Brigham Young University
BYU ScholarsArchive

All Theses and Dissertations

2016-02-01

A Comprehensive Python Toolkit for Harnessing Cloud-Based High-Throughput Computing to Support Hydrologic Modeling Workflows

Scott D. Christensen
Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Civil and Environmental Engineering Commons

BYU ScholarsArchive Citation

Christensen, Scott D., "A Comprehensive Python Toolkit for Harnessing Cloud-Based High-Throughput Computing to Support Hydrologic Modeling Workflows" (2016). *All Theses and Dissertations*. 5667.
https://scholarsarchive.byu.edu/etd/5667

This Dissertation is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.

A Comprehensive Python Toolkit for Harnessing Cloud-Based High-Throughput Computing to Support Hydrologic Modeling Workflows

Scott D. Christensen

A dissertation submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Norman L. Jones, Chair
E. James Nelson
Daniel P. Ames
Gustavious P. Williams
Alan K. Zundel

Department of Civil and Environmental Engineering

Brigham Young University

February 2016

Copyright © 2016 Scott D. Christensen

All Rights Reserved

ABSTRACT

A Comprehensive Python Toolkit for Harnessing Cloud-Based High-Throughput Computing to Support Hydrologic Modeling Workflows

Scott D. Christensen
Department of Civil and Environmental Engineering, BYU
Doctor of Philosophy

Advances in water resources modeling are improving the information that can be supplied to support decisions that affect the safety and sustainability of society, but these advances result in models being more computationally demanding. To facilitate the use of cost-effective computing resources to meet the increased demand through high-throughput computing (HTC) and cloud computing in modeling workflows and web applications, I developed a comprehensive Python toolkit that provides the following features: (1) programmatic access to diverse, dynamically scalable computing resources; (2) a batch scheduling system to queue and dispatch the jobs to the computing resources; (3) data management for job inputs and outputs; and (4) the ability for jobs to be dynamically created, submitted, and monitored from the scripting environment. To compose this comprehensive computing toolkit, I created two Python libraries (TethysCluster and CondorPy) that leverage two existing software tools (StarCluster and HTCondor). I further facilitated access to HTC in web applications by using these libraries to create powerful and flexible computing tools for Tethys Platform, a development and hosting platform for web-based water resources applications. I tested this toolkit while collaborating with other researchers to perform several modeling applications that required scalable computing. These applications included a parameter sweep with 57,600 realizations of a distributed, hydrologic model; a set of web applications for retrieving and formatting data; a web application for evaluating the hydrologic impact of land-use change; and an operational, national-scale, high-resolution, ensemble streamflow forecasting tool. In each of these applications the toolkit was successful in automating the process of running the large-scale modeling computations in an HTC environment.

Keywords: cloud computing, high-throughput computing, Tethys Platform, GSSHA, hydrologic modeling, Python

ACKNOWLEDGEMENTS

The process of earning a doctoral degree is no small undertaking, and to have gotten this far is a tribute to all of the support I have received. While I cannot individually name everyone that has helped me along my journey, I would like to thank a few of the people who have been a significant part of it.

First, I must acknowledge that I have received divine help. I know God has been aware of the details of my life and my research, and I believe that many of the solutions to problems that I encountered along the way came through revelation.

I owe a lot to my parents, who instilled in me a value of education and have set an example of being life-long learners. They taught me to get as much education as I could and have made the opportunities that I have had possible by providing support encouragement.

I thank my advisors Norm Jones and Jim Nelson for the confidence that they showed in me by inviting me to be a part of the CI-WATER team, and for the mentorship and friendship they have given me these past several years. I also want to thank the other members of my committee for their support and mentorship.

I am grateful to have had the opportunity to collaborate with great team members on the CI-WATER project. Nathan Swain provided both vision and leadership in creating Tethys Platform, and has been a good friend and support to me. I am lucky to have worked with Herman Dolder on his Canned GSSHA research, with Jocelynn Anderson on her GSSHA Index Editor app, and with Alan Snow in developing the Streamflow Prediction Tool. I have also benefitted from working with Fidel Perez, Ezra Rice, Curtis Rae, and others.

Finally, I gratefully acknowledge my wife, who has sacrificed a lot to make this possible for me. Her constant support has given me the hope and courage to continue day after day.

This material is based upon work supported by the National Science Foundation under Grant No. 1135483.

TABLE OF CONTENTS

LIST OF TABLES vi

LIST OF FIGURES vii

1 Introduction 1

2 Background and Related Work 6
2.1 Distributed Computing 6
2.1.1 Distributed Computing Architectures 7
2.1.2 Computing Task Classifications 8
2.1.3 Computing Paradigms 10
2.2 Cloud Computing 11
2.3 HTCondor 12
2.4 StarCluster 17
2.5 Tethys Platform 17
2.6 Related Work 20

3 TethysCluster: Automated Provisioning and Configuration of Cloud Resources 26
3.1 Software Description 27
3.1.1 The Node Class 31
3.1.2 Cloud Providers 35
3.2 Discussion 41

4 CondorPy: A Python Interface to HTCondor 44
4.1 Software Description 45
4.1.1 Job-Centric Abstraction 46
4.1.2 Cross-Platform 48
4.1.3 Remote Scheduling 52
4.1.4 Job Templates 54
4.2 Discussion 55

5 Computing Tools in Tethys Platform 59
5.1 Compute API 60
5.1.1 Using TethysCluster in Tethys 61
5.1.2 Schedulers 62
5.2 Jobs API 62
5.2.1 Job Manager 62
5.2.2 Tethys Job Templates 63
5.2.3 Jobs Table Gizmo 63
5.3 Tethys Compute Admin Pages 65
5.4 Discussion 68

6 Applications 71
6.1 Canned GSSHA: Pre-computed Modeling for Flood Warning 71
6.1.1 Methods 72
6.1.2 Results 73
6.1.3 Discussion 76
6.2 Data Access/Formatting Apps 77
6.2.1 Methods 79
6.2.2 Results 82
6.2.3 Discussion 84
6.3 GSSHA Index Editor 85

6.3.1 Methods 86
6.3.2 Results..... 88
6.3.3 Discussion..... 88
6.4 Streamflow Prediction Tool..... 89
6.4.1 Methods 90
6.4.2 Results..... 91
6.4.3 Discussion..... 92
7 Discussion and Conclusions 94
7.1 Technical Contributions 94
7.2 Future Work..... 95
7.2.1 Enhancements to TethysCluster..... 96
7.2.2 Enhancements to Tethys Compute..... 96
7.2.3 Additional Applications..... 97
References..... 99
Appendix A Software Availability..... 105
A.1 TethysCluster..... 105
A.2 CondorPy..... 105
A.3 Tethys Compute (Tethys Platform) 105

LIST OF TABLES

Table 6-1. Summary of Compute Time for the Canned GSSHA Parameter Sweep 74

Table 6-2. Summary of Statistics for Individual Model Runs and the HTCondor Pool............... 75

Table 6-3. Summary of Processing Time on the SSW Downloader Tethys App for Three Datasets.. 84

Table 6-4. Results of Computation Time Based on Area and the Number of Reaches................ 92

LIST OF FIGURES

Figure 1-1. Overview of the software projects I created, and contributed to in my research. 4

Figure 2-1. Overview of distributed systems (Foster, Zhao et al. 2008) 9

Figure 2-2. Possible configurations of an HTCondor pool. 13

Figure 2-3. Major processes in an HTCondor (Thain, Tannenbaum et al. 2005). 14

Figure 2-4. Customized instances of Tethys Portal. 19

Figure 2-5. Component diagram for Tethys Platform. 20

Figure 3-1. StarCluster can provision an HTCondor pool of Linux virtual machines on Amazon Web Services. 27

Figure 3-2. TethysCluster can provision HTCondor pools of Linux virtual machines on Microsoft Azure and both Linux and Windows Machines on Amazon Web Services. 28

Figure 3-3. Unified Modeling Language (UML) diagram showing classes and their relationships in the StarCluster code. 29

Figure 3-4. UML diagram showing classes that I modified and the classes and modules that I added in TethysCluster. 30

Figure 3-5. Implementation of the *set_hostname* method in the LinuxNode subclass. 32

Figure 3-6. Implementation of the *set_hostname* method in the WindowsNode subclass. 32

Figure 3-7. UML inheritance diagram showing the subclasses of the Node class. 33

Figure 3-8. The *make_node* method in the NodeManager class determines the type of node that should be made based on the platform attribute of the VM instance that is passed in. 34

Figure 3-9. Detail UML diagram of the azureutils module showing the parallel structure to the awsutils module. 37

Figure 3-10. Modified method in the TethysClusterConfig class that determines which cloud provider class to instantiate. 38

Figure 4-1. Class relationship UML diagram for the classes in CondorPy. 45

Figure 4-2. UML diagram for the CondorPy Job class. 46

Figure 4-3. UML diagram for the CondorPy Workflow class. 49

Figure 4-4. UML Diagram for the CondorPy Node class. 49

Figure 4-5. Diagrams of various workflows showing the levels of complexity that can be represented in CondorPy. 50

Figure 4-6. Code for the custom decorator that ensures HTCondor CLI commands are executed from the job's working directory. 51

Figure 4-7. The submit method of the HTCondorObjectBase class, which parses the output of a submit command using a regular expression to get the cluster id. 52

Figure 4-8. UML diagram for the CondorPy HTCondorObjectBase class. 54

Figure 4-9. Code sample for submitting an HTCondor job with CondorPy. 55

Figure 5-1. Expanded Tethys Platform component diagram showing the sub-components of Tethys Portal and the Tethys SDK. 60

Figure 5-2. The Jobs Table gizmo. 64

Figure 5-3. Tethys Portal admin page for Tethys Compute. 65

Figure 5-4. Tethys Portal admin page for creating/editing clusters. 66

Figure 5-5. Tethys Portal admin page Tethys Compute Settings. 67

Figure 5-6. Tethys Portal admin page for creating/editing schedulers. 68

Figure 5-7. Tethys Portal admin page for editing jobs. 69

Figure 6-1. Summary of the steps in the Canned GSSHA study. 73

Figure 6-2. The Canned GSSHA Tethys app allows users to explore the solution space of the 57,600 archived simulations. 74

Figure 6-3. Timeline showing the active computing time for running 57,600 GSSHA models using HTCondor. 74

Figure 6-4. The Simple Subset Wizard web tool allows users to obtain subsets of large data sets. 78

Figure 6-5. Links for downloading data from the SSW. 79

Figure 6-6. SSW Downloader Tethys app. 80

Figure 6-7. Convert NetCDF to GSSHA Input Tethys app. 81

Figure 6-8. Summary of steps in developing the data access and formatting apps for use in GSSHA models. 82

Figure 6-9. Soil moisture data from the NLDAS data set over the larger domain. 83

Figure 6-10. Soil moisture data from the NLDAS data set over the smaller domain. 83

Figure 6-11. Hourly precipitation for April 2015 near Austin, Texas. 84

Figure 6-12. Plot of the processing times for various datasets with 1 or 2 variables on a large or small domain. 85

Figure 6-13. Map interface of the GSSHA Index Map Editor Tethys app. 87

Figure 6-14. Summary of steps in creating the GSSHA Index Map Editor app...........................88

Figure 6-15. Example results of GSSHA Index Map Editor app workflow. 89

Figure 6-16. Summary of steps for setting up the Streamflow Prediction Tool. 91

Figure 6-17. The Streamflow Prediction Tool displaying streamflow forecasts for a reach in the Great Basin Region...93

1 INTRODUCTION

Water resources models are often used to provide information needed to support decisions affecting the safety and sustainability of society. Relative to traditional lumped parameter models, spatially distributed, physics-based models such as Gridded Surface/Subsurface Hydrologic Analysis (GSSHA) (Downer and Ogden 2004) have increased applicability and accuracy, allowing for better-informed decisions. However, these advantages come at a cost; sophisticated models are generally (1) more complex, and (2) require more time and computational resources to run. To handle the issue of increased complexity, including the often significant amount of data manipulation and file management in both the pre- and post-processing, and to ensure that the process is repeatable, it is increasingly common to create scripted modeling workflows. When these workflows are integrated into web-based applications they become more broadly accessible and have more wide-reaching influence (Swain, Latu et al. 2015). Still, addressing the issue of needing access to additional computational resources to run sophisticated models remains a challenge. The purpose of this research has been to investigate methods to facilitate access to sufficient resources in computationally demanding modeling workflows and web applications.

Typically water resources models are run using powerful desktop workstations, yet there are many modeling applications that require a significant number (possibly millions) of model runs, such as calibration, parameter sweeps, uncertainty analysis, and parent-child models, which

can take hundreds of thousands of CPU hours and cannot reasonably be supported by even the most up-to-date hardware. To accomplish these modeling tasks, additional computing power is needed. To illustrate, it is often necessary to account for uncertainty in water resources models. Uncertainty is inherent in modeling because of natural variability and parameter uncertainty (Smemoe 2004), structural noise (Doherty and Welter 2010), or unknown future conditions. A common way to understand and characterize this uncertainty is to perform a stochastic analysis that necessitates hundreds or thousands of model runs. It is not uncommon for a single simulation to take several hours to run on a desktop workstation; to run several thousand simulations may take years! Without additional computational resources this type of modeling task would be unfeasible.

The traditional approach for accessing large-scale computing resources is high-performance computing (HPC) using supercomputers or computing clusters. However, for many organizations, acquiring, operating, and maintaining this type of hardware is not cost-effective, especially if the computational demand is sporadic and does not require near full-time utilization of the resources, as is often the case with modeling applications. Yet, without access to the necessary computing resources, scientists and engineers are limited in their ability to perform important modeling tasks (Humphrey, Beekwilder et al. 2012). An alternative method to get access to more computing without the need to invest in additional computing infrastructure is through high-throughput computing (HTC). HTC differs from HPC in that the objective is to optimize the long-term performance, or the amount of computing (or throughput) that can be done over a long time period (e.g. the number of jobs per month), rather than the short term performance (e.g. the number of operations per second) (Livny, Basney et al. 1997). Using HTCondor, a middleware used to configure HTC systems, this type of computing environment

can be set up with ordinary desktop computers (Litzkow, Livny et al. 1988). However, the number of local desktop computers that are available may not be sufficient to handle the computational load of some modeling tasks. Recent advances in cloud computing have made it an ideal resource for large-scale computing because it provides universal access to on-demand, scalable, and cost-effective computing resources. HTC systems can integrate both local desktop computers and cloud resources enabling scientists and engineers to leverage existing hardware while also providing the flexibility to scale using the cloud.

To facilitate the use of HTC and cloud computing in modeling workflows and web applications a scripting toolkit is needed which meets the following requirements: (1) programmatic access to diverse, dynamically scalable computing resources, (2) a batch scheduling system to queue and dispatch the jobs to the computing resources, (3) data management for job inputs and outputs, and (4) the ability for jobs to be dynamically created, submitted, and monitored from the scripting environment. I created two Python libraries, TethysCluster and CondorPy, to meet these requirements. TethysCluster is based on another Python tool called StarCluster and automates provisioning cloud resources and configuring those resources into an HTC environment using HTCondor. CondorPy enables jobs to be created and run in that environment by providing an interface to HTCondor. Together these libraries meet all of the requirements listed above, and thus form a comprehensive Python computing toolkit.

To further facilitate accessing HTC and cloud resources in web applications, I used this computing toolkit to add computing capabilities to Tethys Platform. Tethys Platform is a development environment designed to lower the technical barriers of creating web-based modeling applications (Jones, Nelson et al. 2014). I used TethysCluster and CondorPy as the core elements in building Tethys Compute, a module in Tethys Platform that provides developers

with a software development toolkit (SDK) for integrating large-scale computing into web-based modeling apps.

I tested TethysCluster, CondorPy, and Tethys Compute by using them to support various other research applications including (1) pre-computing model results for flood scenarios using a Latin hypercube sampling parameter sweep of a GSSHA model (Canned GSSHA), (2) developing Tethys apps to access and reformat hydrologic data, (3) the GSSHA Index Map Editor app, which evaluates the hydrologic impact of land-use change, and (4) the Streamflow Prediction Tool, which performs ensemble streamflow forecasts. An overview of the software projects that I developed and the applications that I contributed to is shown in Figure 1-1. The arrows in the diagram indicate software that was used to support other projects.

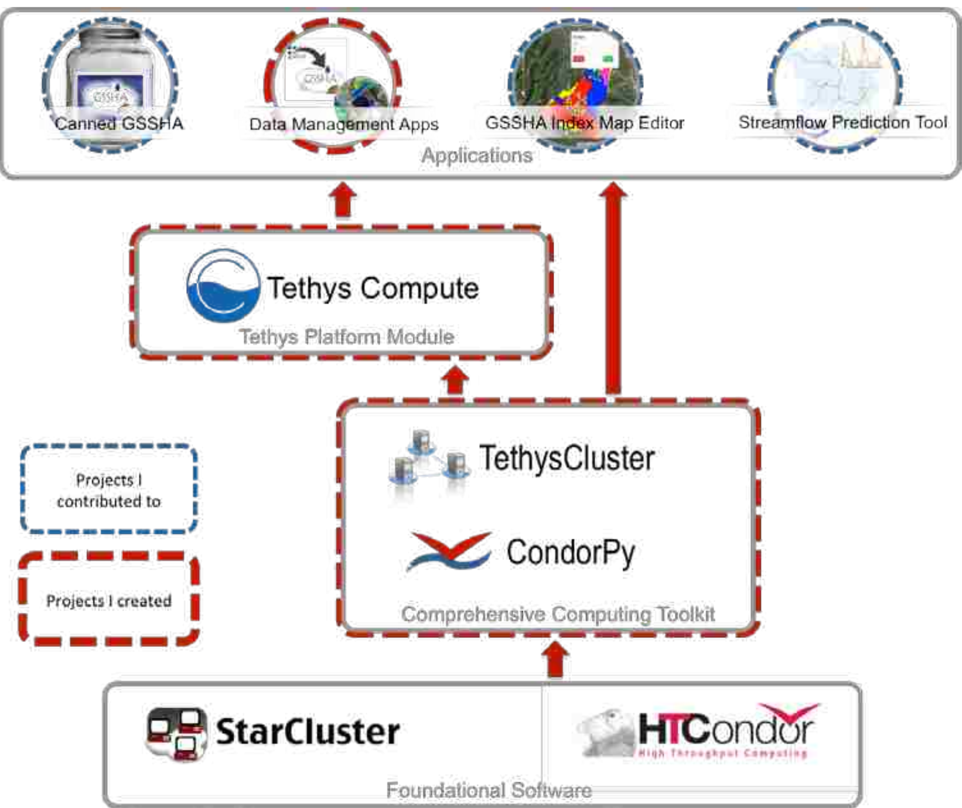


Figure 1-1. Overview of the software projects I created, and contributed to in my research.

This dissertation is organized as follows: Chapter 2 provides background and context, describes related work, and states how my research has built upon and expands previous work. The Python libraries TethysCluster and CondorPy are described in Chapters 3 and 4, respectively. Chapter 5 presents the Tethys Platform computing tools. Chapter 6 discusses several applications and test cases. And finally, I discuss the conclusions and contributions of my research, and possibilities for future research in Chapter 7.

2 BACKGROUND AND RELATED WORK

This chapter provides background on computing technologies and tools relevant to this research and describes advances that have been made in their use for water resources modeling. First, a general background on distributed computing is provided with the goal of clarifying terminology, followed by a more in-depth description of cloud computing and its benefits for large-scale modeling. Next, some background is given on several tools that are of particular relevance to this research, including the resource management and job scheduling system (HTCondor), the cloud resource provisioning software (StarCluster), and the web-app development platform (Tethys Platform). Finally, previous work that has used HTC and cloud computing in water resources modeling is described while highlighting the need for a comprehensive toolkit to facilitate accessing HTC in modeling workflows and web applications.

2.1 Distributed Computing

Large computing tasks, such as those often required by water resources models, generally depend on a distributed computing system. Distributed computing is a general term that refers to any system that allows the decomposition of computing tasks to be computed simultaneously on different processing elements (Buyya, Vecchiola et al. 2013). The field of distributed computing has evolved so quickly that it has outpaced the terminology used to describe it. As a result, many of the terms used in the field are overloaded and ambiguous (Foster, Zhao et al. 2008; Buyya, Vecchiola et al. 2013). Since I use HTC and cloud computing to enable large modeling tasks,

this section is included in an effort to be clear about computing terms and to provide some context for HTC and cloud computing. This section is organized into three subsections: distributed computing architectures, computing task classifications, and computing paradigms.

2.1.1 Distributed Computing Architectures

The architecture of a computing system is the configuration of hardware components and the software required to make it a functioning system. This subsection defines several terms used to describe distributed computing architectures.

Supercomputers: Computers that are at the front line of computing hardware. They typically are parallel systems, meaning that there are multiple, identical processors that simultaneously operate on the same data using a single, shared memory space. This architecture is also known as symmetric multiprocessor systems (SMP) (Buyya, Vecchiola et al. 2013).

Clusters: A form of parallel systems where computing nodes are tightly networked so they can function as a single system. They are generally composed of commodity hardware components and have distributed memory. Clusters are a low-cost alternative to supercomputers and are designed to be capable of the same type of workloads (Buyya, Vecchiola et al. 2013).

Grids: A network of various computing components (traditionally supercomputers or clusters, but more generally any heterogeneous computing elements) that use a software element called middleware to combine the components into a unified system (Rouholahnejad, Abbaspour et al. 2012; Buyya, Vecchiola et al. 2013). Grids came about because there were many supercomputing clusters that were often idle and many computational tasks that required more capacity than could be offered by a single cluster. The term "grid" stems from the utility sector

and captures the idea that computing power can be offered on-demand as a public utility (Foster, Zhao et al. 2008).

Cloud Computing: On-demand computing, network, and storage services, and applications delivered via the Internet with a pay-per-use pricing model (Amazon Web Services 2014). Cloud computing relies heavily on the concept of virtualization, which enables raw hardware to be combined virtually to form a computing system (Buyya, Vecchiola et al. 2013). A more complete definition of cloud computing is given in Section 2.2.

Utility Computing: The idea that computing is a utility that can be distributed like electricity or water (Foster, Zhao et al. 2008). This idea of computing was the inspiration for grid computing and became fully realized with cloud computing.

There are a lot of overlaps and unclear boundaries among the different computing architectures noted above. Figure 2-1, reproduced from work by Foster, Zhao et al. (2008), shows the overlap of the various distributed system architectures.

2.1.2 Computing Task Classifications

Large computing tasks are often parallelized, or divided into sub-tasks, so they can take advantage of distributed computing architectures. These tasks can be classified based on how they are parallelized as either tightly coupled parallel or embarrassingly parallel.

Tightly Coupled Parallelization: The class of computing tasks that can be broken into separate processing units, which can be run on separate processors, but where each unit is not independent of other units, and therefore information must be passed between processors. Information transfer often follows the Message-Passing Interface (MPI) specification that defines

how data are passed from one process to another (Lusk, Huss et al. 2012). This class of tasks must be run on a tightly integrated architecture like a supercomputer or computing cluster.

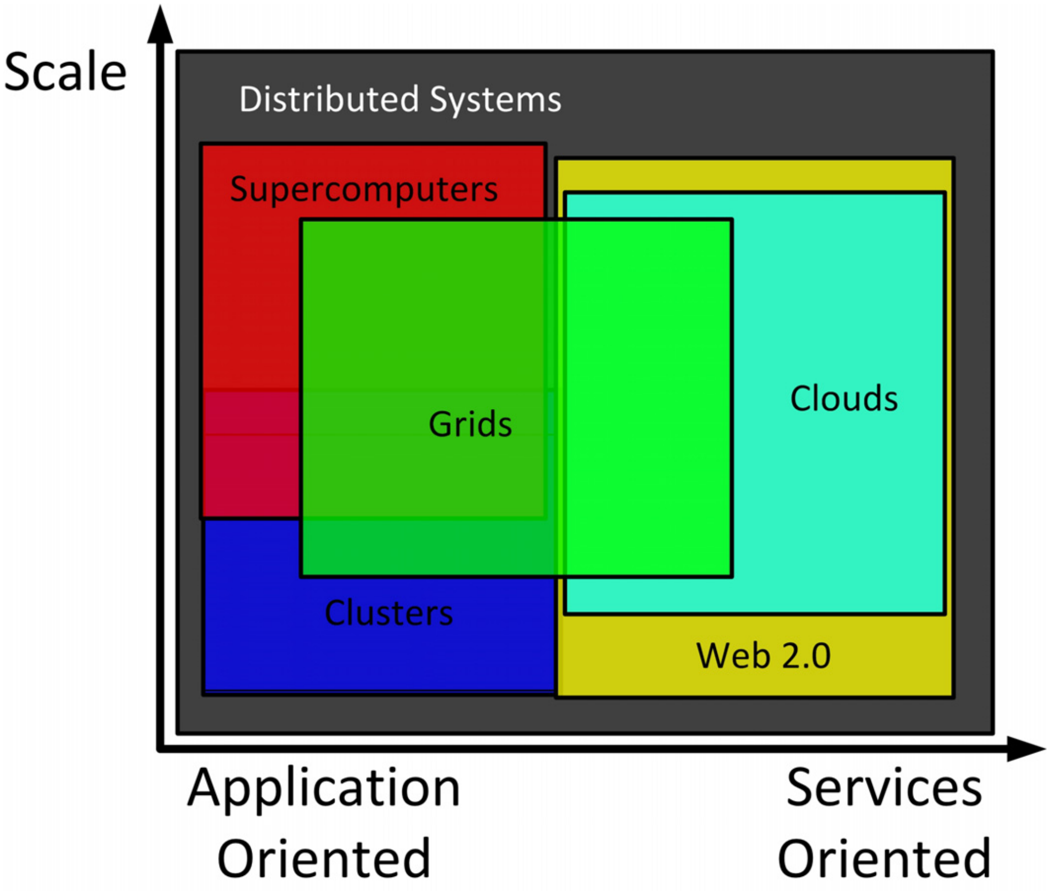


Figure 2-1. Overview of distributed systems (Foster, Zhao et al. 2008)

Embarrassingly Parallel: Denotes tasks that can be broken into sub-tasks that are completely independent from one another. Since each computing task is self-contained, there is no message passing between processes and, therefore, fewer restrictions on the architectures that can support this type of computing problem. Other names for this class of computing tasks are pleasingly parallel, perfectly parallel, happily parallel, or bag-of-tasks.

2.1.3 Computing Paradigms

Several computing paradigms, or ways of handling various classes of computing problems, have been proposed over time. While these paradigms don't define a particular architecture, they do have implicit hardware and middleware requirements (Buyya, Vecchiola et al. 2013).

High-Performance Computing (HPC): Computing where the performance of the processing units is crucial and is typically measured in floating-point operations per second (FLOPS), or more commonly peta-FLOPS (a quadrillion FLOPS). Generally, HPC workloads are performed on supercomputers or clusters where parallel, tightly coupled tasks can be processed in a short period of time. While the strict definition of HPC refers to the performance of the processing units of a system, this term is commonly used in a more generic sense to mean any type of large or intense computing that cannot be done on a standard workstation and necessitates a distributed computing system.

High-Throughput Computing (HTC): Computing where reliability and long-term performance are crucial. HTC is typically measured in cycles (or jobs) per week or month. HTC workloads are often embarrassingly parallel and are executed on heterogeneous distributed systems. Whereas in HPC the performance of the processing units is crucial, in HTC the number of processing units is of more concern. The main challenge of an HTC environment is maximizing the computational resources that are available (Livny, Basney et al. 1997).

