DESIGN AND IMPLEMENTATION OF A LARGE-SCALE SCIENTIFIC GATEWAY SYSTEM USING PROVENANCE

Felipe Figueira Horta

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Marta Lima de Queirós Mattoso
Renato Nascimento Elias

Rio de Janeiro
Agosto de 2015

DESIGN AND IMPLEMENTATION OF A LARGE-SCALE SCIENTIFIC GATEWAY SYSTEM USING PROVENANCE

Felipe Figueira Horta

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Marta Lima de Queirós Mattoso, D.Sc.

Prof. Renato Nascimento Elias, D.Sc.

Prof. Alexandre de Assis Bento Lima, D.Sc.

Prof. José Jerônimo Camata, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
AGOSTO DE 2015

Horta, Felipe Figueira

Design and Implementation of a Large-Scale Scientific Gateway System Using Provenance/Felipe Figueira Horta. – Rio de Janeiro: UFRJ/COPPE, 2015.

X, 92 p.: il.; 29, 7cm.

Orientadores: Marta Lima de Queirós Mattoso

Renato Nascimento Elias

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2015.

Bibliography: p. 82 – 92.

1. scientific gateway system. 2. provenance. 3. visualization. 4. scientific workflows. I. Mattoso, Marta Lima de Queirós *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

À minha família.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

DESIGN E IMPLEMENTAÇÃO DE UM PORTAL CIENTÍFICO PARA EXPERIMENTOS EM LARGA-ESCALA UTILIZANDO A PROVENIÊNCIA

Felipe Figueira Horta

Agosto/2015

Orientadores: Marta Lima de Queirós Mattoso
Renato Nascimento Elias

Programa: Engenharia de Sistemas e Computação

Computação científica em larga escala geralmente se baseia em tarefas intensivas em processamento encadeadas através de um *workflow* executado em um ambiente de alto desempenho (PAD). Neste contexto, cientistas modelam seus *workflows* para posteriormente submetê-los à execução em PAD fazendo uso de Sistemas de Gerenciamento de *Workflows* Científicos (SGWfC), que costumam melhorar o gerenciamento dos dados do experimento. Infelizmente, quando SGWfCs executam o experimento como caixas-pretas, cientistas costumam achar meios de rastrear a execução, *e.g.* seguindo a evolução dos cálculos, abrindo e navegando em arquivos comumente espalhados em uma arquitetura distribuída. Para localizar determinada convergência, *e.g.* cientistas tentam exportar dados parciais do PAD, na tentativa de visualizar a evolução do experimento. No entanto, tal processo pode ser complexo e sensível à erros, principalmente porque o usuário deve "adivinhar" o contexto da geração dos resultados uma vez que dados e metadados não estão conectados. Neste trabalho, propõe-se um portal científico que permite ao usuário gerenciar a execução de *workflows* em larga-escala com base em consultas em tempo de execução à sua proveniência. Proteus oferece uma arquitetura adequada ao suporte e integração de novas aplicações que visem usufruir do acesso desacoplado à proveniência de dados e ambientes de execução. Para avaliar a arquitetura proposta, implementamos e acoplamos ao Proteus um primeiro módulo dedicado à visualização. Este módulo se integra ao portal promovendo a visualização com base na proveniência. Desta forma, Proteus é avaliado enquanto provê suporte à inclusão de novas aplicações, e também enquanto fornece análise visual de resultados parciais de um experimento de Quantificação de Incerteza enriquecido pela proveniência.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

DESIGN AND IMPLEMENTATION OF A LARGE-SCALE SCIENTIFIC GATEWAY SYSTEM USING PROVENANCE

Felipe Figueira Horta

August/2015

Advisors: Marta Lima de Queirós Mattoso
Renato Nascimento Elias

Department: Systems Engineering and Computer Science

Large-scale scientific computing often relies on intensive CPU tasks chained through a workflow running on a high-performance environment (HPC). In this context, scientists model their workflows for later submission on dedicated HPC making use of Scientific Workflow Management Systems (SWfMS), which may improve data management on scientific workflows. Unfortunately, when SWfMSs execute experiments as black-boxes, scientists typically find ways of tracking the execution, *i.e* following the evolution of computations by opening and browsing files commonly spread over a distributed architecture. To track the convergence of a given experiment, scientists even try to stage out some data to visualize the workflow execution. However, such process is complex and error prone, mainly because the user has to "guess" the context of this partial result generation once files and their metadata are not connected. So that, this scenario hides potential issues given the interface between scientists; workflows models; experiment data; and the execution environment. In this work, we propose a scientific portal, which allows users to manage the execution of large-scale scientific workflows based on runtime provenance queries. Proteus provides a suitable architecture that supports and integrates with new applications aimed to take advantage of unbound access to experiment's provenance and execution environment. In order to evaluate the proposed architecture, we implemented and integrated a first Proteus' aplication, dedicated to enhance experiment's visualization. This application integrates with Proteus promoting provenance visualization based on provenance queries. Therefore, Proteus is evaluated while providing support to port new applications, as well as providing partial visual analysis of a Uncertainty Quantification experiment enriched by provenance.

Contents

List of Figures

Chapter 1

Introduction

This first chapter aims at introducing the idea behind of Proteus Scientific Gateway, drawing a first picture of its motivation, related works, problem statement, goals and describing the following chapters. In this very beginning, we point where along this writing each information can be found in details.

1.1 Motivation

Several large-scale experiments demand High-Performance Computing (HPC). These experiments usually rely on compute-intensive activities that may be chained as a scientific workflow [2–4]. The workflow activities may call computer programs and scripts that consume and produce data of different types. Unfortunately, as the workflow scales in terms of the number of activities or complexity, it gets harder to manage its conceptual model, its evolutionary versions, and subsequently to follow the several executions and produced data. Besides data management complexity, the visualization of partial results may also be complex to be performed on large-scale scenarios [5–7]. Finally, when scientists are able to monitor and steer the workflow, they may discover that computations are incorrect or that a lot of information has to be filtered to achieve better results. On another scenario, they simply get lost in the huge amount of data struggling to know which input produced which output. These actions are part of the experiment's lifecycle (Section 2).

Scientific Workflow Management Systems (SWfMS) may improve data management on scientific workflows [2]. They make it easier to represent and reference workflow data through provenance [8]. Provenance is a key feature in SWfMS since it allows for keeping track of everything that happens during workflow execution [9, 10] (see Section 2). So that, in this dissertation, to support the experiment lifecycle we propose and design a scientific gateway platform based on runtime provenance queries provided by Chiron [11] or SciCumulus [12] workflow engines. This solution, named Proteus, intends to support the whole experiment lifecycle by "docking" new

applications on its framework. Therefore, in this work, Proteus is assessed regarding its compatibility since it may port future modules, or integrate old projects adaptations on its application layer. So, after designing and implementing Proteus, we propose and evaluate its first module dedicated on visualizing of partial and final provenance-enriched results of a given experiment. Further, we propose other important modules that could also be of great aid to support experiment lifecycle.

To assist partially - or entirely - the lifecycle of an experiment is a challenging and critical issue that can be accomplished with different approaches. Although this dissertation proposes another approach, we claim that to have a clear understanding of our contributions, it is necessary to assess the gap where the related works didn't manage to fill. And, in advance, this gap lays on runtime provenance access.

Still, describing all works related to this process can be arduous since projects that care about this subject are numerous, and as far as we know, date from 1988 [13]. Therefore, in this section we describe the lifecycle of an experiment with the following steps illustrated in Figure 1.1: (i) Preparation; (ii) Execution, Analysis and Reflection; and (iii) Dissemination. Such steps, which commonly represent the methodology applied to most of the scientific processes, are described below, and may help the reader to understand the related works contributions presented shortly after that. The reader may also notice that it would be quite complicated to describe projects per features since most of them aggregate many among different steps of the experiment lifecycle.

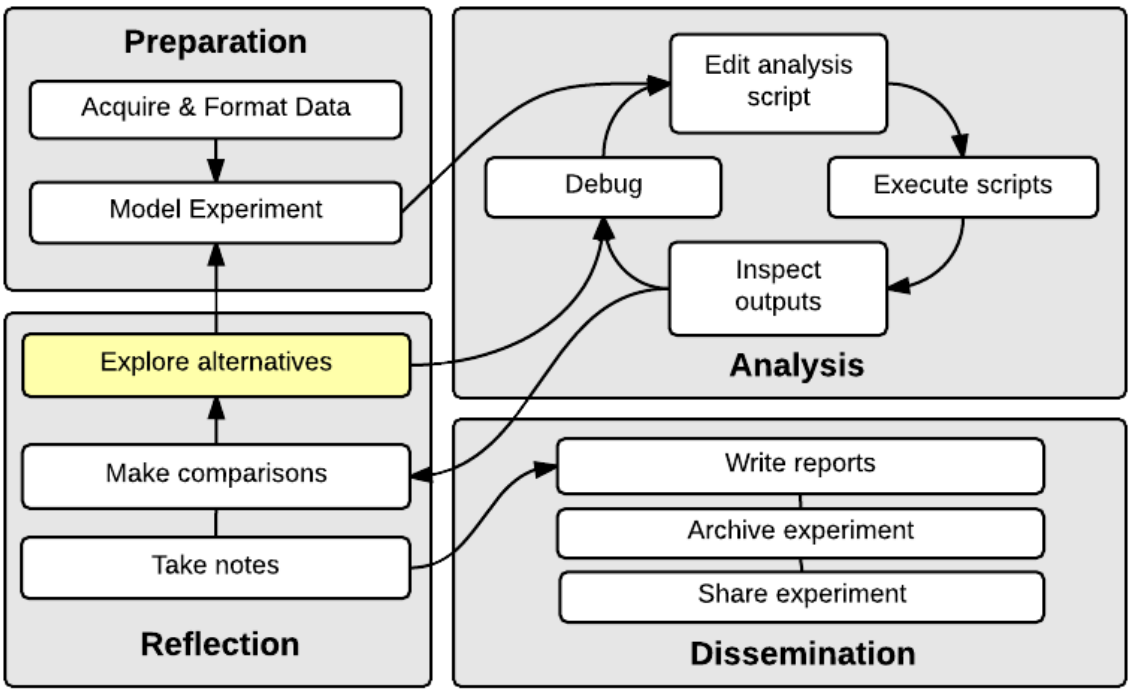

Figure 1.1: Steps involved in a typical data science workflow.

Preparation

According to CALLAHAN *et al.* [14], paradigms for modeling and visualization of complex ecosystems are evolving quickly, creating ample opportunities for scientists and society. But many powerful and integrative modeling and visualization systems have been placed in an extremely challenging position due to the late shift in modeling paradigms. Additionally, authors take the core of Environmental Observation and Forecasting Systems (EOFS) as an example; and explain that the breaking of traditional modeling cycles has been leading EOFS to this kind of problem, putting at risk of losing control of the quality of operational simulations. For such models, tight production schedules - managed by real-time forecasts and multi-decade simulation databases - drive (even today) to multiple complex runs being produced on a daily basis, resulting in a large number of associated visualization products. So that, the preparation phase is extremely valuable, because even after acquiring and formatting a huge amount of data from thousands of sensors, scientists continually revisit how they modeled experiment and how it affects production.

Execution, Analysis, and Reflection

The next step after preparing the experiment is to execute it in a proper environment given the tasks involved. So that, scientists run the experiment looking forward to analyzing its outputs at each round; making comparisons, and exploring new alternatives within the current experiment model; taking notes of each step until get satisfied with the results. For example, after modeling complex ecosystems such as EOFS, it may be necessary to run them in an appropriate execution environment for extensive calculations and further analysis. Thus, often scientists need to encapsulate the whole experiment and dispatch it to, *e.g.* a cluster or cloud environment for each run until they get satisfied with the results. This process can be arduous due to: the large size of the experiment and its fragments; the large number of executions while the experiment model evolve; or the number of times scientist may sweep parameter following a given convergence condition.

Managing the visualization and analysis of results of large-scale experiments manually is laborious and error-prone due the amount of data, its heterogeneity and granularity. One of the unfortunate consequences of this is the high cost of data transfers, knowing that often some retrieved results (from remote execution environment) may not be relevant to the analysis. This kind of inconvenience happens due to data fragments that usually are dissociated from their metadata, making it hard to identify their use; or as each analysis may focus on a subset of the outputs, the relevance of each result data might be obscured.

Dissemination

As in science it is imperative to revisit and reuse past results, the reproducibility of an experiment represents the core component of the scientific process - *i.e.* which pushes science forward. However, this task is not easy. Even after facing earlier challenges - while modeling, running and visualizing data - until finally accomplish a given experiment with success, scientists face the problem of describing and packaging it in enough details, so that other researchers can subsequently reproduce it again [15, 16].

In the scientific community, scientists are used to testing and extending published results looking to practical progress in the way that science moves forward. So, using past works allow them not only to not start from the beginning but also overcome previous knowledge. But, this long tradition in natural science requires experiments to be described in enough detail so that other researchers can reproduce them. This pattern, however, has not been universally applied for computational experiments. Consequently, researchers regularly have to rely on tables, plots and figure captions included in papers. This behavior led to a credibility crisis in computational science since it disrupt the verification and reproducibility of a given publication.

1.2 Related Works

In the context of TeraGrid [17] - one of the world's largest distributed cyberinfrastructures for open scientific research by 2008 - science gateways were formerly proposed to interface scientists and computational resources. Nowadays, most science gateways provide users with either web clients, desktop clients or both, to interact with the applications and simulations breaking geography limitations. Behind these clients, it is hidden the underlying computational complexities that allow scientists to: concept, execute, analyze and disseminate experiments. Running applications without concern over where the computation takes place.

As stated before, by supporting the experiment lifecycle, the computation being launched from the science gateway can be individual applications or a full workflow. As we discuss at Chapter 2, a workflow is an abstract description of tasks required for executing a particular real world process, and interactions between them [18] - what may translate the process of a given experiment. In this context, each task is defined by a set of activities to be conducted, either by people or system functions [19].

According to NGUYEN *et al.* [20], workflow technologies can be categorized into two broad families: business workflow (BWF) and scientific workflow (SWF). While BWF is control-flow oriented - focusing on control of the system -, SWF tends to

have a dataflow-oriented execution model. The workflow execution distribution is inherent in the workflow execution family or engine, and the running machines are associated with a particular scientific workflow management system (SWfMS). An SWfMS provides the construction of scientific workflows, interpreting the workflow described, and running it on a workflow engine. There are many types of SWfMS with different approaches to model or describe workflows and different methods for their executions. During this time, several business workflow systems have also been adapted and used in workflow-based science gateways [21, 22].

Several independent works have developed SWfMSs. Each one with its particular strategy, they approach the same problem covering a set of needed features to support the experiment lifecycle. Among them, we can number Kepler [23], Taverna [24], Vistrails [14], Pegasus [8], Swift [25] and Chiron [11]. And as a maturation process, one by one has been integrated under a Science Gateway System (SGS) - for many reasons we may discuss later - such as CAMERA [26], CrowdLabs [27], BioWep [28], Pegasus Portal [29], Swift Portal [30], and Workways [31]. Fortunately, the fact that each one works over an SWfMS eases our process of evaluation of such SGS. Once knowing the SWfMS, it can help us to judge the expected SGSs' main features.

Many SWfMS such as VisTrails [14] or Kepler [23] are focused on the development and local execution of the workflow. They offer a graphical interface for visual modeling of workflows and propose components to create activities of various types, for example, activities that invoke a local program or web service. Primarily, these SWfMS have execution machines for local scientific workflow, i.e., they will process each activity from the scientist's machine. These systems allow to remotely invoke an activity, such as a web service or running in a cluster or grid, however, remote infrastructure execution is decoupled from the internal control SWfMS and must be pre-configured to be invoked by the manager. The focus of the execution of this type of SWfMS is the of running workflows.

CAMERA [26] and Workways [18] are both Kepler workflow-based science gateways. CAMERA supports microbial ecology researches, and its single target is Metagenomics. It works as infrastructure platform, providing a structure where workflows can be uploaded into the system, and users can interface components for launching and viewing results automatically generated. Still, Workways is a general purpose interactive web portal. It lets the user insert and export data from a workflow running as a service. Workways is also capable of supporting human-in-the-loop (HIL) workflows, offering an alternative approach to assessing produced data at runtime by integrating Kepler with other systems, like Nimrod/K [32] allowing remote distributed execution. However, its decoupled implementation affect how provenance is generated. Moreover, although WorkWays supports HIL, it does

not provide integrated tools for data analysis and level traversing, making it difficult to associate an obtained result with the scenario that produced it.

CrowdLabs [27] works mainly above Vistrails infrastructure, but claims that can be integrated with any workflow management system that runs in server mode and expose an API. CrowdLabs particularly adopts the model used by social Web sites combining a set of usable tools that foster collaboration where scientists can make friends, join groups, write blogs, create projects and disseminate experiments and results together with its provenance. Its scalable infrastructure is provided by Vistrails data servers previous configured that can be triggered by XML-RPC protocol. In this scenario, Vistrails offers a provenance cache generated dynamically per experiments stored at servers while CrowdLabs application maintains another cache for results. Although CrowdLabs is a great tool for scientists to collaboratively analyze and visualize data - leveraging reproducibility [16, 33–35] - as far as we know, it does not present any alternatives for provenance analysis or experiment user-steering at runtime. So, even supporting high-performance computation and manipulation of large volumes of data, it lacks in providing the user the capacity to steer experiments on-the-fly or gather and analyze partial results.

Like Vistrails and Kepler, Taverna [24] also focuses on usability, provenance records and semantics. It is an open source Java-based SWfMS, which directs workflows that leverage bioinformatics based web-services. Consequently, taking advantage of its features, over Taverna one can lay Biowep [28], a web-based client application that allows for the selection and execution of a set of Kepler-predefined workflows. Like CrowdLabs, this system is available on-line. Despite the user interface, the workflow conception (Manager) and execution (Executor) are accomplished within dedicated systems - Taverna or BioWMS [36]. Both are capable to handle the creation and annotation of workflows, and later its execution. For communication between BioWep and Workflow Managers, workflows models and results are carried in and out by agents like FreeFluo[1] or BioAgent/Hermes (BioWep agent). Although this application simplifies access of predefined workflows for bioinformatic researchers that may share experiments within a centralized repository, it lacks by not integrating the workflows engine inside the process. Nevertheless, BioWep also focuses on reproducibility by providing: access management and profiling; a web interface where workflows can be indexed and searched through their annotations; and the ability to save final results.

Swift [25] and Pegasus [8] focus on a strong distributed execution with provenance but have no runtime provenance support. These SWfMS usually use scripting languages and configuration files to describe the scientific workflows, unlike local SWfMS, offering graphical modeling. In this context, over these SWfMS, scientists

[1]http://freefluo.sourceforge.net/

may apply for SGSs like Pegasus Portal [29] and Swift Portal [30] improving scientists experience - but not eliminating their SWfMS-related limitations. The first one is a web-based portal for submitting workflows to the Grid using Pegasus SWfMS. This web application includes components for generating abstract workflows based on a metadata description of the desired data products and application-specific services. The second, Swift Portal, goes further not only by running scientific simulations, but also enabling data analysis and visualization through their web browsers. Swift Portal claims that its framework enables SGSs' developers to import their domain-specific scientific workflow scripts and generate Web 2.0 gadgets for running these embed workflows and visualizing its final results without writing any web related code.

1.3 Problem Definition

Scientific Workflow Management Systems (SWfMS) may improve data management on scientific workflows [2]. They make it easier to represent and reference workflow data through provenance [8]. Provenance is a key feature in SWfMS since it allows for keeping track of everything that happens during workflow execution [9, 10] (see Section 2). Therefore, once data is accessible during execution, scientists should be able to submit high-level and domain-specific provenance queries like: "What are the maximum values for velocity and pressure on a given simulation exploration?" or "According to a simulation exploration, the residual values of pressure are increasing or decreasing?" looking for support while steering the workflow [37]. Provenance data, if available at runtime, enables a powerful association between workflow metadata and strategic workflow results. With provenance support, it is possible to aggregate different types of metadata to the workflow results, making it easier to analyze and draw conclusions from data [38]. In this way, scientists can discover, as soon as possible, any need of changes in the workflow configuration, model, or input data - as opposed to waiting until the end of the whole workflow execution.

However, as we found in our studies, all related works do not allow for scientists to make such provenance-enriched analysis during runtime, preventing them from *e.g.* following workflow execution and checking if something went wrong (see Figure 1.2 for example). Works such [31], lack of integrated tools for analysis and level traversing, or just compromise generated provenance with decoupled implementations - even when runtime interference is allowed. If researchers could monitor time-consuming workflow execution at particular simulation exploration points; analyze enriched provenance data at runtime and decide to stop, re-execute, or re-parameter activities; a lot of energies would be saved - *e.g.* in one of our experiences, the user

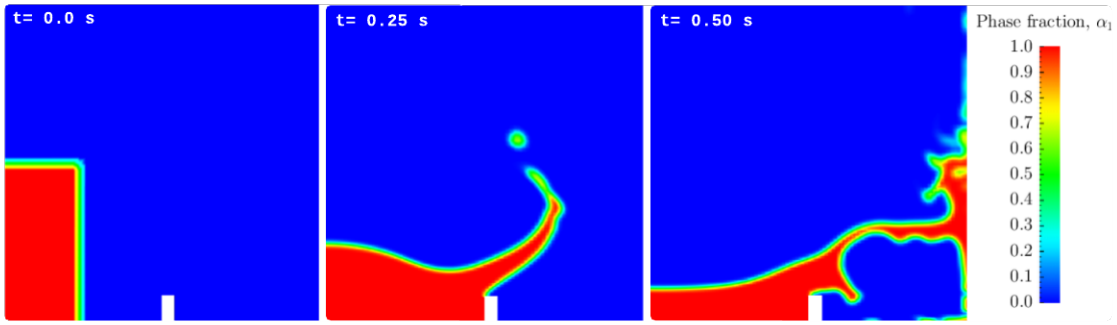

Figure 1.2: Example of a CFD simulation visualization, in which domain-specific queries can be applied. What are the maximum values for velocity and pressure on a given CFD simulation at *0.25s*? (Adapted from [39])

was able to interrupt and replan an execution saving 2.5 to 24 days, compared to non-interactive execution [40]. However, most of the systems execute workflows "offline" and do not allow for runtime analysis and workflow steering even when integrated within well-established Science Gateways Systems.

1.4 Goals and Approach

Finally, to address the problems stated, we propose a Science Gateway System (SGS) framework designed to support the scientific workflow lifecycle by using runtime provenance-enriched data, that also promotes features present at relevant related works, such as Reproducibility.

Support Workflow Lifecycle

This web-based framework, named Proteus, may assist the development and integration of new applications, that should work above it by integrating with its API and database. This database should handle all experiment-related data absent at provenance, among connection settings - for provenance, remote execution or visualization environments; user and groups' permissions; or SWfMS versions applied.

Once docked on Proteus' platform and with proper permissions, modules should be able to accomplish tasks involved in the experiment conception, submission, or analysis by taking advantage of a transparent provenance-layer connection available for runtime features. Security may be provided by a Permission Manager layer, which should control the access to each docked application and intercept all requests from the Web Interface. Figure 1.3 presents a sketch of Proteus architecture and its interaction with users along the experiment lifecycle.

Moreover, since SWfMS are in continual improvement process, Proteus may also take control of their change release, logging the specific version (*e.g.* by git commit

hash) applied for each experiment. This feature should help developers on deploying a new version at remote environments, and keeping track of a particular SWfMS's release execution, evaluating its performance.

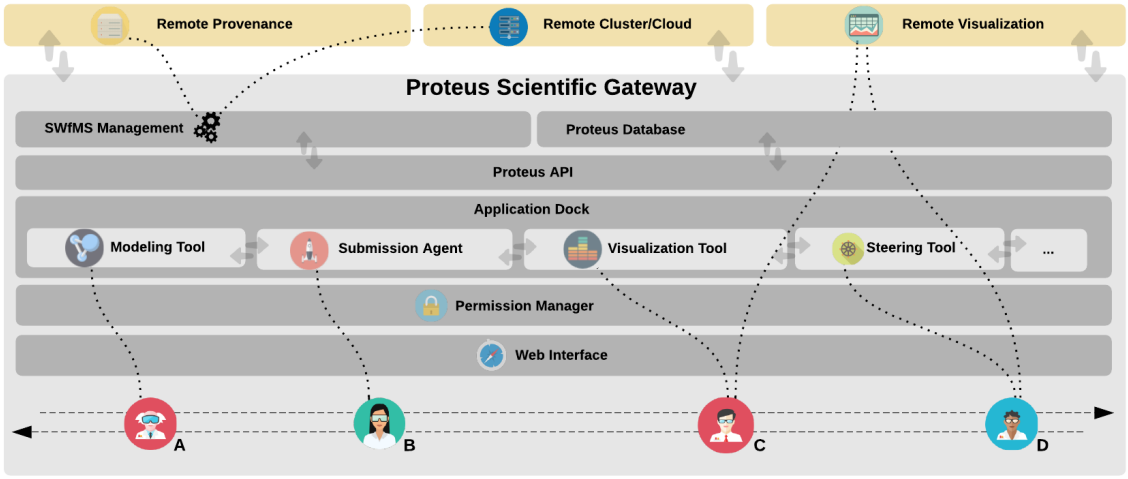

Figure 1.3: Proteus architecture overview. Scientists may interact along experiment lifecycle.

Wrap proper SWfMSs for runtime analysis

Proteus is proposed by integrating with Chiron [11] or SciCumulus' [12] generated provenances. Both solutions allow for *in tandem* execution and enriched data analysis brought by provenance data queries. As we describe later in more details, Chiron [11] presents a data-centric, algebraic based parallel SWfMS. Although Chiron was designed to work in clusters, SciCumulus [12] extend this work with similar features, by enabling cloud execution also considering specificities such as price and other impact factors. Both solutions have a unique native distributed provenance mechanism that enables runtime queries in a relational database, which is the key to our approach.

It is important to stress that both Chiron and SciCumulus, although providing runtime data analysis given by provenance - which fills the related works' unmanaged gap - they do not offer any analysis tool, despite the Relational Database Management Systems (RDBMS) applied over generated provenance. By now, they lack in assisting users along workflow lifecycle, not only while traversing provenance data, but also while managing complicated XML and script files applied for experiment configuration and execution. So far, both solutions work as many other SWfMSs and have not yet been used together with any SGS.

Promote Reproducibility

Proteus might also promote reproducibility, given such importance [16, 35] and application within almost all related works. Inspired by CrowdLabs [27], our solution may adopt a similar model used by social Web sites, combining a set of usable tools to foster collaboration. These functionalities may vary among access management and granular permission control of provenance related data (*e.g.* connections, workflow models or analysis recipes). Offering to scientists the opportunity to share any experiment layer, from the conception model until partial or final results by only assigning permission for a given user or group. Features like these could fit well if sharing a particular experiment credential access within a printed or online publication. Thus, with dedicated modules working above Proteus and given permissions, scientists shall be able to: (re)dispatch the experiment in their own environment; and/or analyze experiments by navigating through workflow metadata, selecting data of interest; and finally visualize provenance-enriched results during runtime or later (Figure 1.4).

