SciPaaS: A Scientific Execution Platform for the Cloud

Wesley H. Brewer[[1]](#footnote-1) , John C. Sanford[[2]](#footnote-2) , Will Scott

*Abstract*— SciPaaS is a prototype development of an execution platform/middleware designed to make it easy for scientists to rapidly deploy their scientific applications (apps) to the cloud. It provides all the necessary infrastructure for running typical IXP (Input-eXecute-Plot) style apps, including: a web interface, post-processing and plotting capabilities, job scheduling, real-time monitoring of running jobs, and even a file/case manager. In this paper, first the system architecture is described and then is demonstrated on an Amazon EC2 instance for two scientific applications: (1) a simple finite-difference solver of the inviscid Burger’s equation, and (2) *Mendel’s Accountant*—a forward-time population genetics simulation model. The implications of the prototype are discussed in terms of ease-of-use and deployment options, especially in cloud environments.

*Keywords*—web-based simulation (WBS), cloud computing, platform-as-a-service (PaaS), rapid application development (RAD), population genetics

# INTRODUCTION

S

cientific programmers very often invest enormous amounts of time and resources developing sophisticated computer programs for specific technical applications. Many of these programs are both brilliant and elegant, and are potentially of interest to a significant number of users beyond just the developers. Most of these programs do not have a broad enough potential user base to justify custom development of sophisticated, easily exported, plug-in, user-friendly input/output, turnkey systems. As a consequence, many wonderful programs are used for just a few years, by just a handful of people (the developers and their extended team), and then fall into disuse. The utilization of such programs could be greatly enhanced if only they were easily accessed, configured, and operated.

Unfortunately, most scientific programs have been cobbled together in a way that is not easily distributed, nor easily installed, nor easily configured. To make matters worse, there is usually a serious learning curve in terms of using a program. Lastly, many people who might like to use a given program are not themselves programmers – they may be, for example, biologists. For a biologist to understand how to use a program, and how to interpret the output, requires an extremely user-friendly user interface. All these problems combine to create an enormous barrier to full utilization of scientific programs. The learning curve is too long and user pain is too great. The cost of getting started is very often too high, and the benefits are often uncertain.

We outline here a generic system for the easy packaging, uploading, configuration, and utilization of any technical program that would otherwise not find application beyond the development team. We illustrate the utility of this system using the technical program called *Mendel’s Accountant* – an advanced numerical simulation of the dynamics of mutation accumulation within biological populations that are undergoing natural selection [1].

With the promise of on-demand computing access, cloud computing has become an invaluable resource for scientists. Cloud computing generally is divided into three separate tiers which each build upon the other as shown in Fig. 1: Infrastructure-as-a-Service (IaaS), Platform-as-a- Service (PaaS), and Software-as-a-Service (SaaS) [2]. IaaS providers, such as Amazon Elastic Compute Cloud (EC2) and Google Compute Engine, provide an infrastructure for running virtual machine servers in the cloud. PaaS providers such as Google App Engine (GAE), Heroku, or Amazon Elastic Beanstalk provides a more simplified way of hosting web applications in the cloud without having to setup the infrastructure. While PaaS makes migrating to the web simpler, PaaS services are more limiting than IaaS in that most of them only support a few languages (e.g. GAE only supports Python, Java, Go, and PHP). SaaS is software that is hosted by a PaaS, and these are may be also referred to as “web applications”.

While the cloud is a very versatile and powerful resource, most scientists have little knowledge about the cloud, and much less about how to build a Software-as-as-Service (SaaS). In general, web interface tools that require little complexity (or programming skill) have little capability (e.g. blog), while more powerful tools (e.g. a full stack web framework such as Ruby on Rails) have significant capabilities, but require significant programming skills. As illustrated in Fig. 2, Karger suggests that *domain tools* can be developed which provide high power while requiring low complexity [3]. In the domain of science, SciPaaS seeks to fill this void.

The concept of SciPaaS is that a scientist could easily create a zip archive of their code containing just the binary executable and a sample input file, upload it to the cloud, and SciPaaS would manage all the cloud infrastructure for them, including the input interface, job scheduling, plotting, etc. This allows the scientists to focus on developing software to solve the problem at hand, without having to worry about the added overhead of developing the infrastructure for the execution and interface environment.

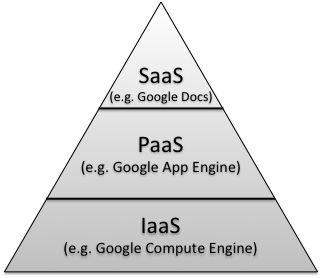


Fig 1 The cloud computing pyramid [2]

# Background

Some of the earliest efforts to do web-based simulations trace back to the 28th Winter Simulation Conference held in 1996 [4]-[6].

A thorough overview of developments of web-based simulations (WBS) and tools has been well documented by Byrne et al. [7]. In this review, the authors emphasize the development of web simulations in light of more recent prominence of technologies such as: Web 2.0 (including cloud computing), service-oriented architectures (SOA), and the Semantic Web. They mention numerous different types of communications protocols such as using WSDL (web-service definition language) or Java remote method invocation (RMI).

