ASTES

Advances in Science, Technology and Engineering Systems Journal
Vol. 3, No. 3, XX-YY (2018)
www.astesj.com

ASTES Journal
ISSN: 2415-6698

Virtual Watershed System: A Web-Service-Based Software Package For Environmental Modeling

Rui Wu[1], Connor Scully-Allison[*2], Moinul Hossain Rifat[2], Jose Painumkal[2], Sergiu Dascalu[2], Frederick C Harris, Jr.[2]

[1]*Department of Computer Science, East Carolina University, 27858, USA*

[2]*Computer Science & Engineering Department, University of Nevada, Reno, 89557, USA*

ARTICLEINFO

Article history:
Received:
Accepted:
Online:

Keywords:
Web-based Virtual Watershed System
Model as service
Cloud-based application
Hydrologic model

ABSTRACT

The physically-based environmental model is a crucial tool used in many scientific inquiries. With physical modeling, different models are used to simulate real world phenomena and most environmental scientists use their own devices to execute the models. A complex simulation can be time-consuming with limited computing power. Also, sharing a scientific model with other researchers can difficult, which means the same model is rebuilt multiple times for similar problems. A web-service-based framework to expose models as services is proposed in this paper to address these problems. The main functions of the framework include model executions in cloud environments, NetCDF file format transmission, model resource management with various web services. As proof of concept, a prototype is introduced,implemented and compared against existing similar software packages. Through a feature comparison with equivalent software, we demonstrate that the Virtual Watershed System (VWS) provides superior customization through its APIs. We also indicate that the VWS uniquely provides friendly, usable UIs enabling researchers to execute models.

1 Introduction

Modeling has become an indispensable tool in growing environmental scientists' understanding of how natural systems react to changing conditions. It sheds light on complex environmental mysteries and helps researchers in formulating policies and decisions on future scenarios. Environmental modeling is highly challenging as it involves complex mathematical computations, rigorous data processing, and convoluted correlations between numerous parameters. Three commonly-used and important scientific software quality measures are maintainability, quality, and scalability. Issues like data storage, coupling models, retrieval, and running are hard problems and need to be addressed with extra efforts from software engineering perspective. It is a challenging job to design integrated systems that can address all these issues.

It is essential to build high-quality software tools and design efficient frameworks for scientific research. Abundant scientific model data are generated and collected in recent years. Software engineering can assist this emergence through the creation of distributed software systems and frameworks enabling scientific collaboration with previously disparate data and models. It is challenging to implement software tools for interdisciplinary research because of the problem of communication and team building among different scientific communities. For example, the same terminology can have different meanings in different domains. This increases difficulties in comprehending software requirements when a project involves stakeholders (e.g. researchers) from different fields.

Most work described in this paper is for Watershed Analysis, Visualization, and Exploration (WC-WAVE), which is a NSF EPSCoR-supported project and initiated by jurisdictions of EPSCoR of Nevada, Idaho and New Mexico . The WC-WAVE project includes three principal components: watershed science, data cyberinfrastructure and data visualization [1]. The project main goal is to implement VWS with the collaborations between cyberinfrastructure team members and hy-

[*]Connor Scully-Allison, 1664 N. Virginia Street, CSE (0171), Reno, NV 89557, (775) 771-1469 & cscully-allison@nevada.unr.edu

drologists. The platform is able to store, share, model and visualize data on-demand through an integrated system. These features are crucial for hydrologic research.

Different hydrologic models, such as ISNOBAL and PRMS, are commonly used by WC-WAVE hydrologists. These models are leveraged to predict or examine hydrologic processes of Lehman Creek in Nevada, Dry Creek and Reynolds Creek in Idaho, and Jemez Creek in New Mexico. We propose a framework for representing model data in a standardized format called the "Network Common Data Format" (NetCDF) and exposing these hydrologic models through web services. This framework is based on our previous work introduced in [2]. To improve on our prior work we have implemented some new Docker APIs to control system components wrapped in docker containers. To simplify data extraction, modification, and storage, NetCDF data format [3] is used in the system for grid-organized scientific data. Most parts of the system framework can be reused for data-intensive purpose because it is designed with the blueprint and template concepts. ISNOBAL and PRMS are physically-based models which can produce very accurate results. However, these two models require abundant computing power. To solve the challenge, we leverage a cluster to execute models in parallel. ISNOBAL and PRMS are used in this paper to demonstrate the ideas and functionality of the proposed framework. Throughout the remainder of this paper, we refer to this prototype system as the VWS.

ISNOBAL is a grid-based DEM (Digital Elevation Model) and created to model the seasonal snow cover melting and development. The model author is Marks et al. [4] and it is initially developed for Utah, California, and Idaho mountain basins. The model determines runoff and snowmelt based on terrain, precipitation, region characteristics, climate, and snow properties [4].

PRMS is short for Precipitation-Runoff Modeling System and is initially written with FORTRAN in 1983. PRMS is prevalent physical process based distributed-parameter hydrologic model and the main function of a PRMS model is to evaluate a watershed response to different climate and land usage cases [5, 6, 7]. It composed of algorithms describing various physical processes as subroutines. The model, now in its fourth version, has become more mature over the years of development. Different hydrology applications, such as measurement of groundwater and surface water interaction, the interaction of climate and atmosphere with surface water, water and natural resource management, have been done with the PRMS model [5, 6, 7].

This paper is organized as follows in its remaining sections: Section 2 introduces background and related work; Section 3 describes the system design; Section 4 describes the prototype system and how the software was built using RESTful APIs; Section 5 compares our work with related tools; and Section 6 contains the papers conclusions and outlines planned future work.

2 Background and Related Work

"How to implement software for interdisciplinary research?" is an interesting question and there exists some successful work on environments and frameworks which seek to answet this question. In this section, relevant, popular earth science applications and frameworks are introduced.

Community Surface Dynamics Modeling System (CSDMS) was a project started in 1999 to conduct expeditious research of earth surface modelers by creating a community driven software platform. CSDMS applies a component-based software engineering approach in the integration of plug-and-play components, as the development of complex scientific modeling system requires the coupling of multiple, independently developed models [8]. CSDMS allows users to write their components in any popular language. Also they can use components created by others in the community for their simulations. CSDMS treats components as pre-compiled units which can be replaced, added to, or deleted from an application at runtime via dynamic linking. Many key requirements drove the design of CSDMS, including the support for multiple operating systems, language interoperability across both procedural and object-oriented programming languages, platform independent graphical user interfaces, use of established software standards, interoperability with other coupling frameworks and use of HPC tools to integrate parallel tools and models into the ecosystem.

