Automatic and Portable Cloud Deployment for Scientific Simulations

Christopher B Horuk, Geoffrey Douglas, Anand Gupta,
Chandra Krintz, Ben Bales, Giovanni Bellesia,
Brian Drawert, Rich Wolski, and Linda Petzold
Dept. of Computer Science
University of California, Santa Barbara

Andreas Hellander
Dept. of Information Technology,
Division of Scientific Computing
Uppsala University

***Abstract*—In this paper, we present CLOUDRUNNER, a framework that extracts arbitrary programs from a source code repository (e.g. GitHub), wraps them in a web service and tasking system, and deploys them over disparate cloud infrastructures and local clusters, automating their portability. In particular, CLOUDRUNNER automatically creates and configures virtual machines so that they can execute the applications, provides a web UI with which users parameterize their applications, deploys instances of the program as background tasks, and collects the results for easy access via a browser. CLOUDRUNNER is an ideal framework for deploying scientific simulation applications portably and as such, we use it to implement StochSS – Stochastic Simulation as-a-service (now available at http://www.stochss.org). We use StochSS to evaluate CLOUDRUNNER overheads and find that they are small, consistent, and amortized for even short running applications.**

***Keywords*—Cloud Computing; Simulation Deployment;**

I. INTRODUCTION

Cloud computing is a highly customizable, pay-per-use, service-oriented methodology that offers many attractive features. Foremost, it simplifies the use of large-scale distributed systems through transparent and adaptive resource management, and automation for configuration and deployment of entire systems and applications. Cloud computing enables arbitrary users to employ potentially vast numbers of multicore cluster resources that are not necessarily owned, managed, or controlled by the users themselves. By reducing the barrier to entry on the use of such distributed systems, cloud technologies encourage innovation and implementation of applications and systems by a broad and diverse developer base – a base that might not otherwise have access to such resource scale.

To date, however, cloud computing has been used primarily to implement commercial information technologies and to support web services (multi-tier systems that encapsulate and integrate business logic, user presentation/interface, and a database engine). As a result, cloud systems (and the tools that facilitate their use) typically focus on support for the web service application domain. Users interested in other domains (e.g. scientific simulation and large scale data analysis) are left to devise their own toolsets. For clouds, such tooling requires that significant expertise, experience, time, and labor be devoted to the customization, configuration, deployment, and management of virtual machines (VMs).

In addition, for competitive reasons, clouds operated by different vendors support different application programming interfaces (APIs) for what amounts to the same (or a similar) set of services. Language bindings, scaling control, service level guarantees, pricing models, and usage policies vary from vendor to vendor although they are logically constituent components of every cloud. The sheer multitude of offerings and options make it challenging for new and expert software developers to determine which set of services is best for their applications, for some definition of "best" (e.g. price, performance, scale, configurability, familiarity, ease of use, other). Moreover, once users choose a cloud technology, code their program to its service interfaces, and configure it for that system, they become "locked in" to both the cloud fabric and to its service interfaces. This lock-in occurs because moving an application to a different provider, even to a similar service, requires significant developer porting effort (coding changes in the application). Developing expertise with each of today's vast offerings is simply untenable and distracts many scientists and data analysts from their investigations.

The goal of our work is to develop a distributed software platform that leverages successful cloud technologies and extends them to support automatic deployment and management of high-performance, compute-intensive scientific simulations. Toward this end, we have designed a toolset called CLOUDRUNNER that takes arbitrary programs and turns them into virtual machine "images" (*i.e.* executables) that workers (agents that execute tasks on behalf of a user) execute using different cloud infrastructures. Users employ the toolset to execute their applications in different settings without requiring that the users become experts in any one of them; CLOUDRUNNER takes only the user's credentials for the clouds she wishes to employ and automates the rest of the cloud deployment and execution process on which ever cloud target she has chosen.

We have designed CLOUDRUNNER for task-parallel workloads and in particular, those for scientific simulations in which potentially thousands of instances of the same program execute

(each with different inputs or seed values). The only constraints we place on CLOUDRUNNER applications is that their individual tasks execute on a single machine (i.e. independent, disconnected tasks with respect to communication), that they take their inputs from arguments or files passed to them, that they produce files as output, and that they execute over a Linux-based operating system.

Our work focuses on simplicity for this class of applications. Unlike past work [1], [2] which requires a new programming language for configuring tasks, we allow developers to implement their applications in any language/environment they wish and provide a portable graphical user interface (the CLOUDRUNNER "app") with which users provide their credentials and manage job deployment. CLOUDRUNNER *automatically constructs this UI* using cloud worker code (described by a configuration file) drawn from the user's code repository (e.g. GitHub). CLOUDRUNNER leverages existing scripting technologies that developers use today to configure their environments (e.g. GNU Bash, Puppet, Chef, and others) and to build their applications from source (e.g. GNU Make, Ant, Maven, and others). CLOUDRUNNER extracts and uses these artifacts from the repository and constructs (and tests) a virtual machine image for each of the deployment targets for which the user has supplied access credentials. CLOUDRUNNER configures this virtual machine with support for database and file storage, and task worker and queue services.

