another remarkable feature of this **WMS** is caching between workflows, not performing redundant computations even if the task was executed in a different workflow or by a different user on the same system [49].

A Snakemake workflow is comprised of *rules*, steps that have input files from which originate output files. A *rule* may be a shell command, an R, Python or Julia script, a block of Python code, a Jupyter notebook or a *wrapper*. *Wrappers* are recurring tools or frequently used libraries in Snakemake workflows, shared among the community via The Snakemake Wrapper Repository[4]. Using *wildcards* it is possible to generalize a *rule*, making it applicable to multiple datasets, for example [70]. By analysing the documentation, we were able to sketch the **WMS** architecture shown in Figure 19.

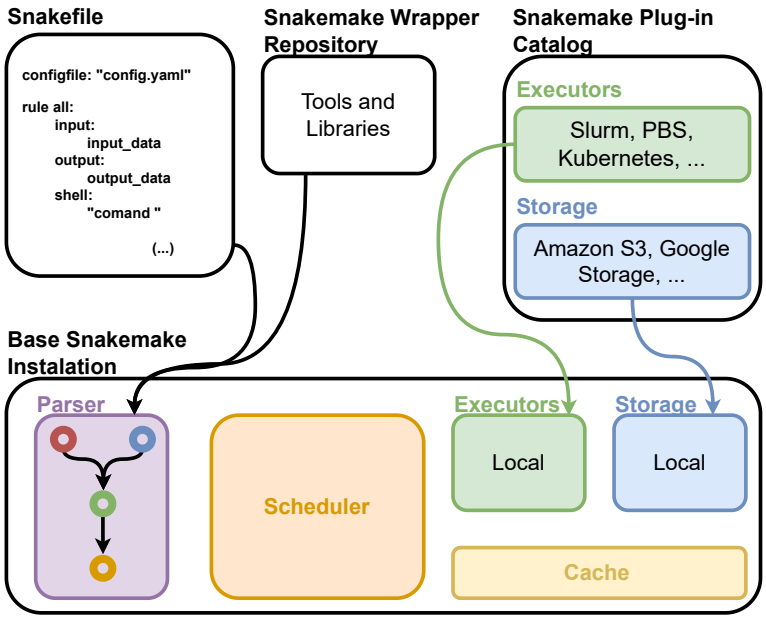



Figure 19: Snakemake architecture.

### StreamFlow [41]

StreamFlow is a container-native **WMS** based on the **CWL** open standard that particularly targets the execution of tasks in a multi-container environment and orchestration of multi-cloud or Cloud-**HPC** hybrid workflows. By adopting the **CWL** it enables users to extend their workflows without needing to change their previous work. StreamFlow is built on top of cwltool [67], using its original features and extending them [41].

Illustrated in Figure 20, the architecture of the **WMS** is designed to manage hybrid workflows that run on heterogeneous and geographically dispersed execution sites. To execute a workflow, StreamFlow needs

[4] https://snakemake-wrappers.readthedocs.io/en/stable/index.html

a workflow description, at least one deployment description ( an infrastructure-as-code representation of the execution site) and a *StreamFlow file* to merge the last two [12]. The architecture is divided into modules. The *Connectors* are used to communicate with the different environments. For **HPC** there are *Connectors* available for Slurm and PBS. The *Deployment Manager* creates and destroys these *Connectors*. The *Scheduler* module uses the *Connectors* available to inspect the best resource to execute the task, according to a *Policy* that the user can previously define. The data's location and transfers are managed by the *Data Manager* [25].

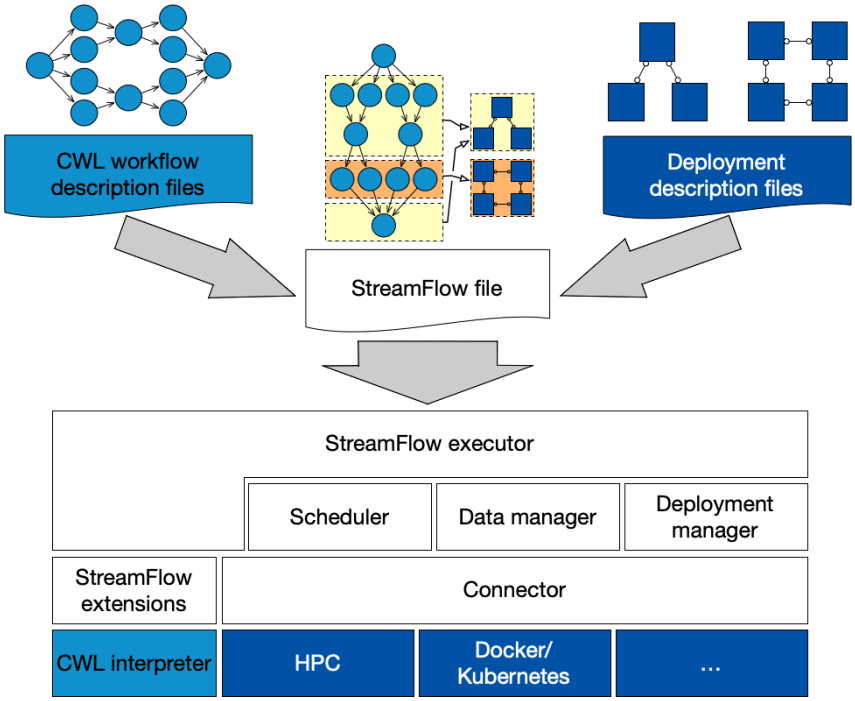



Figure 20: StreamFlow architecture [12].

### Toil [102]

Toil is a fully Python-implemented **WMS** that is scalable, efficient, and cross-platform, developed based on the core principles of functional programming. It can execute workflows described in **CWL** or Python and provides partial support for **WDL** [102].

This **WMS** supports services, such as databases or servers, and can run workflows in a local computer, cloud (Microsoft Azure, Amazon Web Services, Google Compute Engine, OpenStack) or **HPC** (GridEngine or Slurm) environment.

Figure 21 depicts the architecture of Toil. During workflow execution, the leader traverses the graph of job dependencies to decide which jobs to run. Intermediate files and all shared data between components are handled by a job store, which can be implemented in multiple ways, allowing Toil to support different

back-end file stores. The batch system schedules the jobs given to it by the leader, running a worker command for each job and the node provisioner creates worker nodes in which the batch system schedules the workers, temporary processes responsible for running jobs. The stats and logger gathers information about the executions in the job store [13].

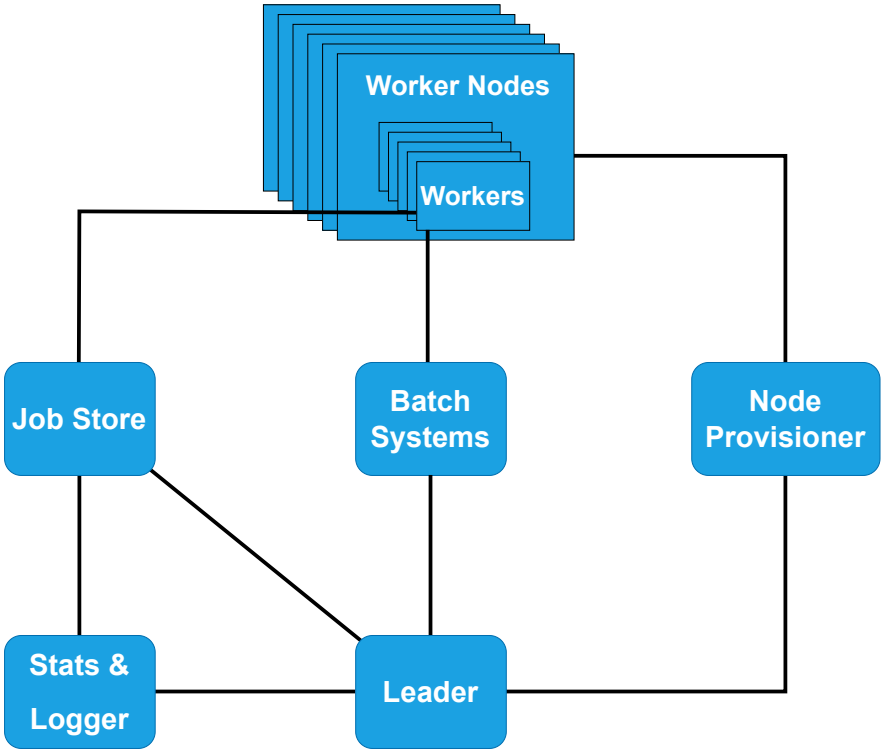



Figure 21: Toil architecture [13].

**Summary**

As expected, the main characteristics of the **WMS**s presented are aligned with the ones presented for their respective workflow specification type, presented in Section 2.1.3. All the **WMS**s have repositories for users to share and re-use workflows.

From a general perspective, all **WMS** are able to execute workflows in multiple distinct environments, enabling the usage of workflows despite the underlying execution environment, but not many can execute tasks of workflows combining multiple environments. Workflow inputs and/or outputs are usually written to the local filesystem. As previously defined, cross-facility workflows use more than one computational site, i.e. remote storage for inputs and outputs or remote execution for tasks of a workflow. The most simple form of defining one of these workflows is through the usage of remote storage, e.g. using input files read from Azure blob storage. This type is common for the majority of the **WMS** presented, thus being

described as **WMS**s able to execute cross-facility workflows. The remote execution of tasks through more than one environment in a single workflow is not as common and harder to achieve [25]. Only StreamFlow and most programming-library-based **WMS**s (mainly with external dependencies) allow remote execution of tasks.

From all the **WMS**s presented in this Section, the **DSL**-based **WMS**s are the most popular. This may be due to the modularisation of tasks in workflows ( using containers and conda packages) and the flexibility that Snakemake and Nextflow provide [45, 46]. Table 2 provides an overview of the **WMS**s analysed in the current section.

| WMS | Workflow Specification Type | Execution | Main Focus | Considered Cross-facility | Allows Remote Execution |
|---|---|---|---|---|---|
| **DagOnStar** | Programming-library | Local; Cluster; Cloud | Portability (**IoT**) | ✓ | ✓ |
| **FireWorks** | Programming-library | Local; Cluster; Cloud | Fault Tolerance | ✓ | ✓ |
| **Galaxy** | Graphical | Cluster;Cloud | Ease-of-Use | ✓ | ✗ |
| **Parsl** | Programming-library | Local; Cluster; Cloud; kubernetes | Scalability | ✓ | ✗ |
| **Pegasus** | Programming-library | Local; Cluster; Cloud | Portability | ✓ | ✓ |
| **PyCOMPSs** | Programming-library | Local; Cluster; Cloud | Performance | ✓ | ✓ |
| **Nextflow** | **DSL** | Local; Cluster; Cloud | Flexibility | ✓ | ✗ |
| **SnakeMake** | **DSL** | Local; Cluster; Cloud; kubernetes | Flexibility | ✓ | ✗ |
| **StreamFlow** | Standard Specification | Local; Cluster; Cloud; kubernetes | Portability (Hybrid Execution) | ✓ | ✓ |
| **Toil** | Standard Specification | Local; Cluster; Cloud; kubernetes | Performance | ✓ | ✗ |

Table 2: Summary of the presented **WMS**s and their characteristics

### 2.2.2 Workflow Management Systems Comparisons and Evaluation

Many studies are made to compare **WMS**s, either regarding their general characteristics, such as ease of use, expressiveness and portability, or more technical ones, i.e. CPU utilisation or memory usage for a specified use case.

Some comparisons, namely articles introducing or presenting new features for **WMS**s [5, 41, 62, 64], may not be completely unbiased. The metrics or characteristics will tend to benefit the presented tool or the approach followed by the developers, instead of performing a full comparison with other **WMS**s. In opposition, some researchers and developers leave these comparisons to other researchers, focusing on presenting the works and features of their **WMS** [20, 49].

From a more technical point of view, there are comparisons analysing the architectural approach developers took and conducting experiments to extract performance metrics from the execution of real workflows. More general comparisons focused on reviewing the architecture and classifying relevant characteristics of **WMS**s, such as their execution model, provided data access methods. Other comparisons are more focused on evaluating performance metrics, including elapsed time, CPU and RAM utilisation and data footprint. These studies usually use one workflow to analyse because of the difficulty of translating workflows to be used with multiple **WMS**s. Less technical evaluations on **WMS**s are integral to researchers who take advantage of these frameworks, as their domain of expertise usually does not require a vast knowledge of computer science. These studies cover a wider range of tools, as they do not require as many computational tasks as the other comparisons, and address topics such as ease of use, language expressiveness, portability across execution environments, scalability, available learning resources and whether there are pipeline-sharing initiatives.

Article [43] focuses on the needs and characteristics of design to handle extreme-scale applications, which operate on multiple computational resources and massive datasets. It presents a characterisation of **WMS**s based on their execution model (cyclic or acyclic), the ability to manage a workflow within a particular computational resource and across systems and possible data access methods. The authors classify 15 **WMS**s (including Nextflow) regarding these parameters. Likewise, [48] presents a comparative analysis of how **WMS**s adapt to the different paradigms like **HPC**, cloud and big data environments. The authors review seven **WMS**s (Pegasus, Kepler, Taverna, Swift, KNIME, Airavata, and Meandre) in key aspects such as scalability, performance, data management, user accessibility, collaboration support, and adaptability to emerging technologies like **AI**.

The authors of [47] make a comprehensive evaluation of **WMS**s for the LodSeq [103] workflow. The article presents a comparison between the execution of the bioinformatics workflow using Cromwell (with

WDL), Nextflow, Pegasus-mpi-cluster, Snakemake and Toil (with CWL) regarding elapsed time, memory, CPU usage among other objective performance metrics.

The paper [25] explores other environments and compares the hybrid cloud-HPC workflow paradigm of StreamFlow with the Snakemake workflow execution targeting only HPC facilities. It presents the advantages and precautions of using a WMS with the capability of executing hybrid workflows.