A leading hydrologic research organization is CUAHSI. "CUAHSI" , and acronym for "Consortium of Universities for the Advancement of Hydrologic Science Inc.," represents universities and international water science-related organizations. One of most highly esteemed products is HydroShare, which is a hydrologic data and model sharing web application. Hydrologists can easily access different model datasets and share their own data. Besides this, this platform offers many distributed data analysis tools. A model instance can be deployed in a grid, cloud or high-performance computing cluster with HydroShare. Also, a hydrologist is able to publish outcomes of their research, such as a dataset or a model. In this way, scientists use the system as a collaboration platform for sharing information. HydroShare exposes its functionality with Application Programming Interfaces (APIs), which means its web application interface layer and service layer are separated. This enables interoperability with other systems and direct client access [9].

Model as a Service is proposed by Li et al. [10] for Geoscience Modeling. It is a cloud-based solution and Li et al. has implemented a prototype system to execute high CPU and memory usage models remotely as a service with third party platform, such as AWS (Amazon Web Service) and Microsoft Azure.

The key idea of MaaS is that model executions can be done through a web interface with user inputs. Computer resources are provisioned with a cloud provider, such as Microsoft Azure. The model registration is done in the framework with a virtual machine image

repository. If a model is registered and placed in the repository, it can be shared by other users and multiple model instances can be executed in parallel based on demand.

McGuire and Roberge designed a social network to promote collaboration between watershed scientists. Despite being highly available, the collaboration between the general public, scientists, and citizen has not been leveraged. Also, hydrologic data is not integrated in any system. The main goal of this work is to design a collaborative social network for multiple watershed scientific and hydrologic user groups [11]. However, more efforts need to be done.

The Demeter Framework by Fritzinger et al. [12] represents another attempt to utilize software frameworks as a scientific aid in the area of climate change research. A software framework named "The Demeter Framework" is introduced in the paper and one of the key ideas is a component-based approach to integrate different components into the system for the "model coupling problem." "The model coupling problem" refers to using a model's outputs as another model's inputs to solve a problem.

Walker and Chapra proposed a web-based client-server approach for solving the problem of environmental modeling compared to the traditional desktop-based approach. The authors assert that, with the improvement in modern day web browsers, client-side approaches offer improved user interfaces compared to traditional desktop software. In addition, powerful servers enable users to perform simulations and visualizations within the browser [13].

The University of New Mexico has implemented a data engine named GSToRE (Geographic Storage, Transformation and Retrieval Engine). The engine is designed for earth scientific research and the main functions of the engine are data delivery, documentation, and discovery. It follows the combination of community and open standards and implemented based on service oriented architecture. [14].

2.1 Service Oriented Architecture

Industry has shown more and more interests on Service Oriented Architecture (SOA) to implement software systems. [15]. The main idea of SOA is to have business logic decomposed into different units (or services). These units are self-contained and can be easily deployed with container techniques, such as Docker.

Representational State Transfer Protocol (REST) is primarily an architectural style for distributed hypermedia systems introduced by Fielding, Roy Thomas in his Ph.D. dissertation [16]. REST defines a way for a client-server architecture on how a client and a server should interact, by using a set of principles. REST has been adopted for building the main architecture of the proposed system. Statelessness, uniform Interface, and cache are the main characteristics of a REST client-server architecture [16]. It is for these characteristics that REST is leveraged in our proposed system.

Statelessness Statelessness is the most important property or constraint for a client-server architecture to be RESTful. The communication between the client and the server must be stateless, which means the server is not responsible for keeping the state of the communication. It is the client's responsibility. A request from the client must contain all the necessary information for the server to understand the request [16, 17]. Two subsequent requests to the server will not have any interdependence between each other. Introducing this property on the client-server architecture presents several benefits regarding visibility, reliability, and scalability [16, 17]. For example, as the server is not responsible for keeping the state and two subsequent requests are not interrelated, multiple servers can be distributed across a load-balanced system where different servers can be responsible for responding to different requests by a client.

Uniform Interface Another important property of a RESTful architecture is it provides a uniform interface for the client to interact with a server. Instead of an application's particular implementation, it forces the system to follow a standardized form. For example HTTP 1.1 which is a RESTful protocol provides a set of verbs (e.g., GET, POST, PUT, DELETE, etc.) for the client to communicate with the server. The verbs, such as "GET" and "POST", work as an interface making the client-server communication generic[16].

Cache REST architecture introduces cache constraint to improve network efficiency [16]. A server can allow a client to reuse data by enabling explicitly for labeling some data cacheable or non-cacheable. A server can serve data that will not change in the future as cached content, allowing the client to eliminate partial interaction with that data in a series of requests.

2.2 REST Components

The main components of REST architecture include resources, representations, and resource identifiers.

Resources The resource is the main abstract representation of data in REST architecture [16]. Any piece of data in a server can be represented as a resource to a client. A document, an image, data on today's weather, a social profile, *everything* is considered a resource in the server. Formally, a resource is a temporarily varying function of $M_R(t)$ that maps to a set of entities for time t [16].

Representations A resource is the abstract building block of the data in a web server. For a client to consume the resource it needs to be presented in a way the client can understand. This is called representation. A representation is a presentation format for representing the current state of a resource to a consumer. Some commonly used resource representation format in the current standard are HTML (Hypertext Markup

Language), XML (Extensible Markup Language), JSON (JavaScript Object Notation), etc. A server can expose data content in different representations so that consumer can access the resources through resource identifiers (discussed in Section 2.2) in the desired format.

Resource Identifiers A resource is uniquely identified through a resource identifier in a RESTful architecture. For example, in HTTP Uniform Resource Identifier (URI) is used to identify a resource in a server. A URI can be thought as the address of a resource in the server [18]. A resource identifier is the key for a client to access and manipulate a resource in the server.

2.3 Microservice Architecture

Software as a Service (SaaS), as a new software delivery architecture, has emerged to leverage the widely-used REST standard for processing in addition to data transfer. The main advantage of SaaS is that it does not require local installation and this is a significant IT trend based on industry analysis [19].

Similarly, "microservice" decomposes an application into small components (or services) and these components communicate with each other through APIs. "Microservice" is a solution to monolithic architecture relevant problems. [20]. Because of the "microservice" characteristics, an application can be easily scaled and the deployment risks have been reduced without interrupting other services.