More recently, there has been a number of software packages developed over the past few years to address the need of being able to run scientific applications on a computer cluster. Wu et al. [8] developed a scientific application framework based on OpenSocial gadgets. Krishnan et al. [9] developed Opal2, a toolkit basically which can be used to wrap scientific applications and expose them as web services. Opal2 also provides plugin integration with EC2 and Hadoop. Opal2 provides much of the backend infrastructure for running applications, but relies on other software such as Kepler for pre-processing, and other codes for post-processing.

During the last couple years, there some new architectures and design methodologies have been proposed for cloud-based simulations. Hu et al. [10] compare four different modern methodologies (simulation model portability [SMP], MyExperiment, NanoHub, and RunMyCode), which promote reuse among common components in cloud-based simulation.

The concept of NanoHub, a scientific hub for web-based simulation for nanotechnology, is based on the HUBZero open source software platform, which uses a typical LAMP-stack (in this case Linux, Apache, MySQL, PHP) approach for the website and content-management system (CMS), while using a Java-based toolkit called Rappture (Rapid Application Infrastructure) to enable legacy scientific applications to run on the web [11].

Di Pierro [12] developed a python-based web framework called web2py. He uses web2py to show a sample scientific computing application in which stores DNA strings and searches for similarities. One of the powerful features of

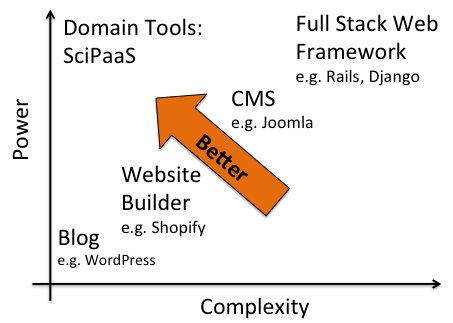


Fig 2 Spectrum of web interface capabilities [3]

web2py is that it uses in data access layer (DAL) such that many different types of database systems can be supported, including both relational and non-relational models. In fact, SciPaaS uses the DAL from web2py.

Liu et al. [13] provide a detailed architecture for Cloud-based Simulation (csim), where they define three key cloud services related to simulation in the cloud: Modeling as a Service (MaaS), Execution as a Service (EaaS), and Analysis as a Service (AaaS). Then they discuss more about more efficient ways of scheduling parallel and distributed applications (PADS) and then present four PADS job scheduling algorithms.

Although other methods exist for creating scientific hubs, they typically require much programming, knowledge and time to setup. To the authors knowledge there are still no available framework or middleware solutions that are dedicated to supporting scientific applications in such a way that: (1) a user can easily upload their program to the cloud, (2) have a user-friendly interface automatically generated for them to run, and (3) provide the infrastructure for all common tasks such as job scheduling, plotting, file/case management, etc.

# System Architecture

The concept for SciPaaS basically came out of developing a Scientific Software-as-a-Service (SaaS) interface for a simulation engine called Mendel’s Accountant [1]. Mendel’s Accountant is a biologically-realistic, forward-time population genetics simulator which can be used for studying mutation accumulation dynamics in genomes. During the development of this application a number of key ingredients required for general scientific applications were identified, which we have termed an IXP (Input-eXecute-Plot) pattern as shown in Fig. 1. The concept for SciPaaS resulted from identifying the common reusable components in many IXP style software systems, such as:

* Interface design
* User authentication
* Job scheduling
* Plotting system
* In-process monitoring
* Management of parallel jobs

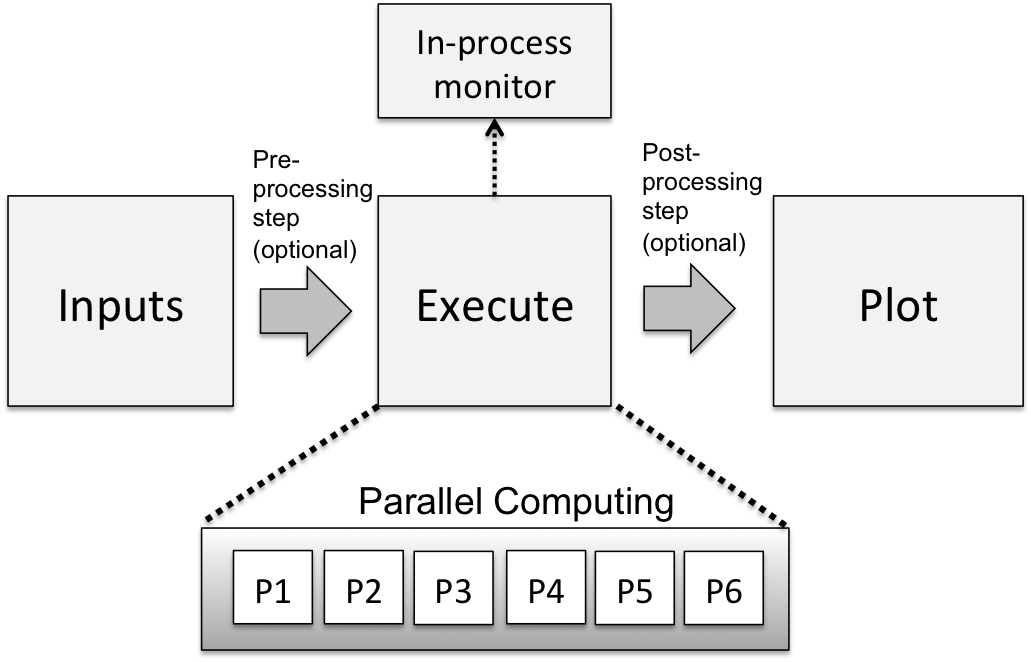


Fig 3 Many scientific applications fall under an Input-eXecute-Plot (IXP) design pattern.