Once a virtual machine is available on one or more public or private cloud systems, the user employs the CLOUDRUNNER web app to specify the program inputs (arguments and files). The user can also optionally specify which cloud and the amount of resources (VM instances, memory, CPUs, and disk) to use to customize her job settings. CLOUDRUNNER then deploys the jobs across the resources in the selected cloud infrastructure, links each to their inputs, and monitors the jobs for termination (normal or exceptional). CLOUDRUNNER also collects and stores the output from the jobs (any files produced including logs if needed). Users access the output files via the user interface in the web app. Users can also execute jobs locally (on the machine that executes the CLOUDRUNNER UI), directly or via a local virtual machine instance. Developers use local execution to work out bugs and to identify inputs they wish to deploy in the large-scale cloud settings.

In this paper, we describe the overall design and implementation of CLOUDRUNNER. We employ only open source technologies for its implementation. In particular, we make use of a lightweight development server for local UI execution from Google App Engine, and connect it to Celery [3] task workers, a RabbitMQ [4] or public cloud task queuing system, an HTTP-based file server (object store), and a database.

To investigate the utility of CLOUDRUNNER for scientific simulation, we use it to implement StochSS – a stochastic simulation system as-a-service. StochSS is a CLOUDRUNNER-wrapped implementation of StochKit [5], an extensible C++ application that executes stochastic and multiscale algorithms

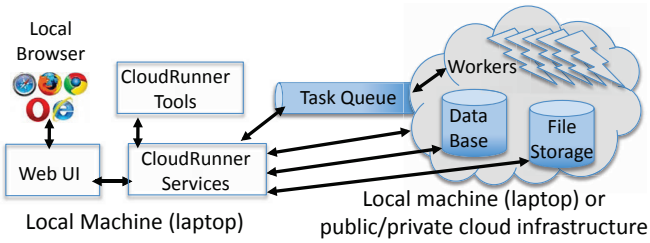

Figure 1. CLOUDRUNNER system overview.

for biochemical systems. We overview the StochSS use case and describe our customizations to the UI. Our current prototype deploys StochSS simulations locally and via Amazon Web Services automatically.

We use StochSS to measure end-to-end performance of CLOUDRUNNER. We find that CLOUDRUNNER introduces a small consistent overhead for cloud deployment (in particular, for task management) and no perceivable overhead for local execution. We also find that without considering task wait time (e.g. when there are more tasks than workers), the overhead for task management is amortized by application execution time for applications that execute longer than one minute.

In the sections that follow, we present the design and implementation of CLOUDRUNNER. Following this, we describe our experience using CLOUDRUNNER to implement StochSS. We then discuss our future and on-going work and conclude.

II. CLOUDRUNNER

CLOUDRUNNER is a toolset that provides its users with the ability to run arbitrary programs on cloud resources without worrying about the specifics of the cloud infrastructure. The entire process, from installing dependencies to running the program is managed by CLOUDRUNNER. Figure 1 overviews our CLOUDRUNNER design. Developers use the CLOUDRUNNER tools to construct automatically an execution engine for their program. The engine abstractly consists of a set of virtual resources through which CLOUDRUNNER achieves portability. The virtual resources enable task execution via *Workers*, task management via a *Task Queue*, *File Storage* for task output, and a *Database* for storing task state and CLOUDRUNNER metadata. As part of CLOUDRUNNER's construction of the execution engine, it also generates a portable web application with which users parameterize and manage their tasks (running instances of their program(s)). The CLOUDRUNNER web app consists of a Web UI and a set of backend services that interface to virtual resources through a set of application programming interfaces (APIs).

Figure 2 depicts the CLOUDRUNNER services, virtual resources, and the primary APIs CLOUDRUNNER uses to access the resources. The Login service provides user authentication, the Job Manager starts tasks, tracks their progress, and collects their output, and the Cloud Manager interfaces to one or more public or private cloud technologies to implement CLOUDRUNNER. We implement the virtual resources with

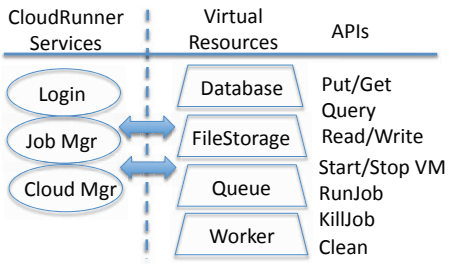

Figure 2. CLOUDRUNNER Backend Services

one or more different implementations located on the local machine or in the cloud. This modular design facilitates integration of multiple disparate deployment targets and thus portability of user programs across cloud infrastructures (and local machines).