Many-Task Computing (MTC): A computing paradigm that seeks to bridge the gap between HPC and HTC. The goal is to leverage a large amount of computing resources (reminiscent of HTC), but over a short period of time to maximize the number of tasks per second (similar to HPC). Tasks are generally heterogeneous and loosely coupled, using a shared

file system to communicate, but may not be embarrassingly parallel. The main concern of MTC is scalability (Raicu 2009).

2.2 Cloud Computing

Terms like "cloud computing" or "the cloud" have become quite commonplace and are used in so many different contexts that their exact meaning is a bit ambiguous. Many definitions have been proposed in the literature, yet the new and quickly evolving field has not yet converged on a stable definition. Armbrust, Fox et al. (2010) gave a concise definition of the cloud stating that it was "applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services." Vaquero, Rodero-Merino et al. (2008) compiled early definitions in the literature and stated the most commonly included characteristics were scalability, virtualization, and pay-per-use utility model. Perhaps the most widely accepted definition is from the National Institute of Standards and Technology (NIST), which states that the essential characteristics of the cloud are: on-demand self-service, broad network access, resource-pooling, rapid elasticity, and measured service. The NIST definition also includes three service models: infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS); in addition to four deployments: public cloud, private cloud, community cloud, and hybrid. Of special note is the hybrid cloud—a combination of a private and public cloud—because it is often used to create the most cost-effective computing cloud with truly elastic scalability by first using local computing resources when available before using commercial cloud resources (Humphrey, Hill et al. 2011). This practice is termed cloud bursting.

Cloud computing is an attractive resource for water resources modeling because cost-effective, scalable computing can be accessed ubiquitously on-demand. It opens the door to perform modeling at a scale that was previously only available to those with access to large computing centers. Furthermore, if a modeling workflow is deployed as a web app, the step to utilizing cloud computing is a natural one. Here a distinction is made between web apps that use the cloud only to deploy their software as a SaaS and hydrologic apps that use IaaS cloud computing to perform large computing tasks.

2.3 HTCondor

HTCondor is a middleware that provides both job and computing-resource management and enables HTC through opportunistic computing. It is probably the most commonly used middleware for managing clusters, idle workstations, and grids (Buyya, Vecchiola et al. 2013). Since I use HTCondor as an essential part of the computing toolkit, it is described here in detail.

HTCondor creates an HTC system by grouping, or "pooling", network-connected computing resources. A common example of an HTCondor pool is a set of lab or office computers that are all on the same network and configured (through HTCondor) to be part of the same computing system. Each computer in the pool is assigned one or more roles such as *matchmaker*, *resource*, or *agent.* Background processes, called daemons, which are specified in the computer's HTCondor configuration file determine the role(s) of a computer. Each pool has only one *matchmaker,* which serves as the central manager. All other computers in the pool are *resources* and/or *agents* and are configured to report to the central manager. A *resource* is also known as a worker, and is a computer designated to run jobs. And an *agent*, also known as a scheduler, is a computer designated to schedule jobs. Any computer in the pool (including the

central manager) may function as both a *resource* and an *agent*. Figure 2-2 shows a possible HTCondor pool configuration.

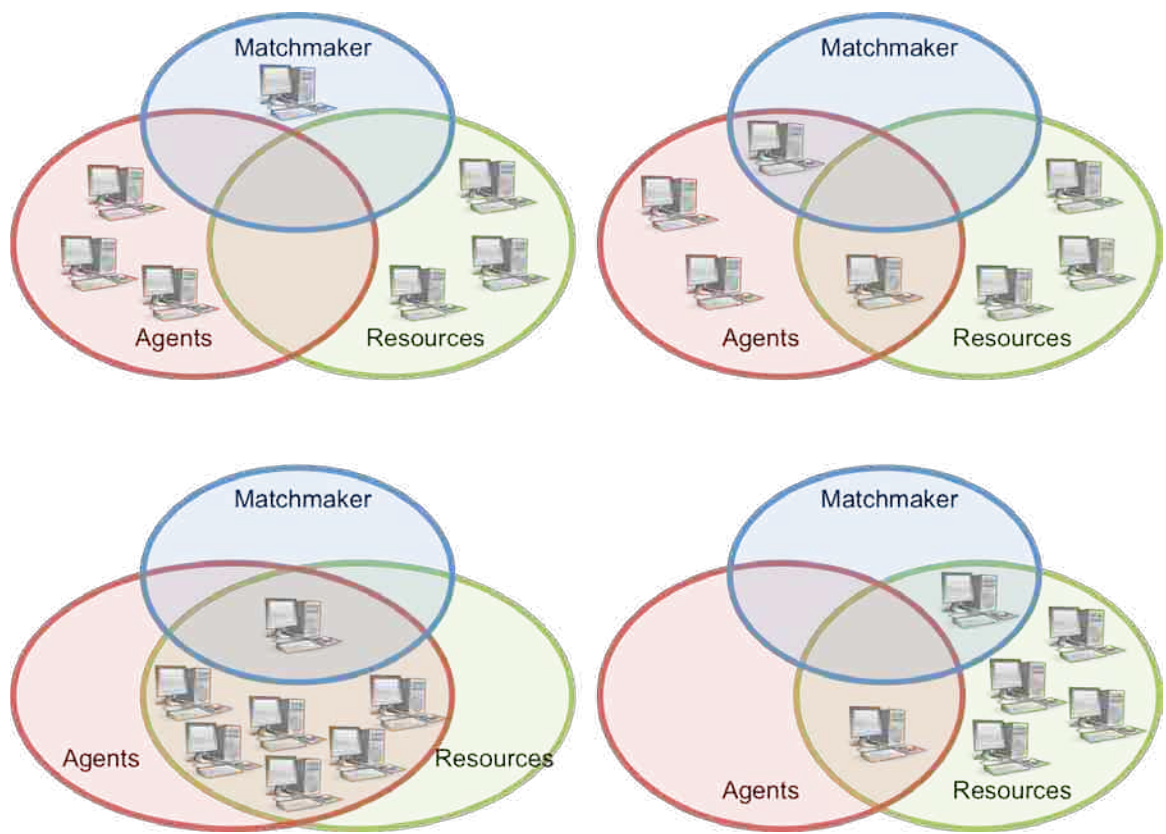


Figure 2-2. Possible configurations of an HTCondor pool.

To submit a job to the pool a user must create a file to describe the job requirements, including the executable and any input or output files. This file is known as a *problem solver*. The user submits the *problem solver* to an *agent,* which advertises the requirements needed to run the job to the central manager. Similarly, each *resource* in the pool also advertises its availability, capabilities, and preferences to the central manager. This advertising, from both the *agents* and the *resources,* is done using a schema-free language called ClassAds. Periodically the central manager scans the ClassAds from *resources* and from *agents* and try to match jobs and resources that are compatible. When a match is made, the central manager notifies the *agent* and

the *resource* of the match. It is then the *agent's* responsibility to contact the *resource* and start the job. This interaction is handled by a process running on the *agent*, known as the *shadow*, which communicates with a process running on the *resource*, called the *sandbox*. The *shadow* provides all of the details required to execute a job, while the *sandbox* is responsible for creating a safe and isolated execution environment for the job.

The entire sequence is known as the HTCondor kernel and is summarized in Figure 2-3 reproduced from (Thain, Tannenbaum et al. 2005).

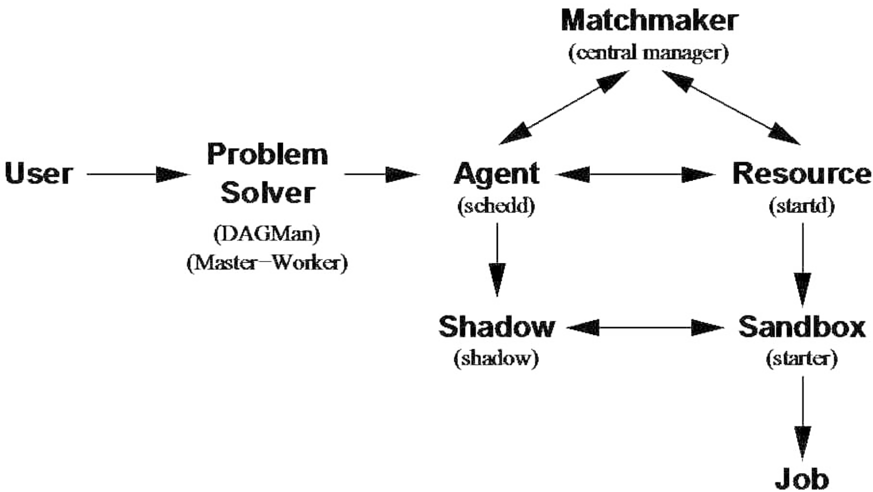


Figure 2-3. Major processes in an HTCondor (Thain, Tannenbaum et al. 2005).

HTCondor was originally conceived with the goal of scavenging processing time from idle desktop workstations (i.e. opportunistic computing) (Litzkow, Livny et al. 1988). This concept alone enables organizations to create HTC systems without the need to invest in additional computing resources by fully utilizing the computers they already own during times when they would otherwise be idle (e.g., nights, weekends, or other times when a user is away from the computer). However, the capabilities of HTCondor now extend beyond just

opportunistic computing. The core design philosophy of HTCondor, and the key that has given the project success, is flexibility (Thain, Tannenbaum et al. 2005), which has allowed HTCondor to expand to be able to combine diverse computing resources (including clusters, grids, and cloud resources in addition to desktop computers), managed by different organizations with differing goals and objectives, into one computing system.

Additional features of HTCondor extend its flexibility and capabilities. For example, HTCondor is ideal for embarrassingly parallel batch jobs, but it also provides a way of executing workflows using directional acyclic graphs (DAGs) (Couvares, Kosar et al. 2007). A DAG specifies a series of jobs, referred to as nodes, that need to be run in a particular order and also defines the relationships between nodes using parent-child notation. This allows for the common situation where the output from a preliminary set of simulations is used as input for a subsequent set of simulations. An alternative scheduler called a DAG Manager (DAGMan) is used to orchestrate submitting jobs in the proper order to the normal scheduler. If a node in the DAG fails, the DAGMan generates a rescue DAG that keeps track of which nodes are completed and those that still need to run. A rescue DAG can be resubmitted, and it will continue the workflow from where it left off. This provides a robust mechanism for executing large workflows or a large number of jobs.

HTCondor provides several runtime environments called *universes*. The two most common universes are the standard universe and the vanilla universe. The standard universe is only available on Unix machines. It enables remote system calls so the resource that is running the job can remotely make calls to the agent to open and read files, which means that job files need not be transferred, but can remain on the submitting machine. It also provides a mechanism called *checkpointing*. Checkpointing enables a job to save its state. Thus when a resource

becomes unavailable in the middle of executing a job (because, for example, a user starts using the computer where the job is running), the job can start on a new resource from the most recent checkpoint. The vanilla universe is the default universe and is the most generic. It requires that the computing resources have a shared file system or that files be transferred from the submitting machine to the computing resource. The vanilla universe does not provide the checkpointing mechanism, and thus, if a job is interrupted mid-execution, it must be restarted on a new resource. Various other universes are available and are described in greater detail in the user manual (Condor Team 2014).

HTCondor also provides a mechanism, called *flocking,* for submitting jobs to more than one pool of resources. Flocking enables organizations to combine resources while maintaining control of their own pool. For a machine to be able to flock to a remote pool the remote scheduler must be identified in the configuration on that machine. Additionally, the remote scheduler must accept the machine to be flocked from in its own configuration. The submitting machine will first try to match jobs in its native pool, but when resources are not available then it will "flock" jobs to the remote pool. Generally, a remote pool will not have a shared file system, so jobs that are flocked must enable the file transfer mechanism.

The standard way for interacting with HTCondor is through the command line interface (CLI). HTCondor also provides several APIs, including a set of Python modules that interact directly with ClassAds and the HTCondor daemons, allowing jobs to be created and submitted with Python scripts (Condor Team 2014). These Python modules, known as the HTCondor Python bindings, are currently only available for Linux and are targeted at a specialized audience of developers (Bockelman 2013).

2.4 StarCluster

StarCluster is a cluster provisioning software created at MIT for Amazon Web Services (AWS) (StarCluster 2014). It allows users to create groups (or clusters) of virtual machines (VMs) (or nodes) that are automatically configured to operate as a unified HTC system. Since I created TethysCluster as a modified version of StarCluster, in this section I describe the functionality of StarCluster before modification.

StarCluster requires the use of pre-built Amazon Machine Images (AMIs) to provision a cluster of VMs, and currently only supports Ubuntu or CentOS images. The pre-build StarCluster AMIs are publicly available and can be customized to include any necessary software for a particular computing task.

StarCluster is written in Python and contains several high-level classes that represent the cloud provider, computing clusters, individual nodes within the cluster, etc. It uses a plug-in framework to configure nodes with the necessary settings to activate various features, like a job management system. By default the Sun Grid Engine (SGE) plugin is included on all clusters for job management, but there is also a plugin for HTCondor. StarCluster is designed with a CLI so it can be used directly from a command line to set up and manage computing clusters. It also has an application programing interface (API) so that provisioning and setting up computing clusters can be automated with Python scripts. StarCluster code is open source and thus available for further customization and adaptation.

2.5 Tethys Platform

Tethys Platform is a free and open source software (FOSS) written in Python and powered by the Django web framework (Jones, Nelson et al. 2014; Swain, Christensen et al. In

Press). It provides an environment for developing and hosting web-based, water resources applications or “web apps.” This platform links together components that are needed to support water resources web apps, including web geographic information systems (GIS), visualization, data management, and computational resources. It provides a Python scripting environment in the form of an SDK to lower the barrier to developing apps, and includes a web portal for easily deploying them.

Tethys Platform leverages several other FOSS projects, which together make up the Tethys software suite. Included in the software suite are: the web GIS projects GeoServer for publishing spatial datasets as web mapping services (WMS), 52 North for providing geoprocessing tools as a web processing services (WPS), and the PostgreSQL database with the PostGIS extension for providing spatial data storage. In addition to the web GIS projects, the software suite also includes a number of JavaScript libraries like OpenLayers and HighCharts to help create interactive maps and charts in the browser. The computing capabilities of Tethys Platform, which I added, rely on HTCondor.

Tethys Platform offers an SDK that helps developers interact with the various components of the software suite and facilitates other aspects of web app development. The Tethys SDK provides APIs to the components in the Tethys software suite as well as to several external resources including CKAN or HydroShare for data storage, and to cloud computing resources. In addition to these APIs, the SDK also has APIs powered by Django to facilitate various aspects of web development including building web pages, user workspaces, and app-to-app communication.

The Tethys Portal is a deployable web site that serves as the runtime environment for apps that are developed with Tethys Platform. Django provides all of the core components of a

website, including a user management system and admin pages where the site can be configured and customized. Different organizations can each host their own instance of a Tethys Portal, brand and customize it, and install the apps that are suitable for their needs (Figure 2-4).

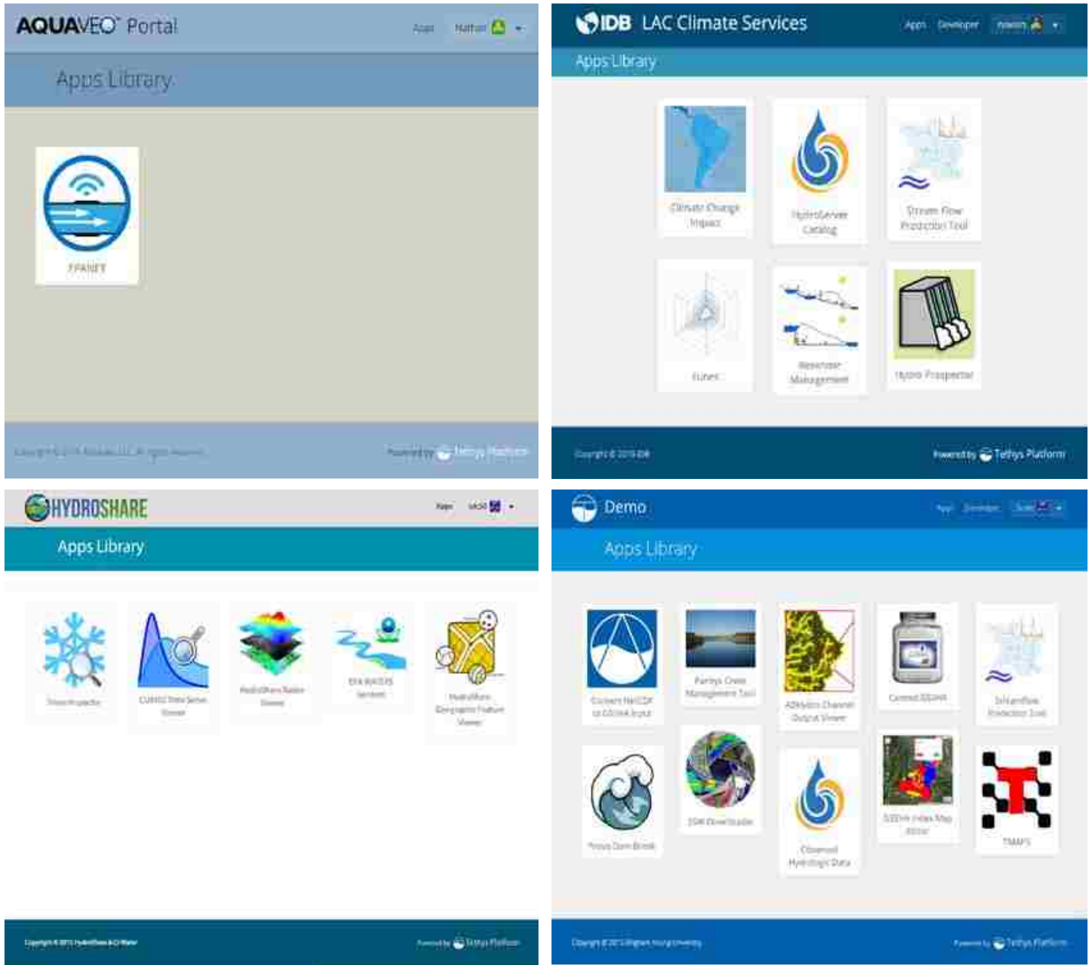


Figure 2-4. Customized instances of Tethys Portal.

In summary, Tethys Platform is a development and runtime environment for web apps. Tools have been included specifically to support the needs of water resources web apps such as web GIS elements and computing resources. An SDK is provided to facilitate the use of the software suite included with Tethys Platform, in addition to external resources. In order for Tethys to provide support for large modeling tasks it needs access to HTC systems and on-

demand, highly scalable computing resources. The Python toolkit developed by this research has been integrated into the Tethys SDK to provide this functionality. The runtime environment for the apps is powered by the Python web framework, Django, which also powers some of the development tools that are part of the SDK. The various components of Tethys Platform that have been described are shown in diagram in Figure 2-5.

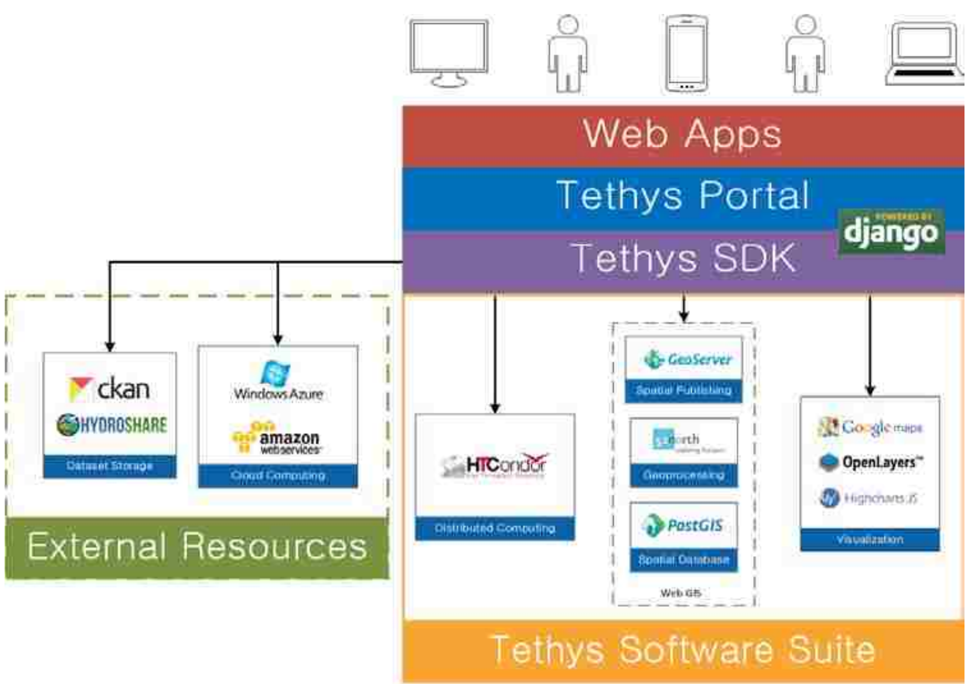


Figure 2-5. Component diagram for Tethys Platform.

2.6 Related Work

The computing tools described in the previous sections have been leveraged to facilitate large-scale computing in various fields, and many examples are present in the literature (Li and Mascagni 2003; Huang and Yang 2011; Fisch, Meißner et al. 2014; Petri, Li et al. 2014; Harvey and Ji 2015). The water resources modeling community, however, has begun to adopt these tools more recently. Efforts that have been made in applying and modifying these tools for water resources applications are described below.

The use of cloud computing in water resources applications is an active field. Sun (2013) created a simple watershed management app using Google services like Fusion Tables and Google Maps. ModflowOnAzure is a cloud service built using the Microsoft Azure cloud and is able to run batch MODFLOW jobs using Dropbox to sync files from a local machine to the cloud (Liu, Sun et al. 2012). The ModflowOnAzure service was used in the CyberGIS project to expand the computing resources from grid computing to the cloud (Behzad, Padmanabhan et al. 2011). Huang, Cao et al. (2014) created a similar system with Azure and Dropbox for running contaminant transport models (MODFLOW and MT3DMS). Wu and Khaliefa (2012) created a local cloud of commodity machines and linked them using Microsoft HPC Pack which they used to run pump scheduling optimizations through a web portal. Hunt, Luchette et al. (2010) also created an HPC Pack system for calibrating SWAT models, with the ability to cloud burst to Azure. Humphrey, Beekwilder et al. (2012) used BeoPEST to calibrate MODFLOW models on GoGrid. A Python module called cloudPEST created at the USGS facilitates provisioning resources for running PEST on the Elastic Compute Cloud (EC2) from AWS (Fienen, Masterson et al. 2013). Other similar cloud modeling applications are described in the literature (Wang, Tao et al. 2008; Juve, Deelman et al. 2009; Luchette, Nelson et al. 2009; Lu, Jackson et al. 2010; Subramanian, Liqiang et al. 2010; Kollet, Schumacher et al. 2011; Kim, Jung et al. 2012; Gunarathne, Zhang et al. 2013; Miras, Jiménez et al. 2013; Delipetrev, Jonoski et al. 2014). All of these cloud applications are specific to a particular modeling purpose, and in many cases a particular model or workflow, and also are created for a specific cloud provider.

Glenis, McGough et al. (2013) created a more generic solution for running models in a cloud-based system that also incorporates HTCondor. A CLI was developed to simplify the task of provisioning a cloud computing cluster. They employ DeltaCloud to abstract the interface of

various clouds so their system is compatible with any cloud provider that is supported by DeltaCloud. A so-called Cloud Enactor manages the jobs and provisions the VMs. HTCondor is used for scheduling if the number of jobs submitted is larger than the number of VM instances. The study used the system to run various flooding scenarios using the model CityCat. The details of the system implementation were not described nor is the code openly available. However, this example serves to illustrate the possibility of using cloud computing and HTCondor to create a generic computing system to support water resources modeling.

There is a relatively new tool developed by the Microsoft Research team, called SimulationRunner, that is designed for running parameter sweeps using either a Windows executable, or one of several other supported programs including Python and Matlab (Liu, Zou et al. 2015). It has a built-in batch scheduling mechanism, but currently only supports job submission through its web interface. Data management is all handled through Azure Storage, and input files can be uploaded through the web interface or retrieved from Dropbox. Output files are automatically stored in an Azure Storage Container. While there are no licensing restrictions for deploying an instance of SimulationRunner, it can only be deployed on Azure, and the code is not publically available. Additionally, there is no mechanism for programmatically provisioning resources or submitting jobs, and it is only capable of handling simple embarrassingly parallel problems.

StarCluster (StarCluster 2014), while providing many of the same benefits as SimulationRunner, also provides a way for programmatically provisioning resources, and since it can use HTCondor, it has the ability to schedule embarrassingly parallel workflows as well as hierarchical job workflows. Unlike many of the other systems previously described, StarCluster creates generic HTC environments that are not specific to any modeling workflow or purpose.

However, it is limited in the type of VMs it can provision (only Linux), and it only works with a single cloud provider (AWS).

In addition to the many examples of water resources applications that have leveraged commercial cloud computing, there are also several examples in the literature of using HTCondor to create HTC systems from local computing resources. For example, the Commonwealth Scientific and Industrial Research Organisation (CSIRO) in Australia is using HTCondor to capture idle time on their organization computers consisting of 5000 nodes and ran over 12,000 agricultural model jobs, each consisting of 325 different management scenarios, in just over 10 days, rather than the estimated 30 years it would have taken to run the simulations on a single computer (Zhao, Bryan et al. 2013). Furthermore, the agricultural model used in this study required a Windows environment to run, an operating system that is uncommon for high performance computing clusters, but was easily obtained using the existing desktop computers of the organization, which eliminated the need to purchase additional, specialized computing infrastructure. HTCondor has also been used to run hundreds of thousands of models while applying the Bayesian inverse modeling technique Method of Anchored Distributions (MAD) to find to find spatial random fields (Heße, Savoy et al. ; Osorio-Murillo, Over et al. 2015). These examples highlight the benefits of using HTCondor to perform large-scale environmental modeling, but they do not provide a way to use HTCondor in a scripting environment for automating modeling workflows or for use in web applications. Taylor (2013) used Python scripts to interact with HTCondor's CLI in a Windows environment to automate the process of using HTCondor to perform a stochastic analysis with the hydrologic model GSSHA. However, these scripts were specific to the application and do not provide a general solution for scripting HTCondor.

As stated earlier, there are four major requirements that are needed to provide a comprehensive scripting toolkit for accessing HTC and cloud computing to support modeling: (1) programmatic access to diverse, dynamically scalable computing resources, (2) a batch scheduling system to queue and dispatch the jobs to the computing resources, (3) data management for job inputs and outputs, and (4) the ability for jobs to be dynamically created, submitted, and monitored from the scripting environment. None of the systems that were found in the literature review provide all of these requirements in a generic, reusable form; however, I leveraged some of these tools and previous work to provide a toolkit that does meet all four requirements.

The features of the first requirement were met by building on the functionality already provided in StarCluster. The only aspect that StarCluster lacked was to provide the desired diversity of computing environments, which has two components: (1) diversity of supported operating systems, and (2) diversity of supported cloud providers. Since StarCluster is open source and written in Python, I modified it to enable creating clusters composed of both Windows and Linux VMs, using either AWS or Azure. The modified form of StarCluster is called TethysCluster and is described in Chapter 3.