During the development of Mendel’s Accountant, third-party software was used to handle each of these components. The problem with using third-party software was that it made it very difficult to setup the environment machine to run the simulation. For example, one of the necessary components, a job scheduler, was implemented using OpenPBS (currently rebranded as “Torque”). This one software alone can take quite some time to setup and also is non-trivial to manage.

Moreover, by considering a number of similar type software, some design goals were identified as follows. SciPaaS should:

* Automatically build a web interface
* Manage job execution, scheduling, and monitoring
* Monitor simulation as it is running
* Provide a plotting library interface
* Handle multiple users
* Provide a file/case manager
* Easily be deployed onto Amazon EC2, Google App Engine (GAE), Google Compute Engine (GCE), or RedHat OpenShift.

To meet these design goals, the following approach was used:

* Use a MVT python-based web framework
* Use a simple DB-based scheduler
* Deploy to free PaaS providers such as Heroku, OpenShift, Google App Engine (GAE), etc.
* Build to support standard scientific application design patterns

After considering a number of alternative languages, such as Java and Ruby, Python was chosen for three reasons: (1) it has one of the largest scientific computing communities, which includes scientific computing libraries such as SciPy (scipy.org), NumPy (numpy.org), and Matplotlib (matplotlib.org), (2) there are numerous open-source python-based web application frameworks available, and (3) because many of the cloud PaaS providers support Python-based applications (e.g. Google App Engine and Heroku). Furthermore, in the design of SciPaaS, a convention-over-configuration philosophy was implemented, such that a number of the files (i.e. the binary executable, the default input deck filename, the name of the view template, etc.) all use the name appname.

# Discussion

SciPaaS features are demonstrated for a typical scientific application called Mendel’s Account, an advanced numerical simulation program for modeling genetic changes over time. The simulation engine of Mendel’s Accountant was developed in Fortran 90 because of its ability to do numerical computation very efficiently.

## Upload app to cloud

A zipfile containing a default input file and binary of the application is uploaded to SciPaaS. The upload process unzips the file to the appropriate locations, reads the default input deck, and then creates an HTML template file views folder named the same as the application. The next section explains how the interface is generated from the input deck.

## Auto-interface generation.

SciPaaS can be used to automatically generate an HTML interface given an input deck as shown in Fig. 4. Currently three different types of standard input deck formats are supported: (1) *namelist* input decks which are typically used in Fortran 90 scientific applications (e.g. NCAR’s Weather Research and Forecasting [WRF] software), and (2) *INI* format which is a standard configuration file typically used in Windows applications, and (3) *xml* format commonly used in Java applications (among others).

While the namelist reader/writer had to be custom written for SciPaaS, the INI reader/writer makes use of Python’s built-in ConfigParser module. In Fig. 4, we show a portion of the Mendel input deck, and then the HTML template file that SciPaaS automatically generates.

Finally, there are some applications that will require a customized reader/writer. In these cases, the user can write their own plugin module with reader/writer methods for reading and writing their own customized input deck.

## Web Framework

The core of SciPaaS is based on a micro-web framework called Bottle (bottlepy.org). This was chosen over a full stack framework to keep the design simple with no external dependencies. Bottle uses a model-view-template architecture as shown in Fig. 5, combined with a separate scheduler, and data access layer, which are later discussed. The main purpose of the web framework is to map URL routes to Python methods, but it also provides a simple, yet powerful templating system. While Bottle is not a full stack framework, it is easily extended via many 3rd party plugins to provide almost any feature that a full stack support, e.g. object-relational mapping (ORM), session management, flash messages, etc.

## Database

SciPaaS stores all information about users, apps, jobs, and plots in a database. Gluino, a data access layer (DAL) ported

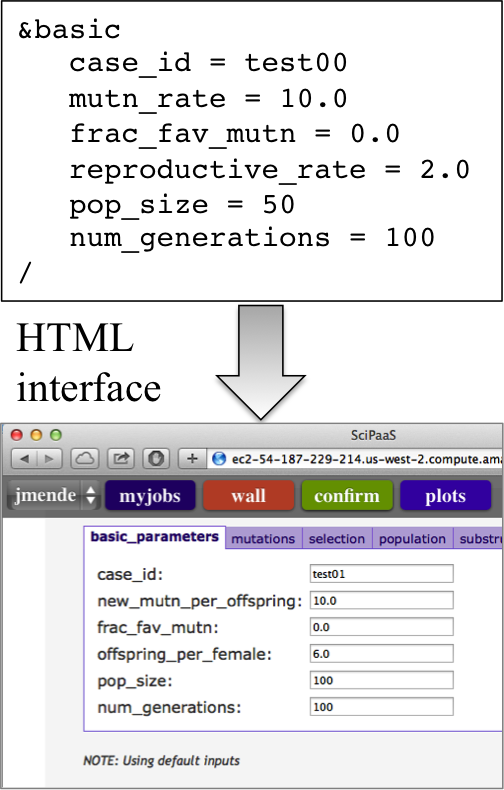


Fig 4 SciPaaS automatically converts input file to HTML form.

from the web2py web framework was implemented so that many different types of databases could be supported, including: SQLite, PostgreSQL, MySQL, Oracle, MSSQL, Google, Firebird, DB2, Informix, Ingres, Cubrid, Sybase, Teradata, SAPDB, MongoDB, and IMAP [14].