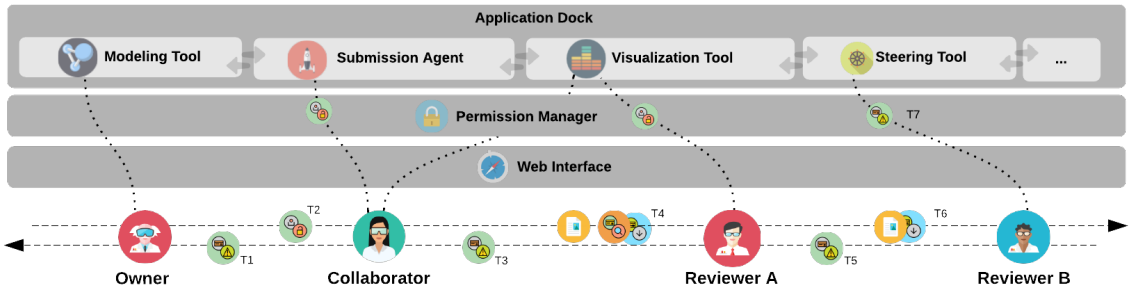

Figure 1.4: Example of a reprocibility interaction, regading a publication workflow.

Docking our first application: Provenance Visualization

Our first application tackle part of visualization needs that we will discuss later in Chapter 3. Laying above Proteus' infrastructure, this application, named Prov-vis, may provide an efficient data filtering and selection; allowing users for subsequent consolidation and data staging of relevant data needed during final or preliminary analysis conclusion. So that, after selecting relevant data, scientists may be able to make use of specific statistical or visualization tools by applying scripts *in situ*. Then, empowering results with provenance data without making costly and laborious data transfers.

Accordingly, to evaluate if Proteus' architecture could suitable port this new application, we integrate a new application dedicated to enhancing experiment's visualization based on features we believe are needed to assist an Uncertainty Quantification (UQ) scientific workflow exploration, with a Stochastic Analysis of sediment

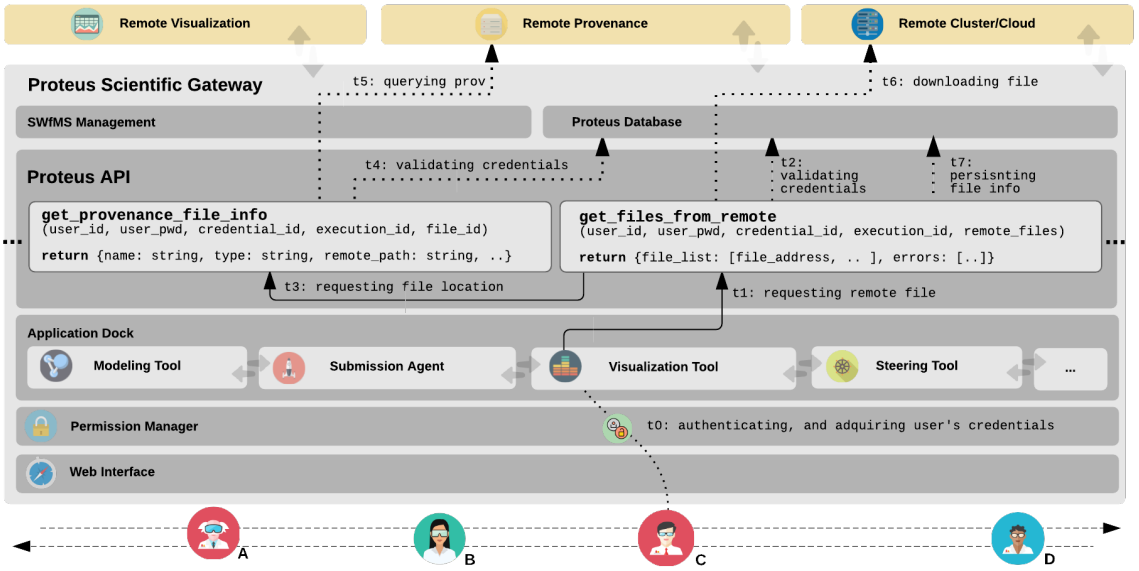

Figure 1.5: Prov-Vis architecture overview, regarding user interaction.

deposition resulted from a turbidity current (see Section 3.3). Our results show that we could explore experiment's partial data *in tandem* with the execution, by traversing and querying provenance; and staging post-processing visualizations that could help us to take decisions whether to stop or not the experiment execution. Figure 1.6 shows an example of post-processing actions during a Stochastic Analysis, when scientists could be able to take decisions *e.g.* among standard deviation and mean of employed experiment's settling velocities during several sparse grid's levels.

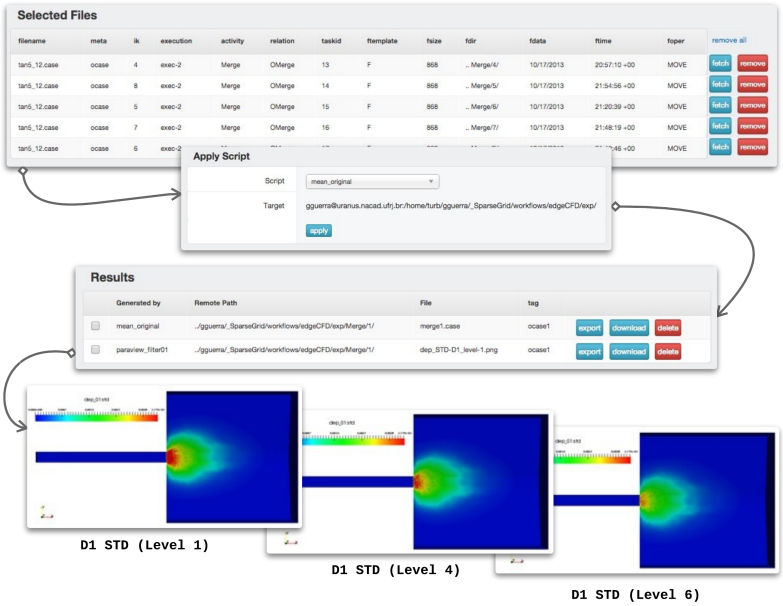

Figure 1.6: Example of post-processed UQ experiment's data generated by Paraview's filters at execution machine.

1.5 Organization

This writing is organized as follows: Chapter 2 provides information about the scientific workflows in much more details, explaining more about the workflow engines, the provenance, and the whole workflow lifecycle according to related works' analysis. In this same chapter, we also present the current scenario for large-scale experiments. The following chapter focuses on our Visualization application, making an overview of Provenance Visualization at some aspects of User-Steering and *in situ* visualization techniques. Additionally, in this chapter we discuss the importance of Uncertainty Quantification within computer science applications, also introducing our case study and its related works. Next, in Chapter 4, we design the whole solution, providing the methodology and technologies applied, also describing the challenges that we found along the way. Finally, in Chapter 5, evaluate Proteus and the visualization application we built. At last, in the final chapter, we conclude this work by summarizing the contributions brought by Proteus and discussing the future works that could be related.

Chapter 2

Scientific Workflow Lifecycle

The lifecycle of a large-scale scientific experiment can be divided into three phases: modeling, execution and analysis [1]. In the composition, scientist defines experimental procedures on an abstract level, modeling the process performed to verify a certain hypothesis. During the implementation phase, the scientist sets up the experimental process defining the applications, scripts - any activity - that will run, the order in which they will be implemented and how they should be configured to run. In the analysis phase, the results obtained in the implementation phase and the way they were hit are analyzed in order to evaluate the hypothesis studied. The result of this analysis can be conclusive or used to feed back the lifecycle generating a new iteration (round) of the experiment. Figure 2.1 shows a graphical diagram of the lifecycle according to MATTOSO *et al.* in [1]. In the following sections will be explained in more detail each phase of the lifecycle of scientific experiment.

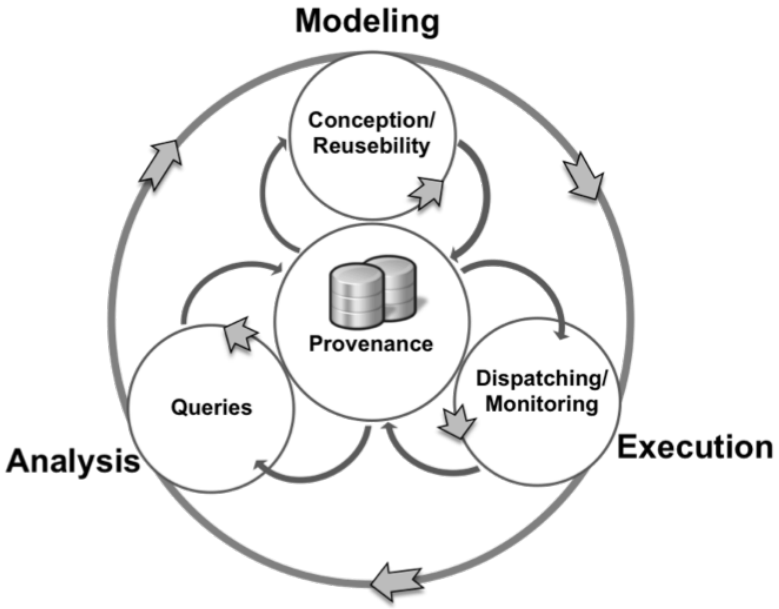

Figure 2.1: Scientific workflow lifecycle according to MATTOSO *et al.* [1]

2.1 Composition

The composition is the step where the experiment is structured and configured (maybe not for the first time), establishing a logical order between its activities. Activity is an abstract concept that defines an action in the context of an experiment; it can be performed either by a computer or a scientist, for example. In this manner, an activity can have different types of input data or parameters, generating different types of output data. While concepting an experiment, scientists define each activity, and its types of input and output, as they compose their experiments. The logical order that defines the sequence in which the activities need to be performed during execution consists in a scientific workflow of that experiment. This scientific workflow may translate the workflow process of a given experiment. The structure of the workflow promotes a consistent and unified view of the process making it easier to maintain and modify. Figure 2.2 shows an example of a scientific workflow. In the figure, each gray rectangle represents an activity. The first activity accounts for pre-processing while the second is the kernel to be run and the third and fourth are the post-processing.

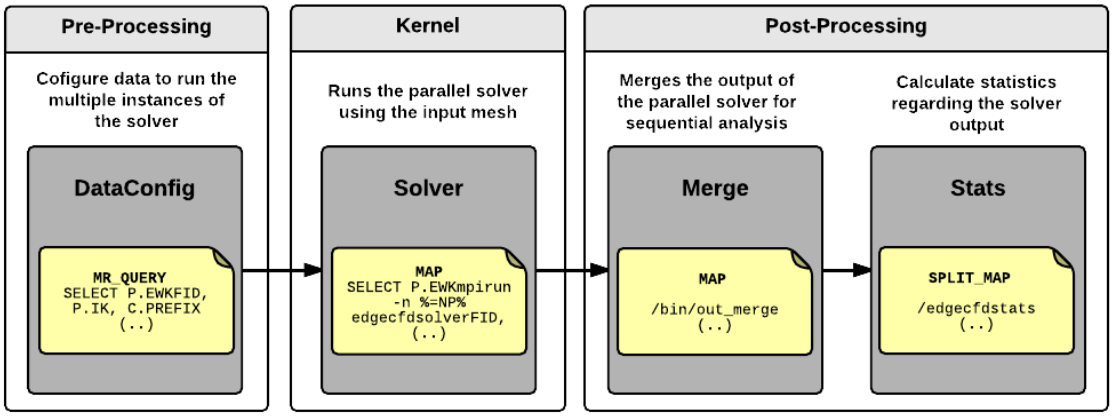

Figure 2.2: Example of a UQ scientific workflow.

A new workflow can be created from scratch or born from an extension of another one. Often, scientists reuse workflows parts or extend workflows from one experiment to another. From this point of view, the workflows of the experiments can be seen as reusable components. The phase of composition, therefore, can be divided into two stages: design and reuse.

The design of the experiment is business workflow modeling of a given experiment, which is the scientific experiment protocol, i.e., the procedure adopted to execute it. From this more conceptual workflow, one can obtain an abstract workflow that add information on which computer programs or procedures will perform each activity. In this way, with an abstract workflow you can have a concrete workflow, which is a ready-to-run workflow that adds the information given by each comput-

ing resource (machine) that performed an activity. Figure 2.2 is an example of an abstract workflow. However, the notes preprocessing, kernel, and post-processing information are inherent to the conceptual workflow experiment.

Another composition's aspect is the reusability. Reuse has an important role in the lifecycle of an experiment. It allows a new experiment to be created based on another one already implemented. According to the results of an analysis of an experiment, it may be desirable to modify the initial workflow or make small changes that add some differential in a new experiment activity or parameter. The reusability aims to design a new workflow from other workflows already modeled. Thus, an alternative is to obtain an abstract or concrete workflow from a particular conceptual one if it is, i.e., in the shape of an "experiment line" [41]."Experiment Line" is a concept inspired by techniques used by software production lines [42, 43]. This concept allows us to transform the initial conceptual workflow of an experiment in a reusable asset able to generate concrete workflows. Therefore, from a unique "experiment line" can derive different workflows since activities have variability - i.e. can be implemented by different procedures or settings - allowing different behaviors during execution.

2.2 Execution

From the concrete workflows obtained at the end of the composition phase, you can start the execution phase. The execution is responsible for performing the actual workflow on a workflow engine. The execution of a workflow involves performing each workflow activity respecting the dependency relationships between them. An activity "A" is dependent on another "B" if a data set input "A" is produced by "B". Relations of dependence ensure that the workflow will run respecting a particular order of activities. The purpose of a workflow execution is to produce the results that will be used in the next phase of analysis. The infrastructure used to run the workflow should be able to run the activities, to log the execution (write its history) and carry out the monitoring.

Some of the workflows activities, or the whole workflow itself, can be very costly to be performed sequentially. A single workflow activity can generate a significant number of instances involving different combinations of parameters. Each of these instances can be visualized as a workflow task. Therefore, it is very common for scientific workflows or part of them to run on distributed computing resources, such as clusters, grids, and clouds. Running in a distributed manner, monitoring the execution of these workflows is even more important, though also more complex. For each stretch of the activity, you must store data that was executed by which resource and how the execution took place. Therefore, the execution phase can be

divided into two stages: **distribution** and **monitoring**.

The workflow execution distribution is inherent in the workflow execution engine, and the running machines are associated with a particular scientific workflow management system (SWfMS). A SWfMS provides the construction of scientific workflows, interpret the workflow described in a particular language running it on a workflow engine. There are many types of SWfMS with different approaches to model or describe workflows and different methods for their executions.

Many SWfMS such as VisTrails [14] or the Kepler [2] are focused on the development and local execution of the workflow. They offer a graphical interface for visual modeling of workflows and propose components to create activities of various types, for example, activities that invoke a local program or a web service. Primarily, these SWfMS have execution machines for local scientific workflow, i.e. processing each activity from the scientist's machine. These systems allow remotely invoke an activity, such as a web service or running in a cluster or grid, however, remote execution's infrastructure is decoupled from the internal control SWfMS and must be pre-configured to be invoked by the manager. The focus of the execution of this type of SWfMS is the local execution and the semantic record of running workflows.

Other SWfMS are focused on the distribution of the workflow execution such as DAGman [44], Pegasus [8] and Swift [25]. These systems focus on a distributed execution but give little support to the semantic record concerning the activities of the workflow and usability through a graphical interface. These SWfMS usually use scripting languages and configuration files to describe the scientific workflows, unlike local SWfMS, offering graphical modeling. Distributed execution managers also usually depend on a grid computing oriented infrastructure to submit their distributed activities more efficiently. Although not a restriction, such systems take greater advantage if the submission of infrastructure uses a scheduler as the Condor [44] or the Falkon [45]. The Condor is a grid computing oriented scheduler. Whereas Falkon is a more a general scheduler, it needs (Falkon Project 2010) the installation of GT4 grid middleware [46] with an authority certification [47]. Such restrictions may be complicators in scenarios where the execution environment uses a system aimed to high-performance computing with traditional schedulers for clusters, such as Torque [48] rather than a grid infrastructure.

In order to unite the benefits of local SWfMS with the distributed execution in high-performance computing resources, solutions such as Hydra [49] and Chiron [11] were presented by OGASAWARA *et al.*.

Hydra is a middleware that, from a local SWfMS as the VisTrails, makes the distribution of an activity of the workflow in a distributed resource as, for example, a cluster. The Hydra parallels an activity following data parallelism stereotypes, such as input data fragmentation or parameters sweep. Data fragmentation involves the

division of a larger input file into smaller files that will be individually processed in parallel. Each generated task consume a fragment of the original file and produce a result; while the parameters sweep concerns the execution of a given program inside an activity using different combinations of parameter sets. Each task will execute the program with a different set of parameters generating (possibly) different results. In the end, the results of each task will be aggregated into a single result of the activity.

Chiron [11] can be considered as a new approach given Hydra limitations. It focuses on the parallel execution of the workflow while managing intermediate resulting data generated by the computations. Chiron coordinates the parallel execution, assigning different input parameter sets to computing nodes that run the workflow for each input in parallel. Chiron engine uses a data-centric approach, where a scientific workflow algebra handles the parallel workflow execution efficiently. Workflows in Chiron are defined through a declarative language (XML), which is transformed into an algebraic expression that enables automatic workflow optimization. The algebra also standardizes data consumption and production as algebraic operands, which allows Chiron to establish an optimized parallel execution plan for the workflow algebraic expression, based on provenance data collected at runtime, that is, during the workflow parallel execution. Moreover, provenance refers to registering all aspects of data generation. Through data provenance queries, it is possible to trace back how data is produced. However, the parallel execution of a workflow introduces several difficulties for collecting provenance data because these data are also distributed across the processing nodes of a high-performance machine.

The monitoring stage of the experiment execution cycle permeates the record of everything that was done and all that is currently running. Runtime, which feature performed which task, results, errors and exceptions that occur during the execution are examples of relevant information for the analysis phase. The feedback that tasks performed as expected, or that they have failed, is also essential to provide a quality service. However, few systems use this data to reschedule tasks automatically. Although the monitoring step is the simplest to explain, it is of great importance for the next phases.

When a workflow activity performs in a distributed manner, it may be difficult to treat the distributed monitoring tasks among execution machines. This management should take into account where data will be recorded being aware of possible concurrency problems while writing to disk or databases. The number of tasks running simultaneously is extremely high - maybe thousands could reach hundreds of thousands. So, monitoring the persistence individually for each task can bring overhead to the management of shared resources. Therefore, you must use efficient distributed algorithms, a hierarchical mechanism or well-defined roles in the

execution resources network to provide an effective access to storage resources.

2.3 Analysis

The scientific experiment analysis phase is associated with: the results obtained in the (previous) execution phase; and the call to the provenance layer [50]. The **prospective provenance** of the experiment records the decisions and parameters defined in the composition phase; while the **retrospective provenance** is the record of decisions and results that occurred during the experiment execution [9]. This collected information is vital to the experimental systematic process and fundamental to the analysis phase. The construction of the provenance base permeates the stages of composition and execution, and is critical to ensure the reproducibility of the experiment - as a way to save the history of what was done and how it was done throughout the experiment lifecycle.

After the execution of the scientific experiment, scientists analyze the results obtained to evaluate that they are satisfactory and sufficient to confirm or refute a given hypothesis. In addition to the results obtained directly in the execution, it is usual also to correlate results with decisions made in the composition phase and to assume relations with provenance data. All these procedures take place in the analysis phase of a scientific experiment.

The analysis may refute the hypothesis of the experiment in question. In this case, the scientist may reformulate the hypothesis and run a new experiment repeating the experiment lifecycle. Even if the examination confirms the experiment hypothesis, it not usual that the scientist state its trueness with complete certainty, as the experiment may be incorrect or may not have taken all the necessary factors into consideration. Therefore, although the results are relevant, such experiment must be reproducible, in order to other scientists run it again. In this scenario, other scientists may adjust the experiment seeking for new results corroborating the confirmation of the hypothesis, or else refuting what before had been confirmed [51].

The analysis process involves a number of provenance queries and results with possible graphical views to facilitate understanding of the set of obtained data. Thus, the analysis phase can be divided into two stages: **Query** and **Visualization/Discovery**.

The **query** and **visualization** process in the experiment lifecycle are related to the study of results - obtained in the execution phase - and the provenance data, in order to achieve the study completion. In other words, this is when scientists assess the experiment hypothesis. There can be applied several methods while querying experiment results and provenance data such as preliminary, analysis or follow-up queries [1, 52]. Through provenance queries, it is already possible to reach conclu-

sions about the experiment. However, due to the large volume of data generated in large-scale experiments, it is common to use scientific visualization techniques to perform better analysis of the dataset and its features. Through graphics, maps, images and videos, scientists can have a whole view of behaviors, trends and general aspects of results.

The discovery stage inside an experiment lifecycle is strongly associated with the provenance base. It is a process that will look for patterns and characteristics traversing data and its relationships inside provenance. Examples of queries concerning the discovery about provenance are [1]: "How many different concrete workflows used the X version of a legacy application in a given experiment?", "How many processors were used in parallel execution or workflow Z ?", "How many workflows were used in experiment Y ?".

One problem concerning the discovery stage is the diversity of provenance data persistence approaches that covers different sort of query languages. Recently, the community has been trying to make the open source provenance model (Open Provenance Model or OPM) [53] a standard, but few solutions take the model natively. Some SWfMS allow you to import and export their provenance data in the OPM's format. However, the exchange of data between different SWfMS by exporting and importing a OPM format is still a challenge. Also, there is no consensus about the best search language on provenance data, not least because the language depends (in general) the way in which data is stored. For instance, if the database is relational, one can use a language such as SQL. However, if the base is writen in XML, some languages such as XQuery [54] must be applied.

2.4 Current Scenario

2.4.1 Scientific Workflows in HPC

Within a scientific experiment lifecycle, workflow parallelism strategies are more related to the execution phase. The scheduling of a workflow means that a given activity was scheduled to run - e.g. in an external resource such as a cluster or grid. The parallelization of the workflow means that the execution of one or more activities this workflow is done in parallel to speed up the process. However, even when an activity is scaled to run on external computer resources, the control over the workflow execution is still held by the SWfMS. There are a several approaches to parallelism or remote execution of workflows activities, according to SWfMS policy. But, we will stick to describe how Chiron [11] approaches and affect our solution.

As we could see in the previous sections, scientific experiments *in silico* are often modeled as scientific workflows and managed by SWfMS. Each SWfMS has its

particular characteristics, notation, and language. They focus on different resources such as scientific visualization, provenance or parallel execution - reasoning why when a workflow is modeled in a given SWfMS, the workflow is dependent on issues relating to that SWfMS technology.

In addition, the representation in a SWfMS, in particular, does not clarify the knowledge behind the workflow model. During an experiment, when scientists need to perform some activity with higher performance, they often call for changing the approach. Thus, they are used to re-model the workflow within a new SWfMS, and probably modify the source code of the activity to achieve the desired level of parallelism, in order to improve performance. However, remodeling a workflow requires effort and can also generate errors. Scientists may not be familiar with programming languages and parallelization methods. For these reasons, SWfMS such as Chiron [11]; Swift [25] and Pegasus [8] believe that the parallel execution of workflows should be made implicitly.

An implicit approach attempts to execute scientist workflow in a parallel way, but with more transparency - without modifying the applications. It means that the approach must be non-invasive, so it does not change the application source code. This fact is very important since many scientific applications have complex and legacy code [55], which is very costly to be modified. Applications can also be proprietary, which means that the scientist does not have access to the source code. But even without accessing the code, you can still run these applications in parallel using, for example, the "bag of tasks" model [56]. This model is related to the execution of many tasks - usually decoupled - in distributed resources. However, performing these tasks does not value the high-performance but the high-flow of execution. Solving this, RAICU *et al.* [45] stands a new computing paradigm - called Many-Task Computing (MTC) - that meets the execution of many tasks comprising the "bag of tasks" model and values the performance in the execution. Therefore, this paradigm permeates between the high-performance computing (HPC) and high flow computation (High Throughput Computing or HTC).

Chiron [11] is a MTC oriented solution. But others studies, such Swift/Falcon [45, 57], or Hydra [58] (a Chiron's precursor) have also explored the MTC features into their solutions. Among them, another popular approach is the MapReduce programming model. This model is explored in scheduling workflows as a particular case of data parallelism. Primarily, a large data set is broken down into smaller pieces that are mapped to be processed in the processing nodes. These pieces are directed to the fragmentation function (Splitter) or Map function as pairs of (key, value). After the Map function is executed, the intermediate values for a given output key are aggregated into a list for an aggregate function (called Reduce). The Reduce function combines the intermediate values in one or more final results related

to a given output key. Although, the application of MapReduce can successfully be applied in MTC problems, its main framework Hadoop [59] and its workflows related applications [60, 61] don't address all SWfMS needs.

Finally, as we can see, even with all the improvements made in recent years in regard to the applications of scientific workflows in HPC, there are still many issues that can strike an experiment execution that relies on distributed resources. One could list these solutions may present: (i) the same problems of parallel programs; (ii) the volatility of computational resources; (iii) the failures occurrences at runtime; and (iv) the difficulties in debugging these failures.

2.4.2 Large-Scale Experiments Scenario

Frequently scientists are tasked with addressing challenging problems in energy, the environment, or national security. Addressing these challenges requires simulation of complex multiscale, multiphysics phenomena and may also involve mathematical optimization and uncertainty quantification to answer broader design and decision questions. Problems like these use to call for large-scale experiments.

However, even with today's mathematical algorithms and petaflop supercomputers, many extreme-scale science problems remain hard to deal with. Among them, we could list: power (where it goes for sustainability); extreme concurrency (power density limitations); limited memory (not scaling as fast as processors); resilience (hardware failure), and data locality (limited memory bandwith).

In preparation for exascale systems, in 2013, the U.S. Department of Energy (DOE) Office of Science Advance Scientific Computing Research (ASCR) program has sponsored a series of workshops leading to comprehensive reports on many of these challenges and opportunities [62]. Yet, in this work we decided to focus on additional problems that - before scaling this far - even in a dozen scale experiment can be noticed.

Time-consuming experiments, like those applied in Computational Fluid Dynamics (CFD) analysis - such as most of DOE addressing problems - present several complex and long activities that, given their black-box behavior, let scientist away of the execution. Each activity involve a execution of programs that can run for weeks straight without notifying the scientist about decisions or results. Consequently, as a typical result of this kind of process, when something goes wrong with the execution, time and money are thrown away.