The second and third requirements of the computing toolkit, a scheduling system and data management, are both provided by HTCondor. TethysCluster uses HTCondor to configure the clusters it provisions so these features are already built in to these clusters. While there are other job management systems that could also have been chosen, HTCondor has the advantages of providing opportunistic computing in addition to various runtime environments and mechanisms of data management available through the different universes as explained in section 2.2.

The fourth requirement, a Python scripting environment for creating, submitting, and monitoring jobs, is the last component needed to provide a comprehensive computing toolkit. While HTCondor does have low-level Python bindings, they are only available on Linux operating systems and do not have the level of abstraction desired to target the scripting tools for scientists and engineers. In response, I created a higher-level, cross-platform Python interface for HTCondor, called CondorPy, which is described in Chapter 4.

3 TETHYSCLUSTER: AUTOMATED PROVISIONING AND CONFIGURATION OF CLOUD RESOURCES

To support various modeling workflows and web applications with unpredictable workloads, a comprehensive computing toolkit needs to provide an automated way to provision scalable computing resources. Furthermore, water resources models have various computational requirements, so the tools need to be able to provide diverse resources to meet those requirements. The flexibility and virtually infinite scalability of the cloud offers the computing resources that are needed. However, provisioning cloud resources and configuring them into an HTC system capable of processing water resources models is a complex and time-consuming task. To make cloud-computing resources accessible from scripted modeling workflows and web apps, the process of provisioning and configuring cloud resources must be automated. StarCluster (described in section 2.4) is a Python tool that can automate provisioning cloud resources and configuring them with HTCondor to operate as an HTC system, but it does not provide the diversity of computing environments required for water resources applications since it can only provision clusters of Linux VMs on AWS (Figure 3-1). I created a modified version of StarCluster, called TethysCluster, that provides the necessary diversity (Figure 3-2).

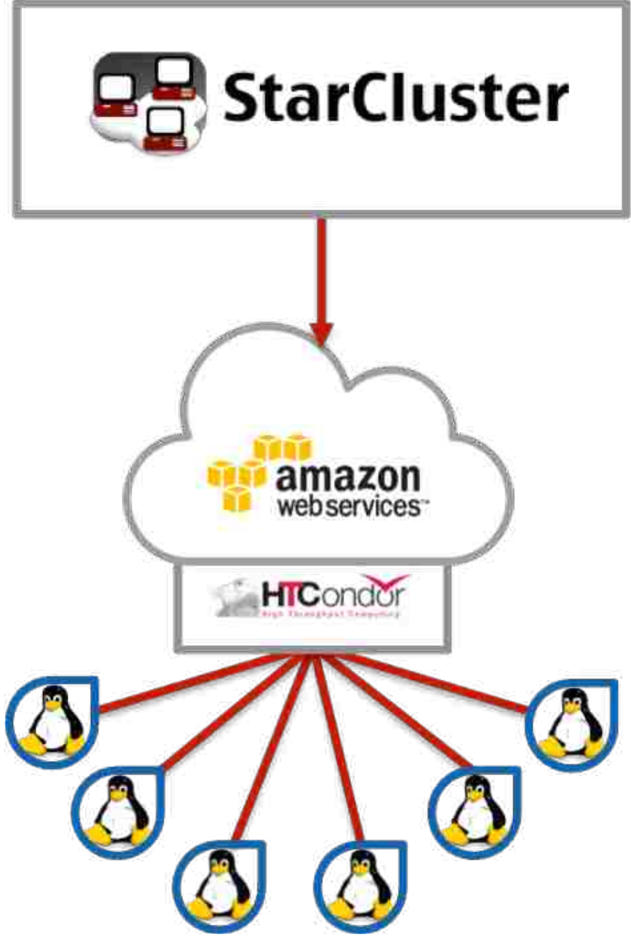


Figure 3-1. StarCluster can provision an HTCondor pool of Linux virtual machines on Amazon Web Services.

3.1 Software Description

TethysCluster is adapted from StarCluster to provide greater flexibility in setting up a computing environment by adding the capability to provision computing clusters with nodes running the Windows operating system and to provision clusters using Microsoft Azure (Figure 3-2). It maintains backwards compatibility with AMIs that are created for StarCluster (although some of the default configuration has changed).

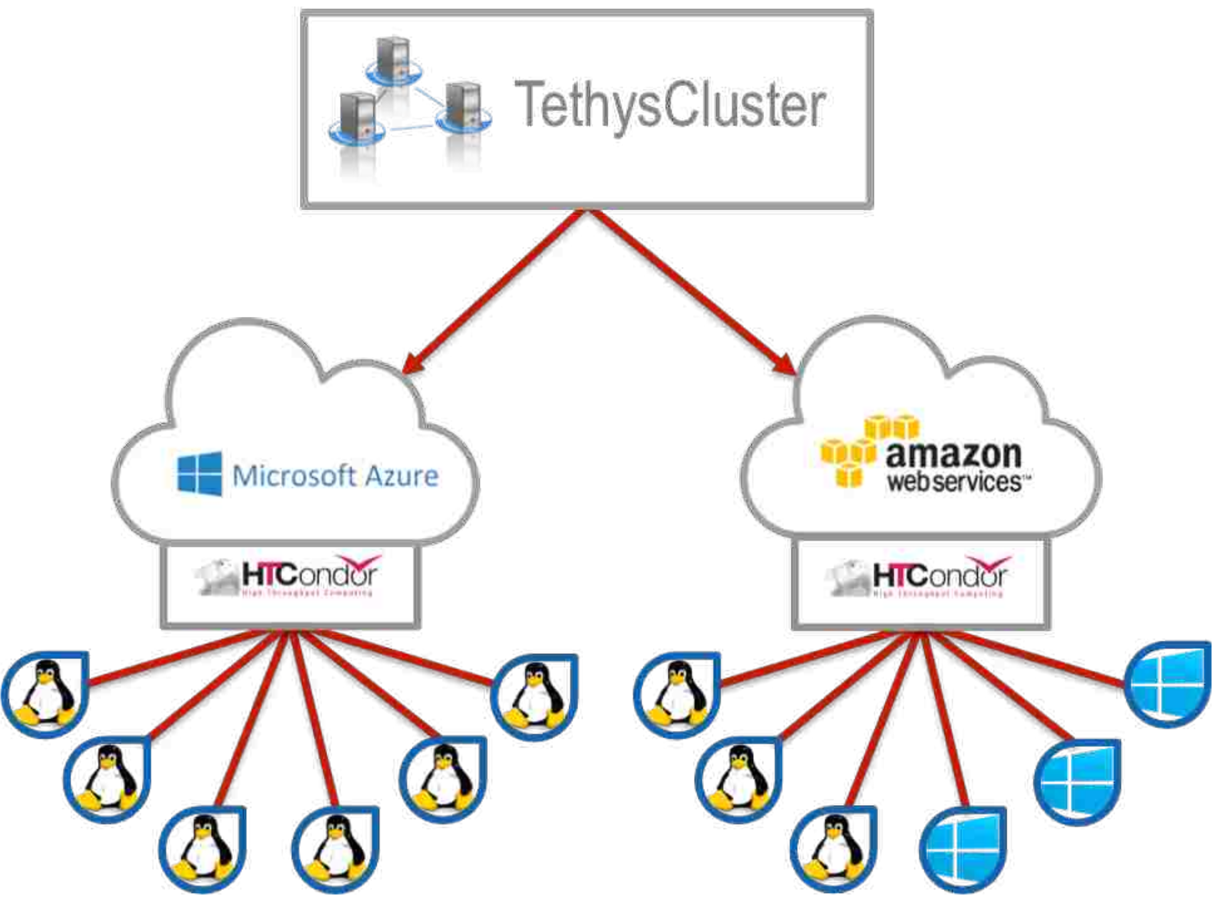


Figure 3-2. TethysCluster can provision HTCondor pools of Linux virtual machines on Microsoft Azure and both Linux and Windows Machines on Amazon Web Services.

The StarCluster code is organized with high-level classes like EasyEC2, Node, and Cluster that represent the cloud provider, a single VM, and the group of VMs that form the computing system, respectively (Figure 3-3). StarCluster is designed to work only with Linux nodes on AWS, thus the Node class has elements that are specific to Linux, and the EasyEC2 class is specific to the AWS EC2 API. TethysCluster maintains the abstraction used by StarCluster, but generalizes some of the classes so that they can provide a greater diversity of computing resources (Figure 3-4).

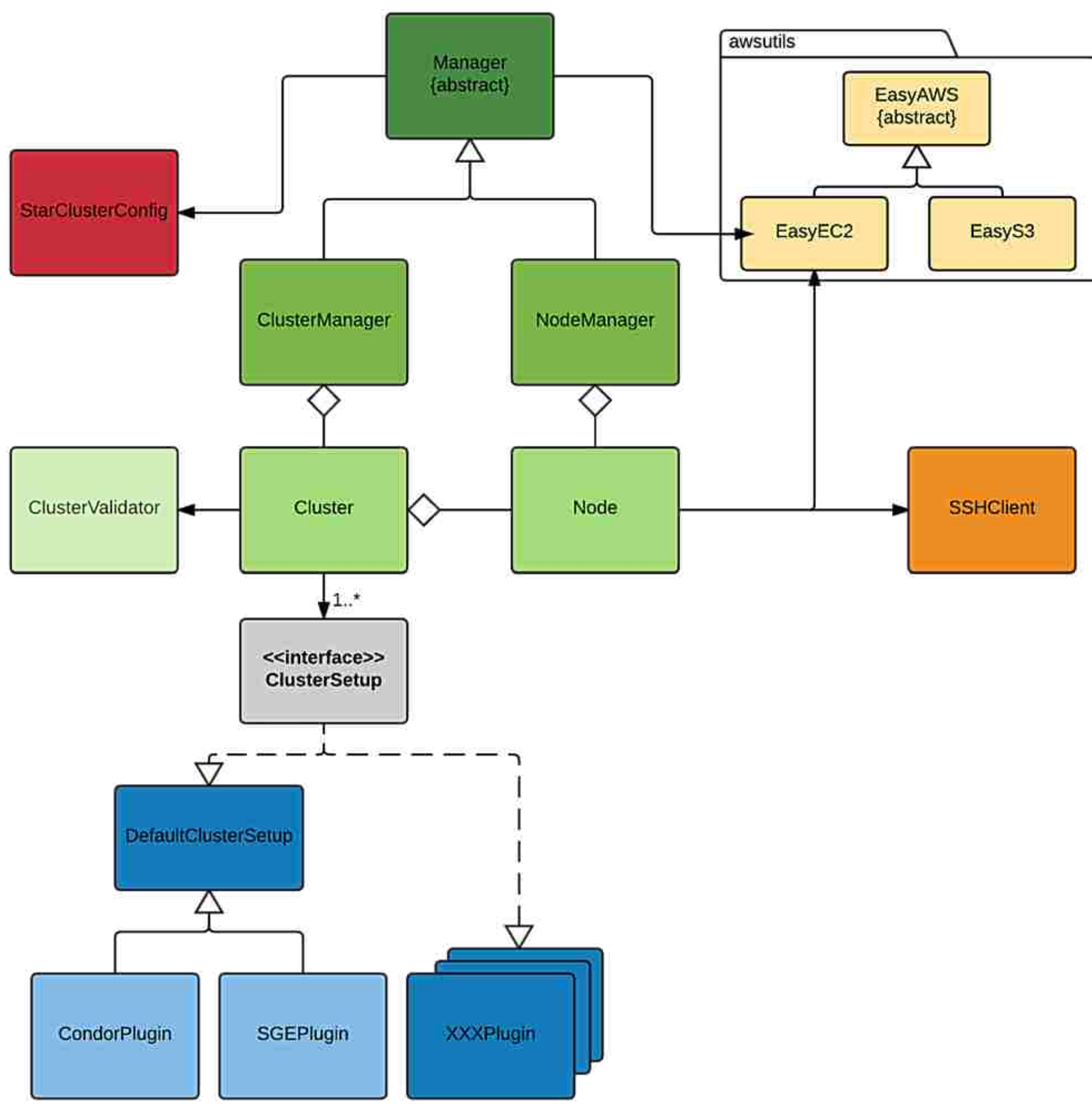


Figure 3-3. Unified Modeling Language (UML) diagram showing classes and their relationships in the StarCluster code.

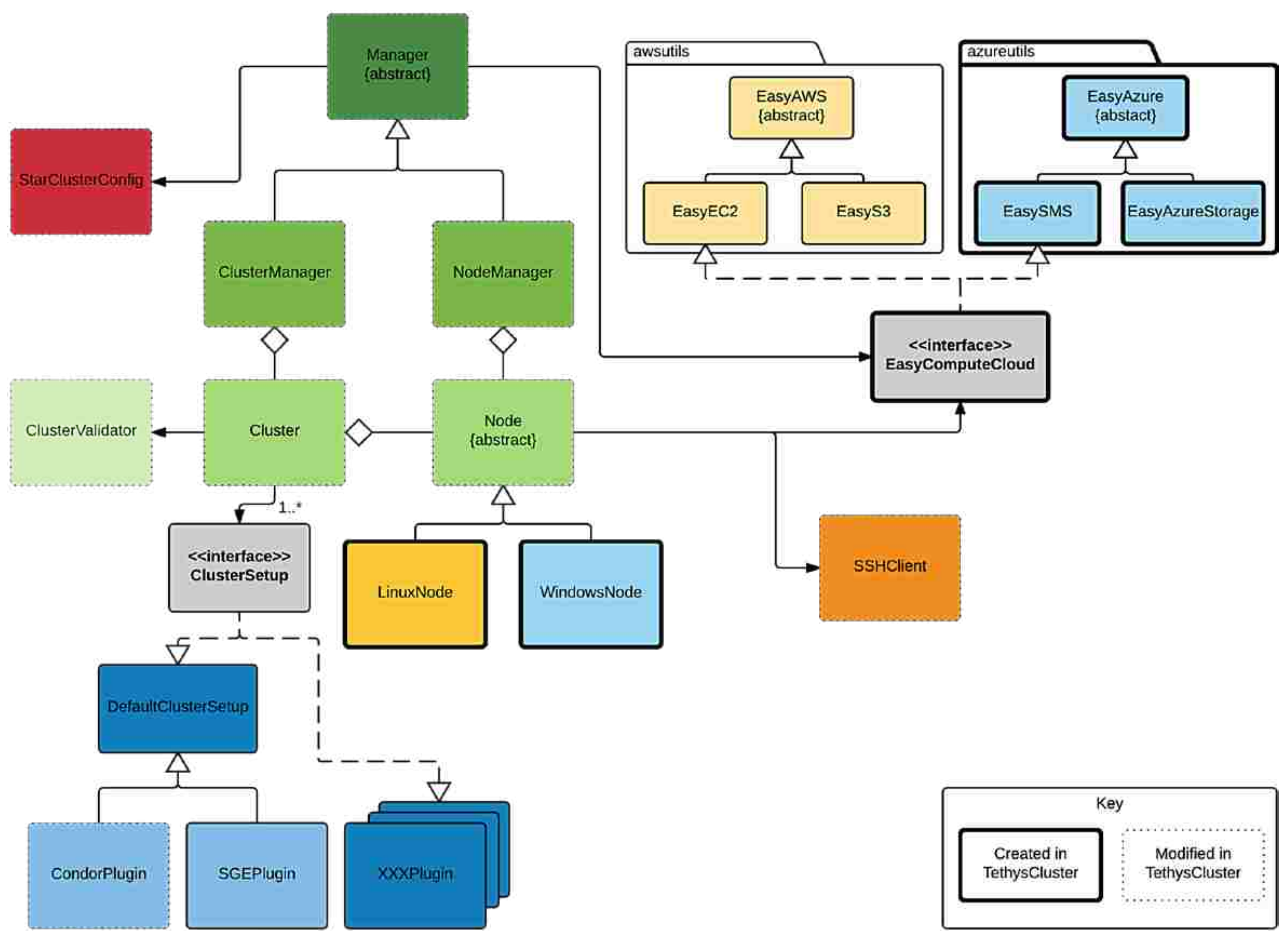


Figure 3-4. UML diagram showing classes that I modified and the classes and modules that I added in TethysCluster.

3.1.1 The Node Class

The Node class represents a single node in a computing cluster and has methods that handle the configuration of that node with the settings to make it function as part of the cluster. The TethysCluster code generalizes the Node class to serve as the base class from which operating system specific subclasses can be made. I extracted the methods of the Node class that had code that was specific to Linux and put then a LinuxNode subclass. These were methods that handled things like hostname configuration, managing user accounts, configuring Network File Systems (NFS), generating SSH keys, and package management. These methods became (in practice) abstract methods on the Node base class, meaning that subclasses would need to implement them with code specific to an operating system. To add support for Windows nodes, I created a WindowsNode subclass where I implemented these methods with code specific to the Windows operating system. For example, the *set_hostname* method in the StarCluster Node class used the Linux-specific program “hostname” to set the hostname of VM represented by the node. I moved this code into the LinuxNode subclass (Figure 3-5) and implemented the *set_hostname* method in the WindowsNode using the Windows Management Instrumentation Command-line (WMIC) tool (Figure 3-6). Additionally, Windows requires that the system be rebooted to reset the hostname, so the method also handles rebooting the system and waiting for it to come back up. A UML diagram with the inheritance relationship between the Node class and its subclasses is shown in Figure 3-7.

```
def set_hostname(self, hostname=None):
    """
    Set this node's hostname to self.alias

    hostname - optional hostname to set (defaults to self.alias)
    """
    hostname = hostname or self.alias
    hostname_file = self.ssh.remote_file("/etc/hostname", "w")
    hostname_file.write(hostname)
    hostname_file.close()
    try:
        self.ssh.execute('hostname -F /etc/hostname')
    except:
        if not utils.is_valid_hostname(hostname):
            raise exception.InvalidHostname(
                "Please terminate and recreate this cluster with a name"
                " that is also a valid hostname.  This hostname is"
                " invalid: %s" % hostname)
        else:
            raise
```

Figure 3-5. Implementation of the *set_hostname* method in the LinuxNode subclass.

```
def set_hostname(self, hostname=None):
    """
    Set this node's hostname to self.alias

    hostname - optional hostname to set (defaults to self.alias)
    """
    hostname = hostname or self.alias
    cmd_rename_computer = 'cmd /C wmic computersystem where ' \
                          'name=\'"%%computername%%"\' call rename ' \
                          'name="%(hostname)s"' % {'hostname': hostname}
    self.ssh.execute(cmd_rename_computer)
    current_hostname = self.ssh.execute('hostname')[0]
    if current_hostname != hostname:
        self.reboot()
    self.ssh.close()
    self._ssh = None
    time.sleep(30)
    self.wait()
```

Figure 3-6. Implementation of the *set_hostname* method in the WindowsNode subclass.

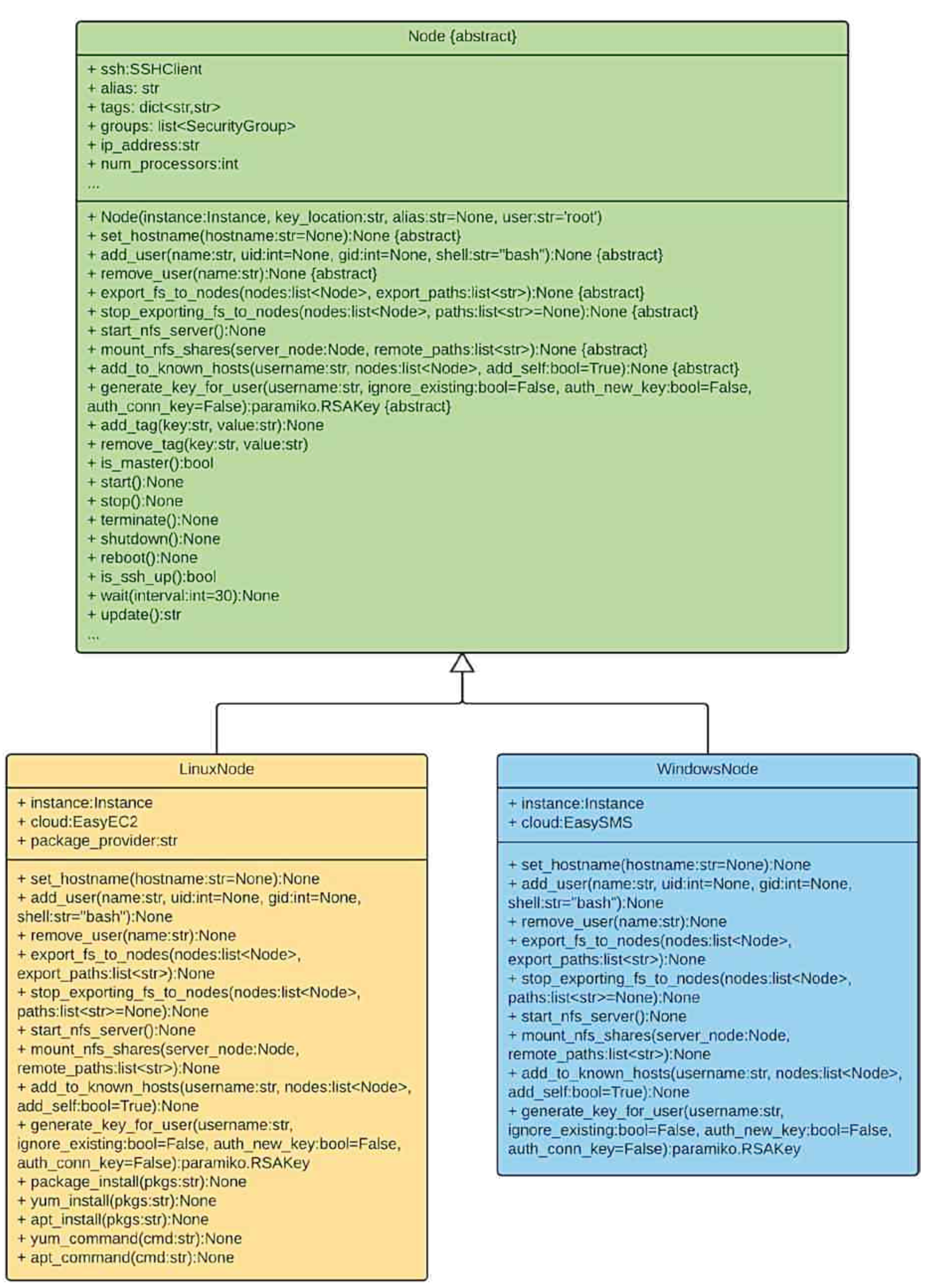


Figure 3-7. UML inheritance diagram showing the subclasses of the Node class.

StarCluster used a NodeManager class to create new nodes. In TethysCluster I modified the *make_node* method of this class to determine the type of node that should be created based on the VM instance that is passed in (Figure 3-8).

```
class NodeManager(managers.Manager):
    """
    Manager class for Node objects
    """
    PLATFORM_NODE_DICT = {
    'windows': WindowsNode,
    'linux': LinuxNode,
    }

    @classmethod
    def make_node(cls, instance, key_location, alias=None, user='root'):
        platform = instance.platform
        log.debug('Making node for platform %s' % (platform,))
        PlatformSpecificNode = cls.PLATFORM_NODE_DICT[platform]
        return PlatformSpecificNode(instance, key_location, alias, user)
```

Figure 3-8. The *make_node* method in the NodeManager class determines the type of node that should be made based on the platform attribute of the VM instance that is passed in.

Nodes are automatically configured into a unified computing cluster by running commands remotely using secure shell (SSH). Linux VMs come with SSH, so configuring them in this way is straightforward. Windows VMs, on the other hand, are generally not running SSH, and therefore a customized Windows image must be created and configured to do so. For the development of TethysCluster I configured the Windows VMs with the Linux emulator Cygwin, which provided the SSH service needed to connect to them remotely.

Providing a Windows VM with SSH is only part of what is required to automatically connect and configure it. SSH keys or certificates must also be configured to authenticate a connection to a VM. Typically, cloud providers automatically configure Linux VMs with the SSH key needed for the user to automatically connect to it, but Windows VMs, which are not

typically provisioned with SSH, do not get configured with the key. AWS provides a mechanism to download instance metadata, including the necessary SSH public key, from the VM by accessing a well-known server with the following permanent URI: http://169.254.169.254/latest/meta-data/public-keys/0/ (Amazon Web Services 2016b). In the Windows Amazon Machine Image (AMI) that I created for TethysCluster, I added a startup script that downloads the SSH key from the well-known server location and adds it to the list of authorized keys. Unfortunately, Azure does not provide a way to automatically retrieve SSH keys on Windows VMs and therefore TethysCluster is not able to automatically configure Windows clusters on Azure.

In summary, to allow TethysCluster to provision and configure Windows VMs (on AWS), I added a WindowsNode subclass to the TethysCluster code. I also created a custom Windows AMI and configured it with Cygwin, to provide an SSH connection, and with a startup script, to retrieve the SSH key required to automatically connect to and configure the VM.

3.1.2 Cloud Providers

StarCluster has a module called awsutils that is used to simplify using the AWS API in the ways that are needed by the StarCluster code. Generalizing this code to allow TethysCluster to function on multiple cloud providers was more complicated than generalizing the Node class. This is because the design of the StarCluster code is so tightly integrated with the AWS API and relies on features that were unique to EC2 that, to truly generalize it, would require redesigning the entire code base. Rather than pursuing this route, I wrote wrappers around the Azure API to give it the same interface as the AWS API in a module called azureutils (see Figure 3-3). Since there are fundamental differences between the architectures of the two clouds, there was not

always a perfect translation from the Azure API to the AWS API, but I made adaptations to allow Azure to function in at least the most vital ways for the purposes of TethysCluster.

StarCluster relies on the following AWS API objects: SecurityGroup, PlacementGroup, Instance, and Reservation. In the azureutils module I created corresponding classes that act as wrappers for Azure API objects, but that have same interface (or partial interface, as needed by the TethysCluster code) as the AWS objects. The SecurityGroup object wraps the Azure HostedService object and is discussed in detail later. The PlacementGroup in AWS represents a group of VMs that are located on the same physical hardware. This is inherently part of an Azure HostedService, so the custom PlacementGroup object is just a wrapper around the custom SecurityGroup object. The Instance object is a wrapper around the Azure RoleInstance, and a Reservation in AWS represents a request to provision one or more VM instances, so the custom Reservation object just contains a list of the custom Instance objects. These relationships and the parallel organization of the awsutils and the azureutils moduels is shown in the UML diagram in Figure 3-9.

The TethysClusterConfig class creates the object that represents the cloud provider. I modified the *get_easy_ec2* method in the StarCluster code to be *get_easy_cloud* and made it determine the cloud provider class to instantiate based on the configuration file Figure 3-10.