The database manages information about currently installed applications, users, and also information about plotting. Fig. 5 shows the general system architecture of the SciPaaS web application framework. Basically the model represents the interface to the database, and the views are essentially HTML templates rendered by Bottle’s template method in a simple way. For example, to render a plot, we can use a simple command such as:

return template('plot', params)

Here, plot refers to the plot.tpl file in the views folder and params contain a Python dictionary of some parameters about the app, the case to plot, and the user.

## Pre-processing

SciPaaS supports defining pre- and post-processing tasks. Pre-processing might be used for creating a non-standard or customized file that the app may need to for inputs. For example, one app that was tested required all program inputs to be specified on the command line, such as:

app.exe -x2 -s2 -n100 -v5 -r10 -k1 -i4 -j0.5

In order to use this app with SciPaaS, a standardized .ini file was created which interfaces well. Whenever, the user submits a case to run a pre-processing step is defined for this app which reads the .ini file and maps all the parameters to command line inputs.

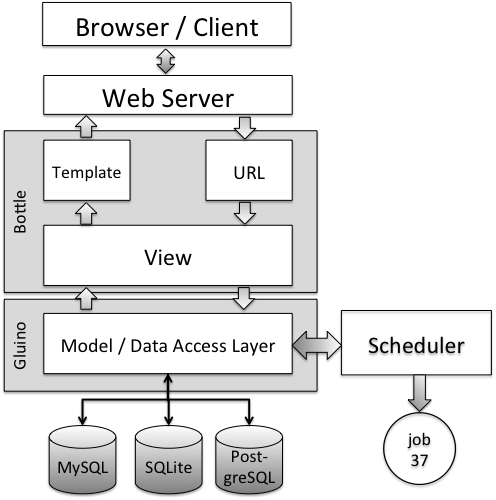
In addition to pre-processing, the user can also defined 

Fig. 5 SciPaaS Architecture

post-processing tasks. Post-processing would be typically used to transform data output by the simulation to the correct format needed to plot, or to compute a subset of data from the data generated by the simulation. An example was an app that output its data not in columnar format, but rather as:

x1 y1 x2 y2 x3 y3 etc.

The post-processing function was used to convert this format into a Python list, which can easily passed to a plotting library,

e.g.:

[[x1, y1], [x2, y2], [x3, y3], … ]

## Executing the Simulation Engine

There are several possible ways to spawn the simulation engine from within Python. One option is to use the subprocess module, which supports either a call method or a pipe. Another method for spawning the engine is simply to use the system call from the OS module. The important point is that the job must be launched in the background so that SciPaaS can continue to handle requests. The way this is handled is by spawning a new thread for every new job that is submitted. The standard output stdout of the simulation is redirected to the file *appname*.out (e.g. mendel.out in the current example).

## Case Management

Each simulation run is assigned a universal unique identifier (UUID) using Python’s built-in uuid module. It is important to have a way to identify each simulation run via SciPaaS for several reasons:

1. Identifiers are needed as a primary key in the jobs table.
2. Virtually every function related to cases and case data are referenced using this identifier.
3. The files and output generated from each run is stored under the relative path user\_data/appname/caseid (e.g. wes/mendel/c13dxg).

## Monitoring the Simulation

Once the simulation has been launched, SciPaaS automatically redirects to the monitor view. The monitor view is essentially a jQuery AJAX call which repeatedly calls a method called tail, which retrieves the last 40 lines of the output file every second.

## Plotting the Data

SciPaas offers two possibilities for plotting: (1) using a jQuery library called Flot, and also (2) using the Matplotlib library to generate static PNG images. The advantage of using a JavaScript or jQuery library is that all the plotting work is offloaded onto the client, rather than putting the burden on the server, and also the user can dynamically interact with the plot (e.g. zooming). The disadvantage of the JavaScript library is it supports only basic line, points, bar, and category charts, and was not specifically designed for scientific plotting (e.g. could not use it for a color contour plot). However, Matplotlib was specifically designed for scientific plotting, and supports many different chart types, including color contours. The primary disadvantage of using Matplotlib is that it requires installing about six additional third-party packages, whereas Flot does not require any additional software to be installed.

At any time during the simulation or after running, the user can plot data from the simulation. When the user clicks the plot button, the first defined plot will be displayed with options to show other possible plots. The user can then choose to plot one of the available plots, or can click the Edit button to define a new custom plot.

Flot is described as “a pure JavaScript plotting library for jQuery, with a focus on simple usage, attractive looks and interactive features” (flotcharts.org). In order to prepare the data for plotting, a plot class is defined within SciPaaS. This class contains a method called get\_data, which extracts the necessary data from the files, and returns it in JSON format.