Traditional monolithic applications are built as a single unit using a single language stack and often composed of three parts, a front end client, a back-end database and an application server sitting in the middle that contains the business logic. Here, the application server is a monolith that serves as a single executable. A monolithic application can be scaled horizontally by replicating the application server behind a load balancer to serve the clients at scale. The biggest issue with monolithic architecture is that, as the application grows, the deployment cycle becomes longer as a small change in the codebase requires the whole monolith to be rebuilt and deployed [20]. It leads to higher risk for maintenance as the application grows. These pitfalls have lead to the idea of decomposing the business capabilities of an application into self-contained services.

Being a relatively new idea, researchers have attempted to formalize a definition and characteristics of *Microservices*. Lewis and Fowler [20] has put together a few essential characteristics of a microservice architecture that are described in brief in the following sections.

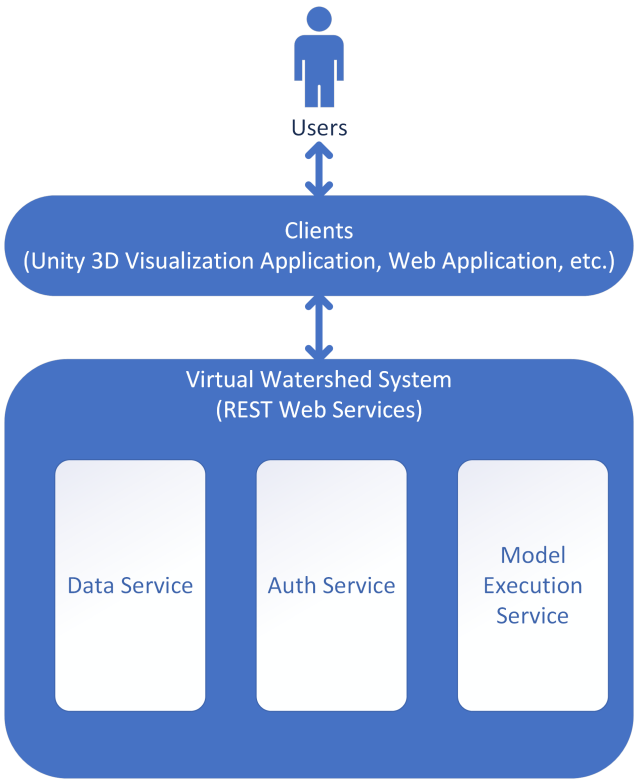

Figure 1: High-level diagram of VWS: clients and VWS communicate through REST Web Services. Possible clients include a script and applications. VWP provide data service, auth service, and model execution service.

Componentization via Services The most important characteristic of a microservice architecture is that service functionalities need to be componentized. Instead of thinking of components as libraries that use in-memory function calls for inter-component communication, we can think of components in terms out-of-process services that communicate over the network, quite often through a web service or a remote procedure call. A service has to be atomic, doing one thing and doing one thing well [21].

Organized Around Business Capabilities Monolithic applications are typically organized around technology layers. For example, a typical multi-tiered application might be split logically into persistence layer, application layer and UI layer and teams are also organized around the technologies. This logical separation creates the need for inter-team communication even for a simple change. In microservices, the product is organized around business capabilities where a service is concentrated on one single business need and owned by a small team of cross-domain members.

Decentralized Governance In a monolithic application, the product is governed in a centralized manner, meaning it restricts the product to a specific platform or language stack. But in microservices, as each service is responsible implementing an independent business capability, it allows for building different services with different technologies. As a result, the team gets to

choose the tools that are best suited for each of the services.

Newman, in his book Building Microservices [21], has discussed several concrete benefits of using microservices over a monolith. Several important benefits are discussed below in brief:

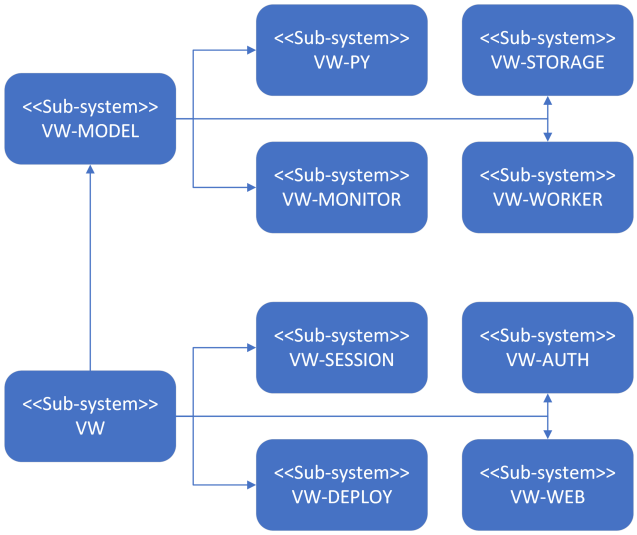

Figure 2: System-level diagram of VWS: two main system components are VW-MODEL and VW. VW-MODEL contains components relative to model execution and VW contains components relative to the platform.

Technology Heterogeneity When independent services are built separately, it allows adoption of different technology stacks for different services. This gives a team multiple advantages: liberty to choose a technology that best suits the need and adopt technology quickly for the needs.

Scaling Monolithic applications are hard to deal with when it comes to scaling. One big problem with monolithic applications is that everything needs to be scaled together as a piece. But microservice allows having control over the scaling application by allowing to scale the services independently.

Ease of Deployment Applying changes for a monolithic application requires the whole application to be re-deployed even for a minor change in the codebase. It poses a high risk as unsuccessful re-deployments even with minor changes can take down the software. Microservices, on the other hand, allow low-risk deployments without interrupting the rest of the services. They also allow having faster development process with small and incremental re-deployments.

3 Proposed Method

Here we introduce detailed design documentation for the VMS. We present the design using many common diagrams used in software engineering, including system diagrams and workflow diagrams. The VWS aims to create a software ecosystem named the *Virtual Watershed* by integrating cyberinfrastructure and visualization tools to advance watershed science research. Fig. 1 shows a general high-level diagram of components of the ecosystem. The envisioned system is centered around services comprised of data, modeling, and visualization components. A high-level description of each depicted component is provided in Fig. 1.

3.1 Data Service and Modeling Service

To create a scalable and maintainable ecosystem of services, a robust data backend is crucial. The envisioned data service exposes a RESTful web service to allow easy storage and management of watershed modeling data. It allows retrieval of data in various OGC (Open Geospatial Consortium) standards like WCS, WMS to allow OGC compliant clients to retrieve data automatically. This feature is very important for data-intensive hydrologic research and is essential for a scientific modeling tool.