In [44], the authors make a systematic analysis of Swift/T, Nextflow, WDL, and CWL regarding features such as language expressiveness, modularity support, robustness, reproducibility and standardisation. Similarly, [45] uses criteria such as ease of use, functionality, installation methods and debugging capabilities. It also provides insights into their popularity, positive reception in the bioinformatics community, open-source nature, and active development. Most WMSs are developed as open-source projects in bioinformatics, so they often lack maintenance and documentation, making it difficult to use them efficiently. This reinforces the importance of the metrics analysed by these articles, particularly the number of available learning resources and pipeline repositories. In the article [46], authors make a more general comparison of WMSs based on ease of use, expressiveness, portability, scalability, amount of available learning resources and pipeline initiatives.

### WfCommons

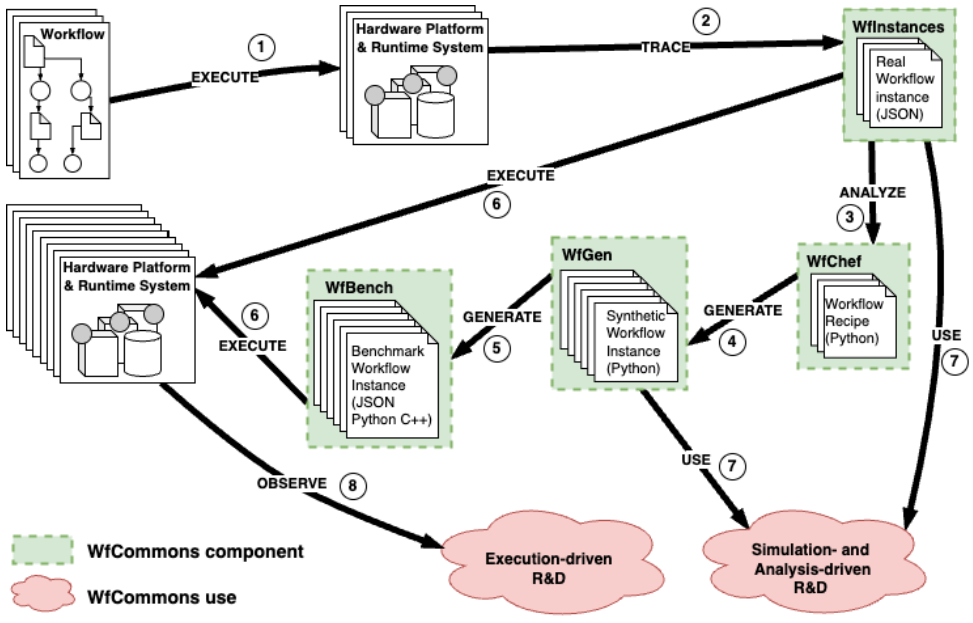



Figure 22: WfCommons architecture [14].

The WfCommons Project [104] is a collaborative open-source framework, distributed as a Python package, focused on providing the basic means for scientific workflow and WMSs research and evolution. This

framework offers tools to analyse and generate synthetic workflow instances, and build realistic workflow benchmark specifications, which can be translated into benchmark code for **WMS**s [14, 104].

WFCommons uses a common workflow instance format JSON specification named WfFormat to represent execution and synthetic workflow instances. These files store all the pertinent information for its use, as it is standard for all tools in this project [14].

Figure 22 depicts the architecture of WfCommons with its four main components: **WfInstances**, **WfChef**, **WfGen** and **WfBench**. The arrows show the overview of a research and development integrating all the components.

Workflow execution instances represent real executions of workflows on a distributed platform and store the workflow execution profile data and relevant information. **WfInstances** gathers information from workflow application executions such as task execution metrics, workflow structure information and compute resource characteristics. This component parses execution logs and creates *workflow instances*. Currently, it can parse logs for Pegasus[20], Nextflow[64] and MakeFlow[14, 105].

**WfChef** analyses a set of *workflow instances* for generating synthetic workflows resembling the workflow application executed and builds *workflow recipes*. A *workflow recipe* stores the discovered pattern occurrences, the statistical models of workflow task characteristics and the workflow **DAG** information. WfCommons incorporates 9 *workflow recipes* ready to use [14].

With a *workflow recipe* it is possible to generate synthetic *workflow instances* using **WfGen**. This component generates *workflow instances* of a based workflow application with an arbitrary number of tasks. Based on the *workflow recipe* provided, the user may choose the factor on which the runtime and/or file sizes will be increased/decreased [14].

**WfBench** creates realistic workflow benchmarks, which can be translated into benchmark code and executed with **WMS**s. This component allows users to generate benchmark specifications with chosen performance characteristics (CPU, memory, and I/O usage) for workflow tasks from realistic or synthetic *workflow instances*. With the workflow benchmark specifications, it is able to translate them into executable workflow applications. Presently, there are translators for MakeFlow[105], Nextflow[64], Pegasus[20] and Swift/T[106] **WMS**s [14, 107].

#### Discussion

In the studies presented, the authors highlight the challenge of choosing a **WMS** due to the diverse range of available options, each with its strengths and weaknesses. It is also important to emphasise that most comparisons focus on using **WMS**s with bioinformatics workflow applications, having fewer studies

addressing their utilisation in other domains.

Another noteworthy consideration in these studies is that using real-world workflows requires application domain knowledge to generate workloads from a single workflow, with small or fixed workloads. Despite the WfCommons Project's capacity to generate standard workflow benchmark code, it still lacks translators for the most used workflow specification languages.

A reproducible method for researchers to analyse and conduct comparisons of **WMS**s using multiple standardised workflows and performance metrics across all **WMS**s and in different environments is currently lacking. Therefore, we aim to develop a framework capable of creating execution reports with performance metrics from workflow applications with arbitrary workloads. To validate the framework, we conducted a comparison of **WMS** in a multi-node environment.

# Chapter 3

# Approach

This chapter presents the practical approach to create, execute and collect workflow execution metrics of **WMS**s. We describe the development of our framework from scratch to the implementation details.

## 3.1 Scope Definition

We aim to develop a framework to automate the evaluation process of **WMS**s. To achieve this, we need to establish the scope of three main components: the workload, the environment, and the metrics collected from the system. To determine the requirements of our approach, we provide an overview of the existing methodologies and the path we have taken, highlighting the considerations and choices made during the process.

The workload, in this case, the origin of workflows, may be synthetic or real. Using synthetic workflows, e.g. the benchmark code created by WFBench [107], we can seamlessly generate different workflows from distinct domains with arbitrary CPU and memory specifications for each type of task. This approach also enables us to choose the number of tasks in each workflow. The downside of this approach is that the workloads and data specifications created may not accurately represent the metrics evaluated in a real practical use case.

On the opposite path, using real workflows as workload reflects and enables the collection of performance metrics in a practical use case, thus being more meaningful than the metrics collected from a synthetic workload. A challenge of this approach is the scope and the scalability of the tests are mitigated, as it is harder to incorporate more workflows from different domains and it is needed further knowledge in each field to change the workflow tasks and use distinct input datasets.

Initially, we explored multiple sources to identify possible real use case workflows to integrate our framework, such as Dockstore [74], Workflowhub [73] and other workflow repositories. These workflows, although reproducible were built for their specific use case, challenging users outside of the workflow

domain to modify and scale the workflow. Furthermore, we also lack the domain knowledge to choose different real datasets, as they often need some preparation prior to the workflow execution. Instead, we decided to use synthetic workflows and workloads. This approach allows for the seamless use of multiple workflows with distinct structures and workloads. From another perspective, by favouring this approach, the workflows only use the basic primitives from the **WMS**s specification, not exploring unique workflow definition options from each **WMS**, such as *gather* and *scatter* workflow structures for parallelisation of tasks, as these structures require further analysis of the workflow and are not as commonly employed by users.

As previously mentioned, one of the main objectives of **WMS**s is to be able to execute workflows across multiple execution environments seamlessly. Therefore, we do not want to limit the execution of our framework to one environment, instead, we want to allow the evaluation in multiple environments. Different **WMS**s may support distinct environments, and even when they support the same environment, developers often make different choices to handle execution challenges, necessitating varied preparations. Therefore, it is preferable to use separate components for each **WMS** to abstract these tasks from the user. By adopting this simple approach, we facilitate the reuse of our scripts by other researchers who wish to analyse different aspects of the **WMS**s we have implemented.

Our framework is designed to collect performance metrics from both the workflow execution and the workflow manager. It then generates report files that summarise the execution. We want to gather base metrics, such as the execution duration, the execution start overhead and the data footprint of the execution. We want to gather other metrics related to the **WMS**s itself, like the CPU utilisation. If possible, we want to avoid the usage of intrusive tools to gather metrics. All log and report files should be stored.

The process of collection of metrics should be automated for consecutive executions with the same **WMS**. An alternative approach would involve executing a single workflow consecutively with multiple **WMS**s. This method is less favourable because of the potential failures associated with workflow executions, which could result in incomplete or unreliable reports.

### 3.1.1 Requirements

Following the analysis and the scope definition of the framework to be developed, we built a list of requirements to be met. These requirements ensure the generation of an unbiased and **FAIR** collection of performance metrics.

1. **Domain Agnostic Synthetic Workflows:** The framework must generate synthetic workflow code and input data, regardless of their domain.

2. **Arbitrary Task Workload:** The framework must support the specification of CPU and memory parameters, as well as input and output sizes for each task type within the workflow to be executed.

3. **Statistic Collection:** The framework must collect the workflow execution time, workflow start overhead, task throughput and data footprint. Each task type in the workflow must summarise the time execution with descriptive statistics. It must also collect hardware events regarding the **WMS**.

4. **Multi Environment:** The framework must allow the user to specify the environment where the **WMS** will execute the workflow.

5. **Modularity:** The framework must be comprised of a set of individual components chained together to automate the report generation process. These components may be used and developed independently from each other.

6. **Reproducibility:** The framework must store all the execution logs and reports of each execution and seamlessly allow the recreation of previous metric collection executions.

7. **Extensibility:** The framework must allow the integration of new **WMS**s and workflows seamlessly.

8. **FAIR Principles Conformity:** The framework must be developed and maintained according to the **FAIR** principles, with complete documentation and examples of use.

Based on the requirements we were able to specify the workflow of the execution of our framework shown in Figure 23. First, the user needs to create or use a specification with the type of workflow, number of tasks and information about the workflow for each type of task. Based on that specification, our framework generates the executable workflow code for a user-defined **WMS**. The workflow code is executed in a specified environment, collecting all metrics and logs created in the execution process. All the information collected is analysed, stored and summarised in a report. All steps of the diagram are different modules of the framework. Inferred from the diagram, we can see that the framework primarily works with two types of files: specification files, which define the workflow to be created, and report files, where the collected metrics are stored.

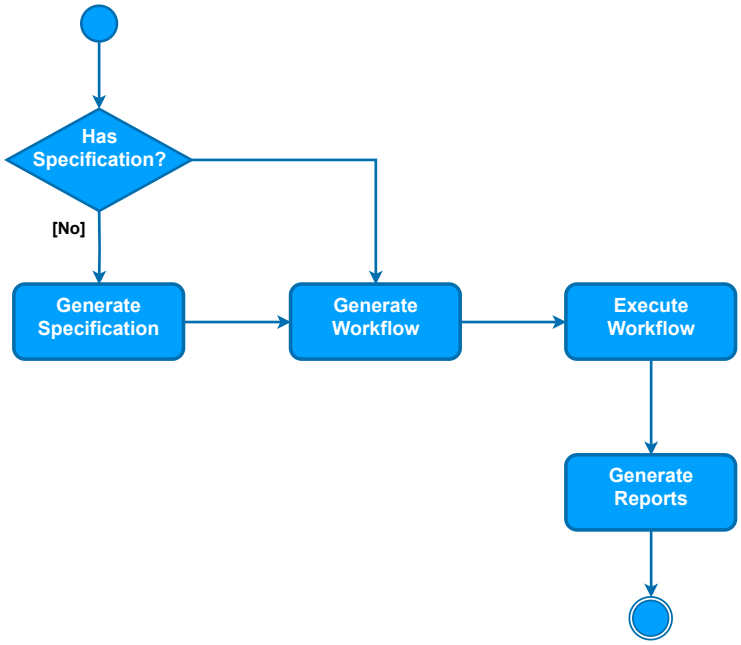



Figure 23: Activity Diagram of *Huginn* for report creation.

## 3.2 Architecture

We decided to name the framework *Huginn* ("Thought" from old norse), after one of Odin's ravens carved on the coin of the Viking king Olaf Guthfrithson[1]. The *Huginn* framework has three individual modules (**wfSpecsGen**, **wfGenerator** and **wfReportsGen**) and is formed by two main components that chain the modules together in order to automate the report generation process (**SpecsGen** and the **Runner Scripts**).

### 3.2.1 Huginn Framework

*Huginn* was built by establishing connections between modules to automate the report generation process., comprising two distinct components: one to create the specifications and the other to generate the workflows and create the reports. The decision to create two separate components was made to allow users to customise the workflow specification at will. Figure 24 depicts the architecture of *Huginn* with all its modules. The arrows show the typical use case of the framework, as described in Figure 23.