## Job Scheduler

A multiprocessing, priority-based FCFS (first-come first-served) scheduler was developed to manage job submissions from the various apps. The jobs in the queue have three possible states: Q for waiting in queue and R for running or C for completed. Jobs are submitted to a jobs table in the database, which maintains state information about each job submission.

Users are assigned priority levels, with 0 having the highest priority. The admin user has a priority of 0, while regular users are given a priority of 1, and guest users are given a priority of 2. If several jobs are in the queued state, the job with the highest priority (or lowest number) is the first to be selected to run.

The scheduler has a separate polling thread, which repeatedly polls the database every second and starts executing any job that is in the front of the queue, which is in the queued state. Each job that is launched is run as a separate process using Python’s multiprocessing module. Administrators can specify the maximum number of

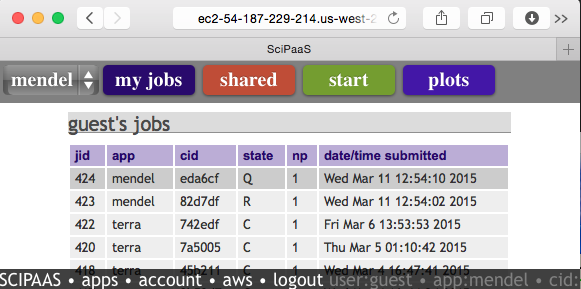


Fig 6 An example of the built-in job scheduler showing jobs in the queue, running and completed jobs

processors *np* that SciPaaS is permitted to use in the config.py file. A semaphore is initialized to this *np* value, when the number of submitted jobs is greater than *np*, the semaphore blocks the process from being launched, and waits until the other jobs finish. Also, a mutex is used to synchronize write access to the database. When a job finishes, the state is changed from R to C. Fig. 6 shows a sample output of the job scheduler.

## Shared Cases

Another feature SciPaaS offers is a way for users to share their results with other users. Since users have access to only their own cases, if they want to share results with others, they can write a comment in the jobs view and click “share”, after which anyone will be able to see their case, comments, outputs, and even be able to run the same case in their own filespace.

## User and Account Management

Users can register for an account. Passwords are hashed before storage in the database using a 256-bit Secure Hash Algorithm (SHA-256). A special account is reserved for the “admin” user, which allows special features, such as email updates whenever a new user registers.

## Worker Management

Integration with the Docker container system is included in SciPaaS in an effort to make deployment of as simple as possible. Docker is a platform and API for application deployment that is supported by most major cloud providers including Google, Microsoft, and Amazon (www.docker.com). When the application finds itself in an environment with a working Docker client, it provides an

interface to deploying new workers for scheduled jobs to run on. Additional interface is shown as part of the input screen, allowing the user to choose how many new worker contexts should be used. After use, workers are automatically stopped, making the deployment process simple enough for non-technical users.

# Example Applications

SciPaaS is exemplified using two different scientific software applications as follows:

1. **Burger** – a simple finite-difference solution to the inviscid Burger’s equation

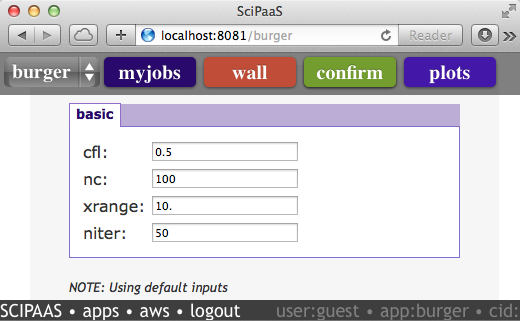


Fig 7 HTML parameter input for Burger's equation

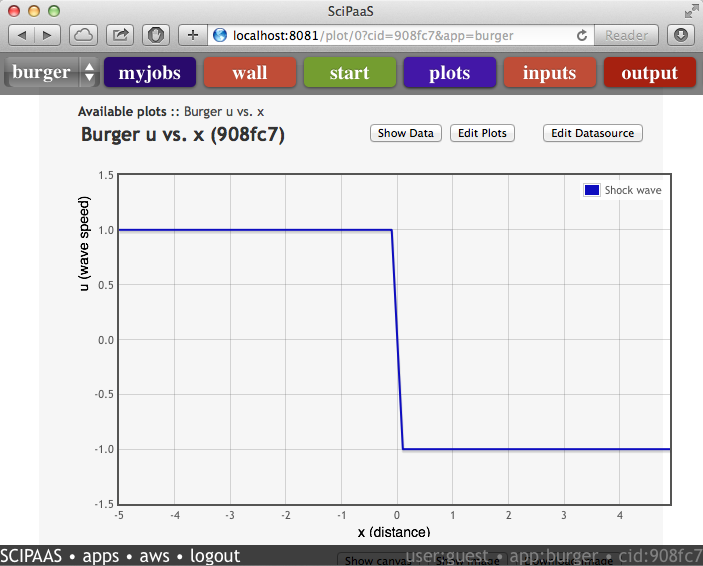


Fig 8 Burger’s output plot showing traveling shock wave

1. **Mendel’s Accountant** – a forward-time population genetics simulation program.