To implement CLOUDRUNNER, we use the Python programming language for portability. In addition, we build upon and extend a number of extant open source technologies and make CLOUDRUNNER available as open source. In the subsections that follow, we provide more details on each of the primary components of CLOUDRUNNER.

A. Deploying and Building Cloud-Enabled Programs with CLOUDRUNNER

We refer to the process of adding new programs to CLOUDRUNNER, i.e. encapsulating programs in a web service and integrating them as part of a background task execution engine, as "service-izing" the programs. To enable this, the CLOUDRUNNER tool CRBuild takes a source code repository URL and downloads, configures, and installs one or more programs. CRBuild does so directly if local execution is desired, and within a virtual machine if remote (cloud-based) execution is desired. To enable this, CLOUDRUNNER relies on a set of named files that invoke standard and popular technologies for installing software to be present in the source code repository. In particular, it expects an executable file called CRBuild.sh (a GNU Bash script) that invokes the installation process for the program, and an executable file called CRTest.sh (a GNU Bash script) that executes a test input on the program. In addition, CLOUDRUNNER expects a text file in the YAML format [6], called CR.yaml, for each program to be service-ized that specifies each of the parameters the program takes as input. If there is more than one program, name expects the CR filename to be suffixed with integers in increasing order, e.g. CR1.yaml and CR2.yaml. Figure 3 depicts a CLOUDRUNNER-built VM configured for a service-ized program. For local execution, CLOUDRUNNER skips the VM creation step and attempts to build and test the program locally.

In our experience, CRBuild.sh typically calls a script that performs the configuration and installation of the program (e.g. INSTALL or configure; make; make install) that developers currently provide. Thus, the CLOUDRUNNER additions are the test script that sets up the environment and invokes the program with representative arguments and the YAML file that describes the programs arguments in a well-formed

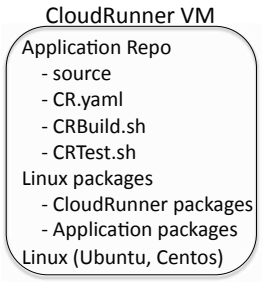

Figure 3. A example of a CLOUDRUNNER VM.

manner. Our current CLOUDRUNNER prototype expects that the developer is able to build and run their program in a Linux distribution (CentOS or Ubuntu is currently supported) and in a modern Mac OS X system. Once scripts are available that automate this process, developers simply wrap them in the appropriate CLOUDRUNNER programs to facilitate service-ization.

CRBuild automatically constructs a virtual machine for each cloud infrastructure on which the user is ultimately interested in executing. CRBuild uses the interface for each of the infrastructures to do so. Our current prototype supports Amazon EC2 [7] and Eucalyptus [8] (an on-premise private cloud infrastructure that is API compatible with Amazon EC2). We are currently working on extending our prototype to other cloud infrastructures (Microsoft Windows Azure and Rackspace); doing so is straightforward as CLOUDRUNNER only requires APIs for starting, stopping, and querying the status on VM instances for which each provider provides a similar API.

The CR.yaml file describes the program's command line parameters, executable name, and how parameters are passed in. CLOUDRUNNER currently expects a minimum set of keys (i.e. key-value pairs) in this file. The first required key is "executable" and the value of this key should be the name of the program's executable. The next required keys are closely related, "parameter_prefix" and "parameter_suffix" and their values represent strings used to prefix and suffix command line parameter key names, respectively. The parameter suffix is only used for file parameters, i.e. parameters that take a file name as an argument, and value parameters, i.e. all other parameters that take an argument. The user is responsible for keeping any user-modified web forms in the CLOUDRUNNER UI synchronized with the `CR[?].yaml` file(s) if they change.

The last two required keys are "required_file_parameters" and "required_value_parameters". The values of both of these keys are lists of the actual parameter names, which can be left empty if the program has no required file or value parameters. The file parameters actually need to be specified as a list of parameter name and input file extension pairs (represented by lists of size 2). If the parameter accepts file names with arbitrary extensions (i.e. text files) then the file extension can be specified as a blank string. The CLOUDRUNNER toolset also currently supports an optional version of both of these keys, i.e. "optional_file_parameters" and "op-

tional_value_parameters", as well as a third optional key called "optional_boolean_parameters". The value of this third optional key, if included, should be a list of parameter names for boolean parameters, i.e. parameters whose inclusion/exclusion signify their value. A sample YAML program description file is included below for a program with only 4 accepted command line parameters, 3 required value parameters and 1 optional boolean parameter.

```
executable: do_something
parameter_prefix: --
parameter_suffix: =
required_file_parameters: []
required_value_parameters:
  - value1
  - value2
  - value3
optional_boolean_parameters:
  - skip-nonsense
```

