Universidad de Cantabria

Facultad de Ciencias

In Situ Workflow Orchestration in HPC to Optimize Extreme Climate Simulation Workflows

Gestión in situ de workflows en HPC para optimizar simulaciones climáticas extremas

Trabajo de Fin de Máster para acceder al

Máster en Ingeniería Informática

Autor: Pablo Goitia González

Director: Borja Pérez Pavón

Co-Director: Manuel Giménez de Castro

Marzo - 2026

Agradecimientos

Este trabajo supone el final de toda una etapa, y me gustaría aprovechar la oportunidad para hacer mi particular reconocimiento a todas las personas que me han acompañado a través de ella.

A mi familia, por haber confiado siempre en mi, por haberme dado todo lo que necesitaba para convertirme en la persona que soy hoy.

A María, que merece una mención de honor por su forma de hacer que todo parezca un poquito más fácil, por nunca dejar de apoyarme.

A mis amigos, por estar ahí en todo momento.

A mis compañeros del Barcelona Supercomputing Center y al equipo de Autosubmit, que han hecho de este trabajo algo apasionante. Especialmente a Manuel Giménez, que estuvo conmigo al pie del cañón durante todo el desarrollo del proyecto. *Moltes gràcies, amics.*

A Borja Pérez, que, junto a Manuel, revisó este trabajo sin descanso hasta asegurarse de que no quedaran hilos sueltos. También a los demás profesores de la Facultad de Ciencias, por su compromiso y por inspirar con su buen hacer.

Finalmente, a quien haya dedicado tiempo a leer este documento y conocer mi trabajo.

Resumen

Las simulaciones climáticas se ejecutan en plataformas de computación de alto rendimiento (HPC, por sus siglas en inglés) organizadas en flujos de tareas que integran todos los pasos para la ejecución completa del modelo, el procesamiento de datos y otras tareas de gestión. Con el aumento continuo en la capacidad de cómputo de estas máquinas a lo largo de los años, la precisión y la resolución de las simulaciones han alcanzado niveles nunca vistos.

En este contexto, la Comisión Europea puso en marcha la iniciativa Destination Earth, en la que el Barcelona Supercomputing Center desempeña un papel fundamental, cuyo objetivo es desarrollar un gemelo digital de la Tierra. Esta iniciativa pretende poner en marcha simulaciones climáticas de muy alta resolución que se acoplan a aplicaciones que consumen sus datos a medida que se producen. Para hacer frente al reto que supone el procesamiento de los cientos de terabytes que genera cada simulación, el proyecto Climate DT, que desarrolla el gemelo digital para la adaptación al cambio climático, implementó un enfoque de streaming de datos. Esto significa que cualquier retraso entre el momento de producción de los datos del modelo climático y su posterior consumo por parte de las aplicaciones de posproceso provoca un desajuste en el flujo de tareas, lo que da lugar a retrasos inaceptables en el tiempo total de ejecución. Esto plantea retos sin precedentes en lo que respecta a la gestión del workflow.

Una de las principales causas de los desajustes que se producen habitualmente radica en el gran tiempo que cada una de las miles de tareas del workflow pasa en las colas de los planificadores de los supercomputadores, como Slurm. Para abordar este problema, la comunidad propuso agregar las tareas del workflow en un único envío a la plataforma sin alterar su lógica de ejecución, una técnica conocida como agregación de tareas. Estudios anteriores han demostrado la eficacia de este enfoque para los workflows climáticos, con resultados prometedores. Sin embargo, la implementación actual es limitada, ya que la ejecución de las tareas dentro de una reserva de recursos sigue dependiendo del gestor de workflows, que no es capaz de realizar la gestión granular de la agregación que una herramienta dedicada podría hacer de forma más práctica.

Para resolver esta limitación, en este trabajo se propone integrar software ya existente en HPC en el gestor de workflows Autosubmit para permitir la gestión *in situ* de las tareas agregadas, como Flux, entre otros. Esta integración tiene como objetivo abstraer tanto a los desarrolladores como a los usuarios de la complejidad de la gestión de los recursos de supercomputación, proporcionando una interfaz fácil de usar. El enfoque propuesto se valida utilizando el workflow de Climate DT en un entorno de supercomputación real.

* * *

Palabras clave – Supercomputación, Autosubmit, Flux Framework, Climate DT, Destination Earth.

Abstract

Climate simulations are executed on High-Performance Computing (HPC) platforms, organized in workflows that involve all the steps for the complete execution of the model, data processing, and management. With the sustained increase in the computing capacity of these machines over the years, the accuracy and resolution of climate simulations have reached levels never seen before.

In this context, the European Commission launched the Destination Earth initiative, in which the Barcelona Supercomputing Center plays a key role, aimed at developing a digital twin of the Earth. This initiative seeks to operationalize the running of very high-resolution climate simulations that are coupled with applications that consume their data as it is produced. In order to address the challenge of processing the hundreds of terabytes that each single simulation involves, the Climate DT project, which develops the digital twin for climate change adaptation, implemented a data streaming approach. This means that any delay between the production time of the climate model data and the subsequent consumption by the post-processing applications results in a workflow misalignment, leading to unacceptable delays in the total execution time. This poses unprecedented challenges on the workflow management side.

One of the main causes of the misalignments that commonly occur lies in the long time that each of the many thousands of tasks of the workflow spends in the queues of the HPC job schedulers, such as Slurm. To address this issue, the community proposed to aggregate workflow tasks into a single submission to the HPC without altering their execution logic—a technique known as *task aggregation*. Previous studies have demonstrated the effectiveness of this approach for climate workflows, yielding promising results. However, the current implementation is limited, as the task execution within an allocation still relies on the workflow manager, which is not able to perform the fine-grained workflow orchestration that a dedicated tool could do in a convenient way.

To overcome this limitation, this work proposes integrating existing HPC software into the Autosubmit Workflow Manager to enable *in situ* orchestration of aggregated tasks, such as the Flux Framework, among others. This integration aims to abstract both developers and users from the complexity of managing supercomputing resources, providing an easy-to-use interface. The proposed approach is validated using the Climate DT workflow in a real supercomputing environment.

* * *

Keywords – Supercomputing, Wrappers, Task aggregation, In situ workflow management.

Contents

List of Figures

List of Tables

List of Codes

Acronyms

API Application Programming Interface.

AR6 Sixth Assessment Report.

ASYPD Actual Simulated Years per Day.

BSC Barcelona Supercomputing Center.

CLI Command Line Interface.

CMIP6 Coupled Model Intercomparison Project Phase 6.

CMS Compact Muon Solenoid.

CNS Centro Nacional de Supercomputación (Spain).

CWL Common Workflow Language.

DAG Directed Acyclic Graph.

DN Data Notifier.

ECMWF European Centre for Medium-Range Weather Forecasts.

ESM Earth System Model.

FIFO First-In First-Out.

GCM Global Climate Model.

GUI Graphical User Interface.

HPC High Performance Computing.

HTC High Throughput Computing.

IFS Integrated Forecasting System.

IPCC Intergovernmental Panel on Climate Change.

LHC Large Hadron Collider.

LLNL Lawrence Livermore National Laboratory.

MPI Message Passing Interface.

NEMO Nucleus for European Modelling of the Ocean.

NUMA Non-Uniform Memory Access.

OPA One Pass Algorithms.

QoS Quality of Service.

WDL Workflow Description Language.

WMS Workflow Management System.

Chapter 1

Introduction

This first chapter describes how the queue waiting times in supercomputer schedulers—often referred to as workload managers—influence the performance of large-scale climate simulations, and an approach designed to help climate scientists alleviate this issue through tools that automate task aggregation is presented. To that end, the chapter outlines the underlying motivation, describes the objectives in detail, and justifies how the proposed solution benefits the community.

1.1 Motivation

Over the years, rapid and sustained advances have transformed the field of High Performance Computing (HPC), also referred to as supercomputing, resulting in the current landscape of large heterogeneous computing machines that are approaching and even surpassing the exascale ("TOP500", 2026). Supercomputing is particularly relevant in modern society, encompassing numerous applications in different disciplines, ranging from Artificial Intelligence to material sciences (Pyzer-Knapp et al., 2022), as well as life sciences with projects such as the development of the digital twin of the human body (Tang et al., 2024), particle physics experiments such as ATLAS (Collado Soto et al., 2025) or Compact Muon Solenoid (CMS) (Delgado Peris et al., 2025) at the Large Hadron Collider (LHC), or Earth sciences, where the development of the digital twin of the Earth is particularly relevant, within the European Commission's *Destination Earth* project (Hoffmann et al., 2023).

Because of the shared nature of the supercomputers, it is natural for applications from each of these and other areas to compete for computing resources on the same machine. As might be expected, due to the limited amount of resources and their high demand, it is necessary to establish scheduling mechanisms that distribute them among users. When the available HPC resources are not enough to satisfy the demand of submitted jobs—which are the compute units sent to be executed remotely—the time they have to spend in the workload manager queue as they wait for the requested resources to be released increases. This period from the job submission by the user until the scheduler assigns the resources for it to be executed is called *queue time*.

In the field of Earth Sciences, which constitutes the focus of this work, the powerful machines available today allow us to take climate simulations to levels never seen before. For this reason, working on the computational optimization of Earth System Models (ESMs) and the respective workflows that run on them is crucial due to the temporal extension and precision of the simulations that are carried out and, likewise, the economic and energy costs that these entail.

At the computational stage, climate workflows are becoming increasingly complex and incorporate advanced techniques for managing the data produced and its post-process.

This involves an increasing number of tasks—the atomic unit of computation within the workflow—to each simulation cycle, each subject to its own queue time, causing the aggregated queue time for the entire workflow to grow.

The impact of these times on climate simulations was quantified in the work of Acosta et al. (2024). The authors indicate that queue time accounts for an overhead of between 10% and 20% of the total execution time in simulations carried out for the Coupled Model Intercomparison Project Phase 6 (CMIP6) project, which is particularly relevant and influential in this field due to its contributions to the development of the Sixth Assessment Report (AR6) of the Intergovernmental Panel on Climate Change (IPCC), with its consequent impact on decisions that shape climate policies.

Being aware of this concern within the community, Marciani et al. (2025) evaluated a technique that consists of grouping multiple workflow tasks into a single submission to the HPC platform, known as *task aggregation*. Task aggregation has a dual purpose. One is to reduce the number of submissions to the HPC platform to meet its Quality of Service (QoS) constraints. Another is to reduce queue times in vertically structured workflows—also known as *chained*—in which tasks are executed sequentially, which is predominant in ESMs workflows.

This work served as a theoretical basis for Goitia et al. (2024), which put the task aggregation approach to the test in production climate simulation workflows, using nearly half a million CPU hours on renowned platforms such as MareNostrum 4, from the Barcelona Supercomputing Center (BSC) in Spain, the subsequent MareNostrum 5, and MeluXina, from LuxProvide in Luxembourg. The results showed that queue times were reduced by up to 12 times, increasing Actual Simulated Years per Day (ASYPD)—a performance metric for climate simulations recognized by the community that considers possible waits or retries in the execution time—by up to 23% in the case of the most congested platform, a value that lines up with the results of the aforementioned studies.

These tests in real environments were carried out using the EC-Earth3 model (Döscher et al., 2022), which is notable for its modularity, allowing users to choose different configurations of coupled Earth system components to simulate, and for participating in the CMIP6 exercise. The workflow associated with this model was developed in Autosubmit (Manubens et al., 2015), a powerful workflow manager designed at the BSC to meet the specific needs of climate research, and the authors adapted it to the needs of the experimentation by pruning the workflow to leave only resource-demanding tasks, which have a greater impact on scheduling, and applying vertical aggregations to them. However, the current implementation of the task aggregation mechanism in Autosubmit, known as "wrappers", is limited and complex, with the workflow manager itself being responsible for coordinating the execution of tasks that run remotely within the resource allocation in the cluster, considering that Autosubmit executes externally to the HPC platform.

Since task aggregation has proved its effectiveness in both simulated and real environments for improving the overall performance of vertically-structured workflows, this work proposes finding an alternative, straightforward, flexible, and scalable way to implement task aggregation in Autosubmit that allows both software users and developers to abstract themselves from the complexity of managing task aggregations on remote HPC platforms.

1.2 Objective

The purpose of this work is to evaluate existing tools in HPC that can serve as *in situ* mechanisms for task aggregation, and integrate the most promising one into the Autosubmit workflow manager. The objective of integrating an *in situ* manager is to overcome

the limitations that default methods for aggregating tasks present by offering better maintainability, scalability, portability, and straightforward resource management. The idea is to design an interface that facilitates communication between Autosubmit and one of these tools, which would be responsible of task and resource management within a single resource allocation in the cluster for all aggregated tasks. Autosubmit already offers up to two different mechanisms for task aggregation, but neither of them delegates management to another tool, so there is a software design challenge to address. The integration of the tool into Autosubmit was carried out following a methodology that guarantees the highest quality of the developed software, so that its users can benefit from a robust task aggregation mechanism.

The new functionality was tested by running the Climate DT workflow (Hadade et al., 2025; Roura-Adserias et al., 2025) with the new task aggregation method in a real environment, using MareNostrum 5. Climate DT is the digital twin for the adaptation to climate change, developed within the framework of the Destination Earth project.

The Climate DT workflow was selected for the experimentation in order to address the queue time problem that concerns its developers. The workflow consists of all the tasks required to execute the model and post-process the output. The simulation is divided into multiple tasks covering different time periods to ensure frequent checkpointing during execution and to prevent the loss of all the data in the case of a failure in the computing infrastructure. This data is then processed by the climate applications that calculate, for example, solar or wind energy indicators (Lacima-Nadolnik et al., 2025).

Given the unprecedented volume of data generated by each simulation task and the need to remove it from the remote file system as quickly as possible, the workflow was designed with the data streaming paradigm in mind to move and use the data on demand (Roura-Adserias et al., 2024). However, this causes a misalignment between the production of data by the model and its consumption by the applications, with queue time being partially responsible for this. This work proposes to aggregate the climate application tasks in the workflow to mitigate this misalignment, reducing the total execution time—or time-to-solution.

1.3 Contribution

This study aims to improve a major workflow manager used for Earth sciences research, and to provide the Earth sciences community with an implementation that they can use as a reference to optimize their own applications and reduce queue times in their simulations. This is a problem that significantly affects the speed with which researchers can make their scientific output available to the community and society. However, given that this is a cross-domain issue, scientists and developers working in other areas are also encouraged to reproduce this solution in their own use cases.

In a more direct way, this work contributes by mitigating the long queue times in the Climate DT workflow, minimizing the time gap between the production and consumption of simulation data and, therefore, increasing the throughput.

This master's dissertation has been developed within the Computational Earth Sciences group of the Earth Sciences Department of the Barcelona Supercomputing Center-CNS, to which the author and co-supervisor of this manuscript belong.

1.4 Working plan

The work plan consisted, first, of exploring existing tools in HPC that could serve as *in situ* managers to enhance task aggregation in Autosubmit.

In a second stage, the selected tools were tested to verify that they meet the requirements of Autosubmit's wrapping functionality. These requirements include, among others, being a portable and robust solution, easy-to-use, and able to assign resources within the allocation dynamically. Based on the results of this analysis, the Flux scheduler was selected for integration into Autosubmit.

The third part of the project involved the development of the interface between Autosubmit and Flux. It was necessary to modify the Autosubmit source code to make it easier for users to use the new functionality while maintaining the existing task aggregation methods.

Then, the quality of the developed software was ensured by running tests in real environments using the Climate DT workflow with different task aggregation configurations. To perform this validation, the MareNostrum 5 supercomputer was used.

1.5 Document organization

This document is divided into seven chapters, the present being Chapter 1, which introduces this work. The remaining content is structured as follows:

- Chapter 2 provides both the theoretical and practical foundations of this work, based on previous studies, since task aggregation in climate simulation workflows is an area that has been recently researched and validated by other authors. In addition, it explains other concepts that are frequently referred to throughout this document: task aggregation, Earth System Models, the Autosubmit workflow manager, and HPC schedulers.
- *Chapter 3* analyzes existing HPC workflow management solutions that may serve as *in situ* engines for task aggregation.
- In *Chapter 4*, the software design is described, including an overview of the current wrapper implementation in Autosubmit and the proposed solution for the integration of the *in situ* manager.
- *Chapter 5* details the methodology for testing the new implementation, featuring reproducible experimentation in a real environment, and software component testing.
- In *Chapter 6*, the results of the experimentation are analyzed in depth. This chapter aims to evaluate whether the implementation is feasible as a wrapping method. It includes an additional analysis on the distribution of the wrapper sizes along the execution with already available data.
- Finally, *Chapter 7* presents the conclusions drawn from this work and discusses what the next steps could be or which aspects are susceptible to improvement.

Chapter 2

Background

This chapter introduces some concepts and tools that are particularly relevant to this work, such as task aggregation, Earth System Models, workflow managers, and workload managers. Subsequently, it contextualizes the work by analyzing the main conclusions of previous studies on task aggregation.

2.1 Task aggregation

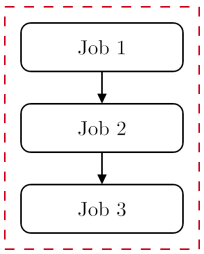
An aggregation of tasks is a set of tasks that belong to a workflow and are grouped into a single submission to the HPC platform. That single job is responsible for remotely managing the underlying workflow tasks, respecting interdependencies, execution logic, and other possible mechanisms of the workflow manager, such as checkpointing. The ways in which tasks can be aggregated are very diverse, so there exist different aggregation structures and policies. The policies will depend on the particular implementation of the workflow manager.

There are different types of aggregation that can be used to group workflow tasks based on the dependencies that exist between them, so the use of one type or another may vary depending on the particular needs of the workflow that contains them. Thus, tasks can be aggregated vertically, horizontally, or through a combination of these, which is referred to as hybrid or mixed aggregation.

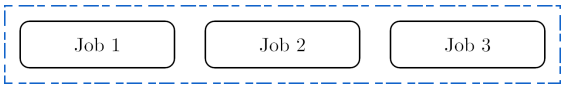
Vertical aggregations (Figure 2.1a) consist of groups of tasks that have sequential dependency. Therefore, the job that executes the entire aggregation will request the walltime corresponding to the sum of the individual walltime of the tasks that comprise it. The highest amount of resources requested by the individual tasks is assumed.

Horizontal aggregations, on the other hand, consist of grouping tasks without dependencies between them. That is, tasks that are executed concurrently (Figure 2.1b). In this case, what is aggregated are the resources needed by the tasks, while the walltime would be the largest individual one.

Hybrid or mixed aggregations emerged to cover more complex aggregation cases. For example, a vertical–horizontal aggregation (Figure 2.2a) is composed of multiple horizontal aggregations that are executed sequentially, and in the case of horizontal–vertical aggregations (Figure 2.2b), there are multiple vertical aggregations that are executed concurrently.

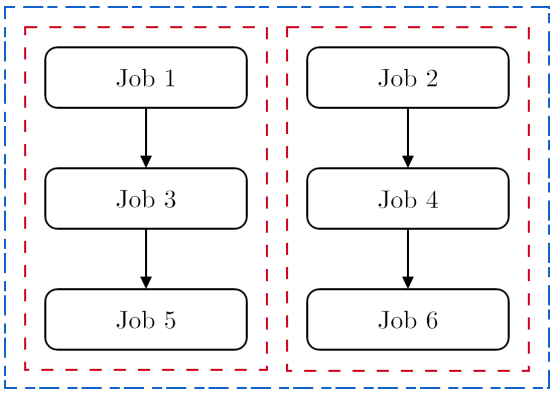


(a) An example of vertical aggregation. Each task inside the job depend on its previous, that is, they follow a sequential execution.