## Solving Burger’s Equation

Burger’s equation is a simple example of a set of nonlinear partial differential equation called reactive diffusion equations [15]. This equation is typically used in gas dynamics (e.g. to model shock waves) or to model traffic flow. The equation is given as:

(1)

where *u* is the speed of the traveling wave in the *x-*direction. This equation can be solved using the method of finite differences where each derivative is approximated to the first order using a simple finite-difference approximation as follows:

(2)

In this program, the user must enter a number of parameters, such as: the Courant number, the number of panels to discretize the domain, the length of the domain, and the number of iterations to run. This program would typically be executed in a UNIX terminal window, making it only accessible to a limited number of UNIX/Linux users around the world. By making the program available on the cloud, it instantly becomes accessible to virtually anyone on the planet.

Fortunately, with SciPaaS it only takes a few minutes to transform this console-based application into a web-based simulation. To create the app in SciPaaS, the user must do the following steps:

1. The program must read an input file with the parameters. If the input file is already in a standard format (such as namelist.input format, or .ini format) nothing additional must be done. If it is in a non-standard format, the user must either (a) modify the code so that it uses a standard
2. input format, or (b) write a custom reader in Python to read their specific input into SciPaaS.
3. Upload a Zip-compressed version of the program, with a sample user input file.
4. Define a Plot. This is simply a matter of clicking which file will be plotted, what columns in the file, and the plot type (e.g. scatter plot, bar chart, etc.).

SciPaaS automatically generates the HTML based on the input deck as shown in Fig. 7. The next step is to define any plots that should be post-processed. In this case, the plot shown in Fig. 8 represents the shock wave, resulting from solving Burger’s equation.

## B. Mendel’s Accountant

Mendel’s Accountant represents a much more complex example, a program which models genetic change over time. The software is part of a growing trend of many geneticists turning to computer simulation as a promising means to better understand population genetics [16]. Mendel’s Accountant is more complicated in two aspects: the simulation engine, and the client interface.

The simulation engine is more complicated because (1) it has more than 60 parameter inputs to the simulation, (2) it is parallelized using the MPI (Message-Passing Interface), (3) generates many output files full of statistical data which needs to be plotted.

*Pre-processing*. Even though Mendel’s Accountant is a quite complex simulation, since it already supports the standard namelist.input input deck, it can be uploaded and running in the cloud in just a matter of minutes. Fig. 9 shows the full input parameter screen.

*Processing*. Fig. 10 shows the console output of Mendel’s Accountant that is auto-updated via AJAX.

*Post-processing*. Once the run has finished and the data needs to be analyzed, clicking the plot button will given a list of pre-defined plots to plot, or can let the user define new plots. Data from multiple sources can be attached to each plot. Fig. 11 shows bar-plot of the distribution of accumulated deleterious mutations.

# Deployment

SciPaaS can be deployed in a number of ways:

1. Run SciPaaS on your personal computer to host a number of locally installed applications. This is a helpful option for development and unit or functional testing.
2. Run SciPaaS on a traditional Linux server.
3. Run SciPaaS on an Infrastructure-as-a-Service machine (e.g. EC2 instance on Amazon Web Services or Google Compute Engine, etc.). This would be the best option in that it could support binaries compiled in any language.
4. Run SciPaaS on top of an existing PaaS, (e.g. Google App Engine [GAE], AppScale, etc.). The limitation here would be that it would only be able to support app binaries that the PaaS supports. For example, running on GAE would require apps to be written in Java, Python, or
5. PHP. The authors have tested SciPaaS both on each of these options except the last option. We have recently developed a cloud test platform using a LEAP cloud stack (Linux, Eucalyptus, AppScale, and Python) in order to test running SciPaaS on top of a generic Platform-as-a-Service solution such as AppScale.

For the results for this paper, most of the testing was performed on an Amazon EC2 machine. Amazon offers a free trial use of a t1.micro EC2 machine. In the AWS page on SciPaas, the user may store their Amazon instances, and it provides three functions: start, stop, and status as shown in Fig 11. Actually, the user must click the status button, and then they can either start or stop the machine. It will also give the user the public DNS name of the machine. In order to utilize this approach, the user must setup the various machines they might want to use in the future, which would include installing SciPaaS on each machine. Then, the user can startup the big machine (e.g. an r3.8xlarge instance which has 32 vCPUs and 244GB of RAM, and also costs $2.80/hour to use such a machine at the datacenter in Northern California), use it for however long they need, and then download the case files to the t1.micro instance, and then shutdown the big, expensive machine. SciPaaS provides a “zip” button on the “myjobs” view, which allows the user to compress any case for download either to their home machine, or to another SciPaaS instance.

# Conclusion and Future Work

A middleware execution platform called SciPaaS was described and demonstrated with both a simple Burger’s equation solver, as well as Mendel’s Accountant, a more complex forward-time population genetics simulator. The software will soon be released in the Open Source domain online at https://bitbucket.org/whbrewer/scipaas.

By providing an automatically generated easy-to-use interface, and an easy way to upload and plugin their application to the platform, SciPaaS allows scientists to rapidly deploy their applications to the cloud.

An unintended benefit of this type of platform solution to web-based simulations is that it also *encourages* developers of scientific simulations towards using standard protocols where appropriate.