CRTest.sh is a script that sets up any necessary environment variables and executes each of the programs for which there are CR.yaml files. The script must exit with a zero value upon successfully executing each of the programs. CLOUDRUNNER considers any non-zero exit value as an error which it reports back to the user and terminates the CRBuild process (after cleaning up all resources it has allocated including virtual machines as necessary). The user is responsible for including all of the necessary executables and arguments in the test to ensure that CLOUDRUNNER can execute the application without error as cloud-based tasks. To help users with this process, we provide them with a VirtualBox VM (Debian-based and RedHat-based Linux distributions) with which they can locally test their build and test scripts directly to validate correctness. The CLOUDRUNNER prototype currently supports git-based source code repositories (e.g. GitHub, Bitbucket, local git repositories). CLOUDRUNNER uses the standard `git clone` command on the HTTPS repository name that the user passes to the CLOUDRUNNER tool CRBuild (i.e. USAGE: `CRBuild https://github.com/myaccount/myrepo.git`).

B. *The* CLOUDRUNNER *Login Service*

The login service provided by CLOUDRUNNER is typical of that employed in modern multi-user web services today. The service allows for both administrator (admin) and regular users to co-exist and provides admin users with the ability to approve each new user before they are allowed to create an account in the system. Prospective users are able to request an account from the admin, who then can approve or reject the request. CLOUDRUNNER provides a login service to securely manage per-user credentials for different cloud infrastructures. Past work [1] does not support security and multi-user capabilities.

To implement the login service, we augment a basic open source login interface (described below as part of the CLOUDRUNNER frontend) with SSL support to facilitate secure login and production use in multi-user settings. To enable this, CLOUDRUNNER automatically launches an Nginx server to serve requests to the CLOUDRUNNER web app. The server is configured for SSL-only access and is used by CLOUDRUNNER as a reverse proxy for the app. CLOUDRUNNER automatically configures and deploys this server as part of its frontend.

To bootstrap the authentication system with the administrator account, a CLOUDRUNNER tool initiates a handshake as part of CLOUDRUNNER installation and initial deployment. This process creates an admin user using a secret token that it stores in the database resource. Once the secret token is injected, the tools return the value of the secret token back to the application which is responsible for displaying it to the user. Using this token, the admin can create her account and log into the system. Upon doing so, the token is invalidated. The admin user has the ability to approve and deny user account requests, explicitly whitelist approved users by their email address and delete current users. The admin is treated as a trusted entity within the system and is also allowed to reset the password of any user to a random string of digits, characters and special characters generated automatically. This tool uses the database resource to store information about each user of the system.

C. *The* CLOUDRUNNER *Job Manager*

CLOUDRUNNER's job manager controls the task queue and worker resources. In particular, the job manager is responsible for executing jobs locally and remotely, querying their status, killing jobs, and cleaning up job artifacts. The CLOUDRUNNER UI interacts with the job manager via the RunJob, KillJob, and Clean APIs. The UI is responsible for indicating whether a user wishes that his job be executed locally or remotely, and with which program arguments (or files). The job manager uses this information to construct and deploy a task-based implementation of the program accordingly. The job manager periodically checks the status of the tasks it deployed and updates the status in the database resource. When a job terminates (normally or exceptionally), the job manager collects the output files and makes them available for user access via a link on a page in the CLOUDRUNNER UI.

The job manager relies on two internal CLOUDRUNNER virtual resources for executing tasks: the task queue and task workers. The task queue is a fault-tolerant and distributed message passing service that facilitates execution of tasks by workers (execution engines). The configuration and setup of this queuing system is handled entirely by CLOUDRUNNER and requires no input from the user. CLOUDRUNNER invokes one or more workers per VM depending upon the resources that the VM has at its disposal (number of CPUs and amount of memory). This is customizable and defaults to one worker per CPU. If a worker crashes while it is executing a task, the task is automatically made available to another worker by the queuing service.

CLOUDRUNNER employs task workers locally and remotely using slightly different semantics. Remote task execution

governs how the workers execute the programs on different cloud infrastructures and store their results. Every task has a unique identifier and an associated status, which is updated by the worker at specific, pre-defined intervals throughout its execution. Depending on the actual value of the tasks status, the task may additionally have some relevant metadata such as the location of the output results or the reason for failure. All of this information about a task is stored in the database resource using the tasks unique identifier as the primary access key. For remote, cloud-based execution, we implement the task queue resource using RabbitMQ and Amazon Simple Queuing Service (SQS); for task workers we integrate Celery. Celery is a scalable, open source interface to a wide range of queue services (including RabbitMQ and SQS) that implements asynchronous dequeuing and execution of tasks. CLOUDRUNNER's cloud tasks store their results locally and upload them to the cloud file system (virtual resource) upon termination. We implement CLOUDRUNNER's cloud file system using Amazon Simple Storage Service (S3). The job manager provides links to the URIs of these files in the CLOUDRUNNER UI when available.