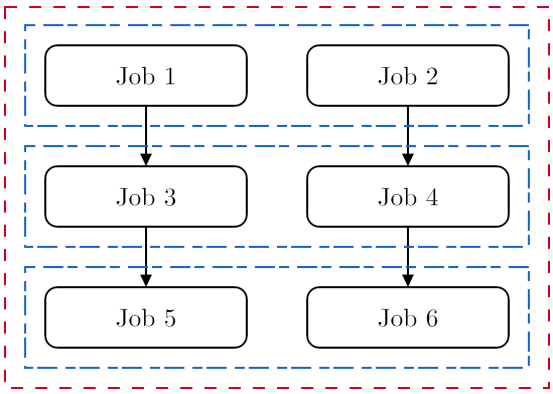


(b) An example of horizontal aggregation. The tasks within the job are independent and can be executed concurrently.

Figure 2.1: Horizontal (a) and vertical (b) aggregations. Figures by Goitia et al. (2024).



(a) Vertical–horizontal aggregation, horizontal aggregation of vertically aggregated tasks.



(b) Horizontal–vertical aggregation, vertical aggregation of horizontally aggregated tasks.

Figure 2.2: The hybrid aggregations: horizontal–vertical (a) and vertical–horizontal (b). Figures by Goitia et al. (2024).

2.2 Earth System Models

Earth System Models are highly complex numerical models that simulate the different components of the Earth system, such as the atmosphere, the ocean, land, and vegetation, among many others. These models are a natural response to the need to understand all events that occur in the Earth system and their causes. According to Döscher et al. (2022), ESMs are "the primary source of information for understanding the Earth's climate feedbacks, for attributing changes to specific drivers, for future climate projections and predictions, and for the development of mitigation policies".

Some well-known climate models include MONARCH (Klose et al., 2021), HERMES (Guevara et al., 2020), CALIOPE (Benavides et al., 2019), and EC-Earth (Döscher et al., 2022), a modular ESM developed by the EC-Earth consortium that, in its latest version (see request and technical report by Yang, 2025a, 2025b), couples different models depending on the desired configuration—GCM, ESM, or LR (Low Resolution)—such as renowned ECMWF's OpenIFS (Buizza et al., 2018) for the atmosphere and land, NEMO (Madec & the NEMO System Team, 2022) for the ocean, LPJ-GUESS (Lindeskog et al., 2013; Smith et al., 2014) for dynamic vegetation, or CaMa-Flood (Yamazaki et al., 2011) for flood/river dynamics.

2.3 The Autosubmit Workflow Manager

A Workflow Management System (WMS) is a tool that facilitates the creation, management, execution, and monitoring of workflows, which are sets of tasks that must be executed in a specific order given by the dependencies between them. Dependencies are often complex and are represented by the topology of the corresponding task graph.

Computational workflows are commonly made up of hundreds of thousands of tasks. While traditional methods to execute simulations consisted of manually accessing each supercomputer to submit the scripts, distributed workflow managers enable automating the submission of the tasks, also facilitating multi-platform execution. Today, there are more than 300 workflow managers (Amstutz et al., 2026), developed by organizations around the world, sometimes with certain use cases in mind.

The Autosubmit Workflow Manager is developed by the Earth Sciences Department of the Barcelona Supercomputing Center to support the specific demands of climate and air quality simulations. Besides a workflow manager, it is an experiment manager capable of coordinating task execution in a fully automated way, storing configurations, and ensuring data provenance. All logs and experiment configurations are stored and tracked, ensuring FAIRness (Puiggros et al., 2025). This makes it possible to know what happened during the execution, and allows the execution to be easily recovered in case of failure, or even to duplicate an experiment and re-execute it. Autosubmit collects information about experiments in individual databases, including the workflow execution history and data on the execution of each of its tasks.

It is a Python-based software that offers a complete working environment for its users through a Command Line Interface (CLI) that allows users to create, modify, and delete experiments. In addition, it offers a Graphical User Interface (GUI) that uses the Autosubmit Application Programming Interface (API) to allow users to monitor the execution of their workflows and access advanced performance statistics. Autosubmit provides advanced management features. One of these, which is the subject of this study, are the "wrappers", its task aggregation mechanism. It supports aggregation configurations that adapt to most workflows, as explained in Section 2.1, and offers multiple wrapping policies, as well as different methods for their remote management.

2.3.1 Autosubmit's hierarchical workflow architecture

As Autosubmit is focused on supporting climate simulations, the tasks in all experiments follow a special hierarchical structure as follows:

- **Startdate:** This is the first level of aggregation in the workflow hierarchy and is used to group different instances of the simulation cycle based on a start date that defines the initial day from which the Earth will be simulated. For example, simulations spanning from the pre-industrial to now start at 1/1/1850. There can be multiple startdates in the workflow.
- **Member:** This is the next level in the hierarchy. Due to the chaotic nature of the dynamics of the Earth's components, multiple concurrent simulations are executed with slightly perturbed initial conditions. Each of these is called a member.
- **Chunk:** is the minimum time simulated by a single task of the ESM. These sequential iterations are common practice in order to meet the constraints of the computing platform in terms of the amount of resources that can be requested each time, and

also to set up implicit checkpointing that enable the simulation experiment to be resumed from any point after a failure.

- **Split:** is the smallest division in the hierarchy. It is used when additional subdivisions need to be made in the chunks. This is used, for example, when the simulation of a chunk produces monthly data but postprocessing expects it in daily batches.

The simplified workflow in Figure 2.3 contains an example of this architecture, which does not take into account possible splits. If splits are used, the *SIM* section and the *"Postprocess subworkflow"* would potentially be divided, although this choice is up to the workflow developer.

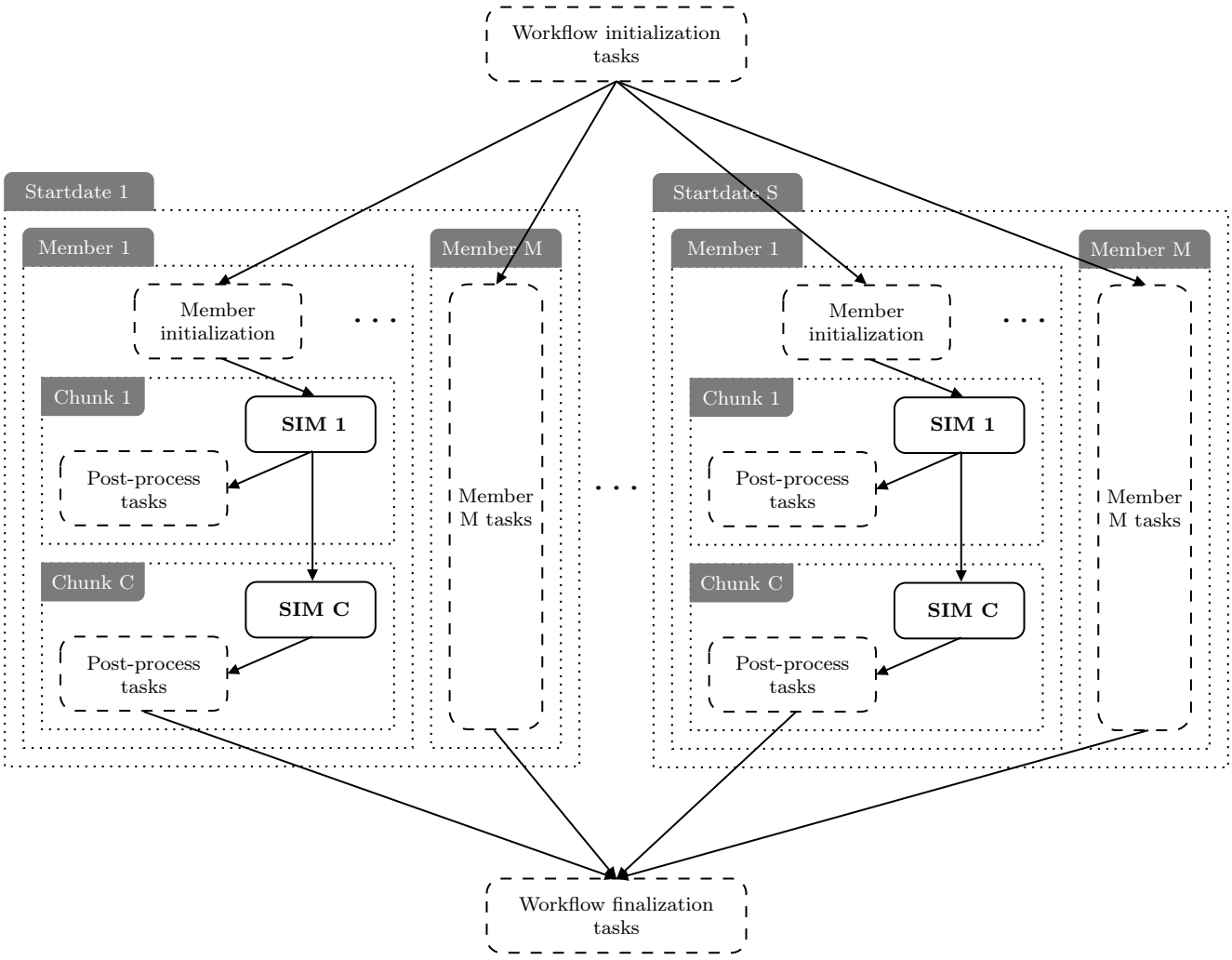


Figure 2.3: The task architecture of the Autosubmit Workflow Manager represented on a fictitious workflow. Graph illustrates a possible configuration for S startdates, M members, and C chunks. Splits are not represented.

2.3.2 Autosubmit support for task aggregation

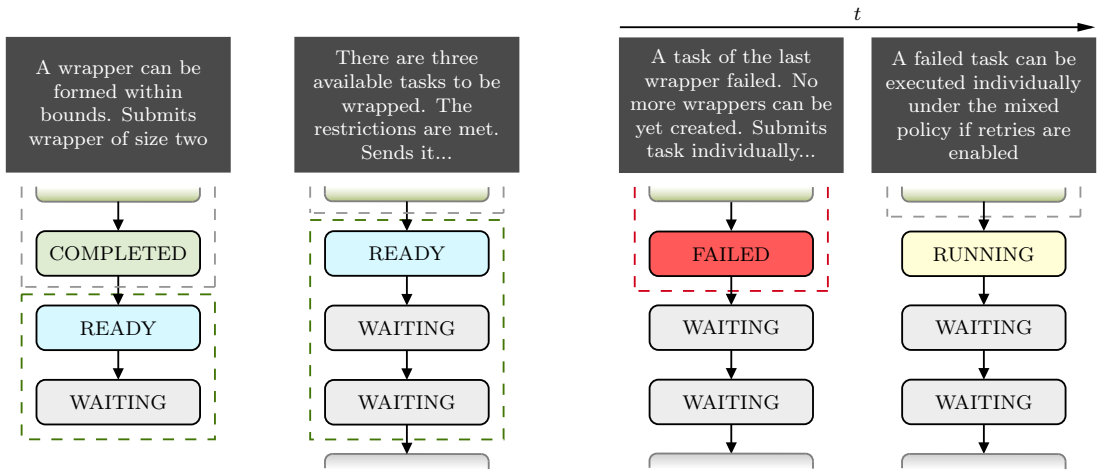
The *wrappers* are the mechanism that Autosubmit provides for aggregating multiple workflow tasks in a single submission to the remote platform. Wrappers can be composed of multiple chunks or splits, in the aggregation form that best suits the workflow structure. It also allows the user to choose between different aggregation and submission policies.

Aggregation policies

The wrapping policies in Autosubmit are *flexible*, *strict*, and *mixed*. Each of these policies offers different behavior when Autosubmit calculates the bounds in runtime—that is, the `min` and `max` size—of the wrapper to be sent in runtime. These limits can be set both vertically—in terms of walltime—and horizontally—in terms of the amount of parallel resources.

- **Flexible:** This policy always tries to build a wrapper that remains within the defined limits. If it cannot produce a wrapper that meets the required minimum size due to the shortage of ready jobs, it instead dispatches the tasks individually.
- **Strict:** This policy always creates wrappers strictly within the defined limits. However, if a condition arises in which the number of tasks ready to execute does not meet the user-defined constraints, a deadlock may occur.
- **Mixed:** This policy behaves like "strict", but if the bounds are not satisfied and a job from the previous wrapper has failed, it resubmits that job individually. As with "strict" mode, deadlocks may still occur.

Figure 2.4 illustrates some different situations that can be found while vertically wrapping tasks and how the respective policy handles it.



(a) Flexible policy. Any wrapper between two and three tasks could be formed. In case there are not a minimum of tasks, they would be sent individually.

(b) Strict policy. Only wrappers with a minimum of two tasks can be sent. In the figure, a wrapper of three tasks can be sent, so it does not need to wait.

(c) Mixed policy. Only wrappers with a minimum of two tasks can be sent, but in case of failure, tasks will be sent individually. In the figure, in a first iteration, a task on a previous wrapper has failed. If retries for that task are enabled, it will be submitted individually later on.

Figure 2.4: Examples of how the wrapping policy influences how the tasks are submitted within vertical wrappers in different scenarios. The vertical bounds are between 2 (`min`) and 3 (`max`) tasks.

Approaches for wrapper execution

Autosubmit provides two different wrapping methods, and the choice of one method or the other is up to the workflow developer based on their needs.

- **ASTHREAD:** This is the default method and is available for any type of aggregation. It launches tasks sequentially or concurrently in the platform's resource allocation by using *machinefiles* associated with each single job to control which compute nodes each one is assigned to.
- **SRUN:** Utilizes Slurm's *srun* command to form wrappers with shared-memory paradigm instead of relying on the *machinefiles*. It is available for *horizontal* and *vertical-horizontal* aggregations. In the last case, it uses CPU mask arrays for precise CPU affinity binding per job. *srun* is the command provided by Slurm to launch and distribute parallel processes across resources within allocations.

2.4 Workload management in HPC

To guarantee that every user of a HPC system can use its resources, it is essential to implement mechanisms that manage how those resources are accessed and distributed. These mechanisms go by the name of *workload managers*, although there are several accepted ways to refer to them, such as *job schedulers* or *resource managers* (Reuther et al., 2018).

2.4.1 An overview on resource management

Some of the best-known workload managers for HPC are Slurm (Jette & Wickberg, 2023), PBS Pro (Nitzberg et al., 2004), TCS (Fujitsu, 2021), and Flux (Ahn et al., 2020), which are used worldwide in well-known supercomputers such as MareNostrum 5, Aurora, Fugaku, or El Capitan, respectively. Currently, Slurm is the predominant workload manager within the European supercomputers ranked in the TOP500 list (Suarez et al., 2025).

One of the responsibilities of a workload manager is to ensure fair access to resources, which represents a complex challenge itself. There is also ongoing research on algorithms aimed at optimizing the energy efficiency of data centers. Representative works in this area include those by Juarez et al. (2018), D'Amico and Gonzalez (2021), and Stafford et al. (2025).

2.4.2 The Slurm Workload Manager

This study will use platforms that utilize Slurm as a workload manager. Slurm is an open-source, widely used job scheduler that stands out for its stability, modularity, scalability, and fault tolerance. Its default behavior can be modified to meet specific needs through plugins.

Slurm schedules the jobs that users send to the platform by assigning them an integer priority value. To do so, it offers the *basic* and *multifactor* priority calculation algorithms. In addition, it also integrates a backfill plugin to allow smaller jobs to run whenever resources are available for them and their execution does not delay any other higher priority jobs, with the aim of increasing the machine throughput.

The basic scheduling approach

The basic algorithm is based on a queue of jobs for scheduling: the priority of each job is assigned in chronological order of arrival. That is, an implementation of First-In

First-Out (FIFO). This algorithm could be particularly beneficial in High Throughput Computing (HTC), where the objective is to be able to execute as many jobs as possible in the shortest time. However, it would not be recommended in other use cases because, due to the nature of the algorithm, head-of-line blocking situations could occur when a very large job that cannot yet be executed delays the execution of lower priority tasks. To remediate this, the backfill plugin could be optionally enabled.

Multifactor scheduling

The multifactor algorithm is the default. In this policy, different factors are involved in the calculation of job priorities, and system administrators assign weights to each of them to determine their impact on the priority of the job. These parameters are *age*, *size*, *QoS*, *partition*, *niceness*, *association*, and *fair share*. In addition, a site-managed factor can be configured with a custom algorithm.

The *age* factor quantifies the time the job has been in queue, and the *size* of the job denotes the amount of nodes or CPUs it requires for execution. The *QoS* is related to the defined level of service quality, which involves some limiting parameters such as the maximum number of jobs that a user can submit at once, or the job's maximum walltime, in exchange of priority. Conversely, the *niceness* is a custom priority value a user may wish to assign to their jobs to indicate the scheduler which job should be executed first. It can only reduce the priority of the job.

The *fair share* which represents a quantitative measure of the user's entitlement to access and utilize the machine, defined as a function of its current and expected utilization level. This factor, by default, is computed by the *Fair Tree* algorithm implements fairness by balancing the responsiveness of the machine among users within accounts—groups of users—while maximizing machine utilization.

2.5 State of the art

Due to the fact that supercomputers are large shared-use machines, there is a need to establish mechanisms that distribute resources among users. Owing to their shared nature, congestion often occurs, having various implications, including system administrators being forced to configure the workload manager to limit, for example, the number of tasks that users can send at the same time, or the volume of resources each task is permitted to consume, with the aim of distributing them properly.

One of the objectives of task aggregation is to adapt the workflow to these restrictions set by the administrators, maximizing the use of the resources permitted for each request. This is not a new approach. There are different implementations of this mechanism or similar ones that are present in various workflow managers in areas such as life sciences, with Snakemake's grouping (Mölder et al., 2021), materials sciences, with the HyperQueue plugin (Beránek et al., 2024) provided by AiiDa (Huber et al., 2020), and other workflow managers such as Pegasus (Deelman et al., 2019), which implements it under the name of *job clustering*. In the case of Autosubmit, the implementation is known as *wrapping*.

Alternatively, a more complex form of aggregation is also possible by using Pilot-Job systems (Turilli et al., 2018), which reserve larger amounts of resources on the platform so that they can be used later by multiple workflow tasks, something that some authors refer to as *binding* (Merzky et al., 2019).

Another implication of using a congested machine is the long wait times in which user

tasks must wait in the scheduler's queues. With the aim of minimizing the impact of queue times on the total execution time of the workflow, the Autosubmit developers implemented a task aggregation approach called "wrappers". However, the first work to validate the effectiveness of this technique, to the best of our knowledge, was the work by Marciani et al. (2025). They simulated the HPC environment to test if vertical task aggregation reduced queue times. The analysis was carried out using a Slurm workload simulator and found that task aggregation is particularly beneficial when applied to workflows with vertical structures, i.e., where each task depends on the previous one.

With the objective of performing a quantitative assessment on the real impact of task aggregation in production workflows, Goitia et al. (2024) executed workflows running the EC-Earth3 model, very demanding in terms of computing resource consumption and executed them with and without wrappers on multiple renowned supercomputers to retrieve queue times and relate the results to the fair share of the user at any given instant. The experimental findings supported those simulated by Marciani et al. (2025): vertical aggregation saved up to approximately 12 times the queue time, increasing the amount of Actual Simulated Years per Day by up to 23% in the case of the platform with the highest congestion.