[1] https://artsandculture.google.com/asset/silver-penny-of-anlaf-guthfrithsson/2AGh76IxA7AWXA

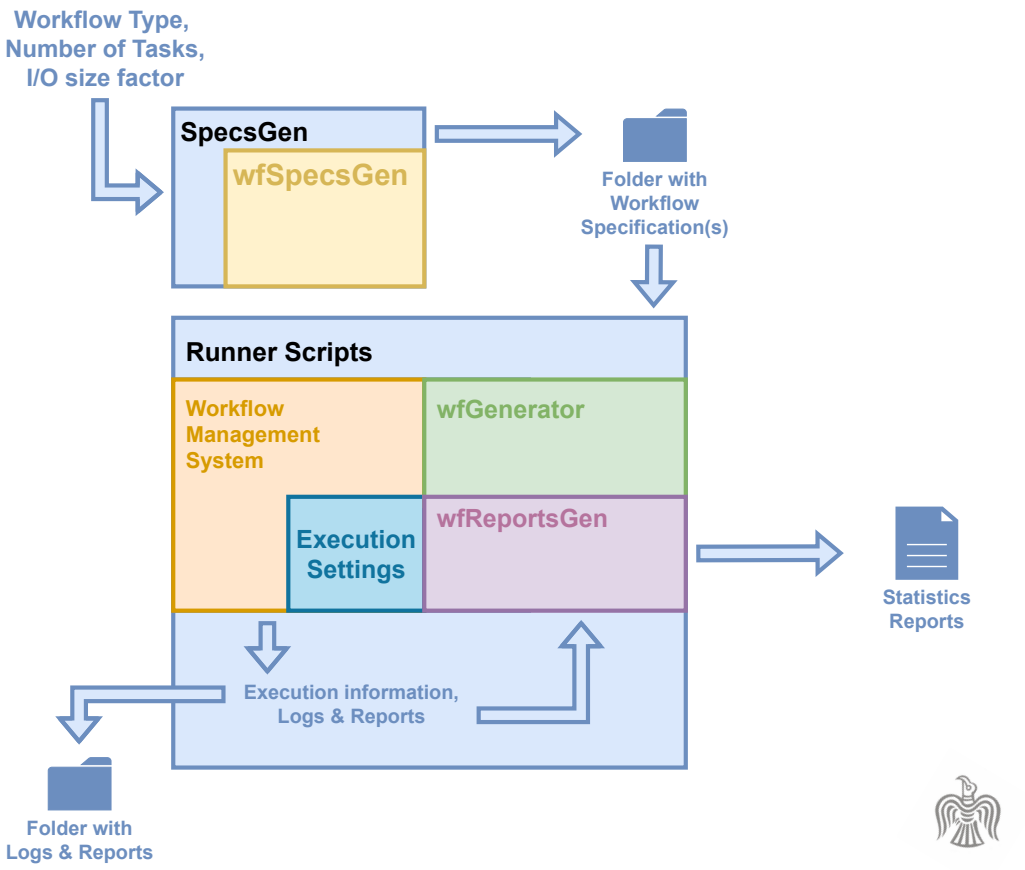



Figure 24: *Huginn* Framework Architecture.

### SpecsGen

The **SpecsGen** is responsible for automating the creation of workflow specification files, allowing users to assemble multiple workflow specifications at once. This component uses the module *wfSpecsGen* to build specification files. This component is separated to allow the customisation of specification files.

### Runner Scripts

The **Runner Scripts** are the core of our framework and are comprised of one **WMS** and its execution settings for the multiple environments and the *wfGenerator* and *wfReportsGen* modules. Although the execution settings are part of the **WMS**, we chose to include them because there is a need to create or prepare directories and files to be able to use the **WMS**s in certain environments.

This component prepares the environment and creates all necessary directories to execute workflows for its **WMS**. It takes as input a folder with multiple workflow specification files and, for each one of them, generates the respective workflow executable code, collects metrics and generates reports regarding the

execution. All the logs, reports, and information collected from the workflow execution are stored alongside the generated reports, and all the intermediate and workflow code files are deleted after the collection of metrics.

### 3.2.2 Individual Modules

The *Huginn* framework has three independent modules, each performing a different role in the report generation process.

#### Workflow Specification - wfSpecsGen

The **wfSpecsGen** module generates JSON files containing all the necessary details to create a synthetic workflow, such as the workflow type and parameters for each task type. Figure 25 provides a simple sketch of the module, where the grey-coloured boxes represent the WFCommons components utilized by *wfSpecsGen*.

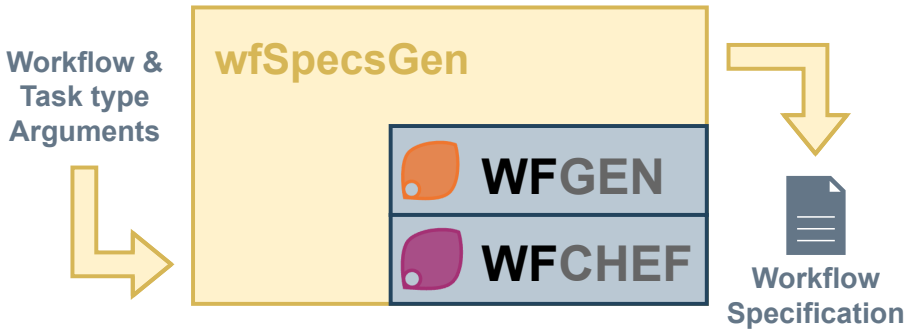



Figure 25: Overview of *wfSpecsGen* module.

The primary goal of this module is to allow users to automatically build specification files by defining the file format and determining the input sizes for each task type. It uses *WFGen* and *WFChef* to get information about workflow types, such as types of tasks and input sizes, easing the user's work.

#### Workflow Executable - wfGenerator

The **wfGenerator** leverages *WFBench* to create workflow applications for a **WMS** based on workflow specification files provided by the user. *WFBench* uses the information of this file to generate a benchmark specification. With a collection of translators, benchmark specifications can be transformed into actual workflow definition files.

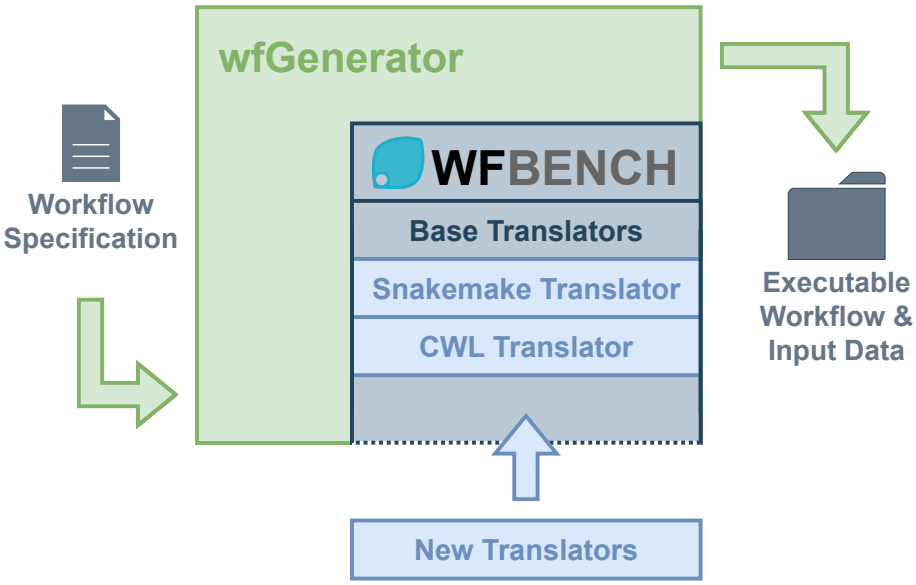



Figure 26: Overview of *wfGenerator* module.

Figure 26 provides an overview of the module. The grey-coloured elements represent the existing software of *WFBench* that the module utilizes to create workflows, while the blue boxes indicate the translators added to the WFCommons project. As illustrated, the *wfGenerator* allows the expansion of translators for *WFBench* without any changes to the source code.

**Report Generation - wfReportGen**

The **wfReportsGen** parses Logs and reports from the **WMS** and other software to create a report with all the gathered information. The objective of these reports is to provide a simple and single source with all information about the workflow execution and performance metrics of the **WMS** used. Inspired by the translators from *WFBench*, we created *parsers* for each **WMS**. *Parsers* gather information from the workflow execution reports generated by the **WMS**. It is possible to create and add a *parser* for *wfReportGen*, similarly to how translators are added in *wfGenerator*. The *wfReportsGen* may be used to generate reports of real workflows as long as the tasks respect the name pattern *"[Task Type]_[Number]"*.

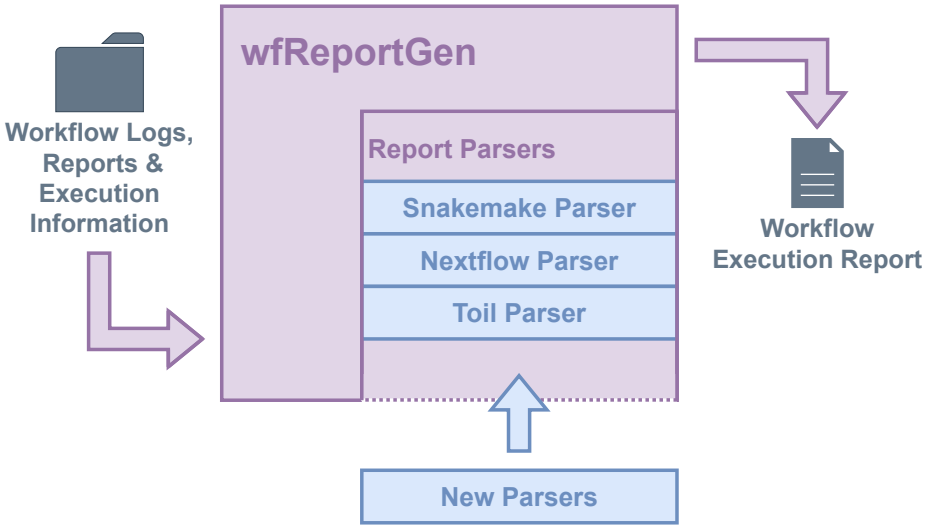



Figure 27: Overview of *wfReportGen* module.

Figure 27 depicts the *wfReportGen* module, where the *parsers* are represented by the blue-coloured boxes.

## 3.3 Implementation

The *Huginn* framework is formed by two main scripts: one to create the specification files for workflows (*SpecsGen*) and the other to generate, execute the workflow and generate the reports (*Runner Scripts*). Here we describe the implementation details of these scripts and the modules they use.

### 3.3.1 SpecsGen

The *SpecsGen* element forges one or more workflow specifications with an optional input/output size factor, workflow type, and the number of tasks in the workflow. Using *wfSpecsGen*, this script generates a workflow specification file and a log file that gives information about the input/outputs of the tasks and the disk usage of all the workflow files. It allows the creation of many workflow specification files by passing more than one value for the number of tasks.

The tasks of the synthetic workflows use a python script called *wfbench.py* as the program executed by a task. To mimic the CPU work, it uses an executable which calculates an increasingly precise value of $\pi$ up until the specified total amount of computation (the values defined for cpu-work) and it leverages *stress-ng*[2] to execute memory-intensive threads. The cpu-work, percentage of threads performing computation

[2] https://github.com/ColinIanKing/stress-ng

instructions and input size must be specified for each type of task.

To create workflow specification files, *wfSpecsGen* uses an existing *WfChef* recipe, a base number of tasks and an input/output size factor, we create a synthetic Workflow execution instance with *WfGen*. The *inputs* parameter values are determined based on the average input size for each task type in the workflow instance generated. The *percent-cpu* and *cpu-work* values are set by default to 0.6 and 10, respectively. Workflow Specification files have the following components:

- **workflow**: Name of the workflow to generate
- **tasks**: base number of tasks of the workflow
- **inputs**: the size of the input by task type in bytes.
- **percent-cpu**: Percentage of threads performing computation instructions per workflow task by type.
- **cpu-work**: CPU work per workflow task by type.

Bellow presented, we have the Workflow specification of a *Seismology* workflow with 103 and an input/output size factor of 100, meaning that the input/output size of each task type is 100 times bigger than in the original *Seismology* workflow.

```
{
    "workflow": "Seismology",
    "tasks": 103,
    "inputs": {
        "sG1IterDecon": 605036.72,
        "wrapper_siftSTFByMisfit": 120121340.0
    },
    "percent-cpu": {
        "sG1IterDecon": 0.6,
        "wrapper_siftSTFByMisfit": 0.6
    },
    "cpu-work": {
        "sG1IterDecon": 10,
        "wrapper_siftSTFByMisfit": 10
```

```
    }
}
```

### 3.3.2 Runner Scripts

Section 2.2.1 introduces multiple state-of-the-art **WMS**s, with multiple distinct features, highlighting the hybrid workflow execution. We determined that the main objective would be to evaluate and compare the hybrid execution workflow model with the single-sited execution models, therefore, we decided to analyse the two most recently cited **DSL**-based **WMS**s (SnakeMake and Nextflow) and compare them with a standard specification **WMS** (Streamflow), regarding execution time, workflow starting overhead, task throughout and data footprint.

Due to the lack of maturity of Streamflow's new version (0.2.0) and incompatibilities with the **HPC** cluster available, we changed the initial scope of our research to analyse **WMS**s with a single-site execution model to assess whether it is necessary to occupy an **HPC** cluster node. Thus, we replaced Streamflow with another standard specification **WMS**, Toil.

Thus, we developed 3 different Runner Scripts, for Snakemake (v8.16.0), Nextflow (v24.4.3) and Toil (v7.0.0), and settings for running locally or on a Slurm cluster. All the log and report files originated from the workflow execution remain on disk.

Using the Runner script for the intended **WMS**, the user specifies the execution environment and a directory containing the specification files for all the workflows intended to be executed. The *wfGenerator* creates a workflow application for the **WMS** based on one of the specification files in the provided directory. With the execution settings for the specified execution environment, the **WMS** executes the workflow application.

The *wfGenerator* leverages *WFBench* to create workflow applications for a **WMS** based on workflow specification files provided by the user. *WFBench* uses the information of this file to generate a benchmark specification. With a collection of translators, benchmark specifications can be transformed into actual workflow definition files.

Currently, *WFBench* (version 1.0) only supports translation for one of the chosen **WMS**s, Nextflow. Thus, we introduced two new translators extending the WfCommons framework: one for SnakeMake **DSL** and other for **CWL**. The SnakeMake translator creates a simple Snakefile based on the benchmark specification, while the **CWL** translator, aside from the workflow definition file, needs to create the YAML file with the workflow input definition. Aside from the workflow executable file definitions, *WFBench* also creates all the input files to be used in the workflow execution. Although provided by *WFBench*, the Nextflow

translator was deprecated and had to be updated. The updates to this translator include a modification to the workflow schema version, from v1.3 to v1.4. Additionally, the name definition of tasks and workflow files was aligned with WfCommons version 0.8, rather than version 1.0.