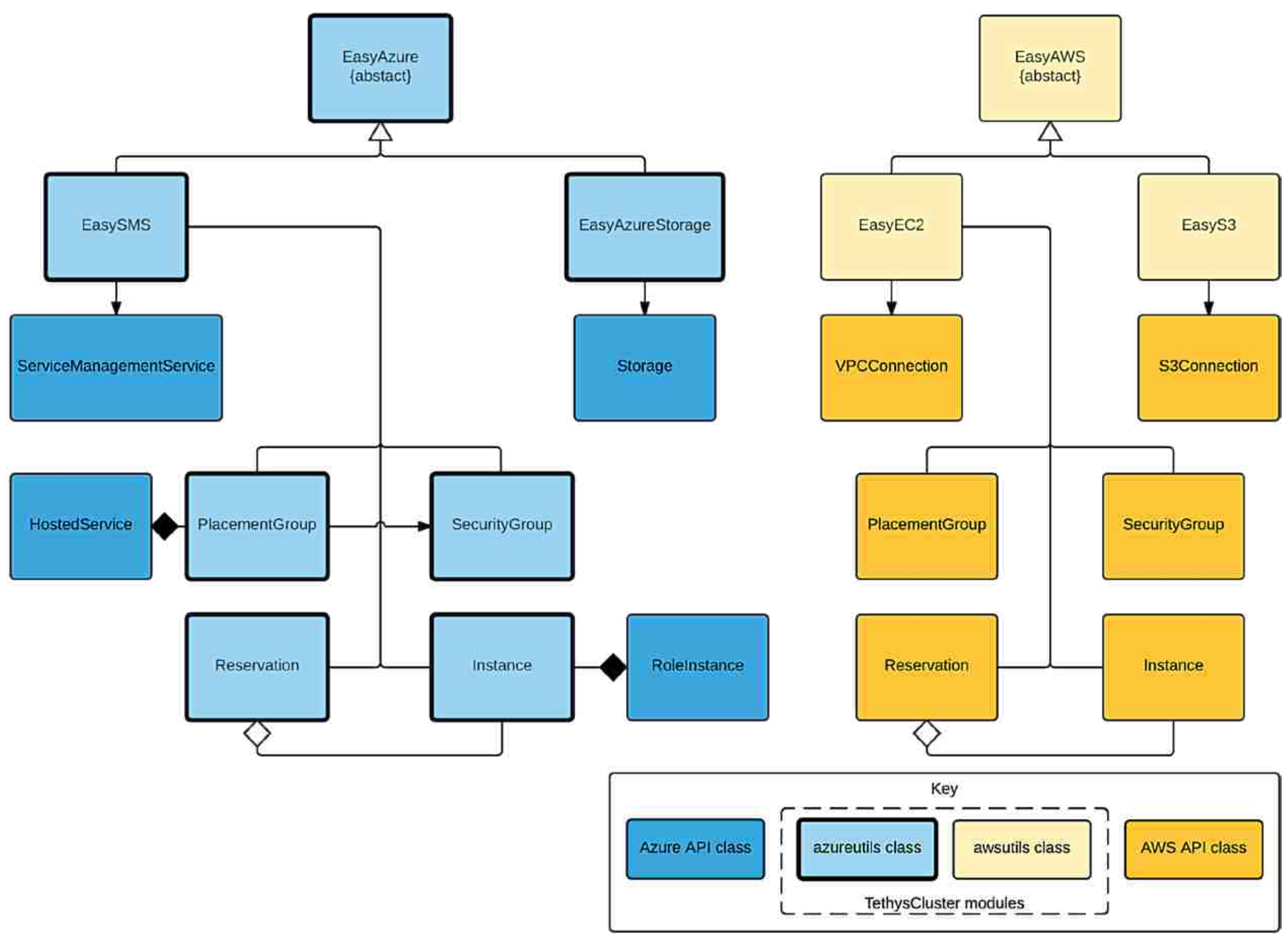


Figure 3-9. Detail UML diagram of the azureutils module showing the parallel structure to the awsutils module.

```python
def get_easy_cloud(self, provider=None):
    if not provider:
        template_name = self.get_default_cluster_template()
        provider = self.clusters[template_name]['cloud_provider']
    if provider == static.CLOUD_PROVIDERS['aws']:
        return self.get_easy_ec2()
    elif provider == static.CLOUD_PROVIDERS['azure']:
        return self.get_easy_sms()
    else:
        raise Exception('cloud provider %s not supported' % (provider,))
```

Figure 3-10. Modified method in the TethysClusterConfig class that determines which cloud provider class to instantiate.

A fundamental requirement that is at the core of the StarCluster design is that all the nodes in a cluster need to be able to communicate with each other. This requires that each node has a predefined hostname (e.g. master, node001, node002, etc.) and that the cloud provider allows network traffic between the nodes and handles hostname resolution (i.e. translating the hostname to the IP address). This way the images that nodes are made from can be pre-configured with the hostnames (and hostname patterns) for other nodes in the cluster. The key to getting this to work is what AWS calls a security group (SG) (Amazon Web Services 2016a). SGs work like a virtual firewall where rules for network traffic are set up. StarCluster uses SGs for two purposes. First, they are used to set network traffic rules allowing all of the VMs in a cluster to communicate with one another; it is important to note that AWS automatically handles hostname resolution within a security group. Second, SGs are used as a way of labeling or tagging a group of VMs with a common name so that they can be identified as a cluster. When StarCluster is used to provision a new cluster, it first creates a corresponding SG. Any VMs that are then provisioned in that cluster are placed in that SG.

Although Azure recently created what is called a network security group (NSG), which has many of the same purposes and functionality as an SG on AWS, there is a key difference:

hostname resolution is not handled automatically. This prevents TethysCluster from being able to use NSGs on Azure in the same way that SGs are used on AWS because the configuration of the clusters will not work without automatic hostname resolution.

To circumvent this problem, TethysCluster relies on what Azure calls cloud services, which are represented by the HostedService class in the Azure API. This gets down to a fundamental difference between AWS and Azure. In AWS's EC2 the basic computing unit is a VM instance, which is essentially a virtual server. On Azure the basic computing unit is a cloud service (formerly hosted service). Cloud services were designed to support scalable web apps, which traditionally have a web front end and a back end that handles the heavier computations. Reflecting this structure, cloud services originally provided two roles (or types of VMs): a web role and a worker role, each of which were configured using code that was bundled into a special file called a cloud service package. Each role could have multiple instances, allowing the web app to scale as needed. Therefore, a cloud service is a container that can host a group of role instances; however, it only has a single public IP address and domain name. Eventually Azure added support for the more traditional virtual machine architecture but did this by just creating a new role type, called a VM role, in a cloud service. The implications of this are that there can be multiple instances of a VM role in a cloud service that are all grouped behind one public IP address (Plankytronixx 2014). Since Azure does handle hostname resolution automatically within a cloud service (Microsoft 2016), and since a cloud service naturally groups VM instances together, TethysCluster can use cloud services to accomplish the same two purposes that SGs do on AWS.

However, a consequence of using cloud services is that although each VM instance has its own private IP address and hostname, which allows them to communicate with each other

internally, there is only a single public IP address and domain name for all of the VMs in the cloud service. This becomes problematic when connecting to individual nodes externally because port 22, the default port for SSH connections, can only be mapped to one of the VMs in the cloud service. TethysCluster solves this problem by mapping the local port 22 on each VM to a unique external port. The master node is mapped to the external port 22 and all other nodes are mapped using the pattern '22' + [node-index], where node index is the index of the node (e.g. the index for node001 is 001).

Another consequence of using cloud services is that the name of the cloud service is used as the domain name. This creates some problems with the naming convention used for SGs. StarCluster uses a naming convention to easily identify the SGs that represent clusters (i.e., that were created by StarCluster), and to ensure that the names are unique from other SGs that are created outside of StarCluster. The slightly modified convention used in TethysCluster is: @tc-[cluster-name], where [cluster-name] is replaced by the name the user gives to the cluster. SG names must be unique within an AWS account, which means that a user cannot create two separate clusters with the same name. On Azure the name of the cloud service is also used as the domain name, thus the naming convention is invalid (the @ symbol cannot be used in a domain name). Moreover, domain names must be universally unique; so if a user of TethysCluster somewhere in the world created a cluster named 'mycluster' then the corresponding domain name would be taken and no other user of TethysCluster anywhere in the world could create a cluster with the same name. It is reasonable to expect a user to create unique names for clusters on their own account, but not have to create a universally unique name. To circumvent these challenges, TethysCluster translates between the naming convention used throughout the code and a separate naming convention used to name cloud services on Azure, which follows the

pattern: tc-[cluster-name]-[azure-subscription-id], where [azure-subscription-id] is a unique identifier for the subscription being used.

StarCluster has utilities to aid in creating and verifying SSH keys that are required to provision and connect to VMs on AWS. Azure uses SSH keys as well as Secure Sockets Layer (SSL) certificates. I added utilities for working with SSL certificates to TethysCluster to facilitate working with VMs on Azure.

To summarize, the design of the StarCluster code is so integrated with the AWS API that generalizing it would require an almost entire rewrite of the code. To avoid this, I added support for Azure by wrapping Azure API objects in custom objects that provide the same interface as the AWS API objects. These custom objects essentially translate between the Azure objects and the interface that is expected throughout the rest of the code and thus prevent the need to completely rewrite the code in TethysCluster.

3.2 Discussion

The primary purpose of creating TethysCluster was to allow clusters to be created with Windows nodes so that computing resources could be automatically provisioned for models that required a Windows environment. While StarCluster offers a rich set of features for working with Linux nodes on AWS, TethysCluster currently only supports the most basic and essential features for Windows nodes. The major features that TethysCluster does not yet support for Windows nodes are being able to attach Elastic Block Store (EBS) volumes, configuring NFS, and running graphical applications with X Windows System (X11). Though these features would be useful on Windows nodes, they were not the primary focus of this research. Future research might look at the possibility of adding these features.

As was discussed earlier, there are a few customizations that are required in order for a Windows AMI to work with TethysCluster. This is not unusual since the Linux AMIs also require some pre-configuration. The standard configuration for Linux nodes requires installing some library dependencies, installing SGE and HTCondor, installing Python and any packages desired, configuring NFS, and configuring root login though SSH. The configuration for the Windows AMIs included installing Cygwin and several of the Cygwin packages (including SSH), configuring the SSH server and allowing root login, opening port 22, setting up a startup script to download the public SSH key, and installing HTCondor. Adding the features mentioned above will likely consist of mostly (if not entirely) additional configuration of the Windows AMI.

It is unfortunate and a bit ironic that Windows nodes could not be configured to work on Azure. The only configuration step that is dependent on the cloud provider is getting the public SSH key. AWS provides a URI that can be accessed by the node to download the key; however, I was unable to find a comparable solution or work-around for Azure. Azure is a quickly evolving cloud platform, and has recently released preview versions of a new Python API. It is possible that a mechanism for retrieving the key will soon be available.

If the primary purpose of TethysCluster was to create a way to provision a cluster of Windows nodes, which was successfully accomplished with AWS but not with Azure, then what was the benefit of providing support for Azure? While the primary purpose was to create diversity of computing environments, the underlying objective of a comprehensive computing toolkit is to make computing more accessible to modelers. It is possible that in some situations a modeler may have access to one cloud and not to another (for example, because of grants,

affiliation, or sponsorship). By making TethysCluster compatible with both Azure and AWS it becomes a more flexible tool that provides greater access to computing resources.

So, if adding support for Azure was beneficial, why weren't other cloud providers also supported? Ideally, TethysCluster would be compatible with any and all cloud computing providers. Unfortunately, because the cloud computing industry originated in the private sector and grew so rapidly, standards were not established, and each provider created their own infrastructure and API (Monteiro, Pinto et al. 2011). There are some efforts to create interoperable APIs that interface with multiple cloud provider APIs, such as DeltaCloud (Apache Software Foundation 2011); however, these currently only provide support for basic capabilities that are common among most providers (like provisioning and deleting a VM). Since TethysCluster has very specific requirements to enable it to configure VMs as a computing cluster, APIs like DeltaCloud do not provide the support needed. StarCluster was designed around the capabilities of AWS, and because of this research, suitable adaptations were made in TethysCluster to allow it to function with Azure. It is possible that there are other clouds that also have the features to support TethysCluster, but additional research would be needed to evaluate other clouds and write wrappers around their APIs to function in TethysCluster.

4 CONDORPY: A PYTHON INTERFACE TO HTCONDOR

Modeling workflows and web applications need the capability to dynamically create, submit, and monitor jobs in a scripting environment. For example, a user of a modeling web app will enter job parameters into a web interface, and the server-side code of the app will need to package the parameters into a job and submit it to a scheduling system to be executed. HTCondor pools provisioned with TethysCluster as described in Chapter 3 provide the compute resources and scheduling system. The next step is to create a Python library that can take advantage of all of the necessary features of HTCondor to create, submit, and monitor jobs. While HTCondor currently has Python bindings that could be used in a scripting environment, they are only available on the Linux distribution, and they are designed to be low-level, targeting power users who have extensive knowledge of HTCondor's ClassAds and daemons. For the objectives of a comprehensive computing toolkit, a high-level, cross-platform HTCondor interface is needed. Work done previously at Brigham Young University used high-level Python scripts for performing stochastic analysis with hydrologic models using HTCondor (Taylor 2013). These scripts would use Python to write a job description file and then would submit it using the HTCondor CLI. This allowed HTCondor to be programmatically accessed from Windows through Python. I used this same pattern to develop CondorPy, but designed it to be more general and expandable.

4.1 Software Description

CondorPy is a simple Python interface to HTCondor that facilitates using HTC to execute large modeling tasks from a scripting environment. To accomplish this, I designed the code to meet the following objectives: (1) provide a high-level, job-centric abstraction; (2) have cross-platform functionality; (3) allow remote job scheduling; and (4) offer a job template system to provide reasonable defaults and eliminate repetitive job configuration. The UML diagram in Figure 4-1 shows a class relationship diagram of the classes that I developed to meet these objectives. The following subsections describe how these classes were implemented.

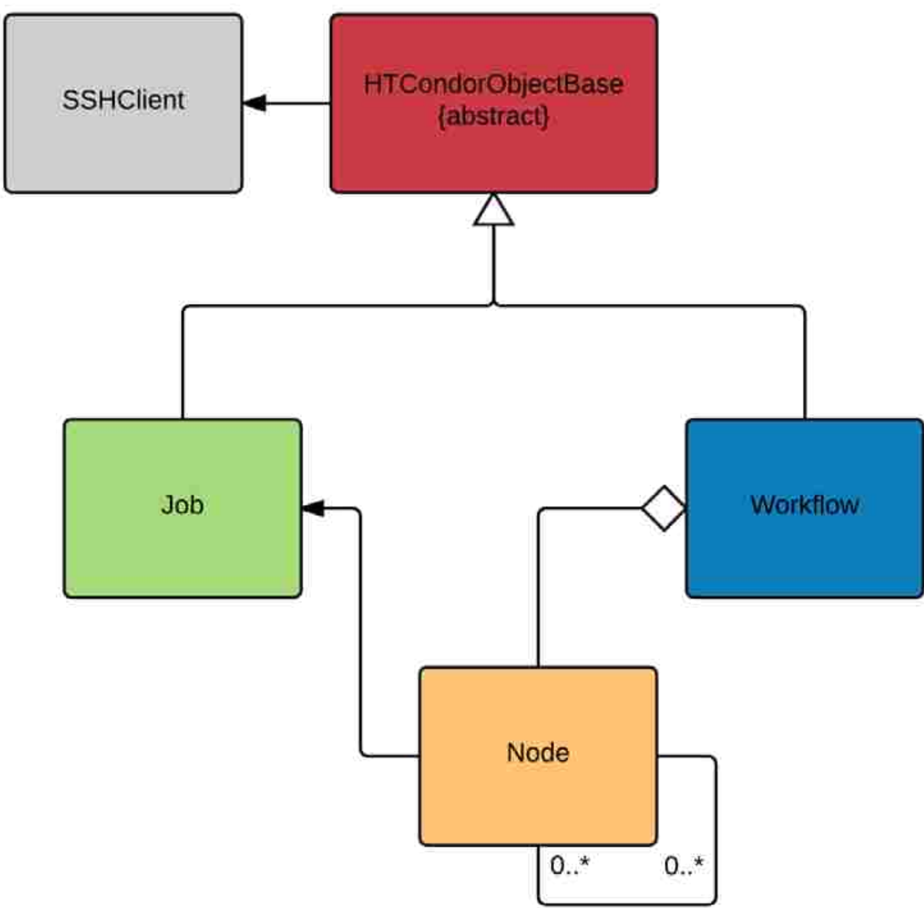


Figure 4-1. Class relationship UML diagram for the classes in CondorPy.

4.1.1 Job-Centric Abstraction

CondorPy uses a high-level, job-centric abstraction, allowing the modeler to focus more intuitively on the computing job and not have to worry about the various system components of HTCondor, such as the scheduler, the job queue, etc. A job-centric abstraction means that a user of CondorPy can create a job object and perform all needed actions (i.e. submit, wait, status, remove, etc.) directly on that object. Figure 4-2 is a detailed UML diagram of the Job class showing its properties and methods.

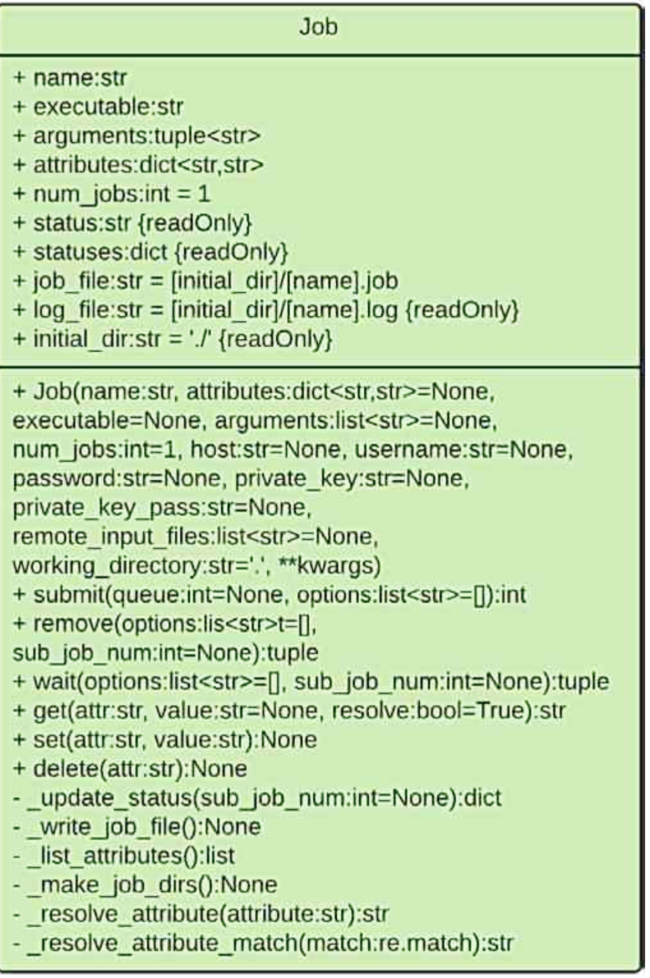


Figure 4-2. UML diagram for the CondorPy Job class.

The only required parameter when creating a job object is the name. The job name not only identifies the job, but also is used to create a job submission file, log files, and the initial directory for the job. The initial directory acts as the home base for that job and is where the job submission file and the log file(s) get written and where any files created by the job are copied back to (when using the file transfer option in HTCondor). The initial directory is automatically created in the job's working directory, which is, by default, the system's current working directory (CWD) when the code is executed, but can be set as any directory. The input files that are required for the job are referenced relative to the initial directory. It is possible to use a different name for the initial directory, and even the name of a directory that already exists, but by default a directory is created using the name given to the job.

Additional job attributes can be passed in as a Python dictionary when creating a job object or added later. Job attributes are key-value pairs and can be any of the submit description file commands that are described in the HTCondor user manual (Condor Team 2014). The executable property, while not required when creating a job object, must be set before the job can be submitted. Unlike the input files, which are defined relative to the initial directory, the executable is defined relative to the job's working directory. This is a result of the way HTCondor handles these parameters.

The Job class in CondorPy provides as set of action methods, such as *submit*, *wait*, *remove*, and *edit,* that allow job objects to interact with HTCondor. The commands rely on the HTCondor CLI, which is discussed more in the following section. Objects created from the Job class also have several properties. Notably, the *status* property is dynamically updated by calling the appropriate CLI command. Since a job can be made up of multiple sub-jobs each with their own status, the *status* property represents the status of all of the sub-jobs. If the status of all of

the sub-jobs is the same, then that is the value of the *status* property. If the sub-jobs have different statuses, then the value of *status* will be '*various*'. There is an additional property called *statuses* that returns a list of the individual sub-job statuses so that the actual status of each sub-job can be examined.

In addition to the Job class, CondorPy also provides a Workflow class to represent a DAG. The Workflow class follows the same pattern as the Job class, but allows users to create a set of jobs, define hierarchical relationships among them, and submit them all as a single workflow. To define the relationships between jobs, CondorPy uses node objects. A node is a wrapper for a job object that has additional attributes to define parent and child relationships to other node objects. Figure 4-3 and Figure 4-4 show UML diagrams listing the properties and methods of the Workflow class and the Node class, respectively. The relationships among the jobs in a workflow can be as simple or complex as needed, or even have no relationship to each other at all (as would be the case for a simple parameter sweep. Figure 4-5 shows various diagrams of possible job relationships within a workflow. The purple boxes represent the nodes and the red arrows indicate a relationship going from the parent node to the child node.

4.1.2 Cross-Platform

One of the objectives of CondorPy is to be cross-platform so that it can be used in any environment that HTCondor can run in, which in many cases for water resources modeling is a Windows environment. This means that it cannot use the native Python bindings of HTCondor since these are only available on the Linux platform. For this reason, CondorPy is built around HTCondor's CLI, which has the same interface on all platforms. Using the CLI also allows

```
Workflow
---
+ name:str
+ cluster_id:int = 0 {readOnly}
+ node_set:list<Node> {readOnly}
+ dag_file:str = [name].dag {readOnly}
---
+ Workflow(name:str)
+ add_node(node:Node):None
+ add_job(job:Job):Node
+ submit(options:list<str>=[]):int
+ wait(options:list<str>=[]):None
+ complete_node_set():None
- _write_dag_file():None
```

Figure 4-3. UML diagram for the CondorPy Workflow class.

```
Node
---
+ job:Job
+ pre_script:str
+pre_script_args:str
+ post_script:str
+ post_script_args:str
+ parent_nodes:set<Node> {readOnly}
+ child_nodes:set<Node> {readOnly}
+ retry:int = 0
---
+ Node(job:Job, parents:set<Node>=None,
children:set<Node>=None, variables:dict=None,
pre_script:str=None, pre_script_args:str=None,
post_script:str=None, post_script_args:str=None,
retry:int=0)
+ add_parent(parent:Node):None
+ remove_parent(parent:Node):None
+ add_child(child:Node):None
+ remove_child(child:Node):None
+ get_all_family_nodes():set<Node>
- _get_all_ancestors():set<Node>
- _get_all_descendents():set<Node>
- _link_parent_nodes():None
- _link_child_nodes():None
+ list_relations():str
+ list_scripts():str
+ list_options():str
- _get_child_names():str
- _get_parent_names():str
- _get_names(nodes:set<Node>):str
```

Figure 4-4. UML Diagram for the CondorPy Node class.

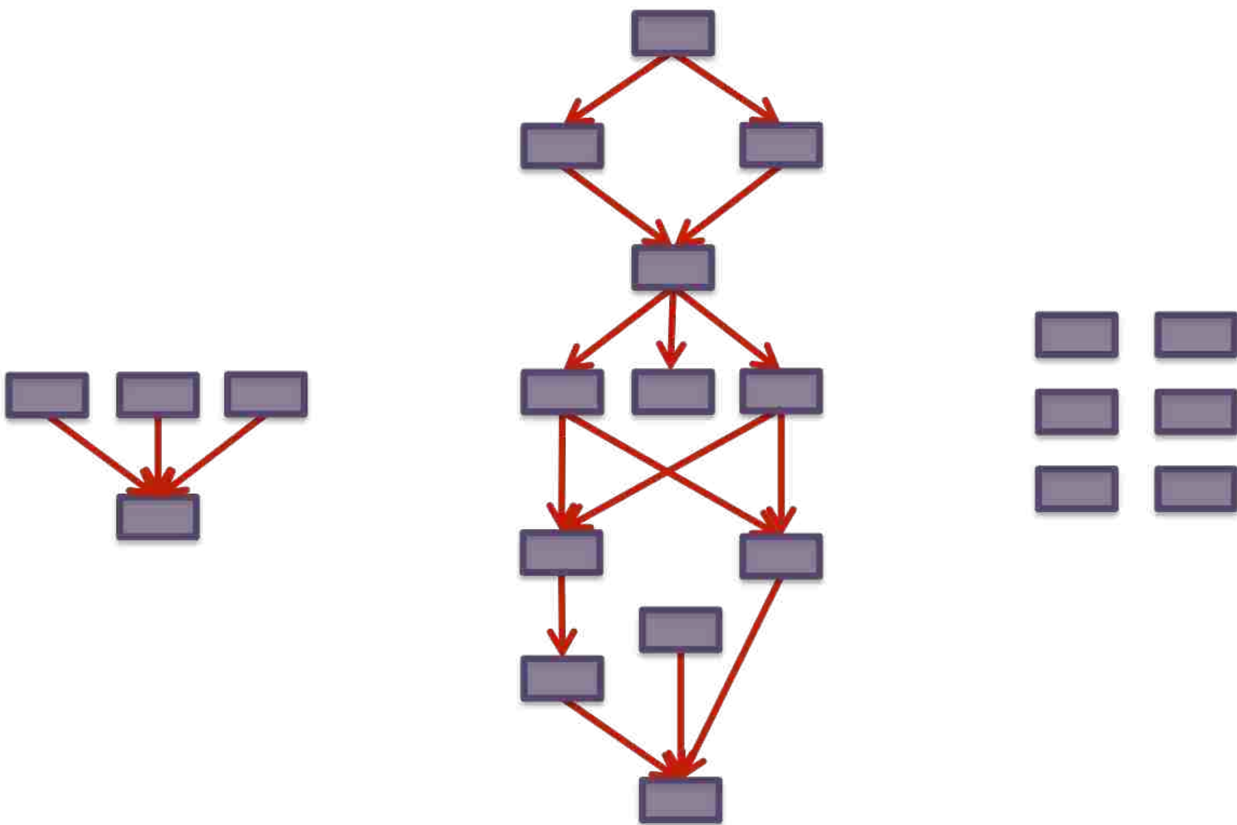


Figure 4-5. Diagrams of various workflows showing the levels of complexity that can be represented in CondorPy.

CondorPy to use the more intuitive language of the job description file, rather than the ClassAd language used by the native Python bindings.

CondorPy wraps the HTCondor CLI commands that interact with the job in methods on the Job and Workflow classes. All of these methods provide reasonable default options, but can accept custom options. Any options that can be passed to the CLI commands can be passed to these methods as a string or list of strings. It is important that the system's CWD is set as the job's working directory when these methods are executed so that relative paths can be resolved properly. By default, the job's working directory is the system's CWD, but if a different working directory is defined for the job then CondorPy changes the system's CWD to be the job's

working directory before executing methods that require it and then change it back once the method returns. This is done using a custom Python decorator that I created (Figure 4-6).

```
def set_cwd(fn):
    """
    Decorator to set the specified working directory to execute the function,
    and then restore the previous cwd.
    """
    def wrapped(self, *args, **kwargs):
        log.info('Calling function: %s with args=%s', fn, args if args else [])
        cwd = os.getcwd()
        log.info('Saved cwd: %s', cwd)
        os.chdir(self._cwd)
        log.info('Changing working directory to: %s', self._cwd)
        result = fn(self, *args, **kwargs)
        os.chdir(cwd)
        log.info('Restored working directory to: %s', os.getcwd())
        return result
    return wrapped
```

Figure 4-6. Code for the custom decorator that ensures HTCondor CLI commands are executed from the job's working directory.