Hydrologists often use different modeling tools to simulate and investigate the change of different hydrologic variables around watersheds. The modeling tools are often complex to setup in local environments and takes up a good amount of time setting up [10]. Besides these modeling tools may require high computational and storage resources that make them hard to run in local environments. The proposed modeling service aims to solve these issues by allowing users to submit model execution tasks through simple RESTful web service API. This approach of allowing model-runs through a generic API solves multiple problems:

- It allows the users to run models on demand without having to worry about setting up environments.
- It allows modelers to accelerate the process of running models with different input parameters by submitting multiple model-runs to be run in parallel which might not be feasible in local environment due to lack of computational and storage resources.
- It opens the door for other services and clients to take advantage of the API to automate the process of running models in their workflow.

The primary goal of this Virtual Watershed system is to allow watershed modelers to share their data resources and execute relevant models through web services without having to install the models locally. The system is designed as a collection of RESTful microservices that communicate internally. The web service sits on top of an extensible backend that allows easy integration of models and scalability over model runs and number of users connected to the service.

The system provides a mechanism for integrating new models by conforming with a simple event driven architecture that allows registering a model in the system by wrapping it with a schema driven adaptor.

As model execution is a CPU intensive process, scaling the server with growing number of parallel model execution is an important issue. The proposed architecture aims to solve this problem by introducing a simple database-oriented job queue that provides options for adding more machines as the system grows.

3.2 System Level Design

Several different submodules comprise The Virtual Watershed system. Each module provides different functionalities to the entire framework. This relationship between modules is described with Figure 2. The following paragraphs will explain each submodule in detail.

VW-PY: Using the VW-PY module, users can define adapters to configure compatibility between different models and the VWS. An adaptor is python code which encapsulates a model and that allows for it to be run programmatically. VW-PY handles many aspects of model "wrapping" with this python code through an interface. Through this API, an user can define the code which handles converting between data formats, executing models, and model-progress-based event triggering. This event triggering system provides model-wrapper developers with the tools to emit model execution progress.

VW-MODEL: The VW-MODEL submodule, through a web service, exposes a RESTful API to the user/client. Users can upload, query, and retrieve model run packages using this API.

VW-WORKER: The VW-WORKER is a service which encapsulates model adaptors in a worker service that is organized in a queue data structure. This component communicates with the VW-MODEL component using a redis data-backend.

VW-STORAGE: The VW-STORAGE module provides an interface for object storage for the VW-MODEL component. Developed as a generic wrapper, sysadmins can configure this interface to work with local or cloud-hosted storage providers.

VW-AUTH: The VW-AUTH module provides authentication services for users/clients connecting to the VWS framework. This service provides clients with a JWT token enabling them to securely utilize the other services described in this section.

VW-SESSION: The VW-SESSION sub module coordinates different components in the VWS to provide a common session backend for a single user. Data about each session is maintained and managed with a Redis data store shared between services.

VW-WEB: This is the web-application front-end which provides users with APIs to interact with the model processing modules described above. In a standard use case of this module, users will get access to a session after logging into this system with the VW-AUTH component. Once given a session, users can use this interface to access resources, run models, track progress and upload/download model run resources.

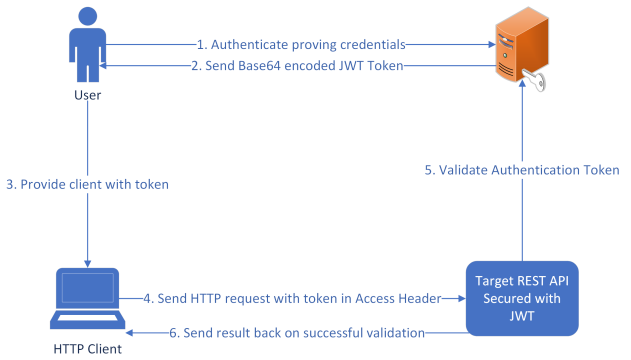

Figure 3: Workflow for accessing secure REST endpoint with JWT token

3.3 Detailed Design

The VWS is comprised of many distinct services and web applications which communicate with one-another to facilitate model runs. We are able to provide secure and centralized communications with the aid of a common authentication gateway. VW-AUTH was developed as a micro service to provide this functionality. It aggregates functionality for registration, authentication, and authorization for system users.

The Authentication component itself exposes RESTful endpoints that provide user access to different services, in addition to endpoints for authentication and registration. To enable the verification of a client's authentication by other services, a JSON Web Token (JWT) based authorization scheme is utilized. As an RFC standard for exchanging information securely between a client and a server, the using JWT ensures a high level of security in this system. A standard workflow for user authorization is depicted in Figure 3.

Users can manage uploaded models via a RESTful API endpoint provided by the Model Web Service. Using this endpoint, a properly authenticated user is able to request a model run, upload necessary input files needed by the model and start model execution. Model run data is stored by the VWS in a dedicated database located on a server operating out of the University of New Mexico [1].

Available models in the system are self-describing with schema that indicates necessary input files, formats, and execution policies. The schema also shows the mapping between user facing and model adaptor parameters. A user/client can use this schema to understand the necessary resources required to run a given model. The a typical model run from the perspective of the clinent side application involves six steps: 1) retrieve the server-side model schema; 2) instantiate a model run session on the server; 3) upload model inputs; 4) run the model; 5) track model run progress; and finally, 6) download outputs.

Figure 4 shows a simple workflow chart from a user or client's perspective. This workflow offers many advantages this workflow and sever-client architecture compared to traditional approaches to model running:

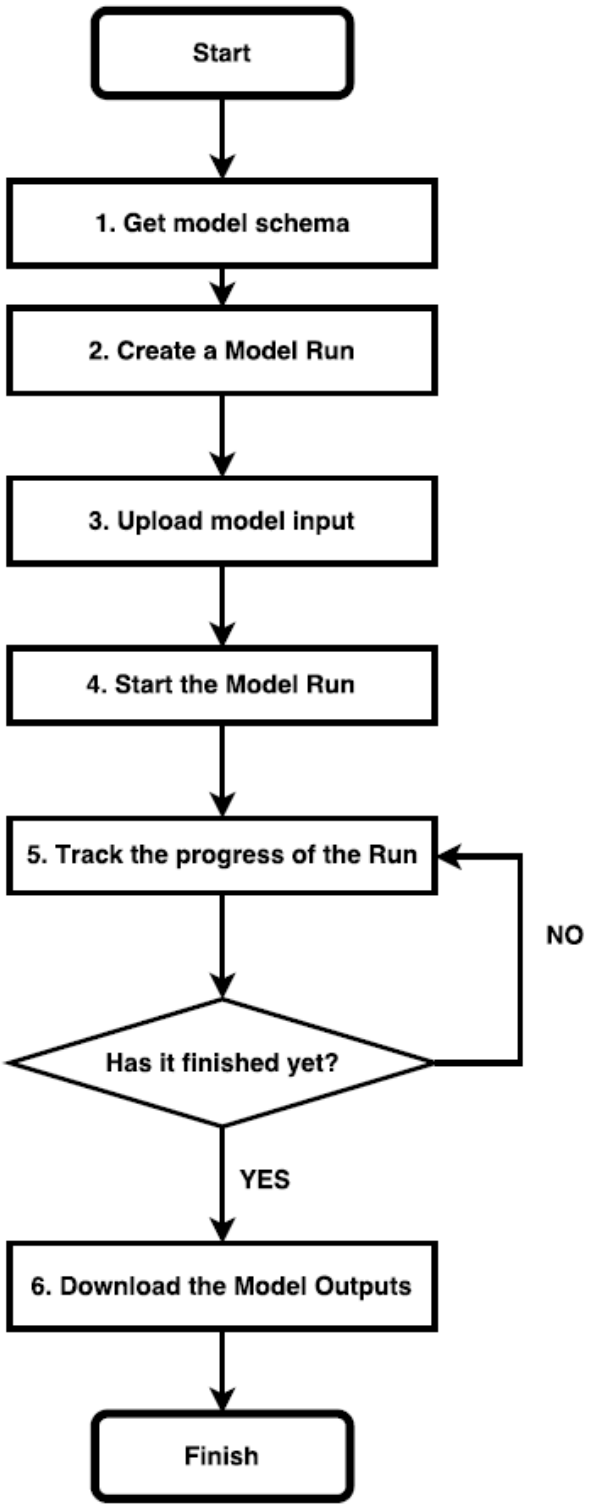

Figure 4: Workflow from user's perspective to run a model

- Users do not need to know internal specifics of the model. Setup details and dependencies are hidden from the user, which provides a more seamless experience.
- Users don't need to worry about installing model dependencies.
- Users can easily initiate multiple parallel model runs on a server, expediting research.
- Users have ubiquitous access to server-side data via provided RESTful APIs.

The first step in creating the architecture for exposing "models as services" is enabling the programmatic running of models. A hurdle to accomplishing this is that a model can have many different dependencies required to run. Programmatic running is accomplished in this program by providing developers the tools to create python wrappers around models. These wrappers expose dependencies to model through container images.

In creating this system we had to strongly consider problems of data format heterogeneity with model inputs and outputs. Different models commonly accept and output data in different file formats. To achieve facilitate greater data interoperability between these models, we provided an option for developers to write NetCDF adapters for each models. A model adaptor is a Python program that handles data format conversion, model execution and progress notification. Adaptor developers must provide converters which define the method of conversion and deconversion between the native formats used by the model and netCDF. An execution function must also be implemented for the model. In this method, resource conversion and model execution occurs. The wrapper reports on model execution events via a provided "event emitter." An event listener can catch these events as they are reported by the emitter. This listener provides a bridge between the internal progress of the model and the user with the aid of a REST endpoint.

Through an adaptor, models can be encapsulated for programmatic execution. However, to bridge frontends with actual model execution a process is required from the model worker module. Utilizing a messaging queue, we create a bridge between the client frontend and worker backend. When a run task is submitted through from the client side, it is placed into the queue and assigned a unique id. The consumer/worker process listens to the queue through a common protocol for new jobs.

The worker service is a server-side python module that has access to a model's executable code and installed dependencies. The worker resides in an isolated server instance that contains the dependencies and libraries of the model installed. This module uses Linux containerization to facilitate easy deployment of model workers.

3.4 Deployment Workflow

A Linux-container-based deployment workflow has been devised for the for many different components of VWS. We utilized the containerization software, Docker, to enable our implementation of this workflow [22]. Docker containers have advantages over traditional virtual machines because they use fewer resources. This workflow allows for iterative deployment, simple scaling of the containerized components, and provides a strategy to register new models in the system.

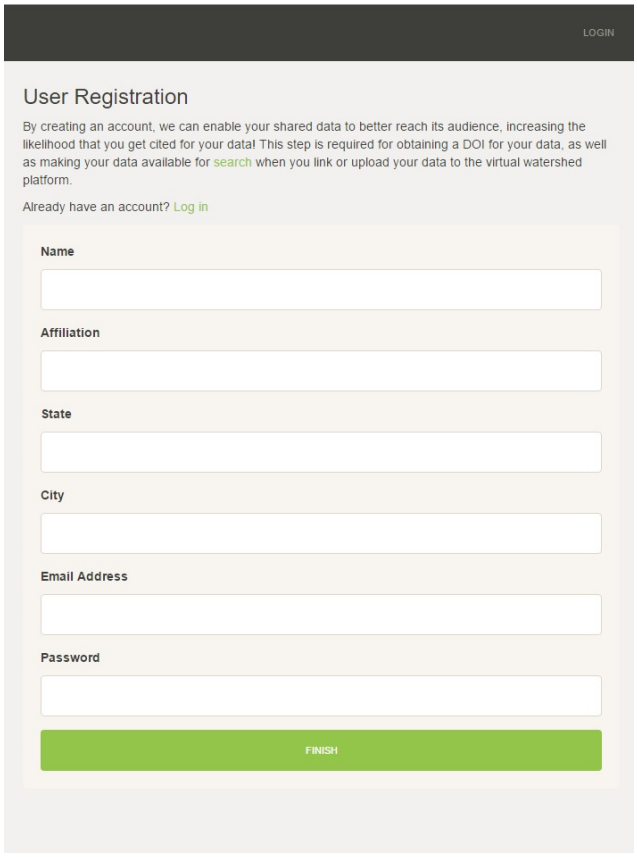

Figure 5: Registration page of Virtual Watershed System

Every VWS component is containerized with Docker. We have a set up a central repository of docker images which contain images for each component in this system. A docker "image" describes a template that provides Docker with the OS, dependencies of an application and the application itself. Docker uses this template to build a working container. Each repository in the Virtual Watershed has a Dockerfile that describes how the component should be built and deployed into a query-able container. The repositories use webhooks to automate the building of docker containers.

It is easy to register new models in this system using this containerization workflow. Using our system, a user may register a model with the creation of docker image describing the model. With this image users can declare the OS, libraries and other dependencies for a model. A typical registration of a model requires the following steps: 1) create a repository for the model; 2) develop the wrapper; 3) specify dependencies within the dockerfile; and finally, 5) create a docker image in the image repository.