*wfReportsGen* receives all information and parses report and log files, gathering all metrics in a statistics report. The metrics regarding the manager of the **WMS**s are collected using *perf*. Although we will only analyse CPU utilisation, we included all events in the statistics report. These metrics include CPU utilisation, context switches, CPU migrations and the number of branches and branch misses.

The elapsed time, workflow start overhead and task time measurement are extracted from the reports of each **WMS**. For each task type, we only store time statistics: the minimum, maximum, mean, median and the first and third quartiles. This approach is adopted due to the large scale that workflows can reach.

The data footprint is simply measured with the *du* command, representing the disk usage that the **WMS** introduces to the workflow execution for cache, logs and statistics collection. Below we present an example of a report from a Seismology workflow execution with toil.


```
{
    "wms": "toil",
    "spec_File": "seism-103.json"
    "data_Footprint": "12G",
    "elapsed_time": 657.905214119,
    "elapsed_time_unit": "seconds",
    "Num_tasks": 103,
    "throughput": 0,156557507,
    "start_Overhead": 43.532575130,
    "user_time": 106.102545,
    "user_time_unit": "seconds",
    "sys_time": 59.690853,
    "sys_time_unit": "seconds",
    "task_clock_time": 162961.75,
    "task_clock_time_unit": "msec",
    "cpu_utilized": 0.248,
    "contex_switches": 240834,
    "cpu_migrations": 809,
    "cycles": 690990433572,
```

```
    "instructions": 1467511516746,
    "branches": 282646423803,
    "branch_misses": 4989868745,
    "tasks": [
        {
            "type": "sG1IterDecon",
            "min": 7.13,
            "25%": 8.13,
            "50%": 8.245,
            "75%": 8.37,
            "max": 9.0,
            "mean": 8.2161
        },
        {
            "type": "wrapper_siftSTFByMisfit",
            "min": 23.27,
            "25%": 23.27,
            "50%": 23.27,
            "75%": 23.27,
            "max": 23.27,
            "mean": 23.27
        }
    ]
}
```


### Complementary Scripts and Documentation

Regarding the **FAIR** Principles, we provide information about all the steps to quickly recreate the metrics collection fo this dissertation. All essential details needed to use or extend either the modules or the Framework are also provided with a template to create other Runner Scripts. This documentation is written in the README markdown file.

To enable the reproducibility of our research we provide a setup script to ease installation. This simple script creates a Micromamba (for SnakeMake) and a Python Virtual Environment (for Toil and Nextflow)

with all the dependencies needed to perform the performance metrics extraction. The script used to make all bar plots present in Section 4.2 is also available. This way, it is possible to recreate our research from scratch.

# Chapter 4

# Evaluation and Results

In this chapter, we present all the tests conducted with our framework, the environment in which they were executed, and the corresponding results, followed by their analysis and interpretation.

## 4.1 Tests

We chose to analyse SnakeMake, Nextflow and Toil by executing 3 workflows from 3 out of the four scientific domains (bioinformatics, physics, geology and astronomy) in which workflows are more commonly used [94]: **Montage**, **Seismology** and **1000Genome**. The objective is to compare the selected **WMS**s in terms of workflow execution, focusing on execution time, execution start overhead, data footprint, and CPU utilisation.

### 4.1.1 Workloads and Test Workflows

As previously mentioned, the workflow benchmark code generated for a specific real use case is designed to replace the tasks' executables with a CPU-Memory benchmarking program named *wfbench.py* to simulate their behaviour.

In order to determine the *percent-cpu* and *cpu-work* parameters for each type of task in the workflows, we first analyse the relationship between the CPU percentage/work parameters and the execution time of *wfbench.py*. The input/output size value for each task is calculated by *wfSpecsGen*.

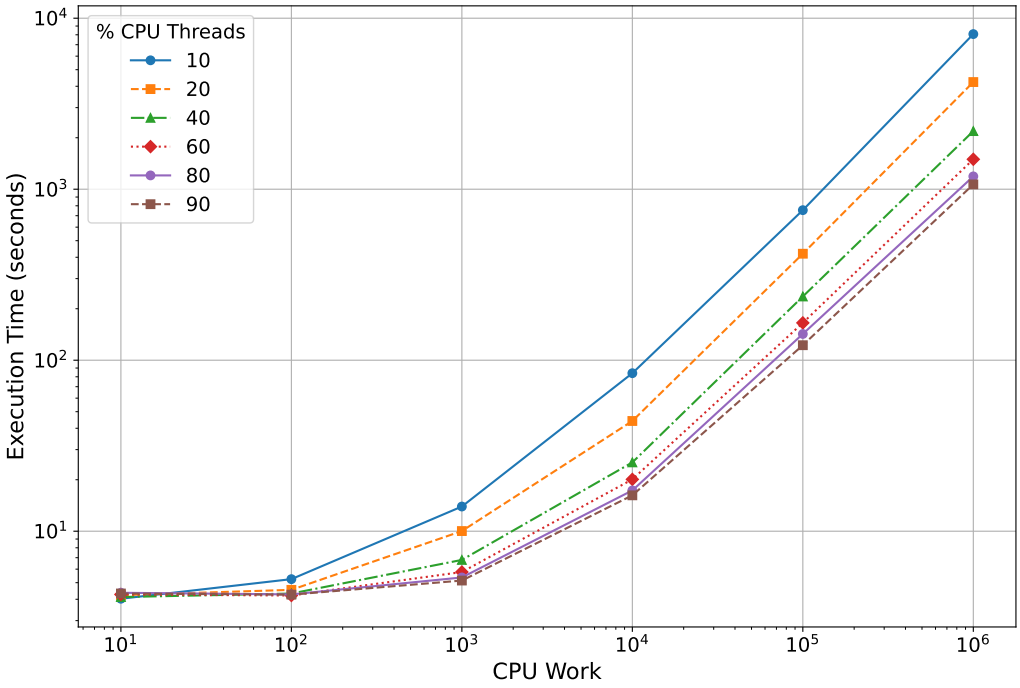



Figure 28: Execution time of *wfbench.py* for multiple *cpu-work* and *percent-cpu* values with an input file of 1 KByte.

As expected, by increasing the percentage of threads doing CPU work, the execution time decreases. This difference becomes significant starting from a *cpu-work* value of 1000 and beyond. Tasks with high CPU utilisation were simulated using more CPU threads (a higher *percent-cpu* value), while the *cpu-work* value was set to simulate the execution time of each task type. Specifically, by knowing the execution time of a given task type, the appropriate *cpu-work* value was selected for the task's *percent-cpu* to match the estimated execution time.

The types of tasks in the generated workflows, although very similar, may differ from those presented by workflow applications in Section 2.1.6 because the former are based on specific workflow use cases which may change according to the application scenario. The workflow applications generated differ little from the ones shown above.

The synthesised workflows will not always contain the exact number of tasks specified, as *WFBench*, utilised within *wfGenerator*, constructs workflows by reusing and replicating specific substructures found in real-world workflows. As a result, the number of tasks in the generated workflows is always less than or equal to the specified number.

### Montage

The workflow code generated by our framework is very similar to the workflow in Section 2.1.6, differing in the last two task types of the workflow. In the generated workflows, the *mShrink* and *mJPEG* task types are merged forming a single type named *mViewer*. Figure 29 shows an example of the **DAG** of *Montage* workflow application generated with *wfGenerator*. The degree of *Montage* workflow applications represents the number of workflow branches, e.g. in Figure 29 we have a 3-degree montage workflow.

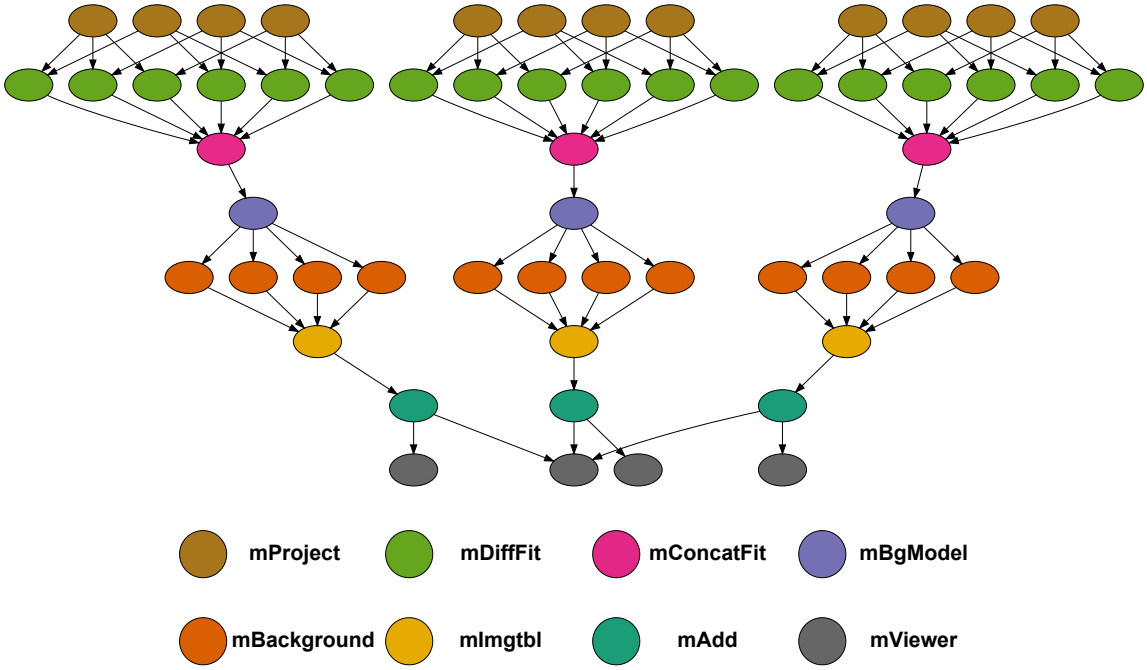



Figure 29: Montage workflow **DAG** with 58 tasks generated with WFBench.

For the task parameters, we defined all the CPU-related parameters according to [82], where the percentage of CPU threads was specified according to the CPU utilisation of the type of task described in the article. The CPU work was then defined based on the expected execution time for each kind of task. All input and output files were automatically specified when using *SpecsGen* with an input/output file factor of *0.02*. Table 3 presents all the specification file parameters we defined.

We decided to test the **WMS**s with different degrees of *Montage* workflow applications: 3-degree (60 tasks), 4-degree (300 tasks), 6-degree (707 tasks) and 11-degree (1500 tasks). Table 4 shows the estimated number of tasks for each type in the *Montage* workflows.

| Task | CPU Work | CPU Threads (%)† |
|---|---|---|
| mProject | 100 | 0.9 |
| mDiffFit | 100 | 0.3 |
| mConcatFit | 80000 | 0.5 |
| mBgModel | 363000 | 0.9 |
| mBackground | 100 | 0.1 |
| mImgtbl | 34750 | 0.1 |
| mAdd | 35000 | 0.1 |
| mViewer | 49500 | 0.8 |

† In the specification described from 0 to 1.

Table 3: Parameters used in the Specification file

| | #Tasks | | | |
|---|---|---|---|---|
| mProject | 12 | 42 | 105 | 226 |
| mDiffFit | 18 | 186 | 469 | 989 |
| mConcatFit | 3 | 4 | 6 | 11 |
| mBgModel | 3 | 4 | 6 | 11 |
| mBackground | 12 | 42 | 105 | 226 |
| mImgtbl | 3 | 4 | 6 | 11 |
| mAdd | 3 | 4 | 6 | 11 |
| mViewer | 4 | 5 | 6 | 12 |
| Workflow‡ | 60 | 300 | 707 | 1500 |

‡ The sum of the tasks is a rounded value.

Table 4: Number of tasks in the Montage workflow

### Cybershake/Seismology

The *Seismology* workflow application is a common data-intensive analysis pattern used by many seismologists, visible twice in Section 2.1.6 with *Cybershake*. Currently, *WFBench* is not able to create full *Cybershake* workflow applications. So, instead of using the original *Seismology* for testing, we simulated the first pattern of data aggregation in the *Cybershake* workflow. In this case, the *sG1IterDecon* tasks represent the *SeismogramSynthesis* and the *wrapper_siftSTFByMisfit* represent the *ZipSeis*. To simplify we will refer to the *Cybershake/Seismology* workflow application as *Seismology*. Figure 30 represents the **DAG** of a small *Seismology* instance with 8 tasks.

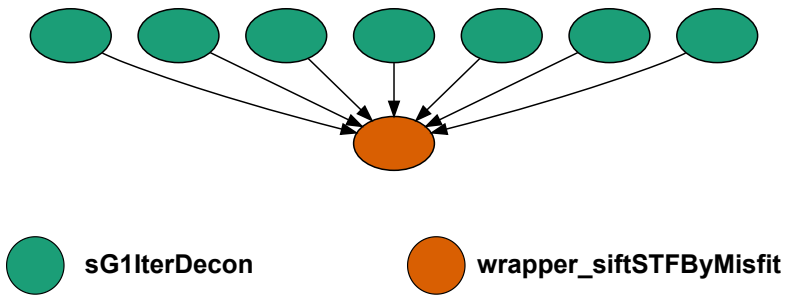



Figure 30: Seismology workflow **DAG** with 8 tasks.

The parameters for the percentage of threads doing CPU work were defined according to the CPU

utilisation of the task type and the CPU work was defined afterwards according to the execution time of each kind of task, both according to [82]. All input and output files were automatically specified when using *SpecsGen* with an input/output file factor of 100. Table 5 presents all the specification file parameters defined by us.

| Task | CPU Work | CPU Threads (%)† |
|---|---|---|
| sG1IterDecon | 67000 | 0.9 |
| wrapper_siftSTFByMisfit | 33000 | 0.1 |

† In the specification described from 0 to 1.

Table 5: Parameters used in the Specification file

| | #Tasks | | | |
|---|---|---|---|---|
| sG1IterDecon | 100 | 122 | 172 | 197 |
| wrapper_siftSTFByMisfit | 1 | 1 | 1 | 1 |
| Workflow‡ | 103 | 125 | 175 | 200 |

‡ The sum of the tasks is a rounded value.