To submit a job using the CLI, CondorPy must write the job description file to disk and then execute the CLI submit command. To identify and interact with the job later, CondorPy saves the job ID that is created when the job is submitted. Since the CLI is used to execute commands to HTCondor, getting feedback from the commands requires parsing the output from the command line. Regular expressions are used to match the output and extract the pertinent parts (e.g. the job ID). This is not as efficient as using the native Python libraries, but the overhead from doing this is generally small in comparison to the runtime of the job.

```python
def submit(self, args):
    """
    Submits job or workflow and parses the cluster id from the output.

    Args:
        args (list or tuple of str): A list of command line arguments with
        the first argument being the submit command.
    """
    out, err = self._execute(args)
    if err:
        if re.match('WARNING',err):
            log.warning(err)
        else:
            raise HTCondorError(err)
    log.info(out)
    try:
        self._cluster_id = int(re.search('(?<=cluster |\*\* Proc )(\d*)', out).group(1))
    except:
        self._cluster_id = -1
    return self.cluster_id
```

Figure 4-7. The submit method of the HTCondorObjectBase class, which parses the output of a submit command using a regular expression to get the cluster id.

4.1.3 Remote Scheduling

Remote job submission decouples the submitting machine from the HTCondor computing resources by not requiring that HTCondor be installed on the submitting machine. This feature of CondorPy accomplishes many of the same things as the HTCondor add-on package RemoteCondor (Condor Team 2010), but it does it in different ways. First, RemoteCondor is written in Bash, whereas the remote scheduling component of CondorPy is written in Python, allowing CondorPy to maintain cross-platform functionality. Secondly, while RemoteCondor is configured to work with a single remote scheduler, in CondorPy each job can be configured to submit to a separate scheduler. Both RemoteCondor and CondorPy execute HTCondor commands on the remote machine via SSH, however RemoteCondor uses SSH Filesystem (SSHFS) to mount the remote file system locally to accommodate file transfer, while CondorPy uses a Python implementation of Secure Copy Protocol (SCP) to transfer files to and from the remote scheduler, again to maintain cross-platform compatibility. RemoteCondor requires that all of the necessary files are located on the mounted remote file system. CondorPy

requires that the user provides a list of local paths to the files that will be copied over to the remote scheduler. This list is the *remote_input_files* property of a CondorPy job object. CondorPy creates a directory on the remote scheduler using a universally unique identifier (UUID) for the name, and all the files listed in the *remote_input_files* property are copied into that directory. The job's initial directory is also created in that remote directory, so when specifying file paths in the job attributes, such as *executable*, or *input_files,* they should be the paths relative to the initial directory on the remote scheduler.

The remote scheduling feature of CondorPy allows modelers to use an HTCondor computing pool without having to install and configure HTCondor on their computer. This is just one of the ways that CondorPy eases the burden of using HTCondor for modeling. Also, this is a valuable feature for integrating HTCondor into web applications where having HTCondor installed on the front end web server may not be desirable.

All of the code that is needed to interact with the remote schedule is the HTCondorObjectBase class, which is the parent class for both the Job class and the Workflow class. This gives both subclasses the remote capability while keeping the code for handling remote interactions in one place. A UML diagram showing the properties and methods of the HTCondorObjectBase class is shown in Figure 4-8.

HTCondorObjectBase

+ cluster_id:int = 0 {readOnly}
+ remote_input_files:tuple<str>
- remote:SSHClient
- remote_id:str
- _cwd:str = '.'

+ HTCondorObjectBase(host:str=None, username:str=None, password:str=None, private_key:str=None, private_key_pass:str=None, remote_input_files:list<str>=None, working_directory:str='.',)
+ submit(queue:int=None, options:list<str>=[]):int
+ sync_remote_output():None
+ close_remote():None
- _execute(args:list<str>):tuple
- _copy_input_files_to_remote():None
- _copy_output_from_remote():None
- _open(file_name:str, mode:str='w'):file_descriptor
- _make_dir(dir_name:str):None

Figure 4-8. UML diagram for the CondorPy HTCondorObjectBase class.

4.1.4 Job Templates

To further simplify job creation, I created job templates, which make it easy to include a group of commonly used job description attributes together with reasonable defaults where possible. For example, the base template configures all of the logging files. The *vanilla_transfer_files* template builds on the base template and adds job attributes needed for a job running in the vanilla universe using the file transfer mechanism. By using job templates, a modeler need only worry about a few key job attributes such as the job executable and the input files, while attributes that deal with the Condor environment are abstracted away. The sample in Figure 4-9 illustrates how to create a job from a job template and submit it with CondorPy using a few short lines of Python code.

```
from condorpy import Job, Templates


job = Job('job_name', Templates.vanilla_transfer_files)
job.executable = 'job_script.exe'
job.transfer_input_files = 'input_1.in input_2.in'
job.transfer_output_files = 'output.out'
job.submit()
```

Figure 4-9. Code sample for submitting an HTCondor job with CondorPy.

While Job templates ease the burden on the modeler by preconfiguring many job attributes, any attribute from a template can be overwritten. Thus templates can be used to simplify the process of creating a job, but they do not restrict the user from accessing any of the options of HTCondor. In addition to the job templates that I included in CondorPy, I also created a way for custom templates to be loaded from file.

4.2 Discussion

I created CondorPy to facilitate automating modeling tasks that require HTC by providing a simple interface for HTCondor. In this section I describe the motivation and rational for the design choices and the broader implications of CondorPy.

First, why does the hydrologic modeling community need a Python interface for HTCondor? Before addressing the specifics of this question it is helpful to first answer the more general question of why a scripting language is needed to access computing resources. As hydrologic models are becoming more sophisticated, they typically require larger amounts of data, and larger amounts of computing resources. The data are often spatially distributed and may come from many different sources and in many different formats. Creating scripts to

retrieve, combine, and reformat data can save a significant amount of time, especially if the process needs to be repeated. Also, these scripts document the provenance of the data that is used in a model, which is critical for research. When modeling requires large-scale computing, such as an HTC system, then the process of executing the models also becomes much more complex, and scripting the process yields the same benefits of time-savings and reproducibility. Furthermore, scripting the entire processes from data manipulation to model execution is necessary for automating modeling workflows, which can be deployed on the web to make them more easily accessible to end users. Thus, having a scripting interface to access computing resources facilitates the modeling processes. So, turning back to the original question of why a Python interface is needed for HTCondor, there are two specifics that still need to be addressed: why Python?, and why HTCondor?

Python is a general-purpose, cross-platform language that has an expressive and easily-readable syntax and is widely adopted by the scientific community for scientific computing, web development, and for "steering" workflows (Oliphant 2007; Pérez, Granger et al. 2011). Since CondorPy was designed to be used by scientists and engineers to facilitate accessing computing resources in modeling workflows and in a web environment, Python was a natural choice.

HTCondor is one of many job scheduling and managing middleware that could be used to provide access to large-scale computing resources. It is a leader, however, in its ability to scavenge idle processing time from regular workstations, making it possible for modelers to gain access to an HTC system without the need to invest in additional infrastructure. Additionally, HTCondor is cross-platform and has the flexibility to be used with almost any computing resource including cloud resources.

If, as has been stated, the modeling community does indeed need a Python interface for HTCondor, then why aren't the native Python bindings sufficient, or at the very least why doesn't CondorPy use these bindings to build from? Although, the native Python bindings are more efficient than parsing the output from the CLI, they are only available on Linux. The CLI, on the other hand, has the same interface on all platforms allowing CondorPy to be cross-platform, which is important as many hydrologic models natively run in Windows. The use of the CLI also allows CondorPy to use the notation of the submit description file, which is more human-readable than the ClassAd notation, making CondorPy more approachable by those not well-versed in HTCondor.

I have explained the rationale for selecting the foundational components that I used to build CondorPy, and I will now focus on the design objectives. The four design objectives of CondorPy emerged while working to generalize the Python scripts created by Taylor (2013) for use in other modeling applications, including web-based applications, and I formed each objective with the goal of simplifying the process of working with HTCondor. The first objective, providing a job-centric abstraction, allows modelers to work with the code in a natural way, and makes the code easy for others to read and maintain. Next, having cross-platform functionality gives modelers the flexibility to work in the environment most suited to their model. Providing remote job scheduling has two purposes: First, it is common for modelers to work from laptops where it may not make sense to install HTCondor; remote job scheduling allows for jobs to be submitted from a computer where HTCondor is not installed as long as it can connect to a remote scheduler. The second reason is for using CondorPy in a web environment; remote submission allows the duties of an HTCondor agent (e.g. maintaining a job queue and advertising jobs to the central manager) to be offloaded from the main web server.

Finally, the job template system was created to eliminate the repetitive parts of creating a new job, such as setting up logging files, and to reduce the number of job settings that the user needs to specify.

5 COMPUTING TOOLS IN TETHYS PLATFORM

One of the main motivations for creating a comprehensive computing toolkit is to provide a way for accessing computing resources in a web development environment. While it is increasingly popular to deploy complex modeling workflows as web applications (or web apps), integrating all of the components required for such apps can be a significant barrier (Swain, Latu et al. 2015). As part of a National Science Foundation-funded project, called CI-WATER, that enhanced the cyberinfrastructure for water resources, we developed a tool called Tethys Platform which lowers this barrier by providing a web development platform that incorporates many of the visualization, geospatial, and computational components needed in water resources web apps (Swain, Christensen et al. In Press). In addition to other contributions, I added the Tethys Compute module to Tethys Platform. Tethys Compute leverages TethysCluster and CondorPy to provide a comprehensive computing toolkit within the Tethys Platform development environment. Access to these tools is provided through the Compute API and the Jobs in the Tethys SDK as well as through the Tethys Compute admin pages in Tethys Portal. Figure 5-1 is an expanded Tethys Platform component diagram that indicates the specific components in Tethys Portal and the Tethys SDK that I created or contributed to as part of the Tethys Compute tools.

Figure 5-1. Expanded Tethys Platform component diagram showing the sub-components of Tethys Portal and the Tethys SDK.

5.1 Compute API

I designed the Compute API to enable provisioning cloud computing resources and getting access to those resources. It is a fairly simple API that offers direct access to the TethysCluster API to support dynamic scalability of computing resources. Accessing those

resources is facilitated by Schedulers, which are database objects that contain the connection information of an HTCondor agent in a computing cluster. The Compute API offers functions to access the Scheduler objects, which are needed to submit jobs to the clusters.

5.1.1 Using TethysCluster in Tethys

The Compute API provides the function *get_cluster_manager* from the TethysCluster API, which returns a ClusterManager object. The ability to provision clusters and scale them up or down can all be done through the cluster manager. The *get_cluster_manager* function accepts the path to a TethysCluster configuration file as an optional argument. This allows Tethys apps to create custom configuration files for TethysCluster that contain specifications for clusters designed to meet the need of the app, such as the number and size of nodes, and the image ID from which to make the nodes. The configuration file must contain credentials for the cloud provider that will be used to provision the clusters. These can either be provided through configuration of the app by the administrator of the Tethys Portal where the app is installed, or the app can collect credentials from users so that the each user is paying for the resources that they use. This is a choice of the app developer. Tethys does not currently have a user payment system, so the only way for a user to pay for the resources he or she uses is to provide the credentials to his or her cloud account. Otherwise the cloud usage of all users will be charged to the account provided by the Tethys Portal administrator.

In addition to being able to manage cloud-computing resources in Tethys apps through the Tethys SDK, resources can also be managed at a portal level through the admin pages. This is discussed more in section 5.3. Whether clusters are provisioned by apps or through the admin pages, accessing the clusters is facilitated through Schedulers.

5.1.2 Schedulers

A scheduler represents an agent in an HTCondor pool, (i.e. a computer that has the ability to submit jobs to the pool). I used the Django's Object Relational Model (ORM) to create database objects to represent schedulers. These scheduler objects store the IP address or hostname of the computer along with the credentials needed to connect to the computer through SSH. To be able to submit jobs to a computing cluster from a Tethys app a scheduler must be defined with the credentials for one of the nodes in the cluster. Creating schedulers can either be done in the app through the Compute API, or they can be created using the admin pages (see section 5.3).

5.2 Jobs API

CondorPy already provides a scriptable interface for creating and submitting jobs to a compute pool. However, running computing jobs in a web environment introduces additional complexities such as dealing with asynchronous interactions, load balancing, and requiring a web-based user interface. This section describes how CondorPy is used in Tethys to address these challenges through the Jobs.

5.2.1 Job Manager

To facilitate the asynchronous nature of computing in the synchronous environment of web apps, I wrap CondorPy job objects in a database model using the Django ORM. The details of interacting with the database are abstracted through a job manager object. The job manager is part of the Tethys SDK and gives the developer a simple interface for creating and retrieving jobs. The job manager automatically filters jobs based on the app and user so the developer can

easily retrieve pertinent jobs. The details of creating new jobs, including the underlying CondorPy job, are also handled by the job manager and facilitated with the use of Tethys job templates.

5.2.2 Tethys Job Templates

In Tethys, jobs are defined through job templates. These are distinct from CondorPy job templates, but serve a similar purpose. Tethys supports multiple job types, but the principle job type is the CondorJob type, which serves as a wrapper for CondorPy Job objects. A job template in Tethys specifies the job type and defines any job parameters that will be common to all jobs created from that template. When creating a job, the name of the job template is passed to the job manager, which then returns a job object that has been initialized with from the job template. Job templates help to organize jobs and simplify the process of creating new jobs.

The CondorJob type can take advantage of CondorPy's remote scheduling capability to offload the task of maintaining the job queue from the main web server. This is done by setting the scheduler property in the job template. If a scheduler is not defined for a job, Tethys assumes that the local machine is configured as a scheduler and will attempt to submit the job locally, which requires that HTCondor be installed on the Tethys server. If a job does have a scheduler defined, then Tethys uses the remote scheduling capability of CondorPy to copy the required files to the scheduler (if needed) and submit the job to the scheduler's HTCondor pool.

5.2.3 Jobs Table Gizmo

We developed Tethys Platform to simplify the process of creating a web application capable of running large computing jobs. One way we accomplished this is through Gizmos.

Gizmos are user interface elements that can be configured through the Tethys SDK. These elements have predefined HTML and JavaScript components that get included automatically. I created the jobs table gizmo to help users interact with their jobs (Figure 5-2).

Jobs Table

Id	Name	Description	Creation Time	Execute Time	Status	
31	job_1	Pending job	Sept. 3, 2015, 9:40 p.m.	None	Run	Delete
32	job_2	Submitted job	Sept. 3, 2015, 9:40 p.m.	None		Delete
33	job_3	Running job	Sept. 3, 2015, 9:40 p.m.	None		Delete
34	job_4	Running multi-process job with various statuses	Sept. 3, 2015, 9:40 p.m.	None		Delete
35	job_5	Job error	Sept. 3, 2015, 9:40 p.m.	None	Error	Delete
36	job_6	Aborted job	Sept. 3, 2015, 9:40 p.m.	None	Aborted	Delete
37	job_7	Completed job	Sept. 3, 2015, 9:40 p.m.	None	View Results	Delete
38	job_8	Completed multi-process job with some errors	Sept. 3, 2015, 9:40 p.m.	None	View Results	Delete

Figure 5-2. The Jobs Table gizmo.

While the jobs table gizmo is accessed through the gizmos API in the Tethys SDK, it is closely connected with the Jobs API because it provides the user with information about computing jobs. The fields of the table are configurable so the app developer can decide which attributes of the jobs to display. The default configuration of the gizmo provides controls that allow a user to run or delete jobs, and a link to view the job results when jobs are completed. It also handles updating the job statuses automatically at a configurable interval. The jobs table gizmo handles much of the background code for creating a web interface for interacting with jobs while still giving developers the flexibility to configure it according to their needs.

5.3 Tethys Compute Admin Pages

Tethys Portal includes admin pages that allow administrators to manage database entries that are part of the Tethys Platform database model. I added a set of Tethys Compute admin pages that allow Tethys Portal administrators to create and manage portal-wide computing clusters and schedulers and to manage jobs from all of the apps in one place (Figure 5-3).

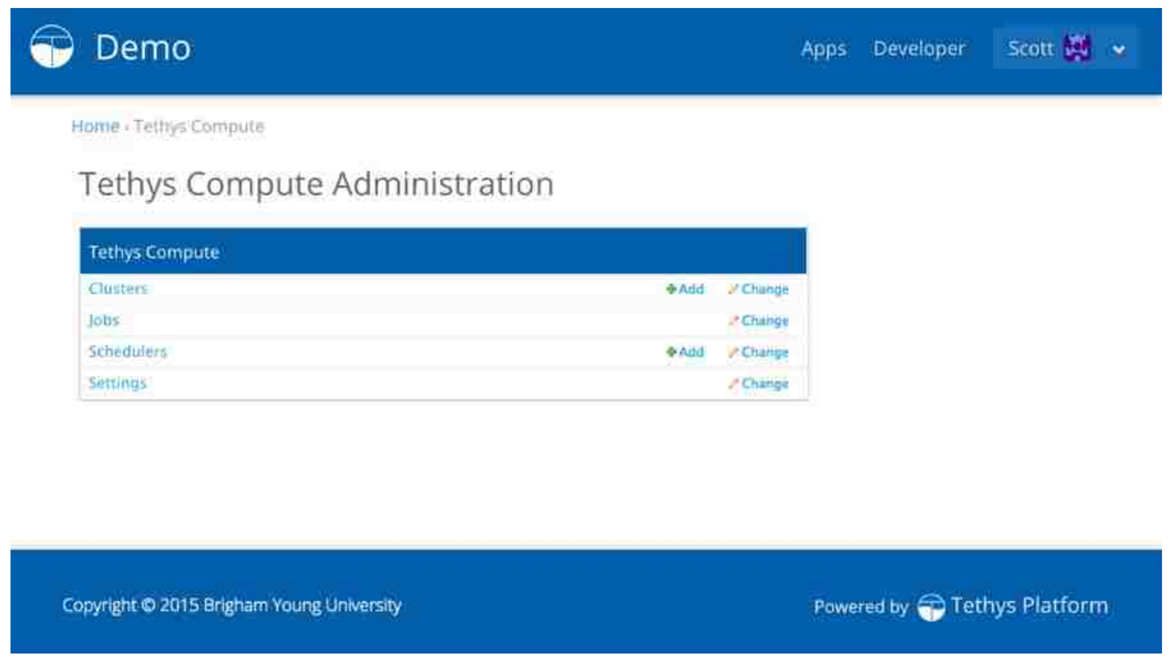


Figure 5-3. Tethys Portal admin page for Tethys Compute.

I created a simple interface for TethysCluster in the admin pages so that setting up a new cluster only requires a name (one that is unique among the clusters on the cloud provider account being used) and a size. Optionally, image IDs can be provided to create the cluster with customized images. I didn’t include all of the features and flexibility that are available through the TethysCluster API in the admin page because the purpose of the web interface was to provide a quick and easy way to create computing clusters. However, It is possible that additional features will be added to the cluster admin page in the future (Figure 5-4).

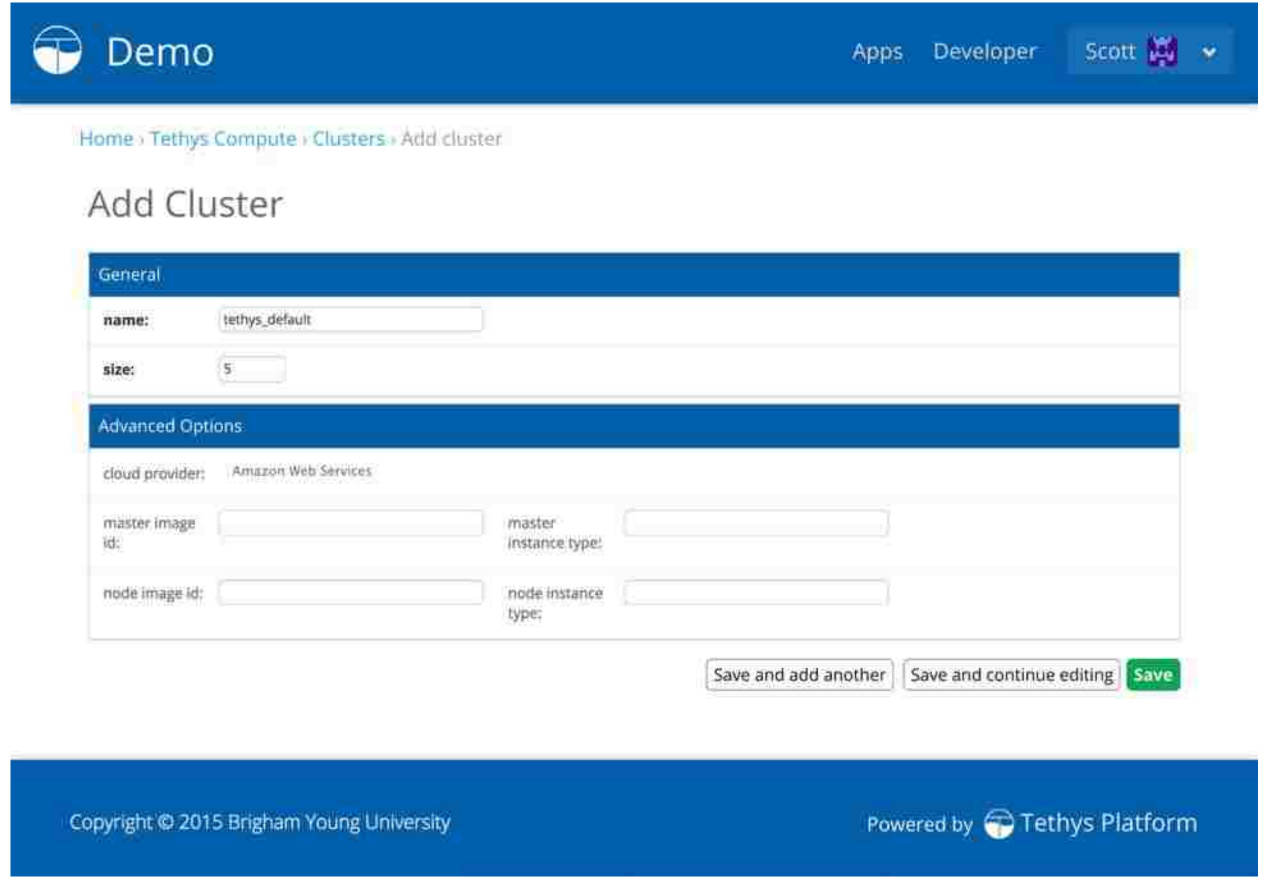


Figure 5-4. Tethys Portal admin page for creating/editing clusters.

Additional settings that aren't specific to a single cluster, such as cloud provider credentials and the default cluster template, are on the Tethys Compute Settings admin page (Figure 5-5).

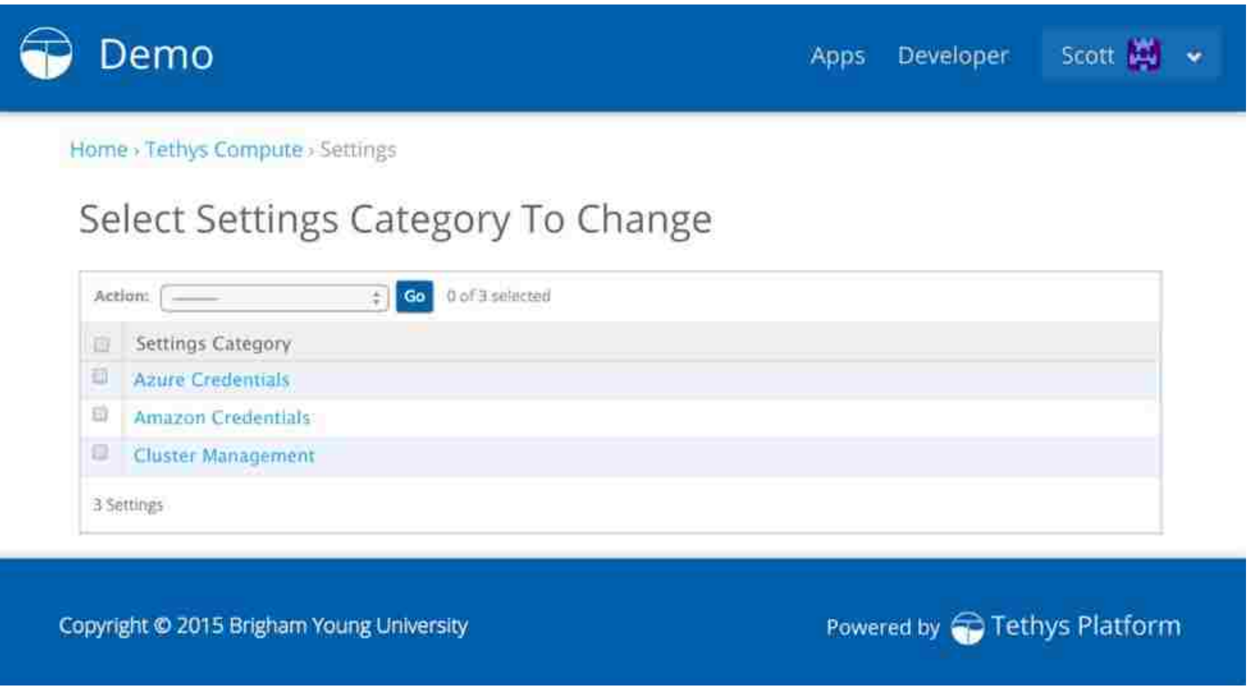


Figure 5-5. Tethys Portal admin page Tethys Compute Settings.

Adding portal-wide schedulers to allow jobs to be submitted to computing resources is done through the Tethys Compute Schedulers admin page (Figure 5-6). Only the *name* and the *host* properties are required. If a username is not provided then default, “root,” is used. Some method for authenticating the user on the scheduler is required whether that is a password, a private key, or a private key/passphrase combination. Schedulers do not necessarily have to be part of one of the clusters that is set up through the admin pages. They can be connected to any HTCondor pool that is accessible from the network where Tethys Portal is running, which can include on-premise pools.

Jobs are also managed through the Tethys Compute admin pages. The Tethys Compute Jobs page (Figure 5-7) provides tools for editing or deleting jobs, but not for creating them; jobs can only be created using the Jobs in the Tethys SDK. Care should be taken when editing certain

properties, like the workspace, the subclass (i.e. the job type), or the status, as jobs may become invalid if these properties are changed.