Chapter 3

In situ workflow managers

In this chapter, it is explored an automated method for executing the wrappers within a single allocation on High Performance Computing platforms. This is done by evaluating various candidate *in situ* managers for task aggregation based on a set of pre-requisites. One of these tools is intended to be integrated into the Autosubmit workflow manager.

3.1 Requisites and main concerns

An *in situ* workflow manager, in this context, is an HPC tool that enables to manage the execution of a subset of tasks within the workflow inside a resource allocation in a cluster. The evaluation and selection process of these *in situ* managers considered the following requirements.

- Should be easy to deploy in the HPC machines. This entails supporting a variety of architectures (x86, RISC-V, ARM), working on a variety of schedulers (Slurm, PBS, Flux, etc.) and and regardless of the underlying technologies configured, as the Message Passing Interface (MPI) implementation (Open MPI, Intel MPI).
- Must be capable of running any kind of task application (MPI, Bash, Python, containerized, etc.).
- Should provide a simple way to define a workflow.
- The in-situ scheduler must be able to deal with dependencies between tasks. The dependency definition should be explicit and straightforward, following a task flow basis, rather than a data flow where tasks are executed as their inputs become available.
- Must be capable of performing precise resource management for each task, also on multi-node configurations.
- It should be fault-tolerant. Ideally, an error on a task would imply the abortion of the execution of the workflow when there are no more runnable tasks, relinquishing the allocation to not misuse resources.
- Must be currently maintained and should be recognized within the workflow or HPC communities.

3.2 Methodology for assessment

To test the operation of the candidate *in situ* workflow managers, multiple dummy workflows have been run both locally and remotely. For local tests, each WMS was deployed in a Docker container based on an Ubuntu 24.04 minimal image (Goitia, 2025a), running on a machine with an `Intel Core Ultra 5 135U` CPU. Remote tests were carried out on MareNostrum 5.

The dummy workflow was based in all cases on the structure of Fig. 3.1. The `INI` task takes responsibility for compiling the application that `TASK` will run. Compiling the workflow on site allows for supporting different hardware without the need to build specific binaries for local and remote executions. Once compiled, it starts running the first group of concurrent `TASK`, which consists of a MPI application that sleeps for five seconds. And so on, until the execution reaches the `CLEAN` task that removes the binaries from the working directory. Furthermore, `TASK_12` served as a failure point to evaluate error handling.

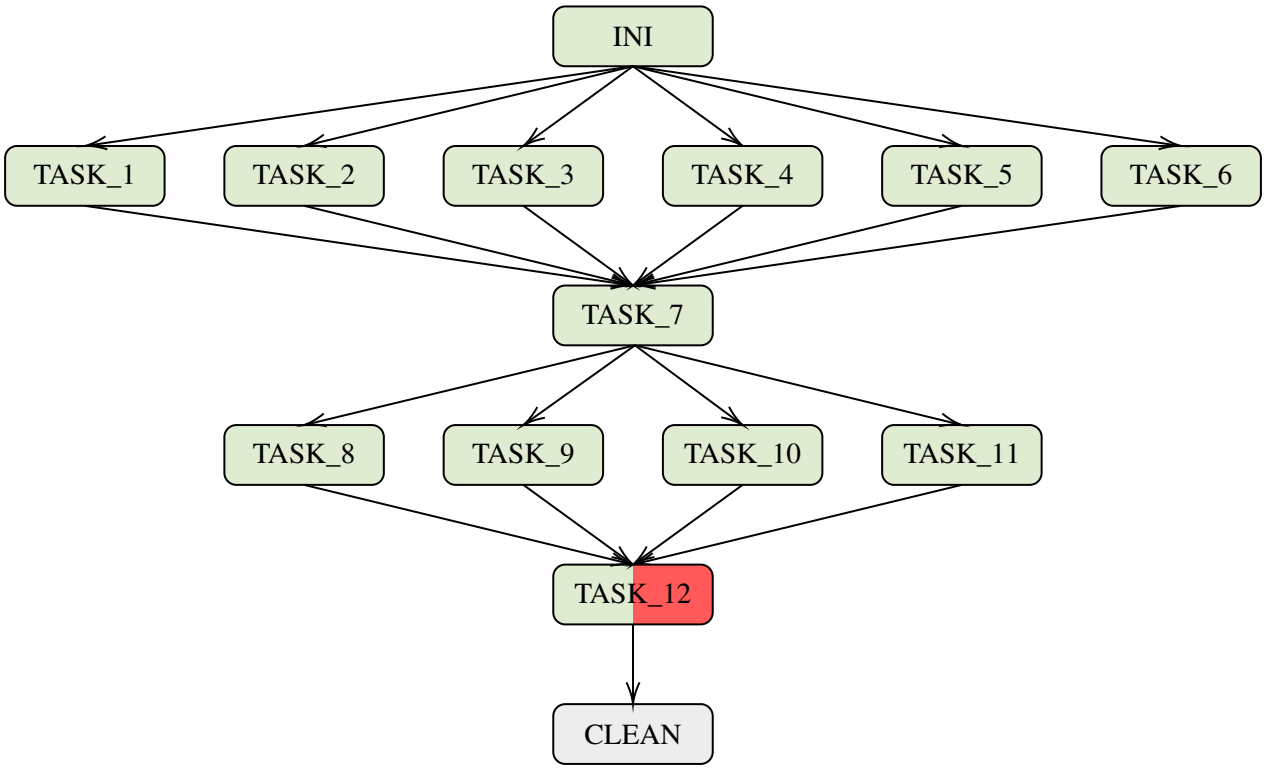


Figure 3.1: The dummy workflow used for testing the Workflow Management Systems (WMSs). Green tasks are expected to always complete. Red tasks are expected to fail. `TASK_12` may succeed of fail depending on the specific stage of the experiment. `CLEAN` is only expected to execute if `TASK_12` completes.

3.3 Evaluation

This section describes the evaluation and selection process of a series of *in situ* workflow managers specifically selected to act as wrapper engines. For each tool, the definition of the workflow, resource management, and failure tolerance is covered, explaining the potential advantages and the drawbacks found. It is important to note that the information is often based on the authors' experience using these software and is limited to functionalities that could be helpful for this purpose.

3.3.1 Toil

Toil (Vivian et al., 2017) is a Python-based workflow engine developed at the Genomics Institute of the UC Santa Cruz that is typically used in genomics applications, although it is multipurpose. It has a straightforward leader-worker architecture, common in workflow management. This type of architecture consists of a central coordinating component—the leader—responsible for scheduling tasks and tracking their progress, and one or more workers assigned to resources that actually execute the tasks and report the result or status of the task executions.

Toil supports the Common Workflow Language (CWL) (Crusoe et al., 2022) and Workflow Description Language (WDL) (Voss et al., 2017) standards, although one can also use their Python API to create and execute workflows. They provide multiple examples for different HPC platforms, including those with Slurm.

After exploring and testing some examples in the documentation, a testing workflow was developed using Toil's Python API, which was expected to be more convenient when facing integration. Toil allows to establish both explicit dependencies between tasks (task-driven) and implicitly defined dependencies by handling input and output data through a method they named "promises" (data-driven). The API provides a command line argument parser so that external applications can be executed.

Regarding resource management, tasks were executed as expected, only when there were enough resources for them. However, although Toil clearly supports multi-node requests, no method has been found in the Python API to specify that a task should use one or more entire nodes. Perhaps this is not possible when workflows are defined in this way. Moreover, core overlapping situations have been observed where some MPI ranks of different tasks shared the same core.

In the fault-tolerance side, Toil has not automatically detected the error code in `TASK_12` in order to interrupt the execution of subsequent dependent tasks. Mechanisms for manually performing this were also not identified.

3.3.2 ExecutorLib

ExecutorLib (Janssen et al., 2025) is a Python library that enables sending Python functions as jobs to remote HPC platforms. It has multiple operating modes that are differentiated by *executors*: `SingleNodeExecutor`, `HPCClusterExecutor`, and `HPCJobExecutor`. Perhaps one of the most notable features of this WMS is that it is capable of communicating with both Slurm and Flux.

Defining workflow tasks is straightforward. However, dependencies are implicit, based on "futures" (data flow). To alleviate this, it would be possible to set explicit synchronization points. It would also be possible to use empty variables to define the data flow. Moreover, as ExecutorLib is designed to facilitate parallelism in Python, it allows only the submission of Python functions as jobs, with the exception of the `FluxExecutor`.

The way ExecutorLib manages task resources depends on the chosen executor. The `SingleNodeExecutor` is conceived to prototype the application locally before executing it on an HPC cluster. It works following a manager-worker architecture in which the number of workers specified in the application is launched. Core overlapping situations have been detected while testing this executor locally.

The `HPCClusterExecutor` and the `HPCJobExecutor` handle the submission of tasks to remote platforms, such as Slurm or Flux, varying the way the tasks are sent: in-

dividually through a batch command or altogether to the same resource allocation, respectively. Attention was directed toward ExecutorLib's `FluxJobExecutor` and the Flux Framework's `FluxExecutor` configuration, which allows for the convenient execution of external executables—essential for the workflows that are executed in this work—alongside Python functions, while letting Flux handle resources.

It has been detected that some scheduling parameters for Flux jobs, such as the *wall-time*, which represents the maximum execution time per task, cannot be set with any executor other than `FluxExecutor`.

Regarding fault-tolerance, it is possible to handle eventual errors manually. This would allow controlling the execution flow and aborting it whenever an error occurs if there are no more runnable tasks.

3.3.3 Greasy

Greasy (Abellan et al., 2021) is a tool developed in the Barcelona Supercomputing Center that facilitates the execution of multiple parallel tasks, covering the scheduling and running stages under the available resources.

Greasy allows for configuring some parameters such as the strict check of the syntax errors of the workflow definition file, the count of maximum retries, the path to the logs, the logging level, the number of workers, a node list where Greasy will run the tasks, and the engine. Parameters can be set for the entire installation through the configuration files or for specific executions by setting environment variables.

The workflow definition consists of assembling a `txt` file with the specification of the tasks and the explicit dependencies between executables.

In Greasy's architecture, the component that allows to schedule and run jobs is called engine, and can be *basic*, *mpi*, or *thread*. The first one is the standard implementation, which allows running locally or remotely if SSH or Slurm is present in the system. This is the only engine that checks the nodelist, which is the list of nodes where Greasy will run tasks. If the nodelist is empty, the execution will be done strictly in the local node. The *mpi* approach launches Greasy as an MPI application, which requires one core for scheduling at runtime. The *thread* engine is the thread implementation for shared memory computers and works only in a single node.

The tests on this tool were performed with the default configuration found in MareNostrum 5, which consisted of a *mpi* engine, without retries. Multi-node experiments on this tool have not been performed because of the absence of nodelists in this engine.

In Greasy, the number of parallel jobs depends on the count of workers, but no information was found regarding the underlying resource management and how resources are distributed between them.

On the error management side, `TASK_12`, which was expected to fail, returned error code `1`, and Greasy stopped the execution of that task and *CLEAN*, which depended on it. Greasy automatically elaborated a new task definition file with all the tasks that failed so that they can be executed independently without having to re-execute the entire workflow, just setting it as the input file. It also counts with a job retry functionality.

3.3.4 RADICAL-Pilot

RADICAL-Pilot (Merzky et al., 2022) is a Python-based *pilot-job system* specialized in executing tasks in heterogeneous HPC environments. What pilot systems do is to launch a job to the HPC to allocate resources and schedule workflow tasks directly on them. RADICAL-Pilot supports a wide variety of HPC schedulers, such as Slurm or PBS. It also supports multiple commands for task execution; including `ssh` or `mpirun`.

RADICAL-Pilot provides a Python API. A RADICAL-Pilot application execution consists of multiple steps, as detailed below. It has been observed that all the steps before task execution can take a variable amount of time to complete.

1. Opens a RADICAL-Pilot session.
2. Creates a pilot manager that will manage the requested resources.
3. Creates pilots indicating the resources to be requested for each of them.
4. Launches the pilots and waits for them to become active.
5. Creates the task manager and assigns the created pilots to it.
6. Defines the tasks.
7. Runs the tasks and waits for them to finish.

Since RADICAL-Pilot is not a workflow manager but a runtime system, there is no method to explicitly specify dependencies, neither based on a task flow nor a data flow. However, RADICAL-Pilot could be used standalone to manage workflows if explicit checkpoints are used to define the task flow. In addition, it allows the execution of external applications if tasks are launched in `rp.TASK_EXECUTABLE` mode.

During some tests in which a workflow was launched with MPI applications, an error `mpirun does not support recursive calls` occurred. Deleting the `mpirun` instruction from the executable command resolved the issue.

On the resource management side, both pilots and jobs can be assigned a set of resources and scheduling parameters. These parameters have been detected to be an adjusted set of those that a workload manager batch command would accept. In a local execution, all tasks were correctly executed as resources became available. For example, on a machine with a total of 12 cores and a horizontal wrapper of six tasks requiring six cores each, the tasks were executed in pairs. Moreover, four horizontally arranged tasks, each requiring four cores, were executed in parallel.

Regarding error management, RADICAL-Pilot was able to detect a failed job. However, since it is not a workflow manager *per se*, but rather a pilot job system, the workflow continued to run despite errors. However, errors can be handled manually.

3.3.5 Pegasus-MPI-Cluster

Pegasus (Deelman et al., 2015, 2019), developed at the Information Sciences Institute of the University of Southern California, is a tool designed to facilitate the execution of workflows without having to think about how the middleware—Slurm, in this case—operates. So far, it has been used to run applications in a wide variety of fields, such as astronomy or bioinformatics.

The Pegasus-MPI-Cluster is a component that belongs to this software, but that can be used separately. It allows running workflows defined as Directed Acyclic Graphs (DAGs) on clusters using MPI.

The abstract definition of the workflow is made in `DAX` format (DAG in XML). The definition of tasks is straightforward. They are assigned IDs so that explicit dependencies can be established later on. The keyword `TASK` defines a new task and `EDGE` defines a one-to-one dependency. Moreover, tasks can be configured with extra options, such as the count of retries.

Regarding resource management, according to the architecture, if a set of cores are allocated for the Pegasus-MPI-Cluster instance, it will use one as the master process and the remaining ones will act as workers. Experiments involving a number of concurrent tasks that exceeded the number of workers succeeded. The testing workflow has been successfully executed within an instance with four MPI processes, one of them being the master (rank `0`), and the other three being the workers (ranks `1-3`).

Running a workflow inside a MareNostrum 5 node (with 112 cores) revealed a problem in resource management. When assigning 60 cores to `TASK_1-6`, all tasks executed concurrently despite resources sufficing for only one, leading the system to time-slice ranks on the same core and delaying the five-second executions to more than 30 seconds. This issue is significant and uncertainty persists regarding whether Pegasus-MPI-Cluster supports the execution of MPI tasks, given its own use of MPI. This involves a non-working nested MPI.

Another concern, now resolved, arose during the first execution of the testing workflow. It failed due to an uncovered case in the Pegasus-MPI-Cluster source code that enforced thread counts proportional to physical cores—a rule not always met in modern processors with hybrid architectures combining performance and low-power cores, for example. The issue was reported through its GitHub repository (Goitia, 2025b), and the Pegasus developers fixed it, being changes now available.

Regarding fault tolerance aspects, Pegasus-MPI-Cluster includes a retry mechanism. To verify it, a retry count of three was set for `TASK_12`. As expected, this task was executed three times before the workflow failed, along with the launched MPI processes. Upon resubmission, the workflow resumes from the failed tasks.

3.3.6 Parsl

Parsl (Babuji et al., 2019) is a library that aims to extend parallelism in Python. One outstanding features is its ability to run Python functions across platforms. Its leader-worker architecture features *executors* that control the tasks to be run based on the data flow. Tasks are executed on workers that are instantiated on the platform by the use of *launchers*. The *providers* behave as bridges between Parsl and the resources that will be utilized. A representation of this leader-worker architecture, applied to the Parsl case, is depicted in Fig. 3.2.

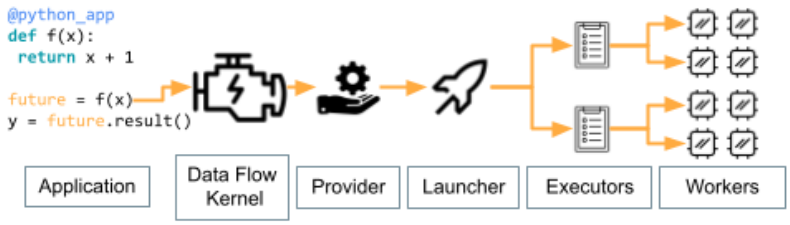


Figure 3.2: High level schema of the Parsl architecture. Image from the "Parsl 1.3.0-dev documentation – Quickstart" (2026).

It supports the execution of different types of task thanks to its decorators: `Python`, as functions; `bash`; and `MPI`, if the `MPIExecutor` is configured.

As with some of the WMSs explored, the task definition is straightforward, but the dependencies between tasks are implicitly set based on the data flow, so explicit barriers must be placed. Tasks, by default, are launched concurrently unless blocking points are entered. These points can be set with the `result()` function.

Among all possible configurations, the combination of `HighThroughputExecutor` with `LocalProvider` and `SimpleLauncher` best suited the studied case. The `HighThroughput-Executor` launches multiple identical Python workers across compute nodes. The `Local-Provider` starts workers on the computer or the allocation that executes the Parsl script. The `SlurmProvider` could also be used, but as it batches scripts directly to Slurm, it requires execution outside the allocation.

So far, the results of the execution of a workflow in MareNostrum 5 have been positive, just by adjusting the number of workers and the resources associated with each. Regarding oversubscription—defined as a situation where tasks exceed available resources—the maximum number of concurrent tasks depends on the number of workers and their associated resources, with no overlaps. To test multi-node executions, the requirements of some workflow tasks were adjusted to request one full node or more, matching expectations.

Error handling in Parsl, according to "Parsl 1.3.0-dev documentation – Error handling" (2026), can be performed from several fronts:

1. **Exceptions:** manual exception catching and handling, as it propagates them from the applications.
2. **Retries:** consists of setting a fixed number of retries.
3. **Retry handlers:** an advanced tool that allows to set a cost function. That is, instead of decrementing the retry value in one unit, it can be decremented in a custom value depending, for example, on the exception or the exit code raised by the application.

Furthermore, the "lazy fail" feature allows for a workflow to continue running even if a task has failed, until it encounters a dependency that cannot be fulfilled.

3.3.7 TaskVine

TaskVine (Sly-Delgado et al., 2023) is a workflow manager developed at the University of Notre Dame, designed for workflows with intense consumption of data, which is committed to keeping track of the "the lifetime of data in a workflow—from archival sources to final outputs—making use of local storage to distribute, and re-use data wherever possible".

The architecture follows a leader-worker basis, broadly based on a machine that runs a manager and a cluster running dynamic workers that adjust to available resources, to which, for example, pending jobs, datasets, and libraries are assigned. The workers are responsible for managing the resources within the nodes. In addition, TaskVine always tries to favor the proximity of data to the tasks that require it.

Executing a workflow requires instantiating both the manager and the desired workers on the corresponding resources. The manager and the workers communicate over the network. To do this, it is necessary to specify the IP and the port at all endpoints. Defining tasks is simple, but execution is data flow-dependent, so the only way to execute a workflow with explicit dependencies is to introduce synchronization points. It allows for the definition of tasks as external applications, Python functions, and serverless function invocations. The latter can be particularly useful for saving the cost of multiple initializations of functions that are repeated between workers.