Currently, SciPaaS should work well with sophisticated simulation packages that do not require significant amount of pre- and post-processing tasks. Future development of SciPaaS should include making improvements to various aspects of the software to handle more robust simulations that may have substantial amounts of pre- and post-processing tasks involved, e.g. the Weather Research and Forecasting (WRF) software. In this case, a separate program must be run before the simulation and multiple programs may be run after the simulation for visualization and analysis.

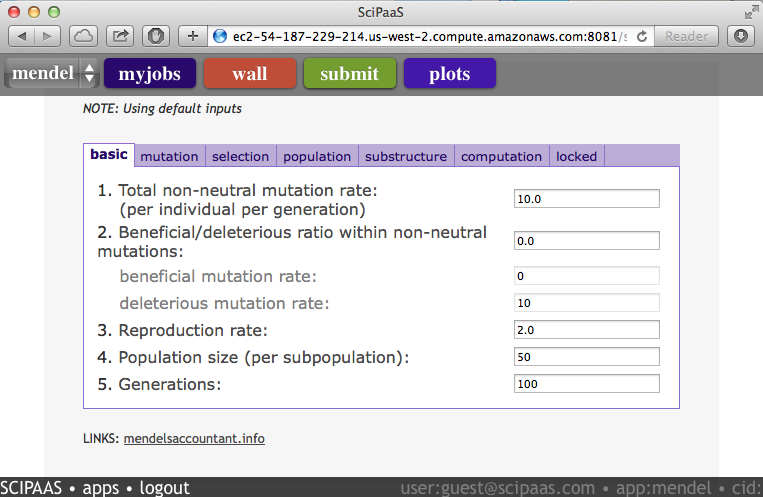


Fig 9 Input parameter screen for Mendel’s Accountant

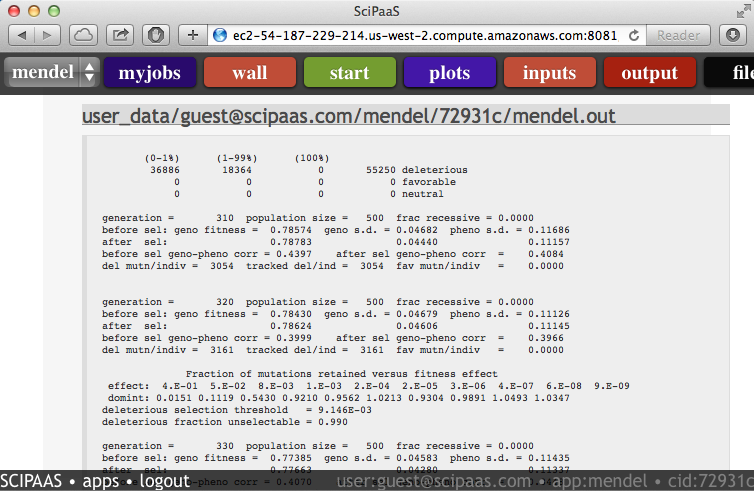


Fig 10 In-process monitoring of Mendel’s Accountant

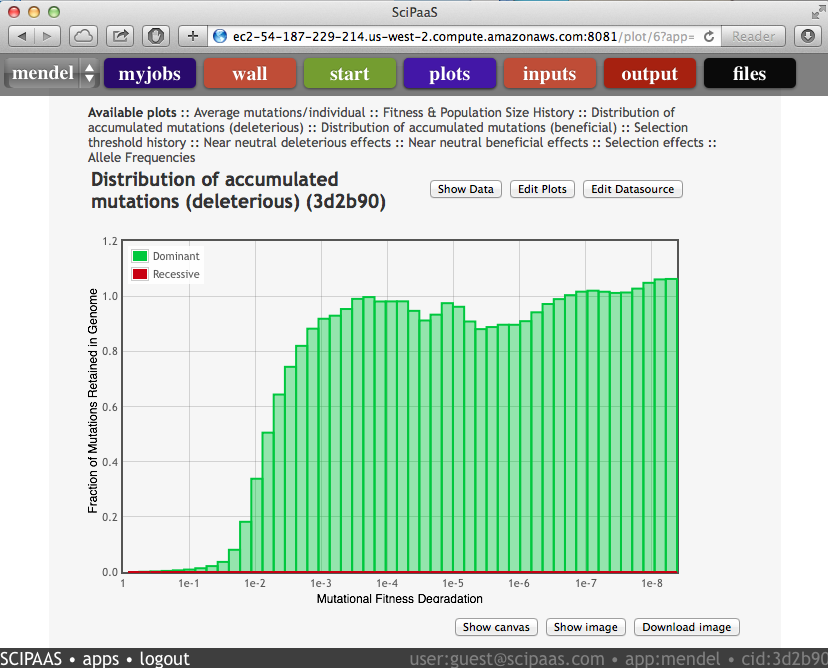


Fig 11 Sample output plot of Mendel’s Accountant showing a log-scale distribution of accumulated deleterious mutations

Acknowledgment

The authors want to acknowledge appreciation for the support graciously provided by both *FMS Foundation* and *GCC Jackson* organizations for this project. Also, many thanks to Tichomir Tenev for reviewing the manuscript

In addition, much future work remains, such as:

* Deployment to Google App Engine
* Implementing a generalized workflow to be able to handle custom-defined and more generalized workflows
* Interfacing with other web services, such as the Opal2 toolkit
* Supporting parallel execution infrastructure, such as interfacing with MPI and Hadoop.