CLOUDRUNNER's local execution spawns processes for task execution on the same physical machine on which the CLOUDRUNNER app is hosted. That is, CLOUDRUNNER bypasses the use of a queue service and implements workers using operating system processes for simplicity (since the local machine is likely to be resource-limited relative to the cloud). The job manager checks the status of a local task by querying its process identifier on the local operating system. CLOUDRUNNER assumes that the local machine employs a Linux-based operating system (Mac OSx included). For Windows users, we provide a virtual machine for the CLOUDRUNNER app that the users can instantiate locally (e.g. using the freely available virtualization system VirtualBox). Local execution allows users to run and debug their applications directly without requiring Internet access and prior to using cloud resources. Local tasks store their output files to the local file system which the job manager provides links to within the frontend UI upon termination.

D. The CLOUDRUNNER Cloud Manager

The final backend service provided by CLOUDRUNNER is the cloud manager. The cloud manager is responsible for interfacing with all of the supported cloud infrastructures and abstracting away all of their differences. The CLOUDRUNNER UI is responsible for transmitting cloud credentials to the cloud manager and for calling the StartVM/StopVM APIs if optionally selected by the user. The user need not manage cloud VMs manually as CLOUDRUNNER will automatically start and stop VMs when jobs are deployed. The user can specify the maximum number of VMs to place a limit on VM instance use. If a user chooses to terminate a VM that has running jobs in it, the jobs are terminated as well. Users also have the option of specifying the instance types to use in the cloud; this dictates how much memory and the number of CPU cores and disk space tasks are given.

As part of the CLOUDRUNNER VM creation process, we also configure timers that identify when a VM instance has been idle for more than ten minutes (a configurable value). When a VM instance is executing within a cloud infrastructure, the cloud manager terminates the VM to keep costs down. Moreover, the cloud manager is able to do so at the pricing granularity of the public cloud so that it keeps VM instances up (warm) for use in task execution, but terminates idle instances before they enter the next billing increment (e.g. hour boundaries in Amazon EC2). In this way CLOUDRUNNER reduces the cost of using the public cloud when possible without involving the user.

E. The CLOUDRUNNER Web-based User Interface

The CLOUDRUNNER front end is a web-based graphical user interface (GUI). CLOUDRUNNER automatically generates a basic implementation of this UI that the user can customize and build from. CLOUDRUNNER constructs web pages for user login, cloud credentials, job input specification and deployment, and job management (and output acquisition). CLOUDRUNNER links these web pages to the CLOUDRUNNER backend services as part of a simple web application (app).

To implement this web app, CLOUDRUNNER builds upon and extends the Google App Engine software development kit (SDK) [9]. The SDK is an open source web server that provides interfaces for basic web application functionality (login, database access, request handling). To construct CLOUDRUNNER, we extend this SDK with new implementations for these basic service APIs to form the CLOUDRUNNER backend services and APIs. In particular, we extend user authentication with SSL support (login service), we extend the database API to use a cloud database service (instead of a flat file) for remote task management by the job manager, and we extend the application runtime (sandbox) with new features that facilitate local file I/O, a wider range of Python libraries, and execution (forking of subprocesses) of locally deployed background tasks by the job manager. Finally, we added support for interacting with a remote task queue by the cloud manager.

F. Design Choices

Our CLOUDRUNNER design is based on two guiding principles. The first is the decoupling of the UI from the backend execution. All of the components of CLOUDRUNNER are provided as individual services each with its own set of APIs. Users of CLOUDRUNNER can take these individual services and use them to build their own custom interfaces to encapsulate the underlying feature provided by the service.

The second is specific to the types of programs that can be executed. CLOUDRUNNER restricts the programs it runs to those that are independent with regard to their communication and interoperation and that execute using a single (virtual) machine. This characteristic is common for scientific simulations

that are run multiple times with a number of different parameters or random seeds (our target application domain). A major benefit of this requirement of communication independence is that a single invocation of a program, i.e. a task, represents a single unit of parallelizable work. The parallel computing aspect comes from ability of CLOUDRUNNER to run as many tasks as the user requests, with the only limit on the number of concurrent tasks being the number of cores in worker virtual machines that CLOUDRUNNER is allowed to launch.

III. CLOUDRUNNER USE CASE: STOCHSS

We next employ CLOUDRUNNER for a scientific simulation application called StochKit. StochKit is a widely used execution engine, written in C++, for stochastic and multiscale algorithms that simulate biochemical systems. In this section, we show how we use CLOUDRUNNER to implement *StochSS – Stochastic Simulation as-a-Service*.

A. "Service-izing" StochKit

We first augment the StochKit git repository (in GitHub) with the structure and files that CLOUDRUNNER expects. In particular, we have a StochKit directory under which there are files `CR.yaml`, `CRBuild.sh`, and `CRTest.sh`. `CRBuild.sh` installs the libxml2 development library (checking first which distribution it is installing into). The script then executes *configure* (StochKit uses the Linux autoconf utility), *make*, and *make install*. To construct `CRBuild.sh`, we create it and call an extant StochKit build script called `install.sh`.