A local experiment to test resource management was carried out by adjusting tasks `TASK_1-6` to request six cores-each out of 12, `TASK_7` requested one, and tasks `TASK_8-12` requested three-each. TaskVine automatically detected the physical cores of the CPU without needing to describe the resources.

The workers behaved as expected, running `TASK_1-6` in pairs, `TASK_7` alone, and `TASK_8-12` concurrently. However, some cores have been detected to be shared between concurrent workers.

Regarding fault tolerance, the retry mechanism in TaskVine intends to prevent a task from finishing in cases of, for example, missing inputs or a worker disconnection, but does not react to non-zero error codes. However, it tracks them, and errors can be manually handled to perform some action after a task failure.

Additionally, it has been detected that no native way to automatically stop the manager when all workers have failed is available, causing the TaskVine manager to remain on hold until a worker becomes available again or the execution time limit is reached.

3.3.8 Flux

Flux (Ahn et al., 2020) is a next-generation HPC scheduler developed by Lawrence Livermore National Laboratory (LLNL) since 2014. Its design is guided by the need to handle leadership-class heterogeneous HPC machines.

The solution that developers of Flux pose to the bottleneck issue that these large machines have is to be hierarchical and recursive. This means that Flux is meant to be launched under an ever smaller resource set. It is capable of identifying resources within an allocation and managing them much like it would be if it were installed natively by the system administrators.

Additionally, due to its strengths, Flux is used by ExecutorLib (see Section 3.3.2) and RADICAL-Pilot (see Section 3.3.4) to handle task execution.

Flux, like Slurm, does not directly claim to support workflows. Both allow for a certain automation of the jobs depending on the outcome of a previously submitted job (i.e., job dependencies). Therefore, although neither scheduler calls workflows, they can be defined by configuring constraints.

In addition to the CLI interface, Flux provides an additional way of interacting through a Python API. Both allow for automating the submission of tasks with dependencies. To achieve this with the CLI, a bash script was created that submits all workflow tasks. Flux

developers considered this case for programming workflows because the `ID` assigned to the task upon submission is the output of the command. Therefore, defining constraints is a matter of saving this output to a variable and then performing the variable substitution.

An example of the above description is given in Code 3.1. A chained workflow with two tasks is defined, where the second depends on the first.

```
previous_task=$(flux batch task_1.sh)
flux batch task_2.sh --dependency=afterany:${previous_task}
```

Code 3.1: A simple way to define a chained workflow though CLI in Flux. `task_2` depends on `task_1`.

Because Flux is a full-fledged scheduler, resource management is an absolute necessity. It tracks resource usage down to the granularity of the definition of the machine: from nodes to CPUs, all the way to Non-Uniform Memory Access (NUMA) cores and particular accelerators.

Defining resources for a job is as straightforward as in Slurm. They can be defined in both the submission command and in the header of the batch script. However, current Flux job submission commands do not allow some resource directives, such as equivalents to Slurm's `ntasks-per-node`, `hetjob`, or `mem`. In Flux, jobs are defined either directly or indirectly by jobspec structures in YAML format, which contains structured information for resources, tasks, attributes, and the version of the schema being followed. These jobspecs follow Flux's RFC 25 specification for "Job Specification Version 1", "which represents a request to run exactly one program" (Herbein, 2014), which is a simplified version of the "Canonical Job Specification" described in RFC 14 (Scogland, 2016). In RFC 25, information is structured as follows:

- **Resources:** can be *node*, *core*, *gpu*, and *slot*. A slot is a reserved resource type that represents the resources available for an individual task. For example, there may be a slot with multiple cores (equivalent to Slurm's `cpus-per-task`), a node with multiple slots (equivalent to `ntasks-per-node`), among many other combinations for resource requests. Only one resource may be defined per jobspec.
- **Tasks:** defines a single task to be executed, specifying the command, the number of tasks, and the slots to be used.
- **Attributes:** define other aspects of the job, such as the maximum execution time, the working directory, or the complete script to be executed, among others.
- **Version:** must always be *1*.

On the fault tolerance side, it is possible to keep track of the running jobs in order to handle errors manually.

3.4 Highlights and conclusions

In this chapter, a series of tools initially considered as candidate wrapper engines were explored. These tools include Workflow Management Systems (mainly distributed), pilot systems, tools or libraries for task parallelism, and workload managers such as Flux. A summary table can be found in Appendix A.

In the case of Toil, it offers the most flexible workflow definition, supporting both implicit and explicit dependencies. However, some drawbacks have been identified regarding resource management and fault tolerance. ExecutorLib, on the other hand, performed correctly in resource management, thanks to its integration with Flux, and in error handling, but the task flow depends on the availability of input data, and some cases of core overlapping were detected when using non-Flux-based executors. Parsl, on the other hand, appears to perform resource management well and facilitates error handling, but dependencies are also data-flow-based.

Greasy, by contrast, facilitates the definition of workflows with explicit dependencies via text files and features a notable error-handling and retry mechanism. However, the underlying resource management is not entirely clear. Pegasus-MPI-Cluster also allows for the explicit definition of dependencies and provides a remarkable error-handling mechanism, but situations of core overlapping and nested MPI were detected. In the case of TaskVine, which also establishes implicit dependencies on the data flow, it supports manual error handling; however, as a drawback, cases of core overlapping were also detected during the execution of the test workflow.

In the case of RADICAL-Pilot, since it is not a workflow manager *per se*, it does not support the definition of workflows, although it does facilitate waiting for executed jobs to finish, which could help mitigate this limitation. It excels in resource and error management. However, it introduces some variable overhead due to initialization tasks before the first task is executed. Flux, on the other hand, is a workload manager for HPC, so resource management is a key component of this software given its nature. It supports establishing both implicit (data-driven) and explicit (task flow) dependencies between submitted jobs, and error management can be performed manually by monitoring the status of submitted jobs.

This analysis reveals that maintaining a compromise among all initially defined requirements proves to be complex. From the task aggregation perspective, each utility exhibits strengths and weaknesses that are often incompatible.

Considering all this information and given the necessity to choose an *in situ* manager, Flux emerges as flexible enough to meet all requirements.

Flux presents an opportunity to integrate state-of-the-art technology into Autosubmit. Clear delimitation of resources for aggregations within the same allocation is critical, and Flux achieves this thanks to its hierarchical architecture, which allows it to execute even within resource allocations of other workload managers. In addition, it allows for easy error management. Flux is already present in supercomputers such as the exascale *El Capitan* at LLNL, the same facility that has been developing it since 2014, positioning it as a stable and actively maintained option.

The other alternatives, while not definitively discarded as wrapper engines, offer somewhat less flexibility or introduce some degree of complexity that is preferable to avoid.

Chapter 4

Design

The final objective of this work is to properly integrate the tool that best satisfies the requirements for the *in situ* orchestration of the aggregated tasks, which has been identified as Flux, into the Autosubmit workflow manager. In doing so, it must be taken into account that Autosubmit is a mission-critical application for the scientific production, and therefore the correctness and highest quality of the implemented functionality must be guaranteed.

The integration design described in this chapter is based on the assumption that Flux is installed and available on the HPC platform, either bare-metal or in a virtual environment, such as Conda or Spack (Gamblin et al., 2015). The implementation, which used as a base the Autosubmit version v4.1.16 (Beltrán Mora et al., 2026), is limited to ensure full functionality with Slurm systems. In addition, unit tests were designed to test the newly developed code and ensure complete coverage.

4.1 Overview

The interaction between Flux, as the *in situ* wrapper manager, and Autosubmit, can be summarized as Flux receiving lists of tasks from Autosubmit that are processed by a remote script that automates the submission of jobs in a specific order depending on the type of aggregation. This wrapper script is responsible for submitting and monitoring the jobs it contains.

The desired behavior is similar to the wrapping methods already present in Autosubmit. What Autosubmit does, at runtime, is to create the wrappers according to the selected policy. For each of the wrappers, it generates a script that is sent to the platform for remote management of internal tasks. The way these tasks are managed depends on the selected method. The idea, therefore, would be to leverage the existing mechanism and introduce the required modifications in order to implement a new `FLUX` wrapping method in Autosubmit.

The creation and management of wrappers under the new `FLUX` method is subject to the following execution loop:

- Autosubmit periodically checks the status of running tasks and verifies whether there are any new tasks that can be executed once their dependencies have been resolved, section by section. If wrappers have been configured for that section, it will also verify whether there are any wrappers that can be sent based on the selected aggregation policy, calculating the bounds based on ready tasks availability.
- If the policy and the status of the workflow allow for creating a wrapper, it creates

both the scripts that execute each job within the wrapper and the wrapper script itself. The scripts of the tasks inside the wrapper will specify the resource reservation they will request from Flux, and the wrapper script will specify the required resource reservation from Slurm to orderly execute the wrapped jobs while preserving possible parallelism. All of these scripts are then sent to the platform for execution.

- In the remote allocation, the wrapper script submits the jobs to Flux and keeps monitoring the execution, checking whether it can submit more jobs as dependencies are solved. In the meantime, Autosubmit keeps tracking the execution of all the tasks, including those in the wrapper.

Regarding the format of the resource specification of the jobs inside a wrapper, it can be performed in two different manners. The first consists of appending a header with the resource specification to the job scripts in a format similar to that of Slurm. Instead of using the `#SBATCH` directives, the equivalent `# flux:` directives would be used. However, it has been found that this form does not yet support all possible directives, omitting necessary directives such as the tasks per node (`ntasks-per-node` in Slurm) or memory directives. Therefore, it was necessary to go with the second option, which is for Autosubmit to generate the jobspecs for the tasks based on the resources specified by the user. The wrapper script will then use Flux's Python API to launch and monitor the jobs in the appropriate order based on the selected policy.

The diagram in Fig. 4.1 provides a high-level conceptualization of the intended integration.

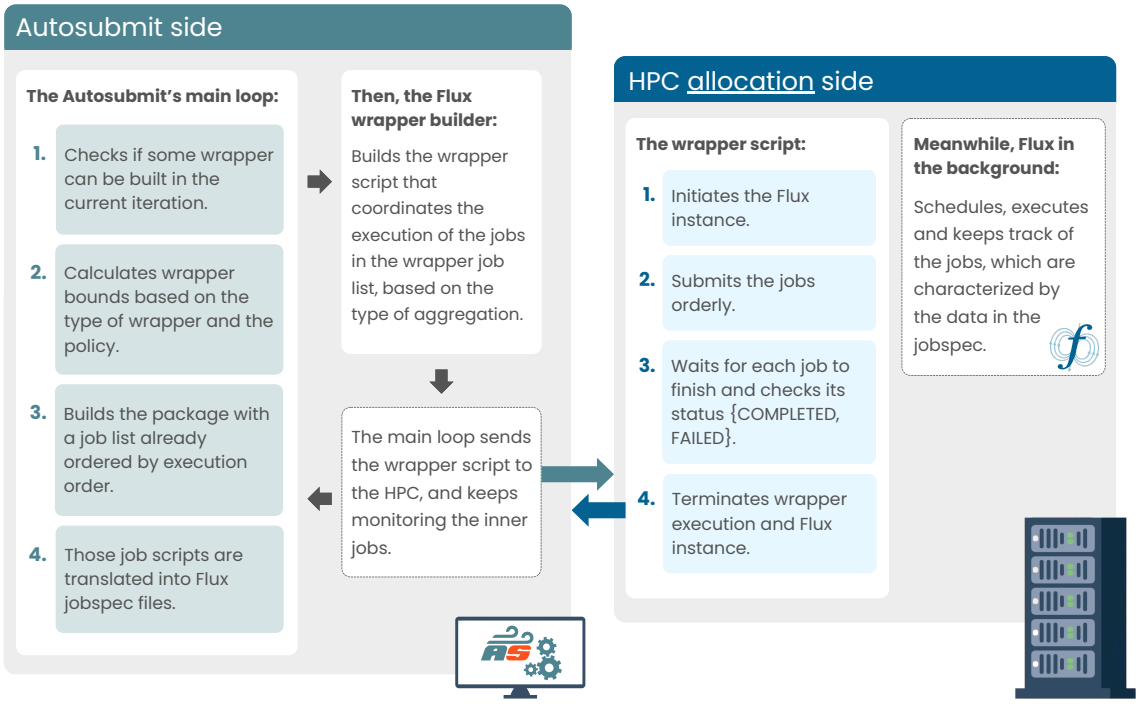


Figure 4.1: High level representation of the interaction between Autosubmit and Flux systems.

4.2 Understanding Autosubmit wrapper-related components

For a clear understanding of the new components that need to be created and where they should be placed, and those that require some modifications, it is essential to have an

overview of the existing components related to wrapper management in Autosubmit and their relationships.

The class diagram in Figs. 4.2 and 4.3 represents the relationships between the classes in the `autosubmit.platforms` package and its subpackages, `headers`, and `wrappers`, which have collected most of the changes. The diagram has been split for better readability, but an expanded version containing all the relationships between the sub-packages can be found in the Figure B.1 of Appendix B.

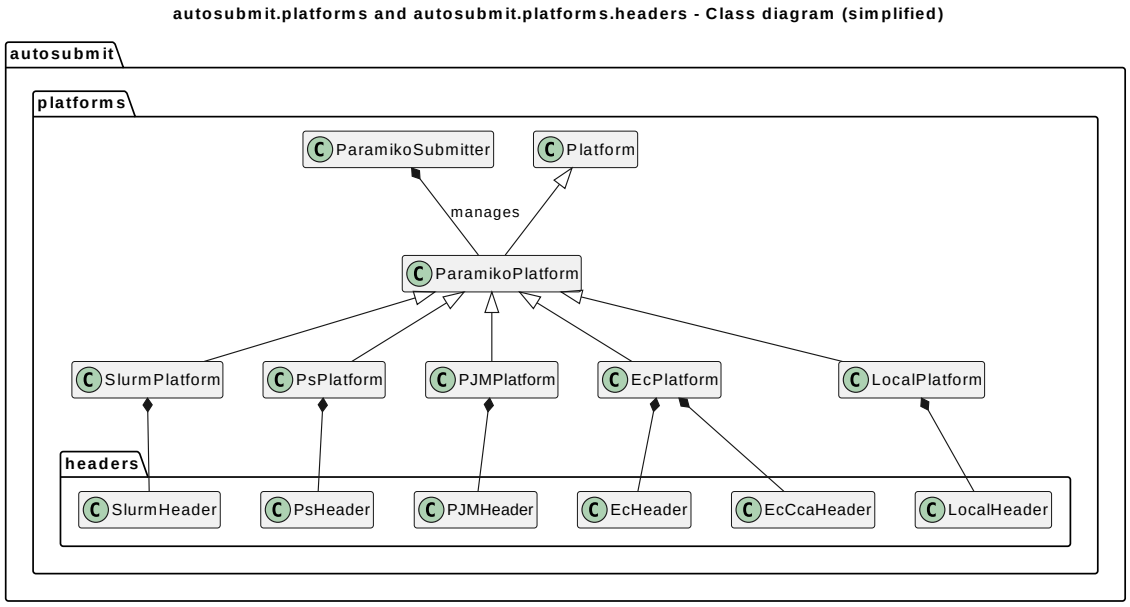


Figure 4.2: Class diagram representing packages `autosubmit.platforms` and its sub-package `headers`.

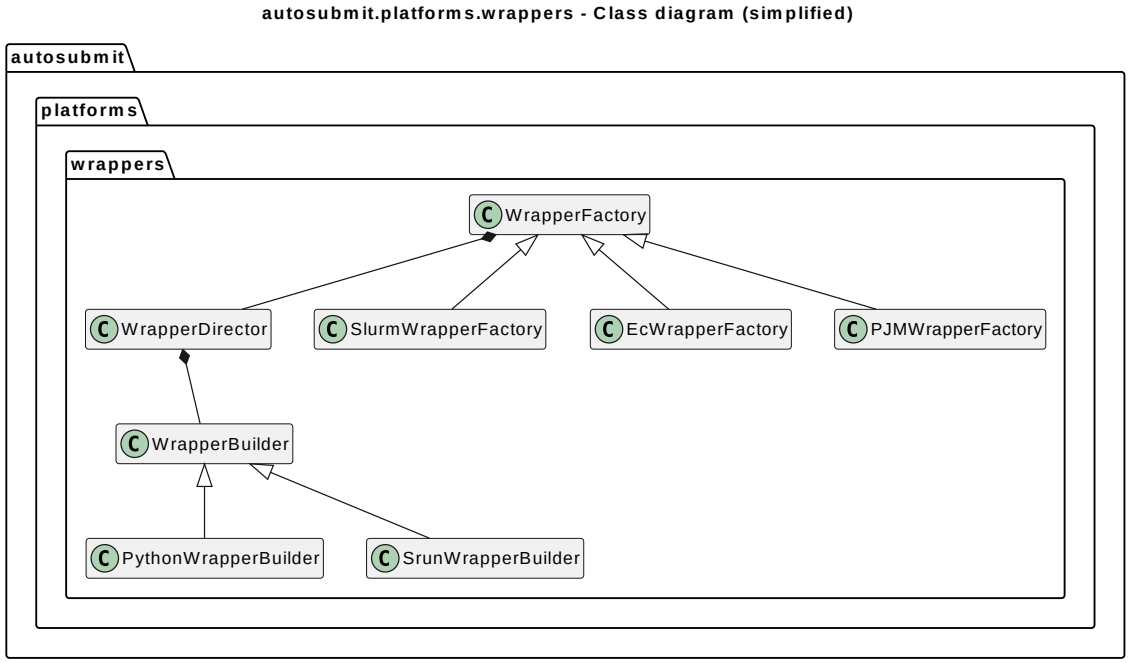


Figure 4.3: Class diagram representing packages `autosubmit.platforms.wrappers`.

As can be seen in Fig. 4.2, there are different classes of *platforms*. These classes are the software components that facilitate communication with HPC clusters, interaction with

the workload manager, and transformation of user configurations into executable functions. The platforms use the `Paramiko` library (Forcier, 2026), an implementation of the SSHv2 protocol for Python.

As a general rule, there is one platform class for each supported workload manager (see `PJMPlatform` or `SlurmPlatform`), although there are also platforms that cover cases such as executions on the local host (`LocalPlatform`) or also on HPC systems in situations where the use of the workload manager is not preferred (`PsPlatform`). Moreover, each `platform` is associated with a `Header` class, which provides all the necessary methods to build the header with the job resource specification based on the workflow configuration.

On the other hand, Fig. 4.3 comprises all the classes related to wrapper creation and management. There exists a `WrapperFactory` per `platform` that implements wrappers (not all supported schedulers have them enabled). It is responsible for generating the wrapper script which coordinates the execution of the tasks inside the wrapper. To do this, it uses the `platforms` themselves and the `WrapperDirector`. The corresponding `platform` provides the wrapper header with the computational resources that will be requested for running all contained tasks, and the `WrapperDirector` serves as an interface for obtaining the script content through the `WrapperBuilders`, which are classes responsible for generating the execution logic for the wrapper script given the wrapping method (`ASThread`, implemented in the `PythonWrapperBuilder`, and `srun`) and the wrapper type. This implies having a `WrapperBuilder` per wrapper type and wrapping method. The wrapper builders receive the lists of jobs that compose a wrapper, as well as other parameters such as the header or the working directory, among others.

Jobs inside the wrapper, as well as the individual jobs, are assigned to scripts composed of a header containing all the scheduler directives regarding resource specification, and the script content itself—which is referred to as the *template* script. In this case, the header is also provided by the `platform`, and the responsible for merging the header and the content of the script is the class `autosubmit.job.job`, which is not present in the diagram.