References

1. Sanford, J., Baumgardner, J., Brewer, W., Gibson, P., & ReMine, W. (2007). Mendel's Accountant: A biologically realistic forward-time population genetics program. *Scalable Computing: Practice and Experience*, *8*(2).
2. Kepes, Ben, “Understanding the Cloud Computing Stack: SaaS, PaaS, IaaS”, Article 35 (2013), <http://www.rackspace.com/ knowledge\_center/whitepaper/understanding-the-cloud-computing- stack-saas-paas-iaas>.
3. Karger, David. “User Interfaces for Data: the Big Picture”, Lecture in *Tackling the Challenges of Big Data* online course, MIT Professional Education, https://mitprofessionalx.edx.org, 27 October 2014.
4. Fishwick, P.A. Web-Based Simulation: Some Personal Observations. *Proceedings of the 1996 Winter Simulation Conference*, Coronado, CA (1996): 772-779.
5. Nair, R. and Miller, J.A. A Java-Based Query Driven Simulation Environment. *Proceedings of the 1996 Winter Simulation Conference,* Coronado, CA 786-793.
6. Buss, A. and Stork, K. Discrete Event Simulation and World-Wide Web Using Java. *Proceedings of the 1996 Winter Simulation Conference*, Coronado, CA (1996): 772-779.
7. Byrne, James, Cathal Heavey, and Peter J. Byrne. "A review of Web-based simulation and supporting tools." *Simulation modeling practice and theory* 18.3 (2010): 253-276.
8. Wu, W., Uram, T., Wilde, M., Herald, M., and Papka, M. “A Web 2.0-Based Scientific Application Framework”, IEEE 2010.
9. Krishnan, S., Clementi, L., Ren, J., Papadopoulos, P., and Li, W. Design and Evaluation of Opal2: A Tookit for Scientific Software as a Service. 2010
10. Hu, C., Xu, C., Fan, G., Li, H., Song, D. “A Simulation Model Design Method for Cloud-Based Simulation Environment. Advances in Mechanical Engineering, <http://dx.doi.org/10.1155/2013/932684> (2013).
11. McLennan, M.; Kennell, R., "HUBzero: A Platform for Dissemination and Collaboration in Computational Science and Engineering," *Computing in Science and Engineering*, **12**(2):48-52 (2010).
12. Di Pierro, Massimo, “web2py for Scientific Applications.” *Computing in Science & Engineering* 13.2 (2011): 64-69.
13. Liu, X., Qiu, X., Chen, B., Huang, K. “Cloud-based Simulation: the State-of-the-art Computer Simulation Paradigm”, 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation (2012).
14. Di Pierro, Massimo, web2py Complete Reference Manual (6th edition), http://www.web2py.com/books/default/chapter/29/06/the-database-abstraction-layer.
15. Debnath, Lokenath. "Nonlinear partial differential equations for scientists and engineers." Birkhäuser (2012).
16. John C. Sanford and Chase W. Nelson (2012). The Next Step in Understanding Population Dynamics: Comprehensive Numerical Simulation, Studies in Population Genetics, Dr. M. Carmen Fusté (Ed.), ISBN: 978-953-51-0588-6, InTech, DOI: 10.5772/34047.

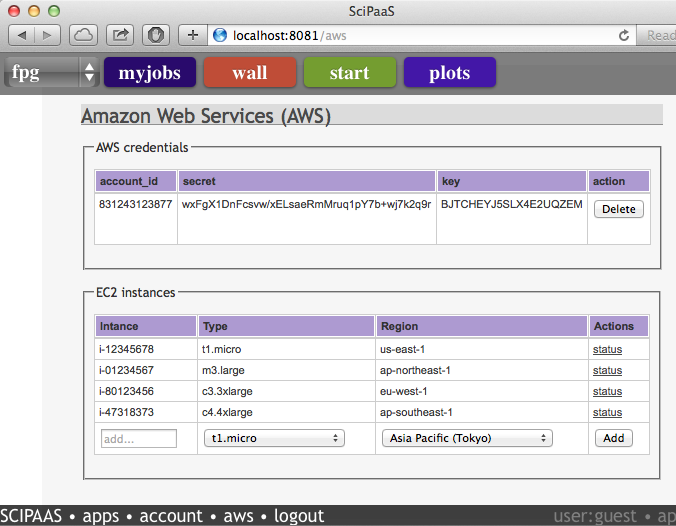


Fig 12 AWS configuration page showing how users can enter their AWS credentials and instances

1. W. H. Brewer is with Fluid Physics International, Houston, TX 77266 USA (phone: +1-713-893-8700; e-mail: wes@fluidphysics.com) [↑](#footnote-ref-1)
2. J. C. Sanford is with the School of Integrative Plant Science, NYSAES, Cornell University (e-mail: jcs21@cornell.edu)

   W. R. Scott is with the Department of Computer Science & Engineering, University of Washington (e-mail: wrs@cs.washington.edu) [↑](#footnote-ref-2)