`CRTest.sh` is a bash script that executes the StochKit application with arguments from the command line. We include each executable that we are service-izing in the script:

```
export STOCHKIT_HOME=$PWD/StochKit
export STH=$STOCHKIT_HOME
mkdir -p /tmp/StochKit
$STH/ssa -m $STH/models
  /examples/dimer_decay.xml -r 1 -t 1 -i 1
  --out-dir /tmp/StochKit --force
$STH/tau_leaping -m
  $STH/models/examples/
  dimer_decay.xml -r 1 -t 1 -i 1
  --out-dir /tmp/StochKit --force
rm -rf /tmp/StochKit
```

The final files that we add to the repository are `CR1.yaml` and `CR2.yaml` because there are two executables to service-ize (ssa and tau_leaping as shown in the test script above). In CR1.yaml, we specify each of the parameters that the application expects for the ssa executable. This file identifies the executable name, the parameter prefix and suffix characters, and the names of the required and optional parameters. CLOUDRUNNER supports file argument, boolean argument, and value (all other) argument types. Our `CR1.yaml` file has the following contents (we omit CR2.yaml for brevity):

```
executable: ssa
parameter_prefix: --
parameter_suffix: ' '
required_file_parameters:
  - [model, xml]
required_value_parameters:
  - time
  - realizations
optional_value_parameters:
  - intervals
  - bins
  - species
  - out-dir
  - seed
  - processes
  - epsilon
  - threshold
optional_boolean_parameters:
  - no-stats
  - keep-trajectories
  - keep-histograms
  - label
  - force
  - no-recompile
```

CLOUDRUNNER uses these files to generate a web form that facilitates user entry for each of the parameters. The form shows each of the required parameters and hides the optional parameters (which can be shown/hidden via button toggle); at the bottom CLOUDRUNNER inserts a `Run Locally` and a `Run in Cloud` button. Users can customize this form to improve esthetics and user experience.

CLOUDRUNNER integrates this web form into its application (app) which provides support for user authentication and login, cloud credential handling and VM management, and task deployment locally and over different cloud infrastructures. The webform is generated via the `--generate-UI` flag to CRBuild. CLOUDRUNNER does not generate it by default to avoid overwriting a previously generated and possibly customized version.

The CRBuild script creates a VM (and credentials page) for each of the cloud infrastructures where a program has been built successfully. Users identify the infrastructures using the `--infra` parameter to CRBuild. This parameter takes one or more, comma delimited infrastructure names. In our case, the complete call to CRBuild is as follows. We then update the web form to provide a customized UI for StochSS.

```
CRBuild --infra=aws --generate-UI
    https://github.com/user/project.git
```

B. Running StochKit Simulations

To use the CLOUDRUNNER app, users simply invoke the CRRun tool. This tools launches the CLOUDRUNNER app and directs a browser window to its server (now running

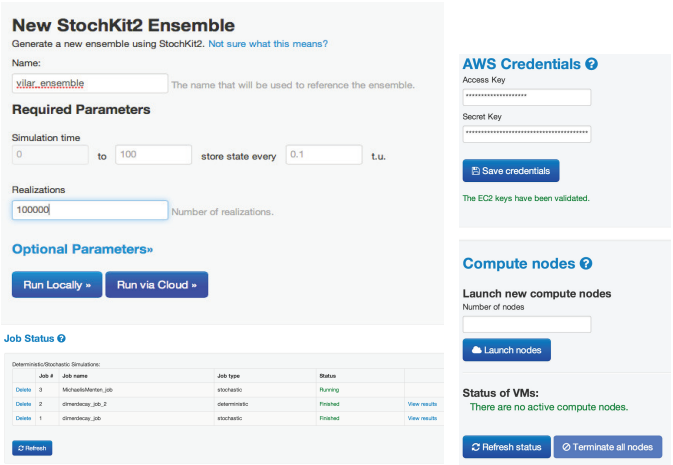

Figure 4. StochSS Web UI constructed using CLOUDRUNNER.

locally). A screen shot of the customized StochSS web form is shown in Figure 4 (top left). The other pages provided by the CLOUDRUNNER app are those for login, for specifying cloud credentials and managing VMs, and for job management. We include the latter two in the figure (bottom left and right, respectively). To invoke a job, the user logs in, navigates to the application's web form and enters arguments for each of the parameters of interest. The user clicks the `Run Locally` button to execute the job locally (to test that CLOUDRUNNER is working correctly). The job status, output, and any error files are accessible via the Job Management page. Users click the job when finished to access the output files.