A typical computational fluid dynamics experiment execution approaches the same basic procedure as the followed [63]:

1. During preprocessing
    1.1. The geometry (physical bounds) of the problem is defined.
    1.2. The volume occupied by the fluid is divided into discrete cells (the mesh). The mesh may be uniform or non-uniform.
    1.3. The physical modeling is defined – for example, the equations of motion + enthalpy + radiation + species conservation
    1.4. Boundary conditions are defined. This involves specifying the fluid behaviour and properties at the boundaries of the problem. For transient problems, the initial conditions are also defined.
2. The simulation is started and the equations are solved iteratively as a steady-state or transient.
3. Finally a postprocessor is used for the analysis and visualization of the resulting solution.

Such experiments can involve multiple explorations with various parameter sets. Furthermore, each activity often needs high-efficient and enormous computational resources concurrently, which often can be executed independently. These requirements explains why CFD applications often take a great advantage of distributed environments with low-latency traffic - such as clusters or grids - dramatically reducing the overall time spent during execution. But as large it goes, so its impact does; turning experiments hard to follow during execution, often behaving as a black box. Moreover, such large-experiments present greater data heterogeneity and granularity given how we found them among distributed resources. Consequently, to visualize their partial results are much harder, mostly because of the effort to perform the traceability of which input produces each output; later, the fact that there data and meta-data are poor connected does not help with the experiment-result association; and finally, when results are found, data transfers are challenging. The next chapter may illustrate these problems along with our Uncertainty Quantification case study.

Chapter 3

Provenance Visualization

This chapter delineates the core of Prov-Vis, our visualization module. Here, we ground Prov-Vis strategy plan and requirements while studying two important surveys published in the last two years. The first one, by MATTOSO *et al.*, brings a discussion about research issues related to scientists' steering in HPC workflows [64] leveraging steering methods such as user interference guided by provenance visualization. In the second one [65], MORELAND presents a literature review of the most prevalent features of traditional visualization pipelines. In his work, the author reveals an interesting parallel between scientific workflows and visualization pipelines; later, ranking provenance as one of the emerging features of this area of study.

3.1 Assisting User-Steering in HPC workflows

For numerical simulations, scientists require snapshots of current simulation results and the possibility to refine their model during runtime [66]. In the Bioinformatics domain, scientists commonly require the possibility to query and traverse partial results and change workflow activities or parameters during the execution [67]. While in the Oil & Gas domain, engineers need to explore the large parameter space in slices, by skipping input data dynamically. Considering this, MATTOSO *et al.* [64] - while discussing research issues related to scientists' steering - bring to us three main issues related to the visualization and user-steering in scientific workflows: monitoring of execution, data analysis at runtime, and dynamic interference in the execution (Figure 3.1).

While monitoring workflow execution, scientists verify its status at particular points looking for issues and assure the experiment finish. Therefore, based on the monitoring results, scientists can decide if it is already possible to analyze specific partial results. Another possibility comes from stopping or re-executing some ac-

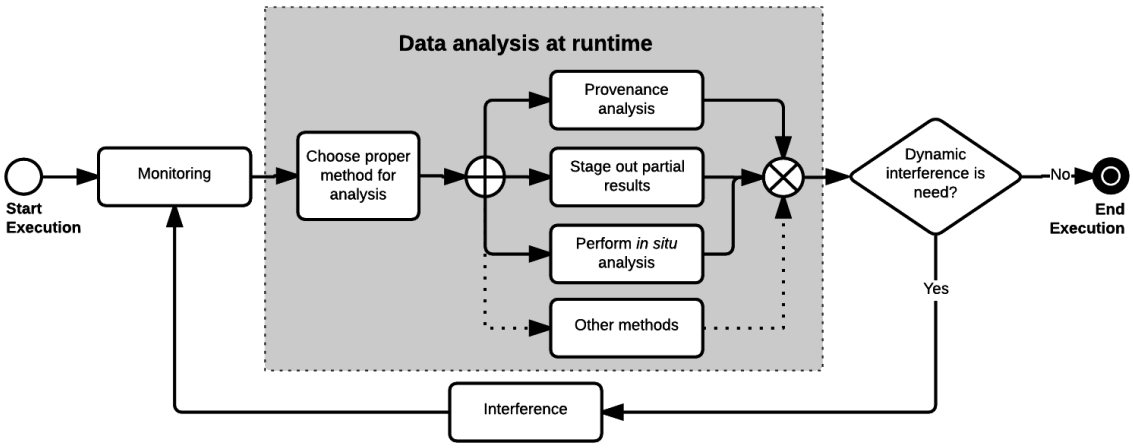

Figure 3.1: Steps related to the visualization and user-steering in scientific workflows: monitoring of execution, data analysis at runtime, and dynamic interference in the execution.

tivities or even the entire workflow. Thus, if they need to analyze partial results, it is important to have tools to handle data staging, consolidation, statistics, and visualization. Succeeding, based on the partial analysis brought by data analysis, scientists shall change parameter values - or even programs - of the workflow during its execution, performing dynamic interventions in the workflow specification.

In MATTOSO *et al.* [64], authors support that scientific workflow steering is strongly related to the dynamic issues. However, they also observe that the complete dynamic support is still a challenging research topic. Additionally, still not distant to GIL *et al.* [68] - which, by 2007 already stated that user-steering of workflows was a step towards dynamic workflows - MATTOSO *et al.* also hold that dynamic workflows include challenging features such as:

i. distributed and collaborative workflow design

ii. workflow adaptation based on external events such as human intervention

iii. an efficient query system to support information browsing and traversing and ways to explore slices of the parameter space, and comparing the results from different configurations

Regarding the monitoring stage of the workflow execution, it can also be leveraged by runtime provenance queries. However, querying provenance at runtime may not be enough. SWfMS may need to react automatically to what is being generated by the activity flow, looking for particular data and providing runtime provenance data analysis for such results. This analysis task is unviable to be hand-operated in

large-scale. Therefore, if existing SWfMS provide online mechanisms for scientists to smartly monitor, analyze and interfere in their workflows, scientists can benefit from such factor as described in [64]:

i. **Awareness:** once scientists are aware of the current status of their workflow execution they can plan actions to avoid failures, anticipate problems and improve results with refinements and fine-tuning;

ii. **Debugging:** if scientists can identify which activity failed in which machine and relate it to provenance data, it might be simpler to solve the problem;

iii. **Completion:** by analyzing provenance data, scientists can analyze failed activities and propose starting other replacement activities, or changing parameter values; and

iv. **Efficiency:** by steering, workflow scientists can reduce the time spent in processing low-quality or even irrelevant data and cut the total execution time.

In this work, we believe that the improvement of such factors is intrinsically related to the visualization features applied during the experiment lifecycle.

3.2 Visualization Pipelines

In [65], MORELAND reviews the visualization pipeline. Beginning with a basic description of what the visualization pipeline is and moving to advancements introduced over the years and current research. The author states that a visualization pipeline comprises a dataflow network in which computation is described as a collection of executable modules that are connected in a directed graph representing how data moves between modules - which can be understood within these three modules:

i. **sources:** A source module produces data that it makes available through an output. File readers and synthetic data generators are typical source modules

ii. **sinks:** A sink module accepts data through an input and performs an operation with no further result (as far as the pipeline is concerned). Typical sinks are file writers and rendering modules that provide images to a user interface

iii. **filters:** A filter module has at least one input from which it transforms data and provides results through at least one output.

According to MORELAND, the goal of this representation is to encapsulate algorithms in interchangeable modules such as source, filter, and sink; with generic connection ports (inputs and outputs). In this way, an output from one module can be connected to the input from another such that the results of one algorithm become the inputs to another algorithm. These connected modules form a pipeline. Moreover, we could parallel the same representation as a workflow, where each module represents an activity connected by its input and output relations (Figure 3.2).

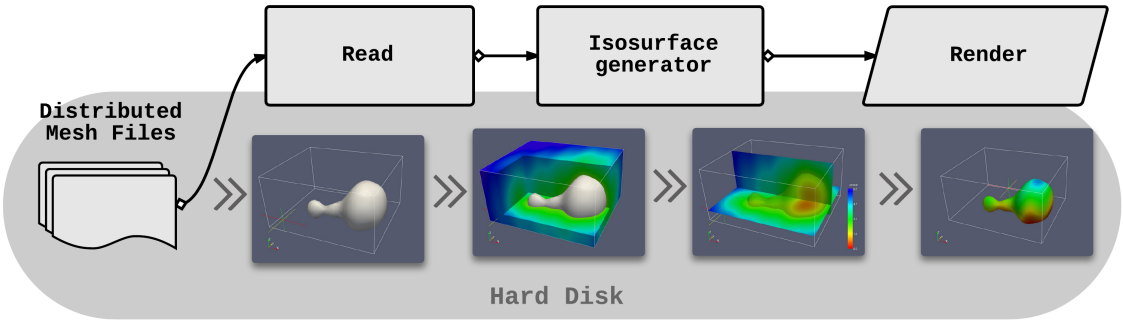

Figure 3.2: Example of a simple visualization pipeline featuring a file reader (source), an isosurface generator (filter), and an image renderer (sink). It is important to note that other filters could be applied, such as streamline, reflect, glyph, tube, etc. This image is an adaptation from [65, 69].

3.2.1 Using Provenance

Although MORELAND [65] considered - along almost his entire work - the visualization pipeline as a static construct that transforms data, he also contemplates provenance as one of the emerging features of visualization pipelines. Like other authors [37, 64, 70] that value the importance of user-steering for scientific exploration, MORELAND - as a Computational Fluid Dynamics (CFD) visualization specialist - states that in real visualization applications, the exploratory process also involves making changes to the visualization pipeline (e.g., adding and removing modules or making parameter changes). This fact leads us that leveraging runtime modifications - and proper recording - might enable scientists to model exploration as transformations to the visualization pipeline [71], promoting advancements in areas of study such as CFD.

There are many ways in that provenance of pipelines transformations can assist exploratory visualization. Among them, we can list:

i. allowing users to fast explore **multiple visualization** methods and compare various parameter changes [72];

ii. supporting **reproducibility** while recording the steps required to achieve a particular visualization. Thus, the user could save the process to automate the same visualization later [16];

iii. provenance information helps to **perform analysis of analysis**, such as:

   (a) **comparing and combining** provenance information to provide revisioning information for collaborative analysis tasks [73]

   (b) revisiting previous analyses to give scientists the ability to **learn with past experiences** for future exploration. Thus, data can be queried for proper visualization pipelines [74] or employed to support users in their exploratory process **automating decisions** [75].

3.2.2 *In situ*

In situ visualization is an old concept [76], but its application has been growing fast in late years. This concept is based on running the simulation while visualizating the results simultaneosly. There are many approaches to in situ visualization. While some approaches directly share memory space; (i) others share data through high-speed message passing (ii) [65]. In one way or another, all *in situ* visualization systems have two important characteristics: visualization runs *in tandem* with the simulation that is generating the results, bypassing the expensive step of writing to or reading from a file on disk storage.

The cost of dedicated interactive visualization computers is increasing [77] and the time spent in `I/O` is beginning to govern the time spent in both the simulation and visualization [78]. Given this fact, the concern in *in situ* visualization has been virtually growing in late years. And consequently, becoming one of the most important research topics in large-scale visualization today.

Even when *in situ* visualization does not really designate to visualization pipelines, many current projects use visualization pipelines for this purpose. Visualization pipeline's flexibility and the large number of existing implementations [6, 79–81] may be the answer. As far as we know, any visualization architecture can be coupled with a simulation and take part of a scientific workflow execution.

3.2.3 *In transit*

A popular form of *in situ* visualization is known as *in transit*. This particular approach is given when *in situ* visualization exploits an `I/O` transport infrastructure that addresses the disk transfer limitations of modern supercomputers. It is well known that the computing rate of a modern supercomputer surpass its disk transfer. However, recent studies have shown that the latency of the disk storage can be hidden by having a "staging" job running separately but concurrently with the main computation job. This job manage to buffer `I/O` transferences, and rather than writing it straight to the disk, performs *in transit* analysis and visualization [7].

3.3 Assisting Uncertainty Quantification

Uncertainty Quantification (UQ) is a general term for a diversity of methodologies, including uncertainty characterization and propagation; parameter estimation and model calibration; and error estimation [82–84]. All these activities address the problem of discovering how ubiquitous uncertainties, in all modeling efforts affect the predictions and understanding of a complex event. The reasons for such problems can include both epistemic (lack of knowledge) and aleatoric (intrinsic variability) uncertainties; which surround uncertainty coming from inaccurate physical measurements, bias in mathematical descriptions, as well as errors coming from numerical approximations of computational simulations [85]. Trying to solve this problem is crucial while dealing with realistic experimental data and judging the reliability of predictions based on numerical simulations. Although UQ has always been an old challenge, recently, advanced research approaches investigating this problem hardly, driven by the massive increase in the number of computing units, such as our path in a few years to the exascale computing [85].

Moreover, according to [85], motivating science applications for UQ investigation involve systems that describe physical and biological processes exhibiting highly nonlinear; or discontinuous; or bifurcating phenomena at a diverse set of length and time scales. For such cases, applied for many significant problems, simulating an entire complex system at the level of resolution necessary to represent such behaviors accurately is extremely challenging. Furthermore, a predictive simulation of these systems requires significantly more computational effort than deterministic high-fidelity simulations do - particularly, cases which approximations and input data (coefficients, boundary conditions, geometry, etc.) are affected by a large amount of uncertainty, such as climate modeling [86]. Thus, to justify a predictive capability by mathematically and scientifically rigorous methods, it is essential that scientists

exhaustively apply tasks such as simulation code and calculation verification; model calibration; validation and bias correction; and a complete quantification of all uncertainties; until discover a satisfactory convergence condition (see Figure 3.3).

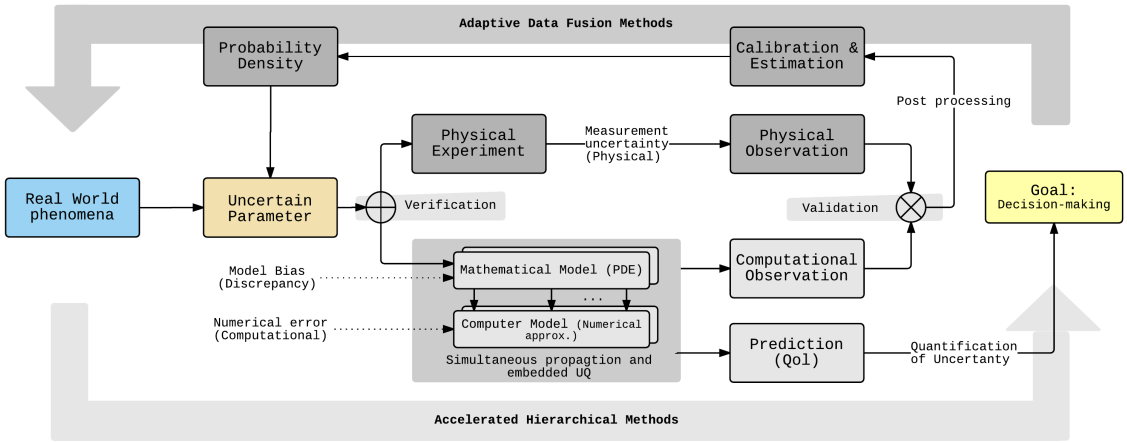

Figure 3.3: Illustration of the capability of HPC to assimilate predictions from computational simulations and data from physical experiments (adapted from [85])

3.3.1 A Provenance Approach

As we mentioned before, computational simulation of complex engineered systems requires intensive computation and a significant amount of data management, often involving a considerable number of processing units, which may lead us to UQ investigations. Regarding this fact, in [66], GUERRA *et al.* apply SWfMS techniques while exploring such problem. In this work, authors state that parameter variability could be inserted in the general context of UQ, providing a rational perspective for analysts and decision makers. For this reason, [66] used scientific workflows to provide a systematic approach while: (i) applying Chiron [11] modeling UQ numerical experiments as scientific workflows, (ii) offering query tools to evaluate UQ processes at runtime, (iii) managing the UQ analysis, and (iv) managing UQ in parallel executions. As results, it was possible, by instrumenting provenance, to collect data in a transparent manner, allowing execution steering; the post assessment of results; and providing the information for re-executing the experiment until convergence.

3.3.2 Our Case Study

As we could see, Uncertainty Quantification (UQ) is a general term for a diversity of methodologies, including uncertainty characterization and propagation, parameter estimation and model calibration, and error estimation; it provides a rational framework by combining sophisticated computational models with a

probabilistic perspective in order to deepen the knowledge about the physics of the problem and to access the reliability of the results obtained with numerical simulations. Laying in this subject, our case study performs a Stochastic Analysis of sediment deposition resulted from a turbidity current (see Figure 3.4). This study considers uncertainties on the initial sediment concentrations and particles settling velocities. In this experiment, also created by GUERRA *et al.*, the statistical moments of the deposition mapping, like other important features of the currents, are approximated by a Sparse Grid Stochastic Collocation method that employs a parallel flow solver for the solution of the deterministic problems associated to the grid points. In this work, as in [66], the whole procedure is supported and steered by Chiron [11]; and persisted at Chiron's provenance.

Figure 3.4: Flow dynamics of our sediment deposition experiment.

So far, even though this work was accepted for publication, it was not published yet. Given this, we are not going further at this work's sensitive decisions. Instead, we are going to work above the available provenance and data generated by its experiment's execution. Also, for abbreviation, we named such Sparse Grid Stochastic Collocation experiment by the acronym **SGSC**.

The following paragraphs, which describe how Chiron worked by generating provenance data at SGSC; and how users behaved during this process, may help us to elaborate our first Proteus' application (at Chapter 5) that will aim at providing better tools for this experiment visualization. Therefore, some possible **(I)**mprovements are raised.

At Section 2.2, we could see that Chiron focuses on the parallel execution of a workflow while managing intermediate resulting data generated by the computations. In the same way, at SGSC, scientists monitored the parallel execution of all deterministic parallel simulations - corresponding to 1073 points of the sparse grids for increasing levels (see Figure 3.5) - using Chiron to manage the whole execution process; assigning different input parameter sets to computing nodes that ran the workflow for each input. Each deterministic run used 12 cores. This two-level parallel approach (Chiron and the applied Solver) allocated 120 cores in total, submitting 108 parallel jobs. Sequentially, the total execution time would be around 202.4 days, but since each ran in parallel, it took only 23.6 days. Further, each execution took, on average, 5:30 hours; and allocated 618Mb of disk space (663.1Gb total). Computer resources were provided by NACAD's cluster machines.

Figure 3.5: Illustration of SGSC's sparse grid at level 6

Since provenance refers to registering all aspects of data generation, in this experiment, through data provenance queries, it was possible to trace back how data was produced; allowing scientists to make the proposed analysis. This process could be accomplish by connecting at provenance and **applying hardcoded SQL queries for each request [I1]** while monitoring SGSC's evolution running from 1 to 6 sparse grids' levels (see Table 3.1). Thus, once scientists knew in beforehand which particular run represented the end of a given level's round, they continuously monitored provenance looking for such moment when sparse grid's level changed. Once a new complete level was done, by analizing data produced so far it was possible to decide whether results were satisfatory or not; and if such analysis demonstrated that SGSC had converged to scientists expectations (a particular UQ convergence condition), the experiment could be considered terminated. In another scenario, at this point, analysis could also show that experiment had diverged (at least) for a particular sparse grid point, which, for our case, would mean that all points would need to be executed again - this time with another approach (*e.g.* changing the `time step` at solver's activity). Moreover, the experiment could have been interrupted by an external event, which also would imply in SGSC's re-execution. In all worst cases, if scientists had not monitored experiments through provenance queries, energy and time could have been thrown away.

Level	# Point/Run
1	7
2	25
3	69
4	177
5	441
6	1073

Table 3.1: Number of points/runs for each "nested" SGSC's level.

The database generated by Chiron stored only provenance data, such as metadata and several extracted results collected along executions; intermediate application data, such as huge output files and other complex computation results, were only referenced in the database. Such reference was used to be related to the dataflow of the execution, but also providing for a direct access to directories and files, *i.e.* corresponding to a specific sample run during analysis. So, these files were kept in the application storage area. Such process involved connecting at the remote cluster environment and making **laborious data transfer [I2]**.

Following the data transfers, scientists also **applied scripts over gathered**

data looking for post-processing informations that could not be accomplished inside data-flow. In particular, for SGSC, scientists downloaded desired partial results files for several conditions; made calculations over particular attributes; and applied a given Paraview [87] filter for desired visualization; for each condition they needed to analyze [**I3**]. For instance, while trying to assess the `mean` and `standard deviation` (STD) of sediments deposition at the end of the experiment (for each particles settling velocity and grid's level), scientists post-processed results generating images like Figure 3.6.

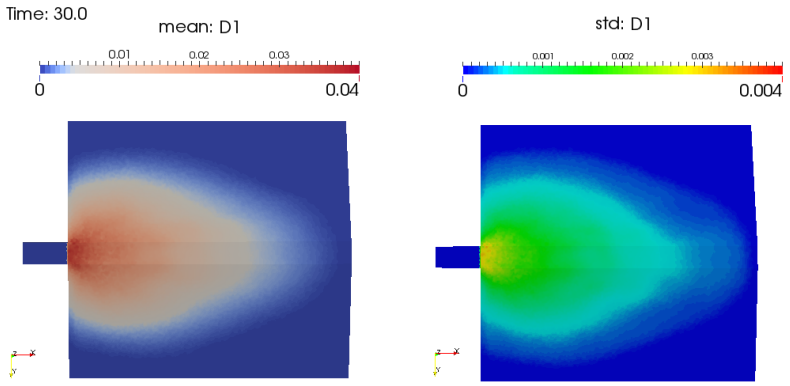

Figure 3.6: Mean and stardard deviation deposition map for one settling velocity, at the sparse grid's level 1.

Chapter 4

Proteus Scientific Gateway

In this chapter, we describe how we reached our goals taking care of how we motivated our choices. First, we describe our research group, facilities, and infrastructure. Next, we take care of the architecture, explaining Proteus design, technologies choices, and implementation. At last, we detail how we achieved our first module implementation by integrating Proteus with an external visualization environment.

4.1 Research Group and Facilities

This work was realized at the High-Performance Computing Center (NACAD) of COPPE/UFRJ. This center is specialized in the application of high-performance computing to engineering problems and large-scale science in general, and other areas of knowledge. NACAD's activities are focused on three main lines: (i) research and development, carried out by NACAD's team of researchers and users of its high-performance computing services; (ii) computer support to teachers groups and researchers of several programs of COPPE, state institutions of Rio de Janeiro and other states; (iii) training of human resources through regular graduate and undergraduate courses, short courses on specific topics and specific courses related to the use of NACAD's infrastructure. NACAD is also part of SINAPAD (Brazilian National Program for High-Performance Processing).

Visualization Environment

Among clusters and storages, inside NACAD facilities students can also take advantage of a Tiled Wall Display (TWD) connected with a dedicated visualization environment. Although this equipment seems to be quite simple, it implements a scalable visualization system based on up-to-date technologies, such as TACC

DisplayCluster[1]. During the elaboration of this work, this dedicated environment configuration was:

- 3 Dell workstations
  - 24 cores (Intel Xeon E5506 @ 2.13GHz)
  - 36 GB RAM
  - 5 display cards NVIDIA Quadro 6000
  - 2,5 TB disk
- 10 DELL displays
  - 31" 2560x1600 resolution
  - 9 visualization displays
  - 1 for adminstration

The two-node cluster (`node2` and `node3`) each have eight processor cores and 12 GB of RAM memory. Each node has two NVIDIA Quadro 6000 graphics cards connected to 2 PCI Express x16 bus on each machine. Graphics cards are configured in dual-head to work together and are connected to four monitors (two monitors for each card). Each node has 500GB hard drive for local storage.

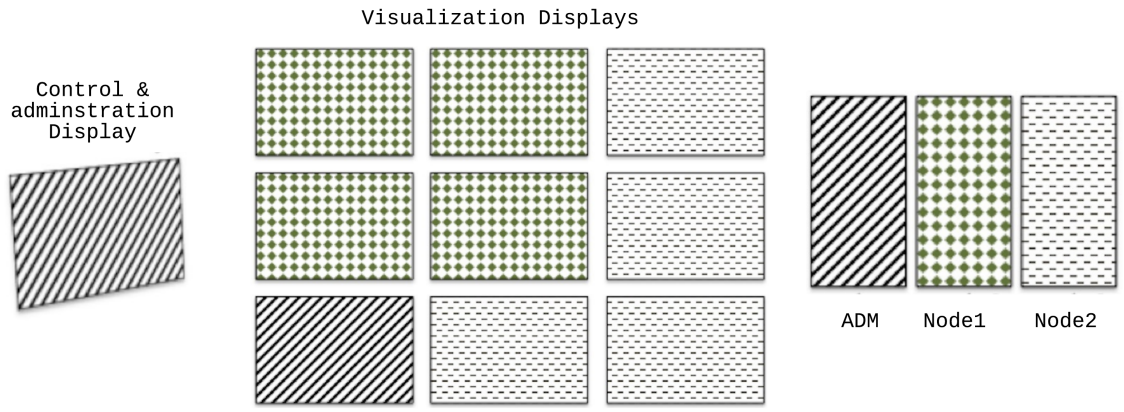

Figure 4.1: Configuration of NACAD's Tiled Wall Display.

The cluster's server (`ADM` node, named Mercury) has a similar configuration of other nodes, with the same number of processors and RAM memory. Though he has only one NVIDIA Quadro 6000 and only controls a display panel monitor. Additionally, the ADM server also controls the cluster management screen. The server disk structure is also different as it has two 500GB disks in RAID1 (originally, Redundant Array of Inexpensive Disks) connected to store the directories of users and two 250GB disks also connected in RAID1 for local storage. In `ADM` machine, the directory of users `/home` and `/sw` are exported through the Network File System

[1]http://www.tacc.utexas.edu/tacc-software/displaycluster

(NFS) so that all nodes can access the files of these directories. Also, the Secure Shell Protocol (SSH) between the nodes is configured for operation without a password, to facilitate the firing of display applications that initiate remote processes on `node2` and `node3`.

As a single physical machine can control up to four monitors, the display panel configuration is as shown in Figure 4.1. In other words, the machine ADM controls a one-panel display, and another control-display for administration. Each one of the other two nodes controls a four-panel display.

This visualization cluster's software configuration is a composition of NACAD's cluster installation tools together with the scientific visualization tool designed by Texas Advanced Computing Center (TACC). As the Operational System (OS), we opted for the Fedora 17 (kernel 3.6.11). The display driver applied is NVIDIA's 310.32. So far, the visualization software installed is the TACC DisplayCluster with Paraview 3.98 [87] Kitware.