4 System Prototype and Testing

The project re-used some code and similar structures to those introduced in [23, 24]. The web service frontends were built using a Python micro-service framework called Flask, with various extensions [25]. Some key libraries used were: 1) Flask-Restless, used to implement the RESTful API endpoints; 2) SQLAlchemy, to map the python data objects with the database schema [26]; 3) PostgreSQL as the database [27]; and, 4) Flask-Security with Flask-JWT which provides the utilities used for security and authentication. Celery was used to implement the task queue for model workers [28]. Redis worked as a repository for model results output by Celery. A REST specification library called Swagger was used to create the specification for the REST APIs. For the web front end, HTML5, CSS, Bootstrap, and Javascript (with ReactsJS) were used.

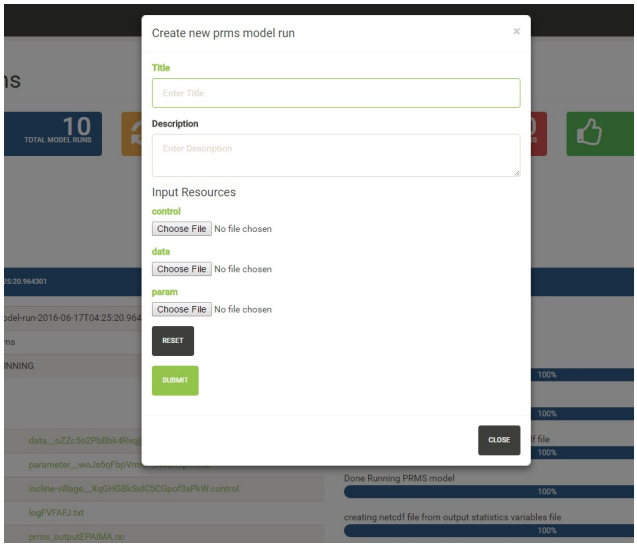

Figure 6: Dynamically generated upload form

To efficiently develop the VWS, an MVC design pattern was used to structure the code. The primary repository used to manage and distribute code was Github. Source code for this project is made publicly available through the Virtual Watersheds GitHub repository [28]. Dockerhub [29] is used for image management, which can automatically update images when Github code is changed.

The prototype system is comprised of two main components 1) the authentication module and 2) the modeling module. First, activities related to the VWS are handled by the authentication module. All standard user management functionality is handled by this module: registration, logins, verification, password management, and authentication token generation. Figure 5 shows the registration page of Virtual Watershed system. In addition to this interface, the VWS also provides endpoints for registration and authentication from user constructed scripts.

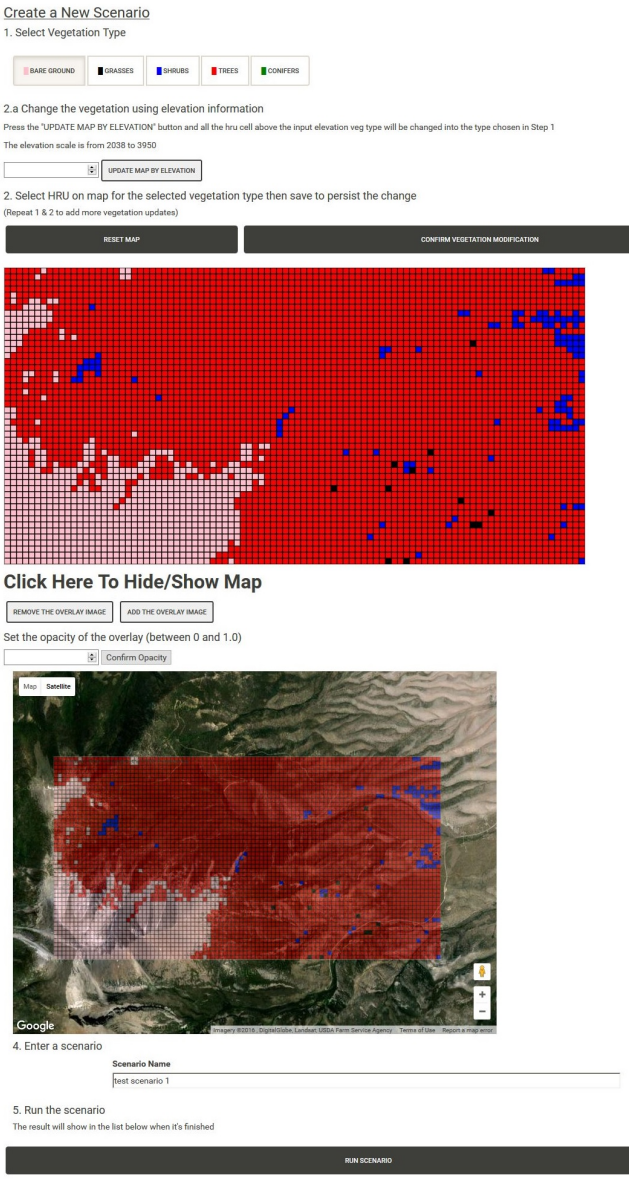

Figure 7: Scenario creation interface for PRMS model that uses Modeling REST API to execute model

Modeling is necessary and commonly used in hydrologic research. Modification of the existing model simulations is a complex activity that hydrologists often have to deal with while analyzing complicated environmental scenarios. Modelers must make frequent modifications to underlying input files followed by lengthy re-runs. Programming languages could help significantly with this easily automated and repetitious task. However, it is complex and time consuming for hydrologists to write their own programs to handle file modifications for scenario based studies. To solve this challenge, the modeling module provides an intuitive user interface where users can create, upload, run and delete models. The UI informs users about the progress of the model run with a bar that displays a percentage of completion. The module also includes a dashboard where users can view the models being run, the finished model runs, and also download the model run files. A user can execute multiple PRMS models in parallel by uploading the three input resources needed for PRMS model. The upload interface as shown in Figure 6 is generated dynamically from the model schema defined for each of the models.

The modeling interface displays peak utility when users use it to programmatically run a bundle of models in parallel without having to worry about resource management. In the prototype system, the client is used to adjust input parameters for models and for model execution.

Model calibration can be a time consuming process for many hydrologists. It often requires that the modeler re-run and re-run a model many time with slightly varied input parameters. For many modelers this is a manual process. They will edit input files with a text editor and execute the model from a command line on their local machine.

Understanding that it was crucial to address this issue, the virtual watershed protoype system was developed with a web-based scenario design tool. This tool provides a handy interface which enables for the rapid adjustment and calibration of PRMS model input variables. It also provides the tools to execute the tweaked models via the modeling web service, and visually compare outputs from differently calibrated runs. Figure 7 shows the interface developed which provides visually oriented tools for data manipulation. This interface abstracts away technical model data representations and allows users to model in intuitive ways. Figure 8 shows the output visualization after a the modeling web service has finished executing the model run.