4.3 Designing the new components

The new interface between Autosubmit and Flux involves, mainly, the jobspecs with the resource specification, metadata, and the template for each single task inside the wrapper, and the wrapper scripts that coordinate their execution. Additionally, since the Flux environment might not be enabled by default in the supercomputer, it is required to implement a new key in the configuration to let users specify how to activate it inside the remote resource allocation.

4.3.1 The Flux jobspec generation

The first step in the integration is to adapt the generation of each of the job scripts that compose a wrapper. This implementation requires Flux jobspec files to be generated for each job in YAML format—which integrate both the resource specification, the template, and metadata—instead of the header-template pair. For that, it became necessary to create an empty `platform` and `header` classes in order to prevent headers with an invalid resource specification from being generated. They have received the names `FluxOverSlurmPlatform` and `FluxOverSlurmHeader`. The new *header* just returns a single-line string with information about the task. The diagram in Fig. 4.4 illustrates where these new components were placed inside the `autosubmit.platforms` package.

Generating the files containing the jobspecs is one of the most delicate parts of this

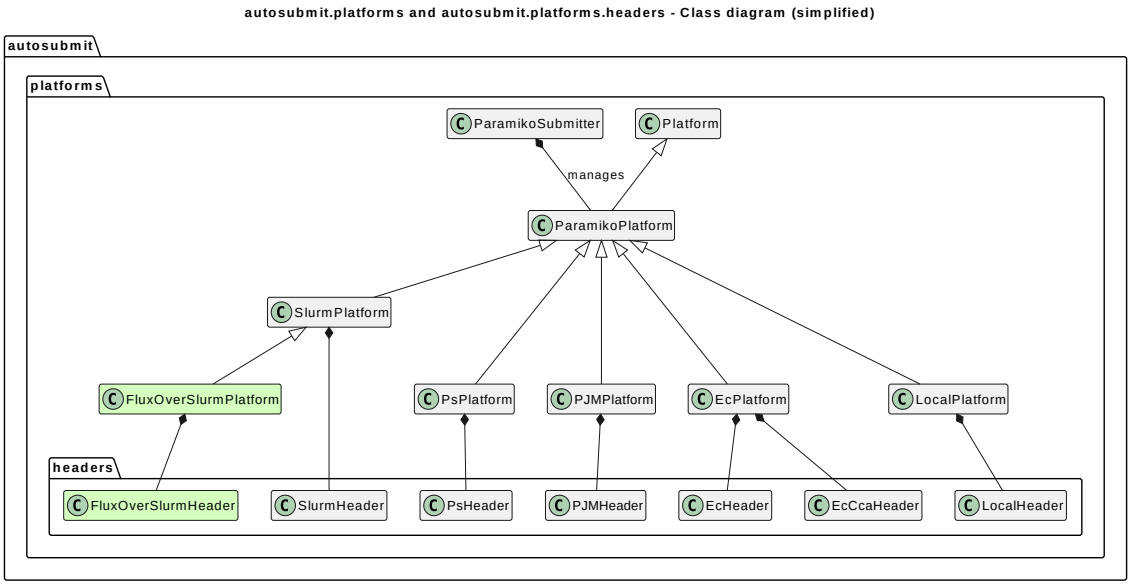


Figure 4.4: Class diagram representing the packages `autosubmit.platforms` and `autosubmit.platforms.headers`, with the new components needed to build the interface with Flux. New components appear highlighted.

work because it serves as the direct interface between Autosubmit and Flux, and a single misplaced attribute or an indentation issue can result in an invalid YAML file that Flux cannot process. For this purpose, it has been designed a new class `FluxYAMLGenerator` which coordinates the creation of a `FluxYAML` object with the resource specification, mapping the job configuration directly into jobspecs, following the Flux's RFC 25 specification. Although this generator is located within the `autosubmit.platforms.wrapper` package given its purpose, it is used externally by the class `autosubmit.job.job` when building the script from the *template* provided by the user.

The diagram in Fig. 4.5 represents a simplified class relation in the `autosubmit.platforms.wrappers` package that includes the new classes mentioned above.

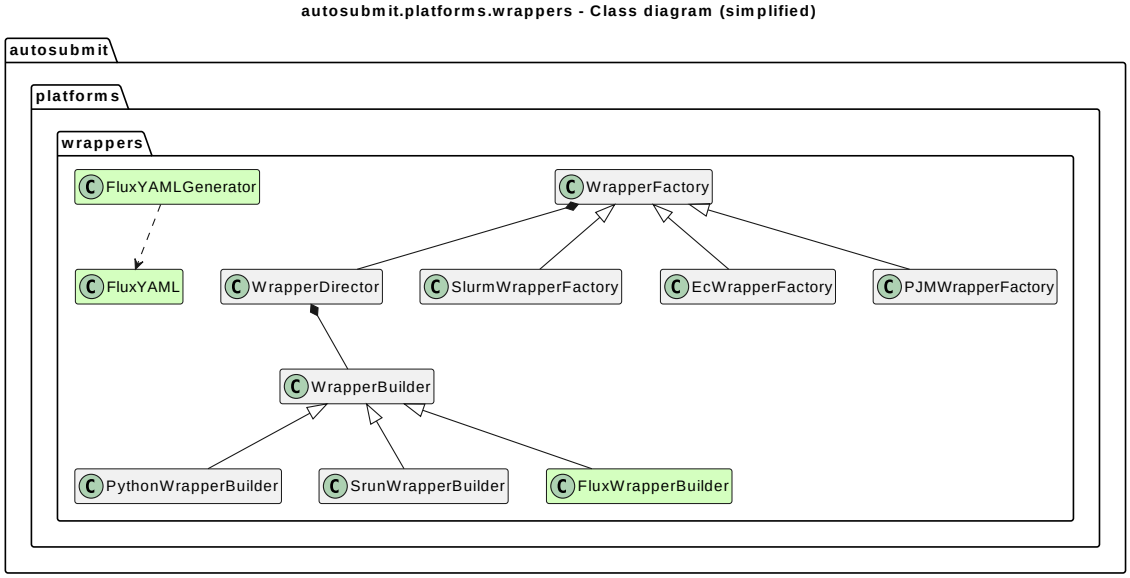


Figure 4.5: Class diagram representing packages `autosubmit.platforms.wrappers`, with the new components needed to build the interface with Flux. New components appear highlighted.

This generator, in addition to being useful for the purpose of this work, could facilitate the inclusion of Flux as another supported HPC scheduler in Autosubmit if required.

4.3.2 The wrapper scripts

The second step is to create the wrapper scripts. For this, it is necessary to introduce a new `WrapperBuilder` in the `autosubmit.platforms.wrappers` package, which has received the name `FluxWrapperBuilder`. It was inserted as the parent class of all the sub-builders that were also created for supporting the different types of wrappers available for the new `FLUX` method: vertical, horizontal, vertical-horizontal, and horizontal-vertical.

Figure 4.5, contains the new classes added in this regard to the `autosubmit.platforms.wrappers` package. For ease of understanding, the classes corresponding to the `WrapperBuilders` of each aggregation type have been excluded. A definitive class diagram for this implementation can be found in Figure B.2 of Appendix B, showing all the relationships between the sub-packages of `autosubmit.platforms` with the new added classes.

The behavior of the wrapper scripts can be summarized as the ordered sequential or parallel submission of the contained jobs to the *in situ* manager—using loops for sequential executions, threads for those in parallel, or a combination of both in the case of hybrid wrappers—of each of the jobs that comprise the wrapper. They consist of a header specifying the resources for the entire aggregation and a body containing the execution logic. To accommodate the new `FLUX` method, both parts have undergone substantial changes.

The header

The script header specifies the resources that Autosubmit requests from the platform for the wrapper. When the aggregation is sequential, the walltime of all tasks is added, and when it is parallel, the CPUs are added.

This logic, while correct, can prevent Flux from taking full control of the allocation if certain considerations are not taken into account. Slurm should only execute a single task to which it assigns all available resources, regardless of the original wrapper resource specification, which may define multiple tasks or assign resources to them, for example. A conditional block in the `WrapperFactory` ensures that a single instance of Flux can manage all the resources in the allocation by defaulting values in Slurm's `ntasks`, *ntasks-per-node*, and *cpus-per-task* directives.

The execution logic

The body of the wrapper scripts has several responsibilities, as detailed below.

- First, it generates a standalone Python script that, through the Flux API, coordinates the proper execution of all the tasks inside the wrapper. This Python script is the only part that varies depending on the type of aggregation, using loops, threads, or both, in an equivalent manner to existing methods for coordinating the execution of sequential or parallel tasks. Specifically, this script does the following:
  1. Retrieves information about the computational resources seen by Flux for debugging.
  2. Loads the jobspecs for the wrapper tasks.

3. Loads the user environment.
4. Iteratively or concurrently, the jobs are submitted to the queue.
5. Retrieves information about the resources requested for the task for debugging purposes.
6. Waits for the jobs to finish.
7. Retrieves the metadata of the completed job to extract the start and end timestamps that Autosubmit will later record in the database.
8. If the job has been completed, there will be a file named `<jobname>_COMPLETED`. If this file does not exist, the wrapper script will eventually create a file named `WRAPPER_FAILED`. These files allow Autosubmit to monitor the status of jobs.

- Next, it ensures execution permissions for the generated script and loads the Flux environment if the user has provided a custom configuration.
- It initializes Flux using `srun` and the `flux start` command, pointing to the generated Python script. The specific command used is the one in Code 4.1. In this command, srun starts the job within the allocation. The option `-cpu-bind=none` explicitly disables task binding to specific CPUs, allowing processes to use any available CPU on the assigned nodes. The option `-verbose=2` in `flux start` sets the Flux verbosity level to obtain detailed information about what is happening in the instance.

```
srun --cpu-bind=none flux start --verbose=2 python {script_name}
```

Code 4.1: The `srun` command utilized to start the Flux broker executing the Python script that coordinates wrapper execution through the API. `{script_name}` is replaced by the actual name of the script.

4.4 Enabling the Flux environment in the platform

Although Flux is a consolidated workload management software, it is not yet present in the software stack of many supercomputers. The European EESSI (Dröge et al., 2023) project, aimed at building a common software stack, already includes Flux in its latest versions. However, the EESSI version in MareNostrum 5 does not contain it yet.

In order for Autosubmit to recognize the environment on platforms where Flux is not directly available, it is necessary to explicitly request the set of bash commands required to enable it. To this end, a new second-level key has been enabled under the `WRAPPERS` key, called `CUSTOM_ENV_SETUP`. Users are responsible for specifying the commands to enable Flux in the remote platform.

4.5 Testing the implementation sources

Unit tests have been developed to test the critical components of this implementation, which include the `FluxYAMLGenerator` and `FluxYAML` classes. The development of these tests serves a dual purpose: to identify scenarios not covered by the design (in this case, invalid resource reservations) and to ensure that future changes to the code do not alter the expected behavior.

For each method in these two classes, pairs of white-box and black-box tests have been developed to detect failure points and erroneous configurations that are inadvertently accepted, and to test them by inputting incorrect values. The goal is to achieve full coverage for both classes. White-box unit tests test a method by knowing and using its internal code (loops, conditions), whereas black-box unit tests evaluate it only through its inputs and outputs, without knowing how it is implemented.

Since various components of Autosubmit were modified during the integration process, it was necessary to update the associated unit tests in some cases. The coverage for the new code reached 100%.

Chapter 5

Methodology

A new software component cannot be released to production unless its quality, correctness, and suitability for its intended purpose have been verified in advance. This chapter describes the methodology followed to validate the functionality of the new task aggregation method in Autosubmit through practical testing in a real computing environment.

5.1 Overview

In High Performance Computing environments, various unpredictable circumstances often arise that hinder the proper execution of a workflow (Chu et al., 2024). For this reason, the implementation developed in this work was evaluated by running an extensive Climate DT workflow on the MareNostrum 5 supercomputer.

This practical validation consisted of running three extensive Climate DT workflows concurrently under a single user. One of the workflows did not have wrappers, so all its tasks were submitted to the machine individually. The other two workflows applied two different wrapping methods, respectively: `ASThread`, the default in Autosubmit; and the new `FLUX` method. Furthermore, the workflows were long enough to span multiple supercomputer utilization cycles (about four entire days considering the results in this work), during which the use of wrappers might have been more or less beneficial. The purpose of this evaluation is to:

1. Determine whether the workflow that applies wrappers using the `FLUX` method completes successfully or, at the very least, whether any errors that arise are due to external causes. The workflow that does not apply wrappers is intended to help identify issues that are not related to the new implementation.

2. Ensure that the new wrapping method achieves an equivalent performance to the default method. The workflow using the default wrappers serves as a reference for what to expect from the workflow using the new method.

3. Obtain performance metrics that allow for a comparison between the three workflows. These metrics include the **time-to-solution**—which is the time elapsed from when the first workflow task is submitted until the last one is completed—the **effective execution time**—which is the time during which at least one task was running, to account for possible overlaps due to parallelism—the **effective queue time**—which is the equivalent for queue time—the **combined effective execution time**—which is the time during which there has been at least one task either in the queue or in execution—and the **workflow management time**—which constitutes those moments when the machine has no workflow task either running or queuing while the workflow manager performs local operations. In addition, throughput and average

execution times are obtained as relative metrics to facilitate fair comparison. The purpose of this is to:

- Quantify the potential overhead introduced by the use of Flux, both on the machine running Autosubmit and on the supercomputer. This includes calculating the wrapper bounds, generating the wrapper script, sending, activating the Flux environment remotely if it is not enabled by default, and the time elapsed from when the wrapper script submits the task to the queue until Flux executes it, among other factors.
- Although not in the main scope, quantify the performance improvement achieved through task aggregation, helping to reinforce the conclusions of previous research.

In addition, as a further step in this study, an exploratory analysis has been carried out taking advantage of the obtained execution data for understanding the distribution of the different wrapper sizes throughout the execution and to conclude whether the selected wrapper configuration proves its effectiveness, so that future executions of the same workflow could benefit from it.

5.2 Obtaining metrics

All metrics that are relevant to this work can be derived from the timestamps of when tasks are submitted to the platforms, when they begin to execute, and when they finish, which are stored in the SQLite databases of the experiments.

These metrics make it possible to assess performance gains or degradations in each of the three executed workflows. The calculations were carried out using a script that adheres to the pipeline below (Goitia, 2026b).

1. For each experiment, it sends a request to the database to retrieve information on all jobs that have been executed in a supercomputer allocation. This is achieved by filtering by the Quality of Service (QoS) assigned to each job. Once loaded, it performs an optional filter based on the date on which a given job was completed. This is useful for excluding jobs from the analysis starting from the moment a failure occurs. The analysis would only account for the jobs that started their execution prior the given failure. Wrappers, which are subjected to their own queue time because they are seen as a single job by Slurm, are considered atomic units (all their jobs are allowed to finish once they have left the queue). The purpose of this pruning is to ensure a fair comparison between different runs.

2. For each experiment, it calculates the union of the time intervals during which tasks are queued or running, both independently and combined. From this information, it obtains the effective queue, execution, and combined times, for then computing relative performance metrics, including the throughput and average times.

3. It stores the computed metrics per experiment and information about the analyzed jobs separately in `CSV` files. Additionally, it optionally generates another file containing the wrappers with their respective job list.

5.3 Workflow configuration

Table 5.1 summarizes the configurations used by each of the validation workflows. With this configuration, the workflow runs were extensive, consisting of 39,564 tasks each, and took several days to complete.

Three *members* were configured, each consisting of 36 chunks of one month. Daily splits were configured for all sections corresponding to the applications, with the exception of `APP_HYDROMET`, which continued to run on a monthly basis.

The workflow itself ran with the apps-only option. This means that no climate models were run; instead, data streaming was disabled by setting Data Notifier (DN) to bypass streaming with direct file reading, and postprocessing applications worked with already available data. In this post-processing intervened both the One Pass Algorithms (OPA) section, which computes the statistics required by the applications, and the `APP`, which transforms the statistics into the climate application's output (e.g. probability of wild fire, energy generation in offshore wind farms, or flood prediction). Of these three sections, only `OPA` and `APP` run on HPC allocations. `DN` runs on the login nodes. The structure of the post-processing task sections is essentially vertical—or chained—with dependencies also existing between tasks in different sections, which makes them perfect candidates for applying vertical task aggregation. Having the ESM simulation enabled would have introduced a higher computational cost and a longer time-to-solution, which would affect the calculation of the performance metrics by having less available data about the post-process (which is the real target in the analysis) for the same period.

All climate applications shown in Table 5.1 were activated, and the wrappers were applied to their `OPA` and `APP` sections. The wrappers, which were vertical according to the workflow organization, had a size ranging from 15 tasks to a value dependent on the time limit for each task established by the QoS of the account running it. That is, when a wrapper for a section has a walltime of 30 minutes and the maximum execution time allowed for the chosen account is 48 hours, up to 96 tasks can be grouped.

Table 5.1: The configurations used for validating the implementation with the Climate DT workflow.

Parameter	No wrappers	ASThread wrappers	Flux wrappers
Common workflow configuration parameters			
Members	3		
Chunks	36		
Chunk size	Month		
Splits	All sections except `APP_HYDROMET`		
Split size	Day		
Task retries	10		
Common Climate DT workflow-specific parameters			
Execution	Apps-only		

Continued on the next page

Parameter	No wrappers	ASThread wrappers	Flux wrappers
Active applications		Energy indicators Wildfires FWI Wildfires WISE Hydroland Hydromet	
Wrapper configuration parameters			
Wrappers	No	Yes	Yes
Policy	—	Flexible	Flexible
Method	—	ASThread	Flux
Type	—	Vertical	Vertical
Min size	—	15 tasks	15 tasks
Max size	—	Unlimited, QoS–dependent	Unlimited, QoS–dependent

Wrapped sections	—	`OPA_ENERGY_INDICATORS`
		`OPA_ENERGYTDIG1`
		`OPA_ENERGYTDIG2`
		`APP_ENERGY_INDICATORS`
		`OPA_WILDFIRES_FWI`
		`APP_WILDFIRES_FWI`
		`OPA_WILDFIRES_WISE`
		`APP_WILDFIRES_WISE`
		`OPA_HYDROLAND`
		`APP_HYDROLAND`
		`OPA_HYDROMET`
		`APP_HYDROMET`
			`OPA_ENERGY_INDICATORS`
			`OPA_ENERGYTDIG1`
			`OPA_ENERGYTDIG2`
			`APP_ENERGY_INDICATORS`
			`OPA_WILDFIRES_FWI`
			`APP_WILDFIRES_FWI`
			`OPA_WILDFIRES_WISE`
			`APP_WILDFIRES_WISE`
			`OPA_HYDROLAND`
			`APP_HYDROLAND`
			`OPA_HYDROMET`
			`APP_HYDROMET`

5.4 Workflow execution

The Appendix C provides a comprehensive step-by-step guide for setting up the experiments in accordance with the descriptions in Table 5.1, explaining the Autosubmit commands used, the modified configurations, and some fine-grained adjustments in resource description parameters. For that, it was utilized the Climate DT workflow version 6.1.0, which is publicly available at its official repository (Arriola et al., 2026). The experiments were managed with the custom Autosubmit version that resulted from this work (Goitia, 2026a)—which is based on v4.1.16—and were executed on the MareNostrum 5 supercomputer, which already had a Flux environment with `flux-core` v0.79.0 and `flux-sched` v0.48.0.