TACC is a center of excellence in scientific visualization and presented its DisplayCluster tool in 2012, at the International Conference for High-Performance Computing, Networking, Storage and Analysis (Supercomputing 2012, or SC12). The configuration presented by the TACC SC12 was exactly the same used in NACAD's TWD.

4.2 The Birth of Proteus

Proteus' origin comes from a simpler implementation of the proposed first module for Provenance Visualization. At first it was a final undergrad project, published at [88], and made by the same author of this current dissertation. This first module version - called here by PV0 - was designed and implemented to provide a tight integration between experiments provenance data and the remote visualization systems, in particular, the NACAD's Tiled Wall display.

In this scenario, Proteus was proposed as an output from an extensive literature survey triggered by several examinations made over PV0. In short, after investigating PV0 faults, our research group discovered that was looking for an after-named Scientific Gateway System (SGS), based on a literature survey. NACAD's team was not only seeking to experience visualization *in tandem* with execution, but also some assistantship along the whole experiment lifecycle. It meant, from conception to analysis, to manage permissions to the execution environment and TWD. However, PV0 did not manage to fill these requirements, nor the later-studied related works' (already mentioned in Chapter 1). Therefore, Proteus Scientific Gateway was conceived, from the very beginning, seeking to attend these motivations knowing

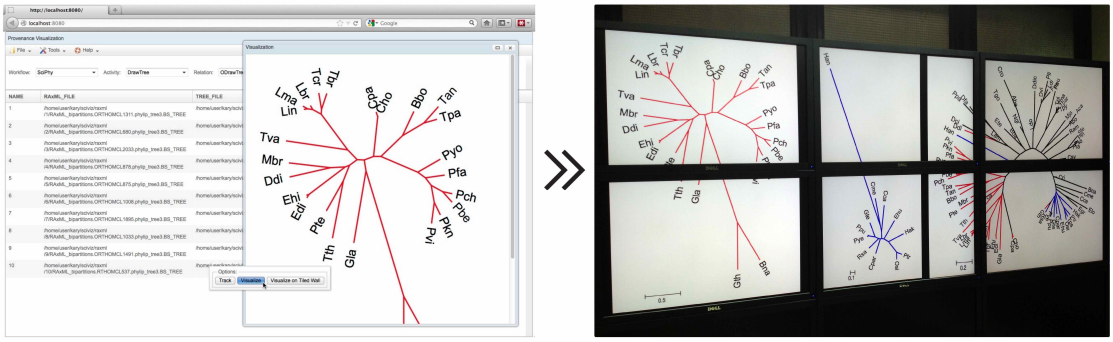

Figure 4.2: Staging data out for a bioinformatics case study using PV0. Designed and implemented to provide a tight integration between experiments provenance data and a particular remote visualization systems

that later PV0 could be integrated into this framework to provide Provenance Visualization.

Since the beginning of this work, it was possible to make many partnerships with several students and researchers that were working with complementary projects. As we shall see it later (Chapter 6), this work could count on the presence of several undergrad students that were in their final undergrad project. With partnerships like these, it was possible to ground Proteus' framework thinking about their contributions that would integrate this tool. So that, it was possible to design Proteus thinking in how we could couple all contributions still providing a consistent and reliable system.

4.3 Design Decisions

Proteus's goals brought us our primary requirements: (**R1**) to provide a runtime large-scale analysis framework based on provenance data; but also (**R2**) assisting scientific workflow lifecycle (**R3**) promoting reproducibility. These conditions, which claim to support a whole visualization of a given experiment since its conception, directed us to take several important decisions. Each one of them we are going to discuss, in later sections, using the references we will create here.

To better understand such decisions, it may be necessary to discretize them among (**C**)onnections, (**D**)atabase, (**P**)rovenance , and (**F**)ramework scopes along a brief description of our assumptions:

To cover condition (R1), we assumed that Proteus would need to establish a new connection with a given provenance every time that it demands to get

relevant information about a particular experiment execution [**C1, D1**]. But foremost, such provenance would need to feed any runtime-requests with updated data regarding experiment execution [**P1, F1**]. Then, following (R1) along runtime needs, Proteus would also need to stage data in or out of remote environments, establishing new connections at each request of data transactions [**C2, D2, F2**].

With (R2), we inferred that such solution would need a considerable effort to provide assistantship to an entire scientific workflow lifecycle. Reasoning that, since Proteus could not accomplish this task alone, a better decision would be to integrate other projects for particular needs. So, the answer was to focus on providing a proper infrastructure, where projects like PV0 could be connected to Proteus. In such case, PV0 would assist with visualization needs taking advantage of Proteus' infrastructure [**F3**].

By (R3) we assumed that Proteus would require interfacing with scientists in a data-centric way [**F4**] since users' sessions could not endure for days continuously, given remote connections availability and human factors. This decision would infer that users' data should persist along multiple interactions [**F5**]. Consequently, users' credentials would need to be validated at each relevant operation [**F6, D3**]. Finally, since we were already considering to manage authentications, we could also help users by enabling access from every connected computer, setting users free from platform constraints or geographic limitations, consequently supporting reproducibility needs [**F7**].

4.3.1 Framework

By considering that our Framework should provide infrastructure to access any provenance; remote machines; and visualization environments at any time regarding particular module request [**F1, F2, F3**], we impose the following character to our solution: **provide functionalities that are independent of their respective implementations**. This character allows definitions and implementations to vary without compromising the interface and it is better known as Application Programming Interface (API).

Providing an API

The main contribution of a well-designed API is to ease the development of other programs by providing all the building blocks. In this way, another programmers should be able to put these blocks together. APIs are common by accessing databases or computer hardware, or *i.e.* helping the work of programming GUI

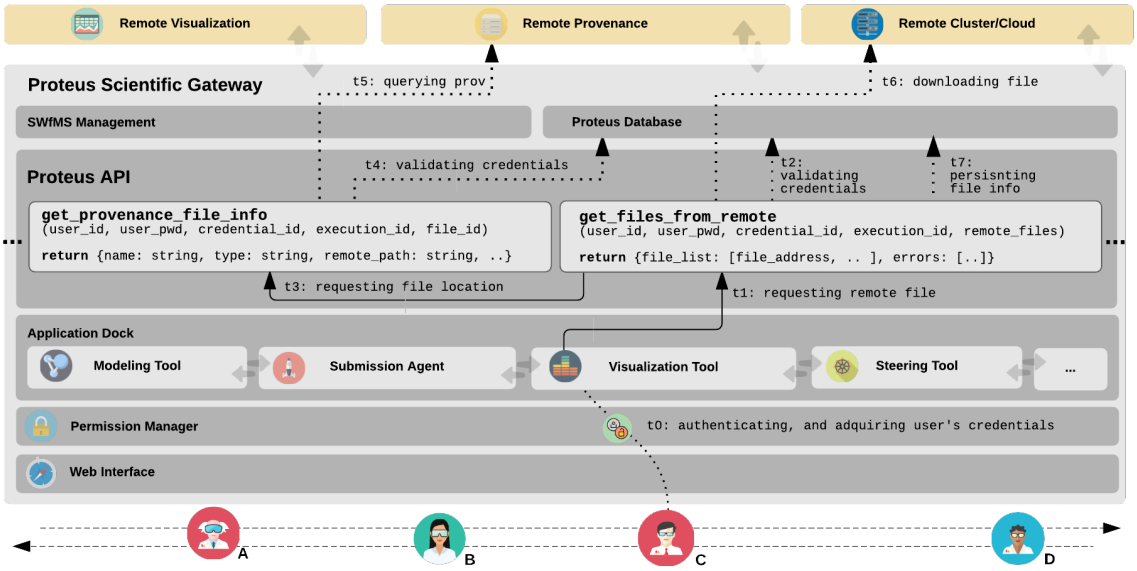

Figure 4.3: Proteus, designed over an Application Programming Interface, provides functionalities that are independent of their respective implementations.

components. In our case, an API could facilitate the integration of new features into the existing applications, bringing the so-called "plug-in" term, or "dock-on". An API could also assist distinct applications with sharing data with each other, which could leverage integration and enhance the functionalities of such applications.

Since API is just a term that refers to the methods a developer would use to interface with the software, we still had the need to define the basic structure underlying our system, the Framework. By definition, a software's framework is a universal, reusable software environment that provides particular functionality as part of a larger software platform to facilitate development of software applications, products and solutions[2]. It would manage tasks such our inversion of control or dependency injection, providing effective tools or templates to make our life easier while implementing our application. In this way, we just needed to adapt this choice to our needs. And by [**F4**], we already knew that such framework would support the development of dynamic websites, web applications, web services or any web resources. Therefore, we chose to apply Django Web Framework[3].

Django Web Framework

Django is an open source high-level Python Web framework that encourages rapid development and clean design. It was designed to help developers taking applications from concept to completion as quickly as possible, still taking security

[2]https://en.wikipedia.org/wiki/Software_Framework
[3]https://www.djangoproject.com

seriously while assisting developers to avoid many common mistakes.

Django offers several features that Proteus' framework may take advantage. Among them, we could list:

- allows programmer to modulate project architecture only by appending back-end or frontend, small and self-contained, applications for a particular group of features (like a `Workflows Manager` or `File Downloader`);
- and by splitting the project in small pieces, Django enhances data model maintenance and coding good practice, which may help Proteus' collaborators porting new applications; and consequently, mitigating integrations conflicts;
- Django also handles database operations through its ORM (Object-Relational Mapping), which facilitates the implementation of domain model pattern, and brings a huge reduction in code.
- at last, Django improves (and beautifies) URL management, which may enhance user's experience while browsing Proteus' applications;

Django Views

Django delivers web pages and other content by `views`. Each `view` is represented by a simple Python function (or method, in the case of class-based `views`). Django framework will choose a `view` by examining the `URL` that's requested (to be accurate, the part of the `URL` after the domain name). See Code 4.1. In this way, Django allows for much more elegant `URL` patterns than other ordinary frameworks such as ASP:

```
http://localhost:8000/dirmod.asp?sid=&type=gen&mod=Core+Pages&gid=..
```

A URL pattern in Django is naturally the general form of a URL, *e.g.*:

```
http://localhost:8000/files/<provenance_id>/<relation_id>/
```

To get from a `URL` to a `View`, Django uses `URLconf` scripts (`urls.py`) which works straightforward and is configurable by within Django's installed applications (which we will discuss later). A `URLconf` maps `URL` patterns (described as regular expressions) to `views`. Look at such example:

Code 4.1: Example of a `urls.py` script.

```
from django.conf.urls import url
from . import views

urlpatterns = [
    #...
    url(r'^executions/([0-9]{4})/$',
        views.workflow_executions,
```

```
        name='workflow-executions'),
    #...
]
```

According to this configuration, the URL for the list of executions corresponding to workflow model `workflow_tag` is `/execution/workflow_tag/`. One can obtain these in template code by using:

```
<a href="{% url 'workflow-executions' UQ_case1 %}">
    Execution List for Workflow UQ_case1
</a>

<!-- Or with the workflow tags in a template context variable: -->
<ul>
    {% for workflow_model in workflow_list %}
    <li>
        <a href="{% url 'workflow-executions' workflow_tag %}">
            Execution List for Workflow  {{ workflow_tag }}
        </a>
    </li>
    {% endfor %}
</ul>
```

Or the same could be written in Python code as:

```
from django.core.urlresolvers import reverse
from django.http import HttpResponseRedirect

def redirect_to_worflow_model(request):
    # ...
    workflow_tag = 'UQ_case1'
    # ...
    return HttpResponseRedirect(
        reverse('workflow-executions', args=(workflow_tag,)))
```

Django Applications

Django brings `Django Packages`, which it inherited from `Python Packages`. Python leverages reusability in various aspects[4]. That is why, Python Package Index (PyPI[5]) has a broad range of packages that developers can use in their Python programs. In a similiar way, `Django Packages` offers Django applications[6]. So that, existing reusable apps can be incorporate to any project without much effort.

Django itself is also simply a Python package, which means that anyone can take existing Python packages, as a `Django Package`, and arrange them into another web project. That is why Django can provide reusability while performing as Proteus web framework. Its implementation would allow collaborators to port new Proteus' applications on the fly, which we could name `Proteus Application`, `Proteus App`,

[4]http://goo.gl/AHDdpe
[5]https://pypi.python.org/pypi
[6]https://www.djangopackages.com/

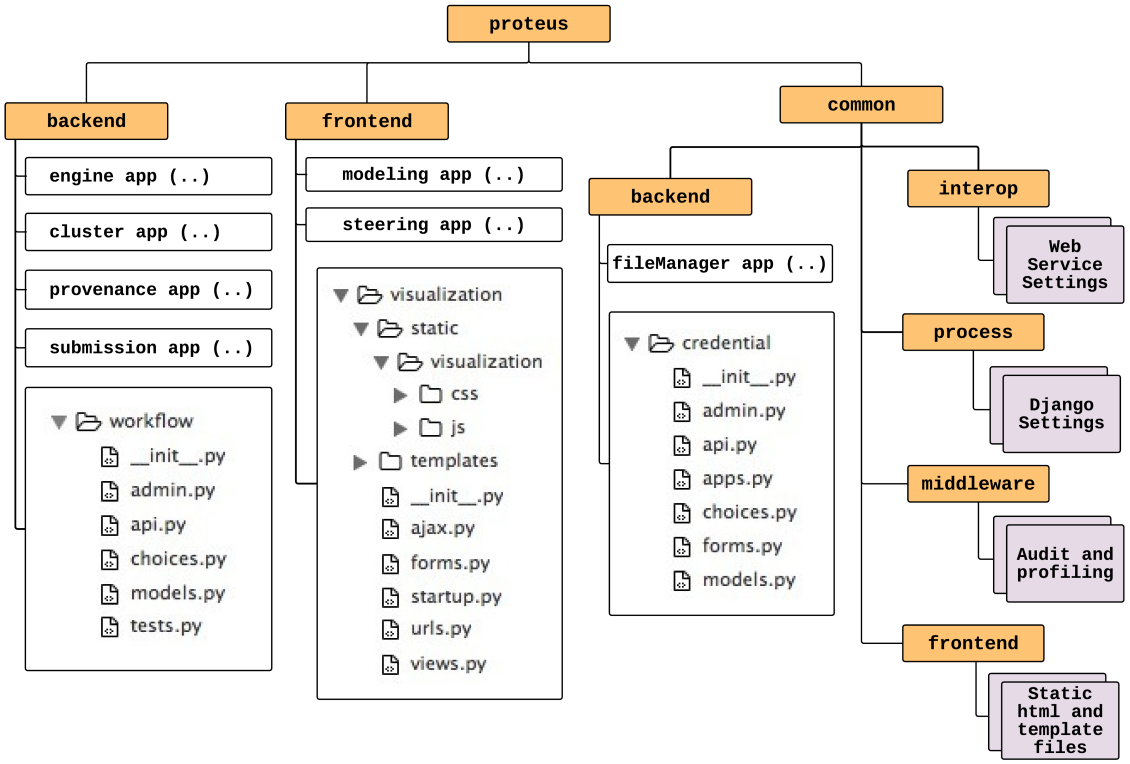

Figure 4.4: Proteus Web Framework directory structure.

or `Proteus Package`.

In Django, any package can be imported using the following python-script directive:

```
import proteus.backend.visualization
# or
from proteus.backend import visualization
```

Or, for a directory (like exposed in Figure 4.4), to determine a package, it must contain a particular file `__init__.py`, which can also be left blank.

As a result of this approach, a `Proteus Application` would be merely a Python package, that once, a programmer build aiming for use in a Django project. So, it may use standard Django conventions, such as having defined `models` (as we will see next), `URLs`, and `views` submodules as shown in Figure 4.4.

Django ORM

As we saw previously, the basis of building dynamic websites with Django is to set up `views` and `URLconfs` scripts. In this way, a `view` handles doing some arbitrary logic and then returning a response to a Web client. However, in a modern web applications, this arbitrary logic often involves interacting with a database, as Proteus should do [**F5**]. And out of sight, a database-driven application connects

to a database server; retrieves some data, and displays on a Web page. Also, this website might provide ways for site visitors to populate the database on their own.

Proteus should provide some combination of the two. For example, Proteus - as a database-driven application - might be able to query into its database looking for provenance-connection data related to a given Proteus' user.

Django Web Framework may be well suited for providing to Proteus a database-driven layer. It comes with powerful tools for performing database queries using Python through its Object-Relational Mapping layer (ORM).

The ORM is a powerful database tool. In computer science, ORM is a programming technique for converting data between incompatible type systems in object-oriented programming languages. Consequently, creating a "virtual object database" that can be used from within the programming language. Although some programmers opt to create their ORM tools, there are both free and commercial packages available that perform object-relational mapping. Django ORM, for instance, handles the creation of the database, as well as insertion, update and delete queries; among others useful operations. Moreover, Django ORM supports multiple databases - MySQL, PostgreSQL, Oracle and SQLite are all supported assuming programmer has the appropriate Python libraries installed.

Django framework, by handling database operations through its ORM, eases the implementation of domain model pattern (and also brings a huge reduction in code). The concept of Django ORM introduces the `Django Models`, which is how Django abstractly translates pieces of information from the database. We shall see `Django Models` in details in the next section. But first, let's see how the one can easily take advantage of ORM.

A given request could be accomplished by connecting to a MySQL database through `MySQLdb` library[7]; and hard-coding database connection parameters, as such:

```
from django.shortcuts import render
import MySQLdb

def provenance_list(request):
    # creating a connection
    db = MySQLdb.connect(user='me', db='mydb', passwd='secret', host='localhost')
    # creating a cursor
    cursor = db.cursor()
    # executing a statement
    cursor.execute('SELECT name, location FROM provenance ORDER BY name')
    data = [row[0] for row in cursor.fetchall()] # list of dictionaries
    # finally, closing the connection
    db.close()
    return render(request, 'provenance_list.html', {'provenance_list': data})
```

[7] http://www.djangoproject.com/r/python-mysql/

But, with the help of Django ORM's API, the same code can be rewritten as:

```
from django.shortcuts import render
from proteus.backend.provenance.models import Provenance

def provenance_list(request):
    # creating a connection; creating a cursor; executing a statement;
    # and closing the connection - all in one
    data = Provenance.objects.order_by('name')
    return render(request, 'provenance_list.html', {'provenance_list': data})
```

Django ORM's API offers this and many other opportunities that may ease Proteus' development. Moreover, with features like an ORM, Proteus can be designed to enhance loose coupling and strict separation between pieces of its application. Following this philosophy, it would be easy to make changes to one particular `Proteus Application` without affecting the other applications. Previously, in `view` functions, for instance, we see the contribution of separating the business logic (server side) from the presentation logic (client side). By choosing such ORM framework, while dealing with the database layer, we're applying that same philosophy to data access logic.

Django Models

In Django, a model is the single definitive source of data about a specific information. A `Django Model` contains the essential fields and behaviors of the information stored. Almost ever, each model maps to a single database table.

`Django Model` is simply a Python class that subclasses `django.db.models.Model`, which each attribute represents a database field. With all of this, this is the way Django directs programmers to an automatically-generated database-access API. The following code, *i.e.* defines the Provenance Model and its relationships.

Code 4.2: `Provenance` model definition at `provenance/models.py`.

```
from django.db import models
from proteus.common.backend.credential.models import *
from .choices import PROV_VERSIONS

class Provenance(models.Model):

    database = models.CharField(max_length=50)
    port = models.CharField(max_length=10)
    server = models.CharField(max_length=50)
    server_port = models.CharField(max_length=10, blank=True)

    credential = models.ForeignKey(Credential)
    prov_version = models.CharField(choices=PROV_VERSIONS, max_length=50)
```

```
    def __unicode__(self):
        return "%s @ %s / :%s" % (self.server, self.database, self.port)
```

The above Provenance model would create a database table like this:

Code 4.3: Provenance model described by Django on database.

```
CREATE TABLE provenance_provenance (
    id INTEGER NOT NULL,
    database VARCHAR NOT NULL,
    port VARCHAR NOT NULL,
    server VARCHAR NOT NULL,
    credential_id INTEGER NOT NULL,
    prov_version VARCHAR NOT NULL,
    server_port VARCHAR NOT NULL,
PRIMARY KEY (id),
CONSTRAINT FOREIGN KEY (credential_id) REFERENCES credential_credential (id)
    ON UPDATE NO ACTION
    ON DELETE NO ACTION);
```

See that Django concatenates the application named `Provenance` with its model's name, resulting in the table `provenance_provenance`. In Django, the name of datatable automatically derives from some model metadata, unless the programmer overrides this. Also, notice that the SQL at Code 4.3 is formatted using SQLite, but as we are going to see later, Django can interface others DBMS.

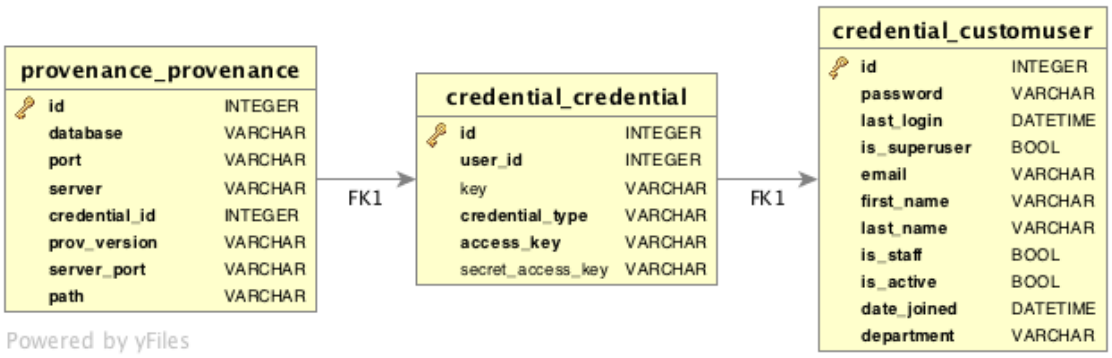

Figure 4.5: Proteus's ER model between Provenance, Credential and Users.

Since `Provenance` data need the user's `Credential` for connection authentication, its model owns a `Credential` field. So, its database table translated by Django, also contemplates a `FOREIGN KEY` to `Credential` table. Figure 4.5 presents the entity-relationship model between `Provenance`, `Credential`, and `User` models, already SQL formatted. And Figure 4.6 shows the same relationship formatted according to Django fields for the respective `models.py` file.

Code 4.4: `User` and `Credential` models defined at `credential/models.py`.

```
# ...
class CustomUser(AbstractBaseUser, PermissionsMixin):
    email = models.EmailField(_('email address'), max_length=254, unique=True)
```

```
    first_name = models.CharField(_('first name'), max_length=30, blank=True)
    last_name = models.CharField(_('last name'), max_length=30, blank=True)
    is_staff = models.BooleanField(
        _('staff status'), default=False,
        help_text=_(
            'Designates whether the user can log into this admin site.'))
    # ...

class Credential(models.Model):
    user = models.ForeignKey(CustomUser, related_name='credentials')
    key = models.FileField(upload_to='keys', null=True, blank=True)
    credential_type = models.CharField(choices=CREDENTIAL_TYPES, max_length=200)
    access_key = models.CharField(max_length=200)
    secret_access_key = models.CharField(max_length=200, blank=True, null=True)

    def owner(self):
        return self.user.get_full_name()
# ...
```

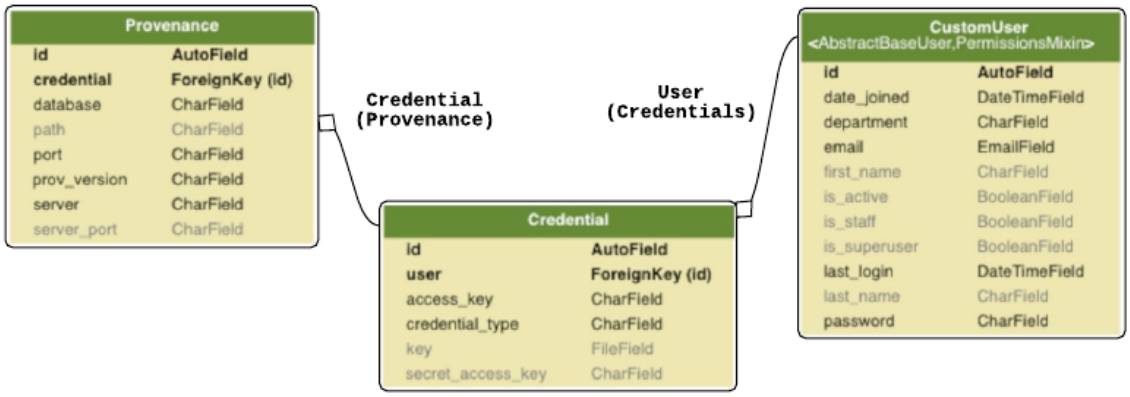

Figure 4.6: Provenance, Credential and Users relationship formatted as Django models.

Applying new models

Once developers have defined their models, they need to tell Django to apply those to the framework. This is made by editing the Django's settings file and changing a variable called `INSTALLED_APPS`, including the name of the module that contains the `models.py`.

Taking Figure 4.4 file structure as example. If a developer implemented a new Proteus'App, *i.e.* to handle workflow management, and its `Python Package` lives in at `proteus.backend.workflow` (the package structure from the root of the project), `INSTALLED_APPS` should read, in part:

Code 4.5: `INSTALLED_APPS` variable users may change to apply new applications models.

```
INSTALLED_APPS = (
    # ...
    'proteus.common.backend',
    'proteus.common.backend.credential',
    'proteus.common.backend.file_management',
    # ...
    'proteus.backend.workflow', # just added
```

```
    'proteus.backend.engine',
    'proteus.backend.register',
    'proteus.backend.provenance',
    'proteus.backend.environment',
    'proteus.backend.submission',
    # ...
    'proteus.frontend.visualization',
    'proteus.frontend.modeling',
    'proteus.frontend.workflow_submission',
    'proteus.frontend.steering',
)
```

After adding new apps to `INSTALLED_APPS`, users need to be sure to run `manage.py migrate`. This `manage.py` script is located at the project basis and it is responsible for many features that improve users productivity, such as `migrate`, which applies model changes for particular models; `startapp`, which helps users to start a new application, generating the file structure at a particular location; or `makemigrations`, that persists model's version at each change, allowing developers to rollback to old versions during development without losing data; among other commands.