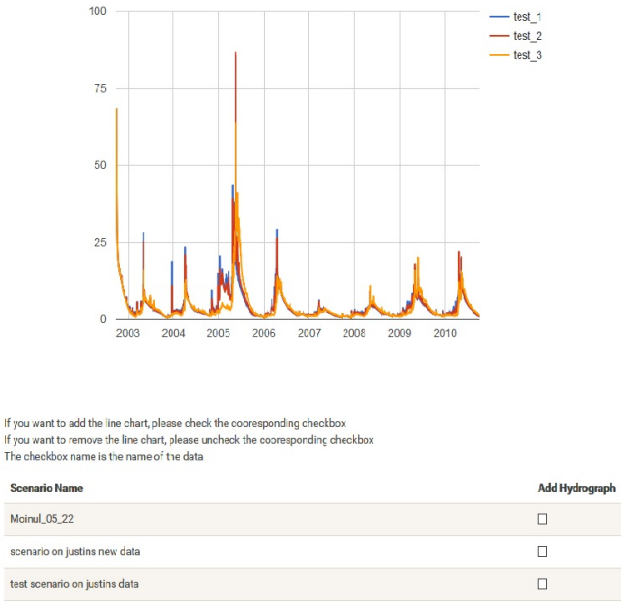

Figure 8: Comparing the results of the scenarios created using the modeling service

We developed multiple IPython [30] notebooks to demonstrate the programmatic approach to running an ensemble of models in parallel by using the modeling API by Matthew Tuner, who is a PhD Student

in Cognitive and Information Sciences at University of California, Merced. Figure 9, shows an important step of modeling with the PRMS model. By using the modeling API, an user can execute multiple models in parallel, programmatically. Users can use server resources to run a model many, many times without concerning themselves about limited time and CPU resources.

```
import itertools
fig, ax = plt.subplots()

cax = ax.matshow(nash_sutcliffe_mat, cmap='viridis')
tix = [1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0]
plt.xticks(range(10), tix)
plt.yticks(range(10), tix)


ax.xaxis.set_ticks_position('bottom')
plt.ylabel('jh_coef_hru factor')
plt.xlabel('rad_trncf factor')

for i, j in itertools.product(range(10), range(10)):
    plt.text(j, i, "%.3f" % nash_sutcliffe_mat[i, j],
             horizontalalignment="center",
             color="w" if nash_sutcliffe_mat[i, j] < .54 else "k")

plt.title('Nash-Sutcliffe Matrix')
plt.grid(b=False)
cbar = fig.colorbar(cax)
```

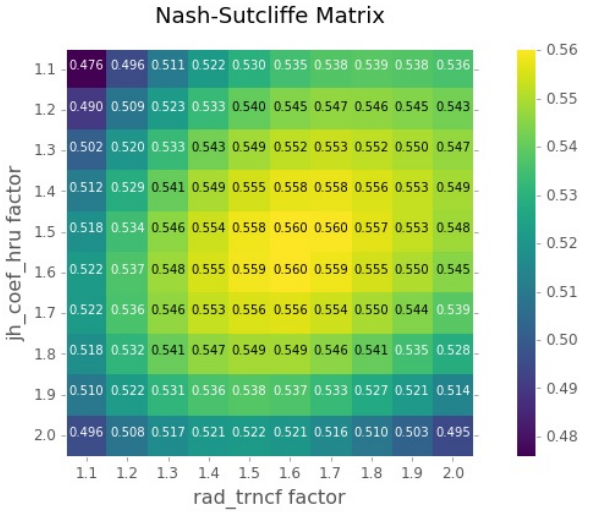

Figure 9: Example of tuning a hundred model through an IPython notebook with different parameters

5 Comparison with Related Tools

A feature based comparison is made between similar software tools focused on hydrologic and environmental modeling in Table 1. The tools discussed in Section 2 are CSDMS [8], Hydroshare [9] and MAAS [10]. Though each of these tools were designed and developed with different goals in mind, all of them are a demonstration of work to assist environmental modeling in general. Each of these tools has their pros and cons from different users perspectives.

CSDMS is a community driver system where modelers can submit their models to CSDMS by implementing model interfaces provided by them. The model gets evaluated and added into the system by the CSDMS committee. CSDMS uses a language interoperability tool called Babel to handle language heterogeneity. From the user's perspective, CSDMS provides a web and desktop based client where users access models and run them using different configurations from within the client. On the server side, CSDMS maintains an HPC cluster where they have all the models and relevant dependencies installed.

Hydroshare [9], on the other hand, is a project started to accommodate data sharing and modeling for hydrologists. The platform is in active development and frequently adds new features. The current version of Hydroshare is more concentrated towards data sharing and discovery for hydrology researchers [31]. Hydroshare also provides programmatic access to its REST API which makes it a good candidate for a data discovery backend for any other similar system.

The MaaS [10] is the most similar project to the Virtual Watershed modeling tool. The main advantage our work has over MaaS is resource management. Dr. Li et al. proposed MaaS, but they use Virtual Machine techniques in their framework. In our system, Docker containers are used, which require fewer resources. Also, we implemented APIs to check and stop a docker container based on model execution status. Existing container orchestration tools, like Docker Swarm, are limited in thier ability to manage containers in this way. Though the approach and implementation of this work differ in various ways, the end goal is similar to what we are trying to achieve. MaaS introduced models as services by creating an infrastructure using cloud platforms. The framework provides users with a web application to submit jobs. The backend that takes care of on-demand provisioning of virtual machines that containing model execution environment setup. MaaS achieves model registration by encapsulating a model as a virtual machine image in an image hub. It provides an FTP based database backend to store the model results.

6 Conclusion and Future Work

A hydrologic model execution web service platform named VWS is described in this paper. The key idea of the proposed platform is to expose model relevant services through python wrapper. This wrapper enables representation of model resources with a common data format (e.g., NetCDF), increasing inseparability. Also, it requires NetCDF format resources, which simplifies model execution on a server node. A user can login once and access all services of the VWS with the authentication/authorization component. Each VWS Component is wrapped in a docker container, which requires less resources than a traditional Virtual Machine. Docker container APIs, including the container termination API, are also implemented to autonomously manage the dynamic resource needs of this system.