5.5 Considerations

Given that errors are common in HPC—for example, due to occasional node failures or network outages—it is possible that a run as extensive as those proposed in this study may not be completed without interruptions, such that the three workflows are executed under equal conditions. Due to the huge computational and temporal cost of the evaluation workflows, and the fact that re-running them in case of failure does not prevent sporadic errors from arising again, this methodology proposes that, in the event of an error, the analysis must be performed using the data available up to the point of failure, excluding the jobs that started their execution after the failure happened in the three workflows.

Moreover, due to the fact that the workflows were executed under the same user account, the utilization of tasks in one workflow can directly penalize the tasks of the others. In an ideal scenario, the evaluation would have been carried out under three different users starting from the same utilization—that is, with the same *fair share* value.

Chapter 6

Results and discussion

This chapter presents and analyzes the results obtained from running the three workflows designed in this work. Also, provides an additional exploratory analysis of the wrapper size distribution based on the obtained data.

6.1 Analysis of the workflow execution

The experiment using Flux wrappers experienced a sporadic failure when sending a wrapper. Although the cause has been determined not to be related to the implementation, this failure resulted in the cancellation of all tasks dependent on it. Consequently, the methodology was followed as described for these cases. All tasks that started execution after the finalization timestamp of the task preceding the failure were excluded from the analysis, except those inside wrappers that had an allocation already assigned.

Table 6.1 presents the analysis results for the individual tasks organized into three blocks of rows. "Individual task" refers to each task unit executed, regardless of whether it has been wrapped or not. Only tasks submitted to the scheduler are considered here (those executed on the login nodes were excluded). The first block consists of information on the workload—that is, the analyzed individual tasks—including the total number of tasks executed considering possible retries after a failure, the number of unique tasks executed excluding retries, the total number of tasks that were excluded from the analysis for each experiment due to the aforementioned failure, and the number of wrappers created per experiment, which are seen as a single job by the scheduler.

For each experiment, the second block presents time-related metrics. First, effective times reflect the concurrent nature of workflows, meaning there are moments when multiple tasks are queued or running, or none at all. For this reason, simply adding the times is not a reasonable approximation. One of these metrics is the *effective execution time*, which represents the fraction of time during which there was at least one job running on the platform. The *effective queue time* is the equivalent for the queue time, representing the time during which there was at least one job that was waiting to be executed. The *effective combined time* integrates the previous ones, taking into account possible overlaps.

Additionally, there are the *time-to-solution*—or *makespan*—which represents the fraction of time from when the first task in the workflow was submitted until the last one was completed, and the *effective workflow management time*, which is the difference between the *time-to-solution* and the *effective combined time*, quantifying those moments when the platform had no running or queuing task due to Autosubmit's processing.

In order to account for these uneven workloads, the final block presents a set of metrics relative to the workload of each workflow. The first is *throughput*, quantified as the rate of

Performance metrics	Unwrapped	ASThread	FLUX
Workload (after pruning)			
Total executed tasks	13,660	33,667	33,488
Unique tasks	13,659	33,413	33,253
Total excluded tasks	22,595	2,863	2,888
Wrapper jobs	–	893	878
Absolute effective times (seconds)			
Effective execution time	140,563	296,629	304,714
Effective queue time	302,832	181,342	185,130
Effective combined time	312,653	344,176	345,031
Time-to-solution	313,061	344,176	345,031
Effective workflow management time	408	0	0
Relative performance metrics (seconds)			
Throughput (tasks/hour)	157.08	352.15	349.41
Effective queue time per task	22.17	5.39	5.53
Effective execution time per task	10.29	8.81	9.10
Effective workflow management time per task	0.03	0.00	0.00

Table 6.1: Number of task per workflow, absolute timing statistics, and relative timing statistics by tasks in the workflow. `Unwrapped` is the workflow without wrappers, `ASThread` is the workflow applying default wrappers, and `FLUX` is the workflow applying wrappers with the new method.

tasks executed per hour. The remaining ones are the average effective queue, execution, and workflow management times per task, defined as the absolute metric divided by the total number of tasks in the workflow. A lower value indicates that the workflow managed to execute more in parallel, thereby increasing the throughput.

The first significant finding revealed by these metrics is that the experiments with wrappers (`ASThread` and `FLUX`) delivered a markedly higher throughput, achieving 2.24x and 2.22x, respectively, compared to the rate offered by the standard reference workflow. All this while maintaining a similar time-to-solution of around four days in all three cases, although the workload of each workload is different. The difference between the three time-to-solution values is due to the fact that, when pruning the tasks scheduled after the failure, the jobs that were already running were allowed to finish. Since wrappers are treated as atomic units and have a longer execution time, it was to be expected that the time-to-solution would be somewhat longer in the experiments that used them.

Another of the most relevant factors in this work is the workflow management time. This metric makes it possible to determine the time during which Autosubmit performed wrapper-related management operations—such as computing the bounds, creating, and sending them—while the platform was not queuing or running any workflow task. It can be seen that this time was 408 seconds in the case of the workflow without wrappers. However, for the two experiments with wrappers, this time was zero. This is because this time is completely overlapped with the execution of the wrappers, meaning that the

additional time Autosubmit needs to build a wrapper has zero cost on the overall execution, whether using the `ASThread` method or the `FLUX` method.

Regarding the average effective queue time, similar values can be observed for the experiments with wrappers, which contrast with the high value in the baseline experiment, thus explaining the substantial improvement in throughput mentioned above. Tasks of workflows executed within wrappers wait on average four times less in queue.

The average effective execution time shows notable differences among the three workflows. In the case of the unwrapped workflow, the average time is 10.28 seconds, a higher value due to the strong fragmentation that occurs because individual tasks spend, on average, more time blocked in the queue and less time running, resulting in very few jobs executing concurrently on the platform. This increases the fraction of effective time that falls on each individual task. In contrast, the two experiments with wrappers manage to sustain a high and steady level of concurrent execution, reducing this average effective time to 8.81 and 9.09 seconds, respectively.

Although it is not possible to determine all the factors involved in the discrepancy between the effective execution time of the `ASThread` and `FLUX` workflows with the available information, it is reasonable to attribute it to the processing overhead that Flux may incur as an *in situ* manager; however, it may also be related to physical factors (performance of the compute nodes). In any case, the difference—2.74 tasks/hour, or 0.78% of the total—supposes a minimal performance degradation that is deemed acceptable for the returns that employing Flux provides, establishing it as a fully viable *in situ* workflow manager with no significant performance penalties compared to Autosubmit's default method.

6.2 Analysis of the wrappers

Section	Size	Walltime	Samples	Mean Queue	Median Queue	Queuing efficiency
OPA	15–40	300–800 min	858	4,660.17	135.00	0.13
	41–66	820–1320 min	122	8,936.44	8,405.00	0.06
	67–92	1340–1840 min	81	3,116.86	169.00	0.48
	93–118	1860–2360 min	89	5,596.53	165.00	0.49
	119–144	2380–2880 min	52	1,164.54	138.00	0.75
APP	15–31	450–930 min	362	2,169.25	150.00	0.12
	32–48	960–1440 min	41	3,023.78	155.00	0.27
	49–64	1470–1920 min	42	4,524.62	3,669.00	0.12
	65–80	1950–2400 min	34	1,843.50	188.50	0.40
	81–96	2430–2880 min	33	7,461.52	167.00	0.62

Table 6.2: Queuing metrics by wrapper size range for both `OPA` and `APP` sections. Wrapper size ranges are measured in tasks, and their corresponding walltimes in minutes. Queue times are given in seconds, and queuing efficiency in tasks/second.

As an additional step in this work, the distribution of wrapper sizes has been studied to understand how the selected wrapping policy and the bounds configured in these experiments impact Slurm queue times and, therefore, affect the workflow performance, with the aim of optimizing the wrapping configuration of future Climate DT workflow runs.

Table 6.2 shows the distribution of the different sizes of the wrappers submitted to

MareNostrum 5. The *size* column shows the wrapper size ranges in tasks, and the *walltime* represents their maximum execution time in minutes. *Samples* represents the number of wrappers counted in each range. The mean and median queue times, in seconds, correspond to the wrappers, not the individual tasks they contain. The *queuing efficiency* represents the average rate of tasks that start execution per second, considering all the samples in the range.

The wrappers of the `OPA` and `APP` sections are analyzed separately because, although they request the same resources (an exclusive node), they specify different walltimes. Since the wrappers are vertical, the walltimes of the tasks they contain are added, so the different sections have a different impact on scheduling. The individual `OPA` tasks have a walltime of 20 minutes, compared to 30 minutes for those in `APP`. This has a direct effect on the maximum aggregation size, as it is adjusted at runtime to group as many tasks as the Quality of Service constraints allow. An `OPA` wrapper can consist of up to 144 tasks, whereas an `APP` wrapper can aggregate up to 96, given the maximum walltime of 48 hours.

The different sizes found are distributed in ranges proportional to the maximum size allowed in each section. The table shows the aggregated data from all the experiments carried out in this work. This approach is valid because, regardless of the wrapping method used, in both experiments, the wrappers from the same section and size request the same allocation from Slurm, thus affecting scheduling in the same way.

One of the most notable details is directly related to the flexible wrapping policy used, with the highest concentration of samples found in the range covered by the smallest wrappers for both sections. This indicates that, in most cases in which Autosubmit calculated the new wrappers to be sent, the number of tasks available to group ranged between 15 and 40 for `OPA` and between 15 and 31 for `APP`. Although larger wrappers are frequently found, the number of samples shows that they represent a smaller fraction in comparison—30% versus 70% for the smaller ranges.

This size distribution is directly dictated by the workflow topology, determined by the dependency (`DN` → `OPA` → `APP`) and the differences in execution speed between these sections. Workflow execution begins with the `DN` tasks which, as they are completed, start to unlock the `OPA` tasks. These `DN` tasks are completed quickly because, in this particular workflow, the data streaming between the simulation—which is not actually run—and the applications is disabled. As soon as 15 `DN` tasks are completed, the first wrappers containing 15 tasks from the `OPA` sections are submitted. The same occurs with the `OPA`–`APP` dependency. However, as execution progresses, the high speed of `DN` compared to `OPA`—and of `OPA` compared to `APP`—causes ready tasks to accumulate faster than they are executed. This accumulation effect is what allows Autosubmit to increase the size of the wrappers in its subsequent evaluations, generating intermediate and large aggregations.

In terms of performance (queuing efficiency), the overall trend seems to indicate that the larger the wrapper, the better the job execution rates obtained. However, a severe performance drop can be observed in the intermediate ranges for both sections, which corresponds to the 41–66 range in `OPA` and 49–64 in `APP`.

When analyzing the metrics in the table, it becomes clear that this penalty is not due to the execution itself, but rather to a disproportionate increase in median queue times attributable to adverse workload conditions in the platform. While the other ranges stay around 150 seconds of queue time, the medium-sized groupings shoot up to 8,405 and 3,669 seconds, respectively.

Initially, this performance degradation could be due to Slurm penalizing medium-sized *walltimes*, as it is unable to find a slot for them to run via backfilling. However, the data suggest that the inefficiency ranges do not necessarily intersect in temporal terms: while

the `OPA` trough occurs between 820–1320 minutes, the `APP` one occurs between 1470–1920 minutes. This discrepancy in the suboptimal range could be due to how the workflow topology interacts with the dynamic load on the supercomputer. In other words, both the workflow topology and the supercomputer load influence when wrappers of a given size are formed. When the supercomputer load is very high, the slow rate at which tasks are completed favors the formation of suboptimal (intermediate) wrapper sizes, which inherit that congested situation and pay for it with longer queue times. On top of this phenomenon, another one could be impacting it. As the workflows execute, there is continuous degradation of the priority of the user who launched the experiments, given that as these progress, the user's utilization increases, negatively impacting their *fair share*.

Ultimately, the smaller wrappers achieve short queue times (135–150 s) because their short walltime allows the backfill algorithm to easily fit them into free time gaps on the machine, while still being large enough to reduce the total number of jobs submitted by the workflow and to minimize the queue time. However, medium-sized wrappers could be too long to benefit from this mechanism, so they must compete in the queue, inheriting the effects of high congestion and the low fair share of the user. The larger wrappers were obtaining median queue times in these executions similar to those of the first segments. The formation of these large aggregations is the result of the progressive, natural accumulation of ready tasks as the workflow is completed during time windows in which the cluster is presumably lightly congested, allowing Slurm to process them quickly. This favorable scenario allows wrappers to face shorter queue times without needing any backfilling. This quick start of execution, combined with the huge number of tasks they complete without interruption, amortizes any penalties resulting from having medium-sized wrappers, increasing the queuing efficiency to 0.75 and 0.62 jobs/s.

In conclusion, the combined analysis of throughput and queue times shows that the size of the wrapper does not exhibit a continuously improving linear relationship, but rather a behavior influenced by the workflow topology, the workload of the machine, and the Slurm scheduling rules in MareNostrum 5. This could serve as the basis for a more comprehensive future analysis, for which additional variables would be needed to understand what was happening on the machine during execution, such as utilization information—global, for the account, and for the user—or how the jobs were scheduled (backfill or standard), among others.

Nevertheless, for future executions of the workflow, the data presented suggest that the chosen wrapping configuration is effective. By allowing the size to adapt at runtime to the dependency-resolution rate, Autosubmit ensured that the vast majority of wrappers (around 70%) remained within small ranges, thus guaranteeing short queue times by benefiting from backfilling and still reducing the total number of jobs submitted by the workflow, thereby increasing throughput compared to the unwrapped workflow in Table 6.1. Also, around 90% of the wrappers experienced fast median queue times. Although the execution conditions inevitably force the creation of medium-sized wrappers that end up trapped in a performance drop, the high performance of small wrappers and the larger ones that are occasionally created compensates for the cost of having wrappers in intermediate size ranges. However, it must be borne in mind that these workflows deactivate the extensive and expensive simulation tasks and that data streaming is simulated, making it difficult to determine its influence on the wrapper size distribution, and therefore on the queuing efficiency.

Chapter 7

Conclusions and future work

Conclusions

This work explores the feasibility of using tools for the *in situ* management of workflow tasks as engines for task aggregation. The need to integrate this type of solution into a workflow manager such as Autosubmit arises from the great complexity involved in developing and maintaining a dedicated submodule for the remote execution of interdependent tasks. By externalizing this remote management, Autosubmit only has to generate a list of the jobs that constitute the aggregation, called a *wrapper*, and execute them through the interface provided by the tool for this purpose.

Multiple candidate *in situ* managers were explored, such as Parsl, Greasy, and ExecutorLib, and, although many proved promising, among the eight analyzed, Flux was the one that best met all the requirements for its integration into Autosubmit. This work describes the software design that was necessary for its correct integration, which establishes a solid foundation both for the initial purpose of this study and for future implementations. Furthermore, to validate the functionality of this new implementation, a rigorous methodology has been designed that tests it in a real execution environment, namely MareNostrum 5, going beyond the developed tests.

This first reproducible analysis running a large Climate DT workflow—the digital twin of the Earth developed for Destination Earth—has made it possible to assess both the feasibility of the new Autosubmit integration with Flux and the overall impact of task aggregation. First, the evaluation of the new Flux-based wrapping mechanism shows that it is capable of managing the *in situ* execution of jobs with a negligible performance difference of 0.78% in favor of the current default wrapping method, attributable to the slight processing overhead introduced by Flux. This discrepancy is minimal in the overall computation. Therefore, Flux is considered to have met the orchestration requirements specified in this work, such as compatibility with multiple platforms and architectures and full resource management of allocations. Hence, it is empirically validated that its adoption does not introduce significant penalties while offering in return a modern, portable, robust, and scalable management ecosystem with strong potential for the *in situ* management of aggregations.

Secondly, the results show that the use of vertical wrappers, in which each task depends on the previous one, is a design-level requirement for the workflow to reduce queue time in congested HPC environments. The wrappers achieved a 2.2× speed-up in workflow performance compared to the execution submitting individual tasks, thanks to the substantial reduction of the queue times, which is further consistent with the conclusions of previous work that stated their benefits. Furthermore, the cost associated with computing the wrapper bounds and creating their scripts overlaps completely with the execution.

Moreover, this work presents an additional exploratory analysis of the distribution of wrapper sizes, using the data obtained from the executions to assess the effectiveness of the selected wrapper configuration. This analysis suggests that the efficiency of the scheduler in dispatching wrappers for execution did not follow a linear pattern with respect to their walltime sizes, but also depends on other factors such as the current workload state of the machine and the topology of the workflow, which strongly determine at what moment the wrappers are created and with what dimensions. The flexible policy and the dynamic wrapper size are presented as an effective option, given that approximately 90% of the wrappers moved to execution quickly, offsetting the penalties incurred in the suboptimal situations that arose during the run.

Finally, although the results of this work provide a strong basis for further action, it should be noted that the analysis was performed on a workflow that did not include the demanding simulation tasks and had the data stream disabled. Nevertheless, the evaluated workflow is considered to be highly demanding and representative of a production environment.

Future work

Throughout the development of this work, new implementation opportunities have been identified that can leverage the integration already in place. In fact, since the wrappers have a dual purpose—reducing queue times and adapting the workflow to the limits of the Quality of Service—they could ideally accept aggregations of tasks that involve more complex interdependencies. An example of this type of dependency is found in the evaluated workflow, which followed a structure that can be summarized as `DN` → `OPA` → `APP`. Aggregating tasks of the two sections that run on supercomputer allocations (`OPA` and `APP`) would be presumably complex with the hybrid wrappers currently supported. Flux would facilitate extending the current implementation to support *subworkflows* with more complex inter-section dependencies, since its API allows one to define dependencies between jobs.

Remote execution of subworkflows is possible, and one example of this is Pilot-Job systems, whose architecture allows a distributed workflow manager—where a machine manages the execution of a workflow remotely—to execute tasks by resource *binding* (tasks are scheduled on remote resources already allocated by the HPC scheduler). An implementation like the one proposed would make the work easier not only for the Autosubmit development team, but also for the users who design and configure the workflows and the wrappers.

In addition, as an immediate subsequent step, it is proposed to develop the required integration and regression tests, as the implementation will be used in real environments where there is no margin for errors. It is also essential to produce the associated documentation.

To conclude, future work focused on testing the wrapping policy would be needed to study the distribution of wrapper sizes as a function of workflow topology, its workload, and machine load, which could be especially useful for optimally configuring aggregations across different types of workflow.

Bibliography

Abellan, X., Naranjo, J., Simarro, C., Rodriguez, J., & Rodenas, P. (2021, June). *GREASY* (Version 2.2.3). https://github.com/BSC-Support-Team/GREASY

Acosta, M. C., Palomas, S., Paronuzzi Ticco, S. V., Utrera, G., Biercamp, J., Bretonniere, P.-A., Budich, R., Castrillo, M., Caubel, A., Doblas-Reyes, F., Epicoco, I., Fladrich, U., Joussaume, S., Kumar Gupta, A., Lawrence, B., Le Sager, P., Lister, G., Moine, M.-P., Rioual, J.-C., … Balaji, V. (2024). The computational and energy cost of simulation and storage for climate science: Lessons from CMIP6. *Geoscientific Model Development*, *17*(8), 3081–3098. DOI: 10.5194/gmd-17-3081-2024.

Ahn, D. H., Bass, N., Chu, A., Garlick, J., Grondona, M., Herbein, S., Ingólfsson, H. I., Koning, J., Patki, T., Scogland, T. R., Springmeyer, B., & Taufer, M. (2020). Flux: Overcoming scheduling challenges for exascale workflows. *Future Generation Computer Systems*, *110*, 202–213. DOI: 10.1016/j.future.2020.04.006.