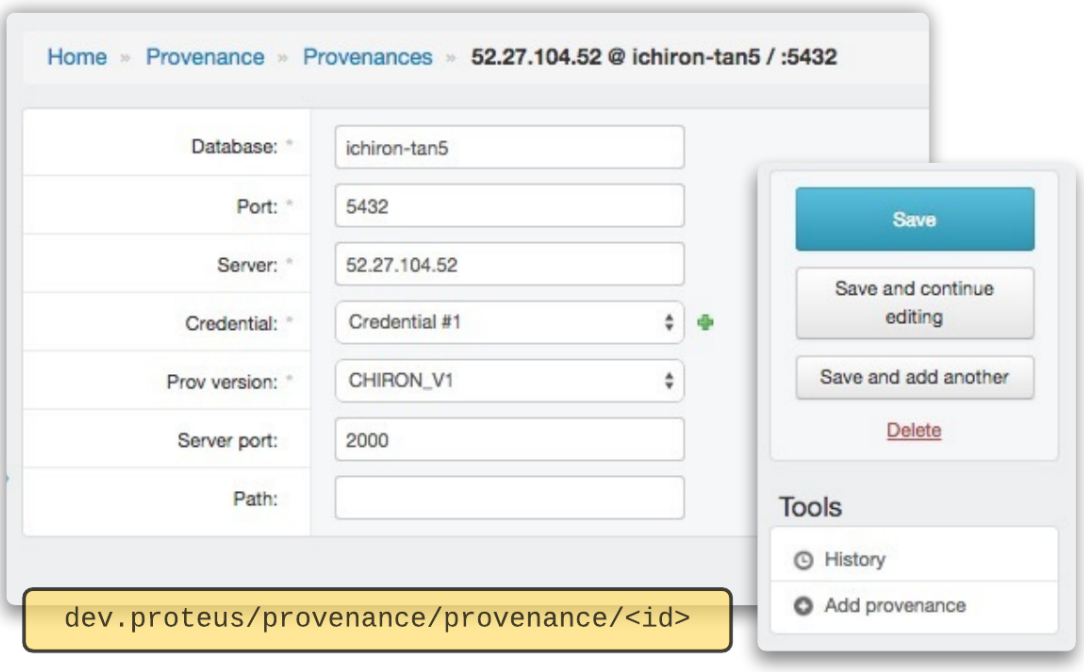

Figure 4.7: Example of a Provenance data-model edition view created by Django for the Admin's interface.

Django also brings a powerful automatic admin interface. It reads metadata in installed applications models to provide a production-ready interface that content producers can immediately use to start adding content to the site. For each model created, Django generates CREATE; RETRIEVE; UPDATE and DELETE (CRUD) operations also for the presentation layer. Figure 4.7 shows an example of an automatic

admin interface built over Code 4.2 description.

Packaging and applying new Apps

Although developers may design new inside-features during the development of a Django Web Framework (such as the `Workflow App` described below), others may develop applications aiming to provide out-of-the-box features. Such features may vary from third-party software's integrations to Proteus' common features update. The greater contribution of this "dock on" feature is that, promoting loose coupling applications, we avoid Proteus to get outdated, being possible to update external applications without much effort.

Moreover, `Proteus Applications` might be able to integrate with Proteus without damaging its architecture. Often, critical issues can be raised from within project's architecture after instrusive integrations. However, `outside` resources, if docked in `Proteus Applications`, could lay over a different software structure already awared of intrusive or bad intentioned operations.

Finally, it is also possible to package and share a given application without much effort. For example, we can package a `Proteus Application` apart from the rest of the project; upload, publish and distribute it over tehcnologies such as Git[8]. This process will be described in the next chapter while we evaluate Proteus installing a visualization purpose application above its architecture.

4.3.2 Provenance

As mentioned at [**P1**], it was crucial that provenance should feed runtime-requests with updated data regarding the experiment execution. So, considering such requirement, we decided to integrate Proteus with Chiron [11] or SciCumulus[12] SWfMSs (described better in sections 1.4 and 2.2). Both solutions allow for *in tandem* experiment execution and data analysis since provenance data is written similarly into a relational database at runtime. Although SciCumulus approaches the same problem leveraging the cloud's infrastructure needs, both engines generate similar provenance model. Hence, until start dealing with execution environment requirements, we are going approach this methodology referring only to Chiron.

As a [**P1**] outcome, we should connect to provenance to retrieve information about the experiment. Thus, it would be necessary to understand how Chiron generates provenance for later start dealing with the connection. Figure 4.8 shows the entity-relationship model of the prospective provenance generated by Chiron.

[8]https://github.com/

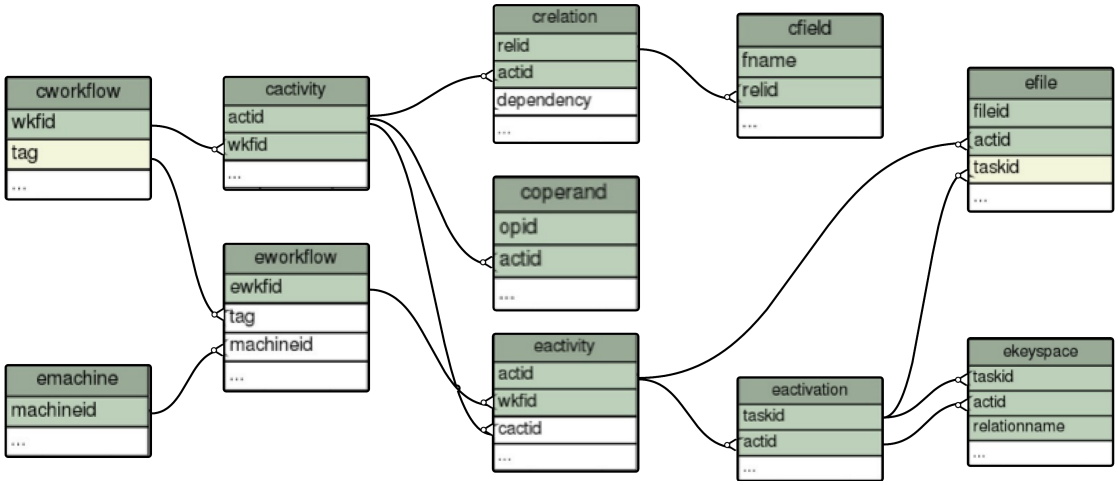

Figure 4.8: Prospective provenance according to Chiron

In this way, a new connection to provenance should be accomplished every time that it demands to get relevant information about an experiment execution [**C1**]. However, an ordinary provenance connection - unlike an ORM database - would not be able to be mapped into `objects` since it dynamically changes its relations. So that, we decided to implement a connection layer to at least abstract the most frequent calls intro Provenance API.

A Provenance API should intercept regular requests, converting them into proper provenance queries while dealing with connection and authentication. Moreover, it should be supported by Proteus' Database [**D1**], which would provide users' credentials on demand. For this reason, proper technologies should be chosen considering that would be needed connecting with a PostgreSQL database [**P1**] through a Python language framework . In this way, considering this, we chose the Psycopg[9] as our PostgreSQL adapter. Code 4.6 illustrates an attempt to connect with a database using this Pyscopg adapter.

Code 4.6: Example of a connection transaction using Psycopg adapter.

```
>>> import psycopg2

# Connect to an existing database
>>> conn = psycopg2.connect("dbname=test user=postgres")

# Open a cursor to perform database operations
>>> cur = conn.cursor()

# Execute a command: this creates a new table
>>> cur.execute("CREATE TABLE test (id serial PRIMARY KEY, num integer, data varchar);")

# Pass data to fill a query placeholders and let
# Psycopg perform the correct conversion
cur.execute("INSERT INTO test (num, data) VALUES (%s, %s)",(100, "abc'def"))
```

[9]http://initd.org/psycopg/

```
# Query the database and obtain data as Python objects
>>> cur.execute("SELECT * FROM test;")
>>> cur.fetchone()
(1, 100, "abc'def")

# Make the changes to the database persistent
>>> conn.commit()

# Close communication with the database
>>> cur.close()
>>> conn.close()
```

4.3.3 Remote Connections

By **[C2]**, requirement raised in Section 4.3, Proteus should also establish a connection with any remote environments providing file upload/download operations between two machines. Despite the nature of the target machine (cluster, clouds or a simple remote node), data would need to be streamlined via some file transfer protocol.

In the same way of **[C1]**, we decided to solve this problem by implementing another connection layer, this time focused on data transfer rather than database reading.

So, since we had already adopted Django, a Python Web Framework supporting **[F1-5]**, the first alternative for such operation would require a python subprocess call, followed by a raw Secure Copy (SCP) - bash command implemented over SSH protocol- such as:

```
    os.system("scp <file> <user>@<server>:<path>")
```

However, this decision would result in taking a lot of effort in parsing the remote output or treating exceptions that may occur in this process. For this reason, a better decision was made while choosing Fabric[10] to handle data transfer.

Fabric is a Python library and command-line tool for streamlining the use of SSH for application deployment or systems administration tasks. It provides a basic suite of operations for executing local or remote shell commands (normally or via sudo, which means administrative privileges) and uploading/downloading files, as well as auxiliary functionality such as prompting the running user for input, or aborting execution. Fabric was designed to be used in a very large scale scenario among huge deploys, and it looked like would fit our needs. Fabric can be configured to handle multiple requests and deal with singular exceptions. Code 4.7 presents a example for a given endpoint implemented at Proteus Connection layer.

Code 4.7: Example of Proteus' API endpoint provided by its Connection layer.

```
#!/usr/bin/python
from fabric.api import *
# import ..

def get_file_from_remote(
```

[10] http://www.fabfile.org/

```
    user_id, user_pwd, credential_id, execution_id, remote_files):

    # (1) getting user by id (..)
    # (2) checking auth (..)
    # (3) retrieving credentials by credential_id and user
    # (4) setting fabric (env)ironment setting file if key is provided
    env.key_filename = credential['key']
    # (5) setting particular permission file to public key
    local('chmod 400 ' + env.key_filename, capture=False)
    # (6) setting connection parameters
    env.skip_bad_hosts=True # (whether skip over hosts it can't connect to)
    env.timeout=2 # (network connection timeout, in seconds)
    env.warn_only=True # (whether Fabric exits when detecting errors)
    env.parallel=True # (whether process transfer of multiple files in parallel)
    # (7) attempting to download file
    result = get(remote_files)
    # (8) evaluating and persisting file instance
    if result.succeeded:
        for _file in result:
            with open(_file, 'r') as _f:
                downloaded_file = dFile(_f)
                # Instantiating new file for retrieved media (..)
                file_list.append(new_file)
    else: # dealing w/ exception (..)
    return file_list
```

4.4 Implementation

This section briefly describes the implementation of Proteus with respect to its architecture. Further, we outline technologies and methods that brought us to our first experimental setup.

4.4.1 Versioning with Git

Since we decided to start Proteus' development based on our design decisions (described at Section 4.3), we also started versioning project's files using Git[11] revision control system served by Bitbucket[12].

Git is a popular distributed revision control system with an emphasis on speed; data integrity; and support for distributed, non-linear workflows. In short, it is a great tool for maintaining a large distributed development project, along with many collaborators.

With Bitbucket web-based hosting Git service, it was possible to host our project's repository for free while maintaining a limited number of collaborators. Figure 4.9 shows a snapshot of SourceTree's [13] GUI while handling many branches of development in late June (2015). So far, this project had collaborations of 3 more developers interested in providing new features to Proteus. This project's first commit was made at early July

[11]https://git-scm.com/
[12]https://bitbucket.org/
[13]https://www.sourcetreeapp.com/

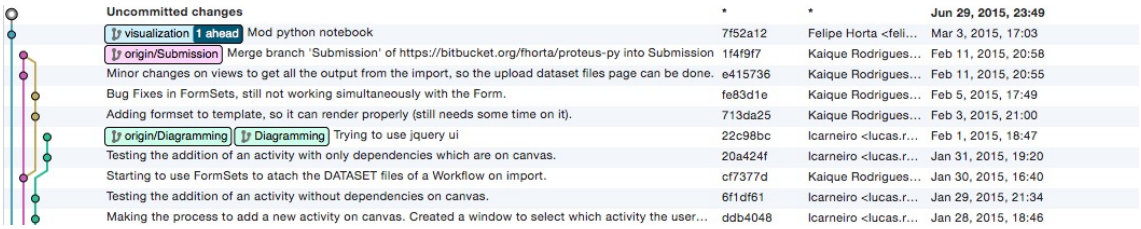

Figure 4.9: SourceTree's GUI while handling several Proteus' coding branches.

(2014). Currently, this project has private access, but potential collaborators may apply for access at `https://bitbucket.org/fhorta/proteus-py`.

4.4.2 Choosing a proper DBMS

At Section 4.3 we made the decision of taking Django as our Web framework for Proteus' application. However, we didn't decide over a particular database management systems (DMBS) to take care of Proteus' database. From Django ORM, we already knew that it could support multiple databases, such as MySQL, PostgreSQL, Oracle, SQLite; among others - since the developer had the appropriate Python library installed. So, as we hadn't had any requirements over a particular database constraint during the Proteus' development phase, we opted for SQLite DBMS, which reasons we may explain.

SQLite[14] is a relational database management system contained in a C programming library. However, in contrast to many other DMBS, SQLite is not a client–server database engine. Rather, it is embedded into the end program. By default, Django configuration uses SQLite, which commonly is the easiest choice to start a project since SQLite is already included in Python. So, programmers, when starting their first real project usually move to a more robust database. In our case, the philosophy was the same. Even though we planned to change the DBMS in a short time, (strictly) while coding Proteus' we didn't find any advantages in keeping a robust database like PostgreSQL or MySQL, quite the opposite. Once we found out that it was possible to keep our database versioned using Git - given the fact it was so small - we decided to continue using SQLite DBMS. Such small database file offered several advantages until now, even if it is not the ideal DMBS for a client-server application. Because we could version an SQLite database file, we didn't need to care about collaborators' model changes until checking out another version, the database would be ready to use.

Additionally, Django Web Framework offers a special feature to load initial data over a database sourced by a JSON file (from Javascript Object Notation). This support was crucial for keeping SQLite DMBS while development. With JSON files, like shown at Code 4.8, we could load initial data for testing a regular workflow, such as first users, workflow models, provenance data, TWD informations, etc.

Code 4.8: An example of initial data loaded at Proteus for testing during development.

[14]https://www.sqlite.org/

```
[{
    "fields": {
        "name": "mercury",
        "tile_height": 1600,
        "tile_padding": 100,
        "tile_width": 2540,
        "address": "146.164.31.201",
        "path": "Users/fhorta/TWD/",
        "port": "9001",
        "tile_y": 3,
        "tile_x": 3
    },
    "model": "environment.visualization",
    "pk": 2
}, {
    "fields": {
        "host": "uranus.nacad.ufrj.br",
        "name": "uranus"
    },
    "model": "environment.cluster",
    "pk": 1
}]
```

4.4.3 Starting Project

Although skipping some steps, showing only the most important ones, we may start from the project's basis configurations, then jumpping to our architecture decisions.

First Steps

The very first step of our project was to include Django package in our `PYTHON_PATH`. And for this, we just needed to check out the latest version of Django present at its Git repository.

```
git clone git://github.com/django/django.git django-trunk
```

After proper installation, which all detailed steps can be found here[15], Proteus' project started as any Django project does. Initial setup was auto-generated by `django-admin` script. Running this assistant script, it was possible to establish our first project files with: a collection of settings for an instance of Django, including database configuration, Django-specific options; and a first application-specific settings. In this way, from the command line, we just ran:

```
# cd into the directory where we'd like to store our code
django-admin startproject proteus
```

This command created our initial version of Proteus file structure. And it looked like this:

```
proteus/                # root directory;
    manage.py           # command-line utility that lets us interact with this Django;
```

[15] https://docs.djangoproject.com/en/1.8/topics/install/

```
    proteus/             # actual Python package for our project;
        __init__.py      # empty file - tells Python this dir should be
                         # considered a Python package;
        settings.py      # settings/configuration for this Django project;
        urls.py          # basis URL declarations for Proteus;
        wsgi.py          # entry-point for WSGI-compatible web servers.
```

Database Setup

So, after creating our initial files for Proteus Web Framework, we could finally configure Django's settings at `proteus/settings.py` script.

By default, as we mentioned, Django's configuration uses SQLite. And this DMBS was our choice during Proteus' development. So that, we configured the `DATABASE` variable at part of our `settings.py` file as:

```
# ..
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3')
    }
}
# ..
```

Proteus' Base Apps

As shown at Figure 4.4, we created several applications composing Proteus' base functionalities. For each of them we first ran the following:

```
python manage.py startapp <application_name> <destination>
```

And as a response to such command, it was created the following file structure beneath Proteus's architecture according to the desired destination.

```
base_application/
    __init__.py
    admin.py
    migrations/
        __init__.py
    models.py
    tests.py
    views.py
```

In this way, for each application created, we changed the `settings.py` variable `INSTALLED_APPS`, including the respective package we'd like to address. Doing this, each package would be automatically included in the `PYTHON_PATH` variable. Consequently, it would be accessible at any script prior Proteus' `Django project` initialization. Code 4.5 illustrates part of `INSTALLED_APPS` variable at `settings.py` script.

Also, as described in Section 4.3.1, for each installed apps, Django helped us automatic creating their presentation views. This feature saves us a lot of time. Following this, we

decided to apply such automatic feature for each new applications that Proteus could integrate later.

Running Proteus

Finally, skipping irrelevant configurations directed for this Chapter, we jump to Proteus' server instantiation. Running Proteus' server could be accomplished since the very beginning of its development regarding implementation tests and debug. So that, after defining any business, presentation or data logic within applications we were developing, we were able to test Proteus by running it on a server. The following command helped us in this task:

```
python manage.py runserver
```

If everything is ok, such command would perform the following response:

```
Performing system checks...

0 errors found
June 30, 2015 - 15:50:53
Django version 1.8, using settings 'proteus.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.

# ..subsequent log for operations..
```

4.4.4 Architecture

In the last section, we provided several insights about Proteus' infrastructure needs which resulted in choosing an API as the interface where developers should interact with its architecture. For this reason, a simple architecture proposed by a default Django application wouldn't be enough. So, we proposed ours.

Since applications, native or external, would provide functionalities in business, presentation and data logic; we needed to organize such destinations within our project looking to provide the best service while integrating and serving each new application.

Applications such as `File Manager`, presented at Figure 4.4, by default, does not support any presentation logic. This app may operate only on the business and data logic since it does not require a user interface. Thus, it would not present any static files, such as HTML, Javascript or CSS. Instead, it would only need to state the definition of its new model classes, such as `File`, `Results` or `Scripts`; as shown in the following piece of the code:

```
# proteus/common/backend/file_management/models.py

class File(models.Model):

    def get_upload_path(instance, filename):
        # ..
        return path
```

```
    owner = models.ForeignKey(CustomUser, related_name='files')
    file = models.FileField(upload_to=get_upload_path, null=True)
    execution = models.ForeignKey(WorkflowExecution)
    host = models.CharField(max_length=100)
    remote_path = models.CharField(max_length=500)
    tag = models.CharField(max_length=100)
    date_modified = models.DateTimeField(auto_now=True)
    file_type = models.CharField(max_length=10)

    class Meta:
        verbose_name = "File"
        verbose_name_plural = "Files"
# ..
class Script(models.Model):
    owner = models.ForeignKey(CustomUser, related_name='scripts')
    tag = models.CharField(max_length=100, unique=True)
    content = models.TextField(blank=False)
    date_modified = models.DateTimeField(auto_now=True)
    file_type = models.CharField(choices=FILE_TYPES, max_length=100)

    class Meta:
        verbose_name = "Script"
        verbose_name_plural = "Scripts"

    def __unicode__(self):
        return '%s' % (self.tag)
# ..
```

The example above presents a common backend application; which may live by particular rules once integrated with Proteus; serving users' needs and also providing functionalities to other applications. This `File Manager` app, especially developed for common backend purpose, would be better located over `proteus.common.backend` package. Following this logic, other packages would also be created looking to cover frontend and backend needs, rather than serving to common utilities. Thus, being addressed to particular locations based on its funcionalities. For this reason we proposed an architecture based on `Backend`; `Frontend`; and `Common` applications, as shown in Figure 4.4. In this way, applications such `File Manager` would only need to:

- define new class models within `models.py` file beneath its package;
- build API's endpoints (`api.py`) which would handle its business logic integrated with other Proteus' endpoints (*i.e.* Code 4.7)
- and publish functionalities, so other apps could integrate later.

4.4.5 Building Proteus' API

Proteus' applications, even distributed among three different structures (`backend`, `frontend` and `common`) should consolidate into a unique API. Mainly in regard to authentication requirements and security rules. Further, proper registration methods should be applied, indexing their frontend or backend contributions. Once registered,

Proteus may contemplate their `static files` at the frontend, if it is the case, listing new funcionalities at the main menu; or their endpoints at the `backend` API, allowing integration with other applications.

For this reason, we developed a startup code that bootstraps all applications registered at project's settings `INSTALLED_APPS` variable, and runs every time Proteus is instantiated (Code 4.9). This code could count on support of another custom application we built, named `Register`.

Code 4.9: Startup script which handles Proteus' app registration

```
# proteus/common/process/startup.py

from django.conf import settings
from django.utils.importlib import import_module
from django.utils.module_loading import module_has_submodule

def import_applications():
    # Get the startup code for all modules, beginning with the Register
    startup_modules = []
    REGISTER_MODULE = 'proteus.backend.register'
    installed_apps = sorted(
        settings.INSTALLED_APPS, key=lambda a: 1 if a == REGISTER_MODULE else 2)
    for app in installed_apps:
        mod = import_module(app)
        try:
            startup_modules.append((app, import_module("%s.startup" % app)))
        except:
            if module_has_submodule(mod, 'startup'):
                raise  # Avoid swallowing other exceptions.

    # Register modules.
    from proteus.common.interop.register import register_module
    for (app, sm) in startup_modules:
        if hasattr(sm, 'module'):
            register_module(sm.module, app)

    # Register pages.
    from proteus.common.interop.register import register_pages
    for (app, sm) in startup_modules:
        if hasattr(sm, 'pages'):
            register_pages(sm.pages)

    # Run other arbitrary code.
    for (app, sm) in startup_modules:
        if hasattr(sm, 'run') and sm.run.__call__:
            sm.run()
```

`Register`, located at `proteus.backend.register` and invoked at Proteus's startup, requires that each application keeps a `startup.py` script beneath its package, declaring its `name`, `description`, and `static pages` definitions. Code 5.2 is an example of startup code for a `Visualization Application`.

Code 4.10: Startup informations for Proteus' Visualization App.

```
# proteus/frontend/visualization/startup.py

from django.core.urlresolvers import reverse_lazy

description = u"""
        Provides experiment Visualization based on Provenance queries (..)
        """

module = {'display_name': u"Workflow Visualization",}

pages = [

    {'identifier': 'vis',                              # main page identifier
    'display_name': u"Visualization",                  # page title
    'description': description,
    'priority': 30,                                    # priority order at main menu
    'icon_class': 'proteus-icon-visualizer-small',     # menu icon filename
    'url': reverse_lazy('vis-submission_list')},       # url which may be addressed
                                                       # (mapped at urls.py)
    # ..

    {'parent': 'vis',                                  # parent identifier
    'identifier': 'vis-tiledwall',                     # this page identifier
    'is_popup': True,                                  # whether is shown as a popup
    'display_name': u"Tiled Wall",
    'url': reverse_lazy('vis-tiledwall')},
]
```

Finally, by the end of Proteus' implementation, we had the following set of applications:

```
INSTALLED_APPS = (

    # PROTEUS BASE APPS ------------------------------------------------------/
    'proteus.common.backend',                 # port and monitor common-backend API

    'proteus.common.backend.register',        # handle/persists installled
                                              # Proteus applications info

    'proteus.common.backend.credential',      # deal with users' profile and
                                              # remote credentials

    'proteus.common.backend.file_management', # manage files/scripts/results
                                              # remote/local operations

    'proteus.common.backend.workflow',        # manage workflow model,relations.
    'proteus.common.backend.submission',      # manage submissions data.
    'proteus.common.backend.provenance',      # manage provenance connections

    'proteus.common.interop',                 # handle API registration and
                                              # intercept any call to API

    'proteus.common.frontend',                # port and monitor common-frontend
                                              # static files

    # DJANGO DEFAULT MODULES -------------------------------------------------/
    'django.contrib.auth',          # authentication and authorization services

    'django.contrib.contenttypes',  # track all of the models installed, providing
                                    # a high-level, generic interface
```

```
                                    # (used at app registration)

    'django.contrib.sessions',      # store and retrieve arbitrary data on a
                                    # per-site-visitor basis.

    'django.contrib.messages',      # handle message services between *views*
    'django.contrib.staticfiles',   # collect static files from each application
    'django.contrib.admin',         # automatic generate admin interface (CRUD*)

    # DJANGO IMPORTED APPS ------------------------------------------------------/
    'filetransfers',        # handle frontend-files uploads
    'compressor',           # handle static files compression (css, html, js ..)
    'dajaxice',             # provide AJAX* > API from frontend
    'suit',                 # provide better layout to admin templates
    'django_extensions',    # support powerful django extensions scripts
                            # while coding

                        # CRUD: Create/Retrieve/Update/Delete Database Operations
                        # AJAX: Javascript Asynchronous Call
)
```

Chapter 5

Evaluating Proteus

This chapter's intent is to evaluate Proteus while providing support for its native applications, such as access control and provenance management, among others. Also, we might see how Proteus supports the development and integration of new applications above its framework, such as our first provenance-based visualization module, named Prov-Vis. And, finally, we will assess our solution while sharing data and access among a group of users.

> All the following evaluated functionalities were assessed from the same machine that Proteus was running. It means that Proteus' server could be accessed by the local web address `http://localhost:8000/`, which for matters of demonstration we named with the alias `dev.proteus`.

In addition, as we mentioned at Section 4.4.2, our architecture enables loading initial data to Proteus at its deploying or during execution. Thus, for the matters of this evaluating process, some data - as credential data (see Code 5.1) - was already loaded prior to evaluation.

Code 5.1: Example of part of initial data loaded prior to evaluation.

```
[{
    "fields": {
        "first_name": "Felipe",
        "last_name": "Horta",
        "is_active": true,
        "is_superuser": true,
        "is_staff": true,
        "last_login": "2014-11-03T18:39:45.904Z",
        "groups": [],
        "department": "",
        "user_permissions": [],
        "password": "pbkdf2_sha256$12000$unv9ZDPdhm279uSPsxiQDUBFLps6jmw4=",
        "email": "felipe.f.horta@gmail.com",
        "date_joined": "2014-10-13T21:54:17.685Z"
    },
    "model": "credential.customuser",
```

```
    "pk": 1
    #   ...
},
```

5.1 Built-in Applications

Built-in applications are those developed by us, and already present at Proteus as-it-is. These applications make part of Proteus' presentation, business and data logic which may integrate its API. As we described at Section 4.4.5, native applications are: Provenance Manager; File Manager; Credential; Workflow Manager; and the Submission Manager.