To run jobs in the cloud, the user navigates to the Cloud page and enters her credentials. She can also start VMs here to avoid VM spin-up time upon initial job submission. The user specifies the program arguments via the web form and presses the `Run in Cloud` button. CLOUDRUNNER then spawns a worker VM if none are running and submits the job to the message broker. A worker in the system retrieves the job from the message broker, executes the task, copies any output files to cloud storage, and contacts the message broker for completion. The user can terminate worker VMs via the Cloud page or wait for CLOUDRUNNER to terminate them automatically (after an idle period of 10 minutes).

IV. RESULTS

We next empirically evaluate CLOUDRUNNER using the StochSS prototype. For local execution, we use a 2011 iMac with 12GB 1333MHz DDR3 memory and a 2.7GHz quad core Intel Core i5 processor. For cloud execution, we use Amazon Web Services (AWS). For AWS, we employ Elastic Compute Cloud (EC2) instance size c3.large, the Simple Storage Service (S3) for cloud storage, and the Simple Queue System (SQS) for the message broker CLOUDRUNNER implementations.

We first present the time for the various stages of CLOUDRUNNER deployment (compilation, installation of dependencies, repository cloning, and image creation). These times are specific to StochKit but provide insight into where

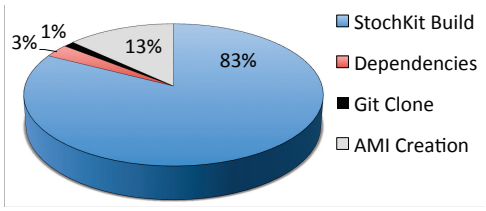

Figure 5. StochSS end-to-end deployment time for StochKit

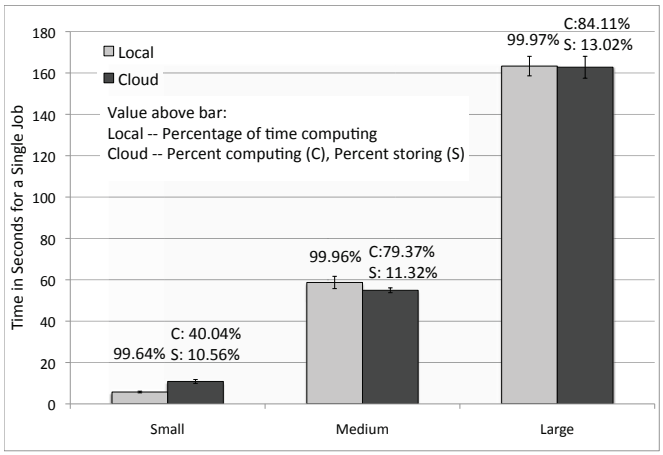

Figure 6. CloudRunner average execution time (in seconds) with 95% confidence for different StochSS job sizes.

time is spent in this process. The StochKit compilation (build) time consumes a majority of time (9.5 minutes). The second largest consumer is image creation in AWS (2 minutes). CLOUDRUNNER introduces no additional overhead that we were able to measure. The percentage breakdown for an end-to-end deployment us shown in Figure 5.

We next consider end-to-end execution time when using CLOUDRUNNER for local and cloud job deployments. End-to-end time consists of CLOUDRUNNER job management, execution of the program, and storage of job output (in the local file system for local jobs and in persistent storage for cloud jobs). For cloud execution, this time also includes task processing time in the queuing system (i.e. the time between job submission and when a worker dequeues a task for execution). End-to-end execution time is job-dependent but our presentation of these values for individual StochSS jobs enables us to distinguish CLOUDRUNNER overheads. The typical use of CLOUDRUNNER, however, is to use the cloud for large numbers of concurrent jobs or for long running jobs.

We chose three different StochKit parameterizations arbitrarily to represent different job lengths (we targeted 5, 60, and 120 second jobs, which we refer to as small, medium, and large in our graph). We execute each job 10 times and present the average and 95% confidence bounds. The results are shown in Figure 6. We show the confidence interval from the student T distribution with 9 degrees of freedom. The Local bars show end-to-end execution time for local job deployment and file system storage. Cloud bars show the same deployment via our distributed tasking service using AWS resources.

Above each Local bar is the percentage of time spent in computing (local storage time is statistically insignificant). Above each Cloud job is the percentage of time spent computing (C) and storing job results (S). CLOUDRUNNER overhead for local deployment is statistically insignificant; CLOUDRUNNER overhead for cloud deployment varies by job size. CLOUDRUNNER imposes overhead on end-to-end execution for queue processing. This overhead can consume a significant portion of this time for short running jobs and is amortized by long running jobs. Cloud storage to persist job output also consumes a significant portion of overall time relative to local file system storage.