Table 6: Number of tasks in the workflow

This workflow is the smallest workflow test case we present as it needs a large amount of resources to be executed. Table 6 shows the number of tasks by type for all *seisnmology* workflows executed. It is important to emphasise that this workflow contains only one *wrapper_siftSTFByMisfit* task, which is always the final task to be executed, as it requires the output from all other tasks.

### 1000Genome

The *1000Genome* workflow has no difference from the workflow application presented in Section 2.1.6. Just like *Montage*, this workflow has multiple branches that may have different numbers of tasks. Figure 31 illustrates a small branch of a *1000Genome* workflow. For the *1000Genome* workflow, the input and output files were automatically specified when using *SpecsGen* with an input/output file factor of 0.01. The parameters for the percentage of threads doing CPU work for the *individuals*, *mutation_overlap*and *frequency* tasks were defined according to the CPU utilisation of the task type in [107]. The others were estimated based on similar tasks of *Montage* and *Seismology* workflow applications.

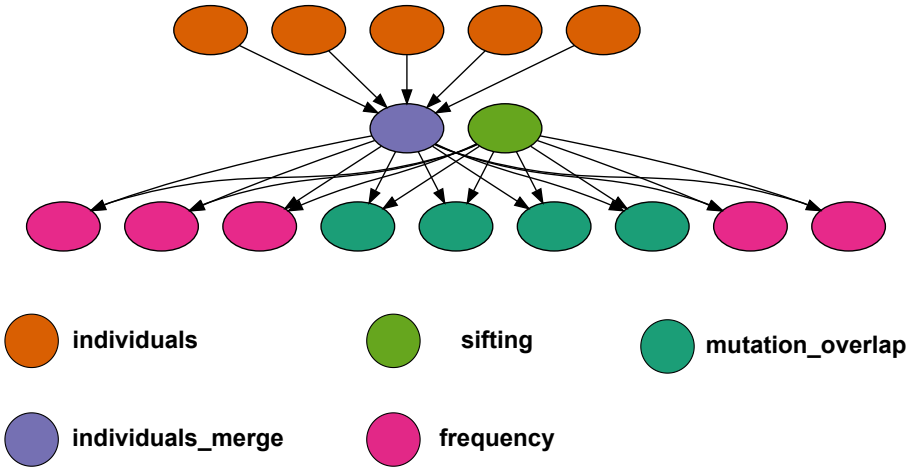



Figure 31: 1000Genome workflow branch **DAG** with 16 tasks.

The CPU work was determined based on the execution time of each task type. Given that the workflow application is very time-consuming, the execution time for the *individuals* tasks was reduced by a factor of 45000, reducing the execution time of each task of this type from 8 hours and 47 minutes to 42 minutes, and by a factor of 100 for the other tasks, in comparison with the time presented in [92]. This was done in order to simulate a smaller workflow application of this type. Table 7 presents all the specification file parameters of *percent-cpu* and *cpu-work*.

Each task of the type *individuals_merge* can merge approximately the output of 25 (or less) *individuals* tasks. In each branch of this workflow, there is only a single task with the type *individuals_merge* and *sifting*, as shown in Table 8 where it is possible to see the number of tasks by type for the *1000Genome* workflows generated. We generated workflows with 2, 5, 15 and 36 branches.

| Task | CPU Work | CPU Threads (%)† |
|---|---|---|
| individuals | 2700 | 0.6 |
| individuals_merge | 100 | 0.6 |
| sifting | 5110 | 0.8 |
| mutation_overlap | 1400 | 0.6 |
| frequency | 680 | 0.5 |

† In the specification described from 0 to 1.

Table 7: Parameters used in the Specification file

| | #Tasks | | | |
|---|---|---|---|---|
| individuals | 25 | 135 | 389 | 925 |
| individuals_merge | 2 | 5 | 15 | 36 |
| sifting | 2 | 5 | 15 | 36 |
| mutation_overlap | 18 | 44 | 113 | 271 |
| frequency | 17 | 41 | 114 | 278 |
| Workflow‡ | 66 | 232 | 648 | 1548 |

‡ The sum of the tasks is a rounded value.

Table 8: Number of tasks in *1000Genome*

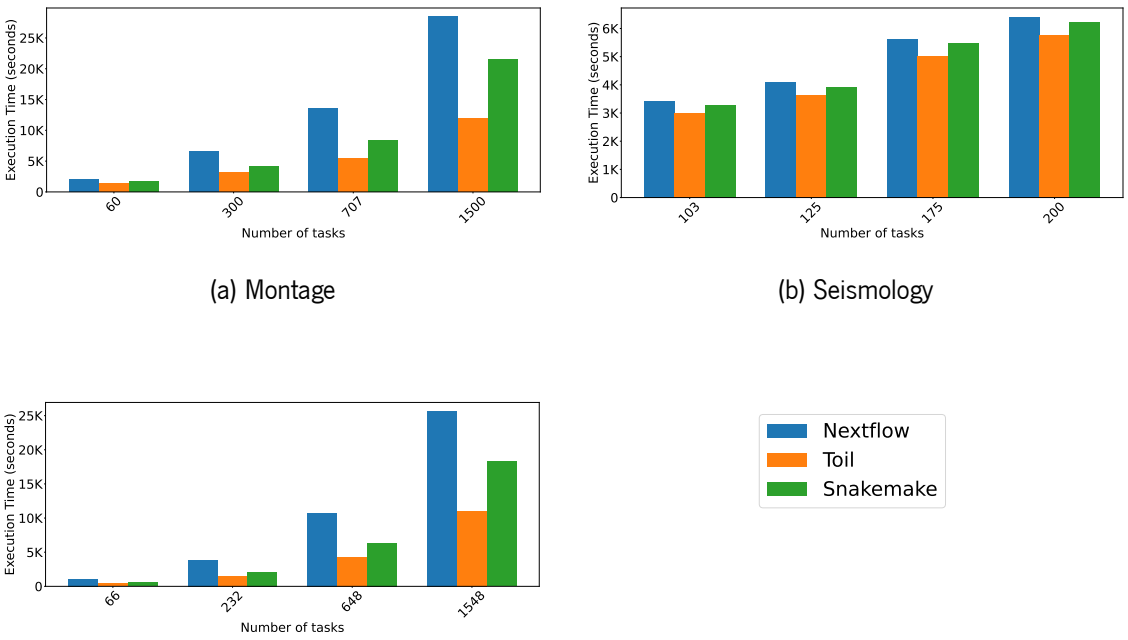
## 4.2 Results

This section presents the results obtained from executing the workflow applications using the *Huggin* framework. A detailed explanation and analysis will follow these results.

All test cases were executed five times, and the results presented are the average values of these executions. The tests were conducted on a Slurm **HPC** cluster with four compute nodes, each equipped with an Intel Core i7-8700 CPU (8 cores, 3.20 GHz), 32GB RAM, and a Toshiba XG6 M.2 NVMe with 240 GB of storage, running Rocky Linux 9.2 with kernel version 6.1.85.

### 4.2.1 Elapsed Time

#### Execution time




(a) Montage

(b) Seismology

(c) 1000Genome

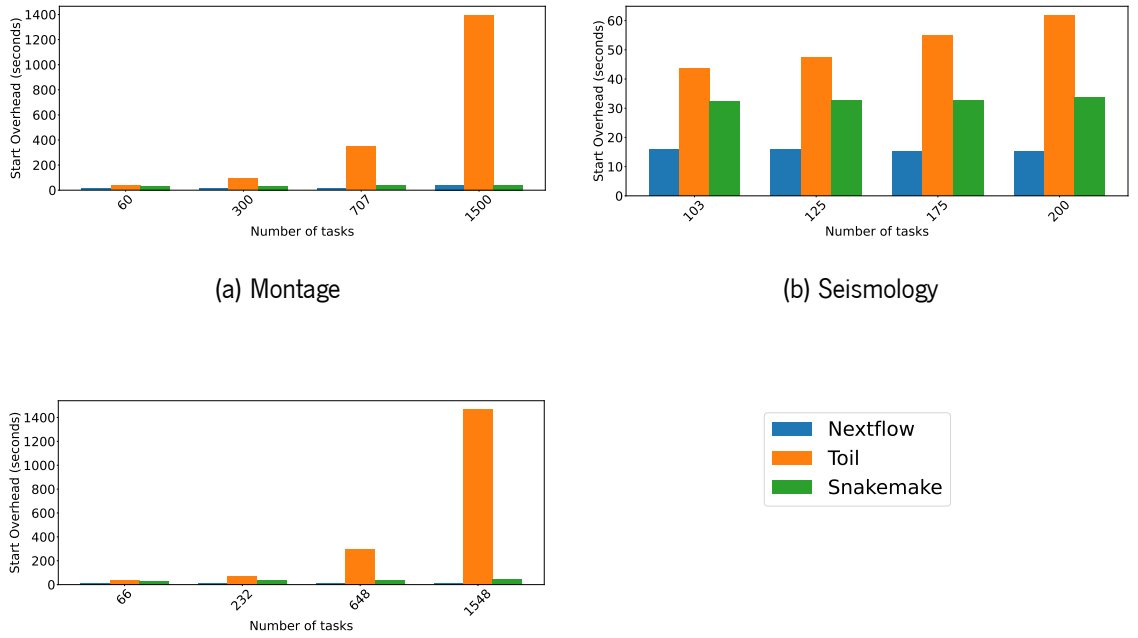
Figure 32: Mean execution time for each workflow application by **WMS**s

As illustrated in Figure 32b, Toil exhibits the shortest execution times among the *Seismology* workflows. Since the number of tasks in this workflow is considerably smaller compared to the other tested workflows, the difference in execution times is less pronounced. This discrepancy becomes more pronounced when the number of tasks exceeds 300, as illustrated in the *Montage* and *1000Genome* workflow applications, depicted in Figure 32a and Figure 32c, respectively. In a general analysis of the elapsed time for work-

flow application executions, it is evident that Toil demonstrates the shortest execution times, followed by Snakemake, with Nextflow showing the longest.

The lower execution times shown by Toil may be explained by the way dependencies are described in **CWL**. While in the Snakemake and Nextflow **DSL**s the task dependencies are implicit, in **CWL** these are explicitly described by the user and Toil builds the workflow **DAG** based on them. Other property that provides an advantage is the creation of the environment for a tasks at the beginning of the workflow. The high workflow execution times recorded with Nextflow may be due to the approach taken regarding the workflow dependencies, as it follows a reactive dataflow model. In this approach, the dependencies are not pre-computed and stored as **DAG**, like in Snakemake and Toil.

### Execution Start Overhead




(a) Montage

(b) Seismology

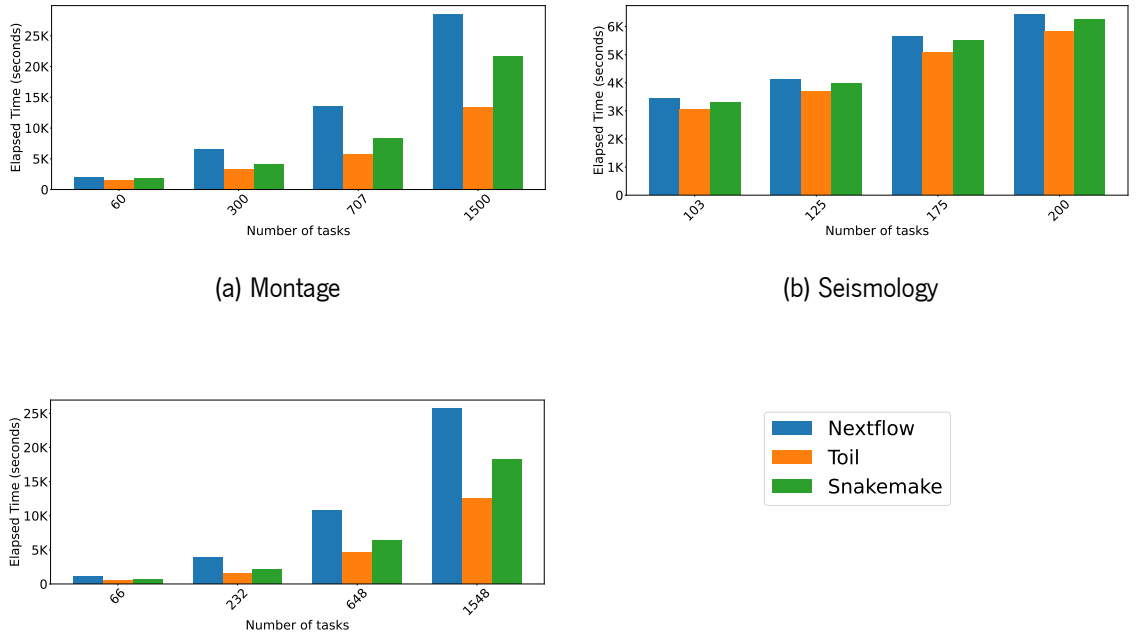
(c) 1000Genome

Figure 33: Mean Start Overhead for each workflow application by **WMS**s

Figures 33a, 33b and 33c reveal that Toil exhibits the highest start overhead among the **WMS**s under consideration. As previously explained, Toil prepares all directories and files needed for each task at the beginning of the workflow, resulting in higher start overheads. This overhead increases exponentially with the number of tasks because with increasing the number of tasks there are more environments to be created/prepared. In contrast, Snakemake and Nextflow both demonstrate relatively constant overheads, independent of the number of tasks in the workflow applications.

Nextflow is the **WMS** with the lowest start overhead as it does not need any preparation prior to the workflow application execution other than the workflow description file parsing.