This software has significant room to be expanded upon with furutre work. For example, the system currently only provides the option to integrate with

Table 1: Feature Comparison with Related Hydrologic and Environmental Modeling tools

Feature \ Tool	CSDMS	HYDRO-SHARE	MAAS	VW-MODEL
Open source	Yes	Yes	No	Yes
Provides REST API	No	Yes	No	Yes
Provides interface for model implementation	Yes	No	No	Yes
Allows model coupling	Yes	No	No	No
Allows model registration	Yes	No	Yes	Yes
Data storage	No	Yes	Yes	Yes
Data sharing and discovery	No	Yes	No	No

GSToRE data backend. This can be improved through implementation of a generic data backend which enables integration with other existing data providers like DataONE, Hydroshare, and CUHASI. This, in turn, could provide better access to data for modeling automation. Developing a pricing component for service usage would be considered for future development. This aspect can help system manager to sustainably maintain the server and hire software developers to implement other useful features.

Conflict of Interest The authors declare no conflict of interest.

Acknowledgment We would like to thank Matthew Turner for his sample python scripts described in Section 4 and all the reviewers for their suggestions.

This material is based upon work supported by the National Science Foundation under grant numbers IIA-1301726 and IIA-1329469.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Nmepscor.org. *The Western Consortium for Water Analysis, Visualization and Exploration — New Mexico EPSCoR*. [Accessed on 10 June 2018]. 2018. URL: https://www.nmepscor.org/science/%5C%5Cwestern-consortium-water-analysis%5C%5C-visualization-and-exploration.

[2] Md Moinul Hossain, Rui Wu, Jose T Painumkal, Mohamed Kettouch, Cristina Luca, Sergiu M Dascalu, and Frederick C Harris. "Web-service framework for environmental models". In: *Internet Technologies and Applications (ITA), 2017*. IEEE. 2017, pp. 104–109.

[3] Russ Rew and Glenn Davis. "NetCDF: an interface for scientific data access". In: *IEEE computer graphics and applications* 10.4 (1990), pp. 76–82.

[4] Danny Marks, James Domingo, Dave Susong, Tim Link, and David Garen. "A spatially distributed energy balance snowmelt model for application in mountain basins". In: *Hydrological processes* 13.12-13 (1999), pp. 1935–1959.

[5] Steven L Markstrom, Robert S Regan, Lauren E Hay, Roland J Viger, Richard M Webb, Robert A Payn, and Jacob H LaFontaine. *PRMS-IV, the precipitation-runoff modeling system, version 4*. Tech. rep. US Geological Survey, 2015.

[6] Chao Chen, Ajay Kalra, and Sajjad Ahmad. "Hydrologic responses to climate change using downscaled GCM data on a watershed scale". In: *Journal of Water and Climate Change* (2018).

[7] Chao Chen, Lynn Fenstermaker, Haroon Stephen, and Sajjad Ahmad. "Distributed Hydrological Modeling for a Snow Dominant Watershed Using a Precipitation and Runoff Modeling System". In: *World Environmental and Water Resources Congress 2015*. 2015, pp. 2527–2536.

[8] Scott D Peckham, Eric WH Hutton, and Boyana Norris. "A component-based approach to integrated modeling in the geosciences: The design of CSDMS". In: *Computers & Geosciences* 53 (2013), pp. 3–12.

[9] David G Tarboton, R Idaszak, JS Horsburgh, D Ames, JL Goodall, LE Band, V Merwade, A Couch, J Arrigo, RP Hooper, et al. "HydroShare: an online, collaborative environment for the sharing of hydrologic data and models". In: *AGU Fall Meeting Abstracts*. 2013.

[10] Zhenlong Li, Chaowei Yang, Qunying Huang, Kai Liu, Min Sun, and Jizhe Xia. "Building Model as a Service to support geosciences". In: *Computers, Environment and Urban Systems* 61 (2017), pp. 141–152.

[11] Michael P McGuire and Martin C Roberge. "The design of a collaborative social network for watershed science". In: *Geo-Informatics in Resource Management and Sustainable Ecosystem*. Springer, 2015, pp. 95–106.

[12] Eric Fritzinger, Sergiu M Dascalu, Daniel P Ames, Karl Benedict, Ivan Gibbs, Michael J McMahon, and Frederick C Harris. "The Demeter framework for model and data interoperabil-

ity". PhD thesis. International Environmental Modelling and Software Society (iEMSs), 2012.

[13] Jeffrey D Walker and Steven C Chapra. "A client-side web application for interactive environmental simulation modeling". In: *Environmental Modelling & Software* 55 (2014), pp. 49–60.

[14] Jon Wheeler. "Extending Data Curation Service Models for Academic Library and Institutional Repositories". In: *Association of College and Research Libraries* (2017).

[15] Thomas Erl. *Service-oriented architecture (SOA): concepts, technology, and design. Prentice Hall*. 2005.

[16] Roy T Fielding. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Doctoral dissertation, 2000.

[17] Alex Rodriguez. "Restful web services: The basics". In: *IBM developerWorks* 33 (2008).

[18] Leonard Richardson and Sam Ruby. *RESTful web services*. O'Reilly Media, Inc., 2008.

[19] Peter Buxmann, Thomas Hess, and Sonja Lehmann. "Software as a Service". In: *Wirtschaftsinformatik* 50.6 (2008), pp. 500–503.

[20] James Lewis and Martin Fowler. "Microservices: a definition of this new architectural term". In: *MartinFowler. com* 25 (2014).

[21] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 2015.

[22] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux Journal* 2014.239 (2014), p. 2.

[23] Md Moinul Hossain. "A Software Environment for Watershed Modeling". PhD thesis. University of Nevada, Reno, 2016.

[24] Rui Wu. "Environment for Large Data Processing and Visualization Using MongoDB". MA thesis. University of Nevada, Reno, 2015.

[25] Miguel Grinberg. *Flask web development: developing web applications with python*. O'Reilly Media, Inc., 2018.

[26] Rick Copeland. *Essential sqlalchemy. "* O'Reilly Media, Inc.", 2008.

[27] Bruce Momjian. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York, 2001.

[28] Ask Solem. *Celery: Distributed Task Queue.* [Accessedon 10 June 2018]. 2018. URL: http://www.%20celeryproject.%20org.

[29] Docker. *Docker Hub*. [Accessed on 10 June 2018]. 2018. URL: https://hub.docker.com/.

[30] Fernando Pérez and Brian E Granger. "IPython: a system for interactive scientific computing". In: *Computing in Science & Engineering* 9.3 (2007).

[31] Daniel P Ames, Jeffery S Horsburgh, Yang Cao, Jiří Kadlec, Timothy Whiteaker, and David Valentine. "HydroDesktop: Web services-based software for hydrologic data discovery, download, visualization, and analysis". In: *Environmental Modelling & Software* 37 (2012), pp. 146–156.