When an application is installed at Django, its presentation; business; and data logic layers are automatically built for CRUD (Create, Retrieve, Update and Delete) operations (Section 4.4.4). This is the case of ours built-in applications, which presented a well-defined CRUD workflow, as illustrated in Figure 5.1. Following this flow, for each built-in App we defined, the illustrated workflow was applied given each model's particular entity-relationship.

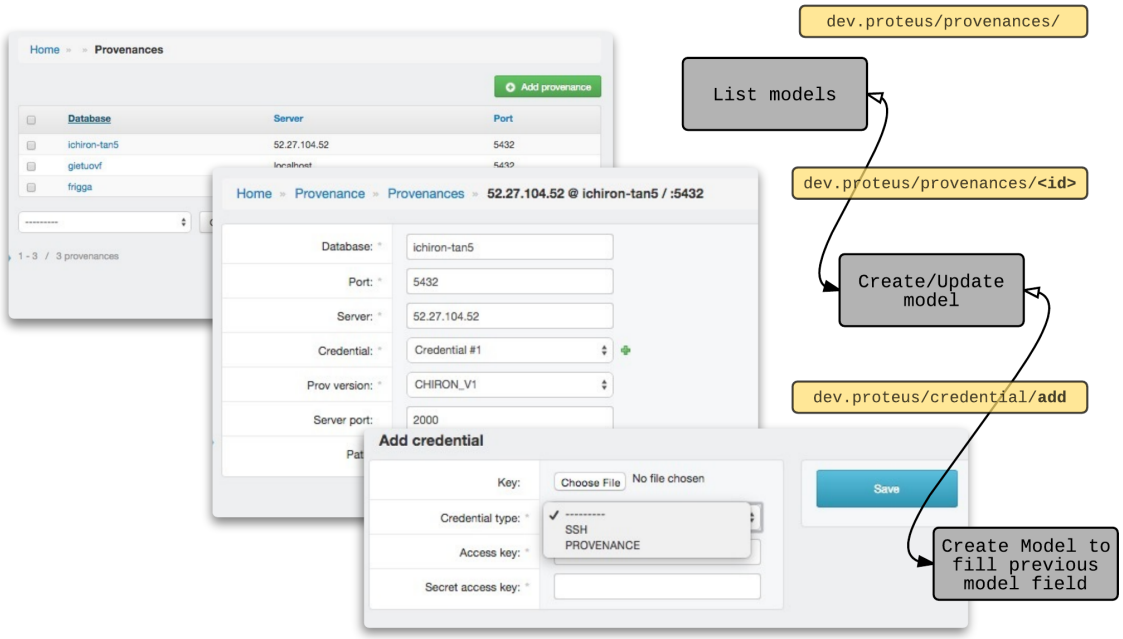

Figure 5.1: Example of a CREATE/UPDATE workflow of a Provenance's model.

5.1.1 Proteus' Access

Proteus, as a web-based platform, can be accessed from everywhere since its server configurations allow for external connection. Thus, once Proteus is deployed, any registered user can sign in by accessing any address beneath the root of its web address. It means that, even when `dev.proteus` is the root, and may present a login form for any sign-out user that attempt to access it, addresses like `dev.proteus/workflows/add` may redirect signout users back to the sign-in form, and later redirects them back to the original URL, if sign-in is successful.

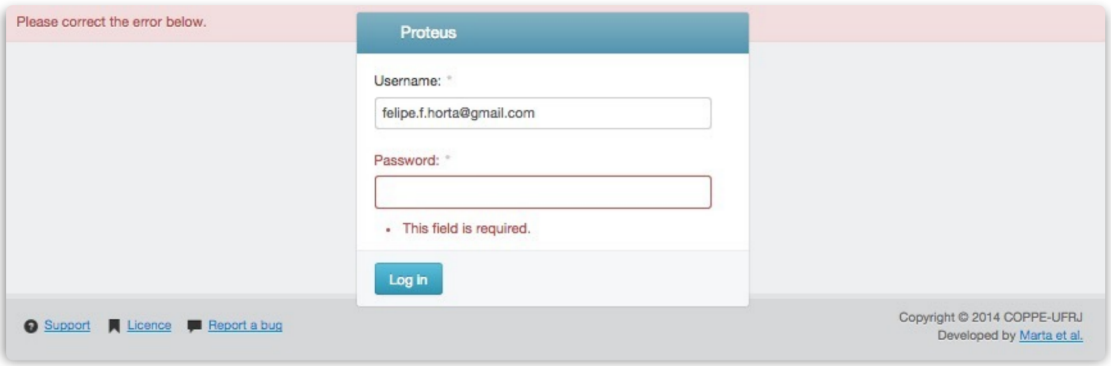

Figure 5.2: Warning of required field at a signin attempt.

Along all Proteus' forms, some fields may be required for a particular purpose. Thus, as shown in Figure 5.2, Proteus handles incomplete forms at the client side before performing a server request. It is possible because even when required fields are defined at application's business logic definitions, these definitions are brought together with the respective `view` prior any request.

5.1.2 Home Screen

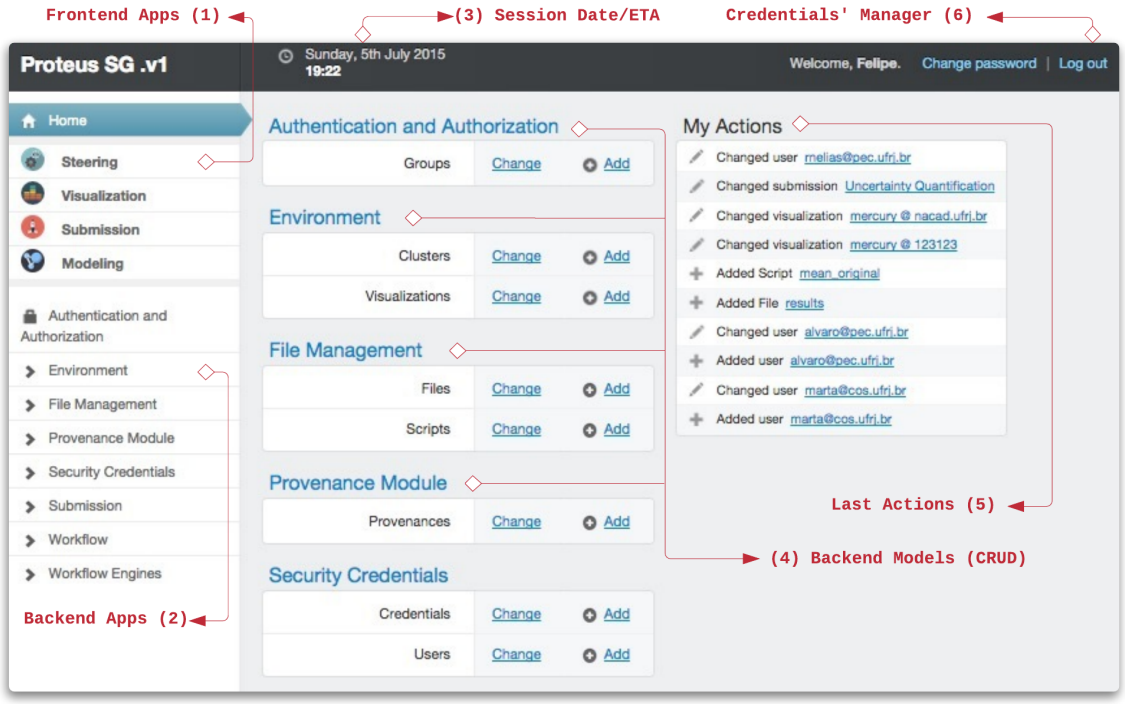

Figure 5.3: Example of Proteus' home screen for a registered user.

Once successfully signed in at Proteus, users are directed to Proteus' home page. At Proteus' Home, the signed in users may find the following widgets (see Figure 5.3):

1. **Frontend Apps:** lists all installed Proteus' Frontend Apps.
2. **Backend Apps:** lists all installed Proteus' Backend Apps.
3. **Session Date/ETA:** monitors user's session ETA and show current date and time already provided by Django.
4. **Backend Models:** lists shortcuts for user's most used Backend Models operations.
5. **Last Actions:** logs for current user's last operations.
6. **Credential Manager:** lists shortcut for user credential's settings and sign out.

Frontend applications' static files, such as HTML, have a limited framed presentation. Such space, in Figure 5.3, is occupied by items 4 and 5. However, frontend applications can apply for popup presentation at startup definitions, which may trigger the web-browser to open applications' `view` in full-frame new separate window (see Code 5.2).

5.1.3 File Manager

Proteus' File Manager helps users to manage local and remote files at its business's layer API. At its automatic generated presentation layer, users may find all files that belong to them, being able to add; edit; or delete files and scripts.

File Manager's API endpoints are able to (i) stablish connections with remote environments downloading files to client machine; (ii) fetch remote files informations (or just check if it exists). Moreover, for each file downloaded, this application also creates a new model, which belong to the requestuser (see Figure 5.4).

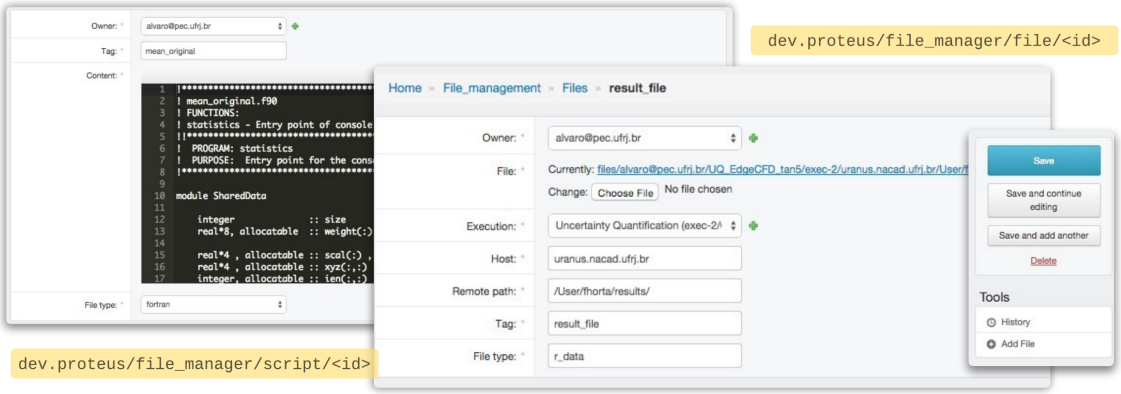

Figure 5.4: Example of File Manager's edition operation.

- Models
  - `File, Script`
- Examples
  - Which scripts belongs to user Y?
  - Which files were generated by Submission X?
  - Which files are results of Workflow B?

5.1.4 Provenance Manager

This application manages provenance's connection information. Provenance models can be shared among users of a such group once are created, but users may need their own credentials. Credentials are unique by users, but can be reused among several provenances. Connections models are used for registering provenance access.

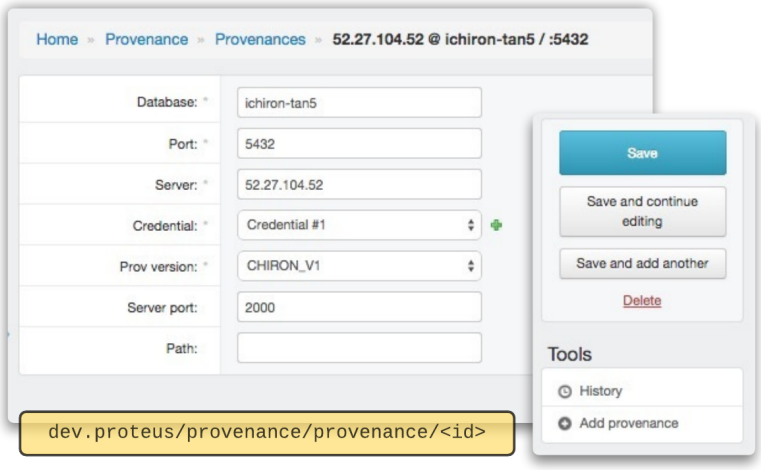

Figure 5.5: Example of Provenance Manager's edition operation.

- Models:
  - `Provenance, Connections`
- Examples
  - Which provenances were created by group A?
  - Which submissions are running in provenance B?
  - How many users have been connecting at provenance C last week?

5.1.5 Credential Manager

Credential Manager App manages permissions among users and their groups. It handles how users belong to groups, and how they manage permissions and connections' credential. Figure 5.6, *i.e.*, shows an example of part of a given group configuration, where Proteus lists all manageable permissions extracted from Proteus' installed Apps. In this way, when users are allowed to, they can manage groups' permissions (Figure 5.6); or overwrite it directly to a single user (Figure 5.7).

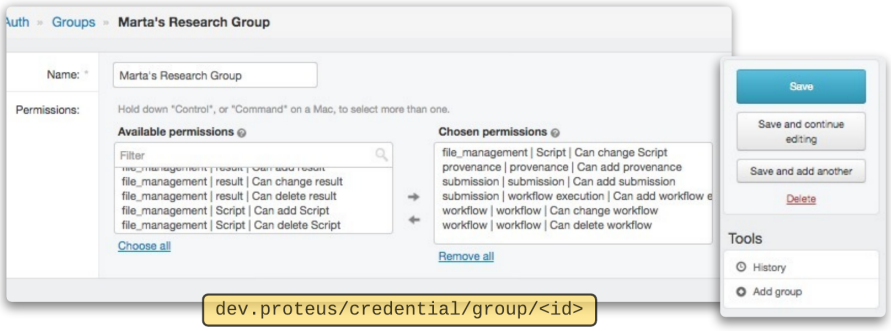

Figure 5.6: Example of Credential Manager's edition over Group's permissions.

- Models:
  - `User, Group, Credential`
- Examples
  - Which users belong to group A?
  - When was the last time user B signed in Proteus?
  - Which users from Group C changed workflow D?

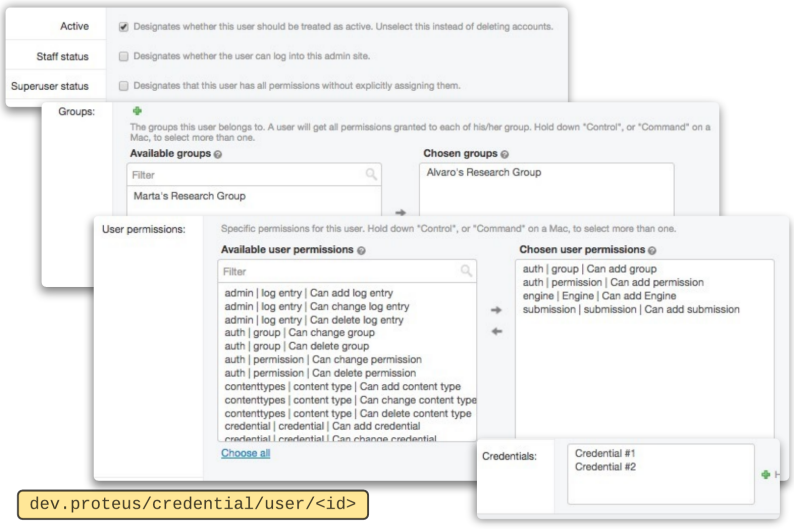

Figure 5.7: Example of Credential Manager's edition over User's permissions.

5.1.6 Workflow Manager

Workflow Manager App defines data-models for workflow related entities, such as activities; relations; and fields. However, besides simple operations, Workflow manager's CRUD interface is not user-friendly given the complexity between workflow's activities, relations and models. As we will see later, even though these applications are essencial by its data logic layer, it has been opening opportunities for other works such as Workflow Modeling, which has been overwriting the current web-form view into a interactive graphical view.

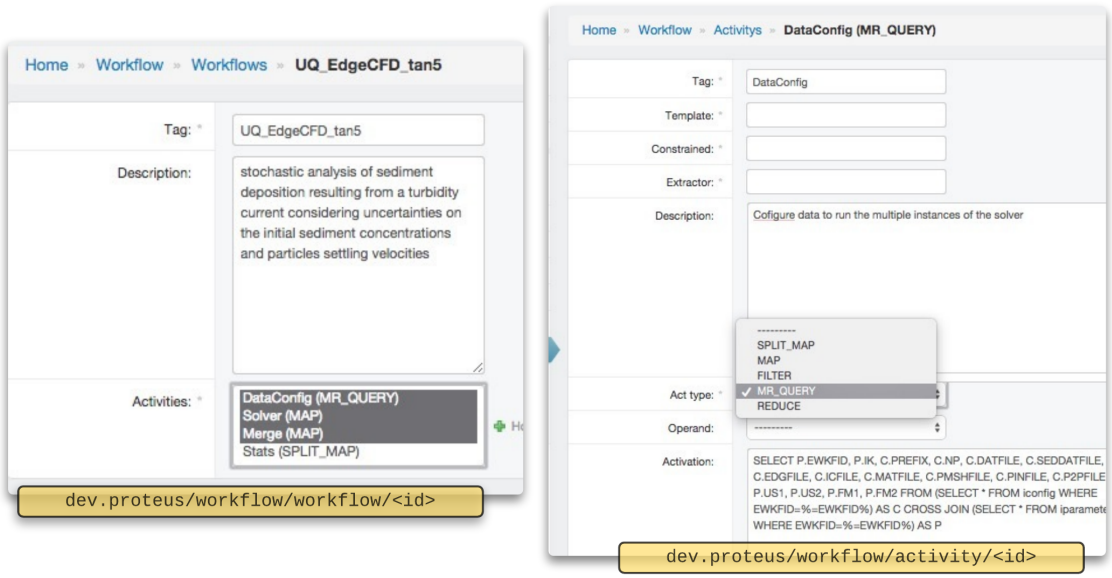

Figure 5.8: Fragment of Workflow Manager's action over a workflow (left) and an activity (right) model .

- Models:
  - `Workflow, Activity, Relation, Fields`
- Examples
  - Which workflows apply activity A at provenance B?
  - Which provenances apply workflows C?

5.1.7 Submission Manager

This app defines several models related to the submission of a workflow using Chiron. Further, Submission Manager can connect to a given provenance and scan looking for workflow models and executions, syncronizing a snapshot of provenance's model on demand. Figure 5.9 shows an example of execution generated by the submission presented at Figure 5.10. Also, this application can perform translations between Chiron XML's files and workflow to submission data-models, in both ways.

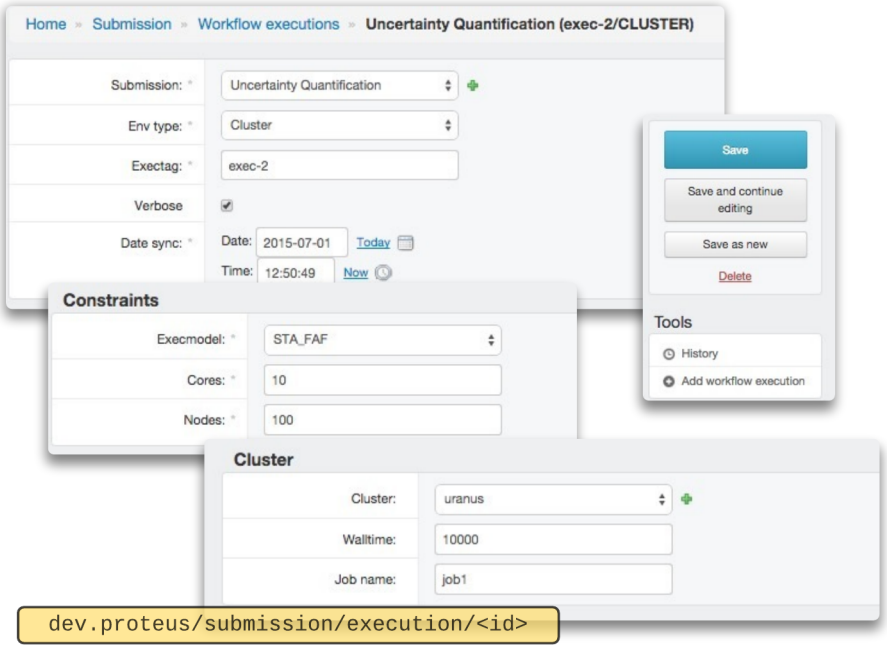

Figure 5.9: Example of Submission Manager's action over an Execution model.

- Models:
  - `Submission, WorkflowExecution, VirtualMachine, NodeConfig`
- Examples
  - How many executions were generated by submission A at provenance B?
  - Which executions were triggered between dates C and D?

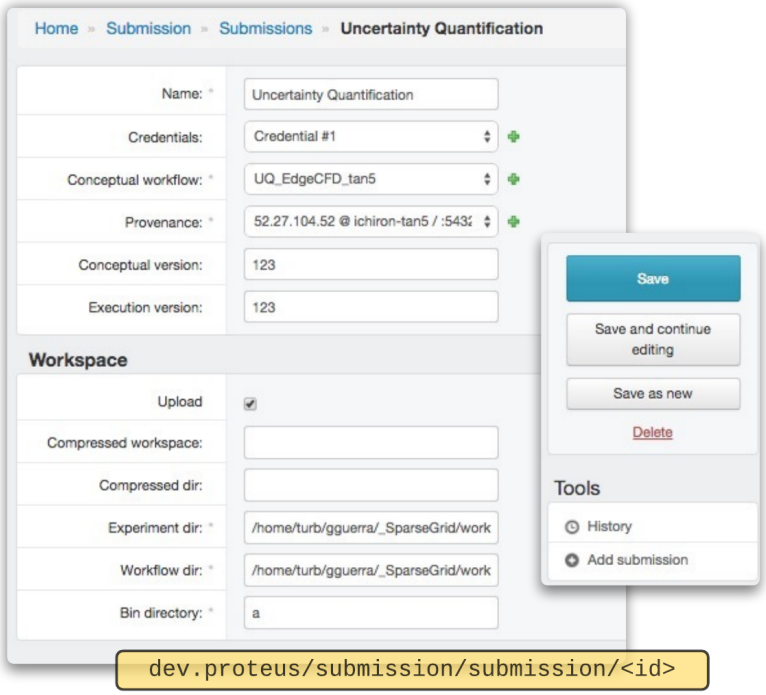

Figure 5.10: Example of Submission Manager's action over a Submission model.

5.2 Prov-vis

With the potential improvements raised at Section 3.3.2 for our UQ case study, we choose a set of features that our first Proteus' application - a provenance-based visualization tool - would perform above the infrastructure we built. Soon, as we needed to choose a shorter name to apply during development, we named it Prov-vis.

Main goal

Prov-vis intended to instrument provenance's exploration, assisting the process described along the SGSC's UQ analysis (3.3.2). This process basically concerned in exploring provenance *in tandem* with the experiment's execution, looking for a convergence criteria along increasing levels of interporlation of a given Sparse grid - which means an increasing number of executions at each level, in exchange (as expected) for more refined results. Thus, with a given criteria condition, scientists decision could be: (i) let experiment continue; (ii) terminate experiment as soon as possible; (iii) or re-execute the experiment again changing input parameters. Therefore, we focused on enhancing such runtime analysis through provenance queries to help scientists to take decisions faster; and easing remote environment connection and instrumentation assisting post-processing tasks that could help taking decisions.

Moreover, as we mentioned before (4.2), PV0's project was used as foundation for Prov-vis application since it already presented UQ target features, such as provenance exploration tools. In this way, we took all of these features and enhanced some of its functionalities at points where it was possible to take advatange of Proteus's API.

> Although based on a UQ case study, once it was simply based on provenance queries, we believe that it can also be applied for other purposes rather than UQ exploration.

Beginning a new application

This application's project started after cloning the last version from its Git versioning repository.

```
git clone git@bitbucket.org:fhorta/proteus-py.git
```

As it aimed at being a Proteus' application, during its development we focused on uncoupling Prov-vis from Proteus, since later we intended to package its first version and serve it as an external application (Section 5.2.5). Thus, as we mentioned at 4.3.1, Prov-vis was built beneath its own package `proteus.frontend.provvis` (see Figure 5.11), and the only configuration we needed prior to coding was to change the file `settings.py`, including such package to the variable `INSTALLED_APPS`.

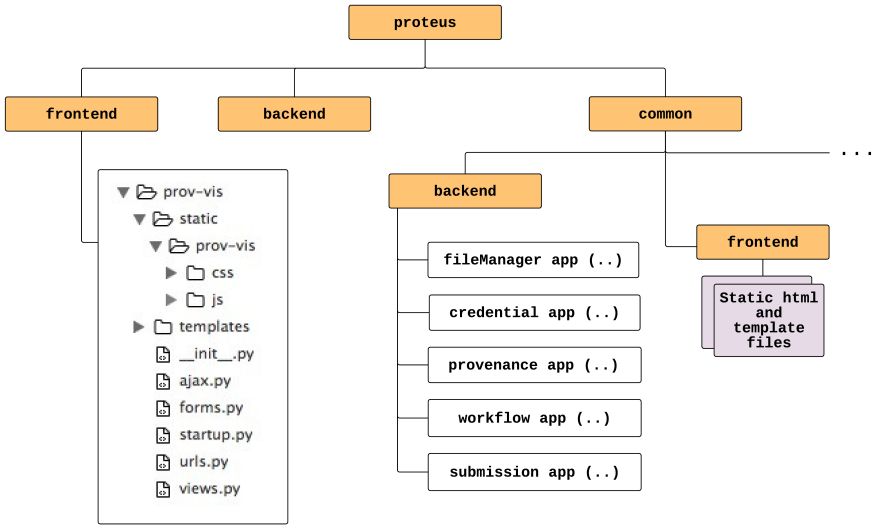

Figure 5.11: Project structure after Prov-Vis development.

In this way, Prov-vis was developed and evaluated using the same provenance generated over our case study. It is worth saying that, even though when this provenance was generated its host machine was located at NACAD (UFRJ), because of connections issues we transfered this database to a Amazon EC2[1] machine, where it stayed until the end of this work. This PostgreSQL database, which was keeping just our target provenance, with 10 executions or our case study, was exceeding the size of 5Gb.

Register

As mentioned at Section 4.4.5, Proteus needs a `startup.py` script, located at each application's root, to perform proper application's static files register during its startup, *i.e* whether application is indexed at main menu or not. In this way, since Prov-vis would also need frontend static pages, with the following script we described its landing page, for submission exploration; and the subsequent one, for the experiment analysis.