### Total Elapsed Time




(a) Montage

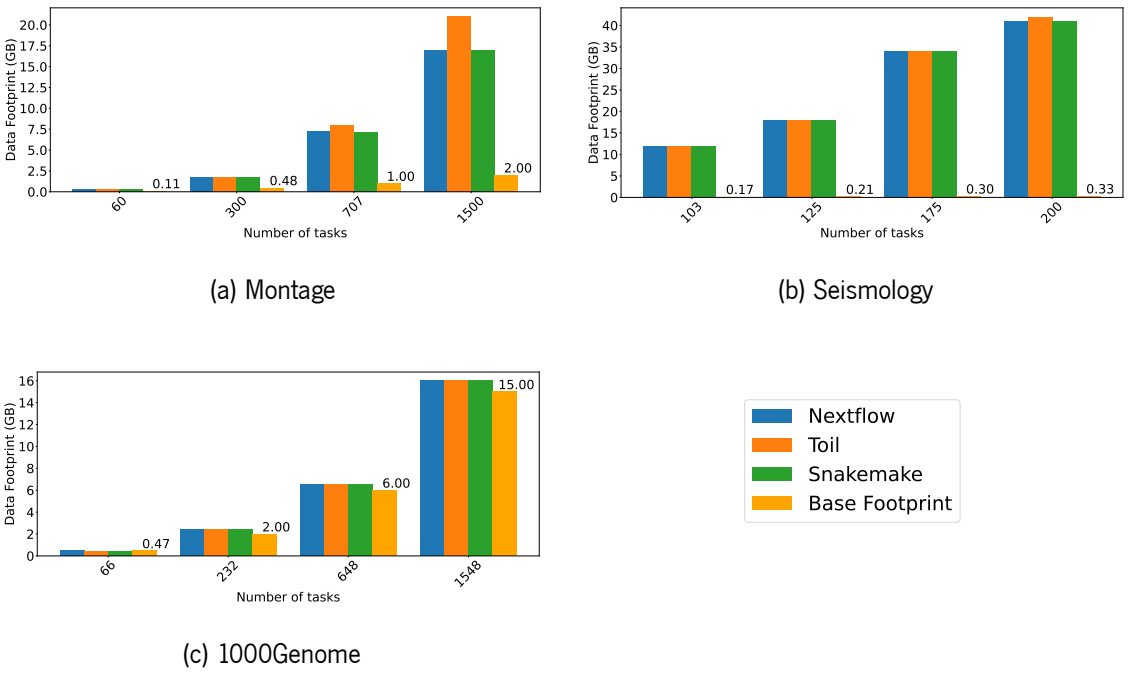
(b) Seismology

(c) 1000Genome

Figure 34: Mean elapsed time (execution time + start overhead time) for each workflow application by **WMS**s

Once the **WMS**s under test have distinct workflow building and scheduling strategies it becomes important to analyse the total workflow execution. Adding the execution and execution start overhead times, shown in Figures 32 and 33, we get the total elapsed time of the workflow application executions depicted in Figures 34a, 34b and 34c. These figures show that despite having the highest start overhead to total elapsed time ratios, Toil still records the shortest elapsed times of all workflow executions. This evidence suggests that the strategy employed by Toil is the most efficient of the **WMS**s tested in a multi-node environment.

### 4.2.2 Data Footprint




(a) Montage

(b) Seismology

(c) 1000Genome

Figure 35: Mean Data Footprint for each workflow application by **WMS**s

Figures 35a, 35b and 35c show the data footprint, i. e. the total amount of data generated, processed and stored in the workflow execution process, including data related to tasks, logs, reports and other **WMS** components. The base footprint, depicted in yellow, represents the data volume of all files involved in manual workflow execution (input, output, and intermediate files).

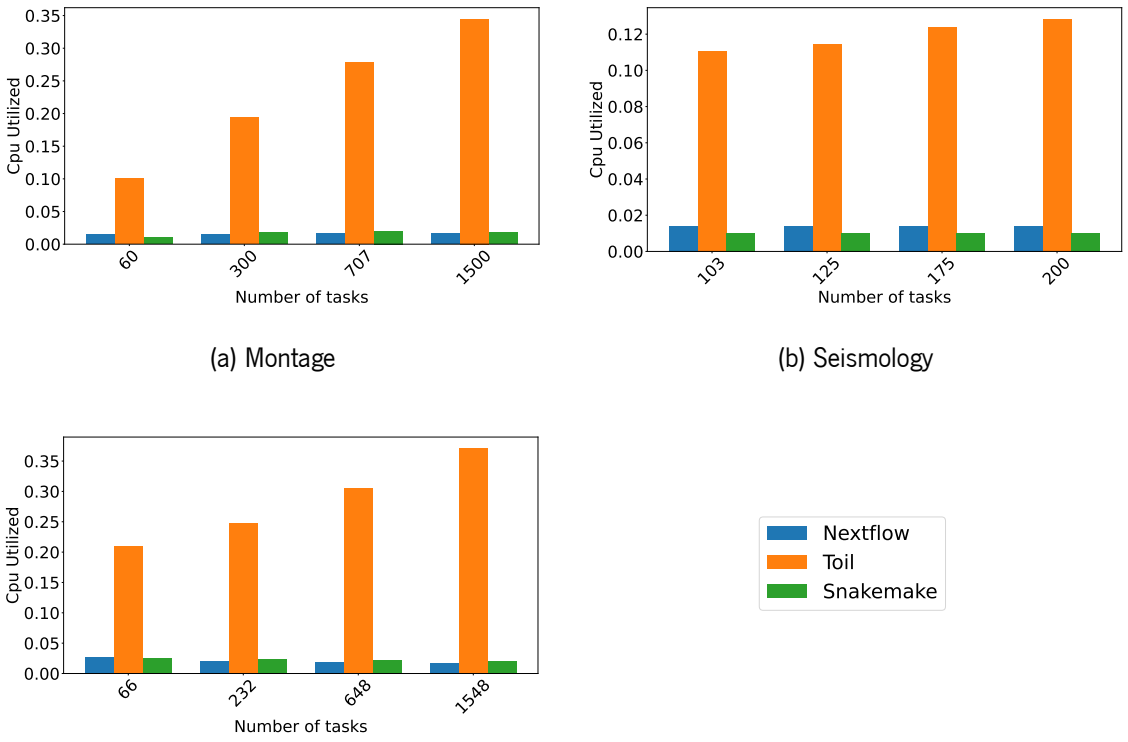
These figures demonstrate that all **WMS**s have a considerable data footprint overhead, reaching more than 10000% in the *Seismology* workflow execution and more than 900% in the *Montage* workflow application. The *1000Genome* workflow application records the smallest data footprint overhead with a maximum of approximately 6% of overhead.

By analysing Figures 35a and 35c, we can infer that the base footprint does not influence the data footprint produced by the **WMS**s, as the *Montage* workflow application has a significantly lower data footprint than the *1000Genome* workflow. The results shown in Figure 35b also indicate that the data footprint is not solely dependent on the number of tasks in workflow applications. This is evidenced by the Seismology workflow application, which has the lowest number of tasks and the highest recorded data footprint. One potential explanation for this phenomenon is that the number of dependencies between tasks influences the data footprint, having *Seismology* with the higher number of dependencies (between the *sG1IterDecon* and *wrapper_siftSTFByMisfit* tasks), followed by *Montage* (between the *mDiffFit* and

*mConcatFit* tasks) and lastly *1000Genome* (between *individuals* and *individuals_merge* tasks).

When examining the data footprint of **WMS**s in the *Montage* workflow (Figure 35a), Toil exhibits a higher data footprint than Snakemake and Nextflow, which show identical values. This difference becomes more significant as the number of tasks increases, more noticeable in Figure 35a. This may be due to the amount of information in the reports generated from workflow executions using Toil, which stores information about task execution metrics for each task executed. For the *Seismology* and the *1000Genome* workflow applications (Figures 35b and 35c) all three **WMS**s show similar data footprint volumes. Figure 35b illustrates that Toil exhibits a greater data footprint when executing a *Seismology* workflow application comprising 200 tasks.

### 4.2.3 CPU Utilisation




(a) Montage

(b) Seismology

(c) 1000Genome

Figure 36: Mean CPU utilisation of the orchestrator of each workflow application by **WMS**s

The figures 36a, 36b and 36c show that Toil has the highest CPU overhead of all the **WMS**s tested. The CPU utilisation of Toil's orchestrator seems to increase with the number of tasks in the workflow. In contrast, Nextflow and Snakemake's orchestrators show approximately constant CPU utilisation, both with similar values, always under 0.02 for *Montage* and *Seismology* workflows (Figures 36a and 36b) and

always under 0.03 for the *1000Genome* workflow application (Figure 36c). In a general perspective, these results show that the node where the **WMS** is running is underutilised, not fully leveraging the performance of the resources available.

### 4.2.4 Discussion

In this dissertation, we analysed and compared Toil (with **CWL**), Snakemake and Nextflow in terms of execution time, execution start overhead, data footprint and CPU utilisation, with the *Huginn* framework. Table 9 shows an abstract comparative analysis between the **WMS**s we tested, to facilitate comparison. It is important to note that this table summarises the results and may omit outliers.

| | Workflow Type | Toil (CWL) | Snakemake | Nextflow |
|---|---|---|---|---|
| Total elapsed time | Montage | 1 | 2 | 3 |
| | Seismology | 1 | 2 | 3 |
| | 1000Genome | 1 | 2 | 3 |
| Data footprint† | Montage | 3 | 2 | 1 |
| | Seismology | 3 | 2 | 1 |
| | 1000Genome | 1 | 1 | 1 |
| CPU utilisation† | Montage | 3 | 2 | 1 |
| | Seismology | 3 | 1 | 2 |
| | 1000Genome | 3 | 2 | 1 |

† The **WMS**s are ordered from 1 (best) to 3 (worst) in a general perspective. When two **WMS**s show similar results, they will be represented by the same number to indicate that they perform comparably or have equivalent metrics.

Table 9: Comparative results of the **WMS**s analysed.

The results of articles analysing single-node executions of workflow applications, for example, article [47], demonstrate that Snakemake, in comparison to Nextflow and Toil (with **CWL**), has the shortest execution times in such execution environments. Contrasting with these articles, when executing workflows in a multi-node execution environment Toil (with **CWL**) exhibits shorter execution times than Nextflow and Snakemake.

In our test cases, Toil records the most favourable execution times for all workflows, although it also

exhibits the highest execution start overheads. This contrasts with Nextflow, which presents the longest executions and the lowest start overheads. The times recorded by Snakemake, both for execution time and start overhead, fall within the range of those recorded by Toil and Nextflow. Although still present, this difference is less significant in workflow applications with a smaller number of tasks (less than 100).

The data footprint was consistent across all workflow applications under examination, exhibiting only slight variations between Toil and the other **WMS**s (none exceeding 1GB). Upon analysing the results of the *Montage* and *Seismology* workflows, it becomes evident that the data footprint of Toil-based executions tends to increase at a comparatively accelerated rate in comparison to those executed with Nextflow and Snakemake. All of our test workflow applications are characterised by a relatively small to medium number of tasks, between 60 and 1550. Consequently, if the data footprint increases in a consistent pattern when larger workflows are employed, this may present a challenge when executing such workflows using Toil.

**WMS**s typically have low CPU usage because the workflow orchestration process itself requires minimal computing power. This is supported by our results, which show that even Toil, the **WMS** with the highest CPU utilisation, does not exceed 0.4 CPUs. The CPU utilisation seems to increase proportionally with the number of tasks present in the workflow executed with Toil. For the Snakemake and Nextflow executions, which have similar CPU usage, the values remain consistently below 0.04, despite the number of tasks in the workflow applications executed.

### Further Considerations

While performance metrics are crucial when selecting the appropriate **WMS** for conducting our workflow applications, it is imperative to consider other features that may assist users in identifying the optimal tool for their specific use case. Thus we compare Toil, Snakemake and Nextflow in some of these general features to provide a complete comparison of these **WMS**s. We summarise all features in Table 10.

Perhaps the most important feature in a **WMS** is the workflow description language, not only limiting the specification of workflows but also allowing the use of workflows in/from other **WMS**s. Toil enables users to describe workflows with the two most used standard workflow specification languages (**CWL** and **WDL**) or using its Python API. Snakemake and Nextflow each have their specific **DSL** to describe workflows, both allowing the usage of their implementation language (Python for Snakemake and Groovy for Nextflow). Snakemake also affords users the ability to use simple **CWL** workflows alongside its **DSL** and to export straightforward workflows to **CWL**. All **WMS**s allow users to define workflows in a modular fashion, enabling users to describe workflows, sub-workflows and tasks in different files easing their reusability.

Toil and Nextflow only allow tasks to execute scripts or command-line tools while Snakemake has a

broader range of options, having integration with multiple programming languages, such as Python and R, and uses wrappers to export and seamlessly use some of the most used tools in the Snakemake community.

Toil and Snakemake are both available via the Python package manager, only needing a Python environment on the machine. In the case of using non-local executors, Slurm in our case, Snakemake requires the installation of the dedicated executor plugin. Nextflow does not require installation but requires Java (Version 11 or later up to 22) to be installed.

All these **WMS**s have recently and constantly updated documentation majorly differing in the amount of training material available online. Toil appears to have the most outdated documentation, with a lack of examples, in comparison to the other two **WMS**s. It leverages the abundance of examples and learning resources on the standard workflow specifications languages, which may discourage users from utilising Toil's Python API. The documentation for Snakemake and Nextflow is more appealing and modern, featuring a large number of examples and tutorials. Additionally, Nextflow offers a training platform, *Nextflow Training*[1], that distinguishes it from Snakemake.

Another key feature of **WMS**s is the report generation to analyse each task's requirements/properties. Toil generates comprehensive reports that illustrate the CPU, memory, and disk usage metrics for the workflow and each task within it. However, a significant limitation of these reports is their format, as they are only available in a text file that lacks a clear visual representation. Snakemake creates HTML reports with a clear visual representation but with less detail. Nextflow stands out for having the option to provide reports in three different ways with different levels of complexity and information: execution report (a single HTML document which includes many useful metrics about a workflow execution), timeline report (timeline for all processes executed in the workflow) and trace report (information about each process executed in your pipeline script and better for parsing). None of the three **WMS**s under examination have built monitoring systems, but all allow the integration of external solutions to do so.