Included in these Cloud execution times is task queue time (due to message brokering) with no contention. We observe this time to be independent of task size (it is 5 seconds on average across job sizes). In addition, we spawned the task worker VMs in this study ahead of time. Spawning time in our experience varies between 1 and 3 minutes in AWS. Since CLOUDRUNNER triggers spawning when there are tasks waiting in the queue (up to the maximum number of VMs identified by the user), tasks may wait this additional duration to be serviced. Moreover, once the maximum number of VMs is spawned, task wait time will be impacted by contention with already running tasks (for which completion time is application specific).

V. RELATED WORK

CLOUDRUNNER is most similar to work that we investigated previously on a domain specific language for deploying high-performance computing applications in the cloud, called Neptune [1], [2]. The work presented herein is a significantly simpler approach that automates much of the specification work required to use Neptune. In particular, CLOUDRUNNER consumes programs from a source code repository, requiring only that developers wrap their existing installation and build scripts in well-defined file names and to specify their program usage (parameters) as a YAML file. CLOUDRUNNER also extends beyond Neptune to provide a web interface to user programs and supports secure, multi-user access to the service.

Other related work includes `cloudinit.d` from Nimbus [10]. `cloudinit.d` provides an API that users employ to automatically launch, configure, and deploy nodes in a cloud infrastructure. Other related tools, such as Nimbus Context Broker [10] and Mesos [11] and emerging IT tools (e.g. Puppet, Chef, and SaltStack) help users to automate configuration of VMs, but none provide the end-to-end service-ization that CLOUDRUNNER provides, which includes automatic UI generation, local web service deployment, and execution of arbitrary programs locally and over disparate cloud infrastructures.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we present a new approach to simplifying the use of cloud infrastructures for the deployment of arbitrary Linux applications. In particular, we present the design and implementation of CLOUDRUNNER, a toolkit that wraps applications within a simple web service so that users can parameterize and execute their programs locally or via one or more cloud infrastructures (public and on-premise) directly through their browser. Users provide CLOUDRUNNER with their Git repository to which they have added simple configuration scripts. CLOUDRUNNER uses these scripts to create and configure a virtual machine image that it then instantiates to execute their applications as a cloud task that the user parameterizes via the CLOUDRUNNER web app. CLOUDRUNNER automates many of the steps in converting an executable to a web and cloud based application so that users can focus on their innovation, science, and results.

To evaluate CLOUDRUNNER, we use it to implement StochSS – a Stochastic Simulation Service, from the StochKit application from UC Santa Barbara. With StochSS, scientists use the CLOUDRUNNER web app to run a wide range of simulation algorithms on biochemical models. We use StochSS to measure end-to-end performance of CLOUDRUNNER. We find that CLOUDRUNNER introduces a small consistent overhead for cloud deployment (in particular, for task management) and no perceivable overhead for local execution. We also find that without considering task wait time (e.g. when there are more tasks than workers), that the overhead for task management is amortized by application execution and storage time for applications that execute longer than one minute. As part of future work, we are augmenting CLOUDRUNNER with support for map-reduce jobs, for providing users with more control over job output and data visualization, and for estimating the costs of cloud use.

This work was funded in part by Google, IBM, NSF grants CNS-0546737, CNS-0905237, CNS-1218808, and NIH grant 1R01EB014877-01.

REFERENCES

[1] C. Bunch, B. Drawert, N. Chohan, C. Krintz, L. Petzold, and K. Shams, "Language and Runtime Support for Automatic Configuration and Deployment of Scientific Computing Software over Cloud Fabrics," *Grid Computing: Special Issue on Data Intensive Clouds*, vol. 10, no. 1, pp. 23–46, 2012.

[2] C. Bunch, N. Chohan, C. Krintz, and K. Shams, "Neptune: A Domain Specific Language for Deploying HPC Software on Cloud Platforms," in *ACM Workshop on Scientific Cloud Computing*, June 2011.

[3] "Celery," http://www.celeryproject.org/ [Online; acc 15-Mar-2014].

[4] "RabbitMQ," https://www.rabbitmq.com/ [Online; acc 15-Mar-2014].

[5] K. Sanft, S. Wu, M. Roh, J. Fu, R. K. Lim, and L. Petzold, "StochKit2: software for discrete stochastic simulation of biochemical systems with events," *Bioinformatics*, vol. 27, no. 17, pp. 2457–2458, 2011.

[6] "YAML," http://www.yaml.org/ [Online; acc 15-Mar-2014].

[7] "AWS Elastic Compute Cloud," http://aws.amazon.com/ec2/ [Online; acc 15-Mar-2014].

[8] "Eucalyptus," https://www.eucalyptus.com/eucalyptus-cloud/iaas [Online; acc 15-Mar-2014].

[9] "Google app engine," http://code.google.com/appengine/ [Online; acc 27-Sept-2013].

[10] K. Keahey and T. Freeman, "Nimbus or an Open Source Cloud Platform or the Best Open Source EC2 No Money Can Buy," in *Supercomputing 2008*, 2008.

[11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," in *Networked Systems Design and Implementation*, 2011.