Code 5.2: Startup informations for Proteus' Visualization App.

```
# proteus/frontend/provvis/startup.py

from django.core.urlresolvers import reverse_lazy

description = u"""
        Provides experiment Visualization based on Provenance queries (..)
        """

module = {'display_name': u"Provenance Visualization",}

pages = [

    {'identifier': 'vis',                          # main page identifier
    'display_name': u"Visualization",              # page title
```

[1] http://aws.amazon.com/pt/ec2

```
    'description': description,
    'priority': 30,                                    # priority order at main menu
    'icon_class': 'proteus-icon-visualizer-small',     # menu icon filename
    'url': '/vis/',                                    # url which may be addressed

    {'parent': 'vis',
    'priority': 100,
    'identifier': 'vis-submission_list',
    'display_name': u"Submission Filter",
    'url': '/vis/'}                                    # our landing page for
                                                       # browsing submissions
    {'parent': 'vis',
    'priority': 100,
    'identifier': 'vis-analysis',
    'display_name': u"Experiment Analysis",
    'url': '/vis/filter'}

]
```

5.2.1 Provenance Discovery

Cases [**I1**] led us to work on a provenance discovery tool, which could enhance scientists process while connecting and browsing experiments executions. So, laying on the already implemented Proteus' API endpoints, this task could be accomplished by **(i) listing discovered submissions at registered provenances**, using `Provenance App` to scan user's available provenances; next, **(ii) listing all discovered executions** of a given workflow, while checking with `Submission App` whether a given execution's data is syncronized or not with Proteus' database; and finally, **(iii) syncronizing a execution's model with Proteus' database**, reading prospective provenance data with `Provenance App`, and persisting discovered workflow's model with `Workflow App` (see Figure 5.12).

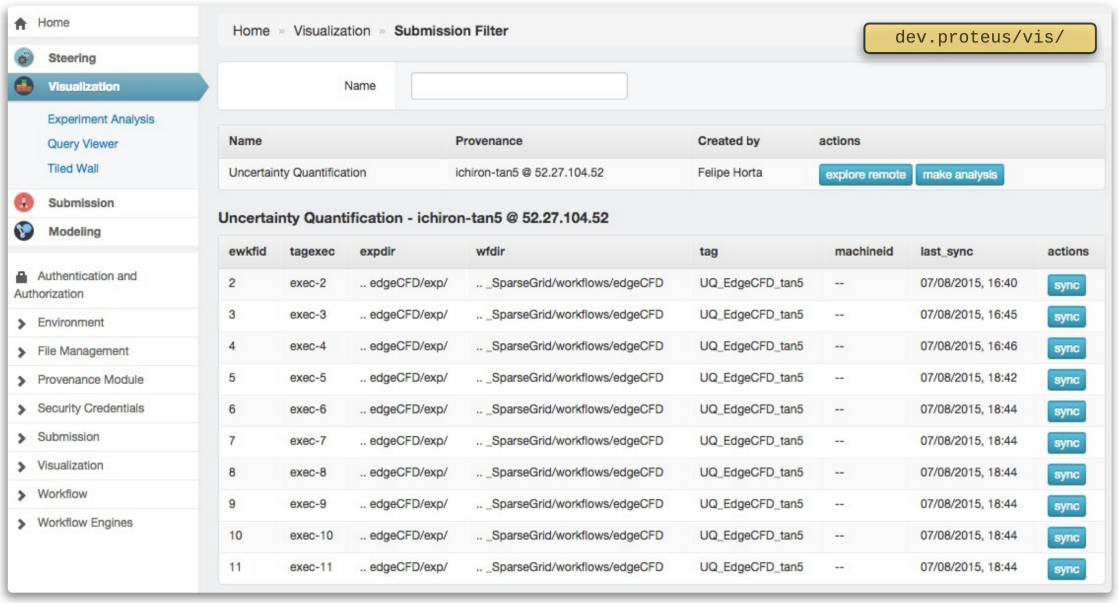

Figure 5.12: Example of provenance discovery using Prov-Vis.

5.2.2 Browsing Provenance

Cases [**I1**] also led us to infer that scientists could save time and have a better view of its experiment, if browsing a given provenance data by its executions; activities; relations and fields given particular conditions. Thus, one more time laying on the already implemented Proteus' API endpoints and following **Provenance Discovery**, this task could accomplished by **(i) allowing users to traverse submission provenance data** - by its executions, activities, relations and fields values - with the aid of `Workflow` and `Submission App` providing all informations needed to build a filter interface; so, at each filter request, `Provenance App` could **(ii) connect and query provenance retrieving filter's results**; additionally, with the help of a new `Search model`, so users could also save their favorite "searchs" for a given submission (see Figure 5.13).

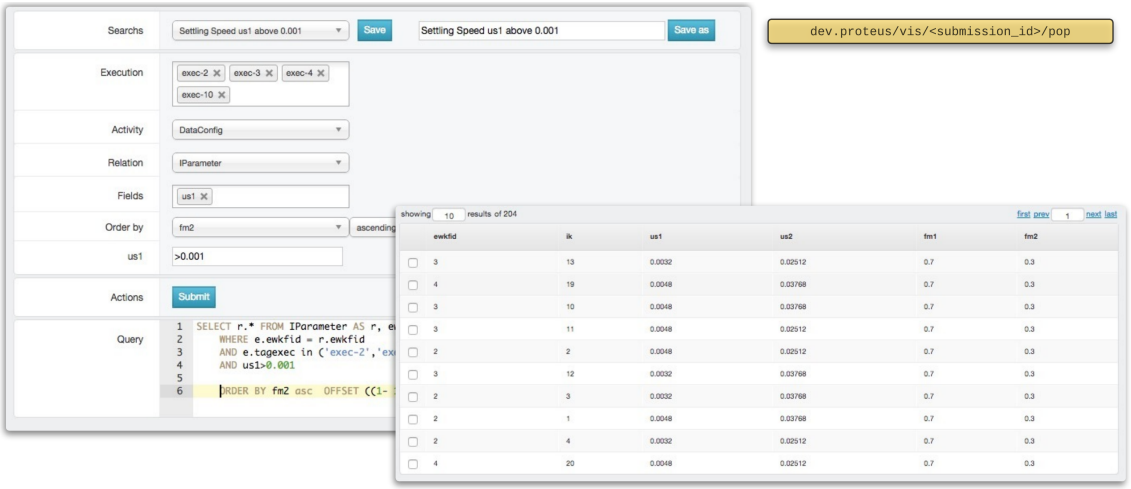

Figure 5.13: Example of provenance browsing applying Prov-Vis filters.

5.2.3 Finding and Downloading Files

The improvement [**I2**] would be, at least in part, successful if scientists could spend less time in the process of acquisition of a given file presented at the execution environment; *i.e.* if it was possible to skip the laborious steps of querying for result's file, and downloading it to a local machine. Further, some files, even presented at provenance data, could have been deleted prior to user's request, following workflow rules sometimes applied to save disk space that could not have been registered. Trying to solve this frustrating experience, the task of recognizing files during the browsing stage and checking if its existence at a given remote machine before attempting to requisition, can be useful.

Following the process of **Browsing Provenance**, the above-described improvements could be accomplished by **(i) identifying file fields within queries results**, which could be done by using `Workflow App` and loolking for each field type, discovering whether it was a file or not (and if positive, building selectable buttons for each field); next, after choosing a given (set of) file results, it was also possible to **(ii) check rather file exists or not** by connecting and querying provenance using `Provenance App`, looking for a particular file

information and checking its existence; at last, **(iii) allowing users to download files** (if feasible given size limits). This last task could be done once `Files App` was able to bring a (set of) file from a remote machine, knowing in beforehand (with the `Submission App`) connections informations, such as particular credentials for a submission's environment (see Figure 5.14).

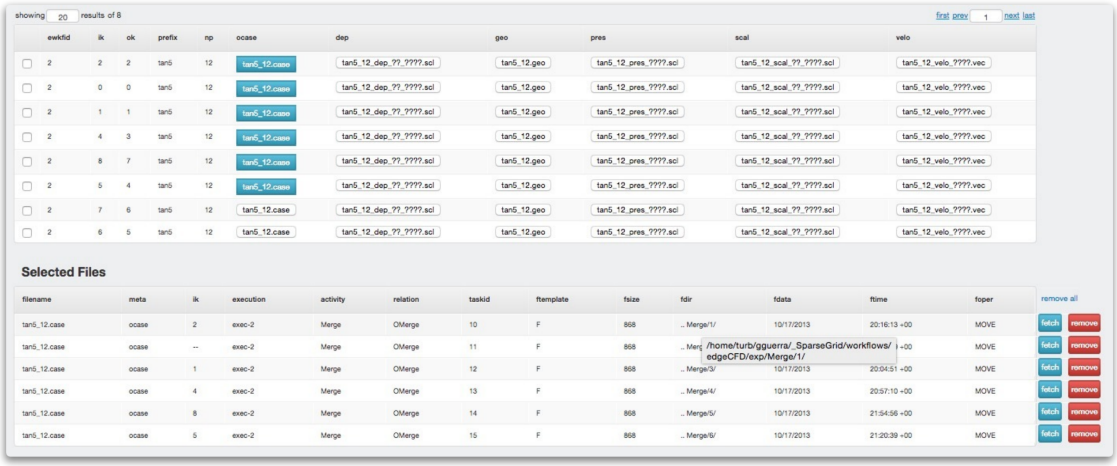

Figure 5.14: Example of finding and downloading files using Prov-Vis.

5.2.4 Post-Processing Results

In addition to previous case, according to **[I3]**, some (set of) files, could oversize the available space at scientist's machine, or simply, ravel the transfer through a given network. In cases like these, some post-processing tasks could be exported to the remote machine, applying desired changes *in situ*. This last useful step would inquire for a mechanism able to gather distributed files in the remote environment, applying such post-processing tasks. Therefore, following the process of **Finding Files**, the above-described improvements could be accomplished by **(i) allowing users to choose and run scripts at remote machines** with the help of `Files App`, sending user's scripts to remote environment with the connections information provided by `Submission App`; next, `File App` was able to **(ii) execute the chosen script with its dependencies**; for finally, **(ii) retrieve results informations, enabling further download** since `Files App` also describes results' information, such as the file path, when files are available (see Figure 5.15).

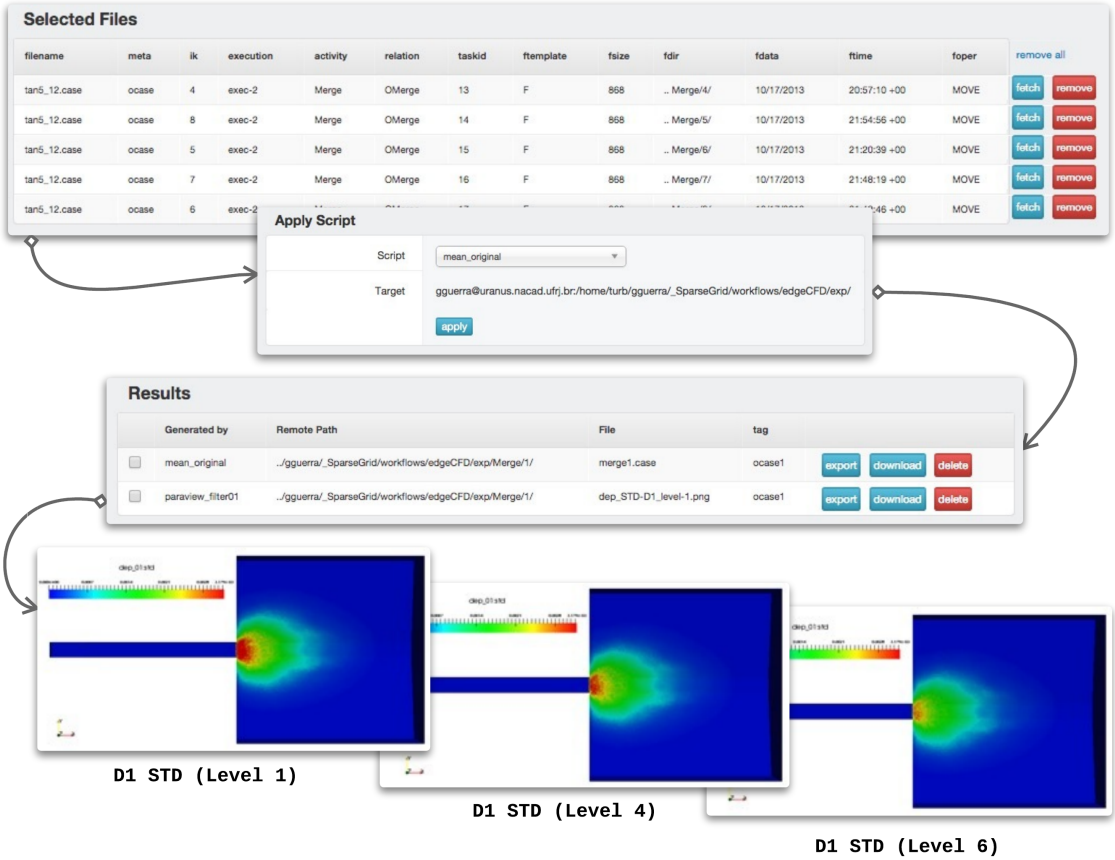

Figure 5.15: Example of post-processing SGSC' data while generating snapshot with Paraview's filters, at remote environment, for a given sparse grid's level.

5.2.5 Packaging

Packaging a Proteus App refers to preparing the app in a specific format that can be easily installed and used; and it behaves as a ordinary Django App. Since we already organized this app uncoupled from Proteus' project; and also given it size, this process wasn't difficult. For this process we also follow a standard Django app packaging tutorial[2].

1. First, we created a parent directory for Prov-Vis, outside of Proteus' project. Then, We named our application by `proteus-provvis`.

```
mkdir ~/tmp/proteus-provvis
mv /proteus/frontend/provvis ~/tmp/proteus-provvis/
```

2. We also created a file proteus-provvis/README.rst with the following contents:

```
.
    =====
    Provenance Visualization
    =====

    Prov-vis is a simple Proteus app to conduct
```

[2]https://docs.djangoproject.com/en/1.8/intro/reusable-apps/

```
    Web-based provenance queries based on Proteus' API.

    Quick start
    -----------

    1. Add "provvis" to your INSTALLED_APPS setting like this::

        INSTALLED_APPS = (
            ...
            'provvis',
        )

    2. Include the provvis URLconf in your project urls.py like this:

        url(r'^vis/', include('provvis.urls')),

    3. Run `python manage.py migrate` to create the provvis models.

    4. Visit http://127.0.0.1:8000/vis/ to query provenance.
.
```

3. Next, we created a `proteus-provvis/LICENSE` file. Choosing a license is beyond the scope of this dissertation, because applications may contain a sort of sensitive contents or third-party softwares. But, as a good practice while distributing software, we choose for BSD License, since Django and many Django-compatible apps are distributed under this license.

4. Following, we created the file `proteus-provvis/setup.py`, which provides details about how to build and install the app. A full explanation of this file is beyond the scope of this work, but the setuptools docs[3] have a good explanation.

```
import os
from setuptools import setup

with open(os.path.join(os.path.dirname(__file__), 'README.rst')) as readme:
    README = readme.read()

# allow setup.py to be run from any path
os.chdir(os.path.normpath(os.path.join(os.path.abspath(__file__), os.pardir)))

setup(
    name='proteus-provvis',
    version='0.1',
    packages=['provvis'],
    include_package_data=True,
    license='BSD License', # example license
    description='A provenance based visualization app to conduct Web-based queries.',
    long_description=README,
    url='http://www.proteus.com/visualizing_experiments',
    author='Felipe Horta',
    author_email='felipe.f.horta@gmail.com',
    classifiers=[
        'Environment :: Web Environment',
        'Framework :: Django',
```

[3] http://pythonhosted.org/setuptools/setuptools.html

```
        'Intended Audience :: Developers',
        'License :: OSI Approved :: BSD License', # example license
        'Operating System :: OS Independent',
        'Programming Language :: Python',
        'Programming Language :: Python :: 2.7',
        'Topic :: Internet :: WWW/HTTP',
        'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
    ],
)
```

5. Since only Python modules and packages are included in a ordinary Python packages by default. To include additional files, such as our static *HTML*; *Javascript*; and *CSS* files, we needed to create a `MANIFEST.in` file. To include the templates; the `README.rst`; and our `LICENSE` file, we created a file `proteus-provvis/MANIFEST.in` with the following contents:

```
include LICENSE
include README.rst
recursive-include provvis/static *
recursive-include provvis/templates *
```

6. Finally, we built our package with `python setup.py sdist`. This creates a directory called `dist` and builds our new package, `proteus-provvis-0.1.tar.gz`.

```
cd ~/tmp/prov-vis/
python setup.py sdist
```

> When choosing a name for our package, we also needed to check out resources like PyPI[4], mentioned at Section 4.3.1 to avoid naming conflicts with existing packages. In this way, we prepended `proteus-` to our module name when creating a package to distribute. This helps others looking for Django apps identify our app as Proteus specific packages. Moreover, application labels (such as "vis" for "proteus-provvis") must be unique in `INSTALLED_APPS`. In this way, we need to avoid using the same label as any of the popular Python packages

5.2.6 Distributing

Since we moved the `prov-vis` original package from our Proteus development branch, it was no longer working. So, with the following help of the `pip` command (PyPI's Package Manager), and adding our new package address "provvis" back to `INSTALLED_APPS` variable at Proteus' settings file, our applications Prov-Vis was back on the fly.

```
pip install ~/tmp/proteus-provvis/dist/proteus-provvis-0.1.tar.gz
```

If the same applications needed to be removed, we could use:

```
pip uninstall proteus-provvis
```

Once we had packaged and tested our Prov-Vis package, it was ready to share. So, following steps could also apply: (i) share package with other collaborator; (ii) upload the package on a Web site; or (ii) post the package on a public repository, such PyPI [5].

[5]https://packaging.python.org/en/latest/

Chapter 6

Conclusions

Large-scale scientific computing often relies on intensive CPU tasks chained through a workflow running on a high-performance environment. In this context, scientists model their workflows for later submission on dedicated execution environments making use of Scientific Workflow Management Systems (SWfMS), which may improve data management on scientific workflows [2]. Unfortunately, when SWfMSs execute experiments as black-boxes, scientists typically find ways of tracking the execution, *i.e* following the evolution of a computation by opening and browsing files commonly spread over a distributed architecture. To track the convergence of given experiment, scientists even try to stage out some data to visualize the workflow execution. However, such process is complex and error prone, mainly because the user has to "guess" the context of this partial result generation once files and their metadata are not connected. So that, this scenario hides potential issues given the interface between scientists; workflows models; experiment data; and the execution environment.

Fortunately, several SWfMSs may make data management on scientific workflows easier by representing and referencing workflow data through provenance [8]. Provenance is a key feature in SWfMS since it allows for keeping track of everything that happens during workflow execution [9, 10] (Section 2.1). Therefore, once data is accessible during execution, scientists should be able to submit high-level and domain-specific provenance queries like: "What are the maximum values for velocity and pressure on a given simulation exploration?" or "According to a simulation exploration, the residual values of pressure are increasing or decreasing?" looking for support while steering the workflow [37, 37, 66, 89] (Section 3.1). In this way, provenance data, if available at runtime, enables a powerful association between workflow metadata and strategic workflow results. With provenance support, it is possible to aggregate different types of metadata to the workflow results, making it easier to analyze and draw conclusions from resulting data [16, 38, 88, 90, 91] (Section 2.3).

Approaching this problem with particular strategies, several independent works have developed SWfMSs covering a set of needed features to support the experiment execution

and analysis. So far, our survey pointed that none of them could allow scientists to make such provenance-enriched analysis during runtime. Further, we notice that one by one was integrated into a Science Gateway System (SGS), that even incorporating new features assisting the experiment's lifecycle, they kept not managing runtime provenance analysis. Nevertheless, if researchers could monitor time-consuming workflow execution at particular simulation exploration points; analyze enriched provenance data at runtime and decide to stop, re-execute, or re-parameter activities; a lot of energies would be saved. However, when executing workflows "offline" they do not allow for runtime analysis and workflow steering even when integrated within well-established Science Gateways Systems.

In this work, we proposed a scientific gateway system, named Proteus, which as demonstrated, supports the lifecycle of large-scale scientific workflow exploring runtime provenance queries. Proteus provides a suitable architecture that supports and integrates with new applications aimed to take advantage of unbound access to experiment's provenance and execution environment. Our work also promotes experiment's reproducibility while managing experiments' metadata and data access from the beginning to the end of its lifecycle. Thus, new applications are capable of arranging and aggregating experiment data at runtime, thanks to the uncoupled access to Chiron or SciCumulus's generated provenance; enhancing experiment's partial analysis, and also promoting cooperative research tool's features.

In order to evaluate Proteus' architecture, we implemented and integrated a new application, dedicated to enhancing experiment's visualization based on features we believe are needed to assist an Uncertainty Quantification scientific workflow exploration, named Prov-vis. This application was successfully built on the Proteus' infrastructure, and evaluated with a Stochastic Analysis of sediment deposition resulted from a turbidity current.

In this case study, since scientists consider uncertainties on the initial sediment concentrations and particles settling velocities, they monitor the statistical moments of the deposition mapping, like other important features of the currents, approximated by a Sparse Grid Stochastic Collocation method that employs a parallel flow solver for the solution of the deterministic problems associated to the grid points. Each grid's point triggers a solver execution, which takes valuable time and energy; and after a particular increasing number of points, the experiment passes to next level of interpolation. Scientists want to increase interpolation level just enough to take decisions over refined partial results. By such analysis, they look for convergence criteria, such as standard deviation and mean of employed settling velocities. This analysis demands laborious tasks, such as applying large hard-coded SQL queries for each request at provenance, and performing heavy data transfer between scientist's machine and execution environment.

Although, in our analysis, the explored experiment was not evaluated *in tandem* with its execution since provenance was generated before the analysis; so, like many other works [89, 91, 92] we could act that experiment was being executed. In this way, we could

explore experiment executions; monitor partials results, and support the post-processing task of applying visualization filters to generate particular views of sensitive data, such as mean and standard deviation, for each interpolation level. Information like those we could retrieve in "runtime" if discovered as fast as possible, are valuable while taking decisions about the experiment's evolution. Therefore, since we could instrument such laborious process, we demonstrated that our solution was able to help scientists to take decisions faster than ordinary methods do, especially for the case study we described.

Additionally, with a precise version of Prov-vis, our Visualization App showed to be capable of being packaged; distributed; and applied again over a Proteus' version, demonstrating that any other standalone application could also be able to port out-of-the-boxes new applications effortlessly. With that, we hope to encourage the development of new applications, enabling new developers to focus energy only on their applications rather than infrastructure or system integrations' challenges.

6.1 Current Collaborations

Proteus was designed to port old and new projects to its architecture, and we have seen that it has been successfully applied. Currently, three insightful collaborations have been developed built on Proteus' infrastructure, as students' final undergrad projects.

- The **Submission App**, which desires to assist Proteus' while launching and monitoring their experiment, has been developed by the student Kaique Rodrigues. So far, this application (i) can translate, in both ways, Chiron's XML models to Proteus's data models; and (ii) launching new experiments at remote environments, by taking advantage of Proteus' remote connections to trigger and monitor Chiron's executions.
- The **Modeling App**, developed by the student Lucas Carneiro, wishes to assist users while modeling their experiments through a rich graphical user interface. This application may (i) bypass the laborious task of editing a Chiron's XML model; (ii) introducing Proteus's users to a better view their experiment. Moreover, this application has been developed using forefront technologies along graphical interface; and the asynchronous communication with Proteus' API.
- The **Steering App**, as we study at Section 3.1 of this work, may bring advances while assisting dynamic interference during execution. Human-in-the-loop features may be add to Proteus while integrating with the Aquiles Project, a dynamic steering engine developed by Jonas Dias (during his Ph.D., 2013). For this development task, we could count with the student Renato Sampaio, which has been interfacing Aquiles and Proteus' API with frontend users.

6.2 Future Works

Supporting any software evolution is a challenging task because crucial architecture's endpoint may vary (even when it shouldn't) from one version to another. Since Chiron is under constant improvement, and it is still walking into a more stable version, changes may happen all the time. Also, considering Proteus as an important tool, which turns Chiron's engine more attractive for users and developers, it is expected that Proteus also provides assistantship during changes releasing. In this way, Proteus could monitor Chiron's development branch, allowing users (in production) or developers (during debugging) to change Chiron's version on the fly. This feature would probably raise some issues while Chiron's API does not offer stability, but some hacks could be done, such as an also versioned Chiron's adaptor for Proteus. Thinking forward in this way, *e.g.* Proteus could enable Chiron's backward-compatibility since workflow data models would be persisted at Proteus' database. So, scientists would not loose (or need to edit) their XML workflow models anymore, even when Chiron updates a new release with crucial improvements.

Another interesting approach to future works could also address other visualization pipelines rather than only a provenance based. Integrating and exploring approaches such as in situ visualization (Section 3.2.2) coupled with dedicated visualization environment could enhance dynamic interference, even when still provenance based. In this way, new Proteus' applications could arise with remote memory systems's uncoupled integrations, *i.e.* using Paraview Coprocessing Library [5, 70] together with enriched provenance data.

Bibliography

[1] MATTOSO, M., WERNER, C., TRAVASSOS, G. H., et al. "Towards Supporting the Life Cycle of Large-scale Scientific Experiments", *International Journal of Business Process Integration and Management*, v. 5, n. 1, pp. 79–92, 2010.

[2] ALTINTAS, I., BERKLEY, C., JAEGER, E., et al. "Kepler: an extensible system for design and execution of scientific workflows". In: *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pp. 423–424. IEEE, 2004. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1311241>.

[3] OINN, T., GREENWOOD, M., ADDIS, M., et al. "Taverna: Lessons in Creating a Workflow Environment for the Life Sciences: Research Articles", *Concurr. Comput. : Pract. Exper.*, v. 18, n. 10, pp. 1067–1100, 2006. ISSN: 1532-0626. doi: 10.1002/cpe.v18:10. Disponível em: <http://dx.doi.org/10.1002/cpe.v18:10>.

[4] CHURCHES, D., GOMBAS, G., HARRISON, A., et al. "Programming Scientific and Distributed Workflow with Triana Services: Research Articles", *Concurr. Comput. : Pract. Exper.*, v. 18, n. 10, pp. 1021–1037, 2006. ISSN: 1532-0626. doi: 10.1002/cpe.v18:10. Disponível em: <http://dx.doi.org/10.1002/cpe.v18:10>.

[5] FABIAN, N., MORELAND, K., THOMPSON, D., et al. "The ParaView Coprocessing Library: A Scalable, General Purpose In Situ Visualization Library". In: *IEEE Symposium on Large-Scale Data Analysis and Visualization*, 2011. doi: 10.1109/LDAV.2011.6092322.

[6] KLASKY, S., ABBASI, H., LOGAN, J., et al. "In situ data processing for extreme-scale computing", *Scientific Discovery through Advanced Computing Program (SciDAC'11)*, 2011. Disponível em: <http://pasl.eng.auburn.edu/pubs/scidac11-adios-insitu.pdf>.

[7] MORELAND, K., HERELD, M., PAPKA, M. E., et al. "Examples of In Transit visualization". p. 1. ACM Press, 2011. ISBN: 9781450311304. doi: 10.1145/2110205.2110207. Disponível em: <http://dl.acm.org/citation.cfm?doid=2110205.2110207>.