Pipeline initiatives store ready-to-use workflows, simplifying their publication and discovery. While general-purpose repositories support multiple workflow languages, Snakemake and Nextflow maintain their own repositories due to high usage. Toil lacks a dedicated repository but is not disadvantaged by this, as it relies on standard workflow specifications that have established repositories.

[1] https://training.nextflow.io/

| | Toil | Snakemake | Nextflow |
|---|---|---|---|
| Workflow Description Languages | Python API; **CWL**; **WDL** (Partially) | Snakemake **DSL**; **CWL** (Partially) | Nextflow **DSL** |
| Installation | Python Package Manager. Requires Python Environment | Python Package Manager. Requires Python Environment | Requires Java Environment. No installation required, just the executable. |
| Documentation | Updated with few examples for each language. | Extensive and updated with examples and tutorials. | Extensive and updated with examples and tutorials. Has a dedicated training platform |
| Monitoring | Only using external software | Only using external software | Only using external software |
| Reports | Detailed reports in a textfile with CPU and memory metrics | Customisable HTML reports, with basic information on the workflow execution. | Multiple detailed and customisable reports, with information on the workflow and task execution. |
| Modularisation | Tasks and workflows can be imported from other files and workflows can consist of a set of linked sub-workflows. | Users can import workflows, **CWL** workflows and rules from other Snakefiles. | Possible to import processes from other files and use other workflows as sub-workflows. |
| Script and Tool Integration | Scripts and command line tools. | Scripts, command line tools and integration with Python, R, R Markdown, Julia, Rust, Bash and Jupyter notebook. Wrappers for most used tools | Scripts and command line tools. |
| Pipeline Initiatives | None specific for Toil python API. Default repositories for **WDL** and **CWL**. (WARP; BioWDL; Dockstore; WorkflowHub) | Dedicated Repositories for the **DSL** (snakePipes; Snakemake-Workflows)† | Dedicated Repositories for the **DSL** (nf-core)† |

† These are repositories specified for the **DSL**s. Workflows for these **WMS**s are also available in default repositories.

Table 10: Summary of other characteristics that influence the choice of a **WMS**.

# Chapter 5

# Conclusions and future work

## 5.1 Conclusions

The role of **Workflow Management System (WMS)**s in scientific research is becoming increasingly significant. They simplify the implementation, execution and reproducibility of robust and complex pipelines across various execution environments.

In this thesis, we developed the *Huginn* framework leveraging the WfCommon project to create workflows, execute them and generate performance reports based on the workflow execution. Using our framework, we conducted several tests in an **HPC**-cluster with three different workflows from different scientific domains to compare three of the most used **WMS**s regarding several performance metrics. In addition, these **WMS**s were evaluated in comparison with other features that are more specifically oriented towards usability.

Furthermore, this dissertation comprises a study of the most commonly used workflows, **WMS**s and workflow community projects. Based on this study, we extended the *WFBench* tool to support the creation of workflow applications in two new workflow specification languages. During this process, probably for being the latest WfCommons Project and frequently under development, we noted that *WFBench* has many inconsistencies and under-tested features. This was the main difficulty of the implementation and extension of this tool.

Finally, we hope that this dissertation will assist users in selecting the most appropriate **WMS** for their requirements, taking into account performance and usability.

## 5.2 Prospect for future work

From a future perspective, we hope to see our framework extended to other popular or emerging **WMS**s to provide a wider comparison. In a different possible direction, this framework might be the launching

pad for a robust **WMS**s benchmark tool, with more features and test scenarios.

In alignment with the objectives of the WfCommons Project, there is considerable potential and interest within the workflow community to expand all tools that have emerged from this project, to ease investigation within the workflow community.

A final point for future research would be the implementation of a remote execution strategy for the **WMS**s, enabling the management of workflows to take place on a separate machine that requires fewer computing resources. Similarly, a comparative analysis of the execution of single-sited and multi-sited workflows with different **WMS**s would be interesting.

# Bibliography


[1] Aleyna Dilan Kiran, Mehmet Can Ay, and Jens Allmer. Criteria for the evaluation of workflow management systems for scientific data analysis. *Journal of Bioinformatics and Systems Biology*, 06(02), 2023.

[2] GLIMPSE: Galactic Legacy Infrared Mid-Plane Survey Extraordinaire, 2024. https://irsa.ipac.caltech.edu/data/SPITZER/GLIMPSE/, accessed at 09/08/2024.

[3] Bharathi, Shishir and Chervenak, Ann and Deelman, Ewa and Mehta, Gaurang and Su, Mei-Hui and Vahi, Karan. Characterization of scientific workflows. In *2008 Third Workshop on Workflows in Support of Large-Scale Science*, pages 1–10, 2008.

[4] Rafael Ferreira Da Silva, George Papadimitriou, Loïc Pottier, and et al. 1000genome-workflow, 2024. https://github.com/pegasus-isi/1000genome-workflow, accessed at 26/08/2024.

[5] Dante Domizzi Sánchez-Gallegos and Diana Di Luccio and Sokol Kosta and J.L. Gonzalez-Compean and Raffaele Montella. An efficient pattern-based approach for workflow supporting large-scale science: The DagOnStar experience. *Future Generation Computer Systems*, 122:187–203, 2021.

[6] Anubhav Jain. FireWorks 2.0.3 documentation. https://materialsproject.github.io/fireworks/, accessed at 21/08/2024.

[7] Helena Rasche, Nate Coraor, and Simon Gladman. Galaxy Pulsar. https://training.galaxyproject.org/training-material/topics/admin/tutorials/ansible-galaxy/tutorial.html, accessed at 22/08/2024.

[8] Aymen Alsaadi, Logan Ward, Andre Merzky, Kyle Chard, Ian Foster, Shantenu Jha, and Matteo Turilli. Radical-pilot and parsl: Executing heterogeneous workflows on hpc platforms. pages 27–34, 11 2022.

[9] Pegasus overview. https://pegasus.isi.edu/about/, accessed at 22/08/2024.

[10] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. PyCOMPSs: Parallel computational workflows in Python. *The International Journal of High Performance Computing Applications*, 31(1):66–82, 2017.

[11] Seqera Team. Nextflow training, 2023. https://training.nextflow.io/, accessed at 30/08/2024.

[12] StreamFlow 0.1 documentation. https://streamflow.di.unito.it/documentation/latest/index.html, accessed at 21/07/2024.

[13] UCSC Computational Genomics Lab. Snakemake 8.16.0 documentation. https://toil.readthedocs.io/en/latest/index.html#toil-documentation, accessed at 24/08/2024.

[14] WfCommons Team. Wfcommons 1.1.dev documentation, 2023. https://docs.wfcommons.org/en/latest/, accessed at 30/08/2024.

[15] Hollingsworth, David. WfMC: Workflow Reference Model. Specification, Workflow Management Coalition, 1995. TC00-1003.

[16] Domenico Talia. Workflow systems for science: Concepts and tools. *ISRN Softw. Eng.*, 2013:1–15, January 2013.

[17] Adam Barker and Jano van Hemert. Scientific workflow: A survey and research directions. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, pages 746–753, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[18] Roger Barga and Dennis Gannon. Scientific versus business workflows. In Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields, editors, *Workflows for e-Science: Scientific Workflows for Grids*, pages 9–16. Springer London, London, 2007.

[19] Emily Gersema. Nobel prize-winning discovery on gravitational waves came about with contributions from usc scientists, Sep 2023.

[20] Ewa Deelman, Karan Vahi, Mats Rynge, Rajiv Mayani, Rafael Ferreira da Silva, George Papadimitriou, and Miron Livny. The Evolution of the Pegasus Workflow Management Software. *Computing in Science Engineering*, 21(4):22–36, 2019.

[21] Sandeep Suresh Cranganore, Vincenzo De Maio, Ivona Brandic, and Ewa Deelman. Paving the way to hybrid quantum–classical scientific workflows. *Future Generation Computer Systems*, 158:346–366, 2024.

[22] Van Rossum, Guido and Drake, Fred L and others. *Python reference manual*, volume 111. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[23] Travis E. Oliphant. Python for scientific computing. *Computing in Science Engineering*, 9(3):10–20, 2007.

[24] Sarah Cohen-Boulakia and Khalid Belhajjame and Olivier Collin and Jérôme Chopard and Christine Froidevaux and Alban Gaignard and Konrad Hinsen and Pierre Larmande and Yvan Le Bras and Frédéric Lemoine and Fabien Mareuil and Hervé Ménager and Christophe Pradal and Christophe Blanchet. Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities. *Future Generation Computer Systems*, 75:284–298, 2017.

[25] Mulone, Alberto and Awad, Sherine and Chiarugi, Davide and Aldinucci, Marco. Porting the Variant Calling Pipeline for NGS data in cloud-HPC environment. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, volume , pages 1858–1863, 2023.

[26] Dennis Gannon, Ewa Deelman, Matthew Shields, and Ian Taylor. Introduction. In Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields, editors, *Workflows for e-Science: Scientific Workflows for Grids*, pages 1–8. Springer London, London, 2007.

[27] Wilkinson, Mark D. and Dumontier, Michel and Aalbersberg, IJsbrand Jan and Appleton, Gabrielle and Axton, Myles and Baak, Arie and Blomberg, Niklas and Boiten, Jan-Willem and da Silva Santos, Luiz Bonino and Bourne, Philip E. and et al. The Fair Guiding Principles for Scientific Data Management and Stewardship. *Scientific Data*, 3(1), 2016.

[28] Jacobsen, Annika and de Miranda Azevedo, Ricardo and Juty, Nick and Batista, Dominique and Coles, Simon and Cornet, Ronald and Courtot, Mélanie and Crosas, Mercè and Dumontier, Michel and et al. FAIR Principles: Interpretations and Implementation Considerations. *Data Intelligence*, 2(1-2):10–29, 01 2020.

[29] Gil, Yolanda and Deelman, Ewa and Ellisman, Mark and Fahringer, Thomas and Fox, Geoffrey and Gannon, Dennis and Goble, Carole and Livny, Miron and Moreau, Luc and Myers, Jim. Examining the Challenges of Scientific Workflows. *Computer*, 40(12):24–32, 2007.

[30] Heinis, T. and Pautasso, C. and Alonso, G. Design and Evaluation of an Autonomic Workflow Engine. In *Second International Conference on Autonomic Computing (ICAC'05)*, pages 27–38, 2005.

[31] Ferreira da Silva, Rafael and Badia, Rosa M. and et al. Workflows Community Summit 2022: A Roadmap Revolution. Technical Report ORNL/TM–2023/2885, Oak Ridge National Laboratory, March 2023.

[32] Jorge Ejarque, Rosa M. Badia, Loïc Albertin, Giovanni Aloisio, Enrico Baglione, Yolanda Becerra, Stefan Boschert, Julian R. Berlin, Alessandro D'Anca, Donatello Elia, François Exertier, Sandro Fiore, José Flich, Arnau Folch, and et. al. Enabling dynamic and intelligent workflows for HPC, data analytics, and AI convergence. *Future Generation Computer Systems*, 134:414–429, 2022.

[33] Jin, Hongwei and Raghavan, Krishnan and Papadimitriou, George and Wang, Cong and Mandal, Anirban and Deelman, Ewa and Balaprakash, Prasanna. Self-supervised Learning for Anomaly Detection in Computational Workflows. *arXiv preprint arXiv:2310.01247*, 2023.

[34] Suter, Frederic and Ferreira da Silva, Rafael and Gainaru, Ana and Klasky, Scott. Driving Next-Generation Workflows from the Data Plane. In *19th IEEE Conference on eScience*, page , 2023.

[35] Ola Spjuth, Erik Bongcam-Rudloff, and Guillermo Carrasco et al. Hernández. "experiences with workflows for automating data-intensive bioinformatics". *Biology Direct*, 10(1), Aug 2015.

[36] Ferreira da Silva, Rafael and Casanova, Henri and Chard, Kyle and Laney, Dan and Ahn, Dong and Jha, Shantenu and et al. Workflows Community Summit: Bringing the Scientific Workflows Community Together. Technical report, 2021.

[37] Hong-Linh Truong and Schahram Dustdar. Composable cost estimation and monitoring for computational applications in cloud computing environments. *Procedia Computer Science*, 1(1):2175–2184, 2010. ICCS 2010.

[38] Elsharkawy, Amr and To, Xiao-Ting Michelle and Seitz, Philipp and Chen, Yanbin and Stade, Yannick and Geiger, Manuel and Huang, Qunsheng and Guo, Xiaorang and Ansari, Muhammad Arslan and Ruefenacht, Martin and Schulz, Laura and Karlsson, Sven and Mendl, Christian B. and Kranzlmüller, Dieter and Schulz, Martin. Challenges in HPCQC Integration. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 02, pages 405–406, 2023.

[39] Benjamin Weder, Johanna Barzen, Frank Leymann, and Daniel Vietz. Quantum software development lifecycle. In Manuel A. Serrano, Ricardo Pérez-Castillo, and Mario Piattini, editors, *Quantum Software Engineering*, pages 61–83, Cham, 2022. Springer International Publishing.

[40] Badia, R.M. and Ayguadé, E. and Labarta, J. Workflows for science: a challenge when facing the convergence of HPC and big data. *Supercomputing frontiers and innovations*, 4(1):27–47, 2017.

[41] Colonnelli, Iacopo and Cantalupo, Barbara and Merelli, Ivan and Aldinucci, Marco. StreamFlow: Cross-Breeding Cloud With HPC. *IEEE Transactions on Emerging Topics in Computing*, 9(4):1723–1737, 2021.

[42] Antypas, K. B. and Bard, D. J. and Blaschke, J. P. and Shane Canon, R. and Enders, Bjoern and Shankar, Mallikarjun Arjun and Somnath, Suhas and Stansberry, Dale and Uram, Thomas D. and Wilkinson, Sean R. Enabling discovery data science through cross-facility workflows. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 3671–3680, 2021.

[43] Rafael Ferreira da Silva, Rosa Filgueira, Ilia Pietri, Ming Jiang, Rizos Sakellariou, and Ewa Deelman. A characterization of workflow management systems for extreme-scale applications. *Future Generation Computer Systems*, 75:228–238, 2017.