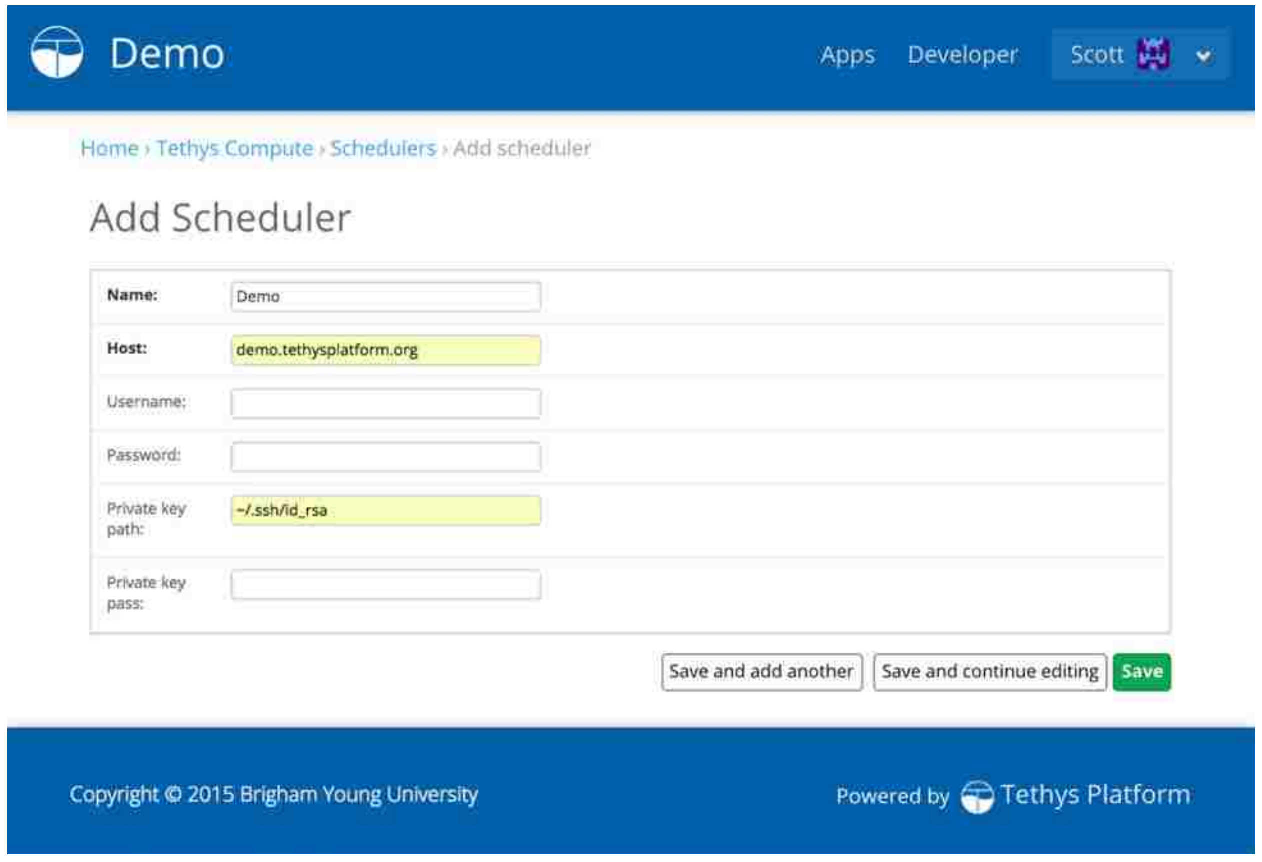


Figure 5-6. Tethys Portal admin page for creating/editing schedulers.

5.4 Discussion

I used the comprehensive computing toolkit provided by TethysCluster and CondorPy to build Tethys Compute, which provides the development tools for HTC computing in a web environment. These tools are integrated into the Tethys SDK through the Compute API and the Jobs. The Compute API is essentially a direct interface to the TethysCluster API, but in the Jobs API I created a set of tools around CondorPy to provide the necessary functionality for the web.

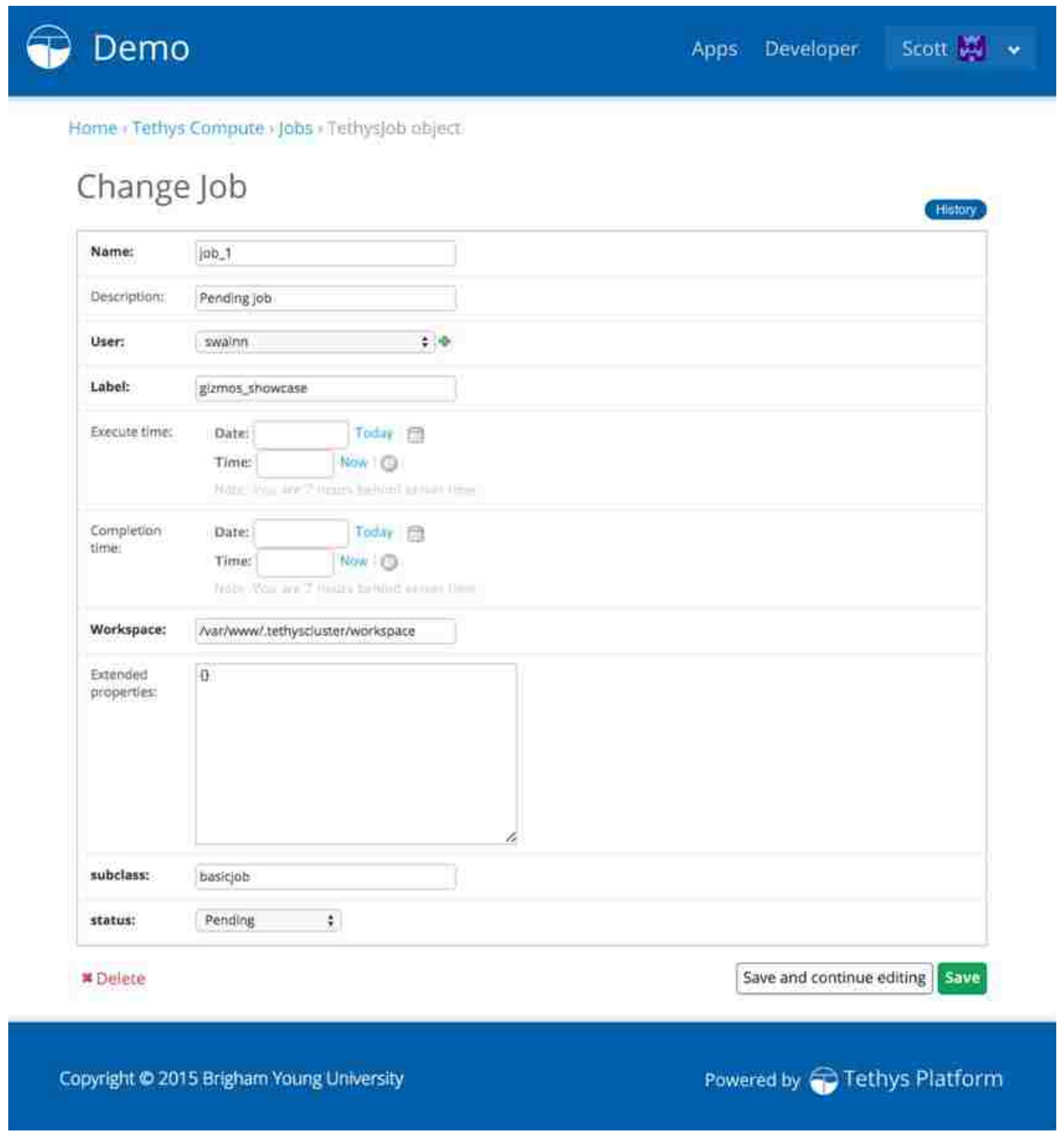


Figure 5-7. Tethys Portal admin page for editing jobs.

These additional tools around CondorPy were needed because the web environment adds complexities that require additional consideration. Some of these issues have already been addressed. However, the primary issue is that of persisting jobs beyond individual server-client transactions. To illustrate this problem, consider a web application that allows a user to execute a job by clicking a button on a web page. Behind the scenes, the browser sends a request to the server indicating that the button to run a job has been clicked. The server responds by executing

some code to process the request; in this case CondorPy is used to create a job and submit it to an HTCondor pool. The server can then wait for the job to finish running before sending a response back to the browser, but if the job takes longer than a minute or two, the connection with the browser will time out. Alternatively, if the server sends a response to the browser before the job is finished then the CondorPy job, which is in the server's memory, is deleted and the server then knows nothing about the job and cannot check the job status or retrieve the job results. To get around this problem the server must save the information about the CondorPy job before it responds to the browser. In Tethys Platform the information from the CondorPy job is stored in a database. This way when another request is made to the server (for example, to check the status of the job), the server can reconstruct the CondorPy job from the information in the database and then use the CondorPy API to retrieve the job status.

6 APPLICATIONS

I used TethysCluster, CondorPy and Tethys Compute to support other research projects. These projects illustrate the need and utility of a comprehensive computing toolkit. Four projects are described and represent research that I did in collaboration with others. I give enough background on each project to provide the context, but I mainly focus on the computing aspects of the projects that I performed.

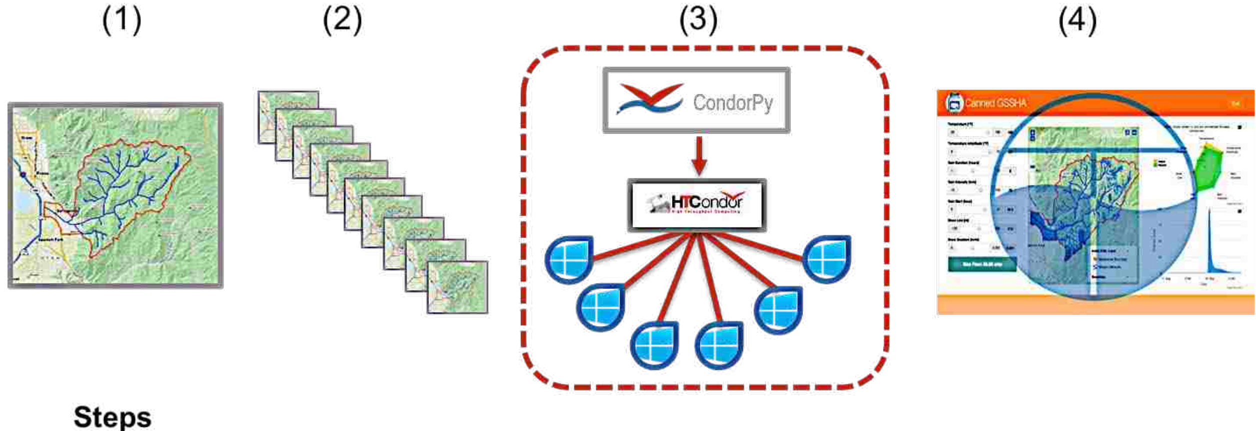
6.1 Canned GSSHA: Pre-computed Modeling for Flood Warning

Parameter sweeps are a common type of modeling task that help characterize the solution space that corresponds to a parameter space. The parameter space is defined by the permutations of a set of parameters that are varied over the possible range of values. Each permutation defines a different parameterization for the model. A parameter sweep involves running the model with each of the parameterizations to determine the distribution of the resulting solutions. One application of this type of modeling in water resources is to pre-compute a large set of parameterizations that represent a set of hydrologic conditions and archive the results so that they can be quickly retrieved to determine if and to what extent forecasted conditions will produce flooding. This requires that many (often tens of thousands) model scenarios (or parameterizations) be computed to adequately cover the parameter space. This method and the work represented here is discussed in grater detail in (Dolder, Jones et al. 2015).

6.1.1 Methods

To demonstrate this application a GSSHA model of the Hobble Creek watershed was developed and a set of seven parameters were selected to vary: temperature, temperature amplitude, precipitation duration, precipitation intensity, rain start time, snow line, and snow gradient. The Latin hypercube method was used to divide each parameter distribution into quantiles. A different number of quantiles was used for different parameters to achieve greater resolution for those that are more important (i.e. have a grater influence on the result). Ultimately this resulted in 57,600 unique scenarios of randomly generated parameterizations that evenly cover the full parameter space.

The GSSHA model I used takes between about a minute to almost a day to run, depending on the parameterization. I used an early version of CondorPy to facilitate creating and submitting jobs to compute each of the 57,600 model instances to an HTCondor pool made up of 55 lab computers that each had 4 cores. At first all of the jobs were submitted into the queue simultaneously; however, this crashed the scheduling daemon on the submitting computer. To prevent the scheduler from becoming overloaded the remaining jobs were placed in a DAG (without any relationships), which enabled the number of jobs that were queued at a single time to be limited. Since the pool consisted of 220 cores the limit on the number of jobs queued was set at around 250, ensuring that there would be no delay in a computing resource from getting a new job from the queue when it became available. A Tethys app was created to visualize the results. The steps involved in this study are summarized in Figure 6-1, and the step that I performed is indicated with the red, dashed box.



Steps

1) Build GSSHA model
2) Generate 57,600 scenarios
3) Create and run HTCondor job for each scenario using CondorPy
4) Create Tethys app to visualize results

Figure 6-1. Summary of the steps in the Canned GSSHA study.

6.1.2 Results

The results of the 57,600 model runs were archived and a Tethys app was designed to be able to explore the parameter space. There are sliders to select the value for each of the parameters. The resulting parameter set is represented by the yellow area on the radar plot. The green area on the plot shows the closest match from the 57,600 parameterizations that were computed. The pre-computed results of that simulation, including the hydrograph and a max flood map, are also shown (Figure 6-2).

Due to the scheduler crashing, the scenarios were not all computed continuously. The computing took place in several stretches over a 46-day period. Figure 6-3 shows a timeline of the entire period with the periods of active computing marked. Table 6-1 summarizes the time statistics for the entire computing task.

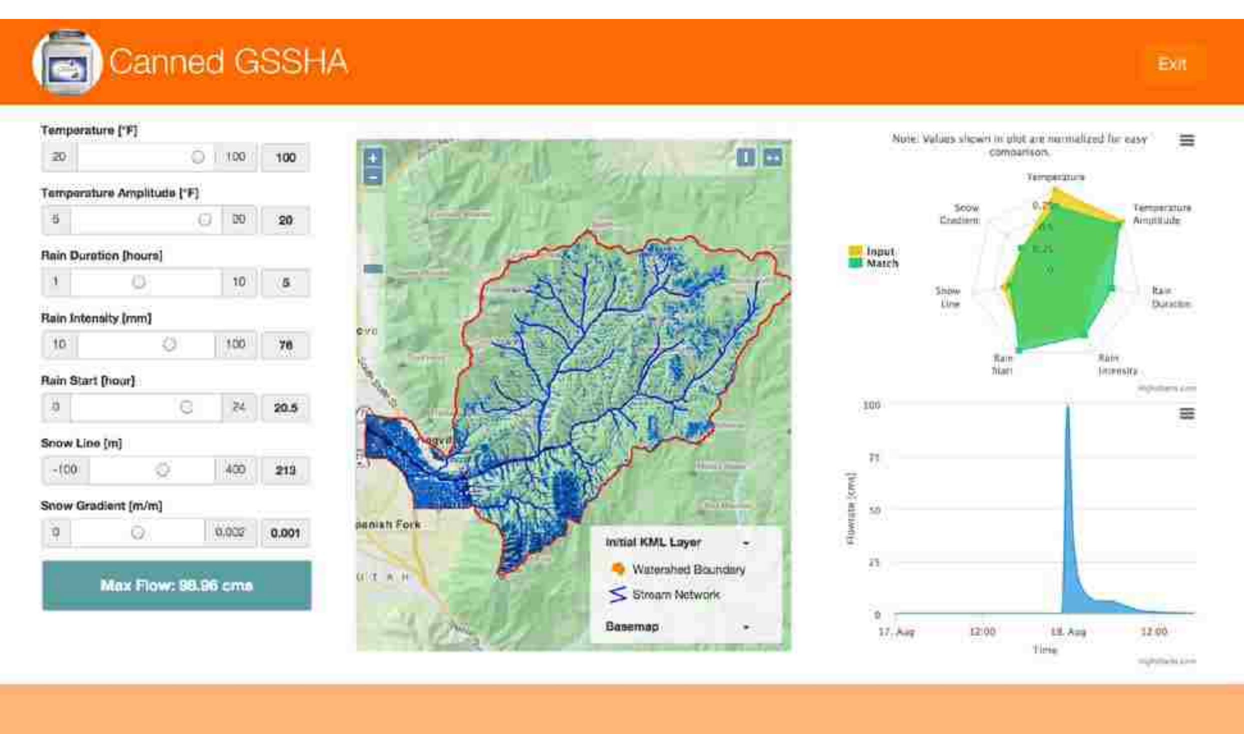


Figure 6-2. The Canned GSSHA Tethys app allows users to explore the solution space of the 57,600 archived simulations.

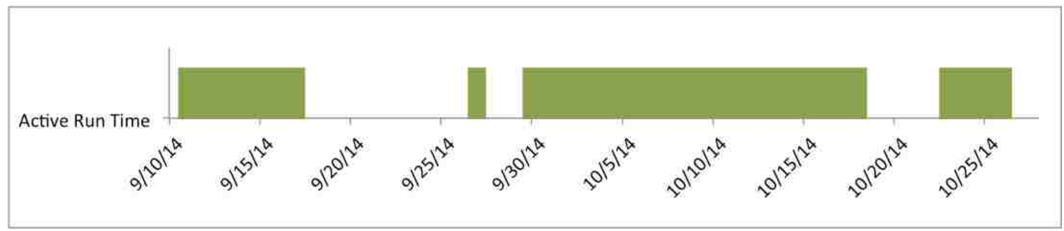


Figure 6-3. Timeline showing the active computing time for running 57,600 GSSHA models using HTCondor.

Table 6-1. Summary of Compute Time for the Canned GSSHA Parameter Sweep

Statistic	Value	Units
Total Time	45.39	days
Active Time	29.39	days
Active Time	64.8	%

Unfortunately, due to the complications in finding the most stable way to run the jobs, the statistics for individual model runs were not saved for all 57,600 jobs. However, the order in

which the parameterizations were run was randomized so the statistics for the 37,244 jobs that were saved should be a representative sample. These statistics are summarized in Table 6-2.

Table 6-2. Summary of Statistics for Individual Model Runs and the HTCondor Pool

Statistic	Value	Units
Jobs	37,244	count
Wall Time	2,752.48	days
Good Time	2,090.73	days
Goodput	76.0	%
Max Job Time	23.73	hours
Min Job Time	< 1	min
Avg. Job Time	1.35	hours
CPUs	220	count
Avg. CPU Time	12.51	days
Pool Utilization	42.6	%

The statistics listed are specific to HTCondor and merit some explanation. *Wall time* is the total amount of time that the 37,244 jobs were being processed on the HTCondor pool. *Good time* is the total amount of time that actually contributed to completed jobs. If a job is evicted from a computing resource during the middle of execution (because the computing resource becomes unavailable) then the job must be restarted on a new resource. Any time that was spent computing the job on the original resources doesn't contributed to the completion of the job, which is why the good time is lower than the wall time. The *goodput* is the percent of the total time that contributes to completed jobs, or the ratio of the good time to the wall time. The *min, max,* and *average job times* are calculated from the good time for the individual jobs. This includes in any overhead, such as time transferring files, in addition to the actual simulation time. The *average CPU time* is the wall time divided by the number of CPUs, and represents the average amount of time each CPU in the pool was actively being used for this task. The *pool*

utilization is the ratio of the average CPU time to the total active time, and represents the how much of the total capacity of the pool was used for this task.

6.1.3 Discussion

This study was done early on in the development of CondorPy and provided valuable insight in how to best run jobs of large magnitude, which directed the development of CondorPy. When the queue on the HTCondor scheduling node is larger than about 5,000 then it becomes unstable. By using a DAG to schedule a large number of jobs then the DAGMan can control how many jobs are submitted to the queue at one time. This keeps the scheduler stable, but also has the added benefit of allowing for recovery and resubmission of partially completed DAGs (using a rescue DAG) if for some reason the DAG is unable to complete all of the jobs. As a result of this study I added the Workflow capabilities to CondorPy to facilitate creating and submitting DAGs.

The goodput indicates that only 76% of the compute time on the HTCondor pool actually contributed to completing jobs. This means that 24% of the compute time was spent on jobs that were evicted before they were completed. This wasted time comes because HTCondor cannot anticipate when a computing resource will become unavailable, or how long a particular job will take to complete. HTCondor has mechanisms that help increase the goodput such as checkpointing (only on Linux nodes), or by having a job that is interrupted mid-execution wait on that resource until the resources becomes available again and then the job will resume where it left off. Deciding whether to have a job wait on a resource or be rescheduled elsewhere is a function of how expensive it is to restart a job. If there is a significant amount of file transfers that have to happen, or if there is the potential for a job to be interrupted after a large amount of

wall time then it may be more advantageous to suspend the job on the current resources and have it wait until the resource becomes available again.

The pool utilization is an indication of how available the pool was for running this task. Since 42.6% of the pool time was used to compute this task it means that 57.4% of the time it was claimed by users of the computers or by other HTCondor jobs. Because of HTCondor almost half of the processing time of 220 cores was made available for HTC tasks without needed to invest in any additional computing resources. However, if tasks are time critical or don't lend themselves well to interruption, then dedicated resources, such as cloud resources, should be used.

6.2 Data Access/Formatting Apps

Obtaining and formatting input data required by a model is part of any modeling process. Because data are now available from many different sources and in many different formats this part of the process can be quite complex and time consuming. As the spatial and temporal resolution of data increases, the computing requirements to process the data can be come large.

The NASA Earth Observing System Data and Information System (EOSDIS) provides many datasets that are useful for water resources modeling. Since the datasets provided are national or global scale and often have a large temporal domain, NASA has an online tool for selecting subsets of the data both spatially and temporally, the Simple Subset Wizard (SSW) (Figure 6-4).

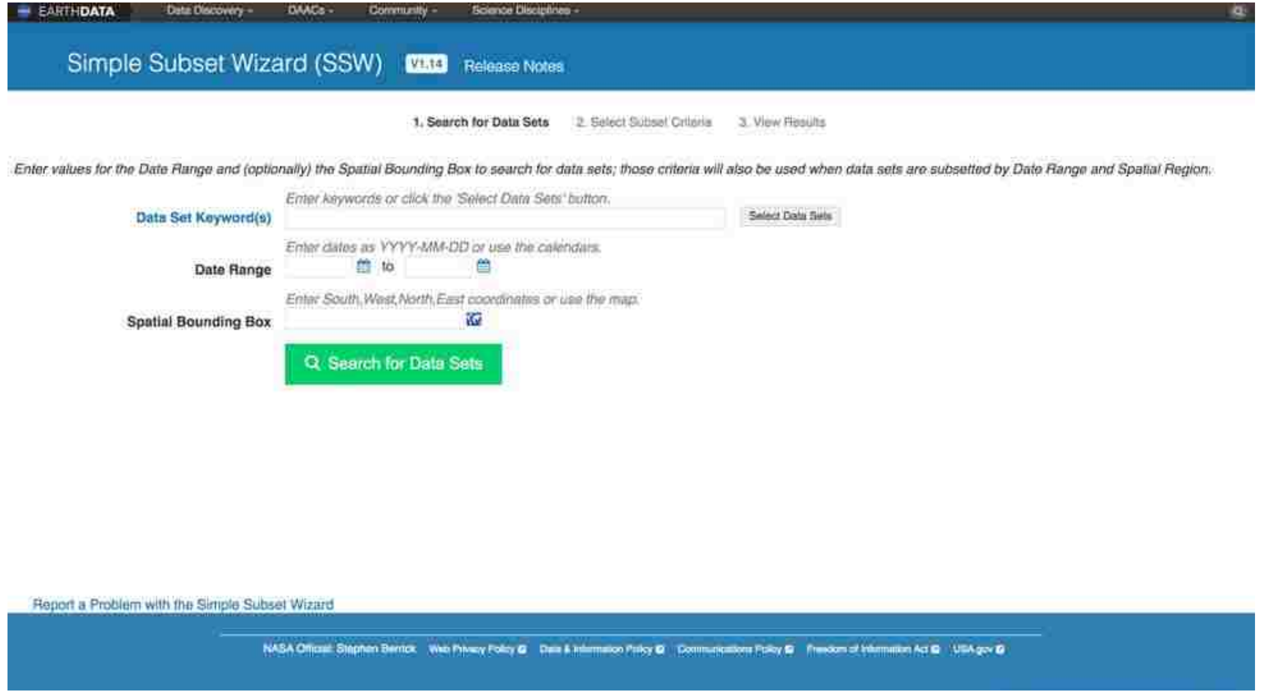


Figure 6-4. The Simple Subset Wizard web tool allows users to obtain subsets of large data sets.

The SSW is a convenient way to get just the data needed without having to manage large datasets; however, the format the data are provided in is not as convenient. The SSW provides a link to download each time step of the requested data individually (Figure 6-5). If data spanning a long period of time are requested, then downloading and managing the data can become unwieldy.

Precipitation and soil moisture data were used from the North American Land Data Assimilation System (NLDAS) data sets to populate GSSHA models for a research project looking at watersheds all over the United States. The data were accessed through the SSW, which provided them in netCDF format, but for the purposes of this project the data were ultimately needed in ASCII grid format so they could be loaded into Watershed Modeling System (WMS) to be used in as GSSHA input. Since the process of obtaining the datasets and reformatting them needed to be done for many watersheds and by many people, I automated the

process with Python scripts and then integrated them into two Tethys apps, which can be accessed by multiple users simultaneously.

Figure 6-5. Links for downloading data from the SSW.

6.2.1 Methods

The first of the two Tethys apps, the SSW Downloader, is designed to download and aggregate data from the SSW. A URL that returns a list of the URLs for the data time steps is copied into the app and then a job is launched to download all of the data time steps and aggregate them into a single netCDF file. The Jobs was used to create a CondorJob that submits a Python script to an HTCondor Pool. The Python script downloads and aggregates the data and then the resulting netCDF file is made available to download or to pass to the second app.

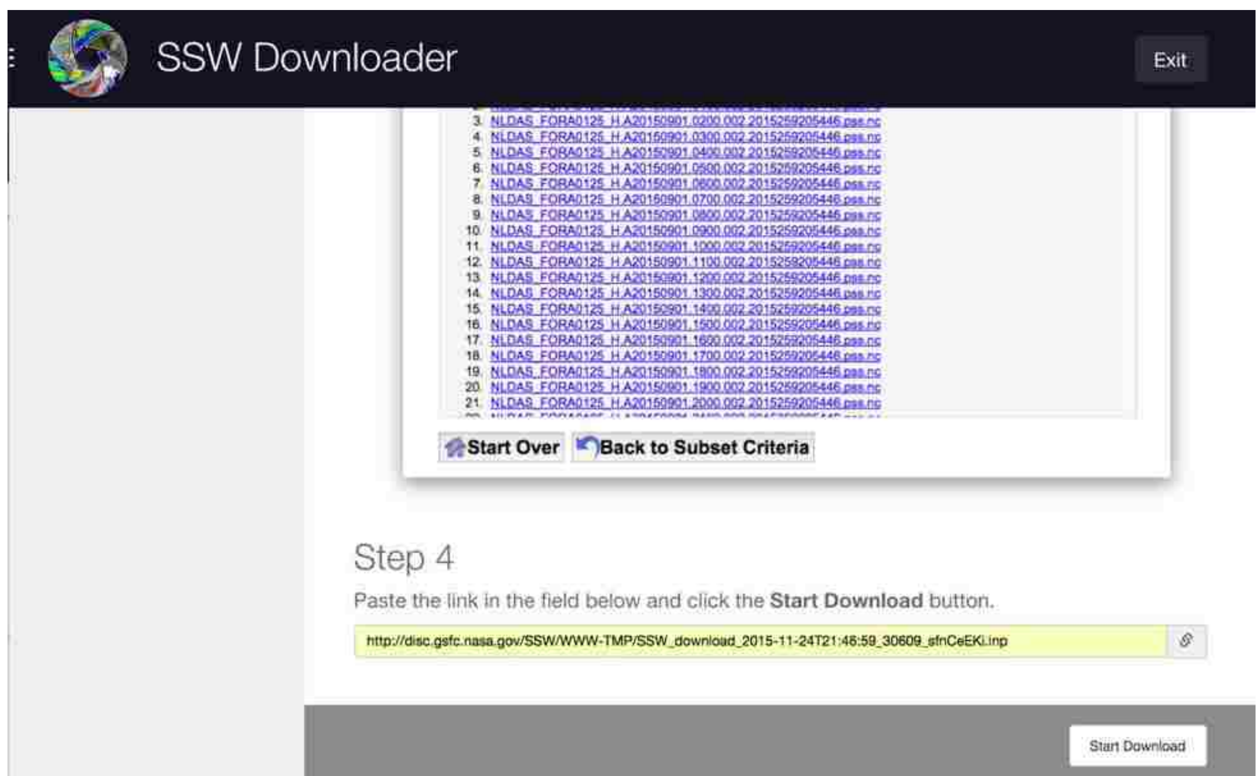


Figure 6-6. SSW Downloader Tethys app.