Amstutz, P., Mikheev, M., Crusoe, M. R., Tijanić, N., Lampa, S., et al. "Existing Workflow systems. Common Workflow Language wiki, GitHub". https://s.apache.org/existing-workflow-systems *Last accessed: February 5, 2026.*

Arriola, L., Gaya, A., Beyer, S., Roura Adserias, F., Alsina, I., McClain, D., & Castrillo, M. "DestinE-Climate-DT/Workflow". https://github.com/DestinE-Climate-DT/Workflow/tree/v6.1.0 *GitHub.*

Babuji, Y., Woodard, A., Li, Z., Katz, D. S., Clifford, B., Kumar, R., Lacinski, L., Chard, R., Wozniak, J. M., Foster, I., Wilde, M., & Chard, K. (2019). Parsl: Pervasive parallel programming in Python. *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 25–36. DOI: 10.1145/3307681.3325400.

Beltrán Mora, D., Castrillo Melguizo, M., Giménez de Castro Marciani, M., de Paula Kinoshita, B., Tenório Ku, L. G., Simó Muñoz, I., Lopes, E., Goitia González, P., Gaya i Àvila, A., Bretonnière, P.-A., Macchia, F., Herrero, L., & Roura Adserias, F. (2026, March). *Autosubmit* (Version v4.1.16). Zenodo. DOI: 10.5281/zenodo.18848441.

Benavides, J., Snyder, M., Guevara, M., Soret, A., Pérez García-Pando, C., Amato, F., Querol, X., & Jorba, O. (2019). CALIOPE-Urban v1.0: coupling R-LINE with a mesoscale air quality modelling system for urban air quality forecasts over Barcelona city (Spain). *Geoscientific Model Development*, *12*(7), 2811–2835. DOI: 10.5194/gmd-12-2811-2019.

Beránek, J., Böhm, A., Palermo, G., Martinovič, J., & Jansík, B. (2024). HyperQueue: Efficient and ergonomic task graphs on hpc clusters. *SoftwareX*, *27*, 101814. DOI: 10.1016/j.softx.2024.101814.

Buizza, R., Alonso-Balmaseda, M., Brown, A., English, S., Forbes, R., Geer, A., Haiden, T., Leutbecher, M., Magnusson, L., Rodwell, M., Sleigh, M., Stockdale, T., Vitart, F., & Wedi, N. (2018). *The development and evaluation process followed at ECMWF to upgrade the Integrated Forecasting System (IFS).* European Centre for Medium Range Weather Forecasts. DOI: 10.21957/xzopnhty9.

Chu, X., Hofstätter, D., Ilager, S., Talluri, S., Kampert, D., Podareanu, D., Duplyakin, D., Brandic, I., & Iosup, A. (2024). Generic and ml workloads in an hpc datacenter: Node energy, job failures, and node-job analysis. *2024 IEEE 30th Interna-*

tional Conference on Parallel and Distributed Systems (ICPADS), 710–719. DOI: 10.1109/ICPADS63350.2024.00097.

Collado Soto, P., Acción García, E., Acin, V., Acosta-Silva, C., Burriel Navarro, H., Cantero, J., Flix Molina, J., García Navarro, J. E., González de la Hoz, S., Pacheco Pages, A., del Peso, J., Planas Teruel, E., Salt, J., Sánchez Martínez, J., Santamaría Riba, M., Torro Pastor, E., Uzum, R., & Villaplana Pérez, M. (2025). Computing activities at the Spanish Tier-1 & Tier-2s for the ATLAS Experiment in the LHC Run-3 period and towards High Luminosity. *EPJ Web Conf.*, *337*, 01098. DOI: 10.1051/epjconf/202533701098.

Crusoe, M. R., Abeln, S., Iosup, A., Amstutz, P., Chilton, J., Tijanić, N., Ménager, H., Soiland-Reyes, S., Gavrilović, B., Goble, C., & Community, T. C. (2022). Methods included: Standardizing computational reuse and portability with the common workflow language. *Commun. ACM*, *65*(6), 54–63. DOI: 10.1145/3486897.

D'Amico, M., & Gonzalez, J. C. (2021). Energy hardware and workload aware job scheduling towards interconnected HPC environments. *IEEE Transactions on Parallel and Distributed Systems*, 1–1. DOI: 10.1109/TPDS.2021.3090334.

Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P. J., Mayani, R., Chen, W., Ferreira da Silva, R., Livny, M., & Wenger, K. (2015). Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, *46*, 17–35. DOI: 10.1016/j.future.2014.10.008.

Deelman, E., Vahi, K., Rynge, M., Mayani, R., da Silva, R. F., Papadimitriou, G., & Livny, M. (2019). The evolution of the Pegasus workflow management software. *Computing in Science & Engineering*, *21*(4), 22–36. DOI: 10.1109/MCSE.2019.2919690.

Delgado Peris, A., Flix Molina, J., Hernández, J., & Pérez-Calero Yzquierdo, A. (2025). Commissioning and exploitation of the MareNostrum5 cluster at the Barcelona Supercomputing Center for CMS computing. *EPJ Web Conf.*, *337*, 01215. DOI: 10.1051/epjconf/202533701215.

Döscher, R., Acosta, M., Alessandri, A., Anthoni, P., Arsouze, T., Bergman, T., Bernardello, R., Boussetta, S., Caron, L. P., Carver, G., Castrillo, M., Catalano, F., Cvijanovic, I., Davini, P., Dekker, E., Doblas-Reyes, F. J., Docquier, D., Echevarria, P., Fladrich, U., … Zhang, Q. (2022). The EC-Earth3 Earth system model for the Coupled Model Intercomparison Project 6. *Geoscientific Model Development*, *15*(7), 2973–3020. DOI: 10.5194/gmd-15-2973-2022.

Dröge, B., Holanda Rusu, V., Hoste, K., van Leeuwen, C., O'Cais, A., & Röblitz, T. (2023). Eessi: A cross-platform ready-to-use optimised scientific software stack. *Software: Practice and Experience*, *53*(1), 176–210. DOI: https://doi.org/10.1002/spe.3075.

"Flux 0.13.0 documentation — 25/Job Specification Version 1". https://flux-framework.readthedocs.io/projects/flux-rfc/en/latest/spec_25.html *Read the Docs. Edited by: Stephen Herbein. Last accessed: March 6, 2026.*

"Flux 0.13.0 documentation — 25/Job Specification Version 1". https://flux-framework.readthedocs.io/projects/flux-rfc/en/latest/spec_14.html *Read the Docs. Edited by: Tom Scogland. Last accessed: March 6, 2026.*

Fujitsu. (2021, March). *FUJITSU Software Technical Computing Suite V4.0L20* [Available at https://software.fujitsu.com/jp/manual/manualindex/p21000155e.html]. Version 2.1. Fujitsu.

Gamblin, T., LeGendre, M., Collette, M. R., Lee, G. L., Moody, A., de Supinski, B. R., & Futral, S. (2015). The Spack Package Manager: Bringing Order to HPC Software Chaos [LLNL-CONF-669890]. DOI: 10.1145/2807591.2807623.

Goitia, P. "Testing in situ wrapper engines within Docker containers". https://gitlab.earth.bsc.es/pgoitia/docker-in-situ-wrapper-engines/-/tree/v1.0.0-tfm *BSC Earth Sciences GitLab.*

Goitia, P. "Pegasus-MPI-cluster: cpuinfo validation fails on hybrid CPU architectures – issue #2119". https://github.com/pegasus-isi/pegasus/issues/2119 *GitHub. Last accessed: February 13, 2026.*

Goitia, P. (2026a). *Autosubmit workflow manager with the Flux integration* (Version v4.1.16-35cd5ac). https://github.com/BSC-ES/autosubmit/tree/35cd5ac

Goitia, P. (2026b). *Script for calculating straightforward performance metrics for autosubmit experiments*. Zenodo. DOI: 10.5281/zenodo.19009262.

Goitia, P., Marciani, M. G., & Bosque, J. L. (2024). Autosubmit wrappers to speed up scientific production. https://hdl.handle.net/10902/33831

Guevara, M., Tena, C., Porquet, M., Jorba, O., & Pérez García-Pando, C. (2020). HERMESv3, a stand-alone multi-scale atmospheric emission modelling framework – Part 2: The bottom-up module. *Geoscientific Model Development*, *13*(3), 873–903. DOI: 10.5194/gmd-13-873-2020.

Hadade, I., Klocke, D., Enkovaara, J., Lunttila, T., Rackow, T., Engels, J. F., Frauen, C., Redler, R., Kontkanen, J., Jung, T., Sein, D., Sandu, I., Reuter, B., Wedi, N., Milinski, S., Doblas-Reyes, F., Castrillo, M., Acosta, M., Girona, S., & Manninen, P. (2025). Destination earth: The climate change adaptation digital twin. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 99–110. DOI: 10.1145/3712285.3771790.

Hoffmann, J., Bauer, P., Sandu, I., Wedi, N., Geenen, T., & Thiemert, D. (2023). Destination Earth – A digital twin in support of climate services. *Climate Services*, *30*, 100394. DOI: 10.1016/j.cliser.2023.100394.

Huber, S. P., Zoupanos, S., Uhrin, M., Talirz, L., Kahle, L., Häuselmann, R., Gresch, D., Müller, T., Yakutovich, A. V., Andersen, C. W., Ramirez, F. F., Adorf, C. S., Gargiulo, F., Kumbhar, S., Passaro, E., Johnston, C., Merkys, A., Cepellotti, A., Mounet, N., ... Pizzi, G. (2020). Aiida 1.0, a scalable computational infrastructure for automated reproducible workflows and data provenance. *Scientific Data*, *7*(300). DOI: 10.1038/s41597-020-00638-4.

Janssen, J., Taylor, M. G., Yang, P., Neugebauer, J., & Perez, D. (2025). Executorlib – up-scaling Python workflows for hierarchical heterogenous high-performance computing. *Journal of Open Source Software*, *10*(108), 7782. DOI: 10.21105/joss.07782.

Jette, M. A., & Wickberg, T. (2023). Architecture of the Slurm Workload Manager. In Klusáček Dalibor, J. Corbalán, & Rodrigo Gonzalo P (Eds.), *Job scheduling strategies for parallel processing. jsspp 2023. lecture notes in computer science, vol 14283* (pp. 3–23). Springer Nature Switzerland. DOI: 10.1007/978-3-031-43943-8_1.

Juarez, F., Ejarque, J., Badia, R. M., González Rocha, S. N., & Esquivel-Flores, O. A. (2018). Energy-aware scheduler for HPC parallel task base applications in cloud computing. *International Journal of Combinatorial Optimization Problems and Informatics*, *9*(1), 54–61. https://ijcopi.org/ojs/article/view/79

Klose, M., Jorba, O., Gonçalves Ageitos, M., Escribano, J., Dawson, M. L., Obiso, V., Di Tomaso, E., Basart, S., Montané Pinto, G., Macchia, F., Ginoux, P., Guerschman, J., Prigent, C., Huang, Y., Kok, J. F., Miller, R. L., & Pérez García-Pando, C. (2021). Mineral dust cycle in the Multiscale Online Nonhydrostatic AtmospheRe CHemistry model (MONARCH) Version 2.0. *Geoscientific Model Development*, *14*(10), 6403–6444. DOI: 10.5194/gmd-14-6403-2021.

Lacima-Nadolnik, A., Grayson, K., Roura-Adserias, F., Ghosh, S., Keller, K., Batlle, M., Gonzalez-Yeregi, I., Samsó-Cabré, M., Soret, A., & Doblas-Reyes, F. J. (2025). Near-term streamed climate information from kilometre-scale global climate models for the wind energy sector. *SSRN*. DOI: 10.2139/ssrn.5509245.

Lindeskog, M., Arneth, A., Bondeau, A., Waha, K., Seaquist, J., Olin, S., & Smith, B. (2013). Implications of accounting for land use in simulations of ecosystem carbon cycling in Africa. *Earth System Dynamics*, *4*(2), 385–407. DOI: 10.5194/esd-4-385-2013.

Madec, G., & the NEMO System Team. (2022). NEMO ocean engine. *Scientific Notes of IPSL Climate Modelling Center, 27. Zenodo.* DOI: 10.5281/zenodo.6334656.

Manubens, D., Vegas, J., & Doblas-Reyes, F. (2015). Autosubmit: A versatile tool to manage weather and climate experiments in diverse supercomputing environments. *Last accessed: January 9, 2026.* https://earth.bsc.es/wiki/lib/exe/fetch.php?media=library:external:20150506_jvegas_ec-earth2015.pdf

Marciani, M. G., Castrillo, M., Utrera, G., Acosta, M. C., Kinoshita, B. P., & Doblas-Reyes, F. (2025). Evaluating the impact of task aggregation in workflows with shared resource environments: Use case for the MONARCH application. *Geoscientific Model Development*, *18*(23), 9709–9721. DOI: 10.5194/gmd-18-9709-2025.

Merzky, A., Turilli, M., Maldonado, M., Santcroos, M., & Jha, S. (2019). Using pilot systems to execute many task workloads on supercomputers. In D. Klusáček, W. Cirne, & N. Desai (Eds.), *Job scheduling strategies for parallel processing* (pp. 61–82). Springer International Publishing. DOI: 10.1007/978-3-030-10632-4_4.

Merzky, A., Turilli, M., Titov, M., Al-Saadi, A., & Jha, S. (2022). Design and performance characterization of RADICAL-Pilot on leadership-class platforms. *IEEE Transactions on Parallel and Distributed Systems*, *33*(4), 818–829. DOI: 10.1109/TPDS.2021.3105994.

Mölder, F., Jablonski, K., Letcher, B., Hall, M., Tomkins-Tinch, C., Sochat, V., Forster, J., Lee, S., Twardziok, S., Kanitz, A., Wilm, A., Holtgrewe, M., Rahmann, S., Nahnsen, S., & Köster, J. (2021). Sustainable data analysis with Snakemake. *F1000Research*, *10*(33). DOI: 10.12688/f1000research.29032.2.

Nitzberg, B., Schopf, J. M., & Jones, J. P. (2004). PBS Pro: Grid computing and scheduling attributes. In J. Nabrzyski, J. M. Schopf, & J. Węglarz (Eds.), *Grid resource management: State of the art and future trends* (pp. 183–190). Springer US. DOI: 10.1007/978-1-4615-0509-9_13.

Forcier, J. "Paramiko documentation". https://www.paramiko.org/ *Last accessed: March 2, 2026.*

"Parsl 1.3.0-dev documentation – Error handling". https://parsl.readthedocs.io/en/latest/userguide/workflows/exceptions.html *Read the Docs. Last accessed: February 13, 2026.*

"Parsl 1.3.0-dev documentation – Quickstart". https://parsl.readthedocs.io/en/latest/quickstart.html *Read the Docs. Last accessed: February 13, 2026.*

Puiggros, A., Castrillo, M., de Paula Kinoshita, B., Bretonniere, P.-A., & Agudetse, V. (2025). Enhancing data provenance in workflow management: Integrating FAIR principles into Autosubmit and SUNSET. *EGU General Assembly 2025.* DOI: 10.5194/egusphere-egu25-2142.

Pyzer-Knapp, E. O., Pitera, J. W., Staar, P. W. J., Takeda, S., Laino, T., Sanders, D. P., Sexton, J., Smith, J. R., & Curioni, A. (2022). Accelerating materials discovery using artificial intelligence, high performance computing and robotics. *npj Computational Materials*, *8*(1), 84. DOI: 10.1038/s41524-022-00765-z.

Reuther, A., Byun, C., Arcand, W., Bestor, D., Bergeron, B., Hubbell, M., Jones, M., Michaleas, P., Prout, A., Rosa, A., & Kepner, J. (2018). Scalable system scheduling for HPC and big data. *Journal of Parallel and Distributed Computing*, *111*, 76–92. DOI: 10.1016/j.jpdc.2017.06.009.

Roura-Adserias, F., Gaya i Avila, A., Arriola i Mikele, L., Andrés-Martínez, M., Beltran Mora, D., Gonzalez Yeregui, I., Grayson, K., De Paula Kinoshita, B., Ahmed, R., Lacima-Nadolnik, A., & Castrillo, M. (2024). The data streaming in the climate adaptation digital twin: A fundamental piece to transform climate data into climate information. *EGU General Assembly 2024.* DOI: 10.5194/egusphere-egu24-2164.

Roura-Adserias, F., Gaya-Avila, A., Arriola i Meikle, L., Gonzalez-Yeregi, I., De Paula Kinoshita, B., Tollander de Balsch, J., & Castrillo, M. (2025). Climatedt workflow: A

containerized climate workflow. *EGU General Assembly 2025*. DOI: 10.5194/egusphere-egu25-4466.

Sly-Delgado, B., Phung, T. S., Thomas, C., Simonetti, D., Hennessee, A., Tovar, B., & Thain, D. (2023). TaskVine: Managing in-cluster storage for high-throughput data intensive workflows. *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, 1978–1988. DOI: 10.1145/3624062.3624277.

Smith, B., Wårlind, D., Arneth, A., Hickler, T., Leadley, P., Siltberg, J., & Zaehle, S. (2014). Implications of incorporating N cycling and N limitations on primary production in an individual-based dynamic vegetation model. *Biogeosciences*, *11*(7), 2027–2054. DOI: 10.5194/bg-11-2027-2014.

Stafford, E., Cruz, L., & Bosque, J. L. (2025). HEAPS: A novel energy-based configurable HPC scheduler. In R. Wyrzykowski, J. Dongarra, E. Deelman, & K. Karczewski (Eds.), *Parallel processing and applied mathematics* (pp. 63–76). Springer Nature Switzerland. DOI: 10.1007/978-3-031-85700-3_5.

Suarez, E., Bockelmann, H., Eicker, N., Eitzinger, J., El Sayed, S., Fieseler, T., Frank, M., Frech, P., Giesselmann, P., Hackenberg, D., Hager, G., Herten, A., Ilsche, T., Koller, B., Laure, E., Manzano, C., Oeste, S., Ott, M., Reuter, K., ... von St. Vieth, B. (2025). Energy-aware operation of HPC systems in germany. *Frontiers in High Performance Computing, Volume 3*. DOI: 10.3389/fhpcp.2025.1520207.

Tang, C., Yi, W., Occhipinti, E., Dai, Y., Gao, S., & Occhipinti, L. G. (2024). A roadmap for the development of human body digital twins. *Nature Reviews Electrical Engineering*, *1*(3), 199–207. DOI: 10.1038/s44287-024-00025-w.

"TOP500". https://top500.org/ *Last accessed: January 24, 2026.*

Turilli, M., Santcroos, M., & Jha, S. (2018). A comprehensive perspective on Pilot-Job systems. *ACM Comput. Surv.*, *51*(2). DOI: 10.1145/3177851.

Vivian, J., Rao, A. A., Nothaft, F. A., Ketchum, C., Armstrong, J., Novak, A., Pfeil, J., Narkizian, J., Deran, A. D., Musselman-Brown, A., Schmidt, H., Amstutz, P., Craft, B., Goldman, M., Rosenbloom, K., Cline, M., O'Connor, B., Hanna, M., Birger, C., ... Paten, B. (2017). Toil enables reproducible, open source, big biomedical data analyses. *Nature Biotechnology*, *35*(4), 314–316. DOI: 10.1038/nbt.3772.

Voss, K., Van der Auwera, G., & Gentry, J. "Full-stack genomics pipelining with GATK4 + WDL + Cromwell". *[version 1; not peer reviewed]. F1000Research, 6(ISCB Comm J):1381 (slides)*. DOI: 10.7490/f1000research.1114634.1.