[8] DEELMAN, E., MEHTA, G., SINGH, G., et al. "Pegasus: Mapping Large-Scale Workflows to Distributed Resources". In: *Workflows for e-Science*, pp. 376–394. Springer, 2007. Disponível em: <http://dx.doi.org/10.1007/978-1-84628-757-2_23>.

[9] FREIRE, J., KOOP, D., SANTOS, E., et al. "Provenance for Computational Tasks: A Survey", *Computing in Science and Engineering, v.10*, , n. 3, pp. 11–21, 2008. Disponível em: <http://portal.acm.org/citation.cfm?id=1399195&jmp=cit&coll=&dl=>.

[10] ALTINTAS, I., BARNEY, O., JAEGER-FRANK, E. "Provenance Collection Support in the Kepler Scientific Workflow System". In: Hutchison, D., Kanade, T., Kittler, J., et al. (Eds.), *Provenance and Annotation of Data*, v. 4145, pp. 118–132, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN: 978-3-540-46302-3, 978-3-540-46303-0. Disponível em: <http://link.springer.com/10.1007/11890850_14>.

[11] OGASAWARA, E., DIAS, J., SILVA, V., et al. "Chiron: a parallel engine for algebraic scientific workflows", *Concurrency and Computation: Practice and Experience*, v. 25, n. 16, pp. 2327–2341, 2013. ISSN: 15320626. doi: 10.1002/cpe.3032. Disponível em: <http://doi.wiley.com/10.1002/cpe.3032>.

[12] OLIVEIRA, D., OGASAWARA, E., BAIÃO, F., et al. "SciCumulus: A Lightweight Cloud Middleware to Explore Many Task Computing Paradigm in Scientific Workflows". In: *3rd International Conference on Cloud Computing*, CLOUD '10, pp. 378–385, Washington, DC, USA, 2010. IEEE Computer Society. ISBN: 978-0-7695-4130-3. doi: 10.1109/CLOUD.2010.64. Disponível em: <http://dx.doi.org/10.1109/CLOUD.2010.64>.

[13] LITZKOW, M., LIVNY, M., MUTKA, M. "Condor-a hunter of idle workstations". pp. 104–111. IEEE Comput. Soc. Press, 1988. ISBN: 0-8186-0865-X. doi: 10.1109/DCS.1988.12507. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=12507>.

[14] CALLAHAN, S. P., FREIRE, J., SANTOS, E., et al. "VisTrails: visualization meets data management". In: *SIGMOD International Conference on Management of Data*, pp. 745–747, Chicago, Illinois, USA, 2006. ACM. ISBN: 1-59593-434-0. doi: 10.1145/1142473.1142574. Disponível em: <http://portal.acm.org/citation.cfm?id=1142473.1142574>.

[15] CHIRIGATI, F., SHASHA, D., FREIRE, J. "Packing Experiments for Sharing and Publication". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pp. 977–980, New York, NY, USA, 2013. ACM. ISBN: 978-1-4503-2037-5. doi: 10.1145/2463676.2465269. Disponível em: <http://doi.acm.org/10.1145/2463676.2465269>.

[16] SILVA, C. T., FREIRE, J., CALLAHAN, S. P. "Provenance for Visualizations: Reproducibility and Beyond", *Computing in Science & Engineering*, v. 9, n. 5, pp. 82–89, 2007. ISSN: 1521-9615. doi: 10.1109/MCSE.2007.106. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4293741>.

[17] WILKINS-DIEHR, N., GANNON, D., KLIMECK, G., et al. "TeraGrid Science Gateways and Their Impact on Science", *Computer*, v. 41, n. 11, pp. 32–41, 2008. ISSN: 0018-9162. doi: 10.1109/MC.2008.470. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4668681>.

[18] NGUYEN, H., ABRAMSON, D., KIPOUROS, T. "The WorkWays Problem Solving Environment", *Procedia Computer Science*, v. 29, pp. 2284–2294, 2014. ISSN: 18770509. doi: 10.1016/j.procs.2014.05.213. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S1877050914003901>.

[19] CURCIN, V., GHANEM, M. "Scientific workflow systems - can one size fit all?" pp. 1–9. IEEE, 2008. ISBN: 978-1-4244-2694-2. doi: 10.1109/CIBEC.2008.4786077. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4786077>.

[20] NGUYEN, H. A., ABRAMSON, D., KIPOROUS, T., et al. "WorkWays: interacting with scientific workflows". In: *Proceedings of the 9th Gateway Computing Environments Workshop*, pp. 21–24. IEEE Press, 2014. Disponível em: <http://dl.acm.org/citation.cfm?id=2690893>.

[21] GIARDINE, B., RIEMER, C., HARDISON, R. C., et al. "Galaxy: a platform for interactive large-scale genome analysis", *Genome research*, v. 15, n. 10, pp. 1451–1455, 2005. Disponível em: <http://genome.cshlp.org/content/15/10/1451.short>.

[22] ESWARAN, S., VECCHIO, D. D., WASSON, G., et al. "Adapting and Evaluating Commercial Workflow Engines for e-Science". In: *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, E-SCIENCE '06, pp. 20–, Washington, DC, USA, 2006. IEEE Computer Society. ISBN: 0-7695-2734-5. doi: 10.1109/E-SCIENCE.2006.18. Disponível em: <http://dx.doi.org/10.1109/E-SCIENCE.2006.18>.

[23] ALTINTAS, I., BERKLEY, C., JAEGER, E., et al. "Kepler: an extensible system for design and execution of scientific workflows". In: *Scientific and Statistical Database Management*, pp. 423–424, Greece, 2004. ISBN: 1099-3371. doi: 10.1109/SSDM.2004.1311241.

[24] OINN, T., ADDIS, M., FERRIS, J., et al. "Taverna: a tool for the composition and enactment of bioinformatics workflows", *Bioinformatics*, v. 20, pp. 3045–3054, 2004.

[25] ZHAO, Y., HATEGAN, M., CLIFFORD, B., et al. "Swift: Fast, Reliable, Loosely Coupled Parallel Computation". In: *3rd IEEE World Congress on Services*, pp. 206, 199, Salt Lake City, USA, 2007. Disponível em: <http://dx.doi.org/10.1109/SERVICES.2007.63>.

[26] ALTINTAS, I., LIN, A. W., CHEN, J., et al. "CAMERA 2.0: A Data-centric Metagenomics Community Infrastructure Driven by Scientific Workflows". pp. 352–359. IEEE, 2010. ISBN: 978-1-4244-8199-6. doi: 10.1109/SERVICES.2010.89. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5577257>.

[27] MATES, P., SANTOS, E., FREIRE, J., et al. "Crowdlabs: Social analysis and visualization for the sciences". In: *Scientific and Statistical Database Management*, pp. 555–564. Springer, 2011. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-642-22351-8_38>.

[28] ROMANO, P., BARTOCCI, E., BERTOLINI, G., et al. "Biowep: a workflow enactment portal for bioinformatics applications", *BMC Bioinformatics*, v. 8, n. Suppl 1, pp. S19, 2007. ISSN: 14712105. doi: 10.1186/1471-2105-8-S1-S19. Disponível em: <http://www.biomedcentral.com/1471-2105/8/S1/S19>.

[29] SINGH, G., DEELMAN, E., MEHTA, G., et al. "The pegasus portal: web based grid computing". In: *Proceedings of the 2005 ACM symposium on Applied computing*, pp. 680–686. ACM, 2005. Disponível em: <http://dl.acm.org/citation.cfm?id=1066834>.

[30] WU, W., URAM, T., WILDE, M., et al. "Accelerating science gateway development with Web 2.0 and Swift". pp. 1–7. ACM Press, 2010. ISBN: 9781605588186. doi: 10.1145/1838574.1838597. Disponível em: <http://portal.acm.org/citation.cfm?doid=1838574.1838597>.

[31] NGUYEN, H., ABRAMSON, D. "WorkWays: Interactive workflow-based science gateways". In: *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pp. 1–8. IEEE, 2012. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6404428>.

[32] ABRAMSON, D., ENTICOTT, C., ALTINAS, I. "Nimrod/K: Towards massively parallel dynamic Grid workflows". pp. 1–11. IEEE, 2008. ISBN: 978-1-4244-2834-2. doi: 10.1109/SC.2008.5215726. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5215726>.

[33] FREIRE, J., KOOP, D., CHIRIGATI, F., et al. "Reproducibility Using VisTrails". In: *Implementing Reproducible Research*, Chapman &Hall/CRC The R Series. Chapman and Hall/CRC, 2014. ISBN: 978-1-4665-6159-5. Disponível em: <http://www.crcnetbase.com/doi/abs/10.1201/b16868-4>.

[34] DONOHO, D. L. "An invitation to reproducible computational research", *Biostatistics*, v. 11, n. 3, pp. 385–388, 2010. ISSN: 1465-4644, 1468-4357. doi: 10.1093/biostatistics/kxq028. Disponível em: <http://biostatistics.oxfordjournals.org/cgi/doi/10.1093/biostatistics/kxq028>.

[35] DONOHO, D. L., MALEKI, A., RAHMAN, I. U., et al. "Reproducible research in computational harmonic analysis", *Computing in Science & Engineering*, v. 11, n. 1, pp. 8–18, 2009. Disponível em: <http://scitation.aip.org/content/aip/journal/cise/11/1/10.1109/MCSE.2009.15>.

[36] BARTOCCI, E., CORRADINI, F., MERELLI, E., et al. "BioWMS: a web-based Workflow Management System for bioinformatics", *BMC bioinformatics*, v. 8 Suppl 1, pp. S2, 2007. ISSN: 1471-2105. doi: 10.1186/1471-2105-8-S1-S2.

[37] MATTOSO, M., OCAÑA, K., HORTA, F., et al. "User-steering of HPC workflows: state-of-the-art and future directions". In: *Proceedings of the 2nd ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, p. 4. ACM, 2013. Disponível em: <http://dl.acm.org/citation.cfm?id=2499900>.

[38] HORTA, F., DIAS, J., ELIAS, R., et al. "Prov-Vis: Large-Scale Scientific Data Visualization using Provenance." SC'13, Denver, CO, 2013. Disponível em: <http://sc13.supercomputing.org/sites/default/files/PostersArchive/tech_posters/post281s2-file3.pdf>.

[39] "OpenFOAM User Guide: 2.3 Breaking of a dam". Disponível em: <http://cfd.direct/openfoam/user-guide/dambreak/>.

[40] DIAS, J., GUERRA, G., ROCHINHA, F., et al. "Data-centric Iteration in Dynamic Workflows", *Future Gener. Comput. Syst.*, v. 46, n. C, pp. 114–126, 2015. ISSN: 0167-739X. doi: 10.1016/j.future.2014.10.021. Disponível em: <http://dx.doi.org/10.1016/j.future.2014.10.021>.

[41] OGASAWARA, E., PAULINO, C., MURTA, L., et al. "Experiment Line: Software Reuse in Scientific Workflows". In: Winslett, M. (Ed.), *Scientific and Statistical Database Management*, n. 5566, Lecture Notes in Computer Science, pp. 264–272. Springer Berlin Heidelberg, 2009. ISBN: 978-3-642-02278-4, 978-3-642-02279-1. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-642-02279-1_20>.

[42] BOSCH, J. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. New York, NY, USA, ACM Press/Addison-Wesley Publishing Co., 2000. ISBN: 0-201-67494-7.

[43] NORTHROP, L. M. "SEI's Software Product Line Tenets", *IEEE Softw.*, v. 19, n. 4, pp. 32–40, 2002. ISSN: 0740-7459. doi: 10.1109/MS.2002.1020285. Disponível em: <http://dx.doi.org/10.1109/MS.2002.1020285>.

[44] COUVARES, P., KOSAR, T., ROY, A., et al. "Workflow management in condor". In: *Workflows for e-Science*, pp. 357–375. Springer, 2007. Disponível em: <http://link.springer.com/chapter/10.1007/978-1-84628-757-2_22>.

[45] RAICU, I., ZHAO, Y., DUMITRESCU, C., et al. "Falkon: a Fast and Light-weight tasK executiON framework". In: *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, 2007. SC '07*, pp. 1–12, 2007. doi: 10.1145/1362622.1362680.

[46] SPOONER, D. P. "Performance-Aware Workflow Management for Grid Computing", *The Computer Journal*, v. 48, n. 3, pp. 347–357, 2005. ISSN: 0010-4620, 1460-2067. doi: 10.1093/comjnl/bxh090. Disponível em: <http://comjnl.oupjournals.org/cgi/doi/10.1093/comjnl/bxh090>.

[47] WELCH, V., SIEBENLIST, F., FOSTER, I., et al. "Security for grid services". In: *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*, pp. 48–57. IEEE, 2003. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1210015>.

[48] STAPLES, G. "TORQUE Resource Manager". In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. ISBN: 0-7695-2700-0. doi: 10.1145/1188455.1188464. Disponível em: <http://doi.acm.org/10.1145/1188455.1188464>.

[49] OGASAWARA, E., DE OLIVEIRA, D., CHIRIGATI, F., et al. "Exploring Many Task Computing in Scientific Workflows". In: *Proceedings of the 2Nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, pp. 2:1–2:10, New York, NY, USA, 2009. ACM. ISBN: 978-1-60558-714-1. doi: 10.1145/1646468.1646470. Disponível em: <http://doi.acm.org/10.1145/1646468.1646470>.

[50] STEVENS, R., ZHAO, J., GOBLE, C. "Using provenance to manage knowledge of in silico experiments", *Briefings in Bioinformatics*, v. 8, n. 3, pp. 183–194, 2007. ISSN: 1467-5463. doi: 10.1093/bib/bbm015.

[51] POPPER, K. R. *Conjectures and Refutations: The Growth of Scientific Knowledge*. Routledge, 1963.

[52] IOANNIDIS, Y. E., WONG, E. "Query Optimization by Simulated Annealing". In: *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, pp. 9–22, New York, NY, USA, 1987. ACM. ISBN: 0-89791-236-5. doi: 10.1145/38713.38722. Disponível em: <http://doi.acm.org/10.1145/38713.38722>.

[53] MOREAU, L., FREIRE, J., FUTRELLE, J., et al. "The open provenance model: An overview". In: *Provenance and Annotation of Data and Processes*, pp. 323–

326. Springer, 2008. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-540-89965-5_31>.

[54] CHAMBERLIN, D. "XQuery: A Query Language for XML". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pp. 682–682, New York, NY, USA, 2003. ACM. ISBN: 1-58113-634-X. doi: 10.1145/872757.872877. Disponível em: <http://doi.acm.org/10.1145/872757.872877>.

[55] CHAN, P., ABRAMSON, D. "A Programming Framework for Incremental Data Distribution in Iterative Applications". In: *International Symposium on Parallel and Distributed Processing with Applications, 2008. ISPA '08*, pp. 244–251, 2008. doi: 10.1109/ISPA.2008.105.

[56] CIRNE, W., BRASILEIRO, F., COSTA, L., et al. "Scheduling in Bag-of-Task grids: the PAUA case". In: *16th Symposium on Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004*, pp. 124–131, 2004. doi: 10.1109/SBAC-PAD.2004.37.

[57] WOZNIAK, J. M., ARMSTRONG, T. G., WILDE, M., et al. "Swift/T: Large-Scale Application Composition via Distributed-Memory Dataflow Processing". pp. 95–102. IEEE, 2013. ISBN: 978-0-7695-4996-5, 978-1-4673-6465-2. doi: 10.1109/CCGrid.2013.99. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6546066>.

[58] OGASAWARA, E., DE OLIVEIRA, D., CHIRIGATI, F., et al. "Exploring many task computing in scientific workflows". In: *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, p. 2. ACM, 2009. Disponível em: <http://dl.acm.org/citation.cfm?id=1646470>.

[59] DEAN, J., GHEMAWAT, S. "MapReduce: Simplified Data Processing on Large Clusters", *Commun. ACM*, v. 51, n. 1, pp. 107–113, 2008. ISSN: 0001-0782. doi: 10.1145/1327452.1327492. Disponível em: <http://doi.acm.org/10.1145/1327452.1327492>.

[60] OLSTON, C., SETH, S., TIAN, C., et al. "Nova: continuous Pig/Hadoop workflows". p. 1081. ACM Press, 2011. ISBN: 9781450306614. doi: 10.1145/1989323.1989439. Disponível em: <http://portal.acm.org/citation.cfm?doid=1989323.1989439>.

[61] ISLAM, M., HUANG, A. K., BATTISHA, M., et al. "Oozie: towards a scalable workflow management system for Hadoop". In: *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, p. 4. ACM, 2012. Disponível em: <http://dl.acm.org/citation.cfm?id=2443420>.

[62] DONGARRA, J., HITTINGER, J., BELL, J. *Applied Mathematics Research for Exascale Computing*. Relatório técnico, DOE/ASCR - U.S. Department of Energy, 2014.

[63] "Computational fluid dynamics". apr, 2015. Disponível em: <http://en.wikipedia.org/w/index.php?title=Computational_fluid_dynamics&oldid=657168413>. Page Version ID: 657168413bibtex: 2015.

[64] MATTOSO, M., DIAS, J., OCAÑA, K. A., et al. "Dynamic steering of HPC scientific workflows: A survey", *Future Generation Computer Systems*, v. 46, pp. 100–113, 2015. ISSN: 0167739X. doi: 10.1016/j.future.2014.11.017. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S0167739X14002519>.

[65] MORELAND, K. "A Survey of Visualization Pipelines", *IEEE Transactions on Visualization and Computer Graphics*, v. 19, n. 3, pp. 367–378, 2013. ISSN: 1077-2626. doi: 10.1109/TVCG.2012.133. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6212499>.

[66] GUERRA, G., ROCHINHA, F. A., ELIAS, R., et al. "Uncertainty Quantification in Computational Predictive Models for Fluid Dynamics Using Workflow Management Engine", *International Journal for Uncertainty Quantification*, v. 2, n. 1, pp. 53–71, 2012. ISSN: 2152-5080. doi: 10.1615/Int.J.UncertaintyQuantification.v2.i1.50. Disponível em: <http://www.dl.begellhouse.com/journals/52034eb04b657aea,69f226067bce0f5b,1b427aac4a956792.html>.

[67] OCAÑA, K. A. C. S., OLIVEIRA, D., OGASAWARA, E., et al. "SciPhy: A Cloud-Based Workflow for Phylogenetic Analysis of Drug Targets in Protozoan Genomes". In: Norberto de Souza, O., Telles, G. P., Palakal, M. (Eds.), *Advances in Bioinformatics and Computational Biology*, v. 6832, pp. 66–70, Berlin, Heidelberg, 2011. Springer Berlin. ISBN: 978-3-642-22824-7. Disponível em: <http://www.springerlink.com/index/10.1007/978-3-642-22825-4_9>.

[68] GIL, Y., DEELMAN, E., ELLISMAN, M., et al. "Examining the Challenges of Scientific Workflows", *Computer*, v. 40, n. 12, pp. 24–32, 2007. ISSN: 0018-9162. doi: 10.1109/MC.2007.421.

[69] LORENSEN, W. E., CLINE, H. E. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pp. 163–169, New York, NY, USA, 1987. ACM. ISBN: 0-89791-227-6. doi: 10.1145/37401.37422. Disponível em: <http://doi.acm.org/10.1145/37401.37422>.

[70] BIDDISCOMBE, J., SOUMAGNE, J., OGER, G., et al. "Parallel Computational Steering for HPC Applications Using HDF5 Files in Distributed Shared Mem-

ory", *IEEE Transactions on Visualization and Computer Graphics*, v. 18, n. 6, pp. 852–864, 2012. ISSN: 1077-2626. doi: 10.1109/TVCG.2012.63.

[71] JANKUN-KELLY, T., MA, K.-L., GERTZ, M. "A model for the visualization exploration process". In: *IEEE Visualization, 2002. VIS 2002*, pp. 323–330, 2002. doi: 10.1109/VISUAL.2002.1183791.

[72] BAVOIL, L., CALLAHAN, S., CROSSNO, P., et al. "VisTrails: Enabling Interactive Multiple-View Visualizations". pp. 135–142. IEEE, 2005. ISBN: 0-7803-9462-3. doi: 10.1109/VISUAL.2005.1532788. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1532788>.

[73] ELLKVIST, T., KOOP, D., ANDERSON, E. W., et al. "Using Provenance to Support Real-Time Collaborative Design of Workflows". In: Freire, J., Koop, D., Moreau, L. (Eds.), *Provenance and Annotation of Data and Processes*, v. 5272, pp. 266–279, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN: 978-3-540-89964-8, 978-3-540-89965-5. Disponível em: <http://link.springer.com/10.1007/978-3-540-89965-5_27>.

[74] SCHEIDEGGER, C., VO, H., KOOP, D., et al. "Querying and Creating Visualizations by Analogy", *IEEE Transactions on Visualization and Computer Graphics*, v. 13, n. 6, pp. 1560–1567, 2007. ISSN: 1077-2626. doi: 10.1109/TVCG.2007.70584. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4376187>.

[75] KOOP, D., SCHEIDEGGER, C., CALLAHAN, S., et al. "VisComplete: Automating Suggestions for Visualization Pipelines", *IEEE Transactions on Visualization and Computer Graphics*, v. 14, n. 6, pp. 1691–1698, 2008. ISSN: 1077-2626. doi: 10.1109/TVCG.2008.174. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4658192>.

[76] MACCORMICK, B., DEFANTI, T., BROWN, M. "Visualization in Scientific Computing". Computer Graphics, ACM Siggraph, 1987.

[77] CHILDS, H. "Architectural challenges and solutions for petascale postprocessing", *Journal of Physics: Conference Series*, v. 78, n. 1, pp. 012012, 2007. ISSN: 1742-6596. doi: 10.1088/1742-6596/78/1/012012. Disponível em: <http://iopscience.iop.org/1742-6596/78/1/012012>.

[78] ROSS, R. B., PETERKA, T., SHEN, H.-W., et al. "Visualization and parallel I/O at extreme scale", *Journal of Physics: Conference Series*, v. 125, pp. 012099, 2008. ISSN: 1742-6596. doi: 10.1088/1742-6596/125/1/012099. Disponível em: <http://stacks.iop.org/1742-6596/125/i=1/a=012099?key=crossref.5eba184b16b9e1a86b37d92687f6beda>.

[79] YU, H., WANG, C., GROUT, R., et al. "In Situ Visualization for Large-Scale Combustion Simulations", *IEEE Computer Graphics and Applications*, v. 30, n. 3, pp. 45–57, 2010. ISSN: 0272-1716. doi: 10.1109/MCG.2010.55.

[80] GAMELL, M., MCCORMICK, P., PAKIN, S., et al. "Exploring power behaviors and trade-offs of in-situ data analytics". pp. 1–12. ACM Press, 2013. ISBN: 9781450323789. doi: 10.1145/2503210.2503303. Disponível em: <http://dl.acm.org/citation.cfm?doid=2503210.2503303>.

[81] ZHENG, F., ABBASI, H., CAO, J., et al. "In-situ I/O processing: a case for location flexibility". In: *Proceedings of the sixth workshop on Parallel Data Storage*, pp. 37–42. ACM, 2011. Disponível em: <http://dl.acm.org/citation.cfm?id=2159362>.

[82] SACKS, J., WELCH, W. J., MITCHELL, T. J., et al. "Design and Analysis of Computer Experiments", *Statistical Science*, v. 4, n. 4, pp. 409–423, 1989. ISSN: 08834237. doi: 10.2307/2245858. Disponível em: <http://dx.doi.org/10.2307/2245858>.

[83] IMAN, R. L., HELTON, J. C. "An Investigation of Uncertainty and Sensitivity Analysis Techniques for Computer Models", *Risk Analysis*, v. 8, n. 1, pp. 71–90, 1988. ISSN: 0272-4332, 1539-6924. doi: 10.1111/j.1539-6924.1988.tb01155.x. Disponível em: <http://doi.wiley.com/10.1111/j.1539-6924.1988.tb01155.x>.

[84] WALKER, W., HARREMOËS, P., ROTMANS, J., et al. "Defining Uncertainty: A Conceptual Basis for Uncertainty Management in Model-Based Decision Support", *Integrated Assessment*, v. 4, n. 1, pp. 5–17, 2003. ISSN: 1389-5176. doi: 10.1076/iaij.4.1.5.16466. Disponível em: <http://www.tandfonline.com/doi/abs/10.1076/iaij.4.1.5.16466>.

[85] DONGARRA, J., HITTINGER, J., BELL, J. *Applied Mathematics Research for Exascale Computing*. Relatório técnico, DOE/ASCR - U.S. Department of Energy, 2014.

[86] TANNAHILL, J., LUCAS, D. D., DOMYANCIC, D., et al. "Poster: Data Intensive Uncertainty Quantification: Applications to Climate Modeling". In: *Proceedings of the 2011 Companion on High Performance Computing Networking, Storage and Analysis Companion*, SC '11 Companion, pp. 17–18, New York, NY, USA, 2011. ACM. ISBN: 978-1-4503-1030-7. doi: 10.1145/2148600.2148610. Disponível em: <http://doi.acm.org/10.1145/2148600.2148610>.

[87] PARAVIEW-3.10.1. *ParaView open-source, multi-platform data analysis and visualization application*. 2011. Disponível em: <http://www.paraview.org/>.

[88] HORTA, F., DIAS, J., OCAÑA, K. A. C. S., et al. "Using Provenance to Visualize Data from Large-Scale Experiments". Salt Lake City, UT, USA, 2012.

[89] DIAS, J., OGASAWARA, E., OLIVEIRA, D., et al. "Supporting Dynamic Parameter Sweep in Adaptive and User-Steered Workflow". In: *6th Workshop on Workflows in Support of Large-Scale Science*, WORKS '11, pp. 31–36, Seattle, WA, USA, 2011. ACM.

[90] SILVA, C., FREIRE, J., SANTOS, E., et al. "Provenance-Enabled Data Exploration and Visualization with VisTrails". pp. 1–9. IEEE, 2010. ISBN: 978-1-4244-8421-8. doi: 10.1109/SIBGRAPI-T.2010.9. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5756325>.

[91] ANDERSON, E. W., AHRENS, J. P., HEITMANN, K., et al. "Provenance in Comparative Analysis: A Study in Cosmology", *Computing in Science & Engineering*, v. 10, n. 3, pp. 30–37, 2008. ISSN: 1521-9615. doi: 10.1109/MCSE.2008.80. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4488062>.

[92] FERREIRA, N., POCO, J., VO, H. T., et al. "Visual Exploration of Big Spatio-Temporal Urban Data: A Study of New York City Taxi Trips", *IEEE Transactions on Visualization and Computer Graphics*, v. 19, n. 12, pp. 2149–2158, 2013. ISSN: 1077-2626. doi: 10.1109/TVCG.2013.226. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6634127>.