The second Tethys app, Convert NetCDF to GSSHA Input, takes a netCDF file passed from the SSW Downloader app and allows the user to select one of the variables to export as either a GRASS ASCII raster or an Arc/Info ASCII gird (Figure 6-7). The same method was used for this app as the previous app to submit a Python script as an HTCondor job using the Jobs. Since the ASCII grid format doesn't support multiple time steps a new grid is generated for each time step and then the whole set is compressed into a zip archive, which is made available to download.

To test the performance of the apps three datasets were downloaded that had the same spatial domain (bounding the state of Texas; see Figure 6-9), but different temporal domains ranging from two days to a month in April 2015. For each dataset both hourly precipitation and

soil moisture were included. That datasets were retrieved with the SSW Downloader app and then the precipitation was reformatted to ASCII grids using the conversion app.

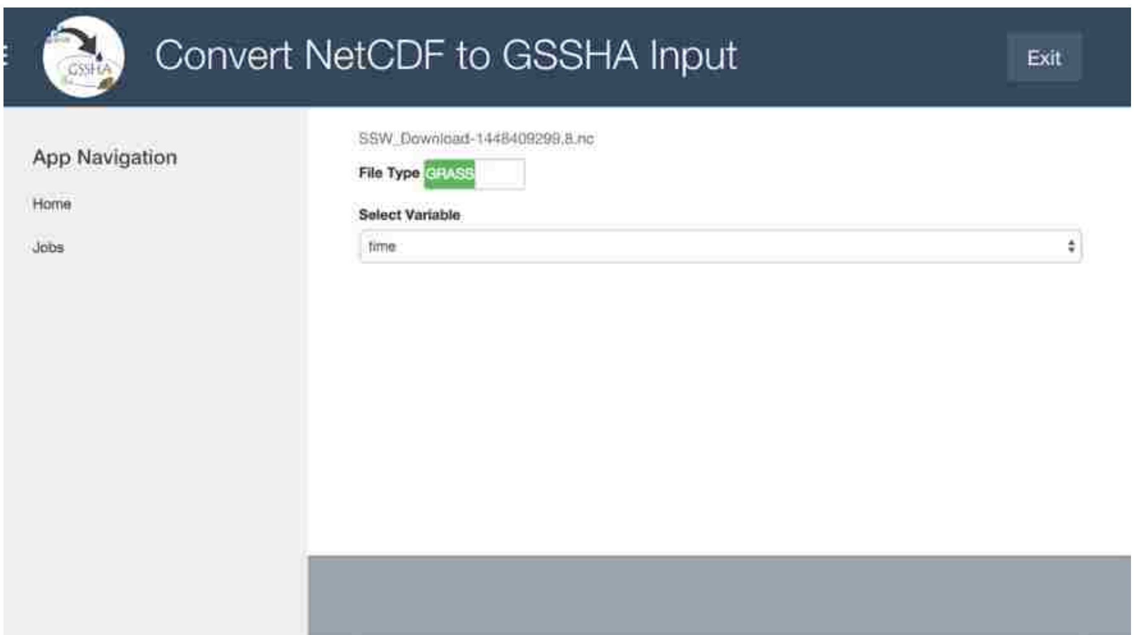


Figure 6-7. Convert NetCDF to GSSHA Input Tethys app.

To compare the effect of the number of variables and the size of the domain two additional sets of three datasets were processed using the same time extents as the first set. One set had a smaller domain surrounding Austin, TX (Figure 6-10), and the other used the original spatial domain, but only downloaded the precipitation data. The processing time of each of the nine datasets were compared.

The steps for developing these apps and using them to format data for GSSHA input is summarized in Figure 6-8 with the steps I performed boxed in the red, dashed line and the steps I helped with boxed in the blue dashed line.

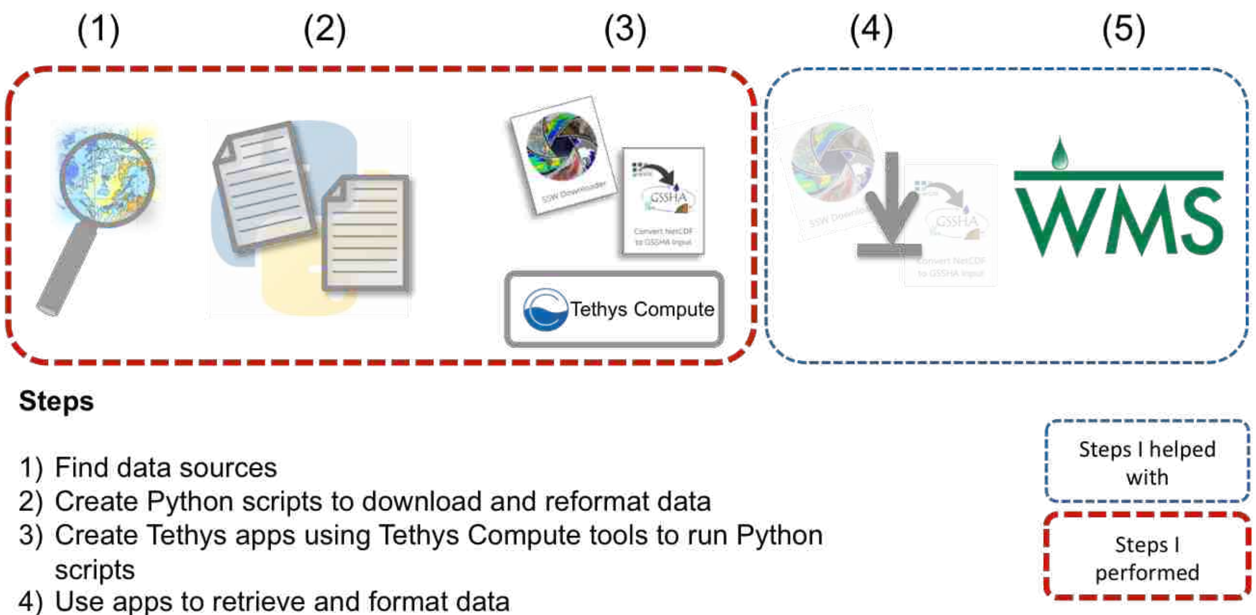


Figure 6-8. Summary of steps in developing the data access and formatting apps for use in GSSHA models.

6.2.2 Results

The data sets resulting from the processing of the first app were netCDF files that contained both the soil moisture and the precipitation data. Figure 6-9 and Figure 6-10 show the soil moisture for April 1, 2015, on the large and small domains respectively, and Figure 6-11 shows the hourly precipitation near Austin, TX for the whole month of April 2015.

The times it took to download and aggregate each of the data sets are listed in Table 6-3 and plotted in Figure 6-12.

The processing times recorded on the Convert NetCDF to GSSHA Input app ranged from 5 seconds to 25 seconds; however, the mechanism for timing only updated every 5 seconds, so it is likely that these times are not accurate.

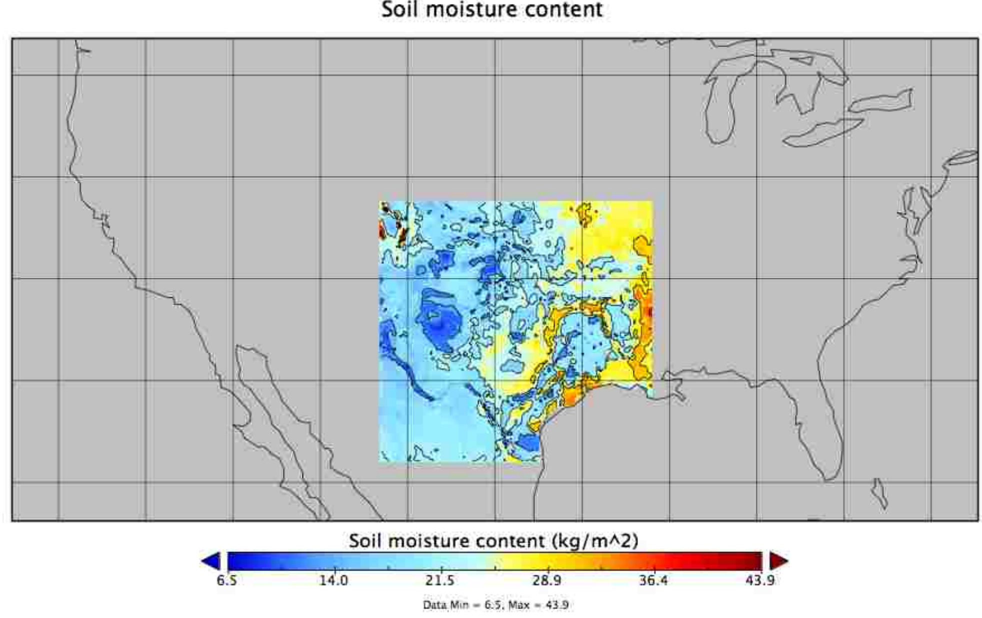


Figure 6-9. Soil moisture data from the NLDAS data set over the larger domain.

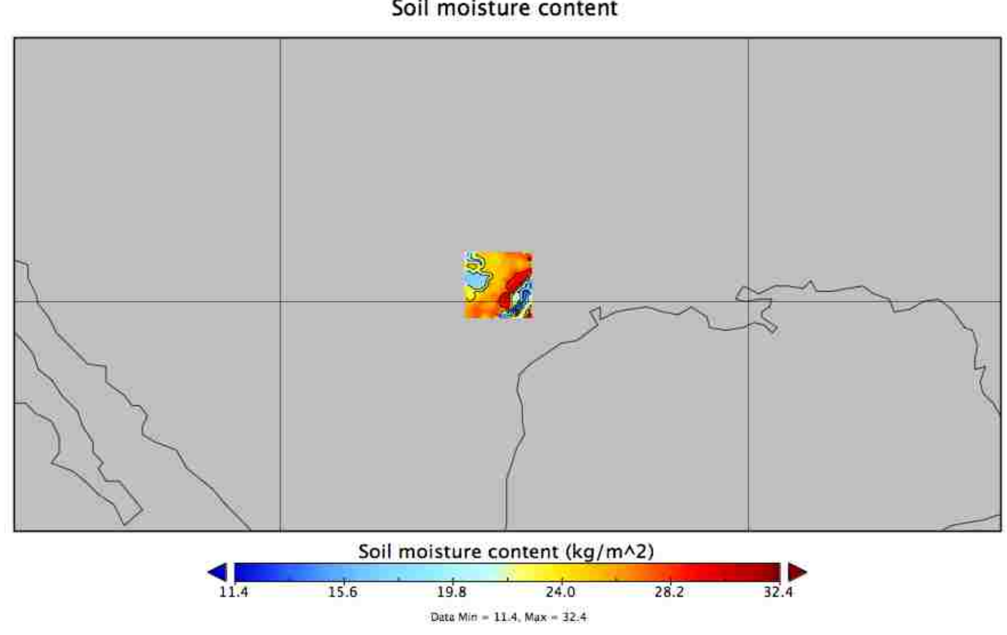


Figure 6-10. Soil moisture data from the NLDAS data set over the smaller domain.

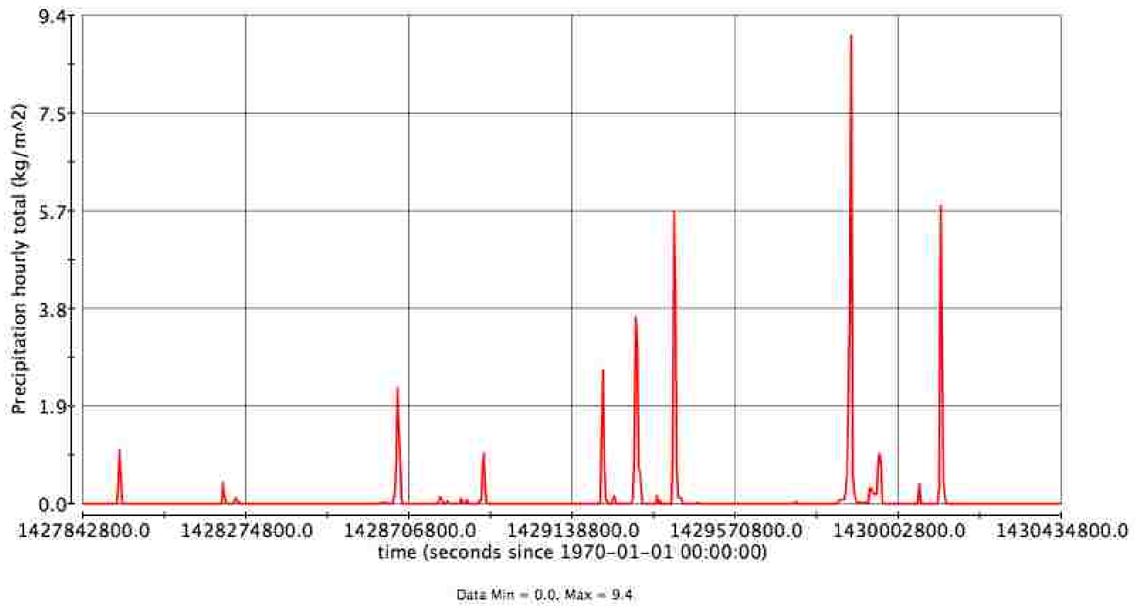


Figure 6-11. Hourly precipitation for April 2015 near Austin, Texas.

Table 6-3. Summary of Processing Time on the SSW Downloader Tethys App for Three Datasets

	Time to Process [sec]		
Days	**2 Vars. Large Domain**	**2 Vars. Small Domain**	**1 Var. Large Domain**
2	274	157	78
7	732	596	665
30	2661	2276	1234

6.2.3 Discussion

From Figure 6-12 it appears that the size of the domain has a slight impact on the compute time, but the number of variables has a significant impact. There was a separate file to download for each variable for each time step, so the downloading time is likely the largest factor in the total time.

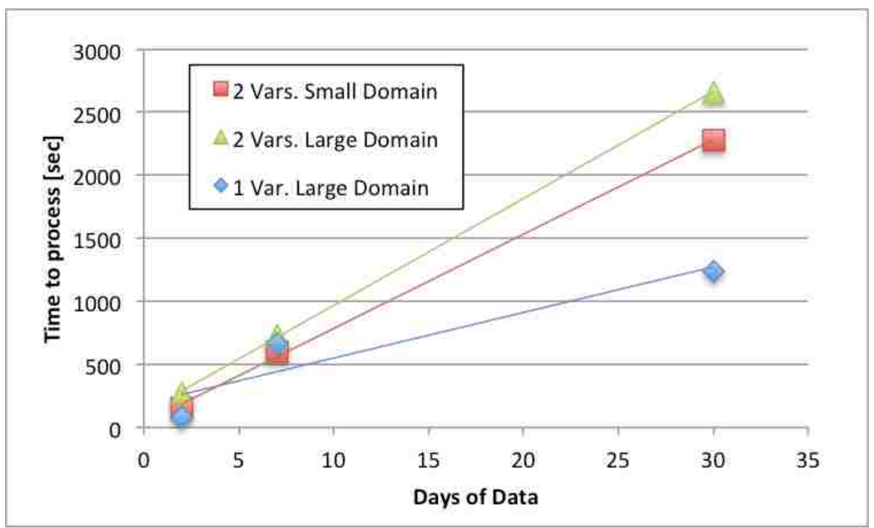


Figure 6-12. Plot of the processing times for various datasets with 1 or 2 variables on a large or small domain.

If the end goal were only to reformat the data, then it would have been more efficient not to have aggregated the netCDF file for each time step into a single dataset since the ASCII grid format can only store a single time step in each file anyway. However, aggregating the datasets made it easier to download and visualize the data before converting it to the ASCII format.

While these examples are not particularly compute intensive, the main benefit of automating the workflow in a web app is that multiple users can access it from anywhere at the same time. If multiple requests are made simultaneously then the computational load could become significant. Using the computing tools allows the computations to be offloaded to a distributed computing system that can queue and processes the jobs as necessary.

6.3 GSSHA Index Editor

Predicting the change in a runoff hydrograph due to changes in land-use is a common modeling task that is important to consider when determining the impact of a new development in a city, or events, such as a forest fire, that significantly change the hydrologic characteristics

of a large area. Since the specific location of the land-use changes can play a large role in the how the hydrology is affected, it is useful to use a spatially-distributed model, like GSSHA (Downer and Ogden 2004), that can represent the spatial characteristics of the land-use change (Ogden, Raj Pradhan et al. 2011). However, distributed hydrologic models are generally more complex than their lumped-parameter counterparts and take more time and expertise both to modify and to run (Singh and Woolhiser 2002). Anderson (2015) addressed these challenges by developing a Tethys app, which provides a simple and straightforward workflow where the user selects an existing model, makes edits to the land-use maps, and submits the modified model to be run (the original is also run if the results have not already been saved). The interface of the app is shown in Figure 6-13.

One of the challenges with running GSSHA in a web app is that it is compiled for a Windows environment, but it is common to use Linux servers to host websites. An additional challenge is that, due to the nature of the web, it is possible for many people to simultaneously access the app and submit jobs, causing a large computing load. These challenges are addressed with the Tethys computing tools.

6.3.1 Methods

The GSSHA Index Map Editor app interface was designed to allow users to manually draw polygons to change the land-use, or to upload shapefiles with new land-use polygons. A Python library called GsshaPy was used to load the GSSHA files into a PostGIS database. GIS functions on the database were used to update the land-use index map.

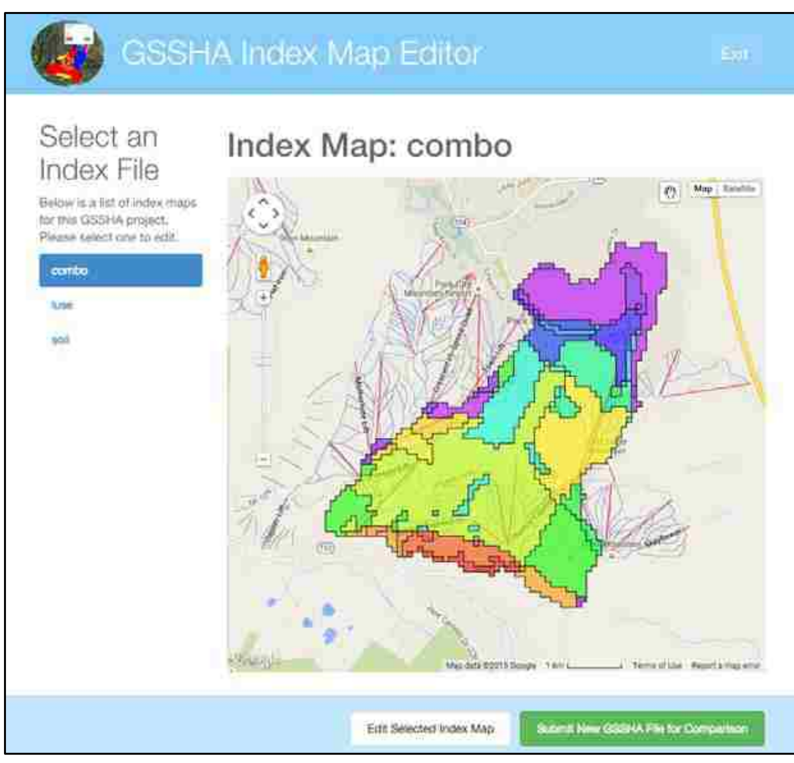
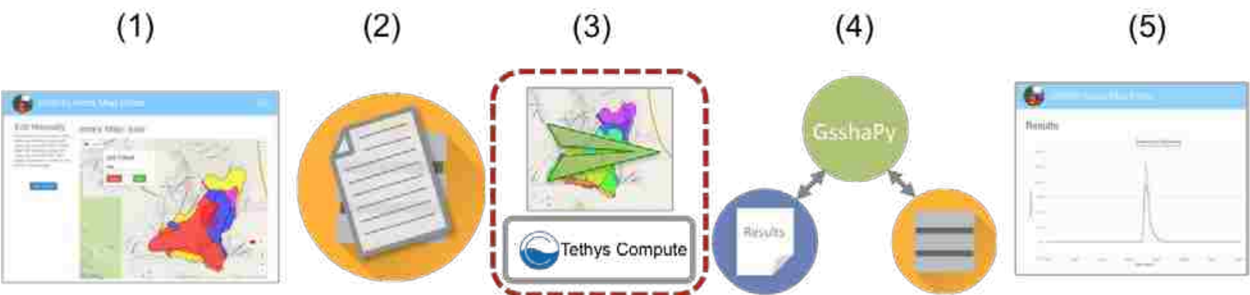


Figure 6-13. Map interface of the GSSHA Index Map Editor Tethys app.

I added the capability for the app to use the Tethys Platform job manager to connect to computing resources that can be scaled to handle the workload and configured with the proper environment for the compute jobs. I used the job manager to create the GSSHA modeling jobs, which are then submitted to an HTCondor pool of Windows nodes that are configured to run GSSHA. CondorPy copies the GSSHA files that were modified through the app to the remote scheduler and then remotely schedules the job to run. Once the job is complete, CondorPy then copies the results back to the local Tethys server so they can be visualized. This file transfer configuration limits the size of the GSSHA models that the app can process to those that can reasonably be copied back and forth over the network.

The steps for creating the GSSHA Index Map Editor app are summarized in Figure 6-14 with the step that I performed boxed in the red, dashed line.



Steps

1) Create web interface for editing index tables
2) Write controllers for submitting edits to GsshaPy database
3) Use Tethys Compute tools to run GSSHA model
4) Use GsshaPy to load results into the database
5) Create web interface for viewing results

Figure 6-14. Summary of steps in creating the GSSHA Index Map Editor app.

6.3.2 Results

The result of interest for the GSSHA Index Map Editor is the difference in the hydrographs produced by the original and the modified models. Once the jobs have finished computing and the results of the models are copied back to the Tethys server, the app can graph both of the hydrographs for comparison. Figure 6-15 shows an example of the results page of the app.

6.3.3 Discussion

The GSSHA models used here ran in approximately three minutes. As in the previous example the computation load is not particularly significant, but the main benefit of deploying this workflow as a web application is to allow ubiquitous and simultaneous access to a simplified modeling workflow where the computational environment only needs to be set up and maintained in one location. Because of the possibility of multiple users accessing the app and submitting jobs at the same time, the Tethys Compute tools are needed to offload the computation to an HTC system.

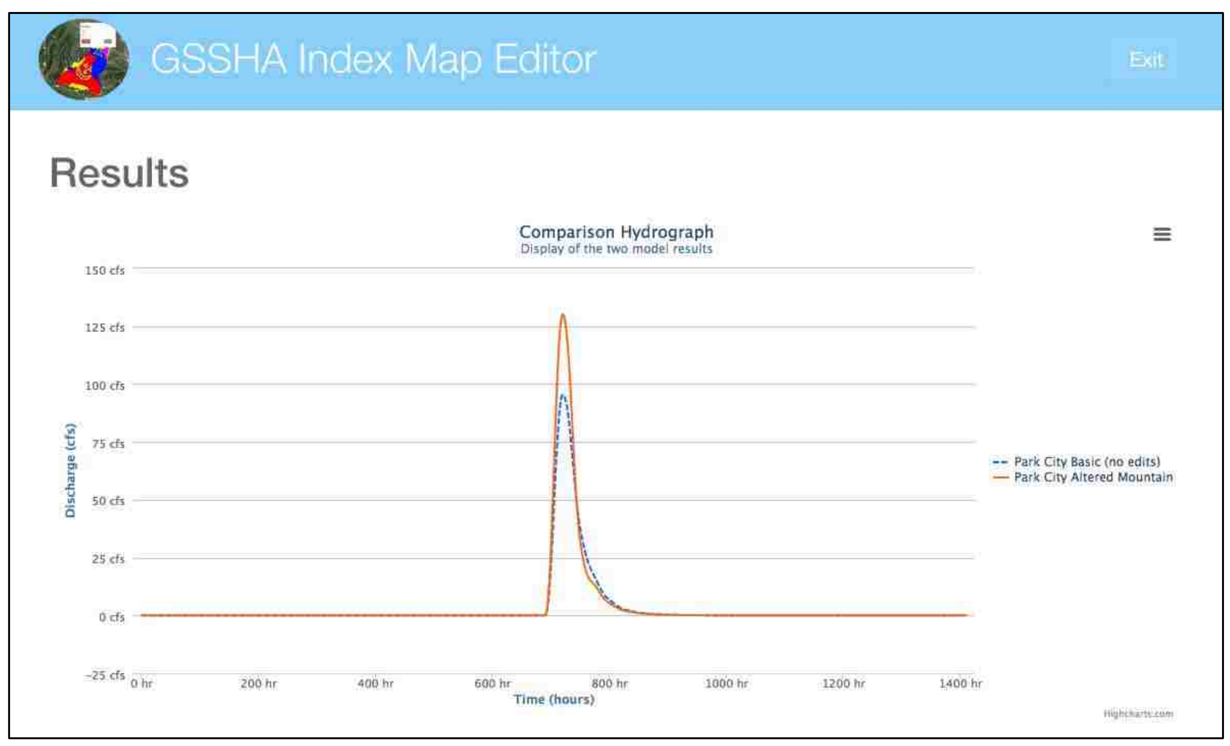


Figure 6-15. Example results of GSSHA Index Map Editor app workflow.

In order for CondorPy to submit jobs to a remote Windows scheduler it is necessary to configure it with an SSH server. If TethysCluster is used to provision the cluster on AWS then the nodes will already have SSH configured.

6.4 Streamflow Prediction Tool

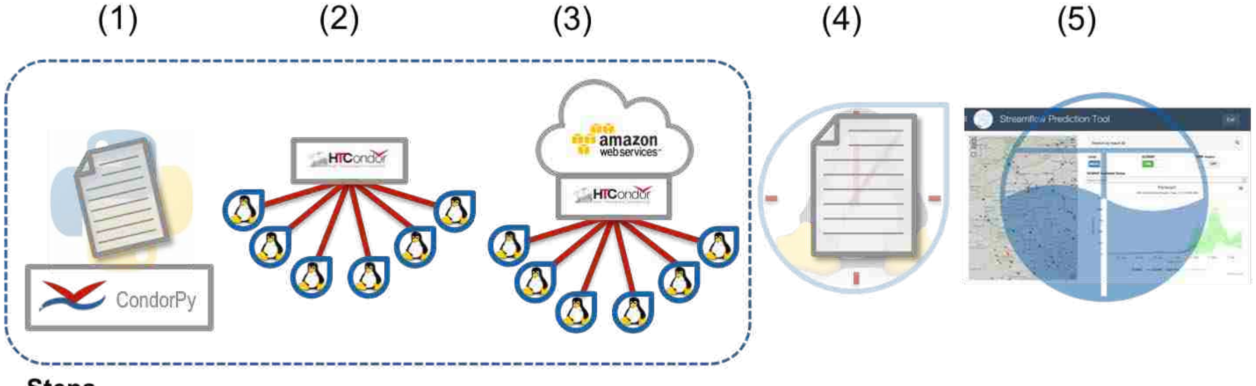
As part of an effort led by the National Water Center to create high-resolution in-stream forecasts for rivers throughout the United States, Snow, Christensen et al. (In Press) developed a method for downscaling and routing global runoff ensemble forecasts generated by the European Centre for Medium-Range Weather Forecasts (ECMWF) using the Routing Application for Parallel computatIon of Discharge (RAPID) model (David, Maidment et al. 2011). This method processes a 52-member ensemble weather forecast by downscaling and routing precipitation runoff through the nearly 2.7 million reaches of the National Hydrography Plus Version 2

Dataset (NHDPlusV2), (Horizons Systems Corporation 2011). While the processing is computationally expensive, the ensemble members are independent of each other and thus can be processed in parallel in a distributed computing system making it an ideal application of HTC. Furthermore, a new forecast is released every 12 hours, requiring a scheduled and automated system to process the computations.