[44] Azza E. Ahmed, Joshua M. Allen, Tajesvi Bhat, Prakruthi Burra, Christina E. Fliege, Steven N. Hart, Jacob R. Heldenbrand, Matthew E. Hudson, Dave Deandre Istanto, Michael T. Kalmbach, and et al. Design considerations for workflow management systems use in production genomics research and the clinic. *Scientific Reports*, 11(1), Nov 2021.

[45] Michael Jackson, Kostas Kavoussanakis, and Edward W. J. Wallace. Using prototyping to choose a bioinformatics workflow management system. *PLOS Computational Biology*, 17(2):1–13, 02 2021.

[46] Wratten, Laura and Wilm, Andreas and Göke, Jonathan. Reproducible, scalable, and shareable analysis pipelines with bioinformatics workflow managers. *Nature Methods*, 18(10):1161–1168, 2021.

[47] Elise Larsonneur, Jonathan Mercier, Nicolas Wiart, Edith Le Floch, Olivier Delhomme, and Vincent Meyer. Evaluating workflow management systems: A bioinformatics use case. In *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 2773–2775, 2018.

[48] Chee Sun Liew, Malcolm P. Atkinson, Michelle Galea, Tan Fong Ang, Paul Martin, and Jano I. Van Hemert. Scientific Workflows: Moving Across Paradigms. *ACM Comput. Surv.*, 49(4), dec 2016.

[49] Mölder, Felix and Jablonski, Kim Philipp and Letcher, Brice and Hall, Michael B. and Tomkins-Tinch, Christopher H. and Sochat, Vanessa and Forster, Jan and Lee, Soohyun and Twardziok, Sven O. and Kanitz, Alexander and et al. Sustainable Data Analysis with snakemake. *F1000Research*, 10:33, 2021.

[50] Ayer, Vidya M. and Miguez, Sheila and Toby, Brian H. Why scientists should learn to program in Python. *Powder Diffraction*, 29(S2):S48–S64, 2014.

[51] Bassi, Sebastian. A Primer on Python for Life Science Researchers. *PLOS Computational Biology*, 3(11):1–6, 11 2007.

[52] TIOBE Software BV. Tiobe index for july 2024, Jul 2024. https://www.tiobe.com/tiobe-index/, accessed: 22/07/2024.

[53] Rachel Stephens. RedMonk Top 20 Languages Over Time: January 2024, 2024.

[54] Carlo Zapponi. Githut 2.0, a small place to discover languages in github. Accessed: 22/07/2024.

[55] Matthew Shields. Control- versus data-driven workflows. In Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields, editors, *Workflows for e-Science: Scientific Workflows for Grids*, pages 167–173, London, 2007. Springer London.

[56] Thulasiraman, Krishnaiyan and Swamy, Madisetti NS. *Graphs: theory and algorithms*. John Wiley & Sons, 1992.

[57] Bertram Ludäscher, Shawn Bowers, and Timothy McPhillips. *Scientific Workflows*, pages 2507–2511. Springer US, Boston, MA, 2009.

[58] Malcolm Atkinson and Sandra Gesing and Johan Montagnat and Ian Taylor. Scientific workflows: Past, present and future. *Future Generation Computer Systems*, 75:216–227, 2017.

[59] Ewa Deelman and Dennis Gannon and Matthew Shields and Ian Taylor. Workflows and e-Science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528–540, 2009.

[60] Marine Djaffardjy, George Marchment, Clémence Sebe, Raphaël Blanchet, Khalid Belhajjame, Alban Gaignard, Frédéric Lemoine, and Sarah Cohen-Boulakia. Developing and reusing bioinformatics data analysis pipelines using scientific workflow systems. *Computational and Structural Biotechnology Journal*, 21:2075–2085, 2023.

[61] The Galaxy Community. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2022 update. *Nucleic Acids Research*, 50(W1):W345–W351, 04 2022.

[62] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. Parsl: Pervasive parallel programming in python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19. ACM, June 2019.

[63] Samuel Lampa, Martin Dahlö, Jonathan Alvarsson, and Ola Spjuth. SciPipe: A workflow library for agile development of complex and dynamic bioinformatics pipelines. *GigaScience*, 8(5):giz044, 04 2019.

[64] Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. Nextflow enables reproducible computational workflows. *Nature Biotechnology*, 35(4):316–319, Apr 2017.

[65] Ivo Jimenez, Michael Sevilla, Noah Watkins, Carlos Maltzahn, Jay Lofstead, Kathryn Mohror, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. The popper convention: Making reproducible systems evaluation practical. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1561–1570, 2017.

[66] YAML Language Development Team. Yaml ain't markup language (yaml™) version 1.2 revision 1.2.2 (2021-10-01).

[67] Michael R. Crusoe, Sanne Abeln, Alexandru Iosup, Peter Amstutz, John Chilton, Nebojša Tijanić, Hervé Ménager, Stian Soiland-Reyes, Bogdan Gavrilović, Carole Goble, and et al. Methods included: Standardizing computational reuse and portability with the common workflow language. *Communications of the ACM*, 65(6):54–63, Jun 2022.

[68] Van der Auwera G Voss K and Gentry J. Full-stack genomics pipelining with gatk4 + wdl + cromwell [version 1; not peer reviewed]., 2017.

[69] Kate Voss, Geraldine Van Der Auwera, and Jeff Gentry. Full-stack genomics pipelining with gatk4 + wdl + cromwell [version 1; not peer reviewed], 2017.

[70] J. Köster, D. Lähnemann, C. Meester, M. Hall, F. Vieira, and M. et al. Lund. Snakemake 8.16.0 documentation. https://snakemake.readthedocs.io/en/stable/, accessed at 24/08/2024.

[71] Pegasus Team. Welcome to Pegasus WMS's documentation! https://pegasus.isi.edu/documentation/index.html, accessed at 24/07/2024.

[72] Goble, Carole and Cohen-Boulakia, Sarah and Soiland-Reyes, Stian and Garijo, Daniel and Gil, Yolanda and Crusoe, Michael R. and Peters, Kristian and Schober, Daniel. FAIR Computational Workflows. *Data Intelligence*, 2(1-2):108–121, 01 2020.

[73] Carole Goble and Stian Soiland-Reyes and Finn Bacall and Stuart Owen and Alan Williams and Ignacio Eguinoa and Bert Droesbeke and Simone Leo and Luca Pireddu and Laura Rodríguez-Navas and José Mª Fernández and Salvador Capella-Gutierrez and Hervé Ménager and Björn Grüning and Beatriz Serrano-Solano and Philip Ewels and Frederik Coppens. Implementing FAIR Digital Objects in the EOSC-Life Workflow Collaboratory, March 2021.

[74] Yuen, Denis and Cabansay, Louise and Duncan, Andrew and Luu, Gary and Hogue, Gregory and Overbeck, Charles and Perez, Natalie and Shands, Walt and Steinberg, David and Reid, Chaz and Olunwa, Nneka and Hansen, Richard and Sheets, Elizabeth and O'Farrell, Ash and Cullion, Kim and O'Connor, Brian D. and Paten, Benedict and Stein, Lincoln. The Dockstore: enhancing a community platform for sharing reproducible and accessible computational protocols. *Nucleic Acids Research*, 49(W1):W624–W632, 05 2021.

[75] Philip A Ewels, Alexander Peltzer, Sven Fillinger, Johannes Alneberg, Harshil Patel, Andreas Wilm, Maxime Ulysse Garcia, Paolo Di Tommaso, and Sven Nahnsen. nf-core: Community curated bioinformatics pipelines. preprint on webpage at https://www.biorxiv.org/content/10.1101/610741v1, Apr 2019.

[76] Workflows Community Initiative, 2024. https://workflows.community/.

[77] Du, Xin and Tang, Songtao and Lu, Zhihui and Gai, Keke and Wu, Jie and Hung, Patrick C. K. Scientific Workflows in IoT Environments: A Data Placement Strategy Based on Heterogeneous Edge-Cloud Computing. *ACM Trans. Manage. Inf. Syst.*, 13(4), aug 2022.

[78] Montage, An Astronomical Image Mosaic Image, 2024. http://montage.ipac.caltech.edu/, accessed at 22/08/2024.

[79] FITS, Flexible Image Transport System, 2024. https://fits.gsfc.nasa.gov/, accessed at 22/08/2024.

[80] Montage Image Mosaic Software: About Montage, 2024. https://montageblog.wordpress.com/about/, accessed at 22/08/2024.

[81] Jonathan Livny, Hidayat Teonadi, Miron Livny, and Matthew K. Waldor. High-Throughput, Kingdom-Wide Prediction and Annotation of Bacterial Non-Coding RNAs. *PLOS ONE*, 3(9):1–12, 09 2008.

[82] Gideon Juve, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3):682–692, 2013. Special Section: Recent Developments in High Performance Computing and Security.

[83] James Frey. Condor DAGMan: Handling inter-job dependencies, 2002.

[84] CyberShake Workflows, 2024. https://strike.scec.org/scecpedia/CyberShake_Workflows, accessed at 23/08/2024.

[85] CyberShake, 2024. https://strike.scec.org/scecpedia/CyberShake, accessed at 23/08/2024.

[86] Collin D Capano. *Searching for gravitational waves from compact binary coalescence using LIGO and virgo data*. Syracuse University, 2011.

[87] LIGO, Laser Interferometer Gravitational-Wave Observatory, 2024. https://www.ligo.caltech.edu/, accessed at 23/08/2024.

[88] Maq: Mapping and Assembly with Qualities, 2024. https://maq.sourceforge.net/, accessed at 23/08/2024.

[89] Illumina. This is the Genome Era., 2024. https://www.illumina.com/, accessed at 23/08/2024.

[90] Adam Auton, Gonçalo R. Abecasis, David M. Altshuler, Lorraine H. Toji, and et al. A global reference for human genetic variation. *Nature*, 526(7571):68–74, Sep 2015.

[91] Susan Fairley, Ernesto Lowy-Gallego, Emily Perry, and Paul Flicek. The International Genome Sample Resource (IGSR) collection of open human genomic variation resources. *Nucleic Acids Research*, 48(D1):D941–D947, 10 2019.

[92] Rafael Ferreira da Silva, Rosa Filgueira, Ewa Deelman, Erola Pairo-Castineira, Ian M. Overton, and Malcolm P. Atkinson. Using simple pid-inspired controllers for online resilient resource management of distributed scientific workflows. *Future Generation Computer Systems*, 95:615–628, 2019.

[93] Rafael Ferreira Da Silva, George Papadimitriou, Loïc Pottier, and et al. Execution Instances for 1000Genome Workflow, 2024. https://github.com/wfcommons/WfInstances/blob/main/pegasus/1000genome/README.md, accessed at 26/08/2024.

[94] Laurens Versluis, Roland Mathá, Sacheendra Talluri, Tim Hegeman, Radu Prodan, Ewa Deelman, and Alexandru Iosup. The workflow trace archive: Open-access data from public and private computing infrastructures. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2170–2184, 2020.

[95] J. Armando Barron-Lugo and Jose Luis Gonzalez-Compean and Jesus Carretero and Ivan Lopez-Arevalo and Raffaele Montella. A novel transversal processing model to build environmental big data services in the cloud. *Environmental Modelling Software*, 144:105173, 2021.

[96] Jain, Anubhav and Ong, Shyue Ping and Chen, Wei and Medasani, Bharat and Qu, Xiaohui and Kocher, Michael and Brafman, Miriam and Petretto, Guido and Rignanese, Gian-Marco and Hautier, Geoffroy and Gunter, Daniel and Persson, Kristin A. FireWorks: a dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience*, 27(17):5037–5059, 2015. CPE-14-0307.R2.

[97] Helena Rasche, Nate Coraor, and Simon et al. Gladman. Galaxy Training: Galaxy Server administration. https://training.galaxyproject.org/training-material/topics/admin/, accessed at 16/08/2024.

[98] Helena Rasche, Nate Coraor, and Simon Gladman. Galaxy Pulsar, 2024. https://github.com/galaxyproject/pulsar/, accessed at 22/08/2024.

[99] What is COMPSs? https://compss-doc.readthedocs.io/en/stable/Sections/0_Intro.html, accessed at 21/07/2024.

[100] Enric Tejedor and Rosa M. Badia. Comp superscalar: Bringing grid superscalar and gcm together. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 185–193, 2008.

[101] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Dataflow variables are spectacularly expressive in concurrent programming, Nextflow Version v24.04. documentation. https://nextflow.io/docs/latest/index.html, accessed at 30/07/2024.

[102] John Vivian, Arjun Arkal Rao, Frank Austin Nothaft, Christopher Ketchum, Joel Armstrong, Adam Novak, Jacob Pfeil, Jake Narkizian, Alden D Deran, Audrey Musselman-Brown, and et al. Toil enables reproducible, open source, big biomedical data analyses. *Nature Biotechnology*, 35(4):314–316, Apr 2017.

[103] Elise Larsonneur and Edith Le Floch, jul 2018. LODSEQ (v1.1.0). Zenodo. https://doi.org/10.5281/zenodo.1321178.

[104] Tainã Coleman and Henri Casanova and Loïc Pottier and Manav Kaushik and Ewa Deelman and Rafael Ferreira da Silva. WfCommons: A framework for enabling scientific workflow research and development. *Future Generation Computer Systems*, 128:16–27, 2022.

[105] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. Makeflow. *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, 1:1–13, May 2012.

[106] Justin M. Wozniak, Timothy G. Armstrong, Michael Wilde, Daniel S. Katz, Ewing Lusk, and Ian T. Foster. Swift/t: scalable data flow programming for many-task applications. *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, Feb 2013.

[107] Tainã Coleman, Henri Casanova, Ketan Maheshwari, Loïc Pottier, Sean R. Wilkinson, Justin Wozniak, Frédéric Suter, Mallikarjun Shankar, and Rafael Ferreira Da Silva. WfBench: Automated Generation of Scientific Workflow Benchmarks. In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 100–111, 2022.