Yamazaki, D., Kanae, S., Kim, H., & Oki, T. (2011). A physically based description of floodplain inundation dynamics in a global river routing model. *Water Resources Research*, *47*(4). DOI: 10.1029/2010WR009726.

Yang, S. (2025a). *ECWMF Special Project request 2025: "EC-EARTH4: developing a next-generation European Earth System model based on ECMWF modelling systems"* (Special Project Request). European Centre for Medium-Range Weather Forecasts (ECMWF). https://www.ecmwf.int/sites/default/files/special_projects/2025/spnltune-2025-request.pdf

Yang, S. (2025b, June). *ECWMF Special Project Progress Report 2025: "EC-EARTH4: developing a next-generation European Earth System model based on ECMWF modelling systems"* (Special Project Progress Report). European Centre for Medium-Range Weather Forecasts (ECMWF). https://www.ecmwf.int/sites/default/files/special_projects/2025/spnltune-2025-report1.pdf

Appendices

Appendix A

Wrapper engines summary

In this chapter, the most significant aspects of each candidate wrapper engine are summarized. Characteristics analyzed include ease of deployment, workflow definition, and the ability to manage resources and errors, in the context of task aggregation.

It is important to note that some of the features of these tools have not been tested nor explored, and that this evaluation is partial, with the aim of determining their suitability as wrapper engines.

The ease of deployment described here states the installation method that has been utilized for each tool, and subjectively classifies the effort made to deploy the tool, either locally or in a HPC platform, according to the column containing the tested machines.

The workflow definition summarizes how a workflow can be declared with the available methods. Then, resource management described the main impressions regarding resource handling and assignation to the workflow tasks.

Failure tolerance describes the available methods, if any, for handling eventual errors in the workflow execution, and the subsequent behavior.

WMS	Ease of deployment	Workflow definition	Resource management	Failure tolerance	Tested machines	Other comments
Toil	Easy (PyPI)	- Python functions as jobs. - Suports both explicit and data-driven dependencies. - Supports CWL and WDL standards.	Unclear core binding and node count recording.	Unclear error handling.	Local	—

ExecutorLib	Easy (Conda)	- Python functions as jobs. - Python script. - Data-driven dependencies.	- Executor-dependent behavior. - Multi-node available. - Jobs are sent to fixed workers, except for `FluxExecutor`. - Workers cannot be assigned resources manually, but tasks.	Errors and retries must be handled manually.	Local	Only `FluxExecutor` allows to specify advanced scheduling parameters and offers native support for bash applications.
Greasy	Easy (source)	- Engine-dependent behavior. - Tasks and dependencies explicitly specified through a plain text file. - Syntax checker available.	- The number of concurrent jobs depends on the count of workers. - Not sure about the resource assignment to workers. - Multi-node configurations expected to be allowed by the *basic* executor, but not tested.	- Detects failures on tasks. - Implements retries. - When a task definitely fails, it generates another plain text file of the remaining workflow to resume the execution from there.	MareNostrum 5	Not tested a fresh install with custom configurations.
RADICAL-Pilot	Easy (PyPI)	- Python script. - No dependencies (barriers required). - Does not allow calls to `mpirun`	- Pilot-based. - Tasks are correctly assigned to resources when available.	- No retries. - Execution continues despite failures. - Manual error handling.	Local MareNostrum 5	- Pilots can take a while to become available. - Certain overhead in the execution of concurrent tasks has been detected.
Pegasus-MPI-Cluster	Medium (source)	Tasks and dependencies explicitly specified through a `DAX` file.	- Manager-worker basis. - One core for the manager, remaining ones for workers. - Time-slicing detected when the amount of resources of the concurrent tasks exceeded the available. - Unclear support for MPI applications.	- Supports retries. - Allows to resume the execution from the failed job.	Local MareNostrum 5	Built over MPI.
Parsl	Easy (PyPI)	- Python script. - Data-driven dependencies.	- Behavior depends on the *executor*, the *provider*, and the *launcher*. - Good resource management, including multi-node configurations.	Not tested, but integrates three mechanisms: *exceptions*, *retries* and *retry handlers*.	Local MareNostrum 5	—

TaskVine	- Easy (Conda) - Medium (source)	- Python script. - Data-driven dependencies. - Manager and workers require explicit instantiation, indicating IP + port for network communication.	- Behavior depends on the manager and worker configuration. - Seems to do it well, but core binding is unclear.	- Retries allow for resubmitting tasks that failed due to misconfiguration or platform-related errors, for example. - Error handling for tasks returning an error code would need be done manually.	Local	—
Flux	Easy (both Spack and Conda)	- Python bindings allow for defining both tasks and data-driven workflows. - CLI allows to define workflows with task dependencies.	Since it is a full scheduler itself, it manages resources within allocations seamlessly.	- Requeues jobs accordingly to failure and configuration. - Allows for manual error handling.	MareNostrum 5	It has been seen Flux allocating more than resources available.

Table A.1: Overview of the key features of the evaluated Workflow Management Systems.

Appendix B

The complete class diagram for `autosubmit.platforms`

B.1 Before Flux integration

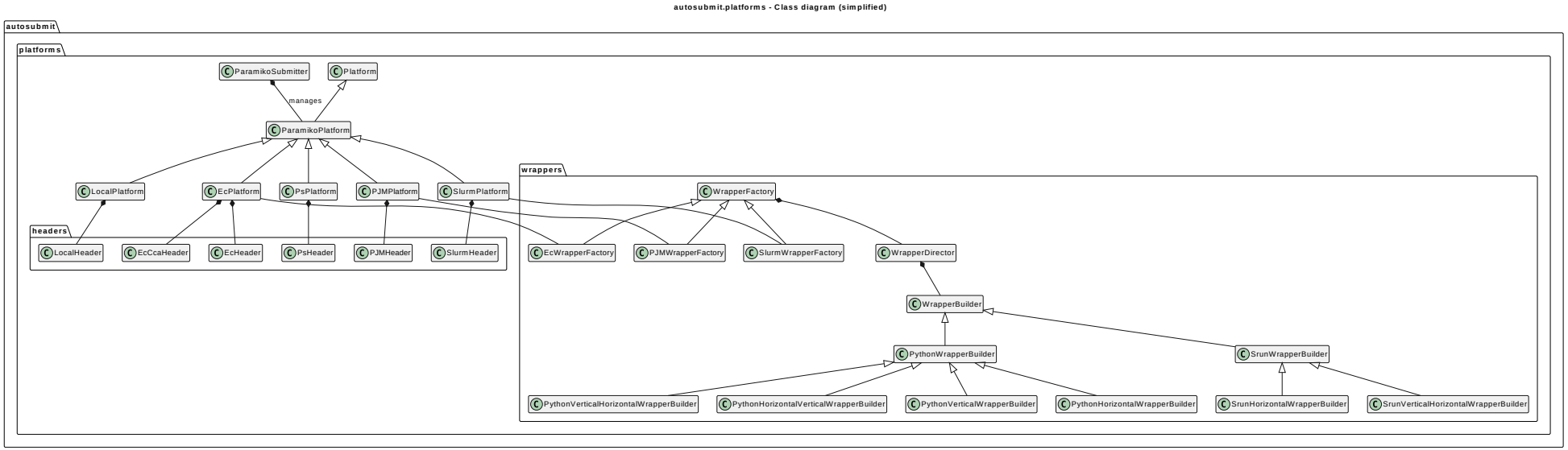


Figure B.1: Class diagram with a complete representation of all the components under the `autosubmit.platforms` package.

B.2 After Flux integration

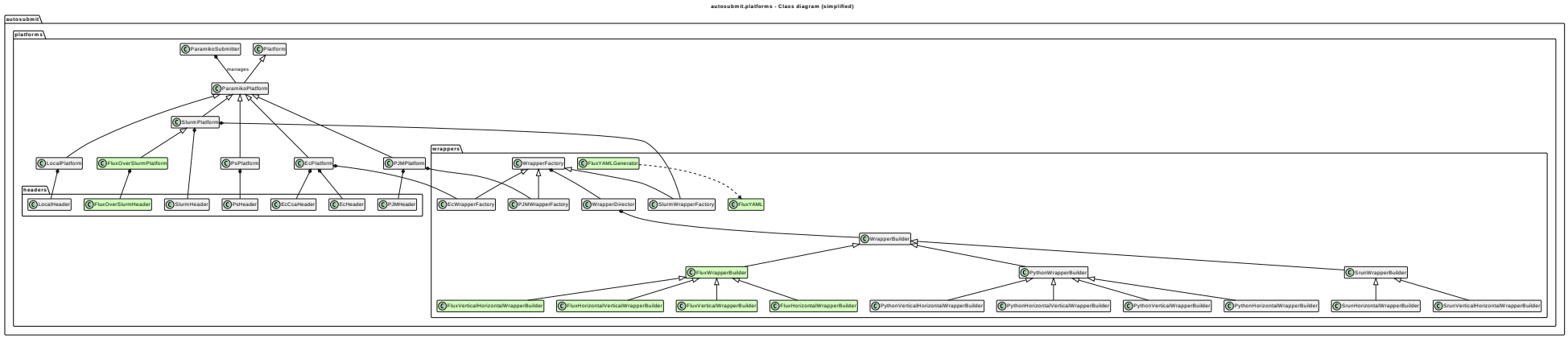


Figure B.2: Class diagram with a complete representation of all the components under the `autosubmit.platforms` package, after the Flux integration. New components appear highlighted.

Appendix C

Steps for executing the workflow

In this appendix, it is described how to reproduce the experiment creation, execution, and monitoring. For that, it was employed the Climate DT workflow v6.1.0 (Arriola et al., 2026). The workflow executed on MareNostrum 5 using a custom Autosubmit build with the integration resulting from this work, based on v4.1.16 (Goitia, 2026a). The `flux-core` version used was v0.79.0, and `flux-sched`'s was v0.48.0.

C.1 Experiment building

In order to build an experiment, it is necessary to execute the `autosubmit expid` command described in C.1. This command creates a new empty experiment of type `git` that will use remote sources. In this case, the `git_repo` flag points to the repository, and the `git_as_conf` indicates where the configuration files are placed in those sources. The `minimal_configuration` is used to indicate Autosubmit to generate only a global configuration file, since the sources already contain custom configuration files. The flag `description` is used to provide a brief explanation of the experiment. The command returns the experiment identifier, or `expid`, of the new experiment and the path to its directory.

```
autosubmit expid --description "Custom Climate DT Workflow with Flux
↪  wrappers" --HPC marenostrum5 --minimal_configuration --git_as_conf
↪  conf/bootstrap/ --git_repo
↪  https://github.com/DestinE-Climate-DT/Workflow.git
```

Code C.1: Example of experiment building from scratch in Autosubmit. A minimal git-based experiment will be created.

C.2 Workflow configuration

The default configuration will fetch the `main` branch of the repository, so the `conf/minimal.yml` file in the experiment folder must be adjusted to point to the `v6.1.0` tag, as detailed in C.2

```
GIT:
  PROJECT_ORIGIN: https://github.com/DestinE-Climate-DT/Workflow.git
  PROJECT_BRANCH: 'v6.1.0' # <-- MODIFIED
```

```
  PROJECT_COMMIT: ''
  PROJECT_SUBMODULES: ''
  FETCH_SINGLE_BRANCH: true
```

Code C.2: Configuration required for the GIT key in conf/minimal.yml.

The experiment can be created by running the autosubmit create command as explained in Code C.3, assuming that the expid is a3b0.

```
autosubmit create a3b0
```

Code C.3: Example of experiment creation in Autosubmit. It is supposed the expid is a3b0.

By creating the experiment for the first time, all the project sources will be fetched. In order to re-fetch the sources, the autosubmit refresh command should be employed. After fetching, an interactive script will be prompted asking the user for the type of workflow that will be used. From all the options, one has to select app. After that, an editor will be displayed with the content of the new conf/minimal.yml file.

The APP key requires to be overridden with the configuration in Code C.4. This configuration will enable all the applications described in Table 5.1. The same applies to the key EXPERIMENT, in which the *startdates*, *members*, *chunks*, *splits*, and their properties are defined. Code C.5 represents how the configuration in Table 5.1 is applied.

```
APP:
  ENERGY_INDICATORS: 'True'
  ENERGY_OFFSHORE: 'False'
  WILDFIRES_FWI: 'True'
  WILDFIRES_WISE: 'True'
  HYDROLAND: 'True'
  HYDROMET: 'True'
  READ_FROM_DATABRIDGE: 'False'
```

Code C.4: Configuration required for the APP key in conf/main.yml.

```
EXPERIMENT:
  DATELIST: 20200101
  MEMBERS: fc0 fc1 fc2
  CHUNKSIZEUNIT: month
  SPLITSIZEUNIT: day
  CHUNKSIZE: 1
  NUMCHUNKS: 36
  CALENDAR: standard
```

Code C.5: Configuration required for the EXPERIMENT key in conf/main.yml.

Another required change in the conf/main.yml file includes the following keys under REQUEST.

```yaml
REQUEST:
  ACTIVITY: projections
  EXPERIMENT: SSP3-7.0
  GENERATION: 2
  REALIZATION: 2
  MODEL: IFS-NEMO
  RESOLUTION: high
  EXPVER: a2ii
  FDB_HOME: "%CONFIGURATION.FDB_DIR%/a2ii/fdb"
```

Code C.6: Configuration required for the `REQUEST` key in `conf/main.yml`.

Creation might take some time to complete due to the size of the workflow. Moreover, some submodules in the repository require authentication.

After creation, some workflow-specific configurations need to be changed to replicate the behavior of the experiments in this work, involving task and platform definition sections, and also other granular adjustments. All the files referenced in the following lines are under the `proj/git_project/conf` directory of the experiment.

The first changes to be made after the experiment creation involve the `applications/opa/opa.yml` file, which contains several configuration parameters related to the `OPA` sections for each platform. This file presents configurations for different platforms, but only MareNostrum 5 needs to receive the changes in Code C.7.

```yaml
PLATFORMS:
  # [...]
  MARENOSTRUM5:
    OPA_EXCLUSIVE: True
    OPA_PROCESSORS: 1
    # [...]
```

Code C.7: `OPA` processors adjustment in file `applications/opa/opa.yml`. The dots indicate the omission of parameters that must not be modified.

The `additional_jobs` directory contains the files for the specification of some jobs, including the applications that have been previously enabled. The configurations in Code C.8 must be applied to the `APP` section for each enabled application. Code C.9, explains the configuration applied to the `OPA` sections instead. Here, the configuration for the *energy indicators* tasks is explained, which can be found in the `energy_indicators-True.yml` file.

```yaml
NODES: 1
PROCESSORS: 1
TASKS: 1
THREADS: 1
EXCLUSIVE: True # <-- NEW
RETRIALS: 10    # <-- NEW
```

Code C.8: Job resource specification for the `APP` section in file `additional_jobs/energy_indicators-True.yml`. This configuration is the same for all the applications.

```
NODES: 1
PROCESSORS: '%CURRENT_OPA_PROCESSORS%'
TASKS: 1
THREADS: 1      # <-- DELETED
RETRIALS: 10    # <-- MODIFIED
```

Code C.9: Job resource specification for the `OPA` section in file `additional_jobs/additional_jobs/energy_indicators-True.yml`. `TASKS` This configuration is the same for all the applications.

These changes in resource specification do not really affect the allocation characteristics the workflow developers specified, but it is an equivalent way needed to make Flux replicate the Slurm behavior while treating the resources within the allocation. The original configuration led Flux wrappers to raise out of memory errors due to how the resource parameters were being parsed.

A total of ten retries have been configured for all the sections that will eventually be wrapped in order to avoid a punctual job failure to stop the entire workflow.

Other small changes must be made in `applications/default_gsv_request.yml` (see Code C.10) and in `applications/container_versions.yml` (see Code C.11) for precise reproducibility.

```
ENERGY_INDICATORS:
  GRID: "0.1/0.1"
  METHOD: "nn"
```

Code C.10: Grid configuration required for the `ENERGY_INDICATORS` application in file `applications/default_gsv_request.yml`.

```
OPA:
  VERSION: "0.9.1"
```

Code C.11: `OPA` version adjustment in file `applications/container_versions.yml`.

C.3 Wrapper configuration

Once both the experiment and its tasks are configured, wrappers need to be applied to the different sections of the enabled applications. This can be done by adding the corresponding keys in the task specification YAML files (in the `additional_jobs` directory).

Code C.12 contains the configuration that must be appended to each task specification file, for each task section they contain, to apply the `ASThead` wrappers specified in Table 5.1. Moreover, Code C.13 contains the equivalent configuration for applying the `FLUX` wrappers. Both configurations belong to the file `additional_jobs/hydroland-True.yml`, but must

be applied to all enabled applications. The reference experiment without wrappers must not contain any of the following configurations.

```
WRAPPERS:
  METHOD: ASTHREAD
  WRAPPER_HYDROLAND_OPA:
    POLICY: flexible
    MIN_WRAPPED: 15
    TYPE: vertical
    JOBS_IN_WRAPPER: OPA_HYDROLAND
  WRAPPER_HYDROLAND_APP:
    POLICY: flexible
    MIN_WRAPPED: 15
    TYPE: vertical
    JOBS_IN_WRAPPER: APP_HYDROLAND
```

Code C.12: The `ASThread` wrapper definition for the `OPA` and `APP` task sections in the *hydroland* application.

```
WRAPPERS:
  METHOD: FLUX
  WRAPPER_HYDROLAND_OPA:
    POLICY: flexible
    MIN_WRAPPED: 15
    TYPE: vertical
    JOBS_IN_WRAPPER: OPA_HYDROLAND
  WRAPPER_HYDROLAND_APP:
    POLICY: flexible
    MIN_WRAPPED: 15
    TYPE: vertical
    JOBS_IN_WRAPPER: APP_HYDROLAND
```

Code C.13: The `FLUX` wrapper definition for the `OPA` and `APP` task sections in the *hydroland* application.

In case the `FLUX` method is used, it is needed to specify the custom configuration script that will be executed in remote to enable the Flux environment on the file `conf/main.yml`. Code C.14 presents the script that was used in this work.

```
WRAPPERS:
  CUSTOM_ENV_SETUP: |
    module load miniconda
    source <absolute_path_to>/conda.sh
    conda activate flux
    conda info
```

Code C.14: The global custom configuration for enabling the Flux environment in remote. The path to `conda.sh` must be replaced.

C.4 Experiment execution

Once the experiment is fully configured, it is needed to run the `autosubmit create` again in order to update the parameters. This second time, the sources will not be fetched. When an experiment has wrappers enabled, it must be indicated in the creation step by setting the `check_wrapper` flag. The entire process can take some time to complete, and after completion, a window with the workflow diagram will be shown. Since the workflow contains thousands of tasks, it is recommended to disable diagram generation to save time by using the `noplot` flag. All these considerations have been reflected in Code C.15.

```
# Experiment with no wrappers
autosubmit create --noplot a3b0
# Experiment with wrappers
autosubmit create --check_wrapper --noplot a3b0
```

Code C.15: Example of experiment creation in Autosubmit, considering wrapper existence. It is supposed that the experiment identifier is a3b0.

Then, the experiment can be executed using the `autosubmit run` command, as in Code C.16.

```
autosubmit run a3b0
```

Code C.16: Example of experiment execution in Autosubmit. It is supposed that the experiment identifier is a3b0.

The execution can be monitored through the GUI, if available, of by using the `autosubmit monitor` command.