6.4.1 Methods

The NHDPlusV2, which covers the continental United States, is divided into 12 hydrologically independent regions. Each of the 52 ensemble members of the runoff forecast was processed for each of the 12 regions, thus the computational load was subdivided both by region and by ensemble member, creating 624 subtasks that needed to be computed every 12 hours. The process of downscaling and routing the forecasted runoff consists of running a geoprocessing tool to map the runoff from the coarse grid to catchments defined by the NHDPlusV2 and then running the RAPID model to route the runoff through a stream network using Muskingum routing. We configured each of the compute nodes, or HTCondor resource nodes, with the geoprocessing script, the RAPID model code, and a shared file system that contained the static input files for each region. We created a Python script to download ECMWF ensemble forecast to the shared file system, thus eliminating the need to transfer a large amount of data between the compute nodes and the scheduling node. The Python scripted used CondorPy to create and submit jobs to process the 52-member ensemble forecast for each of the 12 regions. The script was scheduled to run every 12 hours on the scheduler node.

The steps for setting up the streamflow prediction tool are summarized in Figure 6-16 with the steps that I helped with boxed in the blue, dashed line.



Steps

1) Create Python scripts to download and process ECMWF ensemble forecasts
2) Set up HTCondor pool on local resources
3) Set up HTCondor pool in the cloud
4) Schedule Python scripts to run on compute pools
5) Create Tethys App to Visualize Results

Steps I helped with

Figure 6-16. Summary of steps for setting up the Streamflow Prediction Tool.

6.4.2 Results

This process was tested using a local compute cluster and using a computing cluster on AWS. The time to compute the same jobs serially was estimated based on the individual computation times to determine the savings of using distributed computing vs. serial computing. The results from each region are listed in Table 6-4.

The modeling workflow for each region results in 52 point netCDF files that contain the 15 day in-stream flow prediction for each reach in that region. The netCDF files are uploaded to a data repository where they are accessible to a Tethys app that was created to visualize the results (Figure 6-17).

Table 6-4. Results of Computation Time Based on Area and the Number of Reaches

			Compute Time [sec]		
Watershed Name	Area [sq km]	Number of Reaches	Serial	AWS	Local Cloud
NFIE Souris-Red-Rainey Region	213,488	29,053	7,343	49	141
NFIE Rio Grande Region	564,840	55,854	9,083	64	175
NFIE New England Region	169,445	65,858	10,906	63	210
NFIE Texas-Gulf Region	464,493	66,373	10,417	68	200
NFIE Great Basin Region	367,058	96,269	8,589	43	165
NFIEGreat Lakes Region	324,434	104,645	15,873	115	305
NFIE Mid-Atlantic Region	277,755	125,398	16,900	109	325
NFIE California Region	421,995	140,759	22,367	164	430
NFIE Colorado Region	660,454	187,010	28,105	210	540
NFIE Pacific Northwest Region	814,493	231,806	24,325	180	468
NFIE South Atlantic-Gulf Region	675,734	323,096	41,083	319	790
NFIE Mississippi Region	3,302,913	1,242,008	316,930	1,558	6,095

6.4.3 Discussion

The increased performance of using AWS relative to the local cluster is likely due to faster hardware. The estimated serial run time for the NFIE Mississippi Region alone is much longer than 12 hours (43,200 seconds). Thus the distributed computing, facilitated through CondorPy, is necessary to process the forecasts on the 12-hour release cycle.

Even though a Tethys app was created to visualize the results of the app, the modeling workflow is run as a scheduled service on a cluster that is independent of Tethys; CondorPy is used as a stand-alone library to facilitate submitting jobs to the HTC clusters in the workflow.

While not run operationally, we developed a script that has the ability to use TethysCluster to dynamically provision the clusters needed every 12-hours, and then shut them down once the jobs were finished computing. If AWS or Azure were used to run the forecasts operationally then this could significantly reduce the cost of the computing resources.

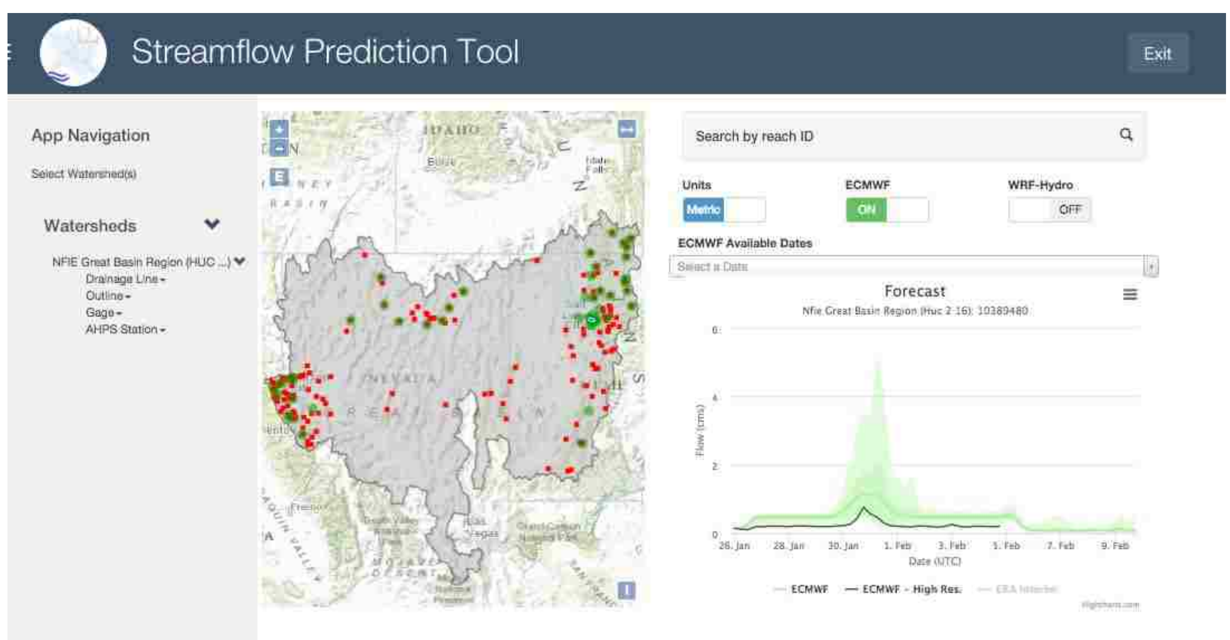


Figure 6-17. The Streamflow Prediction Tool displaying streamflow forecasts for a reach in the Great Basin Region.

7 DISCUSSION AND CONCLUSIONS

The objective of this research has been to facilitate the use of HTC and cloud computing in modeling workflows and web applications. I accomplished this by creating a comprehensive Python toolkit consisting of the two libraries, TethysCluster and CondorPy, which together meet the following requirements: (1) programmatic access to diverse, dynamically scalable computing resources; (2) a batch scheduling system to queue and dispatch the jobs to the computing resources; (3) data management for job inputs and outputs; and (4) the ability for jobs to be dynamically created, submitted, and monitored from the scripting environment. This chapter discusses the contributions that this research has made and describes future work that could be done to expand upon the research presented here.

7.1 Technical Contributions

With the adaptations that I made to StarCluster to create TethysCluster I was able to successfully meet the first three requirements that I defined. The main contribution of TethysCluster is adding the ability to provision cloud-computing clusters composed of Windows VMs on AWS and add the ability to provision Linux VMs on Microsoft Azure. This makes TethysCluster a more flexible tool, able to provide the diverse computing environments that are needed to support water resources modeling. By using HTCondor as the middleware for configuring an HTC system on the clusters configured by TethysCluster I meet the second two requirements. I integrated TethysCluster into Tethys Platform in the Compute API as well as

through admin tools to allow HTC clusters to easily be provisioned from a web app development platform. TethysCluster can also be used to dynamically scale computing resources to support scheduled workflows, as demonstrated with the Streamflow Prediction Tool.

I created CondorPy as an object-oriented interface for HTCondor to meet the last requirement by enabling programmatic creation and submission of HTC jobs. I used CondorPy to facilitate executing 57,600 instances of a GSSHA model for the Canned GSSHA project and helped with the development of a scripted workflow that uses CondorPy to process 624 forecasts for the Streamflow Prediction Tool. To further facilitate the use of CondorPy in web apps, I created the Jobs API in Tethys Platform, which I then used to create two data management apps and to add the ability to submit GSSHA models to be run in the GSSHA Index Map Editor app.

TethysCluster and CondorPy are free and open source software. TethysCluster maintains the GNU Lesser General Public License that StarCluster was licensed under, and CondorPy is licensed under the BSD 2-clause license. Tethys Compute was contributed to the Tethys Platform open source project (under the BSD 2-clause license). All of these code repositories are on GitHub (see Appendix A).

7.2 Future Work

Because of the contributions made by this research there are numerous possibilities to either leverage these contributions or expand upon them in future research. Several of these possibilities are described here.

7.2.1 Enhancements to TethysCluster

TethysCluster provides the basic functionality needed to support hydrologic modeling workflows. Additional features could provide further benefit. Since Microsoft Azure is in the process of developing a new Python API, it may become possible to access the public SSH key that is necessary to support Windows nodes. Also, additional research could find ways to extend the full set of features that are supported on Linux nodes, to Windows nodes. Also, StarCluster supports automated load balancing (i.e. the ability to scale the cluster size) based on the job queue when using SGE. This feature is not supported when using HTCondor. Additional research could extend this feature to both TethysCluster and StarCluster. TethysCluster diversifies the available computing resources that can be used by supporting Azure in addition to AWS. Adding support for additional cloud providers would increase the flexibility of TethysCluster. While not all cloud providers currently support all of the features needed by TethysCluster, a likely candidate that should be explored is OpenStack, which is used by the commercial cloud provider RackSpace, but also is used by many private clouds.

7.2.2 Enhancements to Tethys Compute

Tethys Compute has been proven to support HTC in web applications. However, there are enhancements that could make it an even more powerful and flexible set of tools. A WorkflowJob in the JobsAPI would allow users to take advantage of CondorPy Workflows to provide support for hierarchical jobs in Tethys. Additional options and tools could be integrated into the Tethys Compute admin pages to allow the web interface to take advantage of all of the flexibility in creating cluster that is provided by TethysCluster.

7.2.3 Additional Applications

The comprehensive computing toolkit has supported several research applications that needed programmatic access to HTC resources. Additional research applications could also make use of these tools.

One such application, parent-child modeling with GSSHA, is currently being researched. Hydrologic modeling often requires modeling large domains because downstream locations of interest are dependent on the hydrologic processes of the watershed upstream. When using advanced hydrologic models that are spatially-distributed and physically-based, like GSSHA, modeling large domains can be computationally demanding and time consuming. For some models this problem can be partially alleviated by having a variable discretization with higher resolution over areas of greatest interest. However, many models only support uniform grid size and therefore if a high-resolution grid is desired over part of the model it must be used over the whole domain. While GSSHA only supports uniform grids it does have a mechanism for handling differing grid sizes over a domain through parent-child models.

Parent-child modeling is a method for decomposing a large domain into cascading, hierarchical sub-domains. First level sub-basins, in the highest part of the watershed can all be run independently from each other, and thus in parallel. Second level sub-basins rely on input from the first level sub-basins but are independent from one another. A watershed may be broken into various hierarchical levels ending at the base level, which relies on all previous levels. When the domain is broken up into various sub-domains, each represented by its own model, the sub-domains can have differing grid sizes, making it possible for there to be higher resolution only in the area of interest.

Since the model is structured in a hierarchical way, with sub-basins at the same level being independent of each other, it is a perfect application for a CondorPy workflow. By implementing the parent-child models in a workflow the computation can be more efficient since sub-basins can be run in parallel while sill keeping the necessary parent-child order.

Another application that could facilitate additional research would be to develop a Tethys app to run stochastic GSSHA models. Many study areas could benefit from a stochastic analysis to understand and characterize uncertainty, but the challenge of getting enough computational resources to run the models hundreds or thousands of times is prohibitive. A Tethys app that would allow any model to be uploaded and then run stochastically would allow modelers to spend more time focusing on the hydrologic questions that the model should answer rather than focusing on the modeling process. This research has produced the tools necessary to build this type of application, and it could just as easily be done for models other than GSSHA.

Related research could develop methods for managing, analyzing, and summarizing the large amounts of data that result from a stochastic analysis in a web environment. Tethys Platform provides many tools for visualizing data, but research is needed to find meaningful and manageable ways to visualize large amounts of data produced by stochastic modeling in this environment.

My research has produced tools to support dynamically accessing computing resources for large water resources modeling tasks, which meet the four requirements that I identified. These tools have been shown to support the computing needs of research that has pushed the boundaries of water resources modeling, and have the potential to continue to do so in many other applications.

REFERENCES

Amazon Web Services, I., 2014. What is Cloud Computing?

Amazon Web Services, I., 2016a. Amazon EC2 Security Groups for Linux Instances - Amazon Elastic Compute Cloud. Amazon Web Services, Inc.

Amazon Web Services, I., 2016b. Instance Metadata and User Data - Amazon Elastic Compute Cloud. Amazon Web Services, Inc.

Anderson, J. M., 2015. A Cloud-Based GSSHA Index Map Editor Utility for Watershed Decision Support, Brigham Young University, All Theses and Dissertations.

Apache Software Foundation, 2011. Deltacloud API. Apache Software Foundation.

Armbrust, M., A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica and M. Zaharia, 2010. A view of cloud computing. *Commun. ACM* **53**:50-58.

Behzad, B., A. Padmanabhan, Y. Liu, Y. Liu and S. Wang, 2011. Integrating CyberGIS gateway with Windows Azure: a case study on MODFLOW groundwater simulation. *In*: *Proceedings of the ACM SIGSPATIAL Second International Workshop on High Performance and Distributed Geographic Information Systems.* ACM, Chicago, Illinois, pp. 26-29.

Bockelman, B., 2013. HTCondor Python Tutorial. *In*: *HTCondor Week.* Madison, Wisconsin.

Buyya, R., C. Vecchiola and S. T. Selvi, 2013. *Mastering cloud computing*, Tata McGraw-Hill Education, ISBN 1259029956

Condor Team, 2010. Remote Condor.

Condor Team, 2014. HTCondor™ Version 8.3.1 Manual. University of Wisconsin-Madison.

Couvares, P., T. Kosar, A. Roy, J. Weber and K. Wenger, 2007. Workflow Management in Condor. *In*: *Workflows for e-Science,* I. Taylor, E. Deelman, D. Gannon and M. Shields (I. Taylor, E. Deelman, D. Gannon and M. Shields(I. Taylor, E. Deelman, D. Gannon and M. Shieldss). Springer London, pp. 357-375.

David, C. H., D. R. Maidment, G.-Y. Niu, Z.-L. Yang, F. Habets and V. Eijkhout, 2011. River Network Routing on the NHDPlus Dataset. *Journal of Hydrometeorology* **12**:913-934.

Delipetrev, B., A. Jonoski and D. P. Solomatine, 2014. Development of a web application for water resources based on open source software. *Computers & Geosciences* **62**:35-42.

Doherty, J. and D. Welter, 2010. A short exploration of structural noise. *Water Resources Research* **46**:W05525.

Dolder, H. G., N. L. Jones and E. J. Nelson, 2015. Simple Method for Using Precomputed Hydrologic Models in Flood Forecasting with Uniform Rainfall and Soil Moisture Pattern. *Journal of Hydrologic Engineering* **0**:04015039.

Downer, C. W. and F. L. Ogden, 2004. GSSHA: Model To Simulate Diverse Stream Flow Producing Processes. *Journal of Hydrologic Engineering* **9**:161-174.

Fienen, M. N., J. P. Masterson, N. G. Plant, B. T. Gutierrez and E. R. Thieler, 2013. Bridging groundwater models and decision support with a Bayesian network. *Water Resources Research* **49**:6459-6473.

Fisch, K. M., T. Meißner, L. Gioia, J.-C. Ducom, T. Carland, S. Loguercio and A. I. Su, 2014. Omics Pipe: A Computational Framework for Reproducible Multi-Omics Data Analysis. *bioRxiv* 10.1101/008383.

Foster, I., Y. Zhao, I. Raicu and S. Lu, 2008. Cloud computing and grid computing 360-degree compared. *In*: *Grid Computing Environments Workshop, 2008. GCE'08.* Ieee, pp. 1-10.

Glenis, V., A. S. McGough, V. Kutija, C. Kilsby and S. Woodman, 2013. Flood modelling for cities using Cloud computing. *Journal of Cloud Computing* **2**.

Gunarathne, T., B. Zhang, T.-L. Wu and J. Qiu, 2013. Scalable parallel computing on clouds using Twister4Azure iterative MapReduce. *Future Generation Computer Systems* **29**:1035-1048.

Harvey, B. and S. Ji, 2015. Cloud-Scale Genomic Signals Processing for Robust Large-Scale Cancer Genomic Microarray Data Analysis. *Biomedical and Health Informatics, IEEE Journal of* **PP**:1-1.

Heße, F., H. Savoy, C. A. Osorio-Murillo, J. Sege, S. Attinger and Y. Rubin, Characterizing the impact of roughness and connectivity features of aquifer conductivity using Bayesian inversion. *Journal of Hydrology* http://dx.doi.org/10.1016/j.jhydrol.2015.09.067.

Horizons Systems Corporation, 2011. National Hydrography Plus Version 2 Dataset. *In*: *Weaving the National Hydrologic Geospatial Fabric,* Horizons Systems Corporation (Horizons Systems Corporation)Horizons Systems Corporations).

Huang, Q. and C. Yang, 2011. Optimizing grid computing configuration and scheduling for geospatial analysis: An example with interpolating DEM. *Computers & Geosciences* **37**:165-176.

Huang, X., G. Cao, J. Liu, H. Prommer and C. Zheng, 2014. Reactive transport modeling of thorium in a cloud computing environment. *Journal of Geochemical Exploration* http://dx.doi.org/10.1016/j.gexplo.2014.03.006.

Humphrey, M., N. Beekwilder, J. L. Goodall and M. B. Ercan, 2012. Calibration of watershed models using cloud computing. *In*: *E-Science (e-Science), 2012 IEEE 8th International Conference on.* pp. 1-8.

Humphrey, M., Z. Hill, K. Jackson, C. van Ingen and R. Youngryel, 2011. Assessing the Value of Cloudbursting: A Case Study of Satellite Image Processing on Windows Azure. *In*: *E-Science (e-Science), 2011 IEEE 7th International Conference on.* pp. 126-133.

Hunt, R. J., J. Luchette, W. A. Schreuder, J. O. Rumbaugh, J. Doherty, M. J. Tonkin and D. B. Rumbaugh, 2010. Using a Cloud to Replenish Parched Groundwater Modeling Efforts. *Ground Water* **48**:360-365.

Jones, N., J. Nelson, N. Swain, S. Christensen, D. Tarboton and P. Dash, 2014. Tethys: A Software Framework for Web-Based Modeling and Decision Support Applications. *In*: *Ames, D.P., Quinn, N.W.T., Rizzoli, A.E. (Eds.), Proceedings of the 7th International Congress on Environmental Modelling and Software, June 15-19.* San Diego, California, USA.

Juve, G., E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman and P. Maechling, 2009. Scientific workflow applications on Amazon EC2. *In*: *E-Science Workshops, 2009 5th IEEE International Conference on.* IEEE, pp. 59-66.

Kim, I., J.-Y. Jung, T. F. DeLuca, T. H. Nelson and D. P. Wall, 2012. Cloud computing for comparative genomics with windows azure platform. *Evolutionary bioinformatics online* **8**:527.

Kollet, S., J. Schumacher, C. Bürger and D. Bösel, 2011. Cloud computing with ParFlow: Introduction of a newly developed Web interface. *MODFLOW and More*:52-57.

Li, Y. and M. Mascagni, 2003. Analysis of Large-Scale Grid-Based Monte Carlo Applications. *International Journal of High Performance Computing Applications* **17**:369-382.

Litzkow, M. J., M. Livny and M. W. Mutka, 1988. Condor-a hunter of idle workstations. *In*: *Distributed Computing Systems, 1988., 8th International Conference on.* IEEE, pp. 104-111.

Liu, Y., A. Y. Sun, K. Nelson and W. E. Hipke, 2012. Cloud computing for integrated stochastic groundwater uncertainty analysis. *International Journal of Digital Earth* **6**:313-337.

Liu, Z., H. Zou and W. Ye, 2015. Simulation Runner: A Cloud-Based Parallel and Distributed HPC Platform. *In*: *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on.* IEEE, pp. 885-892.

Livny, M., J. Basney, R. Raman and T. Tannenbaum, 1997. Mechanisms for high throughput computing. *SPEEDUP journal* **11**:36-40.

Lu, W., J. Jackson and R. Barga, 2010. AzureBlast: a case study of developing science applications on the cloud. *In*: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing.* ACM, Chicago, Illinois, pp. 413-420.

Luchette, J., G. K. Nelson, C. McLane and L. I. Cecan, 2009. Unlimited virtual computing capacity using the cloud for automated parameter estimation. *In*: *Proceedings of the 1st PEST Conference.* pp. 1-16.

Lusk, E., S. Huss, B. Saphir and M. Snir, 2012. MPI: A message-passing interface standard 3.0.

Microsoft, 2016. Name Resolution for VMs and Role Instances. Microsoft.

Miras, H., R. Jiménez, C. Miras and C. Gomà, 2013. CloudMC: a cloud computing application for Monte Carlo simulation. *Physics in Medicine and Biology* **58**:N125.

Monteiro, A., J. S. Pinto, C. Teixeira and T. Batista, 2011. Cloud Interchangeability-Redefining Expectations. *In*: *CLOSER.* pp. 180-183.

Ogden, F. L., N. Raj Pradhan, C. W. Downer and J. A. Zahner, 2011. Relative importance of impervious area, drainage density, width function, and subsurface storm drainage on flood runoff from an urbanized catchment. *Water Resour. Res.* **47**:W12503.

Oliphant, T. E., 2007. Python for Scientific Computing. *Computing in Science & Engineering* **9**:10-20.

Osorio-Murillo, C. A., M. W. Over, H. Savoy, D. P. Ames and Y. Rubin, 2015. Software framework for inverse modeling and uncertainty characterization. *Environmental Modelling & Software* **66**:98-109.

Pérez, F., B. E. Granger and J. D. Hunter, 2011. Python: An Ecosystem for Scientific Computing. *Computing in Science & Engineering* **13**:13-21.

Petri, I., H. Li, Y. Rezgui, Y. Chunfeng, B. Yuce and B. Jayan, 2014. A HPC based cloud model for real-time energy optimisation. *Enterprise Information Systems* 10.1080/17517575.2014.919053:1-21.

Plankytronixx, 2014. I'm confused between Azure Cloud Services and Azure VMs – what the ****? Microsoft Corporation, MSDN Blogs.

Raicu, I., 2009. Many-task computing: Bridging the gap between high-throughput computing and high-performance computing, Ph.D., The University of Chicago, Ann Arbor.

Rouholahnejad, E., K. C. Abbaspour, M. Vejdani, R. Srinivasan, R. Schulin and A. Lehmann, 2012. A parallelization framework for calibration of hydrological models. *Environmental Modelling & Software* **31**:28-36.

Singh, V. and D. Woolhiser, 2002. Mathematical Modeling of Watershed Hydrology. *Journal of Hydrologic Engineering* **7**:270-292.

Smemoe, C. M., 2004. Floodplain risk analysis using flood probability and annual exceedance probability maps, Thesis (Ph D), Brigham Young University. Dept. of Civil and Environmental Engineering, 2004.

Snow, A. D., S. D. Christensen, N. R. Swain, E. J. Nelson, D. P. Ames, N. L. Jones, D. Ding, N. Noman, C. H. David and F. Pappenberger, In Press. A Cloud-Based High-Resolution National Hydrologic Forecast System Downscaled from a Global Ensemble Land Surface Model. *JAWRA Journal of the American Water Resources Association*.

StarCluster, 2014. STAR: Cluster - Home. MIT.

Subramanian, V., W. Liqiang, L. En-Jui and C. Po, 2010. Rapid Processing of Synthetic Seismograms Using Windows Azure Cloud. *In*: *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on.* pp. 193-200.

Sun, A., 2013. Enabling collaborative decision-making in watershed management using cloud-computing services. *Environmental Modelling & Software* **41**:93-97.

Swain, N. R., S. D. Christensen, A. D. Snow, H. G. Dolder, E. Goharian, J. Anderson, N. L. Jones, E. J. Nelson, D. P. Ames, G. P. Williams and S. J. Burian, In Press. The Design and Implementation of an Open Source Platform for Water Resources Web App Development. *In*: *Environmental Modelling & Software.*

Swain, N. R., K. Latu, S. D. Christensen, N. L. Jones, E. J. Nelson, D. P. Ames and G. P. Williams, 2015. A review of open source software solutions for developing water resources web applications. *Environmental Modelling & Software* **67**:108-117.

Taylor, S., 2013. High Performance Computing of Hydrologic Models Using HTCondor, Brigham Young University.

Thain, D., T. Tannenbaum and M. Livny, 2005. Distributed computing in practice: the Condor experience. *Concurrency & Computation: Practice & Experience* **17**:323.

Vaquero, L. M., L. Rodero-Merino, J. Caceres and M. Lindner, 2008. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.* **39**:50-55.

Wang, L., J. Tao, M. Kunze, A. C. Castellanos, D. Kramer and W. Karl, 2008. Scientific Cloud Computing: Early Definition and Experience. *In*: *HPCC.* pp. 825-830.

Wu, Z. Y. and M. Khaliefa, 2012. Cloud Computing for High Performance Optimization of Water Distribution Systems. *In*: *World Environmental and Water Resources Congress 2012.* pp. 679-686.

Zhao, G., B. A. Bryan, D. King, Z. Luo, E. Wang, U. Bende-Michl, X. Song and Q. Yu, 2013. Large-scale, high-resolution agricultural systems modeling using a hybrid approach combining grid computing and parallel processing. *Environmental Modelling & Software* **41**:231-238.

APPENDIX A SOFTWARE AVAILABILITY

The software described in this document is open source and publicly available to use and enhance. Links to access each project is listed below:

A.1 TethysCluster

Project Home Page: http://www.tethysplatform.org/TethysCluster/

Code Repository: https://github.com/tethysplatform/TethysCluster/

Documentation: http://tethyscluster.readthedocs.org

A.2 CondorPy

Project Home Page: http://www.tethysplatform.org/condorpy/

Code Repository: https://github.com/tethysplatform/condorpy/

Documentation: http://condorpy.readthedocs.org

A.3 Tethys Compute (Tethys Platform)

Project Home Page: http://www.tethysplatform.org

Code Repository: https://github.com/tethysplatform/tethys/

Documentation: http://docs.tethysplatform.org